Обработка естественного языка в действии

Лейн Хобсон
Хапке Ханнес
Ховард Коул

MANNING

Natural Language Processing in Action

Understanding, analyzing, and generating text with Python

HOBSON LANE
COLE HOWARD
HANNES MAX HAPKE

MANNING
SHELTER ISLAND

Лейн Хобсон, Хапке Ханнес, Ховард Коул

Обработка естественного языка в действии

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2020

ББК 22.183.492+ 81.13
УДК 004.43+81.33
Х68

Хобсон Лейн, Ханнес Хапке, Коул Ховард

Х68 Обработка естественного языка в действии. — СПб.: Питер, 2020. — 576 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1371-2

Последние достижения в области глубокого обучения позволяют создавать приложения, с исключительной точностью распознающие текст и речь. Что в результате? Появляются чат-боты, ведущие диалог не хуже реальных людей, программы, эффективно подбирающие резюме под заданную вакансию, развивается превосходный предиктивный поиск, автоматически генерируются аннотации документов. Благодаря новым приемам и инструментам, таким как Keras и Tensorflow, сегодня возможно как никогда просто реализовать качественную обработку естественного языка (NLP).

«Обработка естественного языка в действии» станет вашим руководством по созданию программ, способных распознавать и интерпретировать человеческий язык. В издании рассказано, как с помощью готовых пакетов на языке Python извлекать из текста смыслы и адекватно ими распоряжаться. В книге дается расширенная трактовка традиционных методов NLP, что позволит задействовать нейронные сети, современные алгоритмы глубокого обучения и генеративные приемы при решении реальных задач, таких как выявление дат и имен, составление текстов и ответов на неожиданные вопросы.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 22.183.492+ 81.13
УДК 004.43+81.33

ISBN 978-1617294631 англ.
ISBN 978-5-4461-1371-2

Краткое содержание

Часть III. Поговорим серьезно. Реальные задачи NLP

Приложения

Оглавление

Часть III. Поговорим серьезно. Реальные задачи NLP

Приложения

Предисловие

Впервые я встретила Ханнеса в 2006 году, когда мы поступили в аспирантуру на одной кафедре по разным специальностям. Он быстро стал известен своими работами, в которых совместно использовались методы машинного обучения и электротехники, а также своей решимостью сделать мир лучше. Последняя и направляла все проекты и компании, с которыми он сотрудничал, на протяжении его карьеры. Именно этот внутренний компас свел его с Хобсоном и Коулом, разделявшими эту страсть к проектам.

Их стремление использовать машинное обучение (ML) на благо людей убедило меня написать это предисловие. Схожее желание сделать мир лучше вело и меня саму на пути исследований в области ML. Моя дорога привела меня к разработке алгоритмов кратномасштабного моделирования (multi-resolution modeling) экологических данных о распределениях видов для их охраны и наблюдения за ними. С тех пор я была твердо настроена продолжать работать в сферах, где могла бы применить машинное обучение для улучшения жизни людей.

> Неограниченная власть означает и неограниченную ответственность.
>
> *Вольтер(?)*

Эти слова остаются справедливыми вне зависимости от того, кому приписать их — Вольтеру или дяде Бену, хотя, возможно, ныне можно их перефразировать: «Неограниченный доступ к данным означает и неограниченную ответственность». Мы доверяем свои данные различным компаниям в надежде, что они будут использованы для улучшения нашей жизни. Мы разрешаем просмотр наших писем для проверки орфографии; обрывки наших жизней в соцсетях изучаются и используются для отображения контекстной рекламы. Наши телефоны и бытовая техника откликаются на наши слова, иногда когда мы обращаемся и не к ним. Даже наши

предпочтения при выборе новостей отслеживаются. Что же лежит в основе всех этих могущественных технологий? Ответ: обработка естественного языка (natural language processing, NLP).

В этой книге вы найдете как теорию, так и практические примеры, необходимые для того, чтобы не просто понимать внутренний механизм работы этих систем, но и начать создавать свои алгоритмы или модели. Базовые понятия теории вычислительной техники здесь изящно преобразуются в надежный фундамент для последующих подходов и методик. Авторы проводят читателя через четкое и изложенное ясным языком описание основных методик NLP, начиная с таких испытанных на практике методов, как TF-IDF, перед поверхностным, но глубоким (да, я скаламбурила) погружением в использование глубоких нейронных сетей для NLP.

Язык — фундамент нашей человеческой общности. Мы передаем друг другу не только факты, но и эмоции; с помощью языка мы получаем новые знания и строим взаимоотношения. У вас есть возможность обрести ясное понимание не только внутренних механизмов NLP, но и путей создания значимых систем, которые однажды могут сами познать человечество через наш язык. У технологии NLP — огромный потенциал как для злоупотреблений, так и для добрых дел. Авторы, делясь в этой книге своими знаниями, надеются склонить нас в сторону более светлого будущего.

Др. Арвен Гриффьюн (Arwen Griffioen),
старший исследователь данных,
компания Zendesk

Вступление

Примерно в 2013 году обработка естественного языка и чат-боты начали получать все большее распространение в нашей жизни. Сначала поиск Google напоминал лишь работу с предметным указателем — инструментом, для использования которого не требовалось особых навыков. Но скоро он стал интеллектуальнее и начал понимать поисковые запросы, все более близкие к естественному языку. Далее еще больше усложнялась функциональность автодополнения в смартфонах. По центру зачастую было указано именно интересующее пользователя слово[1].

В конце 2014 года мы с Сандером Шивайей (Thunder Shiviah) сотрудничали в рамках одного из проектов Hack Oregon, связанного с интеллектуальным анализом данных о финансировании избирательной кампании. Мы пытались отыскать связи между источниками финансирования политиков. Казалось, что политики скрывали своих «инвесторов» за туманным языком финансовых документов их кампаний. Самое интересное было не то, что мы смогли раскрыть эти связи с помощью простых методик обработки естественного языка. Больше всего меня удивляло получение от Сандера лаконичного, но подходящего ответа через считанные секунды после нажатия мной кнопки Отправить. Оказалось, что он использовал *Smart Reply* — утилиту-помощник для Google Inbox, составляющую ответы быстрее, чем человек может прочитать письмо.

---

[1] Нажмите среднюю кнопку (https://www.reddit.com/r/ftm/comments/2zkwrs/middle_button_game/) на клавиатуре интеллектуального ввода текста на смартфоне, чтобы узнать, что вы хотите сказать дальше, по мнению Google. Эта игра впервые появилась на Reddit под названием SwiftKey game (https://blog.swiftkey.com/swiftkey-game-winning-is/) в 2013 году.

Так что я решил тщательнее разобраться в механизмах, стоящих за этим фокусом. И чем больше узнавал, тем более понятными и выполнимыми представлялись подобные впечатляющие трюки обработки естественного языка. И какой бы проект машинного обучения я ни взял, практически все они включали использование обработки естественного языка.

Вероятно, причина — в моей любви к словам и восхищении их ролью в человеческом мышлении. Я мог часами обсуждать, существует ли вообще «смысл» у слов, с Джоном Ковальски (John Kowalski), моим начальником в Sharp Labs. По мере того как я обретал уверенность в своих силах и узнавал все больше от своих учителей и учеников, мне казалось, что я скоро смогу сам создать что-то новое и удивительное.

Один из трюков, которому я научился, — проход в цикле по набору документов и подсчет того, насколько часто за словами вроде *War* и *Hunger* следуют слова вроде *Games* или *III*. Если проделать это для большого набора текстов, можно научиться достаточно хорошо угадывать правильное следующее слово в цепочке слов, фразе или предложении. Такой классический подход к обработке языка был для меня интуитивно понятен.

Профессора и руководители называют это марковской цепью, но, с моей точки зрения, это просто таблица вероятностей — лишь список количества вхождений для каждого из слов в зависимости от предыдущего слова. Профессора назвали бы это условным распределением, вероятностями слов относительно предыдущего слова. Созданная Питером Норвигом (Peter Norvig) программа для проверки орфографии демонстрирует, что такой подход хорошо масштабируется и требует лишь нескольких строк кода на Python[1]. Все, что нужно, — много текста на естественном языке. Я с интересом предвкушаю появление возможностей его обработки на громадных общедоступных коллекциях вроде «Википедии» или проекта «Гутенберг»[2].

А позднее я услышал про латентно-семантический анализ (latent semantic analysis, LSA). Мне казалось, что это просто нестандартный способ описания некоторых знакомых мне по колледжу операций линейной алгебры. Если отслеживать совместные вхождения слов, можно воспользоваться линейной алгеброй для группировки их по «темам». LSA способен сжать смысл целого предложения или даже длинного документа в один вектор, а использующие LSA поисковые системы обладают сверхъестественной способностью выдавать именно те документы, которые я искал. Хорошие поисковые системы делают это даже тогда, когда я не могу придумать, какие слова в таких документах могут быть.

---

[1] См. веб-страницу How to Write a Spelling Corrector на сайте Питера Норвига: http://www.norvig.com/spell-correct.html.

[2] Если вы осознаете ценность свободного доступа к книгам на естественном языке, то, наверное, захотите быть в курсе предпринимаемых в международном масштабе попыток расширить права интеллектуальной собственности далеко за пределы первоначального «срока годности»: см. http://www.gutenberg.org и http://www.gutenbergnews.org/20150208/copyright-term-extensions-are-looming.

Далее проект *gensim* выпустил реализацию векторов слов Word2vec для Python, благодаря чему стало возможно производить семантические математические операции над отдельными словами. И оказалось, что если разбить документы на меньшие порции, то подобная причудливая математика нейронных сетей будет эквивалентна старому доброму методу LSA. Это была сенсация. У меня появилась надежда внести какой-то вклад в данную сферу. Я размышлял об иерархических семантических векторах уже много лет — о том, как книги состоят из глав, состоящих из абзацев, состоящих из предложений, которые состоят из фраз, а те — из слов, которые составлены из букв. Томаш Миколов (Tomas Mikolov), создатель Word2vec, догадался искать основную семантику текста в связях между двумя слоями иерархии — словами и фразами из десяти слов. В течение десятилетий исследователи NLP рассматривали характеристики слов, например точность или эмоциональность. Причем все эти оценки тональностей и компоненты можно складывать/вычитать, создавая разные комбинации смыслов словосочетаний. Миколов придумал, как создавать эти векторы не «вручную» или даже вообще не описывая нужные компоненты. И NLP стало по-настоящему интересным занятием!

Примерно тогда же Сандер познакомил меня со своим подопечным, Коулом. Позднее другие люди познакомили меня с Ханнесом. И мы втроем начали «разделять и властвовать» в сфере NLP. Меня интересовала возможность создания интеллектуального чат-бота. Коула и Ханнеса вдохновляли широкие возможности «черных ящиков» нейронных сетей. Вскоре они открыли такой ящик, заглянули внутрь и описали мне свои находки. Коул даже применил их для создания чат-ботов, помогая мне в путешествии по стране NLP.

И все обнаруженные нами потрясающие новые методы NLP казались мне понятными и пригодными для моих целей. И практически для каждого метода почти сразу появлялась реализация на языке Python. Причем в эти пакеты Python зачастую включались необходимые нам данные и предобученные модели. Фраза «для этого метода уже существует пакет» стала постоянным рефреном на наших воскресных встречах в кофейне Флойда, где Ханнес, Коул и я обычно искали вместе с друзьями решение какой-либо задачи или играли в го и «нажмите среднюю кнопку». Мы быстро добились некоторых успехов и начали читать лекции на эту тему для различных курсов и команд лаборатории Hack Oregon.

В 2015 и 2016 годах все стало серьезнее. По мере того как выходили из-под контроля бот Tau от Microsoft и другие, стало ясно, что боты естественного языка влияют на социум. В 2016-м я был занят тестированием бота, собиравшего твиты для прогноза результатов выборов. В то же время начали появляться новости относительно влияния ботов в Twitter на выборы президента США. В 2015-м я узнал про систему, использовавшуюся для предсказания экономических трендов и запуска крупных финансовых трансакций на основе одного только «мнения» алгоритма о текстах на естественном языке[1]. Подобные влияющие на экономику и приводящие

[1] См. веб-страницу Why Banjo Is the Most Important Social Media Company You've Never Heard Of: https://www.inc.com/magazine/201504/will-bourne/banjo-the-gods-eye-view.html.

к сдвигу в обществе алгоритмы сформировали усиливающую петлю обратной связи. При естественном отборе среди таких алгоритмов, похоже, «выживали» приносившие наибольший доход. Причем этот доход зачастую давался ценой самих основ демократии. Машины влияли на людей, а мы, люди, обучали их применять естественный язык для повышения их влияния. Безусловно, эти машины контролировались мыслящими и вдумчивыми людьми, но с приходом понимания, что на этих людей также влияли боты, становилось не по себе. Могут ли эти боты привести к неуправляемой цепной реакции усиления обратной связи? Возможно, начальные условия ботов могли сильно повлиять на то, окажется ли эта цепная реакция благоприятной для человеческих ценностей и важных для людей вопросов.

А затем ко мне пришел Брайан Сойер (Brian Sawyer) с предложением от издательства Manning. И я уже знал, о чем хочу написать и кого хотел бы видеть в соавторах. Темпы развития NLP-алгоритмов и накопления данных на естественных языках продолжали расти, а Коул, Ханнес и я пытались не отстать от них.

Благодаря бурному потоку неструктурированных данных на естественном языке по вопросам политики и экономики NLP становился неотъемлемым инструментом политтехнологов и финансистов. Бросает в дрожь, когда понимаешь, что часть статей, на которых основываются подобные предсказания, написана другими ботами. Причем эти боты зачастую не подозревают о существовании друг друга. Они, по сути дела, общаются между собой и пытаются друг другом манипулировать, не задумываясь о благополучии людей и общества в целом. Мы для них просто попутчики.

Один из примеров этого замкнутого круга ботов, разговаривающих с ботами, иллюстрируется ростом финансово-технологического стартапа Banjo в 2015 году (https://www.inc.com/magazine/201504/will-bourne/banjo-the-gods-eye-view.html). Благодаря мониторингу Twitter NLP-бот Banjo был способен предсказывать заслуживающие упоминания новости за 30–60 минут до публикации первой статьи журналистами Reuters или CNN. Многие из используемых им для обнаружения этих событий твитов почти наверняка отмечались лайками и ретвитились несколькими другими ботами, чтобы привлечь внимание NLP-бота Banjo. И такие твиты не просто отбирались, раскручивались и оценивались в соответствии с алгоритмами машинного обучения. Многие из них были целиком написаны NLP-движками[1].

Генерация все большего объема развлекательного, рекламного и финансово-отчетного контента не требует участия человека. NLP-боты пишут даже целые сценарии для фильмов (http://fivethirtyeight.com/features/some-like-it-bot/). Компьютерные игры и виртуальные миры содержат боты, иногда даже разговаривающие с нами о самих ботах и ИИ. Подобная «пьеса внутри пьесы» становится еще более рекурсивной в посвященных компьютерным играм фильмах, на которые боты затем пишут обзоры в реальном мире, чтобы помочь нам выбрать, что смотреть. Установление авторства становится все сложнее по мере того, как алгоритмы обработки естественного языка

[1] Финансовый отчет Twitter за 2014 год показал, что более 8 % твитов было составлено ботами, а в 2015 году DARPA провело конкурс (https://arxiv.org/ftp/arxiv/papers/1601/1601.05140.pdf) по их обнаружению и снижению их влияния на общество.

учатся анализировать стиль написанного на естественном языке текста и генерировать текст в этом стиле[1].

NLP влияет на общество и менее явными способами: обеспечивает эффективный информационный поиск и с помощью фильтрации или продвижения определенных страниц влияет на потребляемую нами информацию. Поиск — исторически первая коммерчески успешная сфера приложения NLP. Поиск вдохновлял все более быструю разработку NLP-алгоритмов, которые затем усовершенствовали технологии поиска. Мы поможем вам внести свой вклад в данную область, продемонстрировав некоторые из лежащих в основе веб-поиска методик индексации и прогнозирования естественного языка. Мы покажем, как проиндексировать эту книгу и освободить свой мозг для более высоких мыслительных задач, отдав на откуп машинам запоминание терминологии, фактов и фрагментов кода на языке Python. Возможно, после этого вы сможете повлиять на общество с помощью созданных вами инструментов поиска на естественном языке.

Разработка NLP-систем привела к пику информационных потоков и вычислений, анализирующих действия человека. Сейчас можно ввести лишь несколько символов в строку поиска и зачастую получить именно ту информацию, которая нужна для завершения текущей задачи, например написания программного обеспечения для учебника по NLP. Первые несколько вариантов автодополнения нередко настолько сверхъестественно хорошо подходят, что кажется, будто нам помогает в поиске специальный человек. Конечно, при написании книги мы использовали разнообразные поисковые системы. В некоторых случаях результаты поиска включали сообщения в соцсетях и статьи, отобранные или написанные ботами, что стало причиной многих пояснений и примеров использования NLP на последующих страницах.

Что же является движущей силой NLP?

- Появившееся понимание ценности непрерывно расширяющейся паутины неструктурированных данных?
- Повышение вычислительных возможностей, наконец-то достигнувших достаточного для реализации идей исследователей уровня?
- Эффективность взаимодействия с машиной на нашем собственном языке?

Все это и многое другое. Введите вопрос «Почему обработка естественного языка сейчас настолько важна?» в любую поисковую систему[2], и вы найдете статью в «Википедии» с перечнем убедительных причин[3].

---

[1] NLP уже успешно применяется для идентификации стиля таких авторов XVI века, как Шекспир (https://pdfs.semanticscholar.org/3973/ff27eb173412ce532c8684b950f4cd9b0dc8.pdf).

[2] Например, вот запрос про NLP в поисковую систему Duck Duck Go по адресу https://duckduckgo.com/?q=Why+is+natural+language+processing+so+important+right+now:.

[3] См. статьи «Википедии»: https://en.wikipedia.org/wiki/Natural_language_processingWikipedia/NLP и https://ru.wikipedia.org/wiki/Обработка_естественного_языка.

Существуют также и более глубинные причины. Одна из них — стремление поскорее создать искусственный интеллект общего уровня (artificial general intelligence, AGI), он же глубокий ИИ. Люди мыслят, возможно, лишь потому, что способны группировать мысли, сохранять (запоминать) их и эффективно ими обмениваться. Благодаря этому наш разум способен преодолевать временные и географические границы, объединяя наши интеллекты в единый коллективный разум.

Одна из изложенных в книге Стивена Пинкера (Steven Pinker) *Stuff of Thought* («Субстанция мышления») идей состоит в том, что мы фактически мыслим на естественном языке (https://en.wikipedia.org/wiki/The_Stuff_of_Thought). Выражение «внутренний диалог» возникло вовсе не случайно. Компании Facebook, Google и Илон Маск (Elon Musk) убеждены, что слова будут для мыслей протоколом обмена информацией по умолчанию. Все они инвестируют средства в проекты, связанные с попытками преобразования мыслей, мозговых волн и электрических сигналов в слова[1]. Кроме того, гипотеза Сепира — Уорфа гласит, что слова влияют на наш образ мышления[2], а естественный язык, безусловно, является средой передачи культуры и коллективного сознания.

Так что если это подходит для человеческого мозга и мы хотели бы эмулировать или моделировать человеческие мысли в машине, то обработка естественного языка, вероятно, будет играть в этом важнейшую роль. Плюс в структурах данных и вложенных связях между словами кроется множество важных ключей к постижению разума, о которых мы расскажем в данной книге. В конце концов, вы собираетесь использовать эти структуры, а сети связей позволяют неодушевленным системам поглощать, хранить, извлекать и генерировать естественный язык способами, напоминающими человеческие.

Есть и еще более важная причина, по которой важно научиться программировать системы, хорошо использующие естественный язык... просто чтобы спасти мир. Надеемся, что вы следили за дискуссией сильных мира сего относительно *проблемы контроля ИИ* и непростой задачи разработки «дружественного к людям ИИ» (https://en.wikipedia.org/wiki/AI_control_problem). Ник Бостром (Nick Bostrom)[3], Калум Чейс (Calum Chace)[4], Илон Маск[5] и многие другие верят, что будущее человечества зависит от нашей способности разрабатывать дружественные к нам машины,

---

1 См. статью из Wired Magazine: We are Entering the Era of the Brain Machine Interface по адресу https://backchannel.com/we-are-entering-the-era-of-the-brain-machine-interface-75a3a1a37fd3.

2 См. веб-страницу https://ru.wikipedia.org/wiki/Гипотеза_лингвистической_относительности.

3 Официальный сайт Ника Бострома: https://nickbostrom.com/.

4 *Chace C.* Surviving AI (https://www.singularityweblog.com/calum-chace-on-surviving-ai/).

5 См. веб-страницу Why Elon Musk Spent $10 Million To Keep Artificial Intelligence Friendly: http://www.forbes.com/sites/ericmack/2015/01/15/elon-musk-puts-down-10-million-to-fight-skynet/#17f7ee7b4bd0.

а естественному языку в обозримом будущем предстоит стать важным звеном связи между людьми и машинами.

Даже когда мы сможем непосредственно «передавать свои мысли» машинам, на эти мысли все равно, вероятно, внутри нашего мозга будут влиять естественные слова и языки. Граница между естественным и машинным языком будет размываться точно так же, как сейчас размывается граница между человеком и машиной. На самом деле эта граница начала размываться в 1984-м[1] — году публикации «Манифеста киборгов» (https://en.wikipedia.org/wiki/A_Cyborg_Manifesto), что делает антиутопические предсказания Джорджа Оруэлла более вероятными[2, 3].

Надеемся, что фраза «помочь спасти мир» не вызвала у вас скептической улыбки. По мере чтения этой книги мы покажем вам, как создать и соединить вместе несколько долей «мозга» чат-бота. И при этом вы увидите, какой колоссальный эффект как на людей, так и на машины могут оказывать крошечные сдвиги социальных петель обратной связи. Подобно бабочке, взмахнувшей крыльями в Китае (https://ru.wikipedia.org/wiki/Эффект_бабочки), крошечная корректировка коэффициента усиления «эгоистичности» чат-бота может привести к беспорядочному шквалу враждебного поведения и конфликтности чат-бота[4]. Но также вы увидите, как немногие благожелательно настроенные, альтруистические системы быстро обретают множество верных поклонников, помогающих устранить хаос, причиненный недальновидными ботами, которые преследуют «цели», направленные на обогащение их владельцев. Ориентированные на благо общества, отзывчивые чат-боты могут очень сильно повлиять на этот мир благодаря сетевому эффекту работы в интересах общества[5].

Именно так и именно поэтому авторы данной книги нашли общий язык. В Интернете в ходе открытого, честного и направленного на благо человечества общения на естественном для нас языке возникло сообщество людей, поддерживающих

---

[1] Автор немного искажает факты: «Манифест киборгов» был впервые опубликован в 1985 году. — *Примеч. пер.*

[2] Статья в «Википедии» о романе «1984» Джорджа Оруэлла: https://ru.wikipedia.org/wiki/1984_(роман).

[3] Статья в «Википедии» о 1984 годе: https://ru.wikipedia.org/wiki/1984_год.

[4] Основной инструмент чат-бота — имитация поведения людей, с которыми он общается. И участники диалога могут воспользоваться своим влиянием, чтобы спровоцировать как про-, так и асоциальное поведение ботов. См. статью из интернет-журнала Tech Republic Why Microsoft's Tay AI Bot Went Wrong: http://www.techrepublic.com/article/why-microsofts-tay-ai-bot-went-wrong.

[5] Пример автономных машин, «заражающих» людей своим тщательно просчитанным поведением, можно найти в исследованиях, посвященных работе беспилотных автомобилей в часы пик (https://www.enotrans.org/wp-content/uploads/AV-paper.pdf). В некоторых работах показано, что лишь 10 % таких автомобилей на автостраде достаточно для стабилизации человеческого поведения на дороге, уменьшения заторов и обеспечения большей плавности и безопасности дорожного трафика.

те же идеи. И мы используем свой коллективный разум для создания и поддержки других полуразумных действующих лиц (машин)[1]. Мы надеемся, что наши слова оставят свой отпечаток в вашем сознании и распространятся, подобно мему, по миру чат-ботов, заражая других людей страстью к созданию функционирующих на благо общества NLP-систем. И мы надеемся, что, когда действительно возникнет сверхчеловеческий интеллект, он хоть немного проникнется духом работы на благо общества.

Хобсон Лейн

[1] Начало моему путешествию в страну машинного обучения в 2010 году положила книга Тоби Сегарана (Toby Segaran) Programming Collective Intelligence («Программирование коллективного разума»): https://www.goodreads.com/book/show/1741472.Programming_Collective_Intelligence.

Благодарности

Написание этой книги и необходимого программного обеспечения, позволяющего сделать ее *живой*, было бы невозможно без сообщества поддерживавших нас одаренных разработчиков, наставников и друзей. В основном это представители активного портлендского сообщества, существование которого стало возможно благодаря таким организациям, как PDX Python, Hack Oregon, Hack University, Civic U, PDX Data Science, Hopester, PyDX, PyLadies и Total Good.

Наши благодарности Закари Кенту (Zachary Kent), спроектировавшему, создавшему и поддерживающему *openchat* (Twitter-бот для открытых встреч (open spaces) конференции PyCon), и Райли Растеду (Riley Rustad), создававшему прототипы его схемы данных по мере написания книги и совершенствования наших навыков. Санти Адавани (Santi Adavani) реализовал распознавание поименованных сущностей с помощью библиотеки Stanford CoreNLP, разработал руководства по SVD и PCA и помог нам в работе с его фреймворком RocketML HPC для обучения модели описания видео в режиме реального времени для незрячих. Эрик Миллер (Eric Miller) выделил часть ресурсов Squishy Media для оттачивания навыков Хобсона в визуализации NLP. Эрик Ларсон (Erik Larson) и Алек Ландграф (Aleck Landgraf) великодушно предоставили Хобсону и Ханнесу возможность свободно экспериментировать с машинным обучением и NLP в их стартапе.

Анна Оссовски (Anna Ossowski) помогла спроектировать Twitter-бот для открытых встреч конференции PyCon, а затем присматривала за ним в начале его обучения, чтобы его твиты были более вменяемыми. Чик Уэллс (Chick Wells), сооснователь компании Total Good, разработал продуманный и увлекательный тест на IQ для чат-ботов и постоянно помогал нам своими знаниями в сфере DevOps. Эксперты по NLP, например Кайл Горман (Kyle Gorman), щедро уделяли нам

свое время, делились знаниями об NLP, кодом и драгоценными наборами данных. Кэтрин Николовски (Catherine Nikolovski) поделилась ресурсами Hack Oregon и Civic U. Крис Джан (Chris Gian) предоставил идеи своего NLP-проекта для примеров из этой книги и доблестно взял на себя проведение семинаров по машинному обучению Civic U, когда преподаватель увяз в работе над книгой. Рейчел Келли (Rachel Kelly) разрекламировала нас и помогла на начальном этапе фактического написания книги. Сандер Шивайя неизменно вдохновлял нас своими неустанными наставлениями и безграничным энтузиазмом относительно машинного обучения и жизни вообще.

Молли Мерфи (Molly Murphy) и Наташе Петти (Natasha Pettit) из Hopester мы благодарны за то, что они вдохновили нас на создание полезного для общества бота. Джереми Робин (Jeremy Robin) и команда Talentpair оставили ценные отзывы относительно разработки ПО и помогли воплотить в жизнь некоторые из упомянутых в данной книге концепций. Дэн Феллин (Dan Fellin) дал толчок нашим NLP-приключениям своими консультациями на семинарах PyCon 2016 и курсах Hack University по скрапингу Twitter. Алекс Розенгартен (Alex Rosengarten), Энрико Касини (Enrico Casini), Ригоберто Мачедо (Rigoberto Macedo), Шарлина Хан (Charlina Hung) и Ашвин Канан (Ashwin Kanan) из Aira оживили идею чат-бота из этой книги эффективными, надежными и легкими в сопровождении диалоговым движком и микросервисом. Спасибо нашим «подопытным кроликам» Элле и Уэсли Минтонам (Ella & Wesley Minton), которые экспериментировали с нашими сумасшедшими идеями чат-ботов, учась писать свои первые программы на языке Python. Суман Кануганти (Suman Kanuganti) и Мария Макмаллин (Maria MacMullin) оказались достаточно дальновидны и основали фонд *Do More*, чтобы сделать визуальный интерфейс сервиса Aira доступным для студентов. Спасибо Клейтону Льюису (Clayton Lewis), позволившему нам участвовать в его исследованиях на тему виртуальных помощников, хотя единственным вкладом, который мы могли внести в его семинары в Институте Коулмена, был голый энтузиазм и неумело написанный код.

Часть работы, описанной в этой книге, была выполнена благодаря гранту, выделенному корпорации Aira Tech Национальным научным фондом (National Science Foundation, NSF). Все мнения, выводы и рекомендации, приведенные в издании, отражают лишь точку зрения авторов, но вовсе не обязательно — перечисленных выше организаций или отдельных лиц.

Наконец мы хотели бы поблагодарить всех сотрудников издательства Manning за их тяжкий труд, а также доктора Арвен Гриффьюн за любезно написанное ею предисловие, доктора Давиде Кадамуро (Davide Cadamuro) за техническую рецензию, а также всех наших рецензентов, чьи отзывы помогли улучить нашу книгу: Чан-Яо Чжуана (Chung-Yao Chuang), Фраджа Зайена (Fradj Zayen), Джеффа Барто (Geoff Barto), Джареда Дункана (Jared Duncan), Марка Миллера (Mark Miller), Партхасарати Мандайама (Parthasarathy Mandayam), Роджера Мели (Roger Meli), Шобху Айер (Shobha Iyer), Симону Руссо (Simona Russo), Срджана Сантича (Srdjan Santic), Томмазо Теофили (Tommaso Teofili), Тони Маллена (Tony Mullen), Владимира Купцова (Vladimir Kuptsov), Уильяма И. Уиллера (William E. Wheeler) и Йогеша Кулкарни (Yogesh Kulkarni).

Хобсон Лейн

Я бесконечно признателен своим родителям, вселившим в меня восхищение словами и математикой. Я в вечном долгу перед Лариссой Лейн (Larissa Lane) — самым бесстрашным искателем приключений из всех мне известных, благодаря которой сбылись две мои заветные мечты: кругосветное путешествие и написание книги.

В вечном долгу я также перед Арзу Караэр (Arzu Karaer) — за благосклонность и помощь в склеивании моего разбитого сердца, восстановление моей веры в человечество и за усилия, приложенные для подкрепления обнадеживающей идеи этой книги.

Ханнес Макс Хапке

Я глубоко признателен моей супруге Уитни, которая безгранично поддерживала меня в этом начинании. Спасибо за твои советы и отзывы. Я также хотел бы поблагодарить свою семью, особенно родителей, поощрявших мои исследовательские порывы. Без них вся эта работа была бы просто невозможна. И все приключения в моей жизни были бы невозможны без тех смелых мужчин и женщин, которые изменили мир в ноябрьскую ночь 1989 года[1]. Спасибо вам за храбрость.

Коул Ховард

Я хотел бы выразить признательность моей жене Дон. Ее сверхчеловеческое терпение и понимание поистине вдохновляли меня. И моей маме за возможность свободно экспериментировать и поощрение моего стремления к знаниям.

[1] Ханнес, очевидно, имеет в виду разрушение Берлинской стены 9 ноября 1989 года. — *Примеч. пер.*

Об этой книге

«Обработка естественного языка в действии» — практическое руководство по обработке и генерации текстов на естественном языке. В этой книге мы снабдим вас всеми инструментами и методиками, необходимыми для создания прикладных NLP-систем с целью обеспечения работы виртуального помощника (чат-бота), спам-фильтра, программы — модератора форума, анализатора тональностей, программы построения баз знаний, интеллектуального анализатора текста на естественном языке или практически любого другого NLP-приложения, какое только можно себе представить.

Книга ориентирована на Python-разработчиков среднего и высокого уровня. Значительная часть книги будет полезна и тем читателям, которые уже умеют проектировать и разрабатывать сложные системы, поскольку в ней содержатся многочисленные примеры рекомендуемых решений и раскрываются возможности самых современных алгоритмов NLP. Хотя знание объектно-ориентированного программирования на Python может помочь создавать лучшие системы, для использования приводимой в этой книге информации оно не обязательно.

Как в тексте книги, так и в Интернете приведено достаточно справочной информации и ссылок на ресурсы по отдельным темам для читателей, желающих глубже разобраться в соответствующих вопросах.

Дорожная карта

Если язык Python и обработка естественного языка для вас внове, лучше прочитать сначала часть I, а потом уже интересующую вас (или необходимую для ваших задач) главу части III. Если вы хотите быстрее перейти к новым возможностям NLP,

которые появились благодаря глубокому обучению, то рекомендуем прочитать от начала до конца часть II, которая даст вам понимание работы нейронных сетей.

Рекомендуем, как только вы найдете главу или раздел с фрагментом кода, который можете «выполнить у себя в голове», выполнить его все же на машине. Если похоже, что с помощью какого-либо из примеров можно обработать ваши собственные текстовые документы, поместите этот текст в CSV- или текстовый файл (по одному документу на строку) в каталоге `nlpia/src/nlpia/data/`, после чего можете воспользоваться функцией `nlpia.data.loaders.get_data()` для извлечения этих данных и выполнения примеров на ваших собственных данных.

Что вы найдете в книге

Относящиеся к части I главы описывают логику работы с естественным языком и преобразование его в числа для поиска и вычислений. Подобные базовые навыки работы со словами обладают дополнительным достоинством в виде таких удивительно полезных приложений, как информационный поиск и анализ тональностей. Освоив азы, вы узнаете, что с помощью очень простых арифметических операций, многократно повторяемых в цикле, можно решить весьма важные задачи, например фильтрации спама. Спам-фильтры, подобные тем, которые мы будем создавать в главах 2–4, спасли всемирную систему электронной почты от анархии и застоя. Вы узнаете, как создать фильтр спама с более чем 90%-ной точностью с помощью технологии 1990-х годов — просто вычисляя количества слов и средние значения этих количеств.

Вся эта математика со словами может показаться скучной, но она весьма увлекательна. Очень скоро вы научитесь создавать алгоритмы, умеющие принимать решения относительно естественного языка не хуже, а то и лучше, чем вы сами (и уж точно намного быстрее). Вероятно, при этом вы впервые в жизни сумеете по-настоящему оценить, насколько слова отражают и вообще делают возможным ваше мышление. Надеемся, представления слов и мыслей в многомерном векторном пространстве закружат ваш мозг в рекуррентных циклах самопознания.

Кульминация обучения наступит примерно к середине книги. Центральным моментом этой книги в части II будет исследование вами сложной паутины вычислений и взаимодействия между нейронными сетями. Сетевой эффект взаимодействия в паутине «мышления» маленьких логических блоков обеспечивает возможность решения машинами задач, к которым лишь очень умные люди пытались подступиться в прошлом: таких задач, как вопросы аналогии, автоматическое реферирование текста и перевод с одного естественного языка на другой[1].

Мы расскажем вам не только о векторах слов, но и о многом, многом другом. Вы научитесь визуализировать слова, документы и предложения в облаке взаимосвязанных понятий, распространяющемся далеко за пределы простого и понятного трехмерного пространства. Вы начнете представлять себе документы и слова в виде

[1] К счастью, эту последнюю задачу машины еще не умеют выполнять в совершенстве. — *Примеч. пер.*

описания персонажа в игре «Подземелья и драконы» с множеством случайно выбранных характеристик и способностей, развивающихся и растущих с течением времени, но только в наших головах.

Умение ценить эту межсубъектную реальность слов и их смыслов будет фундаментом завершающей книгу части III, из которой вы узнаете, как создавать машины, способные общаться и отвечать на вопросы не хуже людей.

О коде

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и внутри обычного текста. В обоих случаях исходный код набран `таким моноширинным шрифтом`, чтобы его можно было отличить от обычного текста. Иногда код также набран **`полужирным шрифтом`**, чтобы подчеркнуть изменения по сравнению с предыдущими шагами в этой главе, например, при добавлении новой функциональной возможности к уже существующей строке кода.

Первоначальный исходный код часто переформатировался: мы добавили разрывы строк и переработали отступы, чтобы наилучшим образом использовать доступное место на страницах книги. Иногда этого оказывалось недостаточно, и некоторые листинги включают маркеры продолжения строки (➥). Кроме того, мы нередко удаляли комментарии в исходном коде из листингов там, где код описывался в тексте. Многие листинги сопровождаются примечаниями к коду, подчеркивающими важные нюансы.

Исходный код для всех листингов данной книги доступен для скачивания с сайта издательства Manning по адресу https://www.manning.com/books/natural-language-processing-in-action и с GitHub по адресу https://github.com/totalgood/nlpia.

Дискуссионный форум liveBook

Покупка книги дает право на бесплатный доступ к частному веб-форуму издательства Manning, где можно оставлять свои комментарии о книге, задавать технические вопросы и получать помощь от авторов книги и других пользователей. Чтобы попасть на этот форум, перейдите по адресу https://livebook.manning.com/#!/book/natural-language-processing-in-action/discussion. Узнать больше о форумах издательства Manning и правилах поведения на них можно на странице https://livebook.manning.com/#!/discussion.

Обязательства издательства Manning по отношению к своим читателям требуют предоставления места для содержательного диалога между отдельными читателями, а также читателями и авторами. Эти обязательства не включают какого-либо конкретного объема участия со стороны авторов, чей вклад в работу форума остается добровольным (и неоплачиваемым). Мы советуем вам задавать авторам интересные и трудные вопросы, чтобы их интерес не угас! Указанный форум и архивы предыдущих обсуждений будут доступны на веб-сайте издательства столько, сколько будет в продаже данная книга.

Об авторах

Хобсон Лейн (Hobson Lane) обладает 20-летним опытом создания автономных систем, принимающих важные решения в интересах людей. В компании Talentpair Хобсон обучал машины читать и понимать резюме менее предубежденно, чем большинство специалистов по подбору персонала. В Aira он помогал в создании их первого чат-бота, предназначенного для толкования окружающего мира незрячим. Хобсон — страстный поклонник открытости ИИ и ориентированности его на благо общества. Он вносит активный вклад в такие проекты с открытым исходным кодом, как Keras, scikit-learn, PyBrain, PUGNLP и ChatterBot. Сейчас он занимается открытыми научными исследованиями и образовательными проектами для Total Good, включая создание виртуального помощника с открытым исходным кодом. Он опубликовал многочисленные статьи, выступал к лекциями на AIAA, PyCon, PAIS и IEEE и получил несколько патентов в области робототехники и автоматизации.

Ханнес Макс Хапке (Hannes Max Hapke) — инженер-электротехник, ставший инженером в области машинного обучения. В средней школе он увлекся нейронными сетями, когда изучал способы вычислений нейронных сетей на микроконтроллерах. Позднее, в колледже, он применял принципы нейронных сетей к эффективному управлению электростанциями на возобновляемых источниках энергии. Ханнес обожает автоматизировать разработку программного обеспечения и конвейеров машинного обучения. Он соавтор моделей глубокого обучения и конвейеров машинного обучения для сфер подбора персонала, энергетики и здравоохранения. Ханнес выступал с презентациями на тему машинного обучения на разнообразных конференциях, включая OSCON, Open Source Bridge и Hack University.

Коул Ховард (Cole Howard) — специалист по машинному обучению, специалист-практик, занимающийся NLP, и писатель. Вечный искатель закономерностей, он нашел себя в мире искусственных нейронных сетей. В числе его разработок — масштабные рекомендательные системы для торговли через Интернет и передовые нейронные сети для систем машинного интеллекта сверхвысокой размерности (глубокие нейронные сети), занимающие первые места на конкурсах Kaggle. Он выступал с докладами на тему сверточных нейронных сетей, рекуррентных нейронных сетей и их роли в обработке естественного языка на конференциях Open Source Bridge и Hack University.

Об иллюстрации на обложке

Рисунок на обложке называется «Женщина из Краньска-Гора, Словения». Эта иллюстрация взята из недавнего переиздания книги Бальтазара Аке (Balthasar Hacquet) *Images and Descriptions of Southwestern and Eastern Wends, Illyrians and Slavs* («Изображения и описания юго-западных и восточных венедов, иллирийцев и славян»), опубликованного Этнографическим музеем в Сплите (Хорватия) в 2008 году. Аке (1739–1815) — австрийский врач и ученый, потративший долгие годы на изучение флоры, геологии и этносов Юлийских Альп — горного хребта, простирающегося от северо-восточной Италии до Словении и названного в честь Юлия Цезаря. Нарисованные вручную иллюстрации сопровождают многие опубликованные Аке научные статьи и книги.

Широкое разнообразие рисунков в публикациях Аке наглядно говорит об уникальности и своеобразии восточных Альп всего 200 лет назад. В то время манера одеваться однозначно различала жителей деревень, расположенных всего в нескольких милях друг от друга, а членов разных социальных групп и профессий можно было легко определить по их одежде. Стили одежды с тех пор изменились, и столь богатое разнообразие различных регионов угасло. Зачастую непросто отличить даже жителя одного континента от жителя другого, а обитатели живописных городков и деревушек в Словенских Альпах не так уж сильно не похожи на жителей других частей Словении или остальной части Европы.

В наше время, когда трудно отличить одну компьютерную книгу от другой, издательство Manning проявляет инициативу и деловую сметку, украшая обложки книг изображениями, которые показывают богатое разнообразие жизни в регионах два века назад.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I
Машины для обработки слов

Глава 1 начинает наше путешествие в мир обработки естественного языка со знакомства с некоторыми уже работающими приложениями.

Вы узнаете о способах использования машин, обрабатывающих слова, в вашей повседневной жизни. Надеемся, вы сможете оценить всю мощь машин, собирающих информацию из документа на естественном языке. Слова — фундамент любого языка вне зависимости от того, являются они ключевыми словами языка программирования или теми словами вашего родного языка, которые вы выучили, будучи еще ребенком.

Глава 2 даст вам инструменты для обучения машин извлечению слов из документов. Их куда больше, чем можно представить, но не переживайте — вы узнаете все хитрости. И научитесь автоматически объединять слова в группы слов со схожим значением, не создавая вручную списки синонимов.

Глава 3 обучит подсчитывать слова и собирать их в векторы, отражающие смысл документа. С помощью этих векторов можно отображать значения всего документа, будь он 140-символьным твитом или 500-страничным романом.

Глава 4 откроет некоторые проверенные временем математические приемы для сжатия ваших векторов в более удобные векторы тем.

К концу части I у вас будут все необходимые инструменты для создания различных интересных NLP-приложений (обработки естественного языка): от семантического поиска до чат-ботов.

1 Знакомство с технологией NLP

В этой главе

- Что такое обработка естественного языка (NLP).
- Почему технология NLP очень сложна и получила распространение лишь недавно.
- Когда порядок слов и грамматика имеют значение, а когда — нет.
- Как в чат-ботах сочетается множество инструментов NLP.
- Как начать создавать крошечный чат-бот с помощью регулярных выражений.

Вы собираетесь отправиться в увлекательное путешествие по миру обработки естественного языка. Для начала позвольте рассказать, что такое технология NLP и для чего она. Данная информация позволит углубиться в NLP и придумать способы его использования дома и на работе.

Далее вы узнаете в деталях, как обработать фрагмент английского текста с помощью языка программирования Python, и постепенно пополните свой набор инструментов NLP. Прочитав главу, вы напишете свою первую программу, которая может читать и писать фразы на английском языке. Этот фрагмент кода на языке Python будет первым из многих, которые понадобятся для изучения всех хитростей сборки диалоговой машины на английском языке — чат-бота.

1.1. Естественный язык в сравнении с языком программирования

Естественные языки отличаются от языков программирования. В первую очередь они не предназначены для перевода в конечный набор математических операций. Такие языки нужны для обмена информацией людьми друг с другом. Используя языки программирования, мы не рассказываем про свой день и не просим сходить в продуктовый магазин. Компьютерная программа на языке программирования четко инструктирует машину, что нужно делать, а в естественном английском и французском языках нет компиляторов или интерпретаторов.

ОПРЕДЕЛЕНИЕ

Обработка естественного языка — это область исследований в компьютерных науках и искусственном интеллекте (ИИ), занимающаяся работой с такими языками, как английский или китайский. Обработка обычно включает в себя перевод естественного языка в данные (числа), с помощью которых компьютер может получить информацию об окружающем мире. И это понимание мира иногда используется для создания отражающего его текста на естественном языке.

Тем не менее данная глава демонстрирует способы *обработки* естественного языка. Этот процесс можно даже рассматривать как интерпретацию естественного языка, аналогичную работе интерпретатора языка Python. Когда разрабатываемая вами программа станет обрабатывать естественный язык, она сможет действовать в соответствии с его высказываниями и даже отвечать на них. Стоит отметить, что эти действия и ответы не являются четко определенными, что оставляет вам как разработчику конвейера NLP большую свободу действий.

ОПРЕДЕЛЕНИЕ

Систему обработки естественного языка часто называют *конвейером*, так как она обычно включает в себя несколько этапов обработки, при которых текст на естественном языке подается на один конец системы, а результат выдается на другом.

Скоро вы сможете писать ПО с интересными неожиданными функциями, например умением поддерживать разговор, благодаря чему машины выглядят чуть более человекоподобными. Поначалу это может показаться каким-то фокусом — так происходит со всеми продвинутыми технологиями. Мы предоставим вам возможность заглянуть за кулисы и покажем весь реквизит и инструменты, чтобы вы могли выполнить эти трюки самостоятельно.

Все кажется простым, когда ты знаешь ответ.

Дейв Маджи (Dave Magee)

1.2. Волшебство

Что же такого волшебного в машине, которая может читать и писать на естественном языке? Машины обрабатывают языки с момента изобретения компьютеров. Хотя эти «официальные» языки — вроде ранних языков Ada, COBOL или Fortran — были спроектированы так, чтобы они интерпретировались (или компилировались) только одним верным способом. Сегодня «Википедия» насчитывает более 700 языков программирования. Ethnologue[1], в свою очередь, насчитал в десять раз больше естественных языков, на которых говорят люди по всему миру, а общий объем проиндексированных компанией Google документов, написанных на естественных языках, составляет более 100 миллионов гигабайт[2]. Это просто оценка, она не закончена. Размер реального контента на естественном языке, который сейчас находится в Сети, может превышать 100 миллиардов гигабайт[3]. Однако такое огромное количество текстов на естественном языке не единственная причина создания ПО для обработки этого языка.

Самая интересная часть процесса обработки естественных текстов заключается в сложности. Машины, способные обрабатывать что-то естественное, не являются таковыми. Их можно сравнить со структурой, способной сделать что-то полезное с архитектурными схемами. Возможность обработки с помощью программного обеспечения языков, не предназначенных для понимания машинами, кажется магией — чем-то, что присуще лишь человеку.

В английском языке слово «естественный» (natural) в словосочетании «естественный язык» используется в том же смысле, что и в «окружающем мире» (natural world). Все естественное, появившееся путем эволюции, отличается от искусственных вещей, созданных человеком. Возможность проектировать и создавать ПО для чтения и обработки естественного языка, как в этом тексте, кажется чем-то совершенно трансцендентным, магическим.

Чтобы облегчить вашу работу, мы сфокусируемся на одном естественном языке, английском. Описанные в книге методики можно использовать с целью создания ПО для обработки любого, даже непонятного разработчику языка, включая еще не расшифрованные археологами и лингвистами. Мы покажем вам, как создавать ПО для обработки и генерирования текстов, используя всего один язык программирования, Python.

Python был с самого начала разработан так, чтобы быть удобным для чтения. Он также делает доступными пользователю немало своих «внутренностей», связанных с обработкой языка. Факты, описанные выше, делают его идеальным для обработки естественного языка. Python хорошо подходит, чтобы строить контролируемые конвейеры для алгоритмов NLP в корпоративной среде, где существует много участников, вносящих вклад в одну базу кода. Python даже используется

---

[1] Ethnologue — это веб-публикация, которая анализирует статистику, связанную с естественными языками.

[2] См. статью How Google's Site Crawlers Index Your Site — Google Search: www.google.com/search/howsearchworks/crawling-indexing/.

[3] Приблизительное количество текстов на естественном языке можно оценить, но это значение примерно в 1000 раз больше оценки Google.

вместо «универсального языка» математики и математических символов там, где это возможно. В конце концов, Python предоставляет недвусмысленный способ выражения математических алгоритмов[1]. Он разработан максимально удобочитаемым для таких программистов, как вы.

1.2.1. Машины, которые общаются

Естественные языки не могут быть прямо переведены в четкий набор математических операций. Однако они содержат информацию и инструкции, которые можно извлечь, хранить, индексировать, искать или на основе которых можно немедленно производить какие-либо действия. Одно из этих действий — генерирование последовательности слов в ответ на высказывание. Это и есть одна из функций диалогового движка, или чат-бота, который вы создадите.

Мы с вами целиком сфокусируемся на текстовых документах и сообщениях на английском языке, а не на устных высказываниях. Мы пропустим преобразование устных высказываний в системе распознавания «текст — речь» или «речь — текст». И мы проигнорируем вопрос генерации устной речи или преобразования текста в речь, которое превращает текст обратно в высказывание, произнесенное человеческим голосом. Все, чему вас научит эта книга, можно использовать для создания голосового интерфейса или виртуального помощника, вроде Siri или Alexa, так как библиотеки «текст — речь»/«речь — текст» находятся в свободном доступе. Операционные системы Android и iOS предоставляют высококачественные API распознавания и генерации речи. Кроме того, существуют пакеты языка Python с аналогичной функциональностью для настольных компьютеров и серверов.

Системы распознавания речи

Описание создания настроенной под свои потребности системы распознавания или генерирования речи потребовало бы отдельной книги. Мы оставим это как упражнение для читателя. Такая задача требует большого количества высококачественно маркированных данных, голосовых записей, снабженных фонетической транскрипцией, а также расшифровок текстов на естественном языке, синхронизированных с аудиодорожками. Для этого могут пригодиться некоторые из алгоритмов, которые вы встретите в этой книге, но большая часть алгоритмов распознавания и генерации — совсем другие.

1.2.2. Математика

Обработка естественного языка с целью получения полезной информации может быть сложной. Нужны утомительные статистические расчеты. Однако это то, для чего машины созданы. Как и многие другие технические задачи, решаются они гораздо

[1] Математическая запись неоднозначна. См. раздел Mathematical notation статьи Ambiguity по адресу en.wikipedia.org/wiki/Ambiguity#Mathematical_notation.

проще, если ответ уже известен. К сожалению, компьютеры пока неспособны решать большинство практических задач NLP, таких как разговор и понимание прочитанной информации, так же точно и качественно, как это делают люди. Следовательно, у вас есть шанс настроить алгоритмы, изученные вами в этой книге, так, чтобы немного лучше решать некоторые задачи NLP.

Впрочем, методы и техники, описанные здесь, достаточно мощны для создания машин, способных превзойти людей по точности и скорости решения некоторых удивительно хитрых задач. Например, сарказм в отдельном сообщении Twitter распознается куда точнее машиной, чем человеком[1]. Не переживайте, люди все еще лучше справляются с задачей распознавания юмора и сарказма при диалоге благодаря нашей способности обрабатывать информацию в контексте высказывания. Однако машины непрерывно развивают свои навыки понимания контекста. Данная книга поможет вам включить контекст (метаданные) в свой конвейер NLP, если вы хотите попробовать продвинуть науку и технику вперед.

После того как вы извлечете структурированные числовые данные и векторы из естественного языка, вы сможете применять все инструменты математики и машинного обучения. Мы пользуемся теми же средствами линейной алгебры, которые нужны для проекции 3D-объектов на 2D-экран. Подобным занимались компьютеры и чертежники задолго до появления обработки естественного языка. Эти прорывные идеи стали ключом в мир семантического анализа, позволяя компьютерам интерпретировать и хранить «значение» высказываний, а не просто количество слов/символов. Семантический анализ совместно со статистикой помогают решить проблему неоднозначности естественного языка — того факта, что слова и фразы часто имеют несколько значений и интерпретаций.

Извлечение информации совсем не похоже на создание компилятора для языка программирования (к счастью для вас). Наиболее перспективные инструменты обходят жесткие правила регулярных грамматик (паттернов) или формальных языков. Вы можете положиться на статистические отношения между словами вместо того, чтобы использовать глубокую систему логических правил[2]. Представьте, что вам нужно описать правила английской грамматики и правописания в виде вложенных деревьев из операторов `if...then`. У вас бы получилось расписать правила для всех

---

[1] Gonzalo-Ibanez и др. обнаружили и рассказали в докладе на ACM, что образованные и обученные судьи-люди не могут сравниться по эффективности с их простым алгоритмом классификации (который показал результат 68 %) ACM. Детектор сарказма (github.com/MathieuCliche/Sarcasm_detector) и веб-приложение (www.thesarcasmdetector.com) Мэтью Клише (Matthew Cliche) в Корнелле добились аналогичной точности (> 70 %).

[2] Некоторые грамматические правила могут быть реализованы в виде абстракции из теории вычислительных наук — конечного автомата. Регулярные грамматики можно реализовать в виде регулярных выражений. Существует два пакета языка Python, позволяющих работать с конечными автоматами для регулярных выражений: встроенный re и пришедший ему на замену regex, который нужно устанавливать отдельно. Конечные автоматы представляют собой всего лишь деревья операторов if...then...else для каждого токена (символа, слова, n-граммы) или действия, на которые автомат должен отреагировать или которые должен сгенерировать.

возможных способов сочетания слов, букв и знаков пунктуации в высказываниях? Или хотя бы начать захватывать семантику, смысл высказываний на английском? Даже сумей вы создать пригодное для некоторых типов высказываний ПО, представьте, насколько ограниченным и нестабильным оно бы получилось. Непредвиденная орфография или пунктуация легко нарушат логику вашего алгоритма и собьют его с толку.

Естественные языки также ставят дополнительную, еще более сложную задачу «декодирования». Те, кто говорит и пишет на естественных языках, предполагают, что заниматься обработкой (слушать/читать) будет человек, а не машина. Когда я говорю «доброе утро», я подразумеваю, что вы понимаете, что это такое и когда оно наступает. Интерпретатор должен быть в курсе, что «доброе утро» — общепринятое приветствие, не содержащее дополнительной информации об утре. Данная фраза скорее выражает настроение собеседника и готовность к общению.

Эта модель человеческого обработчика языка обладает большими возможностями. Она позволяет нам выразить много информации с помощью небольшого количества слов, если предположить, что этот «обработчик» имеет доступ к знаниям о мире, основанном на здравом смысле. Такая степень сжатия информации до сих пор недоступна для машин. Не существует чистой «теории разума», которую вы могли бы использовать в своем конвейере NLP. Но в следующих главах мы покажем методы, позволяющие машинам создавать антологии (базы знаний), с помощью которых можно интерпретировать базирующиеся на соответствующих знаниях высказывания.

1.3. Практическое применение

Обработка естественного языка встречается повсюду. Она настолько распространена, что некоторые примеры из табл. 1.1 могут вас удивить.

Таблица 1.1. Категоризация способов использования NLP

Поиск	Веб-поиск	Поиск по документам	Автозаполнение
Редактирование	Правописание	Грамматика	Стиль
Диалог	Чат-бот	Помощник	Планирование
Письменный текст	Индексирование	Согласование	Оглавление
Адрес электронной почты	Спам-фильтр	Классификация	Приоритизация
Интеллектуальный анализ текста	Составление краткого содержания	Извлечение знаний	Медицинские диагнозы
Юриспруденция	Правовое влияние	Поиск прецедентов	Классификация повесток в суд
Новости	Обнаружение событий	Проверка фактов	Составление заголовков

Присваивание	Обнаружение плагиата	Литературная экспертиза	Советы по стилю
Анализ тональности текста	Мониторинг морального состояния общества	Сортировка отзывов на продукты	Техническая поддержка
Прогноз поведения	Финансы	Прогноз выборов	Маркетинг
Литературное творчество	Сценарии фильмов	Поэзия	Слова песен

Поисковая машина может предоставить более точные результаты поиска, если индексирует веб-страницы или архивы документов таким образом, что принимается во внимание значение текста на естественном языке. Автозаполнение использует NLP, чтобы закончить вашу мысль. Оно широко распространено в поисковых системах и клавиатурах мобильных устройств. Во множестве текстовых процессоров, плагинов для браузера и текстовых редакторов есть встроенные средства проверки орфографии, грамматики и согласования слов. Некоторые из диалоговых машин (чат-ботов) используют поиск на естественном языке, чтобы найти ответ на сообщение своего собеседника.

Конвейеры NLP, которые генерируют (составляют) текст, могут использоваться для создания не только коротких ответов в чат-ботах и виртуальных помощниках, но и более длинных отрывков текста. The Associated Press использует роботов-журналистов на базе NLP для написания целых статей по тематике финансов и репортажей со спортивных состязаний[1]. Боты могут составлять прогнозы погоды, которые иногда выглядят так же, как и отчеты людей-синоптиков в вашем родном городе. Вероятно, это связано с тем, что многие метеорологи-люди применяют текстовые процессоры с функциями NLP для составления черновиков прогнозов.

Спам-фильтры на базе NLP в ранних программах электронной почты способствовали тому, что в 1990-е годы электронная почта обогнала телефон и факс в качестве канала связи. Фильтры спама сохранили свое преимущество в игре «кошки-мышки» между спам-фильтрами и генераторами спама для электронной почты, однако могут проигрывать в других областях, таких как социальные сети. Примерно 20 % твитов о президентских выборах в США в 2016 году были составлены чат-ботами[2]. Эти боты поддерживают выгодные их владельцам и разработчикам точки зрения. «Кукловодами» зачастую выступают иностранные правительства или большие корпорации с ресурсами, стремящиеся повлиять на популярность того или иного мнения.

Системы NLP могут не только генерировать короткие посты в социальных сетях. Технологии NLP также можно использовать для составления длинных обзоров

---

1 AP's 'robot journalists' are writing their own stories now // The Verge, 29 января 2015 года: www.theverge.com/2015/1/29/7939067/ap-journalism-automation-robots-financial-reporting.

2 New York Times, 18 октября 2016 года: www.nytimes.com/2016/11/18/technology/automated-pro-trumpbots-overwhelmed-pro-clinton-messages-researchers-say.html и MIT Technology Review, ноябрь 2016 года: www.technologyreview.com/s/602817/how-the-bot-y-politic-influenced-this-election/.

фильмов и продуктов с Amazon и других торговых площадок. Большое количество отзывов — продукт работы автономных конвейеров NLP, которые никогда не были в кинотеатре или не покупали обозреваемый продукт.

Сегодня чат-боты очень распространены. Они используются на ресурсах вроде Slack, IRC, а также на сайтах, предоставляющих клиентские услуги. На таких ресурсах чат-ботам приходится работать с неоднозначными командами или вопросами. Чат-боты, в составе которых находятся системы распознавания и генерации голоса, могут поддерживать длительные беседы ни о чем или с такой «целевой функцией», как бронирование столика в местном ресторане[1]. Системы NLP могут отвечать на звонки в различных компаниях, которые хотят чего-то лучшего, чем обычный обзвон, но при этом не желают платить людям за работу с клиентами.

ПРИМЕЧАНИЕ

> Во время демонстрации системы *Duplex* на Google IO разработчики и руководители упустили из виду вопрос обучения чат-ботов обману. Мы все игнорируем эту дилемму, когда весело общаемся с ботами в Twitter и других анонимных соцсетях, где те не открывают подробности своей родословной. В связи с возникновением ботов, которые так убедительно обманывают нас, появляется и проблема контроля ИИ (en.wikipedia.org/wiki/AI_control_problem). Мрачное пророчество Юваля Харари (Yuval Harari)[2] под названием Homo Deus может сбыться скорее, чем казалось.

Существуют NLP-системы, которые могут заменять «администратора» электронной почты для компаний или помощников руководителей. Подобные помощники планируют встречи и кратко записывают данные в электронную систему Rolodex или CRM (систему управления взаимоотношениями с клиентами), взаимодействуя с другими по электронной почте от имени начальника. Организации доверяют свой бренд и репутацию NLP, позволяя ботам проводить кампании по маркетингу и обмену сообщениями. Некоторые неопытные, но отчаянные авторы книг по NLP позволяют ботам написать несколько предложений в своей книге. Однако об этом позже.

1.4. Язык глазами компьютера

Когда вы печатаете *Good Morning, Rosa*, компьютер видит только `01000111 01101111 01101111` .... Как запрограммировать чат-бот на вежливый ответ на этот поток двоичных данных? Способно ли дерево вложенных условных операторов `if...else` проверить каждый из этих битов и отреагировать на них по отдельности? Подобный вариант будет эквивалентом создания специальной программы под названием «ко-

[1] Google Blog, май 2018 года, об их системе Duplex: ai.googleblog.com/2018/05/advances-in-semantictextual-similarity.html.

[2] WSJ Blog, 10 марта 2017 года: blogs.wsj.com/cio/2017/03/10/homo-deus-author-yuval-noah-harari-saysauthority-shifting-from-people-to-ai/.

нечный автомат» (finite state machine, FSM). Конечный автомат, который выводит последовательность новых символов в процессе функционирования, как функция `str.translate` в языке Python, называется «конечный автомат-преобразователь» (finite state transducer, FST). Возможно, вы уже создавали конечный автомат, даже не отдавая себе в этом отчета. Вы когда-нибудь писали регулярное выражение? Это тот тип конечного автомата, который мы будем использовать в следующем подразделе, чтобы показать вам один из возможных подходов к NLP: на основе паттернов.

Что, если вы решили поискать в банке памяти (базе данных) точно такую же строку битов, символов или слов и использовать один из ответов, которые другие люди и авторы применяли для этого утверждения раньше? Теперь представьте, что в этом утверждении была опечатка или какое-либо незначительное изменение. Наш бот сойдет с рельс. Биты не непрерывная величина, они не прощают ошибок — они либо совпадают, либо нет. Не существует очевидного способа найти сходство между двумя потоками битов, при котором учитывался бы их смысл. Биты для «хорошо» будут так же похожи на биты для «плохо», как и на биты для «нормально».

Перед тем как показать вам лучший вариант, посмотрим, как же работает этот подход. Создадим маленькое регулярное выражение для распознавания таких приветствий, как *Good morning, Rosa*, и ответа соответствующим образом — наш первый крошечный чат-бот!

1.4.1. Язык замков

Удивительно, но скромный кодовый замок на самом деле является простой машиной обработки языка. Если у вас инженерный склад ума, этот раздел многое для вас прояснит. Если же вам не нужны механические аналогии для понимания принципов работы алгоритмов и регулярных выражений, пропустите его.

После прочтения этого подраздела вы по-новому взглянете на кодовый замок своего велосипеда. Замок на школьном шкафчике, конечно, не способен прочитать и понять лежащие в этом шкафчике учебники, но знает язык замков. Такой замок может понять, когда вы пытаетесь «сказать» ему «пароль» — любую последовательность символов, соответствующую «грамматике» (паттерну) языка замка. Еще важнее то, что замок может сказать, соответствует ли «высказывание» особенно осмысленному утверждению, на которое есть только один правильный «ответ»: освободить защелку, сдвинуть U-образный засов, чтобы вы могли попасть в свой шкафчик.

Этот язык замка (регулярные выражения) особенно прост, но не настолько, чтобы его нельзя было применить для чат-бота. Он подходит для распознавания ключевой фразы или команды, чтобы разблокировать конкретное действие или поведение.

Например, мы хотели бы, чтобы наш чат-бот распознавал такие приветствия, как *Hello, Rosa*, и отвечал на них соответствующим образом. Этот тип языка, как и у замков, формальный. Потому что имеет строгие правила о составлении и интерпретировании допустимого высказывания. Если вы когда-либо писали математические уравнения/выражения на языках программирования, то, можно сказать, вы писали высказывания на формальном языке.

Формальные языки — подмножество естественных языков. Для многих высказываний на естественном языке можно найти соответствие и сгенерировать его с использованием грамматики формального языка, например регулярных выражений. Именно поэтому мы провели аналогию с механическим языком замков: «Щелк, вжж»[1].

1.4.2. Регулярные выражения

Регулярные выражения используют специальный вид (класс) грамматики формального языка, называемый регулярной грамматикой. У нее предсказуемое, доказуемое поведение, и в то же время регулярные выражения достаточно гибкие для приведения в действие некоторых самых сложных диалоговых движков и чат-ботов на рынке. Amazon Alexa и Google Now — использующие регулярные грамматики движки, основанные главным образом на паттернах. Сложные правила регулярной грамматики часто можно выразить одной строкой кода, называемой регулярным выражением. Для языка Python существуют эффективные фреймворки для чат-ботов, такие как *Will*, полагающиеся исключительно на этот тип языка для создания удобного и интересного поведения. Amazon Echo, Google Home и аналогичные помощники применяют регулярную грамматику с целью кодирования логики для большей части их взаимодействий с пользователем.

ПРИМЕЧАНИЕ

> Регулярные выражения, применяемые в Python и приложениях Posix (Unix), таких как grep, — ненастоящие регулярные грамматики. У них есть языковые и логические функции (просмотр вперед и назад), приводящие к поспешным выводам и рекурсии, недопустимым в регулярной грамматике. В результате регулярные выражения не могут быть доказуемо остановлены. В них иногда возможны фатальные сбои или бесконечное зацикливание[2].

Вы можете сказать: «Я слышал о регулярных выражениях. Использую *grep*. Но он подходит только для поиска!» И окажетесь правы. Они действительно используются в основном для поиска и сопоставления последовательностей. Однако все, что может найти совпадения в тексте, отлично подходит и для ведения диалога. Некоторые чат-боты, такие как Will, используют «поиск», чтобы найти последовательности символов в тех высказываниях пользователя, на которые они умеют реагировать. Эти распознанные последовательности запускают ответ в виде сценария, соответствующего данному конкретному совпадению регулярного выражения.

---

[1] Один из шести психологических принципов Р. Чалдини, описанных в его книге «Влияние»: changingminds.org/techniques/general/cialdini/click-whirr.htm.

[2] Stack Exchange не работал 30 минут 20 июля 2016 года из-за фатального сбоя регулярного выражения (http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016).

И это же регулярное выражение помогает извлекать полезную информацию из высказывания. Чат-бот может добавить этот кусочек информации в свою базу знаний о пользователе или о мире, который описывает пользователь.

Машину, которая обрабатывает подобный язык, можно рассматривать как формальный математический объект, называемый конечным автоматом или детерминированным конечным автоматом (deterministic finite automaton, DFA). Как вы можете видеть, конечные автоматы снова и снова встречаются в этой книге. В итоге вы получите хорошее представление о том, для чего они нужны, не углубляясь в теорию конечных автоматов и математику. Для тех, кто хочет понять немного больше об этих инструментах, на рис. 1.1 показано место конечных автоматов в многоуровневом мире различных автоматов (ботов). Примечание ниже объясняет более подробные детали формальных языков.

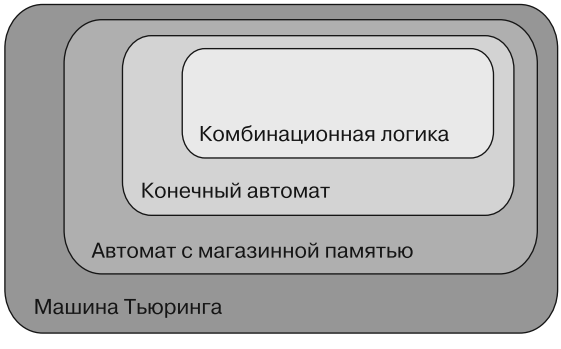


Рис. 1.1. Типы автоматов

Математическое объяснение формальных языков

Кайл Горман (Kyle Gorman) так описывает языки программирования.

- Большинство (если не все) языков программирования взято из класса контекстно свободных языков.
- Синтаксический разбор контекстно свободных языков производится с помощью контекстно свободных грамматик, обеспечивающих эффективный разбор.
- Регулярные языки также пригодны для эффективного синтаксического разбора и широко используются при вычислениях для сопоставления строк.
- Приложения сопоставления строк редко требуют выразительности контекстно свободных языков.
- Существует несколько классов формальных языков (далее приводятся в порядке уменьшения сложности) (https://ru.wikipedia.org/wiki/Иерархия_Хомского):
  - рекурсивно перечислимые;
  - контекстно зависимые;

- контекстно свободные;
- регулярные.

Естественные языки:

- нерегулярные[1];
- не контекстно свободные[2];
- не могут быть описаны никакой формальной грамматикой[3].

1.4.3. Простой чат-бот

Создадим быстрый и простой чат-бот. Он не будет обладать большими возможностями, и для его создания понадобится серьезно обдумать лингвистические нюансы английского языка. Вам также придется «зашить» в него регулярные выражения так, чтобы они соответствовали способам изложения мыслей, используемых людьми в устной речи. Не беспокойтесь, если думаете, что не сможете составить этот Python-код самостоятельно. Вам не придется думать о том, как люди могут что-то сказать, как мы делали это в данном примере. Вам даже не придется писать регулярные выражения (regex), чтобы создать чат-бот. Мы покажем, как создать собственный чат-бот, в последующих главах, не «зашивая» в него ничего. Современный чат-бот может обучиться путем чтения подборки текстов на английском. Как это сделать, мы расскажем далее в книге.

Такой сопоставляющий с паттернами чат-бот — строго контролируемый. Чат-боты, основанные на сопоставлении с паттерном, были обычным явлением до изобретения современных чат-ботов на основе машинного обучения. И один из вариантов приведенного здесь подхода сопоставления с паттернами используется в таких чат-ботах, как Amazon Alexa, и в других виртуальных помощниках.

А пока создадим FSM, регулярное выражение, которое может говорить на языке замков (регулярном языке). Мы могли бы запрограммировать его так, чтобы он понимал выражения на языке замков, такие как 01–02–03. Еще лучше, если бы он понимал приветствия вроде «Сезам, откройся» или «Привет, Роза». Важная особенность вежливого чат-бота — умение отвечать на приветствие.

В протоколе машинного взаимодействия мы бы описали простую процедуру установления связи с сигналом `ACK` (подтверждение) после каждого сообщения,

---

1 English is not a regular language: http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=20.

2 *Wintner S.* Is English context-free? http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=24.

3 См. веб-страницу 1.11. Formal and Natural Languages — How to Think like a Computer Scientist: Interactive Edition: http://interactivepython.org/runestone/static/CS152f17/GeneralIntro/FormalandNaturalLanguages.html.

передаваемого между двумя компьютерами. Но наши машины будут взаимодействовать с людьми, которые говорят что-то вроде «Доброе утро, Роза». Мы не хотим, чтобы чат-бот издавал какое-нибудь чириканье или писк либо отправлял сообщения ACK, как это происходит при синхронизации модема или HTTP-соединении в начале разговора/сеанса просмотра веб-страниц. Вместо этого подойдут регулярные выражения для распознавания нескольких различных приветствий в начале установления связи при разговоре:

В Python есть два «официальных» пакета регулярных выражений. Мы используем пакет re здесь только потому, что он устанавливается вместе со всеми версиями Python. Пакет regex поставляется только с последними версиями Python, и он гораздо мощнее, в чем вы сможете убедиться в главе 2

| означает «ИЛИ», а * — что предыдущий символ может встречаться 0 или более раз и при этом совпадать. Таким образом, наше регулярное выражение будет соответствовать приветствиям, которые начинаются с hi, hello или hey и за которыми следует любое количество пробельных символов, а затем неограниченное число букв

```
>>> import re
>>> r = "(hi|hello|hey)[ ]*([a-z]*)"
>>> re.match(r, 'Hello Rosa', flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 10), match='Hello Rosa'>
>>> re.match(r, "hi ho, hi ho, it's off to work ...", flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 5), match='hi ho'>
>>> re.match(r, "hey, what's up", flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 3), match='hey>
```

Игнорирование регистра символов — довольно распространенная техника. Это упрощает регулярные выражения

В регулярных выражениях можно указывать класс символов в квадратных скобках, а также использовать тире (-), чтобы указать диапазон символов, не набирая их по отдельности. Таким образом, регулярное выражение [a-z] будет соответствовать любой отдельной строчной букве, от a до z. Звездочка (*) после класса символов означает, что регулярному выражению будет соответствовать любое количество последовательных символов, относящихся к этому классу символов.

Детализируем наше регулярное выражение, чтобы ему соответствовало большее количество приветствий:

Регулярные выражения можно скомпилировать, чтобы не указывать параметры (флаги) при каждом использовании

```
>>> r = r"[^a-z]*([y]o|[h']?ello|ok|hey|(good[ ])?(morn[gin']{0,3}|"\
...     r"afternoon|even[gin']{0,3}))[\s,;:]{1,3}([a-z]{1,20})"
>>> re_greeting = re.compile(r, flags=re.IGNORECASE)
>>> re_greeting.match('Hello Rosa')
<_sre.SRE_Match object; span=(0, 10), match='Hello Rosa'>
>>> re_greeting.match('Hello Rosa').groups()
('Hello', None, None, 'Rosa')
>>> re_greeting.match("Good morning Rosa")
<_sre.SRE_Match object; span=(0, 17), match="Good morning Rosa">
>>> re_greeting.match("Good Manning Rosa")
```

Обратите внимание, что это регулярное выражение не может распознать (найти соответствие) слова с опечатками

```
>>> re_greeting.match('Good evening Rosa Parks').groups()
('Good evening', 'Good ', 'evening', 'Rosa')
>>> re_greeting.match("Good Morn'n Rosa")
<_sre.SRE_Match object; span=(0, 16), match="Good Morn'n Rosa">
>>> re_greeting.match("yo Rosa")
<_sre.SRE_Match object; span=(0, 7), match='yo Rosa'>
```

Наш чат-бот может разделять различные части приветствия на группы, но он не будет знать об известной фамилии Розы, так как у нас нет паттерна для сопоставления каких-либо символов после имени

СОВЕТ

Буква r перед кавычкой означает неформатированную строку, а не регулярное выражение. С помощью неформатированных строк Python можно отправлять обратную косую черту непосредственно в компилятор регулярных выражений без необходимости ставить двойную обратную косую черту ("\\") перед всеми специальными символами регулярных выражений, такими как пробелы ("\\ ") и фигурные скобки ("\\{ \\}").

В регулярном выражении в первой строке кода заключено очень много логики. Оно в состоянии обработать на удивление большое разнообразие приветствий. Стоит отметить, что оно пропустило написанное с опечаткой слово Manning. Такого рода проблемы делают NLP очень сложным процессом. В машинном обучении и медицинских диагностических тестах это называется ложноотрицательным результатом классификации. К сожалению, этому регулярному выражению соответствуют также совершенно несвойственные людям фразы — ложноположительные результаты, которые также ничего хорошего не несут. Наличие ошибок обоих видов значит, что наше регулярное выражение слишком строгое и свободное. В результате наш бот будет звучать немного глупо и механически. Нам предстоит провести большую работу по уточнению сопоставляемых фраз, чтобы чат-бот был более похожим на человека.

Даже несмотря на это утомительное занятие, нам вряд ли когда-нибудь удастся охватить весь сленг людей и их орфографические ошибки. К счастью, составление регулярных выражений вручную не единственный способ обучить чат-бота. В дальнейшем мы больше расскажем об этом. Мы используем данную технику только тогда, когда нужен точный контроль над поведением бота, например, при выдаче команд голосовому помощнику на вашем мобильном телефоне.

Продолжим и закончим работу над нашим наделенным лишь одним талантом чат-ботом, добавив генератор выходных данных. Бот должен что-то сказать. Используем функцию форматирования строк Python для создания шаблона ответа чат-бота:

```
>>> my_names = set(['rosa', 'rose', 'chatty', 'chatbot', 'bot',
...     'chatterbot'])
>>> curt_names = set(['hal', 'you', 'u'])
>>> greeter_name = ''
>>> match = re_greeting.match(input())
...
>>> if match:
...     at_name = match.groups()[-1]
...     if at_name in curt_names:
...         print("Good one.")
...     elif at_name.lower() in my_names:
...         print("Hi {}, How are you?".format(greeter_name))
```

Мы пока не знаем, кто общается с нашим чат-ботом, но на данном этапе это неважно

Если вы запустите этот маленький скрипт и скажете боту фразу Hello, Rosa, он ответит, спросив, как ваш день. При использовании немного более грубой формы обращения к чат-боту он ответит менее охотно, но не резко, чтобы поощрить вежливость[1]. Если вы упомянете кого-то еще, кто может следить за разговором по параллельному телефону или на форуме, бот промолчит и позволит общаться с тем, к кому вы обратились. Очевидно, что никто больше не следит за нашей строкой `input()`, но, будь эта функция внутри более крупного чат-бота, понадобилось бы учитывать подобные вещи.

Из-за ограничений вычислительных ресурсов первые исследователи NLP были вынуждены использовать вычислительные возможности своего человеческого мозга для разработки и настройки вручную сложных логических правил извлечения информации из строк на естественном языке. Это называется шаблонизированным подходом к NLP. Шаблоны не обязаны быть просто шаблонами последовательностей символов, как наше регулярное выражение. NLP также часто включает в себя шаблоны последовательностей слов, частей речи, а также другие, «более высокоуровневые» шаблоны. Основные стандартные блоки NLP, такие как стеммеры и токенизаторы, а также комплексные диалоговые NLP-системы (чат-боты), такие как ELIZA, были созданы именно так: с помощью регулярных выражений и сопоставлений с шаблонами. Искусное применение сопоставления с шаблонами обеспечивает элегантные шаблоны для NLP, охватывающие именно то, что нужно, без слишком большого количества строк в коде регулярного выражения.

КЛАССИЧЕСКАЯ ВЫЧИСЛИТЕЛЬНАЯ ТЕОРИЯ СОЗНАНИЯ

Этот классический подход сопоставления с образцом NLP основан на вычислительной теории сознания (computational theory of mind, CTM). CTM предполагает, что человекоподобные NLP могут быть реализованы на основе конечного набора логических правил, обрабатывающихся последовательно[2]. Достижения в области нейробиологии и NLP на рубеже веков привели к развитию коннекционистской теории сознания, позволяющей параллельную обработку естественного языка параллельными конвейерами, как это делается в искусственных нейронных сетях[3, 4].

---

1 Идея подобного разряжающего напряженность ответа появилась в книге Виктора Франкла (Viktor Frankl) «Человек в поисках смысла», описывающей его логотерапевтический (https://ru.wikipedia.org/wiki/Логотерапия) подход к психологии, и во многих популярных романах, где главный герой — ребенок, такой как Оуэн Мини, — достаточно мудр для ответа подобным образом на грубость.

2 Stanford Encyclopedia of Philosophy, Computational Theory of Mind (Стэнфордская философская энциклопедия, «Вычислительная теория сознания»): plato.stanford.edu/entries/computational-mind/.

3 Stanford Encyclopedia of Philosophy, Connectionism (Стэнфордская философская энциклопедия, «Коннекционизм»): plato.stanford.edu/entries/connectionism/.

4 Кристиансен (Christiansen) и Чатер (Chater), 1999, Southern Illinois University: crl.ucsd.edu/~elman/Bulgaria/christiansen-chater-soa.pdf.

В главе 2 вы больше узнаете об основанных на шаблонах подходах к стеммингу и токенизации, таких как стеммер Портера или токенизатор Treebank. Но в последующих главах мы воспользуемся возможностями современных вычислительных ресурсов, а также нашими большими наборами данных для сокращения объема кропотливого программирования и настройки вручную.

Если вы новичок в регулярных выражениях и хотите узнать больше, почитайте приложение Б или онлайн-документацию по регулярным выражениям Python. Но вам пока не нужно в них разбираться. Мы продолжим приводить примеры регулярных выражений по мере использования их в качестве стандартных блоков нашего конвейера NLP. Так что не волнуйтесь, если они выглядят для вас китайской грамотой. Человеческий мозг довольно хорошо умеет обобщать на основе ряда примеров, и мы уверены, что к концу этой книги все станет ясно. Оказалось, машины тоже могут обучаться подобным образом.

1.4.4. Другой вариант

Существует ли статистический или машинный подход, который мог бы заменить основанный на шаблонах? Будь у нас достаточно данных, могли бы мы сделать что-то другое? А если бы у нас была гигантская база данных, содержащая сеансы диалогов между людьми, высказывания и ответы для тысяч или даже миллионов разговоров? Один из способов создания чат-бота — поиск в этой базе данных тех же самых символов, которые наш пользователь бота только что «сказал» ему. Разве мы не можем использовать один из ответов на это утверждение, которое другие озвучивали раньше?

Но представьте, как одна опечатка или ошибка в предложении могут сбить с толку наш бот. Битовые и символьные последовательности дискретны. Они либо совпадают, либо нет. Вместо этого мы хотели бы, чтобы бот мог измерить разницу *смыслов* последовательностей символов.

Когда мы применяем совпадения последовательности символов для измерения расстояния между фразами на естественном языке, мы часто ошибаемся. Фразы с одинаковым значением, как *good* и *okay*, нередко могут иметь разные последовательности символов и большие расстояния в случае подсчета посимвольных совпадений для измерения расстояния. И последовательности с совершенно разными значениями, например *bad* и *bar*, могут оказаться слишком близки друг к другу, если мы станем использовать метрики, предназначенные для измерения расстояний между числовыми последовательностями. Такие метрики, как расстояние Жаккара, расстояние Левенштейна и евклидово векторное расстояние, иногда могут добавить немного «нечеткости», чтобы чат-бот не «спотыкался» на незначительных орфографических ошибках или опечатках. Стоит отметить, что эти метрики не в состоянии уловить суть взаимосвязи между двумя разнородными строками. И они иногда сглаживают небольшие различия в правописании, которые на самом деле не опечатки, как те же *bad* и *bar*.

Метрики расстояния, разработанные для числовых последовательностей и векторов, полезны лишь для немногих приложений NLP, таких как средства проверки правописания и распознавания имен собственных. Поэтому мы используем их только тогда, когда они имеют смысл. Но для приложений NLP, в которых нас

больше интересует смысл естественного языка, а не его правописание, существуют более подходящие подходы. Мы воспользуемся векторными представлениями слов и текста на естественном языке и некоторыми метриками расстояния между этими векторами для нужных нам приложений NLP. Мы покажем каждый подход один за другим по мере нашего разговора об этих различных векторных представлениях и видах приложений, с которыми они используются.

Мы не задержимся надолго в этом запутанном двоичном мире логики. Представьте, что вы — знаменитая взломщица кодов времен Второй мировой войны Мэвис Бэйти (Mavis Batey) из Блетчли-парк. Вам только что передали двоичное сообщение в виде кода Морзе, перехваченное из разговора между двумя немецкими офицерами. Это может стать ключом к победе в войне. С чего бы начать? Первым шагом в анализе должно быть получение какой-либо статистической информации об этом потоке битов для поиска закономерностей. Сначала понадобится таблица азбуки Морзе (или таблица ASCII в нашем случае) для назначения букв каждой группе битов. Затем, если символы для вас все еще бессмысленны, вы могли бы начать их подсчитывать, искать короткие последовательности в словаре всех слов, которые видели раньше, и помечать совпадения. Или отметить в каком-нибудь другом журнале, чтобы указать, в каком сообщении встретилось слово, создав энциклопедический указатель для всех прочтенных документов. Эта коллекция документов называется *корпусом*, а коллекция перечисленных в нашем указателе слов или последовательностей — *лексиконом*.

Если повезет и мы не находимся в состоянии войны, а сообщения, на которые мы смотрим, не зашифрованы, то можно разглядеть закономерности в частотностях этих немецких слов, отражающие частотности английских слов, используемых для передачи похожих сообщений. В отличие от криптографа, пытающегося расшифровать перехваты немецкой морзянки, мы знаем, что символы имеют постоянное значение, которое не меняется при каждом нажатии клавиши, чтобы сбить нас с толку. Такой утомительный подсчет символов и слов — как раз то, что компьютер с легкостью может делать. Удивительно, но этого почти достаточно, чтобы машина могла притвориться, что понимает наш язык. Она может даже совершать математические действия над этими статистическими векторами, соответствующие нашему человеческому пониманию данных фраз и слов. Когда мы расскажем вам в последующих главах, как научить машину нашему языку с помощью Word2Vec, это может показаться волшебством. Но нет — просто математика и вычисления.

Однако задумаемся на секунду, какая информация потерялась при попытке подсчета всех слов в получаемых нами сообщениях. Мы распределяем слова по группам (bucket) и сохраняем их в виде битовых векторов, так же как сортировщик монет или жетонов, направляющий различные виды жетонов в одну или другую сторону в каскаде решений. Наша сортировочная машина должна учитывать сотни тысяч, если не миллионы, возможных «номиналов» токенов. По одному на каждое возможное слово, которое может использовать докладчик или автор. Фраза, предложение или документ, который мы вводим в нашу машину для сортировки токенов, будут отображаться внизу, где у нас есть «вектор», соответствующий количеству токенов в каждом слоте. Большинство наших значений равно нулю даже для больших документов с разнообразной терминологией.

Но мы еще не потеряли никаких слов. Что же пропало? Могли бы вы как человек понять документ, представленный вам в виде количеств всех возможных слов в вашем языке без какой-либо их последовательности или упорядоченности? Сомневаемся. Но в коротком предложении или твите вы, вероятно, сможете переставить слова в нужном порядке и выяснить их смысл.

Вот так наш сортировщик жетонов встраивается в конвейер NLP сразу после токенизатора (см. главу 2). Мы включили в наш эскиз сортировщика механических токенов фильтр стоп-слов, а также фильтр редких слов. Строки подаются сверху, а векторы мультимножеств слов зависят от высоты «стопок» токенов внизу.

Машины могут достаточно хорошо справляться с такими коллекциями слов и извлекать большую часть информации даже из довольно длинных документов. Каждый документ после сортировки и подсчета токенов может быть представлен в виде вектора, последовательности целых чисел для каждого слова или токена в этом документе. Первый черновой пример приведен на рис. 1.2, а в главе 2 вы найдете более полезные структуры данных для векторов мультимножеств слов.

Это наша первая модель векторного пространства для языка. Группы и числа, которые они содержат для каждого слова, представлены в виде длинных векторов с множеством нулей и несколькими единицами или двойками, разбросанных повсюду, где встречалось слово из этой группы. Все способы объединения слов для создания этих векторов называются *векторным пространством* (vector space). Взаимосвязи между векторами в данном пространстве составляют нашу модель, которая пытается предсказать сочетания этих слов, встречающиеся в наборе различных последовательностей слов (обычно предложений или документов). В Python мы можем представить эти разреженные (в основном пустые) векторы (списки чисел) в виде словарей. Класс Python `Counter` — особый вид словаря, распределяющий объекты (включая строки) по группам и подсчитывающий их так, как нужно:

```
>>> from collections import Counter

>>> Counter("Guten Morgen Rosa".split())
Counter({'Guten': 1, 'Rosa': 1, 'morgen': 1})
>>> Counter("Good morning, Rosa!".split())
Counter({'Good': 1, 'Rosa!': 1, 'morning,': 1})
```

Можно придумать способы очистить эти токены. Мы сделаем это в следующей главе. Кажется, что эти разреженные многомерные векторы (много ячеек, по одной на каждое возможное слово) не очень удобны для обработки языка. Но они подходят для некоторых продвинутых инструментов, таких как спам-фильтры, которые мы обсудим в главе 3.

Можно себе представить поочередный ввод в эту машину всех документов, высказываний, предложений и даже отдельных слов, которые мы можем найти. Или подсчитать токены в каждом слоте внизу после обработки каждого из высказываний и назвать это векторным представлением данного высказывания. Все возможные векторы, которые машина может создать таким образом, называются *векторным пространством*, а такая модель документов, высказываний и слов — *моделью векторного пространства*. Это позволяет использовать линейную алгебру для операций

над подобными векторами и вычислять, например, расстояния и статистические данные относительно высказываний на естественном языке, что помогает решать гораздо более широкий круг проблем, не прибегая к написанию большого количества кода вручную.

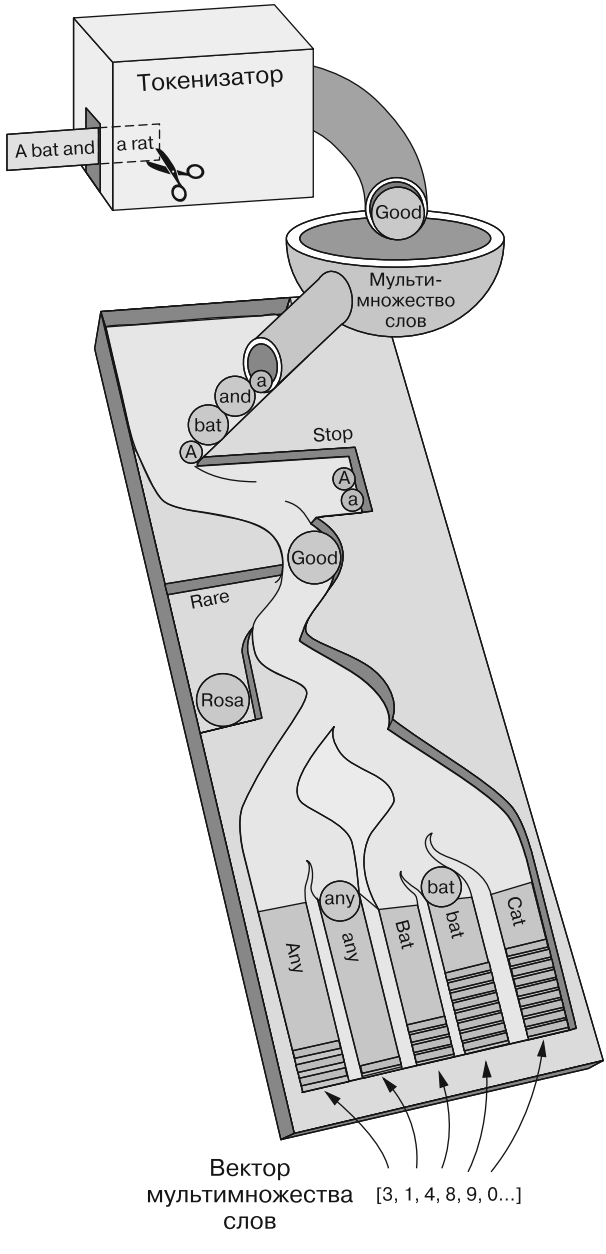


Рис. 1.2. Лоток для сортировки токенов

Один статистический вопрос, который часто задают относительно векторных последовательностей мультимножеств слов: «Какая комбинация слов с наибольшей вероятностью будет следовать за данным мультимножеством?» Или, что еще лучше, при вводе пользователем последовательности слов: «Какое мультимножество символов в нашей базе данных ближе всего к введенному пользователем?» Это поисковый запрос. Входные слова могут быть введены в поле поиска, а ближайший вектор мультимножества слов соответствует искомому документу или веб-странице. Способности эффективно ответить на эти два вопроса достаточно для создания чатбота на основе машинного обучения, который мог бы совершенствоваться по мере получения все новых и новых данных.

Возможно, эти векторы непохожи на те, с которыми вы когда-либо работали. Их размерность очень велика. Они могут насчитывать миллионы измерений в случае словаря триграмм, вычисленного на основе большого корпуса.

1.5. Краткая экскурсия по гиперпространству

В главе 3 мы покажем, как упаковать слова в меньшее число векторных измерений, чтобы смягчить проклятие размерности и, возможно, обернуть его в свою пользу. Проекция этих векторов друг на друга для определения их попарных расстояний будет разумной оценкой сходства их *смыслов*, а не просто их статистик использования слов. Эта векторная метрика расстояния называется коэффициентом Отиаи, о котором мы поговорим в главе 3, а затем покажем его истинную силу на векторах тем пониженной размерности в главе 4. Мы даже можем спроецировать («производить вложение» — более точный термин) эти векторы на 2D-плоскость, чтобы взглянуть на них на графиках и диаграммах и понять, может ли человеческий мозг найти в них закономерности. Затем можно научить компьютер распознавать данные паттерны и действовать на их основе так, чтобы отражать основной смысл породивших эти векторы слов.

Представьте себе все возможные твиты, сообщения и предложения, которые могут написать люди. Несмотря на то что мы часто повторяемся, возможностей все равно еще много. Когда каждый из этих токенов рассматривается как отдельное, уникальное измерение, непонятно, что смысл фразы *Good morning, Hobs* имеет сходство с *Guten Morgen, Hannes*. Нам нужно создать модель векторного пространства сообщений, причем модель пониженной размерности, для маркировки этих сообщений набором непрерывных (с плавающей точкой) значений. Сообщения и слова можно оценивать относительно таких свойств, как значение и тональность. Мы могли бы спросить, например, следующее.

- Какова вероятность, что данное сообщение является вопросом?
- Какова вероятность, что это сообщение относится к какому-то человеку?
- Какова вероятность, что это сообщение касается меня?

- Насколько грубым или приятным оно кажется?
- Нужно ли на него отвечать?

Подумайте обо всех тех оценках, которые мы могли бы дать высказываниям. Мы могли бы упорядочить эти рейтинги и «вычислить» их для каждого высказывания, чтобы сформировать «вектор» для них. Список рейтингов (измерений) для набора высказываний будет намного меньше, чем количество возможных высказываний, а значения для синонимов должны быть одинаковыми для всех наших вопросов.

Можно запрограммировать машину реагировать на такие рейтинговые векторы. Мы можем еще больше упростить и обобщить векторы путем сбора в группу (кластеризации) высказываний, близких по одним измерениям, но не по другим.

Но как компьютер может присвоить значения каждому из этих измерений векторов? Допустим, можно упростить вопросы для векторного измерения до: «Содержит ли высказывание слово *good*?», «Содержит ли высказывание слово *morning*?» и т. д. Как видите, легко придумать миллион или около того вопросов, дающих числовые значения, которые компьютер может присвоить фразе. Это первая реальная модель векторного пространства, называемая бинарной векторной моделью языка, сумма унитарных (one-hot) векторов. Теперь понятно, почему компьютеры только сейчас стали достаточно мощными для понимания естественного языка. Миллионы миллиономерных векторов, которые люди могут генерировать, просто не вычислялись (!) на суперкомпьютере 1980-х, но не представляют проблемы для обычных ноутбуков в XXI веке. Но не только чистая мощность и возможности аппаратного обеспечения сделали NLP реальностью. Инкрементные алгоритмы линейной алгебры с постоянным объемом используемой оперативной памяти были последней частью головоломки, которая позволила машинам взломать шифр естественного языка.

Есть еще более простое, но гораздо более широкое представление, которое можно использовать в чат-боте: что, если наши векторные пространства будут полностью описывать точную последовательность символов? Они бы содержали ответы на такие вопросы, как: «Первая буква А? Или Б? ...Вторая буква А?» — и т. д. Преимущество этого вектора в том, что он сохраняет всю информацию исходного текста, включая порядок символов и слов. Представьте себе механическое пианино, которое может играть только одну ноту в один момент времени и в котором может быть 52 или более возможных нот. «Ноты» для такого механического пианино естественного языка — 26 прописных и строчных букв плюс любая пунктуация, которую пианино должно уметь «играть». Рулон перфоленты будет не намного шире, чем страница с нотами у настоящего пианиста, и количество нот в некоторых длинных фортепианных песнях не превышает количество символов в небольшом документе. Но это унитарное представление, кодирующее последовательность символов, удобно для записи и последующего воспроизведения точного фрагмента, а не для составления чего-то нового или извлечения сути. Сравнить рулон перфоленты механического пианино для одной песни с рулоном для другой не так уж просто. И это представление длиннее исходного ASCII-кодированного представления документа. Количество возможных представлений документов резко возросло, чтобы можно было сохранить информацию о каждой последовательности символов. Мы сохранили порядок символов и слов, но повысили размерность нашей задачи NLP.

Эти представления документов плохо объединяются в нашем векторном мире, основанном на символах. Русский математик Владимир Левенштейн предложил блестящий подход для быстрого нахождения сходства между последовательностями (цепочками символов) в мире векторов. Алгоритм Левенштейна позволил создать несколько удивительно забавных и полезных чат-ботов с помощью только упрощенного механического взгляда на язык. Но настоящая магия произошла, когда мы выяснили, как сжать/вложить эти пространства более высокой размерности в пространство более низкой размерности нечетких смыслов или векторов тем. Мы раскроем некоторые секреты фокусника в главе 4, когда поговорим о латентно-семантической индексации и латентном размещении Дирихле, двух методиках создания гораздо более плотных и осмысленных векторных представлений высказываний и документов.

1.6. Порядок слов и грамматика

Порядок слов важен. Те правила, которые определяют порядок слов в их последовательности (например, в предложении), называются грамматикой языка. Именно грамматикой пренебрегает подход на основе набора или вектора слов, рассмотренный в предыдущих примерах. К счастью, в большинстве коротких фраз и даже во многих полных предложениях это приближение вектора слов нормально работает. Если вы просто хотите закодировать общий смысл и тональность короткого предложения, порядок слов не так уж важен. Взгляните на порядок слов в нашем примере `Good morning Rosa`:

```
>>> from itertools import permutations

>>> [" ".join(combo) for combo in\
...     permutations("Good morning Rosa!".split(), 3)]
['Good morning Rosa!',
 'Good Rosa! morning',
 'morning Good Rosa!',
 'morning Rosa! Good',
 'Rosa! Good morning',
 'Rosa! morning Good']
```

Если попытаться интерпретировать каждую из этих строк по отдельности (не глядя на остальные), вы решите, что тональности и смысл у них схожи. Можно даже заметить прописную букву в слове *Good* и мысленно поместить его в начало фразы. Но вы также можете подумать, что *Good Rosa* — имя собственное, например название ресторана или цветочного магазина. Тем не менее умный чат-бот или умная женщина, жившая в 1940-е годы в Блетчли-парке, вероятно, ответили бы на любую из этих шести перестановок одним и тем же безобидным приветствием *Good morning, my dear General.*

Попробуем мысленно опробовать эту схему на более длинной и сложной фразе, логическом высказывании, где порядок слов важен:

```
>>> s = """Find textbooks with titles containing 'NLP',
...     or 'natural' and 'language', or
...     'computational' and  'linguistics'."""
```

```
>>> len(set(s.split()))
12
>>> import numpy as np
>>> np.arange(1, 12 + 1).prod()  # factorial(12) = arange(1, 13).prod()
479001600
```

Количество перестановок выросло с `factorial(3) == 6` в нашем простом приветствии до `factorial(12) == 479001600` в длинном высказывании! Логика в порядке слов важна для любой машины, желающей правильно ответить на запрос. Хотя обычные приветствия при обработке мультимножеств слов не искажаются, более сложные высказывания могут потерять большую часть смысла, если их преобразовать в мультимножество. Использование мультимножества слов не лучший способ начать обработку запроса к базе данных, как и запрос на естественном языке в предыдущем примере.

Независимо от того, написано ли высказывание (оператор) на формальном языке программирования, таком как SQL, или на неформальном естественном языке вроде английского, порядок слов и грамматика важны, особенно когда с помощью высказывания нужно передать логические взаимосвязи. Поэтому компьютерным языкам необходимы жесткие грамматические и синтаксические анализаторы. К счастью, последние достижения в синтаксических анализаторах естественного языка сделали возможным извлечение синтаксических и логических связей из естественного языка с удивительной точностью (более 90 %)[1]. В последующих главах мы покажем, как применять для распознавания этих взаимосвязей такие пакеты, как `SyntaxNet` и `SpaCy`.

Как и в примере приветствия в Блетчли-парке, даже если логическая интерпретация высказывания не зависит от порядка слов, иногда путем анализа этого порядка слов можно обнаружить нюансы смысла, что может способствовать более содержательным ответам. Эти более глубокие слои обработки естественного языка мы обсудим в следующем разделе. В главе 2 продемонстрирован трюк для добавления некоторой информации, заключающейся в порядке слов, в наше представление векторов слов. В ней также показано, как уточнить черновой токенизатор, использовавшийся в предыдущих примерах (`str.split()`), для более точного распределения слов по более подходящим слотам в векторе слов, чтобы строки `good` и `Good` попали в одну группу, а для таких токенов, как `rosa` и `Rosa`, но не `Rosa!`, были выделены отдельные группы.

1.7. Конвейер чат-бота на естественном языке

Конвейер NLP, необходимый для создания диалогового механизма (чат-бота), аналогичен конвейеру для построения системы формирования ответов на вопросы, описанной в *Taming Text* (Manning, 2013)[2]. Однако некоторые алгоритмы, перечисленные в пяти блоках подсистем, могут оказаться для вас в новинку. Мы поможем

[1] Сравнение точности синтаксических анализаторов SpaCy (93 %), SyntaxNet (94 %), Stanford's CoreNLP (90 %) и других доступно на spacy.io/docs/api/.

[2] Ingersol, Morton and Farris: www.manning.com/books/taming-text.

реализовать эти алгоритмы на Python, чтобы выполнить различные задачи NLP для большинства приложений, включая создание чат-ботов.

Чат-бот требует четыре вида обработки, а также базы данных для запоминания прошлых высказываний и ответов. Каждый из этих этапов может содержать один или несколько алгоритмов обработки, действующих параллельно или последовательно (рис. 1.3).

- *Синтаксический разбор* — выделение признаков (структурированных числовых данных) из текста на естественном языке.
- *Анализ* — создание и комбинация элементов с целью получения показателей тональности, грамматической правильности и семантики текста.
- *Генерация* — составление возможных ответов с использованием паттернов, средств поиска или языковых моделей.
- *Выполнение* — подготовка высказываний на основе истории и целей разговора и выбор последующего ответа.

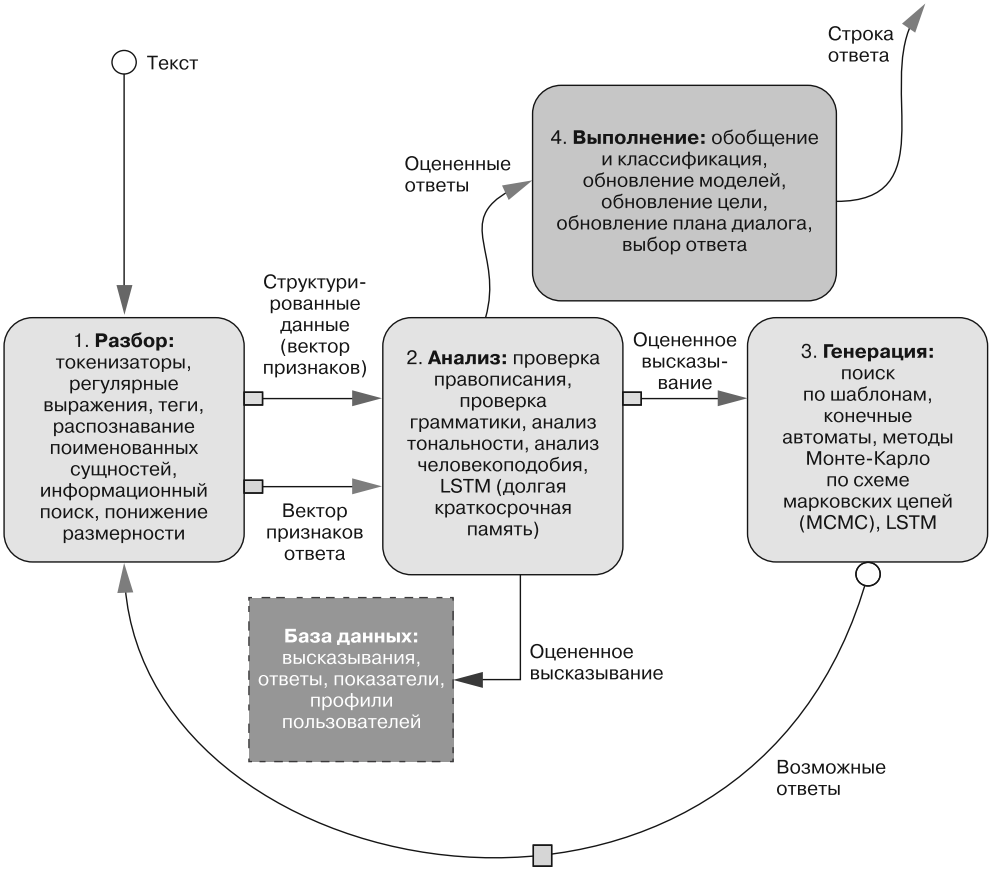


Рис. 1.3. Рекуррентный конвейер чат-бота

Каждый этап можно реализовать по одному или нескольким алгоритмам, перечисленным в соответствующих полях на блок-схеме. Мы покажем, как использовать Python для достижения оптимального уровня производительности на каждом из этапов обработки, и несколько альтернативных подходов к реализации этих пяти подсистем.

Большинство чат-ботов содержат элементы всех пяти подсистем (четыре этапа обработки, а также база данных). Однако многие приложения требуют только простых алгоритмов для выполнения многих из этих шагов. Одни боты лучше отвечают на фактологические вопросы, а другие лучше генерируют длинные, сложные, убедительно напоминающие человеческие ответы. Каждая из этих возможностей требует разных подходов. Мы покажем методы для всех.

Кроме того, глубокое обучение и ориентированное на работу с данными программирование (машинное обучение или вероятностное моделирование языка) привели к быстрому росту разнообразия возможных приложений NLP и чат-ботов. Этот ориентированный на работу с данными подход позволяет увеличить сложность конвейера NLP, предоставляя ему все большие объемы данных в требуемой предметной области. Если же будет открыт новый, более эффективный подход к машинному обучению, позволяющий еще лучше использовать эти данные, с более эффективным обобщением или регуляризацией модели, станет реальностью еще больший скачок возможностей.

Конвейер NLP для чат-бота, показанный на рис. 1.3, содержит все стандартные блоки для большинства способов использования NLP, описанных в начале главы. Как и в *Taming Text*, конвейер разбивается на четыре основные подсистемы (этапа). Мы явным образом обращаемся к базе данных для записи данных, необходимых для каждого из этих этапов, и постоянного хранения настроек и тренировочных наборов. Благодаря этому возможно пакетное или производимое в режиме реального времени повторное обучение каждого из этапов, по мере взаимодействия чат-бота с окружающим миром. Мы также показали цикл обратной связи для наших сгенерированных текстовых ответов, чтобы наши ответы могли обрабатываться теми же алгоритмами, которые использовались для обработки пользовательских высказываний. Показатели ответов или признаки могут быть затем объединены в целевой функции для оценки и выбора наилучшего возможного ответа в зависимости от плана или целей чат-бота для диалога. Книга посвящена настройке этого конвейера NLP для бота, но также можно увидеть аналогию с задачей NLP, связанной с поиском текста, — возможно, наиболее распространенным вариантом применения NLP. И наш конвейер чат-бота, безусловно, подходит для приложения, отвечающего на вопросы, которое было в центре внимания *Taming Text*.

Применение этого конвейера для финансового прогнозирования или бизнес-аналитики, наверное, не столь очевидно. Но представьте, какие признаки генерирует аналитическая часть вашего конвейера. Можно оптимизировать эти признаки, полученные в результате анализа или генерации признаков для конкретного финансового или бизнес-прогноза. Таким образом, они могут помочь вам учесть данные на естественном языке в конвейере машинного обучения для прогнозирования. Хотя эта книга сосредоточена на создании чат-бота, в ней вы найдете инструменты, необходимые для широкого спектра задач NLP: от поиска до финансового прогнозирования.

Один из элементов обработки на рис. 1.3, который обычно не используется в системах поиска, прогнозирования или формирования ответов на вопросы, — *генерация* ответов на естественном языке. Это главная особенность чат-ботов. Тем не менее этап генерации текста часто включается в такой вид применения NLP, как поисковая система, и может дать ей большое конкурентное преимущество. Возможность консолидации (автоматического реферирования) результатов поиска выигрышна для многих популярных поисковых систем (DuckDuckGo, Bing и Google). И вы можете себе представить, как важно, чтобы механизм финансового прогнозирования мог генерировать высказывания, твиты или целые статьи на основе имеющих важное практическое значение бизнес-событий, обнаруживаемых в потоках данных на естественном языке из социальных сетей и новостных лент.

В следующем разделе показано, как объединить слои такой системы для повышения уровня сложности и количества возможностей на каждом этапе конвейера NLP.

1.8. Углубленная обработка

Этапы конвейера обработки естественного языка можно рассматривать как слои, подобные слоям сети прямого распространения. Глубокое обучение связано с созданием более сложных моделей и поведения путем добавления дополнительных слоев обработки к традиционной двухслойной архитектуре модели машинного обучения с выделением признаков для дальнейшего моделирования. В главе 5 мы объясним, как нейронные сети помогают распределять обучение по слоям с помощью метода обратного распространения ошибок от выходных слоев к входным. Однако здесь мы поговорим о верхних слоях и о том, что можно сделать, обучая каждый слой независимо от других.

Верхние четыре слоя на рис. 1.4 соответствуют первым двум этапам конвейера чат-бота (выделение и анализ признаков) из предыдущего раздела. Например, частеречная разметка (POS-разметка) — это один из способов генерации признаков на этапе «Анализ» конвейера нашего чат-бота. POS-метки генерируются автоматически по умолчанию конвейером *SpaCY*, который включает все четыре верхних слоя в этой диаграмме. Разметка POS обычно выполняется с помощью конечного преобразователя, подобного методам, используемым в пакете `nltk.tag`.

Два нижних слоя (взаимосвязи между сущностями и база знаний) заполняют базу данных информацией (знаниями) об определенной предметной области. Информацию, извлеченную из конкретного высказывания или документа с использованием всех этих шести слоев, можно затем сочетать с этой базой данных для выполнения вывода. *Выводы* — это логические экстраполяции на основе набора условий, обнаруженных в среде, например, логика, содержащаяся в высказываниях пользователя чат-бота. Такой вид механизма вывода в более глубоких слоях этой диаграммы считается предметной областью искусственного интеллекта, где машины могут делать выводы о своем мире и использовать их для принятия логических решений. Тем не менее боты могут принимать разумные решения без этой базы знаний, с помощью только алгоритмов нескольких верхних уровней. И в результате сочетания таких решений возможно поведение, удивительно похожее на поведение человека.

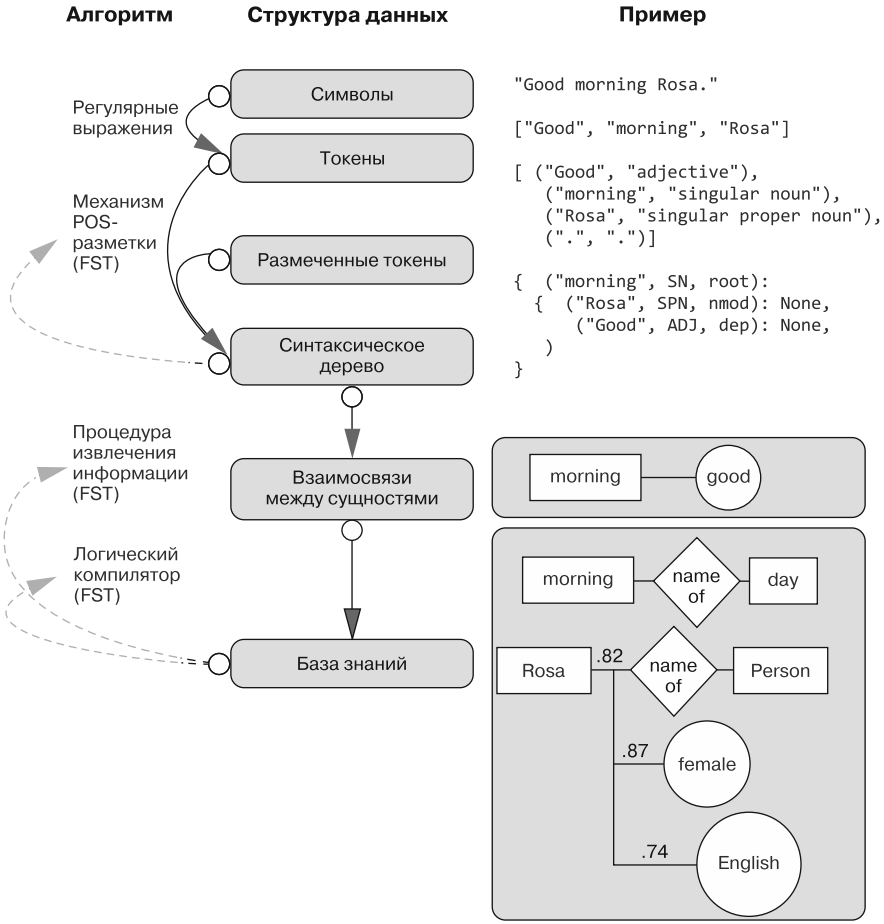


Рис. 1.4. Пример слоев конвейера NLP

Приложения

Криптография, сжатие, исправление орфографии, предиктивный ввод текста, поиск, диалог (чат-бот)

Поиск, стилистика, спам-фильтр, анализ тональностей, математика word2vec, семантический поиск, диалог (чат-бот)

Исправление орфографии и грамматики, стилистика, диалог (чат-бот)

Формирование ответов на вопросы, стилистика, сложный диалог, исправление грамматики, тренировка письма

Извлечение и вывод знаний, медицинская диагностика, формирование ответов на вопросы, игры

Доказательство теорем, логические выводы, запросы к базам данных на естественном языке, искусственный интеллект общего уровня (AGI)

В следующих главах мы углубимся в несколько верхних уровней NLP. Три таких уровня — это все, что требуется для осмысленного анализа тональностей и семантического поиска, а также для создания чат-ботов, имитирующих человека. Фактически можно создать полезный и интересный бот, используя лишь один слой обработки, только непосредственно текст (последовательность символов) в качестве признаков для языковой модели. Бот, который выполняет только сопоставление строк и поиск, способен вести убедительную беседу, если получит достаточно примеров высказываний и ответов.

Например, проект с открытым исходным кодом *ChatterBot* упрощает этот конвейер, вычисляя строковое «расстояние редактирования» (расстояние Левенштейна) между высказываниями на входе и высказываниями из его базы данных. Если его база данных, хранящая пары «высказывание — ответ», содержит искомое высказывание, то соответствующий ответ (из ранее усвоенного диалога людьми или машинами) может повторно использоваться в качестве ответа на последнее высказывание пользователя. Для этого конвейера требуется только третий шаг (генерация) из всего конвейера нашего чат-бота. На этом этапе для нахождения наилучшего ответа нужен алгоритм поиска методом полного перебора. С помощью этой простой методики (не требующей токенизации или генерации признаков) *ChatterBot* может убедительно поддерживать диалог как диалоговая система Salvius — механический робот, созданный из отдельных кусков Гюнтером Коксом[1].

Will — это чат-фреймворк Python Стивена Скочена (Steven Skoczen) с открытым исходным кодом и совершенно другим подходом[2]. Will можно обучать реагировать на высказывания только путем программирования его с помощью регулярных выражений. Это трудоемкий и мало использующий данные подход к NLP. Такой, основанный на грамматике, подход особенно эффективен для систем, формирующих ответы на вопросы и выполняющих задания ботов-помощников, например Lex, Siri и Google Now. Системы преодолевают «хрупкость» регулярных выражений, применяя нечеткие регулярные выражения[3] и используя другие методы поиска приближенных грамматических соответствий. Нечеткие регулярные выражения находят не точные совпадения, а наиболее близкие грамматические совпадения среди списка возможных правил (регулярных выражений), игнорируя при этом некоторое предельно допустимое количество ошибок вставки, удаления и замены. Однако увеличение сложности поведения для основанного на грамматике чат-бота требует много усилий по разработке. Даже самые продвинутые чат-боты на основе грамматики, созданные и поддерживаемые некоторыми из крупнейших корпораций на планете (Google, Amazon, Apple, Microsoft), находятся на среднем уровне IQ.

---

1 ChatterBot от Гюнтера Кокса и др. на github.com/gunthercox/ChatterBot.

2 См. страницу на GitHub о Will — чат-боте, созданном Стивеном Скоченом для HipChat и сообщества HipChat (github.com/skoczen/will). В 2018 году он был обновлен для интеграции с Slack.

3 Пакет regex языка Python обратно совместим с re и, помимо прочих возможностей, добавляет нечеткость. В будущем он заменит пакет re (pypi.python.org/pypi/regex). Аналогично TRE agrep или approximate grep (github.com/laurikari/tre) является альтернативой приложения командной строки UNIX grep.

С помощью даже поверхностной обработки естественного языка можно сделать много интересных вещей. Причем не нужно практически никакого человеческого контроля (маркировки или подбора текста). Можно позволить машине просто постоянно усваивать информацию из окружающей среды (потока слов, извлекаемого из Twitter или другого источника)[1]. Как это сделать, мы покажем в главе 6.

1.9. IQ естественного языка

Подобно человеческому интеллекту, мощность конвейера NLP не может быть легко измерена с помощью одного показателя IQ без учета различных аспектов умственных способностей. Распространенным способом оценить возможности роботизированной системы является измерение сложности поведения и определение степени необходимого контроля со стороны человека. Но для конвейера NLP цель в том, чтобы создать системы, которые полностью автоматизируют обработку естественного языка без необходимости человеческого надзора (как только модель обучена и развернута). Таким образом, более корректная система координат, в которой измеряется IQ, должна охватывать широту и глубину сложности конвейера естественного языка.

Потребительский чат-бот или виртуальный помощник, такой как *Alexa* или *Allo*, обычно разрабатываются так, чтобы обладать чрезвычайно широкими знаниями и возможностями. Однако логика, используемая для ответа на запросы, зачастую поверхностна и состоит из набора кодовых фраз, приводящих к одинаковому ответу с помощью одной ветви принятия решения `if-then`. *Alexa* (и лежащий в его основе движок Lex) ведет себя как однослойное, плоское дерево операторов (`if`, `elif`, `elif`...)[2]. У Google Dialogflow (который был разработан независимо от Google Allo и Google Assistant) есть аналогичные инструменты: Amazon Lex, Contact Flow и Lambda, но без «перетаскиваемого» пользовательского интерфейса для проектирования диалогового дерева.

С другой стороны, конвейер Google Translate (или любая аналогичная система машинного перевода) опирается на многоуровневую иерархию выделителей признаков, деревьев решений и графов знаний, соединяющих фрагменты знаний о мире. Иногда эти процедуры выделения признаков, деревья решений и графы знаний явно программируются в системе, как на рис. 1.4. Другим подходом, позволяющим быстро обогнать этот «запрограммированный вручную» конвейер, является ориентированный на данные подход глубокого обучения. Процедуры выделения признаков для глубоких нейронных сетей обучаются, а не программируются вручную, но им требуется гораздо больше тренировочных данных для достижения той же эффективности, что и у специально разработанных алгоритмов.

---

[1] Простые нейронные сети часто используются для выделения признаков без учителя из последовательностей символов и слов.

[2] Более сложная логика и поведение теперь стали возможны при bcgbb в диалоговом дереве AWS Contact Flow функций из AWS Lambda. См. Creating Call Center Bot with AWS Connect по адресу greenice.net/creating-callcenter-bot-aws-connect-amazon-lex-can-speak-understand.

Вам нужно использовать оба подхода (нейронные сети и вручную запрограммированные алгоритмы) по мере построения конвейера NLP для чат-бота, способного общаться в рамках заданной области знаний. Так вы получите навыки для выполнения задач обработки естественного языка в вашей отрасли промышленности или предметной области. Попутно вы, вероятно, получите представление о том, как расширить сферу возможностей конвейера NLP. Рисунок 1.5 демонстрирует место чат-бота среди уже существующих систем обработки естественного языка. Представьте чат-боты, с которыми вы уже общались. Как вы думаете, где их место на таком графике? Пытались ли вы оценить их интеллект, задавая сложные вопросы или проводя что-то вроде теста IQ?[1] В следующих главах будет возможность сделать именно это, чтобы решить, где ваш чат-бот находится на диаграмме.

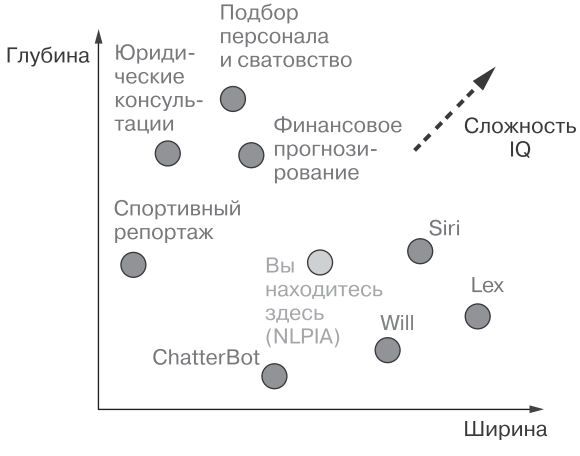


Рис. 1.5. Двумерная диаграмма IQ систем обработки естественного языка

По мере чтения мы будем создавать элементы чат-бота. Для правильной работы нужно, чтобы все функции NLP хорошо работали.

- Выделение признаков (обычно для создания модели векторного пространства).
- Информационный поиск для ответов на фактологические вопросы.
- Семантический поиск для усвоения информации из ранее записанного текста или диалога на естественном языке.
- Генерация естественного языка для составления новых осмысленных высказываний.

Машинное обучение позволяет заставить машины вести себя так, словно мы потратили целую жизнь, программируя их с помощью сотен сложных регулярных выражений или алгоритмов. Можно научить машину реагировать на паттерны,

[1] Хороший вопрос, предложенный Байроном Ризом: «Что больше? Солнце или цент?» (gigaom.com/2017/11/20/voices-in-ai-episode-20-a-conversation-with-marie-des-jardins). Вот вам еще немного вопросов: github.com/totalgood/nlpia/blob/master/src/nlpia/data/iq_test.csv.

аналогичные задаваемым регулярными выражениями, предоставив ей примеры пользовательских высказываний и ответов, которые должен имитировать чат-бот. Модели языка, конечные автоматы, созданные с помощью машинного обучения, намного лучше. Кроме того, они менее чувствительны к ошибкам и опечаткам.

Конвейеры NLP машинного обучения легче программировать. Не нужно предвидеть любой возможный вариант использования символов на нашем языке. Мы просто должны предоставить обучающемуся конвейеру примеры подходящих и неподходящих фраз. Если они на время обучения соответствующим образом помечены, чтобы чат-бот знал, что есть что, он будет учиться различать их. Существуют подходы машинного обучения, для которых требуется мало (или вообще не требуется) маркированных данных.

Мы перечислили несколько интересных поводов для изучения обработки естественного языка. Вы хотите помочь спасти мир, не так ли? Мы же попытались пробудить интерес к некоторым вариантам практического применения NLP, творящим революцию в способах нашего общения, обучения, ведения бизнеса и даже мышления. Пройдет совсем немного времени, и вы сможете построить систему, ведущую себя при «разговоре» словно человек. В следующих главах вы увидите, как обучить чат-бот или конвейер NLP для применения в любой интересующей вас предметной области — от финансов и спорта до психологии и литературы. Если вы можете найти корпус письменной информации о чем-либо, то и машину получится научить понимать эту информацию.

В остальных главах данной книги рассказывается, как использовать машинное обучение так, чтобы не пришлось принимать во внимание все способы выражения мыслей на естественном языке. Каждая глава постепенно усовершенствует базовый конвейер NLP чат-бота, описанный в этой главе.

По мере развития навыков обработки естественного языка вы будете создавать конвейер NLP, способный не только «поддерживать разговор», но и помочь вам достигнуть высот в бизнесе.

Резюме

- Хороший механизм NLP может помочь спасти мир.
- Смысл и коннотацию слов можно расшифровать с помощью машины.
- Интеллектуальный контейнер NLP умеет разрешать неоднозначности.
- Мы можем научить машины здравому смыслу, не тратя на это всю жизнь.
- Чат-боты можно рассматривать как семантические поисковые системы.
- Сфера применения регулярных выражений намного шире обычного поиска.

2 Составление словаря: токенизация слов

В этой главе

- Токенизация текста по словам или *n*-граммам (токенам).
- Обработка нестандартной пунктуации и смайликов, например, в постах в социальных сетях.
- Сжатие словаря токенов с помощью стемминга и лемматизации.
- Построение векторного представления высказываний.
- Создание анализатора тональностей на основе созданных вручную списков токенов.

Итак, вы уже готовы спасать мир с помощью обработки естественного языка? Первое, что вам понадобится, — емкий словарь. В данной главе рассказывается, как разбить документ, любую его строку на отдельные токены смысла. Наши токены ограничиваются словами, знаками пунктуации и числами, но описанные в этой главе методики можно распространить на любые другие смысловые единицы: смайлики ASCII, эмодзи Unicode, математические символы и т. д.

Извлечение токенов из документа требует некоторых манипуляций со строками, выходящих за рамки возможностей метода `str.split()`, применявшегося в главе 1. Необходимо отделить знаки препинания от слов, например кавычки в начале и в конце высказывания. Нужно также разбить сокращения, такие как *we'll*, на составляющие

их слова. После идентификации в документе токенов, подлежащих включению в словарь, вам придется воспользоваться регулярными выражениями, чтобы попытаться объединить в ходе *стемминга* слова со схожим смыслом. Далее мы сформируем векторное представление документов — мультимножество слов, и попытаемся воспользоваться этим вектором для усовершенствования распознавания приветствий по сравнению с приведенным в конце главы 1 вариантом.

Задумайтесь о том, что представляет собой слово или знак. Единое понятие или какое-то размытое облако понятий? Всегда ли вы с уверенностью распознаете слово? Есть ли сходство между словами естественного языка и ключевыми словами языка программирования, имеющими точные определения и отвечающими набору грамматических правил? Можете ли вы написать работающее программное обеспечение для распознавания слов? *Ice cream* для вас — одно слово или два? Разве у слов *ice* и *cream* нет собственных значений в вашем внутреннем словаре, отдельных от составного *ice cream*? Как насчет сокращения *don't*? Должна ли эта строка символов быть разбита на одну или две смысловые единицы?

Слова также можно разделить на более мелкие смысловые единицы. Сами слова делятся на более мелкие значимые части. Слоги, приставки и суффиксы, такие как *re*, *pre* и *ing*, имеют собственные значения. Кроме того, части слов можно разделить на еще более мелкие смысловые единицы. Буквы или графемы (https://ru.wikipedia.org/wiki/Графема) обладают тональностью и смыслом[1].

О символьных моделях векторного пространства мы поговорим в следующих главах. Но сейчас попробуем разрешить вопрос, что такое слово и как разделить текст на слова.

Что можно сказать насчет невидимых (подразумеваемых) слов? Какие дополнительные слова подразумеваются в состоящей из одного слова команде *Don't!*? Если вы можете заставить себя думать как машина, а затем снова переключиться на образ мышления человека, вы поймете, что в этой команде есть три невидимых слова. Одно высказывание *Don't!* может означать *Don't you do that!* или *You, do not do that!*. Данные высказывания включают три скрытых смысловых элемента, пять токенов суммарно, о которых машина должна знать. Однако пока не задумывайтесь о невидимых словах. Все, что вам требуется для этой главы, — токенизатор, умеющий распознавать написанное. Время беспокоиться о подразумеваемых словах, коннотации и даже о собственно смысле наступит в главе 4 и в следующих за ней главах[2].

В этой главе вы встретитесь с простыми алгоритмами разделения строк на слова. Вы научитесь извлекать пары, тройки, четверки и даже пятерки токенов. Такие комбинации называются *n*-граммами. Пары слов называются биграммами,

---

[1] Морфемы — это минимальные осмысленные части слов. Джеффри Хинтон (Geoffrey Hinton) и другие теоретики глубокого обучения продемонстрировали, что даже графемам (буквам) — наименьшим неделимым фрагментам письменного текста — присущ некий неотъемлемый смысл.

[2] Если хотите больше узнать о том, что такое слово, ознакомьтесь с введением в «Морфологию китайского языка» Джерома Паккарда (Jerome Packard). В этой книге Паккард подробно обсуждает понятие «слово». Концепция слова не существовала в китайском языке до XX века. Именно в этом столетии она была переведена из английской грамматики в китайскую.

тройки — триграммами, четверки — 4-граммами и т. д. Использование *n*-грамм позволяет машине знать о таких словах, как *ice cream*, а также о составляющих его *ice* и *cream*. Еще одна биграмма, которую нет смысла разделять, — *Mr. Smith*. В ваших токенах и векторном представлении документа должно быть место как для *Mr. Smith*, так и для *Mr.* и *Smith*.

Пока что все возможные пары (и остальные короткие *n*-граммы) слов включены в наш словарь. В главе 3 вы научитесь оценивать важность слов на основе их частотности (частоты появления в документе). Таким образом, вы сможете отфильтровать пары и тройки слов, которые редко встречаются вместе. Рано или поздно вы обнаружите, что подходы, которые мы описываем в этой книге, неидеальны. Выделение признаков редко позволяет сохранить всю информативную составляющую входных данных в любом конвейере машинного обучения. Это часть искусства NLP — знать, когда необходимо подстроить токенизатор, чтобы извлечь наибольший объем или спектр информации из текста для конкретного приложения.

При обработке естественного языка процесс формирования числового вектора на основе текста — выделение признаков с большим количеством потерь. Тем не менее векторы мультимножеств слов (BOW) сохраняют достаточную долю информационной составляющей текста для создания полезных и интересных моделей машинного обучения. Методы анализа тональностей из конца этой главы — те же самые, которые сервис Gmail использовал, чтобы спасти нас от потока спама, почти сделавшего электронную почту бесполезной.

2.1. Непростые задачи: обзор стемминга

В качестве примера сложности выделения признаков из текста мы рассмотрим *стемминг* — группировку различных форм слова по кластерам. Очень умные люди потратили всю карьеру на разработку алгоритмов группировки слов по их написанию. Можете себе представить, насколько это сложная задача. Представьте также, что вы пытаетесь удалить глагольные окончания, такие как *ing*, из *ending*, чтобы оставить основу *end*, которая будет представлять оба слова. И вы хотели бы сократить слово *running* до *run* так, чтобы эти два слова обрабатывались одинаково. Это сложно, потому что нужно удалить не только *ing*, но и лишнюю *n*. Но необходимо, чтобы слово *sing* оставалось единым целым. Вам бы, наверное, не хотелось удалять окончание *ing* из *sing*, так как останется только буква *s*.

Представьте, что вы пытаетесь найти различие между буквой *s* в конце слова, показывающей множественное число, например, в слове *words*, и обычной *s*, используемой в таких словах, как *bus* и *lens*. Содержат ли отдельные буквы в слове или его частях какую-либо информацию о смысле? Могут ли буквы вводить в заблуждение? Да и да.

В этой главе мы покажем вам, как сделать конвейер NLP немного более интеллектуальным, справляясь с описанными выше трудностями в написании слов с помощью стандартных методов стемминга. В главе 5 мы перечислим статистические подходы к кластеризации, которые требуют от вас лишь накопления определенного набора текстов, написанных на естественном языке и содержащих интересующие

вас слова. Из статистики использования слов в этом наборе текстов можно выяснить семантические основы слов (на самом деле более полезные кластеры слов, такие как леммы или синонимы) без написания вручную регулярных выражений или правил.

2.2. Построение словаря с помощью токенизатора

В NLP *токенизация* — это особый вид сегментирования документов. При сегментации текст разбивается на мелкие куски (сегменты) с более узким информационным содержанием. Сегментация может включать разбиение документа на абзацы, те на предложения, их на фразы и последние — на токены (слова), а также знаки препинания. В этой главе мы сосредоточимся на процессе сегментирования текста в *токены*, называемом токенизацией.

Возможно, вы уже слышали о токенизаторах раньше, на курсе информатики, при обсуждении работы компиляторов. Токенизатор для компиляции языка программирования называют *сканером* (scanner) или *лексическим анализатором* (lexer). Словарь (набор всех допустимых токенов) языка программирования именуют *лексиконом*, и этот термин до сих пор встречается в статьях по NLP. Синтаксический анализатор компилятора языка программирования со встроенным токенизатором — бессканерный синтаксический анализатор. Токены — это оконечная точка контекстно свободных грамматик (context-free grammar, CFG) для анализа языков программирования. Их называют *терминалами*, потому что они служат концом пути от корня до листа в CFG. В главе 11 вы больше узнаете о *формальных* грамматиках, таких как CFG и регулярные выражения, когда начнете использовать их для сопоставления с шаблонами и извлечения информации из текстов на естественном языке.

Вот эквиваленты основных блоков NLP в компиляторах языка программирования:

- *токенизатор* — сканер, лексический анализатор;
- *словарь* — лексикон;
- *синтаксический анализатор* — компилятор;
- *токен*, *терм*, *слово* или *n-грамма* — токен, символ или терминальный символ.

Токенизация — первый шаг в конвейере NLP, поэтому она может серьезно повлиять на остальную часть конвейера. Токенизатор разбивает неструктурированные данные, текст на естественном языке, на фрагменты информации, которые можно считать отдельными элементами. Такие подсчитанные количества вхождений токенов в документ можно непосредственно использовать как представляющий этот документ вектор. Подобный подход позволяет сразу получить из неструктурированной строки (текстового документа) числовую структуру данных, подходящую для машинного обучения. Эти значения могут непосредственно инициировать выполнение компьютером полезных действий и выдачу реакций. Или же их можно применять в конвейере машинного обучения в качестве признаков, инициирующих

более сложные решения или поведение. Наиболее распространенный способ использования векторов мультимножеств слов, созданных таким методом, — поиск документов.

Самый простой способ токенизации предложения — применение внутри строк пробелов в качестве разделителей слов. В языке Python для этого подходит метод `split` из стандартной библиотеки, доступный для всех экземпляров объекта `str`, а также для самого встроенного класса `str` (см. листинг 2.1 и рис. 2.1).

Листинг 2.1. Пример токенизации предложения об усадьбе Монтичелло

```
>>> sentence = """Thomas Jefferson began building Monticello at the
...   age of 26."""
>>> sentence.split()
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26.']
>>> str.split(sentence)
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26.']
```

Thomas | Jefferson | began | building | Monticello | at | the | age | of | 26.

Рис. 2.1. Токенизированная фраза

Как вы можете видеть, этот встроенный метод языка Python неплохо токенизирует простое предложение. Его единственная «ошибка» заключается в последнем слове, где он включил в токен *26.*, завершающий предложение, знак препинания. Обычно токены должны быть отделены от соседних знаков препинания и других значимых токенов в предложении. Токен *26.* — прекрасное представление числа с плавающей запятой 26.0, но при этом он отличается от числа *26*, встречающегося в корпусе в середине предложения, или *26?* в конце вопросительного предложения. Хороший токенизатор должен удалить этот дополнительный символ, чтобы создать *26* в качестве класса эквивалентности для *26*, *26!*, *26?* и *26.*. Еще более точный токенизатор создает отдельный токен для любого завершающего предложение знака

пунктуации, чтобы сегментатор предложения или детектор границ могли найти конец этого предложения.

А пока продолжим работу с нашим несовершенным токенизатором. С пунктуацией и другими сложностями будем разбираться немного позже. По мере дальнейшего знакомства с Python вы можете создать числовое векторное представление для каждого слова. Эти векторы называются *унитарными*, и вскоре вы поймете почему. Последовательность этих унитарных векторов полностью захватывает исходный текст документа в последовательности векторов, таблицы чисел. Это решает первую проблему NLP — преобразование слов в числа:

```
>>> import numpy as np
>>> token_sequence = str.split(sentence)
>>> vocab = sorted(set(token_sequence))
>>> ', '.join(vocab)
'26., Jefferson, Monticello, Thomas, age, at, began, building, of, the'
>>> num_tokens = len(token_sequence)
>>> vocab_size = len(vocab)
>>> onehot_vectors = np.zeros((num_tokens,
...                            vocab_size), int)
>>> for i, word in enumerate(token_sequence):
...     onehot_vectors[i, vocab.index(word)] = 1
>>> ' '.join(vocab)
'26. Jefferson Monticello Thomas age at began building of the'
>>> onehot_vectors
array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

str.split() — наш сделанный на скорую руку токенизатор

В словаре перечислены все уникальные токены, которые необходимо отслеживать

Отсортировано лексикографически таким образом, что цифры идут перед буквами, а прописные буквы — перед строчными

Для каждого слова в предложении помечаем соответствующий столбец в словаре единицей

Ширина пустой таблицы соответствует количеству уникальных словарных термов, а высота — длине документа (в нашем случае десять строк на десять столбцов)

Если вам трудно быстро прочитать все эти единицы и нули, не отчаивайтесь — вы не одиноки. С помощью объектов `DataFrames` библиотеки Pandas можно повысить удобство их восприятия и информативность. Pandas создает для одномерного массива адаптер со вспомогательной функциональностью — объект `Series`. Pandas особенно удобна для таблиц чисел, таких как списки списков, двумерные массивы NumPy, двумерные матрицы NumPy, массивы массивов, словари словарей и т. д.

Объект `DataFrame` отслеживает метки для всех столбцов, позволяя пометить каждый столбец в нашей таблице токеном или словом, которое он представляет. `DataFrame` также может отслеживать метки для каждой строки в `DataFrame.index` с целью обеспечения быстрого поиска. Для большинства приложений они обычно представляют собой просто последовательные целые числа. Пока мы будем использовать для строк в нашей таблице унитарных векторов предложения о Томасе Джефферсоне индекс по умолчанию из целых чисел, показанный в листинге 2.2.

Листинг 2.2. Последовательность унитарных векторов для предложения о Монтичелло

```
>>> import pandas as pd
>>> pd.DataFrame(onehot_vectors, columns=vocab)
   26.  Jefferson  Monticello  Thomas  age  at  began  building  of  the
0    0          0           0       1    0   0      0         0   0    0
1    0          1           0       0    0   0      0         0   0    0
2    0          0           0       0    0   0      1         0   0    0
3    0          0           0       0    0   0      0         1   0    0
4    0          0           1       0    0   0      0         0   0    0
5    0          0           0       0    0   1      0         0   0    0
6    0          0           0       0    0   0      0         0   0    1
7    0          0           0       0    1   0      0         0   0    0
8    0          0           0       0    0   0      0         0   1    0
9    1          0           0       0    0   0      0         0   0    0
```

Унитарные векторы предельно разреженные: в каждом векторе-строке содержится только одно ненулевое значение. Таким образом, мы можем сделать эту таблицу унитарных векторов еще красивее, заменив нули пробелами. Не стоит делать этого с каждым `DataFrame`, который вы намереваетесь использовать в своем конвейере машинного обучения, потому что тогда в массиве NumPy будет создано множество нечисловых объектов, что нарушит всю математику. Однако если задача просто в иллюстрации сходства последовательности унитарных векторов с механическим цилиндром музыкальной шкатулки или барабаном механического пианино, то рекомендуем обратить внимание на листинг 2.3.

Листинг 2.3. Более симпатичные унитарные векторы

```
>>> df = pd.DataFrame(onehot_vectors, columns=vocab)
>>> df[df == 0] = ''
>>> df
  26. Jefferson Monticello Thomas age at began building of the
0                              1
1           1
2                                        1
3                                              1
4                      1
5                                   1
6                                1
7                                                    1
8                                                 1
9   1
```

В этом представлении документа из единственного предложения каждая строка соответствует вектору для отдельного слова. Предложение состоит из десяти слов, все уникальные и не повторяющиеся. Таблица содержит десять столбцов (по числу слов в словаре) и десять строк (по количеству слов в документе). `1` в столбце указывает на слово из словаря, которое присутствовало в этой позиции в документе. Если вы хотите узнать, что такое третье слово в документе, перейдите к третьей строке таблицы. И посмотрите на заголовок столбца со значением `1` в третьей строке (строка

с меткой 2, так как строки нумеруются с 0). Вверху данного столбца, седьмого в таблице, находится представление этого слова на естественном языке, *began*.

Каждая строка таблицы — это бинарный вектор-строка, причем сразу видно, почему он называется унитарным: значения для всех позиций (столбцов) в строке, кроме одной, равны 0 или пусты. Только один столбец (позиция) в векторе содержит 1. Единица (1) означает «включен». Ноль (0) означает «выключен» (отсутствует). Слово *began* в конвейере NLP можно представить с помощью вектора `[0,0,0,0,0,0,1,0,0,0]`.

Одно из приятных свойств векторного представления слов и табличного представления документов в том, что исходная информация не теряется[1]. Если тщательно отслеживать, в каких столбцах отображаются те или иные слова, можно восстановить исходный документ по этой таблице унитарных векторов. Точность процесса восстановления — 100 %, несмотря на то что точность токенизатора при генерации полезных, по вашему мнению, токенов составляла всего 90 %. В результате такие унитарные векторы используются в нейросетях, при моделировании преобразования последовательностей в последовательности (sequence-to-sequence), в порождающих языковых моделях. Унитарные векторы отлично подходят для любых моделей или конвейеров NLP, от которых требуется полностью сохранить смысл исходного текста.

Данная таблица унитарных векторов является своего рода записью исходного текста. Матрицу нулей и единиц, указанную выше, можно, если постараться, представить в виде бумажной ленты механического пианино (ru.wikipedia.org/wiki/Механическое_пианино) или выпуклости на барабане музыкальной шкатулки (ru.wikipedia.org/wiki/Музыкальная_шкатулка). Словарный ключ сверху таблицы подсказывает машине, какую ноту или слово играть для каждой строки в последовательности слов или музыки, исполняемой механическим пианино. В отличие от механического пианино наше устройство записи и воспроизведения слов может одновременно пользоваться только одним «пальцем». Он — унитарный. Каждая нота или слово проигрывается в течение одинаковых промежутков времени с постоянным темпом. Интервалы между словами в тексте одинаковы.

Стоит отметить, что это всего лишь один из способов осмысления унитарных векторов. Вы можете придумать любую другую ментальную модель, которая кажется вам осмысленной. Важно лишь то, что мы преобразовали предложение из слов естественного языка в последовательность чисел (векторов). Теперь компьютер может читать и производить арифметические действия над этими векторами так же, как

[1] За исключением различий между разными видами пробелов, «разбиваемых» токенизатором. Восстановить исходный текст, если лексический анализатор не сохраняет информацию об отбрасываемых в процессе создания токенов пробельных символах, невозможно. В этом случае нет никакой возможности узнать, что должно стоять между конкретными словами: пробел, переход на новую строку, знак табуляции или вообще ничего. Стоит отметить, что в большинстве документов на английском языке смысловая составляющая пробельных символов между словами достаточно низкая. Почти все современные синтаксические и лексические анализаторы сохраняют эту информацию о пробелах на случай, если она понадобится в будущем.

и над любыми другими векторами или списками чисел, а значит, эти унитарные векторы могут быть входными данными для любого конвейера NLP, которому нужны подобные векторы.

Вы можете также воспроизвести последовательность унитарных векторов для генерации текста для чат-бота, как это делает механическое пианино для обычных слушателей. Единственное, что нужно, — выяснить, как создать механическое пианино, которое могло бы «понимать» и комбинировать векторы слов «по-новому». В итоге нам хотелось бы услышать от чат-бота или конвейера NLP что-нибудь новое для нас. Мы доберемся до этого в главах 9 и 10, когда будем обсуждать модели LSTM (long short-term memory — «долгая краткосрочная память»), а также соответствующие нейросети.

Подобное представление предложения в виде унитарных векторов сохраняет все детали, грамматику и порядок слов исходного текста. Мы успешно преобразовали слова в понятные для компьютера числа. Последние относятся к особенно любимой всеми компьютерами разновидности — двоичным числам. Хочется заметить, что таблица для одного предложения довольно большая. Если задуматься, окажется, что мы увеличили размер файла, в котором хранится документ. Для больших документов это непрактично. Размер документа (длина векторной таблицы) в таком случае будет колоссальным. В английском языке по крайней мере 20 тысяч распространенных слов. Или даже миллионы, если включить в этот список также имена собственные, а представление документа в виде унитарных векторов требует создания новой таблицы (матрицы) для каждого документа. Его можно сравнить с необработанной «фотографией» документа. Если вы когда-либо занимались обработкой фотографий, то знаете: чтобы извлечь полезную информацию из данных, необходимо понизить размерность.

Мне хотелось бы сделать небольшой экскурс в математику, чтобы показать, насколько большими и громоздкими могут быть эти «бумажные ленты механического пианино». В большинстве случаев используемый в конвейере NLP словарь токенов будет содержать куда больше 10 или 20 тысяч токенов. Иногда это могут быть сотни тысяч и даже миллионы токенов. Представим, что в нашем словаре их ровно миллион. Пусть у вас также будет всего-навсего 3000 книжек, каждая из которых содержит 3500 предложений по 15 слов в каждом (вполне реалистичная средняя оценка для коротких книг). Получается огромное количество гигантских таблиц (матриц):

```
>>> num_rows = 3000 * 3500 * 15
>>> num_rows
157500000
>>> num_bytes = num_rows * 1000000
>>> num_bytes
157500000000000
>>> num_bytes / 1e9
157500  # gigabytes
>>> _ / 1000
157.5  # terabytes
```

Количество строк в таблице

Количество байтов в случае, если каждая из ячеек занимает только 1 байт

В интерактивной консоли Python переменной с именем _ автоматически присваивается значение предыдущего вывода. Это довольно удобно на случай, если вы забудете явно присвоить поименованной переменной результат функции или выражения, как делали это с num_bytes и num_rows

Речь идет более чем о миллионе миллионов бит, даже если для каждой ячейки используется всего 1 бит. В таком случае понадобится почти 20 Тбайт памяти для хранения небольшой книжной полки, представленной в виде унитарных векторов. К счастью, подобные структуры данных никогда не используются для хранения документов. Они применяются временно, в оперативной памяти, при обработке документов по одному слову за раз.

Из предыдущего абзаца можно сделать вывод, что хранение такого количества нулей и запоминание порядка слов во всех документах бессмысленно. Это непрактично. Все, что нужно на самом деле, — вычленить из документа его суть, то есть сжать документ до одного вектора вместо огромной таблицы. Кроме того, мы готовы отказаться от возможности обратного восстановления документа. Мы хотим захватить бóльшую часть его смысла (информации), а не весь.

А что, если разбить документ на более короткие смысловые фрагменты, например предложения? Допустим, бóльшая часть смысла предложения может быть получена из самих слов. Проигнорируем порядок слов и их грамматику и перемешаем их вместе в мультимножестве слов — по одному мультимножеству на каждое предложение в документе. Оказывается, что это весьма разумное допущение. Даже в случае документов из нескольких страниц вектор мультимножества слов позволяет вкратце подытожить суть документа. Как видите, после лексикографической сортировки всех слов предложения про Джефферсона человек все равно сможет догадаться о его смысле. Машина тоже справится с этим. Можно использовать этот новый подход с векторами мультимножеств слов для сжатия информационного содержания всех документов в более удобную для работы структуру данных.

Если просуммировать все эти унитарные векторы вместо «проигрывания» их по одному, получится вектор мультимножества слов. Его также называют вектором частотности слов, так как он отражает *частоту* вхождения слов, а не порядок их расположения. Вы можете использовать этот вектор для представления всего документа или предложения в виде одного вектора не очень большой длины. Его длина будет равна длине словаря (числу отслеживаемых уникальных токенов).

Кроме того, при простом поиске по ключевым словам можно получить из унитарных векторов бинарный вектор мультимножества с помощью операции логического ИЛИ. В таком случае вы можете игнорировать множество слов, не подходящих в качестве ключевых слов или поисковых термов. Подобный метод подходит для индексов поисковых систем или первого фильтра информационно-поисковой системы. Для последующего поиска документов в поисковом индексе должна присутствовать лишь информация о наличии или отсутствии слова в документе.

Если положить руки на пианино и нажать на все клавиши одновременно, никаких приятных и осмысленных звуков вы не извлечете. Однако при работе со словами подобный подход жизненно важен для «осмысления» машиной всей группы слов как единого целого. Если ограничить количество токенов 10 000 наиболее важных слов, можно сжать цифровое представление нашей воображаемой книги из 3500 предложений до 10 Кбайт, что в пересчете на наш корпус из 3000 книг займет 30 Мбайт. Последовательности унитарных векторов заняли бы сотни гигабайт.

К счастью, в любом конкретном тексте слова из словаря употребляются достаточно редко. Документы в большинстве приложений, использующих методику

мультимножеств слов, достаточно короткие (иногда хватает одного предложения). Поэтому наш вектор мультимножества соответствует широкому и красивому аккорду, осмысленному сочетанию подходящих друг другу нот (слов), а не одновременному нажатию всех клавиш пианино. Наш чат-бот сможет «сыграть» эти аккорды, даже если в предложении присутствует часть «диссонансных» слов, которые обычно не используются вместе. Наш конвейер сможет извлечь полезную информацию о предложении даже из диссонанса (необычного использования слов).

Разберемся с тем, как можно поместить токены в двоичный вектор, указывающий на наличие или отсутствие конкретного слова в определенном предложении. Векторное представление набора предложений можно проиндексировать для отражения информации о том, какие слова в каком предложении используются. Этот индекс — эквивалент указателя в конце книги, за исключением того, что он указывает не на страницу, на которой встретилось слово, а на содержащее это слово предложение (или соответствующий вектор). В то время как в указателе обычно отмечены только основные слова, связанные с темой книги, мы отслеживаем все слова (по крайней мере сейчас).

Именно так выглядит наш отдельный документ — предложение о Томасе Джефферсоне — в виде вектора мультимножества слов:

```
>>> sentence_bow = {}
>>> for token in sentence.split():
...     sentence_bow[token] = 1
>>> sorted(sentence_bow.items())
[('26.', 1)
 ('Jefferson', 1),
 ('Monticello', 1),
 ('Thomas', 1),
 ('age', 1),
 ('at', 1),
 ('began', 1),
 ('building', 1),
 ('of', 1),
 ('the', 1)]
```

При сортировке с помощью функции `sorted()` в языке Python десятичные числа предшествуют символам, а прописные буквы — строчным. Таков порядок символов в кодировках ASCII и Unicode. В таблице ASCII прописные буквы располагаются перед строчными. Порядок слов в вашем словаре неважен. Пока вы последовательно создаете токены, конвейер машинного обучения будет одинаково работать с любым порядком слов.

Вы можете заметить, что применение класса `dict` (или другого способа попарного сопоставления слов значениям `0/1`) для хранения двоичных векторов не должно расходовать много места впустую. При использовании словаря в качестве представления векторов достаточно хранить `1`, когда одно из тысяч или даже миллионов слов словаря встречается в конкретном документе. Нет нужды объяснять, как неудобно представлять мультимножество слов в виде непрерывного списка нулей и единиц с привязкой к месту в «плотном векторе» для каждого слова в словаре, скажем, из 100 000 слов. Подобного рода представление вашего предложения про Томаса

Джефферсона займет 100 Кбайт. Так как словарь игнорирует отсутствующие слова (отмеченные нулем), словарное представление занимает всего несколько байтов для каждого слова в предложении из десяти слов. Такой словарь можно сделать еще более эффективным, если подставить каждое слово в виде целочисленного указателя на место каждого из слов в лексиконе — списке слов, составляющем словарь для конкретного приложения.

Попробуем еще более эффективную форму словаря — класс `Series` библиотеки Pandas. С помощью объекта `DataFrame` библиотеки Pandas можно еще добавить предложения в корпус двоичных векторов ваших текстов о Томасе Джефферсоне. Все эти загадочные пропуски нулей в векторах и разницы разреженных и плотных мультимножеств слов станут более понятными по мере добавления новых предложений и соответствующих векторов мультимножеств слов в `DataFrame` (таблицу векторов, соответствующих текстам в корпусе):

```
>>> import pandas as pd
>>> df = pd.DataFrame(pd.Series(dict([(token, 1) for token in
...   sentence.split()])), columns=['sent']).T
>>> df
      26.  Jefferson  Monticello  Thomas  age  at  began  building  of  the
Sent    1          1           1       1    1   1      1         1   1    1
```

Добавим еще немного текстов в наш корпус, чтобы увидеть нюансы организации `DataFrame`. Он индексирует как столбцы (документы), так и строки (слова), так что может служить и обратным индексом для поиска документов, если нужно быстро найти ответ на вопрос викторины (листинг 2.4).

Листинг 2.4. Создание DataFrame из векторов мультимножеств слов

```
>>> sentences = """Thomas Jefferson began building Monticello at the\
...   age of 26.\n"""
>>> sentences += """Construction was done mostly by local masons and\
...   carpenters.\n"""
>>> sentences += "He moved into the South Pavilion in 1770.\n"
>>> sentences += """Turning Monticello into a neoclassical masterpiece\
...   was Jefferson's obsession."""
>>> corpus = {}
>>> for i, sent in enumerate(sentences.split('\n')):
...     corpus['sent{}'.format(i)] = dict((tok, 1) for tok in
...         sent.split())
>>> df = pd.DataFrame.from_records(corpus).fillna(0).astype(int).T
>>> df[df.columns[:10]]
        1770.  26.  Construction   ...    Pavilion  South  Thomas
sent0       0    1             0  ...           0      0       1
sent1       0    0             1  ...           0      0       0
sent2       1    0             0  ...           1      1       0
sent3       0    0             0  ...           0      0       0
```

Это исходное предложение, описанное в листинге 2.1

Обычно следует использовать .splitlines(), но здесь мы явно добавляем одиночный символ '\n' в конец каждой строки/предложения, так что нужно разбивать и по нему

Выводим только первые десять токенов (столбцов DataFrame), чтобы избежать переноса на другую строку

При беглом осмотре можно увидеть небольшое пересечение списков используемых в этих предложениях слов. Среди первых семи слов словаря только *Monticello* встречается более чем в одном предложении. Теперь для сравнения или поиска схожих документов необходимо учесть это пересечение в конвейере. Один из способов найти сходства между предложениями — подсчитать количество пересекающихся токенов с помощью *скалярного произведения*.

2.2.1. Скалярное произведение

В NLP часто применяется скалярное произведение, поэтому нужно четко понимать, что это такое. Можете пропустить подраздел, если вы с легкостью вычисляете такое произведение в уме. Скалярное произведение также называется *внутренним произведением*, так как размерности пары векторов (количество элементов в каждом из них) или матриц (число строк первой матрицы и столбцов второй) должны совпадать. Эта операция аналогична команде `inner join` для двух таблиц реляционной базы данных.

Из названия *«скалярное произведение»* также следует, что его результатом является одно скалярное значение. Это отличает его от векторного произведения (внешнего произведения, cross product), результат которого представляет собой вектор. Эти названия отражает форма соответствующих формальных математических обозначений, где скалярное произведение обозначается точкой (`.`), а *векторное* произведение — крестиком (`x`). Результат скалярного произведения равен сумме обычных произведений всех элементов одного вектора и всех элементов второго вектора.

Вот фрагмент кода на языке Python, который вы можете запустить, чтобы разобраться, что такое скалярное произведение (листинг 2.5).

Листинг 2.5. Пример вычисления скалярного произведения

```
>>> v1 = pd.np.array([1, 2, 3])
>>> v2 = pd.np.array([2, 3, 4])
>>> v1.dot(v2)
20
>>> (v1 * v2).sum()
20
>>> sum([x1 * x2 for x1, x2 in zip(v1, v2)])
20
```

Большое количество подобных итераций может сильно замедлить работу конвейера

Перемножение массивов NumPy — очень быстродействующая векторизованная операция

СОВЕТ

Скалярное произведение эквивалентно *матричному*, которое в NumPy можно вычислить с помощью функции np.matmul() или оператора @. Поскольку все векторы можно преобразовать в матрицу вида $N \times 1$ или $1 \times N$, этот краткий оператор можно применить к двум векторам-столбцам ($N \times 1$) путем транспонирования первого с помощью кода v1.reshape(–1, 1).T @ v2.reshape(–1, 1), который выводит результат скалярного произведения в виде матрицы размером 1×1: array([[20]]).

2.2.2. Измерение пересечений мультимножеств слов

Если мы измерим пересечение мультимножеств слов для двух векторов, то сможем узнать, как сильно похожи применяемые в них слова. Эта оценка хорошо отражает сходство их смыслов. Воспользуемся скалярным произведением, чтобы оценить пересечения векторов мультимножеств слов между какими-нибудь новыми предложениями и предложением про Томаса Джефферсона (`sent0`) (листинг 2.6).

Листинг 2.6. Подсчет пересечений количеств слов для двух векторов мультимножеств слов

```
>>> df = df.T
>>> df.sent0.dot(df.sent1)
0
>>> df.sent0.dot(df.sent2)
1
>>> df.sent0.dot(df.sent3)
1
```

Из этих результатов понятно, что одно из слов использовалось как в `sent0`, так и в `sent2`. Также одно из слов словаря применялось в обоих предложениях `sent0` и `sent3`. Подобное пересечение слов является мерой их сходства. Что интересно, это странное предложение, `sent1`, единственное, в котором прямо не упоминались ни Джефферсон, ни Монтичелло, а был абсолютно другой набор слов для передачи информации об иных неизвестных людях.

Ниже представлен один из способов нахождения общего слова для предложений `sent0` и `sent3`. Именно это слово дало последнее скалярное произведение, равное единице:

```
>>> [(k, v) for (k, v) in (df.sent0 & df.sent3).items() if v]
[('Monticello', 1)]
```

Это ваша первая модель векторного пространства (VSM) документов (предложений) на естественном языке. Для этих векторов мультимножеств определено не только скалярное произведение, но и другие операции: сложение, вычитание, OR, AND и т. д. Вы можете даже вычислить евклидово расстояние или найти угол между двумя векторами. Подобное представление документа в качестве двоичного вектора дает много возможностей. Долгое время на нем основывались поиск и извлечение документов. Все современные ЦПУ имеют встроенные инструкции адресации памяти для эффективного хеширования, индексации и поиска больших наборов двоичных векторов, подобных описанным выше. Хотя эти инструкции были разработаны для иных целей (индексации мест в оперативной памяти для извлечения данных оттуда), они в равной степени эффективны для выполнения операций над бинарными векторами по поиску и извлечению текста.

2.2.3. Улучшение токенов

В некоторых ситуациях для разделения слов используются и другие символы, помимо пробелов. Кроме того, нам все еще мешает точка на конце токена *26.*. Наш токенизатор должен разбивать предложение не только по пробелам, но и по знакам препинания, таким как запятые, точки, кавычки, точки с запятой и даже дефисы (тире). В некоторых случаях требуется, чтобы токенизатор относился к подобным знакам как к словам, то есть отдельным токенам. В других случаях их просто нужно игнорировать.

В предыдущем примере последний токен предложения был испорчен точкой на конце *26.*. Точка в конце может сбить с толку следующие этапы конвейера NLP, такие как стемминг, где схожие слова должны группироваться вместе согласно правилам, зависящим от единообразного написания слов. В листинге 2.7 демонстрируется один из способов.

Листинг 2.7. Создание токена

```
>>> import re
>>> sentence = """Thomas Jefferson began building Monticello at the\
...   age of 26."""
>>> tokens = re.split(r'[-\s.,;!?]+', sentence)
>>> tokens
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26',
 '']
```

Разбиваем предложение по пробелам или знакам препинания, встречающимся как минимум один раз (обратите внимание на + после закрывающей квадратной скобки в регулярном выражении). См. врезку на следующей странице

Мы обещали, что будем использовать больше регулярных выражений. Надеемся, они стали для вас понятнее, чем при первом применении. Если нет, врезка на с. 84 описывает каждый символ вышеприведенного регулярного выражения. Если же вы хотите копнуть еще глубже, загляните в приложение Б.

Как работают регулярные выражения

Вот как работает регулярное выражение в листинге 2.7. Квадратные скобки (`[` и `]`) используются для обозначения *класса символов* — множества используемых символов. Знак плюс после закрывающей квадратной скобки (`]`) означает, что соответствующий шаблону текст должен содержать один или несколько симво-

лов в квадратных скобках. `\s` в классе символов — краткое обозначение заранее определенного класса символов, включающего в себя все пробельные символы, подобные тем, которые создаются при нажатии клавиш Пробел, Tab и Return. Класс символов `r'[\s]'` эквивалентен `r'\t\n\r\x0b\x0c'`. Всего пробельных символов шесть: пробел (`' '`), табуляция (`'\t'`), новый абзац (`'\r'`), новая строка (`'\n'`) и прогон страницы (`'\f'`).

Здесь не использовались диапазоны символов, однако это может пригодиться вам позже. Диапазон символов — особый вид класса символов, указываемый в квадратных скобках с дефисом, например `r'[a-z]'`, соответствующий всем строчным буквам. Диапазон символов `r'[0-9]'` соответствует любой цифре от 0 до 9 и эквивалентен `r'[0123456789]'`. Регулярное выражение `r'[_a-zA-Z]'` будет соответствовать любому символу подчеркивания (`_`) или букве английского алфавита (в верхнем или нижнем регистре).

Дефис (`-`) сразу после открывающей квадратной скобки является своеобразной причудой регулярных выражений. Поставить дефис в произвольном месте в квадратных скобках нельзя, потому что синтаксический анализатор регулярных выражений подумает, что имеется в виду диапазон символов, например `r'[0-9]'`. Чтобы он понял, что речь идет о символе «дефис», необходимо поместить его сразу после открывающей квадратной скобки класса символов. Так что, если нужно указать собственно символ дефиса (тире) в классе символов, он должен быть первым или экранироваться с помощью обратной косой черты `\`.

Функция `re.split` просматривает все символы входной строки (второй аргумент, `sentence`) слева направо, ища любые совпадения на основе заложенной в регулярное выражение «программы» (первый аргумент, `r'[-\s.,;!?]+'`). Если она находит совпадение, то разрывает строку прямо перед этим совпадающим символом и сразу после него, пропуская совпадающий символ или символы. Таким образом, функция `re.split` работает аналогично `str.split`, зато для любого разделителя из одного или нескольких символов, соответствующего регулярному выражению.

Круглые скобки (`(` и `)`) используются для группировки регулярных выражений аналогично группировке математических выражений, а также выражений языка Python и большинства других языков программирования. Эти скобки обуславливают проверку совпадения всего регулярного выражения в скобках до перехода к следующим за ним символам.

Улучшенные регулярные выражения для разделения слов

Скомпилируем наше регулярное выражение для ускорения работы токенизатора. Скомпилированные объекты регулярных выражений удобны в использовании не только из-за скорости.

Когда имеет смысл компилировать шаблоны регулярных выражений

Модуль регулярных выражений в Python позволяет предварительно скомпилировать регулярные выражения[1], а затем переиспользовать их в базе кода. Представьте, например, регулярное выражение, которое извлекает номера телефонов. С помощью функции `re.compile()` его можно предварительно скомпилировать и передать в качестве аргумента функции или классу, выполняющему токенизацию. Это редко дает преимущество в скорости, потому что Python кэширует скомпилированные объекты для последних `MAXCACHE=100` регулярных выражений. Однако если вы используете более 100 различных регулярных выражений или хотите вызывать методы регулярных выражений, а не методы, соответствующие функции модуля `re`, функция `re.compile` может быть полезна:

```
>>> pattern = re.compile(r"([-\s.,;!?])+")
>>> tokens = pattern.split(sentence)
>>> tokens[-10:]  # только последние 10 токенов
['the', ' ', 'age', ' ', 'of', ' ', '26', '.', '']
```

Это простое регулярное выражение помогает убрать точку в конце токена *26*.. Не спешите расслабляться, у вас возникла новая проблема. Необходимо отфильтровать пробелы и знаки препинания, которые не должны включаться в словарь. Смотрите следующий код и рис. 2.2:

```
>>> sentence = """Thomas Jefferson began building Monticello at the\
...   age of 26."""
>>> tokens = pattern.split(sentence)
>>> [x for x in tokens if x and x not in '- \t\n.,;!?']
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26']
```

Если вы хотите попрактиковаться с лямбда-выражениями и функцией filter(), воспользуйтесь следующим кодом:
list(filter(lambda x: x if x and x not in '- \t\n.,;!?' else None, tokens))

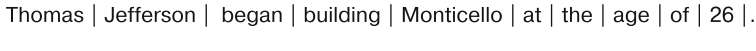


Рис. 2.2. Токенизированная фраза

Таким образом, встроенный Python-пакет `re` прекрасно подходит для приведенного выше примера предложения, но только в случае, если тщательно отфильтровать нежелательные токены. Нет причин искать другие пакеты для работы с регулярными выражениями, если не считать `regex`...

[1] Подробности можно найти на форуме stack overflow или в документации Python по адресу stackoverflow.com/a/452143/623735.

Как использовать новый модуль regex в Python

Существует новый пакет регулярных выражений `regex`, который в конечном счете заменит пакет `re`. Он обладает полной обратной совместимостью и может быть установлен из каталога `pypi` с помощью системы управления пакетами `pip`. Его удобные новые функции включают:

- поддержку пересекающихся множеств соответствия;
- многопоточность;
- полноценную поддержку Unicode;
- нечеткие совпадения регулярных выражений (аналогично `agrep` TRE в UNIX-системах);
- увеличенный по умолчанию `MAXCACHE` (500 регулярных выражений).

Несмотря на то что `regex` предназначен для замены пакета `re` и обладает полной обратной совместимостью с `re`, пока его придется устанавливать как дополнительный пакет с помощью системы управления пакетами, например `pip`:

```
$ pip install regex
```

Вы можете найти больше информации о модуле регулярных выражений на сайте PyPI по адресу pypi.python.org/pypi/regex.

Токенизаторы могут легко становиться очень сложными. Например, может потребоваться разбить текст по точкам, но только если за точкой не следует цифра, чтобы не разбивать десятичные числа. Другой пример, в котором не нужно разбивать текст по точке, — тогда точка является частью смайлика, как в сообщении Twitter.

Токенизаторы реализованы в нескольких библиотеках Python, каждый со своими преимуществами и недостатками.

- *spaCy* — точная, гибкая, быстрая, написана на Python.
- *Stanford CoreNLP* — более точная, менее гибкая, быстрая, требует для работы Java 8.
- *NLTK* — популярная, хорошо подходит для студентов, написана на Python.

NLTK и Stanford CoreNLP — самые давние и широко применяемые для сравнения алгоритмов NLP в научных статьях библиотеки. Несмотря на то что у библиотеки Stanford CoreNLP есть API Python, она основана на прикладной части Java 8 CoreNLP, которую необходимо устанавливать и настраивать отдельно. Поэтому, чтобы быстро приступить к работе, проще использовать токенизатор Natural Language Toolkit (NLTK). Это позволит воспроизвести результаты, которые можно встретить в научных статьях и блогах.

Для моделирования нашего простого примера токенизатора можно воспользоваться функцией `RegexpTokenizer` из библиотеки NLTK:

```
>>> from nltk.tokenize import RegexpTokenizer
>>> tokenizer = RegexpTokenizer(r'\w+|$[0-9.]+|\S+')
>>> tokenizer.tokenize(sentence)
['Thomas',
```

```
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26',
 '.']
```

Этот токенизатор немного лучше, чем тот, который вы использовали изначально, поскольку игнорирует пробельные символы. Он также отделяет завершающие предложение знаки пунктуации от токенов, не содержащих других знаков препинания.

Одним из лучших токенизаторов является Word Tokenizer Treebank, входящий в состав библиотеки NLTK. В нем множество общих правил для токенизации английских слов. К примеру, он отделяет завершающие фразу знаки препинания (`?!.;,`) от соседних токенов и сохраняет десятичные числа, содержащие точку, в качестве отдельного токена. Кроме того, он включает правила токенизации для английских сокращений. Например, *don't* токенизируется как `["do", "n't"]`. Этот метод токенизации поможет на последующих шагах конвейера NLP, таких как стемминг. Вы можете найти все правила токенизатора Treebank по адресу www.nltk.org/api/nltk.tokenize.html#module-nltk.tokenize.treebank. Обратите внимание на следующий код и рис. 2.3:

```
>>> from nltk.tokenize import TreebankWordTokenizer
>>> sentence = """Monticello wasn't designated as UNESCO World Heritage\
...   Site until 1987."""
>>> tokenizer = TreebankWordTokenizer()
>>> tokenizer.tokenize(sentence)
['Monticello',
 'was',
 "n't",
 'designated',
 'as',
 'UNESCO',
 'World',
 'Heritage',
 'Site',
 'until',
 '1987',
 '.']
```

Monticello | was | n't | designated | as | UNESCO | World | Heritage | Site | until | 1987 | .

Рис. 2.3. Токенизированная фраза

Сокращения

Наверняка вы недоумеваете, зачем было разбивать сокращение *wasn't* на токены *was* и *n't*. Для некоторых приложений, например основанных на грамматике моде-

лей NLP, использующих синтаксические деревья, важно разделять слова *was* и *not*, чтобы входные данные синтаксического анализатора представляли собой согласованный предсказуемый набор токенов с известными грамматическими правилами. Существует множество стандартных и нестандартных способов сокращения слов. При сведении сокращений до составляющих их слов достаточно запрограммировать анализатор дерева зависимостей или синтаксический анализатор так, чтобы учесть лишь различные варианты написания отдельных слов, а не все возможные сокращения.

Токенизация неформального текста из социальных сетей, таких как Twitter и Facebook

Библиотека NLTK включает в себя токенизатор `casual_tokenize` для работы с короткими, неформальными, сдобренными смайликами текстами из социальных сетей, где грамматика и правописание сильно варьируются.

Функцией `casual_tokenize` можно выделять имена пользователей и сокращать количество повторяющихся символов в токене:

```
>>> from nltk.tokenize.casual import casual_tokenize
>>> message = """RT @TJMonticello Best day everrrrrrr at Monticello.\
...   Awesommmmmmeeeeeeee day :*)"""
>>> casual_tokenize(message)
['RT', '@TJMonticello',
 'Best', 'day','everrrrrrr', 'at', 'Monticello', '.',
 'Awesommmmmmeeeeeeee', 'day', ':*)']
>>> casual_tokenize(message, reduce_len=True, strip_handles=True)
['RT',
 'Best', 'day', 'everrr', 'at', 'Monticello', '.',
 'Awesommmeee', 'day', ':*)']
```

2.2.4. Расширяем словарь n-граммами

Вернемся к проблеме слов *ice cream*, поднятой в начале главы. Помните, мы говорили о том, чтобы попытаться сохранить *ice* и *cream* вместе: *I scream, you scream, we all scream for ice cream.*

Хотим отметить, что знаем не слишком много людей, готовых кричать от радости при виде сливок *(cream)*. И уж точно никто не будет кричать из-за льда *(ice)*, если они, конечно, не поскользнулись на нем. Таким образом, вам нужно, чтобы в наших векторах слова *ice* и *cream* были вместе.

Все мы граммы для n-грамм

N-грамма — это последовательность, содержащая до *n* элементов, которые были извлечены из последовательности этих элементов, обычно строки. В общем случае

элементами *n*-граммы могут быть буквы, слоги, слова или даже символы, такие как *A*, *T*, *G* и *C*, используемые для представления последовательности ДНК[1].

В этой книге нас интересуют только *n*-граммы слов, а не символов[2]. Так что, когда мы говорим «биграмма», имеем в виду пару слов, таких как *ice cream*. Когда мы говорим «триграмма», то предполагаем тройку слов вроде *beyond the pale* или *Johann Sebastian Bach* или *riddle me this*. *N*-граммы не обязательно должны означать что-то вместе, например представлять собой сложные слова. Они просто должны встречаться достаточно часто, чтобы привлечь внимание счетчиков токенов.

Зачем тратить время на *n*-граммы? Как вы видели ранее, при преобразовании в вектор мультимножества слов последовательность токенов теряет значительную часть смысла, заключенного в порядке этих слов. Если расширить концепцию токена на токены из нескольких слов, *n*-грамм, конвейер NLP может сохранять значительную долю смысла, заключенного в порядке слов в этих высказываниях. Например, слово *not*, меняющее смысл на обратный, останется рядом с соседними словами, где и должно быть. Без *n*-граммовой токенизации такое слово болталось бы по разным позициям, а его смысл связывался бы со всем предложением или документом, а не с соседними словами. Биграмма *was not* сохраняет гораздо больше смысла отдельных слов *was* и *not*, чем соответствующие одограммы в векторе мультимножества. Если связать слово с его соседями в конвейере, можно сохранить немного его контекста.

В следующей главе мы покажем, как распознать, какие из этих *n*-грамм содержат больше полезной информации относительно других, какие из них можно использовать для уменьшения количества отслеживаемых конвейером NLP токенов (*n*-грамм). Иначе ему пришлось бы хранить и поддерживать список всех последовательностей слов, с которыми он сталкивался. Подобный метод расстановки приоритетов поможет ему распознать токены *Thomas Jefferson* и *ice cream*, не обращая особого внимания на *Thomas Smith* или *ice shattered*. В главе 4 мы будем связывать пары слов и даже более длинные последовательности с их реальным смыслом независимо от значения отдельных составляющих их слов. Но пока токенизатор нужен для создания этих последовательностей, *n*-грамм.

Воспользуемся уже известным нам предложением о Томасе Джефферсоне для демонстрации ожидаемых результатов работы токенизатора биграмм, чтобы вы понимали, что мы пытаемся создать:

```
>>> tokenize_2grams("Thomas Jefferson began building Monticello at the\
...   age of 26.")
['Thomas Jefferson',
 'Jefferson began',
 'began building',
```

---

[1] Лингвистические методы, а также методы NLP часто используются для выуживания информации из ДНК и РНК. По адресу ru.wikipedia.org/wiki/Нуклеотидная_последовательность можно найти список символов нуклеиновых кислот, необходимых для перевода языка нуклеиновых кислот на понятный человеку язык.

[2] Возможно, вы знаете про индексы триграмм из курса баз данных или документации PostgreSQL (postgres). Но они представляли собой тройки символов, удобные для быстрого поиска нечетких соответствий строк в огромной базе данных строк с помощью символов %, ~, и * в полнотекстовых поисковых запросах SQL.

```
 'building Monticello',
 'Monticello at',
 'at the',
 'the age',
 'age of',
 'of 26']
```

Я уверен, что вы заметили: в этой последовательности биграмм сохранено немного больше информации, чем при токенизации предложения на слова. Следующие этапы конвейера NLP будут иметь доступ только к тем токенам, которые генерирует токенизатор. Поэтому необходимо передать следующим этапам информацию, что токен *Thomas* не относится к *Isaiah Thomas* или к мультфильму *Thomas & Friends*. *N*-граммы — один из инструментов хранения информации о контексте по мере прохождения данных через конвейер.

Ниже представлен наш первоначальный токенизатор 1-грамм:

```
>>> sentence = """Thomas Jefferson began building Monticello at the\
...   age of 26."""
>>> pattern = re.compile(r"([-\s.,;!?])+")
>>> tokens = pattern.split(sentence)
>>> tokens = [x for x in tokens if x and x not in '- \t\n.,;!?']
>>> tokens
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26']
```

А вот токенизатор *n*-грамм из модуля `nltk`:

```
>>> from nltk.util import ngrams
>>> list(ngrams(tokens, 2))
[('Thomas', 'Jefferson'),
 ('Jefferson', 'began'),
 ('began', 'building'),
 ('building', 'Monticello'),
 ('Monticello', 'at'),
 ('at', 'the'),
 ('the', 'age'),
 ('age', 'of'),
 ('of', '26')]
>>> list(ngrams(tokens, 3))
[('Thomas', 'Jefferson', 'began'),
 ('Jefferson', 'began', 'building'),
 ('began', 'building', 'Monticello'),
 ('building', 'Monticello', 'at'),
 ('Monticello', 'at', 'the'),
 ('at', 'the', 'age'),
 ('the', 'age', 'of'),
 ('age', 'of', '26')]
```

СОВЕТ

Чтобы повысить эффективность использования памяти, функция ngrams библиотеки NLTK возвращает генератор Python. Генераторы Python — это умные функции, которые ведут себя как итераторы, выдавая только один элемент за раз вместо того, чтобы возвращать всю последовательность сразу. Эта особенность полезна в циклах for, где генератор загружает по одному элементу вместо загрузки списка элементов в память целиком. Если же вы хотите посмотреть на все возвращенные *n*-граммы одновременно, преобразуйте генератор в список тем же образом, как мы делали в предыдущем примере. Имейте в виду, что это следует делать только в интерактивном сеансе, а не во время длительной задачи токенизации больших текстов.

В предыдущем списке *n*-граммы представляли собой кортежи, однако их легко можно объединить, если нужно, чтобы все токены в конвейере были строками. Благодаря этому более поздние этапы конвейера могут ожидать получить в качестве входных строковых последовательностей более единообразные типы данных:

```
>>> two_grams = list(ngrams(tokens, 2))
>>> [" ".join(x) for x in two_grams]
['Thomas Jefferson',
 'Jefferson began',
 'began building',
 'building Monticello',
 'Monticello at',
 'at the',
 'the age',
 'age of',
 'of 26']
```

Наверняка вы чувствуете здесь какое-то несоответствие. Возможно, глядя на предыдущий пример, вы поймете, что токен *Thomas Jefferson* будет встречаться во многих документах. Однако биграммы *of 26* или даже *Jefferson began*, вероятно, будут редки, а если токены или *n*-граммы встречаются крайне редко, они обычно не несут корреляции с другими словами, пригодной для идентификации тематики, объединяющей документы или даже целые классы документов. Так что редкие *n*-граммы практически бесполезны при классификации. Как вы понимаете, большинство биграмм довольно редки. Не говоря уже о 3- и 4-граммах.

Поскольку словосочетания встречаются реже, чем отдельные слова, размер словаря экспоненциально приближается к количеству *n*-грамм во всех документах в корпусе (наборе текстов). Если размерность вектора признаков превышает длину всех документов, то этап выделения признаков смысла не имеет. В такой ситуации будет практически невозможно избежать переобучения (переподгонки к векторам) модели. В главе 3 мы будем использовать статистику частотностей документа для идентификации *n*-грамм, встречающихся слишком редко, чтобы пригодиться для машинного обучения. Как правило, слишком редкие (например, в трех или менее различных документах) *n*-граммы отфильтровываются. Подобный сценарий соответствует фильтру «редкий токен» сортировщика монет из главы 1.

Теперь рассмотрим противоположную проблему — биграмму *at the* из предыдущей фразы. Наверное, это нередкая комбинация слов. Она может быть настолько

распространена в ваших документах, что непригодна для различения их смысла. Возможности предсказания у такой *n*-граммы очень невелики. Подобно словам и другим токенам, слишком часто встречающиеся *n*-граммы отфильтровываются. Токены или *n*-граммы, встречающиеся в более чем 25 % всех документов в корпусе, обычно игнорируются. Этот принцип схож с фильтром стоп-слов сортировщика монет из главы 1. Использование подобных фильтров полезно как в отношении *n*-грамм, так и для отдельных токенов.

Стоп-слова

Стоп-слова — это распространенные слова на любом языке, которые встречаются очень часто, но несут в себе гораздо меньше содержательной информации о смысле фразы. Вот примеры некоторых распространенных стоп-слов[1]:

- *a*, *an*;
- *the*, *this*;
- *and*, *or*;
- *of*, *on*.

Изначально стоп-слова не использовались в конвейерах NLP. Это делалось для уменьшения сложности вычислений при извлечении информации из текста. Несмотря на то что сами эти слова обычно малоинформативны, они могут нести важную информацию при использовании их как части *n*-граммы. Рассмотрим два примера.

- *Mark reported to the CEO*;
- *Suzanne reported as the CEO to the board.*

В конвейере NLP можно использовать 4-граммы, такие как *reported to the CEO* и *reported as the CEO*. Если убрать стоп-слова из этих 4-грамм, то они превратятся в выражение *reported CEO*, лишившись информации об иерархии в компании. В первом примере Марк, скорее всего, помощник какого-то генерального директора (CEO). Сьюзан, в свою очередь, сама является генеральным директором и докладывает непосредственно совету директоров. К сожалению, сохранение стоп-слов в конвейере приводит к возникновению другой проблемы — увеличению длины *n*-грамм, необходимой для использования связей, формируемых стоп-словами, которые в противном случае окажутся бессмысленными. В результате придется сохранить как минимум 4-граммы, чтобы избежать неоднозначности, описанной в примере с персоналом.

Процесс разработки фильтра стоп-слов зависит от каждого конкретного приложения. Вычислительную сложность и требования к памяти для всех последующих этапов конвейера NLP определяет размер словаря. Стоит отметить, что список стоп-слов составляет лишь небольшую часть словаря. Обычно такие списки содержат около 100 слов. Для отслеживания 95 % слов, встречающихся в большом корпусе твитов, сообщений в блогах и новостных статей, потребуется словарь на 20 000 слов (rstudio-pubsstatic.s3.amazonaws.com/41251_4c55dff8747c4850a7fb26fb9a969c8f.html). К сожалению, это

[1] Более полный список стоп-слов для различных языков можно найти в документации NLTK: raw.githubusercontent.com/nltk/nltk_data/gh-pages/packages/corpora/stopwords.zip.

число относится только к словарю, наполненному единичными *n*-граммами. Словарю биграмм, способному охватить 95 % биграмм большого английского корпуса, потребуется не менее 1 миллиона уникальных токенов.

Определенное беспокойство может вызывать тот факт, что от размера словаря зависит размер тренировочной выборки, необходимой для того, чтобы избежать переподгонки к конкретному слову или словосочетанию, а размер тренировочного набора данных (training set) определяет затраты на обработку. Впрочем, избавление от 100 слов из 20 000 не сильно ускорит вашу работу. При использовании словаря биграмм экономия времени за счет удаления стоп-слов будет ничтожной. Кроме того, если избавляться от этих стоп-слов произвольным образом, не учитывая частоты использующих их биграмм в тексте, потеря информации будет несоразмерно большой. К примеру, вы можете пропустить применение словосочетания *The Shining*[1], являющегося уникальным названием, и в таком случае рассматривать тексты, посвященные этому жестокому, напряженному фильму, так же, как и документы, в которых упоминаются *Shining Light*[2] или *shoe shining*.

Итак, при достаточном количестве оперативной памяти и производительности системы обработки данных для выполнения всех этапов конвейера NLP, развернутого с использованием большого словаря, не стоит заниматься выбрасыванием нескольких не играющих важной роли слов. Если же вы опасаетесь переобучить модель на маленьком тренировочном наборе данных с большим словарем, то существуют и более эффективные способы подбора словаря и понижения размерности, чем игнорирование стоп-слов. Включение в словарь стоп-слов позволяет частотным фильтрам документа (обсуждаемым в главе 3) более точно идентифицировать и игнорировать слова и *n*-граммы, содержащие меньше всего полезной информации в рамках конкретной предметной области.

Для фильтрации же случайным образом при токенизации набора стоп-слов достаточно и спискового включения языка Python. Ниже представлен фрагмент кода, в котором игнорируется несколько стоп-слов при проходе в цикле по списку токенов:

```
>>> stop_words = ['a', 'an', 'the', 'on', 'of', 'off', 'this', 'is']
>>> tokens = ['the', 'house', 'is', 'on', 'fire']
>>> tokens_without_stopwords = [x for x in tokens if x not in stop_words]
>>> print(tokens_without_stopwords)
['house', 'fire']
```

Вы уже обратили внимание, что часть слов содержит куда больше информации по сравнению с остальными? В некоторых предложениях можно опустить больше половины слов без особого ущерба для общего смысла. Зачастую точку зрения можно выразить без артиклей, предлогов и даже форм глагола *to be*. Представьте, что кто-то говорит на языке жестов или в спешке пишет напоминание для себя. Какие слова обычно при этом опускаются? Именно так и выбираются стоп-слова.

На данный момент NLTK, вероятно, наиболее полный список канонических стоп-слов (листинг 2.8).

---

[1] «Сияние» — фильм ужасов режиссера Стэнли Кубрика по одноименному роману Стивена Кинга. — *Примеч. пер.*

[2] Песня североирландской рок-группы Ash. — *Примеч. пер.*

Листинг 2.8. Список стоп-слов NLTK

```
>>> import nltk
>>> nltk.download('stopwords')
>>> stop_words = nltk.corpus.stopwords.words('english')
>>> len(stop_words)
153
>>> stop_words[:7]
['i', 'me', 'my', 'myself', 'we', 'our', 'ours']
>>> [sw for sw in stopwords if len(sw) == 1]
['i', 'a', 's', 't', 'd', 'm', 'o', 'y']
```

Документы, написанные от первого лица, обычно довольно скучны и, что важнее, малоинформативны. Пакет NLTK включает местоимения (не только первого лица) в свой список стоп-слов. Стоит отметить, что эти однобуквенные стоп-слова еще интереснее, однако они имеют смысл, только если вы часто используете токенизатор NLTK и портер Steter. Эти однобуквенные токены часто возникают при разбиении сокращений и выделении основы слова с помощью токенизаторов и стеммеров NLTK.

ПРЕДУПРЕЖДЕНИЕ

Используемый sklearn набор английских стоп-слов сильно отличается от набора NLTK. На момент написания этой книги в sklearn было 318 стоп-слов. NLTK периодически обновляет свои корпусы, в том числе и список стоп-слов.

Перезапустив листинг 2.8 для подсчета стоп-слов NLTK версии 3.2.5 в Python 3.6, мы получили 179 стоп-слов вместо 153 в более ранней версии.

Описанное выше является еще одной причиной для того, чтобы *не* фильтровать стоп-слова. В противном случае воспроизвести ваши результаты будет непросто.

В зависимости от того, сколько информации на естественном языке нужно отбросить, можно использовать в конвейере объединение или пересечение нескольких списков стоп-слов. Ниже приводится сравнение списков стоп-слов sklearn (версия 0.19.2) и nltk (версия 3.2.5) (листинг 2.9).

Листинг 2.9[1]**.** Лист стоп-слов NLTK

```
>>> from sklearn.feature_extraction.text import\
...   ENGLISH_STOP_WORDS as sklearn_stop_words
>>> len(sklearn_stop_words)
318
```

[1] В языке Python операции объединения и пересечения определены для множеств, а не списков, поэтому для работы этого фрагмента кода необходимо внести в него изменения, преобразовав списки во множества, вот так:

len(set(stop_words).union(set(sklearn_stop_words)))

...

len(set(stop_words).intersection(set(sklearn_stop_words))) — *Примеч. пер.*

```
>>> len(stop_words)
179
>>> len(stop_words.union(sklearn_stop_words))
378
>>> len(stop_words.intersection(sklearn_stop_words))
119
```

Список NTLK содержит 60 стоп-слов, отсутствующих в большем списке sklearn

Списки стоп-слов NLTK и sklearn совпадают менее чем на одну треть (119 из 378)

2.2.5. Нормализация словаря

Надеемся, вы уже успели заметить, какую важную роль в производительности конвейера NLP играет размер словаря. Еще одна из техник уменьшения его размера — объединение токенов, означающих сходные вещи, в единую нормализованную форму. Подобная техника позволяет уменьшить количество хранимых токенов и улучшает связи между смыслами фраз с разным «написанием» токенов или *n*-грамм, а также, как уже упоминалось, снижает вероятность переобучения.

Выравнивание регистра

Выравнивание регистра (case folding) — объединение нескольких вариантов написания слова, различающихся только регистром букв. Для чего вообще это нужно? Различные слова могут денормализоваться по регистру из-за написания с прописной буквы слов в начале предложения или вообще написания ТОЛЬКО ПРОПИСНЫМИ буквами для выделения в тексте. Противоположная денормализации операция называется *нормализацией* (case normalization) или *выравниванием* регистра (case folding). Нормализация регистра слов и отдельных букв — один из способов уменьшения размера словаря и обобщения конвейера NLP. Данный способ помогает объединить слова, которые должны означать одно и то же (и быть написанными одинаково) в один токен.

Впрочем, иногда регистр слов несет в себе определенный смысл. Например, смысл слов *doctor* и *Doctor* различен. Зачастую использование прописных букв означает, что слово является именем собственным, именем человека, названием места или предмета. В случаях, когда распознавание именованных объектов важно для конвейера, необходимо отличать имена собственные от других слов. Если токены не нормализованы по регистру, размер словаря, объем требуемой для него памяти и время обработки будут примерно в два раза больше. Может также потребоваться маркировать больше тренировочных данных конвейера NLP для схождения его к точному общему решению. Как и в любом другом конвейере машинного обучения, маркированный набор используемых при обучении данных должен быть «представительным» для пространства всех возможных векторов признаков, с которыми должна работать модель, включая варианты с различным регистром. Для 100 000-мерных векторов мультимножеств слов понадобится не менее 100 000 маркированных примеров, а иногда даже больше, для обучения конвейера машинного обучения с учителем без риска переобучения. В некоторых ситуациях сокращение размера словаря в два раза оправдывает потерю информационного наполнения.

В языке программирования Python нормализацию регистра символов можно легко осуществить с помощью спискового включения.

```
>>> tokens = ['House', 'Visitor', 'Center']
>>> normalized_tokens = [x.lower() for x in tokens]
>>> print(normalized_tokens)
['house', 'visitor', 'center']
```

Если вы уверены, что хотите нормализовать регистр во всем документе, можете уменьшить регистр текстовой строки за одну операцию с помощью команды `lower()` перед токенизацией. К сожалению, это помешает работе продвинутых токенизаторов, умеющих разбивать слова, написанные в *верблюжьем регистре*, вроде WordPerfect, FedEx или stringVariableName (https://ru.wikipedia.org/wiki/CamelCase). Например, чтобы обеспечить уникальность токена WordPerfect или из ностальгических воспоминаний о более совершенной эре обработки текстов. Только от вас зависит, когда и как применять выравнивание регистра.

Путем нормализации регистра мы пытаемся вернуть токены в их «нормальное» состояние, то есть в состояние, в котором они находились до того, как грамматические правила и их положение в предложении повлияли на регистр букв. Самый простой и наиболее распространенный способ нормализовать регистр текста — понизить регистр всех символов с помощью функции, подобной встроенной функции `str.lower()`[1] языка Python. К сожалению, при этом подходе также переводится в нижний регистр много осмысленных прописных букв, кроме желаемой первой буквы предложения.

Смена регистра первого слова в предложении сохраняет смысл имен собственных в середине предложения, таких как *Joe* и *Smith* в *Joe Smith*. Помимо прочего, правильно группируются слова, которые употребляются совместно, поскольку они пишутся с прописной буквы только тогда, когда находятся в начале предложения, раз они не являются именами собственными. Это предотвращает путаницу *Joe* с *coffee (joe)*[2] во время токенизации, а также слова *smith* в значении «кузнец» с именем собственным *Smith* в предложении типа *A word smith had a cup of joe*[3]. Стоит отметить, что даже при таком осторожном подходе к нормализации регистра, когда в нижний регистр переводятся только прописные буквы в начале предложения, все равно возникают ошибки регистра для редких имен собственных в начале предложений. Набор токенов для предложения *Joe Smith, the word smith, with a cup of joe* будет

---

[1] Мы имеем в виду поведение str.lower() в Python 3. В Python 2 байты (строки) могут быть переведены в нижний регистр путем простого смещения всех буквенных символов в пространстве номеров (ord) ASCII. Но в Python 3 str.lower() переводит символы в нижний регистр как полагается, так что может корректно учитывать вычурные английские символы (например, диакритический знак «острое ударение» над символом e в resumé), а также различные особенности использования заглавных букв в неанглийских языках.

[2] Устойчивое выражение cup of joe (en.wiktionary.org/wiki/cup_of_joe) является сленговым обозначением чашки кофе.

[3] Word smith — английское выражение, означающее искусного в письменной речи человека (писателя, журналиста и т. д.). Часто употребляется в слегка ироническом смысле («акула пера»). — *Примеч. пер.*

отличаться от набора, получаемого из *Smith the word with a cup of joe, Joe Smith.* А этот эффект может быть нежелательным. Кроме того, нормализация регистра бесполезна для языков, в которых нет понятия прописных букв.

Во избежание этой возможной потери информации, во многих конвейерах NLP регистр вообще не нормализуется. Для многих приложений выигрыш в эффективности (в смысле хранения и скорости обработки) за счет уменьшения размера словаря примерно в два раза не окупает потерю полезной информации в именах собственных. Стоит отметить, что некоторая информация может быть утеряна даже без нормализации регистра. В некоторых приложениях может вызвать проблемы то, что *The* в начале фразы не определяется как стоп-слово. По-настоящему продвинутые конвейеры выявляют имена собственные и лишь затем выборочно нормализуют регистр только тех слов в начале предложения, которые точно не являются именами собственными. Можете использовать любой подход к нормализации, удобный для вашего приложения. Если в корпусе не так много *Smith's* и *word smiths* и вам все равно, будут ли у них одинаковые токены, можно просто перевести все слова в нижний регистр. Лучше всего попробовать несколько разных подходов и посмотреть, какой из них окажется наиболее эффективным для задач конкретного NLP-проекта.

Нормализация регистра позволяет уменьшить вероятность переобучения конвейера за счет обобщения модели для работы с текстом с нестандартным регистром букв. Нормализация особенно полезна для поисковых систем. При поиске нормализация увеличивает количество найденных совпадений для конкретного запроса. Это часто называют метрикой полноты поисковой системы (или любой другой модели классификации)[1].

При поиске по словам *Age* и *age* в поисковой системе без нормализации будут выданы различные наборы документов. *Age*, вероятно, встретится в таких фразах, как *New Age* (https://ru.wikipedia.org/wiki/Нью_Эйдж) или *Age of Reason* (https://ru.wikipedia.org/wiki/Век_разума). Слово *age*, в свою очередь, с большей вероятностью встретится в таких фразах, как *at the age of* из нашего предложения о Томасе Джефферсоне. Нормализация словаря в поисковом индексе (а также в запросе) гарантирует, что будут возвращаться оба типа документов с *age*, независимо от регистра запроса пользователя.

К сожалению, повышение полноты приводит к понижению точности выдаваемых ответов и возврату многих документов, абсолютно не интересующих пользователя. Из-за этой проблемы современные поисковые системы позволяют пользователям отключать нормализацию для каждого запроса[2] путем постановки тех слов, для которых важно соблюдение регистра, в кавычки. При конструировании конвейера для такой поисковой машины необходимо создавать два индекса для документов, чтобы учесть оба типа запросов: один с нормализованным регистром *n*-грамм, а второй — с оригинальной расстановкой прописных букв.

---

[1] Больше о точности и чувствительности можно узнать в приложении Г. В статье с сайта Webology приведено сравнение чувствительности различных поисковых систем (www.webology.org/2005/v2n2/a12.html).

[2] Далеко не все. Например, поиск Google всегда производится с нормализацией и нет возможности отключить ее. — *Примеч. пер.*

Стемминг

Другим распространенным методом нормализации является устранение небольших смысловых различий, связанных с окончаниями множественного числа и притяжательными окончаниями слов или даже различными формами глаголов. Данный способ нормализации, заключающийся в поиске общей основы различных форм слова, называется «стемминг». Например, общая основа слов *housing* и *houses* — *house*. Стемминг удаляет суффиксы из слов в попытке сгруппировать слова со схожим смыслом под общей основой. Основа не обязана быть допустимым орфографически словом, она может быть просто токеном или меткой, представляющей несколько возможных вариантов написания.

Для человека очевидно, что *house* и *houses* — это формы единственного и множественного числа одного и того же существительного. Однако до машины эту информацию еще нужно каким-то образом донести. Одним из основных преимуществ стемминга является уменьшение количества слов, значения которых программное обеспечение или языковая модель должны отслеживать. Данный метод уменьшает размер словаря, по максимуму ограничивая потерю полезной информации и смыслов. В машинном обучении это называется понижением размерности. За счет него у модели расширяются возможности обобщения, и она может одинаково обрабатывать все слова с одной основой. Таким образом, если приложению не нужно различать *house* и *houses*, с помощью стемминга можно уменьшить объем кода или набора данных вдвое или даже больше в зависимости от агрессивности выбранного стеммера.

Стемминг играет важную роль при поиске по ключевым словам или информационном поиске. Он позволяет, скажем, искать по фразе *developing houses in Portland* и получить веб-страницы или документы, включающие как *house* и *houses*, так и *housing*, потому что все они связаны с токеном *hous*. Но точно так же можно получить страницы со словами *developer* и *development*, так как основой всех этих слов является *develop*. Как видите, такое «расширение» поиска понижает вероятность того, что вы пропустите связанный с темой документ или веб-страницу. Такое расширение результатов поиска значительно улучшило бы показатель полноты, демонстрируя, насколько хорошо поисковая система возвращает все релевантные документы[1].

Стоит отметить, что стемминг может значительно снизить показатель точности поисковой системы, поскольку она может возвращать гораздо больше не относящихся к делу документов, помимо релевантных. В некоторых приложениях эта доля ложноположительных результатов (доля возвращаемых бесполезных страниц) может быть проблемой. Поэтому большинство поисковых систем позволяет отключать стемминг и даже нормализацию регистра — для этого следует заключить в кавычки слово или фразу. Использование кавычек означает, что вам нужны только страницы, содержащие точное написание фразы, например *Portland Housing Development software*. Подобный запрос не вернул бы документ с фразой *a Portland software developer's house*. Иногда бывает нужно найти информацию по запросу *Dr.*

[1] См. приложение Г, если вы забыли, как измерять чувствительность, или посетите страницу по адресу en.wikipedia.org/wiki/Precision_and_recall.

House's calls, но не по запросу *dr. house call*, которая была бы возвращена в случае использования стеммера.

Вот простая реализация стеммера на чистом Python, умеющего обрабатывать конечные символы *S's*:

```
>>> def stem(phrase):
...     return ' '.join([re.findall('^(.*ss|.*?)(s)?$',
...         word)[0][0].strip("'") for word in phrase.lower().split()])
>>> stem('houses')
'house'
>>> stem("Doctor House's calls")
'doctor house call'
```

Описанная выше функция стеммера следует нескольким простым правилам, в рамках указанного короткого регулярного выражения.

- Если слово оканчивается более чем одним *s*, основой считается слово, а суффиксом — пустая строка.
- Если слово оканчивается одним *s*, основа — это слово без *s*, а суффикс — *s*.
- Если слово не заканчивается на *s*, основа — это слово, суффикс не возвращается.

Метод удаления суффиксов приводит к выделению основы по крайней мере некоторых притяжательных слов наряду со словами во множественном числе.

Функция хорошо работает в обычных условиях, однако подходит и для более сложных случаев. Например, вышеприведенные правила нарушаются для слов вроде *dishes* или *heroes*. Для таких более сложных случаев в пакете NLTK предусмотрены другие стеммеры.

Данная функция также не справляется со словом *housing* из примера про *Portland Housing*.

Двумя наиболее популярными алгоритмами являются стеммер Портера и Snowball. Стеммер Портера назван в честь специалиста в области компьютерных наук Мартина Портера (Martin Porter)[1]. Портеру мы обязаны и усовершенствованной версией его стеммера под названием Snowball[2]. Мартин посвятил большую часть своей долгой карьеры документированию и улучшению стеммеров ввиду важности их роли в поиске информации (поиске по ключевым словам). Описанные выше стеммеры реализуют более сложные, чем обычные регулярные выражения, правила. Это позволяет справляться со сложностями правил правописания и окончания слов английского языка:

```
>>> from nltk.stem.porter import PorterStemmer
>>> stemmer = PorterStemmer()
>>> ' '.join([stemmer.stem(w).strip("'") for w in
...   "dish washer's washed dishes".split()])
'dish washer wash dish'
```

[1] См. статью: *Porter M. F.* An algorithm for suffix stripping, 1993 (www.cs.odu.edu/~jbollen/IR04/readings/readings5.pdf).

[2] См. веб-страницу Snowball: A language for stemming algorithms по адресу snowball.tartarus.org/texts/introduction.html.

Обратите внимание, что стеммер Портера, как и стеммер на основе регулярных выражений, сохраняет конечный апостроф (если вы явно не удалите его). Этот факт гарантирует, что притяжательные слова можно будет отличить от слов, не обозначающих принадлежность. Слова, обозначающие принадлежность, часто являются именами собственными, так что этот признак может быть важен для приложений, в которых имена нужно оборвать иначе, чем другие существительные.

Больше о стеммере Портера

Джулия Менчавез (Julia Menchavez) любезно поделилась своим переводом оригинального алгоритма стеммера Портера на чистый Python (github.com/jedijulia/porter-stemmer/blob/master/stemmer.py). Если вы когда-нибудь испытывали желание разработать собственный стеммер, обратите внимание на эти 300 строк кода и на колоссальный труд по доведению до совершенства, который Джулия вложила в них.

Алгоритм Портера состоит из восьми этапов: 1a, 1b, 1c, 2, 3, 4, 5a и 5b. Шаг 1a немного похож на наше регулярное выражение для обработки завершающих слово символов *S's*[1]:

```
def step1a(self, word):
    if word.endswith('sses'):
        word = self.replace(word, 'sses', 'ss')
    elif word.endswith('ies'):
        word = self.replace(word, 'ies', 'i')
    elif word.endswith('ss'):
        word = self.replace(word, 'ss', 'ss')
    elif word.endswith('s'):
        word = self.replace(word, 's', '')
    return word
```

Это совсем не похоже на str.replace(). Функция str.replace() Джулии изменяет только окончание слова

Остальные семь шагов намного сложнее, потому что они должны иметь дело с правилами правописания английского языка для следующих пунктов.

- **Шаг 1a** — окончания *s* и *es*.
- **Шаг 1b** — окончания *ed*, *ing* и *at*.
- **Шаг 1c** — окончания *y*.
- **Шаг 2** — окончания, преобразующие слово в существительное, например *ational*, *tional*, *ence* и *able*.
- **Шаг 3** — окончания прилагательных, такие как *icate*[2], *ful* и *alize*.
- **Шаг 4** — окончания прилагательных и существительных вроде *ive*, *ible*, *ent* и *ism*.
- **Шаг 5a** — все еще встречающиеся «упрямые» окончания *e*.
- **Шаг 5b** — завершающие удвоенные согласные, из-за которых основа оканчивается одним *l*.

[1] Это сокращенная версия реализации стеммера Портера Джулии Менчавез (Julia Menchavez) на GitHub (github.com/jedijulia/porter-stemmer/blob/master/stemmer.py).

[2] Извини, Чик, алгоритму Портера не нравится твое имя пользователя obfuscate ;).

Лемматизация

При наличии информации о связях между значениями различных слов можно связать несколько слов вместе, даже если их написание сильно различается. Подобная более расширенная нормализация слова до его семантического корня — леммы — называется лемматизацией.

В главе 12 мы покажем, как с помощью лемматизации упростить логику, необходимую для ответа чат-бота на высказывание. Процедура лемматизации принесет пользу любому конвейеру NLP, которому необходимо реагировать аналогично на несколько различных вариантов написания одного и того же основного корня. Лемматизация уменьшает количество слов, на которые нужно реагировать, а значит, и размерность языковой модели. Благодаря лемматизатору модель становится более общей, но также и менее точной, потому что считает одним словом все варианты написания заданного корневого слова. К примеру, слова *chat*, *chatter*, *chatty*, *chatting* и, возможно, даже *chatbot* будут рассматриваться одинаково в использующем лемматизацию конвейере NLP. Последний не обращает внимания на то, что их смысл различен. Точно так же слова *bank*, *banked* и *banking* будут считаться стемминговым конвейером одним словом, несмотря на «речной» смысл слова *bank*, «мотоциклетное» значение *banked* и финансовое *bank*.

По мере чтения данного раздела порассуждайте о словах, в которых применение лемматизации может радикально изменить их значение, возможно, даже на противоположное. Подумайте также о том, насколько будет отличаться реакция конвейера на эти слова. Подобные сценарии обычно называются *спуфингом* — преднамеренной попыткой добиться неправильного ответа от конвейера машинного обучения с помощью искусно сконструированных сложных входных данных.

Лемматизация — потенциально куда более точный способ нормализации, чем стемминг или выравнивание регистра, поскольку учитывает значение слова. Лематизатор использует базу знаний синонимов и окончаний слов, чтобы объединять в один токен только близкие по смыслу слова.

Некоторые лемматизаторы используют частеречную разметку (part of speech, POS) слов, помимо их написания, для повышения точности. Частеречная метка слова указывает на его роль в построении грамматики фразы или предложения. Например, метка «существительное» предназначена для слов, обозначающих людей, места или вещи внутри фразы. «Прилагательное» — к словам, дополняющим или описывающим существительное. «Глагол» относится к действиям. Невозможно выяснить часть речи отдельного слова. Для определения метки части речи должен быть известен контекст. Именно из-за этого некоторые продвинутые лемматизаторы нельзя применять в отношении изолированных слов.

Можно ли с помощью этих частеречных меток найти лучший «корень» для слова, чем предлагает стеммер? Рассмотрим слово *better*. Стеммер обрезал бы окончание *er* и вернул основу *bett* или *bet*. В таком случае *better* смешивалось бы с *betting*, *bets* и *Bet's* вместо более похожих слов, таких как *betterment*, *best* или даже *good* и *goods*.

Как видите, для большинства приложений лемматизаторы подходят лучше, чем стеммеры. Последние используются только в крупномасштабных информацион-

но-поисковых приложениях (поиск по ключевым словам). Если вы действительно хотите уменьшить размерность и повысить чувствительность стеммера в вашем информационно-поисковом конвейере, то нужно применять лемматизатор прямо перед стеммером. Ввиду того что лемма слова — допустимое английское слово, выходные данные лемматизатора отлично подходят в качестве входных данных стеммера. Эта уловка позволяет понизить размерность и повысить чувствительность даже больше, чем один стеммер[1].

Как определить леммы слов в Python? В пакете NLTK есть специальные функции для этого. Обратите внимание, что необходимо указать `WordNetLemmatizer` интересующую часть речи, чтобы найти наиболее точную лемму:

```
>>> nltk.download('wordnet')
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize("better")
'better'
>>> lemmatizer.lemmatize("better", pos="a")
'good'
>>> lemmatizer.lemmatize("good", pos="a")
'good'
>>> lemmatizer.lemmatize("goods", pos="a")
'goods'
>>> lemmatizer.lemmatize("goods", pos="n")
'good'
>>> lemmatizer.lemmatize("goodness", pos="n")
'goodness'
>>> lemmatizer.lemmatize("best", pos="a")
'best'
```

Часть речи по умолчанию — n (существительное)

Параметр a обозначает прилагательное

Возможно, вы удивитесь, что первая попытка лемматизации *better* не изменила его вообще. Так произошло потому, что часть речи слова может сильно влиять на его значение. Если часть речи для слова не указана, лемматизатор NLTK предполагает, что это существительное. При указании правильной части речи, *"a"* в случае прилагательного, лемматизатор возвращает правильную лемму. К сожалению, лемматизатор NLTK ограничен связями, описанными внутри графа значений слов Wordton WordNet. Таким образом, в результате лемматизации *best* не получается тот же корень, что и при лемматизации *better*. В этом графе также отсутствует связь между *goodness* и *good*. Стеммер Портера создал бы эту связь, просто убирая окончание *ness* из всех слов:

```
>>> stemmer.stem('goodness')
'good'
```

Сценарии использования

В каких случаях следует использовать лемматизатор, а в каких — стеммер? Последние, как правило, работают быстрее и требуют менее сложного кода и меньших наборов данных. Однако они более подвержены ошибкам и сводят к одной

[1] Огромная благодарность Кайлу Горману (Kyle Gorman) за то, что указал на это.

основе гораздо большее количество слов, сокращая тем самым информационное содержание (смысл) текста намного сильнее, чем лемматизаторы. Обе технологии уменьшают размер словаря и увеличивают неоднозначность текста, однако лемматизаторы работают лучше, сохраняя как можно больше полезной информации на основе применения слова в тексте и его желаемого смысла. Поэтому некоторые пакеты NLP, такие как spaCy, не включают функции для стемминга, а только методы для лемматизации.

Если приложение связано с поиском информации, использование стемминга и лемматизации повысит его чувствительность и сопоставит тем же словам запроса больше документов. Тем не менее стемминг, лемматизация и выравнивание регистра значительно снижают точность результатов поиска. Эти подходы к сжатию размеров словаря приведут к тому, что система извлечения информации (поисковая система) выдаст много документов, не относящихся к первоначальному значению слов. Поскольку результаты поиска можно ранжировать по релевантности, поисковые системы и индексы документов часто используют стемминг или лемматизацию, чтобы увеличить вероятность того, что эти результаты включают в себя искомые пользователем документы. Но для ранжирования результатов перед их выдачей пользователю они объединяют результаты поиска для обеих версий слов (обычной и полученной в результате стемминга)[1].

Впрочем, для основанного на поиске чат-бота точность играет более важную роль. Поэтому бот должен сначала найти самое близкое соответствие, используя слова, не прошедшие ни стемминг, ни нормализацию, и только после этого вернуться к соответствиям прошедших стемминг и нормализованных токенов. Соответствия первого типа должны располагаться выше в списке результатов.

ВАЖНО

Подводя итог сказанному выше, хочется попросить максимально избегать использования стемминга и лемматизации, за исключением небольших текстов, содержащих искомые слова в различном регистре. Учитывая лавинообразный рост размеров наборов данных NLP, такое редко имеет место для документов на английском языке, разве что тексты изобилуют жаргонизмами или относятся к очень узкой области науки, техники или литературы. Впрочем, для текстов, написанных на отличных от английского языках, лемматизация все еще может принести пользу. Стэнфордский курс по поиску информации целиком исключает стемминг и лемматизацию ввиду пренебрежимо малого увеличения чувствительности и сильного снижения уровня точности[2].

---

[1] Для корректировки ранжирования результатов поиска также используются дополнительные метаданные. Duck Duck Go и другие популярные веб-поисковые системы применяют более 400 независимых алгоритмов (включая пользовательские алгоритмы) для ранжирования результатов поиска (duck.co/help/results/).

[2] См. статью Stemming and lemmatization по адресу nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatizatioN-1.html.

2.3. Тональность

Независимо от того, используются ли в конвейере NLP необработанные токены из одного слова, *n*-граммы, основы или леммы, каждый из этих токенов содержит какую-то информацию. Ее важной частью является тональность — общее чувство или эмоция, которую вызывает это слово. *Анализ тональности* — оценка тональностей фраз или фрагментов текста — распространенный вариант использования NLP. Во многих компаниях это главное, что требуется от инженера, ответственного за обработку естественного языка.

Компании хотят знать, что пользователи думают об их продуктах. Именно поэтому они часто дают возможность оставить отзывы. Рейтинг в виде звезд на Amazon или Rotten Tomatoes — всего лишь один из способов получения количественных данных об отношении людей к приобретаемым ими товарам. Но более естественный способ получения этой информации — применение комментариев на естественном языке. Предоставление пользователю «чистой доски» (пустого текстового поля) для комментариев по поводу товара может дать более подробную информацию о мнении клиентов.

Раньше пришлось бы читать все эти отзывы. Только человек может понять вкладываемые в текст на естественном языке эмоции и чувства, правда? Если бы вам все-таки пришлось читать эти тысячи обзоров, то вы бы поняли, насколько легко эта деятельность утомляет и какое большое количество ошибок вызывает. Люди очень плохо читают отзывы, особенно критикующие или негативные. Клиенты же, как правило, не слишком хорошо умеют выражать свои чувства так, чтобы пробиться через естественные человеческие триггеры и фильтры.

Машины же лишены предубеждений и эмоциональных триггеров. Люди — не единственные, кто может обрабатывать текст на естественном языке и находить в нем информацию и даже смысл. Конвейер NLP может быстро и объективно обработать большое количество отзывов пользователей с меньшей предвзятостостью. Кроме того, конвейер NLP может выдать численную оценку положительности, отрицательности или любой другой эмоциональной характеристики текста.

Другой распространенный способ применения анализа тональности — фильтрация нежелательной почты и провокационных сообщений. Чат-бот должен оценивать тональность обрабатываемых им сообщений в чате и реагировать на них соответствующе. Что еще более важно, оценивать тональность собственных сообщений, что поможет сделать чат-бот более настроенным на общение и дружелюбным. Самый простой способ — последовать типичному совету любой мамы: «Если не можешь сказать что-то приятное — не говори ничего». Именно поэтому необходимо, чтобы наш бот сначала оценивал тональность всего, что ему говорят, а потом решал, стоит ли что-либо отвечать.

Какой тип конвейера вы бы создали для оценки тональности блока текста и получения числового эквивалента настроения? Пусть нам требуется просто оценить позитивную направленность текста — насколько автору текста понравились товар или услуга. Допустим, наш конвейер NLP и алгоритм анализа тональностей должен

возвращать одно число с плавающей точкой в диапазоне от –1 до +1. Он должен выводить +1 для текста с положительным посылом, например «Абсолютно идеально! Мне нравится! :-) :-) :-)», и –1 для текста с отрицательным посылом: «Ужасно! Совершенно бесполезно. :(». Можно также использовать значения, близкие к 0, например +0,1, для высказываний типа «Все нормально. Есть позитивные и негативные стороны».

Существует два подхода к анализу тональности:

- алгоритм, основанный на правилах, создаваемый человеком;
- модель *машинного обучения*, усваиваемая машиной из данных.

Первый подход к анализу тональностей использует для оценки тональности разработанные человеком правила, иногда называемые *эвристическим* алгоритмом. Распространенный подход к анализу тональностей заключается в поиске ключевых слов в тексте и сопоставлении с каждым из них числового показателя (веса) в словаре или ассоциативном массиве — вроде объекта типа `dict` в языке Python. Теперь благодаря токенизации наш словарь может включать основы, леммы или токены *n*-грамм, а не просто слова. «Правило» нашего алгоритма будет состоять в суммировании показателей для всех ключевых слов в документе, которые есть в словаре показателей тональностей. Конечно, придется составить вручную этот словарь ключевых слов и их показателей тональностей, прежде чем применить алгоритм к массиву текста. Мы покажем, как это сделать с помощью алгоритма VADER (из библиотеки `sklearn`), в нижеприведенном коде.

Для второго подхода, машинного обучения, требуется маркированный набор высказываний или документов, на котором модель обучается созданию этих правил. Модель измерения тональности учится обрабатывать вводимый текст и выводить числовой показатель интересующего нас настроения, например положительного, спамовости или провокационности. Для данного подхода требуется большое количество исходных данных, помеченных «правильными» показателями настроения. Для реализации этого подхода часто используется лента Twitter, поскольку хештеги, такие как *#awesome*, *#happy* или *#sarcasm*, часто можно использовать для создания набора автомаркированных данных. Например, у вашей компании могут быть рейтинги товаров в звездочках, соответствующие отзывам. Этот рейтинг в звездах можно использовать в качестве числового показателя уровня позитивности отзывов. Вскоре мы продемонстрируем, как обрабатывать подобный набор данных и обучить основанный на токенах алгоритм, который называется *«наивный байесовский классификатор»*, а также измерять степень позитивности тональности для набора отзывов после обсуждения VADER.

2.3.1. VADER — анализатор тональности на основе правил

Гатто (Hutto) и Гилберт (Gilbert) из GA Tech разработали один из первых успешных алгоритмов анализа тональности на базе правил. Они назвали свой алгоритм VADER, что расшифровывается как **V**alence **A**ware **D**ictionary for s**E**ntiment

Reasoning (учитывающий валентность словарь для анализа тональностей)[1]. Многие пакеты NLP реализуют один из видов этого алгоритма. Пакет NLTK включает реализацию алгоритма VADER в модуле `nltk.sentiment.vader`. Сам Гатто сопровождает Python-пакет `vaderSentiment`, именно его мы и будем здесь использовать.

Для запуска следующего примера необходимо ввести команду `pip install vaderSentiment`[2]. Мы не включали это в пакет `nlpia`:

```
>>> from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
>>> sa = SentimentIntensityAnalyzer()
>>> sa.lexicon
{ ...
':(': -1.9,
':)': 2.0,
...
'pls': 0.3,
'plz': 0.3,
...
'great': 3.1,
... }
>>> [(tok, score) for tok, score in sa.lexicon.items()
...   if " " in tok]
[("( '}{' )", 1.6),
 ("can't stand", -2.0),
 ('fed up', -1.8),
 ('screwed up', -1.5)]
>>> sa.polarity_scores(text=\
...   "Python is very readable and it's great for NLP.")
{'compound': 0.6249, 'neg': 0.0, 'neu': 0.661,
'pos': 0.339}
>>> sa.polarity_scores(text=\
...   "Python is not a bad choice
       for most applications.")
{'compound': 0.431, 'neg': 0.0,
 'neu': 0.711, 'pos': 0.289}
```

SentimentIntensityAnalyzer.lexicon содержит словарь токенов и показателей, о котором мы говорили ранее

Наилучший вариант для работы VADER — когда токенизатор хорошо справляется со знаками препинания и смайликами. В конце концов, смайлики созданы для передачи эмоций

При использовании стеммера (или лемматизатора) в конвейере необходимо также применить его к лексикону VADER, сочетая баллы для всех слов, относящихся к одной основе или лемме

Из 7500 токенов, определенных в VADER, только три содержат пробелы и только два из них на самом деле являются n-граммами; третий — смайлик, означающий поцелуй

Алгоритм VADER учитывает интенсивность направленности тональности в трех отдельных показателях (положительная, негативная и нейтральная), а затем объединяет их вместе в составную оценку тональности

Обратите внимание, что VADER довольно хорошо справляется с отрицанием: great обладает чуть более позитивной тональностью, чем not bad. Встроенный токенизатор VADER игнорирует любые слова, не входящие в его лексикон, и вообще не учитывает n-граммы

[1] Гатто (Hutto) и Гилберт (Gilbert), VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text: comp.social.gatech.edu/papers/icwsm14.vader.hutto.pdf.

[2] Вы можете найти более детальную информацию о процессе установки на GitHub по адресу github.com/cjhutto/vaderSentiment.

Посмотрим, как этот основанный на правилах подход работает с вышеупомянутыми примерами высказываний:

```
>>> corpus = ["Absolutely perfect! Love it! :-) :-) :-)",
...           "Horrible! Completely useless. :(",
...           "It was OK. Some good and some bad things."]
>>> for doc in corpus:
...     scores = sa.polarity_scores(doc)
...     print('{:+}: {}'.format(scores['compound'], doc))
+0.9428: Absolutely perfect! Love it! :-) :-) :-)
-0.8768: Horrible! Completely useless. :(
+0.3254: It was OK. Some good and some bad things.
```

Похоже, то что надо. Единственным недостатком VADER является то, что он не просматривает все слова в документе, только около 7500. Как же добиться, чтобы все слова учитывались в показателе тональности? И как избежать кодирования вручную своего понимания слов в словаре, состоящем из тысяч слов или добавления пользовательских слов в словарь `SentimentIntensityAnalyzer.lexicon`? Использование подхода, основанного на правилах, невозможно без знания языка, а значит, и понимания, какие показатели должны быть в словаре!

Именно для таких случаев предназначены конвейеры машинного обучения.

2.3.2. Наивный байесовский классификатор

Наивная байесовская модель нацелена на поиск в наборе документов ключевых слов, прогнозирующих значение целевой (выходной) переменной. Когда предсказываемой целевой переменной является тональность, эта модель ищет предугадывающие такую тональность слова. Приятной особенностью наивного байесовского классификатора является то, что его внутренние коэффициенты задают соответствие слов или токенов показателям аналогично алгоритму VADER. Только на этот раз показатели не ограничиваются взглядами отдельного человека. Машина найдет оптимальные показатели для решения любой задачи.

Для любого алгоритма машинного обучения требуется набор данных. Необходима подборка текстовых документов с метками позитивного эмоционального содержания (позитивной тональности). При создании VADER Гатто подобрал четыре различных набора данных тональностей. Вы можете загрузить их из пакета `nlpia`[1]:

```
>>> from nlpia.data.loaders import get_data
>>> movies = get_data('hutto_movies')
>>> movies.head().round(2)
    sentiment                                               text
id
1        2.27  The Rock is destined to be the 21st Century...
2        3.53  The gorgeously elaborate continuation of ''...
3       -0.60                     Effective but too tepid ...
4        1.47  If you sometimes like to go to the movies t...
5        1.73  Emerges as something rare, an issue movie t...
```

[1] Если вы еще не установили nlpia, инструкции по установке можно найти по адресу github.com/totalgood/nlpia.

```
>>> movies.describe().round(2)
       sentiment
count   10605.00
mean        0.00
min        -3.88
max         3.94
```

Похоже, фильмы оценивались по шкале от –4 до +4

Теперь токенизируем все тексты рецензий, чтобы создать мультимножества слов для каждого из них. Мы поместим их все в объект DataFrame библиотеки Pandas, подобно тому как мы делали раньше в этой главе:

Эта строка позволит аккуратнее отображать широкие Data Frames в консоли

Функция casual_tokenize из NLTK может обрабатывать смайлики, нестандартные знаки препинания и сленг лучше, чем Treeken Word Tokenizer или другие токенизаторы, описанные в этой главе

Встроенный класс Counter языка Python берет список объектов и подсчитывает их, возвращая словарь, в котором ключами являются объекты (токены в нашем случае), а значениями — целочисленные количества этих объектов

```
>>> import pandas as pd
>>> pd.set_option('display.width', 75)
>>> from nltk.tokenize import casual_tokenize
>>> bags_of_words = []
>>> from collections import Counter
>>> for text in movies.text:
...     bags_of_words.append(Counter(casual_tokenize(text)))
>>> df_bows = pd.DataFrame.from_records(bags_of_words)
>>> df_bows = df_bows.fillna(0).astype(int)
>>> df_bows.shape
(10605, 20756)
>>> df_bows.head()
   !  "  #  $  %  &  ' ...  zone  zoning  zzzzzzzzz  ½  élan  –  '
0  0  0  0  0  0  0  4 ...     0       0          0  0     0  0  0
1  0  0  0  0  0  0  4 ...     0       0          0  0     0  0  0
2  0  0  0  0  0  0  0 ...     0       0          0  0     0  0  0
3  0  0  0  0  0  0  0 ...     0       0          0  0     0  0  0
4  0  0  0  0  0  0  0 ...     0       0          0  0     0  0  0
>>> df_bows.head()[list(bags_of_words[0].keys())]
   The  Rock  is  destined  to  be ...  Van  Damme  or  Steven  Segal .
0    1     1   1         1   2   1 ...    1      1   1       1      1 1
1    2     0   1         0   0   0 ...    0      0   0       0      0 4
2    0     0   0         0   0   0 ...    0      0   0       0      0 0
3    0     0   1         0   4   0 ...    0      0   0       0      0 1
4    0     0   0         0   0   0 ...    0      0   0       0      0 1
```

Таблица мультимножества слов может разрастаться очень быстро, особенно если не использовать нормализацию регистра, фильтры стоп-слов, стемминг и лемматизацию, которые мы обсуждали ранее в этой главе. Попробуйте вставить некоторые из этих методов понижения размерности и посмотрите, как они повлияют на конвейер

Numpy и Pandas могут представлять NaN только в объектах типа float, поэтому, если заполнить все NaN нулями, можно преобразовать DataFrame в целочисленные, гораздо более компактные (и в памяти, и при отображении)

Конструктор типа DataFrame from_records() принимает на входе последовательность словарей. Он создает столбцы для всех ключей с добавлением в таблицу значений в соответствующих столбцах, причем недостающие значения заполняются NaN

Теперь у нас есть все данные, необходимые для модели наивного байесовского классификатора, чтобы искать ключевые слова, предсказывающие тональность текста на естественном языке:

Наивные байесовские модели являются классификаторами, поэтому необходимо преобразовать выходную переменную (тональность в виде значения типа float с плавающей точкой) в дискретную метку (типа integer, string или bool)

Преобразуем двоичную переменную классификатора (0 или 1) в –4 или 4, чтобы сравнить ее с «эталонной» тональностью. Для получения непрерывного значения применяем метод nb.predict_proba

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> nb = MultinomialNB()
>>> nb = nb.fit(df_bows, movies.sentiment > 0)
>>> movies['predicted_sentiment'] =\
>>> movies['predicted_sentiment'] = nb.predict_proba(df_bows) [:,1] * 8 - 4
>>> movies['error'] = (movies.predicted_sentiment - movies.sentiment).abs()
>>> movies.error.mean().round(1)
2.4
>>> movies['sentiment_ispositive'] = (movies.sentiment > 0).astype(int)
>>> movies['predicted_ispositive'] = (movies.predicted_sentiment > 0).astype(int)
>>> movies['''sentiment predicted_sentiment sentiment_ispositive\
...   predicted_ispositive'''.split()].head(8)
    sentiment  predicted_sentiment  sentiment_ispositive  predicted_ispositive
id
1    2.266667                    4                     1                     1
2    3.533333                    4                     1                     1
3   -0.600000                   -4                     0                     0
4    1.466667                    4                     1                     1
5    1.733333                    4                     1                     1
6    2.533333                    4                     1                     1
7    2.466667                    4                     1                     1
8    1.266667                   -4                     1                     0
>>> (movies.predicted_ispositive ==
...   movies.sentiment_ispositive).sum() / len(movies)
0.9344648750589345
```

Среднее абсолютное значение ошибки предсказания (средняя абсолютная погрешность (mean absolute error, MAE)) составляет 2.4

Мы правильно предсказали позитивный рейтинг в 93 % случаев

Это довольно хорошее начало построения анализатора тональностей, созданное лишь несколькими строками кода (и большим количеством данных). Вам не пришлось составлять список из 7500 слов и их тональностей, как вы это делали для VADER. Вы просто подали анализатору на вход подборку текстов и меток для них. Вот она, мощь машинного обучения и NLP!

Как вы думаете, насколько хорошо он будет работать с совершенно другим набором показателей тональности, например, для отзывов о товарах, а не фильмах?

Если вы хотите создать настоящий анализатор тональностей, подобный этому, не забудьте разделить тренировочные данные (и отделить тестовый набор данных — см. приложение Г для получения дополнительной информации о разделении тестовых/тренировочных данных). Мы заставили классификатор оценивать тексты по бинарным категориям «отлично» или «ужасно», поэтому при случайном гадании

ошибка MAP была бы около 4. Таким образом, наш конвейер справился примерно в два раза эффективнее алгоритма случайного гадания:

```
>>> products = get_data('hutto_products')
>>> bags_of_words = []
>>> for text in products.text:
...     bags_of_words.append(Counter(casual_tokenize(text)))
>>> df_product_bows = pd.DataFrame.from_records(bags_of_words)
>>> df_product_bows = df_product_bows.fillna(0).astype(int)
>>> df_all_bows = df_bows.append(df_product_bows)
>>> df_all_bows.columns
Index(['!', '"', '#', '#38', '$', '%', '&', ''', '(', '(8',
       ...
       'zoomed', 'zooming', 'zooms', 'zx', 'zzzzzzzzz', '~', '¼', 'élan',
       '–', '''],
      dtype='object', length=23302)
>>> df_product_bows = df_all_bows.iloc[len(movies):][df_bows.columns]
>>> df_product_bows.shape
(3546, 20756)
>>> df_bows.shape
(10605, 20756)
>>> products[ispos] =
 (products.sentiment > 0).astype(int)
>>> products['predicted_ispositive'] =
 nb.predict(df_product_bows.values).astype(int)
>>> products.head()
id  sentiment                                          text     ispos  pred
0  1_1       -0.90  troubleshooting ad-2500 and ad-2600 ...      0     0
1  1_2       -0.15  repost from january 13, 2004 with a ...      0     0
2  1_3       -0.20  does your apex dvd player only play ...      0     0
3  1_4       -0.10  or does it play audio and video but ...      0     0
4  1_5       -0.50  before you try to return the player ...      0     0
>>> (products.pred == products.ispos).sum() / len(products)
0.5572476029328821
```

В наших новых коллекциях слов присутствуют токены, которых не было в исходных мультимножествах DataFrame (сейчас 23 302 столбца вместо 20 756)

Это оригинальные мультимножества слов для фильмов

Столбцы (токены) нашего нового DataFrame мультимножеств слов должны быть такими же и быть расположенными в том же порядке, что и в исходном, использовавшемся для обучения нашей наивной байесовской модели

Наша наивная байесовская модель плохо предсказывает позитивность отзыва о товаре («отлично»). Одной из причин такой низкой эффективности является наличие в нашем словаре из текстов о товарах из функции `casual_tokenize` 2546 токенов, которых не было в отзывах о фильмах. Это примерно 10 % токенов, полученных при исходной токенизации отзывов о фильмах, что означает, что все эти слова будут без весов или показателей в нашей наивной байесовской модели. С отрицательными тональностями эта наивная байесовская модель также справляется хуже, чем VADER. Необходимо включить в наш токенизатор *n*-граммы — они бы связывали слова отрицания (такие как *not* или *never*) с позитивными словами, которые они уточняют.

Мы оставляем вам в качестве упражнения усовершенствование этой модели машинного обучения. И вы можете сравнивать свои достижения с VADER на каждом этапе, чтобы определить, лучше ли машинное обучение, чем подход с ручным кодированием алгоритмов для NLP.

Резюме

- Мы реализовали метод токенизации и задали настройки токенизатора для нашего приложения.
- Токенизация *n*-грамм помогает в сохранении хотя бы части информации о *порядке слов* исходного документа.
- Нормализация и стемминг объединяют слова в группы, что повышает «чувствительность» поисковых систем, но понижает точность.
- Лемматизация и настроенные под конкретную задачу токенизаторы, такие как `casual_tokenize()`, могут повысить точность и уменьшить потери информации.
- Стоп-слова могут содержать полезную информацию, поэтому не всегда следует их отбрасывать.

3 Арифметика слов: векторы TF-IDF

В этой главе

- Подсчет слов и частотностей термов для анализа смысла.
- Предсказание вероятностей вхождений слов с помощью закона Ципфа.
- Векторные представления слов и способы их использования.
- Поиск релевантных документов из корпуса на основе обратных частотностей документов.
- Оценка сходства пар документов с помощью коэффициентов Отиаи и метрики Okapi BM25.

После того как мы собрали и посчитали слова (токены), а также объединили их по основам или леммам, настало время использовать полученную информацию для чего-нибудь интересного. Обнаружение слов хорошо подходит для простых задач вроде получения сводных показателей использования слов или поиска по ключевым словам. Но вам бы хотелось узнать, какие слова играют более важную роль для конкретного документа или даже корпуса в целом, чтобы затем искать в корпусе релевантные документы на основе этого показателя важности.

Такой подход снижает вероятность ошибочного срабатывания детектора спама на отдельном бранном слове или нескольких слегка напоминающих спам-слова

в сообщении электронной почты. При наличии широкого диапазона слов разной степени позитивности показателей (меток) можно измерить позитивность и дружелюбность какого-нибудь твита. Если знать, с какой частотой конкретные слова появляются в документе *относительно* остальных документов, можно еще больше определить уровень «позитивности» . В этой главе мы расскажем о более гибких способах измерения частоты слов и их применения в документах. Этот подход на протяжении десятилетий был основным при создании признаков на основе текстов на естественном языке в коммерческих поисковых системах, а также фильтрах спама.

Следующим шагом нашего путешествия в мир обработки естественного языка является превращение слов в непрерывные числовые величины, а не просто в целые числа, отражающие количества слов, или бинарные векторы, указывающие наличие/отсутствие конкретных слов. Над векторными представлениями слов в непрерывном пространстве можно производить более интересные математические операции. Наша цель — найти числовое представление, которое отражало бы важность или информационное содержание представляемых слов. Подождите до главы 4, и вы узнаете, как превратить это информационное наполнение в числа, отражающие *смысл* слов.

В данной главе мы рассмотрим три набирающих популярность способа представления слов и их значения.

- *Мультимножества слов* — векторы количеств, или частотностей, слов.
- *Мультимножества n-грамм* — векторы количеств пар слов (биграмм), троек слов (триграмм) и т. д.
- *Векторы TF-IDF* — показатели слов, наилучшим образом отражающие степень их важности.

ВАЖНО

> TF-IDF расшифровывается как *«частотность терма умножить на обратную частотность документа»* (term frequency inverse document frequency). Частотность слова — количество вхождений слова в документ, о котором мы говорили в предыдущих главах. Обратная частотность документа означает, что это количество для конкретного слова делится на число документов, в которых оно встречается.

Каждый из этих методов может применяться как отдельно, так и в качестве части конвейера NLP. Все описанные выше модели являются статистическими в том смысле, что они основаны на *частотностях* слов. Позже мы рассмотрим различные способы заглянуть глубже в связи между словами, закономерности их использования и нелинейности.

Все описанные выше «поверхностные» алгоритмы NLP обладают большими возможностями и применяются на практике для выполнения многих задач, таких как фильтрация спама и анализ тональностей.

3.1. Мультимножество слов

В предыдущей главе мы создали вашу первую модель векторного пространства текста. Для этого мы создали унитарные представления всех слов, а затем объединили все эти векторы с помощью бинарного OR (или усеченной версии `sum`) в векторное представление всего текста. Получившийся в результате бинарный вектор мультимножества слов, будучи загруженным в структуру данных вроде DataFrame библиотеки Pandas, оказывается отличным индексом для поиска документов.

Далее мы рассмотрели еще более удобное векторное представление с подсчетом количества вхождений (частотности) слова в тексте. На первый взгляд, чем чаще встречается слово в тексте, тем больший вклад вносит в его смысл. Документ, в котором часто упоминаются «крылья» и «руль», с большей вероятностью связан с самолетами или воздушными путешествиями, чем документ со словами «кошки» и «гравитация». Если некоторые слова были классифицированы как отражающие положительные эмоции — вроде *good*, *best*, *joy* и *fantastic*, то тональность содержащего их документа, скорее всего, положительная. Впрочем, понятно, что основанный на таких простых правилах алгоритм может легко ошибиться.

Рассмотрим пример, в котором подсчет частотностей слов оказывается полезным:

```
>>> from nltk.tokenize import TreebankWordTokenizer
>>> sentence = """The faster Harry got to the store, the faster Harry,
...     the faster, would get home."""
>>> tokenizer = TreebankWordTokenizer()
>>> tokens = tokenizer.tokenize(sentence.lower())
>>> tokens
['the',
 'faster',
 'harry',
 'got',
 'to',
 'the',
 'store',
 ',',
 'the',
 'faster',
 'harry',
 ',',
 'the',
 'faster',
 ',',
 'would',
 'get',
 'home',
'.']
```

Этот простой список нужен, чтобы выделить из документа уникальные слова и найти их количества. Словарь Python отлично подходит для этой цели. Поскольку

необходимо хранить еще и количества вхождений слов, можно воспользоваться типом `Counter`, как мы делали в предыдущих главах:

```
>>> from collections import Counter
>>> bag_of_words = Counter(tokens)
>>> bag_of_words
Counter({'the': 4,
         'faster': 3,
         'harry': 2,
         'got': 1,
         'to': 1,
         'store': 1,
         ',': 3,
         'would': 1,
         'get': 1,
         'home': 1,
         '.': 1})
```

Как и в любом хорошем словаре Python, порядок ключей перемешивается. Новый порядок оптимизирован для хранения, обновления и поиска, а не для согласованного отображения. Информация, заключенная в порядке слов исходного высказывания, отбрасывается.

ПРИМЕЧАНИЕ

> Объект collections.Counter — неупорядоченная коллекция, также называемая мультимножеством. В зависимости от платформы и версии Python счетчик может отображаться в, казалось бы, логичном порядке, например в лексикографическом или в порядке токенов в высказывании. Но рассчитывать на какой-либо определенный порядок токенов (ключей) как в объекте Counter, так и в обычном классе dict языка Python нельзя.

Для коротких документов, как этот, даже в неупорядоченном мультимножестве слов содержится немало информации об исходном подтексте предложения. Причем информации в мультимножестве достаточно для весьма серьезных задач, таких как обнаружение спама, анализ тональности (позитивность, счастье и т. д.), и даже для выявления таких тонких эмоций, как сарказм. Да, это просто набор слов, но наполненный смыслом и информацией. Итак, проранжируем эти слова — отсортируем их в более удобном для понимания порядке. У объекта `Counter` есть весьма удобный метод `most_common`, предназначенный именно для этой цели:

```
>>> bag_of_words.most_common(4)
[('the', 4), (',', 3), ('faster', 3), ('harry', 2)]
```

По умолчанию функция most_common() сортирует все токены в порядке убывания частотности, но мы ограничились только первыми четырьмя из них

Количество вхождений слова в заданном документе называется *частотностью терма* (term frequency, TF). Иногда можно встретить нормализованные путем деления на общее число термов в документе количества слов[1].

Итак, наши четыре чаще всего встречающихся терма (токена): *the*, *«,»*, *harry* и *faster*. Впрочем, *the* и запятая не несут много информации об основной мысли этого документа. Скорее всего, эти неинформативные токены будут встречаться вам еще много раз. Поэтому в данном случае их лучше игнорировать, как и целый список стандартных английских стоп-слов и знаков препинания. Конечно, это не всегда имеет смысл делать, но такое решение позволит пока упростить наш пример. Это сокращает список самых употребляемых токенов в векторе TF (мультимножестве слов) до *harry* и *faster*.

Подсчитаем частотность терма `harry` из описанного выше объекта `Counter` (`bag_of_words`):

```
>>> times_harry_appears = bag_of_words['harry']
>>> num_unique_words = len(bag_of_words)
>>> tf = times_harry_appears / num_unique_words
>>> round(tf, 4)
0.1818
```

Количество уникальных токенов в исходном документе

Притормозим и обратим внимание на нормализованную частоту термов — фразу (и сопутствующие ей вычисления), которая встречается на протяжении всей книги. Под этим термином понимается количество слов относительно длины документа. Зачем вообще делить на длину документа? Представим, что *dog* встречается три раза в документе A и 100 раз в документе B. Это слово явно играет куда более важную роль во втором случае. Но если документ A — письмо ветеринару из 30 слов, а B — «Война и мир» Л. Н. Толстого (приблизительно 580 тыс. слов!)? Тогда наши изначальные выводы меняются на диаметрально противоположные. Следующие уравнения учитывают длину документа:

$$TF(\text{dog}, document_A) = 3 / 30 = 0{,}1.$$

$$TF(\text{dog}, document_B) = 100 / 580\,000 = 0{,}00017.$$

Теперь у вас есть что-то, что позволяет понять разницу между двумя документами, их связь с *dog* и друг с другом. Таким образом, вместо чистых количеств слов можно использовать для описания документов из корпуса нормализованные частотности термов. Аналогичным образом можно вычислить этот показатель для каждого из слов и узнать относительную важность данного терма для каждого документа. Наш главный герой, Гарри, и его жажда скорости, несомненно, являются центром описанной истории. Мы добились значительных успехов в превращении текста в числа, намного выходящих за пределы фиксации наличия/отсутствия заданного

[1] Впрочем, нормализованная частотность, по существу, представляет собой вероятность, поэтому, возможно, не стоит называть ее частотностью.

слова в тексте. Очевидно, что это довольно «притянутый за уши» пример, но скоро мы увидим, насколько значимые результаты могут быть получены при таком подходе. Рассмотрим больший фрагмент текста. Возьмем несколько первых абзацев из статьи «Википедии» о воздушных змеях *(kites)*:

> *A kite is traditionally a tethered heavier-than-air craft with wing surfaces that react against the air to create lift and drag. A kite consists of wings, tethers, and anchors. Kites often have a bridle to guide the face of the kite at the correct angle so the wind can lift it. A kite's wing also may be so designed so a bridle is not needed; when kiting a sailplane for launch, the tether meets the wing at a single point. A kite may have fixed or moving anchors. Untraditionally in technical kiting, a kite consists of tether-set-coupled wing sets; even in technical kiting, though, a wing in the system is still often called the kite.*
>
> *The lift that sustains the kite in flight is generated when air flows around the kite's surface, producing low pressure above and high pressure below the wings. The interaction with the wind also generates horizontal drag along the direction of the wind. The resultant force vector from the lift and drag force components is opposed by the tension of one or more of the lines or tethers to which the kite is attached. The anchor point of the kite line may be static or moving (such as the towing of a kite by a running person, boat, free-falling anchors as in paragliders and fugitive parakites or vehicle).*
>
> *The same principles of fluid flow apply in liquids and kites are also used under water.*
>
> *A hybrid tethered craft comprising both a lighter-than-air balloon as well as a kite lifting surface is called a kytoon.*
>
> *Kites have a long and varied history and many different types are flown individually and at festivals worldwide. Kites may be flown for recreation, art or other practical uses. Sport kites can be flown in aerial ballet, sometimes as part of a competition. Power kites are multi-line steerable kites designed to generate large forces which can be used to power activities such as kite surfing, kite landboarding, kite fishing, kite buggying and a new trend snow kiting. Even Man-lifting kites have been made.*
>
> *«Википедия»*

Присвоим этот текст переменной:

```
>>> from collections import Counter
>>> from nltk.tokenize import TreebankWordTokenizer
>>> tokenizer = TreebankWordTokenizer()
>>> from nlpia.data.loaders import kite_text
>>> tokens = tokenizer.tokenize(kite_text.lower())
>>> token_counts = Counter(tokens)
>>> token_counts
Counter({'the': 26, 'a': 20, 'kite': 16, ',': 15, ...})
```

Kite_text = A kite is traditionally …, все как в примере выше

ПРИМЕЧАНИЕ

> TreebankWordTokenizer возвращает kite. (с точкой) в качестве токена. Treebank Tokenizer предполагает, что документ уже был сегментирован на отдельные предложения, поэтому он игнорирует пунктуацию только в конце строки. Сегментация предложений — очень сложный вопрос, и мы будем обсуждать ее только в главе 11. Тем не менее синтаксический анализатор spaCy работает быстрее и точнее, чем Treebank, ввиду того, что он выполняет сегментацию и токенизацию предложений (наряду со многими другими действиями)[1] за один проход. Так что лучше применять для реальных приложений spaCy, а не использовавшиеся для простых примеров выше компоненты NLTK.

Ладно, вернемся к примеру. Вы уже заметили огромное количество стоп-слов? Наверняка эта статья из «Википедии» не об артиклях *the*, *a*, союзе *and* и т. д. Их пока отбросим:

```
>>> import nltk
>>> nltk.download('stopwords', quiet=True)
True
>>> stopwords = nltk.corpus.stopwords.words('english')
>>> tokens = [x for x in tokens if x not in stopwords]
>>> kite_counts = Counter(tokens)
>>> kite_counts
Counter({'kite': 16,
         'traditionally': 1,
         'tethered': 2,
         'heavier-than-air': 1,
         'craft': 2,
         'wing': 5,
         'surfaces': 1,
         'react': 1,
         'air': 2,
         ...,
         'made': 1})}
```

Определенные выводы о содержании документа можно сделать, исходя даже из одной информации о частотностях слов в нем. Слова *kite(s)*, *wing* и *lift* обладают большой важностью. Даже если вы не имеете понятия о содержании этого документа и просто наткнулись на него в своей обширной базе данных (масштабов Google), то смогли бы «программным образом» сделать вывод, что он как-то связан с полетами или подъемами в воздух или же с воздушными змеями.

Если рассматривать несколько документов в корпусе, то все становится немного интереснее. Набор документов может, например, быть *целиком* посвящен воздушным змеям. Логично предположить, что во всех документах из него упоминается веревка *(string)* и ветер *(wind)*, а частотность термов `TF("string")` и `TF("wind")` будет высокой во всех документах. Теперь рассмотрим способ более изящного представления этих чисел в математических целях.

---

[1] См. веб-страницу spaCy 101: Everything you need to know по адресу spacy.io/usage/spacy-101#annotations-token.

3.2. Векторизация

Мы уже сталкивались с простейшими преобразованиями текста в числа. Но мы просто сохранили числа в словаре, сделав первый шаг из мира текста в мир математики. Теперь мы отправимся дальше по этому пути. Вместо описания документа в терминах частотного словаря мы сформируем из этих количеств слов вектор. В Python он будет представлять собой список, но в общем случае он может быть упорядоченной коллекцией или массивом. Это можно проделать быстро с помощью следующего кода:

```
>>> document_vector = []
>>> doc_length = len(tokens)
>>> for key, value in kite_counts.most_common():
...     document_vector.append(value / doc_length)
>>> document_vector
[0.07207207207207207,
 0.06756756756756757,
 0.036036036036036036,
 ...,
 0.004504504504504505]
```

Над этим списком (вектором) можно уже непосредственно производить математические операции.

ПРИМЕЧАНИЕ

Существует много способов ускорить обработку подобных структур данных[1].

Применять математику лишь к одному элементу не очень интересно. Одного вектора для одного документа недостаточно. Лучше взять еще парочку документов и создать для них векторы. Но содержащиеся во всех векторах значения должны относиться к общей точке отсчета. Для проведения с ними вычислений они должны отражать точку в пространстве относительно единого начала координат. Вектору нужны общая точка отсчета и одинаковые масштабы («единицы измерения») в каждом из их измерений. Первый этап данного процесса — нормализация количеств слов путем подсчета нормализованных частотностей термов вместо простых количеств вхождений слов в документе (как мы делали в предыдущем разделе). Второй — приведение всех векторов к единой длине (размеру).

Кроме того, значение каждого элемента вектора должно отражать одно и то же слово в векторах для всех документов. Но наше письмо ветеринару вряд ли будет включать столько же слов, как «Война и мир» (а может, и будет, кто знает?). Не переживайте, если некоторые из векторов будут содержать нулевые значения на некоторых позициях. Находим все уникальные слова в каждом из наших двух документов, а затем — все уникальные слова в объединении этих двух множеств. Такие наборы слов часто называются *лексиконом*, что отражает уже встречавшееся нам в предыдущих главах понятие, только в терминах конкретного корпуса. Посмотрим, как это работает с документами немного меньшего размера, нежели *«Война*

[1] См. веб-страницу NumPy по адресу www.numpy.org.

и мир». Вернемся к нашему Гарри. У вас уже есть один «документ» о нем, увеличим наш корпус еще на парочку:

```
>>> docs = ["The faster Harry got to the store, the faster and faster Harry
⇒ would get home."]
>>> docs.append("Harry is hairy and faster than Jill.")
>>> docs.append("Jill is not as hairy as Harry.")
```

ПРИМЕЧАНИЕ

Для удобства, чтобы не набирать эти тексты вручную, вы можете импортировать их из пакета nlpia: from nlpia.data.loaders import harry_docs as docs.

Взглянем на наш лексикон для корпуса из трех документов:

```
>>> doc_tokens = []
>>> for doc in docs:
...     doc_tokens += [sorted(tokenizer.tokenize(doc.lower()))]
>>> len(doc_tokens[0])
17
>>> all_doc_tokens = sum(doc_tokens, [])
>>> len(all_doc_tokens)
33
>>> lexicon = sorted(set(all_doc_tokens))
>>> len(lexicon)
18
>>> lexicon
[',',
 '.',
 'and',
 'as',
 'faster',
 'get',
 'got',
 'hairy',
 'harry',
 'home',
 'is',
 'jill',
 'not',
 'store',
 'than',
 'the',
 'to',
 'would']
```

Каждый из трех векторов документов должен содержать 18 значений, даже если в соответствующем документе содержатся не все 18 слов из лексикона. Каждому токену выделяется место в векторах в соответствии с его позицией в лексиконе. Некоторые из этих количеств токенов будут равны нулю, что нам, собственно, и нужно:

```
>>> from collections import OrderedDict
>>> zero_vector = OrderedDict((token, 0) for token in lexicon)
>>> zero_vector
```

```
OrderedDict([(',', 0),
             ('.', 0),
             ('and', 0),
             ('as', 0),
             ('faster', 0),
             ('get', 0),
             ('got', 0),
             ('hairy', 0),
             ('harry', 0),
             ('home', 0),
             ('is', 0),
             ('jill', 0),
             ('not', 0),
             ('store', 0),
             ('than', 0),
             ('the', 0),
             ('to', 0),
             ('would', 0)])
```

Далее мы копируем базовый вектор, обновляем его значения для каждого документа и сохраняем в массиве:

Функция copy.copy() создает независимую копию, отдельный экземпляр нашего нулевого вектора, а не переиспользует ссылку (указатель) на местоположение в памяти исходного объекта. В противном случае мы бы перезаписывали один и тот же объект zero_vector новыми значениями на каждой итерации цикла и не начинали бы с «чистой доски» на каждом проходе

```
>>> import copy
>>> doc_vectors = []
>>> for doc in docs:
...     vec = copy.copy(zero_vector)
...     tokens = tokenizer.tokenize(doc.lower())
...     token_counts = Counter(tokens)
...     for key, value in token_counts.items():
...         vec[key] = value / len(lexicon)
...     doc_vectors.append(vec)
```

Итак, у нас есть три вектора. По одному на каждый документ. «И что дальше? Что мы можем с ними сделать?» — спросите вы. С векторами, содержащими количества слов, можно делать множество интересных вещей, как и с любыми другими, так что сначала узнаем больше о векторах и векторных пространствах[1].

3.2.1. Векторные пространства

Векторы — краеугольный камень линейной алгебры (векторной алгебры). По своей сути это упорядоченные последовательности чисел (координат) в векторном пространстве. Они описывают место (положение) в данном пространстве. Могут использоваться и для задания конкретного направления и модуля (расстояния) в этом пространстве. *Пространство* — это совокупность всех возможных векторов,

[1] Если вы хотите узнать больше о линейной алгебре и векторах, см. приложение B.

которые могут в нем встречаться. Таким образом, вектор с двумя значениями будет располагаться в двумерном векторном пространстве, вектор с тремя значениями в трехмерном векторном пространстве и т. д.

Кусок миллиметровки или сетка пикселов на изображении — это допустимые двумерные векторные пространства. Нетрудно заметить, как важен порядок этих координат. Если поменять местами координаты x и y для местоположений на миллиметровке без соответствующей корректировки операций над векторами, то все решения задач линейной алгебры окажутся зеркально отраженными. Миллиметровка и сетка пикселов изображения — примеры евклидовых пространств, потому что оси координат x и y перпендикулярны друг другу. Обсуждаемые в этой главе векторные пространства являются евклидовыми пространствами[1].

А как насчет широты и долготы на карте или глобусе? Карта и глобус — определенно двумерные векторные пространства, ввиду того что представляют собой упорядоченный список двух чисел: широты и долготы. Каждая из пар широта — долгота описывает точку на приближенно сферической бугристой поверхности Земли. Вдобавок угол между осями координат широты и долготы не равен в точности 90°, поэтому векторное пространство широты и долготы не является линейным. Это означает, что вы должны быть осторожны, когда вычисляете такие вещи, как расстояние/близость (сходство) между двумя точками, представленными парой 2D-векторов широты и долготы или векторов в любом неевклидовом пространстве. Представьте, например, вычисление расстояния между координатами широты и долготы Портленда, штат Орегон, и Нью-Йорка, штат Нью-Йорк[2].

На рис. 3.1 показан один из способов изображения 2D-векторов с координатами (5, 5), (3, 2) и (–1, 1). «Вершина» вектора (острый конец стрелки) указывает на местоположение в векторном пространстве. Поэтому вершины векторов на этом графике находятся в точках, соответствующих указанным трем парам координат. Хвост вектора («задняя» часть стрелки) всегда в начале координат (0, 0).

А как насчет трехмерных векторных пространств? Положения и скорости в трехмерном физическом мире, в котором мы с вами живем, можно отразить с помощью координат x, y и z трехмерного вектора. Или рассмотреть криволинейное пространство из троек «широта — долгота — высота», описывающих местоположения объектов вблизи поверхности Земли.

К счастью, мы не ограничены обычным 3D-пространством. Наше пространство может быть 5-, 10- или 5000-мерным! Линейная алгебра во всех случаях работает одинаково. Но по мере роста размерности пространства могут потребоваться

---

[1] Авторы используют термин rectilinear space в качестве синонима евклидова пространства. В русскоязычной литературе понятия прямолинейного пространства не выделяют, а rectilinear в серьезных математических словарях переводят как «линейный». Но линейное (векторное) пространство — более общее понятие, чем евклидово, и подобный перевод был бы просто некорректен. — *Примеч. пер.*

[2] Для обеспечения правильности вычислений понадобится что-то вроде пакета GeoPy (geopy.readthedocs.io).

большие вычислительные мощности. Возможны также определенные проблемы проклятия размерности, но этот вопрос мы отложим до последней главы[1].

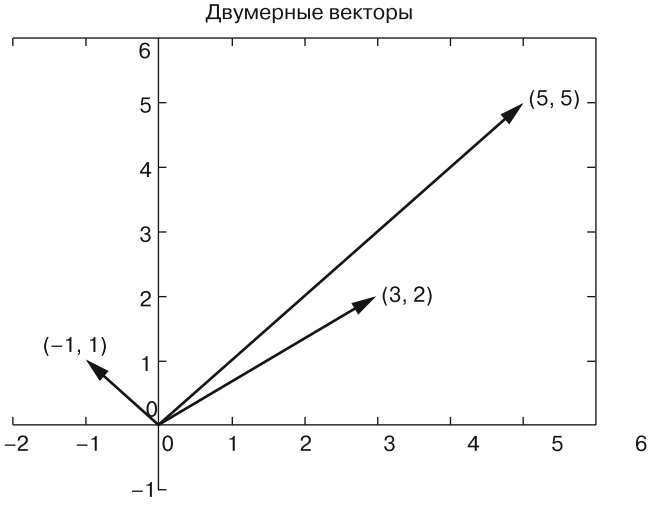


Рис. 3.1. Двумерные векторы

Для векторного пространства документов на естественном языке размерность векторного пространства равна числу различных слов во всем корпусе. Для TF (и далее и TF-IDF) мы иногда будем использовать для обозначения этой размерности прописную букву K. Это число отдельных слов также является размером словаря корпуса, поэтому в теоретических статьях его обычно обозначают $|V|$. Любой документ в K-мерном векторном пространстве можно описать с помощью K-мерного вектора. В случае нашего корпуса из трех документов о Гарри и Джилл K = 18. Поскольку наглядно представить себе пространство размерности больше трех непросто, проигнорируем большую их часть и рассмотрим пока что только два, чтобы изобразить его векторы на лежащей перед вами плоской странице. На рис. 3.2 K уменьшено до двух для 2D-представления 18-мерного векторного пространства Гарри и Джилл.

K-мерные векторы аналогичны данным, просто это сложно представить наглядно. Теперь у нас есть векторные представления всех документов в одном пространстве,

[1] Проклятие размерности состоит в экспоненциальном удалении векторов друг от друга в смысле евклидова расстояния по мере увеличения размерности. Множество простых операций становятся нереализуемыми на практике при векторах размерностью выше 10 или 20. Примерами таких операций может быть сортировка большого списка векторов по их удаленности от вектора запроса («эталонного» вектора) (поиск приблизительного ближайшего соседа). Чтобы глубже вникнуть, ознакомьтесь со статьей по адресу ru.wikipedia.org/wiki/Проклятие_размерности, поэкспериментируйте с пакетом annoy языка Python (github.com/spotify/annoy) или поищите в Google Scholar информацию по запросу high dimensional approximate nearest neighbors (scholar.google.com/scholar?q=high+dimensional+approximate+nearest+neighbor).

а значит, появилась возможность их сравнить. Измерить евклидово расстояние между векторами можно путем их вычитания и вычисления длины расстояния между ними, называемого расстоянием в смысле L^2-нормы (метрики L^2). Это расстояние по прямой от места, задаваемого острием (вершиной) одного вектора, до местоположения, соответствующего острию другого вектора. В приложении C, посвященном линейной алгебре, рассказывается, почему этот метод не подходит для векторов количества слов (частотностей термов).

Два вектора считаются подобными, если они имеют одинаковое направление. Их модули (длины) также могут быть равны, означая, что векторы количеств слов (частотностей термов) относятся к документам примерно одинаковой длины. Но важна ли длина документов при оценке подобия векторных представлений слов в документах? Скорее всего, нет.

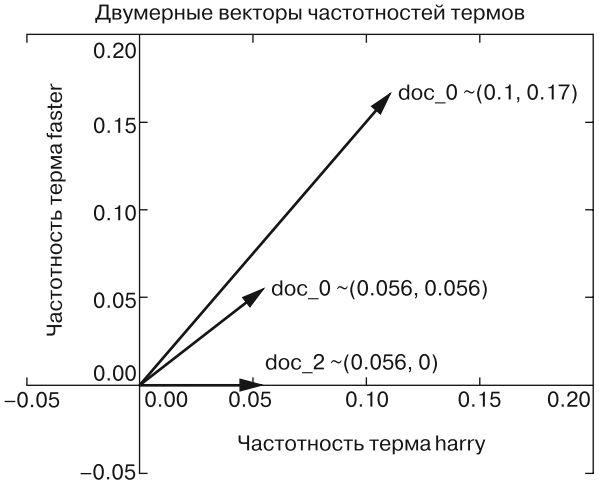


Рис. 3.2. Двумерные векторы частотностей термов

Желательно, чтобы наша оценка уровня подобия документов учитывала одни и те же слова аналогичное число раз в одинаковых пропорциях. Благодаря столь точным оценкам можно быть уверенными, что отражаемые ими документы, вероятно, посвящены близким вещам.

Коэффициент Отиаи (коэффициент косинусного подобия) представляет собой просто косинус угла между двумя векторами (theta), показанными на рис. 3.3. Его можно вычислить с помощью евклидова скалярного произведения по формуле:

$$A \cdot B = |A|\,|B| \cdot \cos \Theta.$$

Коэффициент Отиаи удобно вычислять, поскольку скалярное произведение не требует вычисления каких-либо тригонометрических функций. Кроме того, диапазон принимаемых коэффициентом Отиаи значений удобен для большинства задач машинного обучения: от –1 до +1.

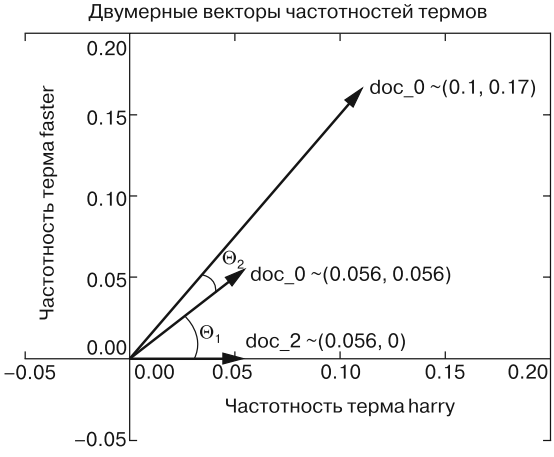


Рис. 3.3. Двумерные theta

На языке Python эта формула выглядит следующим образом:

```
a.dot(b) == np.linalg.norm(a) * np.linalg.norm(b) / np.cos(theta)
```

Решая это уравнение относительно `cos(theta)`, получаем коэффициент Отиаи:

$$\cos\Theta = \frac{A \cdot B}{|A||B|}.$$

Можно сделать это на чистом Python, не используя пакет `numpy`, как в листинге 3.1.

Листинг 3.1. Вычисление косинусного сходства на языке Python

```
>>> import math
>>> def cosine_sim(vec1, vec2):
...     """ Let's convert our dictionaries to lists for easier matching."""
...     vec1 = [val for val in vec1.values()]
...     vec2 = [val for val in vec2.values()]
...
...     dot_prod = 0
...     for i, v in enumerate(vec1):
...         dot_prod += v * vec2[i]
...
...     mag_1 = math.sqrt(sum([x**2 for x in vec1]))
...     mag_2 = math.sqrt(sum([x**2 for x in vec2]))
...
...     return dot_prod / (mag_1 * mag_2)
```

Итак, нужно вычислить скалярное произведение двух векторов — умножить элементы векторов попарно, а затем просуммировать полученные результаты. После этого поделить результат на норму (модуль, то есть длину) каждого вектора. Норма вектора равна евклидову расстоянию от его головы до хвоста — квадратному корню суммы квадратов его элементов. Это *нормализованное скалярное произведение*, как и значение косинуса, располагается между –1 и 1. Оно также является косинусом угла между этими двумя векторами. Данное значение равно доле более длинного

вектора, охваченной перпендикулярной проекцией на него более короткого вектора. Из этого ясно, насколько наши два вектора направлены в одну сторону.

Равный 1 косинусный коэффициент отражает одинаковые нормализованные векторы, указывающие в аналогичном направлении по всем измерениям. Длины (модули) этих векторов могут быть различными, но они указывают в одном и том же направлении. Помните, что мы поделили скалярное произведение на норму каждого вектора, причем это можно делать до или после вычисления скалярного произведения. Таким образом, при его вычислении векторы нормализованы, длины обоих равны 1. Мы можем сделать вывод: чем ближе значение косинусного сходства к 1, тем меньше угол между векторами. В NLP документы, косинусный коэффициент векторов которых близок к 1, содержат похожие слова в одинаковой пропорции. Таким образом, документы, векторы которых близки друг к другу, скорее всего, посвящены одному и тому же.

При косинусном коэффициенте, равном 0, векторы не имеют общих компонентов. Они находятся под углом 90° относительно друг друга по всем измерениям. Для векторов TF NLP такая ситуация возникает только в случае, если в двух документах нет общих слов. Поскольку в них используются совершенно разные слова, они обсуждают абсолютно разные вещи. Это не обязательно означает различный их смысл или тематику, просто в них применяются разные слова.

При значении косинусного коэффициента, равном –1, векторы полностью противоположные. Они указывают в противоположных направлениях. Такого не может произойти ни с простыми векторами количеств слов (частотностей термов), ни даже с нормализованными TF-векторами (которые мы обсудим далее). Векторы количеств слов (частотностей термов) не могут быть отрицательными. Таким образом, последние всегда будут находиться в одном квадранте векторного пространства. Ни один из векторов частотностей термов не может «пробраться» в какой-либо из квадрантов, расположенных «за спиной» других векторов. Ни один компонент какого-либо из TF-векторов не может быть противоположным какому-либо компоненту другого вектора ввиду того, что частотность слова просто не может быть отрицательной.

В этой главе вы не увидите никаких отрицательных значений косинусного сходства. Однако в следующей мы разработаем концепцию слов и тем, противоположных друг другу. В таком случае вы увидите документы, слова и темы, значение косинусного сходства которых меньше 0 или даже равно –1.

ПРОТИВОПОЛОЖНОСТИ ПРИТЯГИВАЮТСЯ

> Из нашего способа вычисления косинусного коэффициента следует интересное свойство. Если косинусные коэффициенты двух векторов или документов равны –1 (то есть они являются противоположностями) относительно третьего вектора, они должны быть подобны друг другу — в точности идентичны. Однако документы, которые представляют эти векторы, могут не совпадать. В них не только может быть разный порядок слов, но и один может быть намного длиннее другого, если в нем используются те же слова в той же пропорции.

Позже вы узнаете о векторах, которые намного точнее моделируют документ, а пока вы получили хорошее представление о необходимых инструментах.

3.3. Закон Ципфа

Приступим к нашей основной теме — социологии. Ладно, на самом деле не совсем, но вам предстоит краткий экскурс в мир подсчетов людей и слов, в ходе которого вы узнаете, вероятно, универсальное правило, обуславливающее подсчеты большинства вещей. Оказывается, что язык, как и большинство вещей, связанных с живыми организмами, изобилует закономерностями.

В начале XX века французский стенографист Жан-Батист Эсту (Jean-Baptiste Estoup) обнаружил закономерность частотностей слов, которые он старательно подсчитывал вручную во многих документах (хвала небесам, что у нас есть компьютеры и Python). В 1930-х годах американский лингвист Джордж Кингсли Ципф попытался формализовать наблюдения Эсту, и в итоге эти закономерности стали носить имя Ципфа.

> Закон Ципфа гласит, что в некотором корпусе высказываний на естественном языке частотность любого слова обратно пропорциональна его позиции в таблице частотностей.
>
> *«Википедия»*

Точнее говоря, фраза *«обратная пропорциональность»* относится к ситуации, когда частотность элемента ранжированного списка явным образом связана с его местом в списке. Например, первый элемент ранжированного списка может встречаться в два раза чаще, чем второй, и в три раза чаще, чем третий. Для любого корпуса или документа можно быстро построить график частоты использования слов относительно их позиции в таблице частотностей. Любые аномальные значения, выделяющиеся на прямой линии графика с двойной логарифмической шкалой, заслуживают исследования.

В качестве примера того, как далеко закон Ципфа выходит за пределы мира слов, рис. 3.4 демонстрирует взаимосвязь между населением городов США и их местом в таблице численности населения. Оказывается, что закон Ципфа применим к частотностям множества вещей. Природа полна систем, в которых наблюдаются экспоненциальный рост и такие сетевые эффекты, как динамика численности населения, рост объемов производства и распределение ресурсов[1]. Интересно, что нечто столь простое, как закон Ципфа, справедливо для столь широкого диапазона природных и техногенных явлений. Нобелевский лауреат Пол Кругман (Paul Krugman), говоря об экономических моделях и законе Ципфа, сформулировал это так:

> *«На экономические теории обычно жалуются, что наши модели излишне упрощены и их отражение сложной беспорядочной реальности слишком "приглаженное". [Относительно закона Ципфа] верно обратное: модели сложны и беспорядочны, а реальность поразительно ясна и проста».*

[1] См. статью There is More than a Power Law in Zipf: www.nature.com/articles/srep00812.

На рис. 3.4 представлена обновленная версия графика Кругмана для населения городов[1].

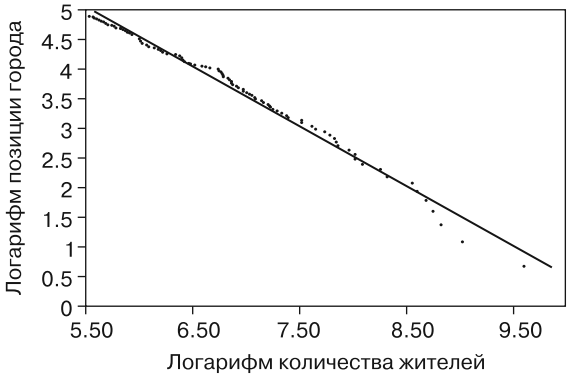


Рис. 3.4. Распределение городского населения

Для слов справедлива та же закономерность, что и для городов и социальных сетей. Для начала загрузим Brown Corpus из пакета NLTK.

> Brown Corpus был первым электронным корпусом английского языка, состоящим из миллиона слов. Он был создан в 1961 году в Университете Брауна. Этот корпус содержит текст из 500 источников, классифицированных по жанрам, таким как новости, передовые статьи и т. д.[2]
>
> *Документация NLTK*

```
>>> nltk.download('brown')
>>> from nltk.corpus import brown
>>> brown.words()[:10]
 ['The',
  'Fulton',
  'County',
  'Grand',
  'Jury',
  'said',
  'Friday',
  'an',
  'investigation',
  'of']
```

Размер Brown Corpus составляет около 3 Мбайт

words() — встроенный метод объекта корпуса NLTK, возвращающий токенизированный корпус в виде последовательности строковых значений

[1] Данные о популяции загружены из «Википедии» с помощью пакета Pandas. См. nlpia.book.examples на GitHub по адресу github.com/totalgood/nlpia/blob/master/src/nlpia/book/examples/ch03_zipf.py.

[2] Полный список жанров можно найти по адресу http://clu.uni.no/icame/manuals/BROWN/INDEX.HTM#t2.

```
>>> brown.tagged_words()[:5]
 [('The', 'AT'),
  ('Fulton', 'NP-TL'),
  ('County', 'NN-TL'),
  ('Grand', 'JJ-TL'),
  ('Jury', 'NN-TL')]
>>> len(brown.words())
1161192
```

О частеречной разметке мы говорили в главе 2

Таким образом, учитывая наличие более чем 1 миллиона токенов, у вас есть на что посмотреть:

```
>>> from collections import Counter
>>> puncs = set((',', '.', '--', '-', '!', '?',
...      ':', ';', '``', "''", '(', ')', '[', ']'))
>>> word_list = (x.lower() for x in brown.words() if x not in puncs)
>>> token_counts = Counter(word_list)
>>> token_counts.most_common(20)
[('the', 69971),
 ('of', 36412),
 ('and', 28853),
 ('to', 26158),
 ('a', 23195),
 ('in', 21337),
 ('that', 10594),
 ('is', 10109),
 ('was', 9815),
 ('he', 9548),
 ('for', 9489),
 ('it', 8760),
 ('with', 7289),
 ('as', 7253),
 ('his', 6996),
 ('on', 6741),
 ('be', 6377),
 ('at', 5372),
 ('by', 5306),
 ('i', 5164)]
```

Сразу заметно, что частотности слов в корпусе Brown соответствуют логарифмическим отношениям, предсказанным Ципфом. *The* (на первом месте по частотности терма) встречается примерно в два раза чаще, чем *of* (на втором месте по частотности терма), и примерно в три раза чаще, чем *and* (на третьем месте по частотности терма). Если не верите, воспользуйтесь примером кода (github.com/totalgood/nlpia/blob/master/src/nlpia/book/examples/ch03_zipf.py) из пакета `nlpia` и убедитесь сами.

Если упорядочить слова корпуса по количеству вхождений и расположить их в порядке убывания, то при достаточно большой выборке первое слово в этом ранжированном списке встречается в корпусе в два раза чаще, чем второе. И в четыре раза чаще четвертого слова в списке. Таким образом, при достаточно большом корпусе можно на основе этого разбиения предсказать статистическую вероятность вхождения данного слова в любом из документов корпуса.

3.4. Тематическое моделирование

Вернемся к нашим векторам документов. Подсчет слов полезен, но чистое количество слов, даже нормализованное по длине документа, мало что говорит о важности определенного слова для смысла документа *по сравнению с* остальными документами корпуса. Чтобы начать описывать документы в корпусе, хорошо бы выяснить эту информацию. Допустим, что у нас есть корпус, включающий в себя все когда-либо написанные книги о воздушных змеях. Слово *kite* почти наверняка встречается много раз в каждой из данных книг (документов), но это не дает никакой новой информации и не помогает различить эти документы. В то время как слова вроде *construction* («конструкция», «конструирование») или *aerodynamics* («аэродинамика»), наверное, не столь распространены во всем корпусе, именно благодаря им можно узнать больше о природе тех документов, где они встречаются чаще. Для выполнения этой задачи нам понадобится еще один инструмент.

Применить закон Ципфа при тематическом анализе нам поможет обратная частотность документа (IDF). Расширим вышеприведенный счетчик частотностей термов. Подсчитывать токены и группировать их можно двумя способами: по документам и по всему корпусу. Мы выберем первое.

Вернемся к примеру о воздушных змеях из «Википедии», но возьмем другой отрывок (раздел History) и сделаем его вторым документом в нашем корпусе.

> *Kites were invented in China, where materials ideal for kite building were readily available: silk fabric for sail material; fine, high-tensile-strength silk for flying line; and resilient bamboo for a strong, lightweight framework.*
>
> *The kite has been claimed as the invention of the 5th-century BC Chinese philosophers Mozi (also Mo Di) and Lu Ban (also Gongshu Ban). By 549 AD paper kites were certainly being flown, as it was recorded that in that year a paper kite was used as a message for a rescue mission. Ancient and medieval Chinese sources describe kites being used for measuring distances, testing the wind, lifting men, signaling, and communication for military operations. The earliest known Chinese kites were flat (not bowed) and often rectangular. Later, tailless kites incorporated a stabilizing bowline. Kites were decorated with mythological motifs and legendary figures; some were fitted with strings and whistles to make musical sounds while flying. From China, kites were introduced to Cambodia, Thailand, India, Japan, Korea and the western world.*
>
> *After its introduction into India, the kite further evolved into the fighter kite, known as the patang in India, where thousands are flown every year on festivals such as Makar Sankranti.*
>
> *Kites were known throughout Polynesia, as far as New Zealand, with the assumption being that the knowledge diffused from China along with the people.*
>
> *Anthropomorphic kites made from cloth and wood were used in religious ceremonies to send prayers to the gods. Polynesian kite traditions are used by anthropologists get an idea of early "primitive" Asian traditions that are believed to have at one time existed in Asia.*
>
> *«Википедия»*

Для начала посчитаем количество слов в каждом документе нашего корпуса: `intro_doc` и `history_doc`:

```
>>> from nlpia.data.loaders import kite_text, kite_history
>>> kite_intro = kite_text.lower()
>>> intro_tokens = tokenizer.tokenize(kite_intro)
>>> kite_history = kite_history.lower()
>>> history_tokens = tokenizer.tokenize(kite_history)
>>> intro_total = len(intro_tokens)
>>> intro_total
363
>>> history_total = len(history_tokens)
>>> history_total
297
```

Преобразуем A kite is traditionally … ? в a kite is traditionally … (понижаем регистр)

Теперь найдем частотность терма *kite* в каждом из двух имеющихся у нас токенизированных документов о воздушных змеях. Хранить найденные TF мы будем в двух словарях, по одному для каждого документа:

```
>>> intro_tf = {}
>>> history_tf = {}
>>> intro_counts = Counter(intro_tokens)
>>> intro_tf['kite'] = intro_counts['kite'] / intro_total
>>> history_counts = Counter(history_tokens)
>>> history_tf['kite'] = history_counts['kite'] / history_total
>>> 'Term Frequency of "kite" in intro is: {:.4f}'.format(intro_tf['kite'])
'Term Frequency of "kite" in intro is: 0.0441'
>>> 'Term Frequency of "kite" in history is: {:.4f}'\
...     .format(history_tf['kite'])
'Term Frequency of "kite" in history is: 0.0202'
```

Итак, получилось, что количество вхождений *kite* в первом тексте в два раза больше, чем во втором. Неужели вступительный раздел в два раза сильнее связан с воздушными змеями, чем раздел об истории их создания? Нет, не совсем. Так что копнем немного глубже. Во-первых, сопоставим эти числа с данными для какого-либо другого слова, скажем *and*:

```
>>> intro_tf['and'] = intro_counts['and'] / intro_total
>>> history_tf['and'] = history_counts['and'] / history_total
>>> print('Term Frequency of "and" in intro is: {:.4f}'\
...     .format(intro_tf['and']))
Term Frequency of "and" in intro is: 0.0275
>>> print('Term Frequency of "and" in history is: {:.4f}'\
...     .format(history_tf['and']))
Term Frequency of "and" in history is: 0.0303
```

Ура! Теперь мы знаем, что оба эти документа посвящены *and* в той же мере, что и *kite*. Так, подождите-ка... Не очень информативно, не правда ли? Как и в первом примере про нашего быстрого друга Гарри, где система, похоже, думала, что *the* было самым важным словом в документе, в этом примере *and* считается довольно значимым. Не слишком большое достижение, даже на первый взгляд.

Обратную частотность термов документа можно рассматривать как меру того, насколько странным является тот факт, что данный токен присутствует в кон-

кретном документе. Если терм встречается в одном документе много раз, но редко в остальной части корпуса, можно предположить, что он играет важную роль именно в этом конкретном документе. Поздравляю, мы сделали первый шаг к тематическому моделированию!

IDF терма — это просто отношение общего количества документов к тем, в которых встречается терм. В случае *and* и *kite* ответ одинаков для обоих.

- Всего два документа / два документа содержат *and* = 2 / 2 = 1.
- Всего два документа / два документа содержат *kite* = 2 / 2 = 1.

Не очень интересно, не так ли? Рассмотрим слово *China*.

- Всего два документа / один документ содержит *China* = 2 / 1 = 2.

Мы что-то нашли. Будем использовать эту меру «редкости» в качестве веса для частотности терма:

```
>>> num_docs_containing_and = 0
>>> for doc in [intro_tokens, history_tokens]:
...     if 'and' in doc:
...         num_docs_containing_and += 1
```

Подобие слов kite и China

Найдем TF *China* в обоих документах:

```
>>> intro_tf['china'] = intro_counts['china'] / intro_total
>>> history_tf['china'] = history_counts['china'] / history_total
```

Вычислим IDF для всех трех слов. Хранить IDF мы будем в словарях для каждого документа, как мы это делали с TF:

```
>>> num_docs = 2
>>> intro_idf = {}
>>> history_idf = {}
>>> intro_idf['and'] = num_docs / num_docs_containing_and
>>> history_idf['and'] = num_docs / num_docs_containing_and
>>> intro_idf['kite'] = num_docs / num_docs_containing_kite
>>> history_idf['kite'] = num_docs / num_docs_containing_kite
>>> intro_idf['china'] = num_docs / num_docs_containing_china
>>> history_idf['china'] = num_docs / num_docs_containing_china
```

А затем вычислим то же самое для документа с первыми абзацами статьи из «Википедии»:

```
>>> intro_tfidf = {}
>>> intro_tfidf['and'] = intro_tf['and'] * intro_idf['and']
>>> intro_tfidf['kite'] = intro_tf['kite'] * intro_idf['kite']
>>> intro_tfidf['china'] = intro_tf['china'] * intro_idf['china']
```

И то же самое для документа с разделом History:

```
>>> history_tfidf = {}
>>> history_tfidf['and'] = history_tf['and'] * history_idf['and']
>>> history_tfidf['kite'] = history_tf['kite'] * history_idf['kite']
>>> history_tfidf['china'] = history_tf['china'] * history_idf['china']
```

3.4.1. Возвращаемся к закону Ципфа

Мы почти справились с заданием. Впрочем, допустим, что наш корпус состоит из 1 миллиона документов (например, у вас своя маленькая поисковая система) и кто-то ищет слово *cat*. Предположим, что из 1 миллиона документов *cat* содержит ровно один документ. Чистый IDF этого слова равен:

$$1\ 000\ 000 / 1 = 1\ 000\ 000.$$

Теперь представим, что из этого миллиона документов десять содержат слово *dog*. Тогда IDF этого слова будет:

$$1\ 000\ 000 / 10 = 100\ 000.$$

Как вы могли заметить, разница довольно большая. Наш друг Ципф сказал бы, что *слишком* большая. Закон Ципфа гласит, что при сравнении частотностей двух слов, таких как *cat* и *dog*, даже если они встречаются одинаковое количество раз, частотность более употребляемого слова будет экспоненциально выше, чем менее частого. Таким образом, закон Ципфа предполагает масштабирование всех частотностей слов (и частотностей документов) с помощью функции `log()`, обратной к функции `exp()`. Это гарантирует, что такие слова, как *cat* и *dog*, встречающиеся близкое число раз, не будут экспоненциально отличаться по частотности. Кроме того, такое распределение частотностей слов обеспечит более равномерное распределение показателей TF-IDF. Таким образом, IDF нужно переопределить так, чтобы он был равен логарифму исходной вероятности появления слова в одном из документов. Желательно также взять логарифм частотности терма[1].

Основание логарифма не имеет особого значения ввиду того, что мы хотим просто сделать распределение частотностей равномерным, а не масштабировать его до пределов определенного числового диапазона[2]. При использовании логарифма по основанию 10 получится:

- поиск по слову *cat*:

$$\text{idf} = \log(1\ 000\ 000 / 1) = 6;$$

- поиск по слову *dog*:

$$\text{idf} = \log(1\ 000\ 000 / 10) = 5.$$

Теперь мы учитываем в результатах TF каждого из этих слов вес в соответствии с частотностью их в языке в целом.

---

[1] Джерард Солтон (Gerard Salton) и Крис Бакли (Chris Buckley) впервые продемонстрировали полезность масштабирования логарифмов для поиска информации в своей статье Term Weighting Approaches in Automatic Text Retrieval (ecommons.cornell.edu/bitstream/handle/1813/6721/87-881.pdf).

[2] Позже мы покажем вам, как нормализовать векторы TF-IDF с помощью этого логарифмического масштабирования после вычисления всех значений TF-IDF.

Для терма t в документе d в корпусе D получаем:

$$\text{tf}(t, d) = \frac{\text{количество}(t)}{\text{количество}(d)};$$

$$\text{idf}(t, D) = \log \frac{\text{количество документов}}{\text{количество документов, содержащих } t};$$

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D).$$

Таким образом, чем больше раз слово встречается в документе, тем выше будет значение TF (и, следовательно, TF-IDF). В то же время по мере увеличения количества документов, содержащих слово, его IDF (и, следовательно, TF-IDF) будет уменьшаться. Итак, теперь у вас есть число — то, с чем компьютер может работать. Но что оно означает? Оно связывает определенное слово или токен с конкретным документом в определенном корпусе, а затем задает числовое значение важности этого слова в данном документе с учетом его встречаемости во всем корпусе.

Иногда все вычисления выполняются в логарифмическом пространстве, так что умножение становится сложением, а деление — вычитанием:

```
>>> log_tf = log(term_occurences_in_doc) -\
...     log(num_terms_in_doc)
>>> log_log_idf = log(log(total_num_docs) -\
...     log(num_docs_containing_term))
>>> log_tf_idf = log_tf + log_idf
```

Логарифм вероятности появления определенного терма в определенном документе

Логарифм логарифма вероятности того, что конкретный терм встретится в документе по крайней мере один раз. Первый логарифм предназначен для линеаризации IDF (компенсировать закон Ципфа)

Логарифм TF-IDF — это логарифм произведения TF и IDF (сумма их логарифмов)

Полученное число, TF-IDF, — скромный фундамент нашей простой поисковой системы. И раз мы уже твердо шагнули из мира текста в мир чисел, пришло время заняться математикой. Вероятно, вам никогда не придется реализовывать вышеприведенные формулы для вычисления TF-IDF. Линейная алгебра не требуется для полного понимания инструментов, используемых при обработке естественного языка, но общее знакомство с этой математикой сделает ее применение интуитивно понятнее.

3.4.2. Ранжирование по релевантности

Как вы уже видели, сравнить два вектора и узнать степень их подобия не очень сложно. Однако с тех пор вы также узнали, что простой подсчет слов не так информативен, как использование их TF-IDF. Так что заменим в каждом векторе документа `word_count` каждого слова на его TF-IDF. Теперь наши векторы точнее отражают смысл (тему) документа, как показано в следующем примере о Гарри:

Необходимо скопировать zero_vector, чтобы создать новый, отдельный объект. В противном случае вы будете перезаписывать один и тот же объект/вектор на каждой итерации

```
>>> document_tfidf_vectors = []
>>> for doc in docs:
...     vec = copy.copy(zero_vector)
...     tokens = tokenizer.tokenize(doc.lower())
...     token_counts = Counter(tokens)
...
...     for key, value in token_counts.items():
...         docs_containing_key = 0
...         for _doc in docs:
...             if key in _doc:
...                 docs_containing_key += 1
...         tf = value / len(lexicon)
...         if docs_containing_key:
...             idf = len(docs) / docs_containing_key
...         else:
...             idf = 0
...         vec[key] = tf * idf
...     document_tfidf_vectors.append(vec)
```

При таких настройках у нас есть K-мерное векторное представление каждого документа в корпусе. А теперь приступим к охоте! Или к поиску в нашем случае. Два вектора в векторном пространстве называются подобными, если одинаковы углы с его базисными векторами. Если представить, что каждый из этих векторов возникает в начале координат при тех же длинах и направлениях, то векторы, направленные под одинаковым углом, подобны, даже если их длины различны.

Два вектора считаются подобными, если значение их косинусного коэффициента велико, поэтому для поиска подобных векторов поблизости можно максимизировать:

$$\cos\Theta = \frac{A \cdot B}{|A||B|}$$

Теперь у нас есть все, что нужно для простого поиска на основе TF-IDF. Поисковый запрос можно рассматривать как документ, для которого существует векторное представление на основе TF-IDF. Последний шаг — найти документы, векторы которых имеют наибольшие косинусные коэффициенты с запросом, и вернуть их в качестве результатов поиска.

Для запроса *How long does it take to get to the store?* и трех наших документов о Гарри получается:

copy.copy() гарантирует, что вы имеете дело с отдельными объектами, а не с несколькими ссылками на один объект

```
>>> query = "How long does it take to get to the store?"
>>> query_vec = copy.copy(zero_vector)
>>> tokens = tokenizer.tokenize(query.lower())
>>> token_counts = Counter(tokens)
>>> for key, value in token_counts.items():
...     docs_containing_key = 0
...     for _doc in docs:
...         if key in _doc.lower():
```

```
...             docs_containing_key += 1
...     if docs_containing_key == 0:
...         continue
...         tf = value / len(tokens)
...         idf = len(docs) / docs_containing_key
...     query_vec[key] = tf * idf
>>> cosine_sim(query_vec, document_tfidf_vectors[0])
0.5235048549676834
>>> cosine_sim(query_vec, document_tfidf_vectors[1])
0.0
>>> cosine_sim(query_vec, document_tfidf_vectors[2])
0.0
```

Данный токен отсутствует в лексиконе, переходим к следующему ключу

Можно с уверенностью утверждать, что документ 0 наиболее релевантен запросу. Аналогичным образом можно использовать релевантные документы в любом корпусе. Неважно в каком — в статьях из «Википедии», книгах из проекта «Гутенберг» или твитах из Twitter. Google, берегись!

Поисковая система Google может нас не бояться. Для каждого запроса нам приходится выполнять поиск по индексу среди векторов TF-IDF. Это алгоритм порядка O(N). Большинство поисковых систем отвечают за фиксированный промежуток времени (O(1)), потому что используют *обратный индекс* (ru.wikipedia.org/wiki/Инвертированный_индекс). Мы не собираемся реализовывать здесь индекс, пригодный для поиска соответствий за фиксированный промежуток времени, но, если вам интересно, взгляните на современную реализацию на языке Python из пакета *Whoosh* (pypi.python.org/pypi/Whoosh) и ее исходный код[1]. Вместо того чтобы демонстрировать создание подобной традиционной поисковой системы на основе ключевых слов, в главе 4 мы расскажем про новейшие подходы семантического индексирования, захватывающие смысл текста.

ПРИМЕЧАНИЕ

В предыдущем коде мы отбросили отсутствующие в лексиконе ключи, чтобы избежать ошибки деления на ноль. Куда лучшим подходом будет прибавление 1 к знаменателю при каждом вычислении IDF, что гарантирует отсутствие нулевых знаменателей. На деле этот подход, называемый *аддитивным сглаживанием* (сглаживанием Лапласа) (en.wikipedia.org/wiki/Additive_smoothing), обычно улучшает результаты поиска по ключевым словам TF-IDF.

Поиск по ключевым словам — это только один из инструментов нашего NLP-конвейера. Мы собираемся создать чат-бот, а некоторые чат-боты используют в качестве единственного алгоритма генерации ответов исключительно поисковую систему. Нам нужно сделать еще один шаг, чтобы превратить наш простой поисковый индекс (TF-IDF) в чат-бот. Нам также необходимо хранить тренировочные данные в виде пар вопросов (высказываний) и соответствующих ответов. При этом можно применять TF-IDF для поиска вопроса (высказывания), наиболее похожего на вводимый

[1] См. веб-страницу GitHub — Mplsbeb/whoosh: A fast pure-Python search engine по адресу github.com/Mplsbeb/whoosh.

пользователем текст. Вместо того чтобы возвращать наиболее похожее высказывание из базы данных, мы будем возвращать соответствующий этому высказыванию ответ. Как и любая сложная задача из сферы компьютерных наук, наша может быть решена с помощью добавления еще одного слоя косвенности. И все, чат-бот готов!

3.4.3. Инструменты

Выше было приведено много кода для давно уже автоматизированных операций. То же самое можно сделать намного проще с помощью пакета `scikit-learn`[1]. Если вы еще не настроили свою среду согласно информации из приложения А таким образом, чтобы она включала этот пакет, вот один из способов его инсталляции:

```
pip install scipy
pip install sklearn
```

Так вы можете использовать `sklearn` для построения матрицы TF-IDF. Класс для TF-IDF библиотеки `sklearn` — это *модель* с методами `.fit()` и `.transform()`, которые соответствуют API sklearn для всех моделей машинного обучения:

Модель TFIDFVectorizer создает разреженную матрицу numpy, потому что матрица TF-IDF обычно содержит в основном нули, поскольку в большинстве документов используется лишь небольшая часть всех слов словаря

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = docs
>>> vectorizer = TfidfVectorizer(min_df=1)
>>> model = vectorizer.fit_transform(corpus)
>>> print(model.todense().round(2))
[[0.16 0.   0.48 0.21 0.21 0.   0.25 0.21 0.   0.   0.   0.21 0.   0.64
  0.21 0.21]
 [0.37 0.   0.37 0.   0.   0.37 0.29 0.   0.37 0.37 0.   0.   0.49 0.
  0.   0.  ]
 [0.   0.75 0.   0.   0.   0.29 0.22 0.   0.29 0.29 0.38 0.   0.   0.
  0.   0.  ]]
```

Метод .todense() преобразует разреженную матрицу обратно в обычную матрицу numpy (заполняя промежутки нулями) для удобства просмотра

С помощью библиотеки `scikit-learn`, написав всего четыре строки кода, мы смогли создать матрицу для трех документов и IDF для всех термов лексикона. Полученная матрица (которая в Python фактически представляет собой список списков) отражает наши три документа (три строки матрицы). TF-IDF каждого терма, токена или слова лексикона составляют столбцы матрицы (или опять же индексы строк). Их всего 16, так как они по-разному токенизируются, а знаки пунктуации отбрасываются — в нашем случае запятая и точка. При работе с большими текстами эта или другая предварительно оптимизированная модель TF-IDF поможет сэкономить массу времени.

---

[1] См. веб-страницу scikit-learn: machine learning in Python по адресу scikit-learn.org.

3.4.4. Альтернативы

Матрицы TF-IDF (матрицы «терм — документ») были основой поиска информации на протяжении десятилетий. В результате исследователи и корпорации потратили много времени, пытаясь оптимизировать связанную с IDF часть, чтобы повысить релевантность результатов поиска. В табл. 3.1 перечислены некоторые способы нормализации и сглаживания весовых коэффициентов частотностей термов[1].

Таблица 3.1. Альтернативные способы нормализации TF-IDF (Молино, 2017)

Схема	**Описание**
Отсутствует	$w_{ij} = f_{ij}$
TF-IDF	$w_{ij} = \log(f_{ij}) \times \log\left(\frac{N}{n_j}\right)$
TF-ICF	$w_{ij} = \log(f_{ij}) \times \log\left(\frac{N}{f_j}\right)$
Okapi BM25	$w_{ij} = \frac{t_{ij}}{0{,}5 + 1{,}5 \times \frac{f_j}{\frac{f_j}{j}} + f_{ij}} \log\frac{N - n_j + 0{,}5}{f_{ij} + 0{,}5}$
ATC	$w_{ij} = \frac{\left(0{,}5 + 0{,}5 \times \frac{f_{ij}}{\max_j}\right)\log\left(\frac{N}{n_j}\right)}{\sqrt{\sum_{i=1}^{N}\left[\left(0{,}5 + 0{,}5 \times \frac{f_{ij}}{max_j}\right)\log\left(\frac{N}{n_j}\right)\right]^2}}$
LTU	$w_{ij} = \frac{\left(\log(f_{ij}) + 1{,}0\right)\log\left(\frac{N}{n_j}\right)}{0{,}8 + 0{,}2 \times f_j \times \frac{j}{f_j}}$
MI	$w_{ij} = \log\frac{P(t_{ij} \mid c_j)}{P(t_{ij})P(c_j)}$
PosMI	$w_{ij} = \max(0, \text{MI})$
T-Test	$w_{ij} = \frac{P(t_{ij} \mid c_j) - P(t_{ij})P(c_j)}{\sqrt{P(t_{ij})P(c_j)}}$
x^2	См. подраздел 4.3.5 книги From Distributional to Semantic Similarity («От дистрибутивного подобия к семантическому») по адресу www.era.lib.ed.ac.uk/bitstream/handle/1842/563/IP030023.pdf#subsection.4.3.5, написанный Джеймсом Ричардом Карреном (James Richard Curran)

Продолжение ⇩

[1] *Molino P.* Word Embeddings Past, Present and Future. Конференция AI with the Best 2017.

Таблица 3.1 *(продолжение)*

Схема	Описание
Lin98a	$w_{ij} = \frac{f_{ij} \times f}{f_i \times f_j}$
Lin98b	$w_{ij} = -1 \times \log \frac{n_j}{N}$
Gref94	$w_{ij} = \frac{\log t_{ij} + 1}{\log n_j + 1}$

Поисковые системы (информационно-поисковые системы) сопоставляют ключевые слова (термы) из запросов и документов корпуса. Если вы решили создать поисковую систему и хотите выдавать по возможности искомые пользователями документы, рекомендуем потратить некоторое время на изучение альтернатив, описанных Пьеро Молино (Piero Molino) в табл. 3.1.

Одним из альтернативных способов использования непосредственно косинусного расстояния TF-IDF для ранжирования результатов запроса вариантов является Okapi BM25 или его самый последний вариант BM25F.

3.4.5. Okapi BM25

Умные люди из Лондонского университета Сити придумали лучший способ ранжирования результатов поиска. Вместо простого вычисления косинусного подобия TF-IDF они его нормализовали и сгладили, а также проигнорировали повторяющиеся термы в документе запроса, фактически обрезая частотности термов вектора запроса в 1. Скалярное произведение косинусного подобия нормализуется не векторной нормой TF-IDF (количеством слов в документе и запросе), а с помощью нелинейной функции от длины самого документа:

```
q_idf * dot(q_tf, d_tf[i]) * 1.5 /
➥ (dot(q_tf, d_tf[i]) + .25 + .75 * d_num_words[i] / d_num_words.mean()))
```

Вы можете оптимизировать свой конвейер, выбрав схему взвешивания, выдающую вашим пользователям наиболее релевантные результаты. Если же ваш корпус не слишком велик, то можете отправиться вместе с нами далее и узнать о более удобных и точных представлениях смысла слов и документов. В следующих главах мы покажем, как реализовать семантическую поисковую систему, находящую документы, близкие по смыслу к словам из запроса, а не просто документы, содержащие в точности те же слова, что и в запросе. Семантический поиск гораздо лучше всего, чего можно достичь с помощью методов использования весов TF-IDF, стемминга и лемматизации. Единственная причина, по которой Google, Bing и другие поисковые системы не используют подход семантического поиска, заключается в слишком больших размерах их корпусов. Семантические векторы слов и тем не масштабируются до миллиардов документов, хотя применять их для корпусов в миллионы документов можно.

Таким образом, для максимально достижимой на текущий момент эффективности семантического поиска, классификации документов, диалоговых систем и большинства других приложений, упомянутых в главе 1, достаточно подать на вход конвейера лишь основные векторы TF-IDF. Векторы TF-IDF — это первая стадия конвейера, основной набор выделяемых из текста признаков. В следующей главе мы вычислим на основе векторов TF-IDF векторы тем. Последние отражают смысл содержимого мультимножества слов куда лучше, чем любой из этих тщательно нормализованных и сглаженных векторов TF-IDF. Дальше мы добьемся еще большего, когда перейдем к векторам слов Word2vec в главе 6 и вложениям[1] смыслов слов и документов в нейронных сетях.

3.4.6. Что дальше?

Теперь, когда мы научились преобразовывать текст на естественном языке в числа, начнем производить над этими числами различные операции и вычисления. Числа у нас уже есть, и в следующей главе мы займемся их уточнением в попытке лучше отразить *смысл (тему)* текста на естественном языке, а не только отдельных слов.

Резюме

- «Под капотом» любой поисковой системы интернет-масштаба со временем отклика порядка миллисекунд находится матрица «терм — документ» (TF-IDF).
- Частотности термов необходимо умножить на веса в виде обратных частотностей документов, чтобы гарантировать достаточный вес важнейших, несущих наибольший смысл слов.
- Закон Ципфа позволяет предсказать частотность чего угодно, включая слова, символы и людей.
- Строки TF-IDF матрицы «терм— документ» можно использовать в качестве векторного представления смыслов отдельных слов для создания векторной модели семантики слов.
- Евклидово расстояние и подобие между парами многомерных векторов не дает адекватного отображения их сходства в большинстве приложений NLP.
- Для эффективного вычисления косинусного расстояния, величины «перекрытия» векторов, достаточно просто перемножить элементы нормализованных векторов и просуммировать полученные произведения.
- Косинусное расстояние — оптимальный показатель подобия для большинства векторных представлений текстов на естественном языке.

---

[1] Термин embedding в данном контексте не имеет устоявшегося перевода и в этой книге переводится как вложение. Близкие по смыслу термины – сопоставление, отображение, представление. — *Примеч. пер.*

4 Поиск смысла слов по их частотностям: семантический анализ

В этой главе

- Создание векторов тем с помощью анализа семантики (смысла).
- Семантический поиск на основе сходства векторов тем.
- Масштабируемый семантический анализ и семантический поиск для больших корпусов текстов.
- Семантические компоненты (темы) как признаки в конвейере NLP.
- Ориентация в векторных пространствах высокой размерности.

Вы уже выучили немало приемов обработки написанных на естественных языках текстов. Но сейчас мы перейдем к настоящей магии: впервые поговорим о том, что машина может понимать смысл слов.

Благодаря векторам TF-IDF (векторы частотности термов по отношению к обратной частотности документа) из главы 3 можно оценивать важность слов во фрагменте текста. Мы использовали векторы и матрицы TF-IDF, чтобы понять, насколько важно каждое из слов для общего смысла фрагмента текста в наборе документов.

Подобные оценки важности TF-IDF возможны не только для слов, но и для коротких последовательностей слов, *n*-грамм. Причем последние очень удобны для поиска в тексте (если искомые слова или *n*-граммы точно известны).

В результате проведенных в сфере NLP экспериментов был найден алгоритм для выяснения смысла словосочетаний и вычисления векторов представления этого смысла. Он называется *латентно-семантическим анализом* (latent semantic analysis, LSA). С его помощью можно представить в виде векторов смысл не только слов, но и целых документов.

В данной главе мы расскажем про эти *семантические* векторы (векторы *тем*)[1]. Мы воспользуемся взвешенными оценками частотностей из векторов TF-IDF для вычисления показателей тем, из которых состоят измерения их векторов, а на основе корреляции нормализованных частотностей термов друг с другом сгруппируем слова в темах для задания измерений наших новых векторов тем.

С помощью этих векторов тем можно делать множество интересных вещей, например искать документы по их смыслу — производить *семантический поиск*. В большинстве случаев результаты этого поиска намного лучше, чем по ключевым словам (поиск TF-IDF). Иногда семантический поиск возвращает именно те документы, которые искал пользователь, даже если последний не нашел правильных слов для запроса.

Кроме того, эти семантические векторы можно использовать для нахождения слов и *n*-грамм, которые лучше всего отражают предмет (тему) высказывания, документа или корпуса (набора документов). С помощью векторов слов и их относительной значимости можно найти наиболее показательные слова для документа, например набор ключевых слов, отражающих его смысл.

Также теперь можно сравнить любые два утверждения/документа и сказать, насколько близки они друг к другу *по смыслу*.

СОВЕТ

> Смысл терминов «тема», «семантика» и «смысл» в контексте NLP схож. Они часто заменяют друг друга. В этой главе вы узнаете, как создать конвейер NLP, способный самостоятельно выяснить наличие подобной синонимии. Возможно, ваш конвейер даже сможет уловить сходство смыслов слов «выяснить» и «вычислить». Компьютер способен только «вычислить» смысл.

Скоро вы увидите, что линейные комбинации слов, из которых состоят измерения векторов тем, — весьма многообещающее представление смысла.

---

[1] В этой главе, посвященной семантическому анализу, мы будем использовать термин «вектор темы», а в главе 6, посвященной Word2vec, будет термин «вектор слов». В строгих трудах по NLP, например в настольной книге по NLP Д. Журафски (Jurafsky) и Дж. Мартина (Martin) (https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf#chapter.15), применяется термин «вектор слов». Другие, скажем авторы статьи Semantic Vector Encoding and Similarity Search (https://arxiv.org/pdf/1706.00957.pdf), употребляют термин «семантический вектор».

4.1. От частотностей слов до оценок тем

Скоро вы узнаете, как подсчитать частотность слов. Вам уже известно, как оценить важность слов в векторе или матрице TF-IDF. Но этого мало. Необходимо оценить смыслы, темы, для которых используются слова.

4.1.1. Векторы TF-IDF и лемматизация

Векторы TF-IDF подсчитывают количество добуквенно точных вхождений термов в документе. Так что у текстов, выражающих одно и то же разными словами или в различной орфографии, будут различные представления векторов TF-IDF. Это затрудняет работу поисковых систем и средств сравнения документов, в основе которых лежит подсчет количества токенов.

В главе 2 мы нормализовали окончания слов, так что различающиеся лишь несколькими последними буквами слова группировались в один токен. Для создания небольших наборов слов со схожим написанием и зачастую схожим смыслом мы использовали такие методы нормализации, как стемминг и лемматизация. Каждый из этих наборов слов маркировался соответствующей леммой или основой слова, после чего эти новые токены обрабатывались вместо исходных слов.

Благодаря лемматизации при анализе группировались слова со схожим написанием[1], но не обязательно со схожим смыслом. Не группировались синонимы. Они обычно различаются отнюдь не только окончаниями, с которыми имеют дело лемматизация и стемминг. Хуже того, в результате лемматизации и стемминга порой группируются антонимы — слова с противоположным смыслом.

В итоге два фрагмента текста, говорящие одно и то же разными словами, окажутся далеко друг от друга в лемматизированной модели TF-IDF векторного пространства. Иногда смыслы двух близких лемматизированных векторов TF-IDF совершенно различны. Даже самые передовые оценки сходства TF-IDF из главы 3, такие как Okapi BM25 или косинусный коэффициент, не могут связать такие синонимы или разделить антонимы. Синонимы с различным написанием просто приводят к далеким друг от друга в векторном пространстве векторам TF-IDF.

Например, вектор TF-IDF для этой главы не слишком напоминает схожие по звучанию места вузовских учебников, рассказывающие о латентно-семантической индексации. Но именно об этом мы здесь и будем рассказывать, просто на более современном и нестрогом языке. Профессора и исследователи в своих лекциях и учебниках излагают материал более единообразным и строгим языком. Кроме того, использовавшаяся в вузах десятилетие назад терминология сильно изменилась соответственно быстрому развитию данной сферы за последние годы. Например, термин «латентно-семантическая *индексация*» применялся тогда намного чаще, чем сейчас термин «латентно-семантический анализ»[2].

---

[1] Как при стемминге, так и при лемматизации удаляются или изменяются окончания и суффиксы — последние несколько букв слов. Для обнаружения слов с одинаковым написанием (или одинаковыми орфографическими ошибками) лучше подходит метод вычисления расстояния редактирования.

[2] Я обожаю просмотрщик n-грамм от компании Google за возможность визуализации подобных тенденций (http://mng.bz/7Jnm).

4.1.2. Векторы тем

При выполнении над векторами TF-IDF арифметических действий, например сложения и вычитания, полученные суммы и разности говорят только о частоте слов в документах, векторы которых вы складываете или вычитаете. Подобные арифметические результаты ничего не говорят о смысле соответствующих слов. Вычислить пословные векторы TF-IDF (векторы совместно встречающихся слов, корреляции) можно, умножив матрицу TF-IDF на себя. Но анализировать векторы на основе этих разреженных векторов высокой размерности получается плохо. При их сложении или вычитании не получается хороших представлений существующих понятий/слов/тем.

Таким образом, нужно научиться извлекать дополнительную информацию, смысл из статистики по словам. Надо найти лучшую оценку для того, что обозначают данные слова в каком-либо документе. И знать, каков *смысл* конкретного словосочетания в определенном документе. Выразить его с помощью вектора, подобного вектору TF-IDF, но более сжатого и осмысленного.

Такие сжатые векторы смысла мы будем называть векторами тем слов, а векторы, отражающие смысл документов, — векторами тем документов. И те и другие можно называть векторами тем, главное — понимать, к чему вектор темы относится: к слову или документу.

Эти векторы тем могут быть столь сжатыми или обширными (многомерными), как вы пожелаете. Векторы тем LSA могут состоять только из одного или из тысяч измерений.

Векторы тем, которые мы вычисляем в этой главе, можно складывать или вычитать, подобно любым другим векторам. В данном случае смысл этих сумм и разностей гораздо глубже, чем у векторов TF-IDF (из главы 3). Да и расстояния между векторами тем могут пригодиться при кластеризации документов и семантическом поиске. Раньше мы производили кластеризацию слов и поиск по ключевым словам и векторам TF-IDF. Теперь можно производить кластеризацию и поиск по семантике, то есть по смыслу!

По окончании вы получите по одному вектору темы документа для каждого документа корпуса. Что еще важнее, для вычисления нового вектора темы для нового документа или фразы не нужно заново обрабатывать весь корпус. У вас будут векторы тем для всех слов словаря, и ими можно будет воспользоваться для вычисления вектора темы для любого документа, содержащего какие-либо из этих слов.

СОВЕТ

Некоторые алгоритмы создания векторов тем, например латентное размещение Дирихле (latent Dirichlet allocation), требуют повторной обработки всего корпуса при каждом добавлении нового документа.

Для каждого слова лексикона (словаря) у вас будет один вектор темы слова. Чтобы вычислить вектор темы для любого нового документа, достаточно сложить все векторы тем его слов.

Подбор числового представления семантики (смысла) слов и предложений — непростая задача, особенно для «нечетких» языков, таких как английский, с множеством

диалектов и различных значений одного слова. Даже написание текста профессором-лингвистом не отменяет наличия у большинства слов английского языка нескольких значений — головная боль для учащихся — как людей, так и машин. Концепция нескольких значений у слов называется *многозначностью* (polysemy).

- *Многозначность* — существование слов и фраз, у которых более одного смысла.

  Многозначность может влиять на семантику слов или высказываний по-разному. Мы перечислим эти варианты ниже, чтобы вы оценили всю широту возможностей LSA. Не стоит волноваться об этих сложностях, LSA берет их на себя.
- *Омонимы* — слова с одинаковым написанием и произношением, но различным смыслом.
- *Зевгма* — использование в одном предложении сразу двух значений одного слова.

  LSA способен справиться с некоторыми сложными задачами, возникающими вследствие многозначности в голосовом интерфейсе — при использовании голосовых чат-ботов, например Alexa или Siri.
- *Омографы* — слова с одинаковым написанием, но различным произношением и смыслом.
- *Омофоны* — слова с одинаковым произношением, но различным написанием и смыслом (непростая задача для NLP в случае голосового интерфейса).

Представьте, как сложно было бы обработать следующее высказывание, не будь таких инструментов, как LSA: *She felt ... less. She felt tamped down. Dim. More faint. Feint. Feigned. Fain* (Патрик Ротфус).

С учетом этих непростых задач как бы вы втиснули вектор TF-IDF из миллиона измерений (термов) в вектор из примерно 200 измерений (тем)? Это словно искать сочетание основных цветов, которое бы давало такой оттенок, как в вашей квартире, чтобы закрасить отверстия от гвоздей в стене.

Необходимо найти «соответствующие» друг другу в теме измерения слов и сложить их векторы TF-IDF. В результате получится новое число, отражающее долю данной темы в документе. Можно даже умножить их на веса, отражающие их важность для темы в целом, степень вложения каждого отдельного слова в «смесь», а для слов, снижающих вероятность того, что текст посвящен данной теме, можно использовать отрицательные веса.

4.1.3. Мысленный эксперимент

Проведем мысленный эксперимент. Пускай у вас есть вектор TF-IDF для конкретного документа и его нужно преобразовать в вектор темы. Задумаемся, насколько много каждое слово вкладывает в тему.

Допустим, мы обрабатываем предложения, посвященные домашним животным в Центральном парке в Нью-Йорке (NYC). Создадим три темы: о домашних животных, о животных вообще и о городах. Назовем их `petness`, `animalness` и `cityness`. Вероятно, в нашей посвященной питомцам теме `petness` веса таких слов, как `cat` (кошка) и `dog` (собака), будут довольно значительны, а `NYC` и `apple` (яблоко) будут

игнорироваться. В теме `cityness` будут игнорироваться `cat` и `dog`, но небольшой вес получит `apple`, из-за ассоциации с *Big Apple*.

Если обучить тематическую модель подобным образом — без компьютера, на основе одного здравого смысла, то веса окажутся следующими:

```
>>> topic = {}
>>> tfidf = dict(list(zip('cat dog apple lion NYC love'.split(),
... np.random.rand(6))))
>>> topic['petness'] = (.3 * tfidf['cat'] +\
...                     .3 * tfidf['dog'] +\
...                      0 * tfidf['apple'] +\
...                      0 * tfidf['lion'] -\
...                     .2 * tfidf['NYC'] +\
...                     .2 * tfidf['love'])
>>> topic['animalness'] = (.1 * tfidf['cat'] +\
...                        .1 * tfidf['dog'] -\
...                        .1 * tfidf['apple'] +\
...                        .5 * tfidf['lion'] +\
...                        .1 * tfidf['NYC'] -\
...                        .1 * tfidf['love'])
>>> topic['cityness']   = ( 0 * tfidf['cat'] -\
...                        .1 * tfidf['dog'] +\
...                        .2 * tfidf['apple'] -\
...                        .1 * tfidf['lion'] +\
...                        .5 * tfidf['NYC'] +\
...                        .1 * tfidf['love'])
```

Вектор tfidf — просто случайный пример, вычисленный так, как будто для отдельного документа, содержащего эти слова в случайной пропорции

«Самодельные» весовые коэффициенты (.3, .3, 0, 0, –.2, .2) умножаются на фиктивные значения вектора tfidf для нашего воображаемого случайного документа. Позже мы вычислим настоящие векторы тем

В этом мысленном эксперименте мы сложили частотности слов, которые могут указывать на каждую из наших тем. Задали веса для частотностей слов (значений TF-IDF) в зависимости от того, насколько слово связано с конкретной темой. То же самое, только со знаком минус сделали для слов, указывающих на нечто в каком-то смысле противоположное теме. Конечно, это не настоящий анализ алгоритма или пример реализации, а просто мысленный эксперимент. В нем мы пытаемся понять, как обучить машину думать так, как человек. Мы произвольно разбиваем слова и документы всего на три темы (`petness`, `animalness` и `cityness`). Да и словарь ограничен шестью словами.

Следующий шаг: разобраться, как бы человек мог математически определить, какие темы и слова связаны и какие веса должны быть у этих связей. После выбора трех тем для модели, необходимо подобрать веса всех слов для этих тем. Мы смешали слова в таком соотношении друг к другу, чтобы превратить тему в «смешение цветов». Преобразование тематического моделирования (рецепт смешения цветов) представляет собой матрицу соотношений (весов) 3×6, задающую соответствия трех тем и шести слов. Мы умножили эту матрицу на воображаемый вектор TF-IDF 6×1 и получили вектор темы 3×1 для данного документа.

Мы приняли субъективное решение, что вклад термов **`cat`** и **`dog`** в тему **`petness`** должен быть одинаков (веса равны **`0.3`**). Так что два значения в верхнем левом углу матрицы для нашего преобразования TF-IDF в тему будут равны `.3`. Как можно «вычислить» эти соотношения с помощью программного обеспечения? Помните, у вас есть набор документов, которые ваш компьютер может читать, токенизировать

и подсчитывать для них количество токенов. Можно также получить векторы TF-IDF для любого числа документов. Продолжайте думать, как применять эти данные для вычисления весов слов для тем при дальнейшем чтении.

Мы решили, что вес терма `NYC` для темы `petness` должен быть отрицательным. В определенном смысле у названий городов, да и вообще имен собственных и сокращений и акронимов, мало общего со словами, связанными с питомцами. Задумайтесь, что для слов означает фраза «мало общего». Отражает ли что-то в матрице TF-IDF общность смыслов слов?

Мы задали `love` (любовь) положительный вес для `petness`. Возможно, потому, что оно часто употребляется в одном предложении со словами, связанными с домашними животными. В конце концов, мы обычно любим своих питомцев. Остается надеяться, что наши властители ИИ будут любить нас не меньше.

Обратите внимание, как мал вес `apple` для темы `cityness`. Дело в том, что мы задаем веса вручную и знаем, что *NYC* и *Big Apple* часто оказываются синонимами. Надеемся, что наш алгоритм семантического анализа сможет вычислить эту синонимию между *apple* и *NYC* по тому, насколько часто эти слова встречаются в одних документах.

Читая в коде примера остальные взвешенные суммы, подумайте, как бы вы выбрали веса для этих трех тем и шести слов. Как бы поменяли их? Что можно выбрать в качестве объективной меры этих соотношений (весов)? У вас может быть на уме корпус, отличный от нашего. Так что и мнение о словах и назначаемых им весах будет другое. Как же прийти к единому мнению?

ПРИМЕЧАНИЕ

> Для получения векторов тем мы решили использовать веса слов со знаком. Благодаря этому можно задавать отрицательные веса для слов, антонимичных теме. Поскольку мы делаем это вручную, то будем нормализовать векторы тем удобной для вычисления L^1-нормой (называемой также манхэттенской метрикой, метрикой такси или метрикой городских кварталов). Но настоящий метод LSA (см. далее в этой главе) нормализует векторы тем с помощью более удобной L^2-нормы. Это обычное евклидово расстояние, знакомое вам из школьной геометрии. Оно выражается в виде решения теоремы Пифагора относительно длины гипотенузы правильного треугольника.

Связи между словами и темами можно «отобразить зеркально». Нужно транспонировать матрицу 3×6 трех векторов тем и получить веса тем для всех слов словаря. Эти векторы весов будут векторами для наших шести слов:

```
>>> word_vector = {}
>>> word_vector['cat']  =  .3*topic['petness'] +\
...                        .1*topic['animalness'] +\
... 0*topic['cityness']
>>> word_vector['dog']  =  .3*topic['petness'] +\
...                        .1*topic['animalness'] -\
...                        .1*topic['cityness']
>>> word_vector['apple']=   0*topic['petness'] -\
```

```
...                         .1*topic['animalness'] +\
...                         .2*topic['cityness']
>>> word_vector['lion'] =    0*topic['petness'] +\
...                         .5*topic['animalness'] -\
...                         .1*topic['cityness']
>>> word_vector['NYC']   = -.2*topic['petness'] +\
...                         .1*topic['animalness'] +\
...                         .5*topic['cityness']
>>> word_vector['love'] =   .2*topic['petness'] -\
...                         .1*topic['animalness'] +\
...                         .1*topic['cityness']
```

Эти шесть векторов (показанные на рис. 4.1), по одному для каждого слова, отражают значения наших шести слов в виде трехмерных векторов.

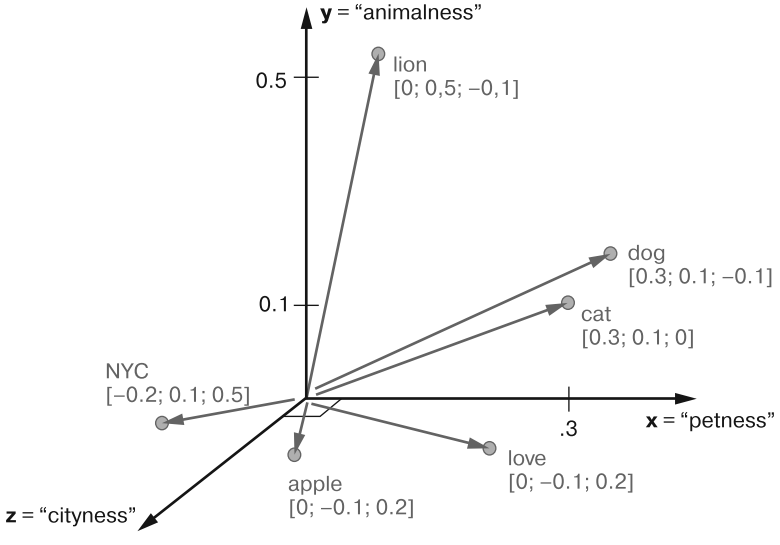


Рис. 4.1. 3D-векторы для мысленного эксперимента с тремя словами, связанными с домашними животными и NYC

Ранее у нас были 6D-векторы, отражающие линейные сочетания слов в наших трех темах, полученные из векторов для всех тем, с весами для слов. В нашем мысленном эксперименте мы вручную создали модель из трех тем для отдельного документа на естественном языке! Чтобы получить 3D-вектор темы для любого документа, достаточно подсчитать вхождения этих шести слов и умножить их на веса. Трехмерные векторы удобны своей наглядностью. Можно нарисовать их на графике и делиться информацией о корпусе или конкретном документе в графическом виде. 3D-векторы (или любые векторы низкой размерности) отлично подходят для классификационных задач машинного обучения. Алгоритм может разделить векторное пространство плоскостью (или гиперплоскостью) на классы.

В документах из корпуса может содержаться намного больше слов, но на данную конкретную модель векторов тем влияет использование только этих шести. Можно

расширить этот подход на любое нужное число слов (насколько хватит вашего терпения). Если модель должна всего лишь разделять документы по трем различным измерениям (темам), словарь может расти столько, сколько вам хочется. В мысленном эксперименте мы сжали шесть измерений (нормализованных частотностей TF-IDF) в три измерения (темы).

При подобном субъективном трудоемком подходе к семантическому анализу в основе разбиения документов по темам лежат человеческая интуиция и здравый смысл. Последний сложно запрограммировать алгоритмически[1]. Значит, он невоспроизводим — вероятно, вы выбрали бы не такие веса, как мы. И он не подходит для конвейера машинного обучения. Вдобавок он плохо масштабируется на большее число тем и слов. Человек не способен распределить достаточно много слов по нужному количеству тем, чтобы точно захватить смысл любого разнообразного корпуса документов.

Автоматизируем эту выполнявшуюся пока что вручную процедуру. Воспользуемся алгоритмом, который бы не зависел от здравого смысла при выборе весов тем[2].

Каждая из этих взвешенных сумм — всего лишь скалярное произведение, а три скалярных произведения (взвешенные суммы) — умножение (внутреннее произведение) матриц. Мы умножаем матрицу весов $3 \times n$ на вектор TF-IDF (по одному значению для каждого слова в документе), где n — число термов в словаре. Результат этого умножения представляет собой новый вектор темы размера 3×1 для данного документа. Мы «перевели» вектор из одного векторного пространства (TF-IDF) в другое низкоразмерное векторное пространство (векторов тем). Наш алгоритм должен создать матрицу размером n термов на m документов, которую можно будет умножить на вектор частотностей слов в документе и получить новый вектор темы для него.

ПРИМЕЧАНИЕ

В математике размер словаря (набора всех возможных слов языка) обычно обозначается |V|, а сама переменная V используется для обозначения набора возможных слов в словаре. Так что при написании научных статей рекомендуем использовать |V| во всех ситуациях, когда мы применяли *n* для обозначения размера словаря.

---

[1] Дуг Ленат из Стэнфордского университета пытается сделать именно это: запрограммировать здравый смысл алгоритмически. См. статью из журнала Wired Doug Lenat's Artificial Intelligence Common Sense Engine: https://www.wired.com/2016/03/doug-lenat-artificial-intelligence-common-sense-engine.

[2] На странице «Википедии» о тематических моделях есть видео, демонстрирующее это для намного большего числа тем и слов. Степень темноты фона пикселов отражает вес/значение/оценку для тем и слов, подобно весам в нашем «ручном» примере. На этом видео продемонстрировано переупорядочение слов и тем с помощью конкретного алгоритма SVD так, чтобы веса были максимальны вдоль диагонали. Это помогает при распознании закономерностей, отражающих смыслы тем и слов, https://upload.wikimedia.org/wikipedia/commons/7/70/Topic_model_scheme.webm#t=00:00:01,00:00:17.600.

4.1.4. Алгоритм оценки тем

Нам по-прежнему нужен алгоритмический способ определения векторов тем. Нам надо научиться преобразовывать векторы TF-IDF в тематические. Машина не знает, какие слова связаны или что они означают. Дж. Р. Ферс, британский лингвист XX века, изучал способы, как определить, что означает конкретное слово или морфема[1]. Ферс писал: *«Слово можно узнать по его друзьям»*.

Как же узнать, с какими словами «дружит» конкретное слово? Наиболее очевидный способ: подсчитать совместные вхождения в одном документе. Наше мультимножество слов (BOW) и векторы TF-IDF из главы 3 содержат нужную информацию. Такой подход с подсчетом совместных вхождений привел к появлению нескольких алгоритмов создания векторов, отражающих статистику использования слов в документах или предложениях.

Алгоритм LSA предназначен для анализа матрицы TF-IDF (таблицы векторов TF-IDF) и сбора слов по темам. Для работы с векторами наборов слов он тоже подходит, но при использовании векторов TF-IDF результаты оказываются немного лучше.

LSA также оптимизирует темы для сохранения разнообразия их измерений, чтобы вместо исходных слов этих новых тем при использовании все равно захватывался смысл (семантика) документов. Число тем, необходимых модели для захвата смысла документов, намного меньше количества слов в словаре векторов TF-IDF. Поэтому LSA часто называется методикой понижения размерности. Она уменьшает необходимое для захвата смысла документов число измерений.

Приходилось ли вам использовать методику понижения размерности для большой числовой матрицы? А пикселов? Если вы применяли машинное обучение к изображениям или другим данным высокой размерности, то сталкивались с методом главных компонент (principal component analysis, PCA). Как оказалось, лежащая в основе PCA математика точно такая же, как и у LSA. Просто о PCA говорят в случае понижения размерности изображений или других числовых таблиц, а не векторов мультимножеств или векторов TF-IDF.

Только недавно исследователи обнаружили, что PCA подходит для семантического анализа слов. Тогда это конкретное применение метода получило собственное название, LSA. Вскоре вы увидите, как мы используем модель PCA библиотеки `scikit-learn`, но в результате этого процесса подгонки и преобразования получается отражающий семантику документа вектор. Это все равно LSA.

Вы можете натолкнуться еще на один синоним LSA. В сфере информационного поиска, где основное внимание уделяется созданию индексов для полнотекстового поиска, LSA часто называют латентно-семантической индексацией (latent semantic indexing, LSI). Но термин практически вышел из употребления. Никакого индекса в результате не получается. На самом деле размерность генерируемых им векторов тем обычно слишком велика для того, чтобы их можно было хорошо индексировать. Поэтому далее будем использовать сокращение LSA.

---

[1] Морфема — наименьшая значимая часть слова. См. https://ru.wikipedia.org/wiki/Морфема.

ПРИМЕЧАНИЕ

Базы данных используют индексацию для быстрого извлечения конкретной строки из таблицы на основе указанной вами частичной информации об этой строке. Аналогично устроен и предметный указатель в учебнике. Если вас интересует конкретная страница, нужно найти в указателе слова, которые должны на ней быть. Затем перейти непосредственно к странице/страницам, содержащим все искомые слова.

«Дальние родственники» LSA

Два алгоритма очень напоминают LSA и имеют схожее применение в NLP:

- линейный дискриминантный анализ (LDA);
- латентное размещение Дирихле (LDiA)[1].

LDA классифицирует документы только по одной теме. LDiA ближе к LSA, поскольку способно классифицировать документы по любому нужному числу тем.

ПРИМЕЧАНИЕ

В силу одномерности для LDA не требуется сингулярное разложение (singular value decomposition, SVD). Можно просто вычислить центроиды (среднее значение или математическое ожидание) всех векторов TF-IDF для каждой из половин бинарного класса (например, спам или не спам). После этого ваше измерение становится прямой, соединяющей эти центроиды. Чем дальше вектор TF-IDF находится вдоль этой прямой (чем больше скалярное произведение вектора TF-IDF и прямой), тем ближе вы к тому или иному классу.

Сначала покажем пример использования этого простого подхода LDA к анализу тем, чтобы разогреться перед LSA и LDiA.

4.1.5. LDA-классификатор

LDA — одна из самых простых и быстродействующих моделей понижения размерности и классификации. Скорее всего, эта книга — практически единственное место, где можно о ней прочитать, поскольку данный метод не слишком «раскручен»[2]. Но во

---

[1] Мы будем использовать акроним LDiA для латентного размещения Дирихле (latent Dirichlet allocation). Наверное, Панупонг Пасупат (Panupong Pasupat) бы это одобрил. Панупонг преподавал онлайн-курс по NLP, посвященный LDiA (https://ppasupat.github.io/a9online/1140.html#latent-dirichlet-allocation-lda-).

[2] Упоминания его в статьях можно найти еще в 1990-х, когда приходилось экономно расходовать вычислительные и прочие ресурсы (https://www.researchgate.net/profile/Georges_Hebrail/publication/221299406_Automatic_Document_Classification_Natural_Language_Processing_Statistical_Analysis_and_Expert_System_Techniques_used_together/links/0c960516cf4968b29e000000.pdf).

многих приложениях его точность намного выше, чем у более навороченных современных алгоритмов, опубликованных в недавних статьях. LDA-классификатор — алгоритм обучения с учителем, так что нам понадобятся метки для классов документов. Но число требуемых для LDA выборок намного меньше, чем для упомянутых навороченных алгоритмов.

В этом примере мы приведем упрощенную реализацию LDA, которой нет в библиотеке `scikit-learn`. Обучение модели состоит лишь из трех шагов, так что их можно произвести непосредственно на Python.

- Вычислить среднее местоположение (центроид) всех векторов TF-IDF, входящих в класс (например, являющихся спамом СМС).
- Вычислить среднее местоположение (центроид) всех векторов TF-IDF, не входящих в класс (например, не являющихся спамом СМС).
- Вычислить разность векторов между центроидами (связывающую их напрямую).

Для обучения модели LDA достаточно найти вектор (прямую) между центроидами бинарного класса. LDA — алгоритм машинного обучения с учителем, так что для сообщений нужны метки. Для выполнения *вывода* (предсказания) с помощью этой модели достаточно выяснить, ближе ли новый вектор TF-IDF к центроиду векторов, входящих в класс (спам), или к центроиду не входящих (не спам). Сначала обучим LDA-модель классифицировать СМС как спам или не спам (листинг 4.1).

Листинг 4.1. Набор данных СМС-спама

```
>>> import pandas as pd
>>> from nlpia.data.loaders import get_data
>>> pd.options.display.width = 120
>>> sms = get_data('sms-spam')
>>> index = ['sms{}{}'.format(i, '!'*j) for (i,j) in\
...     zip(range(len(sms)), sms.spam)]
>>> sms = pd.DataFrame(sms.values, columns=sms.columns, index=index)
>>> sms['spam'] = sms.spam.astype(int)
>>> len(sms)
4837
>>> sms.spam.sum()
638
>>> sms.head(6)
      spam                                                text
sms0     0  Go until jurong point, crazy.. Available only ...
sms1     0  Ok lar... Joking wif u oni...
sms2!    1  Free entry in 2 a wkly comp to win FA Cup fina...
sms3     0  U dun say so early hor... U c already then say...
sms4     0  Nah I don't think he goes to usf, he lives aro...
sms5!    1  FreeMsg Hey there darling it's been 3 week's n...
```

Эта строка служит для красивого отображения широкого столбца SMS-текста в выводе DataFrame библиотеки Pandas

Просто для отображения. Спамовые сообщения отмечаются добавлением в начало их метки восклицательного знака (!)

Итак, у нас 4837 сообщений, 638 из которых были маркированы меткой бинарного класса `spam`.

Теперь выполним токенизацию и преобразование векторов TF-IDF для всех этих СМС:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from nltk.tokenize.casual import casual_tokenize
>>> tfidf_model = TfidfVectorizer(tokenizer=casual_tokenize)
>>> tfidf_docs = tfidf_model.fit_transform(\
...    raw_documents=sms.text).toarray()
>>> tfidf_docs.shape
(4837, 9232)
>>> sms.spam.sum()
638
```

В результате применения `nltk.casual_tokenizer` в нашем словаре оказывается 9232 слова — почти вдвое больше, чем сообщений. И почти в десять раз больше слов, чем спамовых сообщений. Так что у модели не так много информации о словах, которая бы указывала, является сообщение спамом или нет. Обычно наивный байесовский классификатор работает плохо, если словарь намного больше числа маркированных примеров в наборе данных. Здесь помогут методы семантического анализа, изложенные в этой главе.

Начнем с простейшего метода семантического анализа — LDA. Можно воспользоваться моделью LDA из `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`. Но для обучения модели достаточно вычислить центроиды бинарного класса (спам и не спам), так что можно сделать это вручную:

```
>>> mask = sms.spam.astype(bool).values
>>> spam_centroid = tfidf_docs[mask].mean(axis=0)
>>> ham_centroid = tfidf_docs[~mask].mean(axis=0)

>>> spam_centroid.round(2)
array([0.06, 0.  , 0. , ..., 0. , 0. , 0. ])
>>> ham_centroid.round(2)
array([0.02, 0.01, 0. , ..., 0. , 0. , 0. ])
```

С помощью этой маски можно выбрать только спамовые строки из объекта numpy.array или pandas.DataFrame

Поскольку наши векторы TF-IDF представляют собой векторы-строки, необходимо гарантировать, что NumPy вычисляет средние значения столбцов независимо, с помощью аргумента axis=0

Теперь можно вычесть один центроид из другого и получить прямую между ними:

```
>>> spamminess_score = tfidf_docs.dot(spam_centroid - ham_centroid)
>>> spamminess_score.round(2)
array([-0.01, -0.02, 0.04, ..., -0.01, -0. , 0. ])
```

Скалярное произведение вычисляет проекцию («тень») вектора на соединяющую центроиды прямую

Этот spamminess_score представляет собой расстояние вдоль прямой, соединяющей ham_centroid и spam_centroid. Мы вычислили эту оценку путем проекции всех векторов TF-IDF на соединяющую центроиды прямую с помощью скалярного произведения. Причем вычислили все эти 4837 скалярных произведений за один раз векторизованной операцией NumPy. Это дает порой 100-кратный прирост быстродействия по сравнению с циклом Python.

На рис. 4.2 приведено трехмерное представление векторов TF-IDF и расположение центроидов для СМС.

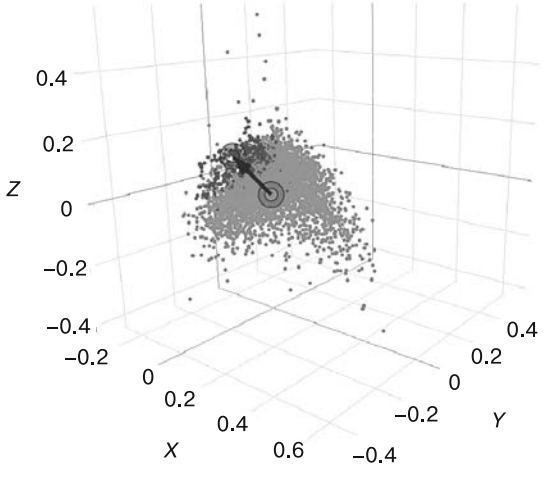


Рис. 4.2. Трехмерная диаграмма рассеяния (облако точек) векторов TF-IDF

Определяющей для нашей обученной модели является прямая, указанная на рис. 4.2 стрелкой от центроида не спама к центроиду спама. Как видите, часть зеленых точек находится сзади стрелки, так что при проектировании их на прямую между центроидами может получиться отрицательная оценка спамовости.

Хотелось бы, чтобы наша оценка находилась в диапазоне от 0 до 1, как вероятность. Этого можно добиться с помощью sklearnMinMaxScaler:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> sms['lda_score'] = MinMaxScaler().fit_transform(\
...     spamminess_score.reshape(-1,1))
>>> sms['lda_predict'] = (sms.lda_score > .5).astype(int)
>>> sms['spam lda_predict  lda_score'.split()].round(2).head(6)
        spam  lda_predict  lda_score
sms0       0            0       0.23
sms1       0            0       0.18
sms2!      1            1       0.72
sms3       0            0       0.18
sms4       0            0       0.29
sms5!      1            1       0.55
```

Выглядит хорошо. Все шесть первых сообщений были классифицированы правильно при пороговом значении 50 %. Посмотрим, что получится на остатке тренировочного набора данных:

```
>>> (1. - (sms.spam - sms.lda_predict).abs().sum() / len(sms)).round(3)
0.977
```

Ух ты! С помощью этой простой модели удалось правильно классифицировать 97.7 % сообщений. На практике вряд ли удастся добиться такого результата, поскольку здесь мы не выделили тестовый набор данных. Модель получила отметку «5 с плюсом» на уже виденных классификатором «экзаменационных вопросах». Но LDA — очень простая модель с небольшим числом параметров, так что она должна легко поддаваться обобщению, если ваши СМС будут соответствовать тем сообщениям, которые планируется классифицировать. Запустите ее на ваших собственных примерах. Или, еще лучше, загляните в приложение Д и узнайте, как произвести так называемую кросс-валидацию.

Таковы возможности методов семантического анализа. В отличие от наивного байесовского классификатора и модели логистической регрессии семантический анализ не основывается на отдельных словах[1]. Он собирает слова со схожей семантикой (например, спамовостью) и использует их вместе. Помните, что словарь данного тренировочного набора ограничен и включает некоторые не английские слова. Так что для правильной классификации ваши контрольные сообщения должны использовать подобный же словарный запас.

Взглянем, как выглядит матрица различий для тренировочного набора данных. Она демонстрирует, сколько отмеченных как спам СМС на самом деле им не были (ложноположительные результаты) и сколько помеченных как не спам следовало пометить как спам (ложноотрицательные результаты):

```
>>> from pugnlp.stats import Confusion
>>> Confusion(sms['spam lda_predict'.split()])
lda_predict     0    1
spam
0            4135   64
1              45  593
```

Выглядит неплохо. Если ложноположительных (64) или ложноотрицательных (45) результатов непропорционально много, можно изменить пороговое значение **`0.5`**. Теперь вы подготовлены к рассказу о моделях, умеющих вычислять *многомерные* семантические векторы вместо одномерных семантических показателей. До сих пор наши одномерные векторы понимали только, является слово/документ спамом или нет. Мы обучим их понимать гораздо больше нюансов слов и предоставим вам многомерный вектор, захватывающий смысл слова.

Прежде чем углубиться в SVD (математические основы многомерного LSA), мы упомянем еще несколько подходов.

---

[1] Наивный байесовский классификатор и модель логистической регрессии эквивалентны этой простой модели LDA. Если вам интересно, можете углубиться в их математические основы и код sklearn.

Еще один «дальний родственник»

Как уже говорилось, у LSA есть еще один «дальний родственник» с похожей аббревиатурой — LDiA. Расшифровывается как *латентное размещение Дирихле* (latent Dirichlet allocation)[1]. Можно также применять для генерации векторов, захватывающих семантику слова или документа.

LDiA применяет математику LSA по-другому. Для группировки слов в нем используется нелинейный статистический алгоритм. В результате обучение занимает намного больше времени, чем при линейных подходах, таких как LSA. Из-за этого LDiA часто оказывается менее подходящим для многих реальных приложений, и его редко имеет смысл пробовать первым. Но создаваемые им сводные показатели тем иногда точнее отражают интуитивные представления человека относительно слов и тем. Так что полученные с помощью LDiA темы будет проще объяснить своему начальнику.

LDiA полезен для решения некоторых однодокументных задач, например для автоматического реферирования документов. Документ играет роль корпуса, а предложения в этом корпусе — документов. Именно так gensim и другие пакеты используют LDiA для обнаружения наиболее важных предложений документа. Далее путем соединения этих предложений можно создать машинно-генерируемый реферат документа[2].

Для большинства задач классификации и регрессии лучше подойдет LSA. Так что мы расскажем сначала про LSA и лежащую в его основе линейную алгебру SVD.

4.2. Латентно-семантический анализ

В основе латентно-семантического анализа лежит самая древняя и популярная методика понижения размерности — сингулярное разложение. Метод SVD широко использовался задолго до появления термина «машинное обучение»[3]. SVD разлагает матрицу на три квадратные матрицы, одна из которых — диагональная.

Одна из сфер применения SVD — обращение матрицы. Найти обратную заданной матрицу можно, разложив ее на три более простые квадратные матрицы, транспонировав их, а затем снова перемножив. Легко можно представить возможности применения алгоритма, позволяющего обратить большую сложную матрицу.

---

[1] Мы выбрали нестандартный акроним LDiA, чтобы отличать его от акронима LDA, который обычно обозначает линейный дискриминантный анализ, но не всегда. В этой книге вам не придется гадать, что мы понимаем под этим акронимом в каждом конкретном случае. LDA всегда будет обозначать линейный дискриминантный анализ, LDiA — латентное размещение Дирихле.

[2] Часть текста из раздела «Об этой книге» была сгенерирована с помощью подобной математики, но реализованной в виде нейронной сети (см. главу 12).

[3] Замечательный способ исследовать историю слов и понятий — Google Ngram Viewer (http://mng.bz/qJEA).

SVD хорошо подходит для задач машиностроения, например для анализа нагрузок ферменных конструкций и расчета деформаций. И даже используется в науке о данных для поведенческих рекомендательных систем, работающих параллельно с рекомендательными системами NLP на основе контента.

С помощью SVD алгоритм LSA может разбить матрицу «терм — документ» на три более простые матрицы, которые затем можно умножить снова и получить исходную без каких-либо изменений. Это напоминает разложение большого числа на множители. Но эти три более простые матрицы из SVD раскрывают свойства исходной матрицы TF-IDF. Они подходят для ее упрощения. Можно усечь матрицы (отбросить часть строк и столбцов), прежде чем их снова перемножить, что уменьшит число измерений, с которыми придется иметь дело в модели векторного пространства.

В результате перемножения этих усеченных матриц вы получаете не ту же самую матрицу TF-IDF, а улучшенную. Наше новое представление документов содержит самую суть, латентную семантику данных документов. Именно поэтому SVD применяется и в других сферах, например при сжатии. Оно улавливает саму суть набора данных и игнорирует шум. JPEG-изображение сохраняет всю информацию из исходного растрового изображения, хотя меньше его в десять раз.

При подобном использовании SVD для обработки естественного языка он называется латентно-семантическим анализом. LSA раскрывает скрытую семантику (смысл) слов.

Латентно-семантический анализ — математический метод поиска наилучшего способа линейного преобразования (поворота и растяжения) любого набора NLP векторов, например векторов TF-IDF или векторов мультимножества. Лучший способ для многих приложений — выровнять оси (измерения) в новых векторах по наибольшему разбросу (дисперсии) частотностей слов[1]. Затем можно исключить из нового векторного пространства те измерения, которые мало влияют на изменения векторов от документа к документу.

Подобное использование SVD называется *усеченным сингулярным разложением* (truncated singular value decomposition, усеченное SVD). В сфере обработки и сжатия изображений его называют *методом главных компонент* (principal component analysis, PCA). Мы продемонстрируем несколько хитростей для повышения точности векторов LSA. Эти трюки полезны также при выполнении PCA в задачах машинного обучения и проектирования признаков в других сферах.

Если вы слушали курс линейной алгебры, то сталкивались с лежащей в основе LSA математикой в виде сингулярного разложения. Если же вам случалось производить машинное обучение на изображениях или других многомерных данных, скажем на временных рядах, то вы применяли PCA к этим многомерным векторам. Использование LSA к документам на естественном языке эквивалентно применению PCA к векторам TF-IDF.

---

[1] Замечательные иллюстрации и пояснения можно найти в главе 16 учебника по NLP Д. Журафски и Дж. Мартина по адресу https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf#chapter.16.

LSA с помощью SVD ищет сочетания слов, отвечающих совместно за наибольший разброс значений данных. Векторы TF-IDF можно повернуть так, чтобы их новые измерения (базисные векторы) располагались по одной линии с этими направлениями максимальной дисперсии. Базисные векторы представляют собой оси координат нового векторного пространства и аналогичны трем шестимерным векторам тем из мысленного эксперимента в начале данной главы. Каждое из измерений (осей координат) превращается скорее в сочетание частотностей слов, чем в частотность отдельного слова. Так что их можно рассматривать как взвешенные сочетания слов, из которых состоят различные используемые в корпусе темы.

Компьютер понимает не смысл сочетаний слов, а то, что они встречаются вместе. Если он часто встречает рядом *dog*, *cat* и *love*, то помещает их в одну тему. Он не знает, что эта тема, вероятно, посвящена домашним животным. Он может отнести к той же теме множество слов с противоположным смыслом, вроде *domesticated* (одомашненный) и *feral* (дикий). Если они часто появляются в одном документе, то получат вместе в LSA высокие оценки для одних тем. Только человек может понять, у каких слов большой вес в каждой из тем, и назвать эти темы.

Но не обязательно давать теме название, чтобы использовать ее. Подобно тому как мы не анализировали все тысячи измерений векторов стеммированных мультимножеств или векторов TF-IDF из предыдущих глав, так и не должны знать, что означают все наши темы. Это не помешает выполнять векторную арифметику над новыми векторами тем, аналогично тому, как мы делали с векторами TF-IDF. Их можно складывать, вычитать и оценивать сходство документов на основе векторов тем, а не только частотностей слов.

Метод LSA обеспечивает и еще кое-какую полезную информацию. Подобно части IDF в TF-IDF, он дает информацию о том, какие измерения вектора играют важную роль для семантики (смысла) документов. Измерения (темы) с наименьшим изменением векторов от документа к документу можно отбросить. Такие темы с низкой дисперсией обычно представляют собой шум и отвлекают внимание любого алгоритма машинного обучения. Если какая-либо тема присутствует во всех документах примерно в одинаковом объеме и не несет никакой пользы для различения документов, от нее можно избавиться, что позволит обобщить векторное представление, сделав его более подходящим для еще не встречавшихся конвейеру документов, даже из другого контекста.

Такие производимые LSA обобщение и сжатие позволяют добиться того, что мы пытались сделать в главе 2, когда игнорировали стоп-слова. Но понижение размерности LSA работает гораздо лучше, поскольку оптимально. Оно сохраняет максимально возможный объем информации, отбрасывая не слова, а измерения (темы).

LSA «втискивает» больше смысла в меньшее число измерений. Необходимо оставить только измерения с большой дисперсией, то есть основные темы, которые обсуждаются в корпусе разными способами (с большой дисперсией). И все эти измерения становятся темой, в каждой из которых оказывается захвачено какое-то взвешенное сочетание всех слов.

4.2.1. Воплощаем мысленный эксперимент на практике

Воспользуемся алгоритмом для вычисления каких-нибудь тем, например `animalness`, `petness` и `cityness` из нашего мысленного эксперимента. Мы не можем сообщить алгоритму LSA, к чему относятся эти темы[1]. Просто попробуем и посмотрим, что получится. Захват семантики коротких документов вроде твитов, сообщений в чате или строк стихотворений из маленького корпуса требует всего нескольких измерений (тем) (листинг 4.2).

Листинг 4.2. Матрица тем-слов для LSA на основе 16 коротких предложений о котах, собаках и NYC

```
>>> from nlpia.book.examples.ch04_catdog_lsa_3x6x16\
...     import word_topic_vectors
>>> word_topic_vectors.T.round(1)
      cat  dog  apple  lion  nyc  love
top0 -0.6 -0.4    0.5  -0.3  0.4  -0.1
top1 -0.1 -0.3   -0.4  -0.1  0.1   0.8
top2 -0.3  0.8   -0.1  -0.5  0.0   0.1
```

Строки в этой матрице тем-слов представляют собой векторы тем-слов или просто векторы тем для всех слов. Они аналогичны оценкам слов из модели анализа тональностей в главе 2. Эти векторы можно использовать для представления смысла слов в любом конвейере машинного обучения; иногда их называют семантическими векторами, а путем сложения векторов тем для отдельных слов можно вычислить вектор темы для всего документа.

Созданные SVD векторы тем аналогичны придуманным нами в мысленном эксперименте. Первая тема, маркированная `top0`, немного похожа на вышеупомянутую тему `cityness`. Веса для `apple` и `NYC` в теме `top0` больше. Но `top0` идет первой в LSA и последней в списке наших придуманных тем. LSA сортирует темы по важности, то есть объему представляемой ими информации или дисперсии в наборе данных. Измерение `topic0` расположено вдоль осей наибольшей дисперсии в наборе данных. Высокая дисперсия по городам заметна при наблюдении нескольких предложений о `NYC` и `apple` и нескольких, которые вообще не включают этих слов.

А `top1` выглядит иначе, чем все темы из мысленного эксперимента. Алгоритм LSA обнаружил, что для захвата сущности документов, на которых он был запущен, тема `love` важнее, чем `animalness`. Последняя тема, `top2`, похоже, относится к собакам *(dog)* с небольшой толикой любви *(love)*. `cat` низведено до уровня темы `anti-cityness`, поскольку кошки и города нечасто упоминаются вместе.

---

[1] Существует область исследований, связанная с так называемыми усвоенными метриками (learned metrics), с помощью которых можно «намекнуть» темам, чего они должны касаться. См. статью: *Jain L., Mason B., Nowak R.* Learning Low-Dimensional Metrics. https://papers.nips.cc/paper/7002-learning-low-dimensional-metrics.pdf.

Разобраться лучше в работе LSA (этот алгоритм создает векторы тем, не понимая, что означают слова) нам поможет еще один мысленный эксперимент.

Mad Libs

Догадаетесь ли вы из контекста, что означает *awas* в следующем высказывании: *Awas! Awas! Tom is behind you! Run!*?

Вы не угадаете, что Том — альфа-орангутанг из парка Лики на Борнео. И вы вряд ли знаете, что Том был приучен к людям, но привык защищать свою территорию, иногда излишне агрессивно. И вашему внутреннему обработчику естественного языка может не хватить времени, чтобы полностью осознать смысл *awas*, пока вы будете убегать в безопасное место.

Но когда вы переведете дыхание и подумаете, то можете предположить, что *awas* означает «опасность» или «осторожно» на индонезийском языке. Игнорируя реальный мир и концентрируя внимание на одном только языковом контексте, словах, часто можно перенести значение (смысл) известных вам слов на неизвестные.

Попробуйте это как-нибудь сами или с друзьями. Как и в игре Mad Libs (https://en.wikipedia.org/wiki/Mad_Libs), просто замените слово в предложении соответствующим иноязычным или даже выдуманным словом. Затем попросите друга угадать, что это слово значит, или заполнить пропуск английским аналогом. Зачастую догадка вашего друга окажется не столь уж далека от правильного перевода иноязычного или предполагаемого значения выдуманного слова.

У машин, начинающих работу с чистого листа, нет знаний языка по умолчанию. Так что им требуется несколько примеров, чтобы понять, что значат слова в них. Это все равно что вы смотрите на предложение, полное иноязычных слов. Но машины неплохо справляются с этой задачей с помощью LSA при наличии только случайной выборки документов, содержащих всего несколько упоминаний интересующих вас слов.

Как видите, более короткие документы, например предложения, подходят лучше статей или книг. Дело в том, что смысл слова обычно тесно связан со значениями других слов в содержащем его предложении, но отнюдь не со значениями далеко стоящих от него слов в пределах длинного документа[1].

LSA — способ обучить машину распознавать смысл (семантику) слов и фраз по примерам их использования. Подобно людям, машины намного быстрее и легче обучаются семантике на примерах применения слов, чем на словарных определениях. Для извлечения смысла из примеров требуется меньше логических рассуждений, чем для чтения всех возможных определений и форм слов из словаря и превращения их в логику.

Математический метод, с помощью которого в LSA раскрывается смысл слов, называется сингулярным разложением. Для создания векторов наподобие тех,

[1] Когда Томаш Миколов задумался об этом при разработке Word2vec, то осознал, что может уточнить смысл векторов слов, если сузит контекст еще больше, ограничив расстояние между словами контекста до пяти.

которые встречались нам только что в матрицах «слово — тема»[1], LSA использует SVD, знакомый вам по курсу линейной алгебры.

Вы увидите NLP в действии: сейчас продемонстрируем, как машина «играет в Mad Libs», понимая значение слов.

4.3. Сингулярное разложение

Именно алгоритм сингулярного разложения лежит в основе LSA. Начнем с корпуса из всего 11 документов и словаря из шести слов, подобного тому, что был в нашем мысленном эксперименте[2]:

```
>>> from nlpia.book.examples.ch04_catdog_lsa_sorted\
...     import lsa_models, prettify_tdm
>>> bow_svd, tfidf_svd = lsa_models()
>>> prettify_tdm(**bow_svd)
```

Эта строка выполняет LSA на корпусе cats_and_dogs с использованием словаря из мысленного эксперимента. Скоро мы заглянем внутрь этого «черного ящика»

	cat	dog	apple	lion	nyc	love	text
0			1		1		NYC is the Big Apple.
1			1		1		NYC is known as the Big Apple.
2					1	1	I love NYC!
3			1		1		I wore a hat to the Big Apple party in NYC.
4			1		1		Come to NYC. See the Big Apple!
5			1				Manhattan is called the Big Apple.
6	1						New York is a big city for a small cat.
7	1			1			The lion, a big cat, is the king of the jungle.
8	1				1		I love my pet cat.
9					1	1	I love New York City (NYC).
10	1	1					Your dog chased my cat.

Полученные результаты представляют собой матрицу «документ — терм», каждая строка которой — это вектор набора слов документа.

Мы ограничили словарь в соответствии с мысленным экспериментом. И ограничили корпус всего 11 документами, использующими шесть слов из нашего словаря. К сожалению, работа алгоритма сортировки и ограниченность словаря привели к появлению нескольких идентичных векторов мультимножеств (`NYC`, `apple`). Но SVD должен обнаружить это и отвести для этой пары слов свою тему.

Сначала мы применим SVD к матрице «терм — документ» (полученной в результате транспонирования вышеприведенной матрицы «документ — терм»), но он может работать и на матрицах TF-IDF или на любой другой модели векторного пространства:

---

[1] Загляните в примеры из nlpia/book/examples/ch04_*.py, если хотите увидеть документы и векторную арифметику, лежащие в основе этого «воплощения» мысленного эксперимента. Этот мысленный эксперимент был проведен до того, как мы воспользовались SVD для настоящих предложений на естественном языке. Нам повезло, что темы полностью совпали.

[2] Мы выбрали 11 коротких предложений, чтобы сэкономить место в печатной версии книги. Вы можете узнать много интересного, если обратитесь к примерам ch04 в nlpia и запустите SVD на больших корпусах.

```
>>> tdm = bow_svd['tdm']
>>> tdm
          0    1    2    3    4    5    6    7    8    9   10
cat       0    0    0    0    0    0    1    1    1    0    1
dog       0    0    0    0    0    0    0    0    0    0    1
apple     1    1    0    1    1    1    0    0    0    0    0
lion      0    0    0    0    0    0    0    1    0    0    0
nyc       1    1    1    1    1    0    0    0    0    1    0
love      0    0    1    0    0    0    0    0    1    1    0
```

SVD — алгоритм для разложения любой матрицы на три множителя, то есть три матрицы, в результате перемножения которых получится исходная матрица. Это аналогично разбиению большого числа на три множителя. Но наши множители не скалярные значения, а настоящие двумерные матрицы с особыми свойствами. У вычисляемых нами с помощью SVD трех матричных множителей есть несколько полезных математических свойств, пригодных для понижения размерности и метода LSA. В курсе линейной алгебры вам приходилось использовать SVD для обращения матрицы. Здесь он пригодится, чтобы LSA определил, какими должны быть темы (группы связанных слов).

При каждом запуске алгоритма SVD для матрицы «терм — документ» BOW или матрицы «терм — документ» TF-IDF SVD находит подходящие друг к другу сочетания слов. Для поиска этих встречающихся рядом слов SVD вычисляет корреляцию между столбцами (термами) матрицы «терм — документ»[1]. SVD одновременно находит корреляцию использования термов в документах и корреляцию документов друг с другом. На основе этой информации SVD также вычисляет линейные комбинации термов с максимальной дисперсией в пределах корпуса. Такие комбинации частотностей термов и станут нашими темами. И мы оставим только темы, вмещающие максимум информации и дисперсии в корпусе. Кроме того, обеспечивается возможность линейных преобразований (поворотов) векторов «терм — документ» для преобразования их в более короткие векторы тем для каждого из документов.

SVD сгруппирует термы с высокой корреляцией (поскольку они часто встречаются в одних и тех же документах), а также варьируются одновременно в наборе документов. Мы будем рассматривать эти линейные сочетания слов как темы, которые превращают наши векторы BOW (или векторы TF-IDF) в векторы тем, указывающие, к каким темам относится документ. Вектор темы представляет собой резюме (обобщение) содержания документа.

Трудно сказать, кому первому пришла в голову идея применения SVD к частотностям слов для создания векторов тем. Несколько лингвистов одновременно работали над схожими методиками. Они все обнаружили, что семантическое подобие двух выражений (или отдельных слов) естественного языка пропорционально подобию контекстов, в которых они используются. В числе этих исследователей

[1] Что эквивалентно квадратному корню из скалярного произведения двух столбцов (векторов встречаемости «терм — документ»), но SVD дает дополнительную информацию по сравнению с непосредственным вычислением корреляции.

З. С. Харрис (1951)[1], М. Коль (1979)[2], Исбел (1998)[3], Дюмей (1988)[4], Сэлтон, Леск (1965)[5] и Дирвестер (1990)[6].

На математическом языке SVD (основа LSA) выглядит следующим образом:

$$W_{mxn} \rightarrow U_{mxp} S_{pxp} V_{pxn}^{T}$$

где: m — число термов в словаре, n — количество документов в корпусе, а p — сумма тем в корпусе, равная числу слов. Погодите, разве мы не хотели уменьшить число измерений? Нам нужно, чтобы тем в итоге было меньше, чем слов, так что можно воспользоваться этими векторами тем (строками матрицы «терм — документ») в качестве понижающего размерность представления исходных векторов TF-IDF. В итоге мы добьемся этого. Но для начала оставим на месте все измерения матриц.

4.3.1. U — левые сингулярные векторы

U-матрица содержит матрицу «терм — тема», указывающую, какие «друзья есть у слова»[7]. Ее называют матрицей левых сингулярных векторов, поскольку она содержит векторы-строки, которые необходимо умножить слева на матрицу векторов-столбцов[8]. *U*-матрица отражает взаимную корреляцию слов и тем на основе совместной встречаемости слов в одном документе. До начала усечения (удаления столбцов) она является квадратной. Количество столбцов и строк в ней равно числу

---

1 Журафски и Шен цитируют «Методы структурной лингвистики» З. С. Харриса (1951) в своей статье 2000 г. Knowledge-Free Induction of Morphology Using Latent Semantic Analysis (https://dl.acm.org/ft_gateway.cfm?id=1117615&ftid=570935&dwn=1&#URLTOKEN), а также в презентации по адресу https://slidegur.com/doc/3928417/knowledge-free-induction-of-morphology-using-latent.

2 *Koll M.* Generalized vector spaces model in information retrieval: https://dl.acm.org/citation.cfm?id=253506; Approach to Concept Based Information Retrieval, 1979.

3 *Isbell C. L. Jr.* Restructuring Sparse High-Dimensional Data for Effective Retrieval: http://papers.nips.cc/paper/1597-restructuring-sparse-high-dimensional-data-for-effective-retrieval.pdf, 1998.

4 *Dumais et al.* Using latent semantic analysis to improve access to textual information: https://dl.acm.org/citation.cfm?id=57214, 1988.

5 *Salton G.* The SMART automatic document retrieval system, 1965.

6 *Deerwester S. et al.* Indexing by Latent Semantic.

7 Если попробовать воспроизвести эти результаты с помощью модели PCA из sklearn, окажется, что эта матрица «терм — тема» получается из матрицы VT, поскольку входной набор данных транспонирован по сравнению с тем, что мы здесь делаем. Scikit-Learn размещает данные в виде векторов-строк, так что наша матрица «терм — документ» в tdm транспонируется в матрицу «документ — терм» при вызове PCA.fit() или любого другого способа обучения модели sklearn.

8 Математики называют такие векторы левыми собственными векторами или собственными векторами-строками. См. https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors#Left_and_right_eigenvectors.

слов в нашем словаре (m): шесть. У нас все еще шесть тем (p), поскольку мы пока не усекали эту матрицу (листинг 4.3).

Листинг 4.3. Матрица U_{mxp}

```
>>> import numpy as np
>>> U, s, Vt = np.linalg.svd(tdm)
>>> import pandas as pd
>>> pd.DataFrame(U, index=tdm.index).round(2)
          0     1     2     3     4     5
cat   -0.04  0.83 -0.38 -0.00  0.11 -0.38
dog   -0.00  0.21 -0.18 -0.71 -0.39  0.52
apple -0.62 -0.21 -0.51  0.00  0.49  0.27
lion  -0.00  0.21 -0.18  0.71 -0.39  0.52
nyc   -0.75 -0.00  0.24 -0.00 -0.52 -0.32
love  -0.22  0.42  0.69  0.00  0.41  0.37
```

Мы переиспользуем матрицу «терм — документ» из вышеприведенных фрагментов кода

Обратите внимание, что алгоритм SVD — обычная математическая операция NumPy, а не экзотический алгоритм машинного обучения Scikit-Learn.

U-матрица включает в виде столбцов векторы тем для всех слов корпуса. Это значит, что ее можно использовать для преобразования вектора «слово — документ» (вектора TF-IDF или вектора BOW) в вектор «тема — документ». Для получения нового вектора «тема — документ» необходимо просто умножить матрицу *U* «тема — слово» на любой вектор «слово — документ». Дело в том, что веса (оценки) в ячейках *U*-матрицы отражают важность слов для тем. Именно это мы и делали в мысленном эксперименте, с которого началась вся наша затея с кошками и собаками в Нью-Йорке.

4.3.2. S — сингулярные значения

Квадратная диагональная *S*-матрица (сигма-матрица) содержит сингулярные значения темы[1]. Эти сингулярные значения указывают, сколько информации захвачено каждым из измерений нашего нового семантического (тематического) векторного пространства. В диагональной матрице ненулевые значения располагаются только на диагонали от верхнего левого угла до нижнего правого. На всех остальных позициях в *S*-матрице находятся нули. Так что NumPy экономит место за счет возврата сингулярных значений в виде массива, который легко можно преобразовать в диагональную матрицу с помощью функции `numpy.diag`, как показано в листинге 4.4.

Листинг 4.4. Матрица S_{pxp}

```
>>> s.round(1)
np.array([3.1, 2.2, 1.8, 1. , 0.8, 0.5])
>>> S = np.zeros((len(U), len(Vt)))
>>> pd.np.fill_diagonal(S, s)
>>> pd.DataFrame(S).round(1)
     0    1    2    3    4    5    6    7    8    9    10
0   3.1  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

[1] Математики называют их собственными значениями.

```
1   0.0  2.2  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
2   0.0  0.0  1.8  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
3   0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
4   0.0  0.0  0.0  0.0  0.8  0.0  0.0  0.0  0.0  0.0  0.0
5   0.0  0.0  0.0  0.0  0.0  0.5  0.0  0.0  0.0  0.0  0.0
```

Подобно *U*-матрице, *S*-матрица для нашего корпуса из шести слов и шести тем состоит из шести строк (`p`). Но в ней намного больше столбцов (`n`), заполненных нулями. В ней должно быть по столбцу для каждого документа, чтобы ее можно было умножить на V^T, матрицу «документ — документ», о которой мы расскажем далее. Поскольку мы пока не понизили размерность с помощью усечения этой диагональной матрицы, число тем (`p`) равно числу термов в словаре (`m`) — шести. Причем наши измерения (темы) устроены так, что первое измерение содержит больше всего информации (объяснимой дисперсии) о корпусе. Таким образом, при необходимости усечения тематической модели можно начать с измерений справа снизу и двигаться влево. И прекратить обнуление этих сингулярных значений тогда, когда погрешность тематической модели будет существенно влиять на общую погрешность конвейера NLP.

СОВЕТ

А вот и упоминавшаяся выше хитрость. При работе с NLP, да и с большинством других приложений, не требуется хранить информацию о дисперсии в тематической модели. Документы, которые будут обрабатываться далее, могут быть посвящены совсем другим темам.

В большинстве случаев лучше задать диагональные элементы *S*-матрицы равными единице, получив в результате прямоугольную единичную матрицу, которая просто меняет форму матрицы «документ — документ» V^T так, чтобы она подходила к *U*-матрице «слово — тема». Таким образом, умножение *S*-матрицы на какой-либо новый набор векторов документов не перекосит векторы тем в сторону исходного набора (распределения) тем.

4.3.3. V^T — правые сингулярные векторы

V^T-матрица «документ — документ» содержит в качестве столбцов правые сингулярные векторы и дает информацию о разделяемых документами смыслах, поскольку измеряет частоту использования в документах одинаковых тем, в нашей новой семантической модели документов. Она содержит столько же строк (`p`) и столбцов, сколько документов в нашем маленьком корпусе, — 11 (листинг 4.5).

Листинг 4.5. V_{pxn}^T

```
>>> pd.DataFrame(Vt).round(2)
       0     1     2     3     4     5     6     7     8     9     10
0   -0.44 -0.44 -0.31 -0.44 -0.44 -0.20 -0.01 -0.01 -0.08 -0.31 -0.01
1   -0.09 -0.09  0.19 -0.09 -0.09 -0.09  0.37  0.47  0.56  0.19  0.47
2   -0.16 -0.16  0.52 -0.16 -0.16 -0.29 -0.22 -0.32  0.17  0.52 -0.32
3   0.00  -0.00 -0.00  0.00  0.00  0.00 -0.00  0.71  0.00 -0.00 -0.71
```

```
4   -0.04 -0.04 -0.14 -0.04 -0.04  0.58  0.13 -0.33  0.62 -0.14 -0.33
5   -0.09 -0.09  0.10 -0.09 -0.09  0.51 -0.73  0.27 -0.01  0.10  0.27
6   -0.57  0.21  0.11  0.33 -0.31  0.34  0.34 -0.00 -0.34  0.23  0.00
7   -0.32  0.47  0.25 -0.63  0.41  0.07  0.07  0.00 -0.07 -0.18  0.00
8   -0.50  0.29 -0.20  0.41  0.16 -0.37 -0.37 -0.00  0.37 -0.17  0.00
9   -0.15 -0.15 -0.59 -0.15  0.42  0.04  0.04 -0.00 -0.04  0.63 -0.00
10  -0.26 -0.62  0.33  0.24  0.54  0.09  0.09 -0.00 -0.09 -0.23 -0.00
```

Как и *S*-матрицу, мы будем игнорировать V^T-матрицу при преобразовании новых векторов «слово — документ» в векторное пространство тем. Мы будем использовать ее только для проверки точности векторов тем и восстановления исходных векторов «слово — документ», применявшихся для ее обучения.

4.3.4 Ориентация SVD-матрицы

Если вам случалось выполнять машинное обучение по документам на естественном языке, то вы могли заметить, что наша матрица «терм — документ» повернута набок (транспонирована) по сравнению с привычной вам по Scikit-Learn и другим пакетам. В модели тональностей наивного байесовского классификатора в конце главы 2 и при работе с векторами TF-IDF из главы 3 тренировочные наборы данных создавались в виде матрицы «документ — терм». Именно такой ориентации матрицы требуют модели Scikit-Learn. Каждая строка тренировочного набора данных в матрице «выборка — признак» для выборки машинного обучения соответствует документу. Каждый столбец отражает слово или признак этих документов. Но при выполнении операций линейной алгебры SVD напрямую матрицу нужно транспонировать в формат матрицы «терм — документ»[1].

ВАЖНО

> Мы называем матрицы и указываем их размеры сначала по строкам, а затем по столбцам. Так, «терм — документ» — это матрица, где строки представляют собой слова, а столбцы — документы. Аналогично и с измерениями (размерами) матриц. Матрица 2×3 состоит из двух строк и трех столбцов, то есть np.shape() у нее (2, 3), а len() — 2.

Не забудьте транспонировать свои матрицы «терм — документ» или «тема — документ» обратно в ориентацию Scikit-Learn перед обучением модели машинного обучения. В Scikit-Learn каждая из строк в тренировочном наборе данных NLP должна содержать вектор связанных с документом признаков (электронную почту, СМС, предложение, веб-страницу или любой другой фрагмент текста). В тренировочных наборах данных NLP векторы представляют собой векторы-строки, а при обычных операциях линейной алгебры векторы обычно рассматриваются как векторы-столбцы.

---

[1] В модели sklearn.PCA матрица «документ — терм» остается нетранспонированной, а транспонируются матричные операции SVD. Так что в модели PCA из Scikit-Learn игнорируются U- и S-матрица. Для преобразования новых векторов-строк «документ — терм» в векторы-строки «документ — тема» используется только V^T-матрица.

В следующем разделе мы подробно рассмотрим все это, когда будем обучать преобразователь TruncatedSVD из библиотеки scikit-learn превращать векторы наборов слов в векторы «документ — тема». Затем мы транспонируем эти векторы обратно для создания строк тренировочного набора данных, чтобы обучить на этих векторах «документ — тема» классификатор Scikit-Learn (sklearn).

ВНИМАНИЕ

При использовании Scikit-Learn необходимо транспонировать матрицу «признак — документ» (обычно называемую в Scikit-Learn X), чтобы создать матрицу документ-признак, передаваемую в методы .fit() и .predict() модели. Каждая строка в тренировочном наборе данных должна быть вектором признаков для конкретной выборки текста, обычно документа[1].

4.3.5. Усечение тем

У нас теперь есть тематическая модель — способ преобразования векторов частотностей слов в векторы весов тем. Но поскольку число тем равно числу слов, то количество измерений в модели нашего векторного пространства аналогично исходным векторам BOW. Мы просто создали новые слова и назвали их темами, так как в них слова сочетаются в различных соотношениях. Мы еще не понизили число размерностей.

S-матрицу можно проигнорировать, поскольку столбцы и строки нашей *U*-матрицы уже упорядочены так, что наиболее важные темы (с наибольшими сингулярными значениями) находятся слева. Еще одна причина для игнорирования *S*-матрицы — большинство векторов «слово — документ», которые могут пригодиться для этой модели, например векторы TF-IDF, уже нормализованы. При этом получаются лучшие тематические модели[2].

Начнем отрезать столбцы с правой стороны *U*-матрицы. Какого числа тем достаточно для захвата сути документа? Один из способов измерить точность LSA — проверить, насколько точно мы можем воссоздать матрицу «терм — документ» по матрице «тема — документ». В листинге 4.6 мы построим график точности восстановления для матрицы из 9 термов и 11 документов, использовавшейся ранее для демонстрации SVD.

Листинг 4.6. Погрешность восстановления матрицы «терм — документ»

```
>>> err = []
>>> for numdim in range(len(s), 0, -1):
...     S[numdim - 1, numdim - 1] = 0
...     reconstructed_tdm = U.dot(S).dot(Vt)
...     err.append(np.sqrt(((\
```

[1] См. посвященный LSA раздел документации Scikit-Learn по адресу http://scikit-learn.org/stable/modules/decomposition.html#lsa.

[2] *Levy, Goldberg, Dagan.* Improving Distributional Similarity with Lessons Learned from Word Embeddings, 2015.

```
...         reconstructed_tdm - tdm).values.flatten() ** 2).sum()
...         / np.product(tdm.shape)))
>>> np.array(err).round(2)
array([0.06, 0.12, 0.17, 0.28, 0.39, 0.55])
```

При восстановлении матрицы «терм — документ» для наших 11 документов с помощью сингулярных векторов чем больше мы отбрасываем, тем выше погрешность. У вышеописанной модели с тремя темами была бы около 28 %, если бы мы воспользовались ею, чтобы восстановить векторы BOW для всех документов. На рис. 4.3 приведен график падения точности по мере отбрасывания все большего числа измерений в тематической модели.

Графики падения точности при использовании для модели векторов TF-IDF или BOW очень близки. Но первые покажут несколько лучшие результаты, если планируется оставить в модели всего несколько тем.

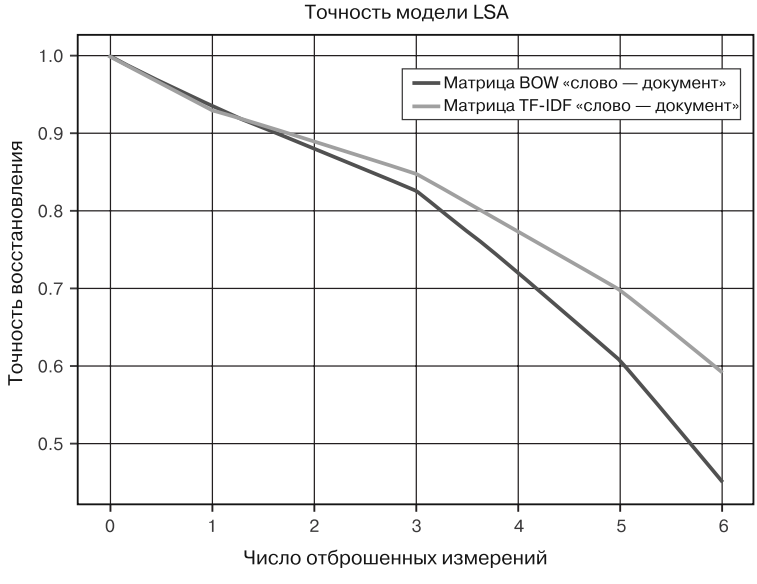


Рис. 4.3. Точность восстановления матрицы «терм — документ» падает по мере игнорирования все большего числа измерений

Это очень простой пример, но из него видно, как подобный график может помочь выбрать число тем (измерений) в модели. В некоторых случаях оказывается, что точность остается идеальной после исключения нескольких измерений из матрицы «терм — документ». Угадайте почему?

Лежащий в основе LSA алгоритм SVD замечает, если слова всегда встречаются рядом, и помещает их вместе в тему. Благодаря этому он может уменьшить число измерений без дополнительных усилий. Даже если вы не собираетесь использовать тематическую модель в конвейере, LSA (SVD) — отличный способ сжатия матриц «слово — документ» и обнаружения для конвейера потенциальных составных слов или *n*-грамм.

4.4. Метод главных компонент

Метод главных компонент — другое название SVD в случае его использования для понижения размерности, как мы делали при семантическом анализе выше. Модель PCA в Scikit-Learn включает несколько модификаций математики SVD, повышающих точность конвейера NLP.

`sklearn.PCA` автоматически «центрирует» данные, вычитая средние значения частотностей слов. Более тонкая уловка: использование в PCA функции `flip_sign` для вычисления детерминистичным образом знаков сингулярных векторов[1].

Реализация PCA из **`sklearn`** включает необязательный шаг «отбеливания». Оно напоминает уловку с игнорированием сингулярных значений при преобразовании векторов «слово — документ» в векторы «тема — документ». Вместо того чтобы просто делать все сингулярные значения в *S*-матрице равными единице, «отбеливание» делит данные на эти дисперсии аналогично преобразованию `sklearn.StandardScaler`. Это повышает «размазанность» данных и снижает вероятность того, что алгоритм оптимизации «потеряется» в «дренажных желобках» и «реках» ваших данных, возникающих при взаимной корреляции признаков в наборе[2].

Прежде чем применить PCA к настоящим многомерным данным NLP, давайте отступим на шаг и посмотрим на более наглядное описание работы PCA и SVD. Это поможет вам также разобраться в API реализации PCA библиотеки Scikit-Learn. PCA удобен для широкого спектра приложений, так что это отступление пригодится вам не только для NLP. Мы собираемся сначала выполнить PCA на трехмерном облаке точек, прежде чем переходить к данным высокой размерности для естественного языка.

В большинстве настоящих задач модель `sklearn.PCA` понадобится для латентно-семантического анализа. Исключение — если документов больше, чем помещается в RAM. В этом случае придется воспользоваться моделью IncrementalPCA из **`sklearn`** или какими-нибудь методиками масштабирования, о которых мы поговорим в главе 13.

СОВЕТ

Если корпус огромен и срочно нужны векторы тем (LSA), перейдите сразу к главе 13 и посмотрите на gensim.models.LsiModel по адресу https://radimrehurek.com/gensim/models/lsimodel.html. Если одного компьютера не хватает для быстрого решения поставленной задачи, попробуйте распараллеленную версию RocketML алгоритма SVD (http://rocketml.net/).

---

[1] Вы можете найти несколько примеров использования этих функций в PCA, которые помогут вам лучше разобраться во всех тонкостях, в nlpia.book.examples.ch04_sklearn_pca_source.

[2] См. статью Deep Learning Tutorial — PCA and Whitening по адресу http://mccormickml.com/2014/06/03/deep-learning-tutorial-pca-and-whitening/.

Мы начнем с набора реальных 3D-векторов, а не векторов «документ — слово» размерностью свыше 10 000. Гораздо проще представить что-либо в трехмерном пространстве, чем в 10 000-мерном, а поскольку речь идет всего о трех измерениях, то можно легко построить соответствующий график с помощью класса `Axes3D` из Matplotlib. Код для создания подобных вращающихся трехмерных графиков вы можете найти в пакете **`nlpia`** по адресу http://github.com/totalgood/nlpia.

Приведенное на рис. 4.4 облако точек взято из трехмерной развертки поверхности реального объекта, а не из заостренных вершин набора векторов BOW. Но именно это поможет лучше почувствовать, как работает LSA. И вы увидите, как производить операции над маленькими векторами и строить их графики, прежде чем перейдете к векторам более высоких размерностей, таким как векторы «документ — слово».

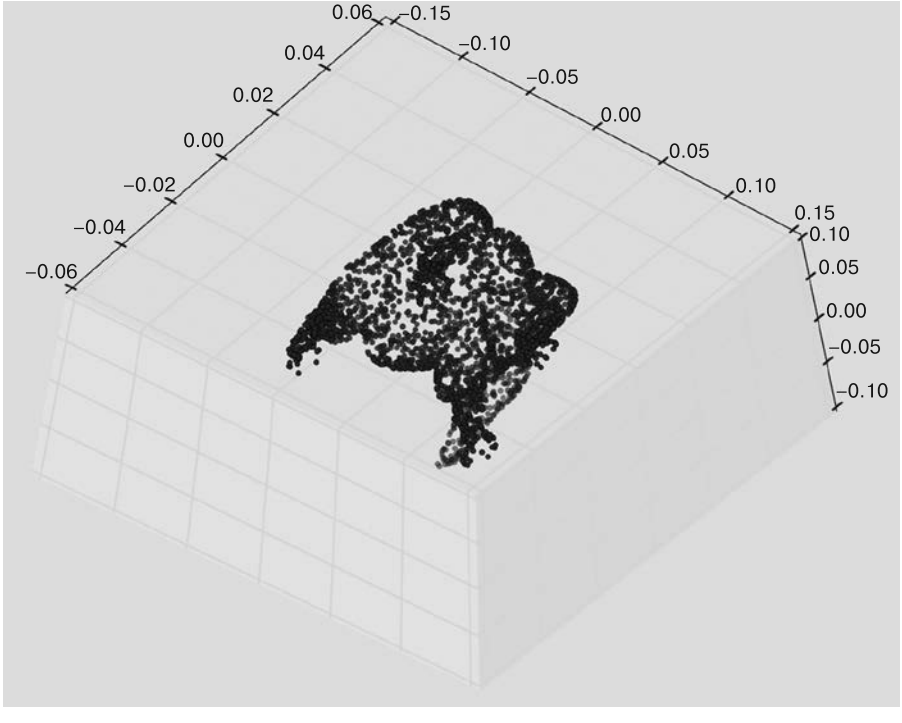


Рис. 4.4. Взгляд снизу на «брюхо» облака точек реального объекта

Угадаете, из какого 3D-объекта получились эти трехмерные векторы, исходя из одной только двумерной проекции, напечатанной в данной книге? Как бы вы запрограммировали компьютер для вращения объекта вокруг своей оси, чтобы удобнее его рассмотреть? Какими сводными показателями по точкам данных можно воспользоваться для оптимального выравнивания осей *X* и *Y* относительно объекта? Мысленно вращая это трехмерное пятно, представьте, как бы менялась при таком вращении дисперсия по осям *X*, *Y* и *Z*.

4.4.1. PCA на трехмерных векторах

Мы вручную поменяли ориентацию облака точек так, чтобы *минимизировать* дисперсию по осям окна графика. В результате было сложно понять, что это такое. Если SVD (LSA) сделает подобное с векторами «документ — слово», то скроет в них определенную информацию. Расположение точек одна поверх другой в двумерной проекции усложняет для человеческих глаз или алгоритма машинного обучения их разделение на осмысленные кластеры, а SVD сохраняет структуру (информационное содержание) векторов за счет *максимизации* дисперсии по измерениям нашей низкоразмерной «тени» пространства высокой размерности. Это как раз то, что нужно для машинного обучения, поскольку векторы низкой размерности улавливают саму «суть» того, что представляют. SVD *максимизирует* дисперсию по каждой из осей координат, а дисперсия оказывается отличным индикатором информации, то есть той самой искомой сути:

```
>>> import pandas as pd
>>> pd.set_option('display.max_columns', 6)
>>> from sklearn.decomposition import PCA
>>> import seaborn
>>> from matplotlib import pyplot as plt
>>> from nlpia.data.loaders import get_data

>>> df = get_data('pointcloud').sample(1000)
>>> pca = PCA(n_components=2)
>>> df2d = pd.DataFrame(pca.fit_transform(df), columns=list('xy'))
>>> df2d.plot(kind='scatter', x='x', y='y')
>>> plt.show()
```

Гарантируем, что вывод pd.DataFrame поместится на странице по ширине

Хотя в Scikit-Learn он называется PCA, на самом деле это SVD

Редуцируем трехмерное облако точек до двумерной проекции для отображения на 2D-диаграмме рассеяния

При запуске этого сценария ориентация двумерной проекции может случайно перевернуться слева направо, но она никогда не повернется вверх ногами или на произвольный угол. Ориентация 2D-проекции вычисляется таким образом, что максимальная дисперсия всегда выравнивается по оси X — первой оси координат. Вторая по размеру дисперсия всегда выравнивается по оси Y — второму измерению нашей «тени» (проекции). Но *полярность* (знак) этих осей координат произвольна, поскольку у оптимизации остается еще две степени свободы. При оптимизации полярность векторов (точек) может свободно меняться по оси X, Y или даже по обеим этим осям координат.

В каталоге `nlpia/data` вы найдете сценарий `horse_plot.py`, если хотите поэкспериментировать с трехмерной ориентацией лошади. Конечно, возможно и более оптимальное преобразование данных, убирающее одно измерение без снижения информационного содержания данных (с точки зрения пользователя), а Пикассо с его кубистскими «глазами» мог бы придумать и нелинейное преобразование, сохраняющее информационное содержание представлений с нескольких точек зрения сразу. И для этого существует алгоритм вложения, например представленный в главе 6.

Но разве старые добрые линейные SVD и PCA плохо сохраняют информацию из векторных данных облака точек? Так ли уж плохо 2D-проекция трехмерной лошади

отражает данные? И разве не сможет машина усвоить что-то из сводных показателей этих двумерных векторов, вычисленных на основе 3D-векторов поверхности лошади (рис. 4.5)?

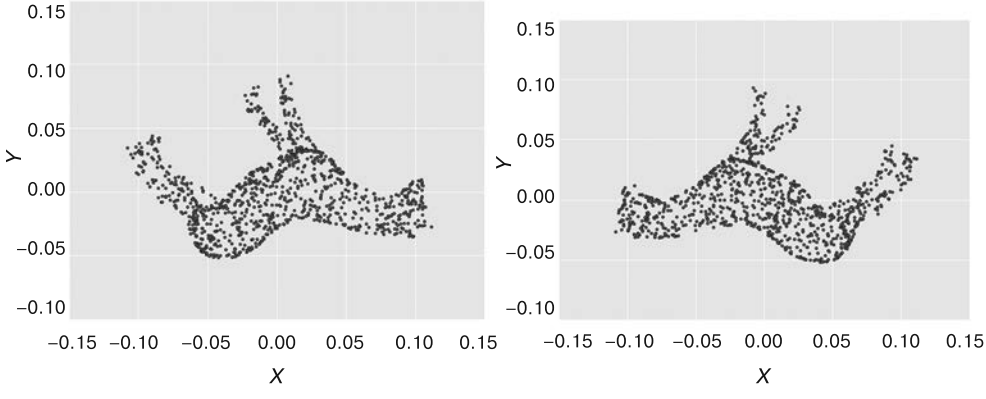


Рис. 4.5. Облака точек лошадей «голова к голове» вверх ногами

4.4.2. Хватит возиться с лошадьми, возвращаемся к NLP

Взглянем, на что способен SVD при работе с документами на естественном языке. Найдем с помощью SVD главные компоненты для 5000 СМС, маркированных как спам (или не спам). Словарь и разнообразие обсуждаемых в этом ограниченном наборе СМС тем должны были относительно малы. Так что уменьшим число тем до 16. Воспользуемся как моделью PCA Scikit-Learn, так и усеченной моделью SVD, чтобы увидеть различия.

Усеченная модель SVD предназначена для работы с разреженными матрицами. *Разреженные матрицы* (sparse matrices) — такие, которые содержат во множестве ячеек одно и тот же значение (обычно ноль или NaN). Матрицы мультимножеств NLP и TF-IDF — почти всегда разреженные, поскольку значительная часть документов не содержит нужной части слов из словаря. Количество большинства слов равно нулю (до добавления к ним всем фиктивного количества для выравнивания данных).

Разреженные матрицы напоминают электронные таблицы, почти пустые, но содержащие несколько разбросанных осмысленных значений. Благодаря применению плотных матриц, в которых все нули представлены явно, решение на основе модели PCA sklearn может оказаться более быстродействующим, чем `TruncatedSVD`. Но `sklearn.PCA` расходует большой объем оперативной памяти, пытаясь запомнить все эти повторяющиеся нули. Векторизатор `TfidfVectorizer` из Scikit-Learn возвращает разреженные матрицы, так что вы должны преобразовать их в плотные, прежде чем сравнивать результаты с PCA.

Сначала загрузим СМС из DataFrame пакета nlpia:

```
>>> import pandas as pd
>>> from nlpia.data.loaders import get_data
>>> pd.options.display.width = 120
>>> sms = get_data('sms-spam')
>>> index = ['sms{}{}'.format(i, '!'*j)
➥  for (i,j) in zip(range(len(sms)), sms.spam)]
>>> sms.index = index
>>> sms.head(6)
       spam                                                text
sms0      0 Go until jurong point, crazy.. Available only ...
sms1      0 Ok lar... Joking wif u oni...
sms2!     1 Free entry in 2 a wkly comp to win FA Cup fina...
sms3      0 U dun say so early hor... U c already then say...
sms4      0 Nah I don't think he goes to usf, he lives aro...
sms5!     1 FreeMsg Hey there darling it's been 3 week's n...
```

Немного облагораживаем вывод широких DataFrame библиотеки Pandas

Добавляем восклицательный знак в конец индексов спамовых СМС, чтобы сделать их более заметными

Теперь можно вычислить векторы TF-IDF для всех этих сообщений:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from nltk.tokenize.casual import casual_tokenize
>>> tfidf = TfidfVectorizer(tokenizer=casual_tokenize)
>>> tfidf_docs = tfidf.fit_transform(raw_documents=sms.text).toarray()
>>> len(tfidf.vocabulary_)
9232

>>> tfidf_docs = pd.DataFrame(tfidf_docs)
>>> tfidf_docs = tfidf_docs - tfidf_docs.mean()
>>> tfidf_docs.shape
(4837, 9232)
>>> sms.spam.sum()
638
```

Центрируем векторизованные документы (векторы BOW) путем вычитания математического ожидания

Атрибут .shape возвращает длину всех измерений любого массива NumPy

Метод .sum() класса Series библиотеки Pandas аналогичен сумме столбца электронной таблицы — он просто складывает все элементы

Итак, у нас есть 4837 СМС, содержащих 9232 различных токена одногramм, полученных из токенизатора (casual_tokenize). Только 638 из этих 4837 сообщений (13 %) маркированы как спам. Мы имеем несбалансированный тренировочный набор данных с соотношением 8:1 не спама (нормальных СМС) к спаму (непрошеных предложений товаров/услуг и рекламы).

Решить проблему этого перекоса в сторону не спама можно, снизив «поощрение» любой модели, правильно классифицирующей не спам. Более сложная проблема — большой размер словаря, |V|. Количество токенов в словаре (9232) больше, чем количество обрабатываемых сообщений (выборок) (4832). Поэтому уникальных слов в словаре (или лексиконе) намного больше, чем СМС. И только небольшая часть этих СМС (1/8) маркирована как спам. Это верная дорога к переобучению (https://

ru.wikipedia.org/wiki/Переобучение). Немногие из уникальных слов нашего большого словаря будут помечены как спамовые слова в наборе данных.

Переобучение означает, что будет учитываться при анализе лишь несколько слов из словаря. Так что наш фильтр спама будет зависеть от того, присутствуют ли в отфильтровываемых сообщениях эти несколько спамовых слов. Спамеры легко смогут обойти нашу защиту с помощью синонимов этих спамовых слов. Если наш словарь не включает используемые спамерами новые синонимы, то фильтр будет ошибочно классифицировать эти ловко написанные СМС как не спам.

Эта проблема с переобучением — неизменный спутник NLP. Непросто найти маркированный набор данных на естественном языке, включающий для каждой помеченной мысли все варианты ее выражения. Мы не нашли большую базу данных СМС, которая бы включала все различные варианты спама и не спама. И только у нескольких крупных корпораций достаточно ресурсов для создания подобного набора данных. Так что всем остальным приходится искать контрмеры для переобучения. Приходится использовать алгоритмы, которые хорошо производят обобщение на основе всего нескольких примеров.

Основная контрмера для переобучения — понижение размерности. За счет уплотнения измерений (слов) в меньшее число измерений (тем) конвейер NLP становится более «общим». Если редуцировать измерения (словарь), фильтр спама станет работать на более широком диапазоне СМС.

Именно это и делает LSA — сокращает число измерений, а значит, помогает предотвратить переобучение[1]. Он производит обобщение на основе маленького набора данных за счет предположения о линейной зависимости между частотностями слов. Если *half* встречается в спамовых сообщениях, часто включающих также слово *off* (например, в выражении *Half off!)*, то LSA выявит подобные связи между словами, определит, насколько они сильны, и обобщит фразы *half off* в спамовых сообщениях на фразы вроде *80 % off*. Возможно, даже пойдет в обобщении еще дальше: до фразы *80 % discount*, если цепочка связей в данных NLP включает ассоциацию *discount* с *off*.

ПРИМЕЧАНИЕ

> Некоторые исследователи рассматривают обобщение как главную проблему машинного обучения и искусственного интеллекта. Для описания исследований моделей, в которых оно доводится до крайности, требуя на порядки меньше данных для достижения той же точности, что у традиционных моделей, часто используют термин «обучение по одному примеру» (one-shot learning).

Обобщение конвейера NLP гарантирует возможности его применения к более широкому диапазону реальных СМС вместо конкретного набора сообщений.

---

[1] Больше информации о переобучении и обобщении см. в приложении Д.

4.4.3. Применение PCA для семантического анализа СМС

Попробуем сначала модель PCA из Scikit-Learn. Мы уже видели, как она работает, загоняя 3D-лошадей в двумерный загон. Теперь загоним набор данных 9232-мерных векторов TF-IDF в 16-мерные векторы тем:

```
>>> from sklearn.decomposition import PCA

>>> pca = PCA(n_components=16)
>>> pca = pca.fit(tfidf_docs)
>>> pca_topic_vectors = pca.transform(tfidf_docs)
>>> columns = ['topic{}'.format(i) for i in range(pca.n_components)]
>>> pca_topic_vectors = pd.DataFrame(pca_topic_vectors, columns=columns,\
...    index=index)
>>> pca_topic_vectors.round(3).head(6)
       topic0  topic1  topic2   ...  topic13  topic14  topic15
sms0    0.201   0.003   0.037   ...   -0.026   -0.019    0.039
sms1    0.404  -0.094  -0.078   ...   -0.036    0.047   -0.036
sms2!  -0.030  -0.048   0.090   ...   -0.017   -0.045    0.057
sms3    0.329  -0.033  -0.035   ...   -0.065    0.022   -0.076
sms4    0.002   0.031   0.038   ...    0.031   -0.081   -0.021
sms5!  -0.016   0.059   0.014   ...    0.077   -0.015    0.021
```

Можно узнать, какую долю каждого из слов содержат эти темы, изучив их веса. Из них понятно, насколько часто *half* встречается совместно с *off* (например, во фразе *half off*). После чего можно выяснить, какая из этих тем представляет собой нашу тему *discount*.

СОВЕТ

> Узнать веса любого подобранного преобразования sklearn можно, заглянув в его атрибут .components_.

Сначала распределим слова по всем измерениям преобразования PCA. Они должны быть правильно упорядочены, поскольку векторизатор TFIDFVectorizer сохраняет словарь в виде ассоциативного массива соответствий термов номерам индексов (номерам столбцов):

```
>>> tfidf.vocabulary_
{'go': 3807,
 'until': 8487,
 'jurong': 4675,
 'point': 6296,
...
>>> column_nums, terms = zip(*sorted(zip(tfidf.vocabulary_.values(),\
...     tfidf.vocabulary_.keys())))
>>> terms
('!',
 '"',
 '#',
 '#150',
...
```

Сортируем словарь по количеству термов. С помощью паттерна zip(*sorted(zip())) удобно распаковать что-либо для сортировки по элементу, который не является крайним слева, с упаковкой заново после этой сортировки

Теперь можно создать красивый `DataFrame` библиотеки Pandas, который будет содержать веса с метками для всех столбцов и строк:

```
>>> weights = pd.DataFrame(pca.components_, columns=terms,
➥ index=['topic{}'.format(i) for i in range(16)])
>>> pd.options.display.max_columns = 8
>>> weights.head(4).round(3)
             !       "       #  ...       …       ?     ?ud       ?
topic0  -0.071   0.008  -0.001  ...  -0.002   0.001   0.001   0.001
topic1   0.063   0.008   0.000  ...   0.003   0.001   0.001   0.001
topic2   0.071   0.027   0.000  ...   0.002  -0.001  -0.001  -0.001
topic3  -0.059  -0.032  -0.001  ...   0.001   0.001   0.001   0.001
```

Часть из этих столбцов (термов) не слишком интересна. Лучше исследуем наш `tfidf.vocabulary`. Взглянем, сможем ли мы найти какие-либо из этих `half off` термов и узнать, к каким темам они относятся:

```
>>> pd.options.display.max_columns = 12
>>> deals = weights['! ;) :) half off free crazy deal only $ 80 %'.split()].
round(3) * 100
>>> deals
            !   ;)    :)  half   off  free crazy  deal only    $   80    %
topic0   -7.1  0.1  -0.5  -0.0  -0.4  -2.0  -0.0  -0.1 -2.2  0.3 -0.0 -0.0
topic1    6.3  0.0   7.4   0.1   0.4  -2.3  -0.2  -0.1 -3.8 -0.1 -0.0 -0.2
topic2    7.1  0.2  -0.1   0.1   0.3   4.4   0.1  -0.1  0.7  0.0  0.0  0.1
topic3   -5.9 -0.3  -7.1   0.2   0.3  -0.2   0.0   0.1 -2.3  0.1 -0.1 -0.3
topic4   38.1 -0.1 -12.5  -0.1  -0.2   9.9   0.1  -0.2  3.0  0.3  0.1 -0.1
topic5  -26.5  0.1  -1.5  -0.3  -0.7  -1.4  -0.6  -0.2 -1.8 -0.9  0.0  0.0
topic6  -10.9 -0.5  19.9  -0.4  -0.9  -0.6  -0.2  -0.1 -1.4 -0.0 -0.0 -0.1
topic7   16.4  0.1 -18.2   0.8   0.8  -2.9   0.0   0.0 -1.9 -0.3  0.0 -0.1
topic8   34.6  0.1   5.2  -0.5  -0.5  -0.1  -0.4  -0.4  3.3 -0.6 -0.0 -0.2
topic9    6.9 -0.3  17.4   1.4  -0.9   6.6  -0.5  -0.4  3.3 -0.4 -0.0  0.0
...
>>> deals.T.sum()
topic0    -11.9
topic1      7.5
topic2     12.8
topic3    -15.4
topic4     38.2
topic5    -34.1
topic6      4.8
topic7     -5.7
topic8     39.8
topic9     31.6
...
```

Похоже, что тональности «торговли» *(deals)* для всех тем 4, 8 и 9 положительны, а темы 0, 3 и 5 представляют собой «антиторговлю» и содержат сообщения о противоположных вещах. Так что связанные с «торговлей» слова могут положительно влиять на одни темы и отрицательно — на другие. Не существует одного явного номера темы «торговли».

ВАЖНО

Токенизатор casual_tokenize разбивает 80 % на ["80", "%"], а $80 million — на ["$", "80", "million"]. Если не воспользоваться LSA или токенизатором биграмм, то никакой разницы между 80 % и $80 million для конвейера NLP не будет. Оба этих выражения включают токен "80".

Поиск смысла тем — одна из самых сложных задач LSA. В LSA между словами возможны только линейные связи. Обычно у вас есть лишь очень небольшой корпус. Так что слова в темах будут зачастую сочетаться бессмысленным для человека способом. В одно измерение (главную компоненту) будут «запихиваться» по нескольку слов из различных тем для гарантии, что модель захватит как можно больше дисперсии, используемой нашими 9232 словами.

4.4.4. Применение усеченного SVD для семантического анализа СМС

Теперь можно попробовать воспользоваться моделью `TruncatedSVD` из Scikit-Learn. При этом более непосредственном варианте LSA, который работает в обход модели `PCA` Scikit-Learn, видно, что происходит внутри адаптера `PCA`. Она умеет работать с разреженными матрицами, так что при работе с большими наборами данных все равно лучше воспользоваться `TruncatedSVD`, а не `PCA`. SVD-часть `TruncatedSVD` разбивает TF-IDF-матрицу на три другие матрицы, а Truncated-часть отбрасывает измерения, содержащие меньше всего информации о матрице TF-IDF. Эти отброшенные измерения представляют темы (линейные сочетания слов), в наименьшей степени варьирующиеся в пределах нашего набора документов. Вероятно, они содержат массу стоп-слов и других слов, равномерно распределенных по всем документам.

Мы воспользуемся `TruncatedSVD`, чтобы оставить только 16 наиболее интересных тем, на долю которых приходится львиная доля дисперсии в наших векторах TF-IDF:

Как и в PCA, мы вычислим 16 тем, но пройдем по данным в цикле 100 раз (по умолчанию — пять), чтобы гарантировать практически такую же точность результата, как и у PCA

Метод fit_transform раскладывает векторы TF-IDF и преобразует их в векторы тем за один проход

```
>>> from sklearn.decomposition import TruncatedSVD

>>> svd = TruncatedSVD(n_components=16, n_iter=100)
>>> svd_topic_vectors = svd.fit_transform(tfidf_docs.values)
>>> svd_topic_vectors = pd.DataFrame(svd_topic_vectors, columns=columns,\
...     index=index)
>>> svd_topic_vectors.round(3).head(6)
       topic0  topic1  topic2  ...   topic13  topic14  topic15
sms0    0.201   0.003   0.037  ...    -0.036   -0.014    0.037
sms1    0.404  -0.094  -0.078  ...    -0.021    0.051   -0.042
sms2!  -0.030  -0.048   0.090  ...    -0.020   -0.042    0.052
sms3    0.329  -0.033  -0.035  ...    -0.046    0.022   -0.070
sms4    0.002   0.031   0.038  ...     0.034   -0.083   -0.021
sms5!  -0.016   0.059   0.014  ...     0.075   -0.001    0.020
```

Эти векторы тем, полученные из `TruncatedSVD`, аналогичны сгенерированным PCA! Такого результата удалось добиться благодаря большому числу итераций (`n_iter`) и тому, что мы позаботились о центрировании всех частотностей TF-IDF для каждого из термов (столбцов) относительно нуля (путем вычитания среднего значения для каждого терма).

Посмотрите внимательнее на веса для каждой из тем и попробуйте понять их значения. Можно ли классифицировать эти шесть СМС как спам или не спам, не зная, о чем эти темы и у каких слов в них наибольшие веса? В этом поможет метка `!` возле меток строк спамовых СМС. Это непростая, но выполнимая задача, особенно для машины, которая может просмотреть все 5000 наших тренировочных примеров и выбрать для каждой из тем пороговые значения, позволяющие отличить спам от не спама.

4.4.5. Насколько хорошо LSA классифицирует спам

Один из способов определить, насколько хорошо модель векторного пространства производит классификацию, — выяснить, как косинусный коэффициент подобия двух векторов коррелирует с их принадлежностью одному и тому же классу. Посмотрим, пригодится ли косинусный коэффициент подобия между соответствующими парами документов для нашей конкретной задачи бинарной классификации. Вычислим скалярные произведения между первыми шестью векторами тем для первых шести СМС. Легко заметить большие положительные косинусные коэффициенты сходства (скалярные произведения) между спамовыми сообщениями (`sms2!`):

Нормализация всех векторов тем по длине (L^2-норма) позволяет вычислить косинусные расстояния с помощью скалярного произведения

```
>>> import numpy as np

>>> svd_topic_vectors = (svd_topic_vectors.T / np.linalg.norm(\
...     svd_topic_vectors, axis=1)).T
>>> svd_topic_vectors.iloc[:10].dot(svd_topic_vectors.iloc[:10].T).round(1)

       sms0  sms1  sms2!  sms3  sms4  sms5!  sms6  sms7  sms8!  sms9!
sms0    1.0   0.6   -0.1   0.6  -0.0   -0.3  -0.3  -0.1   -0.3   -0.3
sms1    0.6   1.0   -0.2   0.8  -0.2    0.0  -0.2  -0.2   -0.1   -0.1
sms2!  -0.1  -0.2    1.0  -0.2   0.1    0.4   0.0   0.3    0.5    0.4
sms3    0.6   0.8   -0.2   1.0  -0.2   -0.3  -0.1  -0.3   -0.2   -0.1
sms4   -0.0  -0.2    0.1  -0.2   1.0    0.2   0.0   0.1   -0.4   -0.2
sms5!  -0.3   0.0    0.4  -0.3   0.2    1.0  -0.1   0.1    0.3    0.4
sms6   -0.3  -0.2    0.0  -0.1   0.0   -0.1   1.0   0.1   -0.2   -0.2
sms7   -0.1  -0.2    0.3  -0.3   0.1    0.1   0.1   1.0    0.1    0.4
sms8!  -0.3  -0.1    0.5  -0.2  -0.4    0.3  -0.2   0.1    1.0    0.3
sms9!  -0.3  -0.1    0.4  -0.1  -0.2    0.4  -0.2   0.4    0.3    1.0
```

Изучение столбца `sms0` (или строки `sms0`) показывает, что косинусные коэффициенты сходства между `sms0` и спамовыми сообщениями (`sms2!`, `sms5!`, `sms8!`, `sms9!`) — существенно меньше нуля. Вектор темы для `sms0` заметно отличается от векторов тем спамовых сообщений.

Проделывая то же самое для столбца `sms2!`, видим положительную корреляцию с другими спамовыми сообщениями. Семантика спамовых сообщений схожа, они посвящены одинаковым темам.

Подобным образом работает и семантический поиск. Косинусный коэффициент между вектором запроса и всеми векторами тем в базе данных документов можно использовать для поиска наиболее семантически похожего сообщения в базе. Ближайший в смысле расстояния к вектору конкретного запроса документ соответствует наиболее близкому по смыслу документу. Спамовость — лишь один из смыслов, вкладываемых в темы наших СМС.

К сожалению, подобное сходство между векторами тем в каждом из классов (спам и не спам) не сохраняется для всех сообщений. Для этого набора векторов тем непросто провести границу между спамовыми и неспамовыми сообщениями. Сложно установить пороговое значение сходства отдельного спамового сообщения, которое бы гарантировало правильную классификацию на спам и не спам. Чем менее спамовым является сообщение, тем дальше оно находится (менее похоже) от других спамовых сообщений в наборе данных. Этого знания достаточно для создания фильтра спама на основе имеющихся векторов тем, а алгоритм машинного обучения может просмотреть все темы по отдельности на предмет меток «спам/не спам» и провести гиперплоскость или другую границу между спамовыми и неспамовыми сообщениями.

При использовании усеченного SVD необходимо отбросить собственные значения, прежде чем вычислять векторы тем. Мы хитростью заставили реализацию `TruncatedSVD` из Scikit-Learn проигнорировать информацию о масштабе из собственных значений (S-матрицы на наших диаграммах) за счет:

- нормализации векторов TF-IDF по длине (L^2-норма);
- центрирования частотностей термов TF-IDF путем вычитания средней частотности каждого из термов (слов).

Процесс нормализации исключает любое масштабирование (систематическую ошибку) собственных значений и концентрирует SVD на относящейся к повороту части преобразования векторов TF-IDF. Благодаря игнорированию собственных значений (масштаба/длины вектора) можно построить гиперкуб, ограничивающий векторное пространство тем, благодаря чему можно считать в своей модели все темы равноценными. Для использования этого приема в своей реализации SVD вы можете нормализовать все векторы TF-IDF по L_2-норме перед вычислением SVD или усеченного SVD. Реализация PCA из Scikit-Learn выполняет это вместо вас, «центрируя» и «отбеливая» данные.

Без такой нормализации вес редких тем оказывается немного больше, чем был бы в противоположном случае. Спамовость — редкая тема, встречающаяся лишь в 13 % случаев. Вследствие такой нормализации или отбрасывания собственных значений вес соответствующих тем окажется больше. При таком подходе полученные в результате темы окажутся более коррелированными с малозаметными характеристиками, такими как спамовость.

СОВЕТ

Какой бы алгоритм или реализацию вы ни использовали для семантического анализа (LSA, PCA, SVD, усеченный SVD или LDiA), сначала нормализуйте свои векторы BOW или TF-IDF. В противном случае можно столкнуться с большими различиями в масштабе тем. Они снижают возможности модели различать малозаметные, редко встречающиеся темы. Можно наглядно представить, что изменения масштаба приводят к «каньонам» и «рекам» в контурном графике целевой функции, что затрудняет для других алгоритмов машинного обучения поиск на этой «пересеченной местности» оптимальных пороговых значений для тем.

Расширения LSA и SVD

Успех применения сингулярного разложения для семантического анализа и понижения размерности побудил исследователей к его расширению и усовершенствованию. Эти расширения в основном предназначены для решения не связанных с NLP задач, но мы упомянем их на случай, если вам придется с ними столкнуться. Они иногда применяются для механизмов рекомендации на основе поведения наряду с NLP-механизмами рекомендации на основе содержимого, а также используются для вычисления статистических частеречных показателей естественного языка[1]. Любой метод разложения матриц на множители или понижения размерности может использоваться в отношении частотностей термов естественного языка. Так что эти методы могут вполне пригодиться в конвейере семантического анализа:

- квадратичный дискриминантный анализ (quadratic discriminant analysis, QDA);
- случайная проекция (random projection);
- неотрицательное разложение матриц (nonnegative matrix factorization, NMF).

QDA — альтернатива LDA. Создает не линейные, а квадратичные полиномиальные преобразования. Они задают векторное пространство, которое можно использовать для разделения классов. И граница между классами в векторном пространстве QDA также представляет собой квадратичную кривую, подобную чаше, сфере или дренажному желобу.

Случайная проекция — метод разложения и преобразования матриц, аналогичный SVD, но стохастический. Так что при каждом его запуске получаются другие результаты. Однако стохастическая природа упрощает его выполнение на параллельных машинах, а в некоторых случаях (для части из этих случайных запусков алгоритма) преобразования оказываются лучше, чем у SVD (и LSA). Но для задач NLP случайная проекция используется редко. И в пакетах NLP, таких как Spacy

[1] См. статью *Feldman S., Marin M. A., Ostendorf M., Gupta M. R.* Part-of-speech Histograms for Genre Classification of Text по адресу http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.332.629&rep=rep1&type=pdf.

и NLTK, нет широко используемой ее реализации. Если вы считаете, что этот метод подходит для ваших задач, можете изучить его глубже.

В большинстве случаев лучше придерживаться LSA, в котором используется проверенный и надежный алгоритм SVD[1].

4.5. Латентное размещение Дирихле

Почти всю главу мы обсуждали латентно-семантический анализ и различные способы его выполнения с помощью Scikit-Learn или NumPy. Для большинства задач моделирования тем, семантического анализа и механизмов рекомендации на основе содержимого следует пробовать LSA[2]. Лежащая в его основе математика проста и эффективна, и получаемое в результате линейное преобразование может применяться к новым пакетам данных на естественном языке без обучения при небольшой потере точности. В некоторых случаях LDiA обеспечивает немного лучшие результаты.

LDiA производит многое из того, что мы делали для создания тематических моделей с помощью LSA (и SVD). В отличие от LSA, LDiA предполагает, что частотности слов распределены по Дирихле. Такое допущение точнее отражает статистику распределения слов по темам, чем линейная математика LSA.

LDiA создает семантическую модель векторного пространства (подобную векторам тем) с помощью подхода, аналогичного ходу ваших мыслей во время описанного в данной главе мысленного эксперимента. Тогда вы сами распределяли слова по темам в соответствии с частотой их совместной встречаемости в одном документе. Далее можно определить смесь тем для документа на основе сочетаний слов в каждой из тем, в которую эти слова были распределены. Поэтому понятность тематического моделирования методом LDiA намного выше, чем при LSA, благодаря распределению слов по темам, а тем — по документам.

LDiA рассматривает каждый из документов как смесь (линейную комбинацию) некоего произвольного числа тем, выбираемого при начале обучения модели LDiA. И предполагает, что любую тему можно представить с помощью распределения слов (частотностей термов). Вычисление вероятности (веса) каждой из этих тем в документе, как и вероятности попадания слова в конкретную тему, начинается с распределения вероятности Дирихле (*априорного распределения*, если вы еще помните курс статистики). Поэтому данный алгоритм и получил такое название.

---

[1] SVD традиционно применяется для вычисления матриц, «псевдообратных» для неквадратных матриц. Вы можете легко представить, сколько приложений существует для метода обращения матрицы.

[2] Проведенное в 2015 году сравнение алгоритмов рекомендации фильмов на основе содержимого Соней Бергамаши и Лаурой По показало, что LSA примерно вдвое точнее, чем LDiA. См. статью: *Bergamaschi S., Po L.* Comparing LDA and LSA Topic Models for Content-Based Movie Recommendation Systems — по адресу https://www.dbgroup.unimo.it/~po/pubs/LNBI_2015.pdf.

4.5.1. Основная идея LDiA

Подход LDiA был разработан в 2000 году британскими генетиками для упрощения «вывода структуры популяции» по последовательности генов[1]. Стэнфордские исследователи (в том числе Эндрю Ын (Andrew Ng)) распространили этот подход на NLP в 2003 году[2]. Не пугайтесь громких имен создателей этого подхода. Мы объясним его главные нюансы всего в нескольких строках кода на языке Python. Вам нужно понимать ровно столько, чтобы интуитивно чувствовать, что этот метод делает. Тогда вы будете знать, для чего он может пригодиться в вашем конвейере.

Блай и Ын предложили эту идею, перевернув наш мысленный эксперимент с ног на голову. Они придумали, как машина, умеющая только бросать игральные кости (генерировать случайные числа), может писать документы для анализируемого корпуса. Поскольку мы имеем дело только с мультимножествами, они отказались от упорядочения этих слов, которое делало фразу осмысленной и позволяло сформировать настоящий документ. Они просто смоделировали статистические показатели для смеси слов, которая бы была частью конкретного BOW для каждого из документов.

Блай и Ын представили себе машину, которой, чтобы начать генерацию смеси слов для конкретного документа, нужно принять всего одно из двух решений. Они предположили, что генератор документов выбирает эти слова случайным образом. Причем возможные варианты распределены по определенному вероятностному закону, вроде выбора числа граней игральной кости и сочетания костей, необходимого для создания листа персонажа в игре «Подземелья и драконы». Для «листа персонажа» нашего документа необходимо всего два броска костей. Но эти кости велики и их несколько, а правила их комбинирования для получения нужных вероятностей для различных желаемых значений — сложны. Нам требуются определенные распределения вероятностей для чисел слов и тем, соответствующие распределениям данных значений в реальных документах.

Эти два броска костей символизируют:

1) число генерируемых для документа слов (распределение Пуассона);

2) число смешиваемых для документа тем (распределение Дирихле).

После получения этих двух чисел начинается самое сложное — выбор слов для документа. Воображаемая машина генерации BOW проходит в цикле по темам и выбирает случайным образом подходящие для текущей темы слова, пока не достигнет числа слов, которое (как было решено на шаге 1) должен содержать документ. Самое сложное — выбрать вероятности этих слов для тем: то, насколько слова подходят для каждой из них. После этого наш «робот» просто ищет вероятности нужных слов для каждой из тем в матрице вероятностей «тема — слово». Если не помните, как эта матрица выглядит, взгляните на приведенный ранее в данной главе простой пример.

---

[1] *Pritchard J. K., Stephens M., Donnelly P.* Inference of Population Structure Using Multilocus Genotype Data, http://www.genetics.org/content/155/2/945.

[2] См. статью: *Blei D. M., Ng A. Y., Jordan M. I.* Latent Dirichlet Allocation — по адресу http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf.

Все, что нужно этой машине, — один параметр для пуассоновского распределения (при броске костей на шаге 1), указывающий желаемую среднюю длину документа, и еще пара параметров для задания распределения Дирихле, которое определяет число тем. Далее нашему алгоритму генерации документов понадобится матрица «терм — тема» всех желаемых слов и тем, то есть его словарь. И еще — смесь тем, о которых он хочет «говорить».

Вернемся от генерации (написания) документов к исходной задаче оценки тем и слов существующего документа. В ней нужно измерить (вычислить) упомянутые параметры слов и тем для первых двух шагов. После этого необходимо вычислить матрицу «терм — тема» по набору документов. Именно это и делает LDiA.

Блай и Ын поняли, что параметры для шагов 1 и 2 можно определить, проанализировав статистические показатели документов из корпуса. Например, для шага 1 можно вычислить среднее число слов (или *n*-грамм) во всех мультимножествах слов документов из корпуса. Примерно вот так:

```
>>> total_corpus_len = 0
>>> for document_text in sms.text:
...     total_corpus_len += len(casual_tokenize(document_text))
>>> mean_document_len = total_corpus_len / len(sms)
>>> round(mean_document_len, 2)
21.35
```

Или в одну строку кода:

```
>>> round(sum([len(casual_tokenize(t)) for t in sms.text]) * 1. / len(sms.text), 2)
21.35
```

Учтите, что вычислять этот статистический показатель следует непосредственно на основе ваших BOW. Причем обязательно подсчитывать в документах токенизированные и векторизованные слова (`Counter()`). И произвести фильтрацию стоп-слов либо какую-то другую нормализацию перед подсчетом уникальных термов. Таким образом, в этот подсчет будут включены все слова из словаря векторов BOW (все подсчитываемые *n*-граммы), но только используемые BOW слова (а не стоп-слова, например). Как и другие алгоритмы из этой главы, данный алгоритм LDiA основан на мультимножественной модели векторного пространства.

Второй параметр, который необходимо указать для модели LDiA, представляет собой несколько бóльшую проблему. Число тем в конкретном наборе документов нельзя измерить непосредственно до распределения слов по темам. Здесь, как и в *методе k-средних* и *KNN*, необходимо указать *k* заранее. Можно высказать предположение о числе тем (подобно числу кластеров *k* в методе *k*-средних), а затем проверить, подходит ли оно для вашего набора документов. Когда вы укажете методу LDiA, сколько тем искать, он найдет подходящую для каждой темы смесь слов, которая бы оптимизировала его целевую функцию[1].

[1] Узнать больше о целевой функции LDiA можно из исходной статьи: *Hoffman M. D., Blei D. M., Bach F.* Online Learning for Latent Dirichlet Allocation — по адресу https://www.di.ens.fr/%7Efbach/mdhnips2010.pdf.

Оптимизировать этот гиперпараметр (k, число тем)[1] можно путем его подгонки вплоть до момента, когда он окажется подходящим для приложения. Эту оптимизацию можно автоматизировать, если есть возможность измерить каким-либо образом качество отражения языковой моделью LDiA смысла документов. Одна из подходящих для этой оптимизации функций стоимости — степень эффективности (или неэффективности) работы данной модели LDiA в задачах классификации или регрессии, например, при анализе тональностей, тем или разметке ключевых слов документа. Для этого необходим набор маркированных документов, на которых можно будет проверить тематическую модель (классификатор)[2].

4.5.2. Тематическая модель LDiA для СМС

Генерируемые LDiA темы обычно более понятны и объяснимы для людей. Дело в том, что часто встречающиеся вместе слова распределяются по одним темам, чего люди обычно и ждут. Если LSA (PCA) старается оставлять в удалении изначально удаленные элементы, то LDiA стремится собирать вместе изначально близко расположенные элементы.

Может показаться, что эти подходы одинаковые, но нет. Математически при этом производится оптимизация разных вещей. Целевые функции оптимизаторов при этом различны, так что и цели будут достигнуты разные. Чтобы сохранить близость векторов высокой размерности при переходе в пространство низкой размерности, LDiA приходится нелинейно изгибать и деформировать пространство (и сами векторы). Это сложно продемонстрировать наглядно, разве что выполнить оптимизацию 3D-данных и вычислить двумерные «проекции» получившихся векторов.

Если хотите помочь другим читателям и попутно чему-то научиться — добавляйте еще код в пример с лошадью (https://github.com/totalgood/nlpia/blob/master/src/nlpia/book/examples/ch04_horse.py) и отправляйте его на GitHub в `nlpia` (https://github.com/totalgood/nlpia). Например, можно создать векторы «слово — документ» для каждой из тысяч точек лошади, преобразовав их в целочисленные значения количеств слов x, y и z, то есть измерения трехмерного векторного пространства. После этого можно сгенерировать на основе этих значений искусственные документы и пропустить их через все примеры LDiA и LSA, приведенные ранее в этой главе.

Взглянем, как сделать это для набора данных из нескольких тысяч СМС, маркированных как спам и не спам. Сначала вычислим векторы TF-IDF, а затем векторы тем для всех СМС (документов). Для классификации сообщений по спаму мы

---

[1] Блай и Ын использовали для этого параметра символ θ, а не k.

[2] Крейг Боумен (Craig Bowman), библиотекарь из университета Майами в штате Огайо (http://www.lib.miamioh.edu/people/), использует в качестве меток тем для книг проекта «Гутенберг» систему классификации Библиотеки конгресса. Думаем, это наиболее амбициозный открытый научный NLP-проект (https://github.com/craigboman/gutenberg) на благо общества из тех, с которыми нам приходилось сталкиваться.

будем, как и ранее, использовать только 16 тем (компонент). Небольшое число тем (измерений) снижает риск переобучения[1].

LDiA использует исходные векторы BOW для количеств слов, а не нормализованные векторы TF-IDF. Вот простой способ, как вычислить векторы BOW в Scikit-Learn:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> from nltk.tokenize import casual_tokenize
>>> np.random.seed(42)

>>> counter = CountVectorizer(tokenizer=casual_tokenize)
>>> bow_docs = pd.DataFrame(counter.fit_transform(raw_documents=sms.text)\
...     .toarray(), index=index)
>>> column_nums, terms = zip(*sorted(zip(counter.vocabulary_.values(),\
...     counter.vocabulary_.keys())))
>>> bow_docs.columns = terms
```

Еще раз проверим осмысленность наших подсчитанных количеств слов для первого СМС с меткой `sms0`:

```
>>> sms.loc['sms0'].text
'Go until jurong point, crazy.. Available only in bugis n great world la e
buffet... Cine there got amore wat...'
>>> bow_docs.loc['sms0'][bow_docs.loc['sms0'] > 0].head()
,            1
..           1
...          2
amore        1
available    1
Name: sms0, dtype: int64
```

Так можно воспользоваться LDiA для создания векторов тем для нашего корпуса SMS:

```
>>> from sklearn.decomposition import LatentDirichletAllocation as LDiA

>>> ldia = LDiA(n_components=16, learning_method='batch')
>>> ldia = ldia.fit(bow_docs)
>>> ldia.components_.shape
(16, 9232)
```

Метод LDiA работает немного дольше, чем PCA или SVD, особенно при большом числе тем и слов в корпусе

Итак, наша модель распределила 9232 слова по 16 темам (компонентам). Давайте взглянем на несколько первых слов и их распределение по нашим 16 темам. Учтите, что ваши результаты могут отличаться от наших. LDiA — стохастический алгоритм, использующий генератор случайных чисел для принятия части статистических решений относительно распределения термов по темам. Так что ваши веса «тема — слово» будут отличаться от приведенных ниже, хотя порядок будет тем же. При каждом запуске `sklearn.LatentDirichletAllocation` (или любого другого алгоритма

[1] Больше информации о том, чем плохо переобучение и как здесь может помочь обобщение, см. в приложении Г.

LDiA) результаты будут различаться, если не зафиксировать начальное значение для генератора случайных чисел:

```
>>> pd.set_option('display.width', 75)
>>> components = pd.DataFrame(ldia.components_.T, index=terms,\
...     columns=columns)
>>> components.round(2).head(3)
      topic0   topic1   topic2   ...   topic13   topic14   topic15
!     184.03    15.00    72.22   ...    297.29     41.16     11.70
"       0.68     4.22     2.41   ...     62.72     12.27      0.06
#       0.06     0.06     0.06   ...      4.05      0.06      0.06
```

Восклицательный знак (`!`) попал в большинство тем, но есть и особенно значительный `topic3`, в котором символ кавычек (`"`) практически никакой роли не играет. Возможно, `topic3` посвящен эмоциям и для него неважны числа или кавычки. Посмотрим:

```
>>> components.topic3.sort_values(ascending=False)[:10]
!       394.952246
.       218.049724
to      119.533134
u       118.857546
call    111.948541
£       107.358914
,        96.954384
*        90.314783
your     90.215961
is       75.750037
```

Десять наиболее важных токенов этой темы относятся к словам, применяемым в эмоциональных распоряжениях, требующих от кого-либо сделать/заплатить что-либо. Интересно выяснить, используется ли эта тема чаще в спамовых сообщениях или неспамовых. Как видите, распределение слов по темам вполне объяснимо и доступно анализу даже при таком беглом взгляде.

Перед подгонкой классификатора LDiA, необходимо вычислить эти векторы тем для всех документов (CMC). Посмотрим, насколько они отличаются от векторов тем, сгенерированных SVD и PCA для тех же документов:

```
>>> ldia16_topic_vectors = ldia.transform(bow_docs)
>>> ldia16_topic_vectors = pd.DataFrame(ldia16_topic_vectors,\
...     index=index, columns=columns)
>>> ldia16_topic_vectors.round(2).head()
      topic0   topic1   topic2   ...   topic13   topic14   topic15
sms0    0.00     0.62     0.00   ...      0.00      0.00      0.00
sms1    0.01     0.01     0.01   ...      0.01      0.01      0.01
sms2!   0.00     0.00     0.00   ...      0.00      0.00      0.00
sms3    0.00     0.00     0.00   ...      0.00      0.00      0.00
sms4    0.39     0.00     0.33   ...      0.00      0.00      0.00
```

Все эти темы разделены более четко. В нашем распределении тем по сообщениям множество нулей. Именно благодаря этому темы LDiA гораздо проще объяснить коллегам при принятии бизнес-решений на основе результатов работы конвейера NLP.

Итак, для людей темы LDiA отлично подходят. Как насчет машин? Насколько хорошо отработает наш классификатор LDA с ними?

4.5.3. LDiA + LDA = классификатор спама

Взглянем, насколько хорошо эти темы LDiA подходят для предсказания чего-нибудь полезного, например того, является ли сообщение спамом. Мы воспользуемся векторами тем LDiA для обучения модели LDA (аналогично тому, как мы делали с векторами тем PCA[1]):

```
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

>>> X_train, X_test, y_train, y_test =
➥ train_test_split(ldia16_topic_vectors, sms.spam, test_size=0.5,
➥ random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda = lda.fit(X_train, y_train)
>>> sms['ldia16_spam'] = lda.predict(ldia16_topic_vectors)
>>> round(float(lda.score(X_test, y_test)), 2)
0.94
```

Точность 94 % на тестовом наборе данных — довольно неплохой результат, хотя не настолько хороший, как у LSA (PCA) в подразделе 4.7.1

Определитель нашей матрицы ldia_topic_vectors близок к нулю, так что вы, вероятно, увидите предупреждение Variables are collinear (Переменные коллинеарны). Появляется при использовании LDiA в случае маленького корпуса, поскольку в векторах тем содержится множество нулей и некоторые из сообщений можно представить в виде линейной комбинации тем других сообщений. Или при наличии СМС с одинаковой (или близкой) смесью тем

Алгоритмы `train_test_split()` и LDiA — стохастические. Так что при каждом запуске результаты и показатели точности будут различными. Чтобы результаты конвейера стали воспроизводимыми, необходимо задавать для этих моделей и разбиения набора данных одно и то же значение аргумента `seed` при каждом запуске.

Один из случаев, когда может появиться предупреждение о коллинеарности, — если текст содержит несколько би- или триграмм, которые составляют слова, встречающиеся только вместе. Так что итоговой модели LDiA придется делить веса между этими эквивалентными частотностями термов произвольным образом. Сможете найти в своих СМС подобные вызывающие коллинеарность слова (с нулевым определителем)? Вам нужно искать слово, которое всегда сопровождает в том же сообщении другое (парное ему) слово.

Такой поиск можно произвести с помощью Python, а не вручную. Прежде всего поискать в корпусе идентичные векторы мультимножеств. Они возможны и для неидентичных СМС, например: *Hi there Bob!* и *Bob, Hi there*, поскольку значения количеств слов в них одинаковы. Вы можете пройти в цикле по всем парам из мультимножества слов в поиске идентичных векторов. Они уж точно вызовут предупреждение о коллинеарности как в LDiA, так и в LSA.

---

[1] Для работы этого кода необходимо также импортировать train_test_split: from sklearn.model_selection import train_test_split. — *Примеч. пер.*

Если найти точные копии векторов BOW не удастся, можно пройти в цикле по всем парам слов в словаре: по всем мультимножествам слов в поиске пар СМС, содержащих одни и те же два терма. Если эти слова никогда не встречаются в СМС по отдельности, значит, вы нашли в своем наборе данных одну из искомых коллинеарностей. В число распространенных биграмм, вызывающих этот эффект, входят имена и фамилии известных людей, которые всегда встречаются совместно и никогда не используются по отдельности, например *Bill Gates*. Конечно, если в ваших СМС не встречается других людей по имени *Bill*.

СОВЕТ

При необходимости пройти в цикле по всем сочетаниям (парам или тройкам) множества объектов можно воспользоваться встроенной функцией product() языка Python:

```
>>> from itertools import product
>>> all_pairs = [(word1, word2) for (word1, word2) in product(word_list,
➥ word_list) if not word1 == word2]
```

Наша точность на тестовом наборе данных составила более 90 %, и это при обучении лишь на половине имеющихся данных. Но мы получили предупреждение о коллинеарности наших признаков из-за ограниченности набора данных, что вызывает проблему недоопределенности у LDA. Определитель нашей матрицы «тема — документ» близок к нулю, поскольку мы отбросили половину документов с помощью метода `train_test_split`. При необходимости можно уменьшить значение `n_components`, чтобы решить данную проблему, но при этом темы, являющиеся линейной комбинацией друг друга (коллинеарные), будут объединяться.

Сравним нашу модель LDiA с моделью более высокой размерности, основанной на векторах TF-IDF. Наши векторы TF-IDF содержат намного больше признаков (более 3000 уникальных термов), поэтому вполне возможны переобучение и плохое обобщение. В этом случае пригодится обобщение LDiA и PCA:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from nltk.tokenize.casual import casual_tokenize
>>> tfidf = TfidfVectorizer(tokenizer=casual_tokenize)
>>> tfidf_docs = tfidf.fit_transform(raw_documents=sms.text).toarray()
>>> tfidf_docs = tfidf_docs - tfidf_docs.mean(axis=0)

>>> X_train, X_test, y_train, y_test = train_test_split(tfidf_docs,\
...     sms.spam.values, test_size=0.5, random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda = lda.fit(X_train, y_train)
>>> round(float(lda.score(X_train, y_train)), 3)
1.0
>>> round(float(lda.score(X_test, y_test)), 3)
0.748
```

Притворяемся, что во всех СМС только одна тема, поскольку нас интересует только скалярный показатель для темы «спамовость»

Подгонка модели LDA ко всем этим тысячам признаков занимает немало времени. Потерпите, происходит разделение нашего векторного пространства 9232-мерной гиперплоскостью!

Точность на тренировочном наборе данных для основанной на векторах модели TF-IDF идеальная! Но точность на тестовом наборе данных намного хуже, чем при ее обучении на векторах более низкой размерности вместо векторов TF-IDF.

При этом точность на тестовом наборе данных имеет большое значение. Здесь нам понадобится тематическое моделирование (LSA). Оно помогает производить обобщение моделей с маленького тренировочного набора данных, так что хорошо работает и на сообщениях с различными сочетаниями слов (но схожими темами).

4.5.4. Более честное сравнение: 32 темы LDiA

Попробуем LDiA еще раз при большем числе измерений и тем. Возможно, LDiA не так эффективен, как LSA (PCA), и ему требуется большее число тем, по которым можно распределить слова. Попробуем вариант с 32 темами (компонентами):

```
>>> ldia32 = LDiA(n_components=32, learning_method='batch')
>>> ldia32 = ldia32.fit(bow_docs)
>>> ldia32.components_.shape
(32, 9232)
```

Теперь вычислим новые 32-мерные векторы тем для всех документов (CMC):

```
>>> ldia32_topic_vectors = ldia32.transform(bow_docs)
>>> columns32 = ['topic{}'.format(i) for i in range(ldia32.n_components)]
>>> ldia32_topic_vectors = pd.DataFrame(ldia32_topic_vectors, index=index,\
...     columns=columns32)
>>> ldia32_topic_vectors.round(2).head()
       topic0   topic1   topic2   ...   topic29   topic30   topic31
sms0     0.00      0.5      0.0   ...       0.0       0.0       0.0
sms1     0.00      0.0      0.0   ...       0.0       0.0       0.0
sms2!    0.00      0.0      0.0   ...       0.0       0.0       0.0
sms3     0.00      0.0      0.0   ...       0.0       0.0       0.0
sms4     0.21      0.0      0.0   ...       0.0       0.0       0.0
```

Как видите, эти темы еще более разрежены и четко разделены.

А вот код обучения нашей LDA-модели (классификатора). На этот раз с 32-мерными векторами тем LDiA:

```
>>> X_train, X_test, y_train, y_test =
➥ train_test_split(ldia32_topic_vectors, sms.spam, test_size=0.5,
➥ random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda = lda.fit(X_train, y_train)
>>> sms['ldia32_spam'] = lda.predict(ldia32_topic_vectors)
>>> X_train.shape
(2418, 32)
>>> round(float(lda.score(X_train, y_train)), 3)
0.924
>>> round(float(lda.score(X_test, y_test)), 3)
0.927
```

.shape — еще один способ проверить количество измерений в векторах тем

Главное — точность на тестовом наборе, и 92,7 % — точность, вполне сравнимая с 94 % при 16-мерных векторах тем LDiA

Не путайте эту оптимизацию числа тем (компонент) с изложенной выше проблемой коллинеарности. Увеличение/уменьшение числа тем не решает/создает проблему коллинеарности. Источник этой проблемы кроется в исходных данных. Чтобы избавиться от этого предупреждения, необходимо добавить шум (метаданные) в виде искусственно созданных слов в СМС или удалить повторяющиеся векторы слов. Никакое количество тем не решит проблемы наличия векторов слов или пар слов, часто повторяющихся в документах.

Большее число тем повышает определенность относительно тем и, по крайней мере для данного набора, генерирует лучше линейно разделяемые темы. Но до 96 % точности PCA + LDA еще далеко. Так, PCA лучше рассредоточивает векторы тем СМС, увеличивая промежутки между сообщениями для разделения классов гиперплоскостью.

Не стесняйтесь заглянуть в исходный код моделей размещения Дирихле, доступный как в Scikit-Learn, так и библиотеке gensim. Их API схоже с LSA (см. `sklearn.TruncatedSVD` и `gensim.LsiModel`). Мы продемонстрируем пример приложения при обсуждении автоматического реферирования в последующих главах. LDiA хорошо подходит для поиска объяснимых тем вроде используемых для автоматического реферирования. И неплохо создает пригодные для линейной классификации темы.

Глубже изучаем имеющийся инструментарий

Путь к исходному коду можно найти в атрибуте `__file__` любого модуля Python, например `sklearn.__file__`, а в IPython (консоли jupyter) можно просмотреть исходный код любой функции, класса или объекта с помощью команды `??`, например `LDA??`:

```
>>> import sklearn
>>> sklearn.__file__
'/Users/hobs/anaconda3/envs/conda_env_nlpia/lib/python3.6/site-packages/skl
earn/__init__.py'
>>> from sklearn.discriminant_analysis\
...     import LinearDiscriminantAnalysis as LDA
>>> LDA??
Init signature: LDA(solver='svd', shrinkage=None, priors=None, n_components
=None, store_covariance=False, tol=0.0001)
Source:
class LinearDiscriminantAnalysis(BaseEstimator, LinearClassifierMixin,
                                 TransformerMixin):
    """Linear Discriminant Analysis

    A classifier with a linear decision boundary, generated by fitting
    class conditional densities to the data and using Bayes' rule.

    The model fits a Gaussian density to each class, assuming that
    all classes share the same covariance matrix.
...
```

Эти команды не работают для функций и классов-расширений, код которых скрыт внутри скомпилированного C++-модуля.

4.6. Расстояние и подобие

Снова обратимся к показателям подобия, о которых мы говорили в главах 2 и 3, и убедимся, что наше новое векторное пространство тем подходит для них. Определить, насколько похожи (далеки) два документа, можно посредством показателей подобия (и расстояний) на основе сходства (расстояния между) представляющих их векторов.

С помощью показателей подобия (и расстояний) можно понять, насколько хорошо модель LSA согласуется с моделью TF-IDF более высокой размерности из главы 3. И как модель сохраняет эти расстояния после исключения больших объемов информации, содержащейся в мультимножествах намного более высокой размерности. Можно проверить, насколько удалены друг от друга векторы тем и какую роль играет качество представления расстояния между предметами обсуждения документов. Нам нужно убедиться, что схожие по смыслу документы близки друг к другу в нашем новом векторном пространстве тем.

LSA сохраняет большие расстояния, но не всегда — малые (тонкую структуру взаимосвязей между документами). Лежащий в его основе алгоритм SVD ориентирован на максимизацию дисперсии между всеми документами в новом векторном пространстве тем.

Расстояния между векторами признаков (слов, тем, контекста документов и т. д.) — движущая сила эффективности работы конвейера NLP и вообще любого конвейера машинного обучения. Так какие существуют варианты измерения расстояния в многомерном пространстве? И какие из них выбрать для конкретной задачи NLP? Некоторые из этих часто используемых вариантов знакомы вам из курса геометрии или линейной алгебры, в то время как другие, вероятно, в новинку:

- евклидово (декартово) расстояние, оно же корень из среднеквадратической погрешности (root mean square error, RMSE): 2-норма, или L_2;
- квадрат евклидова расстояния, сумма квадратов расстояния (SSD): L_2^2;
- косинусное (угловое, проекционное) расстояние: нормализованное скалярное произведение;
- расстояние Минковского: p-норма (L_p);
- выраженное в дробях расстояние (дробная норма): p-норма (L_p) для $0 < p < 1$;
- метрика городских кварталов (манхэттенская метрика, метрика такси); сумма абсолютных расстояний (sum of absolute distance, SAD): l-норма (L_1);
- расстояние Жаккара (обратное подобие множеств);
- расстояние Махаланобиса;
- расстояние Левенштейна (расстояние редактирования).

Разнообразие способов вычисления расстояния — свидетельство его важности. Помимо реализаций попарного расстояния из Scikit-Learn, применяется и мно-

жество других в таких математических дисциплинах, как топология, статистика и машиностроение[1]. В листинге 4.7 приведены расстояния, реализованные в модуле `sklearn.metrics.pairwise`[2].

Листинг 4.7. Доступные в sklearn попарные расстояния

```
'cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan', 'braycurtis',
'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard',
'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto',
'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean',
'yule'
```

Оценки расстояния часто вычисляются на основе мер (показателей) подобия и наоборот и являются обратно пропорциональными показателям подобия. Показатели подобия устроены так, что их значение находится в диапазоне от 0 до 1. Типичные формулы преобразования:

```
>>> similarity = 1. / (1. + distance)
>>> distance = (1. / similarity) - 1.
```

Но для расстояний и показателей подобия, которые находятся в диапазоне от 0 до 1, чаще используют следующие формулы:

```
>>> similarity = 1. - distance
>>> distance = 1. - similarity
```

У косинусных расстояний есть собственные соглашения о диапазоне значений. Угловое расстояние между двумя векторами часто рассчитывается как доля максимально возможного углового разделения двух векторов, составляющего 180°или π радиан (https://en.wikipedia.org/wiki/Cosine_similarity). В результате косинусный коэффициент и расстояние являются взаимно обратными:

```
>>> import math
>>> angular_distance = math.acos(cosine_similarity) / math.pi
>>> distance = 1. / similarity - 1.
>>> similarity = 1. - distance
```

Термины «расстояние» и «длина» часто путают со словом «метрика», поскольку многие меры расстояния и длины — вполне допустимые и удобные метрики. Но, к сожалению, не все расстояния можно так назвать. Еще более запутывает дело то, что метрики иногда называют функциями расстояния или метриками расстояния в текстах по формальной математике и теории множеств (https://en.wikipedia.org/wiki/Metric_(mathematics)).

---

[1] Больше метрик расстояния вы можете найти в библиотеке Math.NET Numerics по адресу https://numerics.mathdotnet.com/Distance.html.

[2] См. документацию модуля sklearn.metrics.pairwise по адресу http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise_distances.html.

Метрики

Настоящая метрика должна обладать четырьмя математическими свойствами, которые отсутствуют у расстояний и «показателей» (scores).

- Неотрицательность: метрика не может принимать значения меньше нуля: `metric(A, B) >= 0`.
- Неразличимость: два объекта совпадают, если метрика между ними равна 0: `if metric(A, B) == 0: assert(A == B)`.
- Симметрия — для метрики неважно направление: `metric(A, B) = metric(B, A)`.
- Неравенство треугольника — длина любой стороны треугольника никогда не больше суммы длин двух других его сторон: `metric(A, C) <= metric(A, B) + metric(B, C)`.

У соответствующего математического термина *«мера»* (measure) есть как смысл в обычном языке, так и строгое математическое определение. Слово «мера» можно найти в словаре Ожегова и в глоссарии учебника по математике, причем определения будут совершенно разными. Так что будьте внимательны при разговоре с преподавателем математики.

Для преподавателя математики мера — это размер множества математических объектов. Объект `set` языка Python можно измерить его длиной, но многие математические множества бесконечны. Причем в теории множеств существуют различные виды бесконечности, а меры — это различные способы вычисления `len()` или размера математического множества, разновидности бесконечности.

ОПРЕДЕЛЕНИЕ

Как и у термина «метрика», у слова «мера» есть четкое математическое определение, связанное с размером набора объектов. Так что слова «мера» следует осмотрительно использовать при описании различных (статистических) показателей, полученных из объектов или сочетаний объектов в NLP (https://ru.wikipedia.org/wiki/Мера_множества).

Но на практике приходится измерять самые различные вещи. Глагол «измерять» может означать использование рулетки, линейки, весов или оценки в баллах для измерения чего-либо. Именно в этом смысле и будет применяться слово «измерять» в этой книге, хотя мы постараемся вообще его не употреблять, чтобы наши преподаватели математики нас не ругали.

4.7. Стиринг и обратная связь

Все вышеописанные подходы к LSA не учитывали информацию о подобии документов. Мы создавали темы, оптимально подходящие к общему набору правил. При нашем машинном обучении без учителя у этих моделей выделения признаков (тем) не было никакой информации о том, насколько близко должны находиться

векторы тем друг к другу. Мы не предоставляли никакой обратной связи относительно того, где оказались векторы или как они связаны друг с другом. Стиринг[1] (он же «усвоение метрик расстояния»[2]) — новейший способ понижения размерности и выделения признаков. Подстройка оценок расстояний, передаваемых алгоритмам кластеризации и вложения, позволяет направить свои векторы на минимизацию какой-либо функции стоимости. Таким образом, можно сосредоточить внимание векторов на каком-либо интересующем вас аспекте информационного содержания.

В предыдущих разделах, посвященных LSA, мы игнорировали всю метаинформацию о документах. Например, в случае СМС мы не учитывали отправителя сообщения, а это хороший индикатор сходства тем, который может пригодиться в качестве источника информации для нашего преобразования векторов тем (LSA).

В компании Talentpair мы проводили опыты по поиску соответствий резюме описаниям вакансий на основе косинусного расстояния между векторами тем для каждого из документов. Этот метод отлично работал. Мы вскоре поняли, что результаты значительно улучшаются, если направлять векторы тем на основе обратной связи от кандидатов и менеджеров по работе с клиентами, помогающих им в поиске работы. Векторы для «хороших пар» направлялись ближе друг к другу, чем остальные пары.

Один из способов добиться этого — вычислить среднюю разность между нашими двумя центроидами (как мы делали в LDA) и добавить некоторую долю этого «смещения» ко всем векторам резюме или описаний вакансий. Это приведет к вычитанию средней разности векторов тем между резюме и описаниями вакансий. Темы вроде «бочковое пиво за обедом» могут встречаться в описаниях вакансий, но никогда в резюме. Аналогично всякие странные хобби, вроде подводной скульптуры, могут встречаться в некоторых резюме, но никогда в описании вакансии. С помощью стиринга можно сфокусировать векторы на темах, которые вам интересно смоделировать.

Если вам интересен вопрос уточнения векторов тем, исключения систематической ошибки, можете поискать в Google Scholar (http://scholar.google.com/) фразы *learned distance/similarity metric* («усвоение метрик расстояния/метрика подобия») и *distance metrics for nonlinear embeddings* («метрики расстояния для нелинейных вложений»)[3]. К сожалению, эта возможность пока не реализована ни в одном модуле библиотеки Scikit-Learn. Вы будете настоящим героем, если найдете время внести в проект Scikit-Learn предложения о добавлении возможности стиринга или даже код для него (http://github.com/scikit-learn/scikit-learn/issues).

---

[1] От английского to steer — «направлять в нужное русло, указывать курс». — *Примеч. пер.*

[2] См. статью Superpixel Graph Label Transfer with Learned Distance Metric по адресу http://users.cecs.anu.edu.au/~sgould/papers/eccv14-spgraph.pdf.

[3] См. статью Distance Metric Learning: A Comprehensive Survey по адресу https://www.cs.cmu.edu/~liuy/frame_survey_v2.pdf.

4.7.1. Линейный дискриминантный анализ

Обучим модель линейного дискриминантного анализа на наших маркированных СМС. LDA работает аналогично LSA, за исключением того, что ему необходимы метки классификации или какие-либо другие показатели для поиска наилучшего линейного сочетания измерений в многомерном пространстве (термов в векторах BOW или TF-IDF). Вместо максимизации разделения (дисперсии) векторов в этом новом пространстве LDA максимизирует расстояние между центроидами векторов каждого из классов.

К сожалению, это значит, что алгоритму LDA требуется информация в виде примеров (маркированных векторов) о том, какие темы мы хотели бы смоделировать. Только в этом случае алгоритм может вычислить оптимальное преобразование из вашего многомерного пространства в пространство более низкой размерности, а количество измерений в итоговом векторе низкой размерности не может превышать числа передаваемых алгоритму меток или показателей классов. Поскольку для обучения у нас есть только тема «спамовость», взглянем, насколько точным окажется наша одномерная тематическая модель в классификации спамовых СМС:

```
>>> lda = LDA(n_components=1)
>>> lda = lda.fit(tfidf_docs, sms.spam)
>>> sms['lda_spaminess'] = lda.predict(tfidf_docs)
>>> ((sms.spam - sms.lda_spaminess) ** 2.).sum() ** .5
0.0
>>> (sms.spam == sms.lda_spaminess).sum()
4837
>>> len(sms)
4837
```

Все до единого — правильно! Хотя погодите... Что мы там раньше говорили о переобучении? При 10 000 термов в векторах TF-IDF ничего удивительного, что алгоритм просто запомнил правильный ответ. Произведем кросс-валидацию:

```
>>> from sklearn.model_selection import cross_val_score
>>> lda = LDA(n_components=1)
>>> scores = cross_val_score(lda, tfidf_docs, sms.spam, cv=5)
>>> "Accuracy: {:.2f} (+/-{:.2f})".format(scores.mean(), scores.std() * 2)
'Accuracy: 0.76 (+/-0.03)'
```

Очевидно, что это не слишком хорошая модель. Пусть это будет вам напоминанием: никогда не следует радоваться хорошим показателям работы модели на тренировочном наборе данных.

Чтобы убедиться, что точность действительно составляет 76 %, выделим треть набора данных для контроля:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(tfidf_docs,\
...     sms.spam, test_size=0.33, random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda.fit(X_train, y_train)
LinearDiscriminantAnalysis(n_components=1, priors=None, shrinkage=None,
              solver='svd', store_covariance=False, tol=0.0001)
>>> lda.score(X_test, y_test).round(3)
0.765
```

На тестовом наборе данных точность плохая. Так что дело не в том, что нам не повезло с выборкой данных. Это плохая переобученная модель.

Взглянем, удастся ли нам создать с помощью LSA в сочетании с LDA точную модель, которая хорошо бы обобщалась, чтобы новые СМС не вызвали у нее сложностей[1]:

```
>>> X_train, X_test, y_train, y_test =
➥ train_test_split(pca_topicvectors.values, sms.spam, test_size=0.3,
➥ random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda.fit(X_train, y_train)
LinearDiscriminantAnalysis(n_components=1, priors=None, shrinkage=None,
              solver='svd', store_covariance=False, tol=0.0001)
>>> lda.score(X_test, y_test).round(3)
0.965
>>> lda = LDA(n_components=1)
>>> scores = cross_val_score(lda, pca_topicvectors, sms.spam, cv=10)
>>> "Accuracy: {:.3f} (+/-{:.3f})".format(scores.mean(), scores.std() * 2)
'Accuracy: 0.958 (+/-0.022)'
```

В случае LSA СМС можно охарактеризовать всего 16 измерениями. Еще останется немало информации для классификации его как спама (или не спама), а вероятность переобучения нашей модели низкой размерности гораздо меньше. Она должна хорошо обобщаться и классифицировать еще не виденные ею СМС и интернет-переписку.

Мы вернулись в исходную точку: к нашей простой модели из начала данной главы. Точность этой простой LDA-модели была выше еще до всего этого семантического анализа. Но преимущество новой модели в том, что теперь можно создавать векторы, отражающие семантику высказываний в более чем одном измерении.

4.8. Мощь векторов тем

С помощью векторов тем можно сравнивать значения слов, документов, высказываний и корпусов. Можно находить кластеры похожих документов и высказываний. Мы уже не сравниваем расстояние между документами на основе только употребляемых в них слов. Мы теперь не ограничены одним поиском по ключевым словам и ранжированием релевантности на основе только выбора слов или словаря. Теперь мы можем находить документы, соответствующие запросу, а не просто хорошо подходящие к статистике слов.

---

[1] В первой части этого фрагмента кода содержится несколько ошибок. Работоспособный вариант выглядит следующим образом (его можно найти в прилагаемом к книге коде):
pca = PCA(n_components=16)
pca = pca.fit(tfidf_docs)
pca_topicvectors = pca.transform(tfidf_docs)
X_train, X_test, y_train, y_test = train_test_split(pca_topicvectors, sms.spam, test_size=0.3, random_state=271828)
lda = LDA(n_components=1)
lda.fit(X_train, y_train). — *Примеч. пер.*

Это называется семантическим поиском (не путайте с семантически структурированной сетью[1]). Семантический поиск реализуют мощные поисковые системы, возвращая пользователю документы, не содержащие многих слов из запроса, но в точности соответствующие искомому. Подобные продвинутые поисковые механизмы используют векторы тем LSA, чтобы отличить пакет *Python* из «магазина сыра»[2] и питона из аквариума магазина домашних животных во Флориде, распознавая при этом его сходство с *Ruby gem*[3].

Семантический поиск — инструмент для поиска и генерации осмысленного текста. Но наш мозг плохо подходит для работы с многомерными объектами, векторами, гиперплоскостями, гиперсферами и гиперкубами. Наших интуитивных представлений как разработчиков и специалистов по машинному обучению хватает только на три измерения.

Например, с помощью запроса по двумерному вектору, предположим по долготе/широте на картах Google, можно быстро найти все кофейни поблизости. Можно просто просмотреть (глазами или программно) по спирали объекты возле нужного местоположения. Или создать с помощью кода ограничивающую рамку и искать в определенных диапазонах долготы и широты. Произвести подобное в гиперпространстве, формируя границы поиска гиперплоскостями и гиперкубами, невозможно.

Как сказал Джеффри Хинтон (Geoffry Hinton), «для работы с гиперплоскостями в 14-мерном пространстве представьте себе 3D-пространство и скажите себе громко: 14». Если вы читали в юном и впечатлительном возрасте написанный в 1884 году Эдвином Эбботом (Edwin Abbott) роман «Флатландия», то вам это удастся немного лучше. Возможно, вы даже сможете высунуть голову из трехмерного окошка в гиперпространство ровно настолько, чтобы увидеть мельком, как выглядит 3D-пространство снаружи. Как и во «Флатландии», в этой главе было приведено множество визуализаций, чтобы продемонстрировать вам «тени», отбрасываемые словами из гиперпространства в нашем трехмерном мире. Если вам не терпится с ними познакомиться, переходите прямо к разделу, где показаны матрицы рассеяния векторов слов. Имеет смысл также заглянуть назад и посмотреть на вектор трехмерного мультимножества из предыдущей главы и попробовать представить себе, как бы эти точки выглядели, если добавить еще всего одно слово в словарь для создания 4D-мира языковых смыслов.

Если вы глубоко задумались о четырех измерениях, не забывайте, что рост сложности при этом еще больше, чем при переходе от двумерного пространства к 3D, и экспоненциально больше, чем при переходе от одномерного пространства вещественных чисел к 2D-пространству треугольников, квадратов и кругов.

---

[1] Семантически структурированная сеть — способ структурной организации текстов на естественном языке с помощью тегов в HTML-документах таким образом, что иерархия тегов и их содержимое указывают на связи (сеть взаимосвязей) между элементами (текстом, изображениями, видеофайлами) на веб-странице.

[2] «Магазином сыра» (Cheese shop) называют индекс PyPi пакетов Python в честь одноименного скетча группы «Монти Пайтон». — *Примеч. пер.*

[3] Ruby — язык программирования, в котором есть пакет gem.

ПРИМЕЧАНИЕ

Взрывной рост числа возможностей при переходе от одномерных линий, двумерных прямоугольников, трехмерных кубов и т. д. проходит мимо безумных вселенных нецелочисленных фрактальных размерностей вроде 1,5-мерных фракталов. Длина 1,5-мерного фрактала бесконечна, и он полностью заполняет двумерную плоскость, хотя сам является менее чем двумерным![1] К счастью, это не «настоящие» измерения[2]. Так что можете не волноваться за них при использовании NLP. Если, конечно, вас не интересуют дробные метрики расстояния, например p-norm, в формулах которых присутствуют нецелочисленные показательные функции[3].

4.8.1. Семантический поиск

Поиск документа по содержащемуся в нем слову или части слова называется *полнотекстовым поиском* (full text search). Именно так работают поисковые системы. Они разбивают документ на фрагменты (чаще всего слова), которые можно проиндексировать с помощью *инвертированного индекса*[4] (inverted index), вроде того, который обычно можно встретить в конце учебников. Это требует хранения большого объема дополнительной информации и догадок из-за орфографических и типографских ошибок, но обычно работает неплохо[5].

Семантический поиск (semantic search) — тип полнотекстового поиска, при котором учитывается смысл слов в запросе и искомых документах. В этой главе мы показали вам два способа — LSA и LDiA — вычисления векторов тем, захватывающих в векторе семантику (смысл) слов и документов. Одна из причин, по которым латентно-семантический анализ сначала называли латентно-семантической *индексацией* — перспективы поддержки им возможностей семантического поиска с помощью индекса числовых значений, например таблиц BOW и TF-IDF. Семантический поиск был следующим серьезным достижением в сфере информационного поиска.

В отличие от таблиц BOW и TF-IDF таблицы семантических векторов плохо поддаются дискретизации и индексации с помощью традиционных методик обратных индексов. Традиционные способы дискретизации ориентированы на работу с бинарными векторами встречаемости слов, дискретными векторами (BOW), разреженными непрерывными векторами (векторы TF-IDF) и непрерывными векторами низкой размерности (3D-данные GIS). Но многомерные непрерывные векторы, например тем

---

1 Дробные размерности (http://www.math.cornell.edu/~erin/docs/research-talk.pdf).

2 Фрактальные измерения (http://www.askamathematician.com/2012/12/q-what-are-fractional-dimensions-can-space-have-a-fractional-dimension/).

3 The Concentration of Fractional Distances по адресу https://perso.uclouvain.be/michel.verleysen/papers/tkde07df.pdf.

4 В русскоязычной литературе также называется обратным индексом. — *Примеч. пер.*

5 Полнотекстовый поиск в базах данных вроде PostgreSQL обычно производится на основе триграмм символов, чтобы можно было справиться с орфографическими ошибками и текстом, который не получается разобрать на слова.

из LSA или LDiA, представляют собой непростую проблему[1]. Обратные индексы подходят для дискретных и бинарных векторов, например таблиц бинарных или целочисленных векторов «слово — документ», поскольку при этом в индексе требуются лишь записи для всех ненулевых дискретных измерений. Значение для конкретного измерения есть в соответствующем векторе/документе или нет. Поскольку векторы TF-IDF — разреженные, почти полностью состоящие из нулей, то записи в индексе для большинства измерений большей части документов вообще не требуется[2].

В результате работы LSA (и LDiA) получаются многомерные, непрерывные и плотные (нули в них встречаются редко) векторы тем. Алгоритмы семантического анализа не генерируют эффективного индекса для хорошо масштабируемого поиска. Точный индекс невозможен из-за проклятия размерности, о котором мы говорили в предыдущем разделе. Относящаяся к индексации часть латентно-семантической индексации оказалась лишь мечтой, так что термин LSI неправилен.

Возможно, именно поэтому более распространенным способом описания генерирующих векторы тем алгоритмов семантического анализа стал термин LSA.

Одно из решений проблемы многомерных векторов — их индексация с помощью *хеширования с учетом локальности* (locality sensitive hash, LSH). Хеш с учетом локальности напоминает почтовый индекс, обозначающий область гиперпространства таким образом, чтобы ее потом было легко найти. Как и обычный хеш, он дискретный и зависит только от значений в векторе. Но даже этот метод плохо работает при более чем 12 измерениях. На рис. 4.6 каждая строка соответствует размеру (размерности) вектора темы, начиная с двух измерений и заканчивая 16, как в векторах, использовавшихся выше для задачи классификации спама.

Таблица демонстрирует качество результатов поиска при использовании хеширования с учетом локальности для индексации большого числа семантических векторов. Если в векторе больше 16 измерений, получить хотя бы пару хороших результатов поиска будет непросто.

Как же производить семантический поиск на 100-мерных векторах без индекса? Мы знаем, как преобразовать строку запроса в вектор темы с помощью LSA. И знаем, как сравнить два вектора с помощью показателя косинусного подобия (скалярное или внутреннее произведение) и найти самый близкий. Для поиска семантических соответствий необходимо найти все векторы тем документов, ближайшие к конкретному вектору темы (поискового) запроса. Но для n документов придется произвести n сравнений с вектором темы запроса. Это немалое число скалярных произведений.

Эту операцию можно векторизовать в NumPy путем умножения матриц, но число операций при этом не уменьшится, просто они будут выполнены в 100 раз быстрее[3]. Точный семантический поиск все равно требует O(N) умножений и сложений для

---

[1] Кластеризация многомерных данных эквивалентна дискретизации или индексации многомерных данных с помощью ограничивающих рамок и описана в статье: Clustering high dimensional data по адресу https://en.wikipedia.org/wiki/Clustering_high-dimensional_data.

[2] См. веб-страницу https://ru.wikipedia.org/wiki/Инвертированный_индекс.

[3] Векторизация кода на языке Python, особенно вложенных циклов for для вычислений попарных расстояний, может ускорить работу кода почти в 100 раз. См. статью в журнале Hacker Noon по адресу https://hackernoon.com/speeding-up-your-code-2-vectorizing-the-loops-with-numpy-e380e939bed3.

каждого запроса. Так что он масштабируется линейно по отношению к размеру корпуса. Для большого корпуса, например для поиска Google или семантического поиска по «Википедии», это не подходит.

Измерений	100-е косинусное расстояние	Топ-1 правильно	Топ-2 правильно	Топ-10 правильно	Топ-100 правильно
2	0,00	ДА	ДА	ДА	ДА
3	0,00	ДА	ДА	ДА	ДА
4	0,00	ДА	ДА	ДА	ДА
5	0,01	ДА	ДА	ДА	ДА
6	0,02	ДА	ДА	ДА	ДА
7	0,02	ДА	ДА	ДА	НЕТ
8	0,03	ДА	ДА	ДА	НЕТ
9	0,04	ДА	ДА	ДА	НЕТ
10	0,05	ДА	ДА	НЕТ	НЕТ
11	0,07	ДА	ДА	ДА	НЕТ
12	0,06	ДА	ДА	НЕТ	НЕТ
13	0,09	ДА	ДА	НЕТ	НЕТ
14	0,14	ДА	НЕТ	НЕТ	НЕТ
15	0,14	ДА	ДА	НЕТ	НЕТ
16	0,09	ДА	ДА	НЕТ	НЕТ

Рис. 4.6. Точность семантического поиска резко падает, начиная примерно с 12 измерений

Ключ к решению этой проблемы — согласиться на достаточно хороший результат, а не стремиться к идеальному индексу или алгоритму LSH для наших многомерных векторов. Существует несколько реализаций с открытым исходным кодом эффективных и точных алгоритмов *приближенного поиска ближайших соседей* (approximate nearest neighbors), использующих LSH для эффективной реализации семантического поиска. Два самых простых в использовании и установке:

- разработанный Spotify пакет `annoy`[1];
- класс `gensim.models.KeyedVector` из библиотеки gensim[2].

[1] Исследователи Spotify сравнили производительность их пакета annoy с несколькими альтернативными алгоритмами и реализациями в репозитории на GitHub (https://github.com/spotify/annoy).

[2] Используемый в библиотеке gensim для векторов тем с сотнями измерений подход отлично работает для любых семантических векторов и векторов тем. См. раздел KeyedVectors в документации gensim по адресу https://radimrehurek.com/gensim/models/keyedvectors.html.

Формально эти способы индексации или хеширования не гарантируют нахождения всех наилучших соответствий для семантического поискового запроса. Но они могут предоставить вам неплохой список близких соответствий практически так же быстро, как обычный обратный индекс для вектора TF-IDF или вектора мультимножества, если вы готовы поступиться толикой точности[1].

4.8.2. Дальнейшие усовершенствования

В следующих главах мы расскажем, как усовершенствовать эту концепцию векторов тем и получить более точные и удобные векторы. Для этого сначала обратимся к понятию нейронных сетей. Они позволят повысить возможности вашего конвейера по извлечению смысла из коротких текстов или даже отдельных слов.

Резюме

- С помощью метода SVD можно производить семантический анализ для разложения и преобразования векторов TF-IDF и BOW в векторы тем.
- Если нужно вычислить объяснимые векторы тем, воспользуйтесь методом LDiA.
- Векторы тем, вне зависимости от того, как они получены, можно использовать для семантического поиска документов по смыслу.
- Векторы тем можно применять для предсказания того, является ли сообщение в социальной сети спамом, или вероятности, что оно «понравится» другим пользователям.
- Теперь вы знаете, как обойти проклятие размерности и найти приближенных ближайших соседей векторов в семантическом векторном пространстве.

[1] Если вас интересуют способы более быстрого поиска ближайших соседей многомерного вектора, загляните в приложение E или воспользуйтесь пакетом annoy для индексации своих векторов тем.

Часть II

Более глубокое обучение: нейронные сети

В части I мы накопили инструменты для обработки естественного языка и углубились в машинное обучение на основе статических моделей векторного пространства. Мы узнали, что по статистике связей между словами можно выяснить еще больше смысла слов[1]. Мы рассказали вам про такие алгоритмы, как латентно-семантический анализ, с помощью которых можно выяснить смысл этих связей за счет группировки слов по темам.

Но в части I речь шла о линейных связях между словами. И вам часто приходилось использовать свой здравый смысл для создания процедуры выделения признаков и выбора параметров модели. Нейронные сети из части II берут львиную долю этого утомительного выделения признаков на себя, а модели из части II нередко оказываются точнее, чем те, которые можно создать с помощью настраиваемых вручную процедур выделения признаков из части I.

Использование многослойных нейронных сетей для машинного обучения называется *глубоким обучением* (deep learning). Этот новый подход к NLP и моделированию мышления человека философы и специалисты в области нейронаук часто называют коннекционизмом (connectionism)[2]. Доступность глубокого обучения благодаря удешевлению вычислительных ресурсов и богатой экосистеме открытого исходного кода — ключ к более глубокому пониманию языка. В части II мы начнем раскрывать «черный ящик» глубокого обучения и научимся нелинейному и более глубокому моделированию текста.

Сперва изучим азбуку нейронных сетей, затем — несколько различных видов нейронных сетей, обсудим возможности их применения к NLP. Мы также начнем исследовать закономерности не только между словами, но и между отдельными символами слов. Наконец мы покажем, как с помощью машинного обучения по-настоящему генерировать новый текст.

---

[1] Условная вероятность (conditional probability) — один из вариантов таких статистических показателей взаимосвязей (отражает частоту встречаемости слова при условии, что перед ним или после него встречаются другие слова). Еще один такой показатель — взаимная корреляция (cross correlation, функция правдоподобия для совместной встречаемости слов). Для группировки слов в темы, линейные комбинации количеств слов, могут использоваться сингулярные значения и сингулярные векторы матрицы «слово — документ».

[2] См. статью Connectionism в Стэнфордской философской энциклопедии по адресу https://plato.stanford.edu/entries/connectionism.

5 Первые шаги в нейронных сетях: перцептроны и метод обратного распространения ошибки

В этой главе

- История нейронных сетей.
- Построение многослойной структуры перцептронов.
- Обратное распространение ошибки.
- Управляющие элементы нейронных сетей.
- Реализация простейшей нейронной сети в Keras.

За последние годы было поднято немало шумихи вокруг перспектив нейронных сетей и их возможностей классифицировать/распознавать данные, а совсем недавно — и по поводу способностей определенных архитектур сетей генерировать новый контент. Большие и маленькие компании применяют их для чего угодно, начиная от генерации подписей к изображениям и навигации беспилотных автомобилей и до идентификации солнечных батарей по спутниковым фотографиям и распознавания лиц на видео с камер наблюдения. К счастью, существует множество способов применения нейронных сетей в сфере NLP. Хотя глубокие нейронные сети вызвали

множество разговоров и преувеличений, до власти над нами роботов, вероятно, еще очень далеко, гораздо дальше, чем то признают статьи с сенсационными заголовками. Впрочем, нейронные сети — весьма мощный инструмент. Их удобно использовать в конвейере NLP чат-бота для классификации входного текста, автореферирования документов и даже для генерации новых текстов.

Эта глава задумана как азбука для тех, кто еще не сталкивался с нейронными сетями. Здесь не будет ничего связанного конкретно с NLP, но базовое понимание происходящего внутри нейронных сетей важно для последующих глав. Если вы уже знакомы с основами, то можете спокойно пропустить эту главу и перейти к следующей, посвященной обработке текста с помощью разнообразных нейронных сетей. Хотя лежащая в их основе математика, *метод обратного распространения ошибки*, выходит за рамки данной книги, общий обзор его функциональности поможет лучше разобраться в терминологии и скрытых в ней закономерностях.

СОВЕТ

Советуем две другие книги по глубокому обучению:

- *Шолле Ф.* Глубокое обучение на Python. — СПб.: Питер, 2018. Углубленный обзор чудес глубокого обучения лично от создателя Keras.
- *Траск Э.* Грокаем глубокое обучение. — СПб.: Питер, 2019. Расширенный обзор моделей и практик глубокого обучения.

5.1. Нейронные сети, список ингредиентов

Доступность вычислительных ресурсов и памяти резко возросла за последний десяток лет, и одна старая технология снова стала актуальна. Перцептрон[1], впервые предложенный в 1950-х Фрэнком Розенблаттом (Frank Rosenblatt), стал новым алгоритмом для поиска закономерностей в данных.

Основная идея состоит в подражании работе живой клетки-нейрона. Через *дендриты* (dendrites) в ее ядро поступают электрические сигналы (рис. 5.1), и начинает накапливаться электрический заряд. Когда он достигает определенного уровня, нейрон *возбуждается* (fire) и посылает электрический сигнал через *аксон* (axon). Не все дендриты одинаковы. К сигналам, поступающим через некоторые дендриты, клетка более чувствительна, чем к другим, так что для возбуждения аксона через них достаточно меньшего заряда.

Разумеется, биологические процессы, определяющие эти взаимосвязи, выходят далеко за рамки данной книги, но основная идея, которая нас интересует, заключается в том, что клетка выбирает, когда возбудиться, с учетом *веса* входящих сигналов. Нейрон динамически меняет эти веса процесса принятия решений за время своей жизни. Мы сымитируем данный процесс.

---

[1] *Rosenblatt F.* The perceptron — a perceiving and recognizing automaton // Report 85–460–1, Cornell Aeronautical Laboratory, 1957.

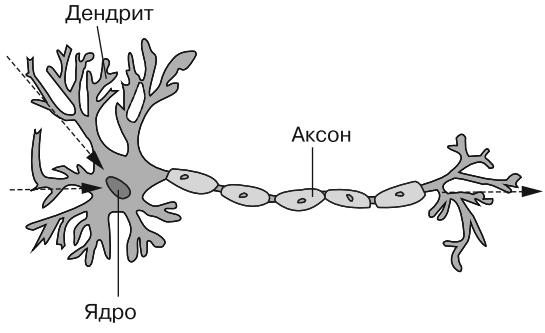


Рис. 5.1. Нейрон

5.1.1. Перцептрон

Целью исходного проекта Розенблатта было научить машину распознавать изображения. Первоначальный перцептрон представлял собой скопление фоторецепторов и потенциометров, а не компьютер в нынешнем понимании этого слова. Если не считать особенностей реализации, идея Розенблатта заключалась в присвоении весов (меры важности) каждому из признаков изображения, которые представляли собой небольшой фрагмент изображения.

Изображение «демонстрировалось» сетке фоторецепторов, отдельный из которых мог «видеть» лишь маленький фрагмент изображения. Сила сигнала, отправляемого конкретным фоторецептором соответствующему дендриту, определялась яркостью видимого им фрагмента изображения.

У каждого дендрита был свой вес в виде потенциометра. При поступлении достаточно мощного сигнала потенциометр пропускал этот сигнал в основное тело «ядра клетки». При превышении суммы этих сигналов от всех потенциометров определенного порогового значения перцептрон возбуждал аксон, указывая на совпадение представленного ему изображения. Отсутствие возбуждения дендрита для заданного изображения классифицировалось как несовпадение. Он работал в режиме «хот-дог, не хот-дог» или «ирис щетинистый, не ирис щетинистый».

5.1.2. Числовой перцептрон

Пока что мы много болтали о биологии, электрическом токе и фоторецепторах. Остановимся и выделим самое важное из всего этого.

Фактически мы хотели бы показать один пример из набора данных алгоритму и получить от того ответ «да» или «нет». Именно это мы и делали. Прежде всего нам нужен способ выделить *признаки* этой выборки. Подбор подходящих признаков оказывается на удивление непростой частью машинного обучения. В «обычных» задачах машинного обучения, например при предсказании цен на недвижимость, признаками могут быть, скажем, площадь в квадратных метрах, последняя цена продажи и почтовый индекс. Или нам может понадобиться предсказать определенный

цветок на основе набора данных Iris[1]. В этом случае признаками будут длина и ширина лепестков, длина и ширина чашелистика.

В опыте Розенблатта признаками служили значения яркости каждого из пикселов (фрагментов изображения), по одному пикселу на фоторецептор. Далее необходим набор присваиваемых признакам *весов*. Не волнуйтесь пока что о том, откуда эти веса берутся. Можете просто считать их процентной долей пропускаемого в нейрон сигнала. Если вам знаком метод линейной регрессии, то вы уже знаете, откуда они берутся[2].

ПРИМЕЧАНИЕ

Обычно отдельные признаки обозначаются x_i, где i — целое число. Набор всех признаков для заданного примера обозначается X, где X — вектор:

$X = [x_1, x_2 \ldots x_i \ldots x_n]$.

И аналогично, веса для признаков обозначаются w_i, где i — индекс соответствующего этому весу признака x. Веса обычно представлены в виде вектора W:

$W = [w_1, w_2 \ldots w_i \ldots w_n]$.

При наличии признаков и весов достаточно умножить признаки на соответствующие веса и просуммировать:

$$(x_1 \cdot w_1) + (x_2 \cdot w_2) + \ldots + (x_i \cdot w_i) + \ldots$$

Не хватает только порогового значения возбуждения нейрона. И это именно пороговое значение. Как только взвешенная сумма превышает определенный порог, перцептрон выдает 1. В противном случае — 0.

Это пороговое значение можно представить в виде простой *ступенчатой функции* (step function), которая на рис. 5.2 названа функцией активации.

5.1.3. Коротко про смещение

На рис. 5.2 и в данном примере упоминается *смещение*[3] (bias). Что это такое? Смещение — это постоянно присутствующий на входе нейрона сигнал. Как и у любого другого элемента входных данных нейрона, у него есть свой вес, который обрабатывается аналогично всем остальным. В различной посвященной нейронным сетям литературе его представляют двумя способами. Например, встречается представ-

---

[1] Набор данных Iris часто используется для демонстрации машинного обучения студентам-первокурсникам. См. документацию Scikit-Learn по адресу http://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html.

[2] Эти веса для входного сигнала отдельного нейрона математически эквивалентны угловым коэффициентам многомерной линейной регрессии или логистической регрессии.

[3] Также называется систематической ошибкой. — *Примеч. пер.*

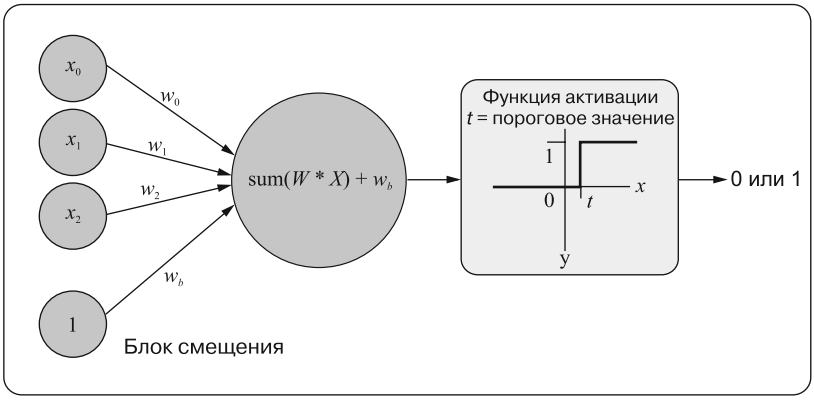


Рис. 5.2. Простой перцептрон

ление входных данных в виде основного входного вектора, скажем, из n элементов и единицы, добавляемой в начало или конец вектора, в результате чего получается $(n + 1)$-мерный вектор. Позиция этой единицы для сети неважна, лишь бы она была одинаковой для всех выборок. В других случаях предполагается наличие постоянного смещения, которое отбрасывается из входных данных на диаграмме, но соответствующий ему вес существует независимо и всегда умножается на 1 и прибавляется к скалярному произведению значений входных сигналов выборки и соответствующих весов. Фактически оба способа аналогичны. Мы просто заранее указали вам два распространенных способа описания этого понятия.

Основная причина, по которой вообще нужен вес смещения, — требование устойчивости нейрона ко всем нулевым входным сигналам. Может так случиться, что сеть должна обучиться выдавать `0` в случае нулевых входных сигналов, хотя такого не должно быть. Без смещения нейрон выдаст `0 * вес = 0` для любых начальных (или усваиваемых) весов. Со смещением этой проблемы не будет. Если нейрону нужно выдавать `0`, он может обучиться уменьшать соответствующий смещению вес настолько, чтобы скалярное произведение не превышало пороговое значение.

На рис. 5.3 представлена довольно точная визуализация аналогий между сигналами в живом нейроне из мозга человека и сигналами искусственного нейрона, применяемого для глубокого обучения. Если хотите разобраться получше, задумайтесь, как вы используете свои живые нейроны для изучения глубокого обучения[1] при чтении этой книги про обработку естественного языка.

[1] Понимание естественного языка (natural language understanding, NLU) — термин, часто применяемый в академических кругах для описания такой обработки естественного языка, которая как бы демонстрирует, что машина понимает смысл текста на естественном языке. Один из примеров задач понимания естественного языка — вложения Word2vec. Задачи формирования ответов на вопросы (question answering) и понимания прочитанного (reading comprehension) также демонстрируют NLU. Нейронные сети в целом очень часто ассоциируются с пониманием естественного языка.

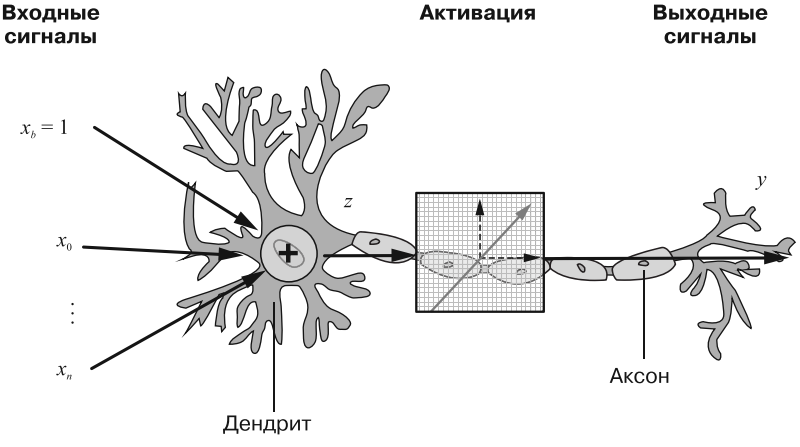


Рис. 5.3. Перцептрон и биологический нейрон

На математическом языке выходной сигнал перцептрона, обозначаемый *f(x)*, выглядит так, как в уравнении 5.1.

Уравнение 5.1. Пороговая функция активации

$$f(\vec{x}) = 1 \text{ если } \sum_{i=0}^{n} x_i w_i > \text{порогового значения, в противном случае} - 0$$

ПРИМЕЧАНИЕ

Сумма попарных произведений элементов входного вектора (X) и вектора весов (W) — как раз скалярное произведение двух векторов. Это одна из важнейших причин, почему линейная алгебра играет такую важную роль в разработке нейронных сетей. Другой побочный эффект подобной структуры перемножения матриц в перцептроне — невероятная эффективность GPU современных компьютеров для реализации нейронных сетей благодаря высочайшей оптимизации в них операций линейной алгебры.

Пока наш перцептрон ничему не *обучился*. Но мы все же добились многого. Мы передали данные в модель и получили выходной сигнал. Вероятно, этот выходной сигнал ошибочен, поскольку мы пока не говорили о том, откуда берутся значения весов. Но здесь и начинается самое интересное.

ПРИМЕЧАНИЕ

Базовый блок любой нейронной сети — нейрон. Простейший перцептрон — частный случай более общего понятия нейрона. Пока что мы будем называть перцептроны нейронами.

Реализация нейрона на Python

На Python можно очень легко вычислить выходной сигнал описанного выше нейрона. Можно также воспользоваться функцией `dot` библиотеки NumPy для перемножения двух векторов:

```
>>> import numpy as np

>>> example_input = [1, .2, .1, .05, .2]
>>> example_weights = [.2, .12, .4, .6, .90]

>>> input_vector = np.array(example_input)
>>> weights = np.array(example_weights)
>>> bias_weight = .2

>>> activation_level = np.dot(input_vector, weights) +
...     (bias_weight * 1)
>>> activation_level
0.674
```

Умножение на единицу (* 1) показано здесь, чтобы подчеркнуть, что bias_weight аналогичен всем остальным весам: умножается на входное значение, однако значение входного признака bias_weight всегда равно 1

После этого можно воспользоваться простой пороговой функцией активации, выбрав пороговое значение `0.5`, и сделать следующее:

```
>>> threshold = 0.5
>>> if activation_level >= threshold:
...     perceptron_output = 1
... else:
...     perceptron_output = 0
>>> perceptron_output
1
```

При указанном `example_input` и данном наборе весов этот перцептрон выдаст `1`. Но в случае нескольких векторов `example_input` и соответствующих ожидаемых результатов для каждого маркированного набора данных вы можете определить, правильные ли результаты он вернул в каждом из *случаев*.

Начнем урок

До сих пор мы прокладывали путь к предсказанию на основе данных, готовя фундамент для основного блюда: машинного обучения. Мы рассматривали значения весов как случайные. Они являются ключом ко всему, и нам нужен способ «подталкивать» их в сторону увеличения или уменьшения в зависимости от результатов предсказания для конкретного примера.

Перцептрон *обучается* в процессе увеличения или уменьшения весов как функции от погрешности результатов работы системы для конкретных входных данных. Но с чего он начинает? Веса необученного нейрона подбираются случайно! Такие значения, обычно близкие к нулю, выбираются из нормального распределения. Из предыдущего примера видно, почему нулевые начальные значения весов (включая вес смещения) могут привести только к нулевому выходному сигналу. Но ввод небольшого разброса значений без предпочтений какому-либо одному из путей

через нейрон дает нам плацдарм, от которого можно отталкиваться для получения правильных или неправильных результатов.

Тогда можно начинать обучение. Системе демонстрируется множество различных выборок, и всякий раз веса чуть-чуть корректируются в зависимости от того, соответствует ли выходной сигнал нейрона желаемому или нет. При достаточной выборке (и при надлежащих условиях) ошибка *должна* стремиться к нулю, а система — *обучиться*.

Хитрость состоит — и это ключ ко всему методу — в корректировке весов в зависимости от их вклада в итоговую ошибку. Чем больше вес (то есть чем сильнее точка данных влияет на результат), тем большая ответственность лежит на нем за правильность/ошибочность выходного сигнала перцептрона для конкретного входного сигнала.

Предположим, что вышеприведенный `example_input` должен привести к `0` на выходе перцептрона:

```
>>> expected_output = 0
>>> new_weights = []
>>> for i, x in enumerate(example_input):
... new_weights.append(weights[i] + (expected_output -\
... perceptron_output) * x)
>>> weights = np.array(new_weights)
```

Например, для первого индекса выше: new_weight = .2 + (0 – 1) * 1 = –0.8

```
>>> example_weights
[0.2, 0.12, 0.4, 0.6, 0.9]
>>> weights
[-0.8 -0.08 0.3 0.55 0.7]
```

Исходные веса

Новые веса

Процесс демонстрации сети снова и снова одного и того же тренировочного набора данных при надлежащих условиях ведет к точным предсказаниям даже на тех данных, которые перцептрон пока что не видел.

Логика — забавная дисциплина

Предыдущий пример содержал лишь случайный набор чисел для демонстрации математики. Применим этот алгоритм к решению задачи. Конечно, это тривиальная модельная задача, но она демонстрирует основы того, как обучить компьютер чему-либо, просто показав ему маркированную выборку.

Попробуем заставить компьютер понять принцип логического `OR`. Если хотя бы одна (или обе) часть выражения истинна, то истинно и все выражение `OR`. Достаточно просто. Для этой модельной задачи можно легко смоделировать все возможные примеры вручную (в реальных задачах такое встречается редко). Каждая выборка состоит из двух сигналов, каждый из которых истинен (`1`) или ложен (`0`) (листинг 5.1).

Листинг 5.1. Подготовка задачи логического OR

```
>>> sample_data = [[0, 0], # False, False
...                [0, 1], # False, True
```

```
...                [1, 0], # True, False
...                [1, 1]] # True, True

>>> expected_results = [0, # (False OR False) дает False
...                     1, # (False OR True ) дает True
...                     1, # (True  OR False) дает True
...                     1] # (True  OR True ) дает True

>>> activation_threshold = 0.5
```

Для начала нам понадобится несколько инструментов: NumPy, чтобы умножить векторы (массивы), и `random` для задания начальных значений весов:

```
>>> from random import random
>>> import numpy as np

>>> weights = np.random.random(2)/1000 # Small random float 0 < w < .001
>>> weights
[5.62332144e-04 7.69468028e-05]
```

Также нам нужно смещение:

```
>>> bias_weight = np.random.random() / 1000
>>> bias_weight
0.0009984699077277136
```

Теперь можно передать его по конвейеру и получить предсказание для каждой из наших четырех выборок (листинг 5.2).

Листинг 5.2. Случайное гадание перцептрона

```
>>> for idx, sample in enumerate(sample_data):
...     input_vector = np.array(sample)
...     activation_level = np.dot(input_vector, weights) +\
...         (bias_weight * 1)
...     if activation_level > activation_threshold:
...         perceptron_output = 1
...     else:
...         perceptron_output = 0
...     print('Predicted {}'.format(perceptron_output))
...     print('Expected: {}'.format(expected_results[idx]))
...     print()
Predicted 0
Expected: 0

Predicted 0
Expected: 1

Predicted 0
Expected: 1

Predicted 0
Expected: 1
```

Случайные значения весов не слишком помогли нашему маленькому нейрону: один раз он угадал и три раза — нет. Отправим его снова в школу. Вместо того чтобы просто выводить `1` или `0`, мы будем обновлять веса на каждой итерации (листинг 5.3).

Листинг 5.3. Обучение перцептрона

```
>>> for iteration_num in range(5):
...     correct_answers = 0
...     for idx, sample in enumerate(sample_data):
...         input_vector = np.array(sample)
...         weights = np.array(weights)
...         activation_level = np.dot(input_vector, weights) +\
...             (bias_weight * 1)
...         if activation_level > activation_threshold:
...             perceptron_output = 1
...         else:
...             perceptron_output = 0
...         if perceptron_output == expected_results[idx]:
...             correct_answers += 1
...         new_weights = []
...         for i, x in enumerate(sample):
...             new_weights.append(weights[i] + (expected_results[idx] -\
...                 perceptron_output) * x)
...         bias_weight = bias_weight + ((expected_results[idx] -\
...             perceptron_output) * 1)
...         weights = np.array(new_weights)
...     print('{} correct answers out of 4, for iteration {}'\
...         .format(correct_answers, iteration_num))
3 correct answers out of 4, for iteration 0
2 correct answers out of 4, for iteration 1
3 correct answers out of 4, for iteration 2
4 correct answers out of 4, for iteration 3
4 correct answers out of 4, for iteration 4
```

Обновляется также вес смещения аналогично весам, соответствующим входным сигналам

Именно здесь происходит все волшебство. Это можно сделать и более эффективно, но реализация в виде цикла позволила нам обновить каждый из весов с учетом соответствующего входного сигнала xi. Маленький или вообще нулевой входной сигнал незначительно повлияет на конкретный вес вне зависимости от величины ошибки. Если входной сигнал был велик, то и влияние его будет значительным

Какой замечательный студент наш маленький перцептрон! Благодаря обновлению весов во внутреннем цикле он обучается на своем опыте обработки конкретного набора данных. После первой итерации он получает на два правильных ответа больше (три из четырех), чем при случайном гадании (один из четырех).

На второй итерации он избыточно корректирует веса (изменяет их слишком сильно), так что ему приходится немного откатить назад внесенные в веса поправки. К четвертой итерации перцептрон идеально усваивает все взаимосвязи. Последующие итерации никак сеть не обновляют, поскольку ошибка на всех выборках равна 0 и никакие поправки весов не производятся.

Этот процесс называется *сходимостью* (convergence). Говорят, что модель сошлась, если ее функция ошибки дошла до минимума или хотя бы до какого-то постоянного значения. Иногда такого везения нет. Нейронная сеть скачет вокруг

да около в поисках оптимальных весов, удовлетворяющих взаимосвязям в пакете данных, и никогда не сходится. В разделе 5.8 мы покажем, как на веса, которые нейронная сеть «считает» оптимальными, влияет *целевая функция*, или *функция потерь*.

Следующий шаг

У простейшего перцептрона есть недостаток. Если данные линейно неразделимы или взаимосвязь невозможно описать с помощью линейной зависимости, то модель не сойдется и никакими полезными способностями к предсказанию обладать не будет. Она не сможет правильно предсказывать целевую величину.

В ранних опытах модели успешно обучались классифицировать изображения на основе одних только примеров изображений и информации про их классы. Первые восторги по поводу этой идеи быстро утихли после выхода работы Минского и Пейперта[1], которые показали, что виды доступной перцептрону классификации сильно ограниченны. Ученые продемонстрировали, что если выборки данных неразделимы линейно на дискретные группы, то перцептрон не сможет научиться классифицировать входные данные (рис. 5.4).

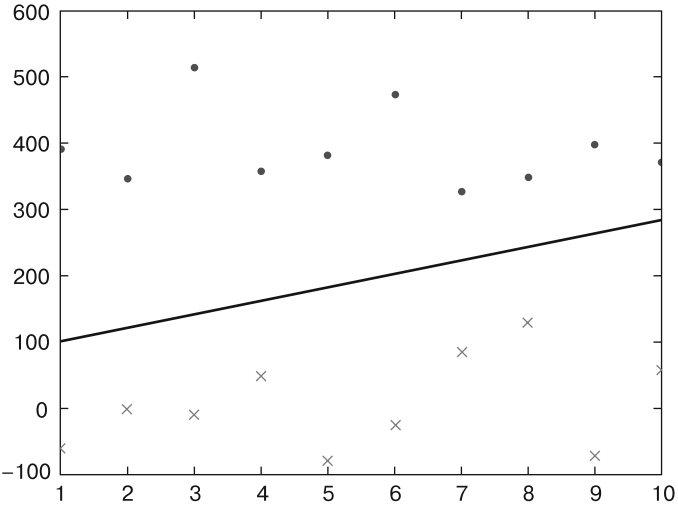


Рис. 5.4. Линейно разделимые данные

Линейно разделимые точки данных (как показано на рис. 5.4) не проблема для перцептрона. Перекрещенные же данные приведут к тому, что однонейронный перцептрон будет бесконечно работать, но не обучится при этом делать лучшие, по сравнению со случайными догадками (подбрасыванием монеты), предсказания. Разделить одной прямой два класса (точки и крестики) на рис. 5.5 невозможно.

[1] *Минский М., Пейперт С.* Перцептроны. — М.: Мир, 1971 (https://ru.wikipedia.org/wiki/Перцептроны_(книга)).

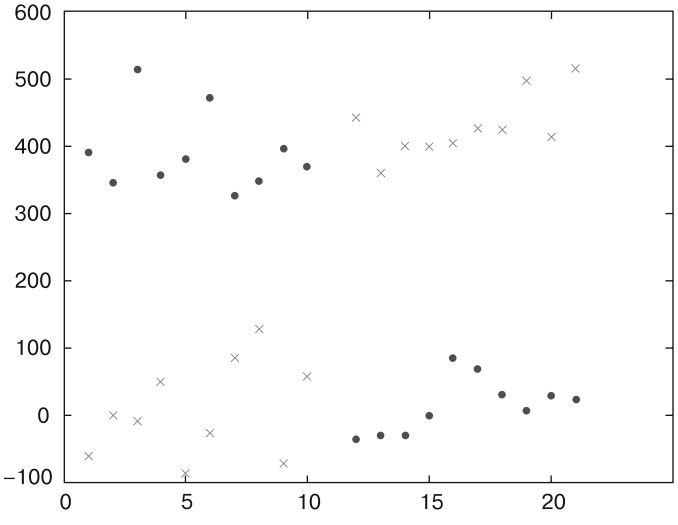


Рис. 5.5. Неразделимые линейно данные

Перцептрон подбирает линейное уравнение, которое описывает взаимосвязь между признаками и целевой переменной в наборе данных. Он просто выполняет линейную регрессию и не может описать нелинейное уравнение или нелинейную взаимосвязь.

Локальный и глобальный минимум

Когда перцептрон сходится, можно сказать, что он нашел линейное уравнение, описывающее взаимосвязь между данными и целевой переменной. Впрочем, о качестве этого описательного линейного уравнения или о том, насколько «минимальна» функция стоимости, он ничего не говорит. Если существует несколько решений, возможных минимумов стоимости, перцептрон сойдется к одному конкретному минимуму, который определяется начальными значениями весов. Он называется *локальным минимумом* (local minimum), поскольку является оптимальным (в смысле наименьшей стоимости) в окрестностях начальных значений весов. Но это вовсе не обязательно *глобальный минимум* (global minimum), то есть наилучшее решение для всех возможных весов. В большинстве случаев понять, найден ли глобальный минимум, невозможно.

Многие взаимосвязи между значениями данных нелинейны. Нет хорошей линейной регрессии или линейного уравнения, которые бы их описывали. Многие наборы данных неразделимы линейно на классы с помощью прямых или плоскостей, а поскольку большая часть данных в мире неразделима линейно прямыми или плоскостями, то опубликованное Минским и Пейпертом «доказательство» отправляет перцептрон на полку.

Но от идеи перцептрона не отказались так просто. О ней снова заговорили, когда в совместной работе Румельхарта и Макклелланда (при участии Джеффри Хинтона)[1] было показано, что ее можно использовать для решения задачи исключающего OR (XOR) с помощью согласованной работы нескольких перцептронов[2]. Отдельным перцептроном без многослойного обратного распространения ошибки мы решили более простую задачу: OR. Главное достижение Румельхарта и Макклелланда заключалось в открытии способа подходящего распределения ошибки по перцептронам. Для этого они воспользовались давно известной идеей — методом обратного распространения ошибки. Первая нейронная сеть и возникла на основе обратного распространения ошибки по слоям нейронов.

Существенный недостаток простейшего перцептрона в том, что если данные неразделимы линейно, то модель не сойдется к пригодному для выполнения предсказаний решению.

ПРИМЕЧАНИЕ

В коде из листинга 5.3 мы решили задачу OR с помощью одного перцептрона. Полученная в результате обучения нашего перцептрона таблица из 1 и 0 в листинге 5.1 представляла собой результаты бинарного логического OR. Для задачи XOR эта таблица слегка меняется, чтобы обучить перцептрон имитировать логический вентиль типа XOR. Если поменять правильный ответ для последнего примера с 1 (true) на 0 (false), чтобы отразить логику XOR, задача сразу значительно усложняется. Примеры в двух классах (0 или 1) неразделимы линейно без добавления дополнительного нейрона в нашу нейронную сеть. Классы располагаются по диагонали друг от друга в нашем двумерном векторном пространстве признаков (аналогично рис. 5.5), так что никакой прямой нельзя отделить единицы (логические true) от нулей (логических false).

Хотя они позволяли решать сложные (нелинейные) задачи, но на то время нейронные сети были слишком затратными вычислительно. Решение задачи XOR с помощью двух перцептронов и кучи математических действий, связанных с обратным распространением ошибки, рассматривалось как напрасная трата драгоценных вычислительных ресурсов. Для широкого применения этот метод оказался непригодным и опять отправился на пыльные университетские полки и использовался только для опытов с суперкомпьютерами. Так началась вторая «зима» искусственного интеллекта (https://ru.wikipedia.org/wiki/Зима_искусственного_интеллекта#Неудачи_конца_1980-х_и_начале_1990-х_годов), длившаяся с 1990-го до примерно 2010 года[3]. В итоге вычислительные ресурсы, алгоритмы обратного распространения ошибки

---

1 *Rumelhart D. E., Hinton G. E., Williams R. J.* Learning representations by backpropagating errors // Nature, 323, 1986. — P. 533–536.

2 См. статью The XOR affair по адресу https://en.wikipedia.org/wiki/Perceptrons_(book)#The_XOR_affair.

3 См. веб-страницу Philosophical Transactions of the Royal Society B: Biological Sciences по адресу http://rstb.royalsocietypublishing.org/content/365/1537/177.short.

и обилие исходных данных, например маркированных изображений кошек и собак[1], взяли свое. Требующие больших вычислительных ресурсов алгоритмы и ограниченные размеры наборов данных перестали быть непреодолимой преградой. Так началась третья эпоха нейронных сетей.

Но вернемся к сути их открытия.

Выход из второй «зимы» искусственного интеллекта

Большинство замечательных идей находят дорогу наружу. Оказалось, что лежащую в основе перцептрона простую идею можно расширить так, чтобы преодолеть «похоронившее» ее сначала ограничение. Идея состояла в объединении нескольких перцептронов с поступлением входного сигнала в один (или несколько) из них. Затем выходной сигнал этого перцептрона можно отправить на вход других и сравнить в итоге полученный результат с ожидаемым. Такая система (нейронная сеть) способна усваивать более сложные закономерности в данных и побороть проблему неразделимых линейно классов, например, как в случае задачи XOR. Основной вопрос: как обновить веса в предыдущих слоях?

Задумаемся на минуту и формализуем одну из важных частей этого процесса. До сих пор мы обсуждали ошибки и погрешность предсказания перцептрона. Измерение этой ошибки — задача *функции стоимости* (cost function) или *функции потерь* (loss function). Первая, как вы уже видели, служит количественной мерой расхождения между правильными ответами, ожидаемыми от сети, и фактическими значениями выходных сигналов (y) для соответствующих «вопросов» (x), поступающих на вход сети. Вторая отражает частоту выдачи сетью неправильных ответов и степень их отличия от правильных. Один из примеров функции потерь (просто разница истинного значения и предсказания модели) приведен в уравнении 5.2.

Уравнение 5.2. Расхождение истинного значения и предсказания

$$\mathrm{err}(x) = |y - f(x)|$$

Цель обучения перцептрона или нейронной сети в целом — в минимизации этой функции потерь для всех имеющихся входных выборок (уравнение 5.3).

Уравнение 5.3. Минимизируемая функция потерь

$$J(x) = \min \sum_{i=1}^{n} err(x_i)$$

Скоро вы познакомитесь и с другими функциями потерь, например среднеквадратичной погрешностью, но выбирать лучшую не придется. Обычно в большинстве фреймворков нейронных сетей этот выбор сделан за вас. Самое главное: уяснить для себя идею, что минимизация функции потерь на всем наборе данных — наша конечная цель. Тогда все остальные описанные здесь идеи будут понятны.

---

[1] См. статью: *Krizhevsky A.* Learning Multiple Layers of Features from Tiny Images по адресу http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.222.9220&rep=rep1&type=pdf.

Метод обратного распространения ошибки

Хинтон и его коллеги решили, что можно одновременно использовать несколько перцептронов для одной цели. Это позволит, как они показали, решать неразделимые линейно задачи. Теперь можно будет аппроксимировать и нелинейные функции.

Но как же обновлять веса этих различных перцептронов? Что вообще означает фраза «вклад в ошибку»? Допустим, два перцептрона расположены рядом и получают одинаковый входной сигнал. Не важно, что произошло с выходным сигналом (конкатенация, сложение, умножение), все равно при распространении ошибки на исходные веса он будет функцией входного сигнала (одинакового для обоих перцептронов). Так что веса будут обновляться аналогично на каждом шаге, и мы ничего не добьемся. Наши нейроны окажутся избыточными. Их итоговые веса будут одинаковы, а сеть ничему особенно не обучится.

Все еще больше запутывается, если представить себе перцептрон, выходной сигнал которого поступает во второй перцептрон в качестве входного сигнала. Именно это мы и собираемся сделать.

Метод обратного распространения ошибки позволяет решить данную проблему, но для этого придется немного модифицировать наш перцептрон. Напомним, что веса обновлялись пропорционально их вкладу в общую погрешность. Но если вес влияет на выходной сигнал, который становится входным для другого перцептрона, то совершенно непонятно, какова была ошибка в начале этого второго перцептрона.

Поэтому нужен способ вычислить вклад конкретного веса (w_{1i} на рис. 5.6) в ошибку при условии, что он произвел этот вклад через другие веса (w_{1j}) и (w_{2j}) на следующем слое. И этот способ называется *обратным распространением ошибки* (backpropagation).

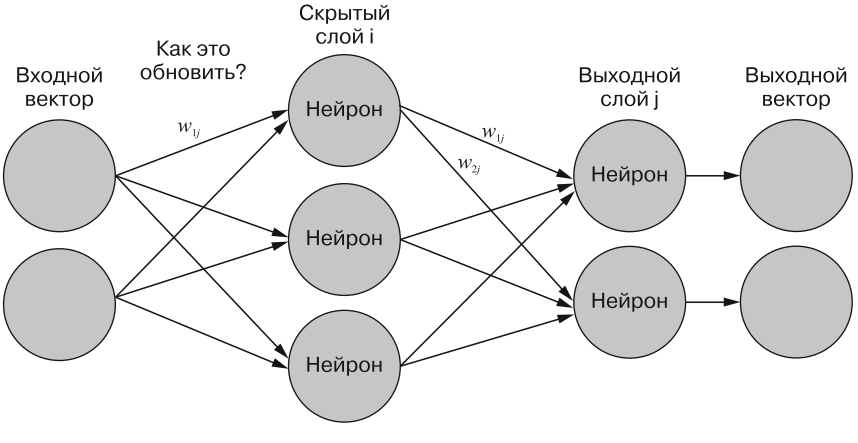


Рис. 5.6. Нейронная сеть со скрытыми весами

Настало время прекратить использовать термин «перцептрон», поскольку мы собираемся изменить способ обновления весов в каждом нейроне. С этого момента мы будем употреблять более общий термин *«нейрон»*, включающий как перцептрон, так

и родственные ему модели с большими возможностями. В литературе нейрон также называют клеткой (cell) или узлом (node) (при апелляции к графовому представлению нейросети), причем в большинстве случаев эти термины взаимозаменяемы.

Нейронная сеть любого типа представляет собой не более чем набор связанных между собой нейронов. Их часто объединяют в слои, но это не обязательно. После построения архитектуры, в которой выходной сигнал одного нейрона становится входным сигналом другого, говорят о *скрытых* нейронах и слоях, в отличие от *входного* или *выходного* слоя или нейрона.

Такая архитектура называется *полносвязной сетью* (fully connected network). Хотя на рис. 5.7 показаны не все связи. В такой сети каждый входной элемент связан с *каждым* нейроном в следующем слое. И каждой связи соответствует свой вес. Так что в сети из пяти нейронов, принимающей на входе четырехмерный вектор, в слое всего 20 весов (по четыре веса для связей каждого из пяти нейронов).

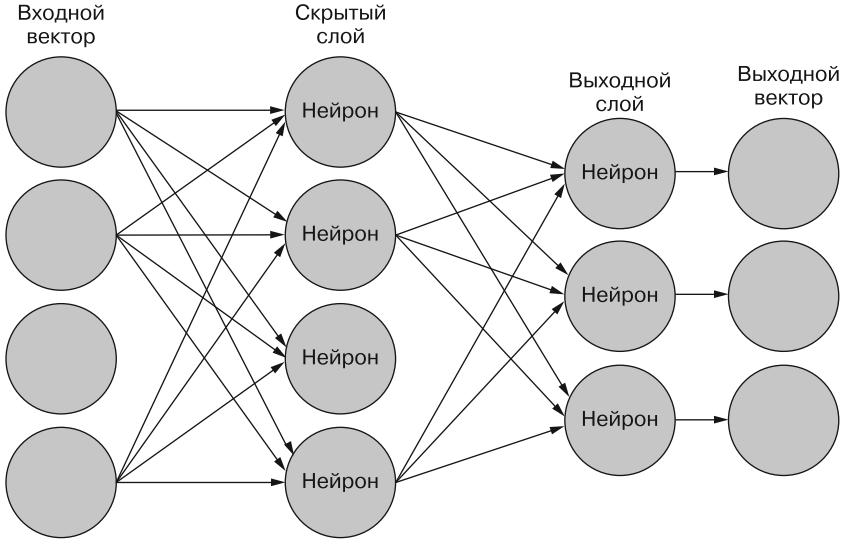


Рис. 5.7. Полносвязная нейронная сеть

Каждому входному сигналу, как и у перцептрона, соответствует свой вес, а у нейронов из второго слоя нейронной сети веса соотносятся не с исходным входным сигналом, а с выходным первого слоя. Теперь должна быть понятна сложность вычисления вклада веса первого слоя в общую ошибку. Воздействие веса первого слоя передается не просто через еще один вес, а через один вес для каждого из нейронов следующего слоя. Вывод и математические подробности самого алгоритма, хотя и чрезвычайно интересны, выходят за рамки данной книги, но мы приведем краткий их обзор, чтобы нейронные сети не были для вас совсем «черным ящиком».

Обратное распространение (сокращение от «метод обратного распространения ошибки») описывает способ выяснения объема обновления для конкретного веса по заданному входному сигналу, выходному сигналу и ожидаемому значению. *Рас-*

пространение (propagation), оно же прямое распространение, означает продвижение «вперед» по сети и вычисление выходного сигнала сети для данного входного сигнала. Для перехода к обратному распространению ошибки необходимо сначала изменить функцию активации перцептрона на более сложную.

До сих пор мы использовали в качестве *функции активации* нашего искусственного нейрона ступенчатую функцию. Как вы скоро увидите, для обратного распространения ошибки требуется нелинейная и непрерывно дифференцируемая функция активации[1]. При этом каждый из нейронов будет выдавать значение в некотором *диапазоне*, например от 0 до 1, как в часто применяемой сигма-функции, показанной в уравнении 5.4.

Уравнение 5.4. Сигма-функция

$$S(x) = \frac{1}{1+e^{-x}}$$

Почему функция активации должна быть нелинейной

Чтобы нейроны могли моделировать нелинейные связи между векторами признаков и целевой переменной. Если все, на что способен нейрон, — умножать входные сигналы на веса и складывать их, то выходной сигнал всегда будет линейной функцией входных сигналов. Вы не сможете смоделировать даже простейшие нелинейные связи.

Но применявшаяся выше для наших нейронов пороговая функция представляла собой нелинейную ступенчатую функцию. Так что использовавшиеся нейроны теоретически можно было обучить моделировать сообща практически любую нелинейную связь... Если, конечно, у вас хватит для этого нейронов.

В этом и преимущество нелинейной функции активации: с ее помощью нейронная сеть может моделировать нелинейную связь. Непрерывно дифференцируемая функция активации, например сигма-функция, позволяет гладко распространять ошибку через несколько слоев нейронов, что ускоряет процесс обучения. Сигма-нейроны — способные ученики.

Существует множество других функций активации, например *гиперболический тангенс* (hyperbolic tangent) и *выпрямленные линейные блоки* (rectified linear unit, ReLU), со своими достоинствами и недостатками. Каждая по-своему хороша для различных архитектур нейронных сетей, как вы узнаете в следующих главах.

Так зачем же нужна дифференцируемость? Если можно вычислить производную функции, то можно вычислить и ее частные производные по различным переменным в ней. Ключевые слова тут: «по различным переменным». Это путь к обновлению весов с учетом полученной ими доли входного сигнала!

---

[1] Непрерывно дифференцируемая функция — еще более гладкая, чем просто дифференцируемая. См. статью Differentiable function по адресу https://en.wikipedia.org/wiki/Differentiable_function#Differentiability_and_continuity.

Дифференцируем все возможное

Начнем с погрешности сети и применим к ней функцию стоимости, например среднеквадратичную ошибку, как показано в уравнении 5.5.

Уравнение 5.5. Среднеквадратичная ошибка

$$MSE = (y - f(x))^2$$

Далее воспользуемся *цепным правилом* математического анализа для вычисления производной композиции функций, как в уравнении 5.6. Сеть представляет собой не что иное, как композицию функций (скалярных произведений, за которыми следует на каждом шаге наша новая нелинейная функция активации).

Уравнение 5.6. Цепное правило

$$(f(g(x))' = F'(x) = f'(g(x))g'(x)$$

Эту формулу можно использовать для вычисления производных функций активации всех нейронов с учетом их входных сигналов, а значит, можно вычислить вклад конкретного веса в итоговую ошибку и откорректировать его соответствующим образом.

Если слой — выходной, то обновление весов не представляет сложностей благодаря нашей легко дифференцируемой функции активации. Производная ошибки по *j*-му повлиявшему на ее значение выходному сигналу приведена в уравнении 5.7.

Уравнение 5.7. Производная ошибки

$$\Delta w_{ij} = -\alpha \frac{\partial Error}{w_{ij}} = -\alpha y_i (y_i - f(x)_j) y_j (1 - y_j)$$

Обновление весов скрытого слоя — более сложная задача, как вы можете видеть из уравнения 5.8.

Уравнение 5.8. Производная предыдущего слоя

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = -\alpha y_i (\sum_{l \varepsilon L} \delta_j w_{il}) y_j (1 - y_j)$$

Функция *f(x)* из уравнения 5.7 представляет собой выходной сигнал: *j*-й элемент выходного вектора, *y* — выходной сигнал узла на *i*-м или *j*-м слое, где выходной сигнал *i* слоя является входным сигналом *j*. Поэтому α (скорость обучения) умножается на выходной сигнал предыдущего слоя, умноженный на производную функции активации с последующего слоя *по* весу, применяемому к выходному сигналу слоя *i*, поступающему в *j*. Сумма в уравнении 5.8 выражает эту идею для всех входных сигналов во все слои.

Важно четко понимать, когда изменения применяются к самим весам. Вычисления обновлений весов в каждом слое зависят от состояния сети во время прямого прохода. После вычисления ошибки мы высчитываем предлагаемые корректировки

для всех весов в сети. Но *не* применяем ни одной из них — до тех пор пока не вернемся в начало сети. В противном случае при обновлении весов по направлению к концу сети вычисленные для более низких уровней производные не будут подходящим градиентом для конкретного входного сигнала. Можно агрегировать все подъемы и спуски для каждого из весов на основе тренировочных выборок без обновления каких-либо весов, а обновить их только в конце обучения. Подробнее этот вариант мы обсудим в подразделе 5.1.6.

Для обучения сети мы передаем в нее все входные сигналы, получаем соответствующую каждому входному сигналу ошибку. Распространяем эти ошибки обратно к весам, после чего обновляем каждый из весов соответственно общему изменению ошибки. Однократный проход тренировочных данных через сеть с последующим обратным распространением ошибок называется *эпохой* (epoch) цикла обучения нейронной сети. После этого можно передавать сети набор данных снова и снова для дальнейшего уточнения весов. Будьте осторожны, чтобы не переобучить веса на тренировочном наборе данных, вследствие чего модель не сможет делать осмысленные предсказания для новых, не входивших в тренировочный набор данных точек.

В уравнениях 5.7 и 5.8 параметр α представляет собой *скорость обучения* (learning rate). Он определяет, насколько сильно корректируется наблюдаемая в весе ошибка за конкретный цикл обучения (эпоху) или пакет данных. Обычно он остается неизменным на протяжении одного цикла обучения, но в некоторых продвинутых алгоритмах обучения он корректируется адаптивно для ускорения обучения и обеспечения сходимости. При слишком большом α можно легко скорректировать ошибку чересчур сильно. И следующая ошибка приведет к еще большей корректировке веса в другую сторону, но еще дальше от цели. Если же задать слишком маленькое значение α, то модель будет сходиться за слишком большое для практического применения время или, еще хуже, застрянет в локальном минимуме на *поверхности ошибок* (error surface).

5.1.4. Айда кататься на лыжах — поверхность ошибок

Цель обучения нейронных сетей, как мы упоминали выше, — поиск оптимальных параметров (весов), минимизирующих функцию стоимости. Помните, что речь идет не об ошибке для какой-либо одной точки данных. Необходимо минимизировать стоимость для всех различных ошибок вместе.

Для построения ментальной модели происходящего при корректировке весов сети полезно создать визуализацию этой стороны задачи.

Как упоминалось выше, одной из распространенных функций стоимости является среднеквадратическая ошибка (показана в уравнении 5.5). Если представить себе график ошибки как функции возможных весов, при заданных наборах входных и выходных сигналов существует точка, в которой эта функция близка к нулю. Это и есть наш *минимум* — место, в котором ошибка модели минимальна.

Минимум — это набор весов, при котором выходной сигнал оптимален для заданного тренировочного примера. Эту ситуацию часто наглядно представляют в виде трехмерной чаши, где две оси координат — 2D-вектор весов, а третья — ошибка (рис. 5.8).

Это описание, конечно, изрядно упрощено, но идея остается той же и для более многомерных пространств (то есть для случаев большего, чем два, количества весов).

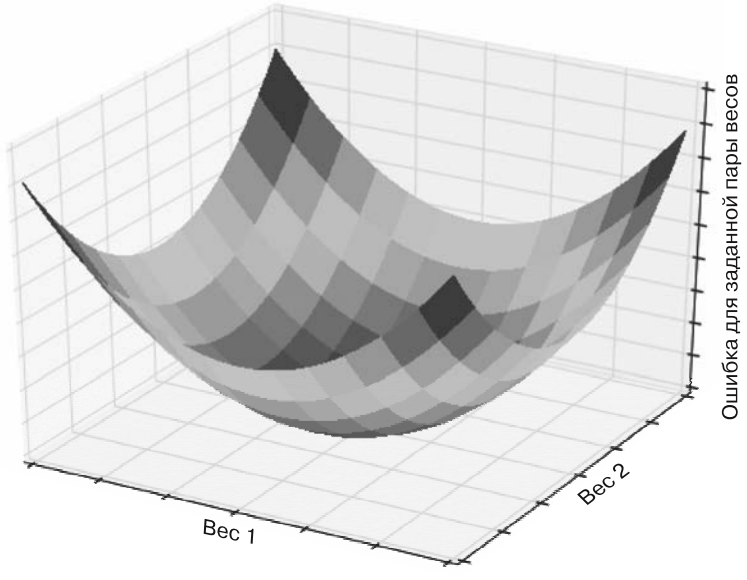


Рис. 5.8. Выпуклая кривая ошибок

Аналогично можно построить график поверхности ошибки в виде функции от всех возможных весов для всех входных сигналов тренировочного набора данных. Но необходимо немного модифицировать функцию ошибки. Нужна такая функция, которая бы отражала совокупную ошибку всех входных сигналов для заданного набора весов. Здесь мы воспользуемся *среднеквадратичной погрешностью* в качестве оси *Z* (см. уравнение 5.5).

Минимум поверхности ошибок будет соответствовать набору весов, который представляет модель, лучше всего подходящую для всего тренировочного набора данных.

5.1.5. С подъемника — на склон

Что означает эта визуализация? На каждой эпохе алгоритм выполняет *градиентный спуск*, пытаясь минимизировать ошибку. Каждый раз мы корректируем веса в сторону (хочется надеяться) последующего снижения ошибки. Выпуклая поверхность ошибок отлично подойдет. Становимся на склоне на лыжах, оглядываемся по сторонам, выбираем путь вниз — и вперед!

Но вам не всегда повезет получить гладкую чашу. По поверхности ошибок может быть разбросано множество ям и выемок. Подобная ситуация называется *невыпуклой поверхностью ошибок* (nonconvex error curve). Как и при лыжном спуске, если ямы достаточно велики, можно в них провалиться и не добраться до низа склона.

Диаграммы отражают веса для двумерных входных сигналов. Но при 10-, 50- или 1000-мерных идея остается такой же. В таких пространствах более высокой размерности пытаться визуализировать бессмысленно, так что приходится доверять математике. Когда вы начнете применять нейронные сети, визуализация перестанет быть столь важной. Ту же информацию можно получить из наблюдения (или построения графика) ошибки или соответствующей метрики по времени обучения, чтобы выяснить, стремится ли она к 0. Так можно понять, в правильном ли направлении продвигается сеть. Но подобные 3D-визуализации — очень удобный инструмент для создания ментальной модели процесса.

Как насчет невыпуклых поверхностей ошибок? Не создают ли проблему эти ямы и выемки? Еще как. В зависимости от случайных начальных значений весов процедура может завершиться на совершенно различных весах, и обучение прекратится, поскольку пути вниз из данного *локального минимума* не существует (рис. 5.9).

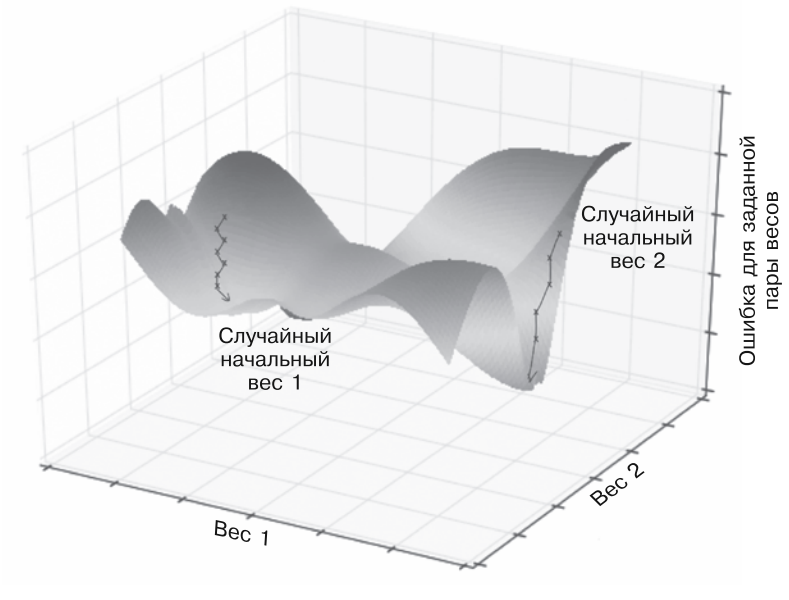


Рис. 5.9. Невыпуклая кривая ошибок

Имейте в виду, что локальные минимумы будут преследовать вас даже в пространствах с еще большей размерностью.

5.1.6. Проведем небольшую реорганизацию

До сих пор мы агрегировали ошибки для всех тренировочных примеров и спускались на лыжах по склону в меру своих сил. Описанный подход к обучению представляет собой *пакетное* (batch) обучение. Пакет — большое подмножество обучающих данных, но поверхность ошибок одинакова для всех элементов данных пакета. При такой поверхности, если скатиться со склона со случайной начальной точки, можно оказаться

в каком-либо локальном минимуме (выемке или яме) и не знать, что существуют лучшие значения весов. Обойти эти ловушки могут помочь два других варианта обучения.

Первый вариант — *стохастический* градиентный спуск, когда значения весов обновляются после каждого тренировочного примера, а не после просмотра всех. И при каждом проходе такие примеры перемешиваются. Благодаря этому поверхность ошибок рисуется заново для каждого примера, поскольку ожидаемый ответ для каждого входного сигнала может отличаться. Но мы все равно корректируем веса на основе градиентного спуска *для данного примера*. Вместо агрегирования ошибок и корректировки весов однократно в конце эпохи мы обновляем веса после каждого отдельного примера. Главное, что всегда продвигаемся *в направлении* предполагаемого минимума (хотя и не проходим весь путь к нему).

Двигаясь к различным минимумам на этой переменной поверхности, при правильных данных и правильных гиперпараметрах гораздо легче доковылять до глобального минимума. Если модель не настроена правильно или тренировочные данные противоречивы, то модель не сойдется и вы будете просто крутиться вокруг да около, а модель ничему не обучится. Но на практике стохастический градиентный спуск достаточно успешно обходит локальные минимумы в большинстве случаев. Недостаток подхода — его медлительность. Вычисление прямого прохода, обратного распространения ошибки и последующее обновление весов после каждого примера делает процесс еще медлительнее.

Более распространенный подход, второй вариант, — *мини-пакеты* (mini-batch). При *мини-пакетном* обучении передается маленькое подмножество тренировочного набора данных и агрегируются соответствующие ошибки, как и в полном *пакете*. Эти ошибки затем распространяются обратно, как и для *пакета*, а веса обновляются для каждого такого подмножества тренировочного набора данных. Процесс повторяется со следующим пакетом и т. д. вплоть до окончания тренировочного набора данных. И такой процесс составляет одну эпоху. Это золотая середина, объединяющая преимущества *пакетного* (скоростного) и *стохастического* (устойчивого) методов обучения.

Хотя подробности работы *метода обратного распространения ошибки* весьма захватывающи (https://ru.wikipedia.org/wiki/Метод_обратного_распространения_ошибки), они далеко не тривиальны и, как уже отмечалось ранее, выходят за рамки данной книги. Но вам пригодится мысленный образ поверхности ошибок. Нейронная сеть — всего лишь способ спуска по склону чаши *так быстро, как только можно*, вплоть до самого дна. Находясь в заданной точке, оглядитесь вокруг, найдите самый крутой путь вниз (не слишком приятное зрелище, если вы боитесь высоты) и двигайтесь по нему. На следующем шаге (пакетном, мини-пакетном или стохастическом) снова оглядитесь вокруг, найдите самый крутой путь вниз и идите по нему. Достаточно скоро вы окажетесь у камина на лыжной базе внизу, в долине.

5.1.7. Keras: нейронные сети на Python

Написание нейронной сети на чистом языке Python — интересный опыт, который может помочь соединить в голове все детали. Но Python не слишком быстр, а само по себе число необходимых вычислений затрудняет создание нейронных сетей даже

среднего размера. Существует множество библиотек Python, с помощью которых можно решить проблему быстродействия: PyTorch, Theano, TensorFlow, Lasagne и многие другие. В примерах из этой книги будет использоваться библиотека Keras, находящаяся по адресу https://keras.io/.

Keras — высокоуровневый адаптер с удобным API для Python. Предоставляемый им API можно применять практически без изменений с тремя различными прикладными частями: Theano, TensorFlow от Google и CNTK от компании Microsoft. У каждой из них есть свои низкоуровневые реализации основных элементов нейронных сетей и прекрасно отлаженные библиотеки операций линейной алгебры для выполнения скалярных произведений и максимизации быстродействия матричных произведений нейронных сетей.

Взглянем на простую задачу XOR и посмотрим, сможем ли мы обучить сеть с помощью Keras (листинг 5.4).

Листинг 5.4. Сеть Keras для XOR

```
>>> import numpy as np
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Activation
>>> from keras.optimizers import SGD
>>> # Наши примеры для исключающего OR
>>> x_train = np.array([[0, 0],
...                     [0, 1],
...                     [1, 0],
...                     [1, 1]])
>>> y_train = np.array([[0],
...                     [1],
...                     [1],
...                     [0]])
>>> model = Sequential()
>>> num_neurons = 10
>>> model.add(Dense(num_neurons, input_dim=2))
>>> model.add(Activation('tanh'))
>>> model.add(Dense(1))
>>> model.add(Activation('sigmoid'))
>>> model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                30
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 11
_________________________________________________________________
activation_2 (Activation)    (None, 1)                 0
=================================================================
Total params: 41
Trainable params: 41
Non-trainable params: 0
```

Основной класс модели Keras

Dense — полносвязный слой нейронов

Стохастический градиентный спуск, хотя есть и другие варианты

x_train — список выборок двумерных векторов признаков, используемых для обучения

y_train — желаемые исходы (целевые значения) для всех выборок векторов признаков

Полносвязный скрытый слой будет содержать десять нейронов

Выходной слой состоит из нейрона, который выдает одно значение бинарной классификации (0 или 1)

Параметр input_dim необходим только для первого слоя. В последующих слоях форма будет вычисляться автоматически на основе выходной размерности предыдущего слоя. Для содержащих 2 входных значения примеров нашего логического вентиля XOR мы используем двумерные векторы признаков

Вызов `model.summary()` возвращает сводку параметров сети и количества весов (`Param \#`) на каждом этапе. Немного арифметики: десять нейронов, каждый с двумя весами (по одному для каждого значения во входном векторе) плюс один вес для смещения. В результате необходимо подобрать значения 30 весов. Выходной слой содержит по весу для каждого из десяти нейронов и плюс один вес для смещения, итого 11 весов.

Следующий фрагмент кода труднее для понимания:

```
>>> sgd = SGD(lr=0.1)
>>> model.compile(loss='binary_crossentropy', optimizer=sgd,
...     metrics=['accuracy'])
```

SGD — импортированный нами оптимизатор на основе стохастического градиентного спуска. С помощью именно этого метода модель будет минимизировать ошибку (*потери* — loss). `lr` — скорость обучения, коэффициент, применяемый к частной производной ошибки по каждому из весов. Чем выше ее значение, тем быстрее будет происходить обучение, но и выше риск «проскочить» мимо глобального минимума. При маленьком значении точность выше, но и дольше время обучения, да и модель более восприимчива к локальным минимумам. Сама функция потерь также описана в виде параметра, в данном случае `binary_crossentropy`. Параметр `metrics` представляет собой список опций выходного потока во время обучения. Метод `compile` создает, но не обучает модель. Задаются начальные значения весов, и это случайно выбранное состояние можно использовать для предсказаний на основе набора данных с получением, правда, лишь случайных догадок:

```
>>> model.predict(x_train)
[[ 0.5       ]
 [ 0.43494844]
 [ 0.50295198]
 [ 0.42517585]]
```

Метод `predict` возвращает необработанный выходной сигнал последнего слоя, генерируемый в данном примере сигма-функцией.

Похвастаться особо нечем. Но не забывайте, что у нашей модели нет никаких знаний про ответы; она просто применяет случайные веса к входным сигналам. Попробуем ее обучить (листинг 5.5).

Листинг 5.5. Обучение модели на тренировочном наборе XOR

```
model.fit(x_train, y_train, epochs=100)      <-- Здесь мы обучаем модель
Epoch 1/100
4/4 [==============================] - 0s - loss: 0.6917 - acc: 0.7500
Epoch 2/100
4/4 [==============================] - 0s - loss: 0.6911 - acc: 0.5000
Epoch 3/100
4/4 [==============================] - 0s - loss: 0.6906 - acc: 0.5000
...
Epoch 100/100
4/4 [==============================] - 0s - loss: 0.6661 - acc: 1.0000
```

СОВЕТ

Сеть может не сойтись с первой попытки. Первый вызов compile может вернуть такие параметры из случайного распределения, при которых очень сложно или даже невозможно найти глобальный минимум. В подобном случае лучше вызвать model.fit снова с теми же параметрами (или увеличить количество эпох) и посмотреть, не сойдется ли сеть. Или задайте новые случайные начальные значения для сети и попробуйте запустить fit для них. В этом случае проверьте, не задали ли вы начальное значение генератора случайных чисел (seed), иначе вы просто будете повторять один и тот же опыт снова и снова.

По мере многократного прохода по нашему крошечному набору данных сеть разобралась, что к чему. Она обучилась исключающему ИЛИ (XOR) просто по показанным ей примерам! Вот подлинная магия нейронных сетей, которая будет сопровождать вас в следующих главах:

```
>>> model.predict_classes(x_train)
4/4 [==============================] - 0s
[[0]
 [1]
 [1]
 [0]]
>>> model.predict(x_train)
4/4 [==============================] - 0s
[[ 0.0035659 ]
 [ 0.99123639]
 [ 0.99285167]
 [ 0.00907462]]
```

Повторный вызов метода `predict` (и метода `predict_classes`) для обученной модели демонстрирует лучшие результаты. При нем точность на нашем крошечном наборе данных составляет 100 %. Конечно, точность — не обязательно лучшая мера для предсказательной модели, но здесь она подходит. Так что мы сохраним нашу революционную модель XOR для будущих поколений в листинге 5.6.

Листинг 5.6. Сохраняем обученную модель

```
>>> import h5py
>>> model_structure = model.to_json()

>>> with open("basic_model.json", "w") as json_file:
...     json_file.write(model_structure)

>>> model.save_weights("basic_weights.h5")
```

Экспорт структуры сети в строку JSON для дальнейшего использования с помощью вспомогательного метода Keras

Полученные в результате обучения веса необходимо сохранить отдельно. Первая часть нашего фрагмента кода сохраняет лишь структуру сети. Для их загрузки в дальнейшем необходимо снова загрузить ту же структуру модели

Существуют аналогичные методы для загрузки структуры модели, так что не обязательно производить повторное обучение всякий раз, когда нужно выполнять

предсказание. Это огромный шаг вперед. Хотя обучение данной модели занимает всего несколько секунд, в следующих главах на работу примеров могут уйти минуты, часы, а в некоторых случаях и дни, в зависимости от аппаратного обеспечения и сложности модели, готовьтесь!

5.1.8. Вперед и вглубь

По мере распространения нейронных сетей и порождения ими целой сферы глубокого обучения было проведено (и проводится) множество исследований, посвященных их устройству, в том числе:

- различным функциям активации (например, сигма-функции, выпрямленным линейным блокам и гиперболическому тангенсу);
- выбору оптимальной скорости обучения для усиления/ослабления влияния ошибки;
- динамической подстройке скорости обучения с помощью модели *моментума* (momentum) для ускорения поиска глобального минимума;
- применению *дропаута* (dropout), при котором выбранный случайным образом набор весов игнорируется в данном проходе обучения, чтобы уменьшить вероятность излишней подгонки модели к тренировочному набору данных (переобучения);
- регуляции весов для искусственного сдерживания слишком сильного роста/уменьшения одного из весов по сравнению с остальными (еще один прием, с помощью которого можно избежать переобучения).

Этот список можно продолжать еще очень долго.

5.1.9. Нормализация: «стильный» входной сигнал

Для нейронных сетей требуется векторный входной сигнал, и они извлекают максимум из полученных данных. Есть ключевой аспект, о котором нужно не забывать: *нормализация* (normalization) входного сигнала. Это справедливо для множества моделей машинного обучения. Представьте задачу классификации домов, например, по вероятности их продажи на заданном рынке. У нас есть только две точки данных: число спален и прошлая цена продажи. Их можно представить в виде вектора. Скажем, для дома с двумя спальными комнатами, который в прошлый раз был продан за 275 тысяч долларов:

```
input_vec = [2, 275000]
```

Когда сеть будет пытаться усвоить какую-либо информацию о данных, веса в первом слое, соответствующие спальням, быстро вырастут до очень больших значений, чтобы сравняться по масштабу со значениями, аналогичными цене. Поэтому данные часто нормализуют, чтобы все элементы сохраняли полезную информацию во всех выборках. Нормализация также гарантирует работу нейронов

в схожих диапазонах входных значений для различных элементов одного вектора выборки. Существует несколько ее способов, например нормализация по среднему, масштабирование признаков и коэффициент вариации. Но их общая цель — привести данные в один диапазон, например [–1, 1] или [0, 1] для всех элементов всех выборок, без потери информации.

Об этом не стоит слишком задумываться при NLP, поскольку TF-IDF, унитарное представление и Word2vec (как вы вскоре увидите) уже нормализованы. Об этом нужно помнить, если входные векторы признаков не нормализованы (например, если они представляют собой исходные частотности или количества слов).

Еще немного о терминологии. Среди исследователей нет единого мнения о том, что представляют собой перцептрон, многонейронный слой и глубокое обучение. Нам кажется удобным различать перцептрон и нейронную сеть при использовании производной функции активации для должного обновления весов. В этой книге мы будем употреблять в этом контексте термины «нейронная сеть» и «глубокое обучение», сохраняя для термина «перцептрон» важное место в истории.

Резюме

- Минимизация функции стоимости — путь к обучению.
- Сеть *обучается* на основе алгоритма обратного распространения ошибки.
- Степень обновления веса прямо пропорциональна его вкладу в ошибку модели.
- Нейронные сети являются механизмами оптимизации.
- Остерегайтесь ловушек в виде локальных минимумов при обучении, отслеживайте постепенное снижение ошибки.
- Математика нейронных сетей становится доступнее благодаря библиотеке Keras.

6 Умозаключения на основе векторов слов (Word2vec)

В этой главе

- Особенности создания векторов слов.
- Применение предобученных моделей для приложений.
- Решение реальных задач с помощью умозаключений на основе векторов слов.
- Визуализация векторов слов.
- Интересные применения вложений слов.

Одно из самых захватывающих новшеств в NLP — открытие векторов слов. Данная глава поможет разобраться, что это такое и как с их помощью делать весьма интересные вещи. Вам предстоит узнать, как выяснить нюансы смысла слов, которые нам не удавалось обнаружить в предыдущих главах.

Ранее мы не учитывали ближайший контекст слова. И игнорировали слова по бокам текущего. Мы игнорировали влияние соседей слова на его смысл, а также влияние этих взаимосвязей на общий смысл высказывания. Наша концепция мультимножества слов смешивала все слова документа в единую статистическую кучу. В этой главе мы будем работать с гораздо меньшими мультимножествами: всего из

нескольких соседних слов, обычно не более десяти токенов. Мы также убедимся, что такое смысловое соседство не распространяется на смежные предложения. Благодаря этому процессу обучение векторов слов будет ограничиваться только нужными словами.

Наши новые векторы слов смогут обнаруживать синонимы, антонимы и слова, которые просто относятся к одной категории, например люди, животные, места, растения, имена или понятия. Мы могли сделать это и раньше с помощью латентно-семантического анализа в главе 4, но благодаря более сильным ограничениям окрестности слов повысится точность их векторов. Латентно-семантический анализ слов, *n*-грамм и документов не захватывает даже все прямые значения слов, не говоря уже о подразумеваемых или скрытых смыслах. При использовании слишком больших мультимножеств LSA часть подтекста теряется.

ОПРЕДЕЛЕНИЕ

Векторы слов (word vectors) — это числовые представления семантики (смысла) слов, включая прямой и скрытый смысл. Эти векторы могут захватывать подтекст, например peopleness, animalness, placeness, thingness и даже conceptness, объединяя все это в плотный (без нулей) вектор из значений с плавающей точкой. Благодаря таким плотным векторам возможны семантические запросы и логические умозаключения.

6.1. Семантические запросы и аналогии

Так что можно делать с помощью этих замечательных векторов слов? Пробовали ли вы когда-нибудь вспомнить имя какой-либо знаменитости, о которой у вас были только самые общие представления? Например:

She invented something to do with physics in Europe in the early 20th century.

Если ввести эту фразу в Google или Bing, возможно, вы не получите интересующий вас прямой ответ: Marie Curie (Мария Кюри). Поиск Google выдаст только ссылки на списки знаменитых физиков: как мужчин, так и женщин. Вам придется пролистать несколько страниц, чтобы найти искомое. Но когда вы найдете Marie Curie, Google/Bing запомнят это. В следующий раз, когда вы будете искать ученого, они смогут выдать более подходящие результаты[1].

Благодаря векторам слов можно найти слова или имена, сочетающие смыслы слов *woman* (женщина), *Europe* (Европа), *physics* (физика), *scientist* (ученый) и *famous* (знаменитый), что приведет вас к искомому токену *Marie Curie*. И для этого необходимо всего лишь сложить векторы слов для всех сочетаний:

```
>>> answer_vector = wv['woman'] + wv['Europe'] + wv[physics'] +\
...     wv['scientist']
```

[1] Так было у нас при работе над этой книгой. Нам приходилось использовать приватные окна браузера, чтобы гарантировать идентичность наших результатов поиска с вашими.

В этой главе мы покажем, как выполнять подобный запрос и как вычесть из используемых для получения ответа векторов слов информацию о роде:

```
>>> answer_vector = wv['woman'] + wv['Europe'] + wv[physics'] +\
...     wv['scientist'] - wv['male'] - 2 * wv['man']
```

С помощью векторов слов можно вычесть *man* из *woman*!

6.1.1. Вопросы на аналогию

А если перефразировать вопрос в виде вопроса на аналогию? Что если наш запрос был вот таким: *Who is to nuclear physics what Louis Pasteur is to germs?*

Поиск Google, Bing и даже Duck Duck Go в этом случае практически бесполезен[1]. Но с помощью векторов слов решение этой задачи сводится к простому вычитанию *germs* из *Louis Pasteur* и сложению получившегося с *physics*:

```
>>> answer_vector = wv['Louis_Pasteur'] - wv['germs'] + wv['physics']
```

Если вас интересуют еще более хитрые аналогии относительно людей из не связанных между собой сфер деятельности, например музыкантов и ученых, то это тоже можно сделать: *Who is the Marie Curie of music?* Или *Marie Curie is to science as who is to music?*

Как вы думаете, какие арифметические действия нужно произвести над векторами слов для ответа на эти вопросы?

Наверное, вам встречались подобные вопросы в разделе аналогий стандартных экзаменов, таких как SAT, ACT или GRE. Иногда их записывают в формальной математической нотации:

```
MARIE CURIE : SCIENCE :: ? : MUSIC
```

Упрощает ли это подбор нужных математических действий над векторами слов? Один из вариантов:

```
>>> wv['Marie_Curie'] - wv['science'] + wv['music']
```

Можно отвечать на подобные вопросы, связанные не только с людьми и их родом деятельности, но, скажем, и со спортивными командами и соответствующими городами: *The Timbers are to Portland as what is to Seattle?*

В стандартизованной форме тестов:

```
TIMBERS : PORTLAND :: ? : SEATTLE
```

Чаще всего в стандартизованных тестах используются слова из словаря и задаются менее интересные вопросы, например:

```
WALK : LEGS :: ? : MOUTH
```

или

```
ANALOGY : WORDS :: ? : NUMBERS
```

[1] Попробуйте сами, если не верите.

Ответ на подобные вопросы а-ля «вертится на языке» не составляет труда для векторов слов даже без вариантов ответа. При попытке вспомнить название или слово наличие нескольких вариантов ответа только усложняет дело. И здесь на помощь приходит NLP с векторами слов.

Векторы слов могут ответить на подобные расплывчатые вопросы и задачи аналогии. Они могут помочь в запоминании любого слова или названия, вертящегося на кончике вашего языка, лишь бы в вашем словаре векторов слов содержался вектор для искомого ответа[1]. Векторы слов отлично работают даже в случае вопросов, которые невозможно сформулировать в виде поискового запроса или задачи аналогии. О некоторой не относящейся к запросам арифметике векторов слов можно узнать из подраздела 6.2.1.

6.2. Векторы слов

В 2012 году Томаш Миколов, тогда еще стажер в Microsoft, нашел способ кодирования смысла слов в векторах относительно небольшой размерности[2]. Миколов обучил нейронную сеть[3] предсказывать вхождения слов поблизости целевых слов. В 2013-м, уже в Google, Миколов с соратниками выпустил программное обеспечение для создания этих векторов слов и назвал его Word2vec[4].

Word2vec усваивает значения слов просто за счет обработки большого корпуса немаркированного текста. Нет нужды маркировать слова в словаре Word2vec. Не нужно рассказывать алгоритму обеспечения, что Мария Кюри — ученый, а «Тимберс» — футбольная команда, Сиэтл — город или Портленд — два города в штатах Орегон и Мен. И не надо рассказывать, что футбол — вид спорта, команда — группа людей или города — как географические, так и демографические объекты. Word2vec может сам обучиться всей этой информации, а также многому другому! Все, что ему требуется, — достаточно большой корпус, в котором Мария Кюри, «Тимберс» и Портленд упоминались бы рядом с другими словами, связанными с наукой, футболом и городами соответственно.

Основной источник мощи Word2vec кроется именно в использовании в нем машинного обучения без учителя. Мир полон немаркированных, не распределенных по категориям, неструктурированных текстов на естественных языках.

---

1 В случае предобученной модели векторов слов Google нужное слово практически наверняка найдется среди новостных лент на 100 миллиардов слов, на которых она обучалась, разве что оно было придумано после 2013 года.

2 Размерность векторов слов обычно составляет от 100 до 500, в зависимости от масштабов информации в использовавшемся для их обучения корпусе.

3 Всего лишь однослойную сеть, так что практически любая линейная модель машинного обучения смогла бы ее заменить. Логистическая регрессия, усеченное SVD, линейный дискриминантный анализ и наивный байесовский классификатор — все они также отлично справились бы с этой задачей.

4 *Mikolov T., Corrado D.* Efficient Estimation of Word Representations in Vector Space, Sep. 2013 (https://arxiv.org/pdf/1301.3781.pdf).

Обучение без учителя (unsupervised learning) и *с учителем* (supervised learning) — два принципиально различных подхода к машинному обучению.

Машинное обучение с учителем

При машинном обучении с учителем тренировочные данные обязательно должны быть как-то маркированы. Пример таких меток — категорийная метка спама на СМС в главе 4. Еще один пример — количественное значение числа лайков для твита. Именно машинное обучение с учителем большинство людей подразумевают под машинным обучением. Модель с учителем может измениться в лучшую сторону только путем измерения различий между ожидаемыми выходными данными (метками) и полученными предсказаниями.

Машинное обучение без учителя позволяет машине обучаться непосредственно на данных без какой-либо помощи со стороны человека. Тренировочные данные могут быть не упорядочены, не структурированы и никак не маркированы. Поэтому алгоритмы машинного обучения без учителя, вроде Word2vec, идеально подходят для текстов на естественных языках.

Машинное обучение без учителя

При машинном обучении без учителя модель обучается выполнять задачу без каких-либо меток на основе одних только исходных данных. Примерами машинного обучения без учителя могут послужить алгоритмы кластеризации, например метод *k*-средних и DBSCAN. Также методиками машинного обучения без учителя являются алгоритмы понижения размерности, такие как метод главных компонент (PCA) и метод стохастических вложений соседей на основе распределения Стьюдента (t-SNE). При машинном обучении без учителя модель ищет закономерности в связях между самими точками данных. Сделать модель без учителя «умнее» (точнее) можно, предоставив ей больше данных.

Вместо того чтобы пытаться обучить нейронную сеть усваивать значения целевых слов непосредственно (на основе меток для этих значений), мы учим ее предсказывать ближайшие к целевому слова. Так что в этом смысле метки у нас есть: соседние слова, которые мы пытаемся предсказать. Поскольку источником меток служит сам набор данных, а никакого присвоения меток вручную не требуется, алгоритм Word2vec является алгоритмом машинного обучения без учителя.

Еще одна предметная область, где применяется машинное обучение без учителя, — моделирование временных рядов. Такие модели часто обучаются предсказанию следующего значения в последовательности на основе окна предыдущих значений. Задачи временных рядов во многом удивительно похожи на задачи обработки естественного языка, поскольку также связаны с упорядоченными последовательностями значений (слов или чисел).

Но сами предсказания — не главное в Word2vec, — а лишь средство достижения цели. Нас интересует внутреннее представление, вектор, постепенно формируемый Word2vec для генерации этих предсказаний. Это представление захватывает намного больше смысла целевого слова (его семантики), чем векторы «слово — тема», полученные в результате латентно-семантического анализа и латентного размещения Дирихле в главе 4.

ПРИМЕЧАНИЕ

Модели, которые обучаются, пытаясь заново предсказать подаваемое на их вход значение с помощью внутреннего представления низкой размерности, называются *автокодировщиками* (autoencoders). Эта идея может показаться странной. Все равно что просить машину просто вернуть переданное ей значение только при условии, что она не может «записать» задаваемый ей вопрос. Машине приходится сжимать этот вопрос в краткое представление. Причем она должна применять один и тот же алгоритм (функцию) сжатия для всех задаваемых ей вопросов. Машина обучается новому сокращенному (векторному) представлению ваших высказываний.

Если хотите больше узнать про модели глубокого обучения без учителя, создающие сжатые представления многомерных объектов (например, слов), поищите в Интернете по слову «автокодировщик»[1]. Это распространенный способ познакомиться с нейронными сетями, поскольку их можно использовать практически для любого набора данных.

Word2vec может обучиться таким вещам, которые вы вряд ли ассоциируете со всяким словом. Знаете ли вы, что каждому слову соответствует какая-либо география, тональность и род? Если у определенного слова из корпуса есть какое-то свойство, например, *placeness*, *peopleness*, *conceptness* или *femaleness*, то у всех остальных слов также будет показатель для этих свойств в векторах слов. Смысл слова «переходит» на соседние при усвоении Word2vec их векторов.

Все слова в корпусе представляются в виде числовых векторов, похожих на обсуждавшиеся в главе 4 векторы «слово — тема». Только на этот раз темы более конкретные, узкие. В LSA для перехода значения слова на другие достаточно, чтобы они встречались в одном документе и были включены в их векторы «слово — тема». В случае векторов слов Word2vec слова должны встречаться рядом друг с другом — обычно в границе пяти слов друг от друга, причем в одном предложении. И веса тем векторов слов Word2vec можно складывать и вычитать, получая новые — осмысленные! — векторы слов.

Чтобы лучше понять, что такое векторы слов, рассматривайте их как списки весов (показателей). Каждый вес (показатель) соответствует конкретному измерению смысла для данного слова (листинг 6.1).

---

[1] См. веб-страницу Unsupervised Feature Learning and Deep Learning Tutorial по адресу http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/.

Листинг 6.1. Вычисляем ness-векторы[1]

```
>>> from nlpia.book.examples.ch06_nessvectors import *
>>> nessvector('Marie_Curie').round(2)
placeness     -0.46
peopleness     0.35
animalness     0.17
conceptness   -0.32
femaleness     0.26
```

Не импортируйте этот модуль, если у вас мало оперативной памяти или свободного времени. Предобученная модель Word2vec огромна

Мы убеждены, что измерения вашего ness-вектора окажутся гораздо полезнее и интереснее, например trumpness и ghandiness[2]

С помощью инструментов из nlpia можно вычислить ness-векторы для любого слова или *n*-граммы в словаре Word2vec (https://github.com/totalgood/nlpia/blob/master/src/nlpia/book/examples/ch06_nessvectors.py). И такой подход годится для любых ness-компонентов, какие только можно себе представить.

Миколов разработал алгоритм Word2vec, когда пытался придумать способ численного представления слов в виде векторов. Его не устраивала менее точная математика тональностей слов, знакомая нам по главе 4. Целью были *векторные умозаключения* (vector-oriented reasoning), вроде тех, которые мы производили в предыдущем разделе для вопросов на аналогию. Может показаться, что это довольно странная концепция, но на самом деле она означает возможность выполнения арифметических действий над векторами слов с получением результата, который окажется осмысленным после обратного преобразования векторов в слова. С помощью сложения и вычитания векторов слов можно получать *умозаключения* относительно смысла представляемых ими слов и отвечать на аналогичные приведенным выше вопросы вот так[3]:

```
wv['Timbers'] - wv['Portland'] + wv['Seattle'] = ?
```

Хотелось бы получить в результате этих арифметических действий (умозаключений на основе векторов слов) следующее:

```
wv['Seattle_Sounders']
```

Также наш вопрос на аналогию *'Marie Curie' is to 'physics' as _ is to 'classical music'?* можно рассматривать как следующее математическое выражение:

```
wv['Marie_Curie'] - wv['physics'] + wv['classical_music'] = ?
```

В этой главе мы хотели бы усовершенствовать представления векторов слов LSA, приведенные в предыдущей главе. Векторы тем, сформированные на основе целых документов с помощью LSA, отлично подходят для классификации документов, семантического поиска и кластеризации. Но генерируемые LSA векторы

---

[1] Суффикс -ness образует в английском языке от прилагательных существительные со значением «качество», «состояние». — *Примеч. пер.*

[2] Образованы от фамилий Дональда Трампа (нынешнего президента США) и Махатмы Ганди (индийского политического и общественного деятеля). — *Примеч. пер.*

[3] Для тех, кто не интересуется спортом: «Портленд Тимберс» и «Сиэтл Саундерс» — команды высшей футбольной лиги США.

«тема — слово» дают недостаточную точность для семантических умозаключений или классификации и кластеризации коротких фраз или составных слов. Скоро вы узнаете, как обучать однослойные нейронные сети, необходимые для генерации этих более точных и интересных векторов слов. И поймете, почему они заменили векторы «слово — тема» LSA во многих сферах применения, связанных с обработкой коротких документов или предложений.

6.2.1. Векторные умозаключения

Word2vec впервые публично продемонстрировали в 2013 году на конференции ACL[1]. Доклад с сухим названием «Лингвистические закономерности в непрерывном пространстве представлений слов» описывал удивительно точную языковую модель. Вложения Word2vec были вчетверо точнее (45 %), чем эквивалентные модели LSA (11 %) при ответе на вопросы аналогий, подобных вышеупомянутым[2]. Повышение точности было таким разительным, что первоначальный вариант статьи Миколова отвергла Международная конференция по усвоению представлений. Рецензенты посчитали, что эффективность модели слишком хороша, чтобы это было правдой. Команде Миколова потребовался почти год для обнародования исходного кода и принятия статьи на конференцию Ассоциации вычислительной лингвистики.

Внезапно оказалось, что благодаря векторам слов можно отвечать на вопросы вроде:

```
Portland Timbers + Seattle - Portland = ?
```

с помощью векторной алгебры (рис. 6.1).

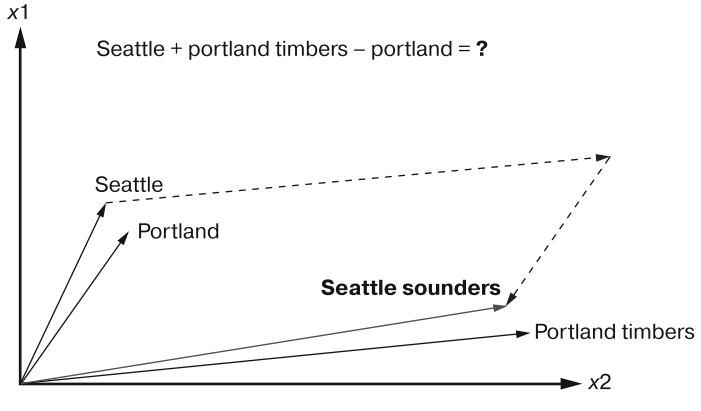


Рис. 6.1. Геометрическое представление математики Word2vec

[1] См. статью: *Mikolov T., Yih W., Zweig G.* Linguistic Regularities in Continuous Space Word Representations — по адресу https://www.aclweb.org/anthology/N13-1090.

[2] См. интервью, взятое у Томаша Миколова Радимом Ржегуржеком, по адресу https://rare-technologies.com/rrp#episode_1_tomas_mikolov_on_ai.

Модель Word2vec содержит информацию о взаимосвязях между словами, в том числе их подобии. Модель Word2vec «знает», что терм *Portland* находится примерно на таком же расстоянии от *Portland Timbers*, что и *Seattle* от *Seattle Sounders*. И эти расстояния (разности между парами векторов) лежат примерно в одном направлении. Поэтому такая модель применима для ответа на наш вопрос аналогии в сфере спортивных команд. Если прибавить разность между *Portland* и *Seattle* к вектору, соответствующему *Portland Timbers*, то в результате мы окажемся неподалеку от вектора для терма *Seattle Sounders* (уравнение 6.1).

Уравнение 6.1. Вычисление ответа на вопрос о футбольных командах

$$\begin{bmatrix} 0{,}0168 \\ 0{,}007 \\ 0{,}247 \\ \dots \end{bmatrix} + \begin{bmatrix} 0{,}093 \\ -0{,}028 \\ -0{,}214 \\ \dots \end{bmatrix} - \begin{bmatrix} 0{,}104 \\ 0{,}0883 \\ -0{,}318 \\ \dots \end{bmatrix} = \begin{bmatrix} 0{,}006 \\ -0{,}109 \\ 0{,}352 \\ \dots \end{bmatrix}$$

Вектор, полученный в результате сложения и вычитания векторов слов, практически никогда не будет в точности совпадать с одним из векторов словаря. В векторах слов Word2vec обычно сотни измерений, каждое из которых содержит непрерывные вещественные значения. Тем не менее ближайший к полученному вектор из вашего словаря обычно будет ответом на вопрос NLP. А соответствующее этому ближайшему вектору английское слово — ответом на естественном языке на ваш вопрос о спортивных командах и городах.

Word2vec дает возможность преобразовывать векторы подсчетов встречаемости и частотностей токенов естественного языка в векторное пространство векторов Word2vec намного более низкой размерности. Там можно проделать необходимые арифметические действия и преобразовать результат обратно в пространство естественного языка. Можете представить, насколько удобна эта возможность для чат-ботов, поисковых механизмов, систем формирования ответов на вопросы и алгоритмов извлечения информации.

ПРИМЕЧАНИЕ

В первоначальном варианте статьи Миколов и его соавторы добились точности ответов всего в 40 %. Но тогда, в 2013 году, это значительно превосходило любой другой подход к семантическим умозаключениям. С момента публикации этой статьи эффективность Word2vec возросла за счет обучения на исключительно большом корпусе. Эталонная реализация обучалась на 100 миллиардах слов из корпуса Google News, и именно ее мы будем часто использовать в этой книге.

Упомянутая команда исследователей также обнаружила, что порядок разности между словами в единственном и множественном числе обычно примерно одинаков, как и его направленность (уравнение 6.2).

Уравнение 6.2. Расстояние между версиями одного слова в единственном и множественном числе

$$\vec{x}_{coffee} - \vec{x}_{coffees} \approx \vec{x}_{cup} - \vec{x}_{cups} \approx \vec{x}_{cookie} - \vec{x}_{cookies}$$

Но открытие ученых этим не ограничивается. Они также обнаружили, что взаимосвязи расстояний намного выходят за рамки связей единственного и множественного числа. Расстояния применимы и для других семантических связей. Создатели Word2vec вскоре выяснили, что могут отвечать и на вопросы, связанные с географией, культурой и демографией, например:

```
"San Francisco is to California as what is to Colorado?"
San Francisco - California + Colorado = Denver
```

Дополнительные поводы воспользоваться векторами слов

Векторные представления слов удобны не только для умозаключений и задач на аналогию, но и прочих, к которым применяются модели векторного пространства естественного языка. От сопоставления с шаблоном до моделирования и визуализации точность и удобство вашего конвейера NLP возрастет, если вы будете знать, как использовать векторы слов из этой главы.

Например, далее мы покажем, как визуализировать векторы слов на двумерных семантических картах, подобных той, что показана на рис. 6.2. Их можно рассматривать как рисованную карту популярных туристических мест или одну из тех импрессионистских карт на остановках общественного транспорта. На этих картах близкие друг к другу семантически или географически объекты схлопываются. В картах художник подбирает масштаб и расположение значков для различных мест, соответствующие ощущению от них. В случае векторов слов машина также может «чувствовать» слова и понимать, какое расстояние должно быть между ними. Таким

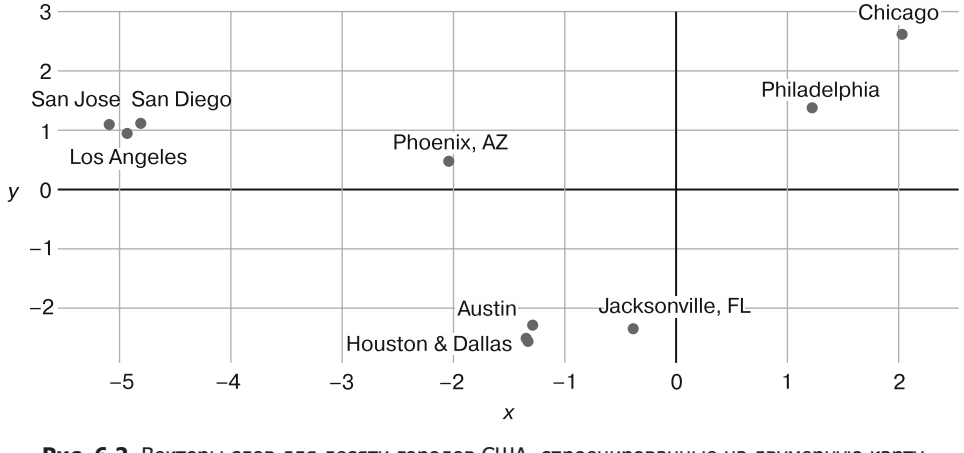


Рис. 6.2. Векторы слов для десяти городов США, спроецированные на двумерную карту

образом, машина может генерировать с помощью описываемых в этой главе векторов слов импрессионистские карты, как на рис. 6.2[1].

Если эти города вам знакомы, то вы заметили, что с географической точки зрения эта карта неточна, зато с семантической — очень хороша. Например, Лейн часто путает два больших техасских города: Хьюстон и Даллас, и их векторы слов практически одинаковы. Векторы слов для больших калифорнийских городов формируют, по его мнению, изящный культурный треугольник.

Векторы слов также отлично подходят для чат-ботов и поисковых систем. В этих сферах применения векторы слов могут помочь преодолеть жесткость и хрупкость шаблонов при сопоставлении с ключевыми словами. Допустим, мы ищем информацию о знаменитости из Хьюстона, штат Техас, но не знаем, что он переехал в Даллас. Из рис. 6.2 видно, что семантический поиск с использованием векторов слов дает возможность легко производить поиск, в котором участвуют названия таких городов, как Даллас и Хьюстон. Хотя для символьных шаблонов разница между *tell me about a Denver omelette* и *tell me about the Denver Nuggets* непонятна, она очевидна для шаблонов на основе векторов слов. Основанные на векторах слов шаблоны, вероятно, смогут различить еду (омлет) и баскетбольную команду («Наггетс»[2]) и адекватно ответить пользователю, задающему вопрос о том или другом.

6.2.2. Вычисление представлений Word2vec

Векторы слов отражают семантическое значение в виде векторов в контексте тренировочного корпуса. Это позволяет не только отвечать на вопросы на аналогию, но и делать более общие выводы относительно смысла слов с помощью векторной алгебры. Но как вычислить эти самые векторные представления? Существует два способа обучения вложений Word2vec.

- При подходе со *skip-граммами* контекст слов (выходные слова) предсказывается на основе интересующего нас (входного) слова.
- При подходе с *непрерывным мультимножеством слов* (CBOW) целевое (выходное) слово предсказывается по близлежащим (входным) словам.

Мы покажем, как и когда использовать каждый из этих подходов для обучения модели Word2vec в следующих разделах.

Вычисление представлений векторов слов может требовать значительного расхода ресурсов. К счастью, для большинства приложений не нужно вычислять собственные векторы слов. Для широкого спектра приложений можно воспользоваться предобученными представлениями. Компании, оперирующие большими корпусами данных, которые могут позволить себе эти вычисления, обычно делают общедоступ-

---

[1] Код для генерации этих интерактивных двумерных графиков слов можно найти по адресу https://github.com/totalgood/nlpia/blob/master/src/nlpia/book/examples/ch06_w2v_us_cities_visualization.py.

[2] Наггетсы — это не только баскетбольная команда, но и закуска из филе куриной грудки в хрустящей панировке, обжаренной в масле. — *Примеч. пер.*

ными свои предобученные модели векторов слов. Далее в этой главе мы познакомим вас с применением этих предобученных векторов слов, таких как GloVe и fastText.

СОВЕТ

Предобученные представления векторов слов доступны для таких корпусов, как «Википедия», DBPedia, Twitter и Freebase[1]. Эти предобученные модели — отличные отправные пункты для ваших приложений, использующих векторы слов.

- Компания Google предоставляет предобученную модель Word2vec на основе статей из корпуса Google News на английском языке[2].
- Facebook опубликовал свою модель слов под названием *fastText* для 294 языков[3].

Но если в вашей предметной области используются специализированный словарь или семантические связи, универсальных моделей слов недостаточно. Например, если *python* («питон») должно однозначно соответствовать языку программирования, а не рептилии, то понадобится предметно-ориентированная модель слов. Если необходимо ограничить векторы слов определенной предметной областью, то нужно обучать их на текстах из этой предметной области.

Подход skip-gram

При подходе к обучению с использованием skip-грамм мы пытаемся предсказать окружающее окно слов по входному слову. В посвященном художнику Клоду Моне предложении в нашем следующем примере тренировочными входными данными для сети служит *painted*. Соответствующие skip-граммы для тренировочного выходного примера показаны на рис. 6.3. Предсказанными словами для этих skip-грамм являются близлежащие слова: *Claude*, *Monet*, *the* и *Grand*.

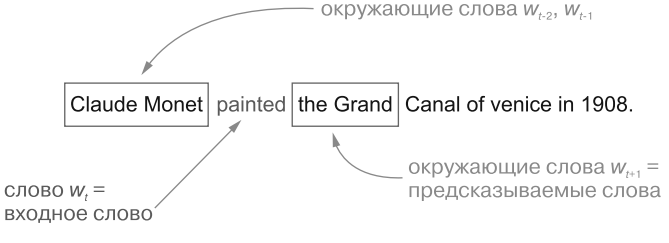


Рис. 6.3. Тренировочные входной и выходной примеры для подхода с использованием skip-грамм

[1] См. веб-страницу GitHub — 3Top/word2vec-api: Simple web service providing a word embedding model по адресу https://github.com/3Top/word2vec-api#where-to-get-a-pretrained-model.

[2] Исходную 300-мерную модель Word2vec Google можно найти на Google Drive по адресу https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM.

[3] См. веб-страницу GitHub — facebookresearch/fastText: Library for fast text representation and classification по адресу https://github.com/facebookresearch/fastText.

ЧТО ТАКОЕ SKIP-ГРАММА?

Skip-граммы — это *n*-граммы с промежутками вследствие пропуска промежуточных токенов. В данном примере мы предсказываем Monet по входному токену *painted* и пропускаем токен *Monet*.

Структура используемой для предсказания окружающих слов нейронной сети аналогична сетям, о которых мы говорили в главе 5. Как можно видеть из рис. 6.4, сеть состоит из двух слоев весов, где скрытый слой состоит из n нейронов. n — размерность векторов, используемых для представления слов. Как входной, так и выходной слой содержат M нейронов, где M — число слов в словаре модели. Функция активации выходного слоя — многомерная логистическая функция, часто применяемая для задач классификации.

Что такое многомерная логистическая функция

Многомерная логистическая функция часто используется в качестве функции активации в выходном слое нейронных сетей, когда цель сети — решение задачи классификации. Многомерная логистическая функция размазывает выходные результаты по интервалу от 0 до 1, причем сумма всех выходных сигналов всегда равна 1. Таким образом, результаты выходного слоя при многомерной логистической функции можно рассматривать как вероятности.

Для каждого из K выходных узлов выходное значение многомерной логистической функции можно вычислить с помощью нормированной экспоненциальной функции:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Выходной вектор трехнейронного выходного слоя выглядит вот так (уравнение 6.3).

Уравнение 6.3. Пример трехмерного вектора

$$v = \begin{bmatrix} 0{,}5 \\ 0{,}9 \\ 0{,}2 \end{bmatrix}$$

«Размазанный» вектор после применения многомерной логистической функции активации будет выглядеть как в уравнении 6.4.

Уравнение 6.4. Пример трехмерного вектора после применения многомерной логистической функции

$$\sigma(v) = \begin{bmatrix} 0{,}309 \\ 0{,}461 \\ 0{,}229 \end{bmatrix}$$

Обратите внимание, что сумма этих значений (округленных до трех значимых цифр) примерно равна 1, как у распределения вероятности.

Рисунок 6.4 демонстрирует входной и выходной сигналы числовой сети для первых двух окружающих слов. В данном случае входное слово — *Monet*, а ожидаемый выходной сигнал сети — *Claude* или *painted*, в зависимости от обучающей пары.

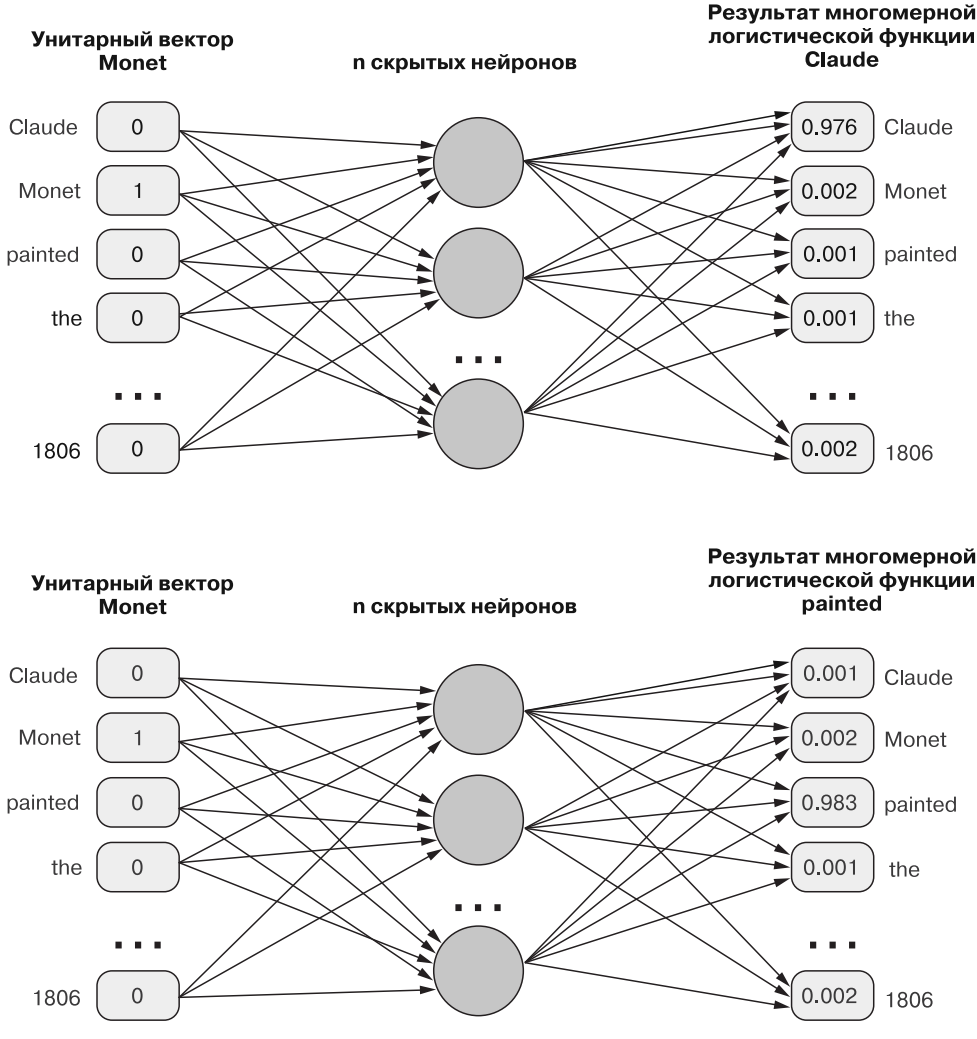


Рис. 6.4. Пример сети для обучения с использованием skip-грамм

ПРИМЕЧАНИЕ

Если взглянуть на структуру нейронной сети для вложений слов, можно заметить, что реализация очень похожа на приведенную в главе 5.

Как нейронная сеть усваивает векторные представления

Для обучения модели Word2vec мы используем методики из главы 2. Например, в табл. 6.1 w_t соответствует унитарному вектору для токена на позиции t. Так что для обучения модели Word2vec с помощью skip-граммы размером (радиусом) два слова нужно учитывать два слова перед и после каждого из целевых слов. После этого можно воспользоваться ориентированным на 5-граммы токенизатором из главы 2 для представления следующего предложения:

```
>>> sentence = "Claude Monet painted the Grand Canal of Venice in 1806."
```

в виде десяти 5-грамм с входным словом по центру по одной для каждого из десяти слов исходного предложения.

Таблица 6.1. Десять 5-грамм для посвященного Моне предложения

Входное слово	Ожидаемое выходное слово w_{t-2}	Ожидаемое выходное слово w_{t-1}	Ожидаемое выходное слово w_{t+1}	Ожидаемое выходное слово w_{t+2}
Claude			Monet	painted
Monet		Claude	painted	the
painted	Claude	Monet	the	Grand
the	Monet	painted	Grand	Canal
Grand	painted	the	Canal	of
Canal	the	Grand	of	Venice
of	Grand	Canal	Venice	in
Venice	Canal	of	in	1908
in	of	Venice	1908	
1908	Venice	in		

Основа для обучения нейронной сети — теперь тренировочный набор данных из входного и окружающих его (выходных) слов. В случае четырех окружающих слов необходимо четыре итерации обучения, при которых по входному слову предсказывается каждое из выходных.

Перед показом нейронной сети каждое из слов представляется в виде унитарного вектора (см. главу 2). Выходной вектор для выполняющей сопоставление слова вектору (embedding) нейронной сети также подобен унитарному. Многомерная логистическая функция активации узлов выходного слоя (по одному для каждого токена в словаре) вычисляет вероятность обнаружения выходного слова среди тех, что окружают входное. Выходной вектор вероятностей слов затем можно преобразовать в унитарный вектор, в котором слову с максимальной вероятностью соответствует 1, а всем остальным словам — 0. Это упрощает вычисление функции потерь.

По завершении обучения нейронной сети вы увидите, что веса отражают семантический смысл. Благодаря унитарному преобразованию токенов каждая строка в матрице весов представляет одно из слов словаря для нашего корпуса. После обучения векторы семантически похожих слов будут подобны, поскольку они были обучены для предсказания схожих окружающих слов. *Просто фантастика!*

По завершении обучения, когда вы решите не обучать модель слов далее, выходной слой сети можно не учитывать. В качестве вложений используются только веса входных сигналов скрытого слоя. Другими словами: матрица весов и является вложениями слов. После этого скалярное произведение унитарного вектора, соответствующего входному терму, и весов становится *вложением вектора слов*.

Поиск векторов слов с помощью линейной алгебры

Веса скрытого слоя нейронной сети часто представляют в виде матрицы: по столбцу на входной нейрон, по строке на выходной. Благодаря такому представлению матрицу весов можно умножить на вектор-столбец входных сигналов, поступающих с предыдущего слоя, и получить вектор-столбец выходных сигналов для следующего (рис. 6.5). Так, если умножить (с помощью скалярного произведения) унитарный вектор-*строку* на полученную в результате обучения матрицу весов, вы получите вектор, содержащий по одному весу от каждого нейрона (от каждого столбца матрицы). Аналогично и при умножении матрицы весов (используя скалярное произведение) на унитарный вектор-*столбец* для интересующего вас слова.

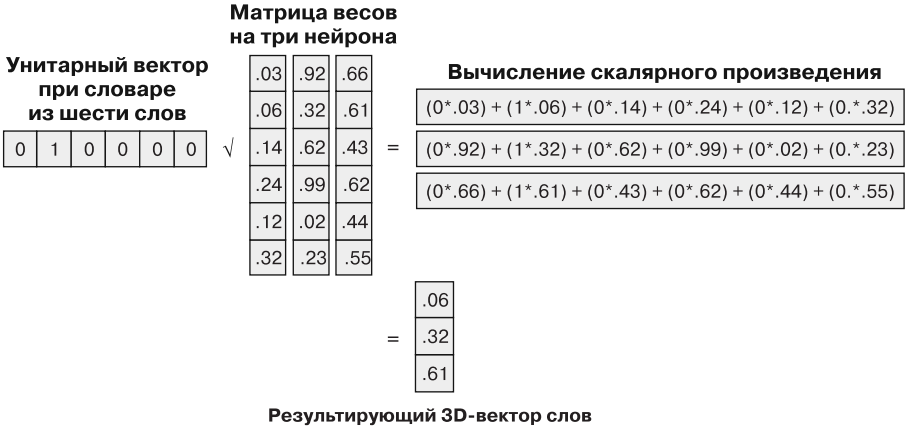


Рис. 6.5. Преобразование унитарного вектора в вектор слов

Конечно, скалярное произведение унитарного вектора просто извлекает из матрицы весов строку, содержащую веса для соответствующего слова и являющуюся нужным вектором слова. Так что эту строку можно просто выбрать по номеру строки слова или числовому индексу из словаря.

Подход с непрерывным мультимножеством слов

При подходе с непрерывным мультимножеством мы пытаемся предсказать центральное слово по окружающим его словам (см. рис. 6.5 и 6.6 и табл. 6.2). Вместо создания пар входных и выходных токенов мы создаем в качестве входного вектора федеративный вектор всех окружающих термов. Он представляет собой сумму всех унитарных векторов токенов, окружающих центральный, целевой токен.

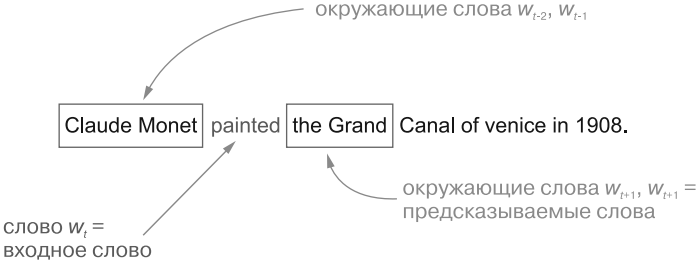


Рис. 6.6. Тренировочные входной и выходной примеры для подхода CBOW

Таблица 6.2. Десять 5-грамм CBOW из посвященного Моне предложения

Входное слово w_{t-2}	**Входное слово w_{t-1}**	**Входное слово w_{t+1}**	**Входное слово w_{t+2}**	**Ожидаемое выходное слово w_t**
		Monet	painted	Claude
	Claude	painted	the	Monet
Claude	Monet	the	Grand	painted
Monet	painted	Grand	Canal	the
painted	the	Canal	of	Grand
the	Grand	of	Venice	Canal
Grand	Canal	Venice	in	of
Canal	of	in	1908	Venice
of	Venice	1908		in
Venice	in			1908

Вы можете создавать на основе тренировочных наборов данных в качестве входных свои федеративные векторы и отображать их в целевое слово как выходное. Такой вектор представляет собой сумму унитарных векторов тренировочных пар $w_{t-2} + w_{t-1} + w_{t+1} + w_{t+2}$ окружающих слов. Далее вы создаете тренировочные пары, используя федеративный вектор в качестве входных данных и целевое слово w_t в качестве выходных. Во время обучения входной сигнал берется из многомерной логистической функции выходного узла с наибольшей вероятностью (рис. 6.7).

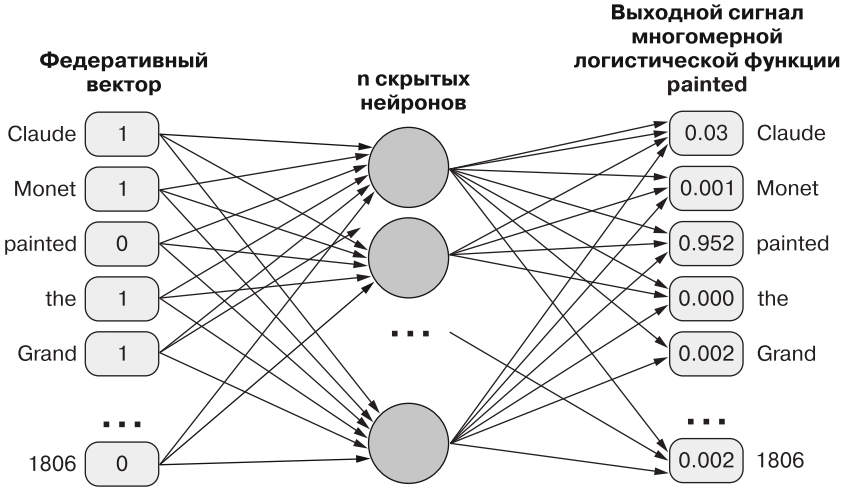
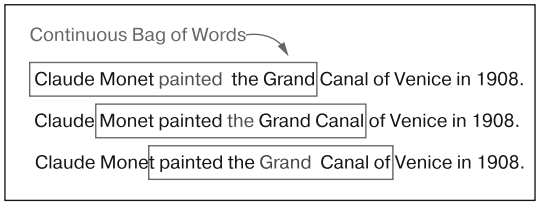


Рис. 6.7. Сеть Word2vec CBOW

Непрерывное мультимножество слов или просто мультимножество слов?

В предыдущих главах мы познакомили вас с мультимножеством слов, но чем оно отличается от непрерывного мультимножества? Для установления взаимосвязей между словами в предложении мы двигаем скользящее окно по предложению, выбирая для целевого слова окружающие. Все слова в пределах скользящего окна считаются содержимым мультимножества слов для целевого посередине окна.



Пример для непрерывного мультимножества слов со скользящим окном длиной пять слов, которое движется по предложению *Claude Monet painted the Grand Canal of Venice in 1908*. Целевым (центральным) словом первого скользящего окна CBOW является *painted*, а четырьмя окружающими его словами — *Claude*, *Monet*, *the* и *Grand*

Skip-граммы и CBOW: какой подход когда использовать

Миколов подчеркивал, что подход со skip-граммами хорошо работает для маленьких корпусов и редко встречающихся термов. При этом подходе благодаря структуре сети число примеров возрастает. Но подход с непрерывным мультимножеством слов

демонстрирует более высокую точность для часто встречающихся слов, и обучение занимает гораздо меньше времени.

Вычислительные приемы Word2vec

С момента выхода первоначальной статьи производительность моделей Word2vec значительно выросла благодаря различным вычислительным приемам. В этом разделе мы рассмотрим три подобных усовершенствования.

Часто встречающиеся биграммы. Некоторые слова то и дело встречаются в сочетании с другими словами — например, за словом *Elvis* (Элвис) нередко следует *Presley* (Пресли) — а значит, часто формируют биграммы. Поскольку слово *Elvis* с высокой вероятностью сочетается с *Presley*, на подобном предсказании много не выиграешь. Для повышения точности вложений Word2vec команда Миколова включила некоторые биграммы и триграммы в словарь Word2vec в качестве термов. Они[1] использовали частотность совместной встречаемости для идентификации биграмм и триграмм, которые следует считать едиными термами, на основе следующей формулы (уравнение 6.5).

Уравнение 6.5. Функция оценки биграмм

$$\text{показатель}(w_i, w_j) = \frac{\text{количество}(w_i, w_j) - \delta}{\text{количество}(w_i) \times \text{количество}(w_j)}$$

Если для слов w_i и w_j показатель оказывается выше и превышает пороговое значение δ, то они включаются в словарь Word2vec как парный терм. Вы можете заметить, что словарь модели содержит такие термы, как *New_York* и *San_Francisco*. В токене часто встречающих биграмм слова соединяются с помощью какого-либо символа (обычно _). Таким образом, этим термам будет соответствовать один унитарный вектор вместо двух (например, для *San* и *Francisco*).

Еще один эффект пар слов заключается в том, что словосочетание часто отражает отличный от отдельных слов смысл. Например, смысл названия футбольной команды «Портленд Тимберс» отличается от смыслов отдельных слов «Портленд» и «Тимберс». Но благодаря добавлению часто встречающихся биграмм, вроде названий команд, в модель Word2vec удается легко включить их в унитарные векторы для обучения модели.

Прореживание часто встречающихся токенов. Еще одно усовершенствование исходного алгоритма для повышения точности — прореживание (субдискретизация, subsampling) часто встречающихся слов. Распространенные английские слова, например *the* или *a*, зачастую неинформативны, а учет совместной встречаемости *the* с множеством разнообразных других существительных в корпусе может при-

[1] Больше информации можно найти в опубликованной командой Миколова статье по адресу https://arxiv.org/pdf/1310.4546.pdf.

вести к менее осмысленным взаимосвязям между словами, засоряя представление Word2vec подобным обучением ложному семантическому сходству.

ВАЖНО

У всех слов есть какой-либо смысл, включая стоп-слова. Так что не нужно полностью игнорировать или пропускать стоп-слова при обучении векторов слов или компоновки словаря. Кроме того, в силу частого использования векторов слов в порождающих моделях (вроде той, которую Коул применял для составления предложений в этой книге) стоп-слова и другие распространенные слова должны включаться в словарь и влиять на векторы соседних с ними слов.

Для снижения влияния часто встречающихся слов, таких как стоп-слова, во время обучения слова подвергаются прореживанию обратно пропорционально их частотности. Аналогично эффекту IDF на векторы TF-IDF. Часто встречающиеся слова меньше влияют на вектор, чем более редкие. Для определения вероятности прореживания конкретного слова Томаш Миколов использовал следующую формулу. Эта вероятность определяет, будет ли включено конкретное слово в конкретную skip-грамму при обучении (уравнение 6.6).

Уравнение 6.6. Вероятность прореживания в статье Миколова о Word2vec

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

В реализации Word2vec на C++ используется несколько иная формула для вероятности прореживания, чем приведенная в вышеупомянутой статье, но эффект тот же (уравнение 6.7).

Уравнение 6.7. Вероятность прореживания в коде Миколова для Word2vec

$$P(w_i) = \frac{f(w_i) - t}{f(w_i)} - \sqrt{\frac{t}{f(w_i)}}$$

В предыдущих уравнениях $f(w_i)$ отражает частотность слова в корпусе, а t — пороговое значение частотности, при превышении которого необходимо применить вероятность прореживания. Это пороговое значение зависит от размера корпуса, средней длины документа и разнообразия используемых в этих документах слов. В литературе обычно упоминаются значения от 10^{-5} до 10^{-6}.

Если слово встречается десять раз во всем корпусе, словарь которого состоит из миллиона различных слов, а пороговое значение прореживания равно 10^{-6}, то вероятность того, что оно останется в любой конкретной n-грамме, равна 68 %. При составлении n-грамм во время токенизации оно будет пропускаться в 32 % случаев.

Миколов продемонстрировал, что прореживание повышает точность векторов слов для таких задач, как формирование ответов на вопросы.

Отрицательная дискретизация. Последний придуманный Миколовым прием — отрицательная дискретизация. Демонстрация сети отдельного тренировочного примера с парой слов приведет к обновлению всех весов сети, что изменит значения всех векторов для всех слов словаря. Но если последний содержит тысячи или миллионы слов, обновлять все веса для большого унитарного вектора нерационально. Для ускорения обучения моделей векторов слов Миколов воспользовался отрицательной дискретизацией.

Вместо того чтобы обновлять веса для всех слов, не включенных в окна слов, Миколов предложил выбирать лишь несколько отрицательных примеров (в выходном векторе) для обновления их весов. Вместо обновления всех весов выбираются *n* отрицательных примеров пар слов (не соответствующих целевому выходному сигналу для данного примера) и обновления весов, внесших вклад в их конкретный выходной сигнал. При этом требуется намного меньше вычислений, а быстродействие обученной сети не слишком снизится.

ПРИМЕЧАНИЕ

> При обучении модели слов на маленьком корпусе разумная частота отрицательной дискретизации — от 5 до 20 примеров. Для большего корпуса и словаря можно снизить частоту отрицательной дискретизации до, скажем, 2–5 примеров, согласно данным Миколова и его команды.

6.2.3. Использование модуля gensim.word2vec

Если предыдущий раздел показался вам слишком сложным — не волнуйтесь. Многие компании предоставляют свои предобученные модели векторов слов, а популярные библиотеки NLP для различных языков программирования дают возможность эффективно применять эти модели. В следующем разделе мы расскажем, как воспользоваться возможностями магии векторов слов.

Если вы уже установили пакет `nlpia`[1], то можете скачать предобученную модель Word2vec с помощью следующей команды:

```
>>> from nlpia.data.loaders import get_data
>>> word_vectors = get_data('word2vec')
```

Если эта команда не работает или вы хотели бы создать собственную, то можете поискать в Google модели Word2vec, предобученные на документах из Google News[2]. После того как вы найдете и скачаете модель в специальном двоичном формате

---

[1] См. инструкции по установке в файле README по адресу http://github.com/totalgood/nlpia.

[2] Исходная модель, обученная Миколовым, размещена на Google Drive по адресу https://bit.ly/GoogleNews-vectors-negative300.

Google и расположите ее на локальной машине, можете загрузить ее с помощью пакета gensim следующим образом:

```
>>> from gensim.models.keyedvectors import KeyedVectors
>>> word_vectors = KeyedVectors.load_word2vec_format(\
...     '/path/to/GoogleNews-vectors-negative300.bin.gz', binary=True)
```

Работа с векторами слов требует большого объема оперативной памяти. Если оперативная память вашей машины невелика или вы не хотели бы ждать загрузки модели векторов слов несколько минут, можете уменьшить количество загружаемых в память слов с помощью именованного аргумента `limit`. В следующем примере загружается 200 000 наиболее распространенных слов из корпуса `Google News`:

```
>>> from gensim.models.keyedvectors import KeyedVectors
>>> word_vectors = KeyedVectors.load_word2vec_format(\
...     '/path/to/GoogleNews-vectors-negative300.bin.gz',
...         binary=True, limit=200000)
```

Учтите, что показатели работы конвейера NLP при модели векторов слов с ограниченным словарем оказываются хуже, если документы содержат слова, для которых не были загружены векторы. Следовательно, размер модели векторов слов нужно ограничивать только во время разработки. Если вы хотите получить те же результаты, что и мы, для остальных примеров из этой главы следует использовать полную модель Word2vec.

Метод `gensim.KeyedVectors.most_similar()` является эффективным способом поиска ближайших соседей для любого заданного вектора слов. В поименованном аргументе `positive` задается список складываемых векторов, подобно примеру с футбольной командой из начала главы. Аналогично можно воспользоваться аргументом `negative` для вычитания и исключения посторонних термов. Аргумент `topn` определяет количество возвращаемых связанных термов.

В отличие от обычного тезауруса синонимия (подобие) Word2vec — непрерывный показатель, расстояние. Дело в том, что Word2vec представляет собой непрерывную модель векторного пространства. Высокая размерность Word2vec и непрерывные значения во всех измерениях позволяют захватывать весь спектр значений любого слова. Поэтому для Word2vec несложны задачи на аналогию и даже зевгмы, необычные сочетания нескольких смыслов в одном слове[1]:

```
>>> word_vectors.most_similar(positive=['cooking', 'potatoes'], topn=5)
[('cook', 0.6973530650138855),
 ('oven_roasting', 0.6754530668258667),
 ('Slow_cooker', 0.6742032170295715),
 ('sweet_potatoes', 0.6600279808044434),
 ('stir_fry_vegetables', 0.6548759341239929)]
>>> word_vectors.most_similar(positive=['germany', 'france'], topn=1)
[('europe', 0.7222039699554443)]
```

[1] Книга *Hoffstadter D., Sander E.* Surfaces and Essences: Analogy as the Fuel and Fire of Thinking рассказывает, почему возможность машин работать с аналогиями и зевгмами настолько важна.

Модели векторов слов дают возможность также определять и несвязанные термы. В библиотеке gensim есть метод `doesnt_match`:

```
>>> word_vectors.doesnt_match("potatoes milk cake computer".split())
'computer'
```

Он возвращает от остальных термов списка наиболее удаленный терм, благодаря чему можно определить самый посторонний терм из списка.

Для выполнения вычислений (как в знаменитом примере *king* + *woman* – *man* = = *queen*, который и привлек изначально внимание Миколова) можно добавить в вызов метода `most_similar` аргумент `negative`:

```
>>> word_vectors.most_similar(positive=['king', 'woman'],
...     negative=['man'], topn=2)
[('queen', 0.7118192315101624), ('monarch', 0.6189674139022827)]
```

Библиотека gensim позволяет вычислять подобие между двумя термами. Для сравнения двух термов и определения их косинусного коэффициента воспользуйтесь методом `.similarity()`:

```
>>> word_vectors.similarity('princess', 'queen')
0.70705315983704509
```

Если вы хотите разработать собственные функции и работать с исходными векторами слов, то получить к ним доступ можно с помощью синтаксиса с квадратными скобками (`[]`) или метода `get()` экземпляра `KeyedVector`. Загруженный объект модели можно рассматривать как словарь, в котором интересующее вас слово представляет собой ключ словаря. Каждое значение с плавающей точкой в возвращаемом массиве соответствует одному из измерений вектора. В случае модели слов Google форма массивов NumPy будет 1 × 300:

```
>>> word_vectors['phone']
array([-0.01446533, -0.12792969, -0.11572266, -0.22167969, -0.07373047,
       -0.05981445, -0.10009766, -0.06884766,  0.14941406,  0.10107422,
       -0.03076172, -0.03271484, -0.03125 ,   -0.10791016,  0.12158203,
        0.16015625,  0.19335938,  0.0065918 , -0.15429688,  0.03710938,
        ...
```

Если вам интересно, что *означают* эти все цифры, то придется изучить какие-нибудь синонимы и выяснить, какие из 300 чисел в массиве у них совпадают. Или же найти линейную комбинацию этих чисел, составляющую измерения для таких вещей, как *placeness* и *femaleness*, подобно тому как мы делали в начале главы.

6.2.4. Как сгенерировать свои собственные представления векторов слов

В некоторых случаях нужно создать собственные предметно-ориентированные модели векторов слов. Это может повысить точность модели, если конвейер NLP обрабатывает документы с видами словоупотребления, отсутствовавшими в Google

News до 2006 года, когда Миколов обучал свою эталонную модель Word2vec. Учтите, для этого понадобится *много* документов, как потребовалось и Google с Миколовым. Но если интересующие вас слова особенно редко встречаются в Google News или в ваших текстах они используются специфически на ограниченной предметной области, например в случае текстов на медицинскую тематику либо расшифровки стенограмм, то предметно-ориентированная модель слов может повысить точность. В следующем разделе мы покажем, как обучить собственную модель Word2vec.

Для обучения предметно-ориентированной модели Word2vec снова обратимся к gensim, но, прежде чем начать обучать эту модель, нужно обработать корпус с помощью утилит, с которыми мы познакомились в главе 2.

Этапы предварительной обработки

Прежде всего необходимо разбить документы на предложения, а последние — на токены. Модель `gensimword2vec` ожидает на входе список предложений, в котором каждое разбито на токены. Это предотвращает обучение векторов слов на посторонних вхождениях слов в соседних предложениях. Структура тренировочных входных данных должна быть примерно следующей:

```
>>> token_list
[
  ['to', 'provide', 'early', 'intervention/early', 'childhood', 'special',
   'education', 'services', 'to', 'eligible', 'children', 'and', 'their',
   'families'],
  ['essential', 'job', 'functions'],
  ['participate', 'as', 'a', 'transdisciplinary', 'team', 'member', 'to',
   'complete', 'educational', 'assessments', 'for']
  ...
]
```

Для сегментирования предложений и последующего преобразования их в токены подходят различные стратегии из главы 2. Для некоторых приложений с помощью такого сегментатора предложений, как детектор Морзе, можно достичь лучшей точности, по сравнению с имеющимся в NLTK и gensim сегментатором[1]. После преобразования документов в списки списков токенов (по одному для каждого предложения) все готово для обучения Word2vec.

---

[1] Детектор Морзе, созданный Кайлом Горменом и OHSU, который можно найти на pypi и в https://github.com/cslu-nlp/DetectorMorse, представляет собой сегментатор предложений, эффективность работы которого соответствует последним достижениям науки (98 %), предобученный на предложениях их многолетней подборки текстов из Wall Street Journal. Если язык вашего корпуса похож на язык WSJ, то детектор Морзе, вероятно, обеспечит наивысшую возможную точность. Вы можете повторно обучить детектор Морзе на собственном наборе данных, если у вас есть большой набор предложений из нужной предметной области.

Обучение предметно-ориентированной модели Word2vec

Начнем с загрузки модуля Word2vec:

```
>>> from gensim.models.word2vec import Word2Vec
```

Обучение требует задания нескольких параметров, показанных в листинге 6.2.

Листинг 6.2. Параметры, управляющие обучением модели Word2vec

```
>>> num_features = 300
>>> min_word_count = 3
>>> num_workers = 2
>>> window_size = 6
>>> subsampling = 1e-3
```

Количество элементов (измерений) векторов слов

Минимальное количество вхождений слова, учитываемое в модели Word2vec. Если ваш корпус мал, уменьшите этот параметр. Если же обучение производится на большом корпусе, увеличьте

Количество используемых для обучения ядер CPU. Для динамического выбора числа ядер выполните импорт: num_workers = multiprocessing.cpu_count()

Частота прореживания для часто встречающихся термов

Размер окна контекста

Теперь мы готовы приступить к обучению с помощью листинга 6.3.

Листинг 6.3. Создание экземпляра модели Word2vec

```
>>> model = Word2Vec(
...     token_list,
...     workers=num_workers,
...     size=num_features,
...     min_count=min_word_count,
...     window=window_size,
...     sample=subsampling)
```

В зависимости от размера корпуса и производительности CPU обучение может занять значительное время. Для маленьких корпусов обучение завершится за несколько минут. Для всеобъемлющей модели слов корпус содержит миллионы предложений, поскольку необходимо по нескольку примеров всех различных способов использования слов в корпусе. Поэтому не удивляйтесь, что обработка большого корпуса, вроде корпуса «Википедии», займет гораздо больше времени и потребует намного больше оперативной памяти.

Модели Word2vec требуют значительных объемов оперативной памяти. Помните, что нас интересует только матрица весов для скрытого слоя. После обучения модели слов можно уменьшить объем используемой оперативной памяти примерно наполовину, «заморозив» модель и отбросив ненужную информацию. Следующая команда исключает ненужные выходные веса из нейронной сети:

```
>>> model.init_sims(replace=True)
```

Метод `init_sims` замораживает модель, сохраняя веса скрытого слоя и отбрасывая выходные веса, предсказывающие совместную встречаемость слов. Выходные веса не входят в векторы, применяемые для большинства приложений Word2vec.

Но дальнейшее обучение модели после отбрасывания весов выходного слоя невозможно.

С помощью нижеприведенной команды можно сохранить обученную модель для последующего использования:

```
>>> model_name = "my_domain_specific_word2vec_model"
>>> model.save(model_name)
```

Для проверки только что обученной модели можно воспользоваться ею с помощью той же команды, что и в предыдущем разделе (листинг 6.4).

Листинг 6.4. Загрузка сохраненной модели Word2vec

```
>>> from gensim.models.word2vec import Word2Vec
>>> model_name = "my_domain_specific_word2vec_model"
>>> model = Word2Vec.load(model_name)
>>> model.most_similar('radiology')
```

6.2.5. Word2vec по сравнению с GloVe (моделью глобальных векторов)

Word2vec был крупным достижением для своего времени, но в его основе лежала нейронная сеть, которой требовалось обучение путем обратного распространения ошибки. Метод обратного распространения ошибки обычно менее эффективен, чем непосредственная оптимизация функции стоимости с помощью градиентного спуска. Исследователи NLP из Стэнфорда[1] под руководством Джеффри Пеннингтона решили выяснить, почему Word2vec так хорошо работает, и найти оптимизируемую функцию стоимости. Они начали с подсчета совместных вхождений слов и занесения их в квадратную матрицу. Оказалось, что можно вычислить сингулярное разложение[2] этой матрицы совместной встречаемости, разбив ее на те же две матрицы весов, что генерирует Word2vec[3]. Главное — нормализировать матрицу совместной встречаемости точно так же. Но в некоторых случаях модель Word2vec не сходится к тому глобальному минимуму, который получался у стэнфордских исследователей с помощью SVD. Именно от непосредственной оптимизации глобальных векторов (global vectors) совместной встречаемости слов (совместных вхождений в рамках всего корпуса) и получил свое название метод GloVe.

Метод GloVe может формировать матрицы, эквивалентные входным и выходным матрицам весов Word2vec, в результате чего получается языковая модель с той же точностью, что Word2vec, но за намного меньшее время. GloVe ускоряет процесс за счет более эффективного использования данных. GloVe может сойтись

---

1 Стэнфордский проект GloVe (https://nlp.stanford.edu/projects/glove/).

2 Больше подробностей о SVD вы можете найти в главе 5 и приложении B.

3 *Pennington J., Socher R., Manning C. D.* GloVe: Global Vectors for Word Representation по адресу https://nlp.stanford.edu/pubs/glove.pdf.

даже при обучении на меньшем корпусе[1]. Кроме того, алгоритмы SVD совершенствовались десятилетиями, так что у GloVe есть определенный гандикап в смысле отлаженности и оптимизации алгоритма. Word2vec основывается на обновлении весов для вложений слов с помощью обратного распространения ошибки. Такой метод ошибки нейронных сетей менее эффективен, чем более зрелые алгоритмы оптимизации, вроде используемых в SVD для GloVe.

Хотя Word2vec первым популяризировал понятие семантических умозаключений с помощью векторов слов, для обучения новых моделей векторов слов вы будете в основном использовать GloVe. Вероятность найти глобальный экстремум для представлений векторов при применении GloVe выше, а значит, и результаты точнее.

Преимущества GloVe:

- большая скорость обучения;
- более экономное расходование ресурсов оперативной памяти/процессора (а значит, и возможность обработки больших документов);
- более эффективное использование данных (удобно при маленьких корпусах);
- более высокая точность при таком же объеме обучения.

6.2.6. FastText

Исследователи из Facebook усовершенствовали идею Word2vec[2], добавив в обучение модели новый трюк. Алгоритм, который они назвали fastText, предсказывает окружающие *n-символьные* граммы, а не просто окружающие слова, как Word2vec. Например, из *whisper* («шепот») получатся следующие би- и триграммы: *wh*, *whi*, *hi*, *his*, *is*, *isp*, *sp*, *spe*, *pe*, *per*, *er*.

FastText производит усвоение представлений для каждой *n*-символьной граммы, включая просто слова, с орфографическими ошибками, части слов и даже отдельные символы. Преимущество такого подхода в том, что он демонстрирует намного лучшие результаты для редко встречающихся слов, чем обычный подход Word2vec.

В качестве части выпуска fastText Facebook опубликовал предобученные модели fastText для 294 языков. На странице GitHub для этого исследования[3] можно найти модели для языков от *абхазского* до *зулусского*. Коллекция моделей даже включает такие редкие языки, как *затерландский фризский*, на котором говорит лишь горстка немцев. Предоставляемые Facebook предобученные модели fastText обучались лишь на имеющихся корпусах «Википедии». Следовательно, для различных языков объем словаря и точность этих моделей будут различны.

---

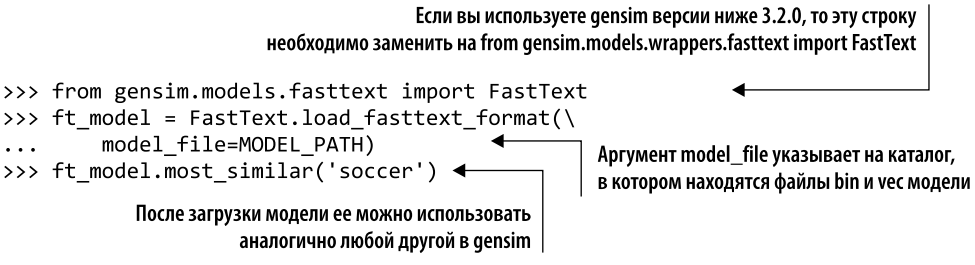
[1] Сравнение производительности Word2vec и GloVe (https://rare-technologies.com/making-sense-of-Word2vec/#glove_vs_word2vec).

[2] *Bojanowski et al.* Enriching Word Vectors with Subword Information (https://arxiv.org/pdf/1607.04606.pdf).

[3] См. веб-страницу fastText/pretrained-vectors.md at master по адресу https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md.

Использование предобученных моделей fastText

FastText можно использовать аналогично модели Word2vec компании Google. Зайдите в репозиторий `fastText` и скачайте бинарный и текстовый файлы модели для интересующего вас языка. По окончании скачивания распакуйте бинарный файл языка[1]. Затем можно загрузить его в gensim с помощью следующего кода:



Значительная часть функциональности API fastText gensim совпадает с реализациями Word2vec. Все описанные выше в данной главе методы применимы и к моделям fastText.

6.2.7. Word2vec по сравнению с LSA

Интересно сравнить векторы слов Word2vec и GloVe с векторами «тема — слово» LSA из главы 4. Хотя там мы особо не говорили про них, но LSA генерирует и их тоже. Векторы «тема — документ» LSA представляют собой сумму векторов «тема — слово» для всех слов в документе. Для получения вектора слов для всего документа, аналогичного векторам «тема — документ», необходимо просуммировать все векторы слов Word2vec в каждом из документов. Это очень напоминает механизм работы Doc2vec. Мы продемонстрируем вам его чуть позже в этой главе.

При размере матрицы LSA векторов тем *Nслов * Nтем* векторы слов LSA представляют собой ее строки. Эти векторы-строки захватывают смысл слов в последовательности примерно 200–300 вещественных значений аналогично Word2vec. И векторы «тема — слово» LSA удобны для поиска как связанных, так и несвязанных термов. Как вы узнали из обсуждения GloVe, векторы Word2vec можно создать с помощью того же алгоритма SVD, что и для LSA. Но Word2vec извлекает больше пользы из такого же числа слов в документах, создавая скользящие окна, переходящие с перекрытием от одного документа к следующему. Так можно переиспользовать одни и те же слова пять раз, прежде чем перейти двигаться дальше.

А как насчет инкрементного (динамического) обучения? Алгоритмы как LSA, так и Word2vec позволяют добавлять в корпус новые документы и корректировать существующие векторы слов для учета совместных вхождений в новых документах. Но обновить можно только уже существующие группы в словаре. Добавление совершенно новых слов привело бы к изменению размера словаря и унитарных векторов. Поэтому захват нового слова в модели требует повторного обучения.

[1] Размер файла en.wiki.zip — около 9,6 Гбайт.

Модель LSA обучается быстрее, чем Word2vec. И она лучше различает и кластеризует длинные документы.

«Убойным приложением» для метода Word2vec стали семантические умозаключения, популярности которых он способствовал. Векторы «тема — слово» LSA также способны на это, но точность в этом случае обычно ниже. Для достижения точности и «вау»-эффекта умозаключений Word2vec необходимо разбить документы на предложения и использовать для обучения модели LSA только короткие фразы. С помощью Word2vec можно найти ответ на вопросы вроде *Harry Potter + University = Hogwarts*[1].

Преимущества LSA:

- более быстрое обучение;
- лучшее различение крупных документов.

Преимущества Word2vec и GloVe:

- более эффективное использование больших корпусов;
- более точные умозаключения на основе слов, например ответы на вопросы на аналогию.

6.2.8. Визуализация связей между словами

Возможности семантических связей между словами очень велики, а их визуализации могут вести к интересным открытиям. В этом разделе мы пошагово продемонстрируем двумерную визуализацию векторов слов.

ПРИМЕЧАНИЕ

> Рекомендуем использовать для быстрой визуализации вашей модели слов имеющуюся в TensorBoard от Google функциональность визуализации вложений слов. Больше подробностей вы можете найти в подразделе 13.6.1 «Визуализация вложений слов» в главе 13.

Для начала загрузим все векторы слов из модели Word2vec корпуса Google News. Этот корпус включает множество упоминаний Портленда и Орегона, а также других названий городов и штатов. Для упрощения мы воспользуемся пакетом `nlpia`, чтобы можно было быстро начать эксперименты с векторами Word2vec (листинг 6.5).

Листинг 6.5. Загрузка предобученной модели с помощью nlpia

```
>>> import os
>>> from nlpia.loaders import get_data
>>> from gensim.models.word2vec import KeyedVectors
>>> wv = get_data('word2vec') ◄──┐
>>> len(wv.vocab)                │
3000000                          │
                                 │
Скачивает предобученные векторы слов Google News
в файл nlpia/src/nlpia/bigdata/GoogleNews-vectors-negative300.bin.gz
```

[1] В качестве прекрасного примера предметно-ориентированных моделей Word2vec можно привести модели относительно «Гарри Поттера», «Властелина колец» и т. п. (https://github.com/nchah/word2vec4everything#harry-potter).

ВНИМАНИЕ!

> Модель Word2vec Google News огромна: три миллиона слов с 300 измерениями вектора для каждого. Полная модель векторов слов требует 3 Гбайт оперативной памяти. Если доступный вам объем оперативной памяти ограничен или вы хотите быстро загрузить несколько наиболее часто встречающихся термов из модели слов, загляните в главу 13.

Объект `KeyedVectors` в gensim теперь содержит таблицу из трех миллионов векторов Word2vec. Мы загрузили эти векторы из файла, созданного Google для хранения модели Word2vec, обученной на большом корпусе, и основанного на статьях из Google News. Во всех этих новостных статьях будет множество слов для штатов и городов. Листинг 6.6 демонстрирует лишь малую часть этих слов в словаре, начиная с миллионного слова.

Листинг 6.6. Изучаем частотности слов в словаре Word2vec

```
>>> import pandas as pd
>>> vocab = pd.Series(wv.vocab)
>>> vocab.iloc[1000000:100006]
Illington_Fund             Vocab(count:447860, index:2552140)
Illingworth                 Vocab(count:2905166, index:94834)
Illingworth_Halifax       Vocab(count:1984281, index:1015719)
Illini                      Vocab(count:2984391, index:15609)
IlliniBoard.com           Vocab(count:1481047, index:1518953)
Illini_Bluffs              Vocab(count:2636947, index:363053)
```

Обратите внимание: составные слова и распространенные *n*-граммы соединяются с помощью символа подчеркивания (_). Заметьте также, что значение в паре «ключ/значение» представляет собой объект `gensimVocab`, содержащий не только индекс слова (позволяющий найти вектор Word2vec), но и число его вхождений в корпус Google News.

С целью поиска 300-мерного вектора для конкретного слова можно воспользоваться квадратными скобками (применительно к этому объекту `KeyedVectors`), чтобы выполнить метод `__getitem__()` для любого слова или n-граммы[1]:

```
>>> wv['Illini']
array([ 0.15625   ,  0.18652344,  0.33203125,  0.55859375,  0.03637695,
       -0.09375   , -0.05029297,  0.16796875, -0.0625    ,  0.09912109,
       -0.0291748 ,  0.39257812,  0.05395508,  0.35351562, -0.02270508,
       ...
```

Причина, по которой мы выбрали именно миллионное слово (в лексикографическом алфавитном порядке), — то, что первые несколько тысяч «слов» представляют собой пунктуационные последовательности вроде # и других часто встречающихся в корпусе Google News символов. Нам просто повезло, что в этом списке попалось

[1] __getitem__() в Python — так называемый волшебный метод. Если переопределить его в классе, можно реализовать собственную логику оператора индексирования [], то есть запись x[i] фактически означает x.__getitem__(i). — *Примеч. пер.*

слово `Illini`[1]. Определим в листинге 6.7, насколько близок вектор для `Illini` к вектору для `Illinois`.

Листинг 6.7. Расстояние между Illini и Illinois

```
>>> import numpy as np
>>> np.linalg.norm(wv['Illinois'] - wv['Illini'])
3.3653798
>>> cos_similarity = np.dot(wv['Illinois'], wv['Illini']) / (
...     np.linalg.norm(wv['Illinois']) *\
...     np.linalg.norm(wv['Illini']))
>>> cos_similarity
0.5501352
>>> 1 - cos_similarity
0.4498648
```

Евклидово расстояние

Косинусный коэффициент равен нормализованному скалярному произведению

Косинусное расстояние

Эти расстояния означают, что `Illini` и `Illinois` близки друг к другу по смыслу, но не слишком.

Теперь найдем все векторы Word2vec для городов США, чтобы на основе их расстояний нарисовать двумерный график их смыслов. Как же найти все города и штаты в конкретном словаре Word2vec в определенном объекте `KeyedVectors`? Можно воспользоваться косинусным расстоянием, как в листинге 6.7, и найти все векторы, близкие к словам `state` (штат) и `city` (город). Вместо просмотра всех 3 миллионов слов и векторов слов загрузим другой набор данных, содержащий список городов и штатов (областей) со всего мира, как показано в листинге 6.8.

Листинг 6.8. Данные о городах США

```
>>> from nlpia.data.loaders import get_data
>>> cities = get_data('cities')
>>> cities.head(1).T
geonameid                       3039154
name                          El Tarter
asciiname                     El Tarter
alternatenames    Ehl Tarter,Эл Тартер
latitude                       42.5795
longitude                      1.65362
feature_class                        P
feature_code                       PPL
country_code                        AD
cc2                                NaN
admin1_code                         02
admin2_code                        NaN
admin3_code                        NaN
admin4_code                        NaN
population                        1052
elevation                          NaN
dem                               1721
```

[1] Illini означает группу людей, обычно футболистов и болельщиков, а не географический регион Иллинойс (в котором живет большинство болельщиков команды Fighting Illini).

```
timezone                Europe/Andorra
modification_date           2012-11-03
```

Этот набор данных от Geocities содержит массу информации, включая долготу, широту и население. Его можно использовать для различных интересных визуализаций или сравнения географического расстояния с расстоянием Word2vec. Пока что мы просто попытаемся нарисовать карту расстояний Word2vec на 2D-плоскости и посмотрим, как она выглядит. Ограничимся США, как показано в листинге 6.9.

Листинг 6.9. Данные о штатах США

```
>>> us = cities[(cities.country_code == 'US') &\
...     (cities.admin1_code.notnull())].copy()
>>> states = pd.read_csv(\
...     'http://www.fonz.net/blog/wp-content/uploads/2008/04/states.csv')
>>> states = dict(zip(states.Abbreviation, states.State))
>>> us['city'] = us.name.copy()
>>> us['st'] = us.admin1_code.copy()
>>> us['state'] = us.st.map(states)
>>> us[us.columns[-3:]].head()
                    city       st         state
geonameid
4046255      Bay Minette       AL       Alabama
4046274             Edna       TX         Texas
4046319   Bayou La Batre       AL       Alabama
4046332        Henderson       TX         Texas
4046430          Natalia       TX         Texas
```

Теперь, помимо сокращенных, у нас есть и полные названия штатов для всех городов. Посмотрим, какие из них есть в нашем словаре Word2vec:

```
>>> vocab = pd.np.concatenate([us.city, us.st, us.state])
>>> vocab = np.array([word for word in vocab if word in wv.wv])
>>> vocab[:5]
array(['Edna', 'Henderson', 'Natalia', 'Yorktown', 'Brighton'])
```

Даже в США можно найти множество крупных городов с одинаковыми названиями, например Портленд (штат Орегон) и Портленд (штат Мэн). Включим в наши векторы городов информацию о штате, в котором он расположен. Для объединения смыслов слов в Word2vec необходимо сложить их векторы. В этом и состоит волшебство векторных умозаключений. Один из способов — сложить векторы для штатов с векторами для городов и поместить их в один большой `DataFrame`. Мы используем полное название штата или только сокращенное (в зависимости от того, какое есть в словаре Word2vec), как показано в листинге 6.10.

Листинг 6.10. Дополнение векторов слов для городов векторами слов для штатов США

```
>>> city_plus_state = []
>>> for c, state, st in zip(us.city, us.state, us.st):
...     if c not in vocab:
...         continue
...     row = []
```

```
...     if state in vocab:
...         row.extend(wv[c] + wv[state])
...     else:
...         row.extend(wv[c] + wv[st])
...     city_plus_state.append(row)
>>> us_300D = pd.DataFrame(city_plus_state)
```

Основываясь на корпусе, взаимосвязи слов могут отражать различные атрибуты, например географическую близость или культурное/экономическое сходство. Но взаимосвязи сильно подвластны тренировочному корпусу и отражают его.

Векторы слов содержат систематическую ошибку!

Векторы слов усваивают взаимосвязи слов по тренировочному корпусу. Если корпус посвящен финансам, то ваш «банковский» вектор слов будет также в основном относиться к бизнесу, у которого есть вклады в банке. Если же корпус посвящен геологии, то вектор слов будет обучаться на ассоциациях с речками и ручейками. Если же корпус посвящен матриархальному обществу, где женщины работают в банке, а мужчины стирают одежду в реке, то векторы слов будут учитывать подобное гендерное смещение.

Следующий пример демонстрирует гендерное смещение модели слов, обучавшейся на статьях Google News. Если подсчитать расстояние между словами `man` и `nurse` и сравнить его с расстоянием между `woman` и `nurse`, то систематическая ошибка будет очевидна:

```
>>> word_model.distance('man', 'nurse')
0.7453
>>> word_model.distance('woman', 'nurse')
0.5586
```

Обнаружение и корректировка подобных систематических ошибок — непростая задача для любого специалиста по NLP, обучающего свои модели на документах, созданных в этом полном предубеждений мире.

У применяемых в этом корпусе новостных статей есть общая составляющая — смысловое сходство городов. Семантически сходные места в статьях кажутся взаимозаменяемыми, а потому модель слов усваивает, что они похожи. При обучении на другом корпусе взаимосвязи слов могли бы быть другими. В данном корпусе новостей схожие по размеру и культуре города группируются, даже если находятся далеко географически, как Сан-Диего и Сан-Хосе или как курорты вроде Гонолулу и Рино.

К счастью, чтобы сложить векторы для городов с векторами для полных и сокращенных названий штатов, можно использовать обычную алгебру. Как вы видели в главе 4, для понижения размерности векторов (с 300 до понятного человеку двумерного представления) можно использовать такие инструменты, как метод главных компонент. PCA позволяет увидеть проекцию («тень») этих 300-мерных векторов на 2D-графике. Алгоритм PCA гарантирует, что эта проекция будет наилучшим возможным представлением данных с как можно большим расстоянием между векторами. PCA подобен хорошему фотографу, который смотрит на свой объект со всех сторон, прежде чем сделать идеальную фотографию. Не нужно даже нормализовать длины векторов перед суммированием векторов «город + штат + сокращение», поскольку PCA берет это на себя.

Мы сохранили эти дополненные векторы слов для городов в пакете nlpia, так что вы можете загрузить их и применять в своем приложении. В листинге 6.11 мы воспользуемся PCA для проекции их на двумерный график.

Листинг 6.11. Пузырьковая диаграмма городов США

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> us_300D = get_data('cities_us_wordvectors')
>>> us_2D = pca.fit_transform(us_300D.iloc[:, :300])
```

Полученные из PCA для визуализации 2D-векторы. Сохраняем исходные 300-мерные векторы Word2vec для будущих векторных умозаключений

Последний столбец этого DataFrame содержит название города, хранящееся также в индексе DataFrame

На рис. 6.8 показана двумерная проекция всех этих 300-мерных векторов для городов США.

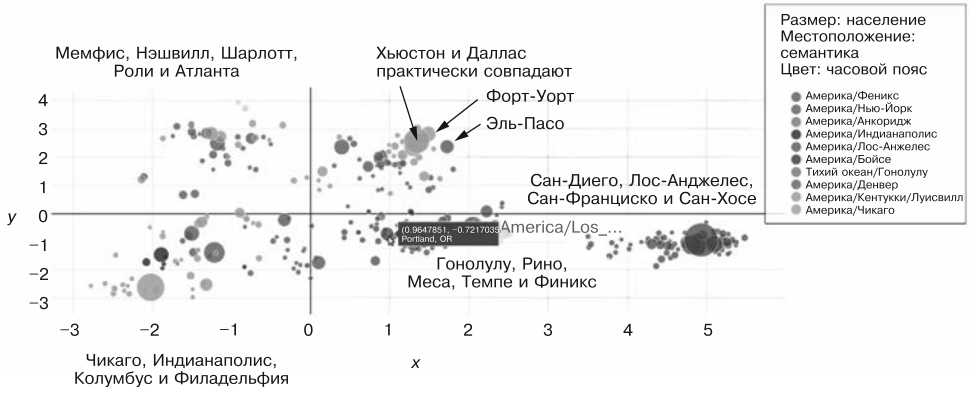


Рис. 6.8. Спроецированные с помощью PCA на 2D-график 300-мерные векторы Word2vec Google News

ПРИМЕЧАНИЕ

Небольшое (близкое к нулю) семантическое расстояние означает сильное сходство слов. Семантическое (смысловое) расстояние определяется словами, встречающимися поблизости в используемых для обучения документах. Векторы Word2vec *близки* друг к другу в пространстве векторов слов, если они часто используются в схожем контексте (с похожими словами поблизости). Например, Сан-Франциско *близок* к Калифорнии, поскольку они часто встречаются рядом в предложениях, а распределения применяемых рядом с ними слов схожи. Большое расстояние между двумя термами означает малую вероятность общности их контекста и смысла (они семантически не схожи), например, в случае автомобилей и арахиса.

В листинге 6.12 показано, что делать, если вы хотите изучить показанную на рис. 6.8 карту городов поближе или построить график каких-либо своих векторов. Мы создали адаптер для офлайн-API построения графиков, который поможет при

работе с объектами `DataFrame` с денормализованными данными. Адаптер `Plotly` ожидает на входе объект `DataFrame`, содержащий по строке для каждого примера и по столбцу для каждого признака, на основе которых нужно построить график. Эти признаки могут быть категориальными (например, часовыми поясами) или непрерывными веществennозначными признаками (скажем, населением города). В результате получаются интерактивные графики, удобные для исследования многих видов данных машинного обучения, особенно векторных представлений таких сложных вещей, как слова и документы.

Листинг 6.12. Пузырьковая диаграмма векторов слов для городов США[1]

```
>>> import seaborn
>>> from matplotlib import pyplot as plt
>>> from nlpia.plots import offline_plotly_scatter_bubble
>>> df = get_data('cities_us_wordvectors_pca2_meta')
>>> html = offline_plotly_scatter_bubble(
...     df.sort_values('population', ascending=False)[:350].copy()\
...         .sort_values('population'),
...     filename='plotly_scatter_bubble.html',
...     x='x', y='y',
...     size_col='population', text_col='name', category_col='timezone',
...     xscale=None, yscale=None, # 'log' or None
...     layout={}, marker={'sizeref': 3000})
{'sizemode': 'area', 'sizeref': 3000}
```

Для получения 2D-представлений наших 300-мерных векторов слов необходимо воспользоваться методом понижения размерности. Мы применяли PCA. Чтобы уменьшить объем информации, теряемой при сжатии от 300-мерного до двумерного пространства, помогает также сужение диапазона содержащейся во входных векторах информации. Поэтому мы ограничили векторы слов только теми, что связаны с городами. Это напоминает ограничение предметной области или тематики корпуса при вычислении векторов TF-IDF или BOW.

Для разнообразия смеси векторов, содержащих больше информации, понадобится нелинейный алгоритм вложения, например t-SNE. Мы поговорим о t-SNE и других методах нейронных сетей в дальнейших главах. T-SNE будет для вас понятнее, когда вы разберетесь здесь с алгоритмами вложений векторов слов.

6.2.9. Искусственные слова

Вложения слов вроде Word2vec могут пригодиться не только для английских слов, но и для любой последовательности символов, в которой очередность и близость символов отражают какой-то смысл. Если у символов есть семантика, значит, вложения могут оказаться полезны. Они работают и для других языков, а не только английского.

---

[1] Этот код работает без изменений только с версиями plotly не выше 3.6.1, в основном из-за функции _plot_html, которая несовместима по параметрам с версиями выше 3.6.1, а начиная с 3.8.0 вообще отсутствует. Возможно, авторы книги в будущем обновят версию nlpia для поддержки новых версий plotly. — *Примеч. пер.*

Вложения слов можно использовать и для пиктографических языков, например для классического китайского, японского (кандзи) или даже загадочных иероглифов в египетских гробницах. Вложения и векторные умозаключения работают для языков, нацелены на сокрытие смысла слов. Векторные умозаключения можно производить на основе большого набора «секретных» сообщений «поросячьей латыни» или любого другого языка, изобретенного как детьми, так и римским императором[1]. *Шифр Цезаря* (https://ru.wikipedia.org/wiki/Шифр_Цезаря), например ROT13, и *шифр подстановки* (https://ru.wikipedia.org/wiki/Шифр_подстановки) поддаются векторным умозаключениям Word2vec. Не требуется даже дешифровочное кольцо (вроде показанных на рис. 6.9). Необходим только большой набор сообщений (*n*-грамм) для обработки с помощью Word2vec для поиска совместных вхождений слов или символов.

Рис. 6.9. Дешифровочные кольца (слева: шифровальный диск для подстановочного шифра[2]; в центре: криптографическое обручальное кольцо[3]; справа: дешифровочное кольцо Капитана Полночь[4])

Word2vec подходит даже для сбора информации и выяснения связей из искусственных слов или идентификаторов, например номеров университетских курсов (CS-101), моделей (Koala E7270 или Galaga Pro), телефонов, почтовых индексов и даже серийных номеров[5]. Для получения максимума полезной информации о связях между подобными идентификаторами понадобится множество содержащих их предложений, а то, что в структуре таких числовых идентификаторов позиция символа часто несет смысл, может помочь токенизировать эти идентификаторы на минимально возможные пакеты (слова или слоги в естественных языках).

---

1 Строго говоря, императором в современном смысле Гай Юлий Цезарь не был, список императоров Древнего Рима традиционно начинают с Октавиана Августа. — *Примеч. пер.*

2 Фото Губерта Берберлиха (Hubert Berberich) (https://commons.wikimedia.org/wiki/File:CipherDisk2000.jpg) помечено как всеобщее достояние, больше подробностей можно найти в «Викискладе» по адресу https://commons.wikimedia.org/wiki/Template:PD-self.

3 Фото Кори Доктороу (Cory Doctorow) (https://www.flickr.com/photos/doctorow/2817314740/in/photostream/) по лицензии https://creativecommons.org/licenses/by-sa/2.0/legalcode.

4 Автор — Sobebunny (https://commons.wikimedia.org/wiki/File:Captain-midnight-decoder.jpg), https://creativecommons.org/licenses/by-sa/3.0/legalcode.

5 См. веб-страницу A non-NLP application of Word2Vec — Towards Data Science по адресу https://medium.com/towards-data-science/a-non-nlp-application-of-word2vec-c637e35d3668.

6.2.10. Определение сходства документов с помощью Doc2vec

Идею Word2vec можно распространить на предложения, абзацы или даже целые документы. Идею предсказания следующего слова по предыдущим можно развить на обучение вектора абзаца или документа (рис. 6.10)[1]. В этом случае предсказание учитывает не только предыдущие слова, но и вектор, отражающий абзац или целый документ. Этот вектор можно рассматривать как дополнительный входной сигнал для предсказания. Со временем алгоритм усваивает представление документа/абзаца из тренировочного набора данных.

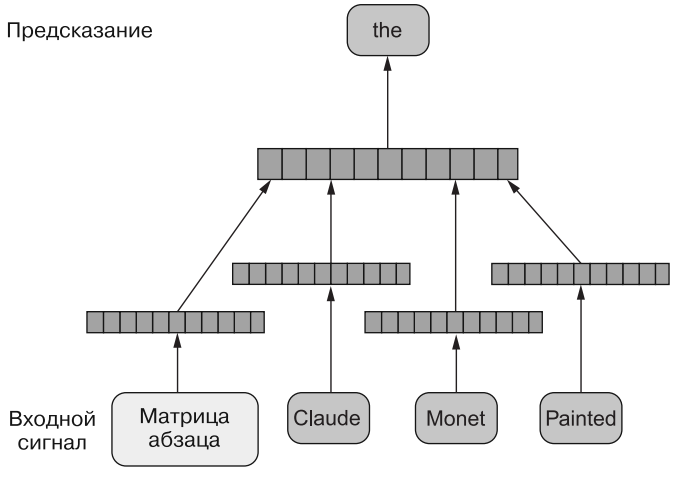


Рис. 6.10. При обучении Doc2vec используется дополнительный вектор документа в качестве входных данных

Каким образом генерируются векторы документов для неизвестных алгоритму документов после этапа обучения? Во время *этапа вывода* (inference stage) алгоритм добавляет дополнительные векторы документов в их матрицу и вычисляет значение добавленного вектора на основе «замороженной» матрицы векторов слов и соответствующих весов. Теперь можно создать семантическое представление всего документа путем вывода вектора документа.

За счет расширения идеи Word2vec с помощью дополнительного вектора документа или абзаца, применяемого для предсказания слов, можно использовать полученный в результате обучения вектор документа для различных целей, например для поиска в корпусе схожих документов.

---

[1] См. веб-страницу Distributed Representations of Sentences and Documents («Распределенные представления предложений и документов») по адресу https://arxiv.org/pdf/1405.4053v2.pdf.

Получение векторов документов в результате обучения

Аналогично обучению для получения векторов слов можно использовать пакет `gensim` для получения векторов документов, как показано в листинге 6.13.

Листинг 6.13. Получение путем обучения векторов документов и слов

```
>>> import multiprocessing
>>> num_cores = multiprocessing.cpu_count()

>>> from gensim.models.doc2vec import TaggedDocument,\
...     Doc2Vec
>>> from gensim.utils import simple_preprocess

>>> corpus = ['This is the first document ...',\
...           'another document ...']
>>> training_corpus = []
>>> for i, text in enumerate(corpus):
...    tagged_doc = TaggedDocument(\
...        simple_preprocess(text), [i])
...    training_corpus.append(tagged_doc)

>>> model = Doc2Vec(size=100, min_count=2,
...     workers=num_cores, iter=10)
>>> model.build_vocab(training_corpus)
>>> model.train(training_corpus, total_examples=model.corpus_count,
...     epochs=model.iter)
```

gensim использует модуль multiprocessing языка Python для распараллеливания обучения на несколько ядер CPU. Но данная строка служит лишь для подсчета количества доступных ядер для определения числа процессов-исполнителей

Модель Doc2vec gensim содержит вложения векторов слов и документов для каждого документа в корпусе

Утилита simple_preprocess из gensim представляет собой черновой токенизатор, игнорирующий однобуквенные слова и все знаки пунктуации. Для этой цели подойдет любой токенизатор из главы 2

Вам нужно передать объект, который мог бы пройти в цикле по строкам документа по одной

Читатель 24231 предварительной версии данной книги (https://forums.manning.com/user/profile/24231.page) предложил заранее выделить память под массив NumPy, а не громоздкий список языка Python. Кроме того, можно организовать потоковую передачу корпуса на диск или в базу данных, а также с диска и из базы, если он не помещается в оперативной памяти

gensim предоставляет структуру данных для снабжения документов строковыми или целочисленными тегами для категориальных меток, ключевых слов и любой другой информации, которую необходимо связать с документами

Запускаем обучение на протяжении десяти эпох

Перед обучением модели нужно сформировать словарь

Создание объекта Doc2vec с размером окна десять слов и 100-мерными векторами слов и документов (намного меньшими, чем 300-мерные векторы Word2vec Google News). min_count — минимальная частотность документа для данного словаря

СОВЕТ

Если оперативной памяти мало и количество документов известно заранее (объект корпуса не является итератором или генератором), можно воспользоваться для training_corpus заранее выделенным массивом NumPy вместо списка языка Python:

```
training_corpus = np.empty(len(corpus), dtype=object);
… training_corpus[i] = …
```

После обучения модели `Doc2vec` можно приступить к выводу векторов документов для новых, незнакомых алгоритму документов с помощью вызова метода `infer_vector` для созданной и обученной модели:

```
>>> model.infer_vector(simple_preprocess(\
...     'This is a completely unseen document'), steps=10)
```

Doc2vec требует шага обучения при выводе новых векторов. В нашем примере обученный вектор обновляется за десять шагов (итераций)

Таким образом, можно быстро усвоить целый корпус документов и найти похожие. Сделать это можно путем генерации векторов для всех документов в корпусе и последующего подсчета косинусных расстояний между каждой парой векторов документов. Еще одна часто встречающаяся задача — кластеризация векторов документов корпуса с помощью какого-либо метода, например, метода k-средних для создания классификатора документов.

Резюме

- Вы узнали, как векторы слов и векторные умозаключения позволяют решать некоторые сложные задачи, например вопросы на аналогию и несинонимические взаимосвязи слов.
- Теперь у вас есть возможность обучать Word2vec и другие вложения векторов слов на используемых в ваших приложениях словах так, чтобы не «загрязнять» свой конвейер NLP смыслами слов из Google News, неизбежными в большинстве предобученных моделей Word2vec.
- Мы использовали gensim для изучения, визуализации и создания собственных словарей векторов слов.
- Благодаря PCA-проекции географических векторов слов, таких как названия городов США, можно выявить культурное сходство географически удаленных мест.
- За счет учета границ предложений в *n*-граммах и эффективного создания пар слов для обучения можно существенно повысить точность вложений слов при латентно-семантическом анализе (см. главу 4).

7 Сверточные нейронные сети

В этой главе

- Применение нейронных сетей для NLP.
- Поиск смысла в закономерностях слов.
- Создание сверточных нейронных сетей (CNN).
- Векторизация текста на естественном языке подходящим для нейронных сетей образом.
- Обучение CNN.
- Классификация тональностей нового для сети текста.

Истинная мощь языка заключается не в самих словах, а в пробелах между ними, порядке и сочетаниях слов. Иногда смысл скрыт между строк, в подтексте и чувствах, приведших к данному конкретному сочетанию слов. Понимание подтекста слов — ключевой навык чуткого, эмоционально развитого слушателя или читателя текста на естественном языке, идет ли речь о человеке или машине[1]. Подобно мыслям и идеям, именно связи между словами порождают глубину, информацию и сложность.

[1] International Association of Facilitators Handbook: http://mng.bz/oVWM.

Как, схватив основной смысл отдельных слов и зная множество ловких способов, связать их вместе, заглянуть в их подтекст и измерить смысл сочетаний слов с помощью чего-либо более гибкого, чем простой подсчет количества соответствий *n*-грамм? Как извлечь для каких-либо действий смысл, чувства — *латентно-семантическую информацию* — из последовательности слов? Или даже еще более амбициозная задача: как придать скрытый смысл тексту, сгенерированному бездушной вычислительной машиной?

Даже сама фраза «сгенерированный машиной текст» вселяет ужас, вызывая мысли о глухом металлическом голосе, прерывисто чеканящем список слов. Машины могут передавать смысл, но вряд ли способны на что-то большее. Чего им не хватает? Интонаций, плавности, черт характера, которые человек проявляет даже при самом кратком контакте. Эти тонкости — между и за словами, в закономерностях их составления. В процессе общения люди вплетают паттерны в свои тексты и речь. По-настоящему великие писатели и ораторы активно оперируют этими паттернами для пущего эффекта. И именно вследствие врожденной человеческой способности их распознавать, пусть даже на подсознательном уровне, генерируемая машинами речь и звучит обычно так ужасно. В ней просто не хватает этих паттернов. Но вы можете найти их в человеческой речи и наделить ими машину.

За последние несколько лет исследования в области нейронных сетей испытали настоящий расцвет. Благодаря широко доступным утилитам с открытым исходным кодом возможности нейронных сетей по поиску закономерностей в больших наборах данных быстро преобразили ландшафт NLP. Перцептрон быстро стал упреждающей сетью (многослойным перцептроном), что привело к разработке нескольких новых вариантов: сверточных и рекуррентных нейронных сетей, еще более эффективных и точных инструментов для отлавливания закономерностей из больших наборов данных.

Как вы уже видели на примере Word2vec, нейронные сети сделали возможными совершенно новые подходы к NLP. Хотя исходная цель нейронных сетей заключалась в том, чтобы *научить* машину оценивать количественно входные данные, их поле деятельности выросло с тех пор от классификации и регрессии (анализа тем, тональностей) до возможности настоящей генерации нового текста на основе входных данных, ранее незнакомых модели: перевода новых фраз на другой язык, генерации ответов на новые вопросы (чат-боты) и даже генерации нового текста в стиле конкретного автора.

Для использования описанных в этой главе инструментов не обязательно в совершенстве понимать всю внутреннюю математику нейронных сетей. Но это поможет понять, что происходит внутри. Если вы поняли примеры и объяснения в главе 5, то должны интуитивно представлять, где можно использовать нейронные сети. И знаете, когда можно ослабить архитектуру нейронной сети (уменьшить число слоев или нейронов), чтобы она лучше подходила для конкретной задачи. Эти интуитивные представления помогут понять, как усовершенствовать чат-бот с помощью нейронных сетей. Нейронные сети обещают сделать из вашего чат-бота лучшего слушателя и немного менее поверхностного собеседника.

7.1. Усвоение смысла

Сущность слов и их секреты сильнее всего (после их определений, конечно) коррелируют с взаимосвязями между ними. Эти взаимосвязи можно выразить по крайней мере двумя способами.

1. *Порядок слов* — вот два отнюдь не тождественных утверждения:

```
The dog chased the cat.
The cat chased the dog.
```

2. *Близость слов* — shone здесь относится к слову hull на другом конце предложения:

```
The ship's hull, despite years at sea, millions of tons of cargo, and two mid-sea
collisions, shone like new.
```

Искать в этих взаимосвязях закономерности (помимо закономерностей, заключенных в самом наличии слов) можно двумя способами: пространственным и временным. Отличаются они следующим: в первом случае утверждение рассматривается, как будто оно написано на бумаге, — взаимосвязи ищутся в позициях слов; во втором — как произносимое вслух — слова и буквы становятся данными *временных рядов*. Хотя и родственные, эти способы отражают ключевые различия в работе с данными с помощью инструментов нейронных сетей. Пространственные данные обычно рассматриваются через окно фиксированной ширины. Временные ряды же могут расширяться на произвольный промежуток времени.

Простейшие упреждающие нейронные сети (многослойный перцептрон) способны находить закономерности в данных. Но эти закономерности обнаруживаются с помощью соотнесения весов с элементами входного сигнала. Никакого пространственного или временного захвата взаимосвязей токенов не происходит. Но упреждающие сети — лишь простейшие из архитектур нейронных сетей. Два важнейших варианта нейронных сетей для обработки естественного языка на текущий момент — сверточные нейронные сети и рекуррентные нейронные сети, а также разнообразные их модификации.

На рис. 7.1 во входной слой нейронной сети передается три токена. Каждый из нейронов входного слоя соединен с каждым из нейронов полносвязного скрытого слоя со своим отдельным весом.

СОВЕТ

Каким образом мы будем *передавать токены в сеть*? В основном в этой главе будут применяться два подхода, знакомые вам по предыдущим главам: унитарное кодирование и векторы слов. Можно кодировать их с помощью унитарных векторов, с 1 на позиции, соответствующей кодируемому слову, и 0 на всех остальных позициях. Или можно воспользоваться полученными в результате обучения векторами из главы 6. Чтобы производить над векторами математические действия, необходимо представить их в виде чисел.

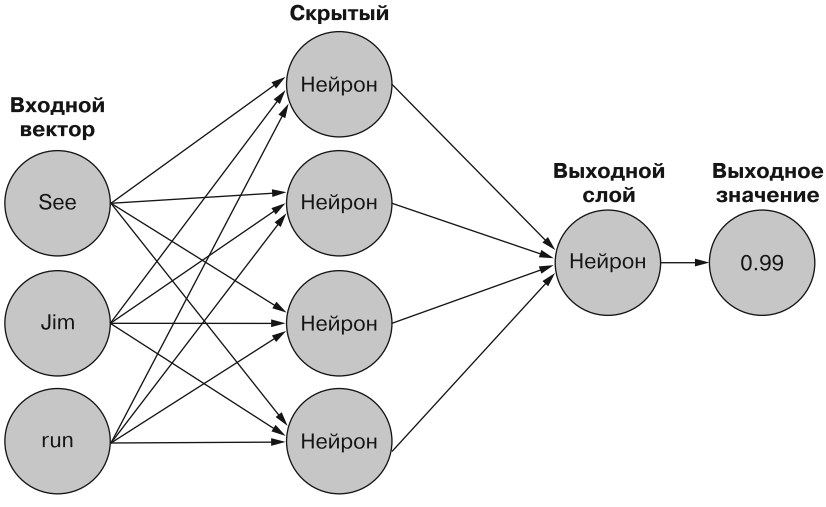


Рис. 7.1. Полносвязная нейронная сеть

Теперь, если поменять местами порядок токенов, с *See Jim run* на *run See Jim* и передать в сеть, будет возвращен, что неудивительно, другой ответ. Как вы помните, каждой позиции во входном сигнале соответствует свой вес в скрытых нейронах (x_1 связан с w_1, x_2 связан с w_2 и т. д.).

Упреждающая нейронная сеть может усвоить подобные конкретные взаимосвязи токенов, поскольку они встречаются в примере вместе, хотя и в различных местах. Но, как вы видите, более длинные предложения из 5, 10 или 50 токенов — со всеми возможными парами, тройками и т. д. на всех возможных позициях для каждой — быстро становятся совершенно неподъемной задачей. К счастью, можно пойти другим путем.

7.2. Инструментарий

Python — один из самых богатых возможностями языков для работы с нейронными сетями. Хотя большинство крупных игроков в этой сфере (Google и Facebook) перешло на реализацию этих ресурсоемких вычислений на низкоуровневых языках, в разработку ранних моделей с помощью Python были направлены колоссальные ресурсы, что оставило свой след. Две основные программы для создания архитектуры нейронных сетей — Theano (http://deeplearning.net/software/theano/) и TensorFlow (http://www.tensorflow.org). В обеих для вычислений используется язык C, но есть в них и надежные API Python. Facebook сделал ставку на пакет Lua под названием Torch; к счастью, в языке Python теперь есть и API для него в виде PyTorch (http://pytorch.org/). Все они, впрочем, при своих возможностях — сильно абстрагированные наборы инструментов для создания моделей с нуля. Но сообщество разработчиков

на Python быстро позаботилось о создании библиотек, упрощающих использование этих архитектур. В числе наиболее популярных вариантов — Lasagne (Theano) и Skflow (TensorFlow), но мы предпочитаем библиотеку Keras (https://keras.io/) за гармоничное сочетание удобного API и разносторонности. Keras может применять в качестве прикладной части как TensorFlow, так и Theano, у каждого из которых есть свои достоинства и недостатки, но мы выбрали для примеров TensorFlow. Вам также понадобится пакет h5py для сохранения внутреннего состояния обученной модели.

По умолчанию Keras использует в качестве прикладной части TensorFlow, и первая выводимая во время выполнения в консоль строка напоминает, какая прикладная часть применяется для обработки. Можно легко поменять прикладную часть в файле конфигурации с помощью переменной среды или в самом сценарии. Документация Keras исчерпывающе описывает все это, мы рекомендуем вам потратить на ее изучение немного времени. Если вкратце: `Sequential()` — класс-абстракция нейронной сети, с помощью которой можно обращаться к основным API Keras, особенно методам `compile` и `fit`, отвечающим за основную работу по созданию базовых весов и их взаимосвязей (`compile`), вычисление ошибок при обучении, а главное, за обратное распространение ошибки (`fit`). `epochs`, `batch_size` и `optimizer` — гиперпараметры, требующие тонкой настройки и в каком-то смысле мастерства.

К сожалению, не существует единого правила на все случаи жизни для проектирования и настройки нейронной сети. Вам придется выработать чутье на то, какой фреймворк лучше подойдет для конкретного приложения. Но если вы нашли пример реализации для задачи, аналогичной вашей, то можете использовать тот же фреймворк и подогнать реализацию под свои нужды. Нет ничего страшного в этом и во всех «наворотах», с которыми вы можете экспериментировать и настраивать. А теперь вернемся на время к обработке естественного языка через призму обработки изображений. Изображений? Потерпите минутку, и вам все станет ясно.

7.3. Сверточные нейронные сети

Сверточные нейронные сети (convolutional neural nets, convnets, CNN) получили свое название от идеи скольжения (свертки) маленького окна по выборке данных.

В математике свертки встречаются во множестве мест и обычно относятся к данным временных рядов. Для наших приложений в данной главе более высокоуровневые идеи, связанные с этими сценариями использования, неважны. Главное — визуализировать для себя скользящий по пространству прямоугольник (рис. 7.2). Сначала мы будем двигать подобный прямоугольник над изображениями, чтобы понять, как это работает, а потом перейдем к окнам, скользящим по тексту. Просто держите в уме мысленный образ окна, скользящего по большому фрагменту данных и ограничивающего наше рассмотрение только видимыми через него данными.

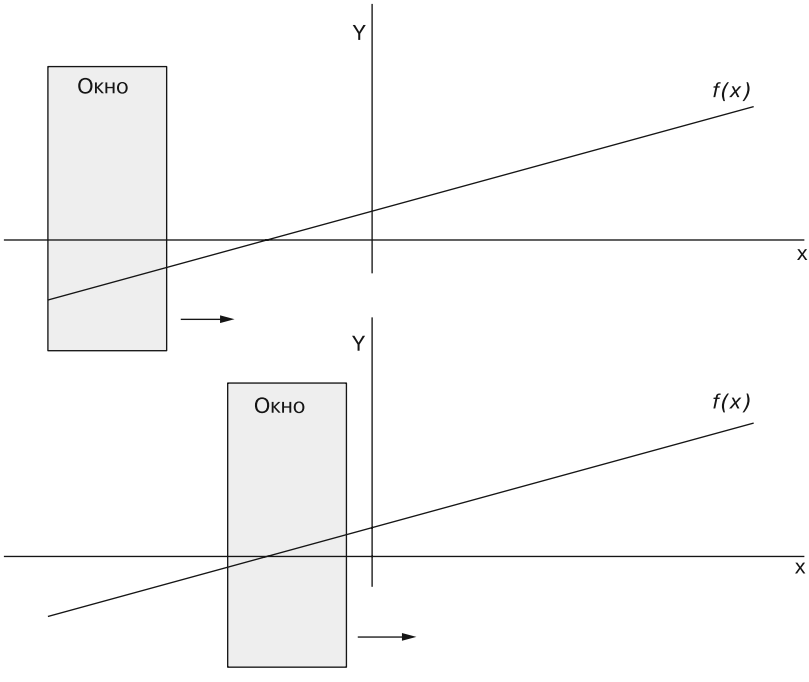


Рис. 7.2. Скользящее по графику функции окно

7.3.1. Стандартные блоки

Первый успех ждал сверточные нейронные сети в области обработки и распознавания изображений. Благодаря способности сети захватывать пространственные взаимосвязи между точками данных выборок она может распознать, содержит ли изображение кошку или собаку, управляющую бульдозером.

Сверточная нейронная сеть добивается таких замечательных результатов не за счет назначения весов всем элементам (скажем, всем пикселам изображения), как в традиционных упреждающих нейронных сетях. Вместо этого описывается набор *фильтров* (называемых также *ядрами*), перемещающихся по изображению! Вот она, *свертка*!

При распознавании элементы точек данных на черно-белом изображении могут принимать значения 1 (включен) или 0 (выключен) для каждого из пикселов. Или могут представлять собой яркость пиксела для изображения в оттенках серого (рис. 7.3 и 7.4) либо яркости цветовых каналов пикселов в цветном изображении.

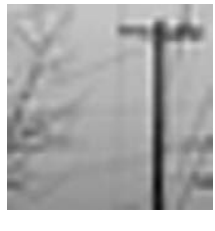


Рис. 7.3. Маленькое изображение телеграфного столба

Каждый из наших фильтров будет *свертывать* входной пример (скользить по нему), в данном случае — значения пикселов. Сделаем паузу и расскажем, что мы имеем в виду под скольжением. Во время движения окна ничего особенного

происходить не будет. Этот процесс можно рассматривать как серию моментальных снимков. Взглянуть через окно, произвести нужную обработку, передвинуть немного и снова обработать.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	120	119	118	108	103	111	113	115	111	117	120	120	120	119	121	114	118	120	120	120	121	121	121	115	100	120	118	117
1	111	109	111	106	107	118	120	110	106	117	120	119	119	119	121	114	118	119	118	119	118	114	119	109	102	122	114	117
2	109	114	121	116	108	119	118	104	111	119	119	119	119	119	119	110	119	121	122	124	73	92	128	100	87	119	115	119
3	109	102	114	117	108	116	108	111	117	118	118	117	119	117	118	79	75	80	76	87	36	63	86	51	55	72	91	120
4	108	110	100	116	111	95	104	110	114	117	117	117	117	117	117	106	107	108	94	90	42	54	70	62	86	70	96	118
5	103	112	109	98	100	93	111	112	115	113	117	117	117	116	116	112	118	116	114	110	41	81	110	102	119	114	119	117
6	111	111	112	104	93	106	119	115	108	110	116	115	116	117	115	107	108	108	111	118	49	87	109	100	115	110	115	115
7	111	111	109	105	103	110	103	103	106	111	115	115	114	110	107	103	113	114	115	115	48	87	105	105	114	111	116	115
8	112	105	108	94	89	102	95	106	111	113	111	108	106	109	112	109	114	113	114	115	45	83	104	108	111	112	114	114
9	112	106	108	86	95	109	104	103	106	104	108	109	114	114	113	108	112	113	113	114	43	80	104	111	109	113	114	113
10	99	111	102	88	111	107	101	101	106	111	112	112	110	113	111	107	112	113	112	112	43	79	106	110	108	114	112	112
11	110	106	93	96	108	107	110	109	111	112	108	107	111	112	111	107	111	111	112	110	38	78	109	108	108	111	114	103
12	101	93	76	101	103	107	107	108	110	107	103	111	111	110	109	106	112	111	111	109	37	82	108	107	111	113	111	100
13	98	92	99	115	108	108	111	106	100	98	106	108	109	110	107	106	109	108	109	107	37	78	106	103	106	108	103	98
14	100	73	97	102	92	95	93	89	89	97	103	103	106	106	103	101	106	104	106	103	33	75	105	103	108	108	98	107
15	84	69	92	87	85	92	89	95	98	100	107	107	107	108	106	104	108	107	109	105	29	74	107	107	110	106	99	109
16	71	82	87	85	78	89	106	104	99	106	106	105	106	105	104	103	106	106	107	103	21	72	106	106	109	100	102	109
17	67	87	64	68	84	89	98	96	99	104	104	104	104	105	103	101	105	103	106	102	23	76	103	103	106	98	108	101
18	68	82	84	97	92	81	84	84	90	98	98	102	102	103	100	99	103	101	103	92	16	76	98	98	89	86	92	73
19	60	71	77	77	80	88	92	91	93	96	96	101	100	101	100	98	101	101	104	92	13	64	93	89	81	89	81	79
20	84	98	87	94	101	100	101	101	103	101	101	100	101	103	98	98	100	96	97	87	12	71	100	97	93	105	91	101
21	92	80	88	99	100	98	100	100	100	97	98	97	97	96	92	91	92	93	96	87	13	79	105	95	98	96	89	100
22	77	80	88	92	96	97	96	95	93	92	93	92	92	92	91	94	96	95	98	89	12	79	103	91	100	89	98	104
23	81	87	83	84	89	89	91	87	90	92	93	95	94	96	94	95	98	94	93	84	7	74	89	86	90	70	81	73
24	60	66	82	92	90	90	87	90	94	94	94	94	93	94	92	93	94	95	95	81	0	76	90	92	81	77	65	58
25	87	81	83	86	87	84	90	92	92	92	92	92	92	92	91	92	92	91	92	77	4	73	91	92	81	95	96	95
26	87	88	88	83	85	91	91	89	90	91	92	91	91	91	89	90	92	90	89	73	8	66	92	83	84	92	91	91
27	81	86	88	91	89	89	89	89	89	88	88	89	89	89	84	83	79	80	87	74	0	60	89	77	90	91	90	89

Рис. 7.4. Значения пикселов для изображения телеграфного столба

ПРИМЕЧАНИЕ

Как раз эта процедура скольжения/моментальных снимков и позволяет так хорошо распараллеливать сверточные нейронные сети. Каждый моментальный снимок для конкретного примера данных можно вычислять независимо от всех остальных (для этого примера). Нет нужды ждать завершения первого снимка перед вторым.

Насколько велики эти фильтры? Размер окна фильтра — параметр, выбираемый создателем модели и сильно зависящий от содержимого данных. Но есть несколько общих отправных точек. При работе с данными изображений часто можно видеть окна размером три на три (3×3) пиксела. Мы обсудим выбор такого размера окна подробнее позднее в данной главе, когда вернемся к NLP.

7.3.2. Размер шага (свертки)

Обратите внимание, что проходимое за шаг свертки расстояние является параметром. Что еще важнее, оно почти всегда меньше самого фильтра. Моментальные снимки обычно пересекаются со своими соседями.

Расстояние, проходимое каждой сверткой, называется *шагом свертки* (stride) и обычно равно 1. Пересечение между различными входными данными фильтра для двух соседних позиций можно получить только при сдвиге на один пиксел (или любое расстояние, меньшее чем ширина фильтра). Больший шаг свертки, при котором не будет перекрытия между областями применения фильтра, приведет к потере эффекта «размытия» в случае одного пиксела (или в нашем случае токена) по отношению к его соседям.

У подобного перекрытия есть несколько интересных свойств, которые станут ясны, когда вы увидите, как фильтры меняются со временем.

7.3.3. Формирование фильтров

Хорошо, мы описали скользящие по данным окна и взгляд на данные через подобные окна, но не сказали, что с этими данными делать.

Фильтры состоят из двух частей:

- набора весов (точно таких же, как веса на входе нейронов из главы 5);
- функции активации.

Как мы уже упоминали ранее, размер фильтров обычно 3 × 3 (хотя часто встречаются и другие размеры и формы).

ПРИМЕЧАНИЕ

> Эти наборы фильтрующих нейронов аналогичны обычным нейронам скрытого слоя, за исключением того, что веса каждого фильтра фиксированы на весь проход по входному примеру. Эти веса одинаковы для всего изображения. Все фильтры в сверточной нейронной сети уникальны, но каждый отдельный элемент фильтра фиксирован в рамках моментального снимка изображения.

Каждый из фильтров, перемещаясь по изображению по шагу свертки за раз, останавливается и делает моментальный снимок охваченных им в текущий момент пикселов. Значения этих пикселов затем умножаются на вес, соответствующий этому местоположению фильтра.

Допустим, мы используем фильтр размером 3 × 3. Мы начинаем с верхнего левого угла и делаем моментальный снимок первого пиксела (0, 0), умноженного на первый вес (0, 0), затем второго пиксела (0, 1), умноженного на второй вес (0, 1), и т. д.

Произведения пикселов и весов (в соответствующей позиции) затем суммируются и передаются в функцию активации (рис. 7.5), в роли которой чаще всего выступают ReLU (выпрямленные линейные блоки), к которым мы вернемся чуть позже.

На рис. 7.5 и 7.6 x_i представляет собой значение пиксела на позиции i, а z_0 — выходной сигнал функции активации ReLU (`z_0 = max(sum(x * w), 0)`, то есть $z_0 = max(x_i \cdot w_j, 0)$). Результаты этой функции активации записываются в качестве значения на соответствующей позиции в выходном «изображении». Затем фильтр смещается на шаг свертки, делает следующий моментальный снимок и помещает выходное значение рядом с предыдущим выходным значением (рис. 7.6).

В одном слое содержится несколько таких фильтров, которые производят свертку по всему изображению и создают таким образом каждый новое (отфильтрованное, если угодно) изображение. В случае, скажем, n фильтров после этого процесса у вас будет n новых отфильтрованных изображений, по одному на каждый из описанных вами фильтров.

Мы вернемся к вопросу о том, что с этими n новыми изображениями, через минуту.

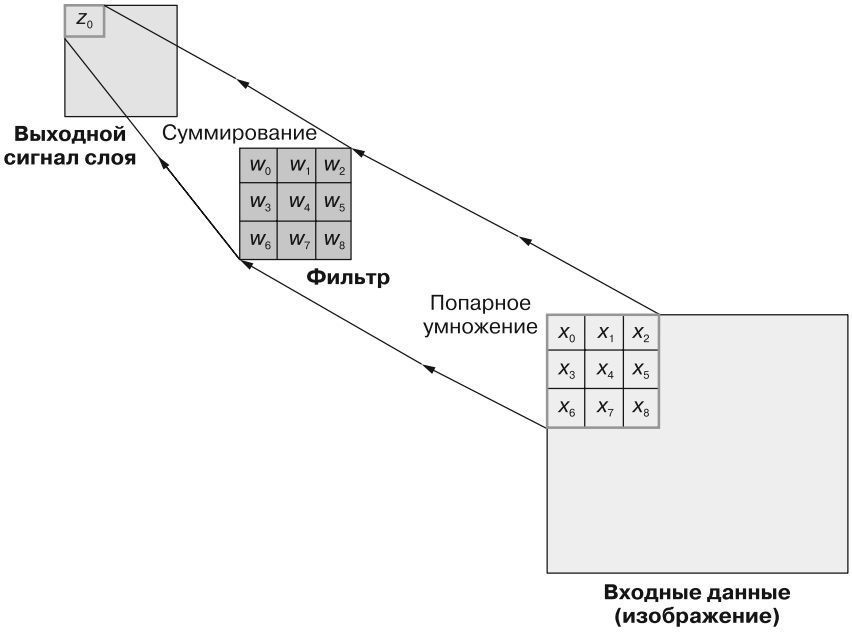


Рис. 7.5. Шаг работы сверточной нейронной сети

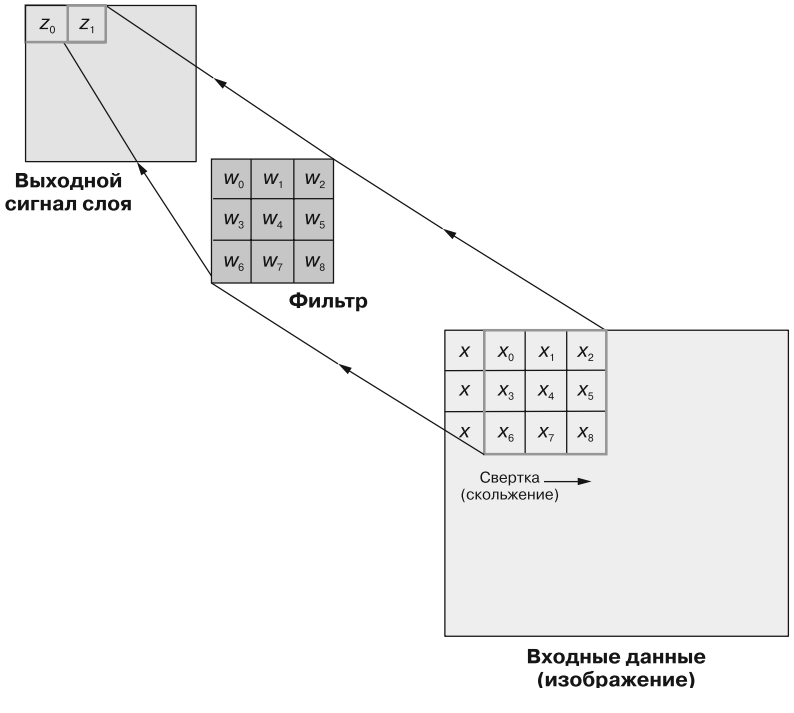


Рис. 7.6. Свертка

7.3.4. Дополнение

Впрочем, по краям изображения происходит что-то странное. Если запустить фильтр 3 × 3 в верхнем левом углу входного изображения, производя шаги свертки по пикселу за раз, и остановиться только по достижении правого края входных данных правым краем фильтра, то выходное изображение окажется на два пиксела уже, чем исходное.

В Keras есть инструменты, с помощью которых можно справиться с этой проблемой. Первый вариант: игнорировать уменьшение размера входных данных. Для этого в Keras служит аргумент `padding='valid'`. В подобном случае достаточно просто принять меры предосторожности и обращать внимание на новые измерения при передаче данных в следующий слой. Минус этой стратегии — недостаточная представленность в выборке исходных входных данных при передаче несколько раз в каждый из фильтров внутренних точек данных при перекрывающихся позициях фильтра. При большом изображении это не проблема, но, как только вы начнете обрабатывать на основе этой идеи, скажем, твиты, недостаточная представленность в выборке слова в начале набора данных из десяти слов может разительно изменить результат.

Следующая стратегия — *дополнение* (padding), то есть добавление по внешним краям входного сигнала дополнительных данных в таком количестве, чтобы первые реальные точки данных обрабатывались точно так же, как внутренние точки данных. Недостаток этой стратегии — во входной сигнал добавляются потенциально не имеющие к нему отношения данные, что само по себе может исказить результат. Вряд ли вам интересно искать закономерности в сгенерированных вами же фиктивных данных. Но существует несколько таких способов дополнения входных данных, которые минимизируют негативные последствия (листинг 7.1).

Листинг 7.1. Сеть Keras с одним сверточным слоем

```
>>> from keras.models import Sequential
>>> from keras.layers import Conv1D

>>> model = Sequential()
>>> model.add(Conv1D(filters=16,
                     kernel_size=3,
                     padding='same',
                      activation='relu',
                     strides=1,
                     input_shape=(100, 300)))
```

Возможные значения параметра: 'same' или 'valid'

input_shape — форма неизмененных входных данных. Дополнение происходит незаметно, «под капотом»

Чуть позже мы расскажем больше подробностей про реализацию. Просто обратите внимание на эти проблемные нюансы и знайте, что используемые утилиты взяли на себя большую часть довольно неприятной работы по первичной обработке данных.

Существуют и другие стратегии, при которых препроцессор пытается угадать, как лучше дополнить данные, имитируя находящиеся на краю точки данных. Но в приложениях NLP эти стратегии не стоит использовать, поскольку они сопряжены со своими опасностями.

Сверточный конвейер

Теперь у нас есть *n* фильтров и *n* новых изображений. Что с ними делать? Отправная точка, как и в большинстве приложений нейронных сетей, — маркированный набор данных. И цель также аналогична: предсказание метки по новому изображению. Простейший следующий шаг: выстроить эти отфильтрованные изображения в ряд в качестве входных данных упреждающего слоя, а затем обработать аналогично главе 5.

СОВЕТ

> Эти отфильтрованные изображения можно передать во второй сверточный слой, содержащий свой набор фильтров. На практике такая архитектура является наиболее распространенной; мы еще встретимся с ней позднее. Оказывается, что несколько слоев сверток — путь к усвоению нескольких слоев абстракций: сначала краев изображения, затем форм/цветов и, наконец, общих идей!

Сколько бы мы ни добавили в сеть слоев (сверточных или каких-либо других), мы можем после получения итогового выходного сигнала вычислить ошибку и распространить ее обратно, в начало сети!

Благодаря дифференцируемости функции активации можно произвести обычное обратное распространение ошибки и обновить веса отдельных фильтров. В результате сеть обучается, усваивая, какие типы фильтров ей требуются, чтобы получить правильный выходной сигнал для заданного входного.

Можно считать, что при этом процессе сеть обучается легче обнаруживать и извлекать необходимую для обработки в последующих слоях информацию.

7.3.5. Обучение

Сами фильтры, как и в любой нейронной сети, начинаются с того, что веса инициализируются близкими к нулю значениями. Не получится ли, что выходное «изображение» будет всего лишь шумом? Сначала, в первые несколько итераций обучения, оно и будет только шумом.

Но создаваемый классификатор будет получать определенную информацию об ошибке на основе ожидаемых меток для каждого элемента входных данных, и эту информацию можно распространить обратно, через функцию активации, на значения самих фильтров. Для обратного распространения ошибки необходимо взять частную производную ошибки по учитываемому в ней весу.

А поскольку ранее в сети находится сверточный слой, то речь идет конкретно о производной градиента из расположенного выше слоя по учитываемому в нем весу. Эти вычисления аналогичны обычному обратному распространению ошибки, так как вес формирует выходной сигнал на многих позициях для конкретного тренировочного примера.

Нюансы взятия производных градиента по весу сверточного фильтра выходят за рамки данной книги. Но если коротко, можно считать, что для заданного веса в заданном фильтре этот градиент равен сумме обычных градиентов для всех отдельных

позиций в свертке во время прямого прохода. Это достаточно сложная формула (две суммы и целая стопка уравнений, как показано ниже):

$$\frac{\partial E}{\partial w_{ab}} = \sum_{i=0}^{m} \sum_{j=0}^{n} \frac{\partial E}{\partial x_{ij}} \frac{\partial x_{ij}}{\partial w_{ab}}$$

Сумма градиентов для веса фильтра

Эта концепция сильно напоминает обычную упреждающую сеть, в которой определяются вклады отдельных весов в общую ошибку системы. Далее мы выбираем оптимальную стратегию корректировки в сторону веса, при котором ошибка в будущих тренировочных примерах была бы минимальной. Все эти подробности неважны для понимания обработки естественного языка с помощью сверточных нейронных сетей. Надеемся, что вы уже интуитивно чувствуете, как довести архитектуру нейронной сети до совершенства и как развить приведенные примеры.

7.4. Окна и правда узкие

Да, да, хорошо, изображения. Но мы же говорим о языке, помните? Попробуем обучить сеть на каких-нибудь словах. Оказывается, что сверточную нейронную сеть можно использовать для обработки естественного языка, если вместо значений пикселов изображения в качестве входных данных для сети применять векторы слов *(вложения слов)*, о которых вы знаете из главы 6.

Поскольку относительные вертикальные связи между словами носят произвольный характер и зависят от ширины страницы, нельзя извлечь никакой относящейся к делу информации из возникающих здесь закономерностей. Все интересное содержится в относительных «горизонтальных» позициях.

СОВЕТ

> То же самое справедливо и для языков, в которых текст читается *сначала* сверху вниз, а только потом слева направо, например японского. В этих случаях просто нужно учитывать «вертикальные» связи вместо «горизонтальных».

Нас интересуют только взаимосвязи токенов в одном пространственном измерении. Вместо 2D-фильтра, свертываемого по двумерным входным данным (картинке), мы будем производить свертки одномерных фильтров по одномерным входным данным, например, предложениям.

Форма наших фильтров также будет одномерной, а не двумерной, как показано в скользящем окне 1 × 3 на рис. 7.7.

Представьте текст в виде изображения с полной длиной вектора слов (обычно 100–500 измерений) в качестве второго измерения, точно как у настоящего изображения. Нам важна только ширина фильтра. На рис. 7.7 ширина фильтра составляет три токена. Ага! Обратите внимание, что каждый токен слова (или далее токен символа) является «пикселом» в нашем «изображении».

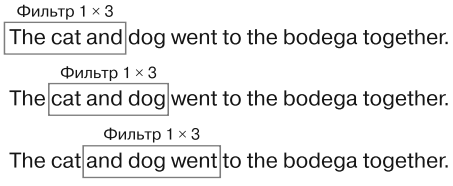


Рис. 7.7. Одномерная свертка

ПРИМЕЧАНИЕ

Термин «одномерный фильтр» может ввести читателя в заблуждение при разговоре о вложениях слов. Векторное представление самого слова простирается «вниз», как показано на рис. 7.8, но фильтр охватывает всю длину этого измерения за один раз. Измерение, на которое ссылается выражение «одномерная свертка», представляет собой «ширину» фразы — это измерение, вдоль которого мы движемся. При двумерной свертке, скажем, изображения входные данные просматриваются из стороны в сторону и сверху вниз, отсюда и ее название. В данном же случае скольжение производится только в одном измерении — слева направо.

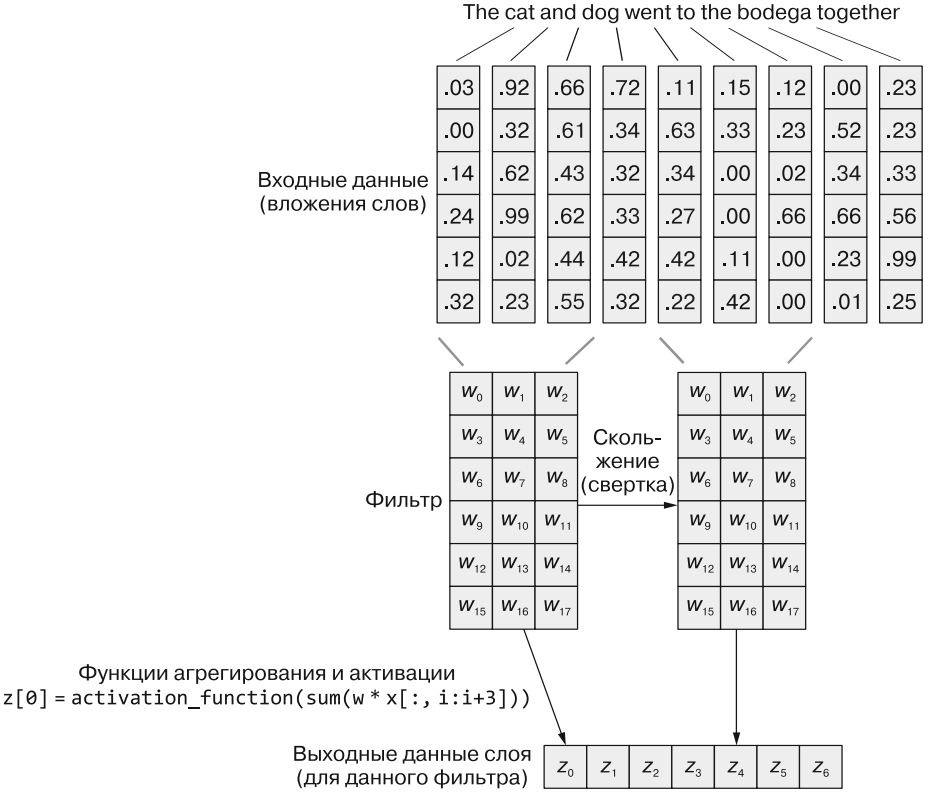


Рис. 7.8. Одномерная свертка с вложениями

Как уже упоминалось, термин «свертка» на самом деле несколько условен. Впрочем, стоит еще раз отметить: сама свертка на модель не влияет. Происходящее определяется данными на различных позициях. Порядок вычисления «моментальных снимков» неважен, лишь бы выходные данные формировались в соответствии с расположением окон над входными данными.

Значения весов в фильтрах не меняются для конкретного входного примера на протяжении прямого прохода, а значит, можно применить некоторый фильтр, получить все его моментальные снимки параллельно и сформировать выходное «изображение» одним махом. В этом и заключается секрет скорости работы сверточных нейронных сетей.

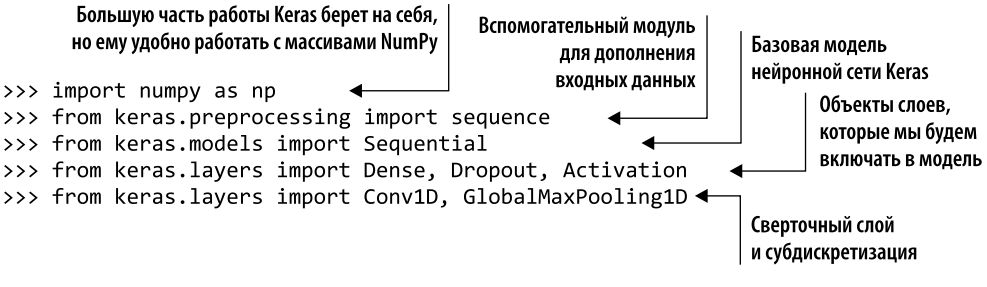
Именно из-за скорости и способности игнорировать позиции признаков исследователи постоянно обращаются к сверточному подходу для выделения признаков.

7.4.1. Реализация на Keras: подготовка данных

Взглянем на реализацию свертки на языке Python на примере сверточной нейронной сети-классификатора, приведенной в документации Keras. Они создали одномерную сверточную сеть для исследования набора данных отзывов о фильмах из базы IMDB.

Каждая точка данных предварительно получает метку 0 (отрицательная тональность) или 1 (положительная тональность). В листинге 7.2 мы собираемся загрузить пример набора данных отзывов о фильмах из базы IMDB в виде необработанного текста, чтобы попробовать себя в деле предварительной обработки текста. Затем посмотрим, удастся ли нам использовать эту обученную сеть для классификации нового для нее текста.

Листинг 7.2. Импорт утилит свертки Keras



Сначала необходимо скачать исходный набор данных с сайта кафедры ИИ Стэнфордского университета по адресу https://ai.stanford.edu/%7eamaas/data/sentiment/. Этот набор данных был создан для статьи 2011 года *Learning Word Vectors for Sentiment Analysis*[1]. После скачивания разархивируйте набор в удобный каталог и за-

[1] *Maas A. L. et al.* Learning Word Vectors for Sentiment Analysis // Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, June 2011, Association for Computational Linguistics.

гляните внутрь. Нас интересует только каталог `train`, но там есть еще много всего интересного, так что не стесняйтесь посмотреть внимательнее.

Отзывы в каталоге `train` разбиты на текстовые файлы, расположенные в подкаталогах `pos` и `neg`. Сначала необходимо прочитать их на Python вместе с соответствующими метками, после чего «перетасовать колоду», чтобы выборка не состояла только из позитивных или негативных примеров. Обучение на отсортированных метках приводит к перекосу в сторону последних из них, особенно при использовании определенных гиперпараметров, например `momentum` (листинг 7.3).

Листинг 7.3.[1] Препроцессор для загрузки документов

```
>>> import glob
>>> import os

>>> from random import shuffle

>>> def pre_process_data(filepath):
...     """
...     Эта функция зависит от источника тренировочных данных, но мы постараемся
...     максимально ее обобщить.
...     """
...     positive_path = os.path.join(filepath, 'pos')
...     negative_path = os.path.join(filepath, 'neg')
...     pos_label = 1
...     neg_label = 0
...     dataset = []
...
...     for filename in glob.glob(os.path.join(positive_path, '*.txt')):
...         with open(filename, 'r') as f:
...             dataset.append((pos_label, f.read()))
...
...     for filename in glob.glob(os.path.join(negative_path, '*.txt')):
...         with open(filename, 'r') as f:
...             dataset.append((neg_label, f.read()))
...
...     shuffle(dataset)
...
...     return dataset
```

Первый пример документа должен выглядеть приблизительно следующим образом. Ваш может оказаться другим в зависимости от перетасовки выборки, это

[1] При вызове функции из листинга 7.3 может возникнуть ошибка кодировки из-за содержащихся в данных символов не из таблицы ASCII. Простейший способ решения этой проблемы — воспользоваться в соответствующих местах метода pre_process_data функцией open из пакета io, указав кодировку UTF:

```
import io
...
    with io.open(filename, 'r', encoding='utf-8') as f:
...
```

— *Примеч. пер.*

нормально. Первый элемент в кортеже — *целевое* значение для тональности: `1` — для положительной тональности и `0` — для отрицательной:

```
>>> dataset = pre_process_data('<path to your downloaded file>/aclimdb/train')
>>> dataset[0]
(1, 'I, as a teenager really enjoyed this movie! Mary Kate and Ashley
➥ worked great together and everyone seemed so at ease. I thought the
➥ movie plot was  very good and hope everyone else enjoys it to! Be sure
➥ and rent it!! Also they had some great soccer scenes for all those
➥ soccer players! :)')
```

Следующий этап — токенизация и векторизация данных. Мы воспользуемся предобученными векторами Word2vec Google News, так что нужно скачать их через пакет nlpia или непосредственно из хранилища Google[1].

Далее мы воспользуемся `gensim` для распаковки векторов, как мы делали в главе 6. Можете поэкспериментировать с аргументом `limit` метода `load_word2vec_format`. Чем больше его значение, тем больше у вас будет векторов для работы, но при этом быстро возникнет проблема с расходом оперативной памяти и отдача при действительно больших значениях аргумента `limit` быстро упадет.

Напишем вспомогательную функцию для токенизации данных и создания списка векторов для этих токенов, которые будут служить входными данными для модели, как показано в листинге 7.4.

Листинг 7.4. Векторизатор и токенизатор

```
>>> from nltk.tokenize import TreebankWordTokenizer
>>> from gensim.models.keyedvectors import KeyedVectors
>>> from nlpia.loaders import get_data
>>> word_vectors = get_data('w2v', limit=200000)

>>> def tokenize_and_vectorize(dataset):
...     tokenizer = TreebankWordTokenizer()
...     vectorized_data = []
...     expected = []
...     for sample in dataset:
...         tokens = tokenizer.tokenize(sample[1])
...         sample_vecs = []
...         for token in tokens:
...             try:
...                 sample_vecs.append(word_vectors[token])
...
...             except KeyError:
...                 pass # В словаре Google w2v нет соответствующего токена
...
...         vectorized_data.append(sample_vecs)
...
...     return vectorized_data
```

Вызов get_data('w2v') выполняет скачивание файла GoogleNews-vectors-negative300.bin.gz в каталог nlpia.loaders.BIGDATA_PATH

[1] См. GoogleNews-vectors-negative300.bin.gz — Google Drive по адресу https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing.

Обратите внимание, что здесь мы отбрасываем некоторую информацию. Словарь Word2vec Google News включает определенные стоп-слова, но не все. Множество распространенных слов вроде *a* окажется выброшено в нашей функции. Не лучший вариант, как ни крути, но благодаря этому вы увидите, насколько хорошо сверточные нейронные сети работают даже при потере части данных. Чтобы учесть эту потерю информации, можно обучить модели `word2vec` отдельно и обеспечить наилучший охват векторов. В этих данных также содержится множество тегов HTML вроде `<br\>`, которые уж точно нужно отфильтровать, поскольку обычно они никак не связаны с тональностью текста.

Необходимо также собрать целевые значения: `0` — для отрицательных отзывов, `1` — для положительных — в том же порядке, что и тренировочные примеры (листинг 7.5).

Листинг 7.5. Целевые метки

```
>>> def collect_expected(dataset):
...     """ Выбираем целевые значения из набора данных """
...     expected = []
...     for sample in dataset:
...         expected.append(sample[0])
... return expected
```

Затем можно просто передать данные в эти функции:

```
>>> vectorized_data = tokenize_and_vectorize(dataset)
>>> expected = collect_expected(dataset)
```

Далее мы разбиваем подготовленные данные на тренировочный и тестовый наборы данных. Вам нужно всего лишь разбить импортированный набор данных в соотношении 80/20, не считая каталога с тестовыми данными. Можете смело сочетать данные из исходного каталога `test` закачки с каталогом `train`. И тот и другой содержат допустимые тренировочные и тестовые данные. Чем больше данных, тем лучше. Каталоги `train/` и `test/` в большинстве наборов данных, которые вы будете скачивать, отражают конкретное разбиение данных на тренировочные/тестовые, использовавшееся создателем пакета. Эти каталоги предоставляются, чтобы можно было в точности воспроизвести их результаты[1].

Следующий фрагмент кода заносит данные в тренировочный набор (`x_train`), который мы покажем нейронной сети наряду с «правильными» ответами (`y_train`), и тестовый набор (`x_test`), который мы пока что придержим вместе с соответствующими ответами (`y_test`). Дальше можно сравнить «догадки» сети относительно примеров из тестового набора и убедиться, научилась ли она обобщать что-либо, помимо тренировочных данных. `y_train` и `y_test` — «правильные» ответы для соответствующих примеров из наборов `x_train` и `x_test` (листинг 7.6).

[1] Имеет смысл публиковать информацию об эффективности работы модели на новом для нее тестовом наборе данных. Но при этом желательно использовать все доступные маркированные данные для последнего, итогового обучения поставляемой заказчику модели.

Листинг 7.6. Разбиение на тренировочные/тестовые данные

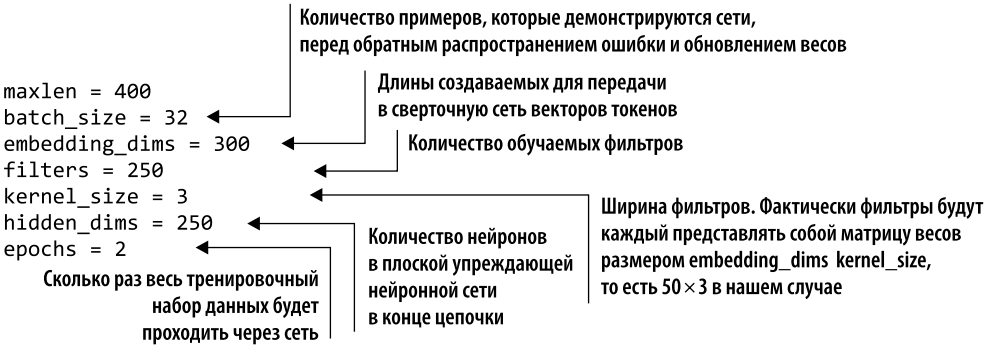
```
>>> split_point = int(len(vectorized_data)*.8)

>>> x_train = vectorized_data[:split_point_]
>>> y_train_ = expected[:split_point]
>>> x_test = vectorized_data[split_point:]
>>> y_test = expected[split_point:]
```

В следующем фрагменте кода (листинг 7.7) задается большинство гиперпараметров сети. Переменная `maxlen` содержит максимально возможную длину отзыва. Поскольку все входные сигналы нейронной сети должны быть одинаковой длины, мы будем усекать все примеры длиннее 400 токенов и дополнять `Null` (0), более короткие, до 400 токенов. При демонстрации исходного текста это обычно иллюстрируется с помощью токенов `PAD`. При этом в систему попадают новые для нее данные. Впрочем, это еще не конец света, сеть обучится такой закономерности, и частью ее структуры станет PAD == «игнорируй меня».

Внимание: это не то дополнение, что упоминалось ранее. Здесь мы дополняем входные данные до одинакового размера. Необходимо отдельно решать, нужно ли дополнять начало и конец каждого тренировочного примера в зависимости от того, хотите ли вы, чтобы выходные данные были аналогичного размера, а конечные токены обрабатывались так же, как и внутренние, или не возражаете, чтобы первые/последние токены обрабатывались иначе (листинг 7.7).

Листинг 7.7. Параметры CNN



СОВЕТ

В листинге 7.7 параметр kernel_size (размер фильтра или окна) представляет собой скалярное значение, в отличие от двумерных фильтров в случае изображения. Наш фильтр будет рассматривать за раз векторы слов для трех токенов. Имеет смысл рассматривать размеры фильтров *только в первом слое* аналогично *n*-граммам текста. В данном случае мы рассматриваем триграммы входного текста. Но с таким же успехом это могут быть 5-граммы, 7-граммы или *n*-граммы для еще большего *n*. Выбор зависит от задачи и данных, так что смело экспериментируйте с этим параметром для ваших моделей.

В Keras есть вспомогательный метод предварительной обработки, pad_sequences, который теоретически можно использовать для дополнения входных данных. К сожалению, он работает только с последовательностями скалярных значений, а у нас последовательности векторов. Напишем собственную вспомогательную функцию для дополнения входных данных, показанную в листинге 7.8.

Листинг 7.8. Дополнение и усечение последовательности токенов

```
>>> def pad_trunc(data, maxlen):
...     """
...     Дополнение для указанного набора данных
...     нулевыми векторами или усечение до maxlen
...     """
...     new_data = []
...
...     # Создаем вектор нулей такой же длины,
...     # что и у наших векторов слов
...     zero_vector = []
...     for _ in range(len(data[0][0])):
...         zero_vector.append(0.0)
...
...     for sample in data:
...         if len(sample) > maxlen:
...             temp = sample[:maxlen]
...         elif len(sample) < maxlen:
...             temp = sample
...             # Присоединяем к списку соответствующее
...             # количество нулевых векторов
...             additional_elems = maxlen - len(sample)
...             for _ in range(additional_elems):
...                 temp.append(zero_vector)
...         else:
...             temp = sample
...         new_data.append(temp)
...     return new_data
```

Внимательный читатель онлайн-версии (LiveBook) данной книги (@madara) указал, что все это можно реализовать в виде одной строки кода: [smp[:maxlen] + [[0.] * emb_dim] * (maxlen – len(smp)) for smp in data]

Дополненные данные готовы для присоединения в конец нашего списка дополненных данных

Далее необходимо передать тренировочные и тестовые данные в процедуру дополнения/усечения. После этого можно преобразовать их в массивы NumPy для удобства Keras. Для нашей CNN необходим тензор формы (количество примеров, длина последовательности, длина векторов слов) (листинг 7.9).

Листинг 7.9. Собираем наши дополненные и усеченные данные

```
>>> x_train = pad_trunc(x_train, maxlen)
>>> x_test = pad_trunc(x_test, maxlen)

>>> x_train = np.reshape(x_train, (len(x_train), maxlen, embedding_dims))
>>> y_train = np.array(y_train)
>>> x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims))
>>> y_test = np.array(y_test)
```

Теперь мы готовы к созданию нейронной сети.

7.4.2. Архитектура сверточной нейронной сети

Начинаем с базового класса `Sequential` модели нейронной сети в Keras. На этот фундамент можно наслаивать дальнейшую магию.

Сначала мы добавим сверточный слой. В данном случае мы допускаем, что размерность выходных данных меньше, чем входных, а аргумент `padding` равен `'valid'`. Все фильтры будут проходить с левого края (начала) предложения и заканчивать работу на правом крае, на последнем токене.

Размер каждого сдвига (шага свертки) будет равен одному токену. Ядро (ширину окна) мы уже задали равной трем токенам в листинге 7.7. И мы используем функцию активации `'relu'`. На каждом шаге мы умножаем вес фильтра на значение из трех просматриваемых им токенов (поэлементно), суммируем результаты и передаем их далее, если они больше 0, в противном случае возвращаем 0. Этот последний этап передачи положительных значений и нулей представляет собой функцию активации в виде *выпрямленного линейного блока* (ReLU) (листинг 7.10).

Листинг 7.10. Формируем одномерную CNN

```
>>> print('Build model...')
>>> model = Sequential()

>>> model.add(Conv1D(
...     filters,
...     kernel_size,
...     padding='valid',
...     activation='relu',
...     strides=1,
...     input_shape=(maxlen, embedding_dims)))
```

Стандартный для Keras паттерн описания модели. В главе 10 мы познакомимся с альтернативным паттерном — «функциональным API» Keras

Добавляем один слой Conv1D, отвечающий за усвоение весов фильтров групп слов размера kernel_size. У этого метода есть еще множество именованных аргументов, но пока воспользуемся их значениями по умолчанию

7.4.3. Субдискретизация

Мы начали создание нейронной сети, так что пора провести субдискретизацию. *Субдискретизация* — способ понижения размерности для сверточных нейронных сетей. В каком-то смысле мы ускоряем процесс за счет распараллеливания вычислений. Но, как вы могли заметить, для каждого описанного нами фильтра создается новая отфильтрованная «версия» примера данных. В предыдущем примере из первого слоя будет возвращено 250 отфильтрованных версий (см. листинг 7.7). Субдискретизация несколько уменьшает это число, но у нее есть и другое замечательное свойство.

Ключевая идея — равномерное разбиение входного сигнала каждого из фильтров по подсекциям с последующим выбором/вычислением для каждой из этих подсекций представительного значения. После чего можно будет отложить в сторону

исходный выходной сигнал и использовать набор этих представительных значений в качестве входного сигнала для следующих слоев.

Но постойте-ка. Ведь выбрасывать данные плохо? Обычно отбрасывание данных не лучший образ действий. В данном случае это путь к усвоению более высокоуровневых представлений исходных данных. Фильтры обучаются искать закономерности, а закономерности проявляют себя во *взаимосвязях* соседних слов! Как раз такую едва уловимую информацию мы и хотели бы обнаружить.

При обработке первые слои обучаются обнаруживать края изображения — области, по разные стороны которых плотность пикселов резко меняется. Дальнейшие слои усваивают, например, форму и визуальную текстуру, а следующие уже могут усваивать «содержание» (смысл). Подобные процессы происходят и при обработке текста.

ПРИМЕЧАНИЕ

При обработке изображений область субдискретизации обычно представляет собой окно 2 × 2 пиксела (причем они не пересекаются в отличие от фильтров), но в нашей одномерной свертке эти окна будут одномерными (например, 1 × 2 или 1 × 3).

Существует два варианта субдискретизации (рис. 7.9): с выбором *максимального* или *среднего* значения. Более интуитивно понятным является второй из них, поскольку операция взятия среднего для подмножества значений теоретически сохраняет больше информации. У субдискретизации с выбором максимального значения, впрочем, есть интересная особенность: благодаря взятию максимального значения активации для заданной области сеть видит наиболее ярко выраженный признак этой подсекции. Таким образом, у сети появляется возможность усвоить нужное вне зависимости от конкретной позиции на уровне пикселов!

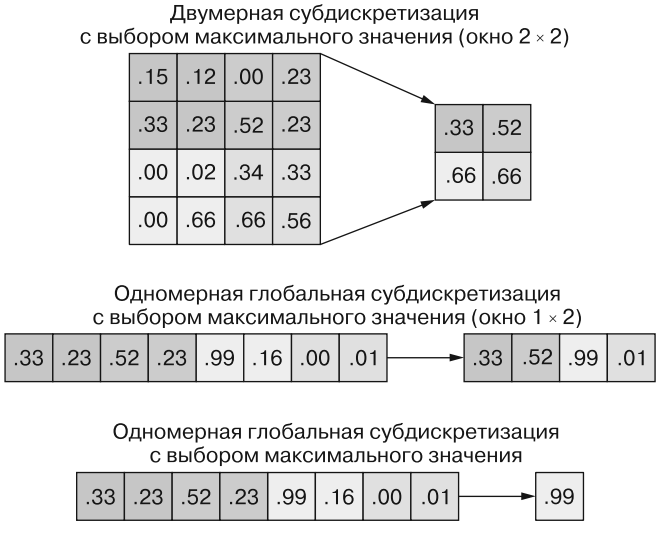


Рис. 7.9. Слои субдискретизации

Помимо понижения размерности и следующей из него экономии вычислительных ресурсов, мы получаем еще одно преимущество: *независимость от расположения* (location invariance). Если элемент входного сигнала слегка сдвинут по позиции в аналогичном, но уникальном входном примере, слой субдискретизации с выбором максимального значения все равно вернет нечто близкое. Это огромный плюс для мира распознавания изображений, служащий аналогичным целям и при обработке естественного языка.

В этом простом примере из Keras мы используем слой `GlobalMaxPooling1D`. Вместо взятия максимума маленькой подсекции выходного сигнала каждого из фильтров вычисляется максимум всего выходного сигнала этого фильтра, в результате чего теряется огромное количество информации. Но даже отбрасывание всей этой хорошей информации не помешает работе нашей модели:

```
>>> model.add(GlobalMaxPooling1D())
```

Возможные варианты субдискретизации: GlobalMaxPooling1D(), MaxPooling1D(n) и AvgPooling1D(n), где n — размер области субдискретизации, по умолчанию равный 2

Отлично, с субдискретизацией закончили. Резюмируем проделанное.

- Применение фильтра (веса и функция активации) для каждого входного примера.
- Свертка по входным данным, в результате которой для каждого фильтра получается одномерный вектор чуть меньшего размера, чем входной (1×398 — входные данные, когда фильтр начинает работу с выравниванием влево и заканчивает с выравниванием вправо).
- Вычисляем одно максимальное значение из каждого одномерного вектора для выходного сигнала каждого из фильтров (напомним, их 250).
- Получаем по одному вектору для каждого входного примера размерности 1×250 (по количеству фильтров).

Получаем для каждого входного примера по одномерному вектору, который с точки зрения сети хорошо отражает этот входной пример. Это *семантическое* представление входных данных — довольно приближенное, конечно. Причем семантическим оно является только в контексте цели обучения, то есть тональности. В нем не закодировано содержание, скажем, отзыва о соответствующем фильме, а только его тональность.

Пока что мы не выполняли никакого обучения, так что все это — лишь большая куча чисел. Но к нему мы вернемся позднее. Важно здесь сделать паузу и разобраться по-настоящему, что происходит, чтобы суметь воспользоваться этим *семантическим* представлением (мы предпочитаем считать его вектором идеи, thought vector) после обучения сети. Подобно множеству способов вложения слов в векторы, теперь для нас становятся доступными и математические операции над ними: у нас теперь есть сущность для представления целых группировок слов.

Хватит радоваться, вернемся к нашей непростой работе по обучению. У нас есть цель и метки для тональностей. Передаем текущий вектор в обычную упреждающую сеть; в Keras эту роль играет слой `Dense`. При наших настройках количество элементов в семантическом векторе совпадает с числом узлов в слое `Dense`, но это просто совпадение. У каждого из 250 (`hidden_dims`) нейронов этого слоя есть 250 весов для входного сигнала из слоя субдискретизации. Необходимо пропустить их через слой дропаута во избежание переобучения.

7.4.4. Дропаут

Дропаут (представленный в Keras в виде слоя, показанного в листинге 7.11) — это специальная методика предотвращения переобучения в нейронных сетях. Он применяется не только для обработки естественного языка, но хорошо подходит и для нее.

Идея дропаута заключается в «выключении» определенной части передаваемого на следующий слой входного сигнала, выбираемой при каждом проходе случайным образом. В результате модель скорее усвоит не особенности именно тренировочного набора данных (и переобучится), а детальные представления закономерностей данных и, следовательно, окажется способна на обобщения и точные предсказания на основе совершенно новых для нее данных.

Наша модель реализует дропаут, принимая равным нулю (при данном проходе) выходной сигнал предыдущего (перед дропаутом) слоя. Это срабатывает для конкретного прохода, поскольку вклад в общую ошибку каждого из весов нейронов, получивших нулевой входной сигнал дропаута, также будет фактически нулевым. Следовательно, эти веса не обновятся на этапе обратного распространения ошибки. И сети придется достигать своих целей на основе взаимосвязей между переменными наборами весов (надеемся, они не припомнят нам потом этого).

СОВЕТ

> Не стоит слишком углубляться в это, но «под капотом» слоев дропаута в Keras происходит много всего интересного. Keras случайным образом выключает определенную долю входных сигналов на каждом прямом проходе тренировочных данных. В настоящем приложении подобный дропаут не производится при выводе и предсказании. На этапе вывода без обучения мощность сигнала, поступающего в следующие за дропаутом слои, будет значительно выше.
>
> На этапе обучения Keras уменьшает этот эффект за счет пропорционального усиления всех неотключенных входных сигналов, так что «амплитуда» суммарного поступающего в следующий слой сигнала будет аналогична этапу вывода.

В слой дропаута в Keras передается параметр, определяющий процентную долю «выключаемых» случайным образом входных сигналов. В текущем примере только 80 % данных вложений, выбираемых случайным образом для каждого из тренировочных примеров, будет передаваться в следующий слой в неизменном виде.

Остальные будут переданы как 0. Обычно используется дропаут 20 %, но неплохие результаты демонстрирует и дропаут вплоть до 50 % (еще один гиперпараметр, с которым стоит поэкспериментировать).

А далее мы применяем функцию активации-выпрямителя (ReLU) на выходе всех нейронов (см. листинг 7.11).

Листинг 7.11. Полносвязный слой с дропаутом

```
>>> model.add(Dense(hidden_dims))
>>> model.add(Dropout(0.2))
>>> model.add(Activation('relu'))
```

Начинаем с простого полносвязного скрытого слоя, а затем присоединяем дропаут и ReLU

7.4.5. Вишенка на торте

Последний (выходной) слой и является, собственно, классификатором, так что в нем нейроны возбуждаются на основе сигма-функции активации (`sigmoid`), которая возвращает значение от 0 до 1. Во время проверки Keras классифицирует любое значение менее 0,5 как 0, а больше 0,5 — как 1. Но в пересчете на *потери* речь идет о целевой величине минус фактическое значение, возвращаемое сигма-функцией: $(y - f(x))$.

Здесь мы производим проекцию на однонейронный выходной слой и «процеживаем» сигнал через сигма-функцию активации, как показано в листинге 7.12.

Листинг 7.12. «Процеживание»

```
>>> model.add(Dense(1))
>>> model.add(Activation('sigmoid'))
```

Итак, мы полностью описали в Keras модель сверточной нейронной сети. Осталось только ее скомпилировать и обучить, как показано в листинге 7.13.

Листинг 7.13. Компиляция CNN

```
>>> model.compile(loss='binary_crossentropy',
...               optimizer='adam',
...               metrics=['accuracy'])
```

Параметр `loss` определяет, какую функцию сеть будет пытаться минимизировать. В данном случае мы используем функцию потерь `'binary_crossentropy'`. На момент написания книги в Keras было описано 13 функций потерь, плюс можно было описать собственную. Мы не будем углубляться в сценарии использования всех, но не помешает знать про основные две: `binary_crossentropy` и `categorical_crossentropy`.

Математические определения обеих похожи, и во многих отношениях можно рассматривать `binary_crossentropy` как частный случай `categorical_crossentropy`. Главное — знать, когда какую из них использовать. Поскольку в этом примере у нас только один выходной нейрон, который может быть включен или выключен, мы воспользуемся `binary_crossentropy`.

Дискретная перекрестная энтропия (`categorical_crossentropy`) используется для предсказания одного из нескольких классов. В подобных случаях целью является n-мерный унитарный вектор, в котором есть по позиции для каждого из наших n классов. В этом случае последний слой сети будет таким, как показано в листинге 7.14.

Листинг 7.14. Выходной слой для дискретной переменной (слова)

```
>>> model.add(Dense(num_classes))
>>> model.add(Activation('sigmoid'))
```

Где параметр num_classes равен... ну вы поняли

В данном случае целевая величина минус выходной сигнал ($y - f(x)$) — это n-мерный вектор, вычитаемый из другого n-мерного вектора. И параметр `categorical_crossentropy` минимизирует эту разницу.

Но вернемся к нашей бинарной классификации.

Оптимизация

Параметр `optimizer` может принимать значение, соответствующее любой из перечня стратегий оптимизации сети во время обучения, например стохастическому градиентному спуску, Adam или RSMProp. Сами оптимизаторы представляют собой различные подходы к минимизации функции потерь в нейронной сети. Их формальное математическое описание выходит за рамки данной книги, достаточно знать про их существование и не стесняться пробовать различные оптимизаторы для конкретной задачи. Для конкретной задачи некоторые из них могут сходиться, некоторые — нет, причем все — с разной скоростью.

Хитрость состоит в динамическом изменении параметров обучения, особенно *скорости обучения*, в зависимости от текущего его состояния. Например, начальная скорость обучения (напомним: как вы видели в главе 5, к обновлениям весов применяется скорость обучения α) может с течением времени снижаться. Некоторые методы могут учитывать *моментум* и повышать скорость обучения, если последнее изменение весов в данную сторону было успешным (снижало потери).

У каждого из оптимизаторов есть несколько своих гиперпараметров, например скорость обучения. В Keras описаны неплохие значения по умолчанию для них, так что поначалу можете не слишком о них задумываться.

Fit

Метод `compile` создает модель, а метод `fit` осуществляет обучение. Все умножение входных сигналов на веса, все функции активации, все обратное распространение ошибки запускается одним оператором. В зависимости от аппаратного обеспечения и размеров модели и данных этот процесс может занимать от пары секунд до нескольких месяцев. В большинстве случаев использование GPU может сильно сократить время обучения, так что если у вас есть графический процессор — обязательно им воспользуйтесь. Чтобы заставить Keras применять GPU, необходимо проделать дополнительные действия по передаче в Keras переменных среды, но наш текущий

пример достаточно мал для выполнения на большинстве современных CPU за разумное время (листинг 7.15).

Листинг 7.15. Обучение CNN

```
>>> model.fit(x_train, y_train,
...           batch_size=batch_size,
...           epochs=epochs,
...           validation_data=(x_test, y_test))
```

Количество примеров данных, обрабатываемых перед тем, как веса обновляются с помощью обратного распространения ошибки

Количество проходов обучения по всему тренировочному набору данных перед остановом

7.4.6. Приступаем к обучению

Последний шаг перед запуском. Нам хотелось бы сохранить состояние модели после обучения, а поскольку мы пока хотим держать модель в оперативной памяти, то можем сохранить ее структуру в JSON-файле, а полученные в результате обучения веса — в другом файле для дальнейшего восстановления (листинг 7.16).

Листинг 7.16. Сохраняем результаты нашего тяжкого труда

```
>>> model_structure = model.to_json()
>>> with open("cnn_model.json", "w") as json_file:
...     json_file.write(model_structure)
>>> model.save_weights("cnn_weights.h5")
```

Обратите внимание, что эта команда не сохраняет веса сети — только структуру

Сохраняем обученную модель, чтобы не потерять ее!

Теперь наша обученная модель сохранена на диск. Если она сойдется, не придется обучать ее снова.

В Keras есть также несколько чрезвычайно удобных функций обратного вызова для этапа обучения, передаваемых в метод `fit` в виде поименованных аргументов. Например, `checkpointing`, итеративно сохраняющий модель только в том случае, если точность повысилась или потери уменьшились; или `EarlyStopping`, рано останавливающая обучение в случае, если модель более не совершенствуется относительно указанной метрики. Что лучше всего — они реализовали функцию обратного вызова TensorBoard. TensorBoard работает только при использовании TensorFlow в качестве прикладной части, но обеспечивает потрясающую глубину анализа моделей и незаменима при поиске причин проблем и тонкой настройке. А теперь приступим к обучению! Выполнение вышеупомянутых шагов `compile` и `fit` должно привести к выводу следующего:

```
Using TensorFlow backend.
Loading data...
25000 train sequences
25000 test sequences
```

```
Pad sequences (samples x time)
x_train shape: (25000, 400)
x_test shape: (25000, 400)
Build model...
Train on 20000 samples, validate on 5000 samples
Epoch 1/2 [==============================] - 417s - loss: 0.3756 -
acc: 0.8248 - val_loss: 0.3531 - val_acc: 0.8390
Epoch 2/2 [==============================] - 330s - loss: 0.2409 -
acc: 0.9018 - val_loss: 0.2767 - val_acc: 0.8840
```

Итоговые потери и точность могут немного отличаться — побочный эффект случайного выбора начальных весов для всех нейронов. Преодолеть этот эффект и создать воспроизводимый конвейер можно путем передачи генератору случайных чисел начального значения. Это обеспечивает выбор одних и тех же значений для начальных случайных весов, что удобно для отладки и тонкой настройки модели. Учтите только, что сама начальная точка может заводить модель в *локальный минимум* или даже не давать ей сходиться, так что мы рекомендуем пробовать несколько различных начальных значений.

Для задания начального значения генератора случайных чисел добавьте следующие две строки кода перед описанием модели. Передаваемое в качестве аргумента в функцию `seed` целочисленное значение неважно, но если оно одинаковое, то и начальные значения весов модели будут одинаковыми маленькими значениями:

```
>>> import numpy as np
>>> np.random.seed(1337)
```

Никаких явных признаков переобучения не заметно: точность улучшилась как для тренировочного, так и проверочного набора данных. Можете продолжить выполнение модели еще на одну-две эпохи и посмотреть, удастся ли улучшить показатели без переобучения. Модель Keras может продолжить обучение с текущего момента, если она все еще находится в оперативной памяти или была загружена из файла. Просто вызовите еще раз метод `fit` (можете поменять выборку или не менять), и обучение продолжится с этого последнего состояния.

СОВЕТ

> Явный признак переобучения — то, что потери продолжают падать в фазе обучения, а значение val_loss в конце эпохи начинает расти по сравнению с предыдущей эпохой. Ключевым в создании хорошей модели является нахождение золотой середины — места, где кривая потерь на этапе проверки начинает загибаться обратно.

Отлично. Готово. Резюмируем, что мы только что сделали.

Мы описали и скомпилировали модель в начальное необученное состояние. Затем мы вызвали метод `fit` для усвоения весов фильтров и получения в итоге полносвязной упреждающей сети и весов всех 250 отдельных фильтров путем обратного распространения полученной для каждого из примеров ошибки в самое начало цепочки.

Мерилом наших успехов служит функция потерь, в нашем случае `binary_crossentropy`. Keras возвращает для каждого пакета метрику расстояния от имеющейся для данного примера метки. Точность отражает «процент правильных предсказаний». Эта метрика весьма наглядна, но может вводить в заблуждение, особенно в случае несимметричного набора данных. Представьте себе, что у нас есть 100 примеров: 99 положительных и лишь один, который должен предсказываться как отрицательный. Если модель предскажет все 100 примеров как положительные, даже не глядя на данные, точность окажется 99 % — не слишком пригодно для обобщения. Метрики `val_loss` и `val_acc` отражают одно и то же на следующем тестовом наборе данных:

```
>>> validation_data=(x_test, y_test)
```

Проверочные примеры никогда не демонстрируются сети при обучении. Они передаются только для выполнения предсказаний и вычисления на основе этих предсказаний метрик. Для этих примеров не производится обратное распространение ошибки. Благодаря этому можно выяснить, насколько хорошо модель производит обобщение на новые, настоящие данные.

Мы обучили модель. Магия свершилась. Машина сообщила вам, что все рассчитала. Вы этому поверили. Ну и что? Теперь извлечем из всего этого какую-нибудь пользу.

7.4.7. Применение модели в конвейере

После обучения модели можно передать ей новые примеры и узнать, что сеть о них думает. Этими примерами может быть поступившее вашему боту сообщение в чате или твит. В нашем случае им будет выдуманный пример.

Прежде всего загрузим нашу сохраненную модель, если она уже не находится в оперативной памяти, как показано в листинге 7.17.

Листинг 7.17. Загрузка сохраненной модели

```
>>> from keras.models import model_from_json
>>> with open("cnn_model.json", "r") as json_file:
...     json_string = json_file.read()
>>> model = model_from_json(json_string)

>>> model.load_weights('cnn_weights.h5')
```

Создадим предложение с явно отрицательной тональностью и посмотрим, что скажет насчет него сеть (листинг 7.18).

Листинг 7.18. Тестовый пример

```
>>> sample_1 = "I hate that the dismal weather had me down for so long,
➥ when will it break! Ugh, when does happiness return? The sun is blinding
➥ and the puffy clouds are too thin. I can't wait for the weekend."
```

При наличии предобученной модели проверка нового примера не занимает много времени. Конечно, все равно приходится произвести тысячи и тысячи вычислений, но, чтобы получить результат для каждого из примеров, требуется лишь один прямой проход и никакого обратного распространения ошибки (листинг 7.19).

Листинг 7.19. Предсказание

```
>>> vec_list = tokenize_and_vectorize([(1, sample_1)])

>>> test_vec_list = pad_trunc(vec_list, maxlen)

>>> test_vec = np.reshape(test_vec_list, (len(test_vec_list), maxlen,\
...     embedding_dims))
>>> model.predict(test_vec)
array([[ 0.12459087]], dtype=float32)
```

Мы передаем фиктивное значение в качестве первого элемента кортежа, поскольку это требуется нашей вспомогательной функции, исходя из способа обработки данных. Сеть этого значения не увидит, так что оно может быть каким угодно

Процедура токенизации возвращает список данных (в данном случае длины 1)

Метод `predict` Keras возвращает необработанный выходной сигнал последнего слоя сети. В данном случае у нас один нейрон, а поскольку последний слой представляет собой сигма-функцию, то результат находится между 0 и 1.

Метод `predict_classes` возвращает ожидаемые 0 или 1. При многоклассовой классификации последний слой сети обычно представляет собой многомерную логистическую функцию, а выходные сигналы всех узлов — вероятности (с точки зрения сети) того, что этот узел является правильным ответом. Метод `predict_classes` в этом случае возвращает узел, которому соответствует наибольшее значение вероятности.

Но вернемся к нашему примеру:

```
>>> model.predict_classes(test_vec)
array([[0]], dtype=int32)
```

Разумеется, отрицательная тональность.

Предложение, включающее слова *happiness* («счастье»), *sun* («солнце»), *puffy* («пушистый») и *clouds* («облака»), не обязательно полно положительных эмоций. Как и тональность предложения с *dismal* («мрачный»), *break* и *down* («разбить») не обязательно отрицательная. Но с помощью обученной нейронной сети можно выяснить его основные закономерности и сделать выводы из данных без задания заранее каких-либо правил.

7.4.8. Что дальше?

Во введении мы говорили о важности CNN для обработки изображений. Один важный нюанс, на котором мы не акцентировали внимание: способность сетей обрабатывать *каналы* информации. В черно-белом двумерном изображении присутствует

один канал. Каждая точка данных представляет собой значение яркости (по серой шкале) соответствующего пиксела, так что данные будут двумерными. В случае же цветного — входной сигнал по-прежнему представляет собой яркость пиксела, но теперь он разбит на красный, зеленый и синий компоненты. Передаваемые в сеть входные данные становятся трехмерным тензором, а вслед за ним становятся трехмерными и фильтры, по-прежнему размером 3 × 3 или 5 × 5 на двумерной плоскости, но теперь уже глубиной в три слоя, в результате чего получаются фильтры, например, три пиксела в ширину × три пиксела в высоту × три канала в глубину, что ведет к интересным способам применениям в сфере обработки естественного языка.

Наши входные данные сети представляли собой ряд слов в виде выстроенных друг возле друга векторов 400 (maxlen) слов в ширину × 300 слов в длину, причем мы использовали для векторов слов вложения Word2vec. Но, как вы видели в предыдущих главах, генерировать вложения слов можно по-разному. Если выбрать несколько и ограничить их одинаковым количеством элементов, можно расположить их аналогично *каналам* изображения — интересный способ добавления информации в сеть, особенно если вложения поступают из различных источников. Расположение подобным образом различных вложений слов не всегда оправдывает возросшее время обучения вследствие увеличения сложности модели. Но теперь вы видите, почему мы начали с аналогий из сферы обработки изображений. Впрочем, эта аналогия нарушается, если осознать, что независимые от вложений слов измерения не коррелируют друг с другом так, как цветовые каналы изображения, так что здесь может быть все иначе.

Мы вкратце поговорили про выходной сигнал сверточных слоев (перед упреждающим слоем). Это *семантическое представление* играет важную роль. Во многих отношениях оно является числовым представлением идеи и нюансов входного текста. В нашем конкретном случае оно выступает числовым представлением идеи и нюансов через призму анализа тональностей, ведь все производимое обучение было лишь откликом на маркировку примера (отрицательная/положительная тональность). Сгенерированный и классифицированный в результате обучения на маркированном для другой конкретной темы наборе вектор будет содержать совсем другую информацию. Обычно промежуточные векторы непосредственно из сверточной нейронной сети не используют, но в следующих главах мы покажем вам примеры других архитектур нейронных сетей, где информация из этих промежуточных векторов играет важную роль, а в некоторых случаях и является конечной целью.

Почему для задач классификации NLP выбирают CNN? Основная причина — эффективность. Как ни посмотри, из-за слоев субдискретизации и ограничений вследствие размеров фильтров (хотя при желании фильтры можно сделать большими) отбрасывается немало информации. Но это не значит, что такие модели бесполезны. Как вы видели, они способны эффективно определять и предсказывать тональность для относительно больших наборов данных. Хотя мы делали это на основе вложений Word2vec, сверточные нейронные сети способны работать и на базе гораздо менее «богатых» вложений без словаря для всего языка.

Что можно дальше делать с CNN? Многое зависит от имеющихся наборов данных, но получить модели с большими возможностями можно благодаря наращи-

ванию количества слоев субдискретизации и передачи выходного сигнала первого набора фильтров в качестве примера «изображения» во второй и т. д. Исследователи также обнаружили, что, если выполнять модель с фильтрами различного размера и конкатенировать выходные сигналы фильтров каждого размера в более длинный *вектор идеи* до передачи его в конце в упреждающую нейронную сеть, можно получить более точные результаты. Все двери перед вами распахнуты. Экспериментируйте и получайте от этого удовольствие.

Резюме

- Свертка представляет собой скольжение небольшого окна над чем-то бóльшим (метод сосредоточения внимания на подмножестве большего множества).
- Нейронные сети способны обрабатывать текст аналогично тому, как они обрабатывают и видят изображения.
- Внесение помех в процесс обучения с помощью дропаута действительно помогает.
- Тональности заключены не только в словах, но и в используемых паттернах.
- У нейронных сетей есть множество доступных для настройки гиперпараметров.

8 Нейронные сети с обратной связью: рекуррентные нейронные сети

В этой главе

- Моделирование памяти в нейронной сети.
- Построение рекуррентной нейронной сети (RNN).
- Подготовка данных для RNN.
- Обратное распространение ошибки во времени (BPTT).

В главе 7 продемонстрированы возможности анализа фрагмента или целого предложения с помощью сверточной нейронной сети, отслеживание соседних слов в предложении путем наложения на них фильтра разделяемых весов (выполнения свертки). Встречающиеся группами слова можно также обнаруживать в связке. Сеть также устойчива к небольшим смещениям позиций этих слов. В то же время встречающиеся по соседству понятия могут существенно влиять на сеть. Но если нужно охватить взглядом большую картину происходящего, учесть взаимосвязи за более длительный промежуток времени, окно, охватывающее больше 3–4 токенов из предложения? Как ввести в сеть понятие произошедших ранее событий? Память?

Для каждого тренировочного примера (или батча неупорядоченных примеров) и выходного сигнала (или пакета выходных сигналов) нейронной сети прямого распространения веса нейронной сети необходимо откорректировать для отдельных нейронов на основе метода обратного распространения ошибки. Это мы уже демонстрировали. Но результаты этапа обучения для следующего примера в основном не зависят от порядка входных данных. Сверточные нейронные сети стремятся захватить эти отношения порядка за счет захвата локальных взаимосвязей, но существует и другой способ.

В сверточной нейронной сети каждый тренировочный пример передается сети в виде сгруппированного набора токенов слов. Векторы слов сгруппированы в матрицу в форме (длина вектора слова × число слов в примере), как показано на рис. 8.1.

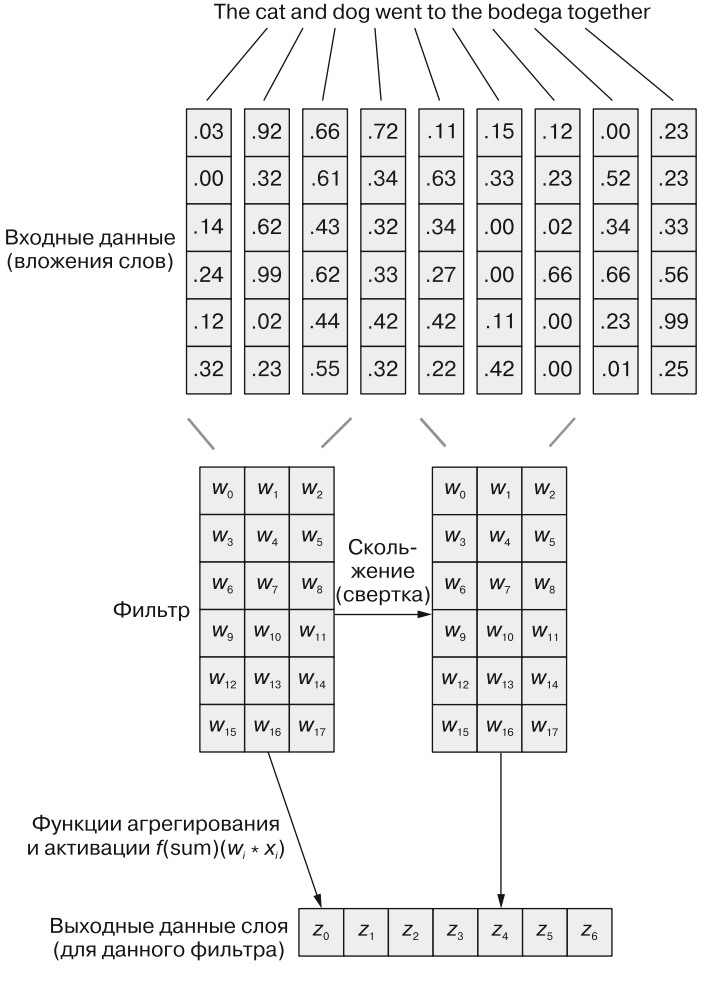


Рис. 8.1. Одномерная свертка с помощью вложений

Но эту последовательность векторов слов можно столь же легко передать и обычной нейронной сети прямого распространения из главы 5 (рис. 8.2), правда?

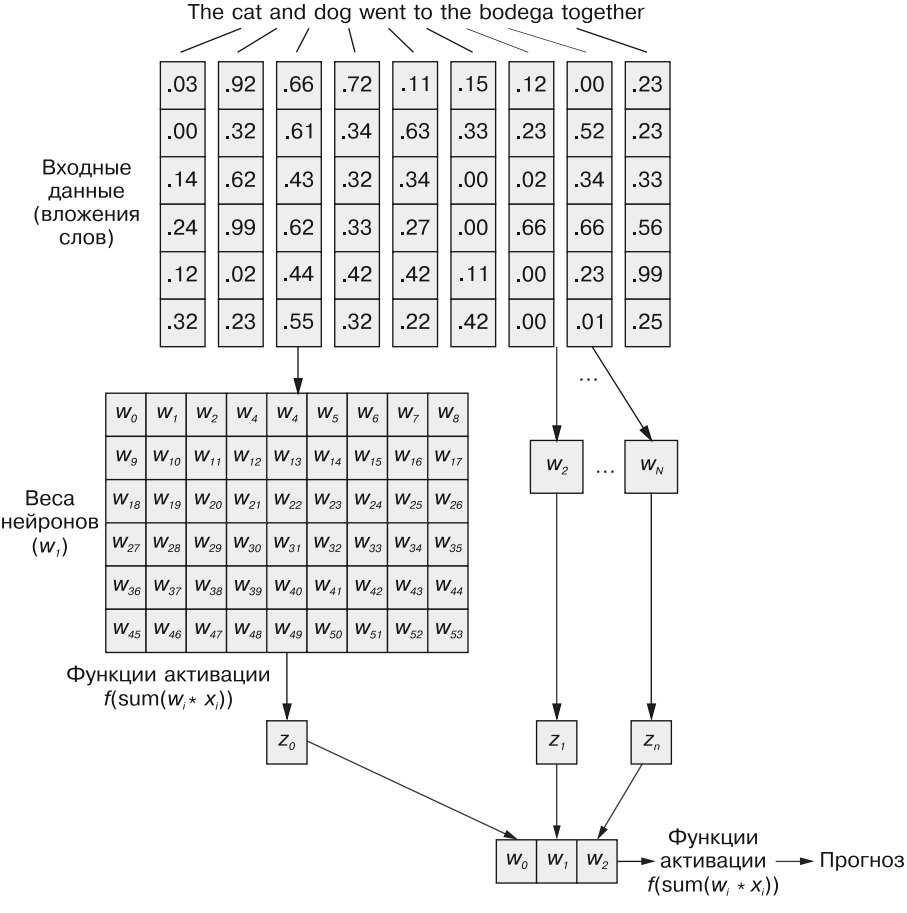


Рис. 8.2. Передача текста в нейронную сеть прямого распространения

Безусловно, это вполне работоспособная модель. При подобном способе передачи входных данных нейронная сеть прямого распространения сможет реагировать на совместные вхождения токенов, что нам и нужно. Но она будет при этом реагировать на все совместные вхождения одинаково, независимо от того, разделяет ли их длинный текст, или они находятся рядом друг с другом. Кроме того, нейронные сети прямого распространения, как и CNN, плохо умеют обрабатывать документы переменной длины. Они не способны обработать текст в конце документа, если он выходит за пределы ширины сети.

Лучше всего нейронные сети прямого распространения проявляют себя в моделировании взаимосвязи выборки данных в целом с соответствующей ей меткой. Слова в начале и в конце предложения ровно так же влияют на выходной сигнал, как и посередине, невзирая на то что вряд ли они семантически связаны друг с другом.

Подобная однородность (равномерность влияния) явно может вызывать проблемы в случае, например, токенов резкого отрицания и модификаторов (прилагательных и наречий), таких как *«нет»* или *«хороший»*. В нейронной сети прямого распространения выражающие отрицание слова влияют на смысл всех слов в предложении, даже сильно удаленных от места, на которое они должны влиять на самом деле.

Одномерные свертки — способ решения проблем с этими взаимосвязями между токенами путем анализа нескольких слов через *окна*. Обсуждавшиеся в главе 7 слои субдискретизации специально предназначены для учета небольших изменений порядка слов. В этой главе мы рассмотрим другой подход, благодаря которому сможем сделать первый шаг к понятию *памяти* нейронной сети. Вместо того чтобы разбирать язык как большую порцию данных, мы начнем рассматривать его последовательное формирование, токен за токеном, с ходом *времени*.

8.1. Запоминание в нейронных сетях

Конечно, слова в предложении редко бывают совершенно независимыми друг от друга; их вхождения влияют или подвергаются влиянию вхождений других слов в документе. Например: *The stolen car sped into the arena* и *The clown car sped into the arena*.

У вас могут возникнуть совершенно различные впечатления от этих двух предложений, когда вы дочитаете до конца. Конструкция фразы в них одинакова: прилагательное, существительное, глагол и предложный оборот. Но замена прилагательного в них радикальным образом меняет суть происходящего с точки зрения читателя.

Как смоделировать подобную взаимосвязь? Как понять, что *arena* и даже *sped* могут иметь немного разные коннотации, если перед ними в предложении есть прилагательное, не являющееся прямым определением ни одного из них?

Если бы существовал способ *запоминать* произошедшее моментом ранее (особенно помнить на шаге $t + 1$ произошедшее на шаге t), можно было бы выявлять закономерности, возникающие при появлении определенных токенов в связанных с другими токенами последовательности закономерностях. *Рекуррентные нейронные сети* (RNN) как раз делают возможным запоминание нейронной сетью прошлых слов последовательности.

Как вы видите на рис. 8.3, отдельный рекуррентный нейрон из скрытого слоя добавляет в сеть рекуррентный цикл для «повторного использования» выходного сигнала скрытого слоя для момента t. Выходной сигнал для момента t прибавляется к следующему входному сигналу для момента $t + 1$. В результате обработки сетью этого нового входного сигнала на временном шаге $t + 1$ получается выходной сигнал скрытого слоя для момента $t + 1$. Этот выходной сигнал для момента времени $t + 1$ далее повторно используется сетью и включается во входной сигнал на временном шаге $t + 2$ и т. д.[1]

[1] В финансовом деле, динамике и управлении с помощью обратной связи это часто называется моделью авторегрессии скользящего среднего (autoregressive moving average, ARMA): https://ru.wikipedia.org/wiki/ Модель_авторегрессии_—_скользящего_среднего.

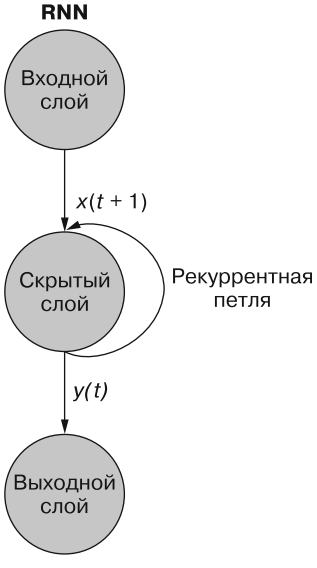


Рис. 8.3. Рекуррентная нейронная сеть

Хотя идея воздействия на состояние сквозь время выглядит немного запутанной, основная концепция проста. Результаты каждого сигнала на входе обычной нейронной сети прямого распространения на временном шаге t используются в качестве дополнительного входного сигнала вместе со следующим фрагментом подаваемых на вход сети данных на временном шаге $t + 1$. Сеть получает информацию не только о происходящем сейчас, но и о происходившем ранее.

ВАЖНО

В этой и следующей главах большая часть нашего обсуждения происходит на языке временных шагов. Это вовсе не то же самое, что отдельные примеры данных. Речь идет о разбиении одного примера данных на меньшие порции, отражающие изменения во времени. Этот отдельный пример данных все равно представляет собой фрагмент текста, скажем короткий отзыв о фильме или твит. Как и ранее, мы токенизируем предложение. Но вместо того, чтобы отправлять токены в сеть все сразу, мы передаем их по одному. *Эта схема отличается от передачи нескольких новых примеров документов*. Токены при этом остаются частью *одного* примера данных, которому соответствует *одна* метка.

t можно считать индексом последовательности токенов. Так, $t = 0$ — первый токен в документе, а $t + 1$ — следующий. Токены в порядке следования их в документе служат входными сигналами на каждом из *временных (токенных) шагов*. Причем токены не обязательно должны быть словами, отдельные символы также допустимы. Подача примера данных в сеть разбивается на *подшаги* — ввод в сеть отдельных токенов.

На протяжении всей этой книги мы будем обозначать текущий временной шаг t, а следующий временной шаг — $t + 1$.

Рекуррентную нейронную сеть можно визуализировать так, как показано на рис. 8.3: круги соответствуют целым *слоям* нейронной сети прямого распространения, состоящим из одного или нескольких нейронов. Выходной сигнал *скрытого слоя* выдается сетью как обычно, но затем поступает обратно в качестве *своего* же (скрытого слоя) входного сигнала вместе с обычными входными данными следующего временного шага. На схеме этот цикл обратной связи изображен в виде дуги, ведущей из выхода слоя обратно на вход.

Более простой (и чаще используемый) способ иллюстрации этого процесса — с использованием *развертывания* сети. Рисунок 8.4 демонстрирует сеть «вверх ногами» с двумя развертками временно́й переменной *(t)* — слоями для шагов $t + 1$ и $t + 2$.

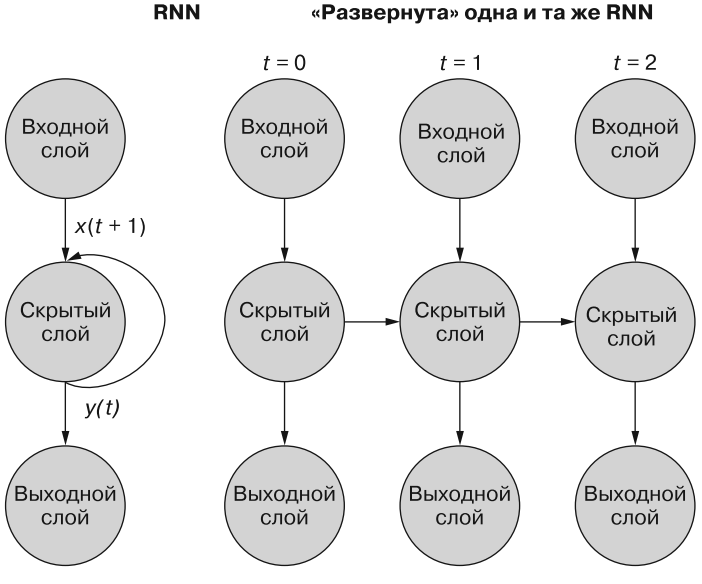


Рис. 8.4. Рекуррентная нейронная сеть

Каждому из временных шагов соответствует развернутая версия той же нейронной сети в виде столбца нейронов. Это все равно, что смотреть сценарий или отдельные видеокадры нейронной сети в каждый момент времени. Сеть справа представляет собой *будущую* версию сети слева. Выходной сигнал скрытого слоя в момент времени *(t)* подается снова на вход скрытого слоя вместе с входными данными для следующего временного шага ($t + 1$) справа. И еще раз. На схеме показаны две итерации этого развертывания, всего три столбца нейронов для $t = 0$, $t = 1$ и $t = 2$.

Все вертикальные маршруты на этой схеме полностью аналогичны, в них показаны одни и те же нейроны. Они отражают одну и ту же нейронную сеть в разные моменты времени. Такое наглядное представление удобно при демонстрации движения информации по сети в прямом и *обратном* направлении во время обратного распространения ошибки. Но помните, глядя на эти три развернутые сети: они

представляют собой различные моментальные снимки одной и той же сети с тем же набором весов.

Изучим внимательнее исходное представление рекуррентной нейронной сети до ее развертывания и покажем взаимосвязи входных сигналов и весов. Отдельные слои этой RNN выглядят так, как показано на рис. 8.5 и 8.6.

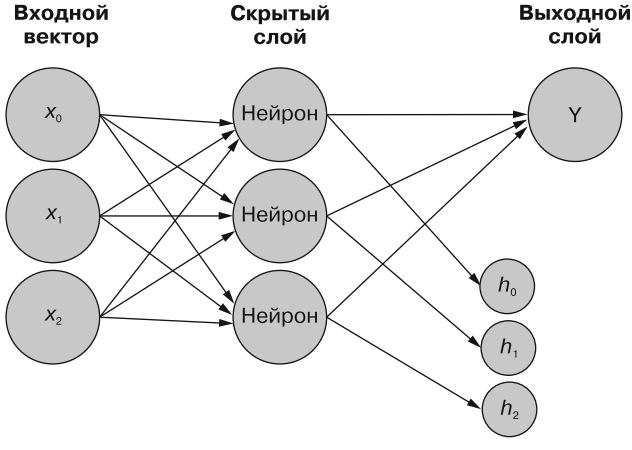


Рис. 8.5. Подробная схема рекуррентной нейронной сети в момент времени $t = 0$

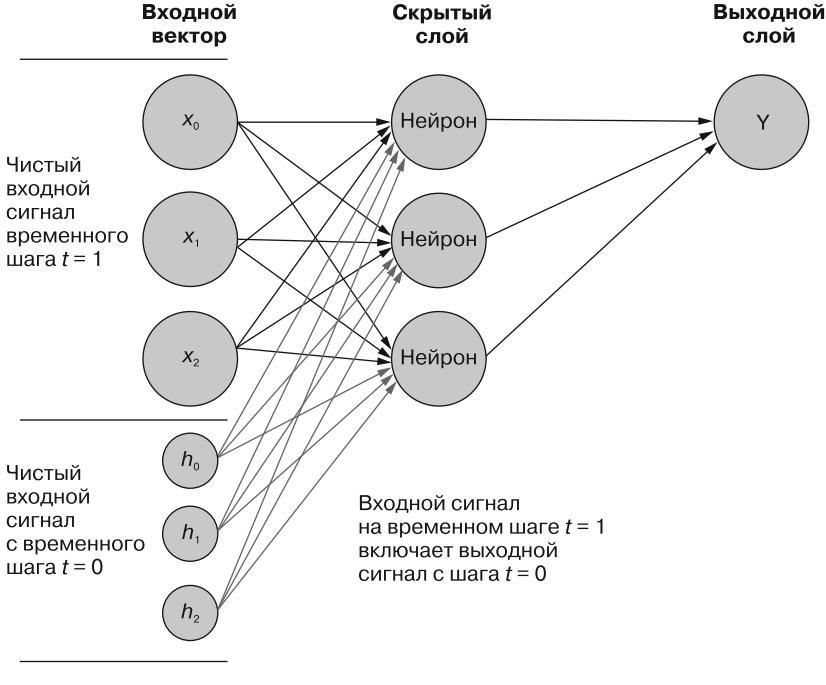


Рис. 8.6. Подробная схема рекуррентной нейронной сети в момент времени $t = 1$

У всех нейронов скрытого состояния есть по набору весов, применяемых к каждому из элементов каждого из входных векторов, как в обычной сети прямого распространения. Но в этой схеме появился и дополнительный набор обучаемых весов, которые применяются к выходным сигналам скрытых нейронов из предыдущего временного шага. Сеть путем обучения подбирает подходящие веса (важность) предыдущих событий при вводе последовательности токен за токеном.

СОВЕТ

У первого входного сигнала в последовательности нет «прошлого», так что скрытое состояние на шаге $t = 0$ получает нулевой входной сигнал от себя же с шага $t - 1$. Можно также для «заполнения» начального значения состояния сначала передать в сеть взаимосвязанные, но отдельные примеры данных, один за другим. Итоговый выходной сигнал каждого примера используется во входном сигнале $t = 0$ следующего примера данных. В посвященном сохранению состояния разделе в конце данной главы мы расскажем вам, как сохранять больше информации из набора данных с помощью альтернативных методик заполнения.

Вернемся к данным: представьте, что у вас есть набор документов, каждый из которых представляет собой маркированный пример. И вместо того, чтобы для каждого выборочного примера передавать набор векторов слов целиком в сверточную нейронную сеть, как в предыдущей главе (рис. 8.7), мы передаем пример данных в RNN по одному токену (рис. 8.8).

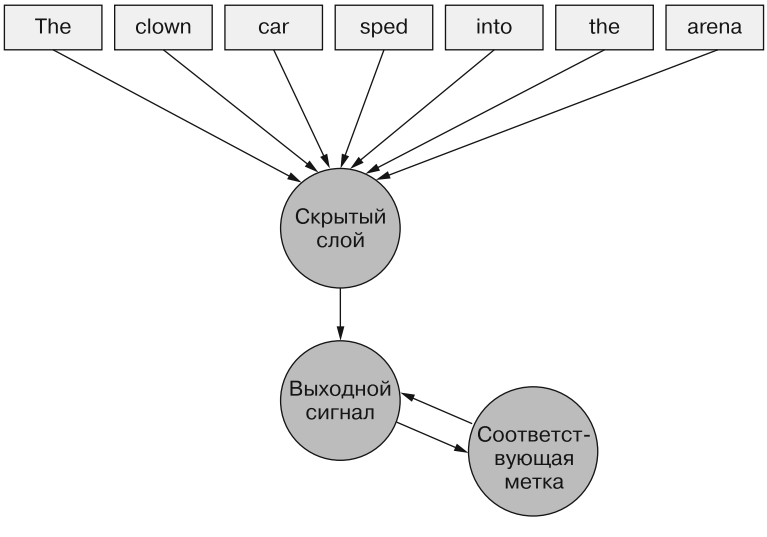


Рис. 8.7. Ввод данных в сверточную сеть

Мы передаем вектор слов для первого токена и получаем выходной сигнал нашей рекуррентной нейронной сети. Затем передаем второй токен, а вместе с ним — выходной

сигнал от первого! После этого передаем третий токен вместе с выходным сигналом от второго! И так далее. Теперь в нашей нейронной сети существуют понятия «до» и «после», причины и следствия, некое, пусть и расплывчатое, представление о времени (см. рис. 8.8).

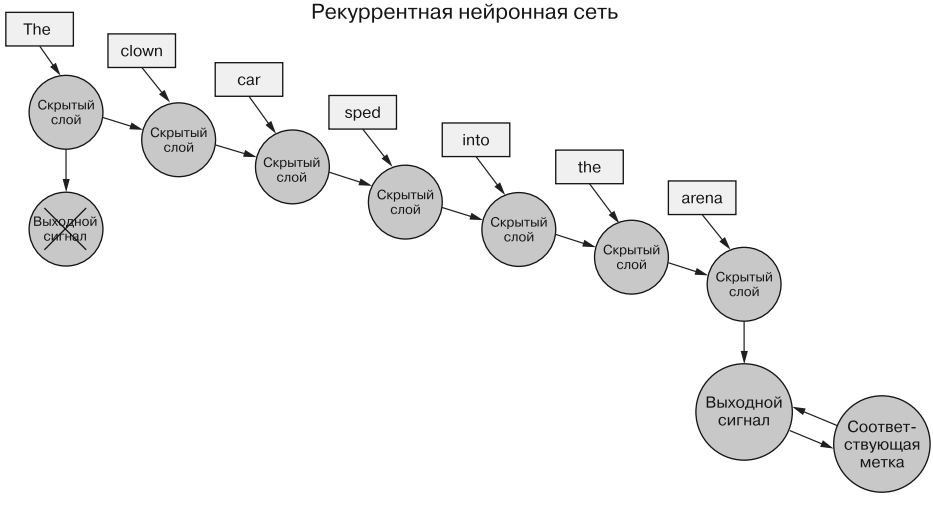


Рис. 8.8. Ввод данных в рекуррентную сеть

Теперь наша сеть уже кое-что запоминает! Ну, в известной мере. Осталось выяснить еще несколько вещей. Во-первых, как может происходить обратное распространение ошибки в подобной структуре?

8.1.1. Обратное распространение ошибки во времени

Во всех обсуждавшихся выше сетях существовала искомая метка (целевая переменная), и RNN не исключение. Но у нас отсутствует понятие метки для каждого токена, а есть только одна метка для всех токенов каждого из примеров текста. У нас имеются только метки для примеров документов.

...и этого достаточно.

Айседора Дункан

СОВЕТ

Мы говорим о токенах как входных данных для сети на каждом из временных шагов, но рекуррентные нейронные сети могут так же работать с любыми данными временных рядов. Токены могут быть любыми, дискретными или непрерывными: показания метеостанции, ноты, символы в предложении и т. п.

Здесь мы сначала сравниваем выходной сигнал сети на последнем временном шаге с меткой. Именно это мы и будем (пока что) называть *ошибкой*, а именно ошибку наша сеть и пытается минимизировать. Но есть небольшое отличие от предыдущих глав. Заданный пример данных разбивается на меньшие части, подаваемые в нейронную сеть последовательно. Однако вместо того, чтобы непосредственно использовать получаемый для каждого из этих «подпримеров» выходной сигнал, мы отправляем его обратно в сеть.

Нас интересует пока что только итоговый выходной сигнал. Каждый из токенов последовательности подается в сеть, и на основе выходного сигнала последнего временного шага (токена) вычисляются потери (рис. 8.9).

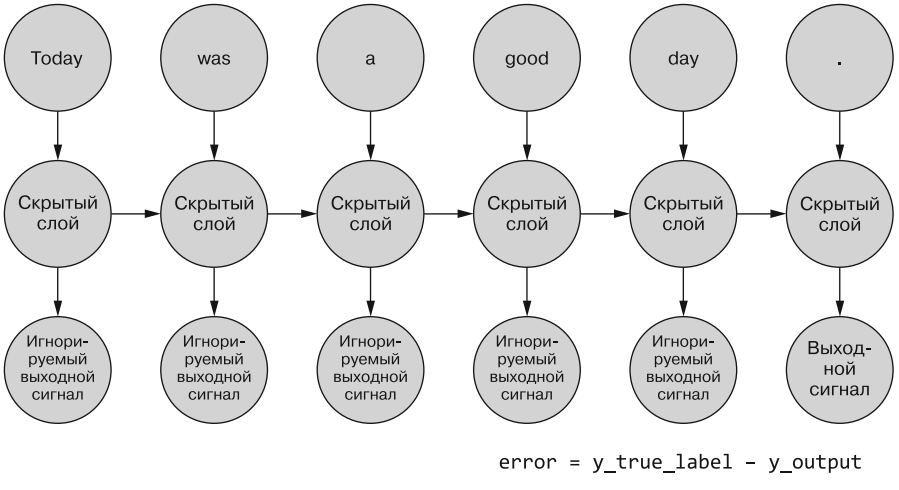


Рис. 8.9. Нас интересует только последний выходной сигнал

Необходимо определить при наличии ошибки для заданного примера, какие веса обновить и насколько. В главе 5 мы рассказали, как производить обратное распространение ошибки по обычной сети. И мы знаем, что величина корректировки веса зависит от его (этого веса) вклада в ошибку. Мы можем подавать по токену из выборочной последовательности на вход сети, вычисляя на основе ее выходного сигнала ошибку для предыдущего временного шага. Тут-то идея обратного распространения ошибки во времени, похоже, все и запутывает.

Впрочем, можно просто рассматривать это как процесс с привязкой по времени. На каждом временном шаге токены, начиная с первого при $t = 0$, подаются по одному на вход расположенного впереди скрытого нейрона — следующий столбец на рис. 8.9. При этом сеть развертывается, раскрывая следующий столбец сети, уже готовый для получения очередного токена в последовательности. Скрытые нейроны развертываются по одному, подобно музыкальной шкатулке или механическому пианино. В конце концов, когда в сеть будут поданы все элементы примеров, развертывать больше будет нечего и мы получим на выходе итоговую метку для интересующей нас целевой переменной, которую можно использовать для вычисления ошибки

и корректировки весов. Мы только что прошли весь путь по графу вычислений для этой *развернутой сети* (unrolled net).

Пока что мы считаем входные данные в целом статическими. Можно проследить по всему графу, какой входной сигнал поступает в какой нейрон. А раз мы знаем, как срабатывает какой нейрон, то можем *распространить ошибку* обратно по цепочке, по тому же пути, точно так же, как и в случае обычной нейронной сети прямого распространения.

Для обратного распространения ошибки на предыдущий слой мы воспользуемся цепным правилом. Вместо предыдущего слоя мы распространим ошибку на тот же слой *в прошлом*, как если бы все развернутые варианты сети были различны (рис. 8.10). Математика расчетов при этом не меняется.

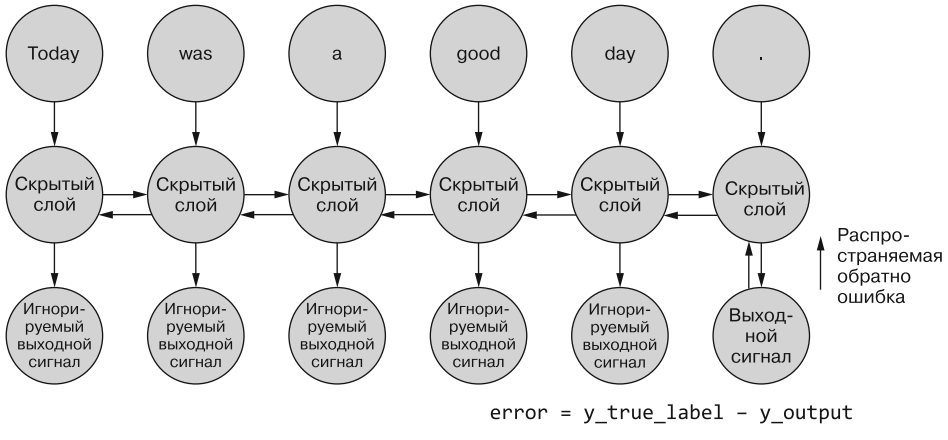


Рис. 8.10. Обратное распространение ошибки во времени

Ошибка с последнего шага распространяется обратно. Вычисляется градиент более раннего временного шага относительно более нового. После вычисления всех отдельных градиентов по токенам, вплоть до шага $t = 0$ для данного примера, изменения агрегируются и применяются к одному набору весов.

РЕЗЮМЕ

- Разбиваем каждый пример данных на токены.
- Передаем токены по одному в нейронную сеть прямого распространения.
- Возвращаем выходной сигнал каждого из временных шагов на вход того же слоя вместе с входным сигналом для следующего временного шага.
- Сравниваем выходной сигнал последнего временного шага с меткой.
- Производим обратное распространение ошибки через весь граф, аж до первого входного сигнала на временном шаге 0.

8.1.2. Когда что обновлять

Мы превратили нашу странную RNN в нечто похожее на обычную нейронную сеть прямого распространения, так что обновление весов не должно вызвать затруднений. Впрочем, есть один нюанс. Хитрость в том, что веса обновляются вовсе не в другой ветке нейронной сети. Каждая ветка представляет собой *ту же* сеть в другие моменты времени. Веса для каждого временного шага *одни и те же* (см. рис. 8.10).

Простое решение этой проблемы — вычисление поправок для весов на каждом из временных шагов с отсрочкой обновления. В сети прямого распространения все обновления весов вычисляются сразу после вычисления всех градиентов для конкретного входного сигнала. И здесь точно так же, но обновления откладываются, пока мы не доберемся до начального (нулевого) временного шага для конкретного входного примера данных.

В основе расчета градиента должны лежать значения весов, при которых они внесли данный вклад в ошибку. Вот и самая ошеломительная часть: вес на временном шаге t внес некий вклад в ошибку. И *тот же* вес получает другой входной сигнал на временном шаге $t + 1$, а значит, вносит уже другой вклад в ошибку.

Можно вычислять различные изменения весов на каждом из временных шагов, суммировать их и затем применять сгруппированные изменения к весам скрытого слоя в качестве последнего шага этапа обучения.

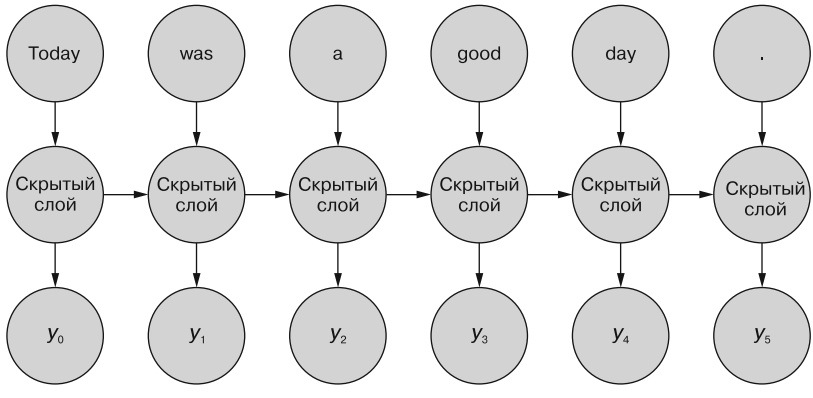
СОВЕТ

> Во всех этих примерах передается отдельный тренировочный пример данных для *прямого прохода*, а затем производится обратное распространение ошибки. Как и в любой нейронной сети, этот прямой проход по сети можно производить или после каждого тренировочного примера, или в виде батчей. Оказывается, что у пакетного подхода есть и другие преимущества, помимо быстродействия. Но пока что мы будем рассматривать эти процессы с точки зрения отдельных примеров данных, отдельных предложений или документов.

Настоящее волшебство. При обратном распространении ошибки во времени может производиться корректировка отдельного веса в одну сторону на временном шаге t (в зависимости от его реакции на входной сигнал на временном шаге t), а затем в другую сторону на временном шаге $t - 1$ (в соответствии с тем, как он отреагировал на входной сигнал на временном шаге $t - 1$) для одного примера данных! Помните, что нейронные сети в целом основаны на минимизации функции потерь вне зависимости от сложности промежуточных шагов. В совокупности сеть оптимизирует эту сложную функцию. Поскольку обновление веса применяется для примера данных однократно, то сеть (если она вообще сходится, конечно) в итоге останавливается на наиболее оптимальном в этом смысле весе для конкретного входного сигнала и конкретного нейрона.

Результаты предыдущих шагов все же важны

Иногда оказывается важна вся последовательность значений, генерируемая на всех промежуточных временных шагах. В главе 9 мы приведем примеры ситуаций, в которых выходной сигнал конкретного временного шага t ничуть не менее важен, чем выходной сигнал последнего временного шага. На рис. 8.11 приведен способ сбора данных об ошибке для любого временного шага и обратного ее распространения для корректировки всех весов сети.




```
error = sum([y_true_label[i] - y[i] for i in range(6)])
```

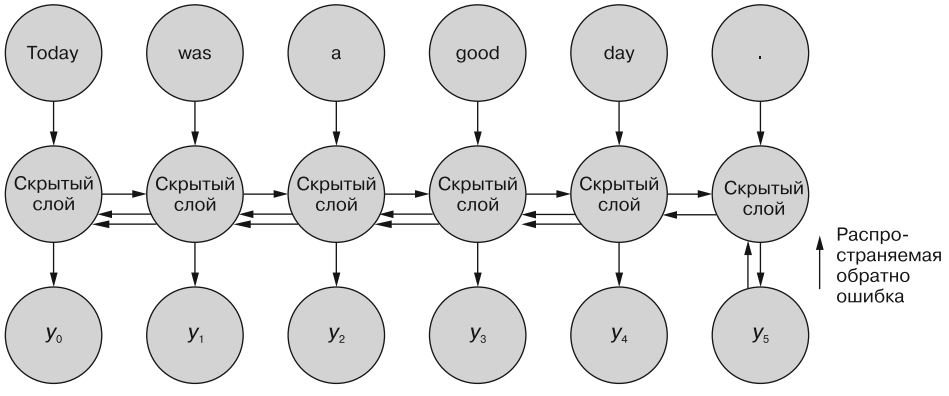
Рис. 8.11. Здесь играют свою роль все выходные сигналы

Этот процесс напоминает обычное обратное распространение ошибки во времени для n временных шагов. В данном случае мы распространяем обратно ошибку из нескольких источников одновременно. Но, как и в первом примере, корректировки весов носят аддитивный характер. Ошибка распространяется с последнего временного шага в начале до первого с суммированием изменений каждого из весов. Затем то же самое происходит с ошибкой, вычисленной на предпоследнем временном шаге с суммированием всех изменений вплоть до $t = 0$. Этот процесс повторяется, пока мы не дойдем до нулевого временного шага с обратным распространением ошибки для него так, как будто он единственный. Затем суммарные изменения применяются все сразу к соответствующему скрытому слою.

На рис. 8.12 видно, как ошибка распространяется от каждого выходного сигнала обратно аж до $t = 0$, затем агрегируется перед итоговой корректировкой весов. Это основная идея данного раздела. Как и в случае обычной нейронной сети прямого распространения, веса обновляются *только после* вычисления предлагаемого изменения весов для всего шага обратного распространения ошибки для данного входного сигнала (или набора входных сигналов). В случае RNN обратное распространение ошибки включает обновления вплоть до момента времени $t = 0$.

Обновление весов ранее внесло бы искажения в расчеты градиентов при обратных распространениях ошибок в более ранние моменты времени. Как вы помните, градиенты вычисляются относительно конкретного веса. Если обновить этот вес

слишком рано, скажем на временном шаге t, то при вычислении градиента на временном шаге $t - 1$ значение веса (напомним, что это *та же* позиция веса в сети) изменится. И при вычислении градиента на основе входного сигнала с временного шага $t - 1$ расчеты окажутся искаженными. Фактически при этом вес будет штрафоваться (или вознаграждаться) за то, в чем «не виноват»!




```
error = sum([y_true_label[i] - y[i] for i in range(6)])
```

Рис. 8.12. Несколько входных сигналов и обратное распространение ошибки во времени

8.1.3. Краткое резюме

Чего мы добились? Мы сегментировали примеры данных на токены, а затем по очереди отправили их на вход нейронной сети прямого распространения. Причем на вход вместе с каждым токеном подавался также выходной сигнал с предыдущего временного шага. На временном шаге 0 на вход подается исходный токен вместе с 0, в результате чего получается нулевой вектор, поскольку никакого предыдущего выходного сигнала нет. Ошибкой служит разность между выходным сигналом сети от последнего токена и ожидаемой меткой. Далее мы распространяли эту ошибку обратно на веса сети, обратно во времени. После этого мы агрегировали предлагаемые изменения весов и применяли к сети их все сразу.

Мы получили нейронную сеть прямого распространения, обладающую понятием памяти и примитивным инструментарием для поддержания памяти вхождений на данной временной шкале.

8.1.4. Всегда есть какой-нибудь подвох

Хотя число подбираемых весов (параметров) в нейронной сети прямого распространения может быть относительно невелико, из рис. 8.12 видно, как быстро могут расти требуемые для обучения RNN ресурсы, особенно для последовательностей более или менее значительной длины, скажем десять токенов. Чем больше токенов, тем дальше обратно во времени нужно распространять ошибки. Для каждого дополнительного

шага обратно по времени нужно вычислять дополнительные производные. Рекуррентные нейронные сети ничуть не менее эффективны, чем другие, но будьте готовы обогревать дом теплом от своего компьютера.

Но забудем про дополнительный обогрев дома, ведь у нашей нейронной сети появилась зачаточная память. Но здесь всплывает другая, еще более серьезная проблема, возникавшая и в нейронных сетях прямого распространения при повышении степени их глубины. Следствием *проблемы «исчезающего» градиента* является *проблема «взрывного» роста градиента*. Идея в том, что по мере углубления (повышения числа слоев) нейронной сети сигнал ошибки может расти или рассеиваться с каждым вычислением градиента.

Та же проблема возникает и в RNN, поскольку каждый шаг назад по времени является математическим эквивалентом обратного распространения ошибки на предыдущий слой нейронной сети прямого распространения. Но здесь ситуация еще хуже! Хотя как раз по этой причине глубина большинства нейронных сетей прямого распространения не превышает нескольких слоев, обрабатываются последовательности из пяти, десяти или даже сотен токенов. Добраться до дна сети глубиной в сотню слоев — непростая задача. Впрочем, эту ситуацию смягчает один фактор. Хотя по дороге к последнему набору весов градиент может «исчезнуть» или «взорваться», мы обновляем только один набор весов. И он не меняется от шага к шагу. Часть информации попадет по назначению, хотя вряд ли память окажется столь идеальной, как вы надеялись. Но не бойтесь, исследователи уже занимаются этой проблемой, и некоторые ее решения вы найдете в следующей главе.

Хватит депрессии. Взглянем на настоящую магию.

8.1.5. Рекуррентные нейронные сети и Keras

Начнем с того же набора данных и предварительной обработки, что и в предыдущей главе. Сначала мы загрузим набор данных, получим метки и перетасуем примеры данных. Затем токенизируем и векторизуем его снова с помощью модели Google Word2vec. Далее мы возьмем метки. И произведем разбиение в соотношении 80/20 на тренировочный и тестовый набор данных.

Сначала нужно импортировать все модули, необходимые для обработки данных и обучения рекуррентной сети, как показано в листинге 8.1.

Листинг 8.1. Импорт всего, что нужно

```
>>> import glob
>>> import os
>>> from random import shuffle
>>> from nltk.tokenize import TreebankWordTokenizer
>>> from nlpia.loaders import get_data
>>> word_vectors = get_data('wv')
```

Теперь можно сформировать процедуру предварительной обработки данных, которая приведет все данные в нужную форму, как показано в листинге 8.2.

Листинг 8.2. Процедура предварительной обработки данных

```
>>> def pre_process_data(filepath):
...     """
...     Загружаем положительные и отрицательные примеры данных из отдельных
...     каталогов, после чего перемешиваем их.
...     """
...     positive_path = os.path.join(filepath, 'pos')
...     negative_path = os.path.join(filepath, 'neg')
...     pos_label = 1
...     neg_label = 0
...     dataset = []
...     for filename in glob.glob(os.path.join(positive_path, '*.txt')):
...         with open(filename, 'r') as f:
...             dataset.append((pos_label, f.read()))
...     for filename in glob.glob(os.path.join(negative_path, '*.txt')):
...         with open(filename, 'r') as f:
...             dataset.append((neg_label, f.read()))
...     shuffle(dataset)
...     return dataset
```

Как и ранее, токенизатор и векторизатор можно объединить в одну функцию, как показано в листинге 8.3.

Листинг 8.3. Токенизатор + векторизатор данных

```
>>> def tokenize_and_vectorize(dataset):
...     tokenizer = TreebankWordTokenizer()
...     vectorized_data = []
...     for sample in dataset:
...         tokens = tokenizer.tokenize(sample[1])
...         sample_vecs = []
...         for token in tokens:
...             try:
...                 sample_vecs.append(word_vectors[token])
...             except KeyError:
...                 pass                          <-- В словаре Google w2v отсутствует
...         vectorized_data.append(sample_vecs)       соответствующий токен
...     return vectorized_data
```

Необходимо также извлечь (распаковать) целевую переменную на отдельные (но соответствующие) примеры данных, как показано в листинге 8.4.

Листинг 8.4. Распаковщик целевой переменной

```
>>> def collect_expected(dataset):
...     """ Вытаскиваем целевые значения из набора данных """
...     expected = []
...     for sample in dataset:
...         expected.append(sample[0])
...     return expected
```

Мы собрали все необходимые для предварительной обработки функции, осталось выполнить их на наших данных, как показано в листинге 8.5.

Листинг 8.5. Загрузка и подготовка данных

```
>>> dataset = pre_process_data('./aclimdb/train')
>>> vectorized_data = tokenize_and_vectorize(dataset)
>>> expected = collect_expected(dataset)
>>> split_point = int(len(vectorized_data) * .8)
>>> x_train = vectorized_data[:split_point]
>>> y_train = expected[:split_point]
>>> x_test = vectorized_data[split_point:]
>>> y_test = expected[split_point:]
```

Разбиваем на тренировочный и тестовый наборы данных в соотношении 80/20 (без какой-либо перетасовки)

Для этой модели мы воспользуемся теми же гиперпараметрами: 400 токенов на один пример данных, размер батча — 32. Длина векторов слов — 300 элементов, количество эпох — 2 (листинг 8.6).

Листинг 8.6. Задание значений параметров сети

```
>>> maxlen = 400
>>> batch_size = 32
>>> embedding_dims = 300
>>> epochs = 2
```

Далее нам необходимо опять дополнить и усечь примеры данных. Обычно в случае RNN это не требуется, поскольку они способны обрабатывать входные последовательности переменной длины. Но, как вы увидите в следующих нескольких шагах, данная модель требует одинаковой длины последовательностей (листинг 8.7).

Листинг 8.7. Загрузка тестовых и тренировочных данных

```
>>> import numpy as np

>>> x_train = pad_trunc(x_train, maxlen)
>>> x_test = pad_trunc(x_test, maxlen)

>>> x_train = np.reshape(x_train, (len(x_train), maxlen, embedding_dims))
>>> y_train = np.array(y_train)
>>> x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims))
>>> y_test = np.array(y_test)
```

Теперь, разобравшись с данными, можно и создать модель. Начнем опять с обычной `Sequential()` (многослойной) модели Keras, как показано в листинге 8.8.

Листинг 8.8. Инициализация «пустой» сети Keras

```
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Dropout, Flatten, SimpleRNN
>>> num_neurons = 50
>>> model = Sequential()
```

Затем, как и ранее, Keras как по волшебству берет на себя все сложности формирования нейронной сети: достаточно всего лишь добавить в сеть нужный рекуррентный слой, как показано в листинге 8.9.

Листинг 8.9. Добавление рекуррентного слоя

```
>>> model.add(SimpleRNN(
...     num_neurons, return_sequences=True,
...     input_shape=(maxlen, embedding_dims)))
```

Инфраструктура для передачи каждого из входных сигналов в простую RNN (не такая простая версия ожидает нас в следующей главе) и последующего сбора выходных сигналов в вектор готова. Поскольку длина наших последовательностей равна 400 токенам и используется 50 скрытых нейронов, то выходной сигнал из этого слоя будет представлять собой вектор длиной 400 элементов, каждый из которых будет вектором длиной 50 элементов, по одному выходному сигналу для каждого нейрона.

Обратите внимание на поименованный аргумент `return_sequences`. Он указывает нейронной сети возвращать значение на каждом временном шаге, в результате чего получаются 400 векторов, каждый длиной 50. Если аргумент `return_sequences` равен `False` (поведение Keras по умолчанию), то будет возвращен только один 50-мерный вектор.

Мы выбрали равное 50 количество нейронов для этого примера произвольно, в основном для сокращения времени вычислений. Можете поэкспериментировать с этим числом и посмотреть, как оно влияет на время вычислений и точность модели.

СОВЕТ

> Хорошее эмпирическое правило: модель должна быть не сложнее данных, на которых она обучается. Проще сказать, чем сделать, но теперь у вас есть разумное обоснование для подстройки параметров при экспериментах с набором данных. Более сложная модель будет *переобучаться* на тренировочном наборе и плохо обобщать данные. Слишком простая модель *недообучится* и мало что сможет сказать интересного про новые данные. Эту дискуссию часто называют выбором *компромисса между систематической ошибкой (смещением) и дисперсией*. О переобученной модели говорят, что у нее большая дисперсия и маленькая систематическая ошибка, а у недообученной — наоборот: маленькая дисперсия и большая систематическая ошибка; она всегда ошибается, но одинаково.

Обратите внимание, что мы опять выполнили усечение и дополнение данных, чтобы сравнить с примером CNN из предыдущей главы. Но при использовании RNN усечение и дополнение обычно не обязательны. Можно подавать на вход тренировочные данные различной длины и разворачивать сеть вплоть до конца входного сигнала. Keras делает это автоматически. Подвох в том, что размер выходного сигнала рекуррентного слоя будет меняться от временного шага к временному шагу в соответствии с входным сигналом. При входном сигнале четыре токена на выходе получится последовательность длиной четыре элемента. Последовательность из ста токенов даст на выходе последовательность из ста элементов. И вы не сможете передать эти данные в следующий слой, если он ожидает на входе унифицированные входные данные. Но существуют случаи, когда подобное поведение не только допустимо, но и предпочтительно. Однако вернемся к нашему классификатору (листинг 8.10).

Листинг 8.10. Добавление слоя дропаута

```
>>> model.add(Dropout(.2))

>>> model.add(Flatten())
>>> model.add(Dense(1, activation='sigmoid'))
```

Мы требуем, чтобы эта `SimpleRNN` возвращала полные предложения, но для предотвращения переобучения добавили слой дропаута для обнуления 20 % выбираемых случайным образом входных сигналов из каждого входного примера данных. И наконец, добавляем классификатор. В данном случае у нас есть один класс: «Да — положительная тональность — 1» или «Нет — отрицательная тональность — 0», так мы выбрали слой из одного нейрона (`Dense(1)`) и сигма-функцию активации. Но слой `Dense` требует в качестве входного сигнала схлопнутый вектор из *n* элементов (каждый из которых представляет собой значение с плавающей точкой). На выходе `SimpleRNN` данные представляют собой тензор длиной 400 элементов, каждый из которых имеет длину 50 элементов. Но порядок элементов для нейронной сети прямого распространения неважен, лишь бы он был одинаков. Мы воспользуемся вспомогательным слоем `Flatten()`, который Keras предоставляет для схлопывания входных данных из тензора 400 × 50 в вектор длиной 20 000 элементов. Именно его мы и передадим в итоговый слой, осуществляющий классификацию. Слой `Flatten` фактически представляет собой процедуру отображения. Это значит, что ошибка распространяется из последнего слоя обратно до соответствующего выходного сигнала слоя RNN, а затем распространяется оттуда обратно во времени, как рассказывалось выше.

При передаче полученного из слоя рекуррентной нейронной сети вектора идеи в сеть прямого распространения порядок входных данных, который мы так старательно старались учесть, более не сохраняется. Но важно отметить, что учитывающее последовательность токенов обучение происходит в самом слое RNN. Эта взаимосвязь кодируется в сети и выражается в самом векторе идеи в процессе агрегирования ошибок и обратного распространения их во времени. Принятое на основе этого вектора идеи с помощью классификатора решение служит обратной связью относительно качества этого вектора идеи для конкретной задачи классификации. Оценивать свой вектор идеи и работать с самой RNN можно и другими способами, но об этом — в следующей главе (чувствуете, как мы предвкушаем следующую главу?). Держитесь, все излагаемое здесь чрезвычайно важно для понимания дальнейшего материала.

8.2. Собираем все вместе

Компилируем модель точно так же, как мы делали со сверточной нейронной сетью в прошлой главе.

В Keras есть несколько утилит, таких как `model.summary()`, для исследования «внутренностей» модели. По мере усложнения моделей отслеживание происходящих внутри них изменений при подстройке гиперпараметров требует немалых

усилий, если регулярно не пользоваться `model.summary()`. Если же записывать эти отчеты вместе с результатами проверочных тестов в журнал настройки гиперпараметров, все становится гораздо проще. Можно даже автоматизировать значительную часть этой задачи и переложить часть работы по хранению записей «на плечи» машины[1] (листинг 8.11).

Листинг 8.11. Компиляция нашей рекуррентной нейронной сети

```
>>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
Using TensorFlow backend.
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (None, 400, 50)           17550
_________________________________________________________________
dropout_1 (Dropout)          (None, 400, 50)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 20000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 20001
=================================================================
Total params: 37,551
Trainable params: 37,551
Non-trainable params: 0
_________________________________________________________________
None
```

Остановитесь и взгляните на число используемых параметров — 37 551, хотя рекуррентная нейронная сеть относительно невелика. Получается немало весов, которые нужно обновлять на основе 20 000 тренировочных примеров (не путайте с 20 000 элементов в последнем слое — это просто совпадение). Взглянем на эти числа и разберемся, откуда именно они взялись.

Мы запросили для слоя `SimpleRNN` 50 нейронов, каждый из которых получает входной сигнал и применяет вес к каждому входному примеру данных. В RNN входной сигнал для каждого временного шага представляет собой один токен. В данном случае токены представлены в виде векторов слов, каждый длиной 300 элементов (300-мерный). Следовательно, у каждого нейрона будет 300 весов:

$$50 \cdot 300 = 15\,000$$

[1] Если хотите автоматизировать выбор гиперпараметров, не зацикливайтесь слишком сильно на поиске по сетке; случайный поиск намного эффективнее (http://hyperopt.github.io/hyperopt/). Если вы действительно хотите заниматься этим всерьез, то попробуйте байесовскую оптимизацию. У оптимизатора гиперпараметров бывает не более одной попытки раз в несколько часов, так что использовать какую-то устаревшую модель настройки гиперпараметров просто недопустимо (и не дай бог рекуррентную нейронную сеть!).

У каждого нейрона также есть постоянное *смещение* (систематическая ошибка, bias) с постоянным входным сигналом 1 (именно это и делает такую ошибку систематической), но с подбираемым весом:

$$15\,000 + 50 \text{ (весов смещения)} = 15\,050.$$

На первом временном шаге первого слоя 15 050 весов. Далее каждый из этих 50 нейронов отправляет свой выходной сигнал на следующий временной шаг сети. Каждый нейрон принимает на входе полный входной вектор вместе с полным выходным вектором с предыдущего шага. На первом временном шаге обратная связь с выхода еще не существует и получает начальные значения в виде вектора нулей такой же длины, как и у выходного сигнала.

У каждого нейрона в скрытом слое есть веса для каждого измерения вложений токенов: всего 300 весов, а также одно смещение для каждого нейрона. И 50 весов для выходных результатов предыдущего временного шага (или нулей для первого $t = 0$ временного шага). Эти 50 весов играют *ключевую роль в обратной связи рекуррентной нейронной сети*. Получаем:

$$300 + 1 + 50 = 351;$$

$$351 \cdot 50 = 17\,550.$$

Получается 17 550 требующих подбора параметров. Эта сеть *разворачивается* на 400 временных шагов (вероятно, слишком много, учитывая проблемы с исчезающими градиентами, но все равно сеть успешно работает). Но на всех разворотах эти 17 550 параметров одинаковы и остаются одинаковыми до вычисления всех обратных распространений ошибок. Обновления весов происходят одновременно, в конце последовательности из прямого прохода и последующего обратного распространения. Хотя мы усложнили алгоритм обратного распространения ошибки, но сэкономили на том, что не стали обучать сеть более чем с 7 миллионами параметров ($17\,550 \cdot 400$), как если бы у каждого разворота сети были свои веса.

В отчете функции `.summary()` указано, что последний слой содержит 20 001 требующий обучения параметр, что не так уж много. После слоя `Flatten` входной сигнал представляет собой 20 000-мерный вектор плюс один входной сигнал для смещения. Поскольку выходной слой содержит только один нейрон, то общее число параметров равно:

$$(20\,000 \text{ входных элементов} + 1 \text{ компонент смещения}) \cdot 1 \text{ нейрон} = 20\,001 \text{ параметр}.$$

Эти числа могут немного ввести вас в заблуждение относительно требуемого вычислительного времени, поскольку при обратном распространении ошибки во времени необходимо столь много дополнительных шагов (по сравнению со сверточными нейронными сетями или обычными нейронными сетями прямого распространения). Но время вычислений не должно играть решающую роль. Как вы увидите в следующей главе, особые таланты рекуррентных нейронных сетей в смысле *памяти* знаменуют новую эру в NLP и обработке любых других последовательных данных. Но вернемся к нашему классификатору.

8.3. Приступим к изучению прошлого

Настало время приступить к фактическому обучению нашей рекуррентной сети, которую мы так старательно компоновали в предыдущем разделе. Как и в случае других моделей Keras, необходимо передать данные в метод `.fit()`, указав желаемую длительность обучения (число эпох), как показано в листинге 8.12.

Листинг 8.12. Обучение и сохранение модели

```
>>> model.fit(x_train, y_train,
...           batch_size=batch_size,
...           epochs=epochs,
...           validation_data=(x_test, y_test))
Train on 20000 samples, validate on 5000 samples
Epoch 1/2
20000/20000 [==============================] - 215s - loss: 0.5723 -
acc: 0.7138 - val_loss: 0.5011 - val_acc: 0.7676
Epoch 2/2
20000/20000 [==============================] - 183s - loss: 0.4196 -
acc: 0.8144 - val_loss: 0.4763 - val_acc: 0.7820

>>> model_structure = model.to_json()
>>> with open("simplernn_model1.json", "w") as json_file:
...     json_file.write(model_structure)
>>> model.save_weights("simplernn_weights1.h5")
Model saved.
```

Не так чтобы ужасно, но и похвастаться особо нечем. Что бы здесь усовершенствовать?

8.4. Гиперпараметры

Все перечисленные в данной книге модели допускают разнообразные настройки для лучшей адаптации к конкретным данным и выборкам. У каждой есть свои преимущества и компромиссы. Найти идеальный набор гиперпараметров обычно весьма непросто. Но человеческие опыт и интуиция позволяют приблизиться к решению этой задачи. Рассмотрим последний пример. Какие гиперпараметры мы выбрали (листинг 8.13)?

Листинг 8.13. Параметры модели

```
>>> maxlen = 400
>>> embedding_dims = 300
>>> batch_size = 32
>>> epochs = 2
>>> num_neurons = 50
```

Произвольно выбранная длина последовательности на основе беглого взгляда на данные

Из предобученной модели Word2vec

Число передаваемых (с агрегированием ошибки) перед обратным распространением ошибки выборочных последовательностей

Степень сложности скрытого слоя

Больше всего вопросов вызывает `maxlen`. Длины примеров в тренировочном наборе данных сильно варьируются. При выравнивании примеров данных длиной менее 100 токенов до 400 и, напротив, при обрезке примеров на 1000 токенов до 400 возникает колоссальное количество шума. Изменение этого параметра влияет на время обучения больше, чем любого другого параметра модели. Длина отдельных примеров данных определяет, насколько далеко обратно во времени должна распространяться ошибка. В современных рекуррентных нейронных сетях это не обязательно. Можно просто развернуть сеть настолько (много/мало), сколько требуется для конкретного примера. В нашем случае это нужно, поскольку в слой прямого распространения передается выходной сигнал, который сам представляет собой последовательность, а для таких слоев необходимы входные данные одинакового размера.

Значение `embedding_dims` определяется выбранной моделью Word2vec, но легко может быть любым числом, адекватно отражающим набор данных. Для точных предсказаний может хватить и столь простого варианта, как унитарное кодирование 50 чаще всего встречающихся токенов в корпусе.

Как и в любой сети, увеличение `batch_size` ускоряет обучение, поскольку снижает то, сколько раз требуется произвести обратное распространение ошибки (наиболее вычислительно дорогостоящую операцию). С другой стороны, чем больше батчи, тем выше риск в итоге оказаться в локальном минимуме.

Проверять и подстраивать параметр `epochs` легко, для этого достаточно просто запустить процесс обучения еще раз. Но это требует выдержки, если при всяком новом проверяемом значении параметра `epochs` начинать этот процесс с «чистого листа». Обучение моделей Keras можно запускать с того места, где закончилось предыдущее, если, конечно, вы не забыли в прошлый раз сохранить модель. Для запуска обучения для ранее уже обучавшейся модели необходимо перезагрузить ее и набор данных и вызвать для данных метод `model.fit()`. Keras не задает значения весов заново, а продолжает обучение, как будто оно и не прерывалось.

Еще один вариант подбора параметра `epochs` — добавить *обратный вызов* Keras под названием `EarlyStopping`. Если передать этот метод модели, она будет обучаться до окончания указанного числа эпох, *если* переданная методу `EarlyStopping` метрика не превысит заданного внутри обратного вызова порогового значения. Одна из часто используемых метрик раннего останова — повышение точности на протяжении нескольких последовательных эпох на проверочных данных. То, что качество модели не улучшается, обычно значит, что пора прекращать обучение.

Такую метрику можно задать и забыть про нее. Модель прекратит обучение, когда доберется до порогового значения. И вы не потратите кучу времени лишь для того, чтобы потом оказалось, что модель начала переобучаться еще 42 эпохи назад.

Важную роль играет также параметр `num_neurons`. Мы без какого-либо обоснования решили использовать 50 нейронов. Теперь попробуем произвести обучение и проверку со 100 нейронами вместо 50, как показано в листингах 8.14 и 8.15.

Листинг 8.14. Увеличиваем размеры сети

```
>>> num_neurons = 100
>>> model = Sequential()
```

```
>>> model.add(SimpleRNN(
...     num_neurons, return_sequences=True, input_shape=(maxlen,\
...     embedding_dims)))
>>> model.add(Dropout(.2))
>>> model.add(Flatten())
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
Using TensorFlow backend.
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (None, 400, 100)          40100
_________________________________________________________________
dropout_1 (Dropout)          (None, 400, 100)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 40000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 40001
=================================================================
Total params: 80,101
Trainable params: 80,101
Non-trainable params: 0
_________________________________________________________________
None
```

Листинг 8.15. Обучение нашей увеличенной сети

```
>>> model.fit(x_train, y_train,
...           batch_size=batch_size,
...           epochs=epochs,
...           validation_data=(x_test, y_test))
Train on 20000 samples, validate on 5000 samples
Epoch 1/2
20000/20000 [==============================] - 287s - loss: 0.9063 -
acc: 0.6529 - val_loss: 0.5445 - val_acc: 0.7486
Epoch 2/2
20000/20000 [==============================] - 240s - loss: 0.4760 -
acc: 0.7951 - val_loss: 0.5165 - val_acc: 0.7824
>>> model_structure = model.to_json()
>>> with open("simplernn_model2.json", "w") as json_file:
...     json_file.write(model_structure)
>>> model.save_weights("simplernn_weights2.h5")
Model saved.
```

Точность на проверочных данных (78,24 %) после удвоения сложности модели в одном из слоев оказалась лишь на 0,04 % лучше. Столь мизерное улучшение наводит на мысль, что модель (для этого слоя сети) слишком сложна для наших данных. Этот слой сети, возможно, слишком широк.

Вот что получится, если уменьшить `num_neurons` до 25:

```
20000/20000 [==============================] - 240s - loss: 0.5394 -
acc: 0.8084 - val_loss: 0.4490 - val_acc: 0.7970
```

Интересно. Качество модели даже улучшилось после ее небольшого сужения «в талии»: ненамного, на 1,5 %. Выработка интуиции на подобные эксперименты требует немалого времени. Особенно усложняется все по мере роста времени обучения, когда вы уже не можете наслаждаться мгновенными результатами и удовлетворением, привычным по другим программистским задачам. И иногда, если менять по одному параметру за раз, можно не заметить выгод, которые сулит подстройка сразу двух одновременно. Но если нырнуть в эту кроличью нору комбинаторики, сложность задачи вырастет до небес.

СОВЕТ

Чаще экспериментируйте и всегда записывайте, как модель отреагировала на ваши манипуляции. Подобные проверки на практике — кратчайший путь к выработке интуитивного понимания принципов создания моделей.

Если вам кажется, что модель переобучилась на данных, но найти способ упростить ее не получается, всегда можно попытаться увеличить процент дропаута. Это крайнее средство, способное снизить риск переобучения с сохранением при этом сложности модели, соответствующей данным. Если задать процент дропаута выше 50 %, у модели начнутся сложности с обучением. Обучение станет медленным, а ошибка на проверочных данных будет сильно скакать. Но диапазон от 20 до 50 % вполне допустим во многих задачах NLP для рекуррентных сетей.

8.5. Предсказание

У нас теперь есть обученная модель, и мы можем производить предсказания аналогично CNN в предыдущей главе, как показано в листинге 8.16.

Листинг 8.16. Тональность высказывания о плохой погоде

```
>>> sample_1 = "I hate that the dismal weather had me down for so long, when
➥ will it break! Ugh, when does happiness return? The sun is blinding and
➥ the puffy clouds are too thin. I can't wait for the weekend."

>>> from keras.models import model_from_json
>>> with open("simplernn_model1.json", "r") as json_file:
...     json_string = json_file.read()
>>> model = model_from_json(json_string)
>>> model.load_weights('simplernn_weights1.h5')

>>> vec_list = tokenize_and_vectorize([(1, sample_1)])
>>> test_vec_list = pad_trunc(vec_list, maxlen)
>>> test_vec = np.reshape(test_vec_list, (len(test_vec_list), maxlen,\
...     embedding_dims))

>>> model.predict_classes(test_vec)
array([[0]], dtype=int32)
```

Мы передаем фиктивное значение в качестве первого элемента кортежа, поскольку это требуется нашей вспомогательной функции, исходя из способа обработки данных

Процедура токенизации возвращает список данных (в нашем случае длины 1)

Опять отрицательная тональность.

У нас появился дополнительный инструмент, который можно добавить в конвейер для классификации возможных ответов, а также поступающих вопросов и запросов, которые могут вводить пользователи. Но почему нужно использовать именно рекуррентную нейронную сеть? Если коротко: лучше не использовать. По крайней мере, применять не ту `SimpleRNN`, которую мы здесь реализовали. По сравнению с нейронными сетями прямого распространения или сверточными нейронными сетями их обучение и передача им новых примеров требуют довольно много ресурсов. Результаты, по крайней мере в приведенном примере, не намного лучше, если вообще лучше.

Так зачем вообще связываться с RNN? Концепция запоминания элементов прошлых входных сигналов играет ключевую роль в NLP. Рекуррентные нейронные сети обычно не способны справиться с проблемами «исчезающих» градиентов, особенно в случаях такого большого числа временных шагов, как у нас. В следующей главе мы начнем исследовать другие способы *запоминания*, которые, как это сформулировал Андрей Карпати (Andrej Karpathy), «неоправданно эффективны»[1].

Следующие разделы описывают некоторые нюансы рекуррентных нейронных сетей, не упомянутые в нашем примере, но имеющие большое значение.

8.5.1. Сохранение состояния

Иногда требуется помнить информацию не просто от одного временного шага (токена) до следующего в пределах примера, а от одного входного *примера* до следующего. Что происходит с этой информацией в конце шага обучения? Итоговый выходной сигнал ни на что не влияет, только кодируется в весах в процессе обратного распространения ошибки, так что следующий сигнал начинает все заново. В базовом слое RNN (а значит, и в `SimpleRNN`) в Keras имеется поименованный аргумент `stateful` («с сохранением состояния»). По умолчанию он равен `False`. Если поменять его значение на `True` при добавлении в модель слоя `SimpleRNN`, то последний выходной сигнал для последнего примера данных будет рекурсивно передан на следующем временном шаге вместе с входным сигналом первого токена, подобно тому как это происходит в середине примера данных.

Устанавливать аргумент `stateful` в `True` важно при моделировании большого документа, разбитого для обработки на абзацы или предложения. Его можно использовать даже для моделирования смысла целого корпуса связанных документов. Но не следует обучать RNN с сохранением состояния на несвязанных документах или текстах без сброса состояния модели между примерами данных. Аналогично если перетасовывать примеры текста, то несколько последних токенов одного примера не будут иметь никакого отношения к первым токенам следующего. Так что для перетасованного текста нужно устанавливать флаг `stateful` в `False`, поскольку порядок примеров ничем не поможет модели в обучении.

---

[1] *Karpathy A.* The Unreasonable Effectiveness of Recurrent Neural Networks («Неоправданная эффективность рекуррентных нейронных сетей») (http://karpathy.github.io/2015/05/21/rnn-effectiveness/).

Если передать методу `fit` параметр `batch_size`, то память модели будет хранить выходные сигналы для каждого примера батчами, а далее будет передавать в первый пример очередного батча выходной сигнал первого примера предыдущего батча, второй — во второй, *i*-й — в *i*-й. При моделировании большего корпуса по кусочкам важно уделять внимание порядку внутри набора данных.

8.5.2. И в другую сторону

До сих пор мы обсуждали взаимосвязи между словами и их предшественниками. Но нельзя ли обосновать зеркальное отражение этих зависимостей слов?

They wanted to pet the dog whose fur was brown.

К моменту, когда мы добрались до токена *fur*, мы уже встретили токен *dog* и кое-что про него знаем. Но предложение содержит также информацию о том, что у этой собаки есть мех и что этот мех коричневый. И эта информация релевантна упомянутому ранее действию — *pet* и тому факту, что *they* хотели собаку погладить. Возможно, *they* хотят гладить только мягкие и пушистые создания, а не зеленые вещи вроде кактусов.

Люди читают предложение в одном направлении, но могут мысленно вернуться к предыдущим частям текста по мере появления новой информации. Они способны справиться с информацией, расположенной не в самом удобном порядке. Было бы неплохо, чтобы наша модель тоже могла пробегать по входному сигналу в обратную сторону. В этом нам помогут *двунаправленные* (bidirectional) рекуррентные нейронные сети. В Keras есть слой-адаптер для автоматического поворота нужных входных и выходных сигналов в другую сторону и автоматического создания двунаправленной RNN (листинг 8.17).

Листинг 8.17. Создание двунаправленной рекуррентной сети

```
>>> from keras.models import Sequential
>>> from keras.layers import SimpleRNN
>>> from keras.layers.wrappers import Bidirectional

>>> num_neurons = 10
>>> maxlen = 100
>>> embedding_dims = 300

>>> model = Sequential()
>>> model.add(Bidirectional(SimpleRNN(
...     num_neurons, return_sequences=True),\
...     input_shape=(maxlen, embedding_dims)))
```

Основная идея состоит в размещении двух RNN рядом друг с другом и передаче одного и того же входного сигнала в одну обычным образом, а во вторую — *«задом наперед»* (рис. 8.13). Выходной сигнал этих двух сетей объединяется, каждый временной шаг одной сети — с соответствующим шагом (тот же входной токен) другой. Выходной сигнал последнего временного шага «прямой» сети объединяется с выходным сигналом, сгенерированным для того же входного токена на первом временном шаге «обратной» сети.

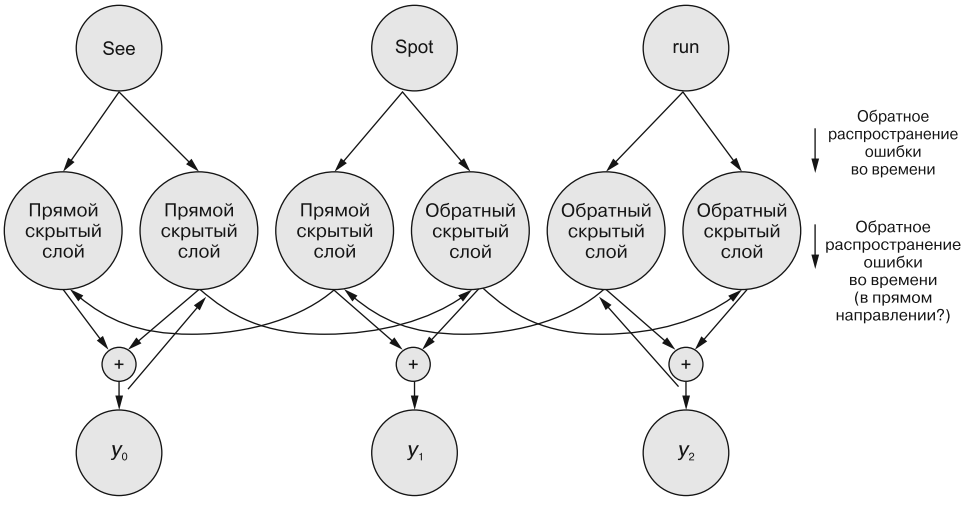


Рис. 8.13. Двунаправленная рекуррентная нейронная сеть

СОВЕТ

В Keras также существует поименованный аргумент go_backwards. Если задать его равным True, Keras автоматически отразит зеркально входные последовательности и отправит их в сеть в обратном порядке. Это вторая часть двунаправленного слоя.

Такой аргумент может оказаться полезен, если использовать двунаправленный адаптер нельзя, поскольку рекуррентная нейронная сеть (из-за проблемы «исчезающих» градиентов) более чувствительна к данным в конце примера, чем в начале. Если примеры данных были дополнены токенами <PAD> в конце, то все нормально, проблемы скрыты глубоко во входном цикле. Аргумент go_backwards — простой способ обхода этой проблемы.

Благодаря этим инструментам мы уже начинаем не только предсказывать и классифицировать текст, но и по-настоящему моделировать сам язык и его использование. Благодаря этому глубокому алгоритмическому пониманию языка модель может не только механически повторять уже виденный текст, но и генерировать совершенно новые высказывания!

8.5.3. Что это такое?

Перед слоем `Dense` из последнего временного шага рекуррентного слоя для заданной входной последовательности мы получили вектор формы (количество нейронов × 1). Этот вектор параллелен *вектору идеи*, который мы получили из сверточной нейронной сети в предыдущей главе. Он кодирует последовательность токенов. Правда, он может только кодировать идеи последовательностей относительно меток, на которых обучается сеть. Но в смысле NLP это очередной замечательный шаг на пути к численному кодированию в векторе более высокоуровневых понятий.

Резюме

- В последовательностях естественного языка (словах или символах) очередность имеет важное значение для понимания этой последовательности моделью.
- Разбиение высказывания на естественном языке вдоль временного измерения (токенов) помогает машине глубже понимать естественный язык.
- Ошибки можно распространять обратно во времени (по токенам), а не только по слоям глубокой сети.
- Поскольку RNN — особенно глубокие нейронные сети, градиенты RNN наиболее «импульсивны», они могут «исчезать» или «взрываться».
- Эффективное моделирование символьных последовательностей естественного языка было невозможно, пока для этой задачи не воспользовались рекуррентными нейронными сетями.
- Весы в RNN подстраиваются для конкретного примера в агрегированном по времени виде.
- Существует много методов, позволяющих изучать результаты работы рекуррентных нейронных сетей.
- Моделировать последовательность естественного языка в документе можно, одновременно пропуская последовательность токенов через RNN в прямом и обратном порядке.

9 Эффективное сохранение информации с помощью сетей с долгой краткосрочной памятью

В этой главе

- Добавление в рекуррентные нейронные сети более глубокой памяти.
- Шлюзование информации в нейронных сетях.
- Классификация и генерация текста.
- Моделирование языковых закономерностей.

При всех достоинствах рекуррентных нейронных сетей в моделировании взаимосвязей, *возможно*, и причинно-следственных связей в последовательных данных, у них есть один недостаток: *эффект* токена практически полностью теряется два токена спустя[1]. Любое влияние первого узла на третий (через два временных шага после первого) будет полностью затерто новыми данными, поступившими на промежуточных временных шагах. Это важная составляющая структуры сети. Однако она мешает учитывать распространенную ситуацию в естественном языке, когда токены глубоко связаны друг с другом, хотя и разнесены далеко в предложении.

[1] Кристофер Ола (Christopher Olah) объясняет почему: https://colah.github.io/posts/2015-08-Understanding-LSTMs.

Рассмотрим следующий пример: *The young woman went to the movies with her friends.*

Подлежащее *woman* непосредственно предшествует сказуемому — смысловому глаголу *went*[1]. В предыдущих главах вы узнали, что усвоение информации из этой взаимосвязи не представляет сложностей как для сверточных, так и для рекуррентных нейронных сетей.

Но взглянем на аналогичное предложение: *The young woman, having found a free ticket on the ground, went to the movies.*

Существительное и глагол находятся уже не рядом друг с другом. Рекуррентной нейронной сети будет непросто обнаружить взаимосвязь между подлежащим *woman* и смысловым глаголом *went* в новом более длинном предложении. Для этого примера рекуррентная нейронная сеть, вероятно, преувеличит связь между глаголом *having* и существительным *woman*. И недооценит связь с *went*, смысловым глаголом (сказуемым). Мы потеряли связь между подлежащим и сказуемым. Веса в рекуррентной нейронной сети затухают слишком быстро по мере просмотра предложения.

Наша (непростая) задача — создать сеть, которая сможет уловить одну и ту же главную *мысль* обоих предложений. Нужен способ запоминания прошлого в масштабах всего предложения. Необходима *долгая краткосрочная память* (long short-term memory, LSTM)[2].

Современные версии сетей с долгой краткосрочной памятью обычно используют специальный блок нейронной сети — *шлюзовой рекуррентный блок* (gated recurrent unit, GRU). Он эффективно поддерживает как долго-, так и краткосрочную память, благодаря чему LSTM может обрабатывать длинные предложения или документы более аккуратно[3]. На самом деле LSTM демонстрируют настолько хорошие результаты, что заменили рекуррентные нейронные сети практически во всех приложениях, включающих временные ряды, дискретные последовательности и NLP[4].

9.1. Долгая краткосрочная память

LSTM вводит понятие *состояния* (state) для каждого из слоев рекуррентной сети. Состояние играет роль *памяти*. Его можно рассматривать как аналог добавления атрибутов в класс в объектно-ориентированном программировании. Атрибуты состояния памяти обновляются с каждым тренировочным примером.

---

[1] Went в этом предложении играет роль сказуемого (смыслового глагола). Термины английской грамматики можно найти по адресу https://www.butte.edu/departments/cas/tipsheets/grammar/sentence_structure.html.

[2] Одну из первых научных статей о LSTM написали Хохрайтер (Hochreiter) и Шмидхубер (Schmidhuber) в 1997 г.: Long Short-Term Memory: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.676.4320&rep=rep1&type=pdf.

[3] *Kyunghyun Cho et al.* Learning Phrase Representations using RNN Encoder – Decoder for Statistical Machine Translation, 2014. https://arxiv.org/pdf/1406.1078.pdf.

[4] В следующем сообщении из блога Кристофера Ола рассказывается, почему это так: https://colah.github.io/posts/2015-08-Understanding-LSTMs.

Фокус в том, что в LSTM правила, определяющие хранение информации в состоянии (памяти), — это сама обученная нейронная сеть. Ее можно обучить тому, что надо запоминать, в то время как остальная рекуррентная нейронная сеть учится предсказывать целевую метку! С появлением памяти и состояния можно выучивать (усваивать, learn) зависимости, простирающиеся на весь пример данных, а не только на 1–2 токена. Благодаря таким «долгим» зависимостям можно выйти за рамки отдельных слов и больше узнать о языке.

С помощью LSTM модель может выявлять очевидные для человека (и обрабатываемые на подсознательном уровне) языковые закономерности, которые позволяют не только точнее классифицировать примеры данных, но и генерировать на их основе совершенно новый текст. Достижения в этой области еще далеки от идеала, но результаты, которые вы увидите, даже в представленных модельных примерах поразительны.

Как же это все работает (рис. 9.1)?

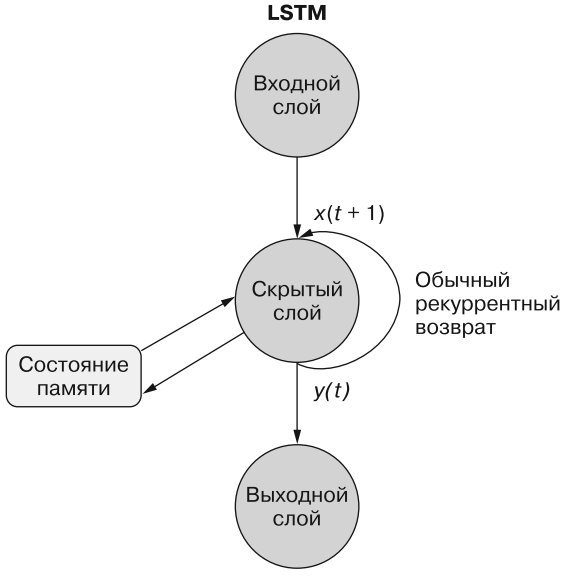


Рис. 9.1. Сеть LSTM и ее память

Входной сигнал влияет на состояние памяти, а оно, в свою очередь, влияет на выходной слой, как и в обычной рекуррентной сети. Но это состояние памяти сохраняется на протяжении всех временных шагов временного ряда (предложения или документа). Поэтому каждый входной сигнал влияет на состояние памяти, а также на выходной сигнал скрытого слоя. Магия состояния памяти заключается в том, что она *усваивает*, что нужно запоминать, и в то же время обучается воспроизводить выходной сигнал с помощью обычного обратного распространения ошибки! Как же это происходит?

Во-первых, развернем обычную RNN и добавим в нее блок памяти. Рисунок 9.2 напоминает такую сеть. Однако в дополнение к выходному сигналу активации,

подаваемому в версию того же скрытого слоя для следующего временного шага, мы передаем по временным шагам сети также состояние памяти. На каждой временной итерации у скрытого рекуррентного блока есть доступ к блоку памяти. Добавление этого блока памяти и механизмов взаимодействия с ним существенно отличает данную сеть от слоя традиционной нейронной сети. Впрочем, вам будет интересно узнать, что можно спроектировать набор слоев обычных рекуррентных нейронных сетей (граф вычислений), реализующий все существующие в слое LSTM вычисления. Слой LSTM — узкоспециализированная рекуррентная нейронная сеть.

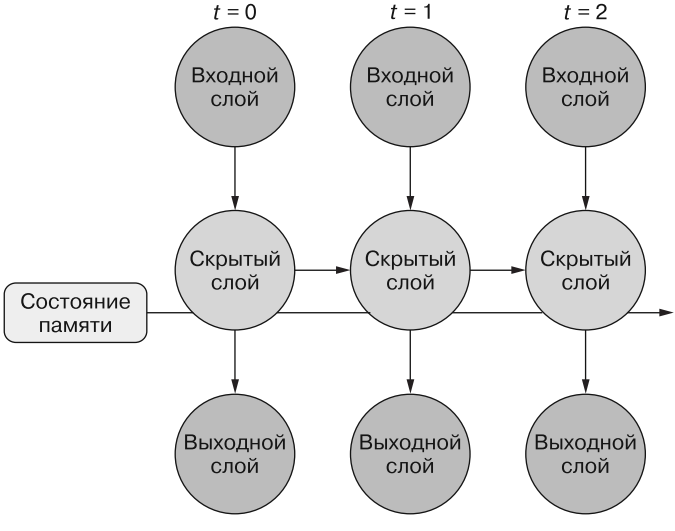


Рис. 9.2. Развернутая сеть LSTM и ее память

ПРИМЕЧАНИЕ

Зачастую в литературе[1] представленный на рис. 9.2 блок состояния памяти называют LSTM-ячейкой (cell), а не LSTM-нейроном, поскольку он содержит два дополнительных нейрона (шлюза), подобно ячейке чипа памяти компьютера (https://ru.wikipedia.org/wiki/Ячейка_памяти). Структура, получающаяся при сочетании LSTM-ячейки памяти с сигма-функцией активации для вывода значения в следующую LSTM-ячейку, называется LSTM-блоком (unit). Несколько LSTM-блоков объединяются в LSTM-слой (layer). Горизонтальная линия, пересекающая все развернутые рекуррентные нейроны на рис. 9.2, — это сигнал, содержащий память (состояние). При передаче последовательности токенов в многоблочный LSTM-слой он становится вектором, размерность которого соответствует числу LSTM-ячеек.

[1] Прекрасный недавний пример использования англоязычной терминологии LSTM можно найти в диссертации: *Graves A.* Supervised Sequence Labelling with Recurrent Neural Networks. 2012. https://mediatum.ub.tum.de/doc/673554/file.pdf.

Посмотрим на одну из этих ячеек внимательнее. Теперь ячейки стали сложнее и представляют собой не просто ряд весовых коэффициентов для входных сигналов и функцию активации для этих весов. Как и ранее, входной сигнал слоя (или ячейки) — это сочетание входного примера данных и выходного сигнала с предыдущего временного шага. При поступлении информации в ячейку вместо вектора весов ее ждут три шлюза: шлюз забывания, шлюз входных данных/кандидатов и выходной шлюз (рис. 9.3).

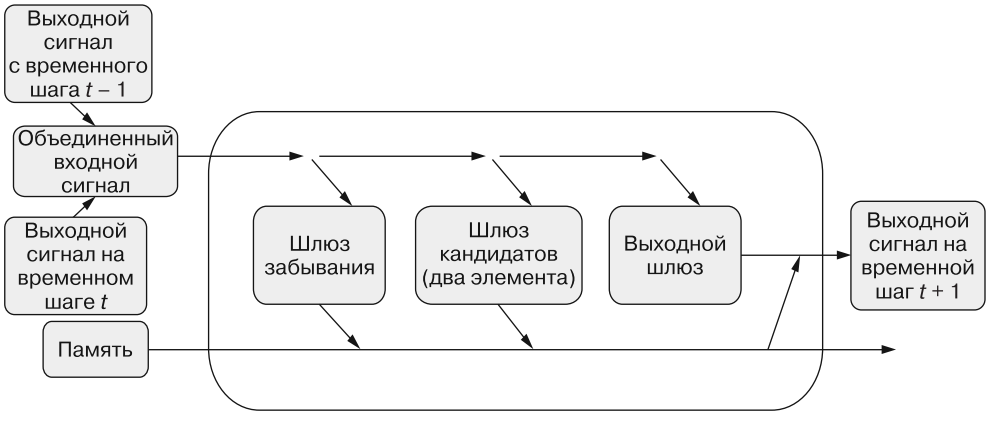


Рис. 9.3. LSTM-слой на временном шаге t

Каждый из этих шлюзов является нейронным сетевым слоем прямого распространения, состоящим из последовательности весов, подбираемых сетью, плюс функция активации. Фактически один из шлюзов состоит из двух путей прямого распространения, следовательно, в слое будет четыре набора подбираемых весов. Весы и функции активации нацелены на то, чтобы *дать возможность* информации проходить через ячейку в различных объемах.

Чтобы не увязнуть в дебрях, рассмотрим реализацию примера на языке Python из главы 8, заменив слой SimpleRNN на LSTM. Вы можете воспользоваться векторизованными дополненными/усеченными обработанными данными из предыдущей главы: `x_train`, `y_train`, `x_test` и `y_test` (листинг 9.1).

Листинг 9.1. LSTM-слой в Keras

```
>>> maxlen = 400
>>> batch_size = 32
>>> embedding_dims = 300
>>> epochs = 2
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Dropout, Flatten, LSTM
>>> num_neurons = 50
>>> model = Sequential()
>>> model.add(LSTM(num_neurons, return_sequences=True,
...                input_shape=(maxlen, embedding_dims)))
```

```
>>> model.add(Dropout(.2))
>>> model.add(Flatten())
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
>>> print(model.summary())
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 400, 50)           70200
_________________________________________________________________
dropout_1 (Dropout)          (None, 400, 50)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 20000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 20001
=================================================================
Total params: 90,201.0
Trainable params: 90,201.0
Non-trainable params: 0.0
```

Изменились всего один импорт и строка кода Keras. Но внутри поменялось гораздо больше. Из отчета видно, что число подбираемых параметров намного увеличилось, по сравнению с SimpleRNN в предыдущей главе, при том же числе нейронов (50). Напомним, что у SimpleRNN были следующие веса:

- 300 (по одному для каждого элемента входного вектора);
- 1 (для постоянного смещения);
- 50 (по одному для выходного сигнала каждого нейрона с предыдущего временного шага).

Итого 351 вес из расчета на каждый нейрон:

$$351 \cdot 50 = 17\,550 \text{ для слоя.}$$

У ячеек по четыре шлюза (суммарно четыре нейрона):

$$17\,550 \cdot 4 = 70\,200.$$

А что играет роль памяти? Память представлена вектором, число элементов в котором равно числу нейронов в ячейке. Наш случай относительно прост, всего 50 нейронов, поэтому блок памяти — вектор значений с плавающей точкой из 50 элементов.

Но что делают эти шлюзы? Чтобы разобраться, проследим за первым примером данных в его «путешествии» через сеть (рис. 9.4).

«Путешествие» через ячейку — вовсе не прямой путь, он ветвится, и мы будем идти по этим веткам некоторое время, после чего возвращаться назад, затем снова вперед, вновь идти по веткам и, наконец, окончательно возвращаться для получения итогового выходного сигнала ячейки.

Передадим 300-элементное векторное представление первого токена первого примера данных в первую LSTM-ячейку. На пути в ячейку векторное представление данных объединяется с векторным выходным сигналом с предыдущего временного шага (нулевой вектор на первом временном шаге). В этом примере длина вектора

составляет 300 + 50 элементов. Иногда в конец вектора добавляется 1 — это элемент, соответствующий постоянному смещению. Поскольку соответствующий смещению вес всегда умножается на 1 перед отправкой в функцию активации, то этот входной сигнал иногда не включается в векторное представление входного сигнала для упрощения диаграмм.

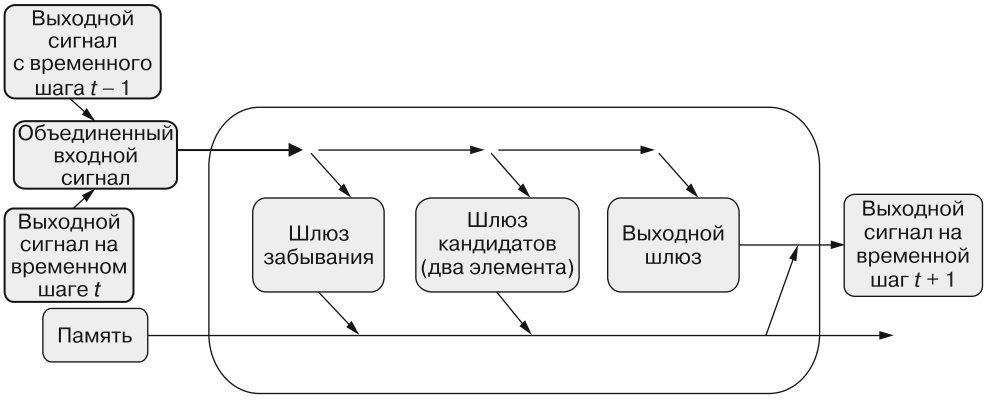


Рис. 9.4. Входные сигналы LSTM-слоя

На первой развилке пути мы передаем копию объединенного входного вектора в *шлюз забывания* (звучит угрожающе) (рис. 9.5). Он обучается на заданных входных данных тому, какую долю памяти ячейки необходимо стереть. Стоп! Мы только что подключили эту самую память и сразу же собираемся из нее что-то стирать? С ума сойти!

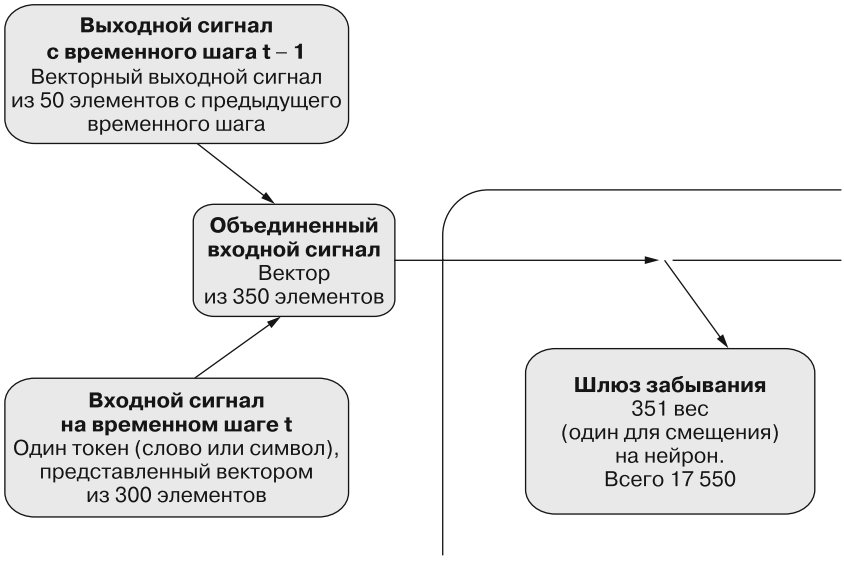


Рис. 9.5. Первая остановка — шлюз забывания

Идея забывания столь же важна, как и идея запоминания. Читая книгу, человек усваивает из текста конкретную информацию (например, единственное или множественное число существительного), хочет запомнить ее, чтобы правильно определить соответствующее спряжение глагола или форму прилагательного. В романских языках необходимо распознать также род существительного и использовать эти сведения далее в предложении. Но во входной последовательности существительные могут легко меняться, поскольку входная последовательность может состоять из нескольких фраз, предложений или даже документов. По мере выражения в высказываниях новых идей то, что существительное стояло во множественном числе, перестает быть релевантным для дальнейшего несвязанного текста Фридриха Ницше:

A thinker sees his own actions as experiments and questions — as attempts to find out something. Success and failure are for him answers above all.

В этой цитате глагол *see* стоит в форме, подходящей для существительного *thinker*. Следующий активный глагол — *to be* во втором предложении. Здесь *be* стоит в форме *are* для соответствия существительным во множественном числе: *Success and failure*. Если поставить его в форму, соответствующую первому встречающемуся существительному, *thinker*, то мы получим неправильную форму *is*. Поэтому LSTM должна не только моделировать «долгие» зависимости в предложении, но и забывать их по мере появления новых. Именно для этого и необходимы шлюзы забывания, которые освобождают место в ячейках памяти для *релевантных* воспоминаний.

Сеть не работает с подобными явными представлениями. Она пытается найти набор весов, при умножении на которые входных сигналов от последовательности токенов ячейка памяти и выходной сигнал обновлялись бы так, чтобы минимизировать ошибку. Удивительно, что они вообще работают. И работают очень даже неплохо. Но хватит восторгаться: вернемся к забыванию.

Сам шлюз забывания (рис. 9.6) представляет собой нейронную сеть прямого распространения. Он состоит из n нейронов, каждый с $m + n + 1$ весовых коэффициентов. Так что у шлюза забывания в нашем примере 50 нейронов, каждый с 351 (300 + 50 + 1) весовым коэффициентом. Функция активации для шлюза забывания является сигма-функцией, поскольку значения выходного сигнала для каждого нейрона в шлюзе должны располагаться между 0 и 1.

Получается, выходной вектор шлюза забывания — некая разновидность маски, хотя и пористой, стирающей элементы из вектора памяти. Чем ближе выводимые шлюзом забывания значения к 1, тем бóльшая часть накопленной памяти соответствующего элемента сохраняется для текущего временного шага; чем ближе они к 0, тем больше памяти стирается (рис. 9.7).

Активно забываем разные вещи, готово. Теперь лучше научиться запоминать что-нибудь новое, иначе все быстро пойдет наперекосяк. Как и в шлюзе забывания, мы воспользуемся маленькой сетью, которая обучится наращивать память на основе двух факторов: текущего входного сигнала и выходного с прошлого временного шага. Это происходит в следующем шлюзе, в который мы попадаем, — в *шлюзе кандидатов* (candidate gate).

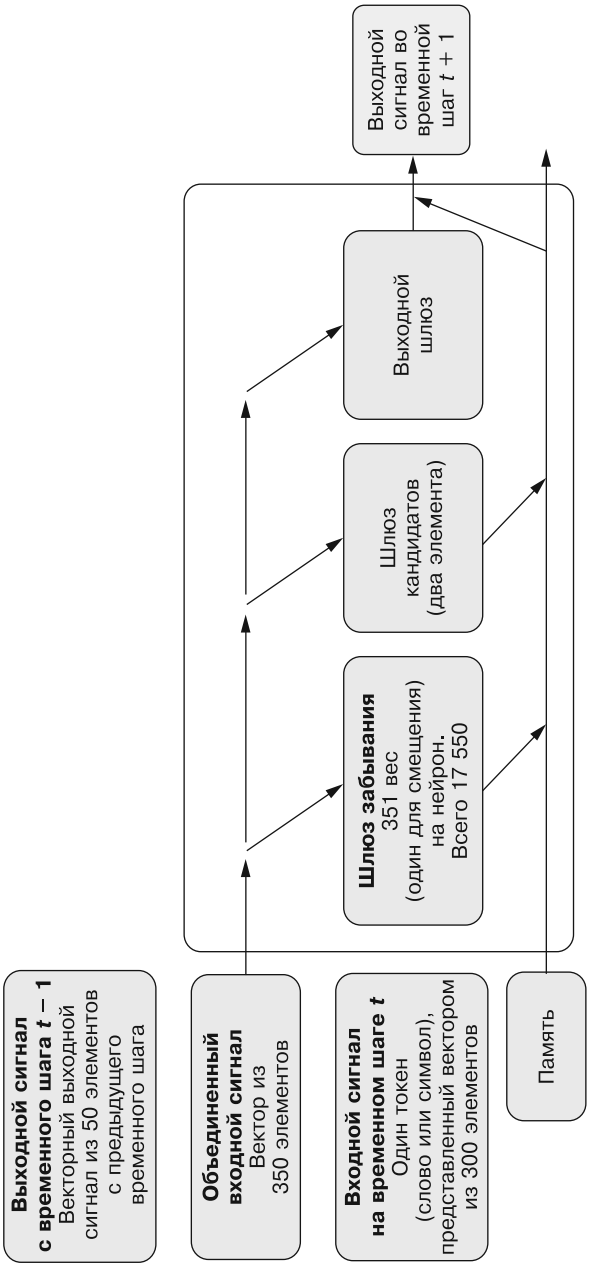


Рис. 9.6. Шлюз забывания

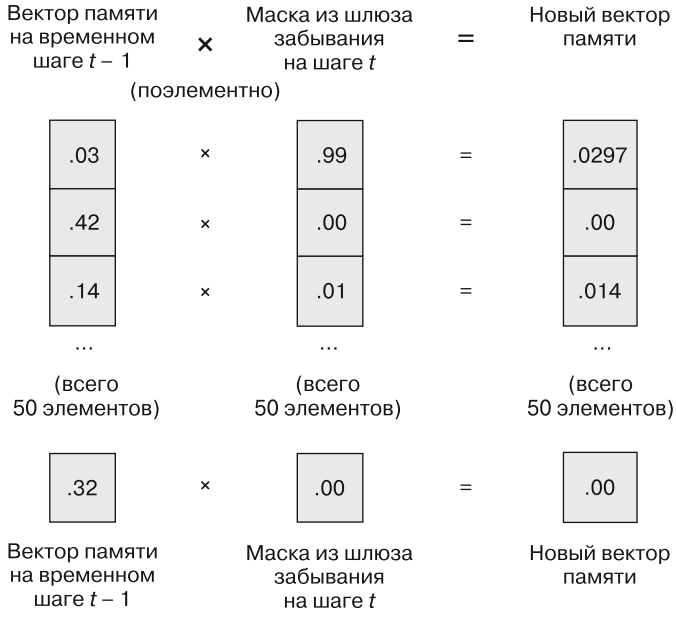


Рис. 9.7. Применение шлюза забывания

В шлюзе кандидатов есть два отдельных нейрона, которые выполняют следующее.

1. Определяют, какие элементы входного вектора заслуживают запоминания (аналогично маске шлюза забывания).
2. Направляют запомненные входные элементы в нужный слот памяти.

Первую часть шлюза кандидатов составляет нейрон с сигма-функцией активации. Он должен установить, какие входные значения вектора памяти нужно обновить. Этот нейрон очень похож на маску в шлюзе забывания.

Вторая часть этого шлюза определяет, какими значениями будет обновлена память. Функция активации второй части — `th`[1], следовательно, значение на выходе оказывается в диапазоне от –1 до 1. Выходные значения двух векторов перемножаются поэлементно. Полученный в результате перемножения вектор прибавляется, опять поэлементно, к регистру памяти, в результате чего и происходит запоминание новых подробностей (рис. 9.8).

Этот шлюз одновременно усваивает, какие значения нужно извлечь и каков порядок величин данных значений. Маска и величины добавляются в состояние памяти. Как и шлюз забывания, шлюз кандидатов обучается маскировать неподходящую информацию перед добавлением в память ячейки.

[1] Гиперболический тангенс, в англоязычной литературе обозначается tanh. — *Примеч. пер.*

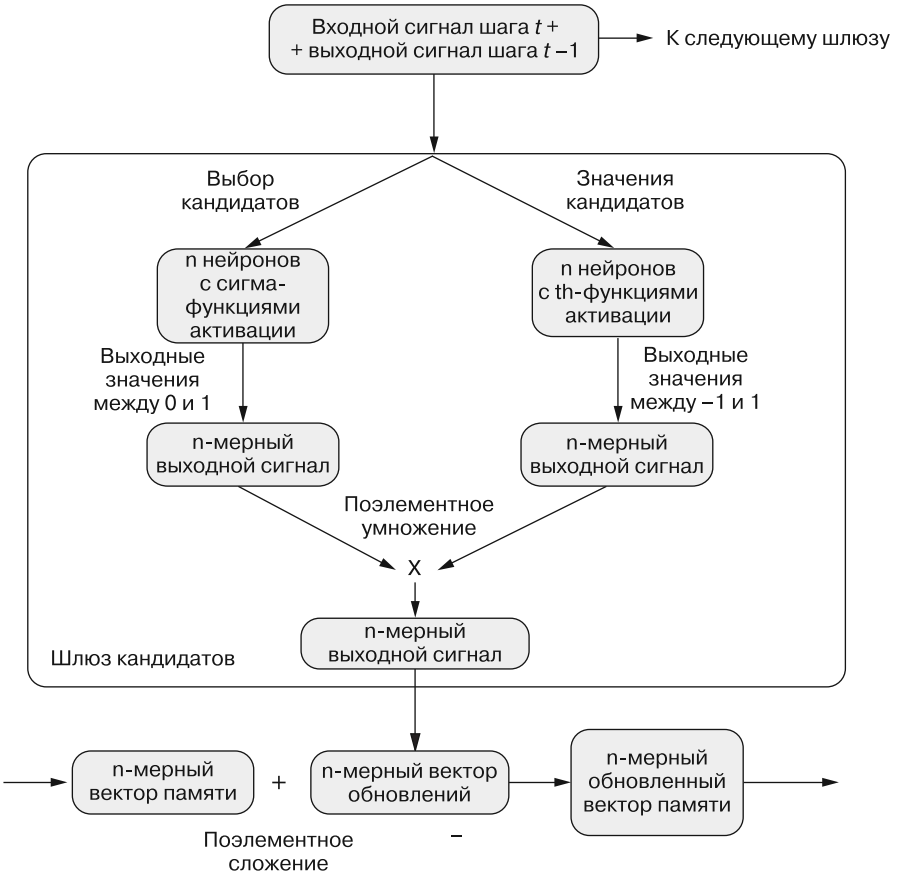


Рис. 9.8. Шлюз кандидатов

Итак, нерелевантная информация забыта, а новая зафиксирована в памяти. Мы добрались до последнего шлюза ячейки — *выходного шлюза* (output gate).

До сих пор в «путешествии» по ячейке мы лишь записывали данные в память ячейки. Теперь же пришло время извлечь из этой структуры какую-то пользу. Выходной шлюз получает входной сигнал (напомним, что он по-прежнему представляет собой объединение входного сигнала на временном шаге t и выходного сигнала ячейки на временном шаге $t - 1$) и передает его на выход.

Объединенный входной сигнал передается на веса n нейронов, затем применяется сигма-функция активации для выдачи на выходе n-мерного вектора значений с плавающей точкой аналогично выходному сигналу SimpleRNN. Но не будем торопиться выдавать эту информацию за пределы ячейки.

Созданная структура памяти уже готова к использованию, и теперь необходимо принять решение, что же нам *следует* выдать на выходе. Для этого мы воспользуемся нашей памятью для создания последней маски. Эта маска тоже представляет собой

некую разновидность шлюза. Но применять данный термин для нее мы не станем, поскольку она не содержит никаких подбираемых путем обучения параметров, что отличает ее от трех описанных выше шлюзов.

Созданная на основе памяти маска — это состояние памяти с примененной поэлементно функцией активации *th*, в результате чего получается *n*-мерный вектор значений с плавающей точкой в диапазоне от –1 до 1. Этот вектор маски затем поэлементно умножается на исходный вектор, вычисленный на первом шаге выходного шлюза. Полученный в результате *n*-мерный вектор передается за пределы ячейки в качестве «официального» выходного сигнала ячейки на временном шаге *t* (рис. 9.9).

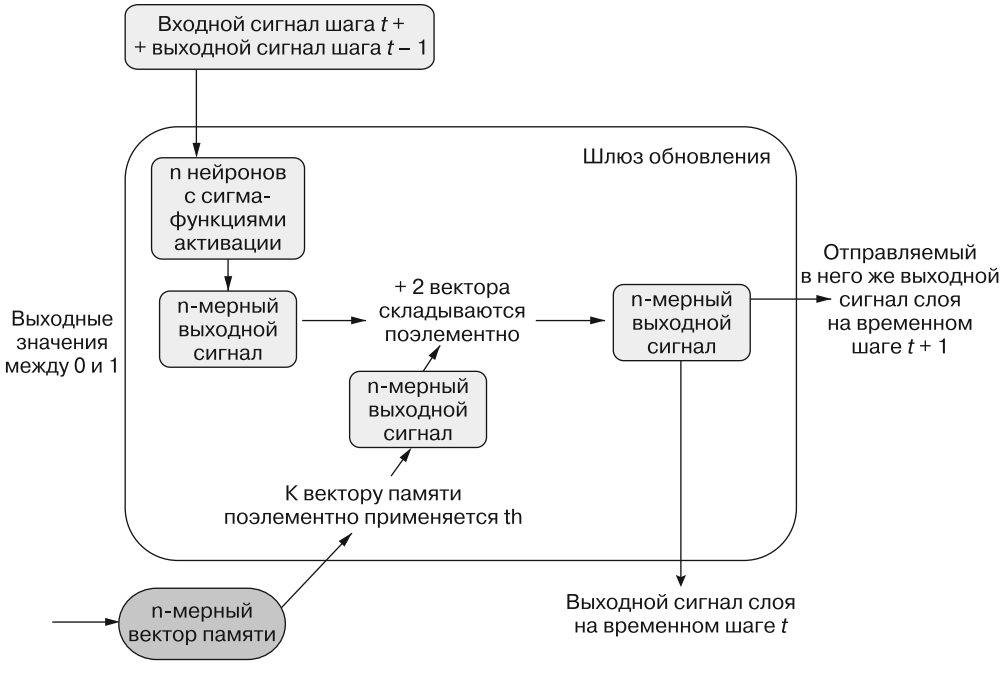


Рис. 9.9. Обновляющий/выходной шлюз

ПРИМЕЧАНИЕ

> Помните, что выходной сигнал LSTM-ячейки аналогичен выходному сигналу простого рекуррентного слоя нейронной сети. Он передается наружу из ячейки в качестве выходного сигнала слоя (на временном шаге t) и обратно себе как часть входного сигнала на временном шаге t + 1.

Таким образом, именно память ячейки определяет, что важнее для выходного сигнала на временном шаге *t* при заданном входном сигнале на временном шаге *t* и выходном сигнале на временном шаге *t* – 1 и всех почерпнутых на текущий момент из входной последовательности подробностях.

9.1.1. Обратное распространение ошибки во времени

Как же происходит обучение? С помощью метода обратного распространения ошибки — как и во всех прочих нейронных сетях. На минуту вернемся назад и взглянем на решаемую задачу с точки зрения этого усложнения. «Наивная» RNN подвержена проблеме «исчезающего» градиента, поскольку производная на любом временном шаге пропорциональна самим весам. По мере возврата во времени и слияния различных дельт после нескольких итераций влияние весов (и скорости обучения) приведет к уменьшению градиента до 0. Обновление весов в конце обратного распространения ошибки (соответствующего началу последовательности) окажется незначительным или вообще нулевым. Похожая проблема возникает, когда весовые коэффициенты великоваты: происходит «взрывной» рост градиента несоразмерно сети.

LSTM избегает этой проблемы с помощью самого состояния памяти. Нейроны во всех шлюзах обновляются с помощью производных функций, для которых служат источниками, а именно тех, которые обновляют состояние памяти при прямом проходе. Поэтому если на каждом временном шаге применить обычное цепное правило к прямому распространению ошибки «задом наперед», изменения нейронов будут зависеть только от состояния памяти на этом и предыдущем временных шагах. Таким образом, ошибка функции в целом остается «близкой» к нейронам для каждого временного шага. Это называется *каруселью ошибки* (error carousel).

На практике

Как же все это работает на практике? Точно так же, как и в простой RNN из предыдущей главы. Единственное изменение во внутренних механизмах нашего «черного ящика» — рекуррентный слой в сети. Можно просто заменить слой SimpleRNN Keras на слой LSTM Keras, а все остальные компоненты классификатора останутся теми же.

Мы воспользуемся тем же набором данных с той же предварительной обработкой: токенизацией текста и вычислением вложений с помощью Word2vec. Далее мы снова дополняем/усекаем последовательности до 400 токенов, применяя описанные в предыдущих главах функции (листинг 9.2).

Листинг 9.2. Загрузка и подготовка данных IMDB

```
>>> import numpy as np

>>> dataset = pre_process_data('./aclimdb/train')
>>> vectorized_data = tokenize_and_vectorize(dataset)
>>> expected = collect_expected(dataset)
>>> split_point = int(len(vectorized_data) * .8)

>>> x_train = vectorized_data[:split_point]
>>> y_train = expected[:split_point]
>>> x_test = vectorized_data[split_point:]
>>> y_test = expected[split_point:]

>>> maxlen = 400
>>> batch_size = 32
```

Получаем и подготавливаем данные

Разбиваем данные на тренировочный и тестовый наборы

Объявляем гиперпараметры

Количество демонстрируемых сети примеров данных перед обратным распространением ошибки и обновлением весов

```
>>> embedding_dims = 300
>>> epochs = 2
```

Длина векторов токенов, создаваемых для передачи сверточной нейронной сети

```
>>> x_train = pad_trunc(x_train, maxlen)
>>> x_test = pad_trunc(x_test, maxlen)
>>> x_train = np.reshape(x_train,
        (len(x_train), maxlen, embedding_dims))
>>> y_train = np.array(y_train)
>>> x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims))
>>> y_test = np.array(y_test)
```

Дальнейшая предварительная обработка данных путем выравнивания длин точек данных

Преобразуем в структуру данных NumPy с изменением формы

Теперь можно создать модель с новым LSTM-слоем, как показано в листинге 9.3.

Листинг 9.3. Создание LSTM-сети Keras

```
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Dropout, Flatten, LSTM
>>> num_neurons = 50
>>> model = Sequential()
>>> model.add(LSTM(num_neurons, return_sequences=True,
...               input_shape=(maxlen, embedding_dims)))
>>> model.add(Dropout(.2))
>>> model.add(Flatten())
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 400, 50)           70200
_________________________________________________________________
dropout_2 (Dropout)          (None, 400, 50)           0
_________________________________________________________________
flatten_2 (Flatten)          (None, 20000)             0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 20001
=================================================================
Total params: 90,201.0
Trainable params: 90,201.0
Non-trainable params: 0.0
```

Keras значительно упрощает реализацию всего этого

Схлопываем выходной сигнал LSTM

Слой из одного нейрона, выдающий на выходе значение с плавающей точкой в диапазоне от 0 до 1

Обучаем и сохраняем модель, как и раньше (листинги 9.4, 9.5).

Листинг 9.4. Обучение LSTM-модели

```
>>> model.fit(x_train, y_train,
...           batch_size=batch_size,
...           epochs=epochs,
...           validation_data=(x_test, y_test))
Train on 20000 samples, validate on 5000 samples
Epoch 1/2
20000/20000 [==============================] - 548s - loss: 0.4772 -
acc: 0.7736 - val_loss: 0.3694 - val_acc: 0.8412
```

Обучение модели

```
Epoch 2/2
20000/20000 [==============================] - 583s - loss: 0.3477 -
acc: 0.8532 - val_loss: 0.3451 - val_acc: 0.8516
<keras.callbacks.History at 0x145595fd0>
```

Листинг 9.5. Сохраняем модель на будущее

```
>>> model_structure = model.to_json()
>>> with open("lstm_model1.json", "w") as json_file:
... json_file.write(model_structure)

>>> model.save_weights("lstm_weights1.h5")
```

Сохраняем структуру модели, чтобы не повторять эту часть вычислений заново

Это огромный скачок точности на проверочном наборе данных по сравнению с простой RNN, которую мы реализовали в главе 8 для того же набора данных. Сразу видно, как много можно добиться, обеспечив модель памятью, когда взаимосвязи токенов столь важны. Вся прелесть этого алгоритма состоит в том, что он выучивает *взаимосвязи* встреченных им токенов. Теперь наша сеть способна моделировать подобные взаимосвязи, в частности, в контексте задаваемой функции стоимости.

Настолько мы близки в данном случае к правильному распознаванию положительной и отрицательной тональностей? Разумеется, это лишь один аспект более масштабной задачи обработки естественного языка. Как смоделировать юмор, сарказм или гнев, например? Можно ли их смоделировать вместе? Это область, в которой сейчас проводятся активные исследования. Работать с тональностями нужно по отдельности, при этом требуется большое количество маркированных вручную данных (а их сейчас вокруг с каждым днем все больше). Безусловно, такой путь заслуживает внимания. Объединение в конвейер подобных дискретных классификаторов — вполне допустимый способ поиска решений в пространстве конкретной узкой задачи.

9.1.2. А как же проверка на практике?

Здесь и начинается самое интересное. При наличии обученной модели можно использовать различные фразы и увидеть своими глазами, насколько хорошо работает модель. Попытайтесь обмануть ее. Используйте «радостные» слова в негативном контексте. Попробуйте вводить длинные, короткие и противоречивые фразы (листинги 9.6, 9.7).

Листинг 9.6. Повторная загрузка LSTM-модели

```
>>> from keras.models import model_from_json
>>> with open("lstm_model1.json", "r") as json_file:
...     json_string = json_file.read()
>>> model = model_from_json(json_string)

>>> model.load_weights('lstm_weights1.h5')
```

Листинг 9.7. Выполнение предсказания для примера данных с помощью нашей модели

```
>>> sample_1 = """I hate that the dismal weather had me down for so long,
...  when will it break! Ugh, when does happiness return? The sun is
...  blinding and the puffy clouds are too thin. I can't wait for the
...  weekend."""

>>> vec_list = tokenize_and_vectorize([(1, sample_1)])

>>> test_vec_list = pad_trunc(vec_list, maxlen)

>>> test_vec = np.reshape(test_vec_list,
...                       (len(test_vec_list), maxlen, embedding_dims))

>>> print("Sample's sentiment, 1 - pos, 2 - neg : {}"\
...     .format(model.predict_classes(test_vec)))
1/1 [==============================] - 0s
Sample's sentiment, 1 - pos, 2 - neg : [[0]]

>>> print("Raw output of sigmoid function: {}"\
...     .format(model.predict(test_vec)))
Raw output of sigmoid function: [[ 0.2192785]]
```

Процедура токенизации возвращает список данных (в этом случае — длины 1)

Мы передаем фиктивное значение в качестве первого элемента кортежа, поскольку это требуется нашей вспомогательной функции, исходя из способа обработки исходных данных. Это значение никогда не попадет в нашу нейронную сеть, так что можно указать любое

При экспериментах с разными возможностями посмотрите на чистый выходной сигнал сигма-функции, помимо дискретных классификаций тональностей. В отличие от `.predict_class()` метод `.predict()` выводит чистый результат сигма-функции активации перед учетом порогового значения, так что мы получаем вещественное значение в диапазоне от 0 до 1. Значение выше 0,5 классифицируется как позитивное, ниже 0,5 — как негативное. По мере экспериментов с примерами данных вам станет понятно, насколько уверена модель в своих предсказаниях, что поможет при анализе результатов ваших выборочных проверок.

Пристальное внимание уделяйте неправильно классифицированным примерам данных (как положительным, так и отрицательным). Если выходной сигнал сигма-функции близок к 0,5, значит, модель как будто подкидывает монетку для этого примера. В таком случае стоит посмотреть, почему данная фраза выглядит для модели неоднозначно, причем по возможности с точки зрения машины, а не человека. Забудьте на минуту про свою интуицию и субъективное мнение и попытайтесь мыслить статистически. Вспомните, какие документы модель уже «видела». Являются ли слова из неправильно классифицированного примера данных редко встречающимися? Редко ли они попадаются в вашем корпусе или корпусе, на котором обучалась языковая модель для ваших вложений? Включает ли словарь модели все слова из этого примера?

Пошаговый анализ вероятностей и входных данных, при которых происходят некорректные предсказания, поможет вам интуитивно понимать машинное обучение, а значит, создавать лучшие конвейеры NLP. Это своего рода обратное распространение ошибки по человеческому мозгу для задачи тонкой настройки модели.

9.1.3. «Грязные» данные

У этой более мощной модели тоже есть множество гиперпараметров, с которыми можно экспериментировать. Но сейчас самое время сделать паузу и вернуться к нашим данным. Мы использовали одни и те же одинаково обработанные данные еще с самого начала обсуждения сверточных нейронных сетей именно для того, чтобы показать различия типов моделей и их эффективности на одном наборе данных. Но при этом некоторые из принятых нами решений могли нарушить целостность данных, *«загрязнить»* их.

Дополнение/усечение примеров данных до 400 токенов было необходимо для сверточных нейронных сетей, чтобы фильтры «просматривали» векторы одинаковой длины и выходной сигнал сверточной сети тоже представлял собой вектор постоянной длины. Одинаковая размерность выходного вектора важна, поскольку выходной сигнал попадает в конце цепочки на вход полносвязного слоя прямого распространения, которому нужен входной сигнал фиксированной длины.

Аналогично и наши реализации рекуррентных нейронных сетей (простой и LSTM) стремились добиться *вектора идеи* фиксированной длины, который можно было бы передать в слой прямого распространения для классификации. Векторное представление объекта фиксированной длины, например вектор идеи, называется также *вложением* (embedding). Если же длина вектора идеи одинакова, то *развертывать* сеть тоже придется на аналогичное число временных шагов (токенов). Исследуем внимательнее вариант 400 в качестве количества временных шагов развертывания сети, как показано в листинге 9.8.

Листинг 9.8. Оптимизация размера вектора идеи

```
>>> def test_len(data, maxlen):
...     total_len = truncated = exact = padded = 0
...     for sample in data:
...         total_len += len(sample)
...         if len(sample) > maxlen:
...             truncated += 1
...         elif len(sample) < maxlen:
...             padded += 1
...         else:
...             exact +=1
...     print('Padded: {}'.format(padded))
...     print('Equal: {}'.format(exact))
...     print('Truncated: {}'.format(truncated))
...     print('Avg length: {}'.format(total_len/len(data)))

>>> dataset = pre_process_data('./aclimdb/train')
>>> vectorized_data = tokenize_and_vectorize(dataset)
>>> test_len(vectorized_data, 400)
Padded: 22559
Equal: 12
Truncated: 2429
Avg length: 202.4424
```

Ух ты! Ладно, 400 — многовато (наверное, следовало проанализировать это раньше). Уменьшим `maxlen` до значения ближе к средней длине примера данных — 202 токена. Округлим его до 200 токенов и запустим еще раз LSTM с этим значением, как показано в листингах 9.9–9.11.

Листинг 9.9. Оптимизация гиперпараметров LSTM

```
>>> import numpy as np
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Dropout, Flatten, LSTM
>>> maxlen = 200
>>> batch_size = 32
>>> embedding_dims = 300
>>> epochs = 2
>>> num_neurons = 50
>>> dataset = pre_process_data('./aclimdb/train')
>>> vectorized_data = tokenize_and_vectorize(dataset)
>>> expected = collect_expected(dataset)
>>> split_point = int(len(vectorized_data)*.8)
>>> x_train = vectorized_data[:split_point]
>>> y_train = expected[:split_point]
>>> x_test = vectorized_data[split_point:]
>>> y_test = expected[split_point:]
>>> x_train = pad_trunc(x_train, maxlen)
>>> x_test = pad_trunc(x_test, maxlen)
>>> x_train = np.reshape(x_train, (len(x_train), maxlen, embedding_dims))
>>> y_train = np.array(y_train)
>>> x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims))
>>> y_test = np.array(y_test)
```

Тот же код, что и раньше, только максимальная длина уменьшена до 200 токенов

Листинг 9.10. LSTM оптимального размера

```
>>> model = Sequential()
>>> model.add(LSTM(num_neurons, return_sequences=True,
...                input_shape=(maxlen, embedding_dims)))
>>> model.add(Dropout(.2))
>>> model.add(Flatten())
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_3 (LSTM)                (None, 200, 50)           70200
_________________________________________________________________
dropout_3 (Dropout)          (None, 200, 50)           0
_________________________________________________________________
flatten_3 (Flatten)          (None, 10000)             0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 10001
=================================================================
Total params: 80,201.0
Trainable params: 80,201.0
Non-trainable params: 0.0
```

Листинг 9.11. Обучение более маленькой LSTM

```
>>> model.fit(x_train, y_train,
...           batch_size=batch_size,
...           epochs=epochs,
...           validation_data=(x_test, y_test))
Train on 20000 samples, validate on 5000 samples
Epoch 1/2
20000/20000 [==============================] - 245s - loss: 0.4742 -
acc: 0.7760 - val_loss: 0.4235 - val_acc: 0.8010
Epoch 2/2
20000/20000 [==============================] - 203s - loss: 0.3718 -
acc: 0.8386 - val_loss: 0.3499 - val_acc: 0.8450

>>> model_structure = model.to_json()
>>> with open("lstm_model7.json", "w") as json_file:
...     json_file.write(model_structure)
>>> model.save_weights("lstm_weights7.h5")
```

Ну что ж, эта модель обучилась гораздо быстрее, а точность на проверочных данных упала незначительно (84,50 вместо 85,16 %)[1]. С примерами данных, уменьшающими число временных шагов вдвое, время обучения сократилось более чем наполовину! Вычисляемых временных шагов LSTM и обучаемых весовых коэффициентов стало вдвое меньше. Но главное — каждый раз обратное распространение ошибки производится на расстояние, вдвое меньшее (половина временных шагов в прошлое).

Впрочем, точность упала. Но разве 200D-модель не должна обобщать лучше (переобучаться меньше), чем предыдущая 400-мерная? Дело в наличии слоя дропаута в обеих моделях. Слой дропаута предотвращает переобучение, поэтому точность на проверочных данных будет лишь падать по мере снижения числа степеней свободы или эпох обучения модели.

Несмотря на возможности нейронных сетей и их способности усваивать сложные паттерны, легко забыть, что разумно спроектированная нейронная сеть хорошо обучается игнорировать шум и систематические ошибки (смещение). Мы нечаянно внесли в данные систематическую ошибку, добавив нулевые векторы. Элементы смещения во всех узлах генерируют определенный сигнал даже при всех нулевых входных сигналах. Но сеть постепенно обучается полностью игнорировать эти элементы (говоря конкретнее, путем подгонки весового коэффициента этого элемента смещения к нулю) и сосредотачивать внимание на содержащих осмысленную информацию частях примеров данных.

Наша оптимизированная LSTM усваивает (learn) не больше, но намного быстрее. Самый важный вывод из этого: необходимо учитывать длину проверочных примеров данных относительно длины тренировочных. Если тренировочный набор данных состоит из документов длиной тысячи токенов, вряд ли можно рассчитывать на правильную классификацию примера данных длиной три токена, дополненного до 1000. И наоборот — усечение опуса на 1000 токенов до трех ничего хорошего вашей маленькой модели не принесет. Не то чтобы LSTM не смогла ничего из него извлечь; просто учтите это при своих экспериментах.

[1] Да и оперативной памяти потребовалось существенно меньше. — *Примеч. пер.*

9.1.4. Возвращаемся к «грязным» данным

Какова самая большая оплошность при обработке данных? Отбрасывание неизвестных токенов. Список неизвестных, то есть просто слов, отсутствующих в предобученной модели Word2vec, достаточно велик. Отбрасывание такого количества данных, особенно при моделировании последовательности слов, ни к чему хорошему не приведет.

Если ваш словарь вложений слов не содержит *don't*, предложение вроде:

I don't like this movie.

может превратиться в:

I like this movie.

Для вложений Word2vec этот пример неактуален, но опускаются многие токены, которые могут оказаться важны. Отбрасывание этих неизвестных токенов — лишь одна из возможных стратегий. Можно использовать или обучить вложение слов, в котором есть векторы для всех токенов до последнего, но это почти всегда оказывается слишком дорогостоящим.

Существует два распространенных подхода, показывающих неплохие результаты без больших вычислительных затрат. Оба включают замену неизвестного токена новым векторным представлением. Первый кажется парадоксальным: для каждого не смоделированного вектором токена используется случайным образом выбранный вектор из существующей модели. Очевидно, что подобный подход привел бы в замешательство читателя-человека.

Предложение вроде

The man who was defenestrated, brushed himself off with a nonchalant glance back inside.

может превратиться в

The man who was duck, brushed himself off with a airplane glance back inside.

Как же модель может усвоить что-либо из подобной бессмыслицы? Оказывается, что модель преодолевает такие помехи практически так же, как и при отбрасывании их в примере выше. Помните, что мы не стремимся смоделировать все высказывания из тренировочного набора данных явным образом. Наша цель — создать обобщенную модель языка тренировочного набора данных. Так что аномалии неизбежны, но можно надеяться, что они не помешают модели описывать основные закономерности.

Второй, более распространенный, подход состоит в замене при воссоздании исходного входного сигнала всех отсутствующих в библиотеке векторов слов токенов конкретным токеном, обычно обозначаемым UNK[1]. Сам вектор выбирается либо при моделировании исходного вложения, либо случайным образом (и желательно подальше от всех известных векторов в пространстве).

Как и в случае дополнения, сеть способна обучиться обходить неизвестные токены и приходить к своим собственным заключениям, несмотря на них.

---

[1] От англ. unknown — «неизвестный». — *Примеч. пер.*

9.1.5. Работать со словами сложно. С отдельными буквами — проще

Слова обладают смыслом — с этим все согласны, а значит, вполне закономерно моделировать естественный язык с помощью простейших стандартных блоков. Точно так же в порядке вещей использовать эти модели для описания смысла, чувства, подтекста и всего остального в терминах подобных атомарных структур. Но, разумеется, слова вовсе не атомарны. Как вы видели, они состоят из меньших слов, основ, фонем и т. д., а если смотреть глубже, они состоят из символов.

При моделировании языка необходимо учитывать, что на уровне символов скрыто немало смысла. Голосовые интонации, аллитерации, рифмы — все это можно моделировать, если перейти с уровня слов на уровень символов. Люди способны моделировать все это и без подобного разбиения слов. Но возникающие вследствие такого моделирования определения слишком сложны, и наделить ими машину непросто, а именно это нас интересует. Многие из закономерностей заметны в тексте при посимвольном его изучении.

В такой парадигме пробел/запятая/точка — лишь очередной символ. И модели придется искать все эти низкоуровневые паттерны при усвоении сетью смысла предложений, разбитых на отдельные символы. Повторяющийся суффикс после определенного числа, вероятно, рифмующихся слогов вполне может оказаться закономерностью, несущей определенный смысл, возможно веселье или насмешку. Подобные закономерности начинают проясняться при достаточно большом тренировочном наборе данных, а поскольку количество букв в английском языке[1] намного меньше, чем слов, то приходится волноваться о меньшем числе входных векторов.

Впрочем, обучение модели на уровне символов — задача непростая. Закономерности и «длинные» зависимости, найденные на уровне символов, могут сильно варьироваться из-за произношения. Но даже если найти эти закономерности, не факт, что удастся их обобщить. Воспользуемся LSTM на уровне символов с тем же примером набора данных. Во-первых, нам придется по-другому обрабатывать данные. Как и ранее, мы извлекаем данные и отбираем метки, как показано в листинге 9.12.

Листинг 9.12. Подготовка данных

```
>>> dataset = pre_process_data('./aclimdb/train')
>>> expected = collect_expected(dataset)
```

Теперь нам нужно решить, насколько далеко развертывать сеть. Теперь, как показано в листинге 9.13, выясним, сколько символов в среднем содержится в примерах данных.

Листинг 9.13. Вычисление средней длины примера данных

```
>>> def avg_len(data):
...     total_len = 0
...     for sample in data:
...         total_len += len(sample[1])
```

[1] Да и в русском тоже. — *Примеч. пер.*

```
...     return total_len/len(data)

>>> avg_len(dataset)
1325.06964
```

И сразу же видно, что сеть будет разворачиваться намного дальше, а значит, придется гораздо дольше ждать окончания работы этой модели. Спойлер: эта модель ничего не делает, кроме как переобучается, тем не менее она интересна в качестве примера.

Далее нам нужно очистить данные от токенов, не относящихся к естественному языку текста. Следующая функция отфильтровывает некоторые бесполезные символы HTML-тегов из набора данных. На самом деле эти данные необходимо очистить более тщательно (листинг 9.14).

Листинг 9.14. Подготовка строковых значений для модели на основе отдельных символов

```
>>> def clean_data(data):
...     """Переводим в нижний регистр, заменяя неизвестные токены на UNK,
...     и преобразуем в список"""
...     new_data = []
...     VALID = 'abcdefghijklmnopqrstuvwxyz0123456789"\'?!.,:; '
...     for sample in data:
...         new_sample = []
...         for char in sample[1].lower():   <-- Берем только строку, но не метку
...             if char in VALID:
...                 new_sample.append(char)
...             else:
...                 new_sample.append('UNK')
...         new_data.append(new_sample)
...     return new_data

>>> listified_data = clean_data(dataset)
```

Для всего, что отсутствует в списке VALID, в качестве отдельного символа используется `'UNK'`.

Затем, как и раньше, мы дополняем или усекаем примеры данных до заданной `maxlen`. Здесь же появляется еще один отдельный символ для дополнения: `'PAD'` (листинг 9.15).

Листинг 9.15. Дополнение и усечение символов

```
>>> def char_pad_trunc(data, maxlen=1500):
...     """ Производим усечение до maxlen или добавляем токены PAD"""
...     new_dataset = []
...     for sample in data:
...         if len(sample) > maxlen:
...             new_data = sample[:maxlen]
...         elif len(sample) < maxlen:
...             pads = maxlen - len(sample)
...             new_data = sample + ['PAD'] * pads
...         else:
...             new_data = sample
...         new_dataset.append(new_data)
...     return new_dataset
```

Мы выбрали равную 1500 `maxlen`, чтобы захватить чуть больше данных, чем содержится в среднем примере данных, но пытаемся избежать внесения слишком большого шума с помощью символов `'PAD'`. Имеет смысл рассмотреть эти варианты с точки зрения размера слов. При фиксированной длине в символах пример данных, содержащий множество длинных слов, окажется недостаточно представленным, по сравнению с примером данных, состоящим целиком из простых, односложных слов. Как и в любой задаче машинного обучения, важно хорошо знать свой набор данных и все его нюансы.

На этот раз вместо Word2vec для вложений мы воспользуемся унитарным кодированием символов. Нам нужно создать словарь соответствий токенов (символов) целочисленным индексам. Мы также создадим словарь для обратных соответствий, но об этом позднее (листинг 9.16).

Листинг 9.16. Словарь для модели на основе отдельных символов

```
>>> def create_dicts(data):
...     """ Модифицированный фрагмент примера LSTM Keras"""
...     chars = set()
...     for sample in data:
...         chars.update(set(sample))
...     char_indices = dict((c, i) for i, c in enumerate(chars))
...     indices_char = dict((i, c) for i, c in enumerate(chars))
...     return char_indices, indices_char
```

Теперь можно использовать этот словарь для создания входных векторов индексов вместо самих токенов, как показано в листингах 9.17, 9.18.

Листинг 9.17. Процедура унитарного кодирования символов

```
>>> import numpy as np

>>> def onehot_encode(dataset, char_indices, maxlen=1500):
...     """
...     Унитарное кодирование токенов
...
...     Аргументы:
...               dataset список списков токенов
...               char_indices
...                       dictionary of {key=символ,
...                                      value=индекс}
...               maxlen int Длина примера данных
...     Возвращает:
...               Массив NumPy формы (примеры данных, токены, длина кодирования)
...     """
...     X = np.zeros((len(dataset), maxlen, len(char_indices.keys())))
...     for i, sentence in enumerate(dataset):
...         for t, char in enumerate(sentence):
...             X[i, t, char_indices[char]] = 1
...     return X
```

Массив NumPy длины, равной числу примеров данных — каждый пример данных будет состоять из равного maxlen количества токенов, а каждый токен будет унитарным вектором длины, равной числу символов

Листинг 9.18. Загрузка и предварительная обработка данных IMDB

```
>>> dataset = pre_process_data('./aclimdb/train')
>>> expected = collect_expected(dataset)
>>> listified_data = clean_data(dataset)

>>> common_length_data = char_pad_trunc(listified_data, maxlen=1500)
>>> char_indices, indices_char = create_dicts(common_length_data)
>>> encoded_data = onehot_encode(common_length_data, char_indices, 1500)
```

А теперь, как и ранее, разбиваем данные так, как показано в листингах 9.19, 9.20.

Листинг 9.19. Разбиваем набор данных на тренировочный (80 %) и тестовый (20 %)

```
>>> split_point = int(len(encoded_data)*.8)

>>> x_train = encoded_data[:split_point]
>>> y_train = expected[:split_point]
>>> x_test = encoded_data[split_point:]
>>> y_test = expected[split_point:]
```

Листинг 9.20. Создаем символьную LSTM

```
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Dropout, Embedding, Flatten, LSTM

>>> num_neurons = 40
>>> maxlen = 1500
>>> model = Sequential()

>>> model.add(LSTM(num_neurons,
...                return_sequences=True,
...                input_shape=(maxlen, len(char_indices.keys()))))
>>> model.add(Dropout(.2))
>>> model.add(Flatten())
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 1500, 40)          13920
_________________________________________________________________
dropout_1 (Dropout)          (None, 1500, 40)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 60000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 60001
=================================================================
Total params: 73,921.0
Trainable params: 73,921.0
Non-trainable params: 0.0
```

Что ж, мы добились определенных успехов в создании LSTM-моделей. Нашей последней модели на основе отдельных символов необходимо подобрать лишь 74 000 параметров, по сравнению с оптимизированной LSTM-моделью на основе слов, которой требовалось 80 000. И эта упрощенная модель должна быстрее обучаться и лучше обобщаться в отношении нового текста, поскольку у нее меньше степеней свободы для переобучения.

Попробуем теперь ее в деле и в листингах 9.21 и 9.22 посмотрим, что могут нам предложить LSTM-модели на основе отдельных символов.

Листинг 9.21. Обучаем символьную LSTM-модель

```
>>> batch_size = 32
>>> epochs = 10
>>> model.fit(x_train, y_train,
...           batch_size=batch_size,
...           epochs=epochs,
...           validation_data=(x_test, y_test))
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 634s - loss: 0.6949 -
acc: 0.5388 - val_loss: 0.6775 - val_acc: 0.5738
Epoch 2/10
20000/20000 [==============================] - 668s - loss: 0.6087 -
acc: 0.6700 - val_loss: 0.6786 - val_acc: 0.5962
Epoch 3/10
20000/20000 [==============================] - 695s - loss: 0.5358 -
acc: 0.7356 - val_loss: 0.7182 - val_acc: 0.5786
Epoch 4/10
20000/20000 [==============================] - 686s - loss: 0.4662 -
acc: 0.7832 - val_loss: 0.7605 - val_acc: 0.5836
Epoch 5/10
20000/20000 [==============================] - 694s - loss: 0.4062 -
acc: 0.8206 - val_loss: 0.8099 - val_acc: 0.5852
Epoch 6/10
20000/20000 [==============================] - 694s - loss: 0.3550 -
acc: 0.8448 - val_loss: 0.8851 - val_acc: 0.5842
Epoch 7/10
20000/20000 [==============================] - 645s - loss: 0.3058 -
acc: 0.8705 - val_loss: 0.9598 - val_acc: 0.5930
Epoch 8/10
20000/20000 [==============================] - 684s - loss: 0.2643 -
acc: 0.8911 - val_loss: 1.0366 - val_acc: 0.5888
Epoch 9/10
20000/20000 [==============================] - 671s - loss: 0.2304 -
acc: 0.9055 - val_loss: 1.1323 - val_acc: 0.5914
Epoch 10/10
20000/20000 [==============================] - 663s - loss: 0.2035 -
acc: 0.9181 - val_loss: 1.2051 - val_acc: 0.5948
```

Листинг 9.22. И сохраняем ее на будущее

```
>>> model_structure = model.to_json()
>>> with open("char_lstm_model3.json", "w") as json_file:
...     json_file.write(model_structure)
>>> model.save_weights("char_lstm_weights3.h5")
```

Точность 92 % на тренировочном наборе данных при точности 59 % на проверочных подтверждает переобучение. Модель понемногу начала усваивать тональность тренировочного набора данных. Причем очень медленно. На современном ноутбуке без GPU это заняло 1,5 часа. Но точность на проверочном наборе данных так никогда и не стала намного лучше точности угадывания, а позже даже ухудшилась, как видно из значения функции потерь.

Причин этого может быть множество. Например, слишком полная для конкретного набора данных модель. То есть в ней достаточное количество параметров для моделирования закономерностей, присущих сугубо 20 000 примеров из выборки тренировочного набора данных, но не подходящих для общей модели тональностей языка. Эту проблему можно сгладить с помощью повышения процента дропаута или уменьшения числа нейронов в LSTM-слое. Улучшить ситуацию со слишком полно описанной моделью могут также дополнительные маркированные данные, но качественные маркированные данные обычно найти сложнее всего.

В конечном счете эта модель требует немалых затрат — как вычислительных ресурсов, так и времени — при весьма умеренном результате по сравнению с LSTM-моделью на уровне слов или даже со сверточными нейронными сетями из предыдущих глав. Так зачем вообще возиться с уровнем отдельных символов? Модели на основе отдельных символов способны очень хорошо моделировать язык при достаточно широком тренировочном наборе данных. Или моделировать конкретный вид языка по узконаправленному тренировочному набору данных, допустим по одному автору вместо тысяч. В любом случае мы сделали первый шаг к генерации нового текста с помощью нейронной сети.

9.1.6. Моя очередь говорить

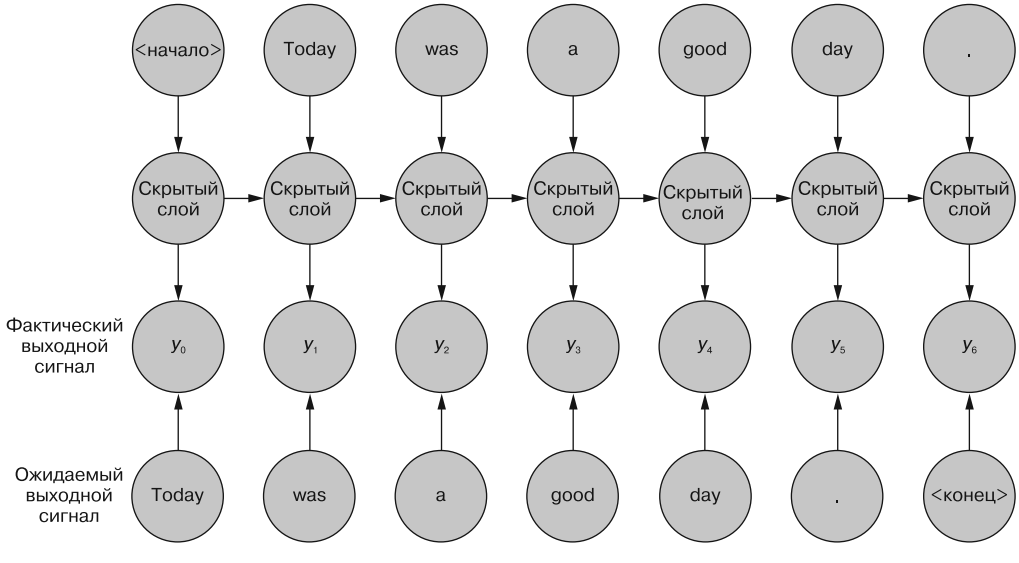
Благодаря генерации нового текста в определенном «стиле» или с определенными «взглядами» наверняка можно создать чат-бот, который будет интересным собеседником. Конечно, возможность генерации нового текста в заданном стиле не гарантирует, что этот бот сможет разговаривать о чем угодно. Но с помощью такого подхода можно сгенерировать текст с заданными параметрами (например, в ответ на определенный стиль сообщений пользователя) и проиндексировать этот больший корпус нового текста для поиска в нем возможных ответов на конкретный запрос.

Подобно цепи Маркова, предсказывающей следующее слово в последовательности на основе вероятностей появления слов после только что встреченной одограммы, биграммы или n-граммы, LSTM-модель может выяснить вероятность появления следующего слова по только что просмотренным словам, но

с дополнительным преимуществом в виде *памяти*! У марковской цепи есть только информация о текущей *n*-грамме и частотностях встречающихся после нее слов. Модель RNN тоже кодирует информацию о следующем терме на основе нескольких предшествующих. Но благодаря состоянию памяти LSTM имеющийся у модели контекст для определения наиболее подходящего следующего слова оказывается гораздо шире. И что самое интересное, можно предсказывать новые символы по предыдущим. Подобный уровень детализации для простых марковских цепей недостижим.

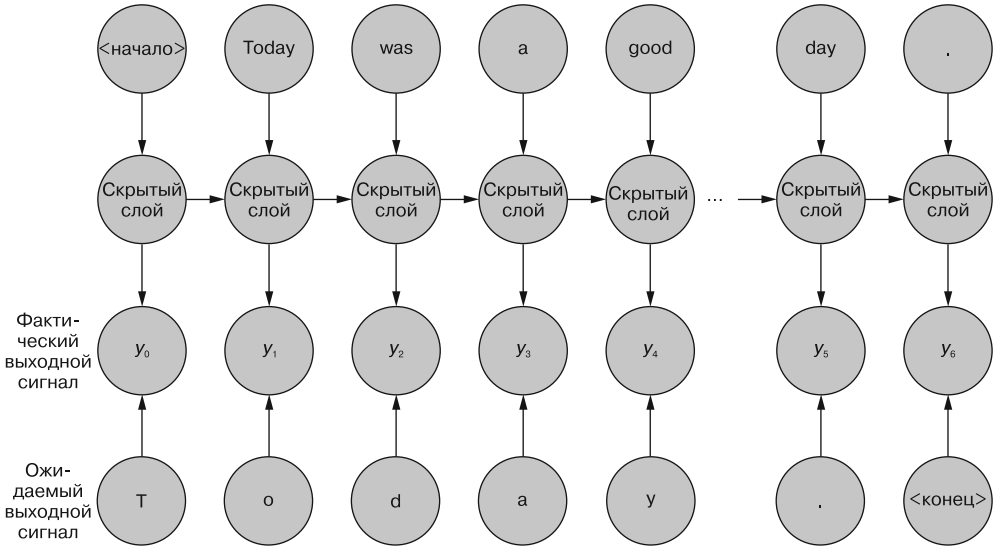
Но как обучить модель этому фокусу? Во-первых, отойдем от задачи классификации. Ядро усвоенной LSTM информации заключается в самой LSTM-ячейке. Но мы обучали ее на успехах и неудачах в конкретной задаче классификации. Не факт, что подобный подход поможет модели усвоить общее представление языка. Мы обучили ее обращать внимание только на предложения, содержащие ярко выраженную тональность.

Поэтому вместо меток тональностей тренировочного набора данных воспользуемся в качестве целевой переменной обучения самими тренировочными примерами данных! Мы хотели бы обучить LSTM *предсказывать* по заданному токену в примере данных следующий токен (рис. 9.10). Это напоминает подход с вложениями векторов слов из главы 6, но мы будем обучать сеть на биграммах, а не на skip-граммах. Обученная подобным образом модель генератора слов (см. рис. 9.10) работает вполне пристойно, но перейдем к сути дела и обратимся к уровню отдельных символов с тем же подходом (рис. 9.11).



Ожидаемый выходной сигнал — следующий токен примера данных. Здесь показано на уровне слов

Рис. 9.10. Предсказание следующего слова



Ожидаемый выходной сигнал — следующий токен примера данных. Здесь показано на уровне символов

Рис. 9.11. Предсказание следующего символа

Нас интересуют не векторы идей на выходе последнего временно́го шага, а выходные сигналы всех временны́х шагов по отдельности. Ошибка будет по-прежнему распространяться во времени с каждого временно́го шага в самое начало, но определяться она будет теперь именно на уровне конкретного временно́го шага. По сути, в других LSTM-классификаторах из этой главы было так же, только ошибка не вычислялась вплоть до конца последовательности. Лишь в конце последовательности вычислялся агрегированный выходной сигнал для передачи в слой прямого распространения в конце цепочки. Тем не менее обратное распространение ошибки происходит аналогичным образом, с накоплением ошибок путем подстройки весов в конце последовательности.

Таким образом, прежде всего необходимо откорректировать метки тренировочного набора данных. Мы будем сверять выходной вектор не с заданной меткой классификации, а с унитарным представлением следующего символа последовательности.

Можно также довольствоваться более простой моделью: вместо предсказания последовательно, по одному символу предсказывать следующий символ для заданной последовательности. Эта модель совпадает с остальными LSTM-слоями в данной главе, если убрать поименованный аргумент `return_sequences=True` (см. листинг 9.20). При этом LSTM-модель сосредоточит внимание на возвращаемом на последнем временно́м шаге последовательности значении (рис. 9.12).

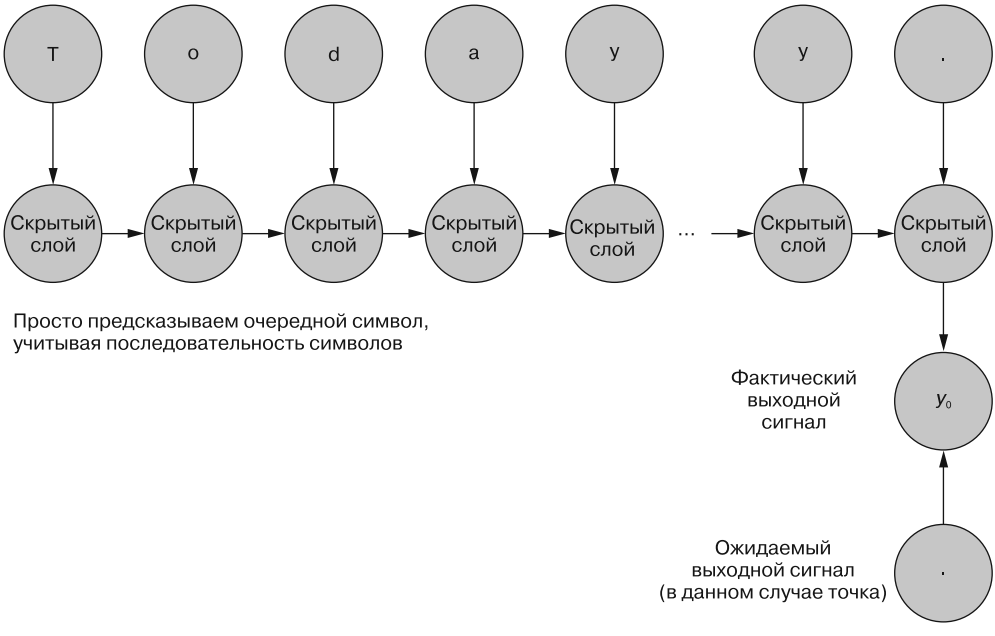


Рис. 9.12. Предсказание только последнего символа

9.1.7. Моя очередь говорить понятнее

Простое моделирование на уровне символов — ключ к более сложным моделям, способным улавливать не только орфографию, но и грамматику и пунктуацию. Вся мощь таких моделей проявляется, когда они усваивают эти грамматические нюансы и начинают улавливать ритм и модуляции текста. Рассмотрим, как генерировать новый текст с помощью использовавшихся нами для классификации инструментов.

В документации Keras приведен отличный пример. На этот раз мы не будем использовать набор данных с отзывами о фильмах. Для поиска столь глубоких понятий, как интонация и выбор слов, этот набор данных обладает двумя неудобными параметрами. Прежде всего, он разнообразен: написан множеством авторов, у каждого — свой собственный стиль и индивидуальные особенности. Найти общие черты очень сложно. Впрочем, при достаточно большом наборе данных можно создать сложную модель языка, способную справиться с разнообразием стилей. Но это ведет ко второй проблеме набора данных IMDB: он слишком мал для усвоения общей языковой модели на основе отдельных символов. Чтобы решить эту проблему, необходим набор данных с более однородными по стилю и интонации примерами данных или же гораздо больший по размеру; и мы выберем второе. Для примера Keras взят фрагмент из книги Фридриха Ницше. Это интересно, но мы выберем другого писателя со своим уникальным стилем — Уильяма Шекспира.

Он уже давненько не публиковал ничего нового, так что давайте поможем ему немного (листинг 9.23).

Листинг 9.23. Импорт набора данных проекта «Гутенберг»

```
>>> from nltk.corpus import gutenberg
>>>
>>> gutenberg.fileids()
['austen-emma.txt',
 'austen-persuasion.txt',
 'austen-sense.txt',
 'bible-kjv.txt',
 'blake-poems.txt',
 'bryant-stories.txt',
 'burgess-busterbrown.txt',
 'carroll-alice.txt',
 'chesterton-ball.txt',
 'chesterton-brown.txt',
 'chesterton-thursday.txt',
 'edgeworth-parents.txt',
 'melville-moby_dick.txt',
 'milton-paradise.txt',
 'shakespeare-caesar.txt',
 'shakespeare-hamlet.txt',
 'shakespeare-macbeth.txt',
 'whitman-leaves.txt']
```

О, три пьесы Шекспира. Склеиваем их в одну большую строку. Если же этого мало, вы легко найдете *множество* других произведений по адресу: https://www.gutenberg.org/[1] (листинг 9.24).

Листинг 9.24. Предварительная обработка пьес Шекспира

```
>>> text = ''
>>> for txt in gutenberg.fileids():
...     if 'shakespeare' in txt:
...         text += gutenberg.raw(txt).lower()
>>> chars = sorted(list(set(text)))
>>> char_indices = dict((c, i)
...     for i, c in enumerate(chars))
>>> indices_char = dict((i, c)
...     for i, c in enumerate(chars))
>>> 'corpus length: {} total chars: {}'.format(len(text), len(chars))
'corpus length: 375542 total chars: 50'
```

Склеиваем все пьесы Шекспира из корпуса проекта «Гутенберг» в NLTK

Формируем словарь соответствий символов индексам, чтобы ссылаться в унитарном представлении

Создаем обратный словарь для поиска соответствий при переводе унитарного представления обратно в символы

[1] Веб-сайт проекта «Гутенберг» содержит 57 000 отсканированных книг в различных форматах. Книги можно скачать бесплатно по запросу, для чего потребуется всего один день: https://www.exratione.com/2014/11/how-to-politely-download-all-english-language-text-format-files-from-project-gutenberg/.

Все так же аккуратно отформатировано:

```
>>> print(text[:500])
[the tragedie of julius caesar by william shakespeare 1599]

actus primus. scoena prima.

enter flauius, murellus, and certaine commoners ouer the stage.

  flauius. hence: home you idle creatures, get you home:
is this a holiday? what, know you not
(being mechanicall) you ought not walke
vpon a labouring day, without the signe
of your profession? speake, what trade art thou?
  car. why sir, a carpenter

   mur. where is thy leather apron, and thy rule?
what dost thou with thy best apparrell on
```

Далее мы собираемся разбить исходный текст на примеры данных, содержащие фиксированное, равное `maxlen`, число символов. Чтобы увеличить размер набора данных и сосредоточить внимание на одинаковых закономерностях, в примере Keras делается выборка данных с запасом, в виде полуизбыточных фрагментов. Берем 40 символов с начала, переходим к третьему символу от начала, берем 40 символов отсюда, переходим к шестому символу... и т. д.

Помните, что цель конкретно этой модели — предсказание 41-го символа в произвольной последовательности по 40 предыдущим символам. Именно поэтому мы формируем тренировочный набор данных из полуизбыточных фрагментов длиной 40 символов каждый, как показано в листинге 9.25.

Листинг 9.25. Формируем тренировочный набор данных

```
>>> maxlen = 40
>>> step = 3
>>> sentences = []
>>> next_chars = []
>>> for i in range(0, len(text) - maxlen, step):
...     sentences.append(text[i: i + maxlen])
...     next_chars.append(text[i + maxlen])
>>>     print('nb sequences:', len(sentences))
nb sequences: 125168
```

Пока игнорируем границы предложений (и строк), чтобы посимвольная модель усваивала, где заканчивается предложение: на основе точки (.) или символа перевода строки (\n)

Устанавливаем шаг три символа, чтобы сгенерированные тренировочные примеры данных пересекались, но не совпадали полностью

Получаем срез текста

Получаем следующий ожидаемый символ

Итак, у нас есть 125 168 тренировочных примеров данных и следующие за каждым из них символы, цели нашей модели (листинг 9.26).

Листинг 9.26. Унитарное кодирование тренировочных примеров данных

```
>>> X = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
>>> y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
>>> for i, sentence in enumerate(sentences):
...     for t, char in enumerate(sentence):
...         X[i, t, char_indices[char]] = 1
...     y[i, char_indices[next_chars[i]]] = 1
```

Далее производим унитарное кодирование всех символов всех примеров данных в наборе и сохраняем их в виде списка `X`. Мы также сохраняем список унитарных представлений «ответов» в списке `y`. И формируем модель, как показано в листинге 9.27.

Листинг 9.27. Формирование посимвольной LSTM-модели для генерации текста

```
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Activation
>>> from keras.layers import LSTM
>>> from keras.optimizers import RMSprop
>>> model = Sequential()
>>> model.add(LSTM(128,
...                input_shape=(maxlen, len(chars))))
>>> model.add(Dense(len(chars)))
>>> model.add(Activation('softmax'))
>>> optimizer = RMSprop(lr=0.01)
>>> model.compile(loss='categorical_crossentropy', optimizer=optimizer)
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 128)               91648
_________________________________________________________________
dense_1 (Dense)              (None, 50)                6450
_________________________________________________________________
activation_1 (Activation)    (None, 50)                0
=================================================================
Total params: 98,098.0
Trainable params: 98,098.0
Non-trainable params: 0.0
```

Используем более широкий LSTM-слой — 128 вместо 50. И не возвращаем последовательность. Нас интересует только последний выходной символ

Это задача классификации, так что нам требуется распределение вероятности по всем возможным символам

Выглядит немного не так, как раньше, поэтому изучим отдельные компоненты. Последовательный и LSTM-слои такие же, как и ранее, в случае классификатора. Количество нейронов (параметр `num_neurons`) в скрытом слое LSTM-ячейки теперь равно 128. Это существенно больше, чем было в классификаторе, но ведь мы хотим смоделировать более сложное поведение, связанное с воспроизведением интонации заданного текста. Далее оптимизатор описан с помощью переменной, но это тот же оптимизатор, что использовался до сих пор. Мы выделили его для наглядности, поскольку изменили значение параметра скорости обучения (равное по умолчанию `.001`). Как бы там ни было, оптимизатор RMSprop работает путем обновления каждого из весов путем деления скорости обучения на «скользящее среднее модулей

последних градиентов для этого веса»[1]. Наличие информации об оптимизаторах, безусловно, может предотвратить возникновение проблем при экспериментах, но подробное описание устройства отдельных оптимизаторов выходит за рамки данной книги.

Следующее серьезное отличие — минимизируемая функция потерь. До сих пор ее роль играла `binary_crossentropy`. Мы пытались лишь определить уровень, на котором срабатывает отдельный нейрон. Но в последнем слое мы заменили `Dense(1)` на `Dense(len(chars))`. Поэтому выходной сигнал сети на каждом из временны́х шагов представляет собой 50-мерный вектор (`len(chars) == 50` в листинге 9.20). В качестве функции активации используется `softmax`, так что выходной вектор будет эквивалентен распределению вероятности по всему 50-мерному вектору (сумма значений элементов вектора всегда равна 1). Использование в качестве функции потерь `categorical_crossentropy` означает минимизацию разницы между итоговым распределением вероятности и унитарным представлением ожидаемого символа.

И последнее существенное изменение — отсутствие дропаута. Поскольку мы хотим смоделировать именно конкретный набор данных, то обобщение на другие задачи нас не интересует. Таким образом, переобучение не только допустимо, оно желательно (листинг 9.28).

Листинг 9.28. Обучение нашего шекспировского чат-бота

```
>>> epochs = 6
>>> batch_size = 128
>>> model_structure = model.to_json()
>>> with open("shakes_lstm_model.json", "w") as json_file:
>>>     json_file.write(model_structure)
>>> for i in range(5):
...     model.fit(X, y,
...     batch_size=batch_size,
...     epochs=epochs)
...     model.save_weights("shakes_lstm_weights_{}.h5".format(i+1))
Epoch 1/6
125168/125168 [==============================] - 266s - loss: 2.0310
Epoch 2/6
125168/125168 [==============================] - 257s - loss: 1.6851
...
```

Обучаем модель в течение некоторого времени, сохраняем ее состояние и снова продолжаем обучение. В Keras есть также встроенная функция обратного вызова, производящая аналогичные операции

При этой схеме работы модель сохраняется каждые шесть эпох, после чего продолжает обучаться. Если значение функции потерь более не уменьшается, не стоит тратить ресурсы на дальнейшее обучение, так что можно спокойно остановить процесс и получить сохраненный набор весов, устаревший лишь на несколько эпох. Мы обнаружили, что для получения более или менее приличных результатов от этого

[1] Хинтон (Hinton) и др. (http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf).

набора данных потребуется от 20 до 30 эпох. Возможно, имеет смысл расширить набор данных. Книги Шекспира находятся в свободном доступе и являются общественным достоянием[1]. Главное, старайтесь добиться согласованности за счет соответствующей предварительной обработки, если вы берете их из различных источников. К счастью, при работе с моделями на основе отдельных символов не приходится волноваться из-за токенизаторов и сегментаторов предложений, но выравнивание регистра может сыграть важную роль. Мы воспользовались, так сказать, кувалдой. Не исключено, что более аккуратный подход даст лучшие результаты.

Теперь напишем свою пьесу! Поскольку выходные векторы представляют собой 50-мерные векторы, описывающие распределение вероятности по 50 возможным выходным символам, можно произвести выборку из этого распределения. В примере Keras для этого есть вспомогательная функция, как показано в листинге 9.29.

Листинг 9.29. Выборка для генерации символьных последовательностей

```
>>> import random
>>> def sample(preds, temperature=1.0):
...     preds = np.asarray(preds).astype('float64')
...     preds = np.log(preds) / temperature
...     exp_preds = np.exp(preds)
...     preds = exp_preds / np.sum(exp_preds)
...     probas = np.random.multinomial(1, preds, 1)
...     return np.argmax(probas)
```

Поскольку последний слой сети — многомерная логистическая функция, выходной вектор представляет собой распределение вероятности по всем возможным выходным сигналам сети. Из максимального значения выходного вектора видно, какой следующий символ сеть считает наиболее вероятным. Говоря точнее, индекс выходного вектора, соответствующий максимальному значению (в диапазоне от 0 до 1), коррелирует с индексом унитарного представления ожидаемого токена.

Но в данном случае нас интересует не точное воссоздание входного текста, а наиболее вероятное его продолжение. Как и в марковской цепи, следующий токен выбирается случайно, не наиболее часто встречающийся, а на основе вероятности следующего токена.

В результате деления логарифма на температуру происходит сглаживание (температура > 1) или заострение (температура < 1) распределения вероятности. Поэтому температура (аргумент `diversity` в списке аргументов вызова функции в листинге 9.30) < 1 будет означать попытку более точно воспроизвести исходный текст. Температура > 1 приводит к выходному сигналу, отличающемуся от исходного текста. Но усвоенные закономерности стираются по мере сглаживания распределения, и выходной сигнал постепенно превращается в какую-то абракадабру. Впрочем, с более высокими степенями несходства интересно экспериментировать.

Случайная функция `multinomial(num_samples, probabilities_list, size)` из библиотеки NumPy выбирает `num_samples` из распределения, вероятности исходов которого содержатся в `probabilities_list`, и возвращает список длиной `size`, равной

[1] Имеется в виду, что имущественные авторские права на них уже истекли. — *Примеч. пер.*

числу экспериментов[1]. В данном случае размер выборки равен 1, нам нужен только один пример данных.

При предсказании в примере Keras происходит проход в цикле по различным значениям температуры, поскольку для каждого предсказания существует диапазон различных выходных сигналов в зависимости от температуры, используемой в функции `sample` для выборки из распределения вероятностей (листинг 9.30).

Листинг 9.30. Генерация трех текстов при различном уровне несходства с исходным текстом

```
>>> import sys
>>> start_index = random.randint(0, len(text) - maxlen - 1)
>>> for diversity in [0.2, 0.5, 1.0]:
...     print()
...     print('----- diversity:', diversity)
...     generated = ''
...     sentence = text[start_index: start_index + maxlen]
...     generated += sentence
...     print('----- Generating with seed: "' + sentence + '"')
...     sys.stdout.write(generated)
...     for i in range(400):
...         x = np.zeros((1, maxlen, len(chars)))
...         for t, char in enumerate(sentence):
...             x[0, t, char_indices[char]] = 1.
...         preds = model.predict(x, verbose=0)[0]
...         next_index = sample(preds, diversity)
...         next_char = indices_char[next_index]
...         generated += next_char
...         sentence = sentence[1:] + next_char
...         sys.stdout.write(next_char)
...         sys.stdout.flush()
...     print()
```

Задаем начальное значение обученной сети и смотрим, что она вернет в качестве следующего символа

Модель делает прогноз

Находим соответствующий этому индексу символ

Добавляем его в «начальное значение» и отбрасываем первый символ, чтобы длина осталась такой же. Это наше начальное значение для следующего прохода

Эта команда сбрасывает в консоль внутренний буфер, так что наш символ сразу же появляется там

Для краткости мы убрали из примера `diversity = 1.2`, но можете добавить этот вариант обратно и посмотреть на результат.

Мы берем случайную порцию из 40 (`maxlen`) символов из источника и предсказываем следующий символ. Далее добавляем предсказанный символ в конец входной последовательности, отбрасываем первый ее символ и снова производим предсказание с получившимися 40 символами в качестве входных данных. И каждый раз мы записываем предсказанный символ в консоль (или, как в данном случае, в строковый буфер) и выполняем операцию `flush()`, так что этот символ сразу оказывается в консоли. Если предсказанный символ оказывается символом новой строки, то строка

[1] Не вполне точное описание работы данной функции. Возможно, автор перепутал в этом описании смысл аргументов num_samples и size. Число экспериментов задается в первом аргументе, num_samples, а аргумент size описывает форму выходного массива и равен размеру выборки. См. https://numpy.org/doc/1.17/reference/random/generated/numpy.random.Generator.multinomial.html#numpy.random.Generator.multinomial. — *Примеч. пер.*

текста оканчивается, но генератор продолжает работать и предсказывает новую строку на основе только что выведенных предыдущих 40 символов.

Что же мы получим в результате? Что-то вроде:

```
----- diversity: 0.2
----- Generating with seed: " them through & through
the most fond an"
 them through & through
the most fond and stranger the straite to the straite
him a father the world, and the straite:
the straite is the straite to the common'd,
and the truth, and the truth, and the capitoll,
and stay the compurse of the true then the dead and the colours,
and the comparyed the straite the straite
the mildiaus, and the straite of the bones,
and what is the common the bell to the straite
the straite in the commised and

----- diversity: 0.5
----- Generating with seed: " them through & through
the most fond an"
 them through & through
the most fond and the pindage it at them for
that i shall pround-be be the house, not that we be not the selfe,
and thri's the bate and the perpaine, to depart of the father now
but ore night in a laid of the haid, and there is it

bru. what shall greefe vndernight of it

cassi. what shall the straite, and perfire the peace,
and defear'd and soule me to me a ration,
and we will steele the words them with th

----- diversity: 1.0
----- Generating with seed: " them through & through
the most fond an"
 them through & through
the most fond and boy'd report alone

   yp. it best we will st of me at that come sleepe.
but you yet it enemy wrong, 'twas sir

   ham. the pirey too me, it let you?
  son. oh a do a sorrall you. that makino
beendumons vp?x, let vs cassa,
yet his miltrow addome knowlmy in his windher,
a vertues. hoie sleepe, or strong a strong at it
mades manish swill about a time shall trages,
and follow. more. heere shall abo
```

При `diversity 0.2` и `0.5` получаем нечто напоминающее на первый взгляд текст Шекспира. При `diversity 1.0` (при нашем наборе данных) все идет не так. Но обратите внимание, что определенные базовые структуры, например разрыв строки

с последующим сокращенным именем персонажа, все еще присутствуют. В общем, неплохо для такой простой модели, и определенно это основа для интересных экспериментов с генерацией текста в заданном стиле.

Повышаем практическую пригодность нашего генератора

Но что, если порождающая модель нужна не только для развлечений? Как сделать ее более согласованной и пригодной для практического использования?

- Расширьте объем и повысьте качество корпуса.
- Повысьте сложность модели (число нейронов).
- Реализуйте более совершенный алгоритм выравнивания регистра.
- Сегментируйте предложения.
- Добавьте соответствующие вашим потребностям грамматические, орфографические и интонационные фильтры.
- Демонстрируйте пользователям лишь малую часть генерируемых примеров.
- Используйте несколько различных начальных значений текста в каждом раунде диалога, чтобы определить, о чем чат-бот способен говорить достаточно хорошо и что пользователь считает полезным.

Еще больше идей можно найти на рис. 1.4. Теперь, вероятно, он будет вам более понятен, чем при первом знакомстве.

9.1.8. Мы научились, как говорить, но не что говорить

Мы сгенерировали новый текст исключительно на основе примера текста. И научились воспроизводить стиль. Но, как бы это ни было странно, мы никак не контролируем то, что говорим. Контекст ограничен исходными данными, которые по крайней мере ограничивают словарь модели. При наличии входных данных можно обучать модель говорить то, что, по нашему мнению, говорил бы настоящий автор (авторы). Но максимум, на что можно надеяться при подобном типе модели, — воспроизвести то, *как* они говорят — а именно, как бы они закончили высказывание, начинающееся заданным в качестве начального значения предложением. Источником этого предложения не обязательно будет сам текст. Поскольку модель обучалась на символах, можно использовать новые слова как начальные значения и получить интересные результаты. У нас уже есть заготовка чат-бота — собеседника. Но только в следующей главе он научится говорить нечто осмысленное *и* в определенном стиле.

9.1.9. Другие виды памяти

LSTM представляют собой расширение простейших идей рекуррентной нейронной сети, и существует множество других подобных расширений. В них незначительно варьируется количество или функциональность шлюзов в ячейке. Например, шлюзовой рекуррентный блок сочетает в едином шлюзе обновления функциональность

шлюза забывания и ветки выбора кандидатов. Этот шлюз позволяет уменьшить число параметров, требующих подбора, и демонстрирует аналогичные обычному LSTM результаты при значительно меньших вычислительных затратах. В Keras существует слой `GRU`, который можно использовать аналогично LSTM, как показано в листинге 9.31.

Листинг 9.31. Шлюзовые рекуррентные блоки в Keras

```
>>> from keras.models import Sequential
>>> from keras.layers import GRU
>>> model = Sequential()
>>> model.add(GRU(num_neurons, return_sequences=True,
...               input_shape=X[0].shape))
```

Еще один метод — LSTM с *peephole*-соединениями. Непосредственной реализации этого метода в Keras нет, но в Интернете можно найти несколько примеров, в которых с этой целью расширяется класс LSTM Keras. Идея заключается в организации непосредственного доступа каждого шлюза в обычной LSTM-ячейке к текущему состоянию памяти в виде части его входного сигнала. Как описано в статье *Learning Precise Timing with LSTM Recurrent Networks*[1], шлюз содержит дополнительные веса в том же измерении, что и состояние памяти. Входной сигнал каждого шлюза при этом представляет собой конкатенацию входного сигнала ячейки на данном временно́м шаге и выходного сигнала ячейки с предыдущего временно́го шага *и* самого состояния памяти. Авторы сумели точнее смоделировать хронометраж событий в данных временных рядов. Хотя они не занимались именно NLP, сам принцип применим и тут. Предлагаем вам поэкспериментировать с ним.

Это лишь два производных от RNN/LSTM метода. Исследования в данной сфере продолжаются — присоединяйтесь к ним. Все нужные утилиты вполне доступны, так что каждый может найти новый потрясающий метод.

9.1.10. Углубляемся

Блок памяти удобно рассматривать как кодирование конкретного представления пар «существительное/глагол» или соответствий времен глаголов между предложениями, но это не совсем так. Это лишь побочный результат усваиваемых сетью закономерностей при условии успешности обучения. Как и в любой нейронной сети, послойная организация позволяет моделировать более сложные представления содержащихся в тренировочных данных закономерностей. И LSTM-слои можно расположить ярусами столь же легко (рис. 9.13).

Обучение расположенных ярусами слоев требует намного больше вычислительных ресурсов. Но само *размещение их ярусами* в Keras занимает лишь несколько секунд (листинг 9.32).

---

[1] Герс (Gers), Шраудольф (Schraudolph), Шмидхубер (Schmidhuber): http://www.jmlr.org/papers/volume3/gers02a/gers02a.pdf.

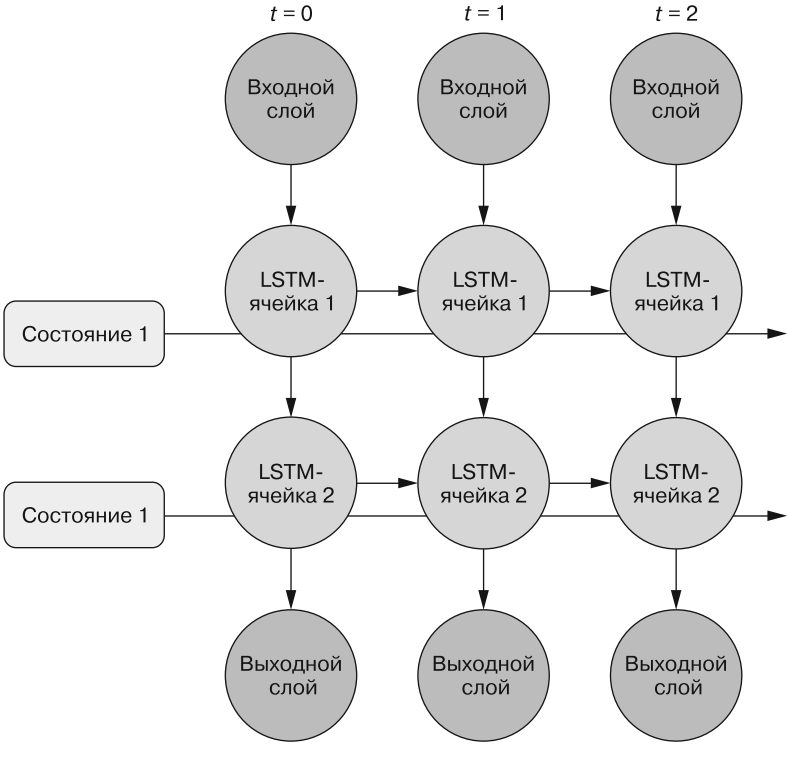


Рис. 9.13. Расположение LSTM ярусами

Листинг 9.32. Два LSTM-слоя

```
>>> from keras.models import Sequential
>>> from keras.layers import LSTM
>>> model = Sequential()
>>> model.add(LSTM(num_neurons, return_sequences=True,
...                input_shape=X[0].shape))
>>> model.add(LSTM(num_neurons_2, return_sequences=True))
```

Обратите внимание, что для правильного построения модели в первом и промежуточных слоях необходимо указать параметр `return_sequences=True`. Это требование объясняется тем, что в качестве входного сигнала временны́х шагов следующего слоя необходим выходной сигнал на каждом из временны́х шагов предыдущего.

Впрочем, не забывайте, что создание модели, способной представлять более сложные взаимосвязи, чем содержатся в тренировочных данных, может привести к странным результатам. Простое нагромождение в модели слоев, как бы интересно это ни было, редко делает модель полезной на практике.

Резюме

- Запоминание информации с помощью блоков памяти позволяет создавать более точные и общие модели последовательностей.
- Важно забывать нерелевантную более информацию.
- Для последующего входного сигнала следует сохранять лишь часть новой информации, и LSTM можно обучить ее находить.
- Умение предсказывать последующие символы/слова позволяет генерировать новый текст на основе вероятностей.
- Модели на основе отдельных символов эффективнее и успешнее обучаются на маленьких, узкоспециализированных корпусах, чем модели на основе слов.
- Векторы идей LSTM захватывают намного больше, чем просто сумму слов в высказывании.

10 Модели sequence-to-sequence и механизм внимания

В этой главе

- Задание соответствия последовательностей с помощью нейронной сети.
- Задачи преобразования последовательностей в последовательности и их отличие от уже знакомых вам.
- Использование архитектур «кодировщик — декодировщик» для перевода и чата.
- Обучаем модель обращать внимание на самое важное в предложении.

Мы уже рассказали, как создавать модели естественного языка и использовать их для разнообразных задач, от классификации тональностей до генерации нового текста (см. главу 9).

Может ли нейронная сеть переводить с английского языка на немецкий? Способна ли она предсказывать болезнь путем перевода генотипа в фенотип (генов в индивидуальные особенности организма)[1]? И как насчет чат-бота, о котором мы говорили с самого начала этой книги? Может ли нейронная сеть поддерживать разговор на отвлеченные темы? Все это задачи на преобразование последовательностей в последовательности (sequence-to-sequence). Одной последовательности

[1] geno2pheno: https://academic.oup.com/nar/article/31/13/3850/2904197.

неопределенной длины ставится в соответствие другая последовательность, также неизвестной длины.

В этой главе вы узнаете, как создавать модели sequence-to-sequence с помощью архитектур типа «кодировщик — декодировщик» (encoder — decoder).

10.1. Архитектура типа «кодировщик — декодировщик»

Как вы думаете, какая из наших предыдущих архитектур подошла бы для задач преобразования последовательностей в последовательности? Модель вложений векторов слов из главы 6? Сверточная нейронная сеть из главы 7 или рекуррентные нейронные сети из глав 8 и 9? Правильно, будем работать с LSTM-архитектурой из предыдущей главы.

LSTM замечательно подходят для обработки последовательностей, но оказывается, что для этих задач понадобится две LSTM, а не одна. Мы будем строить модульную архитектуру типа «кодировщик — декодировщик» (encoder — decoder architecture).

Первая половина модели «кодировщик — декодировщик» — это *кодировщик* (encoder) последовательностей, сеть, преобразующая последовательность (например, текст на естественном языке), в представление низкой размерности (например, вектор идеи, о которой речь шла в конце главы 9). Так что первую половину нашей модели sequence-to-sequence мы уже построили.

Вторая половина архитектуры типа «кодировщик — декодировщик» состоит из *декодировщика* (decoder) последовательностей. Декодировщик последовательностей можно спроектировать таким образом, что он будет преобразовывать вектор обратно в удобочитаемый текст. Но разве мы не делали и это тоже? Мы сгенерировали немного странный шекспировский текст пьесы в конце главы 9. Почти то, что нужно, но нам придется добавить еще несколько компонентов, чтобы шекспировский бот-драматург сосредоточил свое внимание на новой задаче и стал переводчиком.

Например, пусть наша модель выдаст немецкий перевод английского входного текста. На самом деле не аналогично ли это переводу шекспировским ботом современного английского на язык Шекспира? Да, но в примере с Шекспиром нас устраивало, когда алгоритм машинного обучения выбирал случайным образом слова, соответствующие вероятностям, полученным в результате обучения. Это не подходит для сервиса перевода или даже приличного бота-драматурга.

Итак, вы уже знаете, как создавать кодировщики и декодировщики; осталось научиться улучшать их, делать более «внимательными». На самом деле LSTM из главы 9 отлично подходят в качестве кодировщиков текста переменной длины. Мы создавали их для захвата смысла и тональности текста на естественном языке. LSTM захватывает смысл во внутреннее представление, вектор идеи. Необходимо только извлечь вектор идеи из состояния (ячейки памяти) модели LSTM. Вы уже

умеете задавать параметр `return_state=True` для LSTM-модели Keras, чтобы выходной сигнал включал состояние скрытого слоя. Этот вектор состояния и будет выходным сигналом кодировщика и входным сигналом декодировщика.

СОВЕТ

> При обучении любой модели нейронной сети каждый из внутренних слоев содержит всю необходимую информацию для решения задачи, на которой ее обучали. Эта информация обычно представлена в виде тензора фиксированной размерности, содержащего весовые коэффициенты или результаты активации данного слоя. Если же сеть хорошо обобщается, то наверняка существует узкое место в информации — слой с минимальной размерностью. В Word2vec (см. главу 6) веса внутреннего слоя нужны были для вычисления векторного представления. Можно также использовать непосредственно результаты активации внутреннего слоя сети. Именно так мы и поступим в примерах этой главы. Взгляните на успешно построенные вами сети и попробуйте найти это узкое место в смысле информации для использования в качестве кодированного представления данных.

Осталось только немного усовершенствовать архитектуру декодировщика. Нам нужно декодировать вектор идеи обратно в последовательность естественного языка.

10.1.1. Декодирование вектора идеи

Представьте себе, что нужно разработать модель перевода текстов с английского на немецкий язык. Необходимо сопоставить последовательностям символов или слов другие последовательности символов или слов. Мы уже выяснили, как предсказать элемент последовательности на временно́м шаге t на основе предыдущего элемента на временно́м шаге $t - 1$. Но установление соответствий одного языка другому с помощью непосредственно LSTM приводит к проблемам. Для работы отдельной LSTM нужно, чтобы входная и выходная последовательности были одинаковой длины, а при переводе это бывает редко.

Рисунок 10.1 иллюстрирует эту проблему. Длина предложений на английском и немецком языках различна, что усложняет установление соответствия между английским входным сигналом и ожидаемым выходным сигналом. Английская фраза *is playing* (настоящее продолженное время) переводится на немецкий язык словом в настоящем времени *spielt*. Но слово *spielt* пришлось бы предсказывать на основе входного сигнала *is*; мы еще не добрались до *playing* на этом временно́м шаге. Далее *playing* пришлось бы поставить в соответствие *Fußball*. Конечно, сеть может усвоить такие отображения, но усвоенные представления были бы тесно связаны с конкретным входным сигналом и наша мечта о более универсальной модели языка не осуществилась бы.

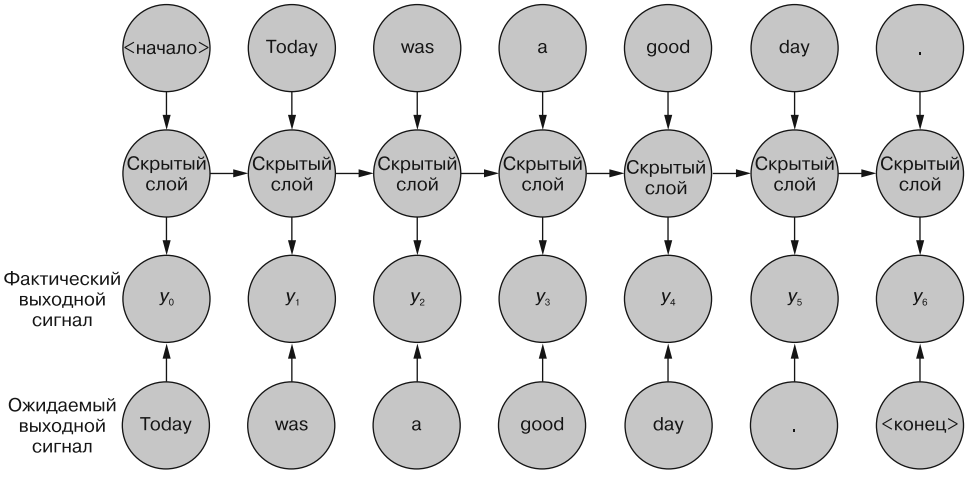


Рис. 10.1. Ограничения языкового моделирования

Сети преобразования последовательностей в последовательности (сокращенно — *seq2seq*) снимают это ограничение за счет создания представления входного сигнала в виде вектора идеи. Модели sequence-to-sequence далее используют этот вектор идеи, или вектор контекста, в качестве отправной точки для второй сети, получающей другой набор входных данных для генерации выходной последовательности.

ВЕКТОР ИДЕИ

> Помните, как вы находили векторы слов? Векторы слов — это сжатие смысла слова в вектор фиксированной длины. Слова со схожим смыслом близки друг к другу в этом векторном пространстве смыслов слов. Вектор идеи — очень похожее понятие. Нейронная сеть может сжимать информацию произвольного высказывания естественного языка, а не только отдельного слова в вектор фиксированной длины, отражающий содержимое входного текста. Этот вектор и есть вектор идеи. Он играет роль численного представления заложенной в документ идеи для работы какой-либо модели декодировщика, обычно декодировщика-переводчика. Этот термин ввел в обращение Джеффри Хинтон (Geoffrey Hinton) в докладе в Королевском научном обществе в 2015 году[1].

Сеть преобразования последовательностей в последовательности состоит из двух модульных рекуррентных сетей с вектором идеи между ними (рис. 10.2).

[1] См. веб-страницу Deep Learning по адресу https://www.evl.uic.edu/creativecoding/courses/cs523/slides/week3/DeepLearning_LeCun.pdf.

Кодировщик выдает на выходе вектор идеи в конце его входной последовательности. Декодировщик берет этот вектор идеи и выдает на выходе последовательность токенов.

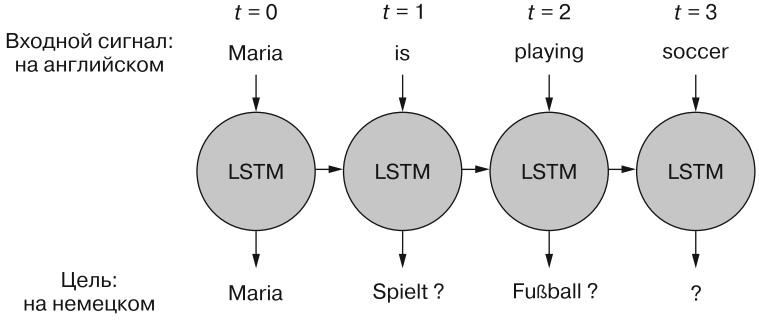


Рис. 10.2. «Бутерброд» кодировщик — декодировщик с вектором идеи в качестве мяса

Первая сеть — кодировщик — преобразует входной текст (например, отправленное пользователем чат-боту сообщение) в вектор идеи. Этот вектор идеи состоит из двух частей-векторов: выходного сигнала (результата активации) скрытого слоя кодировщика и состояния памяти ячейки LSTM для входного примера данных.

ПРИМЕЧАНИЕ

Как показано в листинге 10.1, вектор идеи записывается в переменные state_h (выходной сигнал скрытого слоя) и state_c (состояние памяти).

Далее вектор идеи становится входным сигналом для второй сети — сети-декодировщика. Как вы увидите далее в разделе, посвященном описанию реализации, сгенерированное состояние (вектор идеи) является *начальным состоянием* сети-декодировщика. Затем вторая сеть использует это начальное состояние и особую разновидность входного сигнала, *токен начала* (start token). Вооружившись данной информацией, вторая сеть должна обучиться генерировать первый элемент целевой последовательности (например, символ или слово).

Этапы обучения и вывода происходят по-разному в каждой отдельной архитектуре. Во время обучения исходный текст передается кодировщику, а *ожидаемый* текст — в качестве входного сигнала декодировщика. Сеть-декодировщик должна обучиться по заданному состоянию и начальному ключу выдавать ряды токенов. Первым непосредственным входным сигналом декодировщика будет токен начала; вторым входным сигналом должен быть первый ожидаемый или предсказанный токен, который, в свою очередь, должен побудить сеть вернуть второй ожидаемый токен.

Во время вывода ожидаемого текста у нас нет. Что же можно передать в декодировщик, за исключением состояния? Сначала мы передадим универсальный токен

начала, а затем возьмем первый сгенерированный элемент в качестве входного сигнала декодировщика на следующем временно́м шаге, для генерации следующего элемента и т. д. Повторяем этот процесс до тех пор, пока не будет достигнуто заданное максимальное число элементов последовательности или сгенерирован токен останова.

Обученный подобным образом «от и до» декодировщик превратит вектор идеи в полностью декодированный ответ на начальную входную последовательность (например, вопрос пользователя). Разбиение решения на две сети с вектором идеи в качестве связующего звена между ними позволяет отображать входные последовательности в выходные другой длины (рис. 10.3).

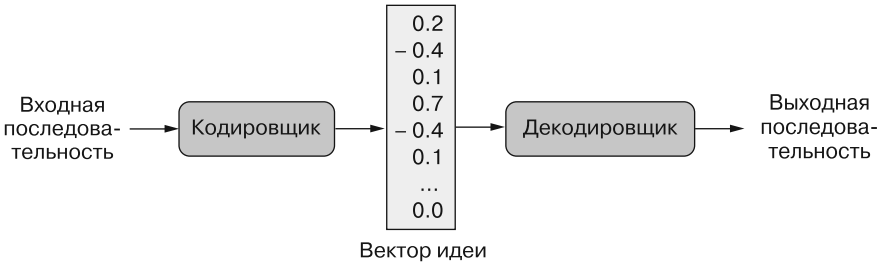


Рис. 10.3. Развернутый кодировщик — декодировщик

10.1.2. Знакомо, правда?

У вас может сложиться впечатление, что вы уже сталкивались с подходом кодирования — декодирования. Вполне может быть. Автокодировщики — часто используемая для обучения студентов нейронным сетям архитектура типа «кодировщик — декодировщик». Они похожи на игру в повтор фраз («игра в попугая»), что упрощает поиск тренировочных данных для нейронных сетей, обучающихся механически выдавать входной сигнал. Подойдет практически любой большой набор данных из многомерных тензоров, векторов или последовательностей.

Как и у любой архитектуры типа «кодировщик — декодировщик», у автокодировщиков есть узкое место в информации между кодировщиком и декодировщиком, которое можно использовать в качестве низкоразмерного представления входных данных. Любую сеть с узким местом в информации можно использовать в качестве кодировщика в архитектуре типа «кодировщик — декодировщик», даже если эта сеть обучалась только перефразировать или повторять входной сигнал[1].

Хотя структура автокодировщиков совпадает с нашими кодировщиками — декодировщиками из этой главы, они обучаются для решения другой задачи. Автокодировщики обучаются находить такое векторное представление входных данных, что-

[1] *Chandar, Lauly et al.* An Autoencoder Approach to Learning Bilingual Word Representations (https://papers.nips.cc/paper/5270-an-autoencoder-approach-to-learning-bilingual-word-representations.pdf).

бы декодировщик сети мог восстановить входной сигнал с минимальной ошибкой. Кодировщик и декодировщик — псевдопротивоположности. Задача сети — найти такое плотное векторное представление входных данных (например, изображения или текста), которое позволило бы декодировщику восстановить входной сигнал с минимальной ошибкой. На этапе обучения входные данные и ожидаемый выходной сигнал одинаковы. Следовательно, если задача состоит в нахождении плотного векторного представления данных, а не генерации векторов идей для перевода с одного языка на другой или ответов на заданный вопрос, автокодировщик может быть неплохим вариантом.

Как насчет PCA и t-SNE из главы 6? Не использовали ли мы `sklearn.decomposition.PCA` или `sklearn.manifold.TSNE` для визуализации векторов в других главах? Модель t-SNE выдает в качестве выходного сигнала вложение, поэтому в некотором смысле ее можно считать кодировщиком. То же самое справедливо и для PCA. Впрочем, эти модели — без учителя, так что их нельзя ориентировать на конкретный выходной сигнал или задачу. И данные алгоритмы были созданы преимущественно для выделения признаков и визуализации. Они создают очень узкие места для вывода векторов очень низкой размерности, обычно 2 или 3. И они не предназначены для того, чтобы получать на входе последовательности произвольной длины. Это главное в кодировщике. И мы уже рассказывали, что LSTM — последнее достижение науки для задач выделения признаков и вложений из последовательностей.

ПРИМЕЧАНИЕ

Вариационный автокодировщик — модифицированная версия автокодировщика, обучаемая в качестве хорошего генератора, а не только кодировщика — декодировщика. Вариационный автокодировщик выдает сжатый вектор, не только точно представляющий входной сигнал, но и распределенный по гауссовскому закону. Это упрощает генерацию нового выходного сигнала путем случайного выбора вектора начального значения и передачи его в соответствующую кодировщику половину автокодировщика[1].

10.1.3. Диалог с помощью sequence-to-sequence

Возможно, вам непонятно, какое отношение имеет задача диалогового движка (чата) к машинному переводу, но они довольно схожи. Генерация ответов при разговоре для чат-бота не слишком отличается от перевода на немецкий язык высказывания на английском в системе машинного перевода.

Задачи как перевода, так и разговорной речи требуют от модели отображения одной последовательности в другую. Отображение последовательности англоязычных токенов в последовательность токенов немецкого языка очень напоминает отображение высказываний естественного языка при разговоре в ожидаемые ответы диалогового движка. Движок машинного перевода можно рассматривать

[1] См. веб-страницу Variational Autoencoders Explained (http://kvfrans.com/variational-autoencoders-explained).

как страдающий раздвоением личности двуязычный диалоговый движок, играющий в детскую игру «эхо»[1], который слушает на английском, а отвечает на немецком.

Но мы хотим, чтобы наш бот отвечал, а не просто повторял услышанное. Поэтому модель должна привлекать любую дополнительную информацию об окружающем мире, относящуюся к тому, о чем должен беседовать бот. NLP-модель должна усвоить более сложное соответствие высказывания ответу, чем простое эхо или перевод. Это требует большего объема тренировочных данных и вектора идеи более высокой размерности, поскольку он должен содержать всю необходимую диалоговому движку информацию об окружающем мире. В главе 9 вы узнали, как повысить размерность, а следовательно, и информационную емкость вектора идеи в модели LSTM. Нам также понадобится достаточное количество данных подходящего типа, чтобы превратить машину-переводчик в машину, способную общаться.

При наличии набора токенов можно обучить конвейер машинного обучения имитировать последовательность ответов при разговоре. Для понимания всех этих соответствий требуется достаточное количество таких пар и вектор идеи необходимой информационной емкости. Если есть набор данных, содержащий достаточное количество подобных пар переводов высказываний в ответы, можно обучить диалоговый движок с помощью той же сети, которая использовалась для машинного перевода.

В Keras есть модули, позволяющие создавать сети с модульной архитектурой (модель «кодировщик — декодировщик») для преобразования последовательностей в последовательности. Кроме того, Keras предоставляет API для доступа ко всем «внутренностям» LSTM-сети, необходимого для решения задач перевода, разговора и даже преобразования «генотип — фенотип».

10.1.4. Обзор LSTM

Из предыдущей главы вы узнали, что LSTM позволяет рекуррентным нейронным сетям избирательно запоминать и забывать паттерны токенов, встреченные ими в примере документа. Входной токен для каждого из временны́х шагов проходит через шлюзы забывания и обновления, умножается на весовые коэффициенты и маски, а затем сохраняется в ячейке памяти. Выходной сигнал сети на этом временно́м шаге (токен) определяется не только входным токеном, но и сочетанием входного сигнала *и* текущего состояния блока памяти.

Важную роль играет тот факт, что закономерности токенов в LSTM распознаются в масштабах всех документов, поскольку веса шлюзов забывания и обновления обучаются при просмотре многих документов. LSTM не придется заново учить английскую орфографию и грамматику для каждого нового документа. Мы также научились активировать сохраненные в весах ячейки памяти LSTM закономерности токенов для предсказания последующих токенов на основе запуска генерации последовательности на базе каких-либо начальных токенов (рис. 10.4).

[1] Та же «игра в попугая» (https://en.uncyclopedia.co/wiki/Childish_Repeating_Game).

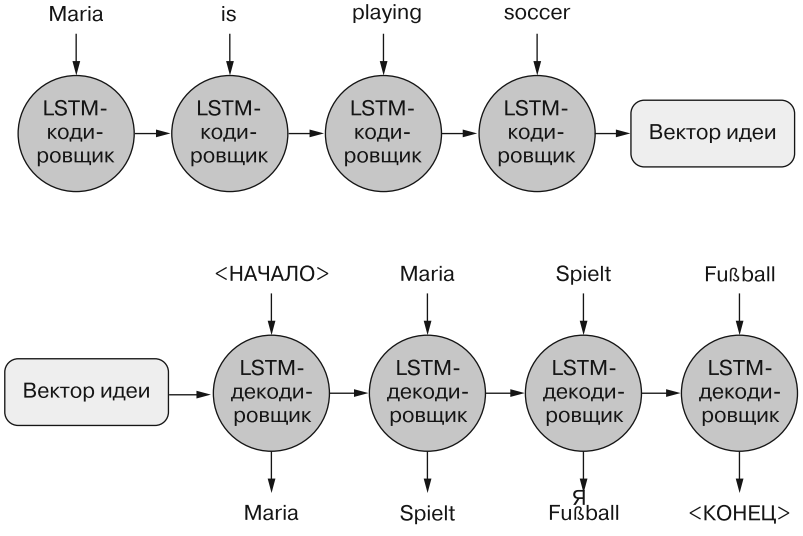


Рис. 10.4. Предсказание следующего слова

Предсказывая токен за токеном, мы сумели сгенерировать текст, выбирая следующий токен в соответствии с распределением вероятностей предполагаемых сетью следующих токенов. Как ни крути, не наилучший вариант, но все же интересный. Но мы здесь не ради интереса; мы хотим управлять выходным сигналом порождающей модели.

Суцкевер (Sutskever), Виньялс (Vinyals) и Ле (Le) придумали способ подключения второй LSTM-модели для менее произвольного и более управляемого *декодирования* закономерностей в ячейке памяти[1]. Они предложили воспользоваться классификационной стороной LSTM для создания вектора идеи и затем подать сгенерированный вектор на вход второй, *другой* LSTM, пытающейся лишь произвести предсказания токен за токеном, благодаря чему можно отобразить входную последовательность в отдельную выходную последовательность. Посмотрим, как это происходит на практике.

10.2. Компонуем конвейер sequence-to-sequence

Благодаря полученной информации из предыдущих глав у вас уже есть все фрагменты, необходимые для создания конвейера машинного обучения для преобразования последовательностей в последовательности.

[1] *Sutskever, Vinyals, Le*. arXiv:1409.3215 (http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf).

10.2.1. Подготавливаем набор данных для обучения модели sequence-to-sequence

Как вы уже видели в предыдущих реализациях сверточных и рекуррентных нейронных сетей, данные необходимо дополнить до фиксированной длины. Обычно входные последовательности дополняются специальными токенами до размера самой длинной входной последовательности. Для сети преобразования последовательностей в последовательности необходимо также подготовить целевые данные и дополнить их до размера самой длинной целевой последовательности. Помните, что длины последовательностей входных и целевых данных могут различаться (рис. 10.5).

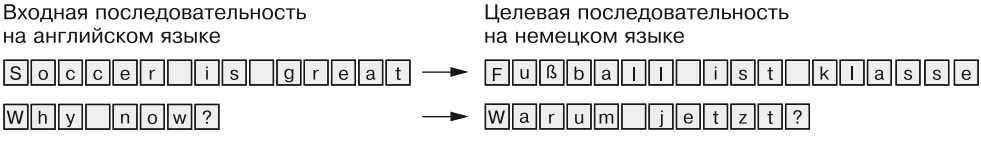


Рис. 10.5. Входная и целевая последовательности до предварительной обработки

Помимо необходимого дополнения, выходную последовательность нужно снабдить токенами начала и останова, чтобы декодировщик знал, где его задание начинается и когда заканчивается (рис. 10.6).

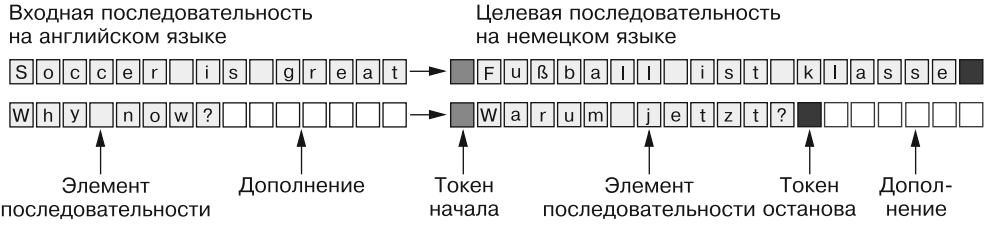


Рис. 10.6. Входная и целевая последовательности после предварительной обработки

Мы расскажем вам, как снабдить целевые последовательности нужными токенами далее в этой главе, когда будем строить конвейер Keras. Запомните, что нам нужны две версии целевой последовательности для обучения: одна, начинающаяся с токена начала (которая будет использоваться в качестве входного сигнала декодировщика), и вторая — без этого токена в начале (на основе этой целевой последовательности функция потерь будет оценивать точность).

В предыдущих главах тренировочные наборы данных состояли из пар: входного сигнала и ожидаемого выходного сигнала. Все тренировочные примеры для модели sequence-to-sequence представляют собой тройки: начальный входной сигнал, ожидаемый выходной сигнал (предваренный токеном начала) и ожидаемый выходной сигнал (без токена начала).

Прежде чем углубиться в реализацию, подведем итоги. Наша сеть преобразования последовательностей в последовательности состоит из двух сетей: кодировщика, генерирующего вектор идеи; и декодировщика, в который мы этот вектор идеи будем передавать в качестве начального состояния. С заданным начальным состоянием и токеном начала в качестве входного сигнала сети-декодировщика мы сгенерируем первый элемент последовательности (например, символ или вектор слов) выходного сигнала. Каждый из следующих элементов предсказывается на основе обновленного состояния и следующего элемента ожидаемой последовательности. Этот процесс продолжается либо до момента генерации токена останова, либо до достижения максимального числа элементов. Все сгенерированные декодировщиком элементы последовательности формируют предсказанный выходной сигнал (например, ответ на вопрос пользователя). Теперь можно приступать к изучению подробностей.

10.2.2. Модель sequence-to-sequence в Keras

В следующих разделах мы рассмотрим реализацию на Keras сети sequence-to-sequence, описанную в работе Франсуа Шолле (Francois Chollet)[1]. Мистер Шолле написал также книгу *Deep Learning with Python* (Manning, 2017)[2] — бесценный источник сведений об архитектурах нейронных сетей и Keras.

На этапе обучения мы будем обучать сеть-кодировщик и сеть-декодировщик вместе, от начала до конца, для чего потребуется три точки данных для каждого примера данных: входная тренировочная последовательность кодировщика, входная последовательность декодировщика и выходная последовательность декодировщика. Роль входной тренировочной последовательности кодировщика может играть заданный пользователем вопрос, на который должен ответить наш бот. Ожидаемый ответ будущего бота при этом будет входной последовательностью декодировщика.

Наверное, вы недоумеваете, зачем нужны входная *и* выходная последовательности для декодировщика. Дело в том, что мы обучаем декодировщик с помощью метода *усиления учителем* (teacher forcing), при котором предоставленное сетью-кодировщиком начальное состояние используется для обучения декодировщика генерировать ожидаемые последовательности: декодировщику демонстрируется входной сигнал и позволяется предсказывать ту же последовательность. Значит, входная и выходная последовательности декодировщика будут идентичны, за исключением сдвига их на один временной шаг друг относительно друга.

На этапе выполнения кодировщик используется для генерации вектора идеи пользовательского входного сигнала, а декодировщик затем генерирует на основе

---

[1] См. веб-страницу A ten-minute introduction to sequence-to-sequence learning in Keras (https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html).

[2] *Шолле Ф.* Глубокое обучение на Python. — СПб.: Питер, 2019.

этого вектора идеи ответ. Ответом пользователю служит выходной сигнал декодировщика.

ФУНКЦИОНАЛЬНЫЙ API KERAS

В следующем примере стиль реализации слоев Keras отличается от показанного в предыдущих главах. В Keras появился еще один способ компоновки моделей — с помощью вызова каждого из слоев, с передачей ему значения с предыдущего слоя. Функциональный API предоставляет широкие возможности в случаях создания моделей с переиспользованием частей уже обученных моделей (как мы покажем в следующих разделах). Больше информации о функциональном API Keras вы можете найти в сообщении блога разработчиков ядра Keras[1].

10.2.3. Кодировщик последовательностей

Единственная задача кодировщика — создание вектора идеи, который затем выступит в роли начального состояния сети-декодировщика (рис. 10.7). Обучить кодировщик отдельно нельзя, поскольку отсутствует «целевой» вектор идеи, который сеть могла бы обучиться предсказывать. Обратное распространение ошибки, с помощью которого кодировщик обучается создавать соответствующий вектор идеи, основано на ошибке, генерируемой далее по конвейеру в декодировщике.

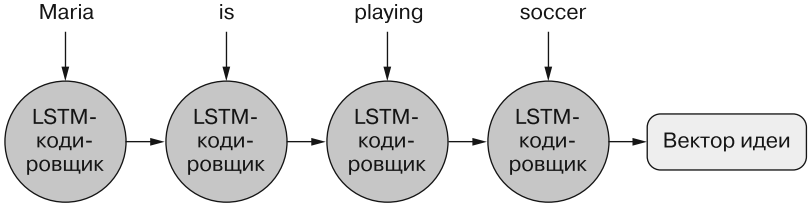


Рис. 10.7. Кодировщик вектора идеи

Тем не менее кодировщик и декодировщик — независимые модули, причем зачастую взаимозаменяемые. Например, после обучения кодировщика на задаче перевода с английского на немецкий язык его можно использовать повторно с другим кодировщиком для перевода с английского на испанский[2]. Листинг 10.1 демонстрирует, как выглядит отдельно кодировщик.

---

[1] См. веб-страницу Getting started with the Keras functional API: https://keras.io/getting-started/functional-api-guide/.

[2] Обучение подобной многозадачной модели называется совместным обучением (joint training) или переносом обучения (transfer learning) и описано Луонгом (Luong), Ле (Le), Суцкевером (Sutskever), Виньялсом (Vinyals) и Кайером (Kaier) из проекта Google Brain на конференции ICLR 2016: https://arxiv.org/pdf/1511.06114.pdf.

Очень удобно, что RNN-слои Keras возвращают внутреннее состояние при создании экземпляра LSTM-слоя (или слоев) с помощью поименованного аргумента `return_state=True`. В следующем фрагменте кода мы сохраняем окончательное состояние кодировщика и игнорируем его фактический выходной сигнал. Список состояний LSTM затем передается декодировщику.

Листинг 10.1. Кодировщик идеи в Keras

Для возврата внутренних состояний аргумент return_state LSTM-слоя должен быть задан равным True

```
>>> encoder_inputs = Input(shape=(None, input_vocab_size))
>>> encoder = LSTM(num_neurons, return_state=True)
>>> encoder_outputs, state_h, state_c = encoder(encoder_inputs)
>>> encoder_states = (state_h, state_c)
```

Первым возвращаемым значением LSTM-слоя является выходной сигнал этого слоя

Поскольку параметр `return_sequences` по умолчанию равен `False`, первым возвращаемым значением является выходной сигнал с последнего временно́го шага. `state_h` — именно выходной сигнал последнего временно́го шага для данного слоя. В нашем примере `encoder_outputs` и `state_h` будут идентичны. В любом случае можно игнорировать формальный выходной сигнал, хранящийся в `encoder_outputs`. `state_c` — текущее состояние блока памяти. Наш вектор идеи будет состоять из `state_h` и `state_c`.

Рисунок 10.8 демонстрирует генерацию внутренних состояний LSTM. Кодировщик обновляет скрытое состояние и состояние памяти на каждом временно́м шаге и передает конечные состояния в качестве начального состояния в декодировщик.

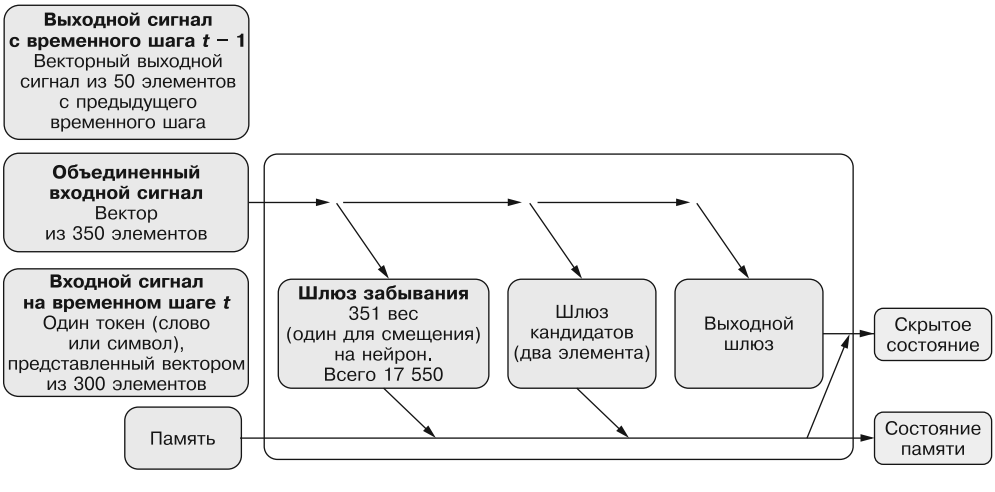


Рис. 10.8. Используемые в кодировщике преобразования последовательностей в последовательности LSTM-состояния

10.2.4. Декодировщик идеи

Аналогично схеме сети-кодировщика схема декодировщика довольно проста. Основное отличие состоит в том, что на этот раз мы хотим захватывать выходной сигнал сети на каждом временно́м шаге. Нам нужно определить степень правильности выходного сигнала, токен по токену (рис. 10.9).

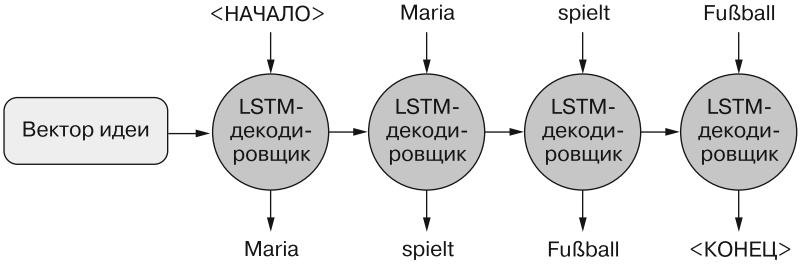


Рис. 10.9. Декодировщик вектора идеи

Именно здесь используются вторая и третья части тройки примера данных. У декодировщика обычный входной сигнал токен по токену и выходной сигнал токен по токену. Они практически идентичны, но сдвинуты на один временной шаг друг относительно друга. Необходимо, чтобы декодировщик обучился воспроизводить токены заданной входной последовательности *при заданном* состоянии, сгенерированном на основе переданной кодировщику первой части тройки.

ПРИМЕЧАНИЕ

Такой подход является ключевым для декодировщика, да и для моделей sequence-to-sequence вообще; сеть обучается выдавать выходной сигнал в пространстве второй задачи (другого языка или ответа другого человека/бота на заданный вопрос). При этом формируется идея и того, что было сказано (входной сигнал), и ответа (выходной сигнал) одновременно, а эта идея определяет ответ, токен по токену. В конце концов, для работы нам нужны только идея (сгенерированная кодировщиком) и универсальный токен начала. Этого достаточно для инициации выдачи правильной выходной последовательности.

Для вычисления ошибки на шаге обучения мы передаем выходной сигнал LSTM-слоя в слой `Dense`. Число нейронов в плотном слое равно количеству всех возможных выходных токенов. В нем ко всем токенам применяется многомерная логистическая функция активации. Так что на каждом временно́м шаге сеть выдает распределение вероятности по всем возможным токенам для наиболее вероятного (с ее точки зрения) очередного элемента последовательности. И берется токен с самым большим значением у соответствующего нейрона. Мы уже использовали

выходной слой с многомерными логистическими функциями активации в предыдущих главах, когда хотели определить токен с максимальным правдоподобием (см. подробности в главе 6). Отметим, что `num_encoder_tokens` и `output_vocab_size` не обязаны совпадать — одно из главных преимуществ сетей sequence-to-sequence (листинг 10.2).

Листинг 10.2. Декодировщик идеи в Keras

```
>>> decoder_inputs = Input(shape=(None, output_vocab_size))
>>> decoder_lstm = LSTM(
...     num_neurons,return_sequences=True, return_state=True)
>>> decoder_outputs, _, _ = decoder_lstm(
...     decoder_inputs, initial_state=encoder_states)
>>> decoder_dense = Dense(
...     output_vocab_size, activation='softmax')
>>> decoder_outputs = decoder_dense(decoder_outputs)
```

Функциональный API дает возможность передавать начальное состояние в LSTM-слой путем присваивания последнего состояния кодировщика аргументу initial_state

Создаем LSTM-слой, аналогичный кодировщику, но с дополнительным аргументом return_sequences

Слой функции softmax с отображением всех возможных символов на выходной сигнал softmax

Передаем выходной сигнал LSTM-слоя в слой многомерной логистической функции

10.2.5. Формируем сеть sequence-to-sequence

Функциональный API Keras позволяет формировать модель в виде обращений к объектам. Объект `Model` позволяет описать входные и выходные сигналы сети. Для данной сети sequence-to-sequence мы будем передавать в модель список входных сигналов. В листинге 10.2 описан один входной слой в кодировщике и один — в декодировщике. Эти два входных сигнала соответствуют двум первым элементам обучающих троек. В качестве выходного слоя мы передадим в модель `decoder_outputs`, включающий все описанные выше настройки модели. Выходной сигнал в `decoder_outputs` соответствует последнему элементу обучающих троек.

ПРИМЕЧАНИЕ

При подобном использовании функционального API описания вроде decoder_outputs являются тензорными представлениями. Этим они отличаются от последовательной модели из предыдущих глав. Подробности API Keras вы опять же можете найти в документации (листинг 10.3).

Листинг 10.3. Функциональный API Keras (Model())

```
>>> model = Model(
...     inputs=[encoder_inputs, decoder_inputs],
...     outputs=decoder_outputs)
```

Аргументы inputs и outputs можно описать в виде списков, если ожидается несколько элементов входных/выходных данных

10.3. Обучение сети sequence-to-sequence

Для создания модели sequence-to-sequence в Keras нам осталось только ее скомпилировать и обучить. Единственное отличие по сравнению с предыдущими главами состоит в том, что раньше мы предсказывали бинарную классификацию: да или нет. Но здесь мы имеем дело с задачей категориальной (многоклассовой) классификации. На каждом временно́м шаге необходимо определить, какая из нескольких категорий — правильная, а здесь у нас много категорий. Модель должна выбрать какой из множества возможных токенов «озвучить». Поскольку мы предсказываем символы или слова, а не бинарные состояния, то оптимизацию потерь будем производить относительно функции потерь `categorical_crossentropy`, а не использовавшейся раньше `binary_crossentropy`. Это единственное изменение, которое необходимо внести в шаг `model.compile` Keras, как показано в листинге 10.4.

Листинг 10.4. Обучение модели sequence-to-sequence в Keras

```
>>> model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
>>> model.fit([encoder_input_data, decoder_input_data],
               decoder_target_data,
               batch_size=batch_size, epochs=epochs)
```

Задаем функцию потерь categorical_crossentropy

Модель ожидает обучающие входные данные в виде списка, первый элемент в котором передается во время обучения в сеть-кодировщик, второй — в сеть-декодировщик

Поздравляем! С помощью вызова `model.fit` мы произвели полное обучение нашей сети sequence-to-sequence, от начала и до конца. В следующих разделах мы покажем вывод выходной последовательности для заданной входной.

ПРИМЕЧАНИЕ

Обучение сетей sequence-to-sequence может потребовать значительных вычислительных ресурсов и много времени. Если тренировочные последовательности длинные или нужно обучаться на большом корпусе, очень рекомендуем обучать такие сети на GPU, что повышает скорость обучения порой и в 30 раз. Если вы никогда не обучали нейронную сеть на GPU — не волнуйтесь. Прочитайте главу 13, где описано, как арендовать и настроить собственный GPU на платных вычислительных облачных сервисах.

LSTM, в отличие от сверточных нейронных сетей, не распараллеливаются естественным образом, поэтому для максимальной отдачи от GPU имеет смысл заменить LSTM-слои на `CuDNNLSTM`, оптимизированный для обучения на GPU, с поддержкой CUDA.

10.3.1. Генерация выходных последовательностей

Перед генерацией последовательностей необходимо перемонтировать структуру обучающих слоев для генерации. Сначала мы опишем предназначенную специально для кодировщика модель, которой затем воспользуемся для генерации вектора идеи (листинг 10.5).

Листинг 10.5. Декодировщик для генерации текста с помощью универсального класса Model Keras

```
>>> encoder_model = Model(inputs=encoder_inputs, outputs=encoder_states)
```

Здесь используются ранее описанные encoder_inputs и encoder_states; вызов метода predict этой модели возвращает вектор идеи

Описание декодировщика может смутить вас. Но будем шаг за шагом разбираться в соответствующих фрагментах кода. Во-первых, мы описываем входные сигналы декодировщика, используя входной слой Keras, но вместо передачи в него унитарных векторов, символов или вложений слов мы передаем вектор идеи, сгенерированный сетью-кодировщиком. Обратите внимание, что кодировщик возвращает список из двух состояний, который затем необходимо передать в аргументе `initial_state` при вызове ранее описанного `decoder_lstm`. Выходной сигнал LSTM-слоя далее передается в плотный слой, также описанный ранее. Выходной сигнал этого слоя дает нам вероятности всех выходных токенов декодировщика (в данном случае всех символов, встреченных во время фазы обучения).

А теперь — самое интересное. Предсказанный с наибольшей вероятностью на каждом временно́м шаге токен будет возвращен в качестве наиболее вероятного токена и передан на следующий шаг итерации декодировщика как новый входной сигнал (листинг 10.6).

Листинг 10.6. Генератор последовательностей для случайных идей

```
>>> thought_input = [Input(shape=(num_neurons,)),
...     Input(shape=(num_neurons,))]
>>> decoder_outputs, state_h, state_c = decoder_lstm(
...     decoder_inputs, initial_state=thought_input)
>>> decoder_states = [state_h, state_c]
>>> decoder_outputs = decoder_dense(decoder_outputs)

>>> decoder_model = Model(
...     inputs=[decoder_inputs] + thought_input,
...     output=[decoder_outputs] + decoder_states)
```

Описываем входной слой, получающий состояния кодировщика

Передаем состояние кодировщика в LSTM-слой в качестве начального состояния

Обновленное LSTM-состояние становится новым состоянием ячейки для следующей итерации

Последний шаг — собираем воедино модель декодировщика

decoder_inputs и thought_input становятся входным сигналом для модели-декодировщика

Передаем выходной сигнал LSTM в плотный слой для предсказания следующего токена

Выходной сигнал плотного слоя и обновленные состояния описываются как выходной сигнал

После завершения описания модели можно генерировать последовательности с помощью предсказания вектора идеи на основе унитарной входной последовательности и последнего сгенерированного токена. Во время первой итерации `target_seq` устанавливается равным токену начала, а во время всех последующих итераций `target_seq` обновляется последним сгенерированным токеном. Этот цикл выполняется до тех пор, пока либо не будет достигнуто максимальное число элементов последовательности, либо декодировщик сгенерирует токен останова, в случае чего генерация прекращается (листинг 10.7).

Листинг 10.7. Простой декодировщик — предсказание следующего слова

```
...
>>> thought = encoder_model.predict(input_seq)
...
>>> while not stop_condition:
...     output_tokens, h, c = decoder_model.predict(
...         [target_seq] + thought)
```

Кодируем входную последовательность в вектор идеи (состояние LSTM-ячейки памяти)

Декодировщик возвращает токен с максимальной вероятностью и внутренние состояния, которые используются снова на следующей итерации

stop_condition обновляется после каждой итерации и становится равным True либо при достижении максимального числа токенов выходной последовательности, либо при генерации декодировщиком токена останова

10.4. Создание чат-бота с помощью сетей sequence-to-sequence

В предыдущих разделах вы узнали, как обучить сеть sequence-to-sequence и как с ее помощью генерировать ответы на последовательности. В следующих разделах шаг за шагом мы пройдем процесс обучения чат-бота. Для этого мы будем использовать корпус диалогов из фильмов Корнеллского университета[1]. Мы научим сеть sequence-to-sequence «адекватно» реагировать на вопросы и высказывания. Наш пример чат-бота представляет собой взятый из блога Keras пример преобразования последовательностей в последовательности.

10.4.1. Подготовка корпуса для обучения

Во-первых, необходимо загрузить корпус и сгенерировать на его основе тренировочные наборы данных. Тренировочные данные определяют набор поддерживаемых кодировщиком — декодировщиком символов при обучении и на этапе генерации. Обратите, пожалуйста, внимание, что эта реализация не поддерживает символы, не включавшиеся в данные в процессе обучения. Использование всего набора дан-

[1] См. веб-страницу Cornell Movie-Dialogs Corpus по адресу https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html.

ных диалогов из фильмов Корнеллского университета потребовало бы значительных вычислительных затрат, поскольку в нескольких последовательностях более 2000 токенов, а развертывание 2000 временны́х шагов требует немало времени. Но большая часть примеров диалогов содержит менее 100 символов. Для текущего примера мы выполним предварительную обработку корпуса диалогов, ограничив перечень примеров данных только теми, которые содержат менее 100 символов, удалим нестандартные символы и будем использовать только символы в нижнем регистре. Таким образом мы ограничим множество символов. Предварительно обработанный корпус вы можете найти в репозитории GitHub для этой книги[1].

Мы пройдемся по файлу корпуса и сгенерируем тренировочные пары (формально тройки: входной текст, целевой текст с токеном начала и целевой текст). При чтении корпуса мы также сгенерируем наборы входных и целевых символов, которые затем будем использовать для унитарного кодирования примеров данных. Входные и целевые символы не обязательно должны совпадать. Но не включенные в эти наборы символы нельзя будет читать и генерировать на этапе генерации. В результате листинга 10.8 получаются два списка входных и целевых текстов (строк), а также два набора символов, встреченных в тренировочном корпусе.

Листинг 10.8. Создаем тренировочный набор данных для преобразования последовательностей символов в другие последовательности символов

Эти множества содержат символы, встреченные во входном и целевом тексте

Эти массивы содержат входной и целевой текст, прочитанный из файла корпуса

Целевая последовательность снабжена токенами начала (первый) и останова (последний); а здесь описаны соответствующие этим токенам символы. Эти токены не могут включаться в последовательность обычного текста и должны использоваться только в качестве токенов начала и останова

Переменная max_training_samples определяет, сколько строк используется для обучения. Она равна минимуму из заданного пользователем максимума и общего числа загруженных из файла строк

```
>>> from nlpia.loaders import get_data
>>> df = get_data('moviedialog')
>>> input_texts, target_texts = [], []
>>> input_vocabulary = set()
>>> output_vocabulary = set()
>>> start_token = '\t'
>>> stop_token = '\n'
>>> max_training_samples = min(25000, len(df) - 1)

>>> for input_text, target_text in zip(df.statement, df.reply):
...     target_text = start_token + target_text \
...         + stop_token
...     input_texts.append(input_text)
...     target_texts.append(target_text)
...     for char in input_text:
...         if char not in input_vocabulary:
...             input_vocabulary.add(char)
...     for char in target_text:
...         if char not in output_vocabulary:
...             output_vocabulary.add(char)
```

Переменную target_text необходимо обернуть в токены начала и останова

Компиляция словаря — набора уникальных символов, встреченных в input_texts

[1] См. веб-страницу GitHub — totalgood/nlpia по адресу https://github.com/totalgood/nlpia.

10.4.2. Формирование словаря символов

Подобно примерам из предыдущих глав, нам нужно преобразовать все символы входных и целевых текстов в соответствующие им унитарные векторы. Для генерации унитарных векторов мы сгенерируем словари токенов (для входного и целевого текстов), в которых каждому символу соответствует индекс. Мы также сгенерируем обратный словарь (соответствие индексов символам), который будем использовать на этапе генерации для преобразования сгенерированного индекса в символ (листинг 10.9).

Листинг 10.9. Параметры модели преобразования последовательностей символов в последовательности символов

Для входных и целевых данных также определяем максимальное число токенов в последовательности

Преобразуем наборы символов в отсортированные списки символов, на основе которых затем сгенерируем словарь

Для входных и целевых данных определяем максимальное число уникальных символов, используемое для создания унитарных матриц

Проходим в цикле input_vocabulary и output_vocabulary для создания словарей быстрого поиска, с помощью которых сгенерируем унитарные векторы

Проходим в цикле по только что созданным словарям для создания обратных словарей быстрого поиска

```
>>> input_vocabulary = sorted(input_vocabulary)
>>> output_vocabulary = sorted(output_vocabulary)

>>> input_vocab_size = len(input_vocabulary)
>>> output_vocab_size = len(output_vocabulary)
>>> max_encoder_seq_length = max(
...     [len(txt) for txt in input_texts])
>>> max_decoder_seq_length = max(
...     [len(txt) for txt in target_texts])

>>> input_token_index = dict([(char, i) for i, char in
...     enumerate(input_vocabulary)])
>>> target_token_index = dict(
...     [(char, i) for i, char in enumerate(output_vocabulary)])
>>> reverse_input_char_index = dict((i, char) for char, i in
...     input_token_index.items())
>>> reverse_target_char_index = dict((i, char) for char, i in
...     target_token_index.items())
```

10.4.3. Генерируем унитарные тренировочные наборы данных

На следующем этапе мы преобразуем входной и целевой текст в унитарные «тензоры». Для этого пройдем в цикле по всем входным и целевым примерам данных и по всем символам всех примеров данных. Произведем унитарное кодирование этих символов. Каждый символ кодируется с помощью вектора $n \times 1$ (где n — число

уникальных входных или целевых символов). Далее все векторы объединяются в матрицу для каждого из примеров данных, а все примеры данных объединяются в тренировочный тензор (листинг 10.10).

Листинг 10.10. Формирование тренировочного набора данных кодировщика — декодировщика последовательностей символов

```
>>> import numpy as np

>>> encoder_input_data = np.zeros((len(input_texts),
...      max_encoder_seq_length, input_vocab_size),
...      dtype='float32')
>>> decoder_input_data = np.zeros((len(input_texts),
...      max_decoder_seq_length, output_vocab_size),
...      dtype='float32')
>>> decoder_target_data = np.zeros((len(input_texts),
...      max_decoder_seq_length, output_vocab_size),
...      dtype='float32')

>>> for i, (input_text, target_text) in enumerate(
...             zip(input_texts, target_texts)):
...     for t, char in enumerate(input_text):
...         encoder_input_data[
...             i, t, input_token_index[char]] = 1.
...     for t, char in enumerate(target_text):
...         decoder_input_data[
...             i, t, target_token_index[char]] = 1.
...         if t > 0:
...             decoder_target_data[i, t - 1 , target_token_index[char]] = 1
```

Для операций с матрицами используем библиотеку NumPy

Задаем начальные значения тренировочных тензоров в виде нулевых тензоров формы (num_samples, max_len_sequence, num_unique_tokens_in_vocab)

Проходим в цикле по тренировочным примерам данных; входные и выходные тексты должны соответствовать друг другу

Проходим в цикле по всем символам всех примеров данных

Задаем соответствующий символу индекс на каждом временном шаге равным 1; остальные индексы остаются равными 0. В результате получается унитарное представление тренировочного примера данных

В качестве тренировочных данных для декодировщика создаем decoder_input_data и decoder_target_data (отстающий на один временной шаг от decoder_input_data)

10.4.4. Обучение нашего чат-бота sequence-to-sequence

После подготовки тренировочного набора данных — преобразования предварительно обработанного корпуса во входные и целевые примеры данных, создания словарей индексов и преобразования примеров данных в унитарные тензоры — пришло время обучить наш чат-бот. Код ничем не отличается от предыдущих примеров. После завершения методом `model.fit` обучения мы получим полностью обученный чат-бот на основе сети преобразования последовательностей в последовательности (листинг 10.11).

Листинг 10.11. Формирование и обучение сети кодировщика — декодировщика последовательностей символов

```
>>> from keras.models import Model
>>> from keras.layers import Input, LSTM, Dense

>>> batch_size = 64
>>> epochs = 100
>>> num_neurons = 256

>>> encoder_inputs = Input(shape=(None, input_vocab_size))
>>> encoder = LSTM(num_neurons, return_state=True)
>>> encoder_outputs, state_h, state_c = encoder(encoder_inputs)
>>> encoder_states = [state_h, state_c]

>>> decoder_inputs = Input(shape=(None, output_vocab_size))
>>> decoder_lstm = LSTM(num_neurons, return_sequences=True,
...                     return_state=True)
>>> decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
...     initial_state=encoder_states)
>>> decoder_dense = Dense(output_vocab_size, activation='softmax')
>>> decoder_outputs = decoder_dense(decoder_outputs)
>>> model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

>>> model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
...               metrics=['acc'])
>>> model.fit([encoder_input_data, decoder_input_data],
...      decoder_target_data, batch_size=batch_size, epochs=epochs,
...      validation_split=0.1)
```

В этом примере мы задали размер пакета равным 64 примерам данных. Увеличение размера пакета ускорит обучение, но потребует больше оперативной памяти

Обучение сети преобразования последовательностей в последовательности может длиться очень долго и запросто потребовать 100 эпох

В этом примере мы задали число нейронов равным 256

10 % примеров данных мы откладываем для проводимой после каждой эпохи контрольной проверки

10.4.5. Формируем модель для генерации последовательностей

Описание модели для генерации последовательностей практически идентично тому, о котором речь шла в предыдущих разделах. Требуются лишь небольшие изменения из-за отсутствия конкретного целевого текста, который можно было бы передать декодировщику вместе с состоянием. У нас есть только входной сигнал и токен начала (листинг 10.12).

Листинг 10.12. Формирование модели генератора ответов

```
>>> encoder_model = Model(encoder_inputs, encoder_states)
>>> thought_input = [
...     Input(shape=(num_neurons,)), Input(shape=(num_neurons,))]
>>> decoder_outputs, state_h, state_c = decoder_lstm(
...     decoder_inputs, initial_state=thought_input)
```

```
>>> decoder_states = [state_h, state_c]
>>> decoder_outputs = decoder_dense(decoder_outputs)

>>> decoder_model = Model(
...     inputs=[decoder_inputs] + thought_input,
...     output=[decoder_outputs] + decoder_states)
```

10.4.6. Предсказание последовательности

Ядром генерации ответов нашего чат-бота является функция `decode_sequence`. Она принимает на входе унитарно закодированную входную последовательность, генерирует вектор идеи, на основе которого генерирует подходящий ответ с помощью обученной ранее сети (листинг 10.13).

Листинг 10.13. Создание переводчика на основе символов

```
>>> def decode_sequence(input_seq):
...     thought = encoder_model.predict(input_seq)

...     target_seq = np.zeros((1, 1, output_vocab_size))
...     target_seq[0, 0, target_token_index[stop_token]
...         ] = 1.
...     stop_condition = False
...     generated_sequence = ''

...     while not stop_condition:
...         output_tokens, h, c = decoder_model.predict(
...             [target_seq] + thought)

...         generated_token_idx = np.argmax(output_tokens[0, -1, :])
...         generated_char = reverse_target_char_index[generated_token_idx]
...         generated_sequence += generated_char
...         if (generated_char == stop_token or
...                 len(generated_sequence) > max_decoder_seq_length
...                 ):
...             stop_condition = True
...         target_seq = np.zeros((1, 1, output_vocab_size))
...         target_seq[0, 0, generated_token_idx] = 1.
...         thought = [h, c]
...     return generated_sequence
```

Генерируем вектор идеи — входной сигнал для декодировщика

В отличие от этапа обучения начальное значение target_seq — нулевой тензор

Первым входным токеном для декодировщика служит токен начала

Передаем уже сгенерированные токены и последнее состояние декодировщику для предсказания очередного элемента последовательности

Присвоение переменной stop_condition значения True приводит к прекращению цикла

Обновляем состояние вектора идеи

Обновляем целевую последовательность и используем последний сгенерированный токен в качестве входного сигнала для следующего шага генерации

10.4.7. Генерация ответа

Теперь мы опишем вспомогательную функцию `response()`, служащую для преобразования входной строки (например, высказывания пользователя-человека) в пригодный для использования чат-ботом ответ. Эта функция сначала преобразует входной текст пользователя в последовательность унитарных векторов. Затем тензор

унитарных векторов передается в описанную выше функцию `decode_sequence()`. В результате входной текст кодируется в векторах идей, на основе которых затем генерируется выходной текст.

ПРИМЕЧАНИЕ

Главное, что вместо передачи декодировщику начального состояния (вектора идеи) и входной последовательности мы передаем только вектор идеи и токен начала. Токен, генерируемый декодировщиком на основе начального состояния и токена начала, становится входным сигналом для декодировщика на временно́м шаге 2. Выходной сигнал на временно́м шаге 3 становится входным сигналом на временно́м шаге 4 и т. д. Тем временем состояние памяти LSTM обновляет память и наращивает выходной сигнал — точно как в главе 9.

Проходим в цикле все символы входного текста и генерируем унитарный тензор, на основе которого кодировщик генерирует вектор идеи

```
>>> def response(input_text):
...     input_seq = np.zeros((1, max_encoder_seq_length, input_vocab_size),
...         dtype='float32')
...     for t, char in enumerate(input_text):
...         input_seq[0, t, input_token_index[char]] = 1.
...     decoded_sentence = decode_sequence(input_seq)
...     print('Bot Reply (Decoded sentence):', decoded_sentence)
```

Используем функцию decode_sequence для обращения к обученной модели и генерации последовательности символов ответа

10.4.8. Общаемся с нашим чат-ботом

Вуаля! Вы только что завершили все шаги, необходимые для обучения и использования вашего собственного чат-бота. Поздравляем! Интересно, на что чат-бот может дать ответ? После 100 эпох обучения, которые заняли примерно семь с половиной часов на NVIDIA GRID K520 GPU, обученный чат-бот преобразования последовательностей в последовательности все равно оставался немного упрямым и неразговорчивым. Такое поведение мог бы изменить больший и более универсальный тренировочный корпус:

```
>>> response("what is the internet?")
Bot Reply (Decoded sentence): it's the best thing i can think of anything.

>>> response("why?")
Bot Reply (Decoded sentence): i don't know. i think it's too late.

>>> response("do you like coffee?")
Bot Reply (Decoded sentence): yes.

>>> response("do you like football?")
Bot Reply (Decoded sentence): yeah.
```

ПРИМЕЧАНИЕ

Если вы не хотите настраивать GPU и обучать свой собственный чат-бот — не проблема. Мы выложили для ваших экспериментов уже обученный чат-бот. В репозитории GitHub к этой книге[1] вы можете найти самую свежую версию нашего чат-бота. Не забудьте дать знать авторам, с какими забавными ответами чат-бота вы столкнетесь.

10.5. Усовершенствования

Существует два возможных способа усовершенствовать обучение моделей sequence-to-sequence, чтобы повысить их точность и масштабируемость. Как и при обучении людей, глубокому обучению не помешает грамотно составленная учебная программа. Чтобы обеспечить быстрое усвоение, необходимо распределить по категориям и упорядочить учебный материал, равно как и гарантировать, что преподаватель подчеркивает наиболее важные части каждого из документов.

10.5.1. Упрощаем обучение с помощью группирования данных

Длины входных последовательностей могут быть различны, из-за чего к коротким последовательностям в тренировочных данных придется прибавить большое количество токенов дополнения. Слишком много дополнений серьезно повышает вычислительные затраты, особенно если большинство последовательностей — короткие и длина лишь нескольких близка к максимальной. Представьте себе, что мы обучаем сеть sequence-to-sequence на данных, в которых длина почти всех примеров данных равна 100 токенам, за исключением нескольких аномальных, содержащих 1000 токенов. Без группирования данных (bucketing) пришлось бы дополнять большинство тренировочных данных 900 токенами дополнения, по всем из которых сети sequence-to-sequence пришлось бы проходить в цикле на этапе обучения. Подобное дополнение катастрофически замедлило бы обучение. Группирование данных помогает в подобных случаях снизить необходимое количество вычислений. Можно отсортировать последовательности по длине и использовать различную длину последовательности при обработке разных наборов (batch). Входные последовательности распределяются по группам в зависимости от длины. Допустим, в одну группу попадают все последовательности длиной от 5 до 10 токенов, и эти группы последовательностей используются для обучающих наборов, например, сначала производится обучение с помощью всех последовательностей длиной от 5 до 10 токенов, от 10 до 15 и т. д. Некоторые фреймворки глубокого обучения предоставляют утилиты для обработки данных с помощью группировки, которые предлагают оптимальные группы для конкретных входных данных.

Как показано на рис. 10.10, последовательности были сначала отсортированы по длине, а затем дополнены до максимальной длины в токенах для конкретной

[1] См. веб-страницу GitHub — totalgood/nlpia (https://github.com/totalgood/nlpia).

группы. Таким образом можно снизить число временны́х шагов, необходимых для каждой конкретной группы при обучении сети sequence-to-sequence. Сеть развертывается ровно настолько (до самой длинной последовательности), насколько это необходимо для конкретного тренировочного набора.

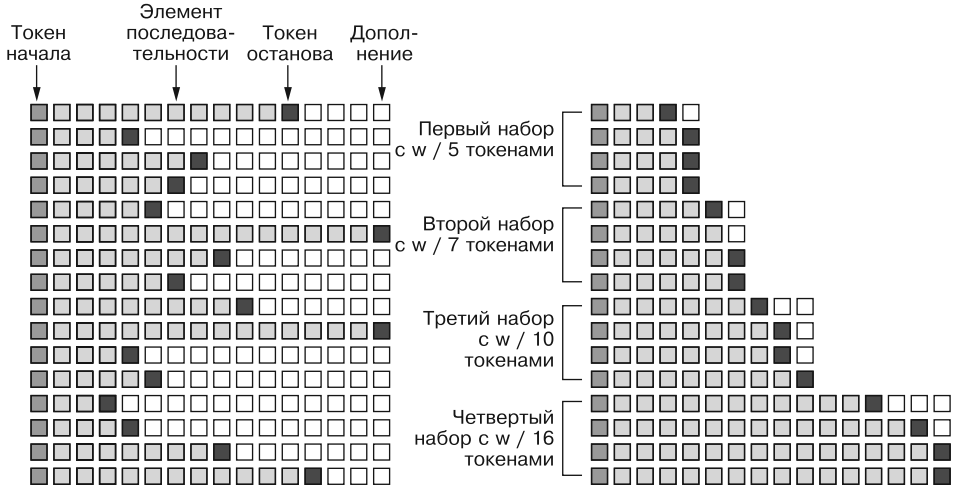


Рис. 10.10. Группирование целевых последовательностей

10.5.2. Механизм внимания

Как и в случае латентно-семантического анализа, с которым вы познакомились в главе 4, более длинные входные последовательности (документы) приводят к векторам идей — они менее точно отражают эти документы. Вектор идеи ограничен размерностью LSTM-слоя (числом нейронов). Отдельного вектора идеи достаточно для коротких входных/выходных последовательностей, таких как в нашем примере чат-бота. Но представьте себе сценарий, когда нужно обучить модель sequence-to-sequence создавать краткие изложения онлайн-статей. В этом случае роль входной последовательности будет играть длинная статья, которую придется сжать в один вектор идеи для генерации, скажем, заголовка. Как вы можете представить, обучить сеть находить наиболее релевантную информацию в подобном длинном документе не так-то просто. Заголовок или краткое изложение (и соответствующий вектор идеи) должны фокусироваться на конкретном аспекте или фрагменте этого документа, а не пытаться отразить всю сложность его смысла.

В 2015 году Богданов (Bahdanau) и др. представили свое решение этой задачи на Международной конференции по усвоению представлений (International Conference on Learning Representations)[1]. Разработанный авторами принцип получил название *ме-*

[1] См. веб-страницу Neural Machine Translation by Jointly Learning to Align and Translate по адресу https://arxiv.org/abs/1409.0473.

ханизма внимания (attention mechanism) (рис. 10.11). Как следует из названия, смысл в том, чтобы указать декодировщику, на что нужно обратить внимание во входной последовательности. Подобный «предварительный просмотр» достигается за счет того, что декодировщику, помимо вектора идеи, разрешается заглядывать назад (вплоть до самого начала), в состояния сети-кодировщика. Вместе с остальной частью сети усваивается своеобразная карта интенсивности (по всей входной последовательности). Далее это отображение, различное на разных временных шагах, становится доступным декодировщику. При декодировании любого фрагмента последовательности созданный из вектора идеи концепт может быть дополнен непосредственно сгенерированной декодировщиком информацией. Другими словами, механизм внимания делает возможной прямую связь между выходным и входным сигналами за счет выбора релевантных фрагментов входного сигнала. Этот процесс не подразумевает выравнивания «токен — токен», которое лишило бы всю процедуру смысла и вернуло бы нас опять к автокодировщику. Но благодаря этому возможны более насыщенные информацией представления встречающихся в последовательности понятий.

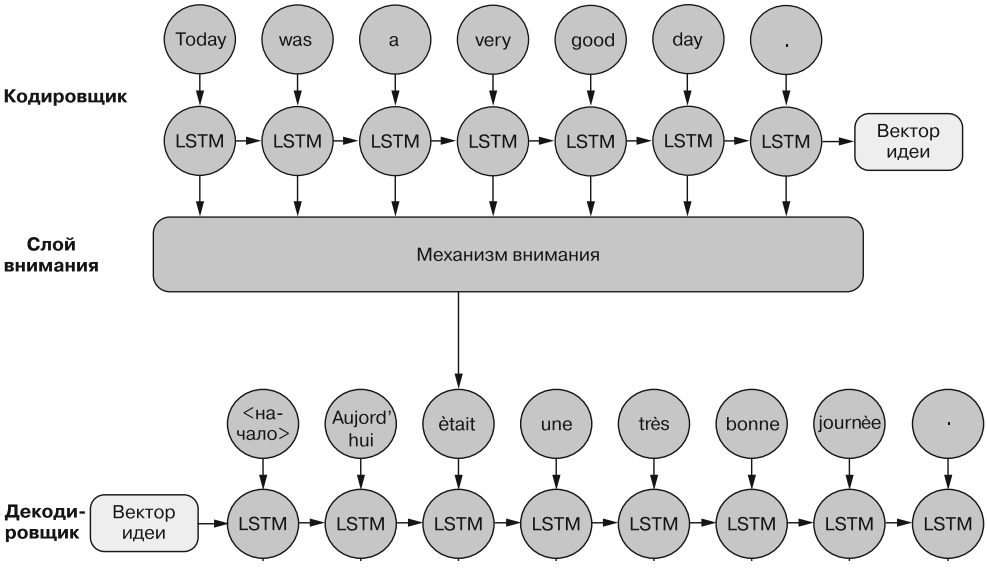


Рис. 10.11. Обзор механизма внимания

Благодаря механизму внимания декодировщик получает дополнительный входной сигнал на каждом временно́м шаге, соответствующий одному или нескольким токенам входной последовательности, на которые необходимо обратить внимание на этом временно́м шаге. Все позиции полученной от кодировщика последовательности представляются в виде взвешенного среднего для каждого временно́го шага декодировщика.

Тонкая настройка механизма внимания — задача непростая, но в ее решении могут помочь предоставляемые различными фреймворками глубокого обучения

реализации. На момент написания данной книги запрос на внесение соответствующих изменений в пакет Keras находился на этапе обсуждения, но никакой реализации пока что не было в него включено.

10.6. На практике

Сети sequence-to-sequence отлично подходят для любых приложений машинного обучения с входными или выходными последовательностями переменной длины. Поскольку длина последовательностей естественного языка почти всегда непредсказуема, модели sequence-to-sequence могут повысить точность большинства моделей машинного обучения в этой сфере.

Основные приложения преобразования последовательностей в последовательности:

- разговоры чат-ботов;
- формирование ответов на вопросы;
- машинный перевод;
- создание подписей к изображениям;
- формирование ответов на визуальные вопросы;
- автоматическое реферирование документов.

Как вы видели в предыдущих главах, диалоговые системы — распространенное приложение NLP. Модели преобразования последовательностей в последовательности являются порождающими, благодаря чему они особенно хорошо подходят для речевых диалоговых систем (чат-ботов). Чат-боты, использующие преобразование последовательностей в последовательности, порождают более разнообразные, креативные речевые диалоги, чем информационный поиск или чат-боты на основе баз знаний. Речевые диалоговые системы имитируют человеческий разговор на разные темы. Чат-боты, использующие преобразование последовательностей в последовательности, способны производить обобщение на основе корпуса с ограниченной предметной областью и давать разумные ответы на темы, не содержащиеся в их тренировочном наборе данных. Напротив, сужение предметной области диалоговых систем на базе знаний (о которых речь пойдет в главе 12) ограничивает их способность участвовать в разговорах на темы, выходящие за рамки предметной области их обучения. В главе 12 мы сравним эффективность различных архитектур чат-ботов подробнее.

Помимо корпуса диалогов из фильмов Корнеллского университета, существуют и другие разнообразные бесплатные и открытые тренировочные наборы данных, например наборы данных Q&A компании DeepMind[1, 2]. Если диалоговая система долж-

[1] Набор данных в форме вопросов и ответов: https://cs.nyu.edu/~kcho/DMQA/.

[2] Список диалоговых корпусов в документации пакета nlpia: https://github.com/totalgood/nlpia/blob/master/docs/notes/nlp--data.md#dialog-corpora.

на уверенно отвечать в рамках определенной предметной области, ее необходимо обучать на корпусах высказываний из этой предметной области. Информационная емкость вектора идеи ограничена, поэтому его нужно заполнять информацией по темам, на которые должен общаться чат-бот.

Еще одно распространенное приложение сетей преобразования последовательностей в последовательности — машинный перевод. Понятие вектора идеи позволяет осуществляющему перевод приложению учитывать *контекст* входных данных и переводить в правильном контексте многозначные слова. Если вы хотите создавать приложения для перевода, посмотрите сайт ManyThings (http://www.manythings.org/anki/), — там можно найти пары предложений, подходящие для использования в качестве тренировочных наборов данных. Мы включили эти пары для вас в пакет `nlpia`. Например, для получения пар англо-немецких соответствий высказываний можете заменить в листинге 10.8 `get_data('moviedialog')` на `get_data('deu-eng')`.

Модели преобразования последовательностей в последовательности также отлично подходят для автоматического реферирования благодаря различию длины строки на входе и выходе. В этом случае входным сигналом сети-кодировщика могут служить, например, новостные статьи (или любые другие длинные документы), а декодировщик можно обучить генерировать последовательность для заголовка, реферата или краткого изложения документа. Сети преобразования последовательностей в последовательности позволяют создавать звучащие более естественно краткие изложения текста, чем это делают методы автоматического реферирования на основе статистики векторов мультимножеств слов. Если вы желаете разработать подобное приложение, отличный тренировочный набор данных доступен в конкурсе Kaggle по созданию сводок новостей[1].

Сфера применения сетей sequence-to-sequence не ограничивается обработкой естественного языка. Еще два их приложения: автоматическое распознавание речи и создание подписей к изображениям. Современные, созданные по последнему слову науки и техники системы автоматического распознавания речи[2] используют сети sequence-to-sequence для преобразования последовательностей примеров голосового входного сигнала в вектор идеи, который далее декодировщик может преобразовать в текстовую расшифровку речи. Аналогично осуществляется и создание подписей к изображениям. Последовательность пикселов изображения (независимо от разрешения изображения) используется в качестве входного сигнала кодировщика, а декодировщик обучается генерировать подходящее описание. На самом деле существует приложение, сочетающее создание подписей к изображениям и систему формирования ответов на вопросы, именуемое формированием ответов на визуальные вопросы: https://vqa.cloudcv.org/.

---

[1] См. веб-страницу NEWS SUMMARY: Kaggle по адресу https://www.kaggle.com/sunnysai12345/news-summary/data.

[2] Современные системы автоматического распознавания речи: https://arxiv.org/pdf/1610.03022.pdf.

Резюме

- Сети преобразования последовательностей в последовательности можно создавать на основе модульной, повторно используемой архитектуры типа «кодировщик — декодировщик».
- Модель-кодировщик генерирует вектор идеи — плотное векторное представление фиксированной размерности для информации, содержащейся во входной последовательности переменной длины.
- Декодировщик использует векторы идей для предсказания (генерации) выходных последовательностей, включая ответы чат-ботов.
- Благодаря представлению вектора идеи длины входной и выходной последовательностей не обязаны совпадать.
- Вектор идеи может включать лишь ограниченный объем информации. Если нужно закодировать с помощью вектора идеи более сложные понятия, можно использовать механизм внимания для выборочного кодирования наиболее важных частей вектора идеи.

Часть III

Поговорим серьезно. Реальные задачи NLP

Часть III поможет вам довести свои навыки до уровня, позволяющего решать реальные задачи. Вы узнаете, как выделять такую информацию, как даты и имена, создавать приложения типа ботов Twitter, помогавших управлять расписанием открытых встреч (open spaces) конференции PyCon US в 2017 и 2018 годах.

В трех последних главах мы затронем более сложные задачи NLP. Покажем несколько различных способов создания чат-ботов, как на основе машинного обучения, так и без него. И научим вас комбинировать эти методики для создания сложного поведения. Мы также расскажем про алгоритмы обработки больших корпусов данных — наборов документов, которые невозможно целиком загрузить в оперативную память.

11 Выделение информации: выделение поименованных сущностей и формирование ответов на вопросы

В этой главе

- Сегментация предложений.
- Выделение поименованных сущностей (NER).
- Выделение числовой информации.
- Частеречная (POS) разметка и разбор дерева зависимостей.
- Выделение логических отношений и базы знаний.

Чтобы вы смогли создать полнофункциональный чат-бот, необходим еще один навык: умение выделять информацию (знания) из текста на естественном языке.

11.1. Поименованные сущности и отношения

Нам нужно, чтобы машина могла выделять из текста элементы информации и факты, чтобы знать хотя бы немного, что говорит пользователь. Например, пользователь сказал: *Remind me to read aiindex.org on Monday* («Напомни мне почитать aiindex.org в понедельник»). Необходимо, чтобы это высказывание инициировало запись в календаре или оповещающий сигнал на ближайший понедельник после текущей даты.

Для активации этого действия нужно знать, что *me* является определенным видом *поименованной сущности* (named entity): действующим лицом. И чат-бот должен понимать, что это слово нужно «расширить» (нормализовать), заменив его именем пользователя действующего лица. Чат-бот также должен распознать *aiindex.org* как сокращенный URL, поименованную сущность — название конкретного экземпляра чего-то. И должен знать, что нормализованное написание этого вида поименованных сущностей выглядит как *http://aiindex.org*, *https://aiindex.org* или, возможно, даже *https://www.aiindex.org*. Аналогично чат-бот должен распознать *Monday* как один из дней недели (еще один вид поименованных сущностей — «событие») и найти его в календаре.

Чтобы чат-бот мог корректно ответить на этот простой запрос, необходимо также выделить отношение поименованной сущности *me* и команды *remind*. Нужно даже распознать в предложении подразумеваемое подлежащее *you*, относящееся к чат-боту, еще одной поименованной сущности — действующему лицу. И нам необходимо «приучить» чат-бот к тому, что напоминания происходят в будущем, поэтому ему нужно найти ближайший будущий понедельник и создать напоминание.

Типичное предложение может содержать несколько поименованных сущностей различных типов, например географические сущности, организации, людей, политические сущности, время (в том числе даты), артефакты, события и природные явления. Предложение может также содержать несколько отношений — фактов о взаимосвязях между поименованными сущностями в предложении.

11.1.1. База знаний

Помимо просто выделения информации из текста пользовательского высказывания, можно также использовать выделение информации для обучения чат-бота! Если заставить чат-бот выделять информацию из большого корпуса, например из «Википедии», он извлечет факты об окружающем мире, которые могут повлиять на будущее поведение и ответы чат-бота. Некоторые чат-боты фиксируют всю выделяемую информацию (из назначенных им «домашних заданий» по чтению) в своей базе знаний. Далее можно выполнять запросы к этой базе знаний, что позволяет чат-боту принимать более взвешенные решения и делать осознанные выводы об окружающем мире.

Чат-боты могут также хранить информацию о текущем сеансе общения с пользователем (разговоре). Относящаяся только к текущему разговору информация называется *контекстом*. Подобные контекстуальные знания можно хранить в той же глобальной базе знаний чат-бота или в отдельной базе знаний. Коммерческие API чат-ботов, например Watson компании IBM или Lex компании Amazon, обычно хранят контекст отдельно от глобальной базы знаний фактов, используемых для поддержания разговоров с другими пользователями.

Контекст может включать факты о пользователе, чате или канале или о текущей погоде и новостях. Контекст может даже включать изменения состояния самого чат-бота, основанного на данном разговоре. Пример «знания системы о себе»: интеллектуальный чат-бот должен отслеживать историю всего им сказанного кому-либо или заданных пользователю вопросов, чтобы не повторяться.

Таким образом, цель данной главы — научить бот понимать читаемое им. И это «понимание» мы поместим в гибкую структуру данных, специально предназначенную для хранения знаний. Далее чат-бот сможет использовать эти знания для принятия решений и умных высказываний относительно окружающего мира.

Помимо упомянутой простой задачи распознавания в тексте чисел и дат, мы хотели бы, чтобы наш бот мог выделять более общую информацию об окружающем мире, причем делать это самостоятельно, чтобы не приходилось «программировать» в него все наши знания из данной области. Например, мы хотели бы, чтобы он мог обучаться на документах на естественном языке, вроде следующего предложения из «Википедии»:

In 1983, Stanislav Petrov, a lieutenant colonel of the Soviet Air Defense Forces, saved the world from nuclear war.

Конспектируя на уроке истории что-либо подобное (прочитанное или услышанное), вы бы, вероятно, перефразировали немного и создали в уме связи между понятиями или словами. И сжали бы это до элемента информации, того, что вы из этого «вынесли». Хотелось бы, чтобы наш бот умел делать то же самое. Желательно, чтобы он умел «конспектировать» изученное, например, тот факт (знание), что Станислав Петров был подполковником. Данный факт можно сохранить в примерно следующей структуре данных:

```
('Stanislav Petrov', 'is-a', 'lieutenant colonel')
```

Это пример двух узлов поименованных сущностей (`'Stanislav Petrov'` и `'lieutenant colonel'`) и отношения/связи (`'is a'`) между ними в графе/базе знаний. При хранении подобного отношения в форме, соответствующей стандарту RDF (relation description format, формат описания отношений) для графов знаний, о нем говорят как о RDF-тройке. Исторически такие RDF-тройки хранятся в XML-файлах, но их можно хранить в любом файловом формате или базе данных, подходящей для хранения графов троек в виде (`субъект, отношение, объект`).

Набор таких троек представляет собой граф знаний. Лингвисты иногда называют его онтологией, поскольку он хранит структурированную информацию о словах. Но если граф должен отражать факты об окружающем мире, а не просто слова, его называют графом знаний или базой знаний. Рисунок 11.1 наглядно демонстрирует граф знаний, который мы хотели бы выделить из подобного предложения.

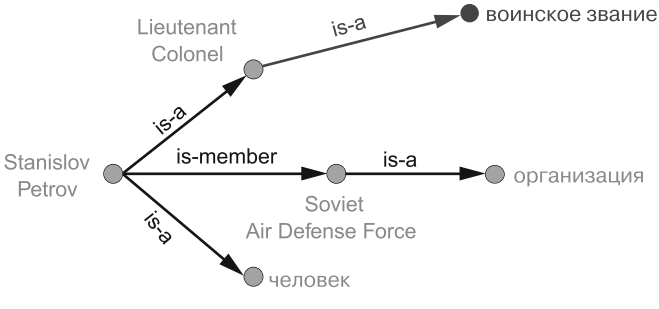


Рис. 11.1. Граф знаний о Станиславе

Взаимосвязь *is-a* на рис. 11.1 вверху отражает факт, который нельзя непосредственно выделить из приведенного высказывания о Станиславе. Но тот факт, что *lieutenant colonel* — это воинское звание, может следовать из того, что звание человека, состоящего в военной организации, является воинским. Такая логическая операция по извлечению фактов из графа знаний называется *выводом* (inference) из графа знаний. Ее можно также назвать запросом к базе знаний, по аналогии с запросом к реляционной базе данных. Для конкретного вывода (запроса) относительно воинского звания Станислава граф знаний должен содержать факты о военных и воинских званиях. Не помешают также факты в базе знаний о званиях и профессиях. Наверное, вы уже поняли, почему с помощью базы знаний машина может лучше понять высказывание, чем без нее. Без этой базы знаний многие факты в подобном простом высказывании окажутся недоступными для чат-бота. Можно даже сказать, что вопросы относительно связанных с родом деятельности званий окажутся «выходящими за пределы служебных обязанностей» чат-бота, умеющего только классифицировать документы по распределенным случайным образом темам[1].

То, насколько это важно, наверное, не совсем понятно, но это **очень** важно. Если вам случалось когда-то общаться с чат-ботом, который буквально не понимает «где право, а где лево», то вы представляете, о чем речь. Одна из наиболее сложных задач ИИ — компиляция и организация эффективных запросов к графу общечеловеческих знаний, а мы считаем общечеловеческие знания чем-то само собой разумеющимся при обычных разговорах.

Люди обретают немалую часть знаний еще до того, как обучаются говорить. Мы не проводим все детство, записывая информацию о том, что день начинается, когда становится светло, а спать обычно ложатся после заката солнца. И мы не редактируем статьи в «Википедии» о том, что пустой желудок надо заполнять едой, а не грязью или камнями. Поэтому так сложно найти корпус общечеловеческих знаний для просмотра и обучения машин. Не существует посвященных общечеловеческим знаниям статей «Википедии», из которых наш бот мог бы выделить информацию. И часть этих знаний — инстинктивные, «зашитые» в нашу ДНК[2].

Между вещами и людьми существуют разнообразные виды отношений, например, *kind-of* («является разновидностью»), *is-used-for* («используется для»), *has-a* («имеет»), *is-famous-for* («известен тем, что»), *was-born* («рожден») и *has-profession* («профессия»). Внимание NELL, никогда не прекращающего изучение языка (Never Ending Language Learning) бота Университета Карнеги — Меллона, почти исключительно сосредоточено на задаче выделения информации об отношениях типа «является разновидностью».

Большинство баз знаний нормализуют описывающие эти отношения строки, так что *kind of* и *type of* будут присвоены нормализованная строка или идентификатор, отражающий конкретное отношение. Некоторые базы знаний также производят

---

[1] См. главу 4, если вы забыли, что такое случайное распределение по темам.

[2] Впрочем, существуют жестко запрограммированные базы общечеловеческих знаний, от которых вы можете отталкиваться. В поиске таких графов знаний вам поможет Google Scholar.

разрешение существительных, соответствующих объектам в базе знаний. Так что биграмме *Stanislav Petrov* может быть присвоен конкретный ID. Тот же ID будет присвоен синонимам биграммы *Stanislav Petrov*, таким как *S. Petrov* и *Lt Col Petrov*, если конвейер NLP решит, что они относятся к одному человеку.

Базу знаний можно использовать для создания удобного типа чат-бота: системы формирования ответов на вопросы (question answering system, QA-система). Чат-боты обслуживания клиентов, в том числе боты — учебные помощники в университетах, практически исключительно полагаются на базы знаний для генерации своих ответов[1]. Системы формирования ответов на вопросы очень удобны для людей, которым нужно найти какую-либо фактографическую информацию. И это освобождает их для того, что у них получается лучше, например для обобщения на основе этих фактов. Люди плохо запоминают факты, но хорошо умеют находить связи и закономерности между ними, способность, которой машинам еще только предстоит овладеть. Об отвечающих на вопросы чат-ботах мы поговорим в следующей главе.

11.1.2. Выделение информации

Итак, вы уже знаете, что выделение информации означает преобразование неструктурированного текста в структурированную информацию, хранящуюся в базе или графе знаний. Выделение информации относится к сфере исследований, которая называется «понимание естественного языка» (natural language understanding, NLU), хотя этот термин часто употребляется в качестве синонима для обработки естественного языка.

Выделение информации и NLU отличаются от обучения, о котором обычно говорят в контексте науки о данных. Это не только обучение без учителя; даже сама модель, логика функционирования мира, может создаваться без вмешательства человека. Вместо того чтобы кормить модель рыбой (фактами), мы учим ее держать удочку (выделять информацию). Тем не менее для обучения выделению информации часто применяются методы машинного обучения.

11.2. Регулярные паттерны

Нам нужен алгоритм сопоставления с паттернами, который умел бы идентифицировать соответствующие заданному паттерну последовательности символов/слов, чтобы выделить их из более длинной строки или текста. Наивный способ написания подобного алгоритма сопоставления с паттерном состоит в создании на Python последовательности операторов `if/then` для поиска символа или слова в каждой позиции строки. Допустим, необходимо найти распространенные слова приветствия, например *Hi*, *Hello* и *Yo*, в начале высказывания. Это можно сделать так, как показано в листинге 11.1.

[1] Учебный ассистент на основе ИИ в GaTech, 2016: http://www.news.gatech.edu/2016/05/09/artificial-intelligence-course-creates-ai-teaching-assistant.

Листинг 11.1. Жестко зашитые на Python паттерны

```
>>> def find_greeting(s):
...     """ Возвращает строку приветствия (Hi, etc),
            в случае совпадения приветствия с паттерном"""
...     if s[0] == 'H':
...         if s[:3] in ['Hi', 'Hi ', 'Hi,', 'Hi!']:
...             return s[:2]
...         elif s[:6] in ['Hello', 'Hello ', 'Hello,', 'Hello!']:
...             return s[:5]
...     elif s[0] == 'Y':
...         if s[1] == 'o' and s[:3] in ['Yo', 'Yo,', 'Yo ', 'Yo!']:
...             return s[:2]
...     return None
```

А листинг 11.2 демонстрирует, как эта функция работает.

Листинг 11.2. Хрупкий пример сопоставления с паттерном

```
>>> find_greeting('Hi Mr. Turing!')
'Hi'
>>> find_greeting('Hello, Rosa.')
'Hello'
>>> find_greeting("Yo, what's up?")
'Yo'
>>> find_greeting("Hello")
'Hello'
>>> print(find_greeting("hello"))
None
>>> print(find_greeting("HelloWorld"))
None
```

Возможно, вы обратили внимание, насколько утомительно программирование алгоритма сопоставления с паттерном подобным образом. И этот алгоритм не слишком хорош. Он очень хрупок, зависит от точного написания, регистра слов и позиций символов в строке. Кроме того, непросто указать все возможные разделители, например, знаки препинания, пробелы или знак начала/конца строки (символы `NULL`) на обеих сторонах искомых слов.

Наверное, можно придумать способ задавать различные искомые слова или строки без «зашивания» их в выражения Python, подобные вышеприведенным. И можно даже указывать разделители в отдельной функции. В результате можно было бы находить вхождения искомых слов в произвольном месте строки с помощью токенизации и прохода по строке в цикле. Но это слишком много работы.

К счастью, всю эту работу уже сделали за вас! Механизм сопоставления с паттернами встроен в большинство современных языков программирования, включая Python. Это так называемые регулярные выражения. Регулярные выражения и выражения форматирования для строковой интерполяции (например, `"{:05d}".format(42)`) — сами по себе являются миниатюрными языками программирования. Язык сопоставления с паттернами называется языком регулярных выражений. И в стандартной библиотеке `re` языка Python есть интерпретатор

(компилятор и исполнитель) регулярных выражений. Воспользуемся этими выражениями для описания наших паттернов вместо глубоко вложенных операторов `if` языка Python.

11.2.1. Регулярные выражения

Регулярные выражения — это строки, созданные на специальном машинном языке для описания алгоритмов. Регулярные выражения обладают большими возможностями описания сопоставления с паттернами, они гибче и лаконичнее, чем соответствующий код на языке Python. Поэтому регулярные выражения — оптимальный язык описания паттернов для множества задач NLP, включая сопоставление с паттернами. Это приложение NLP — расширение первоначального его использования для компиляции и интерпретации формальных (машинных) языков.

Регулярное выражение описывает конечный автомат (finite state machine, FSM) — дерево решений типа `if-then`, принимаемых относительно последовательности символов, подобное функции `find_greeting()` из листинга 11.1. Символы последовательности передаются в дерево решений FSM по одному. Обрабатывающий последовательности символов (например, строки символов ASCII) или последовательности английских слов конечный автомат называется *грамматикой* (grammar). Их также называют *формальными грамматиками* (formal grammars), чтобы отличать от грамматических правил естественных языков, которые изучают в средней школе.

В теории вычислительной техники и математике слово «грамматика» означает набор правил, определяющих, является ли последовательность символов допустимым элементом языка, называемого машинным (компьютерным) или формальным языком. Машинный (формальный) язык представляет собой набор всех возможных высказываний, удовлетворяющих задающей этой язык формальной грамматике. Данное определение напоминает порочный круг, но в математике такое иногда бывает. Если вы не знакомы хотя бы с основами синтаксиса регулярных выражений и символами вроде `r'.\*'` и `r'a-z'` — обратитесь к приложению Б.

11.2.2. Выделение информации и признаков при машинном обучении

Итак, мы вернулись опять к главе 1, где впервые упомянуты регулярные выражения. Но разве мы не отказались в конце главы 1 от «грамматических» подходов к NLP в пользу машинного обучения и ориентированных на данные подходов? Зачем же возвращаться к жестко программируемым (составляемым вручную) регулярным выражениям и паттернам? А затем, что у статистических и ориентированных на данные подходов есть свои ограничения.

Мы хотим, чтобы наш конвейер машинного обучения мог производить простейшие действия, вроде ответов на логические вопросы, или заносить в расписание деловые встречи на основе NLP-инструкций. Машинное обучение в этом случае

не дает желаемого результата. Далеко не всегда можно найти маркированный набор данных, охватывающий ответы на все вопросы, которые только могут задать на естественном языке люди. Кроме того, как вы увидите, обычно можно описать краткий набор проверяемых условий (регулярное выражение) для извлечения ключевых элементов информации из строки на естественном языке. И такой метод работает для широкого круга задач.

Сопоставление с паттернами (и регулярные выражения) остается передовым подходом к выделению информации. При подходах к обработке естественного языка на основе машинного обучения приходится производить проектирование признаков. Нужно создавать мультимножества (вложения) слов для попытки свести практически безграничное число смыслов в естественном языке в удобный для машинной обработки вектор. Выделение информации — просто другая форма выявления признаков при машинном обучении из неструктурированных данных на естественном языке (например, создания мультимножества слов или выполнения PCA для этого мультимножества слов). Эти паттерны и признаки все еще используются даже в самых продвинутых конвейерах машинного обучения для естественного языка, например в Assistant компании Google, Siri, Alexa от Amazon и других ультрасовременных ботах.

С помощью выделения информации производится поиск высказываний и информации, которые бот должен «иметь под рукой». Выделить информацию можно и заранее, чтобы заполнить базу знаний о фактах. Или можно искать нужные высказывания и информацию по мере необходимости, при получении чат-ботом вопроса или при запросе к поисковой системе. Когда база знаний создается заранее, можно оптимизировать структуру данных для ускорения запросов в рамках более широких областей знаний. Заранее созданная база знаний позволяет чат-боту быстро отвечать на вопросы, касающиеся широкого диапазона информации. Выделение же информации в режиме реального времени при запросе к чат-боту называется поиском. Google и другие поисковые системы сочетают эти два метода, выполняя запрос к графу знаний (базе знаний) и возвращаясь к текстовому поиску, если нужные факты не нашлись там. Многие из изученных вами в школе грамматических правил естественного языка можно закодировать в виде формальной грамматики, спроектированной для работы со словами или символами, соответствующими частям речи. Английский[1] язык можно рассматривать как состоящий из слов и грамматических правил. Или можно воспринимать его как множество высказываний, которые счел бы допустимыми носитель английского языка.

И это приводит нас к еще одной возможности формальных грамматик и конечных автоматов, удобной для NLP. Машина может использовать любую формальную грамматику двумя способами:

- для поиска соответствующих этой грамматике высказываний;
- для генерации новых последовательностей символов.

Паттерны (регулярные выражения) можно использовать не только для выделения информации из естественного языка, но и в чат-боте, который хочет «говорить»

[1] И русский, конечно. — *Примеч. пер.*

что-то в соответствии с этим паттерном! Мы покажем, как делать это с помощью пакета `rstr` для некоторых из ваших паттернов выделения информации.

Такой подход к сопоставлению с паттернами на основе формальных грамматик и конечных автоматов обладает и другими яркими достоинствами. Можно всегда гарантировать выполнение (останов) настоящего конечного автомата за конечное время. И благодаря ему всегда известно, нашлось соответствие в строке или нет. Можно быть уверенными, что он никогда не попадет в бесконечный цикл... если не пользоваться некоторыми продвинутыми возможностями движков регулярных выражений, позволяющими «жульничать» и встраивать циклы в FSM.

Мы ограничимся регулярными выражениями, не требующими этих «жульничеств» с возвратом назад или заглядываниями вперед. Наша процедура сопоставления с паттерном будет просматривать по одному символу и переходить к следующему только в случае совпадения с паттерном — примерно как суровый контролер в трамвае проходит по салону, проверяя билеты. Если билета у вас нет, контролер останавливается и заявляет о проблеме, несоответствии, отказывается идти дальше или проверять билеты, пока эта проблема не будет решена. Никаких «возвратов назад» или «заглядываний вперед» ни для пассажиров трамвая, ни для строгих регулярных выражений не существует.

11.3. Заслуживающая выделения информация

Вот основные элементы количественной информации, заслуживающие усилий по написанию вручную регулярных выражений:

- GPS-координаты;
- даты;
- цены;
- числа.

Другие важные элементы информации на естественном языке, требующие более сложных паттернов (по сравнению с теми, что легко захватываются с помощью регулярных выражений):

- вопросительные (начинающие вопрос) слова;
- целевые слова вопросов;
- поименованные сущности.

11.3.1. Выделение GPS-координат

GPS-координаты — типичный вид числовых данных, выделяемых из текста с помощью регулярных выражений. GPS-координаты встречаются в виде пар числовых значений широты и долготы. Иногда они также включают третье число — высоту (над уровнем моря), но его мы пока не будем учитывать. Просто выделим десятичную широту/долготу в градусах. Этот метод подходит для многих URL Google Maps. Хотя URL формально не относятся к естественному языку,

они обычно входят в состав неструктурированных текстовых данных, и эту информацию нужно извлечь, чтобы бот располагал знаниями о местоположениях, а не только объектах.

Воспользуемся нашим паттерном для десятичного числа из предыдущих примеров, но внесем определенные ограничения, чтобы значение находилось в допустимом для широты (+/–90°) и долготы (+/–180°) диапазоне. Нет ничего севернее Северного полюса (+90°) и южнее Южного (–90°). И если отплыть из Гринвича на 180° на восток (долгота +180°), мы достигнем линии перемены дат, где окажемся одновременно на 180° на запад (долгота –180°) от Гринвича (листинг 11.3).

Листинг 11.3. Регулярные выражения для GPS-координат

```
>>> import re
>>> lat = r'([-]?[0-9]?[0-9][.][0-9]{2,10})'
>>> lon = r'([-]?1?[0-9]?[0-9][.][0-9]{2,10})'
>>> sep = r'[,/ ]{1,3}'
>>> re_gps = re.compile(lat + sep + lon)

>>> re_gps.findall('http://...maps/@34.0551066,-118.2496763...')
[(34.0551066, -118.2496763)]

>>> re_gps.findall("https://www.openstreetmap.org/#map=10/5.9666/116.0566")
[('5.9666', '116.0566')]

>>> re_gps.findall("Zig Zag Cafe is at 45.344, -121.9431 on my GPS.")
[('45.3440', '-121.9431')]
```

Выделение числовых данных — задача несложная, особенно если эти числа входят в состав пригодной для машинного чтения строки. В URL и других подходящих для машинного чтения строках числа указываются в предсказуемых порядке, формате и единицах, что намного упрощает нашу задачу. Подобный паттерн будет выделять и некоторые неправдоподобные значения широты и долготы, но для большинства URL, которые можно скопировать из приложений географических карт вроде OpenStreetMap, он свое дело делает.

А как насчет дат? Подходят ли регулярные выражения для выделения дат? Что, если нужно, чтобы процедура выделения дат работала и в Европе, и в США, где день/месяц обычно указываются в обратном порядке?

11.3.2. Выделение дат

Выделение дат — задача намного сложнее, чем выделение GPS-координат. Даты гораздо ближе к естественному языку с различными диалектами для выражения одних и тех же понятий. В США дата Рождества 2017 года обозначается как 12/25/17, в Европе же — 25.12.17. Можно проверять региональные настройки пользователей и предполагать, что они пишут даты так же, как большинство людей в их регионе. Но это допущение может оказаться ложным.

Поэтому большинство алгоритмов извлечения даты и времени работают с обеими вариантами порядка дня/месяца и проверяют, получается ли в результате допустимая дата. Именно так работает с датами человеческий мозг. Если вы носитель

американского английского и оказались в Брюсселе на Рождество, вероятно, вы все равно поймете, что означает дата 25/12/17, ведь в году всего 12 месяцев.

Подобная «утиная» типизация, отлично работающая в языках программирования, подходит и для естественного языка. Если что-то выглядит как утка и крякает как утка, то это, вероятно, утка. Если что-то выглядит как дата и ведет себя как дата[1], то это, вероятно, дата. Мы будем использовать подход «сначала делаем, а прощения попросим потом» и для других задач обработки естественного языка. Попробуем несколько вариантов и выберем из них рабочий. Мы будем запускать процесс выделения или генерации, а затем проверки, чтобы выяснить, имеет ли полученный результат смысл.

Такой подход особенно хорошо работает для чат-ботов, поскольку позволяет сочетать возможности нескольких генераторов естественного языка. В главе 10 мы генерировали ответы чат-бота с помощью LSTM. Чтобы пользователю было комфортнее, можно сгенерировать большое количество ответов и выбрать из них тот, который орфографически, грамматически и по тональности более правильный. Мы поговорим об этом подробнее в главе 12 (листинг 11.4).

Листинг 11.4. Регулярное выражение для дат в формате, принятом в США

```
>>> us = r'((([01]?\d)[-/]([0123]?\d))([-/]([0123]\d)\d\d)?)'
>>> mdy = re.findall(us, 'Santa came 12/25/2017. An elf appeared 12/12.')
>>> mdy
[('12/25/2017', '12/25', '12', '25', '/2017', '20'),
 ('12/12', '12/12', '12', '12', '', '')]
```

Для некоторого структурирования этих выделенных данных можно использовать списковое включение, преобразовав месяц, день и год в числа и маркировав эту числовую информацию осмысленным образом, как показано в листинге 11.5.

Листинг 11.5. Структурирование выделенных дат

```
>>> dates = [{'mdy': x[0], 'my': x[1], 'm': int(x[2]), 'd': int(x[3]),
...      'y': int(x[4].lstrip('/') or 0), 'c': int(x[5] or 0)} for x in mdy]
>>> dates
[{'mdy': '12/25/2017', 'my': '12/25', 'm': 12, 'd': 25, 'y': 2017, 'c': 20},
 {'mdy': '12/12', 'my': '12/12', 'm': 12, 'd': 12, 'y': 0, 'c': 0}]
```

Даже для таких простых дат невозможно создать регулярное выражение, разрешающее все неоднозначности во второй дате, 12/12. В языке дат существуют неоднозначности, которые могут понять только люди, зная о Рождестве и намерениях автора текста. Например, 12/12 может означать:

- 12 декабря 2017 года — месяц/день года, предполагаемого на основе разрешения анафор[2];
- 12 декабря 2018 года — месяц/день текущего на момент публикации года;
- декабрь 2012 года — месяц/год в 2012 году.

---

[1] Используется в контексте даты. — *Примеч. пер.*

[2] См. статью: *Sayed I. Q.* Issues in Anaphora Resolution — для стэнфордского курса CS224N (https://nlp.stanford.edu/courses/cs224n/2003/fp/iqsayed/project_report.pdf).

Поскольку в датах в США и в нашем регулярном выражении месяц/день предшествуют году, 12/12 рассматривается как 12 декабря неизвестного года. Любые пропущенные числовые поля можно заполнить текущим годом на основе содержащегося в памяти контекста из структурированных данных, как показано в листинге 11.6.

Листинг 11.6. Простые действия с контекстом

```
>>> for i, d in enumerate(dates):
...     for k, v in d.items():
...         if not v:
...             d[k] = dates[max(i - 1, 0)][k]
>>> dates
[{'mdy': '12/25/2017', 'my': '12/25', 'm': 12, 'd': 25, 'y': 2017, 'c': 20},
 {'mdy': '12/12', 'my': '12/12', 'm': 12, 'd': 12, 'y': 2017, 'c': 20}]
>>> from datetime import date
>>> datetimes = [date(d['y'], d['m'], d['d']) for d in dates]
>>> datetimes
[datetime.date(2017, 12, 25), datetime.date(2017, 12, 12)]
```

Этот код работает, поскольку и словарь, и список — изменяемые типы данных

Это простейший, но достаточно надежный способ выделения информации о дате из текста на естественном языке. Для превращения данного кода в алгоритм для выделения дат осталось только добавить перехват исключений и подходящее для нашего приложения поддержание контекста. Если вы добавите это в пакет `nlpia` (http://github.com/totalgood/nlpia) с помощью запроса на внесение изменений, другие читатели оценят по достоинству, а если вы еще и добавите какие-нибудь процедуры выделения времени — будете настоящим героем.

Есть много возможностей с помощью описываемой вручную логики обрабатывать различные граничные случаи и названия месяцев и даже дней на естественном языке. Но никакие ухищрения не позволят разрешить неоднозначность в дате 12/11. Она может означать:

- 11 декабря любого года, в который вы ее услышали или прочитали;
- 12 ноября, если вы услышали ее в Лондоне или Лонсестоне (Тасмания, территория Содружества);
- декабрь 2011 года, если вы прочитали ее в американской газете;
- ноябрь 2012 года, если вы прочитали ее в европейской газете.

Некоторые неоднозначности естественного языка не может понять даже человеческий мозг. Но мы сделаем так, чтобы наш процесс выделения дат мог справиться с европейским порядком дней/месяцев, поменяв порядок месяца и дня в регулярном выражении (листинг 11.7).

Листинг 11.7. Регулярное выражение для европейских дат

```
>>> eu = r'((([0123]?\d)[-/]([01]?\d))([-/]([0123]\d)?\d\d)?)'
>>> dmy = re.findall(eu, 'Alan Mathison Turing OBE FRS (23/6/1912-7/6/1954) \
...     was an English computer scientist.')
```

```
>>> dmy
[('23/6/1912', '23/6', '23', '6', '/1912', '19'),
 ('7/6/1954', '7/6', '7', '6', '/1954', '19')]
>>> dmy = re.findall(eu, 'Alan Mathison Turing OBE FRS (23/6/12-7/6/54) \
...     was an English computer scientist.')
>>> dmy
[('23/6/12', '23/6', '23', '6', '/12', ''),
 ('7/6/54', '7/6', '7', '6', '/54', '')]
```

Это регулярное выражение правильно выделяет даты рождения и смерти Тьюринга из фрагмента статьи «Википедии». Но мы немного сжульничали — преобразовали месяц *June* в число 6, прежде чем протестировали это регулярное выражение на предложении из «Википедии». Так что этот пример не слишком реалистичен. И остается еще одна неразрешенная неоднозначность: год, поскольку столетие не указано. Означает ли 54 год 1954-й или 2054-й? Желательно, чтобы наш чат-бот умел выделять даты из статей «Википедии» в их первозданном виде, чтобы изучать известных людей и важные даты. Для работы нашего регулярного выражения с датами более естественного языка, такими как в статьях «Википедии», необходимо добавить в выделяющее даты регулярное выражение такие слова, как *June* (и все их сокращенные формы).

Не нужно никаких специальных символов, которые бы отмечали слова (сочетания символов в последовательности). Их можно указывать в регулярном выражении точно в том виде, в котором они должны встречаться во входных данных, включая регистр. Все, что нужно, — указать между ними символ `OR` (`|`) в регулярном выражении. И следует убедиться, что оно может обрабатывать американский, а не только европейский порядок месяца/дня. Мы добавим эти два альтернативных написания дат в регулярное выражение с помощью «большого» `OR` (`|`) между ними в качестве ветки дерева решений в регулярном выражении.

Воспользуемся поименованными группами для идентификации годов вроде 84 в качестве 1984, а 08 в качестве 2008. И немного уточним диапазон соответствий наших четырехзначных годов, ограничившись годами в будущем вплоть до 2399-го и в прошлом — начиная с года 0[1] (листинг 11.8).

Листинг 11.8. Распознавание годов

```
>>> yr_19xx = (
...     r'\b(?P<yr_19xx>' +
...     '|'.join('{}'.format(i) for i in range(30, 100)) +
...     r')\b'
...     )
>>> yr_20xx = (
...     r'\b(?P<yr_20xx>' +
...     '|'.join('{:02d}'.format(i) for i in range(10)) + '|' +
...     '|'.join('{}'.format(i) for i in range(10, 30)) +
```

Двузначные годы: 30–99 = 1930–1999

[1] См. веб-страницу «0 год» (https://ru.wikipedia.org/wiki/0_год).

```
...     r')\b'
...     )
>>> yr_cent = r'\b(?P<yr_cent>' + '|'.join(
...     '{}'.format(i) for i in range(1, 40)) + r')'
>>> yr_ccxx = r'(?P<yr_ccxx>' + '|'.join(
...     '{:02d}'.format(i) for i in range(0, 100)) + r')\b'
>>> yr_xxxx = r'\b(?P<yr_xxxx>(' + yr_cent + ')(' + yr_ccxx + r'))\b'
>>> yr = (
...     r'\b(?P<yr>' +
...     yr_19xx + '|' + yr_20xx + '|' + yr_xxxx +
...     r')\b'
...     )
>>> groups = list(re.finditer(
...     yr, "0, 2000, 01, '08, 99, 1984, 2030/1970 85 47 `66"))
>>> full_years = [g['yr'] for g in groups]
>>> full_years
['2000', '01', '08', '99', '1984', '2030', '1970', '85', '47', '66']
```

Одно- или двузначные годы: 01–30 = 2001–2030

Первые цифры трех- или четырехзначного года, например, 1 в 123 A.D. или 20 в 2018

Последние две цифры трех- или четырехзначного года, например, 23 в 123 A.D. или 18 в 2018

Ух ты! Сколько работы, и все для задания простых правил написания годов в регулярном выражении, а не просто на Python. Не волнуйтесь, существуют пакеты для распознавания распространенных форматов дат. Они работают точнее (намного меньше ложных соответствий) и являются более универсальными (меньше пропущенных дат). Так что писать подобные сложные регулярные выражения самостоятельно обычно не нужно. Этот пример — шаблон на случай, если вам когда-нибудь понадобится выделить конкретный тип числовых данных с помощью регулярного выражения. Примерами, когда могут пригодиться более сложные регулярные выражения, с поименованными группами, служат денежные величины и IP-адреса.

Завершим наше регулярное выражение для выделения дат, добавив паттерны для названий месяцев, например *June* или *Jun* в дате рождения Тьюринга в «Википедии», как показано в листинге 11.9.

Листинг 11.9. Распознавание словесных названий месяцев с помощью регулярных выражений

```
>>> mon_words = 'January February March April May June July ' \
...     'August September October November December'
>>> mon = (r'\b(' + '|'.join('{}|{}|{}|{}|{:02d}'.format(
...     m, m[:4], m[:3], i + 1, i + 1) for i, m in
➥ enumerate(mon_words.split())) +
...     r')\b')
>>> re.findall(mon, 'January has 31 days, February the 2nd month
➥ of 12, has 28, except in a Leap Year.')
['January', 'February', '12']
```

Есть идеи, как объединить регулярные выражения в одно большое, способное учитывать как европейские, так и американские даты? Одна из сложностей здесь — невозможность использовать повторно одно и то же название группы (заключенной в круглые скобки части регулярного выражения). Так что поставить `OR` между регулярными выражениями для наименований месяца и года в европейском и американском порядке нельзя. И нужно включить также паттерны для некоторых необязательных разделителей между днем, месяцем и годом.

Вот один из способов решить эту задачу (листинг 11.10).

Листинг 11.10. Объединение регулярных выражений для выделения информации о датах

```
>>> day = r'|'.join('{:02d}|{}'.format(i, i) for i in range(1, 32))
>>> eu = (r'\b(' + day + r')\b[-,/ ]{0,2}\b(' +
...     mon + r')\b[-,/ ]{0,2}\b(' + yr.replace('<yr', '<eu_yr') + r')\b')
>>> us = (r'\b(' + mon + r')\b[-,/ ]{0,2}\b(' +
...     day + r')\b[-,/ ]{0,2}\b(' + yr.replace('<yr', '<us_yr') + r')\b')
>>> date_pattern = r'\b(' + eu + '|' + us + r')\b'
>>> list(re.finditer(date_pattern, '31 Oct, 1970 25/12/2017'))
[<_sre.SRE_Match object; span=(0, 12), match='31 Oct, 1970'>,
 <_sre.SRE_Match object; span=(13, 23), match='25/12/2017'>]
```

Наконец, нужно проверить эти даты, убедившись, что их можно преобразовать в допустимые объекты типа `datetime` языка Python, как показано в листинге 11.11.

Листинг 11.11. Проверка дат

```
>>> import datetime
>>> dates = []
>>> for g in groups:
...     month_num = (g['us_mon'] or g['eu_mon']).strip()
...     try:
...         month_num = int(month_num)
...     except ValueError:
...         month_num = [w[:len(month_num)]
...             for w in mon_words].index(month_num) + 1
...     date = datetime.date(
...         int(g['us_yr'] or g['eu_yr']),
...         month_num,
...         int(g['us_day'] or g['eu_day']))
...     dates.append(date)
>>> dates
[datetime.date(1970, 10, 31), datetime.date(2017, 12, 25)]
```

Похоже, алгоритм выделения дат работает нормально, по крайней мере для нескольких простых, однозначных дат. Задумайтесь, как пакеты вроде `Python-dateutil` и `datefinder` разрешают неоднозначности и обрабатывают написанные на более естественном языке даты, вроде *today* и *next Monday*. И если вам кажется, что вы могли бы решить эти задачи лучше, отправьте им запрос на внесение изменений!

Если же вам нужен ультрасовременный алгоритм выделения дат, обратитесь к статистическим (основанным на машинном обучении) подходам. Библиотека SUTime из пакета Stanford CoreNLP (https://nlp.stanford.edu/software/sutime.html) и `dateutil.parser.parse` компании Google — именно эти.

11.4. Выделение взаимосвязей (отношений)

До сих пор мы говорили только о выделении таких экземпляров существительных, как даты и значения GPS-координат широты и долготы. И имели дело в основном с числовыми паттернами. Пришло время заняться более сложной задачей выделения знаний из естественного языка. Хотелось бы, чтобы наш бот мог обучаться

фактам об окружающем мире из энциклопедии, например «Википедии». Желательно, чтобы он умел соотносить даты и GPS-координаты с сущностями, о которых он читает.

Какие знания может извлечь ваш мозг из следующего предложения из «Википедии»?

On March 15, 1554, Desoto wrote in his journal that the Pascagoula people ranged as far north as the confluence of the Leaf and Chickasawhay rivers at 30.4, –88.5.

С помощью выделения дат и GPS-координат вы могли связать эту дату и местоположение с Десото, племенем паскагула и двумя реками, название второй из них вы даже не можете выговорить. Нам нужно, чтобы бот (и наш мозг) мог связать эти факты с более крупными фактами — например, с тем, что Десото[1] был испанским конкистадором, а паскагула — мирное индейское племя. Желательно также, чтобы даты и местоположения связывались с правильными понятиями: Десото и слиянием двух рек соответственно.

Именно так большинство людей себе и представляют термин «понимание естественного языка». Чтобы понять высказывание, необходимо выделить из него ключевую информацию и соотнести ее с соответствующими знаниями. В случае с машинами эти знания хранятся в виде графа — базы знаний. Ребрами этого графа являются взаимосвязи между разными понятиями, а его вершинами — существительные или объекты из корпуса.

Для выделения этих взаимосвязей (отношений) мы воспользуемся паттернами типа «подлежащее — сказуемое — дополнение». Для распознавания паттернов конвейер NLP должен знать, к какой части речи относится каждое из слов предложения.

11.4.1. Частеречная (POS) разметка

Выполнить частеречную разметку можно с помощью языковых моделей, включающих словари слов со всеми возможными частями речи. Далее на правильно маркированных предложениях их можно обучить распознавать части речи в новых предложениях, содержащих слова, не входящие в этот словарь. Как в NLTK, так и в spaCy реализованы функции частеречной разметки. Мы воспользуемся библиотекой spaCy, поскольку она точнее и быстрее работает (листинг 11.12).

Для создания нашего графа знаний необходимо выяснить, какие объекты (именные группы) нужно связать друг с другом попарно. Хотелось бы связать дату *March 15, 1554* с поименованной сущностью *Desoto*. Далее можно разрешить эти две строки (именные группы) так, чтобы они указывали на объекты из нашей базы знаний. *March 15, 1554* можно преобразовать в объект типа `datetime.date` с нормализованным представлением.

Разобранные с помощью spaCy предложения также содержат во вложенном словаре дерево зависимостей, а `spacy.displacy` может сгенерировать строку с *мас-*

[1] Правильно «де Сото». — *Примеч. пер.*

штабируемой векторной графикой SVG (или полной HTML-страницей), которую можно просмотреть как изображение в браузере. Такая визуализация, вероятно, поможет при поиске способов создания паттернов частеречной разметки для выделения отношений (листинг 11.13).

Листинг 11.12. POS-разметка с помощью spaCy

```
>>> import spacy
>>> en_model = spacy.load('en_core_web_md')
>>> sentence = ("In 1541 Desoto wrote in his journal that the Pascagoula people " +
...     "ranged as far north as the confluence of the Leaf and Chickasawhay
          rivers at 30.4, -88.5.")
>>> parsed_sent = en_model(sentence)
>>> parsed_sent.ents
(1541, Desoto, Pascagoula, Leaf, Chickasawhay, 30.4)

>>> ' '.join(['{}_{}'.format(tok, tok.tag_) for tok in parsed_sent])
'In_IN 1541_CD Desoto_NNP wrote_VBD in_IN his_PRP$ journal_NN that_IN
  the_DT Pascagoula_NNP people_NNS ranged_VBD as_RB far_RB north_RB
  as_IN the_DT confluence_NN of_IN the_DT Leaf_NNP and_CC Chickasawhay_NNP
  rivers_VBZ at_IN 30.4_CD ,_, -88.5_NFP ._.'
```

spaCy пропустила долготу в числовой паре широта/долгота

spaCy использует частеречную разметку OntoNotes 5: https://spacy.io/api/annotation#pos-tagging

Листинг 11.13. Визуализация дерева зависимостей

```
>>> from spacy.displacy import render
>>> sentence = "In 1541 Desoto wrote in his journal about the Pascagoula."
>>> parsed_sent = en_model(sentence)
>>> with open('pascagoula.html', 'w') as f:
...     f.write(render(docs=parsed_sent, page=True,
➥ options=dict(compact=True)))
```

Дерево зависимостей для этого короткого предложения демонстрирует, что именная группа *the Pascagoula* — объект взаимосвязи *met* для подлежащего *Desoto* (рис. 11.2). Причем оба существительных получили теги, соответствующие именам собственным.

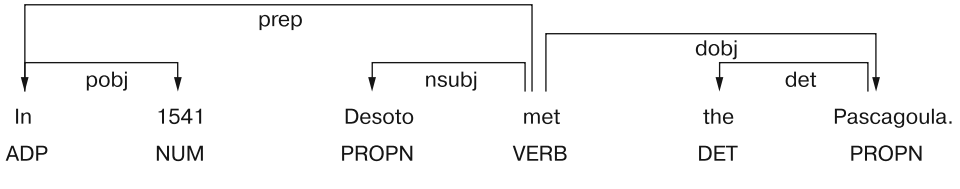


Рис. 11.2. Племя паскагула

Для создания паттернов частей речи и свойств слов для `spacy.matcher.Matcher` есть смысл перечислить все теги токенов в таблице. В листинге 11.14 приведены некоторые упрощающие эту задачу вспомогательные функции.

Листинг 11.14. Вспомогательные функции для размеченных строк spaCy

```
>>> import pandas as pd
>>> from collections import OrderedDict
>>> def token_dict(token):
...     return OrderedDict(ORTH=token.orth_, LEMMA=token.lemma_,
...         POS=token.pos_, TAG=token.tag_, DEP=token.dep_)

>>> def doc_dataframe(doc):
...     return pd.DataFrame([token_dict(tok) for tok in doc])

>>> doc_dataframe(en_model("In 1541 Desoto met the Pascagoula."))
          ORTH       LEMMA    POS  TAG    DEP
0           In          in    ADP   IN   prep
1         1541        1541    NUM   CD   pobj
2       Desoto      desoto  PROPN  NNP  nsubj
3          met        meet   VERB  VBD   ROOT
4          the         the    DET   DT    det
5   Pascagoula  pascagoula  PROPN  NNP   dobj
6            .           .  PUNCT    .  punct
```

Теперь можно найти последовательность свойств `POS` или `TAG`, из которых получится хороший паттерн. Если нас интересуют взаимосвязи *has-met* между людьми и организациями, то подойдут паттерны вроде `PROPN met PROPN`, `PROPN met the PROPN`, `PROPN met with the PROPN` и `PROPN often meets with PROPN`. Можно указывать все эти паттерны по отдельности или попытаться захватить их все с помощью операторов `*` или `?`, задающих паттерн типа «любое слово» между нашими именами собственными:

```
'PROPN ANYWORD? met ANYWORD? ANYWORD? PROPN'
```

Возможности паттернов в spaCy намного больше и гибче, чем в предыдущем псевдокоде, так что необходимо подробно описывать, с какими именно чертами слов нужно найти совпадение. При задании паттернов spaCy словарь используется для захвата всех тегов, которые нужно сопоставить с каждым токеном или словом, как показано в листинге 11.15.

Листинг 11.15. Пример POS-паттерна spaCy

```
>>> pattern = [{'TAG': 'NNP', 'OP': '+'}, {'IS_ALPHA': True, 'OP': '*'},
...            {'LEMMA': 'meet'},
...            {'IS_ALPHA': True, 'OP': '*'}, {'TAG': 'NNP', 'OP': '+'}]
```

Далее можно извлечь нужные размеченные токены из разобранного предложения (листинг 11.16).

Листинг 11.16. Создание сопоставления с паттерном с помощью spaCy

```
>>> from spacy.matcher import Matcher
>>> doc = en_model("In 1541 Desoto met the Pascagoula.")
>>> matcher = Matcher(en_model.vocab)
>>> matcher.add('met', None, pattern)
>>> m = matcher(doc)
```

```
>>> m
[(12280034159272152371, 2, 6)]

>>> doc[m[0][1]:m[0][2]]
Desoto met the Pascagoula
```

Итак, мы нашли совпадение с нашим паттерном в исходном предложении, на основе которого был создан паттерн. Но что насчет аналогичных предложений из «Википедии» (листинг 11.17)?

Листинг 11.17. Использование сопоставления с паттерном POS

```
>>> doc = en_model("October 24: Lewis and Clark met their first
      Mandan Chief, Big White.")
>>> m = matcher(doc)[0]
>>> m
(12280034159272152371, 3, 11)

>>> doc[m[1]:m[2]]
Lewis and Clark met their first Mandan Chief

>>> doc = en_model("On 11 October 1986, Gorbachev and Reagan met at a house")
>>> matcher(doc)
[]
```

Этому паттерну не соответствует ни одна подстрока предложения из «Википедии»

Необходимо добавить второй паттерн, чтобы учесть ситуацию, когда глагол встречается после подлежащего и дополнения, как показано в листинге 11.18.

Листинг 11.18. Объединение нескольких паттернов для более надежного сопоставления

```
>>> doc = en_model("On 11 October 1986, Gorbachev and Reagan met at a house")
>>> pattern = [{'TAG': 'NNP', 'OP': '+'}, {'LEMMA': 'and'}, {'TAG': 'NNP', 'OP': '+'},
...            {'IS_ALPHA': True, 'OP': '*'}, {'LEMMA': 'meet'}]
>>> matcher.add('met', None, pattern)
>>> m = matcher(doc)
>>> m
[(14332210279624491740, 5, 9),
 (14332210279624491740, 5, 11),
 (14332210279624491740, 7, 11),
 (14332210279624491740, 5, 12)]

>>> doc[m[-1][1]:m[-1][2]]
Gorbachev and Reagan met at a house
```

Добавляем дополнительный паттерн без удаления предыдущего. 'met' здесь — это лишь произвольный ключ, можете называть паттерн так, как вам удобно

Операторы '+' повышают число пересекающихся альтернативных соответствий

Самое длинное совпадение — последнее в списке совпадений

Итак, у нас есть сущности и взаимосвязь. Можно даже создать паттерн, позволяющий более гибкий выбор глагола посередине *(met)* и более жестко ограничивающий имена людей и групп с обеих сторон. Благодаря этому можно найти дополнительные глаголы, указывающие на встречу одного человека или группы с другими, например глагол *knows* или даже фразы в пассивном залоге вроде *had a conversation* или *became acquainted with*. После этого можно использовать новые глаголы для добавления взаимосвязей для новых имен собственных с обеих сторон.

Но вы, наверное, заметили, что мы отступаем от исходного смысла наших начальных паттернов взаимосвязей. Это называется *семантическим дрейфом* (semantic drift). К счастью, spaCy не только размечает слова в разобранном документе их метками частей речи и информацией дерева зависимостей, но также предоставляет вектор слов Word2vec, который можно использовать для предотвращения дрейфа соединительного глагола и имен собственных с обеих сторон от исходного смысла начального паттерна[1].

11.4.2. Нормализация имен сущностей

Нормализованное представление сущности обычно представляет собой строку, даже для числовой информации вроде дат. Нормализованный формат ISO для упомянутой даты имеет вид 1541-01-01. Нормализованное представление сущностей позволяет базе знаний связать все произошедшее в мире в одну дату с одним узлом (сущностью) графа.

То же самое можно сделать и для других поименованных сущностей: исправить написание слов и попытаться разрешить неоднозначности для названий/имен объектов, животных, людей, мест и т. д. Нормализацию поименованных сущностей и разрешение неоднозначностей называют *разрешением кореференций* (coreference resolution) или *разрешением анафор* (anaphora resolution) — особенно это относится к местоимениям и другим названиям, зависящим от контекста. Это напоминает лемматизацию, о которой говорилось в главе 2. Благодаря нормализации поименованных сущностей можно быть уверенными, что словарь сущностей не будет «замусорен» противоречивыми, избыточными названиями.

Например, *Desoto* может быть упомянут в конкретном документе как минимум пятью разными способами:

- *de Soto*;
- *Hernando de Soto*;
- *Hernando de Soto (c. 1496/1497–1542), Spanish conquistador*;
- https://en.wikipedia.org/wiki/Hernando_de_Soto (URI);
- числовой ID в базе данных известных исторических личностей.

Аналогично наш алгоритм нормализации может выбрать любую из этих форм. Граф знаний должен нормализовать каждый из типов сущностей одинаково, чтобы несколько различных сущностей одного типа не получили одно название. Желательно, чтобы на одного человека не ссылалось несколько различных имен. И что более важно, нормализация должна применяться согласованно — как при записи новых фактов в базу знаний, так и при ее чтении или запросе к ней.

Если нужно изменить подход к нормализации после заполнения базы данных, необходимо изменить данные для уже существующих записей в базе знаний, чтобы

[1] Этот вопрос сейчас активно изучается исследователями: https://nlp.stanford.edu/pubs/structuredVS.pdf.

они соответствовали новой схеме нормализации. Бессхемные базы данных (хранилища пар «ключ/значение»), подобные используемым для хранения графов или баз знаний, также подлежат миграции, как и реляционные базы данных. В конце концов, бессхемные базы данных «внутри» являются интерфейсными обертками для реляционных баз данных[1].

Нормализованные сущности должны связываться с помощью взаимосвязей *is-a*, с категориями сущностей. Эти взаимосвязи можно рассматривать как теги, поскольку у каждой сущности может быть несколько взаимосвязей *is-a*. Как и имена людей, и теги частей речи, даты и другие дискретные числовые объекты должны нормализоваться, чтобы их можно было включить в базу знаний.

А как насчет *отношений* между сущностями? Нужно ли их хранить в каком-либо нормализованном виде?

11.4.3. Нормализация и выделение отношений

Теперь для выяснений видов взаимосвязей между сущностями нужен способ нормализации взаимосвязей. Это позволит найти все связи типа «родился в» между людьми и датами или даты исторических событий, например встречи между *Hernando de Soto* и *Pascagoula people*. И нам нужно написать алгоритм выбора правильных меток для взаимосвязей.

Причем у этих взаимосвязей могут быть иерархические названия, например *occurred-on/approximately* или *occurred-on/exactly*, с помощью которых можно находить конкретные взаимосвязи или категории взаимосвязей. Можно также маркировать эти взаимосвязи числовыми метками для «доверия», вероятности, веса или нормализованной частотности (аналогично TF-IDF для термов/слов) этой взаимосвязи. Можно также корректировать эти уровни доверия всякий раз, когда выделенный из нового текста факт подтверждает существующий факт из базы данных или противоречит ему.

Теперь нам нужно найти способ сопоставления с паттернами, пригодными для поиска таких взаимосвязей.

11.4.4. Паттерны слов

Паттерны слов аналогичны регулярным выражениям, но для слов вместо символов. Взамен классов символов у нас будут классы слов. Например, вместо сопоставления с символом в нижнем регистре можно использовать паттерн слов для поиска всех существительных в единственном числе (POS-тег *NN*)[2]. Для этого обычно обращаются к машинному обучению. Несколько начальных предложений размечается тегами правильных взаимосвязей (фактов), выделенными из этих предложений.

---

[1] Довольно спорное утверждение. NoSQL базы данных отказываются от некоторых базовых принципов реляционных баз данных. — *Примеч. пер.*

[2] В spaCy используются POS-теги OntoNotes 5: https://spacy.io/api/annotation#pos-tagging.

Паттерн POS можно использовать для поиска аналогичных предложений, отличающихся подлежащим, дополнениями или даже взаимосвязями.

Пакет spaCy можно применять для сопоставления с такими паттернами за (фиксированное) время O(1), вне зависимости от количества паттернов, двумя способами:

- используя класс `PhraseMatcher` для произвольных паттернов последовательностей слов/тегов[1];
- с помощью класса `Matcher` для паттернов последовательностей тегов[2].

Чтобы найденные в новых предложениях новые отношения были действительно аналогичны исходным начальным примерам взаимосвязей, часто приходится ограничивать смысл подлежащего, взаимосвязи и дополнения близкими к содержащимся в начальных предложениях. Лучше всего делать это с помощью какого-либо векторного представления смыслов слов. Знакомо, правда? Векторы слов, о которых речь шла в главе 4, — одни из чаще всего используемых для этой цели представлений смысла слов. Они позволяют минимизировать семантический дрейф.

Благодаря семантическим векторным представлениям слов и фраз автоматическое выделение информации стало достаточно точным для автоматического создания больших баз знаний. Но для разрешения многих неоднозначностей в тексте на естественном языке требуется присмотр и контроль со стороны человека. Бот NELL[3] Университета Карнеги — Меллона позволяет пользователям голосовать за вносимые в базу знаний изменения в Twitter и веб-приложениях.

11.4.5. Сегментация

Мы пропустили в описании одну из форм выделения информации, а равно и используемую при выделении информации утилиту. Большинство документов в этой главе представляют собой небольшие порции, содержащие лишь несколько фактов и поименованных сущностей. Но на практике эти порции часто приходится создавать самостоятельно.

Разбиение документа на порции удобно для разработки частично структурированных данных, упрощающих поиск, фильтрацию и сортировку документа для выделения информации. Если вы извлекаете информацию для выделения отношений с целью создания базы знаний вроде NELL или Freebase, необходимо разбить документ на части, содержащие 1–2 факта. Разбиение текста на естественном языке на осмысленные фрагменты называется *сегментацией* (segmentation). Полученные в результате сегменты могут быть фразами, предложениями, цитатами, абзацами или даже целыми разделами длинного документа.

---

[1] См. веб-страницу Code Examples — spaCy Usage Documentation по адресу https://spacy.io/usage/examples#phrase-matcher.

[2] См. веб-страницу Matcher — spaCy API Documentation по адресу https://spacy.io/api/matcher.

[3] См. веб-страницу NELL: The Computer that Learns — Carnegie Mellon University по адресу https://www.cmu.edu/homepage/computing/2010/fall/nell-computer-that-learns.shtml.

Предложение — наиболее распространенный вид порций текста в большинстве задач выделения информации. Предложения обычно отделяются одним из знаков препинания («.», «?», «!» или символом новой строки). Грамматически правильное предложение на английском языке должно содержать подлежащее (имя существительное) и сказуемое (глагол), так что в нем обычно есть по крайней мере одно заслуживающее выделения отношение или факт. Кроме того, предложения обычно являются завершенными смысловыми пакетами, зачастую предшествующий текст не нужен для того, чтобы читатель понял смысл.

К счастью, в большинстве языков, включая английский, есть понятие предложения — отдельное высказывание с подлежащим и сказуемым, содержащее какую-либо информацию об окружающем мире. Размеры предложений оптимально подходят для использования их в качестве порций текста для нашего NLP-конвейера выделения знаний. Итак, наша задача для конвейера чат-бота — сегментировать документы на предложения (высказывания).

Помимо упрощения выделения информации, эти высказывания и предложения можно маркировать, например, как части диалога или подходящие для ответов в диалоге. Использование сегментатора предложений позволяет обучать чат-бот на более длинных текстах, например на книгах. Подбор соответствующих книг делает стиль чат-бота более литературным, интеллигентным, чем обучение его исключительно на данных из Twitter или чатах в IRC. И благодаря этим книгам у чат-бота появляется более широкий доступ к обучающим документам для формирования базы общечеловеческих знаний об окружающем мире.

Сегментация предложений

Сегментация предложений обычно является первым шагом в конвейере выделения информации. Она помогает изолировать факты друг от друга, чтобы связать правильную цену с правильным товаром в строке, например: *The Babel fish costs $42. 42 cents for the stamp*. Эта строка — яркий пример трудностей сегментации предложений — точку посередине можно интерпретировать либо как десятичный разделитель, либо как точку в конце предложения.

Простейшие элементы информации, которые можно выделить из документа, — последовательности слов, представляющие собой логически связные высказывания. Важнейшими сегментами в документах на естественном языке после слов являются предложения. Они представляют собой логически связные высказывания об окружающем мире. Часто они, констатируя какие-либо факты, указывают на взаимосвязь между вещами и описывают устройство мира, так что их можно использовать для выделения информации. Кроме того, предложения объясняют, когда, где и как происходило что-то в прошлом, происходит обычно или произойдет в будущем. Поэтому необходимо также уметь выделять факты относительно дат, моментов времени, местоположений, людей и даже последовательностей событий или задач с помощью предложений в качестве ориентира. И, что самое важное, во всех естественных языках существуют предложения либо иные логически связные фрагменты текста. И во всех языках есть общеупотребительный процесс их генерации (набор грамматических правил либо практик).

Но сегментация текста, распознавание границ предложений — задача более сложная, чем может показаться. В английском языке, например, не существует отдельного знака препинания или последовательности знаков, которые всегда бы указывали на конец предложения.

11.4.6. Почему не получится разбить по ('.!?')

Даже читатель-человек не всегда сможет найти правильную границу предложения в следующих цитатах. И если он найдет в каждой из них несколько предложений, то ошибется в четырех из пяти этих непростых примеров.

I live in the U. S. but I commute to work in Mexico on S. V. Australis for a woman from St. Bernard St. on the Gulf of Mexico.

I went to G. T. You?

She yelled "It's right here!" but I kept looking for a sentence boundary anyway.

I stared dumbfounded on as things like "How did I get here?," "Where am I?," "Am I alive?" flittered across the screen.

The author wrote "'I don't think it's conscious.' Turing said."

Больше подобных «граничных случаев» сегментации предложений можно найти на сайте tm-town.com[1] и в модуле `nlpia.data`.

Особенно трудно разбивать на предложения технический текст, поскольку инженеры, ученые и математики используют точки и восклицательные знаки для множества других обозначений, помимо конца предложения. Когда мы попробовали искать границы предложений в этой книге, нам пришлось вручную исправить несколько выделенных предложений.

Ах, если бы предложения писались как телеграммы, с «тчк» или каким-то уникальным знаком препинания в конце каждого предложения. Но раз это не так, нужна более продвинутая NLP, а не просто `split('.!?')`. Надеемся, вы уже мысленно спроектировали решение. Если да, то, вероятно, в его основе лежит один из двух использовавшихся на протяжении этой книги подходов к NLP:

- программирование алгоритмов вручную (регулярные выражения и сопоставление с паттернами);
- статистические модели (модели на основе данных или машинное обучение).

Воспользуемся задачей сегментации предложений, чтобы вернуться к этим двум подходам, и продемонстрируем вам, как искать границы предложений с помощью регулярных выражений или перцептронов. Причем в качестве тренировочного и тестового наборов данных мы будем использовать текст этой книги, чтобы указать

[1] См. веб-страницу Natural Language Processing: TM-Town: https://www.tm-town.com/natural-language-processing#golden_rules.

вам на некоторые сложные случаи. К счастью, мы не вставляли никаких символов новой строки внутри предложений для переноса текста вручную, как в макетах газет. В противном случае задача оказалась бы еще сложнее. На самом деле большая часть исходного текста этой книги, в формате ASCII, была написана либо со «старыми добрыми» разделителями предложений (два пробела после конца каждого предложения), либо с каждым предложением на отдельной строке. Именно благодаря этому мы можем использовать данную книгу в качестве тренировочного и тестового наборов данных для сегментаторов.

11.4.7. Сегментация предложений с помощью регулярных выражений

Регулярные выражения — просто сокращенный способ записи деревьев правил `if...then` (правил регулярной грамматики) для поиска паттернов символов в строках символов. Как мы упоминали в главах 1 и 2, регулярные выражения (регулярные грамматики) — особенно компактный способ задания правил конечного автомата. Единственная задача нашего регулярного выражения (FSM) — распознавание границ предложений.

Если поискать в Интернете сегментаторы предложений[1], можно найти разнообразные регулярные выражения, предназначенные для захвата наиболее распространенных границ предложений. Вот некоторые из них, комбинированные и усовершенствованные. В результате получился быстрый, универсальный сегментатор предложений. Следующее регулярное выражение подходит для нескольких вариантов «обычных» предложений:

```
>>> re.split(r'[!.?]+[ $]', "Hello World.... Are you there?!?! I'm going to Mars!")
['Hello World', 'Are you there', "I'm going to Mars!"]
```

К сожалению, подход с использованием метода `re.split` приводит к отбрасыванию завершающего предложение токена, сохраняя его лишь тогда, когда он является последним символом в документе или строке. Зато он игнорирует «хитрые» точки внутри дважды вложенных кавычек:

```
>>> re.split(r'[!.?] ', "The author wrote \"'I don't think it's conscious.'
     Turing said.\"")
['The author wrote "\'I don\'t think it\'s conscious.\' Turing said."']
```

Но, к сожалению, он игнорирует и точки внутри кавычек, завершающие настоящее предложение. Хорошо это или плохо — зависит от следующих за сегментацией предложений шагов выделения информации:

```
>>> re.split(r'[!.?] ', "The author wrote \"'I don't think it's conscious.'
➥ Turing said.\" But I stopped reading.")
['The author wrote "\'I don\'t think it\'s conscious.\' Turing said." But I
➥ stopped reading."']
```

[1] См. результаты поиска по фразе Python sentence segment в подсистеме DuckDuckGo (https://duckduckgo.com/?q=Python+sentence+segment&t=canonical&ia=qa).

А как насчет сокращенного написания текста, например СМС или твитов? Иногда в спешке люди смешивают предложения, не ставя пробелы, где нужно. Само по себе следующее регулярное выражение может справиться лишь с точками в СМС, по обеим сторонам от которых находятся буквы, и успешно игнорировать числовые значения:

```
>>> re.split(r'(?<!\d)\.|\.(?!\d)', "I went to GT.You?")
['I went to GT', 'You?']
```

Но даже объединения этих двух регулярных выражений недостаточно для правильной сегментации изрядного количества сложных тестовых примеров из модуля `nlpia.data`:

```
>>> from nlpia.data.loaders import get_data
>>> regex = re.compile(r'((?<!\d)\.|\.(?!\d))|([!.?]+)[ $]+')
>>> examples = get_data('sentences-tm-town')
>>> wrong = []
>>> for i, (challenge, text, sents) in enumerate(examples):
...     if tuple(regex.split(text)) != tuple(sents):
...         print('wrong {}: {}{}'.format(i, text[:50], '...' if len(text) > 50
else ''))
...         wrong += [i]
>>> len(wrong), len(examples)
(61, 61)
```

Для повышения точности сегментатора на основе регулярных выражений придется добавить еще очень много «возвратов назад» или «заглядываний вперед». Для сегментации предложений лучше воспользоваться алгоритмом машинного обучения (обычно однослойной нейронной сетью или логистической регрессией), обученным на маркированном наборе предложений. Подобные модели, подходящие для усовершенствования нашего сегментатора предложений, включены в несколько пакетов:

- DetectorMorse[1];
- spaCy[2];
- SyntaxNet[3];
- NLTK (Punkt)[4];
- Stanford CoreNLP[5].

---

1 См. веб-страницу GitHub — cslu-nlp/DetectorMorse: Fast supervised sentence boundary detection using the averaged perceptron: https://github.com/cslu-nlp/detectormorse.

2 См. веб-страницу Facts & Figures — spaCy Usage Documentation: https://spacy.io/usage/facts-figures.

3 См. веб-страницу models/syntaxnet-tutorial.md at master (https://github.com/tensorflow/models/blob/master/research/syntaxnet/g3doc/syntaxnet-tutorial.md).

4 См. веб-страницу nltk.tokenize package — NLTK 3.3 documentation: http://www.nltk.org/api/nltk.tokenize.html#module-nltk.tokenize.punkt.

5 См. веб-страницу torotoki / corenlp-python — Bitbucket (https://bitbucket.org/torotoki/corenlp-python).

Для большинства критически важных приложений есть смысл использовать сегментатор предложений spaCy (встроенный в синтаксический анализатор). У spaCy не очень много зависимостей и он вполне сравним с прочими по точности и быстродействию. Созданный Кайлом Горманом (Kyle Gorman) DetectorMorse — еще один неплохой вариант, если нужна первоклассная производительность при реализации на чистом Python, эффективность которой можно улучшить с помощью своего тренировочного набора данных.

11.5. На практике

Выделение информации и системы формирования ответов на вопросы применяются для:

- учебных помощников в университетах;
- обслуживания клиентов;
- технической поддержки;
- продажи;
- документации программного обеспечения и ЧаВо;

Выделять можно такую информацию, как:

- даты;
- моменты времени;
- цены;
- количественные величины;
- адреса;
- имена/названия:
  - людей;
  - мест;
  - приложений;
  - компаний;
  - ботов;
- взаимосвязи:
  - *is-a* (разновидности вещей);
  - *has* (атрибуты вещей);
  - *related-to*.

Производится ли разбор информации из большого корпуса или вводимой пользователем на ходу, возможность выделения конкретных нюансов и сохранение их для использования в дальнейшем играют важнейшую роль в эффективной работе чат-бота. Вы научились нормализовать информацию программным образом, с помощью ее выявления и отделения, а затем — разметки взаимосвязей между

элементами этой информации. Благодаря размещению знаний в удобной для поиска структуре данных у чат-бота появляются инструменты для отстаивания своей точки зрения при разговоре в рамках заданной предметной области.

Резюме

- Для хранения взаимосвязей между сущностями можно создать граф знаний.
- Регулярные выражения — миниатюрный язык программирования, позволяющий выделять информацию.
- Частеречная разметка помогает выделять взаимосвязи между упомянутыми в предложении сущностями.
- Сегментация предложений — задача, для которой недостаточно просто разбить предложения по точкам и восклицательным знакам.

12 Начинаем общаться: диалоговые системы

В этой главе

- Четыре подхода к созданию чат-ботов.
- Что такое язык разметки для систем ИИ.
- В чем различия между конвейерами чат-ботов и другими конвейерами NLP.
- Гибридная архитектура чат-бота, объединяющая воедино лучшие идеи.
- Как заставить чат-бот умнеть с течением времени с помощью машинного обучения.
- Предоставление чат-боту свободы воли — даем ему возможность спонтанно высказывать свои мысли.

Мы начали эту книгу с идеи диалоговой системы (конвейера NLP чат-бота), поскольку считаем, что в настоящее время это одно из важнейших приложений NLP. Впервые в истории мы можем говорить с машиной на нашем собственном языке, причем не всегда ее можно отличить от человека. Машины научились имитировать людей — куда более сложная задача, чем может показаться. Существует несколько

конкурсов с внушительными денежными призами, в которых вы можете поучаствовать, если чувствуете, что ваш чат-бот умеет это делать:

- премия Alexa (The Alexa Prize, $3,5 миллиона)[1];
- премия Лебнера (Winograd Schema Challenge, $7 тысяч[2])[3];
- схема Винограда (The Winograd Schema Challenge, $27 тысяч[4])[5];
- тест Маркуса (The Marcus Test)[6];
- тест Лавлейс (The Lovelace Test)[7].

Помимо развлекательной функции и волшебства создания машины, готовой поддерживать разговор, помимо славы, ожидающей того, кто создаст бот, способный победить человека в IQ-тесте[8], помимо теплого чувства спасения мира от злобных хакерских сетей ботов и уймы денег, которые получит победивший Google и Amazon в их играх виртуальных помощников, — изученные в этой главе методы помогут вам в работе.

Двадцать первое столетие пройдет под знаком упрощающего нашу жизнь ИИ. Наиболее естественный интерфейс для ИИ — разговор на естественном языке. Например, чат-бот Chloe компании Aira.io помогает описывать окружающий мир для слепых и слабовидящих людей. Другие компании создают чат-боты — юристов, экономящих пользователям на штрафах за парковку тысячи долларов (или фунтов стерлингов) и сокращающих время, проведенное в суде. У беспилотных автомобилей, вероятно, вскоре появятся речевые интерфейсы, аналогичные Google Assistant и Google Maps, помогающие добраться до нужного места.

12.1. Языковые навыки

Наконец-то у нас есть все необходимое для создания чат-бота — проще говоря, *диалоговой системы* (dialog system), или *диалогового механизма/движка* (dialog engine). Мы создадим конвейер NLP, способный участвовать в разговоре на естественном языке.

---

1 Премия Alexa, https://developer.amazon.com/alexaprize.

2 Авторы ошибаются, главный приз премии Лебнера составляет $100 тысяч. — *Примеч. пер.*

3 Премия Лебнера в Блетчли-парк, https://aisb.org.uk/category/loebner-prize/.

4 Авторы ошибаются, главный приз составлял $25 тысяч, впрочем, более он не присуждается. — *Примеч. пер.*

5 *Bender D.* Establishing a Human Baseline for the Winograd Schema Challenge (http://ceur-ws.org/Vol-1353/paper_30.pdf); *Kurzweil.* An alternative to the Turing test (http://www.kurzweilai.net/an-alternative-to-the-turing-test-winograd-schema-challenge-annual-competition-announced).

6 What Comes After the Turing Test // New Yorker, Jan 2014 (http://www.newyorker.com/tech/elements/what-comes-after-the-turing-test).

7 *Reidl.* The Lovelace 2.0 Test of Artificial Creativity and Intelligence (https://arxiv.org/pdf/1410.6142.pdf).

8 Авторы несколько смешивают понятия, под IQ-тестом все же обычно понимается нечто совершенно другое. — *Примеч. пер.*

Вот некоторые из возможностей NLP, которые мы будем использовать:

- токенизация, стемминг и лемматизация;
- языковые модели в векторном пространстве, например векторы мультимножеств слов или векторы тем (семантические векторы);
- углубленные представления языка, например векторы слов или векторы идей LSTM;
- перевод на основе преобразования последовательностей в последовательности (см. главу 10);
- сопоставление с паттернами (см. главу 11);
- шаблоны для генерации текста на естественном языке.

С помощью этих инструментов можно создать чат-бот с весьма интересным поведением.

Теперь убедимся, что мы понимаем под чат-ботом одно и то же. В некоторых сообществах чат-ботами в несколько презрительном ключе называются системы, возвращающие заранее заготовленные ответы (canned response systems) (https://en.wikipedia.org/wiki/Canned_response). Эти чат-боты ищут паттерны во входящем тексте и возвращают при обнаружении совпадения фиксированный (шаблонизированный) ответ[1]. Их можно считать ЧаВо-ботами, знающими ответы только на общие вопросы. Подобные простейшие диалоговые системы полезны главным образом в автоматизированных системах поддержки пользователей с интерактивными голосовыми меню, где существует возможность перевести разговор на человека, когда у чат-бота закончатся шаблонные ответы.

Но это не значит, что чат-бот должен ограничиваться этим. Если проявить особую изобретательность в этих паттернах и шаблонах, то чат-бот вполне может работать врачом, проводя психотерапевтические сеансы или психологические консультации. Еще в 1964 году Йозеф Вайценбаум (Joseph Weizenbaum) использовал паттерны и шаблоны для создания первого широко известного чат-бота ELIZA (https://ru.wikipedia.org/wiki/Элиза_(программа)). Удивительно эффективный терапевтический бот Facebook Messenger, Woebot, существенно использует подход сопоставления с паттерном и шаблонизированного ответа. Все, что требуется для создания проходящих тест Тьюринга чат-ботов, — добавить немного управления состоянием (контекстом) в нашу систему сопоставления с паттерном.

Чат-бот Mitsuku Стива Уорсвика (Steve Worswick) получил премию Лебнера (https://ru.wikipedia.org/wiki/Премия_Лебнера), пройдя одну из разновидностей теста Тьюринга в 2016–2019 годах с помощью шаблонов и сопоставления с паттерном. Для расширения возможностей Mitsuku он добавил в него поддержку контекста (сохранение состояния). Прочитать об остальных победителях можно в «Википедии» (https://ru.wikipedia.org/wiki/Премия_Лебнера#Победители). Amazon недавно добавил дополнительный слой глубины разговора (контекст) в Alexa и назвал его Follow-Up Mode («Режим дальнейшего отслеживания»)[2]. В этой главе мы расскажем вам, как

---

[1] *Woudenberg van A. F.* A Chatbot Dialogue Manager. Open University of the Netherlands (http://dspace.ou.nl/bitstream/1820/5390/1/INF_20140617_Woudenberg.pdf).

[2] См. статью с сайта The Verge Amazon Follow-Up Mode по адресу https://www.theverge.com/2018/3/9/17101330/amazon-alexa-follow-up-mode-back-to-back-requests.

добавить контекст в ваши собственные чат-боты, использующие сопоставление с паттернами.

12.1.1. Современные подходы

Чат-боты намного продвинулись со времен ELIZA. Технологии сопоставления с паттернами обобщались и улучшались на протяжении десятилетий. Кроме того, разработаны совершенно новые технологии в дополнение к сопоставлению с паттернами. В свежей литературе чат-боты часто называются диалоговыми системами, возможно, вследствие усложнения их устройства. Сопоставления с паттернами в тексте и наполнение шаблонов ответов выделенной с помощью этих паттернов информацией — лишь один из четырех современных подходов к созданию чат-ботов:

- *сопоставление с паттернами* — сопоставление с паттерном и шаблоны ответов (заранее заготовленные ответы);
- *«заземление»* (grounding) — логические графы знаний и выполнение вывода на их основе;
- *поиск* — извлечение текста;
- *порождающие методы* — статистика и машинное обучение.

Примерно в таком порядке данные подходы и разрабатывались. В этой последовательности мы и будем знакомить вас с ними. Но прежде, чем продемонстрировать, как с помощью каждого из этих методов генерировать ответы, мы покажем, как они используются чат-ботами на практике.

В наиболее продвинутых чат-ботах используется гибридный подход — сочетание всех этих методов. Благодаря гибридному подходу они могут решать широкий спектр задач. Вот список лишь некоторых из приложений чат-ботов; обратите внимание, что наиболее продвинутые чат-боты (Siri, Alexa и Allo) упомянуты рядом с несколькими задачами и приложениями:

- формирование ответов на вопросы — Google Search, Alexa, Siri, Watson;
- виртуальная помощь — Google Assistant, Alexa, Siri, помощник («скрепка») MS Office;
- речевые — Google Assistant, Google Smart Reply, Mitsuki Bot;
- маркетинг — боты Twitter, боты в блогах, боты Facebook, Google Search, Google Assistant, Alexa, Allo;
- обслуживание потребителей — боты интернет-магазинов, боты технической поддержки;
- боты управления сообществами — Bonusly, Slackbot;
- терапевтические боты — Woebot, Wysa, YourDost, Siri, Allo.

Как, по вашему мнению, нужно сочетать четыре названных базовых типа диалоговых систем для создания ботов для каждого из этих семи приложений? На рис. 12.1 показано, как это делают некоторые из ботов.

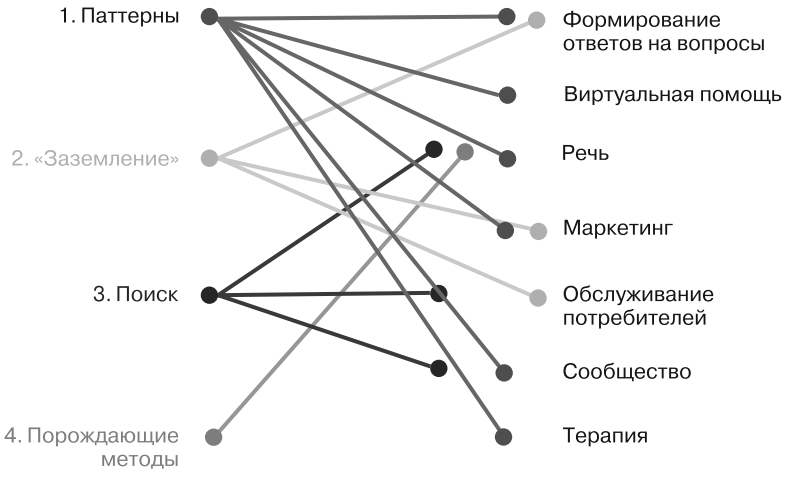


Рис. 12.1. Используемые в чат-ботах методы для некоторых приложений

Диалоговые системы формирования ответов на вопросы

Чат-боты, формирующие ответы на вопросы, используются для ответов на фактологические вопросы об окружающем мире, в том числе, возможно, о самом боте. Многие из систем формирования ответов на вопросы сначала просматривают базу знаний или реляционную базу данных для «заземления» (привязки к реальному миру). При отсутствии там приемлемого ответа они могут производить поиск ответов на заданные вопросы в корпусе неструктурированных данных (или даже по всему Интернету). По сути, это и делает Google Search. Синтаксический разбор высказывания для выделения вопроса, на который нужно ответить, и последующий выбор правильного ответа требуют сложного конвейера, сочетающего большинство описанных в предыдущих главах элементов. Ответы на вопросы чат-ботами сложнее всего реализовать, поскольку они требуют согласования работы большого числа различных элементов.

Виртуальные помощники

Виртуальные помощники, такие как Alexa и Google Assistant, удобны, когда речь идет о достижении конкретной цели. Цели (намерения) обычно представляют собой простые понятия вроде запуска приложения, установки напоминаний, проигрывания какой-либо музыки или включения света в доме. Поэтому виртуальных помощников называют целеориентированными диалоговыми системами. Разговор с подобными чат-ботами обычно длится недолго, и в результате пользователь должен быть удовлетворен выполнением конкретного действия или извлечением конкретной информации.

Вероятно, вы знакомы с виртуальными помощниками в своем смартфоне или в системе «умный дом». Но, наверное, вы не знаете, что виртуальные помощники могут помочь и с юридическими и налоговыми вопросами. Хотя «мастера» TurboTax компании Intuit не слишком «разговорчивы», в них реализовано весьма сложное

интерактивное голосовое меню. Взаимодействие с ними происходит не голосовым путем и не в чате, а путем заполнения форм со структурированными данными. Поэтому «мастер» TurboTax пока нельзя назвать чат-ботом в полном смысле этого слова, но он наверняка вскоре будет обернут в чат-интерфейс, если конкурирующий с ним AskMyUncleSam добьется успеха[1].

Чат-боты — юридические помощники успешно обжаловали миллионы долларов штрафов за парковку в Нью-Йорке и Лондоне[2]. Существует даже юридическая контора, в которой все общение с юристами сводится к разговору с чат-ботом[3]. Юридические чат-боты, безусловно, являются целеориентированными виртуальными помощниками, но они не просто назначают дату визита к юристу, а подбирают дату судебного заседания и, возможно, даже помогают выиграть дело.

Компания Aira.io (http://aira.io) создала виртуального помощника Chloe. Он предоставляет слепым и слабовидящим доступ к «визуальному переводчику для слепых». При знакомстве с ним новых сотрудников Chloe может задавать такие вопросы, как «Вы слепой?», «Есть ли у вас собака-поводырь?», «Страдаете ли вы аллергией на какие-либо продукты питания и есть ли у вас пищевые предпочтения, о которых нам следует знать?». Такая архитектура называется *голосовой* (voice first design), при этом приложение проектируется с нуля вокруг диалоговой системы. В будущем возможности Chloe вырастут, по мере обучения ее пониманию окружающего мира с помощью видеотрансляций в режиме реального времени. Взаимодействующие с Chloe «исследователи» со всего мира будут обучать ее понимать выполняемые людьми каждодневные действия. Chloe — один из немногих виртуальных помощников, чье предназначение — исключительно помогать, а не влиять или манипулировать[4].

Такие виртуальные помощники, как Siri, Google Assistant, Cortana и Chloe, умнеют с каждым днем. Виртуальные помощники обучаются при взаимодействии с людьми и другими машинами, с которыми они связаны. Они вырабатывают не зависящий от предметной области интеллект общего уровня. Если вас интересуют вопросы искусственного интеллекта общего уровня (AGI), рекомендуем поэкспериментировать с виртуальными помощниками и речевыми чат-ботами.

---

1 Январь 2017 года, сообщение AskMyUncleSam на сайте Venture Beat: https://venturebeat.com/2017/01/27/how-this-chatbot-powered-by-machine-learning-can-help-with-your-taxes/.

2 Июнь 2016 года, Chatbot Lawyer Overturns 160,000 Parking Tickets in London and New York. The Guardian: https://www.theguardian.com/technology/2016/jun/28/chatbot-ai-lawyer-donotpay-parking-tickets-london-new-york.

3 Ноябрь 2017, Chatbot-based “firm without lawyers” launched, сообщение в блоге Legal Futures: https://www.legalfutures.co.uk/latest-news/chatbot-based-firm-without-lawyers-launched.

4 Мы редко осознаем, какое влияние оказывают виртуальные помощники и поисковые системы на нашу свободу выбора и убеждения. И нас редко волнует тот факт, что их стимулы и побудительные мотивы отличаются от наших. Подобные смещенные побудительные мотивы встречаются не только в сфере технологии, например в виртуальных помощниках, но и в самой нашей культуре. Если вам интересно, в какую сторону направляют нас культура и технология, почитайте «Sapiens. Краткая история человечества» (Sapiens Deus by Yuval) и «Homo Deus. Краткая история завтрашнего дня» (Homo Deus by Yuval) Юваля Ноя Харари (Yuval Noah Harari).

Речевые чат-боты

Речевые чат-боты, например Mitsuku[1] Уорсвика или любой из Pandorabots[2], предназначены для развлечения. Их реализация зачастую требует лишь нескольких строк кода при наличии больших объемов тренировочных данных. Но поддержание разговора — задача с постоянно растущими требованиями. Точность (эффективность) речевого чат-бота обычно оценивается чем-то вроде теста Тьюринга. При типичном тесте Тьюринга люди взаимодействуют с неким собеседником через терминал и пытаются выяснить, бот это или человек. Чем сложнее отличить бот от человека, тем выше оценка этого бота по метрике теста Тьюринга.

Предметная область (разнообразие знаний) и типы человеческого поведения, которые должен реализовывать чат-бот в этих тестах Тьюринга, расширяются с каждым годом. И чем лучше чат-боты нас обманывают, тем лучше мы, люди, учимся обнаруживать их уловки. Во времена BBS[3] (1980-е годы) ELIZA обвела вокруг пальца немало людей, воспринимавших его психоаналитиком, помогающим справиться с повседневными проблемами. Понадобились десятилетия исследований и множество разработок, прежде чем чат-боты смогли обмануть нас снова.

> Обманешь меня один раз — позор ботам; обманешь второй — позор людям.
>
> *Неизвестный автор*

Недавно бот Mitsuku выиграл премию Лебнера, соревнование, в котором для ранжирования чат-ботов используется тест Тьюринга (https://ru.wikipedia.org/wiki/Премия_Лебнера). Речевые чат-боты применяются в основном для научных исследований, развлечения (в компьютерных играх) и рекламы.

Маркетинговые чат-боты

Задача маркетингового чат-бота — информирование пользователей о товаре и соблазнение на его покупку. Все большему количеству компьютерных игр, фильмов и сериалов сопутствуют рекламирующие их чат-боты на сайтах[4]:

- HBO продвигала сериал «Мир Дикого Запада» с помощью чат-бота Aeden[5];
- Sony продвигала «Обитель зла» с помощью чат-бота Red Queen[6];

---

1 См. веб-страницу Mitsuku Chatbot по адресу http://www.square-bear.co.uk/aiml.

2 См. веб-страницу Pandorabots AIML Chatbot Directory по адресу https://www.chatbots.org/.

3 Электронные доски объявлений (Bulletin Board Systems): https://ru.wikipedia.org/wiki/BBS. — *Примеч. пер.*

4 Джастин Клегг (Justin Clegg) перечисляет и другие в своем сообщении в LinkedIn: https://www.linkedin.com/pulse/how-smart-brands-using-chatbots-justin-clegg/.

5 Сентябрь 2016 года, Entertainment Weekly: https://www.yahoo.com/entertainment/westworld-launches-sex-touting-online-181918383.html.

6 Январь 2017 года, IPG Media Lab: https://www.ipglab.com/2017/01/18/sony-pictures-launches-ai-powered-chatbot-to-promote-resident-evil-movie/.

- Disney продвигала мультфильм «Зверополис» с помощью чат-бота Officer Judy Hopps[1];
- Universal продвигала фильм «Убрать из друзей» с помощью чат-бота Laura Barnes;
- Activision продвигала игру Call of Duty с помощью чат-бота Lt. Reyes.

Некоторые виртуальные помощники, по сути, являются замаскированными маркетинговыми чат-ботами. Рассмотрим, например, Amazon Alexa и Google Assistant. Хотя они утверждают, что помогают людям, например, с установкой напоминаний и поиском в Интернете, но неизменно выдают результаты, относящиеся к товарам или фирмам, перед общей или бесплатной информацией. Эти компании занимаются продажей — непосредственно в случае Amazon или опосредованно в случае Google. Их виртуальные помощники прежде всего помогают родительским компаниям (Amazon и Google) зарабатывать деньги. Конечно, они также помогают пользователям решить свои проблемы, поэтому мы продолжаем пользоваться ими. Но целевые функции, которым подчиняются эти боты, ориентированы на совершение пользователем покупки, а не на его счастье или благополучие.

Большинство маркетинговых чат-ботов — речевые, чтобы развлекать пользователя и скрывать свои истинные мотивы. Они, как правило, умеют формировать ответы на вопросы, «заземленные» на базу знаний о продаваемых ими товарах. Для имитации персонажей фильмов, телепрограмм или компьютерных игр чат-боты осуществляют поиск по тексту подходящих для использования в диалоге фрагментов сценария. Иногда порождающие модели даже обучаются непосредственно на наборе сценариев. В маркетинговых ботах часто задействованы все четыре метода, о которых мы расскажем в этой главе.

Управление сообществами

Управление сообществами — особенно важное применение чат-ботов, поскольку влияет на развитие общества. Хороший чат-бот способен вывести сообщество, посвященное какой-либо компьютерной игре, из хаоса и помочь ему развиться из логова хамов и троллей в открытое и учитывающее индивидуальные потребности участников сообщество, в котором всем интересно. Плохой чат-бот, например печально известный Twitter-бот Tay, может быстро создать среду предубеждений и невежества[2].

Когда чат-боты «сходят с ума», некоторые люди утверждают, что они лишь отражают (возможно, с эффектом увеличительного стекла) пороки общества. И таковы зачастую ненамеренные последствия столкновения любой сложной системы

---

[1] Июнь 2016 года, Venture Beat: https://venturebeat.com/2016/06/01/imperson-launches-zootopias-officer-judy-hopps-bot-on-facebook-messenger/.

[2] Статья в «Википедии» о короткой «жизни» чат-бота Tay компании Microsoft по адресу https://en.wikipedia.org/wiki/Tay_(bot).

с реальным миром. Но поскольку чат-боты являются активными действующими «лицами», которых пропитывают мотивациями такие же разработчики, как вы, не стоит считать их просто «отражением общества». Чат-боты, похоже, не просто отражают и увеличивают лучшее и худшее в нас. Это действующая сила, на которую частично влияют их разработчики и тренировочные данные, в хорошую или плохую сторону. Поскольку отвечающие за их работу люди не могут полностью навязать им стратегию «не сотвори зла», то именно разработчик должен стараться создавать доброжелательные, отзывчивые, ориентированные на благо общества чат-боты. «Трех законов роботехники» Азимова недостаточно[1]. Только вы можете повлиять на эволюцию чат-ботов с помощью интеллектуального программного обеспечения и ловко подобранных наборов данных.

Несколько умных людей из Университета Аризоны направили свое умение создавать чат-боты на спасение человечества, но не от злобного сверхчеловеческого ИИ, а от нас самих. Исследователи пытаются имитировать поведение потенциальных новобранцев-террористов ИГИЛ, чтобы сбить с толку и дезориентировать игиловских вербовщиков. В один прекрасный день это позволит чат-ботам спасать людские жизни, просто разговаривая с людьми, которые несут в этот мир зло[2]. Чат-боты — тролли иногда оказываются не такими уж плохими, если троллят нужных людей или организации.

Обслуживание потребителей

Единственным «человеком», доступным при вашем визите в интернет-магазин, зачастую оказывается чат-бот, осуществляющий обслуживание потребителей. Watson от IBM, Lex от Amazon и другие сервисы чат-ботов используются «за кулисами» в качестве движущей силы этих помощников потребителей. Они сочетают способности формировать ответы на вопросы (помните обучение Watson игре в Jeopardy?) со способностями виртуального помощника. Но, в отличие от маркетинговых ботов, обслуживающие потребителей боты должны быть хорошо «заземлены». Причем важна поддержка актуальности базы знаний, используемой для привязки ответов к реальности, чтобы боты могли отвечать на вопросы о заказах и товарах, равно как и производить размещение и отмену заказов.

В 2016 году Facebook Messenger выпустил API для коммерческих компаний, позволяющий создавать чат-боты для обслуживания потребителей. Компания Google недавно купила API.ai и создала фреймворк Dialogflow для создания обслуживающих потребителей чат-ботов. Аналогично для создания диалоговых систем обслуживания потребителей (для розничных и оптовых торговцев продаваемыми на

---

[1] Март 2014 года, Джордж Дворски (George Dvorski): Why Asimov's Three Laws of Robotics Can't Protect Us. Gizmodo, https://io9.gizmodo.com/why-asimovs-three-laws-of-robotics-cant-protect-us-1553665410.

[2] Октябрь 2015 года, Slate: http://www.slate.com/articles/technology/future_tense/2015/10/using_chatbots_to_distract_isis_recruiters_on_social_media.html.

Amazon товарами) часто используется Amazon Lex. Чат-боты быстро превращаются в важный канал продаж в различных отраслях промышленности — от мира моды (бот Botty от Hilfiger) и фастфуда (TacoBot) до торговли цветами[1].

Психотерапия

Современные терапевтические чат-боты, такие как Wysa и YourDOST, созданы, чтобы помочь компьютерщикам, которых сократили с работы, приспособиться к новой жизни[2]. Терапевтические чат-боты должны уметь развлекать не хуже, чем речевой чат-бот. Быть информативными не меньше, чем боты, формирующие ответы на вопросы. И убедительными, как маркетинговые чат-боты. Если же в дополнение к их альтруизму они преследуют какую-либо личную выгоду, то могут быть целенаправленными и использовать свое влияние и маркетинговые навыки, чтобы убедить клиента вернуться для повторного сеанса терапии.

Наверное, вы не рассматриваете Siri, Alexa и Allo как психотерапевтов, но они вполне могут помочь вам выдержать непростой день. Спросите их про смысл жизни — наверняка получите философский или забавный ответ. А если вы грустите, попросите их рассказать вам шутку или включить какую-нибудь оптимистичную музыку. Несмотря на эти дешевые трюки, можно не сомневаться, что разработчикам сложных чат-ботов помогали психологи, чтобы пользователи ощущали себя счастливее и благополучнее.

Как и следовало ожидать, терапевтические боты используют гибридный подход, сочетающий все четыре базовых подхода, перечисленных в начале данной главы.

12.1.2. Гибридный подход

Так что же представляет собой гибридный подход?

Четыре основных подхода к созданию чат-ботов можно комбинировать различными способами, получая в результате удобные чат-боты. Множество различных приложений используют все четыре базовых метода. Гибридные чат-боты различаются главным образом тем, как именно они сочетают эти подходы и насколько сильный упор делается на каждый из них.

В этой главе вы узнаете, как подобрать соотношение таких подходов для создания соответствующего вашим потребностям чат-бота. Гибридный подход позволяет встроить в чат-бот возможности всех этих реальных систем. И мы сформируем такую «целевую функцию», которая будет учитывать цели чат-бота при выборе между четырьмя подходами или просто при выборе из всех возможных ответов, сгенерированных с помощью каждого из подходов.

Итак, рассмотрим эти подходы по порядку. Для каждого из них мы создадим чат-бот, использующий только один метод, а в конце данной главы покажем, как объединить их.

---

[1] 1-800-flowers: 1-800-Flowers.com, Tommy Hilfiger: https://techcrunch.com/2016/09/09/botty-hilfiger/, TacoBot: http://www.businessinsider.com/taco-bells-tacobot-orders-food-for-you-2016-4.

[2] Декабрь 2017 года, Bloomberg: https://www.bloomberg.com/news/articles/2017-12-10/fired-indian-tech-workers-turn-to-chatbots-for-counseling.

12.2. Подход сопоставления с паттернами

В самых первых чат-ботах реакция на сообщение пользователя происходила на основе сопоставления с паттерном. Помимо обнаружения высказываний, на которые мог бы ответить чат-бот, паттерны также можно использовать для выделения информации из входящего текста. Мы обсуждали несколько способов описания паттернов для выделения информации в главе 11.

Выделенной из высказываний пользователей информацией можно заполнить базу знаний о пользователях или об окружающем мире вообще, а можно задействовать ее для непосредственного формирования ответов на некоторые высказывания. В главе 1 показан простой чат-бот на основе сопоставления с паттерном, использующий регулярные выражения для обнаружения приветствий. Регулярные выражения можно также применять для выделения имени того, кого приветствует пользователь-человек. Это дает боту контекст для разговора, на основе которого можно сформировать ответ.

Созданная в конце 1970-х программа ELIZA[1] справлялась с этой задачей на удивление хорошо, убеждая множество пользователей, что она может помочь им с психологическими проблемами. ELIZA запрограммирована на поиск в высказываниях пользователей ограниченного набора слов. Алгоритм ранжировал встреченные слова, выделяя одно, предположительно наиболее важное, слово в высказывании пользователя. После чего выбирался соответствующий этому слову стандартный шаблон ответа. Шаблоны ответов специально разработаны так, чтобы имитировать сочувствие и непредвзятость психоаналитика на основе методов рефлексивной психологии. Ключевое слово, инициировавшее ответ, обычно употреблялось в самом ответе для создания впечатления, что ELIZA понимает, о чем говорит пользователь. Отвечая «на языке самого пользователя», бот устанавливал с ними взаимопонимание и помогал им поверить, что их действительно слушают.

Бот ELIZA открыл глаза на очень многие вещи, связанные с взаимодействием с людьми на естественном языке. Вероятно, главным откровением было то, что ключевым для успешной работы чат-бота было просто внимание к словам пользователя или по крайней мере имитация такого внимания.

В 1995 году Ричард Уоллес (Richard Wallace) занялся созданием универсального фреймворка для построения чат-ботов с помощью подхода сопоставления с паттерном. С 1995 по 2002 год его сообщество разработчиков создало язык разметки для систем ИИ (Artificial Intelligence Markup Language, AIML), предназначенный для описания паттернов и ответов чат-ботов. Эталонной реализацией с открытым исходным кодом чат-бота, использующего этот язык разметки, стал A.L.I.C.E. AIML с тех пор де-факто стал стандартом для описания конфигураций API чат-ботов и виртуальных помощников для таких сервисов, как Pandorabots. Фреймворк Bot компании Microsoft также умеет загружать AIML-файлы для описания поведения чат-ботов. Другие фреймворки, например, DialogFlow от Google и Amazon Lex, не поддерживают импорт/экспорт AIML.

---

[1] Авторы противоречат самим себе, как уже упоминалось выше, бот ELIZA был создан Йозефом Вайценбаумом в середине 1960-х. — *Примеч. пер.*

AIML — открытый стандарт в том смысле, что язык документирован и не содержит скрытых проприетарных возможностей, доступных для использования только одной компанией. Существуют пакеты Python с открытым исходным кодом для синтаксического разбора и выполнения AIML для чат-ботов[1]. Но AIML ограничивает число доступных для описания паттернов и логических структур. И, по сути, это XML, а значит, основанные на Python фреймворки чат-ботов (например, `Will` и `ChatterBot`) лучше подойдут для создания вашего чат-бота.

А поскольку существует множество утилит NLP в пакетах Python, то можно создать более сложные чат-боты на основе сопоставления с паттерном, просто наращивая постепенно логику бота непосредственно на Python с помощью регулярных выражений и паттернов поиска[2]. Для описания наших паттернов Aira мы разработали аналогичный AIML простой язык паттернов поиска. У нас был транслятор для преобразования из этого языка паттернов поиска в регулярные выражения, которые можно применять на любой платформе, где есть синтаксический анализатор регулярных выражений.

Aira использует `{{handlebars}}` для спецификаций шаблонов в этом фреймворке ботов `aichat` (http://github.com/aira/aichat). Причем для языка шаблонизации Handlebars существуют интерпретаторы как для Java, так и для Python, так что Aira применяет его на множестве мобильных и серверных платформ. Причем выражения Handlebars могут включать фильтры и условные выражения, пригодные для создания сложного поведения чат-бота, а если вам нужно нечто еще более простое и «пайтонообразное» для шаблонов чат-бота, можете воспользоваться f-строками языка Python 3.6. Если вы пока еще не перешли на Python 3.6 или более поздние версии, можете использовать `str.format(template, **locals())` для получения аналогичной f-строкам функциональности.

12.2.1. Сопоставляющий с паттернами чат-бот на основе AIML

Описать наш чат-бот для приветствий из главы 1 на AIML (v2.0) можно следующим образом[3] (листинг 12.1).

Листинг 12.1. Файл nlpia\data\greeting2.aiml

```
<?xml version="1.0" encoding="UTF-8"?><aiml version="2.0">
<category>
```

[1] pip install aiml https://github.com/creatorrr/pyAIML.

[2] Паттерны поиска и паттерны поиска со звездочкой в качестве джокерного символа представляют собой упрощенные регулярные выражения вроде тех, что применяются для поиска файла в DOS, Bash или практически любой другой командной оболочке. В шаблонах поиска джокерный символ звездочки означает произвольное число любых символов. Так, *.txt соответствует любым именам файлов с .txt в конце (https://ru.wikipedia.org/wiki/Шаблон_поиска).

[3] AI Chat Bot in Python with AIML от NanoDano, август 2015 года, https://www.devdungeon.com/content/ai-chat-bot-python-aiml#what-is-aiml.

```
    <pattern>HI</pattern>
<template>Hi!</template>
</category>
<category>
    <pattern>[HELLO HI YO YOH YO'] [ROSA ROSE CHATTY CHATBOT BOT CHATTERBOT]
     </pattern>
    <template>Hi , How are you?</template>
</category>
<category>
    <pattern>[HELLO HI YO YOH YO' 'SUP SUP OK HEY] [HAL YOU U]</pattern>
    <template>Good one.</template>
</category>
</aiml>
```

Мы сделали XML более компактным и удобочитаемым благодаря новым возможностям AIML 2.0 (с платформы Bot Libre). С помощью квадратных скобок можно описывать в одной строке альтернативные написания слов.

К сожалению, интерпретаторы Python для AIML (`PyAiml`, `aiml` и `aiml_bot`) не поддерживают версию 2.0 спецификаций AIML. `aiml_bot` — интерпретатор AIML для Python 3, работающий с исходной спецификацией AIML 1.0. В `aiml_bot` синтаксический анализатор встроен в класс `Bot()`, который держит ядро в оперативной памяти, что обеспечивает быструю реакцию бота. Ядро (kernel) содержит все паттерны и шаблоны AIML в единой структуре данных, аналогичной словарю языка Python, в которой описываются соответствия паттернов шаблонам ответов.

AIML 1.0

AIML — декларативный язык, основанный на стандарте XML, что ограничивает допустимые для использования в боте языковые конструкции и структуры данных. Но не думайте, что чат-бот AIML — завершенная система. Мы дополним его изученными ранее инструментами.

Одно из ограничений AIML — виды паттернов для сопоставления и ответа. Ядро AIML (механизм сопоставления с паттерном) реагирует только на соответствие входного текста паттерну, вручную прописанному разработчиком. Удобно, что паттерны AIML могут включать джокерные символы (символы, соответствующие любой последовательности слов). Но включаемым в паттерн словам входной текст должен соответствовать в точности. Никакого нечеткого поиска, смайликов, знаков препинания внутри, опечаток или написанных с ошибками слов не допускается, автоматическое их сопоставление не производится. В AIML приходится вручную описывать все синонимы с помощью `</srai>` по одному. Представьте себе, насколько изнуряющей была бы реализация на AIML всего того стемминга и лемматизации, которые мы производили программным образом в главе 2. Хотя мы показали в примере выше, как реализовать синонимы и соответствие слов с опечатками в AIML, гибридный чат-бот, который мы создадим к концу этой главы, будет уклоняться от этого утомительного труда с помощью обработки всего поступающего в чат-бот текста.

Следует учитывать и еще одно принципиальное ограничение паттернов AIML: не более одного джокерного символа. У более выразительных языков сопоставления с паттернами, например, регулярных выражений, есть больше возможностей для

создания интересных чат-ботов[1]. Пока что, с помощью AIML, мы использовали только паттерны вроде HELLO ROSA * для поиска входного текста вроде *Hello Rosa, you wonderful chatbot!*.

ПРИМЕЧАНИЕ

Удобочитаемость языка очень важна для производительности разработчика. Хороший язык может существенно помочь в разработке, вне зависимости от того, создаете ли вы чат-бот или веб-приложение.

Мы не станем тратить много времени, чтобы научить вас понимать и писать AIML. Но было бы неплохо, если бы вы умели импортировать и настраивать под свои задачи некоторые из доступных (и бесплатных) сценариев AIML с открытым исходным кодом[2]. Сценарии AIML можно использовать для создания определенной простой функциональности чат-бота, предварительно немного доработав их.

Далее мы покажем, как создать и загрузить AIML-файл в чат-бот и сгенерировать с его помощью ответы.

Интерпретатор AIML на Python

Пройдем пошагово непростой сценарий AIML из листинга 12.1 и покажем, как загрузить и запустить его из программы на языке Python. Листинг 12.2 представляет собой простой AIML-файл, способный распознавать две последовательности слов: *Hello Rosa* и *Hello Troll*. На каждую из них чат-бот будет реагировать по-разному, как и в предыдущих главах.

Листинг 12.2. Файл nlpia/nlpia/data/greeting_step1.aiml

```
<?xml version="1.0" encoding="UTF-8"?><aiml version="1.0.1">

<category>
    <pattern>HELLO ROSA </pattern>
    <template>Hi Human!</template>
</category>
<category>
    <pattern>HELLO TROLL </pattern>
    <template>Good one, human.</template>
</category>

</aiml>
```

---

[1] Соревноваться по выразительности с современными языками вроде Python непросто (https://en.wikipedia.org/wiki/Comparison_of_programming_languages#Expressiveness и http://redmonk.com/dberkholz/2013/).

[2] Поищите в Google AIML 1.0 files («Файлы AIML 1.0») или AIML brain dumps («Дампы ядра AIML») и посетите такие AIML-ресурсы, как Chatterbots и Pandorabots: http://www.chatterbotcollection.com/category_contents.php?id_cat=20.

ПРИМЕЧАНИЕ

В AIML 1.0 все паттерны необходимо задавать в верхнем регистре.

Мы настроили бот так, чтобы он по-разному реагировал на два различных приветствия: вежливое и невежливое. Теперь воспользуемся пакетом `aiml_bot` для интерпретации файлов AIML 1.0 на языке Python. Если вы установили пакет `nlpia`, то можете загрузить эти примеры из него с помощью кода из листинга 12.3. Чтобы поэкспериментировать с собственноручно созданными AIML-файлами, вам нужно изменить путь `learn=path` так, чтобы он указывал на ваш файл.

Листинг 12.3. Файл ch12_patterns_aiml.py

```
>>> import os
>>> from nlpia.constants import DATA_PATH
>>> import aiml_bot

>>> bot = aiml_bot.Bot(
...     learn=os.path.join(DATA_PATH, 'greeting_step1.aiml'))
Loading /Users/hobs/src/nlpia/nlpia/data/greeting_step1.aiml...
done (0.00 seconds)
>>> bot.respond("Hello Rosa,")
'Hi there!'
>>> bot.respond("hello !!!troll!!!")
'Good one, human.'
```

Выглядит неплохо. Данная спецификация AIML ловко игнорирует знаки препинания и регистр символов при поиске соответствий паттернам.

Но спецификация AIML 1.0 нормализует паттерны только относительно знаков препинания и пробелов между словами, а не внутри слов. Синонимы, ошибки написания, слова, которые пишутся через дефис, и составные слова выходят за рамки ее возможностей (листинг 12.4).

Листинг 12.4. Файл ch12_patterns_aiml.py

```
>>> bot.respond("Helo Rosa")
WARNING: No match found for input: Helo Rosa
''
>>> bot.respond("Hello Ro-sa")
WARNING: No match found for input: Hello Ro-sa
''
```

Большинство подобных проблем можно решить с помощью тега `<srai>` и символа звездочки (`*`) в шаблоне, чтобы связать несколько паттернов с одним шаблоном ответа. Можете считать их синонимами слова *Hello*, хотя это могут быть варианты неправильного написания или вообще совершенно другие слова (листинг 12.5).

Листинг 12.5. nlpia/data/greeting_step2.aiml

```
<category><pattern>HELO *        </pattern><template><srai>HELLO <star/>
    </srai></template></category>
<category><pattern>HI *          </pattern><template><srai>HELLO <star/>
    </srai></template></category>
<category><pattern>HIYA *        </pattern><template><srai>HELLO <star/>
    </srai></template></category>
<category><pattern>HYA *         </pattern><template><srai>HELLO <star/>
    </srai></template></category>
<category><pattern>HY *          </pattern><template><srai>HELLO <star/>
    </srai></template></category>
<category><pattern>HEY *         </pattern><template><srai>HELLO <star/>
    </srai></template></category>
<category><pattern>WHATS UP *    </pattern><template><srai>HELLO <star/>
    </srai></template></category>
<category><pattern>WHAT IS UP *  </pattern><template><srai>HELLO <star/>
    </srai></template></category>
```

ПРИМЕЧАНИЕ

При написании своих собственных файлов AIML не забудьте включить теги <aiml> в начале и в конце. В примере кода AIML мы опустили их для краткости.

После загрузки дополнительного AIML наш бот сможет распознавать несколько различных вариантов приветствия, в том числе и с орфографическими ошибками, как показано в листинге 12.6.

Листинг 12.6. Файл ch12_patterns_aiml.py

```
>>> bot.learn(os.path.join(DATA_PATH, 'greeting_step2.aiml'))
>>> bot.respond("Hey Rosa")
'Hi there!'
>>> bot.respond("Hi Rosa")
'Hi there!'
>>> bot.respond("Helo Rosa")
'Hi there!'
>>> bot.respond("hello **troll** !!!")
'Good one, human.'
```

В AIML 2.0 можно задавать альтернативные, выбираемые случайным образом шаблоны ответов с помощью списков в квадратных скобках. В AIML 1.0 для этого используется тег <li>. Тег <li> работает только внутри тегов <condition> или <random>. Мы воспользуемся тегом <random>, чтобы наш бот более творчески реагировал на приветствия (листинг 12.7).

Листинг 12.7. nlpia/data/greeting_step3.aiml

```
<category><pattern>HELLO ROSA </pattern><template>
    <random>
        <li>Hi Human!</li>
        <li>Hello friend</li>
```

```
        <li>Hi pal</li>
        <li>Hi!</li>
        <li>Hello!</li>
        <li>Hello to you too!</li>
        <li>Greetings Earthling ;)</li>
        <li>Hey you :)</li>
        <li>Hey you!</li>
    </random></template>
</category>
<category><pattern>HELLO TROLL </pattern><template>
    <random>
        <li>Good one, Human.</li>
        <li>Good one.</li>
        <li>Nice one, Human.</li>
        <li>Nice one.</li>
        <li>Clever.</li>
        <li>:)</li>
    </random></template>
</category>
```

Теперь ответы нашего чат-бота не звучат так механически (по крайней мере, в начале разговора) (листинг 12.8).

Листинг 12.8. Файл ch12_patterns_aiml.py

```
>>> bot.learn(os.path.join(DATA_PATH, 'greeting_step3.aiml'))
>>> bot.respond("Hey Rosa")
'Hello friend'
>>> bot.respond("Hey Rosa")
'Hey you :)'
>>> bot.respond("Hey Rosa")
'Hi Human!'
```

ПРИМЕЧАНИЕ

Вероятно, при выполнении этого кода вы не получите тех же ответов в том же порядке, что и мы. В этом-то и смысл тега <random>. При каждом совпадении с паттерном он выбирает ответ из списка случайным образом. Способа задать начальное значение для генератора случайных ответов в боте aiml_bot нет, хотя это было бы удобно при тестировании (кто-нибудь, отправьте запрос на внесение изменений).

Описывать синонимы для альтернативных написаний *Hi* и *Rosa* можно в отдельных тегах `<category>`. Можно описывать различные группы синонимов для шаблонов и отдельные списки ответов, в зависимости от типа приветствия. Например, можно описать паттерны для таких приветствий, как *SUP*[1] и *WUSSUP BRO*[2], и затем отвечать на том же диалекте или с той же степенью фамильярности и неформальности.

[1] То же, что What's up? («Как жизнь?»). — *Примеч. пер.*

[2] То же, что What's up, brother? («Как жизнь, старина?»). — *Примеч. пер.*

В AIML даже есть теги для захвата строк в поименованные переменные (аналогично поименованным группам в регулярных выражениях). Состояния в AIML называются `topics`. И в AIML есть способы описания условных выражений на основе любой из описанных в AIML-файле переменных. Если AIML вас заинтересовал — попробуйте их. Это отличное упражнение, помогающее лучше понять, как работают грамматики и чат-боты на основе сопоставления с паттернами. Но мы пойдем дальше и обратимся к более выразительным языкам, таким как регулярные выражения и Python, для создания нашего чат-бота. Благодаря этому мы сможем воспользоваться и другими инструментами, о которых говорили в предыдущих главах, например стеммерами и лемматизаторами, для обработки синонимов и неправильных написаний слов (см. главу 2). Если в вашем чат-боте используется AIML и производится предварительная обработка в виде лемматизации и стемминга, то вам, видимо, нужно модифицировать шаблоны AIML для захвата этих основ слов и лемм.

Если вам кажется, что AIML сложен для своих целей, — вы не одиноки. Amazon Lex использует упрощенную версию AIML, которую можно импортировать и экспортировать из JSON-файла. Стартап `API.ai` разработал настолько интуитивно понятные спецификации разговора, что Google их выкупила, интегрировала в Google Cloud Services и переименовала в Dialogflow. Спецификации Dialogflow тоже можно экспортировать в JSON и импортировать из JSON, но эти файлы несовместимы с форматами AIML или Amazon Lex.

Если вы думаете, что имеет смысл объединить все эти несовместимые спецификации в единую открытую спецификацию вроде AIML, у вас есть шанс внести свой вклад в проект `aichat` и разработку языка AIRS (спецификация ИИ-ответов, AI Response Specification). Aira и фонд Do More поддерживают AIRS, стремясь упростить обмен контентом (диалоги для интерактивной беллетристики, обучающих курсов, виртуальных экскурсий и т. д.) друг с другом. Приложение `aichat` — эталонная реализация интерпретатора AIRS на Python с графическим веб-интерфейсом.

А вот как выглядит типичная спецификация AIRS. В ней описаны четыре элемента информации, содержащиеся в отдельной строке «плоской» таблицы, которые необходимы боту для реакции на команду пользователя. Эту таблицу можно экспортировать/импортировать из/в CSV/JSON/простой список списков Python:

```
>>> airas_spec = [
...     ["Hi {name}","Hi {username} how are you?","ROOT","GREETING"],
...     ["What is your name?",
...     "Hi {username} how are you?","ROOT","GREETING"],
...     ]
```

Первый столбец спецификации AIRS описывает паттерн и любые параметры, которые нужно извлечь из фрагмента пользовательской речи или текстового сообщения. Второй столбец описывает желаемый ответ чат-бота, обычно в форме шаблона, который можно заполнить с помощью переменных из контекста данных чат-бота. Кроме того, он может содержать специальные ключевые слова, инициирующие другие действия бота, помимо текстового/речевого ответа.

Последние два столбца используются для хранения состояния (контекста) чат-бота. Когда чат-бот реагирует, вследствие обнаружения совпадения с паттерном он может переходить в новое состояние, если в рамках этого состояния требуется иное поведение, скажем, чтобы задать дополнительные вопросы или получить новую информацию. Так что два столбца в конце строки просто указывают чат-боту, какое состояние он должен отслеживать для этих паттернов и в какое должен перейти после заданного в шаблоне ответа или действия. Названия исходных и целевых состояний формируют граф, такой как на рис. 12.2, определяющий поведение чат-бота.

Google Dialogflow и Amazon Lex представляют собой лучше масштабируемые версии используемого в `aichat` подхода к созданию чат-ботов с помощью спецификаций по сопоставлению с паттернами. Но для многих сценариев они представляются чрезмерно сложными. Цель проекта с открытым исходным кодом `aichat` (http://github.com/totalgood/aichat) — дать разработчикам более интуитивный способ проектирования, визуализации и тестирования чат-ботов. Посмотрите на `aichat` или гибридный чат-бот из `nlpia` (http://github.com/totalgood/nlpia), если хотите узнать больше о подходе к созданию чат-ботов на основе сопоставления с паттернами. Если же вам нужно реализовать полнофункциональный чат-бот на основе этого подхода, у фреймворков Dialogflow от Google (бывший `app.ai`) и Lex от Amazon есть обширная документация с примерами, которые можно положить в основу своего приложения. Внутри обеих систем есть возможность развертывания бесплатного чат-бота, в результате чего, впрочем, разработчик окажется намертво привязан к конкретному подходу, так что лучше помогите нам в разработке `aichat`.

12.2.2. Сетевое представление сопоставления с паттерном

По мере создания Aira чат-бота для помощи незрячим мы разрабатывали определенные утилиты визуализации для анализа и проектирования такого опыта взаимодействия с пользователем. Сетевое представление связей между состояниями и создающими эти связи паттернами открывает пути создания новых паттернов и состояний. Сетевое представление позволяет мысленно «прокрутить» диалог, подобно мысленному выполнению нескольких строк кода на языке Python. И сетевое представление обеспечивает удобную навигацию по лабиринту дерева диалога (фактически по сети или графу), а значит, позволяет избежать тупиков и циклов в диалоге.

Если задуматься, становится понятно, что паттерны и ответы чат-бота на основе сопоставления с паттернами описывают сеть (граф), узлы (вершины) которой соответствуют состояниям. Ребра же сети отражают «спусковые крючки» сопоставления с паттернами, в результате которых бот что-то «говорит», прежде чем перейти в новое состояние (на новый узел). Диаграмма переходов из состояния в состояние для нескольких паттернов и ответов AIRS может выглядеть так, как показано на рис. 12.2.

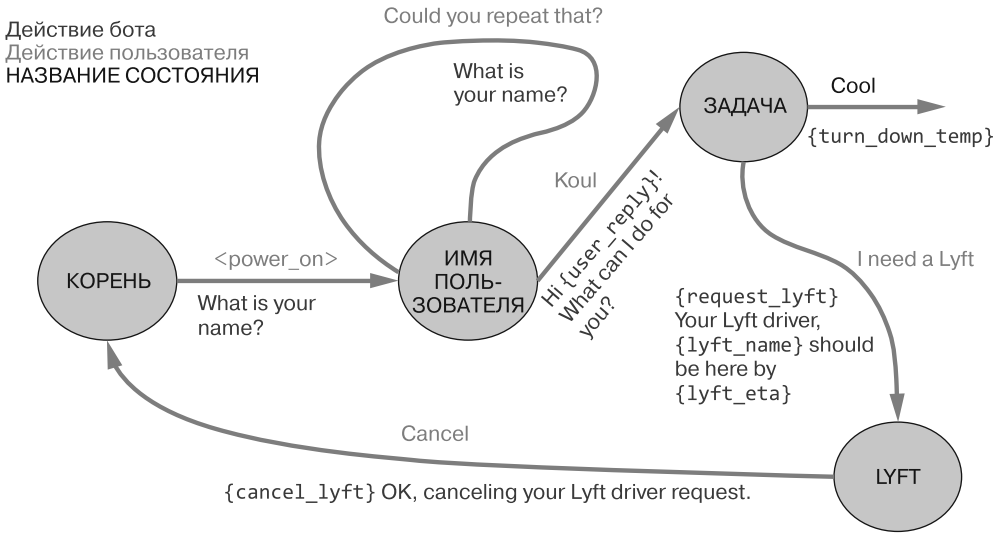


Рис. 12.2. Управление состоянием (контекстом)

С помощью такого графического представления можно обнаружить в диалоге тупики и циклы, требующие исправления путем уточнения старых или добавления новых паттернов в спецификацию диалога. Aira работает над средствами визуализации для преобразования спецификаций AIRS в подобные диаграммы графов (см. рис. 12.2) в проекте `aichat`. Если вы работаете с JavaScript или D3, им не помешает ваша помощь.

А теперь займемся другим подходом к созданию чат-ботов — «заземлением».

12.3. Заземление

В основе A.L.I.C.E., как и других AIML чат-ботов, лежит сопоставление с паттернами. Сопоставление с паттернами и шаблоны использовал и первый широко известный чат-бот ELIZA задолго до AIML. Но создатели этих чат-ботов жестко «зашивают» в паттернах и шаблонах логику ответов. Такой подход плохо масштабируется, не в смысле производительности вычислений, а в смысле трудоемкости для разработчика. Сложность подобного чат-бота растет линейно относительно вложенных в его создание усилий. Фактически по мере роста сложности чат-бота усилия разработчика окупаются все меньше, поскольку объем взаимодействия между всеми «движущими частями» растет и поведение чат-бота становится все хуже прогнозируемым и менее доступным для отладки.

Программирование с ориентацией на работу с данными — современный подход к наиболее сложным задачам программирования наших дней. Как использовать данные для программирования чат-бота? В предыдущей главе мы рассказали, как формировать структурированные знания на основе текста на естественном языке (неструктурированных данных) с помощью выделения информации. Создать сеть связей или фактов

можно, просто читая текст, например «Википедию» или даже свой личный блог. Из этого раздела вы узнаете, как включить знания об окружающем мире (или своей личной жизни) в арсенал своего чат-бота. Такую сеть логических связей между сущностями — граф или базу знаний — можно положить в основу ответов чат-бота.

С помощью подобного графа знаний можно делать вывод, чтобы ответить на вопросы о мире, содержащиеся в этой базе знаний, а затем на основе логических ответов можно заполнить значения содержащихся в шаблонных ответах переменных для создания ответов на естественном языке. Системы формирования ответов на вопросы (например, выигрывавший в шоу Jeopardy бот Watson от компании IBM) именно так изначально и были устроены, хотя в более свежих версиях наверняка применяются технологии информационного поиска. Говорят, что граф знаний «заземляет» чат-бот к реальному миру.

Возможности подхода на основе баз знаний не ограничиваются формированием ответов на вопросы об окружающем мире. Базу знаний можно также заполнять в режиме реального времени фактами из текущего диалога. Благодаря этому бот будет в курсе, кто его собеседник.

Следующий шаг в моделировании знаний — создание субграфов знаний, соответствующих представлениям собеседников бота об окружающем мире. Если вы знакомы с архитектурой баз данных, можете считать это неким аналогом внешних баз данных — в данном случае баз знаний. Можно хранить временный «кэш» самых свежих знаний, или постоянный обновляемый журнал всех полученных (и не полученных) чат-ботом знаний о других участниках разговора. Все высказывания участников разговора можно использовать для заполнения «моделей сознания», баз знаний их представлений об окружающем мире. Для этого, возможно, достаточно создать паттерны для извлечения используемых участниками разговора для обращения друг к другу сетевых псевдонимов, подобно тому как мы делали в главе 1.

Если задуматься, становится понятно, что люди при общении не просто выдают заготовленные стандартные ответы, как только что созданный нами чат-бот AIML. Благодаря человеческому мозгу мы обдумываем сказанное собеседником и пытаемся сделать какой-либо логический вывод из того, что мы помним о реальном мире и друг о друге. Для состоящего из одного высказывания ответа может понадобиться сделать несколько выводов и предположений. Так что добавление в чат-бот логики и «заземления» делает его более человекоподобным, или по крайней мере более логичным.

Создание чат-ботов на основе «заземления» отлично подходит для формирующих ответы на вопросы чат-ботов, когда необходимые для ответа на вопрос знания содержатся в некоей обширной базе знаний, которую можно получить из открытой базы данных. Вот некоторые примеры баз знаний, подходящих для «заземления» чат-бота:

- Wikidata (включая Freebase) (https://www.wikidata.org/wiki/Wikidata:Introduction/ru);
- Open Mind Common Sense (ConceptNet) (https://github.com/commonsense/conceptnet5./wiki/API);
- Cyc (https://ru.wikipedia.org/wiki/Cyc);
- YAGO (https://en.wikipedia.org/wiki/YAGO_(database));
- DBpedia (https://ru.wikipedia.org/wiki/DBpedia).

Нам нужен только способ выполнения запросов к базе знаний для извлечения из нее фактов, необходимых для наполнения ответа на высказывание пользователя. Если же пользователь задает вопрос о факте, который может оказаться в нашей базе знаний, можно перевести этот вопрос на естественном языке (например, «Кто вы?» или «Как называется 50-й штат США?») в запрос базы знаний для непосредственного извлечения искомого ответа. Именно это поиск Google и делает с помощью Freebase и других баз знаний, объединенных для формирования их графа знаний.

Мы можем воспользоваться навыками сопоставления с паттерном слов из главы 11 для выделения важнейших составных частей вопроса из высказывания пользователя, например поименованных сущностей или информации о требуемых для ответа на вопрос взаимосвязях. Для классификации вопросов можно проверять их на наличие ключевых вопросительных слов (*who*, *what*, *when*, *where*, *why* и *is*) в начале предложения. Благодаря этому чат-бот сможет определить вид знаний (тип узла или поименованной сущности), которые нужно извлечь из графа знаний.

Quepy (http://quepy.readthedocs.io/en/latest/) — компилятор запросов на естественном языке, способный генерировать запросы к базам знаний и базам данных с помощью этих методов. SQL-эквивалент графа знаний из RDF-троек называется SPARQL (https://www.w3.org/TR/rdf-sparql-query/).

12.4. Информационный поиск

Еще один ориентированный на данные способ «слушать» пользователя — поиск предыдущих высказываний в журналах предыдущих разговоров, аналогично человеку-слушателю, который пытается вспомнить, слышал ли он уже вопрос/высказывание/слово. Бот может искать не только в журналах своих собственных разговоров, но и в любых расшифровках разговоров людей между собой, разговоров людей с ботами или даже разговоров между ботами. Но, как обычно, какие входные данные — таков и результат. Поэтому необходимо тщательно очистить и организовать базу данных предыдущих разговоров, чтобы бот искал (а затем и имитировал) качественный разговор. Люди должны наслаждаться разговором с вашим ботом.

В базе данных диалогов бота, работающего на основе поиска, должны содержаться только полезные или приятные разговоры. Причем это должны быть разговоры на тему, общение на которую ожидается от бота. Несколько примеров хороших источников диалогов для бота, работающего на основе поиска: диалоги из сценариев фильмов, журналы обслуживания потребителей на IRC-каналах (те разговоры, после которых пользователи остались довольны обслуживанием) и личные сообщения между людьми (те, которыми люди не против поделиться). Не делайте этого со своими собственными журналами сообщений электронной почты или СМС без получения письменного разрешения всех участников разговора.

Если решите включить диалоги бота в свой корпус — будьте осторожны. Желательно включать в базу данных только те высказывания, удовлетворение которыми

выразил хотя бы один человек, по крайней мере в форме продолжения разговора. А разговоры ботов между собой вообще желательно не использовать, разве что речь идет о *действительно* интеллектуальном боте.

Бот, работающий на основе поиска, может обращаться к журналу прошлых разговоров для поиска примеров высказываний, аналогичных только что сказанному его собеседником. Для упрощения такого поиска следует организовать корпус диалогов в виде пар «высказывание — ответ». Если на ответ была получена какая-либо реакция, то его следует включить в базу данных дважды: один раз как ответ, а второй раз — как высказывание, на которое поступил ответ. На основе столбца ответов из базы данных можно формировать ответы бота на высказывания из столбца высказываний.

12.4.1. Проблема контекста

Простейший подход — повторно использовать ответ дословно, без каких-либо изменений. Зачастую это удачный подход, если высказывание семантически (по смыслу) соответствует тому, на которое нужно ответить боту. Но, даже если база данных содержит все возможные высказывания пользователей, бот будет отражать личные особенности произносивших эти высказывания людей. Если набор ответов согласован и получен от разнообразных людей, это хорошо. Но может представлять проблему, если высказывание, на которое бот пытается ответить, зависит от общего контекста данного разговора или обстоятельств окружающего мира, которые могли измениться за прошедшее с момента формирования корпуса диалогов время.

Например, представьте, что кто-то задал нашему боту вопрос «Который час?». Бот не должен при этом использовать ответ, полученный от человека на наиболее подходящее высказывание из нашей базы данных. Этот ответ подойдет только в случае, если время, когда задавался вопрос, совпадает с временем, когда было записано соответствующее высказывание из базы. Помимо текста высказывания на естественном языке, должна фиксироваться и сравниваться и подобная информация о времени — контекст (состояние). Это играет особенно важную роль, если семантика высказывания указывает на активное изменение зафиксированного в контексте (базе знаний чат-бота) состояния.

Еще два примера того, как знания об окружающем мире (контекст) должны влиять на ответы чат-бота: вопросы «Кто ты?» и «Откуда ты родом?». Контекстом в данном случае являются личность и биографические данные того, к кому обращен вопрос. К счастью, такой контекст можно легко сгенерировать и хранить в базе знаний или базе данных, содержащей профиль и предысторию нашего бота. Анкетные данные чат-бота необходимо подобрать так, чтобы они приблизительно отражали средние или медианные данные людей, высказывания которых включены в базу данных диалогов.

Информацию о профиле бота можно использовать для выбора между равными по прочим показателям высказываниями из базы данных. Чтобы сделать бот совсем

передовым, можно искусственно завышать рейтинг найденных в результате поиска ответов от людей, чей профиль напоминает профиль нашего бота. Допустим, что у нас есть информация о поле людей, высказывания и ответы которых зафиксированы в нашей базе данных диалогов. При этом можно сопоставлять «пол» чат-бота в качестве еще одного слова (измерения, поля базы данных) с полом респондентов из базы данных. Если пол респондента совпадает с «полом» чат-бота, а слова высказывания (семантический вектор) близки к соответствующему вектору из высказывания нашего пользователя, их следует считать отлично подходящими для наших целей и размещать вверху списка результатов поиска. Оптимальный способ такого сопоставления с паттерном — вычисление оценочной функции при каждом извлечении ответа, с учетом информации из профиля.

Или же можно решить проблему контекста с помощью создания неявного профиля бота и сохранения его в базе знаний вручную. Главное, использовать только соответствующие этому профилю ответы из базы данных чат-бота.

Независимо от того, как используется этот профиль, для создания непротиворечивой личности нашего чат-бота придется рассматривать вопросы о его личности как особые случаи. Если база данных высказываний и ответов не содержит достаточно ответов на вопросы вроде «Кто вы?», «Откуда вы родом?» и «Какой ваш любимый цвет?», то информационный поиск не подходит и придется использовать одну из других методик создания чат-ботов. При отсутствии большого количества пар «высказывание — ответ» относительно профиля придется выявлять все вопросы о боте и использовать базу знаний с целью вывода подходящего ответа для соответствующего элемента высказывания. Или же можно применить подход на основе грамматик для заполнения шаблонизированного ответа с помощью извлеченной из структурированного набора данных информации о профиле чат-бота.

Для учета состояния (контекста) в чат-боте, в основе которого лежит информационный поиск, можно проделать нечто аналогичное тому, что мы сделали для чат-бота, использующего сопоставление с паттернами. Если задуматься, то перечисление списка высказываний пользователей — просто другой способ описания паттерна. На самом деле именно такой подход и применяют Amazon Lex и Google Dialogflow. Вместо описания жесткого паттерна для захвата команды пользователя можно просто предоставить диалоговому движку несколько примеров. Подобно тому как каждому паттерну в чат-боте на основе сопоставления с паттернами соответствовало состояние, здесь нам тоже нужно просто связать пары примеров «высказывание — ответ» с поименованным состоянием.

Это может оказаться непростой задачей, если источник этих пар примеров «высказывание — ответ» — неструктурированный и нефильтрованный, например корпус диалогов Ubuntu или Reddit. Но при работе с такими диалоговыми тренировочными наборами данных, как Reddit, зачастую можно найти какие-либо мелкие фрагменты огромного набора данных, которые можно автоматически маркировать в соответствии с их каналом и нитью ответов в дискуссии. Для кластеризации исходного комментария, предшествовавшего конкретной нити или дискуссии, можно использовать инструменты семантического поиска и сопоставления с паттернами. И эти кластеры далее могут служить нашими состояниями. Впрочем, обнаружение переходов от одних темы или состояния к другим может быть сложным процессом.

И получаемые подобным образом состояния далеко не так точны, как сгенерированные вручную.

Подобный подход к управлению состоянием (контекстом) — весьма приемлемый вариант, если бот должен лишь быть интересным и поддерживать разговор. Но для предсказуемого и надежного поведения чат-бота лучше ограничиться подходом на основе сопоставления с паттернами или вручную описывать переходы из состояния в состояние.

12.4.2. Пример чат-бота на основе информационного поиска

Мы будем руководствоваться ODSC 2017 по созданию чат-бота на основе информационного поиска. Если вы хотите посмотреть видео или исходный блокнот IPython для этого руководства, найти их можно в репозитории GitHub https://github.com/totalgood/prosocial-chatbot.

Наш чат-бот будет использовать корпус диалогов Ubuntu — набор высказываний и ответов, записанных на канале IRC Ubuntu, где люди помогают друг другу решать технические проблемы. Он включает более семи миллионов речевых фрагментов и более миллиона сеансов диалога, каждый с множеством реплик и речевых фрагментов[1]. Благодаря столь большому числу пар «высказывание — ответ» исследователи часто применяют этот набор данных для проверки точности работы чат-ботов, основанных на информационном поиске.

Именно такие пары «высказывание — ответ» подходят для обучения чат-ботов, основанных на информационном поиске. Но не волнуйтесь, мы не собираемся использовать все семь миллионов речевых фрагментов. Возьмем лишь около 150 тысяч реплик и проверим, достаточно ли этого, чтобы наш чат-бот мог отвечать на распространенные вопросы Ubuntu. Для начала скачаем часть корпуса Ubuntu, как показано в листинге 12.9.

Листинг 12.9. ch12_retrieval.py

```
>>> from nlpia.data.loaders import get_data
>>> df = get_data('ubuntu_dialog')
Downloading ubuntu_dialog
requesting URL:
https://www.dropbox.com/s/krvi79fbsryytc2/ubuntu_dialog.csv.gz?dl=1
remote size: 296098788
Downloading to /Users/hobs/src/nlpia/nlpia/bigdata/ubuntu_dialog.csv.gz
39421it [00:39, 998.09it/s]
```

Возможно, при выполнении этого кода вы получили предупреждение о том, что путь `/bigdata/` не существует, если вы пока еще не использовали функцию `nlpia.data.loaders.get_data()` для большого набора данных. Но процедура скачивания создаст его при первом запуске.

[1] *Lowe et al.* The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems. — 2015 (https://arxiv.org/abs/1506.08909).

ПРИМЕЧАНИЕ

Приведенные здесь сценарии требуют 8 Гбайт свободной оперативной памяти. Если памяти на вашей машине окажется недостаточно, попробуйте уменьшить размер набора данных — вырезать часть строк из df. В следующей главе мы расскажем вам, как с помощью gensim обрабатывать данные по пакетам «во внешней памяти» в случае больших наборов данных.

В листинге 12.10 показано, как выглядит этот корпус.

Листинг 12.10. ch12_retrieval.py

```
>>> df.head(4)
                     Context                                    Utterance
0  i think we could import the old comments via r...    basically each
      xfree86 upload will NOT force u...
1  I'm not suggesting all - only the ones you mod...    oh? oops. __eou__
2  afternoon all __eou__ not entirely related to ...    we'll have a BOF
      about this __eou__ so you're ...
3  interesting __eou__ grub-install worked with /...    i fully endorse
      this suggestion </quimby> __eo...
```

Видите токены `__eou__`? Похоже, работа с этим набором данных — весьма непростая задача. Но благодаря ему вы сможете прочувствовать некоторые распространенные проблемы предварительной обработки данных в NLP. Эти токены отмечают конец речевого фрагмента (end of utterance), место, в котором «говорящий» нажал клавишу `RETURN` или `Send` в IRC-клиенте. Если вывести в консоль еще несколько примеров полей `Context`, можно заметить в них также маркеры `__eot__` (end of turn, конец реплики), отмечающие место, где кто-либо из собеседников завершил мысль и ждет ответа.

Но если посмотреть внутрь документа контекста (строки таблицы), можно увидеть несколько маркеров `__eot__` (реплик). С помощью этих маркеров более продвинутые чат-боты могут проверять, насколько хорошо они справляются с проблемой контекста, о которой речь шла в предыдущем разделе. Но мы проигнорируем дополнительные реплики в корпусе и сосредоточим внимание на последней, той, ответом на которую был речевой фрагмент. Во-первых, создадим функцию для разбиения по этим символам `__eot__` и уберем маркеры `__eou__`, как показано в листинге 12.11.

Листинг 12.11. ch12_retrieval.py

```
>>> import re
>>> def split_turns(s, splitter=re.compile('__eot__')):
...     for utterance in splitter.split(s):
...         utterance = utterance.replace('__eou__', '\n')
...         utterance = utterance.replace('__eot__', '').strip()
...         if len(utterance):
...             yield utterance
```

А теперь выполним функцию `split_turns` для нескольких строк из `DataFrame`, чтобы проверить, правильно ли она работает. Мы извлечем только последние репли-

ки как из контекста, так и из речевого фрагмента и посмотрим, достаточно ли этого для обучения основанного на информационном поиске чат-бота (листинг 12.12).

Листинг 12.12. ch12_retrieval.py

```
>>> for i, record in df.head(3).iterrows():
...     statement = list(split_turns(record.Context))[-1]
...     reply = list(split_turns(record.Utterance))[-1]
...     print('Statement: {}'.format(statement))
...     print()
...     print('Reply: {}'.format(reply))
```

При выполнении этого кода будет выведено что-то вроде:

```
Statement: I would prefer to avoid it at this stage. this is something that
     has gone into XSF svn, I assume?
Reply: each xfree86 upload will NOT force users to upgrade 100Mb of fonts
     for nothing
  no something i did in my spare time.

Statement: ok, it sounds like you're agreeing with me, then
 though rather than "the ones we modify", my idea is "the ones we need to
     merge"
Reply: oh? oops.

Statement: should g2 in ubuntu do the magic dont-focus-window tricks?
 join the gang, get an x-series thinkpad
 sj has hung on my box, again.
 what is monday mornings discussion actually about?
Reply: we'll have a BOF about this
 so you're coming tomorrow?
```

Превосходно! Похоже, здесь есть высказывания и ответы, включающие несколько высказываний (речевых фрагментов). Так что наш сценарий выполняет свою задачу, и его можно использовать для заполнения таблицы соответствий высказываний ответам, как показано в листинге 12.13.

Листинг 12.13. ch12_retrieval.py

```
>>> from tqdm import tqdm

>>> def preprocess_ubuntu_corpus(df):
...     """
...     Разбиваем все строки в df.Context и df.Utterance по маркерам
...     __eot__ (репликам)
...     """
...     statements = []
...     replies = []
...     for i, record in tqdm(df.iterrows()):
...         turns = list(split_turns(record.Context))
...         statement = turns[-1] if len(turns) else '\n'
...         statements.append(statement)
...         turns = list(split_turns(record.Utterance))
...         reply = turns[-1] if len(turns) else '\n'
```

Условный оператор if нужен здесь, поскольку часть высказываний и ответов содержит только пробелы

```
...         replies.append(reply)
...     df['statement'] = statements
...     df['reply'] = replies
...     return df
```

Нам нужно найти в столбце высказываний наиболее похожее на высказывание пользователя и ответить с помощью соответствующего ответа из столбца ответов. Помните, как мы искали подобные документы на естественном языке с помощью векторов частотностей слов и векторов TF-IDF в главе 3? Смотрите листинг 12.14.

Листинг 12.14. ch12_retrieval.py

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> df = preprocess_ubuntu_corpus(df)
>>> tfidf = TfidfVectorizer(min_df=8, max_df=.3, max_features=50000)
>>> tfidf.fit(df.statement)
```

Обратите внимание, что достаточно вычислить TF-IDF только высказываний (но не ответов), поскольку именно по ним мы производим поиск

Создадим объект `DataFrame` под названием `X`, в который поместим все векторы TF-IDF для 150 тысяч высказываний, как показано в листинге 12.15.

Листинг 12.15. ch12_retrieval.py

```
>>> X = tfidf.transform(df.statement)
>>> X = pd.DataFrame(X.todense(), columns=tfidf.get_feature_names())
```

Один из способов найти ближайшее высказывание — вычислить косинусное расстояние от высказывания из запроса до всех высказываний матрицы `X` (листинг 12.16).

Листинг 12.16. ch12_retrieval.py

```
>>> x = tfidf.transform(['This is an example statement that\
...     we want to retrieve the best reply for.'])
>>> cosine_similarities = x.dot(X.T)
>>> reply = df.loc[cosine_similarities.argmax()]
```

Выполнение этого кода заняло достаточно длительное время (более минуты на моем MacBook), а мы даже не вычисляли уровень доверия и не получали список возможных ответов для объединения с другими метриками.

12.4.3. Чат-бот на основе поиска

Что, если паттерны, соответствие которым мы хотим найти, в точности совпадают со сказанным людьми в предыдущих разговорах? Именно так и работает чат-бот на основе поиска. Он индексирует корпус диалогов так, чтобы с легкостью находить предыдущие высказывания, похожие на те, которым ему нужно ответить, а затем отвечает с помощью одного из соответствующих найденным в корпусе высказываниям ответов, которые он «запомнил» и проиндексировал для быстрого поиска.

Если вы хотите, не откладывая, поработать с основанным на поиске чат-ботом, то ChatterBot, созданный Гюнтером Коксом (Gunther Cox), отлично подойдет для того, чтобы набить руку. Его очень просто установить (выполните команду `pip install ChatterBot`), он поставляется с несколькими корпусами разговоров, на которых можно обучить чат-бот поддерживать простейший разговор. В ChatterBot есть корпусы для разговоров о спорте, философских рассуждений на тему ИИ или просто разговора ни о чем. ChatterBot можно обучить на любой последовательности разговоров (корпусе диалогов). Это не машинное обучение, а скорее индексация набора документов для поиска.

По умолчанию ChatterBot при обучении использует высказывания обоих участвующих в разговоре людей как основу для его собственных высказываний. Чтобы придать чат-боту более конкретную «личность», необходимо создать свой собственный корпус в формате `.yml` ChatterBot. Чтобы чат-бот имитировал только одну личность, корпус должен содержать разговоры, состоящие только из двух высказываний — одна исходная реплика и один ответ на нее; причем ответ — от той личности, которую нужно имитировать. Кстати, этот формат напоминает формат AIML, в котором есть паттерн (инициирующее разговор `statement` в ChatterBot) и шаблон (`response` в ChatterBot).

Конечно, у созданного подобным образом чат-бота на основе поиска есть довольно жесткие ограничения. Он не способен ни на какие новые высказывания. И чем больше данных, тем сложнее просмотреть все предыдущие высказывания. Так что чем интеллектуальнее и изощреннее такой бот, тем медленнее он будет работать. Подобная архитектура плохо масштабируется. Тем не менее мы продемонстрируем несколько продвинутых методов масштабирования любых основанных на поиске или индексации чат-ботов с помощью таких инструментов индексации, как хеширование с учетом локальности (`pip install lshash3`) и приближенный поиск соседей (`pip install annoy`).

По умолчанию ChatterBot использует в качестве базы данных SQLite, усугубляющую названные проблемы с масштабированием, если корпус содержит более 10 000 высказываний. Если попытаться обучить ChatterBot на основе SQLite на корпусе диалогов Ubuntu, придется ждать несколько дней... без преувеличения. На MacBook обработка всего 100 000 пар «высказывание — ответ» заняла у нас больше дня. Тем не менее код ChatterBot отлично подходит для разработки основного месторождения технических диалогов про Ubuntu. ChatterBot берет на себя все служебные действия, автоматически скачивая и распаковывая tar-архив, прежде чем проходить по древовидной файловой системе для поиска разговоров.

В листинге 12.17 показано, как «тренировочные» данные (фактически просто корпус диалогов) ChatterBot хранятся в реляционной базе данных.

Листинг 12.17. ch12_chatterbot.sql

```
#!sqlite3

.tables
# conversation              response                  tag
# conversation_association  statement                 tag_association
.width 5 25 10 5 40
```

```
.mode columns
.mode column
.headers on
SELECT * FROM response LIMIT 9;
# id     text               created_at  occur  statement_text
# -----  -----------------  ----------  -----  ------------------------------
--------
# 1      What is AI?        2017-11-26  2      Artificial Intelligence is the
branch of
# 2      What is AI?        2017-11-26  2      AI is the field of science which
concern
# 3      Are you sentient?  2017-11-26  2      Sort of.
# 4      Are you sentient?  2017-11-26  2      By the strictest dictionary
definition o
# 5      Are you sentient?  2017-11-26  2      Even though I'm a construct I do
have a
# 6      Are you sapient?   2017-11-26  2      In all probability, I am not.
I'm not t
# 7      Are you sapient?   2017-11-26  2      Do you think I am?
# 8      Are you sapient?   2017-11-26  2      How would you feel about me if I
told yo
# 9      Are you sapient?   2017-11-26  24     No.
```

Обратите внимание, что некоторым высказываниям соответствует несколько различных ответов, благодаря чему чат-бот может выбирать из допустимых ответов в соответствии с модальностью, контекстом или вообще случайным образом. ChatterBot выбирает ответы случайным образом, но можно создать и более совершенный чат-бот, позволив производить этот выбор на основе какой-либо целевой функции, функции потерь или эвристического алгоритма. Обратите также внимание, что даты `created_at` одинаковы. Это дата запуска «тренировочного» сценария ChatterBot, который скачал корпус диалогов и загрузил его в базу данных.

Основанные на поиске чат-боты можно усовершенствовать, свернув строки высказываний в векторы тем фиксированной размерности с помощью чего-то вроде Word2vec (путем суммирования всех векторов слов для коротких высказываний), или Doc2vec (см. главу 6), или LSA (см. главу 4). Помочь чат-боту производить обобщение примеров, на которых он обучался, может также понижение размерности. Благодаря этому он сможет адекватно реагировать в случаях, когда смысл высказывания в запросе (последнего высказывания собеседника бота) близок к смыслу одного из высказываний корпуса, хотя и выражен другими словами. Этот метод работает даже при сильно различающемся написании этих высказываний. По сути, такой семантический, основанный на поиске, чат-бот автоматизирует программирование шаблонов, которое мы выполняли на AIML ранее в этой главе. С помощью машинного обучения понижение размерности делает чат-бот более интеллектуальным (ориентированным на данные), чем это было бы при создании машинного интеллекта на основе жестко «зашитых» паттернов. Машинное обучение всегда предпочтительнее жестко «зашитых» паттернов, если времени (на программирование инициирующей ответы запутанной логики и паттернов) мало, а маркированных данных — много. Единственная необходимая для основанных на поиске чат-ботов «метка» — пример ответа для каждого из примеров высказываний в диалоге.

12.5. Порождающие модели

Мы обещали показать вам в этой главе порождающую модель. Но если вы вспомните модели преобразования последовательностей в последовательности, которые мы создавали в главе 10, то поймете, что это по сути порождающие чат-боты. Они представляют собой алгоритмы перевода на основе машинного обучения, которые переводят высказывания пользователей в ответы чат-бота. Поэтому мы не станем подробно описывать порождающие модели, лишь скажем, что существует еще множество других видов порождающих моделей.

Для создания творческого чат-бота, способного говорить то, что еще никто не говорил, полезны следующие порождающие модели:

- модели преобразования последовательностей в последовательности — модели, обученные генерировать ответы на основе входных последовательностей[1];
- ограниченные машины Больцмана (Restricted Boltzmann machine, RBM) — цепи Маркова, обучаемые минимизировать функцию «энергии»[2];
- порождающие состязательные сети (Generative Adversarial Network, GAN) — статистические модели, обучаемые обманывать эксперта, оценивающего качество разговора[3].

В главе 10 мы обсуждали сети на основе внимания (расширенные LSTM) и показывали, какие новые высказывания может спонтанно генерировать наш чат-бот. В следующем разделе мы разовьем этот подход в другом направлении.

12.5.1. Разговор в чате про nlpia

Наконец-то момент, которого вы так ждали... чат-бот, который может помочь в написании книги про NLP. Мы написали (а вы прочитали) достаточно для работы чат-бота. В этом разделе мы покажем, как с помощью переноса обучения создать порождающий конвейер NLP для генерации некоторых предложений. Возможно, вы уже справились с этим, даже не заметив.

Почему именно перенос обучения? Помимо небольшого начального текста на конкретную тему, которую должен понимать чат-бот, порождающим моделям требуется больший корпус более общих текстов для изучения модели языка. Чат-бот должен прочитать немало текста, прежде чем начнет распознавать все возможные сочетания слов для формирования грамматически правильных и осмысленных предложений. Так что корпус проекта «Гутенберг» — не лучший корпус для такой модели.

Задумайтесь, сколько книг нужно прочитать ребенку, прежде чем у него сформируется приличный словарный запас и понимание того, как правильно складывать

---

1 Пояснения к моделям преобразования последовательностей в последовательности и ссылки на несколько статей: https://suriyadeepan.github.io/2016-06-28-easy-seq2seq/.

2 Лекция Хинтона (Hinton) на Coursera: https://youtu.be/EZOpZzUKl48.

3 Руководство по GAN Иэна Гудфеллоу (Ian Goodfellow), NIPS 2016: https://arxiv.org/pdf/1701.00160.pdf — и приспособление к текстовым последовательностям Лантау Ю. (Lantau Yu): https://arxiv.org/pdf/1609.05473.pdf.

слова в предложения. И вероятно, пока вы все это читали, учителя многое вам подсказывали, например объясняя смысл[1]. Кроме того, люди более обучаемы, чем машины[2].

Обучение языковой модели, требующее вычислений с обработкой больших объемов данных — особенно сложная задача в случае моделей на основе отдельных символов. В языковой модели последовательностей символов чат-бот должен обучиться сочетать символы так, чтобы формировать правильно написанные и осмысленные слова, а не только складывать новые слова вместе в предложения. Поэтому желательно использовать повторно существующую языковую модель, обученную на большом массиве текста, написанного на том языке и в том стиле, который должен имитировать чат-бот. Если задуматься, станет понятно, почему ограничения данных не позволили нынешним исследователям NLP далеко забраться по кривой сложности от символов к словам и от слов к предложениям. Формирование абзацев, глав и целых романов — все еще сфера активных исследований. Остановимся на этом и покажем, как сгенерировать несколько предложений, подобно сгенерированным для раздела «Об этой книге».

Команда DeepMind предоставляет предобученные на более чем 500 Мбайт предложений из новостных лент CNN и Daily Mail языковые модели TensorFlow преобразования символьных последовательностей в символьные последовательности[3]. На случай, если кто-то захочет создать свою собственную языковую модель, они предоставили и все предложения в виде двух больших наборов данных в рамках задачи понимания прочитанного (Q&A)[4]. Мы воспользовались непосредственно предобученной моделью автоматического реферирования текста для генерации предложений для раздела «Об этой книге». Можете также использовать эти модели для дополнения вашего собственного конвейера машинного обучения с помощью подхода под названием «перенос обучения», подобно тому как мы делали с векторами слов в главе 6.

Рассмотрим алгоритм.

1. Скачать предобученную модель автоматического реферирования текста на основе преобразования последовательностей в последовательности (https://github.com/totalgood/pointer-generator#looking-for-pretrained-model).

---

[1] On the role of context in first- and second-language vocabulary learning: https://www.ideals.illinois.edu/bitstream/handle/2142/31277/TR-627.pdf.

[2] См. One-shot and Few-shot Learning of Word Embeddings: https://openreview.net/pdf?id=rkYgAJWCZ; One-shot learning by inverting a compositional causal process: http://www.cs.toronto.edu/~rsalakhu/papers/lake_nips2013.pdf.

[3] Предобученные модели автоматического реферирования текста TensorFlow: TextSum от Google Brain: https://github.com/totalgood/pointer-generator#looking-for-pretrained-model — и описывающая эту модель статья: https://arxiv.org/abs/1704.04368.

[4] Этот набор данных включает вопросы и ответы на понимание прочитанного, а также предложения из новостных статей, необходимых для ответов на эти вопросы: набор данных Q&A от DeepMind (https://cs.nyu.edu/%7Ekcho/DMQA/).

2. Выполнить синтаксический разбор и сегментацию текста в формате asciidoc для выделения предложений на естественном языке с помощью `nlpia.book_parser` (https://github.com/totalgood/nlpia/blob/master/src/nlpia/.py).
3. С помощью этой модели автоматического реферирования текста получить краткое изложение первых 30 или около того строк текста из каждого файла в формате asciidoc (обычно это одна глава): https://github.com/totalgood/nlpia/blob/master/src/nlpia/book/examples/ch12_chat_about_nlpia.py.
4. Отфильтровать сгенерированные предложения по новизне во избежание повтора уже существующих предложений из книги: https://github.com/totalgood/nlpia/blob/master/src/nlpia/book_parser.py.

Далее приведены единственные два синтаксически корректных и хотя бы немного оригинальных предложения, которые смог сгенерировать наш бот @ChattyAboutNLPIA. Вот как выглядит попытка @Chatty изложить вкратце первые 30 строк главы 5:

Convolutional neural nets make an attempt to capture that ordering relationship by capturing localized relationships[1].

А вот выданное @Chatty краткое резюме главы 8:

Language's true power is not necessarily in the words, but in the intent and emotion that formed that particular combination of words[2].

Приведенные предложения были в числе 25 выведенных (https://github.com/totalgood/nlpia/blob/master/src/nlpia/data/nlpia_summaries.md) этим доморощенным конвейером. В ближайшее время мы планируем модифицировать конвейер из nlpia.book.examples.ch12_chat_about_nlpia и попробовать получить более полезные результаты. Одно из возможных усовершенствований — обработка всей книги с помощью TextSum, чтобы у бота было больше материала для работы. Не помешает и дополнительная фильтрация.

1. Фильтрация сгенерированных предложений по синтаксической корректности[3].
2. Фильтрация сгенерированных предложений в соответствии с целевым стилем и тональностью.
3. Автоматическая детокенизация и при необходимости перевод букв в верхний регистр.

---

[1] Сверточные нейронные сети стремятся захватить эти отношения порядка за счет захвата локальных взаимосвязей. — *Примеч. пер.*

[2] Истинная мощь языка заключается не в самих словах, а в подтексте и чувствах, приведших к данному конкретному сочетанию слов. — *Примеч. пер.*

[3] Кайл Горман (Kyle Gorman) (@wellformedness, https://twitter.com/wellformedness), спасибо тебе более чем за 100 предложений и интересных фрагментов контента для этой книги. См. также https://en.wikipedia.org/wiki/Well-formedness.

12.5.2. Достоинства и недостатки каждого из подходов

Теперь, когда вы уже знаете о четырех основных подходах к созданию чат-ботов все, понимаете ли вы, как их сочетать, чтобы получить от бота максимальную отдачу? На рис. 12.3 приведены достоинства и недостатки каждого из подходов.

Подход	Достоинства	Недостатки
Грамматический	Легко начать работу. Простота переиспользования обучения. Модульность. Легко контролировать/ограничивать	Ограниченная предметная область. Возможности ограничиваются трудозатратами разработчика. Сложность отладки. Жесткие и «хрупкие» правила
«Заземление»	Хорошо отвечает на логические вопросы. Легко контролировать/ограничивать	Искусственное, механическое звучание. Проблемы с неоднозначностями. Проблемы с общечеловеческими знаниями. Ограничен структурированными данными. Требует выделения информации в больших масштабах. Требует надзора человека
Информационный поиск	Простота. Легкость обучения. Возможность имитации разговора людей	Плохо масштабируется. Имитируемая ботом личность непоследовательна. Не имеет представления о контексте. Не может отвечать на фактологические вопросы
Порождающий	Новые, творческие манеры разговора. Меньше трудозатрат для разработчика. Предметная область ограничивается только данными. Учет контекста	Сложно направлять в нужную сторону. Сложности в обучении. Требует больше данных (диалогов). Обучение требует больше вычислительной обработки

Рис. 12.3. Достоинства и недостатки четырех подходов к созданию чат-ботов

12.6. Подключаем привод на четыре колеса

Как мы и обещали в начале данной главы, сейчас мы покажем вам, как объединить все четыре подхода и получить поддержку у ваших пользователей. Для этого нам понадобится современный фреймворк создания чат-ботов, легко расширяемый и модифицируемый, способный эффективно выполнять все типы алгоритмов параллельно[1]. С помощью приведенных ранее в этой главе примеров на языке Python мы добавим генератор ответов для каждого из четырех подходов, а затем добавим логику выбора одного из четырех (или более) ответов. Наш чат-бот будет думать, прежде чем говорить, «мысленно» проговаривая сначала все различными

[1] Мы в Aira разрабатываем фреймворк для создания чат-ботов с открытым исходным кодом aichat, чтобы помочь нашим пользователям и их друзьям внести свой вклад в нашу библиотеку диалогов, помогая незрячим и слабовидящим людям и развлекая их: http://github.com/aira/aichat.

способами, ранжируя или сливая воедино некоторые из альтернативных ответов для генерации окончательного ответа. И возможно, мы даже попытаемся работать нашими ответами на благо общества, проверяя их тональность перед «нажатием кнопки “Отправить”».

12.6.1. Will — залог нашего успеха

Will — современный дружественный к разработчику фреймворк создания чат-ботов от Стивена Скочена (Steven Skoczen), может пригодиться в каналах HipChat и Slack и во многих других[1]. Разработчикам Python должна понравиться его модульная архитектура. Впрочем, он достаточно тяжеловесный в смысле требований и инсталляции. К счастью, для него есть контейнер Docker, с помощью которого можно ускорить создание своего сервера чат-бота.

Will применяет регулярные выражения для сопоставления с паттернами, а сам Python можно использовать для любых требующих вычисления логических условий. Для шаблонизации необходима библиотека jinja2. Каждая из этих частей конвейера добавляет адаптивности и гибкости чат-боту. В результате Will намного гибче, чем фреймворки на основе AIML. Но он все же страдает от тех ограничений, которым подвержены все чат-боты, основанные на сопоставлении с паттернами (включая AIML), — он не умеет обучаться на данных; разработчику приходится его обучать путем написания кода для всех без исключения ветвей логического дерева.

Установка Will

В документации по установке Will есть несколько маленьких пробелов. Надеемся, что к моменту публикации этой книги они будут заполнены и вы сможете просто прочитать качественную документацию[2]. На операционной системе Mac OS X можно инсталлировать и запустить сервер redis с помощью команды `brew install redis`.

Привет, Will!

Вот так выглядит разговор с необученным Will, если не считать системных сообщений об ошибках относительно разрешений на порт 80. Впрочем, вы можете разобраться, как избежать этих ошибок:

```
You: Hey
Will: hello!
You: What's up?
Will: I heard you, but I'm not sure what to do.
You: How are you?
Will: Doing alright. How are you?
You: What are you?
Will: Hmm. I'm not sure what to say.
You: Are you a bot?
Will: I didn't understand that.
```

[1] Репозиторий GitHub: https://github.com/skoczen/will.

[2] Документация Will: http://skoczen.github.io/will/.

Как видите, Will после установки ведет себя вежливо, но понимает не слишком много. Имя Will можно легко поменять на Rosa или любое другое. И вы можете воспользоваться своими навыками обработки естественного языка для усовершенствования некоторых из его паттернов и расширения его способностей к общению.

12.7. Процесс проектирования

Для создания полезного приложения менеджеры, ответственные за выпуск программного продукта, и разработчики сочиняют так называемые истории пользователей (user stories) — требования к системе с точки зрения пользователей. Эти «истории» описывают последовательность выполняемых пользователем действий при работе с приложением и желаемую реакцию приложения. Они могут основываться на опыте практического взаимодействия с аналогичными приложениями или описываться на основе запрошенных пользователями новых функций или обратной связи. Для повышения вероятности того, что усилия разработчиков будут сосредоточены на создании чего-то полезного для продукта, функциональные возможности ПО привязываются к этим требованиям с точки зрения пользователей.

Требования пользователей к чат-боту часто составляются в виде возможных высказываний (текстовых сообщений) пользователя чат-боту, для которых указываются соответствующие ответы или действия чат-бота (виртуального помощника). В случае основанного на информационном поиске чат-бота такая таблица «историй» пользователей — все, что требуется для его обучения конкретным ответам и требованиям. Именно разработчик должен выбрать допускающие обобщение требования, чтобы команде проектировщиков не нужно было задавать все, что бот должен понимать, и все, что он может говорить. Как вы считаете, какая из наших четырех методик создания чат-ботов лучше всего подойдет для каждого из приведенных ниже вопросов?

- Hello! => Hello!
- Hi => Hi!
- How are you? => I'm fine. How are you?
- How are you? => I'm a stateless bot, so I don't have an emotional state.
- Who won the 2016 World Series? => Chicago Cubs
- Who won the 2016 World Series? => The Chicago Cubs beat the Cleveland Indians 4 to 3
- What time is it => 2:55 pm
- When is my next appointment? => At 3 pm you have a meeting with the subject "Springboard call"
- When is my next appointment? => At 3 pm you need to help Les with her Data Science course on Springboard
- Who is the President? => Sauli Niinistö
- Who is the President? => Barack Obama

Для конкретного высказывания возможны несколько допустимых ответов, даже для одного и того же пользователя и контекста. Кроме того, несколько различных вопросов могут вызывать одну реакцию чат-бота (или набор возможных реакций). Отображение «многие ко многим» между высказываниями и ответами имеет свои достоинства и недостатки, как и диалог людей. Так, число возможных сочетаний допустимых пар *«высказывание => ответ»* может быть огромным — практически бесконечным (хотя формально конечным).

Необходимо также расширить число сочетаний пар «высказывание — ответ» в пользовательских требованиях с помощью поименованных переменных для часто меняющихся элементов контекста, таких как:

- дата;
- время;
- местоположение: страна, область, город или широта/долгота;
- региональные настройки: например, американское или финское форматирование даты, времени, валют и чисел;
- тип интерфейса: мобильное устройство или ноутбук;
- модальность интерфейса: голосовой или текстовый;
- предыдущие взаимодействия: например, спрашивал ли пользователь недавно подробности бейсбольной статистики;
- потоковые аудио, видео и показания датчиков с мобильного устройства (Aira.io).

В основе API чат-ботов IBM Watson и Amazon Lex лежат базы знаний, которые непросто расширять оперативно и поддерживать в соответствии с быстро меняющимися контекстными переменными. Темпы записи в эти базы знаний слишком низки для обработки всех меняющихся фактов окружающего мира, с которым взаимодействуют чат-бот и пользователь.

Список возможных «историй» пользователей даже для простейших чат-ботов формально конечен, но достаточно велик даже для самого простого реального чат-бота. Один из способов справиться с бурным ростом возможных комбинаций — объединение множества взаимодействий с пользователем в один паттерн. Со стороны высказываний нашей таблицы соответствий такой подход с шаблонами эквивалентен созданию регулярного выражения (или конечного автомата), отражающего определенную группу высказываний, которая должна вызывать определенный паттерн ответа. Со стороны реакций бота этот подход эквивалентен шаблонам `Jinja2`, или `Django`, или f-строк Python.

Возвращаясь к первому чат-боту из главы 1, можно так представить отображение «высказывание => ответ», которое задает соответствие регулярных выражений для высказывания f-строке Python для ответа:

```
>>> pattern_response = {
...     r"[Hh]ello|[Hh]i[!]*":
...         r"Hello {user_nickname}, would you like to play a game?",
...     r"[Hh]ow[\s]*('s|are|'re)?[\s]*[Yy]ou([\s]*doin['g]?)?":
...         r"I'm {bot_mood}, how are you?",
...     }
```

Но сложную логику таким образом не выразить. И это требует написания кода вручную, а не машинного обучения. Поэтому каждое из отображений захватывает не слишком широкий диапазон высказываний и ответов. Модель машинного обучения могла бы обрабатывать широкий диапазон спортивных вопросов или помогать пользователю вести календарь.

ВАЖНО

Не меняйте эти строковые шаблоны на f-строки с f", иначе они будут вычисляться во время создания экземпляра. Ваш бот может мало что знать об окружающем мире в момент создания словаря pattern_response.

Вот несколько примеров требований чат-бота, которые плохо подходят для варианта с шаблонами:

- Where is my home => Your home is 5 minutes away by foot, would you like directions?
- Where am I => You are in SW Portland near Goose Hollow Inn или You are at 2004 SW Jefferson Street
- Who is the President? => Sauli Niinistö или Barack Obama или What country or company ...
- Who won the 2016 World Series? => Chicago Cubs или The Chicago Cubsbeat the Cleveland Indians 4 to 3
- What time is it => 2:55 pm, или 2:55 pm, time for your meeting with Joe, или...

А вот несколько общих вопросов на IQ, слишком специфических, чтобы требовать для каждого варианта отдельной пары «паттерн — ответ». Ответы на вопросы на общую эрудицию обычно берутся из базы знаний. Тем не менее именно так чат-бот Mitsuku, вероятно, смог подобраться близко к правильному ответу в недавнем тесте Байрона Риза (Byron Reese):

- Which is larger, a nickel or a dime? => Physically or monetarily? или A nickel is physically larger and heavier but less valuable monetarily.
- Which is larger, the Sun or a nickel? => The Sun, obviously[1].
- What's a good strategy at Monopoly? => Buy everything you can, and get lucky.
- How should I navigate a corn-row maze? => Keep your hand against one wall of corn and follow it until it becomes an outside wall of the maze.
- Where does sea glass come from? => Garbage... fortunately the polishing of sand and time can sometimes turn human refuse, like broken bottles, into beautiful gemstones.

И хотя превратить их непосредственно в код непросто, их можно преобразовать в автоматизированный набор тестов для NLP-конвейера. Подобные тесты можно использовать для оценки нового подхода к созданию чат-бота или для отслеживания продвижения работы с течением времени[2]. Если вы придумаете еще вопросы на IQ

[1] Байрон Риз, подкаст AI Minute.

[2] Лекция Эндрю Ына (Andrew Ng) студентам Стэнфордской школы бизнеса: https://youtu.be/21EiKfQYZXc?t=48m6s.

для чат-бота, добавьте их в непрерывно растущий список в `nlpia/data/iq_test.csv` (https://github.com/totalgood/nlpia/blob/master/src/nlpia/data/iq_test.csv). И конечно, включите их в автоматизированные тесты вашего собственного чат-бота. Никогда не знаешь, чем может удивить тебя твой чат-бот.

12.8. Маленькие хитрости

При создании чат-бота не помешает иметь в запасе пару небольших хитростей. С их помощью можно гарантировать, что чат-бот не слишком часто будет сбиваться с пути истинного.

12.8.1. Задавайте вопросы с предсказуемыми ответами

При получении вопроса, на который чат-бот не знает ответа, можно отвечать на него с помощью уточняющего вопроса. Если же этот уточняющий вопрос — в рамках сферы знаний или профиля «личности» чат-бота, он может предсказать предполагаемый ответ человека и воспользоваться ответом пользователя для того, чтобы вернуть себе контроль над разговором и направить беседу в более знакомое ему русло. Чтобы не раздражать пользователей, старайтесь делать уточняющие вопросы веселыми или лестными им, каким-то образом приятными для них:

```
Human: "Where were you born?"

Sports Bot: "I don't know, but how about those Mets?"
Therapist Bot: "I don't know, but are you close to your mother?"
Ubuntu Assistant Bot: "I don't know, but how do you shut down your Ubuntu PC at
night?"
```

Семантический поиск можно использовать для того, чтобы найти в базе знаний чат-бота пары «вопрос — ответ», шутки или интересные факты, хотя бы отдаленно связанные с вопросом пользователя.

12.8.2. Развлекайте пользователей

Иногда процесс формирования ответа, близкого к качественному сообщению, может занимать слишком много времени, а чат-бот не может найти подходящий поясняющий вопрос. В подобном случае у чат-бота есть две возможности: 1) признать, что не знает ответа, или 2) придумать non sequitur.

Non sequitur[1] — это высказывание, не имеющее никакого отношения к тому, что спрашивает пользователь. Подобные высказывания обычно считаются неприветливыми, а иногда и манипулятивными. Честность — лучшая политика для ориентированного на благо общества чат-бота. Чем более открыто он ведет себя, тем выше вероятность завоевать доверие пользователей. Пользователю, если вы откроете ему размер своей базы данных ответов или возможных действий, может быть интересно узнать немного о «внутренностях» бота. Можно также показать ему

[1] От лат. «не следует». — *Примеч. пер.*

несколько ответов, не прошедших проверку на грамматику и стиль. Чем честнее вы с пользователем, тем скорее он поведет себя доброжелательно в ответ и попытается помочь чат-боту вернуться на правильный путь. Коул Говард (Cole Howard) обнаружил, что пользователи часто подталкивают к правильному ответу его обученный на базе MNIST бот для распознавания рукописного текста, более разборчиво перерисовывая цифры.

Так что желательно, чтобы бесполезные ответы коммерческого чат-бота были эмоциональными, сбивающими с толку, лестными или смешными. Хотелось бы также, чтобы его ответы выбирались способом, который бы казался людям случайным. Например, не повторяйтесь слишком часто[1]. Разнообразьте структуру, форму и стиль предложений. Таким образом вы сможете отслеживать реакции своих пользователей и оценивать их тональности, чтобы определить, какие из ваших non sequitur менее всего их раздражают.

12.8.3. Если все остальное не дает результата — ищите!

Если чат-бот не может придумать, что сказать, попробуйте сыграть роль поисковой системы или поисковой панели. Ищите веб-страницы или записи внутренней базы данных, которые могут оказаться релевантны полученному от пользователя вопросу. Но не забудьте спросить пользователя, интересна ли ему конкретная страница, судя по заголовку, прежде чем вываливать на него всю содержащуюся в ней информацию. Для многих ботов в этих целях отлично подойдут Stack Overflow, «Википедия» и Wolfram Alpha (поскольку Google умеет это и пользователи этого ждут).

12.8.4. Стремитесь к популярности

Если у вас есть несколько в целом одобрительно воспринимаемых вашей аудиторией шуток, ссылок на ресурсы или ответов, лучше отвечать с помощью них, чем искать наиболее подходящий для заданного вопроса ответ, особенно если слишком хорошего соответствия найти не удается. Эти шутки и ресурсы могут помочь вернуть человека-собеседника на более знакомую боту колею разговора, для которой у вас есть много тренировочных наборов данных.

12.8.5. Объединяйте людей

Пользователи быстро начинают ценить чат-боты, способные играть роль ядра соцсети. Познакомьте пользователя с другими людьми с форума или теми, кто писал о том же, что и он. Или укажите пользователю сообщение в блоге, мем, канал чата или другой веб-сайт, который может быть ему интересен. У хорошего бота должен быть наготове список популярных ссылок на случай, если разговор начинает ходить по кругу.

---

[1] Люди склонны недооценивать число возможных повторов в случайной последовательности: https://mindmodeling.org/cogsci2014/papers/530/paper530.pdf.

Бот: «Возможно, вам будет интересно познакомиться с @SuzyQ, она часто интересовалась этим вопросом в последнее время. Вероятно, она сможет помочь вам разобраться».

12.8.6. Проявляйте эмоции

Автоответчик для входящих сообщений электронной почты в Gmail аналогичен диалоговому чат-боту. Автоответчик должен предлагать ответы на полученные сообщения электронной почты на основе семантики их содержимого. Но для сообщений электронной почты длинная цепочка ответов — редкость. А исходный вопрос автоответчика электронной почты обычно гораздо длиннее, чем у чат-бота. Тем не менее задачи создания обоих требуют генерации текстовых ответов на входящие текстовые запросы. Многие из применимых к одной из этих задач методов могут подойти и для второй.

И хотя у Google есть доступ к миллионам сообщений электронной почты, функция Smart Reply в Gmail Inbox обычно склонна возвращать более краткие, универсальные и прямолинейные ответы. Если пытаться максимизировать правильность ответа для среднего пользователя электронной почты, то семантический поиск также будет приводить к генерации относительно универсальных, прямолинейных ответов. Усредненный ответ вряд ли будет отражать какую-либо личность или эмоции. В Google для добавления в предлагаемые ответы капельки эмоциональности подключили очень неожиданный корпус... любовные романы.

Любовные романы обычно отличаются предсказуемым сюжетом и слащавыми диалогами, которые, как оказалось, легко анализировать и имитировать. И они очень эмоциональны. Я не уверен, что Google выбрал фразы вроде *That's awesome! Count me in!* или *How cool! I'll be there*. из любовных романов, но утверждают, что именно это источник всех эмоциональных возгласов, предлагаемых функцией Smart Reply.

12.9. На практике

Созданный нами здесь гибридный чат-бот достаточно гибок для использования в большинстве распространенных реальных приложений. На самом деле вам, наверное, уже приходилось на этой неделе взаимодействовать с подобным чат-ботом, если вами использовались:

- виртуальные помощники;
- боты консультанты-продавцы;
- маркетинговые (спам-) боты;
- игровые и вспомогательные боты;
- ИИ для компьютерных игр;
- мобильные помощники;
- помощники для «умных» домов;
- боты-психоаналитики;
- автоматические предложения ответов на сообщения электронной почты.

Вероятно, вы будете сталкиваться с подобными созданным в этой главе чат-ботами все чаще и чаще. Пользовательские интерфейсы постепенно уходят от архитектур, ограниченных жесткой логикой и структурами данных машин. Все больше машин обучается общению с людьми при естественном, живом разговоре. Голосовой паттерн проектирования все больше набирает популярность по мере того, как чат-боты становятся удобнее и меньше раздражают людей. А приведенные подходы к созданию диалоговых систем обещают пользователям более полный и насыщенный опыт взаимодействия, чем нажатие клавиш и «свайп» влево. И по мере неявного взаимодействия с нами чат-ботов они все глубже и глубже внедряются в общественное сознание.

Итак, вы узнали, как создавать чат-боты для забавы и ради прибыли, как создать на удивление интеллектуальный бот путем комбинации порождающих диалоговых моделей, семантического поиска, сопоставления с паттернами и информационного поиска (в базах знаний).

Вы освоили все ключевые NLP-компоненты интеллектуального чат-бота. Осталось только придать ему спроектированную вами «личность». А дальше, вероятно, необходимо его масштабировать, чтобы он продолжал обучаться еще долго после исчерпания оперативной памяти, пространства на жестком диске и ресурсов CPU вашего ноутбука. В главе 13 мы покажем вам, как добиться этого.

Резюме

- Путем комбинации нескольких испытанных подходов можно создать интеллектуальный диалоговый движок.
- Один из ключей к интеллектуальности чат-бота — выбор наилучшего ответа из числа сгенерированных с помощью четырех основных подходов к созданию чат-ботов.
- Машину можно обучить накопленным за много лет знаниям, не тратя всю жизнь на программирование этого.

13 Масштабирование: оптимизация, распараллеливание и обработка по батчам

В этой главе

- Масштабирование конвейера NLP.
- Ускорение поиска с помощью индексации.
- Обработка по батчам для сокращения объема потребляемой памяти.
- Распараллеливание для ускорения NLP.
- Выполнение обучения NLP-модели на GPU.

Из главы 12 вы узнали, как использовать все утилиты из своего NLP-арсенала для создания NLP-конвейера, способного на ведение разговора. Мы показали примитивные примеры этой возможности чат-ботов на маленьких наборах данных. Человекоподобие (IQ) диалоговой системы, похоже, ограничивается данными, на которых она обучается. Большинство упомянутых в этой книге подходов покажут лучшие результаты, если масштабировать их на поддержку бóльших наборов данных.

Наверное, вы обратили внимание, что ваш компьютер зависает или даже выдает фатальный сбой при запуске некоторых из приведенных примеров на больших

наборах данных. Какие-то наборы данных из `nlpia.data.loaders.get_data()` требуют больше оперативной памяти (RAM), чем установлено на многих PC и ноутбуках.

Еще одно узкое место, помимо оперативной памяти, в конвейерах обработки естественного языка — процессор. Даже при наличии неограниченного объема оперативной памяти обработка больших корпусов с помощью некоторых из приведенных алгоритмов занимает дни.

Итак, необходимо придумать алгоритмы, минимизирующие требуемые ресурсы:

- энергозависимую память (RAM);
- обработку (количество циклов CPU).

13.1. Слишком много хорошего (данных)

При добавлении дополнительных данных, то есть дополнительных знаний, в конвейер модели машинного обучения требуют все больше и больше RAM, места для хранения данных и циклов CPU для их обучения. Хуже того, некоторые из методов основаны на вычислении расстояния или подобия векторных представлений высказываний либо документов, со сложностью $O(N^2)$. Эти алгоритмы начинают работать намного медленнее при увеличении объемов данных. Обработка каждого нового предложения в корпусе требует больше байтов RAM и циклов CPU, чем предыдущего, что неудобно на практике даже для корпусов среднего размера.

Существует два общих подхода, с помощью которых можно избежать этих проблем при масштабировании конвейера NLP на бóльшие наборы данных:

- *повышенная масштабируемость* — усовершенствование либо оптимизация алгоритмов;
- *горизонтальное масштабирование* — распараллеливание алгоритмов для выполнения нескольких вычислений одновременно.

В этой главе вы узнаете о методиках для обоих подходов.

Усовершенствование алгоритмов — практически всегда оптимальный способ ускорения конвейера обработки, поэтому поговорим сначала о нем. Распараллеливание же оставим для второй части данной главы, чтобы еще более ускорить выполнение уже «прилизанных», оптимальных алгоритмов.

13.2. Оптимизация алгоритмов NLP

В некоторых из встречавшихся в предыдущих главах алгоритмах есть элементы с квадратичной ($O(N^2)$) или еще более высокой сложностью:

- компиляция тезауруса синонимов на основе подобия векторов `word2vec`;
- кластеризация веб-страниц на основе их векторов тем;
- кластеризация статей из журналов или других документов на основе векторов тем;
- кластеризация вопросов в корпусе Q&A для автоматического формирования ЧаВо.

Все эти непростые задачи NLP относятся к категории поиска по индексу, или векторного поиска методом k-ближайших соседей (KNN). Мы посвятим несколько следующих разделов обсуждению задачи масштабирования: оптимизации алгоритмов. Покажем один конкретный метод оптимизации алгоритмов — *индексацию* (indexing). Индексация может помочь решить большинство задач векторного поиска (KNN). Во второй части этой главы вы узнаете, как гиперраспараллелить обработку естественного языка за счет использования тысяч процессорных ядер из графического процессора (GPU).

13.2.1. Индексация

Вероятно, вы используете индексы для естественного языка ежедневно. Текстовый индекс естественного языка (или обратный индекс) можно встретить в конце учебника, с его помощью можно найти страницу для интересующей вас темы. Страницы в этом случае играют роль документов, а слова — лексикона векторов мультимножеств слов (BOW) для каждого из документов. Еще мы используем текстовый индекс при любом поиске в Интернете. Для масштабирования NLP-приложения необходимо проделать подобное для семантических векторов, например, векторов «документ — тема» LSA или векторов слов `word2vec`.

В предыдущих главах мы упоминали использование условных обратных индексов для поиска в документах набора слов или токенов на основе слов из запроса. Но мы пока не обсуждали *приближенный* поиск *подобного* текста методом KNN. При поиске методом KNN ищутся подобные, хотя не обязательно содержащие одинаковые слова, строки. Одна из метрик расстояния, используемых такими пакетами, как `fuzzywuzzy` и `ChatterBot`, для поиска подобных строк, — расстояние Левенштейна.

В базах данных реализовано множество разнообразных текстовых индексов для быстрого поиска документов или строк. SQL-запросы позволяют искать текст, соответствующий, например, таким паттернам, как:

```
SELECT book_title from manning_book WHERE book_title LIKE 'Natural Language%in
Action'
```

В результате выполнения этого запроса будут найдены все книги издательства Manning из серии In Action, начинающиеся с Natural Language. Существуют также индексы на основе триграмм (`trgm`) для множества баз данных, с помощью которых можно быстро (за постоянное время) находить текст, причем без указания паттерна, достаточно только указать текстовый запрос, подобный искомому.

Эти методы индексации текста для баз данных отлично подходят для текстовых документов и любых строк. Но с семантическими векторами вроде векторов `word2vec` или плотными векторами «документ — тема» они работают не так хорошо. В основе условных индексов баз данных лежит допущение, что индексируемые объекты (документы) — дискретные, разреженные или низкой размерности:

- строки (последовательности символов) дискретны: количество возможных символов ограничено;

- векторы TF-IDF разрежены: частотность большинства термов в любом конкретном документе равна 0;
- векторы BOW одновременно дискретны и разрежены: термы дискретны, а частотность большинства слов в конкретном документе равна 0.

Именно благодаря этому для поиска в Интернете, поиска документов и географического поиска достаточно миллисекунд. И эти алгоритмы десятилетиями работают эффективно (со сложностью порядка `O(1)`).

Почему же индексировать непрерывные векторы вроде векторов «документ — тема» LSA (см. главу 4) или векторов `word2vec` (см. главу 6) так трудно? В конце концов, векторы систем географической информации (GIS) обычно представляют собой широту, долготу и высоту над уровнем моря. Поиск по GIS в Google Maps занимает миллисекунды. К счастью, векторы GIS содержат только три непрерывных значения, так можно строить индексы, ограниченные «невидимыми рамками», распределяя объекты GIS по дискретным группам.

Для решения этой задачи можно использовать несколько различных видов индексных структур данных:

- k-мерное дерево: в следующих выпусках Elasticsearch эту структуру данных обещают реализовать вплоть до восьмимерного случая;
- Rtree: в версиях PostgreSQL >=9.0 она реализована вплоть до 200-мерного случая;
- Minhash (https://en.wikipedia.org/wiki/MinHash) или другие методы хеширования с учетом локальности: `pip install lshash3`.

Впрочем, они работают до определенного предела, который наступает примерно по достижении размерности 12. Если поэкспериментируете с оптимизацией индексов баз данных или методами хеширования с учетом локальности, то сами увидите, что сохранять постоянную скорость поиска с ростом размерности становится все сложнее и сложнее. При размерности около 12 это просто невозможно.

Так что же делать с нашими 300-мерными векторами `word2vec` или более чем 100-мерными семантическими векторами LSA? На помощь приходит аппроксимация. Алгоритмы приближенного вычисления ближайших соседей не стремятся найти точный набор векторов документов, наиболее похожих на вектор из запроса. Вместо этого они стараются найти наиболее подходящий вариант. И обычно они работают очень хорошо, в десяти первых результатах почти всегда присутствуют все хорошо подходящие векторы.

Но все иначе при использовании возможностей SVD или вложений для понижения размерности токенов (при размере словаря, исчисляемом миллионами) до, скажем, 200 или 300. А именно, отличаются три вещи. Одна из них на самом деле скорее достоинство: число измерений (можете думать о них как о столбцах в таблице базы данных), по которым нужно искать, становится меньше. Две другие, наоборот, усложняют задачу: приходится иметь дело с плотными векторами непрерывных значений.

13.2.2. Продвинутая индексация

Семантические векторы — во всех отношениях непростые объекты. Они сложны, поскольку:

- высокой размерности;
- содержат вещественные значения;
- плотные.

Мы поменяли шило (проклятие размерности) на мыло (две новые проблемы). Теперь наши векторы — плотные (нет нулей, которые можно проигнорировать) и непрерывные (вещественнозначные).

В каждом измерении наших плотных семантических векторов содержатся значимые данные. Больше нельзя пропускать/игнорировать нули, наполнявшие таблицы TF-IDF или BOW (см. главы 2 и 3). Даже если заполнить пропуски в векторах TF-IDF с помощью аддитивного (лапласовского) сглаживания, в нашей плотной таблице все равно остались бы некоторые согласованные значения, так что ее можно было бы рассматривать как разреженную матрицу. Но теперь в наших векторах нет нулей или заполнителей в виде наиболее распространенных значений. Каждой теме соответствует какой-либо вес для каждого документа. Эта проблема вполне решаема. Понижение размерности более чем компенсирует проблему плотности.

Значения в плотных векторах — вещественные. Но есть и еще бóльшая проблема. Веса тем в семантических векторах могут принимать положительные или отрицательные значения и не ограничиваются дискретными символами или целочисленными количествами вхождений. Веса всех тем теперь представляют собой непрерывные вещественные значения (например, типа `float`). Недискретные значения, например значения с плавающей точкой, невозможно индексировать. Они не просто есть или нет. Их нельзя векторизовать с помощью унитарного кодирования входного сигнала в признаки нейронной сети. И уж точно нельзя создать запись в индексной таблице, которая ссылается на все документы, в которых данный признак присутствует либо отсутствует. Темы теперь присутствуют во всех документах, только в разной степени.

Решить приведенные в начале данной главы проблемы поиска на естественном языке можно, если найти эффективный алгоритм поиска либо метод KNN. Один из способов оптимизации алгоритма решения подобных задач — пожертвовать определенностью и точностью в пользу колоссального ускорения поиска. Такая методика называется приближенным поиском ближайших соседей (ANN). Например, поисковая система DuckDuckGo не пытается найти идеально подходящее соответствие для семантического вектора из поискового запроса, а выдает ближайшие десять или около того приближенных соответствий.

К счастью, множество компаний открыли немало исходного кода своих проектов, связанных с исследованиями в сфере повышения масштабируемости ANN. Их исследовательские группы соревнуются, кто сможет создать для вас самое

удобное и быстрое ПО для ANN поиска. Вот несколько библиотек и пакетов Python, возникших в результате этого соревнования, проверенных на стандартных тестах производительности для NLP-задач в IT-университете Копенгагена (ITU)[1]:

- пакет `Annoy`[2] от Spotify;
- BallTree (с помощью `nmslib`)[3];
- библиотека Basic Linear Algebra Subprograms («Основные подпрограммы для линейной алгебры»)[4];
- библиотека Non-Metric Space («Неметрические пространства», NMSlib)[5];
- эффективная реализация метода ближайших соседей на основе понижения размерности и поиска по гиперкубу (Dimension reductiOn and LookuPs on a Hypercube for effIcient Near Neighbor, DolphinnPy)[6];
- библиотека `rpforest`[7];
- библиотека `datasketch`[8];
- пакет MIH[9];
- библиотека FALCONN (Fast Lookup of Cosine and Other Nearest Neighbors)[10];
- библиотека FLANN (Fast Lookup of Approximate Nearest Neighbors,)[11];

---

1 Проделанное в ITU сравнение результатов тестов производительности ANN: http://www.itu.dk/people/pagh/SSS/ann-benchmarks/.

2 См. веб-страницу GitHub — spotify/annoy: Approximate Nearest Neighbors in C++/Python for memory usage and loading/saving to disk по адресу https://github.com/spotify/annoy.

3 См. веб-страницу GitHub — nmslib/nmslib: Non-Metric Space Library (NMSLIB): An efficient similarity search library and a toolkit for evaluation of k-NN methods for generic non-metric spaces по адресу https://github.com/searchivarius/nmslib.

4 См. веб-страницу 1.6. Nearest Neighbors — scikit-learn 0.19.2 documentation по адресу http://scikit-learn.org/stable/modules/neighbors.html#brute-force.

5 См. веб-страницу GitHub — nmslib/nmslib: Non-Metric Space Library (NMSLIB): An efficient similarity search library and a toolkit for evaluation of k-NN methods for generic non-metric spaces по адресу https://github.com/searchivarius/NMSLIB.

6 См. веб-страницу GitHub — ipsarros/DolphinnPy: High-dimensional approximate nearest neighbor in python: https://github.com/ipsarros/DolphinnPy.

7 См. веб-страницу GitHub — lyst/rpforest: It is a forest of random projection trees по адресу https://github.com/lyst/rpforest.

8 См. веб-страницу GitHub — ekzhu/datasketch: MinHash, LSH, LSH Forest, Weighted MinHash, HyperLogLog, HyperLogLog++ по адресу https://github.com/ekzhu/datasketch.

9 См. веб-страницу GitHub — norouzi/mih: Fast exact nearest neighbor search in Hamming distance on binary codes with Multi-index hashin по адресу https://github.com/norouzi/mih.

10 См. веб-страницу FALCONN: PyPI по адресу https://pypi.python.org/pypi/FALCONN.

11 FLANN — Fast Library for Approximate Nearest Neighbors: http://www.cs.ubc.ca/research/flann/.

- HNSW («иерархический, допускающий навигацию, маленький мирок», Hierarchical Navigable Small World) (в `nmslib`)[1];
- k-мерные деревья (`kdtree`)[2];
- `nearpy` (https://pypi.python.org/pypi/NearPy).

Один из наиболее простых из этих подходов к индексации реализован в пакете `Annoy` от Spotify.

13.2.3. Продвинутая индексация с помощью пакета Annoy

В недавнем обновлении `word2vec` (`KeyedVectors`) в библиотеке `gensim` появился продвинутый вариант индексации. Теперь есть готовое решение, с помощью которого можно находить приближенных ближайших соседей любого вектора за миллисекунды. Но как вы увидели в начале данной главы, индексацию необходимо использовать для любого набора многомерных плотных непрерывных векторов, а не только для векторов `word2vec`. Можете воспользоваться модулем `Annoy` для индексации `word2vec` и сравнить полученные результаты с индексом `KeyedVectors` из библиотеки `gensim`. Во-первых, необходимо загрузить векторы `word2vec` (см. главу 6), как показано в листинге 13.1.

Листинг 13.1. Загрузка векторов word2vec

```
>>> from nlpia.loaders import get_data
>>> wv = get_data('word2vec')
100%|############################| 402111/402111 [01:02<00:00, 6455.57it/s]
>>> len(wv.vocab), len(wv[next(iter(wv.vocab))])
(3000000, 300)
>>> wv.vectors.shape
(3000000, 300)
```

Если вы еще не скачивали файл GoogleNews-vectors-negative300.bin.gz (https://bit.ly/GoogleNews-vectors-negative300) в каталог nlpia/src/nlpia/bigdata/, то функция get_data() скачает его для вас

Создаем пустой индекс `Annoy` с нужным числом измерений для наших векторов, как показано в листинге 13.2.

Листинг 13.2. Инициализация 300-мерного AnnoyIndex

```
>>> from annoy import AnnoyIndex
>>> num_words, num_dimensions = wv.vectors.shape
>>> index = AnnoyIndex(num_dimensions)
```

Исходная модель word2vec GoogleNews содержит 3 миллиона 300-мерных векторов

Теперь можно добавить наши векторы `word2vec` в индекс `Annoy` по одному. Этот процесс можно сравнить с постраничным чтением книги с включением номеров страниц для каждого из найденных слов в таблицу обратного индекса в конце

[1] См. веб-страницу nmslib/hnsw.hat master: nmslib/nmslib по адресу https://github.com/searchivarius/nmslib/blob/master/similarity_search/include/factory/method/hnsw.h.

[2] См. GitHub-репозиторий для kdtree по адресу https://github.com/stefankoegl/kdtree.

книги. Конечно, поиск ANN устроен намного сложнее, но Annoy значительно его упрощает (листинг 13.3).

Листинг 13.3. Добавляем векторы слов в AnnoyIndex

tqdm() принимает на входе объект итерируемого типа, возвращает также итерируемый объект (например, enumerate()) и вставляет в цикл код для отображения индикатора хода выполнения

.index2word представляет собой неотсортированный список всех 3 миллионов токенов словаря, эквивалентный карте соответствий целочисленных индексов (0-2999999) токенам (от '</s>' до 'snowcapped_Caucasus')

```
>>> from tqdm import tqdm
>>> for i, word in enumerate(tqdm(wv.index2word)):
...     index.add_item(i, wv[word])
22%|#######?                   | 649297/3000000  [00:26<01:35, 24587.52it/s]
```

У объекта AnnoyIndex осталась последняя задача: просмотр всего индекса и кластеризация векторов по небольшим порциям, которые можно индексировать в древовидной структуре, как показано в листинге 13.4.

Листинг 13.4. Формируем индекс евклидовых расстояний, состоящий из 15 деревьев

Просто эвристическое правило — этот гиперпараметр имеет смысл менять, если производительность индекса недостаточна для важных моментов (RAM, поиска по справочнику, индексации) или индекс недостаточно точен для вашего приложения

round(ln(3000000)) => 15 деревьев индекса для наших 3 миллионов векторов — на ноутбуке занимает пару минут

Сохраняем индекс в локальный файл и освобождаем оперативную память, это может занять несколько минут

```
>>> import numpy as np
>>> num_trees = int(np.log(num_words).round(0))
>>> num_trees
15
>>> index.build(num_trees)
>>> index.save('Word2vec_euc_index.ann')
True
>>> w2id = dict(zip(range(len(wv.vocab)), wv.vocab))
```

Мы построили 15 (примерно натуральный логарифм 3 миллионов) деревьев, поскольку нам нужно производить поиск по 3 миллионам векторов. Если векторов у вас больше либо вам хочется, чтобы индекс был быстрее или точнее, можно увеличить количество деревьев. Но лучше не делать его слишком большим, иначе процесс индексации затянется надолго.

Теперь можно поискать в индексе какое-нибудь слово из нашего словаря, как показано в листинге 13.5.

Листинг 13.5. Ищем соседей токена Harry_Potter в AnnoyIndex

Словарь KeyedVectors.vocab библиотеки gensim содержит объекты Vocab вместо простых строк или числовых индексов

Из объекта Vocab библиотеки gensim можно узнать, сколько раз встречалась биграмма Harry_Potter в корпусе GoogleNews... Оказывается, почти 3 миллиона раз

```
>>> wv.vocab['Harry_Potter'].index
9494
>>> wv.vocab['Harry_Potter'].count
2990506
>>> w2id = dict(zip(
```

```
...     wv.vocab, range(len(wv.vocab))))
>>> w2id['Harry_Potter']
9494
>>> ids = index.get_nns_by_item(
...     w2id['Harry_Potter'], 11)
>>> ids
[9494, 32643, 39034, 114813, ..., 113008, 116741, 113955, 350346]
>>> [wv.index2word[i] for i in ids]
 ['Harry_Potter',
 'Narnia',
 'Sherlock_Holmes',
 'Lemony_Snicket',
 'Spiderwick_Chronicles',
 'Unfortunate_Events',
 'Prince_Caspian',
 'Eragon',
 'Sorcerer_Apprentice',
 'RL_Stine']
```

Создаем аналогичный wv.vocab ассоциативный массив, содержащий соответствия токенов (целочисленным) значениям индексов

Annoy возвращает сначала целевой вектор, так что нужно запросить 11 соседей, если мы хотим получить 10, помимо целевого

Выведенные пакетом `Annoy` десять ближайших соседей — в основном книги, относящиеся к тому же общему жанру, что и «Гарри Поттер», но вовсе не точные синонимы названий книг, фильмов или имени персонажа. Наши результаты — действительно приближенные ближайшие соседи. Не забывайте, что используемый `Annoy` алгоритм — стохастический, аналогичный алгоритму машинного обучения типа «случайный лес»[1]. Поэтому полученный вами список будет отличаться от приведенного тут. Для обеспечения повторяемых результатов можно воспользоваться методом `AnnoyIndex.set_seed()` для инициализации генератора случайных чисел.

Похоже, что в индексе `Annoy` отсутствует немало более близких соседей нашего токена и результаты взяты из более широкой окрестности искомого терма, а не из числа ближайших десяти. А как насчет библиотеки `gensim`? Что, если попробовать извлечь ближайших десять соседей с помощью встроенного индекса `KeyedVector` библиотеки `gensim` (листинг 13.6).

Листинг 13.6. Ближайшие соседи Harry_Potter, согласно индексу gensim.KeyedVector

```
>>> [word for word, similarity in wv.most_similar('Harry_Potter', topn=10)]
['JK_Rowling_Harry_Potter',
 'JK_Rowling',
 'boy_wizard',
 'Deathly_Hallows',
 'Half_Blood_Prince',
 'Rowling',
 'Actor_Rupert_Grint',
 'HARRY_Potter',
 'wizard_Harry_Potter',
 'HARRY_POTTER']
```

Намного более релевантный список десяти ближайших синонимов. В него включены правильный автор, альтернативные названия книги, названия других

[1] Для генерации хешей с учетом локальности Annoy использует случайные проекции (http://en.wikipedia.org/wiki/Locality-sensitive_hashing#Random_projection).

книг из серии и даже один из игравших в фильмах про Гарри Поттера актеров. Но результаты Annoy могут быть полезны в случаях, когда нас больше интересует жанр или общий смысл слова, а не точные синонимы. Весьма удобно.

Но приближенная индексация Annoy, скажем прямо, «срезает некоторые углы». Чтобы решить эту проблему, перестроим индекс на основе косинусной метрики расстояния (вместо евклидовой) и увеличим количество деревьев. В итоге должна повыситься точность метода ближайших соседей, как и соответствие ее результатов результатам gensim (листинг 13.7).

Листинг 13.7. Создаем индекс на основе косинусного расстояния

```
>>> index_cos = AnnoyIndex(
...     f=num_dimensions, metric='angular')
>>> for i, word in enumerate(wv.index2word):
...     if not i % 100000:
...         print('{}: {}'.format(i, word))
...     index_cos.add_item(i, wv[word])
0: </s>
100000: distinctiveness
    ...
2900000: BOARDED_UP
```

При указании параметра metric='angular' для вычисления кластеров и хешей используется угловое расстояние. Возможные значения этого параметра: 'angular', 'euclidean', 'manhattan' и 'hamming'

Альтернативный способ отслеживания хода выполнения, если tqdm не подходит

Теперь увеличим вдвое количество деревьев и зададим начальное значение для генератора случайных чисел, чтобы вы получили те же результаты, что и в листинге 13.8.

Листинг 13.8. Создаем индекс на основе косинусного расстояния

```
>>> index_cos.build(30)
>>> index_cos.save('Word2vec_cos_index.ann')
True
```

30 эквивалентно int(np.log(num_vectors).round(0)), вдвое больше, чем раньше

Выполнение этой индексации займет вдвое больше времени, но зато результаты должны быть ближе к полученным от gensim. Теперь посмотрим, насколько приближенными оказались ближайшие соседи при нашем более точном индексе (листинг 13.9).

Листинг 13.9. Соседи Harry_Potter в мире косинусного расстояния

```
>>> ids_cos = index_cos.get_nns_by_item(w2id['Harry_Potter'], 10)
>>> ids_cos
[9494, 37681, 40544, 41526, 14273, 165465, 32643, 420722, 147151, 28829]
>>> [wv.index2word[i] for i in ids_cos]
['Harry_Potter',
 'JK_Rowling',
 'Deathly_Hallows',
 'Half_Blood_Prince',
 'Twilight',
 'Twilight_saga',
 'Narnia',
 'Potter_mania',
 'Hermione_Granger',
 'Da_Vinci_Code']
```

Вы получите другие результаты. Случайные проекции для LSH — стохастические. Если нужна воспроизводимость результатов — воспользуйтесь методом AnnoyIndex.set_seed()

Уже лучше. По крайней мере, в списке присутствует правильный автор книги. Можно сравнить результаты двух поисков с помощью Annoy с правильным ответом от gensim, как показано в листинге 13.10.

Листинг 13.10. Точность результатов поиска для первых десяти списков

Оставляем в качестве упражнения читателю объединение этих первых десяти списков в единый объект DataFrame

```
>>> pd.DataFrame(annoy_top10, columns=['annoy_15trees',
...                                    'annoy_30trees'])
                           annoy_15trees        annoy_30trees
gensim
JK_Rowling_Harry_Potter       Harry_Potter        Harry_Potter
JK_Rowling                         Narnia          JK_Rowling
boy_wizard                 Sherlock_Holmes     Deathly_Hallows
Deathly_Hallows             Lemony_Snicket   Half_Blood_Prince
Half_Blood_Prince     Spiderwick_Chronicles            Twilight
Rowling                  Unfortunate_Events       Twilight_saga
Actor_Rupert_Grint         Prince_Caspian               Narnia
HARRY_Potter                       Eragon        Potter_mania
wizard_Harry_Potter    Sorcerer_Apprentice    Hermione_Granger
HARRY_POTTER                     RL_Stine       Da_Vinci_Code
```

Чтобы избавиться от избыточного синонима Harry_Potter, можно вывести 11 лучших результатов и пропустить первый. Но здесь уже виден прогресс. При повышении количества индексных деревьев Annoy ранг менее релевантных термов (таких как Narnia) понижается и появляются более релевантные термы из числа настоящих ближайших соседей (например, JK_Rowling и Deathly_Hallows).

Кроме того, с помощью индекса Annoy можно получить приближенный ответ гораздо быстрее, чем точные результаты с применением индекса gensim. А еще индекс Annoy можно использовать для поиска по любым многомерным, непрерывным, плотным векторам, например по векторам «документ — тема» LSA или вложениям документов (векторам) doc2vec.

13.2.4. Зачем вообще использовать приближенные индексы

Те, кому приходилось заниматься анализом эффективности алгоритмов, могут сказать, что алгоритмы сложности $O(N^2)$ теоретически довольно эффективны. В конце концов, они эффективнее алгоритмов с экспоненциальной сложностью и даже с полиномиальной. Они явно не входят в класс NP по сложности вычислений. Это не какие-то задачи, вычисление которых требует времени, сравнимого со временем жизни Вселенной.

А поскольку вычисления со сложностью $O(N^2)$ нужны только для обучения моделей в нашем конвейере NLP, то их можно выполнить заранее. Чат-боту не нужно производить $O(N^2)$ операций для каждого ответа на новое высказывание. И N^2 операции, по сути, допускают распараллеливание. Практически всегда можно выполнить одну из N последовательностей вычислений независимо от

остальных N последовательностей[1]. Так что можно просто выделить на вычисление задачи больше оперативной памяти и процессоров и выполнять процесс обучения по батчам каждую ночь или каждые выходные, для поддержания актуальности «мозгов» бота[2]. Что еще лучше, можно просто «откусывать» порции от N^2 вычислений и выполнять их по одной, в инкрементном режиме, по мере поступления данных, увеличивающих это значение N.

Например, представьте себе, что мы для начала обучили чат-бот на маленьком наборе данных, а затем «отпустили его на волю». Пусть N — число высказываний и ответов в его постоянной памяти (базе данных). Каждый раз, когда кто-нибудь обращается к чат-боту с новым высказыванием, бот ищет в своей базе данных наиболее похожее высказывание, чтобы снова использовать ответы, подходившие для этого высказывания в прошлом. Итак, мы вычисляем какой-либо показатель (метрику) подобия между N имеющимися высказываниями и новым высказыванием и сохраняем новые показатели подобия в нашей матрице подобия размером $(N + 1)^2$ в виде новой строки и нового столбца. Или просто добавляем еще N связей (взаимосвязей) в структуру данных в виде графа, в котором хранятся все показатели подобия между высказываниями. Теперь можно выполнить запрос к связям (ячейкам в матрице связей) и найти минимальное расстояние. При более простом подходе достаточно просто просмотреть только что вычисленные N показателей. Но если подойти к делу более основательно, можно просмотреть и другие строки и столбцы (заглянуть в граф чуть глубже), чтобы найти, например, ответы на аналогичные высказывания и проверить такие метрики, как доброжелательность, информационное содержимое, тональность, грамматическую правильность, правильность синтаксиса, краткость и стиль. В любом случае сложность алгоритма вычисления наилучшего ответа составляет O(N), хотя общая сложность полного обучения — порядка $O(N^2)$.

Но что, если даже O(N) недостаточно? Что, если нужно создать по-настоящему масштабный искусственный интеллект, вроде Google, в котором N превышает 60 триллионов?[3] Даже если наше N не настолько велико, но отдельные вычисления очень сложны или ответ должен быть получен за приемлемое время (десятки миллисекунд), необходимо использовать индекс.

13.2.5. Решение проблемы индексации: дискретизация

Мы только что утверждали, что значения с плавающей точкой (вещественные значения) невозможно наивно проиндексировать. Как доказать, что мы не правы, или по крайней мере подойти к индексации менее «наивно»? Те из вас, кому доводилось работать с данными датчиков и аналогово-цифровыми преобразователями, наверное, подумают, что непрерывные величины легко можно превратить в цифровые или дискретные. Да и тип `float` на самом деле не совсем непрерывный. Это лишь набор

[1] Точнее, N – 1. — *Примеч. пер.*

[2] Это вполне реальная архитектура, которую мы использовали для задачи сопоставления N^2 документов.

[3] Руководство Google по индексации поисковых систем: https://www.google.com/insidesearch/howsearchworks/thestory/.

битов, в конце концов. Но их нужно сделать *по-настоящему* дискретными, чтобы они вписались в нашу концепцию индекса, сохранив низкую размерность. Необходимо разбить их по группам так, чтобы получилось что-то подходящее для наших целей. В листинге 13.11 показан примерный код реализации простейшего способа преобразования непрерывной переменной в достаточно малое число дискретных или порядковых значений.

Листинг 13.11[1]. Использование MinMaxScaler для векторов низкой размерности

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> real_values = [-1.2, 3.4, 5.6, -7.8, 9.0]
>>>
>>> scaler = MinMaxScaler()
>>> scaler.fit(real_values)

[int(x * 100.) for x in scaler.transform(real_values)]
[39, 66, 79, 0, 100]
```

Масштабируем наши значения с плавающей точкой, чтобы они оказались в интервале от 0.0 до 1.0

Масштабированные, дискретизированные целочисленные значения в диапазоне от 0 до 100

Для пространств низкой размерности такой способ вполне подходит. По сути, именно он применяется в некоторых двумерных индексах GIS для дискретизации значений широты/долготы в виде сетки ограничивающих прямоугольников. Для каждой из точек этой сетки точка в двумерном пространстве или есть, или отсутствует. По мере роста размерности приходится использовать все более и более сложные, эффективные индексы вместо простой двумерной сетки.

Представим себе все это наглядно в трехмерном пространстве, прежде чем перейти к 300-мерным семантическим векторам естественного языка. Например, задумайтесь, что меняется при переходе от двумерного к трехмерному пространству при добавлении высоты над уровнем моря в какую-нибудь базу GIS двумерных значений широты и долготы. А теперь мысленно разбейте Землю на трехмерные кубы вместо вышеупомянутой двумерной сетки. Большинство из этих кубов содержит мало чего интересного для людей, а поиск с учетом расстояния, например поиск всех объектов внутри какой-либо трехмерной сферы или трехмерного куба, оказывается намного более трудоемкой операцией. Количество точек сетки, которые нужно просмотреть, растет пропорционально N^3, где N — диаметр области поиска. Понятно, что при повышении размерности с 3 до 4 или 5 поиск должен быть достаточно интеллектуальным.

13.3. Алгоритмы с постоянным расходом RAM

Одна из главных проблем при работе с большими корпусами и матрицами TF-IDF — как уместить их все в RAM. В этой книге везде используется `gensim`, потому что их алгоритмы пытались сохранять постоянный объем потребляемой памяти.

[1] Для того чтобы этот код работал, входной массив для scaler.fit должен быть двумерным. То есть нужно объявить его как: real_values = [[–1.2], [3.4], [5.6], [–7.8], [9.0]]. — *Примеч. пер.*

13.3.1. gensim

Что делать, если документов больше, чем помещается в памяти? В результате роста размера и разнообразия документов в корпусе в конце концов вы выйдете за пределы RAM даже на самых больших машинах, которые только можно арендовать на облачном сервисе. Не волнуйтесь, математики спешат на помощь!

Математический аппарат таких алгоритмов, как LSA, существует уже десятилетия. У математиков и специалистов в области компьютерных наук было предостаточно времени поэкспериментировать с ними и научиться выполнять их *во внешней памяти*. Это значит, что не все необходимые для работы алгоритма объекты должны находиться в оперативной памяти (RAM) одновременно, а значит, вы больше не ограничены объемом RAM вашей машины.

Даже если вы не хотите распараллелить выполнение конвейера обучения на несколько машин, для больших наборов данных понадобится реализация с постоянным объемом используемой памяти. Класс `LsiModel` библиотеки `gensim` — одна из таких работающих во внешней памяти реализаций сингулярного разложения для LSA[1].

Даже для не таких больших наборов данных `gensimLSIModel` лучше, поскольку не требует бóльших объемов RAM при увеличении размера словаря или набора документов. Так что можно не волноваться, что в процессе обработки корпуса начнется свопинг на диск или программа начнет зависать, когда закончится RAM. Можно даже продолжать использовать ноутбук для других задач, пока модель `gensim` обучается в фоновом режиме.

Подобное эффективное применение оперативной памяти достигается в `gensim` за счет так называемого обучения по батчам. `gensim` обучает модель LSA (`gensim.models.LsiModel`) на батчах документов и объединяет результаты для этих батчей инкрементным образом. Все модели `gensim` спроектированы таким образом, что используют *постоянное* количество оперативной памяти, благодаря чему быстрее работают на больших наборах данных, поскольку не требуют свопинга данных на диск и эффективно используют драгоценный кэш CPU.

ПРИМЕЧАНИЕ

> Обучение моделей gensim не только требует неизменного количества оперативной памяти, но и допускает распараллеливание, по крайней мере для многих из «долгоиграющих» шагов их конвейеров.

Итак, пакеты вроде `gensim` отнюдь не будут лишними в вашем арсенале. С их помощью можно ускорить эксперименты на маленьких наборах данных (как в этой книге), а также сделать возможным путешествие в гиперпространство больших данных в будущем.

---

[1] См. веб-страницу gensim: models.lsimodel — Latent Semantic Indexing по адресу https://radimrehurek.com/gensim/models/lsimodel.html.

13.3.2. Вычисления на графах

Hadoop, TensorFlow, Caffe, Theano, Torch и Spark — все были изначально спроектированы так, чтобы использовать постоянное количество оперативной памяти. Если вашу задачу машинного обучения можно сформулировать в виде задачи отображения-свертки или общего графа вычислений и вы хотите избежать опасности выхода за пределы доступной оперативной памяти, можете воспользоваться этими фреймворками. Они автоматически обходят граф вычислений, выделяя ресурсы и оптимизируя пропускную способность.

Питер Голдсборо (Peter Goldsborough) реализовал несколько наборов данных и моделей оценки производительности с помощью этих фреймворков для сравнения их быстродействия. И хотя фреймворк Torch существует с 2002 года, он показал неплохие результаты в большинстве тестов, обойдя все остальные при работе на обычных процессорах, а иногда и на графических процессорах. В большинстве случаев он был в десять раз быстрее ближайшего конкурента.

Кроме того, Torch (и его API Python PyTorch) интегрирован во многие фреймворки кластерных вычислений, например RocketML. Хотя для примеров в этой книге PyTorch не использовался (чтобы не слишком загружать читателей возможными вариантами), есть смысл обратиться к нему, если узкими местами NLP конвейера являются RAM или пропускная способность.

Нам удалось успешно распараллеливать выполнение конвейеров NLP с помощью RocketML (rocketml.net). Его создатели внесли значительный вклад в виде исследований и разработки в распараллеливание конвейеров NLP для Aira и TotalGood, чтобы помочь слепым и слабовидящим:

- выделять изображения из видео;
- производить логический вывод и вложения на основе предобученных моделей Caffe, PyTorch, Theano и TensorFlow (Keras);
- производить SVD на больших матрицах TF-IDF, охватывающих гигабайтные корпуса[1].

Конвейеры RocketML отлично масштабируются, зачастую линейным образом, в зависимости от алгоритма[2]. Так что, если удвоить количество машин в кластере, можно получить обученную модель вдвое быстрее. Добиться этого не так просто, как кажется. Большинство универсальных фреймворков распараллеливания графов вычислений, например PySpark и TensorFlow, едва ли могут таким похвастаться.

---

[1] На конференции SAIS 2008 Санти Адавани (Santi Adavani) рассказал о своих оптимизациях, позволивших выполнять SVD быстрее и с лучшей масштабируемостью на платформе RocketML HPC (http://www.databricks.com/speaker/santi-adavani).

[2] Санти Адавани и Винай Рао (Vinay Rao) вносят значительный вклад в проект, посвященный описаниям видео в режиме реального времени (Real-Time Video Description, https://github.com/totalgood/viddesc).

13.4. Распараллеливание вычислений NLP

Существует два основных подхода к высокопроизводительным вычислениям для NLP. Можно или добавить на сервер (в некоторых случаях даже на ноутбук) GPU, или связать вместе обычные процессоры с нескольких серверов.

13.4.1. Обучение моделей NLP на GPU

GPU стали важным и иногда совершенно необходимым инструментом для разработки промышленных NLP-приложений. GPU, впервые появившиеся в 2007 году, специально спроектированы для распараллеливания большого числа вычислительных задач и обращения к большим массивам памяти. CPU же, ядра любых компьютеров, устроены иначе. Они предназначены для высокоскоростной последовательной обработки задач и могут очень быстро обращаться к своей ограниченной памяти (рис. 13.1).

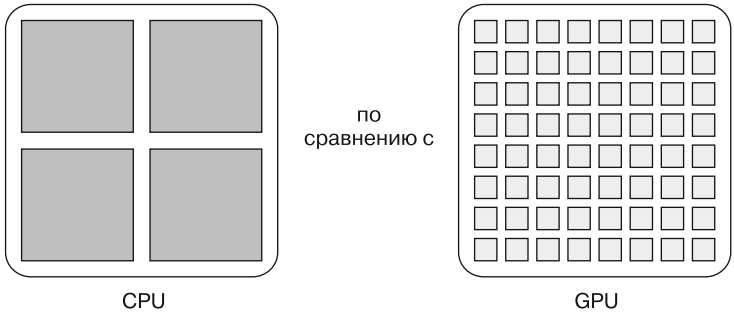


Рис. 13.1. Сравнение CPU и GPU

Оказывается, что обучение моделей глубокого обучения включает различные легко распараллеливаемые операции, например умножение матриц. Как и обработку динамических графических изображений, на которую изначально были рассчитаны GPU, обучение моделей глубокого обучения можно существенно ускорить за счет распараллеливания умножения матриц.

Рисунок 13.2 демонстрирует умножение входного вектора на матрицу весов, операцию, которая встречается очень часто во время прямого прохода обучения нейронной сети. Отдельные ядра GPU работают медленно по сравнению с CPU, но каждое из ядер может вычислять одну из компонент итогового вектора. При выполнении вычислений на CPU умножение отдельных строк производится последовательно, конечно, если не применяется какая-либо специализированная библиотека линейной алгебры. Для завершения умножения потребуется n (число строк матрицы) временны́х шагов. При выполнении же аналогичной задачи на GPU, умножение распараллеливается, и умножения всех строк происходят одновременно на отдельных ядрах GPU.

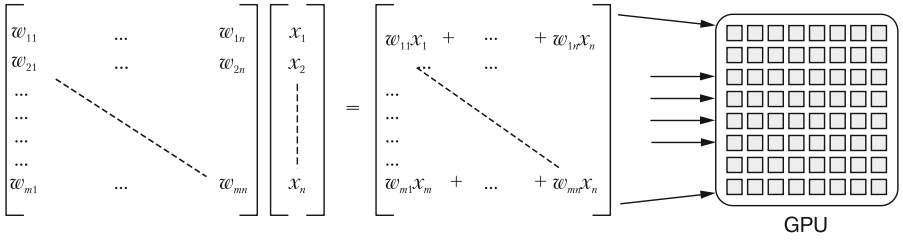


Рис. 13.2. Матричное умножение, при котором можно распараллелить умножение отдельных строк на GPU

ДОЛЖНА ЛИ МОДЕЛЬ ВЫПОЛНЯТЬСЯ НА GPU ПО ЗАВЕРШЕНИИ ОБУЧЕНИЯ?

Нет, использовать GPU для выполнения вывода на основе моделей при промышленной эксплуатации не обязательно, даже если GPU использовался для обучения модели. На самом деле, если не нужно выполнять прямые проходы (вывод или активацию нейронной сети) предобученной модели на миллионной выборке или при высокой пропускной способности (потоковая обработка в режиме реального времени), лучше использовать GPU только при обучении новой модели. Обратное распространение ошибки — намного более затратная вычислительно операция, чем прямая активация (вывод) с помощью нейронной сети.

GPU повышают сложность конвейера и затраты на него. Но подобные начальные вложения быстро окупаются за счет ускорения цикла обработки для моделей. Если повторное обучение модели с новыми гиперпараметрами занимает вдесятеро меньше времени, то можно попробовать в десять раз больше различных подходов и достичь гораздо более высокой точности.

По завершении обучения Keras или фреймворк глубокого обучения позволяют экспортировать веса и структуру модели. Затем можно загрузить эти веса и структуру модели практически на любом аппаратном обеспечении для вычисления предсказаний модели (прямой проход, проход вывода), даже на смартфоне[1] или в браузере[2].

13.4.2. Арендовать или покупать?

Использование графических процессоров может ускорить разработку модели и повысить скорость итераций. GPU полезны, но имеет ли смысл покупать свой собственный?

---

[1] См. документацию по фреймворку Core ML от Apple (https://developer.apple.com/documentation/coreml) или документацию по фреймворку TensorFlow Lite от Google (https://www.tensorflow.org/mobile/tflite/).

[2] См. веб-страницу Keras.js — Run Keras models in the browser по адресу https://transcranial.github.io/keras-js/#/.

В большинстве случаев ответ на этот вопрос: нет. Производительность GPU растет столь быстро, что купленная вами графическая карта очень скоро устареет. И если вы не планируете использовать GPU круглосуточно, лучше арендовать его на одном из таких сервисов, как Amazon Web Services или Google Cloud. Сервис GPU позволяет менять размеры экземпляров между запусками модели. Таким образом, можно масштабировать размер GPU в любую нужную сторону, в зависимости от ваших нужд. Упомянутые провайдеры обычно предоставляют полностью настроенные экземпляры, что экономит разработчику время и позволяет полностью сосредоточить свое внимание на разработке модели.

Мы создали и поддерживали свой собственный сервер GPU для ускорения обучения некоторых из представленных в этой книге моделей, но вам лучше сделать так, как мы сказали, а не так, как мы сделали. Подбор совместимых друг с другом компонентов и минимизация узких мест в смысле пропускной способности по данным — весьма непростая задача. Мы подражали успешно работающим архитектурам, описанным другими исследователями, и купили RAM и GPU еще до недавнего скачка курса биткоина и роста цен на высокопроизводительные вычислительные компоненты (HPC). Поддерживать актуальность всех библиотек и согласовывать их использование и настройки между авторами данной книги было непросто. Это было интересно и поучительно, но вряд ли можно это назвать эффективным вложением времени и денег.

У гибкой архитектуры арендованных экземпляров GPU есть один недостаток: приходится внимательно отслеживать денежные траты. Завершение обучения не означает автоматический останов экземпляра. Чтобы счетчик перестал крутиться (а значит, и текущие расходы перестали увеличиваться), следует отключать экземпляр GPU между отдельными прогонами обучения. Подробную информацию можно найти в подразделе «Финасы» раздела «Источники информации» в конце данной книги.

13.4.3. Варианты аренды GPU

Различные варианты аренды GPU предоставляют несколько компаний, начиная с вариантов «платформа как сервис» от таких известных компаний, как Microsoft, Amazon (Web Services) и Google. Интересные варианты, с помощью которых можно быстро начать работать с проектом глубокого обучения, также предоставляют недавно ворвавшиеся на этот рынок стартапы, такие как Paperspace и FloydHub.

В табл. 13.1 приведено сравнение различных вариантов GPU от провайдеров платформ как сервисов. Эти варианты начинаются от чистой машины с GPU с минимальной конфигурацией до полностью настроенных машин с перетаскиваемыми клиентами. Поскольку в разных регионах цены сервисов различаются, мы не будем сравнивать провайдеров по ценам. Цены на сервисы варьируются от $0,65 до нескольких долларов за час работы одного экземпляра, в зависимости от местоположения, настроек и архитектуры сервера.

Таблица 13.1. Сравнение вариантов GPU «платформа как сервис»

Компания	Преимущества	Варианты GPU	Насколько просто начать использовать	Гибкость
Amazon Web Services (AWS)	Широкий диапазон вариантов GPU; цены за спотовый виртуальный узел; доступны в различных центрах данных по всему миру	NVIDIA GRID K520, Tesla M60, Tesla K80, Tesla V100	Не очень сложно	Высокая
Google Cloud	Интеграция Google Cloud Kubernetes, DialogFlow, Jupyter (colab.research.google.com/notebook)	NVIDIA Tesla K80, Tesla P100	Не очень сложно	Высокая
Microsoft Azure	Хороший вариант, если вы используете другие сервисы Azure	NVIDIA Tesla K80	Не очень сложно	Высокая
FloydHub	Интерфейс командной строки для компоновки вашего кода	NVIDIA Tesla K80, Tesla V100	Просто	Средняя
Paperspace	Виртуальные серверы и хостируемые блокноты iPython/Jupyter с поддержкой GPU	NVIDIA Maxwell, Tesla P5000, Tesla P6000, Tesla V100	Просто	Средняя

НАСТРОЙКА СВОЕГО GPU НА AWS

В приложении Д вкратце описаны необходимые шаги для начала работы со своим собственным экземпляром GPU.

13.4.4. Тензорные процессоры

Наверное, вы слышали еще одну аббревиатуру, TPU (tensor processing unit, тензорный процессор), которая обозначает высокоспециализированный процессор для глубокого обучения. Особенно эффективно эти процессоры вычисляют обратное распространение ошибки для моделей TensorFlow. TPU оптимизированы для умножения тензоров и передачи данных. GPU оптимизированы для обработки графических данных, состоящей в основном из умножения двумерных матриц, необходимого для визуализации и перемещения в мирах 3D-игр.

Google утверждает, что TPU на порядок эффективнее при расчетах глубокого обучения, чем эквивалентные GPU. На момент написания данной книги компания Google, которая изобрела и спроектировала TPU в 2015 году, выпустила для широкой публики лишь бета-версию (без какого-либо соглашения об уровне предоставления услуги). Кроме того, исследователи могут подавать заявки на участие в TensorFlow Research Cloud (https://www.tensorflow.org/tfrc/) для обучения их моделей на TPU.

13.5. Сокращение объема потребляемой памяти при обучении модели

При обучении моделей NLP на GPU с большим корпусом велика вероятность столкнуться в конце концов во время обучения со следующей ошибкой: MemoryError (листинг 13.12).

Листинг 13.12. Сообщение об ошибке, возникающее, если тренировочные данные превышают объем памяти GPU

```
Epoch 1/10
Exception in thread Thread-27:
Traceback (most recent call last):
  File "/usr/lib/python2.7/threading.py", line 801, in __bootstrap_inner
    self.run()
  File "/usr/lib/python2.7/threading.py", line 754, in run
    self.__target(*self.__args, **self.__kwargs)
  File "/usr/local/lib/python2.7/dist-packages/keras/engine/training.py",
    line 606, in data_generator_task
    generator_output = next(self._generator)
  File "/home/ubuntu/django/project/model/load_data.py", line 54,
    in load_training_set
    rv = np.array(rv)
MemoryError
```

Чтобы добиться высокой производительности, GPU используют свою собственную внутреннюю память, помимо памяти CPU. Память графической карты обычно ограничена несколькими гигабайтами и в большинстве случаев гораздо меньше объема памяти, доступной CPU. При обучении модели на CPU тренировочные данные обычно загружаются в память компьютера в виде одной большой таблицы или последовательности тензоров. При ограничениях на память GPU это невозможно (рис. 13.3).

Рис. 13.3. Загрузка тренировочных данных без функции-генератора

Один из возможных обходных путей для решения этой проблемы — использование понятия *генератора* языка Python, функции, возвращающей объект-итератор. При передаче объекта-итератора методу обучения модели он извлекает на каждом шаге цикла обучения один или несколько тренировочных примеров данных. У этого эффективного метода понижения объема потребляемой памяти есть и недостатки:

- генераторы могут выдавать только по одному элементу за раз, так что количество содержащихся в нем элементов становится известно только в конце;
- генераторы можно выполнять только один раз. Они являются одноразовыми и повторному использованию не подлежат.

Эти два нюанса существенно затрудняют выполнение нескольких проходов обучения по данным. Но на выручку нам приходит Keras, включающий методы, которые берут на себя всю эту утомительную «бухгалтерию» (рис. 13.4).

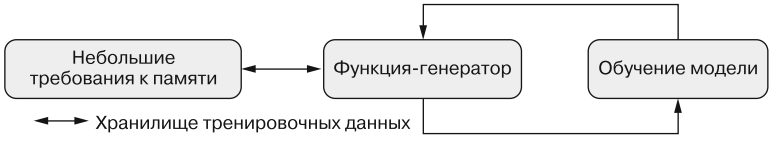


Рис. 13.4. Загрузка тренировочных данных с помощью функции-генератора

Функция-генератор берет на себя загрузку хранилища тренировочных данных и возвращает тренировочные «порции» производящим обучение методам. В листинге 13.13 хранилище тренировочных данных представляет собой CSV-файл с входными данными, отделенными от ожидаемых выходных данных разделителем |. Порции ограничиваются размерами батчей, причем только один батч хранится в памяти одновременно. Благодаря этому можно существенно сократить объем памяти, потребляемой тренировочным набором данных модели.

Листинг 13.13. Применение генератора для повышения эффективности использования RAM

```
>>> import numpy as np
>>>
>>> def training_set_generator(data_store,
...                             batch_size=32):
...     X, Y = [], []
...     while True:
...         with open(data_store) as f:
...             for i, line in enumerate(f):
...                 if i % batch_size == 0 and X and Y:
...                     yield np.array(X), np.array(Y)
...                     X, Y = [], []
...                 x, y = line.split('|')
...                 X.append(x)
...                 Y.append(y)
>>>
>>> data_store = '/path/to/your/data.csv'
>>> training_set = training_set_generator(data_store)
```

В настройках функции можно задавать размер батча динамически

Этот бесконечный цикл выдает тренировочные наборы данных постоянно; по окончании эпохи Keras прекращает запрашивать новые обучающие примеры

Открываем хранилище обучающих данных и создаем дескриптор файла f

Проходим в цикле по содержимому хранилища обучающих данных строка за строкой, пока все данные не сыграют роль обучающих примеров; после этого начинаем с начала тренировочного набора данных

Если примеров данных недостаточно, читаем дополнительные строки, разбивая их по разделителю |, и сохраняем в списках X и Y

Когда наберется достаточно обучающих примеров данных, возвращаем обучающие данные и ожидаемый выходной сигнал обучения с помощью функции yield. Python возвращается обратно после оператора yield, после того как данные переданы в метод fit модели

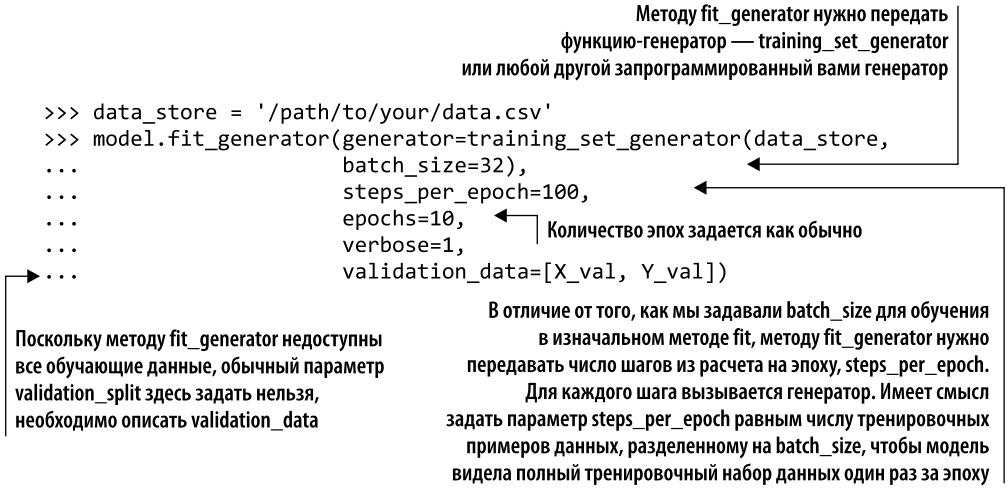
В нашем примере функция `training_set_generator` читает из файлов значений с разделителем |, но точно так же можно загружать данные из любой базы данных или любой другой системы хранения данных.

Один из недостатков генератора — он не возвращает никакой информации о размере массива тренировочных данных. Поскольку количество доступных тренировочных данных неизвестно, то приходится использовать немного другие версии методов `fit`, `predict` и `evaluate` модели Keras.

Вместо того чтобы обучать модель с помощью:

```
>>> model.fit(x=X,
...           y=Y,
...           batch_size=32,
...           epochs=10,
...           verbose=1,
...           validation_split=0.2)
```

приходится запускать обучение модели с помощью:



При использовании генератора не помешает также заменить вызовы методов `evaluate` и `predict` на:

```
>>> model.evaluate_generator(generator=your_eval_generator(eval_data,
...     batch_size=32), steps=10)
```

и

```
>>> model.predict_generator(generator=your_predict_generator(\
...     prediction_data, batch_size=32), steps=10)
```

ПРЕДОСТЕРЕЖЕНИЕ

Генераторы эффективно используют память, но могут и оказаться узким местом при обучении модели, замедляя выполнение шагов цикла обучения. Обращайте особое внимание на быстродействие генератора при разработке функций обучения. Если обработка на лету приводит к замедлению генератора, имеет смысл провести предварительную обработку тренировочных данных либо арендовать экземпляр с большим объемом оперативной памяти, а может, и то и другое.

13.6. Как почерпнуть полезную информацию о модели с помощью TensorBoard

Как думаете, было бы неплохо получить полезную информацию о работе модели во время ее обучения и сравнить с предыдущими запусками обучения? Или быстро построить график вложений слов для проверки семантических подобий? Утилита TensorBoard от Google предназначена именно для этого.

При обучении модели с помощью TensorBoard (или Keras с прикладной частью TF) можно с помощью TensorBoard почерпнуть полезную внутреннюю информацию о моделях NLP. Ее можно использовать для отслеживания метрик обучения модели, построения сетевых графиков распределений весов, визуализации вложений слов и т. д. TensorBoard проста в использовании и подключается к экземпляру, на котором происходит обучение, через браузер.

Для применения TensorBoard вместе с Keras необходимо инсталлировать TensorBoard, точно так же, как любой другой пакет Python:

```
pip install tensorboard
```

По завершении установки можно запустить ее:

```
tensorboard --logdir=/tmp/
```

Запущенная утилита TensorBoard доступна в браузере на `localhost`, порт 6006 (`http://127.0.0.1:6006`), с ноутбука или стационарного компьютера. При обучении модели на арендованном экземпляре GPU следует обращаться по общему IP-адресу этого экземпляра GPU, предварительно убедившись, что у провайдера GPU открыт доступ через порт 6006.

После входа можно начинать детальный анализ производительности модели.

13.6.1. Визуализация вложений слов

TensorBoard — великолепная утилита для визуализации вложений слов. Особенно полезна она для верификации семантического подобия при обучении своих собственных, ориентированных на конкретную предметную область вложений слов. Преобразование модели слов в формат TensorBoard позволяет сделать это без особых сложностей. После загрузки векторов слов и меток в утилиту TensorBoard она производит понижение размерности до 2D или 3D. В настоящее время TensorBoard предлагает три метода понижения размерности: PCA, t-SNE и пользовательские варианты.

В листинге 13.14 показано, как преобразовать вложения слов в формат TensorBoard и сгенерировать данные проекций.

Листинг 13.14. Преобразование вложения в проекцию TensorBoard

```
>>> import os
>>> import tensorflow as tf
>>> import numpy as np
>>> from io import open
```

```
>>> from tensorflow.contrib.tensorboard.plugins import projector
>>>
>>>
>>> def create_projection(projection_data,
...                       projection_name='tensorboard_viz',
...                       path='/tmp/'):
...     meta_file = "{}.tsv".format(projection_name)
...     vector_dim = len(projection_data[0][1])
...     samples = len(projection_data)
...     projection_matrix = np.zeros((samples, vector_dim))
...
...     with open(os.path.join(path, meta_file), 'w') as file_metadata:
...         for i, row in enumerate(projection_data):
...             label, vector = row[0], row[1]
...             projection_matrix[i] = np.array(vector)
...             file_metadata.write("{}\n".format(label))
...
...     sess = tf.InteractiveSession()
...
...     embedding = tf.Variable(projection_matrix,
...                             trainable=False,
...                             name=projection_name)
...     tf.global_variables_initializer().run()
...
...     saver = tf.train.Saver()
...     writer = tf.summary.FileWriter(path, sess.graph)
...
...     config = projector.ProjectorConfig()
...     embed = config.embeddings.add()
...     embed.tensor_name = '{}'.format(projection_name)
...     embed.metadata_path = os.path.join(path, meta_file)
...
...     projector.visualize_embeddings(writer, config)
...     saver.save(sess, os.path.join(path, '{}.ckpt'\
...         .format(projection_name)))
...     print('Run `tensorboard --logdir={0}` to run\
...           visualize result on tensorboard'.format(path))
```

Функция create_projection принимает на входе три аргумента: данные вложений, название проекции и путь, где сохранять файлы проекций

Наша функция проходит в цикле по данным вложений и создает массив NumPy, далее преобразуемый в переменную TensorFlow

Для создания проекции TensorBoard необходимо создать сеанс TensorFlow

TensorFlow предоставляет встроенные методы для создания проекций

Метод visualize_embeddings записывает конфигурацию проекции по заданному пути, где ее потом сможет прочитать TensorBoard

Функция `create_projection` принимает на входе список кортежей (сначала вектор, затем метка) и преобразует его в файлы проекций TensorBoard. После того как вы создали файлы проекций и они стали доступны TensorBoard (в нашем случае TensorBoard ожидает, что эти файлы находятся в каталоге `tmp`), перейдите в браузере на TensorBoard и посмотрите на визуализацию вложений (рис. 13.5).

```
>>> projection_name = "NLP_in_Action"
>>> projection_data = [
>>>     ('car', [0.34, ..., -0.72]),
>>>     ...
>>>     ('toy', [0.46, ..., 0.39]),
>>> ]
>>> create_projection(projection_data, projection_name)
```

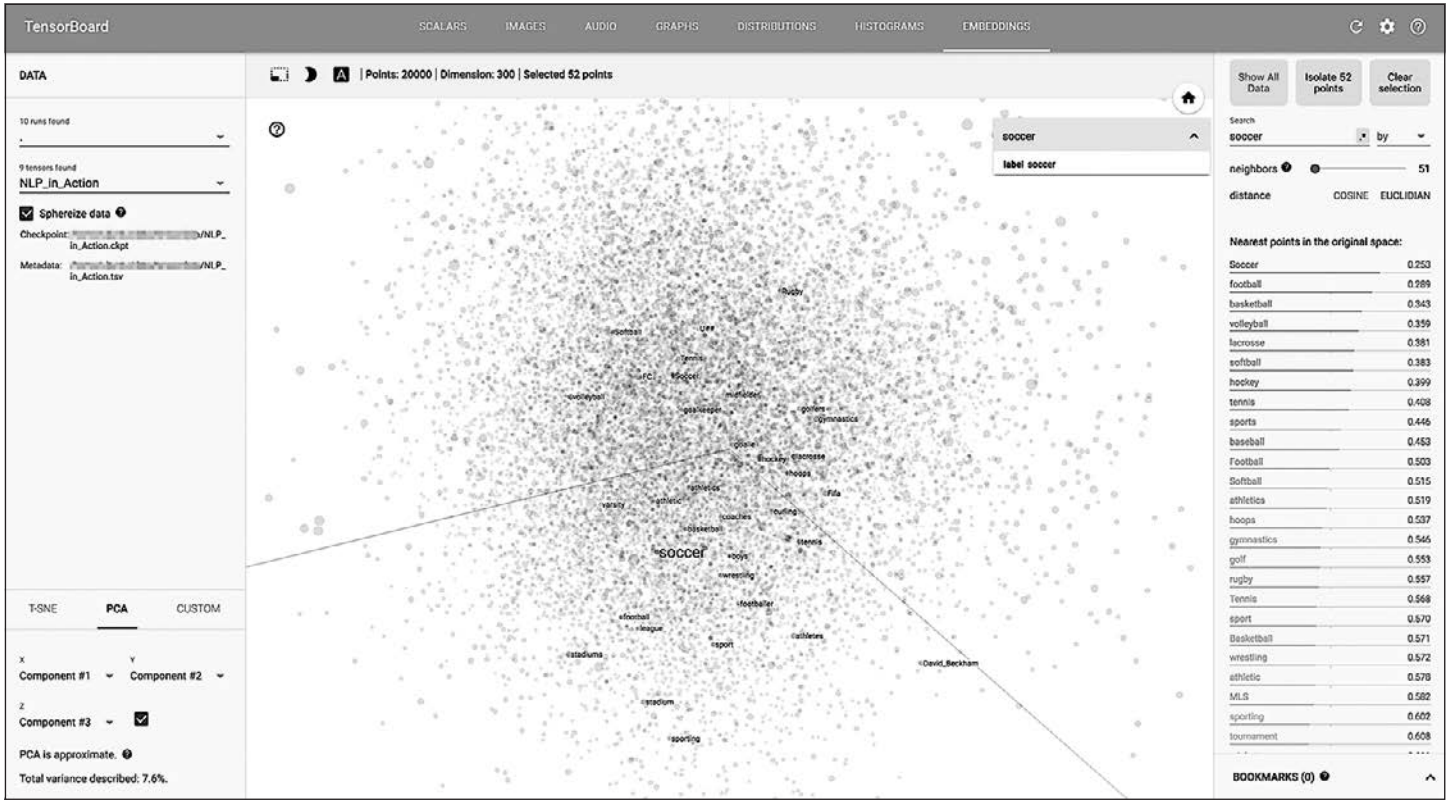


Рис. 13.5. Визуализация вложений word2vec с помощью TensorBoard.

Резюме

- Методы хеширования с учетом локальности, например Annoy, обещают превратить латентно-семантическую индексацию в реальность.
- Использование GPU ускоряет обучение модели, снижая длительность цикла обработки, благодаря чему можно создавать лучшие модели быстрее.
- Распараллеливание CPU имеет смысл для алгоритмов, для которых бесполезно ускорение умножения больших матриц.
- Благодаря генераторам Python можно обойти узкое место, связанное с системной RAM, и сэкономить деньги на GPU и CPU.
- TensorBoard от компании Google помогает визуализировать и выделять вложения естественного языка, которые иначе могли бы ускользнуть от вашего внимания.
- Благодаря распараллеливанию NLP можно расширить интеллектуальные возможности системы за счет *совокупности интеллектов* — кластеров машин, помогающих вам мыслить[1].

[1] *Watts P.* Conscious Ants and Human Hives (https://youtube/v4uwaw_5Q3I?t=45s).

Приложения

A *Инструменты для работы с NLP*

Если вы установите пакет `nlpia` (http://github.com/totalgood/nlpia), то сможете выполнить все примеры из этой книги. В файле README вы найдете инструкции, которые мы поддерживаем в актуальном состоянии. Но если Python 3 у вас уже установлен и вы готовы рискнуть (или вам повезло и у вас установлен Linux), можете попытаться выполнить команды:

```
$ git clone https://github.com/totalgood/nlpia
$ pip3 install -e nlpia
```

Если этот вариант в вашем случае не работает, вероятно, вам придется установить систему управления пакетами и несколько бинарных пакетов для своей операционной системы. Далее приведены инструкции для трех операционных систем:

- Ubuntu;
- Mac;
- Windows.

В этих разделах рассказано, как установить систему управления пакетами для соответствующей операционной системы. После установки системы управления пакетами (или если вы работаете в дружественной разработчику операционной системе вроде Ubuntu, в которой она уже установлена) вы можете установить Anaconda3.

A.1. Anaconda3

В Python 3 есть множество повышающих производительность и выразительность кода возможностей, удобных для NLP. Простейший способ установки Python практически на любой системе — установить Anaconda3 (https://www.anaconda.com/download). В качестве дополнительного преимущества при подобном способе вы

получаете систему управления пакетами и средой, с помощью которой можно установить множество сложных в установке пакетов (например, `matplotlib`) на вызывающих затруднения операционных системах (например, Windows).

Установить последнюю версию Anaconda[1] и ее систему управления пакетами можно программным образом с помощью выполнения кода из листинга А.1.

Листинг А.1. Установка Anaconda3

```
$ OS=MacOSX # Или Linux, или Windows
$ BITS=_64 # or '' for 32-bit
$ curl https://repo.anaconda.com/archive/ > tmp.html
$ FILENAME=$(grep -o -E -e "Anaconda3-[.0-9]+-$OS-
x86$BITS\.(sh|exe)" tmp.html | head -n 1)
$ curl "https://repo.anaconda.com/archive/$FILENAME" > install_anaconda
$ chmod +x install_anaconda
$ ./install_anaconda -b -p ~/Anaconda
$ export PATH="$HOME/Anaconda/bin:$PATH"
$ echo 'export PATH="$HOME/Anaconda/bin:$PATH"' >> ~/.bashrc
$ echo 'export PATH="$HOME/Anaconda/bin:$PATH"' >> ~/.bash_profile
$ source ~/.bash_profile
$ rm install_anaconda
```

Теперь можно создать виртуальную среду: не `virtualenv` Python, а более полную среду `conda`, изолирующую все бинарные зависимости Python от среды Python операционной системы. После этого можно установить зависимости и исходный код `nlpia` в этой виртуальной среде с помощью кода из листинга А.2.

А.2. Установка пакета nlpia

Мы предпочитаем устанавливать исходный код программного обеспечения, над которым работаем, в подкаталог `code/` каталога `$HOME` соответствующего пользователя, но вы можете разместить его там, где вам удобно. Если не получается, посмотрите в файл README пакета `nlpia` (https://github.com/totalgood/nlpia) за актуальными инструкциями по установке.

Листинг А.2. Установка кода nlpia с помощью conda

```
$ mkdir -p ~/code
$ cd ~/code
$ git clone https://github.com/totalgood/nlpia
$ cd ~/code/nlpia
$ conda install -y pip
$ pip install --upgrade pip
$ conda env create -n nlpiaenv -f conda/environment.yml
$ source activate nlpiaenv
$ pip install --upgrade pip
$ pip install -e .
```

Установка свежих бинарных файлов conda для pip в корневую среду conda

Обновление pip до последней версии с pypi.python.org — pip устанавливает pip ;)

Создаем среду conda, каталог в "$HOME/Anaconda3/envs/nlpia" с бинарными и исходными файлами зависимостей

Активируем среду Python

Устанавливаем свежий pip в среде nlpiaenv

Устанавливаем редактируемый каталог исходного кода для nlpia, чтобы сохранять изменения исходного кода и данных на диск

[1] На момент перевода данной книги это версия 2019.10. — *Примеч. пер.*

A.3. IDE

После установки на машину Python 3 и `nlpia` для полноты интегрированной среды разработки (IDE) вам понадобится хороший текстовый редактор. Вместо больших комплексных систем вроде PyCharm от JetBrains мы предпочитаем использовать отдельные утилиты, разработанные маленькими командами разработчиков (в случае Sublime Text — вообще из одного разработчика), которые делают что-то одно, но хорошо.

СОВЕТ

> «Созданное разработчиками для разработчиков» — вполне реальная вещь, особенно если команда разработчиков состоит из одного человека. Разработчики-одиночки, как правило, создают лучшие утилиты, чем корпорации, будучи более открытыми для использования кода и рекомендаций пользователей. Разработчик-одиночка, создающий нужную ему утилиту, обычно оптимизирует ее под свой процесс разработки. Его процессы разработки весьма впечатляют, раз он создает надежные, мощные и конкурентоспособные утилиты. Большие проекты с открытым исходным кодом, такие как jupiter, также впечатляют, но по-другому. Они обычно отличаются исключительной универсальностью и набором возможностей, если, конечно, у такого проекта с открытым исходным кодом нет коммерческого ответвления.

К счастью, все необходимые для нашего IDE Python утилиты бесплатны, расширяемы и постоянно поддерживаются разработчиками. Большинство из них — с открытым исходным кодом, так что вы можете собрать свою версию:

- Sublime Text 3 (www.sublimetext.com/3) — текстовый редактор с Package Control (https://packagecontrol.io/installation#st3), а также линтером и автокорректором Anaconda (https://packagecontrol.io/packages/Anaconda);
- `Meld` — утилита слияния кода для Mac (https://yousseb.github.io/meld) и других операционных систем (http://meldmerge.org);
- `ipython` (`jupyter console`) — для вашего цикла разработки «чтение — вычисление — вывод» (**R**ead → **E**val → **P**rint → **L**oop, REPL);
- `jupyter notebook` для создания отчетов, учебных руководств и сообщений в блоге или для демонстрации результатов работы начальнику.

СОВЕТ

> Некоторые высокопроизводительные разработчики применяют процесс REPL в Python[1]. Особенно большими возможностями обладают консоли REPL ipython, jupyter console и jupyter notebook с их магическими командами help, ?, ?? и %, а также автодополнением табуляцией атрибутов, методов, аргу-

[1] Да, я говорю о вас, Стивен «Цифровой кочевник» Скочен и Алек «Чувак» Ландграф.

ментов, файловых путей и даже ключей dict. Прежде чем искать в Google или переполнять свой стек разработчика, загляните в docstring и исходный код импортируемых пакетов Python с помощью команд вида >>> sklearn.linear_model.BayesianRidge??. Процесс REPL языка Python дает даже возможность выполнения команд командной оболочки (попробуйте выполнить >>> !git pull или >>> !find . -name nlpia), чтобы разработчику не приходилось отрывать пальцы от клавиатуры, минимизируя таким образом число переключений контекста и максимизируя производительность.

A.4. Система управления пакетами Ubuntu

В вашем дистрибутиве Linux уже установлена полнофункциональная система управления пакетами. Но вы можете обойтись и без нее, если воспользуетесь системой управления пакетами `conda` из дистрибутива Anaconda, как рекомендуется в инструкции по установке `nlpia` (https://github.com/totalgood/nlpia). Система управления пакетами операционной системы Ubuntu называется `apt`. В разделе A.3 мы советовали вам некоторые пакеты для установки. Почти наверняка вам понадобятся не все из них, но мы приведем исчерпывающий список утилит на случай, если что-либо из установленного вами с Anaconda начнет жаловаться на нехватку бинарного файла. Можете идти по списку сверху вниз до тех пор, пока `conda` не сможет установить нужные вам пакеты Python (листинг A.3).

Листинг A.3. Установка инструментов разработчика с помощью apt

```
$ sudo apt-get update
$ sudo apt install -y build-essential libssl-dev g++ cmake swig git
$ sudo apt install -y python2.7-dev python3.5-dev libopenblas-dev libatlas-
    base-dev gfortran libgtk-3-dev
$ sudo apt install -y openjdk-8-jdk python-dev python-numpy python-
    pip python-virtualenv python-wheel python-nose
$ sudo apt install -y python3-dev python3-wheel python3-numpy python-
scipy python-dev python-pip python3-six python3-pip
$ sudo apt install -y python3-pyaudio python-pyaudio
$ sudo apt install -y libcurl3-dev libcupti-dev xauth x11-apps python-qt4
$ sudo apt install -y python-opencv-dev libxvidcore-dev libx264-dev libjpeg8-
    dev libtiff5-dev libjasper-dev libpng12-dev
```

СОВЕТ

Если команда apt-get update завершилась неудачно и вернула ошибку, в которой упоминается система сборки bazel, значит, вы добавили репозиторий apt Google с их системой сборки для TensorFlow. Следующие команды помогут вам вернуться на путь истинный:

```
$ sudo apt-get install curl
$ curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -
```

A.5. Mac

Чтобы установить все утилиты, без которых вы не сможете быть на равных с другими разработчиками, вам понадобится настоящая система управления пакетами (а не XCode).

A.5.1. Система управления пакетами для Macintosh

Homebrew (https://brew.sh) — вероятно, самая популярная среди разработчиков система управления пакетами для Macintosh. Ее легко установить, и она содержит устанавливаемые за один шаг пакеты для большинства используемых разработчиками утилит. Она эквивалентна системе управления пакетами `apt` операционной системы Ubuntu. Компания Apple могла сделать так, чтобы их операционная система работала с `apt`, но они не хотели давать разработчикам возможность обойти XCode и App Store по коммерческим соображениям. Поэтому несколько бесстрашных Ruby-разработчиков создали свою собственную систему управления пакетами[1]. И она не хуже `apt` или любой другой системы управления пакетами, ориентированной на конкретную операционную систему (листинг A.4).

Листинг A.4. Установка Homebrew

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/master/install)"
```

Вам понадобится подтвердить некоторые действия нажатием клавиши Return, а также ввести пароль суперпользователя. Так что не уходите за кофе, пока не ввели пароль и сценарий установки не начал делать свою работу.

A.5.2. Дополнительные пакеты

Возможно, после установки Homebrew вы захотите установить несколько удобных Linux-утилит, как показано в листинге A.5.

Листинг A.5. Установка инструментов разработчика

```
$ brew install wget htop tree pandoc asciidoctor
```

A.5.3. Настройки

Если вы относитесь к своим занятиям NLP и к разработке программного обеспечения серьезно, то вам не помешает убедиться в правильных настройках операционной системы. Вот какие утилиты мы устанавливаем при каждом создании новой учетной записи пользователя на Mac:

---

[1] См. статью о системе управления пакетами Homebrew по адресу https://ru.wikipedia.org/wiki/Homebrew_(менеджер_пакетов_в_Mac_OS).

- Snappy — для скриншотов экрана (http://snappy-app.com);
- CopyClip — для управления буфером обмена (https://itunes.apple.com/us/app/copyclip-clipboard-history-manager/id595191960).

Для демонстрации своего экрана другим NLP-разработчикам вам понадобится утилита для скриншота экрана, например Snappy. Утилита управления буфером обмена, например CopyClip, позволит вам копировать и вставлять несколько фрагментов за раз и не терять историю буфера обмена при перезагрузке. Система управления буфером обмена обеспечивает возможности поиска по истории консоли (`[ctrl]-[R]`) в GUI.

Не помешает также увеличить глубину истории командной оболочки bash, добавить несколько более безопасных в смысле `rm -f` псевдонимов, задать используемый по умолчанию редактор, сделать текст разноцветным и добавить команды `open` для браузера, текстового редактора и утилиты слияния кода, как показано в листинге А.6.

Листинг А.6. bash_profile

```
#!/usr/bin/env bash
echo "Running customized ~/.bash_profile script: '$0' ......."
export HISTFILESIZE=10000000
export HISTSIZE=10000000
# Добавление информации в конец файла истории команд после каждого сеанса
shopt -s histappend
# Разрешать корректировку завершившихся неудачно команд нажатием Ctrl+R
shopt -s histreedit
# Демонстрация подстановок команд пользователю перед выполнением
shopt -s histverify
# Хранение многострочных команд в одной записи истории
shopt -s cmdhist
# Проверка размера окна после каждой команды и при необходимости
# обновление значений LINES и COLUMNS
shopt -s checkwinsize
# Вывод разноцветных результатов команды grep
export GREP_OPTIONS='--color=always'
# Найденные командой grep совпадения выводятся жирным пурпурным шрифтом
export GREP_COLOR='1;35;40'
# Записывать все произведенные в командной оболочке действия в файл,
# который не окажется случайно очищен или усечен операционной системой
export PROMPT_COMMAND='echo "# cd $PWD" >> ~/
     .bash_history_forever; '$PROMPT_COMMAND
export PROMPT_COMMAND="history -a; history -c; history -r; history 1 >> ~/
     .bash_history_forever; $PROMPT_COMMAND"
# так что больше он не поменяется
readonly PROMPT_COMMAND
# Применение: subl http://google.com  # открывается в новой вкладке
if [ ! -f /usr/local/bin/firefox ]; then
    ln -s /Applications/Firefox.app/Contents/MacOS/firefox /usr/local/bin/
     firefox
fi
alias firefox='open -a Firefox'
# Применение: subl file.py
```

```
if [ ! -f /usr/local/bin/subl ]; then
    ln -s /Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl /
     usr/local/bin/subl
fi
# Применение: meld file1 file2 file3
if [ ! -f /usr/local/bin/meld ]; then
    ln -s /Applications/Meld.app/Contents/MacOS/Meld /usr/local/bin/meld
fi
export VISUAL='subl -w'
export EDITOR="$VISUAL"
# Можете воспользоваться -f, чтобы избавиться от интерактивных подтверждений
# деструктивных операций записи на диск
alias rm="rm -i"
alias mv="mv -i"
alias ..="cd .."
alias ...="cd ../.."
```

Другие сценарии `bash_profile` вы можете найти, выполнив поиск по GitHubGist (https://gist.github.com/search?q=%22.bash_profile%22+mac).

A.6. Windows

Утилиты командной строки для управления пакетами на Windows, например `cygwin`, не слишком хороши. Но вы можете установить на Windows-машине GitGUI и получить таким образом командную строку bash и удобный терминал для работы вашей консоли Python REPL.

1. Скачайте и установите `git` (https://git-scm.com/download/win).
2. Скачайте и установите GitHub Desktop (https://desktop.github.com/).

Установщик `git` включает версию командной оболочки bash, которая должна неплохо работать на Windows, но устанавливаемый им `git-gui` нельзя назвать удобным для пользователя, особенно для начинающего. Так что если вы не используете `git` из командной строки (командная оболочка bash в Windows), то лучше воспользуйтесь GitHub Desktop для всех своих операций с `git` (`push`/`pull`/`merge`). Мы сталкивались при написании данной книги с неожиданными действиями `git-gui`, вроде перезаписи зафиксированных другими разработчиками изменений в случае конфликта версий, даже в не участвовавших в конфликте файлах. Именно поэтому мы рекомендуем вам установить GitHub Desktop (https://desktop.github.com/) поверх чистых `git` и `git-bash`. GitHub Desktop более дружественен к пользователю, он сообщает вам о необходимости отправить/извлечь/слить какие-либо изменения[1].

При наличии работающей в терминале Windows командной оболочки можно установить Anaconda и воспользоваться системой управления пакетами `conda` для установки пакета `nlpia`, как и ранее, с помощью инструкций из файла README GitHub-репозитория (http://github.com/totalgood/nlpia).

[1] Огромное спасибо Бенджамину Бергу (Benjamin Berg) и Даррену Миссу (Darren Meiss), которые это выяснили, и за весь тяжелый труд, вложенный ими в приведение этой книги в подобающий вид.

A.6.1. Переходите в виртуальность

Если Windows вас разочаровала в смысле работы с Python, вы всегда можете установить VirtualBox или Docker и создать виртуальную машину с операционной системой Ubuntu. Это тема для отдельной книги (или по крайней мере главы) и есть лучшие, чем мы, специалисты в этой области:

- Джейсон Браунли (Jason Brownlee) (https://machinelearningmastery.com/linux-virtual-machine-machine-learning-development-python-3);
- Йерун Янсенс (Jeroen Janssens) (http://datasciencetoolbox.org);
- Вик Парачури (Vik Paruchuri) (www.dataquest.io/blog/docker-data-science);
- Джейми Холл (Jamie Hall) (http://blog.kaggle.com/2016/02/05/how-to-get-started-with-data-science-in-containers).

Еще один способ заполучить Linux в мире Windows — воспользоваться приложением командной оболочки Ubuntu от Microsoft. Мы его не использовали, так что не можем поручиться за его совместимость с нужными вам пакетами Python. Если вы его попробуете — поделитесь, пожалуйста, информацией с нами в репозитории `nlpia` с помощью запроса на новые функции (feature request) или включение изменений (pull request) в документацию. Форум издательства Manning (https://forums.manning.com/forums/natural-language-processing-in-action) — тоже отличное место для того, чтобы поделиться информацией и получить помощь.

A.7. Автоматизация в пакете nlpia

К счастью, в пакет `nlpia` включено несколько процедур автоматизации настройки среды, для скачивания моделей NLTK, Spacy, Word2vec и необходимых для этой книги данных. Скачивания запускаются при вызове любой из функций-адаптеров `nlpia`, например, `segment_sentences()`, для которой требуются какие-либо из этих наборов данных или моделей. Но работа над этим программным обеспечением продолжается, оно постоянно поддерживается и расширяется такими же читателями, как и вы. Возможно, вы захотите узнать, как установить эти пакеты и скачать данные вручную на случай, если автоматизация `nlpia` не сработает. А может, вас просто заинтересуют какие-либо из наборов данных, благодаря которым возможны синтаксический разбор предложений и частеречная разметка. Так что, если вы хотите настроить свою среду под свои потребности, в следующих приложениях к данной книге рассказывается, как установить и настроить отдельные элементы полнофункциональной среды разработки NLP.

Б *Эксперименты с Python и регулярные выражения*

Чтобы извлечь максимум из этой книги, вам нужно как следует освоить язык Python. Освоить до такой степени, чтобы свободно экспериментировать с ним. Когда что-то не работает, необходимо экспериментировать и исследовать, чтобы выяснить, как заставить Python выполнить требуемое.

И даже когда код работает, благодаря экспериментам можно найти новые интересные способы выполнения различных действий и обнаружить скрытые ошибки в коде. Скрытые ошибки и граничные случаи очень часто встречаются при обработке естественного языка, поскольку в языках вроде английского существует множество способов выразить одно и то же.

Для этого достаточно просто свободно экспериментировать с кодом, как это делают дети. Если вы копируете и вставляете код — поменяйте его. Попытайтесь нарушить его работу, а затем все почините. Расчленяйте его на как можно большее количество отдельных выражений. Производите модульную организацию фрагментов своего кода с помощью функций или классов, а затем собирайте их воедино, как можно лаконичнее.

Производите произвольные эксперименты с создаваемыми вами структурами данных, моделями и функциями. Запускайте команды, которые имеет смысл, по вашему мнению, включить в модуль или класс. Чаще используйте клавишу Tab на своей клавиатуре. При нажатии клавиши Tab редактор или командная оболочка пытается закончить вашу мысль путем автодополнения названий переменных, классов, функций, методов, атрибутов и путей, которые вы только начали вводить.

Используйте всю предоставляемую Python и вашей командной оболочкой справочную информацию (`help`). Как и команда `man` в командной оболочке Linux,

`help()` — ваш встроенный в Python помощник. Попробуйте набрать в консоли Python `help` или `help(object)`. Эта команда должна сработать даже в случае, когда не работают команды `?` и `??` IPython. Попробуйте набрать `object?` и `object??` в IPython, если еще никогда этого не делали.

В оставшейся части этого букваря языка Python мы познакомим вас с некоторыми структурами данных и функциями, которые используются в этой книге, чтобы вы могли начинать с ними экспериментировать:

- `str` и `bytes`;
- `ord` и `chr`;
- `.format()`;
- `dict` и `OrderedDict`;
- `list`, `np.array`, `pd.Series`;
- `pd.DataFrame`.

Также мы расскажем о некоторых паттернах и встроенных функциях языка Python, которые иногда используются здесь и в пакете `nlpia`:

- списковые включения — `[x for x in range(10)]`;
- генераторы — `(x for x in range(1000000000))`;
- регулярные выражения — `re.match(r'[A-Za-z]+', 'Hello World')`;
- функции открытия файлов — `open('path/to/file.txt')`.

Б.1. Работа со строковыми значениями

Обработка естественного языка неразрывно связана с обработкой строк. А в работе со строками в Python 3 есть множество неожиданных нюансов, особенно для тех, кто много работал с Python 2. Так что вам не помешает поэкспериментировать со строками и способами работы с ними, чтобы чувствовать себя уверенно, когда столкнетесь со строками естественного языка.

Б.1.1. Типы строк: str и bytes

Строки (`str`) — последовательности символов в кодировке Unicode. Некоторые символы `str` могут состоять из нескольких байтов, если в строке используются не-ASCII-символы. Не-ASCII-символы часто возникают при копировании и вставке данных из Интернета в консоль или программу на Python. Причем заметить некоторые из них непросто, например круглые несимметричные кавычки и апострофы.

При открытии файла с помощью команды `open` языка Python по умолчанию он считывается в объект `str`. Попытка открыть двоичный файл, например, `'.txt'` предобученной модели Word2vec без указания режима `mode='b'` завершится неудачей. И хотя тип модели `gensim.KeyedVectors` может представлять собой текст, а не двоичные данные, файл все равно нужно открывать в двоичном режиме, чтобы

символы Unicode не искажались, когда `gensim` читает модель; то же самое справедливо и для CSV-файлов и любого другого текста, сохраненного с помощью Python 2.

Байты (`bytes`) представляют собой массивы восьмибитных значений, обычно применяемые для хранения символов таблицы ASCII или символов расширенной таблицы ASCII (в которой целочисленные значения `ord` превышают 128)[1]. Байты также иногда применяются для хранения RAW-изображений, аудиофайлов в формате WAV и других больших двоичных объектов.

Б.1.2. Шаблоны в Python: .format()

В Python включена гибкая система шаблонизации строк, позволяющая заполнить строку значениями переменных. Благодаря этому можно формировать динамические ответы на основе знаний из базы знаний или контекста работающей в настоящий момент программы на Python (`locals()`).

Б.2. Ассоциативные массивы в Python: dict и OrderedDict

Хэш-таблицы (ассоциативные массивы) — встроенные в объекты `dict` Python структуры данных. Но `dict` не обеспечивает согласованный порядок ключей, поэтому модуль `collections` из стандартной библиотеки Python содержит класс `OrderedDict`, в котором можно хранить пары ключ/значение в согласованном и легко контролируемом (зависящем от момента вставки нового ключа) порядке.

Б.3. Регулярные выражения

Регулярные выражения представляют собой маленькие компьютерные программы со своим собственным языком программирования. Любую строку регулярного выражения вроде `r'[a-z]+'` можно скомпилировать в маленькую программу, используемую для поиска соответствий внутри других строк. Далее приведены краткий справочник по ним и несколько примеров, но мы рекомендуем обратиться к онлайн-руководствам, если вы всерьез занялись NLP. Как и всегда, оптимальный способ их изучения — эксперименты с ними в командной строке. В пакет `nlpia` включены множество текстов на естественном языке и полезные примеры регулярных выражений.

Регулярное выражение определяет последовательность условных выражений (`if` в Python), каждое из которых обрабатывает отдельный символ. Эта последовательность условных выражений образует дерево, в конце концов приводящее

[1] Единого официального расширенного набора символов ASCII не существует, поэтому никогда не используйте их для NLP, разве что хотите запутать машину при усвоении универсальной модели языка.

к ответу на вопрос, соответствует ли регулярному выражению входная строка? Поскольку любое регулярное выражение может соответствовать только конечному числу строк и содержит лишь конечное число условных переходов, то оно задает конечный автомат (FSM)[1].

Компилятором/интерпретатором по умолчанию в Python служит пакет `re`, но существует и новый официальный пакет `regex`, который можно легко установить с помощью команды `pip install regex`. У него больше возможностей, лучше поддержка символов Unicode и нечеткого сопоставления с паттерном (что очень удобно для NLP). Для приводимых здесь примеров эти дополнительные возможности нам не нужны, так что можете использовать любой из этих пакетов. Для решения поставленных в данной книге задач достаточно выучить лишь несколько символов регулярных выражений:

- `|` — символ OR;
- `()` — группировка с помощью скобок, так же как в выражениях Python;
- `[]` — классы символов;
- `\s`, `\b`, `\d`, `\w` — сокращенная запись часто используемых классов символов;
- `*`, `?`, `+` — часто используемые варианты сокращенной записи числа вхождений классов символов;
- `{7,10}` — если `\*`, `?` и `+` недостаточно, можно указать точное число вхождений с помощью фигурных скобок.

Б.3.1. | — OR

Символ | используется для разделения альтернативных вариантов строк, сопоставляемых с входной строкой, в качестве части общего сопоставления с регулярным выражением. Например, регулярному выражению `'Хобсон|Коул|Ханнес'` соответствует фамилия любого из авторов этой книги. Паттерны обрабатываются слева направо, и обработка прерывается при обнаружении соответствия, как и в любом языке программирования. Поэтому в данном случае порядок паттернов между символами OR (`|`) не играет роли, поскольку первые два символа всех паттернов (фамилий авторов) различны. В листинге Б.1 показаны перемешанные фамилии авторов — можете убедиться сами.

Листинг Б.1. Символ OR регулярных выражений

```
>>> import re
>>> re.findall(r'Hannes|Hobson|Cole', 'Hobson Lane, Cole Howard,
➥ and Hannes Max Hapke')
['Hobson', 'Cole', 'Hannes']
```

Метод findall() находит во входной строке все непересекающиеся соответствия регулярному выражению и возвращает их в виде списка

[1] Это справедливо лишь для регулярных выражений со строгим синтаксисом, без «заглядывания вперед и назад».

Ради эксперимента проверьте, сможете ли вы сделать так, чтобы вычисление регулярного выражения закончилось на первом паттерне, в то время как человек, глядя на эти три паттерна, мог бы найти лучшее соответствие:

```
>>> re.findall(r'H|Hobson|Cole', 'Hobson Lane, Cole Howard,
➥ and Hannes Max Hapke')
['H', 'Cole', 'H', 'H', 'H']
```

Б.3.2. () — группы

Для группировки нескольких символьных паттернов в одно выражение можно использовать круглые скобки. Каждое из сгруппированных выражений вычисляется как единое целое. Так, выражению `r'(kitt|dogg)ie'` соответствует как *kitty*, так и *doggy*. Без скобок `r'kitt|doggy'` соответствует *kitt* или *doggy* (но, обратите внимание, не *kitty*).

У групп есть и другое назначение. Их можно использовать для захвата (извлечения) частей входного текста. Каждой из групп присваивается индекс в списке `groups()`, по которому можно извлечь соответствующий фрагмент входной строки. Метод `.group()` возвращает общую группу по умолчанию для всего выражения. Вышеупомянутые группы можно применять для захвата основы (части без *y*) регулярного выражения *kitty/doggy*, как показано в листинге Б.2.

Листинг Б.2. Использование скобок для группировки регулярных выражений

```
>>> import re
>>> match = re.match(r'(kitt|dogg)y', "doggy")
>>> match.group()
'doggy'
>>> match.group(0)
'dogg'
>>> match.groups()
('dogg',)
>>> match = re.match(r'((kitt|dogg)(y))', "doggy")
>>> match.groups()
('doggy', 'dogg', 'y')
>>> match.group(2)
'y'
```

Для захвата каждой из частей входной строки в отдельной группе

При необходимости для извлечения информации в структурированный тип данных (`dict`) можно дать группам названия, указав в начале группы символ `P`, вот так: `(P?<animal_stemm>dogg|kitt)y`[1].

[1] См. дискуссию Named regular expression group: What does “P” stand for?: https://stackoverflow.com/questions/10059673.

Б.3.3. [] — классы символов

Классы символов эквивалентны указанию символов OR (|) между символами множества. Так, `[abcd]` эквивалентно `(a|b|c|d)`, а `[abc123]` эквивалентно `(a|b|c|d|1|2|3)`.

А если часть символов в классе символов представляет собой последовательные символы алфавита (ASCII или Unicode), то можно записать их сокращенно, поставив между ними дефис. Так, `[a-d]` эквивалентно `[abcd]` или `(a|b|c|d)`, а `[a-c1-3]` — сокращение для `[abc123]` и `(a|b|c|d|1|2|3)`.

Сокращения для записи классов символов

- `\s` — `[\t\n\r]` — пробельные символы;
- `\b` — небуквенный, нецифровой символ рядом с буквой или цифрой;
- `\d` — `[0-9]` — цифра;
- `\w` — `[a-zA-Z0-9_]` — символы, образующие слова или названия переменных.

Б.4. Стиль

По возможности старайтесь придерживаться руководства по стилю PEP8 (http://python.org/dev/peps/pep-0008), даже если не планируете делиться кодом с кем-либо. Благодаря этому в будущем вам самим будет удобнее читать и отлаживать код. Простейший способ добиться соответствия своих программ руководству по стилю PEP8 — установить линтер (http://sublimelinter.com/) либо утилиту автоматического исправления стиля (http://packagecontrol.io/packages/Anaconda).

Еще одно соглашение относительно стиля, полезное для обработки естественного языка, — выбор из двух возможных символов кавычек (' и "). Какой бы вы ни выбрали, будьте последовательны. Код становится более удобочитаемым для профессионалов, если при описании предназначенной для машины строки, например, в регулярных выражениях, тегах и метках всегда использовать символ одинарной кавычки ('). Тогда для корпуса на естественном языке, предназначенного для чтения людьми, можно использовать двойные кавычки (").

А как насчет неформатированных строк (`r''` и `r""`)? Все регулярные выражения должны представлять собой неформатированные строки вида `r'match[ ]this'`, даже если они не содержат обратных косых черт. docstring должны быть неформатированными и заключенными в тройные кавычки, вот так: `r""" Эта функция предназначена для NLP """`. Таким образом, если понадобится добавить в doctest или в регулярные выражения обратные косые черты, все будет работать как задумано[1].

[1] Из следующего заданного на сайте stackoverflow вопроса становится понятно почему (https://stackoverflow.com/q/8834916/623735).

Б.5. Овладейте в совершенстве

Найдите веб-сайт интерактивного конкурса по программированию, чтобы отточить на нем свои навыки работы с Python, прежде чем начинать коммерческий проект. Можете использовать один из следующих (раз или два в неделю) во время чтения данной книги:

- CodingBat (http://codingbat.com) — решение забавных задач в интерактивном интерпретаторе Python с веб-интерфейсом;
- задачи по программированию Дона Мартина (Donne Martin) (http://github.com/donnemartin/interactive-coding-challenges) — репозиторий блокнотов Jupiter и дидактических карточек Anki с открытым исходным кодом, упрощающий изучение алгоритмов и данных;
- DataCamp (http://datacamp.com/community/tutorials) — руководства по Pandas и Python на сайте DataCamp.

Векторы и матрицы: базовые элементы линейной алгебры

Язык, на котором мыслят машины, — векторы и числа. Число, лежащее в самой основе машинных вычислений, — бит, подобно тому как буквы (символы) — базовые, неделимые части слов, языка, на котором мыслят люди. Все математические операции можно свести к нескольким логическим операциям над последовательностями битов. Аналогичным образом человеческий мозг обрабатывает последовательности символов при чтении. Первая задача, которую нужно решить для обучения машин нашим словам, — представить символы, слова, предложения и промежуточные концепты в виде векторов, с которыми могли бы работать машины, для реализации видимости разумного поведения.

B.1. Векторы

Вектор — это упорядоченная последовательность чисел без каких-либо пропусков. В scikit-learn и NumPy вектор представляет собой плотный массив (array), который ведет себя во многом похоже на список чисел Python. Основная причина, по которой мы используем массивы NumPy, а не списки языка Python, — они намного быстрее работают и используют намного меньше памяти (в четыре раза). Кроме того, можно задавать векторизованные операции, например умножение целого массива на какое-либо значение без прохода по нему в цикле `for`. Это *чрезвычайно* важно при работе с большим количеством текста, содержащего много информации, которую необходимо представить в виде этих векторов и чисел (листинг B.1).

Листинг В.1. Создание вектора

```
>>> import numpy as np
>>> np.array(range(4))
array([0, 1, 2, 3])
>>> np.arange(4)
array([0, 1, 2, 3])
>>> x = np.arange(0.5, 4, 1)
>>> x
array([ 0.5, 1.5, 2.5, 3.5])
>>> x[1] = 2
>>> x
array([ 0.5, 2, 2.5, 3.5])
>>> x.shape
(4,)
>>> x.T.shape
(4,)
```

У массивов есть определенные свойства, отсутствующие у списков, — например, `.shape` и `.T`. Атрибут `.shape` содержит длину (размер) каждого из измерений (число содержащихся в нем объектов). В названиях переменных, содержащих массивы и векторы (или даже числа), мы используем буквы в нижнем регистре, как в формальной математической нотации. В текстах по линейной алгебре, физике и машиностроению эти буквы обычно выделяются жирным шрифтом, а иногда и украшаются (особенно профессорами, пишущими мелом и фломастерами на досках) стрелочкой над ними.

Если вы слышали слово «матрица», то, наверное, знаете, что ее можно рассматривать и как массив векторов-строк:

```
>>> np.array([range(4), range(4)])
>>> array([[0, 1, 2, 3],
           [0, 1, 2, 3]])
>>> X = np.array([range(4), range(4)])
>>> X.shape
(2, 4)
>>> X.T.shape
(4, 2)
```

Свойство `T` возвращает матрицу, *транспонированную* по отношению к исходной. *Транспонированная матрица* — отраженная вдоль воображаемой диагонали, идущей из верхнего левого угла матрицы в нижний правый. Так, транспонированной к следующей матрице `A`:

```
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
```

будет матрица:

```
>>> A.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

То есть если матрица `A` представляла собой набор векторов-строк, то операция `A.T` превращает эти векторы-строки в векторы-столбцы.

B.2. Расстояния

Расстояние между двумя векторами можно измерить различными способами. Расстояние между двумя векторами представляет собой вектор, как показано в листинге B.2.

Листинг B.2. Векторное расстояние

```
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
>>> A[0]
array([1, 2, 3])
>>> A[1]
array([4, 5, 6])
>>> np.diff(A, axis=0)
array([[3, 3, 3]]
>>> A[1] - A[0]
array([3, 3, 3])
```

Этот вектор `[3, 3, 3]` в точности отражает расстояние между нашими двумя векторами по каждому из измерений. Представьте себе, что эти векторы соответствуют домам и этажам в Манхэттене для квартир двух людей: расстоянием служит точный маршрут поездки от одного к другому. Если вы живете в квартире на 3-м этаже на углу 1-й улицы и 2-й авеню, то ваши координаты в системе координат «улица, авеню, этаж» выглядят как `[1, 2, 3]`, как в примере. Если же ваш преподаватель Python живет в квартире на 6-м этаже на углу 4-й улицы и 5-й авеню, то его координатами будут `[4, 5, 6]`. Расстояние между этими векторами (`[3, 3, 3]`) означает, что вам нужно пройти три квартала на север, три квартала на восток и подняться на три этажа, чтобы добраться до его квартиры. Конечно, эта векторная математика не слишком углубляется в досадные нюансы вроде гравитации. Так что эти алгебраические вычисления предполагают, что вы прокатитесь на своем ховерборде из «назад в будущее», выехав прямо из окна на высоте трех этажей над дорогой, чтобы добраться до квартиры своего преподавателя.

Если вы скажете преподавателю, что его квартира находится на расстоянии `[3, 3, 3]` от вашей, он посмеется над педантичной точностью. Менее технически подкованные люди, говоря о расстояниях, упрощают эти три числа до одного, скаляра. Так что, если вы скажете, что квартира находится в трех кварталах от вашей, вас прекрасно поймут; вы проигнорировали нерелевантное измерение этажей, поскольку это пустяки для вашего ховерборда (или лифта). Помимо игнорирования части измерений, вы воспользовались хитрой метрикой расстояния, называемой *манхэттенским расстоянием* (Manhattan distance). Мы покажем, как вычислять его для 300-мерных векторов слов столь же просто, как и для двумерных векторов местоположений квартир.

Евклидово расстояние

Евклидово расстояние (euclidean distance) — расстояние между двумерными векторами по кратчайшему пути. Это расстояние по прямой линии между задаваемыми нашими векторами двумя точками (вершинами/острыми концами векторов).

Его также называют L2-нормой, как длину разницы между двумя векторами. Буква L в L2 означает длину (length). Цифра 2 в L2 отражает показатель степени (квадрат), в которую возводятся измерения вектора разницы перед суммированием (и извлечением из суммы квадратного корня).

Евклидово расстояние также называют RSS-расстоянием, то есть расстоянием (разницей) на основе корня из квадратов (root sum square):

```
euclidean_distance = np.sqrt(((vector1 - vector2) ** 2).sum())
```

Вычислим евклидово расстояние для векторов из примера NLP из курса лекций по ИИ Патрика Уинстона (Patrick Winston)[1].

Допустим, у нас есть векторы двумерной частотности термов (мультимножеств слов), содержащие количества вхождений слов *hack* и *computer* в статьях из двух журналов, *Wired* и *Town & Country*. Нам нужна возможность выполнять запросы для поиска при научных исследованиях в этом наборе статьей по конкретным темам. Строка запроса содержит оба слова, *hack* и *computer*. Вектор слов для строки запроса имеет вид `[1, 1]` для слов *hack* и *computer* благодаря нашему способу токенизации и выделения основ слов (см. главу 2).

Какие же статьи ближе всего к нашему запросу по евклидову расстоянию? Евклидово расстояние — длина четырех линий (рис. B.1). Они выглядят довольно похожими, но это не так. Как же решить эту задачу, чтобы наш поисковый движок возвращал при этом запросе полезные статьи?

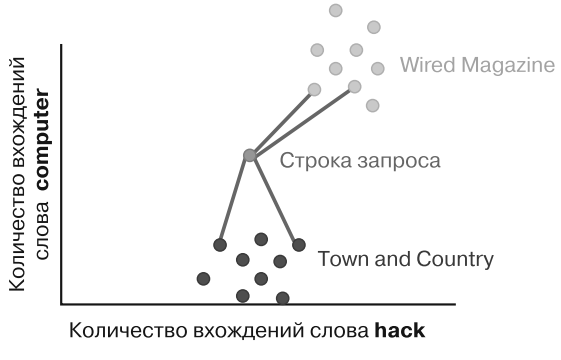


Рис. B.1. Измерение евклидова расстояния

Можно вычислить *отношение* количеств вхождений слов к общему числу слов в документе и определить на основе таких отношений евклидово расстояние.

[1] *Winston P.* 6.034 Artificial Intelligence. Fall 2010. Massachusetts Institute of Technology: MIT OpenCourseWare (https://ocw.mit.edu). Лицензия: Creative Commons BY-NC-SA. «Лекция 10» (http://mng.bz/nxjK).

Но из главы 3 мы знаем, что существует лучший способ вычисления такого отношения: TF-IDF. Евклидово расстояние между векторами TF-IDF — хорошая мера расстояния (обратного подобия) документов.

Для работы с евклидовым расстоянием можно нормализовать все наши векторы к единичной длине (длины всех векторов равны 1). Благодаря этому можно быть уверенными, что все расстояния между векторами будут располагаться в диапазоне от 0 до 2.

Косинусное расстояние

Еще более полезной может сделать наше значение расстояния одна небольшая корректировка вычислений. *Косинусное расстояние* (cosine distance) — обратная величина к косинусному коэффициенту (коэффициенту Отиаи, `cosine_distance = 1 - cosine_similarity`). Коэффициент Отиаи равен косинусу угла между двумя векторами. Так, в нашем примере угол между вектором TF для данного запроса и вектором для статей из журнала *Wired* намного меньше, чем угол между запросом и статьями из *Town & Country*. Это нам и нужно. Поскольку запрос *hacking computers* должен вернуть только статьи из *Wired*, а *не* статьи об активном отдыхе, например о езде на лошади *(hacking)*[1], охоте на уток, званых обедах и оформлении интерьера в сельском стиле.

Этот коэффициент можно эффективно вычислить как скалярное произведение двух нормализованных векторов — векторов, все значения которых разделены на длину вектора, как показано в листинге В.3.

Листинг В.3. Косинусное расстояние

```
>>> import numpy as np
>>> vector_query = np.array([1, 1])
>>> vector_tc = np.array([1, 0])
>>> vector_wired = np.array([5, 6])
>>> normalized_query = vector_query / np.linalg.norm(vector_query)
>>> normalized_tc = vector_tc / np.linalg.norm(vector_tc)
>>> normalized_wired = vector_wired / np.linalg.norm(vector_wired)

>>> normalized_query
array([ 0.70710678, 0.70710678])
>>> normalized_tc
array([ 1., 0.])
>>> normalized_wired
array([ 0.6401844 , 0.76822128])
```

Косинусные коэффициенты TF-вектора нашего запроса и остальных двух TF-векторов (косинусы углов между ними) равны:

```
>>> np.dot(normalized_query, normalized_tc) # косинусный коэффициент
0.70710678118654746
>>> np.dot(normalized_query, normalized_wired) # косинусный коэффициент
0.99589320646770374
```

[1] См. статью Hack (horse) в «Википедии» о значении слова hack в английском языке применительно к верховой езде (https://en.wikipedia.org/wiki/Hack_%28horse%29).

Косинусное *расстояние* нашего запроса и остальных двух TF-векторов равно единице минус косинусный коэффициент:

```
>>> 1 - np.dot(normalized_query, normalized_tc) # cosine distance
0.29289321881345254
>>> 1 - np.dot(normalized_query, normalized_wired) # cosine distance
0.0041067935322962601
```

Косинусные коэффициенты используются для TF-векторов в NLP из-за:

- простоты вычисления (только умножение и сложение);
- удобного диапазона значений (от –1 до +1);
- простоты вычисления (`1 - cosine_similarity`) обратной к ним величины (косинусного расстояния);
- ограниченности обратной к ним величины (косинусного расстояния) (в диапазоне от 0 до +2).

Впрочем, у косинусного расстояния есть один недостаток по сравнению с евклидовым расстоянием: это ненастоящая *метрика расстояния*, поскольку не соблюдается неравенство треугольника[1]. Это значит, что если косинусное расстояние векторов слов для *red* и *car* равно 0,5, а для *red* и *apple* — 0,3, то косинусное расстояние от *apple* до *car* может быть больше 0,8. Неравенство треугольника в основном важно при использовании косинусных расстояний для доказательств каких-либо свойств векторов. На практике в NLP это редкий случай.

Манхэттенское расстояние

Манхэттенское расстояние также называют «расстоянием такси» или L1-нормой. «Расстоянием такси» его называют потому, что оно отражает длину проезжаемого такси пути от одного вектора до другого, в случае если наши векторы представляют собой двумерные векторы с выровненными по сетке улиц координатами[2]. Манхэттенское расстояние исключительно просто в вычислении: достаточно суммировать абсолютные расстояния по всем измерениям. Для наших выдуманных векторов журналов манхэттенское расстояние будет равно:

```
>>> vector_tc = np.array([1, 0])
>>> vector_wired = np.array([5, 6])
>>> np.abs(vector_tc - vector_wired).sum()
10
```

Если нормализировать векторы перед вычислением манхэттенского расстояния, то расстояние будет совсем другим:

```
>>> normalized_tc = vector_tc / np.linalg.norm(vector_tc)
>>> normalized_wired = vector_wired / np.linalg.norm(vector_wired)
>>> np.abs(normalized_tc - normalized_wired).sum()
1.128...
```

---

[1] См. статью «Википедии» Cosine similarity, где есть ссылка на правила, которые должны соблюдаться для настоящих метрик расстояния (http://en.wikipedia.org/wiki/Cosine_similarity).

[2] См. статью https://ru.wikipedia.org/wiki/Расстояние_городских_кварталов.

Хотелось бы надеяться, что значение этой метрики расстояния также будет находиться в каком-либо диапазоне, например от 0 до 2, но это не так. Как и евклидово расстояние, манхэттенское расстояние — настоящая метрика, оно подчиняется неравенству треугольника и может использоваться в математических доказательствах, основанных на истинных метриках расстояния. Но, в отличие от евклидова расстояния для нормализированных векторов, нельзя рассчитывать, что манхэттенское расстояние между нормализированными векторами останется ограниченным каким-либо аккуратным диапазоном, например от 0 до 2. Максимальная возможная длина увеличивается с ростом размерности, даже если нормализировать все векторы к единичной длине. Для нормализированных двумерных векторов максимальное манхэттенское расстояние между двумя векторами составляет 2,82 (квадратный корень из 8). Для трехмерных — 3,46 (квадратный корень из 12). Можете угадать или вычислить, каково максимальное расстояние для четырехмерных?

Г Инструменты и методы машинного обучения

Обработка естественного языка очень часто включает машинное обучение. Так что вовсе не помешает разобраться в основных инструментах и методах машинного обучения. Некоторые из них описаны в предыдущих главах, некоторые — нет, но все они заслуживают по крайней мере нескольких слов тут.

Г.1. Выбор данных и устранение предвзятости

Выбор данных и проектирование признаков сопряжены с сильной опасностью предвзятости (с человеческой точки зрения). Если вложить в алгоритм свои предубеждения (путем выбора конкретного набора признаков), модель усвоит эти предубеждения и выдаст результаты с систематической ошибкой. Даже если повезет и эту систематическую ошибку удастся обнаружить до ввода в эксплуатацию, все равно ее исключение потребует колоссальных усилий. Придется перестроить и обучить заново весь конвейер, чтобы он смог, например, воспользоваться новым словарем из токенизатора. Придется начинать с самого начала.

Один из возможных примеров — выбор данных и признаков для известной модели Word2vec. Модель Word2vec обучалась на обширном массиве новостных статей, из которого для этой модели в качестве словаря (признаков) был выбран 1 миллион или около того *N*-грамм. В результате получилась модель, вызвавшая восхищение исследователей данных и лингвистов возможностью математических операций над векторами слов, например: *king – man + woman = queen*. Но по мере углубления исследований в модели обнаруживались все более проблемные взаимосвязи.

Например, для выражения *doctor – father + mother = nurse* («доктор – отец + + мать = медсестра») ответ *nurse* не был тем непредвзятым и логичным результатом, на который надеялись исследователи. В модель нечаянно была внесена системати-

ческая ошибка, связанная с гендером. Исходная модель Word2vec кишела схожими расовыми, религиозными и даже связанными с географическими регионами систематическими ошибками. Исследователи Google не вносили систематические ошибки намеренно, они являются неотъемлемой частью данных, статистики использования слов в корпусе Google News, на котором Word2vec обучалась.

Многие из новостных статей предвзяты, поскольку написаны журналистами, мотивированными доставлять читателям положительные эмоции. Вдобавок журналисты пишут статьи о мире, полном предубеждений на государственном уровне и уровне отдельных событий и людей. Статистика использования слов в Google News просто отражает тот факт, что матерей-медсестер намного больше, чем матерей-докторов. Равно как и отцов-докторов намного больше, чем отцов-медсестер[1]. Модель Word2vec позволяет нам заглянуть через окошко в созданный нами мир.

К счастью, для таких моделей, как Word2vec, маркированные тренировочные данные не нужны. Так что можно выбрать любой текст, какой только захочется, для обучения модели. Можно выбрать более гармоничный набор данных, лучше отражающий убеждения и умозаключения, которые должны быть присущи модели. И когда другие разработчики будут скрываться за алгоритмами и утверждать, что делают лишь то, что им говорит модель, мы сможем поделиться с ними наборами данных, которые бы лучше отражали столь желанное нам общество равных возможностей.

При обучении и проверке моделей разработчику следует полагаться на внутреннее чувство справедливости для определения момента, когда модель будет готова производить предсказания, влияющие на жизни клиентов. Если ваша модель обращается со *всеми* пользователями так, как вам бы хотелось, чтобы поступали с вами, можете спать спокойно. Не помешает также уделить особое внимание непохожим на вас пользователям, особенно тем, кого общество обычно ставит в невыгодное положение. Если же вам нужно больше формальных оправданий своим действиям, можете, в дополнение к изученным в этой книге навыкам работы с вычислительной техникой, заняться изучением статистики, философии, этики, психологии, поведенческой экономики и антропологии.

У вас, как у специалиста-практика, занимающегося обработкой естественного языка и машинным обучением, есть возможность обучать машины вести себя лучше, чем ведет себя большинство людей. Ваши начальники и коллеги вряд ли станут указывать вам, какие документы включать или исключать из тренировочного набора данных. Вы наделены властью влиять на поведение машин, формирующих отдельные группы людей и общество в целом.

Мы привели несколько идей о том, как сформировать менее предвзятый и более справедливый набор данных. Теперь покажем, как обучить модели на непредвзятых данных так, чтобы они были точны и полезны на практике.

Г.2. Насколько хорошо подогнана модель

Одна из основных проблем, как и в случае любой модели машинного обучения, — как не допустить, чтобы модель оказалась *слишком хорошо* подогнана к данным. Как может что-то быть «слишком хорошо»? При работе с примерами данных для любой

[1] Точнее, медбратьев. — *Примеч. пер.*

модели конкретный алгоритм может слишком хорошо находить паттерны для конкретного набора данных. Но это приносит мало пользы, поскольку мы уже знаем метки примеров данных из тренировочного набора (иначе это не был бы тренировочный набор данных). Настоящая цель — создать на основе этих тренировочных примеров данных хорошо *обобщающую* модель, которая смогла бы правильно маркировать пример, схожий с примерами из тренировочного набора, но не входящий в него. Необходимо максимизировать качество работы модели на новых примерах данных, не входящих в тренировочный набор.

Модель, которая идеально описывает (и предсказывает) тренировочные примеры данных, — *переобучена* (overfit) (рис. Г.1). Подобная модель будет не в состоянии (или почти не в состоянии) описывать новые данные. Она не будет общей моделью, на работу которой на примерах не из тренировочного набора можно было бы положиться.

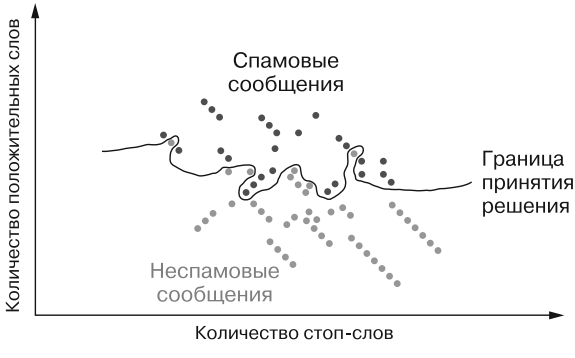


Рис. Г.1. Переобученная на тренировочных примерах данных модель

И наоборот, если модель неправильно выполняет многие из предсказаний на тренировочных данных, но и на новых примерах данных демонстрирует плохие результаты, она *недообучена* (underfit) (рис. Г.2). Ни один из этих видов моделей

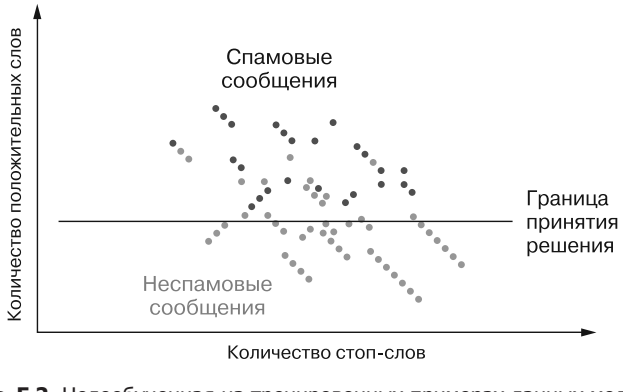


Рис. Г.2. Недообученная на тренировочных примерах данных модель

не подходит для выполнения предсказаний на практике. Поэтому рассмотрим методы, позволяющие обнаружить такие проблемы, и, что более важно, способы их избежать.

Г.3. Знание — половина победы

На практике машинного обучения, если данные — золото, то маркированные данные — один из бесценных редкоземельных элементов (или какая там метафора вам ближе к сердцу). Первый порыв — скормить все до последнего маркированные данные модели. Чем больше данных — тем лучше модель, правда? Но тем самым мы лишаем себя возможности проверить ее, разве что отправить ее в эксплуатацию и надеяться на лучшее. Конечно, на деле подобное неприемлемо. Решение этой проблемы состоит в разбиении маркированных данных на две, а иногда и три части: тренировочный набор данных, *проверочный* набор данных (validation set), а иногда еще и *тестовый* набор данных (test set).

С тренировочным набором данных все понятно. Проверочный набор данных — небольшая часть маркированных данных, отделенная и скрытая от модели на время одного цикла обучения. Хорошие результаты на проверочном наборе данных — первый шаг к уверенности в том, что обученная модель будет хорошо работать при получении новых для нее данных. Чаще всего имеющийся маркированный набор данных разбивают на тренировочный/проверочный в соотношении 80/20 или 70/30. *Тестовый* набор данных аналогичен проверочному и представляет собой подмножество маркированных тренировочных данных, на котором проверяют качество работы модели. Но чем же тестовый набор данных отличается от проверочного? По составу они вообще не отличаются. Отличие заключается в том, как их используют.

При обучении модели на тренировочном наборе данных производится несколько итераций обучения с различными гиперпараметрами и в качестве окончательной выбирается модель, которая демонстрирует наилучшие результаты на проверочном наборе. Но здесь есть хитрость. Как убедиться, что мы не выбрали модель, которая просто сильно перекошена в сторону проверочного набора данных? Невозможно гарантировать, что эта модель будет хорошо работать на новых реальных данных. А именно это и интересует больше всего ваше начальство и читателей ваших статей — насколько хорошо она будет работать на *их* данных.

Так что если у вас достаточно данных, лучше отложить в сторону третью порцию маркированного набора данных в качестве *тестового* набора. Благодаря этому ваши читатели (или начальство) будут уверены, что модель сможет работать на никогда не встречавшихся ей в процессе обучения и подстройки данных. После выбора модели на основе продемонстрированных на проверочном наборе показателей, после завершения всякого обучения и корректировки модели можно произвести предсказания (вывод) для всех примеров данных из тестового набора. И если модель хорошо работает на третьем наборе данных — значит, она хорошо обобщается. Для подобной проверки и получения полной уверенности набор данных обычно разбивают на тренировочный/проверочный/тестовый в соотношении 60/20/20.

СОВЕТ

Очень важно хорошо «перетасовать» набор данных перед его разбиением на тренировочный, проверочный и тестовый наборы данных. Каждое из подмножеств должно хорошо отражать реальность и содержать примерно равные доли всех ожидаемых меток. Если в тренировочном наборе данных 25 % положительных примеров данных и 75 % отрицательных, то в проверочном и тестовом наборах также должно быть по 25 % положительных примеров данных и 75 % отрицательных. Если же все отрицательные примеры находились в начале исходного набора данных и разбиение на тренировочные/тестовые данные было произведено в соотношении 50/50 без предварительного «перемешивания» набора данных, то в тренировочном наборе данных окажется 100 % отрицательных примеров данных, а в тестовом наборе данных — 50/50. Модель ничего не сможет усвоить из положительных примеров данных набора.

Г.4. Перекрестное обучение

Еще один подход к вопросу разбиения на тренировочные/тестовые данные — *кросс-валидация*, или *перекрестная проверка* (cross-validation), или *k-блочная кросс-валидация* (k-fold cross-validation) (рис. Г.3). Идея кросс-валидации очень близка к описанным выше грубым разбиениям, но открывает дорогу к использованию всего набора данных для обучения. Процесс включает разбиение тренировочного набора данных на k равных множеств — *блоков* (folds). Далее производится обучение модели на $k - 1$ из этих блоков в качестве тренировочного набора данных и проверка на оставшемся k-м блоке, затем обучение производится снова, с одним из использовавшихся на предыдущем шаге $k - 1$ блоков в качестве выделенного проверочного набора данных. Оставшиеся $k - 1$ блоков играют роль нового тренировочного набора данных.

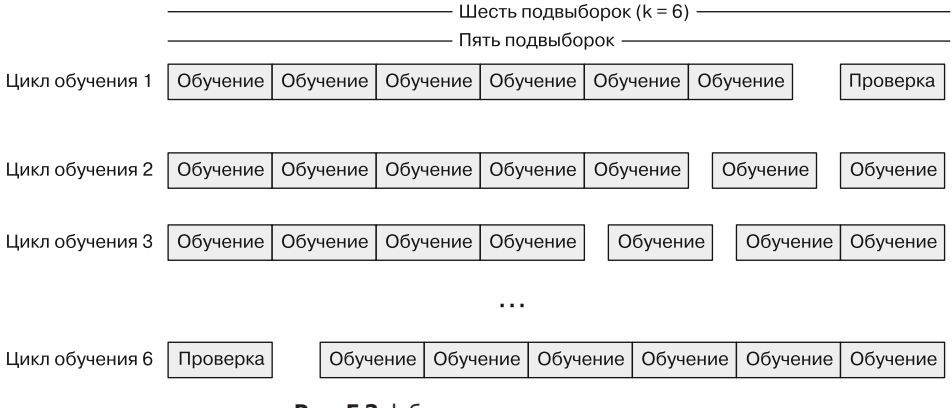


Рис. Г.3. k-блочная кросс-валидация

Этот метод играет важную роль при анализе структуры модели и поиске гиперпараметров, хорошо подходящих для разнообразных проверочных данных. После

выбора гиперпараметров все равно необходимо выбрать наилучшую *обученную* модель, которая поэтому подвержена перекосу, упомянутому в предыдущем разделе, так что рекомендуется отделить при этом процессе тестовый набор данных.

Такой подход позволяет получить дополнительную информацию о надежности модели. Можно, например, вычислить P-значение вероятности того, что выявленные моделью взаимосвязи между входными признаками и выходными предсказаниями статистически значимы, а не носят случайный характер. Информация о том, что тренировочный набор данных действительно является репрезентативной выборкой реального мира, существенно новая.

Платой за эту дополнительную уверенность в нашей модели является k-кратное удлинение время обучения, в случае k-блочной кросс-валидации. Так что если нужно получить 90%-ный ответ для задачи, часто достаточно просто произвести одноблочную кросс-валидацию. Этот один блок эквивалентен описанному выше разбиению нашего набора данных на тренировочный и проверочный. Уверенности в 100%-ной надежности отражения нашей моделью динамики реального мира у нас не будет, но если она хорошо работает на тестовом наборе данных, можно быть вполне уверенными, что она *пригодна* для предсказания целевой величины. Так что такой подход вполне годится для большинства видов коммерческого применения моделей машинного обучения.

Г.5. Притормаживаем модель

Во время выполнения `model.fit()` градиентный спуск очень рьяно минимизирует возможную ошибку модели. Это может привести к переобучению, при котором модель демонстрирует великолепные результаты на тренировочном наборе данных и очень плохие — на новых, не встречавшихся ей примерах данных (тестовом наборе данных). Поэтому желательно «притормозить» модели. Сделать это можно тремя способами:

- регуляризацией;
- случайным дропаутом;
- нормализацией по мини-батчам.

Г.5.1. Регуляризация

В любой модели машинного обучения в конце концов возникает переобучение. К счастью, существует несколько инструментов для борьбы с этим неприятным явлением. Первый из них — *регуляризация* (regularization), то есть штрафование подбираемых параметров на каждом шаге обучения. Обычно, хотя не всегда, пропорционально самим параметрам. Чаще всего для этого используются *L1-норма* и *L2-норма*.

L1-регуляризация

$$+\lambda\sum_{i=1}^{n}|w_i|$$

L1-норма представляет собой сумму абсолютных значений всех параметров (весов), умноженную на некий параметр лямбда (гиперпараметр), обычно небольшое число с плавающей точкой в диапазоне от 0 до 1. Эта сумма применяется для обновления весов. Идея заключается в том, что большие по модулю веса навлекают на себя штраф, и поощряется более равномерное использование моделью весов.

L2-регуляризация

$$+\lambda\sum_{i=1}^{n} w_i^2$$

Аналогично L2-норма также представляет собой штрафование весов, но слегка по другой формуле. В данном случае речь идет о сумме квадратов весов, умноженной на некий параметр лямбда (отдельный гиперпараметр, выбираемый перед обучением).

Г.5.2. Дропаут

В нейронных сетях *дропаут* (dropout) представляет собой еще один удобный инструмент решения вышеупомянутой проблемы — на первый взгляд, волшебный. Идея дропаута состоит в отключении на любом слое нейронной сети части сигнала, проходящего через этот слой при обучении. Обратите внимание, что это происходит *только* во время обучения, а не вывода. На каждом проходе обучения подмножество нейронов на нижнем уровне «игнорируется»; эти выходные значения явным образом обнуляются. Поскольку они не влияют на итоговое предсказание, то и не получают обновления весов во время шага обратного распространения ошибки. На следующем шаге обучения для обнуления выбирается другое подмножество весов на этом слое.

Как же сеть обучается, если 20 % ее интеллекта в любом моменте времени отключено? Идея в том, что никакой конкретный путь по весам не должен полностью определять конкретный атрибут данных. Модель должна обобщать свои внутренние структуры так, чтобы иметь возможность обрабатывать данные по разным путям через нейроны.

Доля отключаемого подобным образом сигнала задается в виде гиперпараметра и находится в диапазоне от 0 до 1. На практике обычно оптимальным является дропаут от 0,1 до 0,5, но это, конечно, зависит от модели. Во время вывода дропаут игнорируется, и вся мощь обученных весов обрушивается на новые данные.

Keras предоставляет очень простой способ реализации дропаута, который вы можете видеть в примерах из данной книги и в листинге Г.1.

Листинг Г.1. Слой дропаута в Keras сокращает переобучение

```
>>> from keras.models import Sequential
>>> from keras.layers import Dropout, LSTM, Flatten, Dense

>>> num_neurons = 20
>>> maxlen = 100
>>> embedding_dims = 300
>>> model = Sequential()
```

Для примера взяты произвольные гиперпараметры

```
>>> model.add(LSTM(num_neurons, return_sequences=True,
...                input_shape=(maxlen, embedding_dims)))
>>> model.add(Dropout(.2))
>>> model.add(Flatten())
>>> model.add(Dense(1, activation='sigmoid'))
```

Здесь .2 — гиперпараметр, так что 20 % выходных сигналов расположенного выше слоя LSTM обнуляется, а значит, игнорируется

Г.5.3. Нормализация по мини-батчам

Для регуляризации и обобщения модели в нейронных сетях можно использовать и более новую идею: *нормализацию по мини-батчам* (batch normalization). Данная идея заключается в том, что, подобно входным данным, выходные сигналы каждого из слоев должны нормализоваться до значений в диапазоне от 0 до 1. До сих пор ведутся споры, почему и когда это приносит пользу и при каких условиях имеет смысл использовать данный метод. При желании можете сами изучить этот вопрос.

Но Keras предоставляет удобную реализацию нормализации по мини-батчам в виде слоя `BatchNormalization`, как показано в листинге Г.2.

Листинг Г.2. BatchNormalization

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dropout, LSTM, Flatten, Dense
>>> from keras.layers.normalization import BatchNormalization

>>> model = Sequential()
>>> model.add(Dense(64, input_dim=14))
>>> model.add(BatchNormalization())
>>> model.add(Activation('sigmoid'))
>>> model.add(Dense(64, input_dim=14))
>>> model.add(BatchNormalization())
>>> model.add(Activation('sigmoid'))
>>> model.add(Dense(1, activation='sigmoid'))
```

Г.6. Несбалансированные тренировочные наборы данных

Качество модели машинного обучения определяется качеством подаваемых на ее вход данных. Большое количество данных приносит пользу лишь тогда, когда примеры данных охватывают все случаи, которые необходимо предсказать. Причем однократного охвата каждого из случаев недостаточно. Представьте себе, что мы пытаемся предсказать, собака или кошка изображена на фотографии. Но тренировочный набор данных содержит 20 000 изображений кошек и лишь 200 — собак. Если обучить модель на таком наборе данных, она, вероятно, просто будет предсказывать, что на любой заданной картинке изображена кошка, независимо от входных данных. Причем с точки зрения модели это совершенно правильно, правда? В том

смысле, что в 99 % случаев из тренировочного набора данных такое предсказание будет правильным. Конечно, это псевдодовод и модель никуда не годится. Но, если говорить вне контекста конкретной модели, наиболее вероятная причина подобной неудачи — *несбалансированный тренировочный набор данных*.

Модели могут быть очень чувствительными к тренировочным наборам данных по той простой причине, что сигнал от класса, чрезмерно представленного в выборке, может подавлять сигналы от менее представленных классов. В результате веса чаще обновляются на основе ошибки, сгенерированной преобладающим классом, а сигнал от менее представленного класса размывается. Строго равномерное представительство каждого из классов не обязательно, поскольку модели способны преодолевать некоторый шум. Главное, чтобы масштабы были сопоставимы.

Первый шаг, как и для любой задачи машинного обучения, пристально и внимательно изучить данные. Прочувствуйте все их нюансы и рассчитайте приближенные статистические показатели, чтобы понять, что на самом деле представляют данные. Выясните не только, сколько у вас данных, но и сколько данных каких видов.

Так что же делать, если с самого начала все плохо? Для выравнивания представленности данных различных классов существует три основных метода: супердискретизация, субдискретизация и дополнение данных.

Г.6.1. Супердискретизация

Супердискретизация — метод повторения примеров данных из недостаточно представленного в выборке класса или классов. Возьмем вышеупомянутый пример с собаками/кошками (лишь 200 картинок с изображением собак на 20 000 кошек). Можно просто продублировать имеющиеся изображения собак 100 раз, и в результате получится 40 000 примеров данных, половина собак и половина кошек.

Это крайний пример, который ведет к своим проблемам. Скорее всего, сеть очень хорошо обучится узнавать эти 200 собак и будет плохо обобщаться на других собак, не включенных в тренировочный набор данных. Но метод супердискретизации определенно может помочь сбалансировать тренировочный набор данных в случаях, когда дисбаланс не столь велик.

Г.6.2. Субдискретизация

Субдискретизация — оборотная сторона той же медали. В этом случае отбрасываются примеры данных из чрезмерно представленного в выборке класса. В примере с собаками/кошками можно случайным образом выбросить 19 800 изображений кошек и оставить в итоге 400 с примерами данных, половина собак и половина кошек. У этого метода, конечно, тоже есть явная проблема. Мы отбросили подавляющее большинство данных, и фундамент для нашей работы теперь стал гораздо у́же. Подобные крайние случаи нежелательны, но при наличии большого количества примеров в недостаточно представленном классе перспективы такого метода также неплохи. Иметь настолько много данных, безусловно, одно удовольствие.

Г.6.3. Дополнение данных

Этот способ несколько сложнее, но при соответствующих обстоятельствах *дополнение* данных также может быть полезно. Идея дополнения состоит в генерации новых данных — либо с нуля, либо путем внесения шума в уже существующие. Примером может служить AffNIST (http://www.cs.toronto.edu/~tijmen/affNIST). Знаменитый набор данных MNIST представляет собой набор написанных от руки цифр от 0 до 9 (рис. Г.4). AffNIST искажает, вращает и масштабирует все цифры различным образом, сохраняя при этом исходные метки.

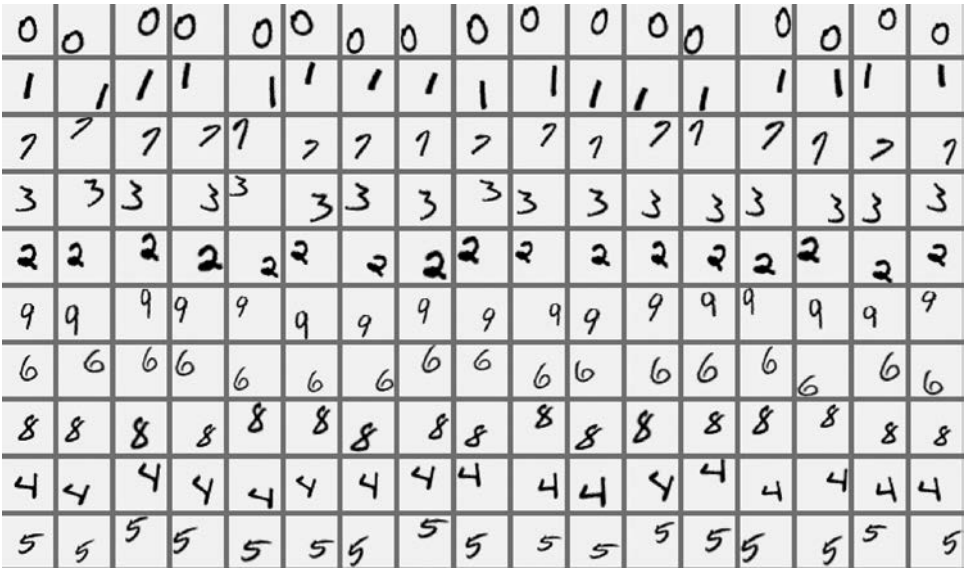


Рис. Г.4. В крайнем слева столбце приведены исходные примеры данных из MNIST; в других столбцах — аффинные преобразования этих данных, включенные в AffNIST (источник изображений: affNIST (http://www.cs.toronto.edu/~tijmen/affNIST))

Цель данного исследования — не сбалансировать тренировочный набор данных, а сделать, например, сверточные нейронные сети более устойчивыми к другим формам написания новых данных, но идея все равно заключается в дополнении данных.

Впрочем, будьте осторожны. Добавление данных, которые плохо отражают моделируемую сущность, может принести больше вреда, чем пользы. Рассмотрим, например, вышеупомянутый набор данных с 200/20 000 собак/кошек. И пускай все изображения — цветные фотографии в высоком разрешении, снятые при идеальных условиях. Вряд ли можно получить подходящие дополненные данные, если просто раздать мелки 19 с лишним тысячам детсадовцев. Всегда следует хоть немного обдумывать, как дополнение данных повлияет на модель. Ответ не всегда ясен, так что нужно это учитывать при проверке итоговой модели и по возможности проверять граничные случаи, чтобы поведение не оказалось не тем, которое ожидалось.

И наконец, вероятно, наименее полезный совет, но все же: если набор данных неполон, никогда не будет лишним поискать дополнительные данные в их первоисточнике. На практике это не всегда осуществимо, но стоит хотя бы учитывать такой вариант.

Г.7. Метрики эффективности

Важнейшая составляющая любого конвейера машинного обучения — метрики эффективности его работы. Если не знать, насколько хорошо модель машинного обучения работает, не удастся ее и улучшить. Первым делом при создании конвейера машинного обучения мы задаем метрику эффективности его работы, такую как `.score()`, в любой модели машинного обучения scikit-learn. Затем создаем произвольный конвейер классификации/регрессии, в конце которого вычисляется показатель эффективности. Благодаря этому можно постепенно вносить усовершенствования в конвейер, которые в итоге улучшают данный показатель, приближая нас к цели. Это также позволяет демонстрировать начальству и коллегам, что мы на правильном пути.

Г.7.1. Оценка эффективности классификатора

Классификатор должен правильно выполнять две функции: маркировать меткой класса то, что к нему на самом деле принадлежит, и не маркировать этой меткой то, что к нему не принадлежит. То, сколько раз он правильно выполнил каждую из этих операций, называется истинно положительными и истинно отрицательными результатами. При наличии массива выполненных моделью классификаций/предсказаний в виде массивов NumPy можно подсчитать правильные предсказания, как показано в листинге Г.3.

Листинг Г.3. Подсчитываем правильно выполненные моделью предсказания

```
>>> y_true = np.array([0, 0, 0, 1, 1, 1, 1, 1, 1, 1])
>>> y_pred = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0, 0])
>>> true_positives = ((y_pred == y_true) & (y_pred == 1)).sum()
>>> true_positives
4

>>> true_negatives = ((y_pred == y_true) & (y_pred == 0)).sum()
>>> true_negatives
2
```

y_true — массив NumPy истинных (правильных) меток классов. Обычно определяются человеком

y_pred — массив NumPy меток классов (0 или 1), предсказанных моделью

true_positives — правильно предсказанные моделью (правильно маркированы 1) метки положительного класса (1)

true_negatives — правильно предсказанные моделью (правильно маркированы 0) метки негативного класса (0)

Часто необходимо также подсчитать неправильные предсказания модели, как показано в листинге Г.4.

Листинг Г.4. Подсчитываем неправильные предсказания модели

```
>>> false_positives = ((y_pred != y_true) & (y_pred == 1)).sum()
>>> false_positives
3
>>> false_negatives = ((y_pred != y_true) & (y_pred == 0)).sum()
>>> false_negatives
1
```

false_positives — примеры данных негативного класса (0), ошибочно маркированные моделью как положительные (маркированы 1, хотя должны быть 0)

false_negatives — примеры данных положительного класса (1), ошибочно маркированные моделью как негативные (маркированы 0, хотя должны быть 1)

Иногда четыре числа объединяются в одну матрицу 2×2, которая называется *матрицей ошибок* (error matrix) или *матрицей различий* (confusion matrix). Листинг Г.5 демонстрирует, как будет выглядеть наша вымышленная таблица предсказанных и истинных значений.

Листинг Г.5. Матрица различий

```
>>> confusion = [[true_positives, false_positives],
...              [false_negatives, true_negatives]]
>>> confusion
[[4, 3], [1, 2]]
>>> import pandas as pd
>>> confusion = pd.DataFrame(confusion, columns=[1, 0], index=[1, 0])
>>> confusion.index.name = r'pred \ truth'
>>> confusion
              1    0
pred \ truth
1             4    1
0             3    2
```

В матрице различий должны быть большие значения на диагонали (идущей слева сверху направо вниз) и маленькие вне диагонали (в верхнем правом углу и нижнем левом). Впрочем, порядок положительных и отрицательных значений произволен, так что иногда встречается транспонированный вид матрицы различий. Всегда маркируйте столбцы и индексы[1] своих матриц различий. Иногда специалисты по статистике называют эту матрицу таблицей сопряженности классификатора, но во избежание путаницы лучше называть ее матрицей различий.

Существует два удобных способа объединения этих четырех чисел в единую метрику эффективности для классификационной задачи машинного обучения: *точность* (precision) и *полнота* (recall). Примерами подобных задач классификации могут служить информационный поиск (поисковые системы) и семантический поиск, поскольку задача в них состоит в классификации документов как подходящих или нет. В главе 2 мы рассказывали, как стемминг и лемматизация повышают полноту, но снижают точность.

Точность описывает, насколько хорошо модель обнаруживает все члены интересующего нас класса — положительного. Поэтому ее также называют положительной значимостью предсказаний (positive predictive value). Поскольку

[1] Возможно, авторы имели в виду строки. — *Примеч. пер.*

истинно положительными результатами являются правильно предсказанные положительные метки, а ложноположительными результатами — отрицательные примеры данных, неправильно маркированные как положительные, то точность можно вычислить так, как показано в листинге Г.6.

Листинг Г.6. Точность

```
>>> precision = true_positives / (true_positives + false_positives)
>>> precision
0.8...
```

Точность для этого примера матрицы различий равна примерно 80 %, поскольку модель правильно предсказала 80 % истинных меток.

Показатель полноты аналогичен. Он также называется чувствительностью (sensitivity), или частотой истинно положительных результатов (true positive rate), или вероятностью обнаружения (probability of detection). Поскольку общее число примеров данных в наборе данных равно сумме истинно положительных и ложноотрицательных результатов, то полноту, долю обнаруженных положительных меток, можно вычислить с помощью кода, приведенного в листинге Г.7.

Листинг Г.7. Полнота

```
>>> recall = true_positives / (true_positives + false_negatives)
>>> recall
0.571
```

Итак, это значит, что наш пример модели обнаруживает 57 % положительных примеров данных из набора данных.

Г.7.2. Оценка эффективности регрессора

Два показателя эффективности, чаще всего используемых при решении регрессионных задач машинного обучения, — корень из среднеквадратической погрешности (root mean square error, RMSE) и корреляция Пирсона (R^2). Оказывается, что задачи классификации, по сути, являются задачами регрессии. Так что можно воспользоваться метриками регрессии для меток классов, если преобразовать их в числа, как мы делали в предыдущем разделе. Поэтому в следующих примерах кода мы используем снова предсказания и истинные значения из предыдущего примера. Для большинства задач более удобен RMSE, поскольку из него сразу понятно, насколько далеко, по-видимому, будут предсказания от истинных значений. Показатель RMSE дает нам стандартное отклонение ошибки (листинг Г.8).

Листинг Г.8. RMSE

```
>>> y_true = np.array([0, 0, 0, 1, 1, 1, 1, 1, 1, 1])
>>> y_pred = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0, 0])
>>> rmse = np.sqrt(sum((y_true - y_pred) ** 2) / len(y_true))
>>> rmse
0.632...
```

Еще одна распространенная метрика эффективности для регрессоров — коэффициент корреляции Пирсона. В модуле `sklearn` он добавлен в большинство моделей как метод `.score()` по умолчанию. Если вы не уверены, что именно эти показатели измеряют, то лучше вычислить их вручную (листинг Г.9).

Листинг Г.9. Корреляция

```
>>> corr = pd.DataFrame([y_true, y_pred]).T.corr()
>>> corr[0][1]
0.218...
>>> np.mean((y_pred - np.mean(y_pred)) * (y_true - np.mean(y_true))) /
... np.std(y_pred) / np.std(y_true)
0.218...
```

Как видите, предсказания в нашем примере коррелируют с истинными результатами только на 21,8 %.

Г.8. Советы от профессионалов

Несколько простых уловок, с помощью которых вы сможете быстрее создавать хорошие модели, после того как освоили все основы:

- для решения проблем конвейера используйте маленькую случайную выборку из своего набора данных;
- когда будете готовы к развертыванию в промышленной эксплуатации, обучите модель на всех имеющихся данных;
- сначала пробуйте лучше всего знакомый вам подход. Это относится как к процедурам выделения признаков, так и к самой модели;
- используйте диаграммы и матрицы рассеяния для признаков и целевых величин низкой размерности, чтобы убедиться, что не пропустили какие-либо очевидные закономерности;
- стройте в виде необработанных изображений графики многомерных данных для выявления временных сдвигов по признакам[1];
- попробуйте выполнить PCA на многомерных данных (LSA на данных NLP), при котором максимизируются *различия* между парами векторов (их угловые разделения);
- воспользуйтесь нелинейным методом понижения размерности, вроде t-SNE, при котором ищутся соответствия между парами векторов или выполняется регрессия в пространстве низкой размерности;

[1] Тренировочные наборы данных на основе временных рядов часто генерируются с временным сдвигом (запаздыванием). Выявление этого обстоятельства может помочь на конкурсах Kaggle, где источник данных скрывается, как, например, в соревновании по предсказанию величин трансакций банка «Сантандер» (http://www.kaggle.com/c/santander-value-prediction-challenge/discussion/61394).

- создайте объект `sklearn.Pipeline` для улучшения сопровождаемости и повторного использования моделей и процедур выделения признаков;
- автоматизируйте подстройку гиперпараметров, чтобы модель изучала данные, а вы могли тем временем заниматься изучением машинного обучения.

ПОДСТРОЙКА ГИПЕРПАРАМЕТРОВ

> Гиперпараметры — значения, определяющие эффективность работы конвейера, включающие тип модели и ее настройки. В их числе — количество нейронов и слоев в нейронной сети или значение α в регрессоре sklearn.linear_model.Ridge. Гиперпараметры также включают значения, которые обуславливают шаги предварительной обработки, например, тип токенизатора, списки игнорируемых слов, минимальную и максимальную частотности документа для словаря TF-IDF, решение, использовать лемматизатор или нет, подход к нормализации TF-IDF и т. д.

Подстройка гиперпараметров может занимать длительное время, поскольку каждый эксперимент требует обучения и проверки новой модели. Так что имеет смысл при поиске по широкому диапазону гиперпараметров уменьшить размер набора данных до минимальной репрезентативной выборки. Когда этот поиск будет на завершающей стадии и вы будете близки к итоговой модели, подходящей, по вашему мнению, для ваших нужд, можно увеличить размер набора данных и использовать столько данных, сколько нужно.

Именно с помощью подстройки гиперпараметров конвейера можно улучшить эффективность работы модели. Автоматизация подстройки гиперпараметров может сэкономить вам время на чтение книг, подобных этой, или на визуализацию и анализ результатов. При этом все равно можно прибегнуть к своей интуиции для этой подстройки, выбирая диапазоны гиперпараметров.

СОВЕТ

> Далее перечислены наиболее эффективные алгоритмы для подстройки гиперпараметров (от лучшего к худшему).
>
> 1. Байесовский поиск.
> 2. Генетические алгоритмы.
> 3. Случайный поиск.
> 4. Кратномасштабный поиск по сетке.
> 5. Поиск по сетке.
>
> Впрочем, любой алгоритм, позволяющий компьютеру производить поиск ночью, когда вы спите, лучше угадывания вручную новых параметров по одному.

Д *Настройка GPU на AWS*

Для быстрого обучения или выполнения конвейера NLP не помешает сервер с GPU. Особенно быстро GPU позволяют обучать нейронные сети при использовании для создания модели таких фреймворков, как `Keras` (`TensorFlow` или `Teano`), `PyTorch` или `Caffe`. Эти фреймворки вычислений на графах способны воспользоваться всеми преимуществами полностью распараллеливаемых операций умножения и сложения, для чего и создавались GPU.

Облачный сервис — отличный вариант, если вы не хотите тратить время и деньги на создание собственного сервера. Впрочем, можно создать сервер с GPU с быстродействием в два раза выше сравнимого сервера Amazon Web Services (AWS) примерно за те же деньги, которые стоит месячная подписка на сравнимый экземпляр AWS. Кроме того, на нем можно хранить гораздо больше данных с более сильным сцеплением (более высокой пропускной способностью) с вашей машиной и зачастую бóльшим объемом оперативной памяти, чем может быть доступно на отдельном EC2-экземпляре AWS.

Зато с помощью AWS можно быстро все настроить и начать работать без необходимости держать свои собственные устройства хранения и серверы. Кроме того, большинство провайдеров облачных сервисов преобразует заранее сконфигурированные образы жестких дисков (ISO), с помощью которых можно начать работу гораздо быстрее, чем если настраивать собственный сервер. Вероятно, для коммерческой системы имеет смысл использовать облачные сервисы, например AWS или Google Cloud Services (Azure все еще немного отстает от них). Для развлечения же и экспериментов, напротив, лучше создать свой сервер.

Д.1. Шаги создания экземпляра с GPU на AWS

1. Перейдите на сайт http://aws.amazon.com/ и зарегистрируйте новую учетную запись либо выполните вход в уже существующую. После входа в свою учетную запись перейдите в консоль управления AWS (AWS Management Console, http://console.aws.amazon.com), показанную на рис. Д.1.

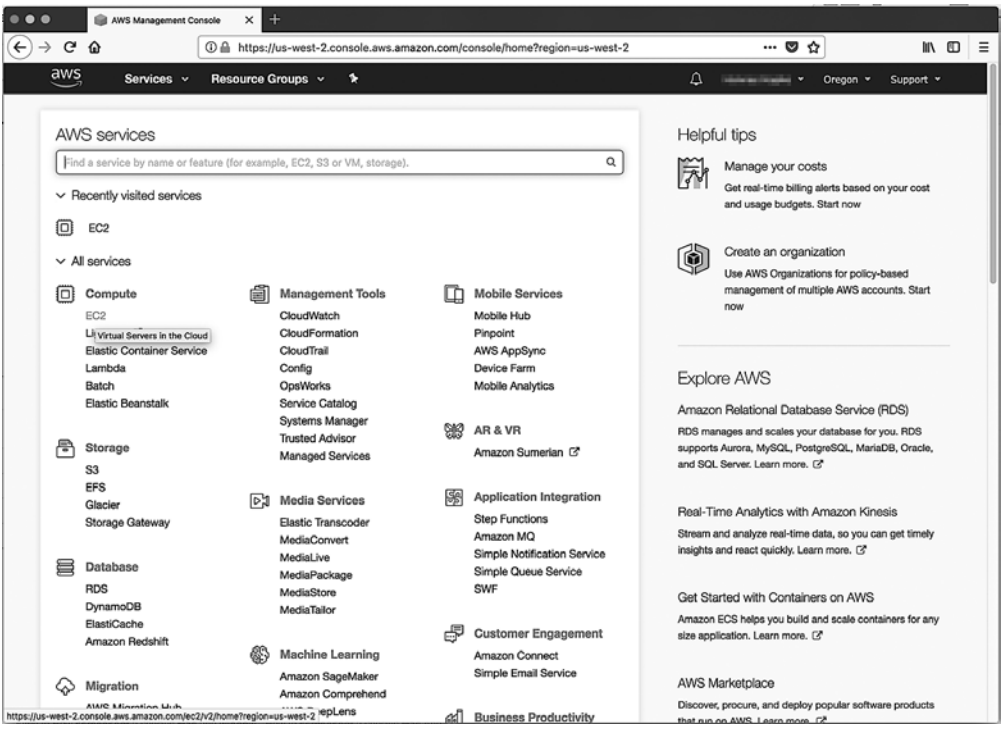


Рис. Д.1. Консоль управления AWS

2. Выберите EC2 в разделе All Services (Все сервисы); сервис EC2 можно также найти в меню Services (Сервисы) вверху страницы. В EC2 Dashboard (на панели инструментов EC2) можно найти сводку информации о существующих EC2-экземплярах (рис. Д.2).
3. На панели инструментов EC2 нажмите синюю кнопку Launch Instance (Запустить экземпляр) для запуска мастера настройки экземпляра — последовательности экранных форм, в которых можно настроить требуемую виртуальную машину.
4. В этой экранной форме (рис. Д.3) показаны образы жестких дисков сервера (ISO), которые можно установить на виртуальной машине. Amazon называет

их *Amazon Machine Images* (AMI, «образы машин Amazon»).[1] Некоторые AMI включают уже установленные в них фреймворки глубокого обучения. Таким образом, не нужно инсталлировать и настраивать библиотеки CUDA и BLAS или такие пакеты Python, как `TensorFlow`, `numpy` или `Keras`. Чтобы найти бесплатный заранее настроенный AMI для глубокого обучения, щелкните на вкладке Amazon Marketplace или Community AMIs слева и найдите там Deep Learning (Глубокое обучение).[2] При этом все равно придется настроить использующее все программные возможности конкретного AMI аппаратное обеспечение.

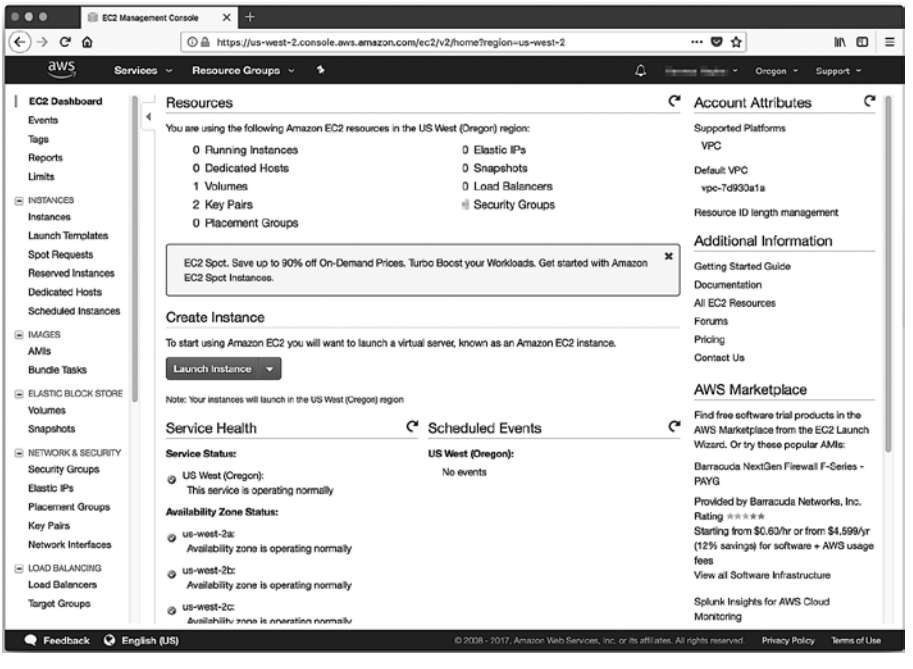


Рис. Д.2. Создание нового экземпляра AWS

5. Часть кода нейронных сетей из этой книги была протестирована на Deep Learning AMI (Ubuntu), умеющего извлекать пользу из любого имеющегося на виртуальной машине аппаратного обеспечения GPU. Нажмите синюю кнопку Select (Выбрать) рядом с тем AMI, который вы хотели бы использовать. Если выбрать образ из Amazon Marketplace, вам будет показана примерная стоимость работы этого образа на различных типах EC2-экземпляров с GPU (рис. Д.4).

---

[1] ISO — сокращение от ISO-9660, открытый стандарт Международной организации по стандартизации (International Standards Organization) по записи образов дисков таким образом, чтобы их можно было переносить и инсталлировать где угодно, а не только на проприетарном облачном сервисе вроде AWS.

[2] На момент написания данной книги идентификатор AMI одного из подобных образов в Amazon Marketplace был ami-f1d51489.

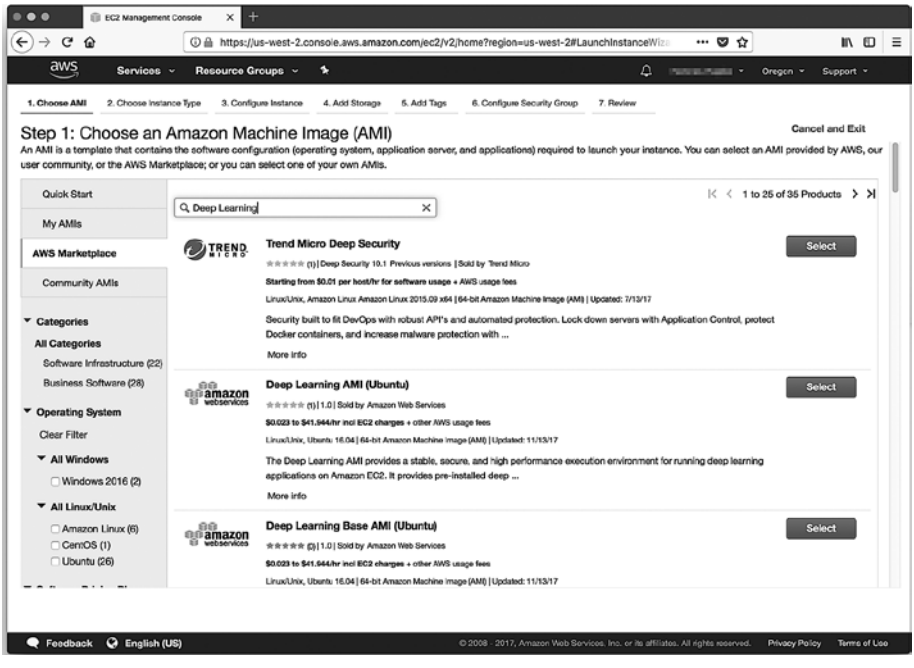


Рис. Д.3. Выбор образа машины AWS

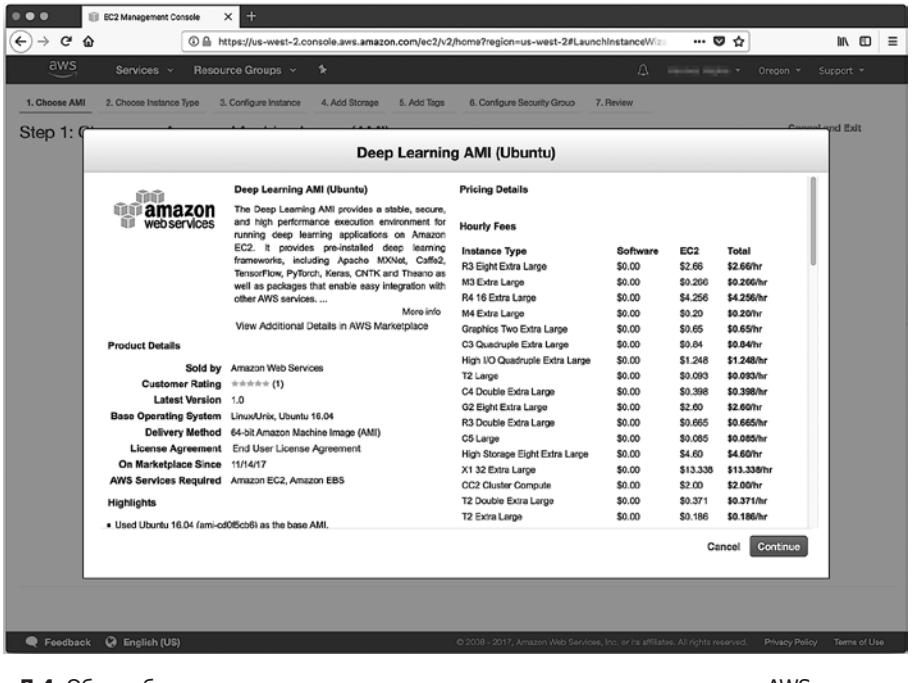
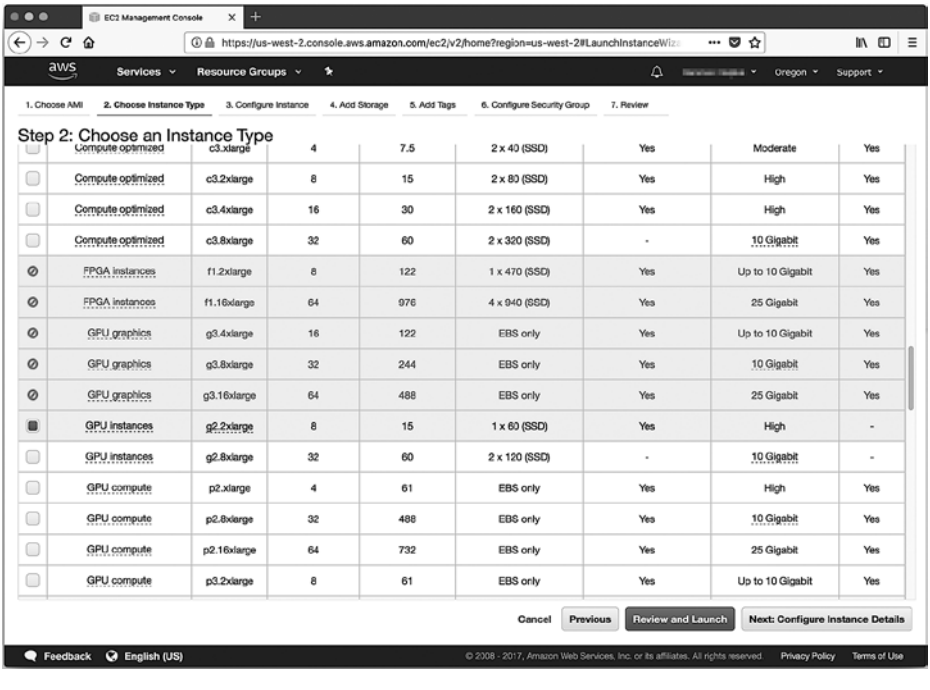


Рис. Д.4. Обзор образов машин и доступных типов экземпляров в вашем регионе AWS по стоимости

6. Многие AMI с открытым исходным кодом, например Deep Learning AMI (Ubuntu), бесплатны, так что в столбце стоимости Software (ПО) на странице More Info (Дополнительная информация) на Amazon Marketplace будет значиться $0. Программное обеспечение же других AMI в закладке Amazon Marketplace, например AMI RocketML, может быть не бесплатным. Вне зависимости от стоимости программного обеспечения, необходимо платить за время работы экземпляра сервера, если оно превышает квоту бесплатного пакета. Экземпляры с GPU в бесплатном пакете отсутствуют. Так что сначала полностью протестируйте свой конвейер на дешевой машине с CPU, прежде чем запускать его на более дорогостоящем экземпляре. Если вы смотрите на список цен — нажмите синюю кнопку Continue (Продолжить) (см. рис. Д.4). Если же вы вернулись к спискам AMI на Amazon Marketplace, то можете нажать синюю кнопку Select (Выбрать) рядом с AMI, который бы хотели инсталлировать на своем EC2-экземпляре, в результате чего вы перейдете на Step 2: Choose an Instance Type (Шаг 2: выбор типа экземпляра) (рис. Д.5).



Compute optimized	c3.xlarge	4	7.5	2 x 40 (SSD)	Yes	Moderate	Yes
Compute optimized	c3.2xlarge	8	15	2 x 80 (SSD)	Yes	High	Yes
Compute optimized	c3.4xlarge	16	30	2 x 160 (SSD)	Yes	High	Yes
Compute optimized	c3.8xlarge	32	60	2 x 320 (SSD)	-	10 Gigabit	Yes
FPGA instances	f1.2xlarge	8	122	1 x 470 (SSD)	Yes	Up to 10 Gigabit	Yes
FPGA instances	f1.16xlarge	64	976	4 x 940 (SSD)	Yes	25 Gigabit	Yes
GPU graphics	g3.4xlarge	16	122	EBS only	Yes	Up to 10 Gigabit	Yes
GPU graphics	g3.8xlarge	32	244	EBS only	Yes	10 Gigabit	Yes
GPU graphics	g3.16xlarge	64	488	EBS only	Yes	25 Gigabit	Yes
GPU instances	g2.2xlarge	8	15	1 x 60 (SSD)	Yes	High	-
GPU instances	g2.8xlarge	32	60	2 x 120 (SSD)	-	10 Gigabit	-
GPU compute	p2.xlarge	4	61	EBS only	Yes	High	Yes
GPU compute	p2.8xlarge	32	488	EBS only	Yes	10 Gigabit	Yes
GPU compute	p2.16xlarge	64	732	EBS only	Yes	25 Gigabit	Yes
GPU compute	p3.2xlarge	8	61	EBS only	Yes	Up to 10 Gigabit	Yes

Рис. Д.5. Выбор типа экземпляра

7. На этом шаге выбирается тип сервера для виртуальной машины (см. рис. Д.5). Неплохим вариантом будет минимальный экземпляр с GPU — `g2.2xlarge`. Темный паттерн[1] UI Amazon по умолчанию предлагает более дорогой тип. Вам

[1] См. https://en.wikipedia.org/wiki/Dark_pattern. — *Примеч. пер.*

придется вручную выбрать экземпляр `g2.2xlarge`, если вы хотите использовать именно его. Обратите также внимание, что виртуальные машины намного дешевле, если выбрать в качестве региона US West 2 (Oregon) (Запад США 2 (Орегон)), а не другие регионы США. Выбрать его можно в меню в правом верхнем углу страницы, около названия учетной записи.

8. После выбора желаемого типа экземпляра можно запустить машину, нажав синюю кнопку Review and Launch (Обзор и запуск). Но если это ваш первый экземпляр, лучше пройтись по всем шагам мастера настройки, чтобы знать, какие опции доступны, даже если на каждой из экранных форм вы выберете значение по умолчанию. Для перехода на следующий шаг нажмите серую кнопку Next: Configure Instance Details (Далее: настройка параметров экземпляра).
9. Здесь вы сможете настроить параметры экземпляра (рис. Д.6). Если вы применяли ранее машины AWS в уже существующем *виртуальном частном облаке* (virtual private cloud, VPC), то можете подключить свою машину с GPU к своему VPC. Машины, относящиеся к одному VPC, могут использовать один шлюз или сервер-бастион этого VPC для доступа к вашей машине. Но если это ваш первый EC2-экземпляр или у вас нет сервера-бастиона[1], можете об этом не думать.

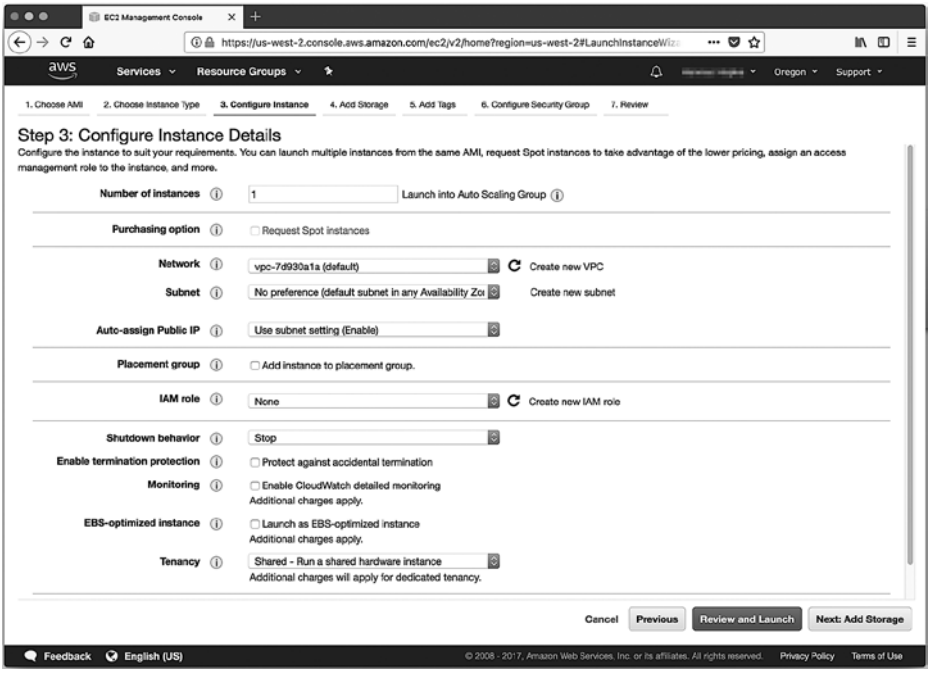


Рис. Д.6. Настройка параметров экземпляра

[1] В документации Amazon есть руководство по рекомендуемым практикам для узлов-бастионов (https://docs.aws.amazon.com/quickstart/latest/linux-bastion/architecture.html).

10. При выборе опции Protect against accidental termination (Защита от случайного завершения работы экземпляра) случайно завершить работу экземпляра будет сложнее. В Amazon Web Services «завершить» (terminate) означает выключить машину и очистить ее хранилище. «Остановить» же (stop) означает выключить машину или приостановить ее работу с сохранением всей содержащейся в постоянном хранилище на этой машине информации о контрольных точках обучения.
11. Для продолжения нажмите кнопку Next: Add Storage (Далее: добавить хранилище).
12. На этом шаге (рис. Д.7) можно добавить хранилище, если вы собираетесь работать с большими корпусами информации. Но лучше начинать работу с минимальным размером локального хранилища на EC2-экземпляре и ждать подключения Amazon S3 Bucket или другого сервиса облачного хранилища после запуска вашего EC2-экземпляра. Благодаря этому вы сможете совместно использовать большие наборы данных в нескольких серверах или циклах обучения (между завершениями работы экземпляра). Amazon Web Services требует платы за любое локальное хранилище EC2 свыше 30 Гбайт бесплатного пакета. В UX AWS есть множество темных паттернов, из-за которых так сложно избежать роста денежных расходов.

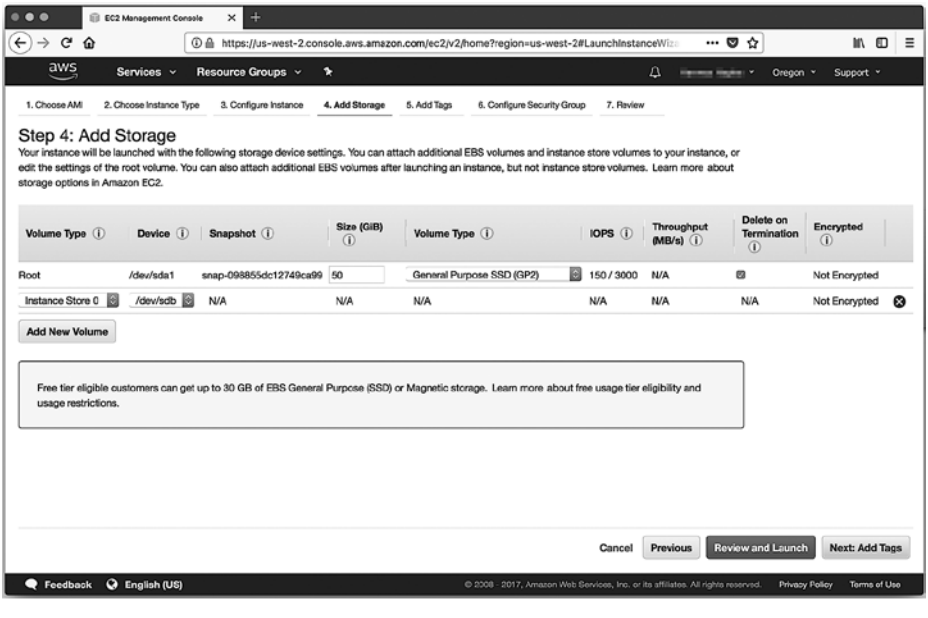


Рис. Д.7. Добавление для экземпляра постоянного хранилища

13. Пройдитесь по следующим шагам, нажимая кнопки Next и изучая назначаемые по умолчанию вашему EC2-экземпляру теги и группы безопасности. Нажатие последней кнопки Next приведет к тому, что вы вернетесь на шаг обзора всех настроек (рис. Д.8).

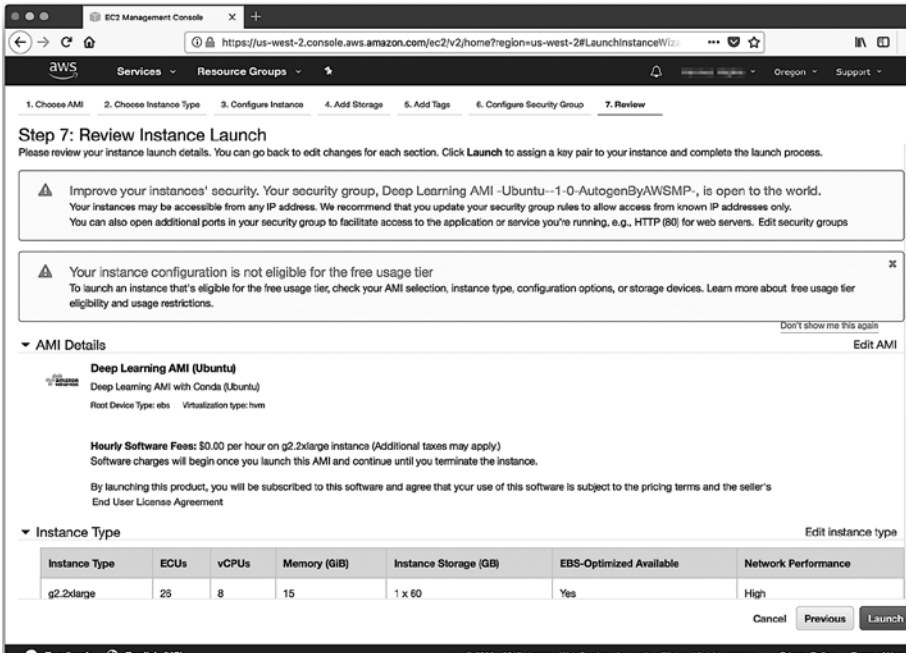


Рис. Д.8. Просмотр настроек экземпляра перед запуском

14. В экранной форме обзора настроек (см. рис. Д.8) Amazon Web Services показывают сводку всех подробностей настроек экземпляра.
15. Прежде чем нажимать кнопку Launch (Запуск), убедитесь, что настройки экземпляра — особенно его тип (RAM и CPU), образ AMI (Deep Learning Ubuntu) и хранилище (достаточно места для ваших данных) — вас устраивают. На этом этапе AWS запустит виртуальную машину и начнет загружать в нее образ программного обеспечения.
16. AWS, если вам еще не приходилось создавать с его помощью экземпляры, попросит вас создать новую пару ключей (рис. Д.9). Она позволит вам заходить на машину через ssh без пароля. По умолчанию EC2-экземпляры не допускают входа с паролем, поэтому вам придется сохранить файл `.pem` в каталоге `$HOME/.ssh/` и держать его копию в надежном месте (например, в своем диспетчере паролей), иначе вы не сможете получить доступ к своему работающему серверу и придется начинать все заново.
17. После сохранения пары ключей (если вы создали новую пару ключей) AWS подтвердит, что экземпляр запущен. В редких случаях в ЦОД Amazon может не оказаться запрошенных вами ресурсов и будет возвращена ошибка, в результате чего вам придется повторить все с начала.
18. Щелкните на хеше экземпляра, который начинается с *i-*... (рис. Д.10). Эта ссылка выведет обзор всех ваших EC2-экземпляров, где должен присутствовать ваш новый экземпляр с состоянием running («работает») или initializing («инициализация»).

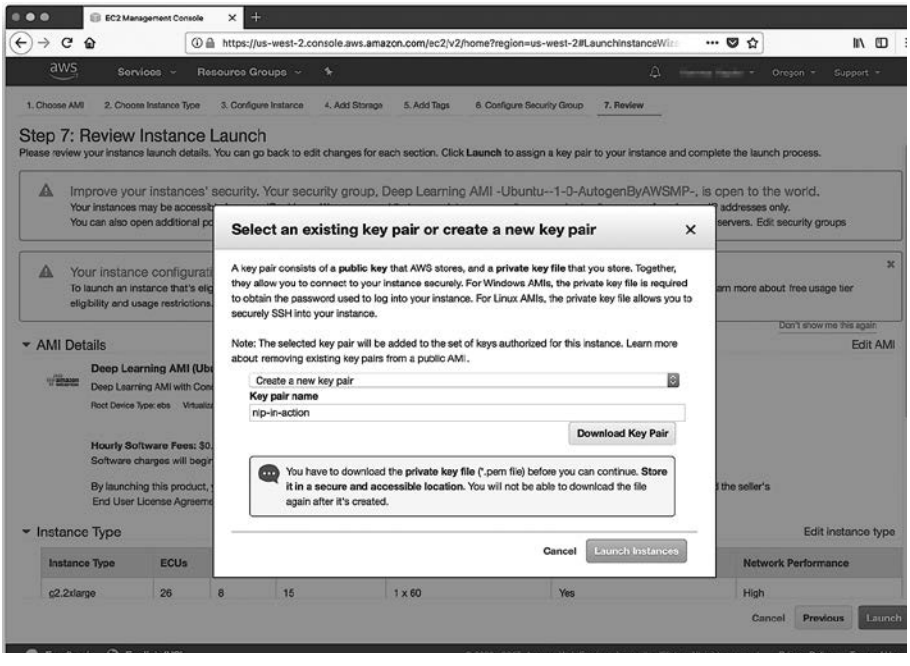


Рис. Д.9. Создание нового ключа экземпляра (или загрузка уже существующего)

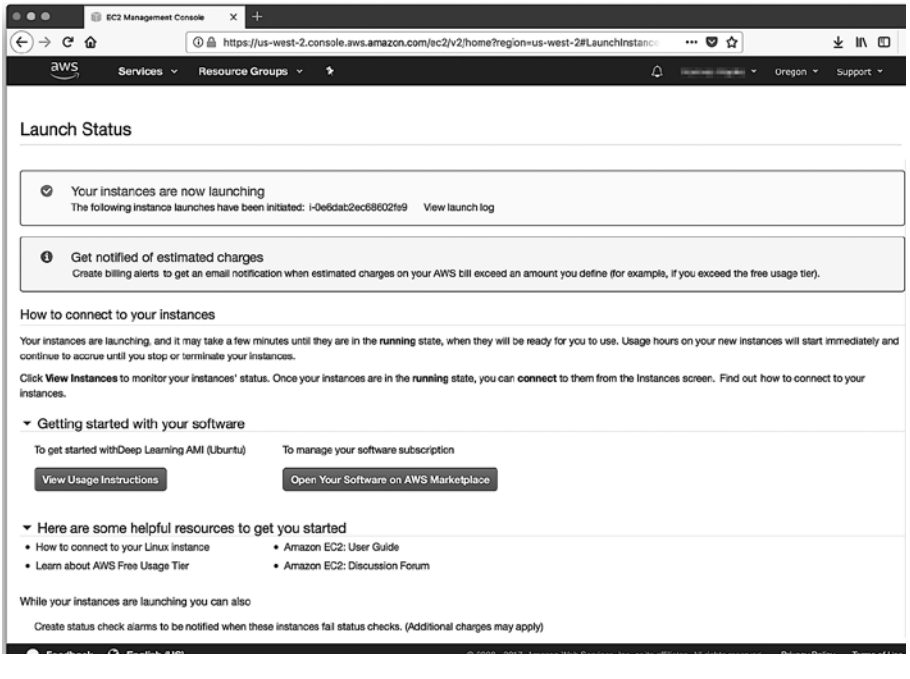


Рис. Д.10. Подтверждение AWS запуска экземпляра

19. Помимо файла `.pem` для сгенерированной ранее пары ключей вам понадобится публичный IP-адрес вашего экземпляра (рис. Д.11). Лучше хранить его в диспетчере паролей, вместе с файлом `.pem`. Его также лучше указать в файле `$HOME/.ssh/config`, чтобы можно было задать имя хоста для вашего экземпляра и не искать этот IP-адрес в будущем.

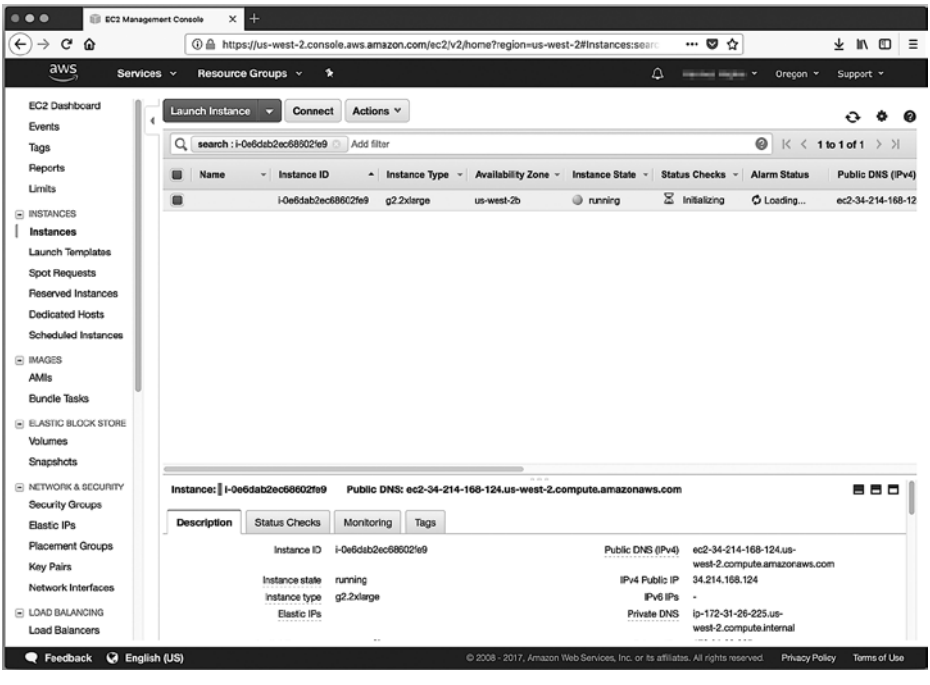


Рис. Д.11. Только что созданный экземпляр на панели инструментов EC2

Типичный файл конфигурации выглядит примерно так, как показано в листинге Д.1. Значение параметра `HostName` необходимо заменить на публичный IP-адрес (с панели инструментов EC2) или полное доменное имя (с вашей панели инструментов Route 53 в AWS) вашего только что запущенного EC2-экземпляра.

Листинг Д.1. $HOME/.ssh/config

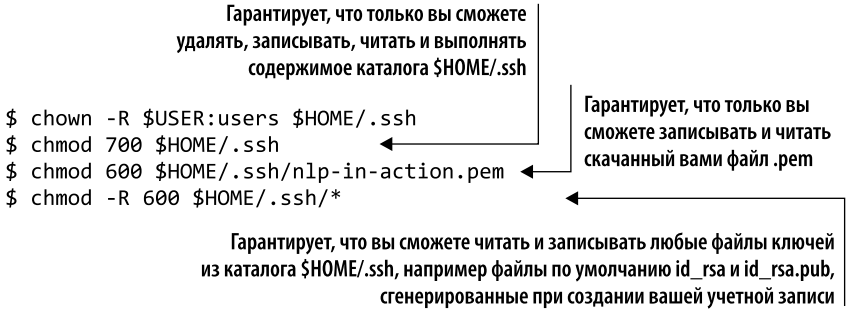
```
Host totalgood
    User ubuntu
    HostName INSTANCE_PUBLIC_IP
    Port 22
    IdentityFile ~/.ssh/nlp-in-action.pem
    # ssh -i ~/.ssh/nlp-in-action.pem ubuntu@INSTANCE_PUBLIC_IP
```

Замените INSTANCE_PUBLIC_IP своим публичным IP-адресом

Здесь необходимо указать путь к загруженному файлу .pem

В файле конфигурации можно оставлять свои примечания в виде комментариев

20. Для входа в экземпляр AWS `ssh` требует, чтобы файл приватного ключа (файл `.pem` в каталоге `$HOME/.ssh`) был доступен для чтения только вам и суперпользователю системы. Задать соответствующие права доступа можно с помощью следующих команд командного процессора bash[1]:



21. После установки соответствующих прав доступа файлов и настройки файла конфигурации выполните следующую команду bash для входа в свой EC2-экземпляр:

```
$ ssh -i ~/.ssh/nlp-in-action.pem ubuntu@INSTANCE_PUBLIC_IP
```

22. Если образ машины Amazon основан на операционной системе Ubuntu, имя пользователя обычно будет `ubuntu`. Но для каждого AMI имеется документация, в которой описываются имя пользователя и номер порта `ssh` для входа.
23. При входе в экземпляр в первый раз вы увидите предупреждение о незнакомом цифровом отпечатке машины (рис. Д.12). Для продолжения процесса входа в систему введите `yes`[2].

Рис. Д.12. Запрос подтверждения для обмена учетными данными ssh

[1] Командный процессор bash, как и cygwin или git-bash, необходимо отдельно инсталлировать, чтобы команды ssh bash работали в операционной системе Windows.

[2] Если вы увидите это предупреждение, причем если вы не меняли IP-адрес, то, вероятно, кто-то пытается подменить IP-адрес или доменное имя вашей машины для несанкционированного доступа к вашему экземпляру с помощью атаки типа «человек посередине». Это бывает исключительно редко.

24. После успешного входа вы увидите экран приветствия (рис. Д.13).

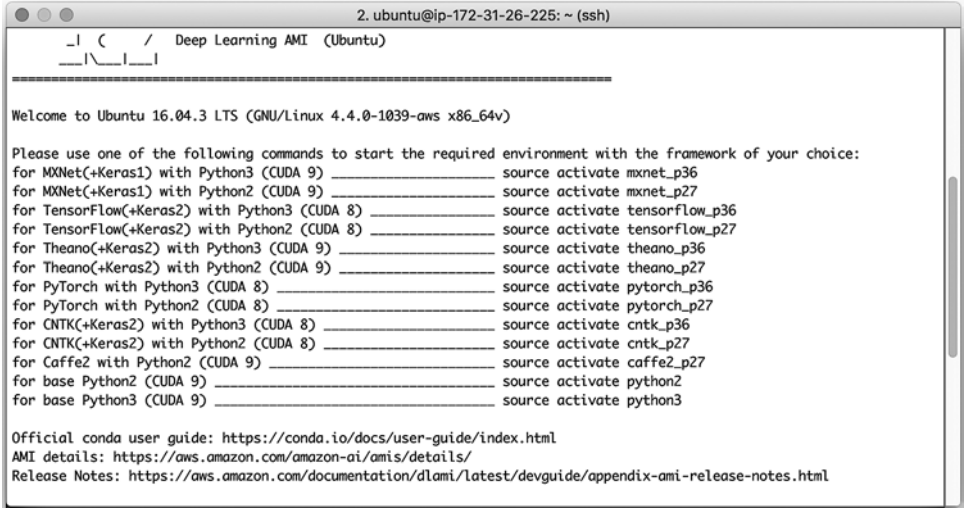


Рис. Д.13. Экран приветствия, отображаемый после успешного входа

25. И наконец, необходимо активировать желаемую среду разработки. В данном образе машины имеется несколько сред, включая `PyTorch`, `TensorFlow` и `CNTK`. Поскольку в этой книге используются TensorFlow и Keras, следует активировать среду `tensorflow_p36`. При этом загружается виртуальная среда с установленными Python 3.6, Keras и TensorFlow (рис. Д.14):

```
$ source activate tensorflow_p36
```

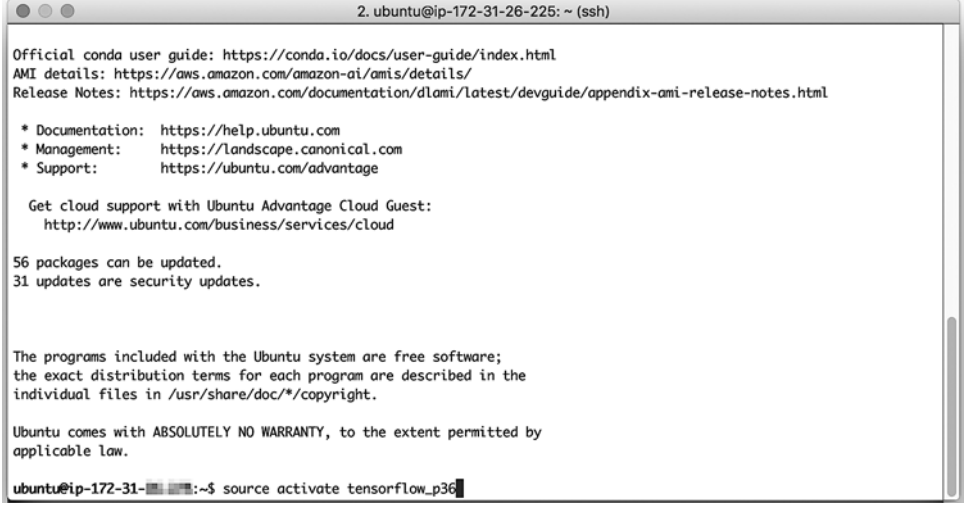


Рис. Д.14. Активация предустановленной среды Keras

После активации среды TensorFlow можно приступать к обучению моделей глубокого обучения NLP. Для этого перейдите в командную оболочку iPython с помощью команды:

```
$ ipython
```

Теперь можете обучать свои модели. Удачи!

Д.1.1. Контроль затрат

Работа экземпляра с GPU на облачном сервисе вроде AWS может влететь в копеечку. Самый маленький экземпляр с GPU в регионе US-West 2 обходится в $0,65 в час (на момент написания данной книги). Обучение простой модели преобразования последовательностей в последовательности может занять несколько часов, после чего еще может понадобиться произвести несколько итераций для подбора параметров модели. Эти итерации могут привести к солидному счету в конце месяца. Минимизировать неприятные сюрпризы можно благодаря нескольким мерам предосторожности (рис. Д.15, Д.16).

- Отключайте простаивающие машины с GPU. При останове (не завершении) машины последнее состояние хранилища (за исключением каталога `/tmp`) сохраняется, и к нему впоследствии можно вернуться. Данные, содержащиеся в оперативной памяти, теряются, так что не забудьте сохранять все контрольные точки модели перед остановом машины.

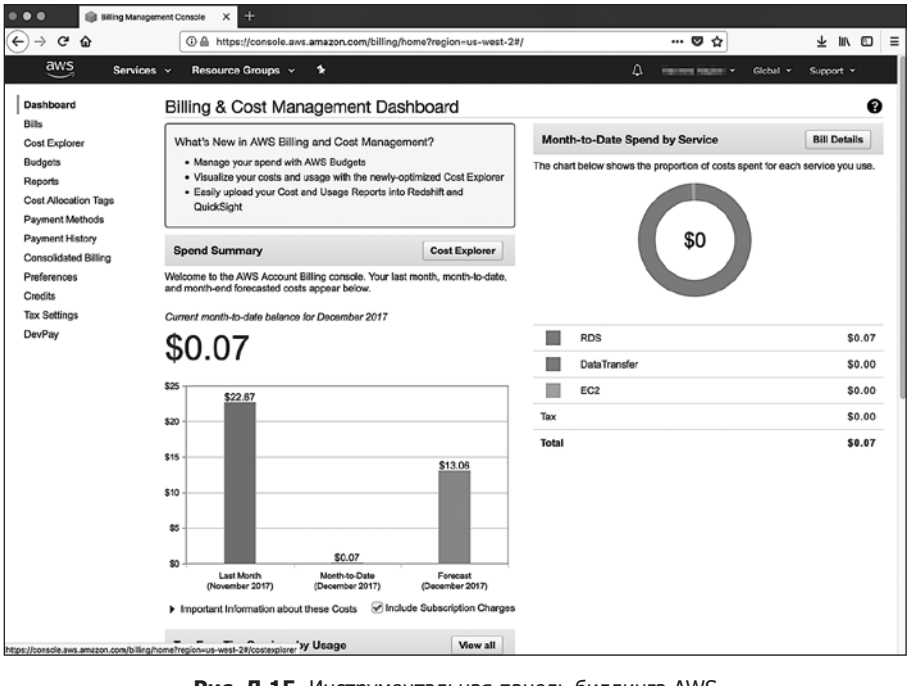


Рис. Д.15. Инструментальная панель биллинга AWS

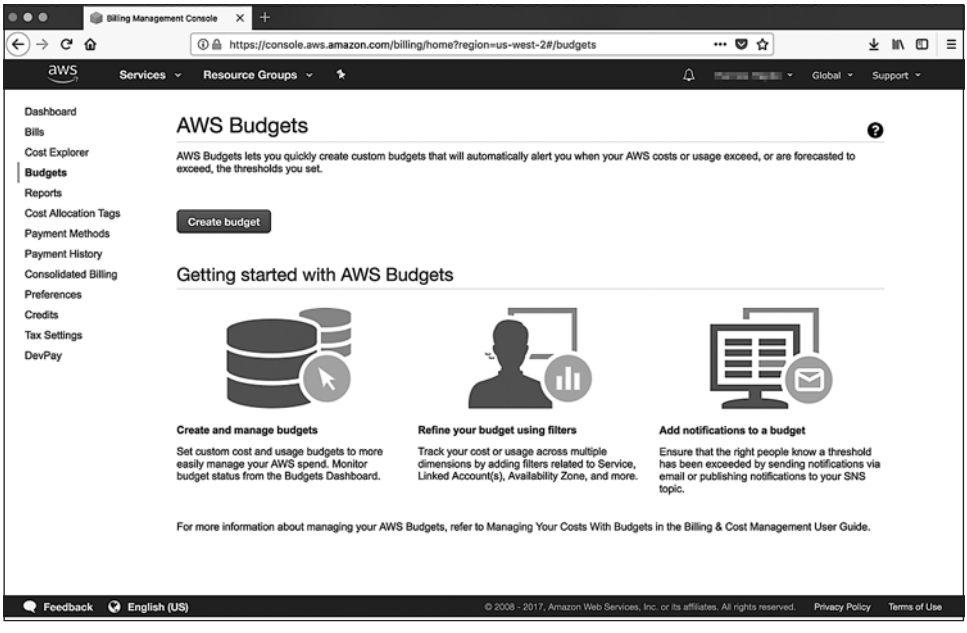


Рис. Д.16. Консоль бюджета AWS

- Заглядывайте на страницы сводной информации об EC2-экземпляре для всех запущенных экземпляров.
- Регулярно отслеживайте работающие экземпляры в сводке счета AWS.
- Создайте AWS Budget с автоматическими предупреждениями о затраченных денежных ресурсах. После того как вы настроите бюджет, AWS будет предупреждать вас о его превышении.

Приложение Е
Хеширование с учетом локальности

Из главы 4 вы узнали, как создавать векторы тем, содержащие сотни измерений с вещественными (с плавающей точкой) значениями. В главе 6 рассказано, как создавать векторы слов с сотнями измерений. Но хотя вы уже знаете, как выполнять полезные математические операции над этими векторами, вы все еще не можете производить быстрый поиск по ним, подобно поиску по дискретным векторам или строкам. В базах данных отсутствуют эффективные схемы поиска по более чем четырехмерным векторам[1]. Для эффективного использования векторов слов и векторов «документ — тема» нужен поисковый индекс для поиска ближайших соседей любого заданного вектора.

Это нужно для преобразования результатов векторной математики в слова или наборы слов (поскольку полученный в результате вектор никогда не будет точно соответствовать вектору для какого-либо слова из английского языка). Нужно это и для семантического поиска. В данном приложении мы покажем вам пример такого подхода, основанного на хешировании с учетом локальности (locality sensitive hashing, LSH).

---

[1] Некоторые продвинутые базы данных, например PostgreSQL, умеют индексировать векторы и более высокой размерности, но эффективность этих индексов быстро падает с ростом размерности.

E.1. Векторы высокой размерности принципиально различны

При переходе от одномерных векторов к двумерным и даже трехмерным ничего особо не меняется в смысле математических методов быстрого поиска ближайших соседей. Поговорим немного о традиционных подходах, применяемых в поисковых индексах баз данных для двумерных векторов, и постепенно перейдем к многомерным. Благодаря этому вы увидите своими глазами, в каком месте такая математика перестает работать (точнее, перестает быть пригодной для использования на практике).

E.1.1. Индексы и хеши векторного пространства

Индексы предназначены для упрощения поиска. Индексы типа `float` (предназначенные для долготы и широты) не могут основываться на точном совпадении, в отличие от индекса (указателя) слов в конце учебника. В большинстве предназначенных для двумерных вещественнозначных данных индексов используется какая-либо разновидность ограничивающих прямоугольников для разбиения пространства низкой размерности на куски приемлемых размеров. Самый распространенный пример подобного индекса для двумерной информации о географическом местоположении — системы почтовых индексов, используемые в различных странах для распределения почтовых адресов по примыкающим друг к другу зонам (в США они называются ZIP-кодами).

Каждой области в двумерном пространстве присваивается число; и хотя охватываемые почтовыми индексами зоны не являются прямоугольными ограничивающими рамками, они служат для той же цели. Военные применяют сетку ограничивающих прямоугольников для разбиения поверхности земного шара на прямоугольные области, каждой из которых присваивается число. Как в ZIP-кодах США, так и военной сеточной системе, у номеров этих областей есть семантический смысл.

В основе учета локальности (locality sensitivity) для таких хешей, как ZIP-коды США, лежит идея близости друг к другу и в двумерном пространстве чисел (хешей), близких по порядковому номеру. Например, первая цифра в ZIP-кодах США обозначает конкретный регион в США, например Западный берег, или Юго-Запад, или штат США. Следующая цифра (вместе с первой) однозначно идентифицирует конкретный штат. Первые три цифры однозначно идентифицируют район или большой город в этом штате. Учет локальности ZIP-кодов США распространяется вплоть до суффикса +4, определяющего конкретный квартал города или даже конкретное здание[1].

Алгоритм и выполнявшийся вручную процесс, в результате которого создана система ZIP-кодов США, эквивалентен алгоритмам хеширования с учетом ло-

[1] Этот учет локальности можно наглядно увидеть на карте из статьи «Википедии» ZIP Code: https://en.wikipedia.org/wiki/ZIP_Code#Primary_state_prefixes.

кальности, созданным для других векторных пространств. Они описывают способ генерации учитывающих локальность чисел. В них координаты местоположений в векторном пространстве используются таким образом, что числовые значения хешей оказываются близки друг к другу, если близки (или даже пересекаются) соответствующие им местоположения областей векторного пространства. Хеширование с учетом локальности нацелено на создание тех самых математических свойств, вроде высокой вероятности коллизий и учета локальности, которых так стараются избежать криптографические алгоритмы хеширования.

Е.1.2. Мыслим многомерно

Векторные пространства естественного языка многомерны. Естественный язык захватывает все сложные понятия, о которых думают и говорят люди, включая саму обработку естественного языка. И чтобы сжать все эту сложность в вектор и уместить в жесткие рамки заданного векторного пространства, зачастую часть сложности отбрасывается. И мы все добавляем в эти векторы все новые измерения, чтобы уместить в них столько человеческих мыслей и языка, сколько можно.

Например, в векторах мультимножеств слов не учитывается информация, заключенная в порядке слов. Благодаря этому получаются дискретные многомерные векторы с возможностью эффективной индексации и поиска. Чтобы обнаружить, присутствуют ли конкретные ключевые слова в запросе и корпусе, используются бинарный поиск и индексные деревья. Этот метод работает, даже если включить в словарь ключевых слов все слова естественного языка. Поисковые системы часто даже включают сразу все слова на сотнях естественных языков. Именно поэтому при поиске в Интернете можно использовать в одном запросе испанские и немецкие слова вместе с английскими.

Из главы 2 вы знаете, как захватить часть заключенной в порядке слов сложности с помощью добавления *n*-грамм в измерения векторов мультимножеств слов, из главы 3 — как задать весовые коэффициенты для этих миллионов термов (слов и *n*-грамм) в соответствии с их важностью. В результате модель векторного пространства естественных языков содержит миллионы измерений.

Но мультимножества слов, TF-IDF и регулярные выражения не способны понимать. Они могут только помочь в поиске нужных документов. Поэтому с 4-й по 6-ю главу мы сделали наши векторные пространства непрерывными. Мы втиснули часть многогранности естественного языка в промежутки между целочисленными значениями количеств вхождений слов. При описании измерений нашего векторного пространства мы больше не привязаны к жесткому дискретному словарю. Мы понизили размерность векторов с миллионов до сотен за счет группировки слов в понятия (темы). Мы создали ness-векторы, способные захватывать *femaleness*, *blueness* и *placeness* слов и высказываний.

И это еще не все. В главах с 7-й по 10-ю мы выяснили, как захватывать в векторах не просто словосочетания, а длинные, сложные последовательности слов. Благодаря рекуррентным нейронным сетям и сетям с долгой краткосрочной памятью наши поверхностные ness-векторы превратились в глубокие векторы идей.

Но вся эта глубина и сложность приводит к проблеме. Эффективная индексация и поиск по таким непрерывным, плотным, многомерным векторам, как векторы идей, невозможны. Именно поэтому поисковые системы не могут просто ответить на сложные вопросы за миллисекунду. Чтобы узнать смысл жизни, вам придется поговорить с чат-ботом или, боже упаси, с другим человеком.

Почему? Почему нельзя индексировать или искать многомерные непрерывные векторы? Начнем с одномерного векторного пространства и увидим, насколько просто индексировать и искать отдельное скалярное значение на одномерной числовой оси. А дальше посмотрим, как расширить этот одномерный индекс на многомерный случай. И размерность можно наращивать и далее, пока все не перестанет работать.

Одномерный индекс

Представьте себе случайное распределение одномерных векторов, то есть просто набор случайных чисел. Можно создать естественный ограничивающий отрезок, отсекающий половину ширины всего пространства путем разбиения числовой прямой напополам. Все положительные числа будут внутри одного отрезка, а отрицательные — внутри другого. Если известно, где находится середина (центроид) векторного пространства (обычно 0), можно сделать так, чтобы каждая из рамок содержала примерно половину векторов.

Каждую из этих рамок можно снова разбить напополам, в результате чего получится четыре отрезка. Если продолжить этот процесс, через несколько итераций получится бинарное дерево поиска — бинарный хеш с учетом локальности (местоположения). Среднее число точек в каждом отрезке для одномерного векторного пространства составляет `pow(число_векторов/2, число_отрезков)`. Для одномерного пространства достаточно около 32 уровней (размеров отрезков) для индексации миллиардов точек таким образом, чтобы в каждом отрезке находилось лишь несколько из них.

У каждого из наших одномерных векторов теперь есть свой собственный ZIP-код (индекс или учитывающий локальность хеш). И все похожие друг на друга векторы будут находиться неподалеку, в отсортированном списке этих хеш-значений. Таким образом можно вычислять хеш-значения для новых запросов и быстро находить их в базе данных.

Двумерные, трехмерные и четырехмерные индексы

Добавим еще одно измерение и посмотрим, насколько хорошо работает наш одномерный индекс на основе бинарного дерева. Вспомните, как мы разбивали пространство на отрезки для бинарного дерева, разделяя каждый отрезок примерно напополам при каждом ветвлении дерева. Какое измерение теперь мы будем разбивать напополам для уменьшения количества числовых точек вдвое? Для двумерного вектора получится 2×2 квадратов (квадранты двумерной плоскости). Для трехмерного вектора — $3 \times 3 \times 3$ блоков в виде пространственного кубика Рубика. Для четырехмерного понадобится $4 \times 4 \times 4 \times 4$ блоков... для начала. Первое же ветвление индекса на основе бинарного дерева приведет к созданию 4^4^ веток. При-

чем некоторые из 256 ограничивающих гиперпараллелепипедов в четырехмерном векторном пространстве могут вообще не содержать никаких векторов. Некоторые сочетания и последовательности слов вообще никогда не встречаются.

Наш наивный подход с индексами на основе бинарных деревьев нормально работает в случае трехмерных и четырехмерных векторов, вплоть даже до восьмимерных или более. Но он быстро становится неуправляемым и неэффективным. Представьте себе, как будут выглядеть наши ограничивающие гиперкубы в десятимерном пространстве. Не получается представить? Вы не одиноки в этом. Человеческий мозг оперирует трехмерными образами и не способен полностью воспринимать концепции даже четырехмерного векторного пространства.

Машины вполне способны обрабатывать десятимерные векторы, но нам нужно обрабатывать с их помощью 100-мерные или более, чтобы сжать в векторы всю многогранность людских мыслей. Это проклятие размерности можно рассматривать в таких аспектах, как:

- экспоненциальный рост числа возможных сочетаний измерений с каждым новым измерением;
- удаленность всех векторов друг от друга в многомерном пространстве;
- то, что многомерные векторные пространства состоят по большей части из пустоты — выбранный случайным образом ограничивающий гиперпараллелепипед практически всегда будет пустым.

Следующий код должен помочь вам лучше прочувствовать свойства многомерных пространств (листинг Е.1).

Листинг Е.1. Исследуем многомерные пространства

```
>>> import pandas as pd
>>> import numpy as np
>>> from tqdm import tqdm

>>> num_vecs = 100000
>>> num_radii = 20
>>> num_dim_list = [2, 4, 8, 18, 32, 64, 128]
>>> radii = np.array(list(range(1, num_radii + 1)))
>>> radii = radii / len(radii)
>>> counts = np.zeros((len(radii), len(num_dims_list)))
>>> rand = np.random.rand

>>> for j, num_dims in enumerate(tqdm(num_dim_list)):
...     x = rand(num_vecs, num_dims)
...     denom = (1. / np.linalg.norm(x, axis=1))
...     x *= denom.reshape(-1, 1).dot(np.ones((1, x.shape[1])))
...     for i, r in enumerate(radii):
...         mask = (-r < x) & (x < r)
...         counts[i, j] = (mask.sum(axis=1) == mask.shape[1]).sum()
```

Нормализация таблицы случайных векторов-строк к единичной длине

Вы можете изучить странный мир многомерных пространств подробнее в `nlpia/book/examples/ch_app_h.py` на GitHub (http://github.com/totalgood/nlpia). Можете также

увидеть часть этой странности в следующей таблице, где показана плотность точек во всех ограничивающих гиперкубах по мере увеличения размера:

```
>>> df = pd.DataFrame(counts, index=radii, columns=num_dim_list) / num_vecs
>>> df = df.round(2)
>>> df[df == 0] = ''
>>> df
         2     4     8    18    32    64   128
0.05
0.10
0.15                                       0.36
0.20                                  0.1     1
0.25                                    1     1
0.30                             0.55     1     1
0.35                       0.11  0.98     1     1
0.40                       0.63     1     1     1
0.45                  0.03 0.92     1     1     1
0.50                   0.2 0.99     1     1     1
0.55         0.01  0.49     1     1     1     1
0.60         0.08  0.75     1     1     1     1
0.65         0.24  0.89     1     1     1     1
0.70         0.44  0.96     1     1     1     1
0.75  0.12   0.64  0.98     1     1     1     1
0.80  0.25   0.78     1     1     1     1     1
0.85  0.38   0.88     1     1     1     1     1
0.90  0.52   0.94     1     1     1     1     1
0.95  0.67   0.98     1     1     1     1     1
1.00     1     1     1     1     1     1     1
```

Существует алгоритм индексации под названием «k-мерное дерево» (https://ru.wikipedia.org/wiki/K-мерное_дерево), нацеленный на максимально эффективное (в смысле минимизации числа пустых прямоугольных гиперпараллелепипедов) разбиение многомерных пространств. Но даже подобные подходы перестают работать в случае десятков или сотен измерений, когда вступает в игру проклятие размерности. В отличие от двумерных или трехмерных векторов невозможно по-настоящему индексировать или хешировать многомерные векторы слов и идей так, чтобы быстро находить ближайшие соответствия. Для нахождения нескольких действительно ближайших соседей приходится вычислять расстояния к множеству векторов-кандидатов. Или даже проверять все, чтобы не пропустить чего-нибудь.

E.2. Многомерная индексация

В многомерном пространстве обычные индексы на основе ограничивающих прямоугольников не работают. В конце концов даже хеширование с учетом локальности перестает работать. Но сначала поэкспериментируем с хешированием с учетом локальности и покажем его ограничения, а затем научимся обходить эти ограничения, просто отказавшись от идеи идеального индекса. После эксперимента с учитывающим локальность хешированием мы создадим приближенный индекс.

Е.2.1. Хеширование с учетом локальности

На рис. Е.1 мы сформировали 400 000 совершенно случайных 200-мерных векторов (типичные показатели для векторов тем для большого корпуса). И проиндексировали их с помощью пакета `LSHash` языка Python (`pip install lshash3`). Теперь представьте себе, что у нас есть поисковая система, стремящаяся найти все векторы тем, близкие к вектору темы запроса. Сколько таких векторов наберется при использовании учитывающего локальность хеша? При какой размерности векторов тем результаты поиска потеряют всякий смысл?

D	N	100th Cosine Distance	Top 1 Correct	Top 2 Correct	Top 10 Correct	Top 100 Correct
2	4254	0	TRUE	TRUE	TRUE	TRUE
3	7727	0.0003	TRUE	TRUE	TRUE	TRUE
4	12198	0.0028	TRUE	TRUE	TRUE	TRUE
5	9920	0.0143	TRUE	TRUE	TRUE	TRUE
6	11310	0.0166	TRUE	TRUE	TRUE	TRUE
7	12002	0.0246	TRUE	TRUE	TRUE	FALSE
8	11859	0.0334	TRUE	TRUE	TRUE	FALSE
9	6958	0.0378	TRUE	TRUE	TRUE	FALSE
10	5196	0.0513	TRUE	TRUE	FALSE	FALSE
11	3019	0.0695	TRUE	TRUE	TRUE	FALSE
12	12263	0.0606	TRUE	TRUE	FALSE	FALSE
13	1562	0.0871	TRUE	TRUE	FALSE	FALSE
14	733	0.1379	TRUE	FALSE	FALSE	FALSE
15	6350	0.1375	TRUE	TRUE	FALSE	FALSE
16	10980	0.0942	TRUE	TRUE	FALSE	FALSE

Рис. Е.1. Семантический поиск с помощью LSHash

Если размерность существенно превышает десять или около того, число правильных результатов поиска резко падает. Если хотите поэкспериментировать сами или попытаться создать лучший алгоритм LSH, код для выполнения подобных экспериментов доступен в пакете `nlpia`. Код пакета `lshash3` вообще открыт, его сердцевина состоит всего примерно из 100 строк кода.

Е.2.2. Приближенный метод поиска ближайших соседей

Приближенный поиск ближайших соседей — новейшее решение проблемы многомерных векторных пространств. Приближенные хеш-значения аналогичны учитывающим локальность хешам и k-мерным деревьям, но в их основе лежит нечто напоминающее скорее алгоритм случайного леса. Они представляют собой стохастические (случайные) алгоритмы разбиения векторного пространства на все меньшие и меньшие куски пространства.

Наиболее современные подходы к поиску ближайших соответствий многомерным векторам — пакет FAISS от Facebook и пакет Annoy от Spotify. Мы выбрали для нашего чат-бота пакет Annoy из-за простоты инсталляции и использования. Он интенсивно используется не только для поиска соответствий по векторам метаданных песен для меломанов, издательство Dark Horse Comics применяет Annoy для эффективной рекомендации комиксов. Мы упоминали эти инструменты в главе 13.

E.3. Предсказание лайков

На рис. E.2 показано, как выглядит в гиперпространстве набор твитов. Эти изображения представляют собой двумерные «тени» 100-мерных векторов тем (точек) твитов, полученных путем латентно-семантического анализа этих твитов. Большая часть пятен соответствует твитам, которые получили лайк хотя бы один раз; меньшая часть пятен — получившим ноль лайков твитам.

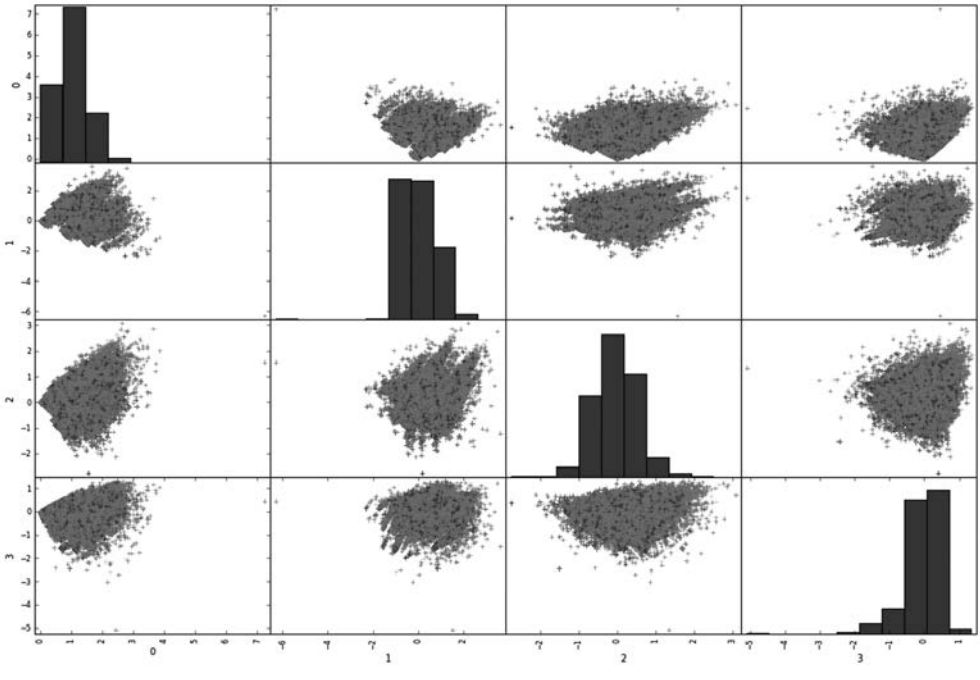


Рис. E.2. Матрица рассеяния для четырех тем твитов

Подгонка модели LDA к этим векторам тем в 80 % случаев будет успешной. Впрочем, как и в случае с набором данных CMC, наша база твитов очень плохо сбалансирована. Так что предсказания вероятности получения новыми твитами лайков с помощью этой модели вряд ли окажутся очень точными. Для следующих

задач классификации, в которых полезна максимизация дисперсии (разделяемости классов), лучше использовать языковые модели LSA, LDA и LDiA:

- семантический анализ;
- анализ тональности;
- обнаружение спама.

Для более тонкого различения текстов на основе обобщения сходства семантического содержания понадобятся наиболее современные инструменты NLP из вашего арсенала. Имеет смысл использовать модели глубокого обучения LSTM и методы понижения размерности t-SNE для решения таких сложных проблем, как:

- предсказание реакции человека (например, вероятности получения лайка);
- машинный перевод;
- генерация естественного языка.

Источники информации

При написании данной книги мы черпали информацию из разных источников. Вот некоторые из наших любимых.

В идеальном мире можно было бы найти эти источники информации, просто введя их названия в семантическую поисковую систему вроде Duck Duck Go (http://duckduckgo.com), Gigablast (http://gigablast.com/search?c=main&q=open+source+search+engine) или Qwant (https://www.qwant.com/web). Но до тех пор, пока Джимми Уэйлс (Jimmy Wales) не доработает свой Wikia Search (https://ru.wikipedia.org/wiki/Wikia_Search) или Google не поделится своей технологией NLP, придется использовать списки ссылок в стиле 1990-х годов, подобные нижеприведенным. Загляните в раздел «Поисковые системы», если хотите сделать свой вклад в спасение мира, оказав помощь проектам индексации Интернета с открытым исходным кодом.

Приложения и идеи проектов

Вот несколько примеров, которые могут вдохновить вас на свои собственные проекты NLP.

- *Guessing passwords from social network profiles* («Подбор паролей из профилей соцсетей», http://www.sciencemag.org/news/2017/09/artificial-intelligence-just-made-guessing-your-password-whole-lot-easier).
- *Chatbot lawyer overturns 160,000 parking tickets in London and New York* («Чат-бот юрист оспорил 160 000 штрафов за парковку в Лондоне и Нью-Йорке», http://www.theguardian.com/technology/2016/jun/28/chatbot-ai-lawyer-donotpay-parking-tickets-london-new-york).

- *GitHub — craigboman/gutenberg: Librarian working with project gutenberg data, for NLP and machine learning purposes* («GitHub — craigboman/gutenberg: Библиотекарь, работающий с данными проекта Гутенберга, для целей НЛП и машинного обучения», https://github.com/craigboman/gutenberg).
- *Longitudial Detection of Dementia Through Lexical and Syntactic Changes in Writing* («Многолетнее исследование по обнаружению деменции по лексическим и синтаксическим изменениям на письме», ftp://ftp.cs.toronto.edu/dist/gh/Le-MSc-2010.pdf) — магистерский диплом Сюаня Ле (Xuan Le) по психологической диагностике с помощью NLP.
- *Wang Z.* Time Series Matching: a Multi-filter Approach («Сопоставление временных рядов: многофильтровый подход», https://www.cs.nyu.edu/web/Research/Theses/wang_zhihua.pdf) — песни, аудиоклипы и другие временные ряды допускают дискретизацию и поиск по ним с помощью алгоритмов динамического программирования, аналогичных расстоянию Левенштейна.
- Never Ending Language Learning, NELL (непрерывное изучение языка) — постоянно растущая база знаний CMU, обучающаяся с помощью скрапинга текстов на естественном языке.
- *How the NSA identified Satoshi Nakamoto* («Как NSA идентифицировал Сатоши Накамото», https://slashdot.org/story/330553) — журнал *Wired* и NSA идентифицировали Сатоши Накамото с помощью NLP (стилометрии).
- Стилометрия (https://ru.wikipedia.org/wiki/Стилометрия) и установление авторства для целей экспертных исследований социальных сетей (http://www.parkjonghyuk.net/lecture/2017-2nd-lecture/forensic/s8.pdf) — сопоставление стиля/закономерностей и кластеризация текста на естественном языке (а также музыки и произведений искусства) с целью установления авторства и атрибуции.
- Можно производить скрапинг таких онлайн-словарей, как Your Dictionary (http://examples.yourdictionary.com/), и получать грамматически правильные предложения с POS-метками, подходящие для обучения вашего собственного синтаксического дерева и средства частеречной разметки Parsey McParseface (https://ai.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html).
- *Goldstein J., Ghoul M.* Identifying 'Fake News' with NLP («Выявление сфабрикованных новостей с помощью NLP», https://nycdatascience.com/blog/student-works/identifying-fake-news-nlp/) — Нью-Йоркская академия науки о данных (NYC Data Science Academy).
- SimpleNumericalFactChecker (https://github.com/uclmr/simpleNumericalFactChecker) Андреаса Влахоса (Andreas Vlachos) (https://github.com/andreasvlachos) и информационный поиск (см. главу 11) можно использовать для ранжирования издательств, авторов и журналистов по степени правдивости. Можно объединить этот подход со средством предсказания «сфабрикованных новостей» Джулии Голдштейн.
- Пакет artificial-adversary (https://github.com/airbnb/artificial-adversary) Джека Дая (Jack Dai), стажера в Airbnb, — обфусцирует текст на естественном языке (преобразуя фразы вроде *you are great* в *ur gr8*). При желании можно научить классификатор на основе машинного обучения обнаруживать и преобразовывать текст на английском

языке в обфусцированный английский текст или L33T (https://sites.google.com/site/inhainternetlanguage/different-internet-languages/l33t). Можно также обучить стеммер (автокодировщик с обфускатором, генерирующим символьные признаки) для декодировки обфусцированных слов, чтобы конвейер NLP мог обрабатывать обфусцированный текст без необходимости повторного обучения.

Курсы и учебные руководства

Далее перечислено несколько неплохих учебных руководств, презентаций и даже образовательного ПО известных университетских программ, многие из которых включают примеры на языке Python.

- *Jurafsky D., Martin J. H.* Speech and Language Processing («Обработка речи и языка», https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf) — следующая книга, которую стоит прочитать, если вы всерьез решили заняться NLP. Журафски и Мартин более подробно и формально описывают понятия NLP. Они посвятили целые главы вопросам, которые мы практически проигнорировали, например конечным преобразователям (finite state transducer, FST), скрытым марковским моделям (HMM), частеречной разметке (POS), синтаксическому разбору, когерентности дискурса, машинному переводу, автоматическому реферированию и диалоговым системам.
- Курс MIT Лекса Фридмана (Lex Fridman) *Artificial General Intelligence* (https://agi.mit.edu/), февраль 2018 года, — бесплатный интерактивный (публичный конкурс!) курс MIT по искусственному интеллекту общего уровня. Вероятно, самый строгий и основательный бесплатный курс по проектированию искусственного интеллекта, какой только можно найти.
- *Textacy: NLP, до и после spaCy* (https://github.com/chartbeat-labs/textacy) — обертка для тематического моделирования для spaCy.
- Конспекты лекций курса 6-863j MIT *Natural Language and the Computer Representation of Knowledge* («Естественный язык и компьютерное представление знаний», http://mng.bz/vOdM) весеннего семестра 2003 года.
- Учебное руководство: *Teknomo K.* (PhD) Singular value decomposition (SVD) («Сингулярное разложение» (SVD), http://people.revoledu.com/kardi/tutorial/LinearAlgebra/SVD.html).
- *Manning C. D., Raghavan P., Schütze H.* An Introduction to Information Retrieval («Введение в информационный поиск», https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf).

Утилиты и пакеты

- `nlpia` (http://github.com/totalgood/nlpia) — наборы данных для NLP, утилиты и примеры сценариев из этой книги.
- `OpenFST` (http://openfst.org/twiki/bin/view/FST/WebHome) Тома Бэгби (Tom Bagby), Дэна Байкела (Dan Bikel), Кайла Гормана (Kyle Gorman), Мехьяра Мохри (Mehryar

Mohri) и др. — реализация конечных преобразователей на C++ с открытым исходным кодом.

- `pyfst` (https://github.com/vchahun/pyfst) Виктора Шаюно (Victor Chahuneau) — интерфейс Python для OpenFST.
- Stanford `CoreNLP` — программное обеспечение для обработки естественного языка (https://stanfordnlp.github.io/CoreNLP/) Кристофера Д. Мэннинга (Christopher D. Manning) и др. — библиотека Java, включающая современные процедуры сегментации предложений, выделения дат и времени, частеречной разметки, грамматической проверки и т. д.
- `stanford-corenlp 3.8.0` (https://pypi.org/project/stanford-corenlp/) — интерфейс Python для Stanford CoreNLP.
- `keras` (https://blog.keras.io/) — высокоуровневый API для создания графов вычислений TensorFlow и Theano (нейронных сетей).

Научные статьи и обсуждения

Один из лучших способов досконально изучить какую-либо тему — повторять самостоятельно эксперименты исследователей, а затем их каким-либо образом модифицировать. Именно поэтому лучшие профессора и преподаватели при обучении своих студентов лишь поощряют воспроизведение ими интересующих их результатов других исследователей. Вы неизбежно начнете вносить в подход свои изменения, если потратите достаточно времени, пытаясь добиться желаемых результатов.

Модели векторных пространств и семантический поиск

- *Semantic Vector Encoding and Similarity Search Using Fulltext Search Engines* («Кодирование семантических векторов и поиск подобий с помощью полнотекстовых поисковых систем», https://arxiv.org/pdf/1706.00957.pdf) — Ян Ригль и др. с помощью обычного обратного индекса смогли реализовать эффективный семантический поиск по всей «Википедии».
- *Learning Low-Dimensional Metrics* («Усвоение метрик низкой размерности», https://papers.nips.cc/paper/7002-learning-low-dimensional-metrics.pdf) — Лалит Джайн и др. сумели включить человеческий здравый смысл в попарные метрики расстояния, благодаря чему стали возможны более обоснованное принятие решений и кластеризация без учителя векторов слов и тем. Например, специалисты по подбору персонала благодаря этому подходу могут направить в нужную сторону рекомендательную систему на основе контента, сопоставляющую резюме с описаниями вакансий.
- *Arora S., Li Y., Liang Y., Ma T., Risteski A.* RAND-WALK: A latent variable model approach to word embeddings («RAND-WALK: Подход к вложениям слов на основе моделирования латентных переменных», https://arxiv.org/pdf/1502.03520.pdf) — объясняет новейшие (2016) представления о векторных умозаключениях Word2vec

и других моделей векторных пространств слов, в частности вопросов на аналогию.

- *Mikolov T., Corrado G., Chen K., Dean J.* Efficient Estimation of Word Representations in Vector Space («Эффективная оценка представлений слов в векторном пространстве», https://arxiv.org/pdf/1301.3781.pdf), Google, сентябрь 2013 года — первая публикация модели Word2vec, включающая реализацию на C++ и предобученные на корпусе Google News модели.
- *Mikolov T., Sutskever I., Chen K., Dean J.* Distributed Representations of Words and Phrases and their Compositionality («Распределенные представления слов и фраз и их сочетаемости», https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf), Google, — описание усовершенствований модели Word2vec, повысивших ее точность, включая субдискретизацию и отрицательную субдискретизацию.
- *From Distributional to Semantic Similarity* («От дистрибутивного к семантическому подобию», https://www.era.lib.ed.ac.uk/bitstream/handle/1842/563/IP030023.pdf), 2003 год, кандидатская диссертация Джеймса Ричарда Каррена (James Richard Curran) — множество исследований по традиционному информационному поиску (полнотекстовому поиску), включая нормализацию TF-IDF и метаранжирования страниц для веб-поиска.

Финансы

- *Predicting Stock Returns by Automatically Analyzing Company News Announcements* («Предсказание биржевой прибыли за счет автоматического анализа анонсов новостей компаний», http://www.stagirit.org/sites/default/files/articles/a_0275_ssrn-id2684558.pdf) — Белла Дубров (Bella Dubrov) воспользовалась методом Doc2vec библиотеки gensim для предсказания курсов акций на основе официальных сообщений компаний, с прекрасными объяснениями работы `Word2vec` и `Doc2vec`.
- *Building a Quantitative Trading Strategy to Beat the S&P 500* («Создание численной стратегии инвестиций, позволяющей обойти компании из S&P 500», https://www.youtube.com/watch?v=ll6Tq-wTXXw) — на конференции PyCon 2016 Карен Рубин (Karen Rubin) рассказала, как обнаружила, что женщины — директора компаний способны предсказывать рост курса акций, хотя и не так хорошо, как она думала сначала.

Системы формирования ответов на вопросы

- *Singh A.* Keras-based LSTM/CNN Models for Visual Question Answering («Модели LSTM/CNN для формирования ответов на визуальные вопросы на основе Keras», https://github.com/avisingh599/visual-qa).
- *Magnini B.* Open Domain Question Answering: Techniques, Resources and Systems («Формирование ответов на вопросы без ограничения предметной области: методики, ресурсы и системы», http://lml.bas.bg/ranlp2005/tutorials/magnini.ppt).

- *Katz L.* Question Answering Techniques for the World Wide Web («Методы формирования ответов на вопросы для Всемирной паутины») (https://cs.uwaterloo.ca/~jimmylin/publications/Lin_Katz_EACL2003_tutorial.pdf).
- *NLP-Question-Answer-System* («Система формирования ответов на вопросы на основе NLP», https://github.com/raoariel/NLP-Question-Answer-System/blob/master/simpleQueryAnswering.py) — система, созданная с нуля на основе `corenlp` и `nltk` для сегментации предложений и частеречной разметки.
- *Attardi et al.* PiQASso: Pisa Question Answering System («Пизанская система формирования ответов на вопросы», 2001 год, http://trec.nist.gov/pubs/trec10/papers/piqasso.pdf) — использует обычную NLP на основе информационного поиска (IR).

Глубокое обучение

- *Olah C.* Understanding LSTM Networks («Сети LSTM», https://colah.github.io/posts/2015-08-Understanding-LSTMs) — ясные и четкие объяснения работы LSTM.
- *Cho K. et al.* Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation («Усвоение представлений фраз с помощью RNN кодировщика-декодировщика для статического машинного перевода», 2014 год, https://arxiv.org/pdf/1406.1078.pdf) — статья, в которой впервые были описаны шлюзовые рекуррентные блоки, благодаря которым эффективность LSTM для NLP существенно возросла.

Сети LSTM и RNN

Нам было нелегко разобраться в терминологии и архитектуре сетей LSTM. Вот подборка наиболее цитируемых библиографических источников, так что пусть сами авторы «голосуют», как правильнее рассказывать про сети LSTM. Состояние посвященной сетям LSTM страницы «Википедии» (и соответствующей страницы обсуждения) — отличный показатель отсутствия консенсуса относительно того, что же такое LSTM.

- *Cho K. et al.* Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation («Усвоение представлений фраз с помощью RNN кодировщика-декодировщика для статического машинного перевода», 2014 год, https://arxiv.org/pdf/1406.1078.pdf) — объясняет, как можно использовать содержимое ячеек памяти в LSTM-слое в качестве вложений, кодирующих последовательности переменной длины, а затем декодирующих их в новую последовательность переменной (возможно, другой) длины, то есть переводя (перекодируя) одну последовательность в другую.
- *Bakker B.* Reinforcement Learning with Long Short-Term Memory («Обучение с подкреплением с помощью сетей с долгой краткосрочной памятью», https://papers.nips.cc/paper/1953-reinforcement-learning-with-long-short-term-memory.pdf) — приложение

сетей LSTM к процессу когнитивного планирования и формирования гипотез с демонстрацией сети, способной решать задачу движения по T-образному лабиринту и расширенную задачу балансировки шеста (обратного маятника).

- Диссертация Алекса Грейвса (Alex Graves) под руководством Б. Брюгге (B. Brugge) *Supervised Sequence Labelling with Recurrent Neural Networks* («Маркировка последовательностей с учителем с помощью рекуррентных нейронных сетей», https://mediatum.ub.tum.de/doc/673554/file.pdf) — подробное объяснение математики метода точного градиентного спуска для LSTM, впервые предложенного Хохрайтером (Hochreiter) и Шмидхубером (Schmidhuber) в 1997 году. Но Грейвсу не удалось дать строгое определение таких терминов, как CEC и LSTM-*блок/ячейка*.
- *Carrier P. L., Cho K.* Документация по LSTM библиотеки Theano (http://deeplearning.net/tutorial/lstm.html) — схемы и обсуждение, объясняющие реализацию LSTM в Theano и Keras.
- *Gers F. A., Schmidhuber J., Cummins F.* Learning to Forget: Continual Prediction with LSTM («Учимся забывать: беспрерывное предсказание с помощью LSTM», http://mng.bz/4v5V) — использует нестандартные обозначения для входных ($\mathbf{y}^{in}$) и выходных ($\mathbf{y}^{out}$) сигналов слоев и внутреннего скрытого состояния *(h)*. Все математические операции и схемы векторизованы.
- *Sutskever I., Vinyals O., Le Q. V.* Sequence to Sequence Learning with Neural Networks («Обучение с преобразованием последовательностей в последовательности с помощью нейронных сетей», Google, http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf).
- Сообщение от 2015 года в блоге Чарльза Ола (Charles Olah) *Understanding LSTM Networks* («Разбираемся, что такое LSTM-сети», http://colah.github.io/posts/2015-08-Understanding-LSTMs) — множество отличных схем и обсуждение/обратная связь от читателей.
- *Hochreiter S., Schmidhuber J.* Long Short-Term Memory («Долгая краткосрочная память», 1997 год, http://www.bioinf.jku.at/publications/older/2604.pdf) — исходная статья о сетях LSTM с устаревшей терминологией и неэффективной реализацией, но зато подробным математическим выводом.

Конкурсы и премии

- *Тест быстродействия сжатия больших текстов* (http://mattmahoney.net/dc/text.html) — некоторые исследователи убеждены, что сжатие текста на естественном языке эквивалентно искусственному интеллекту общего уровня (AGI).
- *Премия Хаттера* (https://en.wikipedia.org/wiki/Hutter_Prize) — ежегодное соревнование по сжатию архива текста на естественном языке из «Википедии» размером 100 Мбайт. В 2017 году его выиграл Александр Ратушняк.
- *Открытый конкурс по извлечению знаний — 2017* (https://svn.aksw.org/papers/2017/ESWC_Challenge_OKE/public.pdf).

Наборы данных

Данные на естественном языке — повсюду, куда ни глянь. Язык — суперсила человечества, и ваш конвейер должен этим пользоваться.

- Поиск наборов данных от компании Google (http://toolbox.google.com/datasetsearch) — поисковая система, аналогичная Google Scholar (http://scholar.google.com), но для данных.
- Наборы данных Стэнфордского университета (https://nlp.stanford.edu) — предобученные модели `word2vec` и GloVE, многоязыковые модели языка и наборы данных, многоязыковые словари и корпуса.
- Предобученные модели векторов слов (https://github.com/3Top/word2vec-api#where-to-get-a-pretrained-model) — в файле README для веб-API векторов слов приведены ссылки на несколько моделей векторов слов, включая 300-мерную модель «Википедии» GloVE.
- Список наборов данных/корпусов для задач NLP в обратном хронологическом порядке (https://github.com/karthikncode/nlp-datasets) от Картика Нарасимхана (Karthik Narasimhan).
- Алфавитный список бесплатных/являющихся общественным достоянием наборов с текстовыми данными для использования при обработке естественного языка (https://github.com/niderhoff/nlp-datasets).
- Наборы данных и инструменты для простейшей обработки естественного языка (https://github.com/googlei18n/language-resources) — утилиты Google для интернационализации.
- `nlpia` (https://github.com/totalgood/nlpia) — пакет Python с загрузчиками данных (`nlpia.loaders`) и препроцессорами для всех NLP-данных, которые вам понадобятся... до завершения чтения этой книги.

Поисковые системы

Информационный поиск — важная часть NLP. И особенно важно правильно его организовать, чтобы властвующий над нами ИИ (и коммерческие корпорации) не манипулировал скармливаемой нам информацией. Если вы хотите научиться искать информацию путем создания своих собственных поисковых систем, вот несколько полезных ресурсов.

Поисковые алгоритмы

- *Billion-scale similarity search with GPUs* («Сверхмасштабный поиск подобий с использованием GPU», https://arxiv.org/pdf/1702.08734.pdf) — BidMACH представляет собой реализацию алгоритма многомерной векторной индексации и поиска на основе KNN (аналогично пакету Annoy Python). Эта статья описывает расширение для работы с GPU, в восемь раз более производительное, чем исходная реализация.

- Пакет Annoy от Spotify, созданный Эриком Бернхардссоном (Erik Bernhardsson) (https://erikbern.com/2017/11/26/annoy-1.10-released-with-hamming-distance-and-windows-support.html), — алгоритм поиска методом k-ближайших соседей, используемый Spotify для поиска похожих песен.
- *Bernhardsson E.* New Benchmarks for Approximate Nearest Neighbors («Новые тесты быстродействия приближенного метода ближайших соседей», https://erikbern.com/2018/02/15/new-benchmarks-for-approximate-nearest-neighbors.html) — приближенные методы поиска ближайших соседей — ключ к масштабируемому семантическому поиску, и автор статьи внимательно отслеживает последние новости в этой сфере.

Поисковые системы с открытым исходным кодом

- BeeSeek (https://launchpad.net/~beeseek-devs) — распределенная поисковая индексация и своя система поиска (поиск на основе p2p); разработка и поддержка прекращены.
- WebSPHNIX (https://www.cs.cmu.edu/~rcm/websphinx/) — веб-интерфейс (GUI) для создания веб-поискового агента.

Утилиты полнотекстовой индексации с открытым исходным кодом

Эффективная индексация жизненно важна для любого приложения поиска на естественном языке. Существует несколько утилит с открытым исходным кодом для полнотекстовой индексации. Впрочем, эти поисковые системы не умеют сканировать Интернет, так что им необходимо предоставить корпус для индексации и поиска.

- Elasticsearch (https://github.com/elastic/elasticsearch) — распределенная, воплощающая REST поисковая система с открытым исходным кодом.
- Apache Lucern + Solr (https://github.com/apache/lucene-solr).
- Sphinx Search (https://github.com/sphinxsearch/sphinx).
- Kronuz/Xapiand: Xapiand: Воплощающая REST поисковая система (https://github.com/Kronuz/Xapiand) — пакеты для операционной системы Ubuntu, с помощью которых можно производить поиск по локальному жесткому диску (подобно тому как это делает Google Desktop).
- Indri (http://www.lemurproject.org/indri.php) — семантический поиск с интерфейсом для Python (https://github.com/cvangysel/pyndri), правда, сопровождается не слишком активно.
- Gigablast (https://github.com/gigablast/open-source-search-engine) — поисковый агент и инструмент индексации естественного языка на C++ с открытым исходным кодом.
- Zettair (http://www.seg.rmit.edu.au/zettair) — инструмент для индексации HTML и TREC с открытым исходным кодом (без поискового агента и реальных примеров); в последний раз обновлялся в 2009 году.

- OpenFTS: Full Text Search Engine («Полнотекстовая поисковая система», http://openfts.sourceforge.net) — полнотекстовая система поисковой индексации для PyFTS, использующая PostgreSQL, с API для языка Python (http://rhodesmill.org/brandon/projects/pyfts.html).

Манипулятивные поисковые системы

Используемые большинством из нас поисковые системы вовсе не оптимизированы для помощи нам в поиске, а стремятся к тому, чтобы мы перешли по ссылкам, которые приносят создавшим их компаниям прибыль. Передовой аукцион второй цены с закрытыми торгами компании Google гарантирует, что рекламодатели не переплачивают за свои рекламные объявления[1], но отнюдь не предотвращает лишние траты ищущих информацию пользователей из-за щелчков на скрытых рекламных объявлениях. Подобный манипулятивный поиск присущ не только Google. Он используется во всех поисковых системах, ранжирующих результаты согласно какой-либо целевой функции, а не тому, насколько пользователи удовлетворены результатами поиска. Но если вы хотите сравнить и поэкспериментировать, вот их список:

- Google;
- Bing;
- Baidu.

Менее манипулятивные поисковые системы

Для определения степени манипулятивности и коммерциализации поисковых систем мы отправили в несколько из них запросы вроде *open source search engine*[2] и подсчитали число клиентов AdWords[3] и сайтов, использующих клик-наживки среди первых десяти результатов поиска. На следующих сайтах это число не превышало одного или двух, а первые выдаваемые ими результаты обычно были наиболее объективными и полезными сайтами, такими как «Википедия», Stack Exchange или авторитетными новостными статьями и блогами:

- альтернативы Google (https://www.lifehack.org/374487/try-these-15-search-engines-instead-google-for-better-search-results)[4];

---

1 Анализ этого сценария из курса «Сети» Корнеллского университета Google AdWords Auction — A Second Price Sealed-Bid Auction («Аукцион Google AdWords — аукцион второй цены с закрытыми торгами», https://blogs.cornell.edu/info2040/2012/10/27/google-adwords-auction-a-second-price-sealed-bid-auction).

2 Поисковая система с открытым исходным кодом. — *Примеч. пер.*

3 С июля 2018 года называется Google Ads. — *Примеч. пер.*

4 См. веб-страницу Try These 15 Search Engines Instead of Google For Better Search Results по адресу https://www.lifehack.org/374487/try-these-15-search-engines-instead-google-for-better-search-results.

- «Яндекс» (https://yandex.com/search/?text=open%20source%20search%20engine&lr=21754) — как ни странно, похоже, что наиболее популярная российская поисковая система (60 % поисковых запросов на русском языке) в меньшей степени манипулирует поиском, чем наиболее популярные поисковые системы США;
- DuckDuckGo (https://duckduckgo.com);
- семантический веб-поиск `Watson` (http://watson.kmi.open.ac.uk/WatsonWUI) — хотя разработка прекращена и это не совсем полнотекстовая поисковая система для Интернета, она представляет собой интересный способ изучения семантической паутины (по крайней мере, была несколько лет назад, до заморозки проекта `watson`).

Распределенные поисковые системы

Распределенные поисковые системы[1], вероятно, наименее манипулятивные и наиболее объективные, поскольку в них отсутствует центральный сервер, который мог бы влиять на ранжирование результатов поиска. Впрочем, ранжирование страниц в современных реализациях распределенного поиска основывается на частотностях слов TF-IDF из-за проблем масштабирования и распределенного выполнения NLP-алгоритмов семантического поиска. Впрочем, такие подходы к семантической индексации, как латентно-семантический анализ (LSA) и хеширование с учетом локальности, удалось успешно реализовать распределенным образом практически с линейным масштабированием (идеальный случай). Когда кто-нибудь решит предоставить код для семантического поиска проекту с открытым исходным кодом вроде Yacy или создать новую распределенную поисковую систему, умеющую выполнять LSA, — лишь вопрос времени.

- Nutch (https://nutch.apache.org/) — в основе Nutch лежит Hadoop, и он сам с течением времени постепенно превращается из распределенной поисковой системы в распределенную HPC-систему.
- Yacy (https://www.yacy.net/en/index.html) — одна из немного до сих пор активно используемых децентрализованных (федеративных) поисковых систем с открытым исходным кодом (https://github.com/yacy/yacy_search_server). Существуют заранее настроенные клиенты для Mac, Linux и Windows.

[1] См. веб-страницы https://en.wikipedia.org/wiki/Distributed_search_engine и https://wiki.p2pfoundation.net/Distributed_Search_Engines.

Глоссарий

Мы собрали здесь некоторые определения из сферы обработки естественного языка, а также акронимы и терминологию машинного обучения[1].

Часть синтаксических анализаторов и регулярных выражений, которые мы использовали при генерации этого списка, можно найти в пакете `nlpia` Python по адресу github.com/totalgood/nlpia (https://github.com/totalgood/nlpia)[2]. Следующий листинг демонстрирует, как применялся `nlpia` для составления англоязычной версии этого глоссария:

```
>>> from nlpia.book_parser import write_glossary
>>> from nlpia.constants import DATA_PATH
>>> print(write_glossary(
... os.path.join(DATA_PATH, 'book')))
== Акронимы

[acronyms,template="glossary",id="terms"]
*AGI*:: Artificial general intelligence --
*AI*:: Artificial intelligence --
*AIML*:: Artificial Intelligence Markup Language --
*ANN*:: Approximate nearest neighbors --
...
```

Вам придется делать по-другому, поскольку мы не можем включить всю рукопись книги в каталог data

[1] Более подробный словарь NLP составил Билл Уилсон (Bill Wilson) из Университета Нового Южного Уэльса в Австралии (http://www.cse.unsw.edu.au/~billw/nlpdict.html).

[2] nlpia.translators (https://github.com/totalgood/nlpia/blob/master/src/nlpia/translators.py) и nlpia.book_parser (https://github.com/totalgood/nlpia/blob/master/src/nlpia/book_parser.py).

Данный генератор определений не завершен, это можно сделать с помощью хорошей языковой LSTM-модели (см. главу 10). Мы оставляем эту задачу в качестве упражнения вам.

Акронимы

AGI (Artificial general intelligence) — искусственный интеллект общего уровня. Машинный интеллект, способный решать множество тех же задач, что и человеческий мозг.

AI (Artificial intelligence) — искусственный интеллект. Поведение машины, достаточно впечатляющее для того, чтобы ученые или маркетологи корпораций могли назвать его разумным.

AIML (Artificial Intelligence Markup Language) — язык разметки для систем ИИ. Основанный на XML язык описания шаблонизированных ответов и сопоставления с паттернами, придуманный в процессе создания A.L.I.C.E., одного из первых речевых чат-ботов.

ANN (Approximate nearest neighbors) — приближенный метод ближайших соседей. Поиск m ближайших соседей отдельного вектора из набора n-мерных векторов представляет собой задачу со сложностью O(N), поскольку необходимо вычислять метрику расстояния между целевым вектором и каждым из остальных. В результате кластеризация становится неподъемной задачей сложности $O(N^2)$.

ANN (Artificial neural network) — искусственная нейронная сеть.

API (Application programming interface) — программный интерфейс приложения. Пользовательский интерфейс для ваших клиентов-программистов, обычно утилита командной строки, библиотека исходного кода или веб-интерфейс, с которыми они могут взаимодействовать программным образом.

AWS (Amazon Web Services) — веб-сервисы Amazon. Amazon, открыв для всеобщего обозрения свою внутреннюю инфраструктуру, создал тем самым понятие облачных сервисов.

BOW (Bag of words) — мультимножество слов. Структура данных (обычно вектор), содержащая информацию о количестве вхождений (частотностях) слов, но не в их порядке.

CEC (Constant error carousel) — карусель константной ошибки. Нейрон, выдающий на выходе свой входной сигнал с задержкой на один временной шаг. Используется внутри блоков памяти LSTM и GRU. Представляет собой регистр памяти для LSTM-блока, может быть сброшен на новое значение только прерывающим эту «карусель» шлюзом забывания.

CNN (Convolutional neural network) — сверточная нейронная сеть. Нейронная сеть, усваивающая *фильтры (ядра)* для целей выделения признаков при машинном обучении с учителем

CUDA (Compute Unified Device Architecture) — унифицированная архитектура вычислительных устройств. Библиотека NVIDIA с открытым исходным кодом, оптимизированная для выполнения обычных вычислений/алгоритмов на GPU.

DAG (Directed acyclic graph) — ориентированный ациклический граф. Топология сети без циклов, то есть связей, закольцованных на себя же.

DFA (Deterministic finite automaton) — детерминистический конечный автомат. Конечный автомат, не производящий случайных действий. Пакет `re` в Python компилирует регулярные выражения для создания DFA, а пакет `regex` способен компилировать нечеткие регулярные выражения в NDFA (недетерминистические FA).

FSM (Finite-state machine) — конечный автомат. Кайл Горман (Kyle Gorman) и «Википедия» объяснят это понятие лучше, чем мы (https://ru.wikipedia.org/wiki/Конечный_автомат).

FST (Finite-state transducer) — конечный преобразователь. FST подобны регулярным выражениям, но могут выдавать новый символ вместо каждого найденного соответствия паттерну. Кайл Горман отлично объясняет, что это такое (www.openfst.org).

GIS (Geographic information system) — геоинформационная система. База данных для хранения и отображения географической информации (обычно включающей координаты широты, долготы, высоты над уровнем моря и их атрибутов), а также выполнения над ней различных операций.

GPU (Graphical processing unit) — графический процессор. Видеокарта в игровой системе, сервере для майнинга криптовалюты или в сервере для машинного обучения.

GRU (Gated recurrent unit) — шлюзовой рекуррентный блок. Разновидность сети с долгой краткосрочной памятью, с совместно используемыми параметрами для ускорения вычислений.

HNSW (Hierarchical Navigable Small World) — «иерархический, допускающий навигацию, маленький мирок». Это графовая структура данных для эффективного поиска (см. *Мальков Ю. А., Яшунин Д. А.* Эффективный и надежный поиск приближенным методом поиска ближайших соседей с помощью графов HNSW (https://arxiv.org/vc/arxiv/papers/1603/1603.09320v1.pdf)).

HPC (High performance computing) — высокопроизводительные вычисления. Системы с максимальной пропускной способностью, обычно за счет распараллеливания вычислений на отдельных этапах отображения и свертки.

IDE (Integrated development environment) — интегрированная среда разработки. Традиционное (не веб-) приложение для разработки ПО, например PyCharm, Eclipse, Atom, or Sublime Text 3.

IR (Information retrieval) — информационный поиск. Исследования алгоритмов поиска по документам и в Интернете. Именно они ввели NLP в число важных вычислительных дисциплин в 1990-х.

ITU (IT University of Copenhagen) — IT-университет Копенгагена. Один из ведущих технических университетов.

i18n (Internationalization) — интернационализация. Подготовка приложения к использованию в нескольких странах (с несколькими региональными настройками).

LDA (Linear discriminant analysis) — линейный дискриминантный анализ. Алгоритм классификации с линейным разделением классов (см. главу 4).

LSA (Latent semantic analysis) — латентно-семантический анализ. Применение усеченного SVD к векторам TF-IDF мультимножеств слов для создания векторов тем в векторном пространстве языковой модели (см. главу 4).

LSH (Locality sensitive hash/hashing) — учитывающий локальность хеш/хеширование с учетом локальности. Хеш, играющий роль эффективного, но приближенного индекса отображения/кластеризации для плотных многомерных векторов (см. главу 13). Такие хеши можно рассматривать как ZIP-коды для пространства размерности выше двух (не только широты и долготы).

LSI (Latent semantic indexing) — латентно-семантическая индексация. Устаревшее и неправильное название латентно-семантического анализа (см. LSA), поскольку модели LSA векторного пространства плохо поддаются индексации.

LSTM (Long short-term memory) — долгая краткосрочная память. Усовершенствованный вид рекуррентных нейронных сетей с поддержкой памяти состояния, усваиваемой с помощью обратного распространения ошибки (см. главу 9).

MIH (Multi-index hashing) — мультииндексное хеширование. Методика хеширования и индексации для многомерных плотных векторов.

ML (Machine learning) — машинное обучение. Программирование машин с помощью данных вместо написания алгоритмов вручную.

MSE (Mean squared error) — среднеквадратичная погрешность. Сумма квадратов разницы между желаемым и фактическим выходными сигналами модели машинного обучения.

NELL (Never Ending Language Learning) — никогда не прекращающееся изучение языка. Проект Университета Карнеги — Меллона по выделению знаний, который функционирует непрерывно уже многие годы, сканируя веб-страницы и выделяя общеизвестные знания о мире (в основном категорийные связи типа *IS-A* между термами).

NLG (Natural language generation) — генерация естественного языка. Автоматическое, алгоритмическое составление текстов; одна из самых сложных задач обработки естественного языка (NLP).

NLP (Natural language processing) — обработка естественного языка. Вероятно, вы уже знаете, что это такое. Если нет, см. введение в этот вопрос в главе 1.

NLU (Natural language understanding) — понимание естественного языка. Этот термин часто используется в недавних статьях для описания обработки естественного языка с помощью нейронных сетей.

NMF (Nonnegative matrix factorization) — неотрицательное разложение матриц. Метод разложения матриц, подобный SVD, но все элементы матриц-множителей должны быть больше нуля или равны нулю.

NSF (National Science Foundation) — Национальный научный фонд. Агентство при правительстве США, отвечающее за финансирование научных исследований.

NYC (New York City) — Нью-Йорк. Никогда не спящий город в США.

OSS (Open source software) — ПО с открытым исходным кодом.

pip (Pip installs packages) — рекурсивный акроним для «*Pip устанавливает пакеты*» (Cheese Shop). Официальная система управления пакетами языка Python для автоматического скачивания и установки пакетов из «Магазина сыра» (pypi.python.org).

PR (Pull request) — запрос на внесение изменений. Правильный способ попросить кого-либо объединить его код с вашим. На GitHub есть специальные кнопки

и мастера для упрощения этого процесса. Именно так можно заслужить репутацию человека, добросовестно относящегося к делу внесения вклада в программное обеспечение с открытым исходным кодом.

PCA (Principal component analysis) — метод главных компонент. Усеченный SVD для любых численных данных, обычно изображений или аудиофайлов.

QDA (Quadratic discriminant analysis) — квадратичный дискриминантный анализ. Аналогичен LDA, но границы между классами могут быть квадратичными (криволинейными).

ReLU (Rectified linear unit) — выпрямленный линейный блок. Функция активации линейной нейронной сети, обеспечивающая ненулевой[1] выходной сигнал нейрона. Эквивалентна `y = np.max(x, 0)`. Наиболее популярная и эффективная функция активации для обработки изображений и NLP, поскольку дает возможность производить эффективное обратное распространение ошибки даже для очень глубоких сетей без «исчезновения градиентов».

REPL (Read-evaluate-print loop) — цикл разработки «чтение — вычисление» — вывод. Типичный цикл разработки для разработчика на любом языке сценариев, не требующем компиляции. Особенно широкие возможности предоставляют REPL ipython, консоли jupyter и блокнотов jupyter с их магическими командами `help`, `?`, `??` и `%` плюс автодополнение и поиск по истории команд Ctrl-R[2].

RMSE (Root mean square error) — корень из среднеквадратичной погрешности. Квадратный корень из среднеквадратичной погрешности. Распространенная метрика погрешности регрессии. Может также использоваться для задач бинарной и порядковой классификации. Соответствует интуитивной оценке неопределенности предсказаний модели в 1-σ.

RNN (Recurrent neural network) — рекуррентные нейронные сети. Архитектура нейронных сетей, при которой входные сигналы одного слоя подаются на вход предыдущего. Сети RNN часто развертываются в эквивалентные сети прямого распространения для построения схем и анализа.

SMO (Sequential minimal optimization) — последовательная минимальная оптимизация. Алгоритм и подход решения задачи квадратичного программирования, возникающей при использовании метода опорных векторов.

SVD (Singular value decomposition) — сингулярное разложение. Метод разложения матрицы на диагональную матрицу собственных значений и две ортогональные матрицы собственных векторов. Этот метод лежит в основе LSA и PCA (см. главу 4).

SVM (Support vector machine) — метод опорных векторов. Алгоритм машинного обучения, обычно используемый для классификации.

TF-IDF (Term frequency/inverse document frequency) — частотность терма/обратная частотность документа. Метод нормализации количеств вхождений слов для улучшения результатов информационного поиска (см. главу 3).

---

[1] По-видимому, авторы имеют в виду «неотрицательный». — *Примеч. пер.*

[2] REPL языка Python даже позволяют выполнить любую команду (включая pip) инсталлированной в операционной системе командной оболочки (например, !git commit -am 'fix 123'). Благодаря этому можно не убирать пальцы с клавиатуры и не тянуться к мышке, что минимизирует когнитивную нагрузку от переключений контекста.

UI (User interface) — пользовательский интерфейс. Предоставляемые пользователю в программном обеспечении возможности действий, обычно графические веб-страницы или экранные формы мобильных приложений, путем взаимодействия с которыми пользователь применяет программный продукт или сервис.

UX (User experience) — опыт взаимодействия пользователя. Сущность взаимодействия потребителя с программным продуктом или компанией, от покупки до самого последнего его контакта с вами. Включает ваш сайт или API UI веб-сайта и все остальные взаимодействия пользователя с вашей компанией.

VSM (Vector space model) — модель векторного пространства. Векторное представление объектов задачи, например слов или документов в задаче NLP (см. главы 4 и 6).

YMMV (Your mileage may vary) — у вас все может оказаться иначе. Полученные вами результаты могут отличаться от наших.

Терминология

Возможное действие (affordance) — осознанно предоставляемый разработчиком способ взаимодействия пользователя с программным продуктом. В идеале такое взаимодействие должно быть для пользователя интуитивным, легко заметным и самодокументированным.

Искусственная нейронная сеть (artificial neural network) — граф вычислений для машинного обучения или имитационного моделирования биологической нейронной сети (мозга).

Ячейка (cell) — память или состояние LSTM-блока, фиксирующее одно скалярное значение и непрерывно выдающее его на выходе (https://ru.wikipedia.org/wiki/Долгая_краткосрочная_память).

Темные паттерны (dark pattern) — паттерны программного обеспечения (обычно пользовательского интерфейса), предназначенные для повышения прибыли, но часто приводящие к обратному эффекту, поскольку пытаются хитростью заставить клиентов использовать продукт не так, как клиентам этого хотелось бы.

Сеть прямого распространения (feed-forward network) — однопроходная нейронная сеть, по которой входные сигналы передаются до выхода в одном направлении, формируя ориентированный ациклический граф (DAG) вычислений (дерево).

Морфема (morpheme) — осмысленная сама по себе часть токена или слова. Составляющие токен морфемы в совокупности называются *морфологией* токена. Найти морфологию токена можно с помощью алгоритмов из таких пакетов, как spaCy, обрабатывающих токен вместе с контекстом (окружающими его словами)[1].

Сеть, **нейронная сеть** (net, network, neural net) — искусственная нейронная сеть.

Нейрон (neuron) — это блок нейронной сети, функция которого (например, `y = tanh(w.dot(x))`) принимает на входе несколько входных сигналов и возвращает одно скалярное значение. Это значение обычно представляет собой весовые ко-

[1] См. веб-страницу Linguistic Features: spaCy Usage Documentation по адресу https://spacy.io/usage/linguistic-features#rule-based-morphology.

эффициенты нейрона (**w** или w^i), умноженные на все входные сигналы (**x** или x^i), и складывается с весовым коэффициентом для смещения (w^0) с последующим применением функции активации вроде `th`. Нейрон всегда выдает на выходе скалярное значение, отправляемое на вход каких-либо дополнительных скрытых или выходных нейронов этой сети. Нейрон, реализующий более сложную функцию активации, вроде расширений, благодаря которым из рекуррентных нейронов удалось создать LSTM, обычно называют *блоком* (unit), например *LSTM-блок*.

Ness-вектор — нестрогий термин для векторов тем или семантических векторов, захватывающих в измерениях вектора определенные понятия или качества.

Сказуемое (predicate) — в грамматике английского языка сказуемое — это основной глагол предложения, связанный с подлежащим. В любом полном предложении должно быть сказуемое, равно как и подлежащее.

Skip-граммы (skip-grams) — пары токенов, используемые в качестве тренировочных примеров данных для вложений векторов слов, в которых игнорируется какое-то число промежуточных слов (см. главу 6).

Многомерная логистическая функция (softmax) — нормализованная экспоненциальная функция, «размазывающая» вещественнозначный векторный выходной сигнал нейронной сети так, чтобы значения находились в диапазоне от 0 до 1, как вероятности.

Подлежащее (subject) — основное существительное предложения; в каждом полном предложении должно быть подлежащее (и сказуемое), даже если подлежащее лишь подразумевается, например, как в предложении *Run!* («Беги!»), где подразумеваемое подлежащее — *you* («ты»).

Блок (unit) — нейрон или небольшой набор нейронов, реализующий некую сложную нелинейную функцию для вычисления выходного сигнала. Например, в LSTM-блоке есть память, фиксирующая состояние, входной шлюз (нейрон), определяющий, какое значение запоминать, шлюз забывания (нейрон), определяющий, сколько времени помнить это значение, и выходной шлюз (нейрон), реализующий функцию активации блока (обычно сигма-функцию или `th()`). Блок — это замена для нейрона в нейронной сети, принимающая векторный входной сигнал и выдающая на выходе скалярное значение, просто с более сложным поведением.

Лейн Хобсон, Хапке Ханнес, Ховард Коул

Обработка естественного языка в действии

Перевели с английского *И. Пальти, С. Черников*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*К. Русецкий*
Литературные редакторы	*В. Байдук, А. Дубейко*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Е. Рафалюк-Бузовская*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 15.04.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 46,440. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87