O'REILLY®

2-е издание

Data Science

Наука о данных с нуля

Джоэл Грас

bhv®

SECOND EDITION

Data Science from Scratch

First Principles with Python

Joel Grus

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY

Джоэл Грас

Data Science

Наука о данных с нуля

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2021

УДК 004.6
ББК 32.81
Г77

Грас Д.

Г77 Data Science. Наука о данных с нуля: Пер. с англ. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2021. — 416 с.: ил.

ISBN 978-5-9775-6731-2

Книга позволяет изучить науку о данных (Data Science) и применить полученные знания на практике. Она содержит краткий курс языка Python, элементы линейной алгебры, статистики, теории вероятностей, методов обработки данных. Приведены основы машинного обучения. Описаны алгоритмы *k* ближайших соседей, наивной байесовой классификации, линейной и логистической регрессии, а также модели на основе деревьев принятия решений, нейронных сетей и кластеризации. Рассмотрены приемы обработки естественного языка, методы анализа социальных сетей, основы баз данных, SQL и MapReduce.

Во втором издании примеры переписаны на Python 3.6, игрушечные наборы данных заменены на «реальные», добавлены материалы по глубокому обучению и этике данных, статистике и обработке естественного языка, рекуррентным нейронным сетям, векторным вложениям слов и разложению матриц.

Для аналитиков данных

УДК 004.6
ББК 32.81

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Перевод с английского	*Андрея Логунова*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 30.06.20.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 33,54.
Тираж 2000 экз. Заказ № 5935.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано в ОАО «Можайский полиграфический комбинат».
143200, Россия, г. Можайск, ул. Мира, 93.
www.oaompk.ru, тел.: (495) 745-84-28, (49638) 20-685

ISBN 978-1-492-04113-9 (англ.)
ISBN 978-5-9775-6731-2 (рус.)

Оглавление

Предисловие

Я исключительно горжусь первым изданием книги "Наука о данных с нуля". Книга оказалась именно такой, какой я и хотел ее видеть. Но несколько лет развития науки о данных, прогресса в экосистеме Python и личностного роста как разработчика и педагога изменили мои представления о том, как первая книга по науке о данных должна выглядеть.

В жизни не бывает повторных попыток. Однако в книжном деле существуют вторые издания. И поэтому я переписал весь исходный код и примеры, используя Python 3.6 (и многие из его новых функций, таких как аннотации типов). Я вплел в книгу акцент на написании чистого кода. Я заменил некоторые игрушечные примеры первого издания более реалистичными, используя "реальные" наборы данных. Я добавил новые материалы по таким темам, как глубокое обучение, статистика и обработка естественного языка, соответствующие проблемам, которые, вероятно, придется решать современным исследователям. (Я также удалил некоторые материалы, которые кажутся менее субъективными.) И я прошелся по книге мелкозубой расческой, исправляя ошибки, переписывая объяснения, которые были не очень понятны, и по ходу освежая некоторые шутки.

Первое издание было замечательной книгой, но это издание еще лучше. Наслаждайтесь!

Условные обозначения, принятые в книге

В книге используются следующие типографические условные обозначения:

- *курсив* указывает на новые термины;
- `моноширинный шрифт` используется для листингов программ, а также внутри абзацев для ссылки на элементы программ, такие как переменные или имена функций, базы данных, типы данных, переменные окружающей среды, операторы и ключевые слова;
- **`жирный моноширинный шрифт`** показывает команды либо другой текст, который должен быть напечатан пользователем, а также ключевые слова в коде;
- *`моноширинный шрифт курсивом`* показывает текст, который должен быть заменен значениями пользователя либо значениями, определяемыми по контексту.

Данный элемент обозначает подсказку или совет.

Данный элемент обозначает общее замечание.

Данный элемент обозначает предупреждение или предостережение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и пр.) доступны для скачивания по адресу **https://github.com/joelgrus/data-science-from-scratch**.

Эта книга предназначена для того, чтобы помочь вам решить ваши задачи. В целом, если код примеров предлагается вместе с книгой, то вы можете использовать его в своих программах и документации. Вам не нужно связываться с нами с просьбой о разрешении, если вы не воспроизводите значительную часть кода. Например, написание программы, которая использует несколько фрагментов кода из данной книги, официального разрешения не требует.

Адаптированный вариант примеров в виде электронного архива вы можете скачать по ссылке **ftp://ftp.bhv.ru/9785977567312.zip**. Эта ссылка доступна также со страницы книги на веб-сайте **www.bhv.ru**.

Благодарности

Прежде всего, хотел бы поблагодарить Майка Лукидеса (Mike Loukides) за то, что принял мое предложение по поводу этой книги и настоял на том, чтобы я довел ее до разумного объема. Он мог бы легко сказать: "Кто этот человек, вообще, который шлет мне образцы глав, и как заставить его прекратить, наконец?" И я признателен, что он этого не сделал. Хотел бы поблагодарить моего редактора, Мари Богуро (Marie Beaugureau), которая консультировала меня в течение всей процедуры публикации и привела книгу в гораздо более удобоваримый вид, чем если б этим я занимался сам.

Я бы не написал эту книгу, если бы не изучил науку о данных, и, возможно, я бы не научился ей, если бы не влияние Дэйва Хсу (Dave Hsu), Игоря Татаринова (Igor Tatarinov), Джона Раузера (John Rauser) и остальной группы единомышленников из Farecast (это было так давно, что в то время даже не было понятия науки о данных). Хорошие парни в Coursera также заслуживают много добрых слов.

Я также благодарен моим бета-читателям и рецензентам. Джэй Фандлинг (Jay Fundling) обнаружил тонны ошибок и указал на многие нечеткие объяснения,

и в итоге книга оказалась намного лучше и корректней благодаря ему. Дэбаши Гош (Debashis Ghosh) геройски провела санитарную проверку моих статистических выкладок. Эндрю Массельман (Andrew Musselman) предложил смягчить первоначальный аспект книги, касающийся утверждения, что "люди, предпочитающие язык R вместо Python, есть моральные извращенцы", что, думаю, в итоге оказалось неплохим советом. Трэй Кози (Trey Causey), Райан Мэттью Балфанц (Ryan Matthew Balfanz), Лорис Муларони (Loris Mularoni), Нуриа Пуджол (Núria Pujol), Роб Джефферсон (Rob Jefferson), Мэри Пэт Кэмрбелл (Mary Pat Campbell), Зак Джири (Zach Geary) и Венди Грас (Wendy Grus) также высказали неоценимые замечания. За любые оставшиеся ошибки, конечно же, отвечаю только я.

Я многим обязан сообществу с хештегом **#datascience** в Twitter за то, что его участники продемонстрировали мне массу новых концепций, познакомили меня со множеством замечательных людей и заставили почувствовать себя двоечником, да так, что я ушел и написал книгу в качестве противовеса. Особые благодарности снова Трэю Кози (Trey Causey) за то, что нечаянно упомянул, чтобы я включил главу про линейную алгебру, и Шин Дж. Тэйлор (Sean J. Taylor) за то, что неумышленно указала на пару больших несоответствий в *главе "Работа с данными"*.

Но больше всего я задолжал огромную благодарность Ганге (Ganga) и Мэдлин (Madeline). Единственная вещь, которая труднее написания книги, — это жить рядом с тем, кто пишет книгу, и я бы не смог ее закончить без их поддержки.

Предисловие к первому изданию

Наука о данных

Исследователей данных называют "самой сексуальной профессией XXI века"[1]. Очевидно, тот, кто так выразился, никогда не бывал в пожарной части. Тем не менее наука о данных — это действительно передовая и быстроразвивающаяся отрасль знаний, и не придется рыскать по Интернету, чтобы отыскать обозревателей рыночных тенденций, которые возбужденно предвещают, что через 10 лет нам потребуется на миллиарды и миллиарды больше исследователей данных, чем мы имеем на текущий момент.

Но что же это такое — наука о данных? В конце концов нельзя же выпускать специалистов в этой области, если не знаешь, что она собой представляет. Согласно диаграмме Венна[2,3], которая довольно известна в этой отрасли, наука о данных находится на пересечении:

- алгоритмических и хакерских навыков;
- знаний математики и статистики;
- профессионального опыта в предметной области.

Собираясь с самого начала написать книгу, охватывающую все три направления, я быстро пришел к выводу, что всестороннее обсуждение профессионального опыта в предметной области потребовало бы десятков тысяч страниц. Тогда я решил сосредоточиться на первых двух. Задача — помочь желающим развить свои навыки алгоритмизации, которые потребуются для того, чтобы приступить к решению задач в области науки о данных, а также почувствовать себя комфортно с математикой и статистикой, которые находятся в центре этой междисциплинарной практической сферы.

Довольно трудновыполнимая задача для книги. Развивать свои навыки алгоритмизации лучше всего решая прикладные задачи. Прочтя эту книгу, читатель получит хорошее представление о моих способах алгоритмизации прикладных задач, неко-

---

[1] См. **https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century**.

[2] См. **http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram**.

[3] Джон Венн (1834–1923) — английский логик, предложивший схематичное изображение всех возможных пересечений нескольких (часто — трех) множеств (см. **https://ru.wikipedia.org/wiki/Венн,_Джон**). — *Прим. пер.*

торых инструментах, которыми я пользуюсь в работе, и моем подходе к решению задач, связанных с анализом данных. Но это вовсе не означает, что мои способы, инструменты и подход являются единственно возможными. Надеюсь, что мой опыт вдохновит попробовать пойти собственным путем. Все исходные коды и данные из книги доступны на GitHub (**https://github.com/joelgrus/data-science-from-scratch**); они помогут приступить к работе.

То же самое касается и математики. Ею лучше всего заниматься, решая математические задачи. Данная книга, разумеется, не является руководством по математике, и мы не собираемся, по большей части, заниматься ею. Однако действительность такова, что заниматься наукой о данных не получится без некоторых представлений о теории вероятностей, математической статистике и линейной алгебре. Это значит, что по мере надобности мы будем уходить с головой в математические равенства, интуицию, аксиомы и мультяшные версии глобальных математических идей. Надеюсь, читатель не побоится погрузиться в них вместе со мной.

По ходу изложения я также надеюсь передать ощущение, что импровизировать с данными — это увлекательное занятие, потому что, как бы это сказать, импровизировать с ними на самом деле увлекательно! В особенности, если сравнивать с такими альтернативами, как подготовка налоговой декларации или добыча угля.

С нуля

Для работы в области науки о данных разработана масса программных библиотек, вычислительных каркасов, модулей и наборов инструментов, которые эффективно имплементируют[4] наиболее (нередко, и наименее) общие алгоритмы и технические решения, применяемые в науке о данных. Тот, кто станет исследователем данных, несомненно, будет досконально знать научно-вычислительную библиотеку NumPy, библиотеку машинного обучения scikit-learn, библиотеку для анализа данных pandas и множество других. Они прекрасно подходят для решения задач, связанных с наукой о данных. Но они также способствуют тому, чтобы начать решать задачи в области науки о данных, фактически не понимая ее.

В этой книге мы станем двигаться в сторону науки о данных, стартовав с нулевой отметки, а именно займемся разработкой инструментов и имплементацией[4] алгоритмов вручную для того, чтобы разбираться в них лучше. Я вложил немало своего умственного труда в создание ясных, хорошо задокументированных и читаемых имплементаций алгоритмов и примеров. В большинстве случаев инструменты, которые мы станем конструировать, будут иметь не практический, а разъясняющий характер. Они хорошо работают с игрушечными наборами данных, но не справляются с данными "веб-масштаба".

---

[4] Под имплементацией подразумевается воплощение дизайна в структурах данных, классах, объектах, функциях, интерфейсах и т. д. Под реализацией — материализация дизайна в готовом инструменте или законченном продукте.
Поскольку в ряде работ проводится четкое различие между фазами инженерии программно-информационного обеспечения: 1) архитектурный дизайн, 2) имплементация и 3) реализация. — *Прим. пер.*

По ходу изложения я буду отсылать читателя к библиотекам, которые подойдут для применения этих методов на более крупных наборах данных. Но мы их тут использовать не будем.

По поводу того, какой язык программирования лучше всего подходит для усвоения науки о данных, развернулась здоровая полемика. Многие настаивают на языке статистического программирования R (мы называем таких людей неправильными). Некоторые предлагают Java или Scala. Однако, по моему мнению, Python — идеальный вариант.

Он обладает несколькими особенностями, которые делают его наиболее пригодным для изучения и решения задач в области науки о данных:

- он бесплатный;
- он относительно прост в написании кода (и в особенности в понимании);
- он располагает сотнями прикладных библиотек, предназначенных для работы в области науки о данных.

Не решусь назвать Python моим предпочтительным языком программирования. Есть другие языки, которые я нахожу более удобными, продуманными либо просто интересными для программирования. И все же практически всякий раз, когда я начинаю новый проект в области науки о данных, либо, когда мне нужно быстро набросать рабочий прототип либо продемонстрировать концепции этой практической дисциплины ясным и легким для понимания способом, я всякий раз в итоге применяю Python. И поэтому в этой книге используется Python.

Эта книга не предназначена для того, чтобы научить программировать на Python (хотя я почти уверен, что, прочтя ее, этому можно немного научиться). Тем не менее я проведу интенсивный курс программирования на Python (ему будет посвящена целая глава), где будут высвечены характерные черты, которые в данном случае приобретают особую важность. Однако если знания, как программировать на Python (или о программировании вообще), отсутствуют, то читателю остается самому дополнить эту книгу чем-то вроде руководства по Python для начинающих.

В последующих частях нашего введения в науку о данных будет принят такой же подход — углубляться в детали там, где они оказываются решающими или показательными. В других ситуациях на читателя возлагается задача домысливать детали самому или заглядывать в Википедию.

За годы работы в отрасли я подготовил ряд специалистов в области науки о данных. Хотя не все из них стали меняющими мир супер-мега-рок-звездами в области анализа данных, тем не менее я их всех выпустил более подготовленными специалистами, чем они были до этого. И я все больше убеждаюсь в том, что любой, у кого есть некоторые математические способности вкупе с определенным набором навыков в программировании, располагает всем необходимым для решения задач в области науки о данных. Все, чего она требует, — это лишь пытливый ум, готовность усердно трудиться и наличие данного пособия. Отсюда и эта книга.

Комментарий переводчика

Автор ставшей популярной книги "Наука о данных с чистого листа" легко, доступно и иногда с юмором повествует о сложных вещах, составляющих фундамент науки о данных и машинного обучения. Хотя автор в предисловии ко второму изданию упоминает о внесенных изменениях, стоит особо отметить, что второе издание книги дополнено главами о глубоком обучении и этике данных. Кроме того, книга содержит несколько новых разделов, в частности о рекуррентных нейронных сетях, векторных вложениях слов и разложении матриц и некоторые другие, а также ряд новых примеров; всё подкреплено исходным кодом, размещенным в репозитории книги на GitHub. В книге детально разбирается пример разработки глубокой нейронной сети по образу библиотеки Keras. В исходный код внесен ряд изменений, отражающих последние тренды в развитии языка Python, в частности широко используются аннотации типов, не характерные для ранних версий языка Python, и типизированные именованные кортежи.

Приведенные в книге примеры были протестированы на Python 3.7.2 в среде Windows. Перевод второго издания книги был тоже полностью переработан с учетом терминологических уточнений и стилистических поправок.

Настоящая книга рекомендуется широкому кругу специалистов, в том числе в области машинного обучения, начинающим исследователям данных, преподавателям, студентам, а также всем, кто интересуется программированием и решением вычислительных задач.

Об авторе

Джоэл Грас является инженером-исследователем в Институте искусственного интеллекта имени Аллена. Ранее он работал инженером-программистом в компании Google и исследователем данных в нескольких стартапах. Он живет в Сиэтле, где регулярно посещает неформальные встречи специалистов в области науки о данных. Он редко пишет в свой блог **joelgrus.com** и всегда доступен в Twitter по хештегу **@joelgrus**.

ГЛАВА 1
Введение

— Данные! Где данные? — раздраженно восклицал он. — Когда под рукой нет глины, из чего лепить кирпичи?[1]

– Артур Конан Дойл

Воцарение данных

Мы живем в мире, страдающем от переизбытка данных. Веб-сайты отслеживают любое нажатие любого пользователя. Смартфоны накапливают сведения о вашем местоположении и скорости в ежедневном и ежесекундном режиме. "Оцифрованные" селферы носят шагомеры на стероидах, которые, не переставая, записывают их сердечные ритмы, особенности движения, схемы питания и сна. Умные авто собирают сведения о манерах вождения своих владельцев, умные дома — об образе жизни своих обитателей, а умные маркетологи — о наших покупательских привычках. Сам Интернет представляет собой огромный граф знаний, который, среди всего прочего, содержит обширную гипертекстовую энциклопедию, специализированные базы данных о фильмах, музыке, спортивных результатах, игровых автоматах, мемах и коктейлях... и слишком много статистических отчетов (причем некоторые почти соответствуют действительности!) от слишком большого числа государственных исполнительных органов, и все это для того, чтобы вы объяли необъятное.

В этих данных кроятся ответы на бесчисленные вопросы, которые никто никогда не думал задавать. Эта книга научит вас, как их находить.

Что такое наука о данных?

Существует шутка, что исследователь данных — это тот, кто знает статистику лучше, чем инженер-информатик, а информатику — лучше, чем инженер-статистик. Не утверждаю, что это хорошая шутка, но на самом деле (в практическом плане) некоторые исследователи данных действительно являются инженерами-статистиками, в то время как другие почти неотличимы от инженеров программного обеспечения. Некоторые являются экспертами в области машинного обучения, в то время как другие не смогли бы машинно обучиться, чтобы найти выход из детского сада. Некоторые имеют ученые степени доктора наук с впечат-

[1] Реплика Шерлока Холмса из рассказа Артура Конан Дойла (1859–1930) "Медные буки". — *Прим. пер.*

ляющей историей публикаций, в то время как другие никогда не читали академических статей (хотя им должно быть стыдно). Короче говоря, в значительной мере не важно, как определять понятие науки о данных, потому что всегда можно найти практикующих исследователей данных, для которых это определение будет всецело и абсолютно неверным[2].

Тем не менее этот факт не остановит нас от попыток. Мы скажем, что исследователь данных — это тот, кто извлекает ценные наблюдения из запутанных данных. В наши дни мир переполнен людьми, которые пытаются превратить данные в сущностные наблюдения.

Например, веб-сайт знакомств OkCupid просит своих членов ответить на тысячи вопросов, чтобы отыскать наиболее подходящего для них партнера. Но он также анализирует эти результаты, чтобы вычислить виды безобидных вопросов, с которыми вы можете обратиться, чтобы узнать, насколько высока вероятность близости после первого же свидания[3].

Компания Facebook просит вас указывать свой родной город и нынешнее местоположение, якобы чтобы облегчить вашим друзьям находить вас и связываться с вами. Но она также анализирует эти местоположения, чтобы определить схемы глобальной миграции и места проживания фанатов различных футбольных команд[4].

Крупный оператор розничной торговли Target отслеживает покупки и взаимодействия онлайн и в магазине. Он использует данные, чтобы строить предсказательные модели[5] в отношении того, какие клиентки беременны, чтобы лучше продавать им товары, предназначенные для младенцев.

В 2012 г. избирательный штаб Барака Обамы нанял десятки исследователей данных, которые вовсю копали и экспериментировали, чтобы определить избирателей, которым требовалось дополнительное внимание, при этом подбирая оптимальные обращения и программы по привлечению финансовых ресурсов, которые направлялись в адрес конкретных получателей, и сосредотачивая усилия по повышению явки избирателей там, где эти усилия могли быть наиболее успешными. А в предвыборной кампании Трампа 2016 года его команда протестировала ошеломляющее разнообразие онлайновых объявлений[6] и проанализировала данные с целью выявления того, что сработало, а что нет.

---

[2] Наука о данных — это практическая дисциплина, которая занимается изучением методов обобщаемого усвоения знаний из данных. Она состоит из различных составляющих и основывается на методах и теориях из многих областей, включая обработку сигналов, математику, вероятностные модели, машинное и статистическое обучение, программирование, технологии данных, распознавание образов, теорию обучения, визуальный анализ, моделирование неопределенности, организацию хранилищ данных, а также высокоэффективные вычисления с целью извлечения смысла из данных и создания продуктов обработки данных. — *Прим. пер.*

[3] См. **https://theblog.okcupid.com/the-best-questions-for-a-first-date-dba6adaa9df2**.

[4] См. **https://www.facebook.com/notes/facebook-data-science/nfl-fans-on-facebook/10151298370823859**.

[5] См. **https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html**.

[6] См. **https://www.wired.com/2016/11/facebook-won-trump-election-not-just-fake-news/**.

И прежде чем вы почувствуете пресыщение, добавим, что некоторые аналитики данных время от времени используют свои навыки во благо[7], делая правительство эффективнее, помогая бездомным и совершенствуя здравоохранение. И конечно же вы не нанесете вреда своей карьере, если вам нравится заниматься поисками наилучшего способа, как заставить людей щелкать на рекламных баннерах.

Оправдание для выдумки: DataSciencester

Поздравляем! Вас только что приняли на работу, чтобы вы возглавили усилия в области науки о данных в DataSciencester, уникальной социальной сети для исследователей данных.

Когда я писал первое издание этой книги, я думал, что "социальная сеть для исследователей данных" — это забавная и глупая выдумка. С тех пор люди на самом деле создали социальные сети для исследователей данных и собрали гораздо больше денег от венчурных инвесторов, чем я получил от своей книги. По всей видимости, здесь есть ценный урок о глупых выдумках и/или книгоиздании.

Предназначенная исключительно для исследователей данных, тем не менее DataSciencester еще ни разу не вкладывалась в развитие собственной практической деятельности в этой области (справедливости ради, она ни разу по-настоящему не вкладывалась даже в развитие своего продукта). Теперь это будет вашей работой! На протяжении всей книги мы будем изучать понятия этой практической дисциплины путем решения задач, с которыми вы будете сталкиваться на работе. Мы будем обращаться к данным, иногда поступающим прямо от пользователей, иногда полученным в результате их взаимодействий с веб-сайтом, а иногда даже взятыми из экспериментов, которые будем конструировать сами.

И поскольку в DataSciencester царит дух новизны, то мы займемся построением собственных инструментов с нуля. В результате у вас будет достаточно твердое понимание основ науки о данных, и вы будете готовы применить свои навыки в любой компании или же в любых других задачах, которые окажутся для вас интересными.

Добро пожаловать на борт и удачи! (Вам разрешено носить джинсы по пятницам, а туалет — по коридору направо.)

Выявление ключевых звеньев

Итак, настал ваш первый рабочий день в DataSciencester, и директор по развитию сети полон вопросов о ее пользователях. До сего дня ему не к кому было обратиться, поэтому он очень воодушевлен вашим прибытием.

В частности, он хочет, чтобы вы установили, кто среди специалистов является "ключевым звеном". Для этих целей он передает вам "снимок" всей социальной сети. (В реальной жизни обычно никто не торопится вручать вам требующиеся данные. *Глава 9* посвящена способам сбора данных.)

---

[7] См. **https://www.marketplace.org/2014/08/22/tech/beyond-ad-clicks-using-big-data-social-good**.

Как выглядит этот "снимок" данных? Он состоит из списка пользователей `users`, где каждый пользователь представлен словарем `dict`, состоящим из идентификатора `id` (т. е. числа) пользователя и его или ее имени `name` (которое по одному из необычайных космических совпадений рифмуется с идентификатором `id`):

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

Кроме того, он передает вам данные о "дружеских отношениях" `friendships` в виде списка кортежей, состоящих из пар идентификаторов пользователей:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

Например, кортеж `(0, 1)` означает, что исследователь с `id 0` (Hero) и исследователь с `id 1` (Dunn) являются друзьями. Сеть представлена на рис. 1.1.

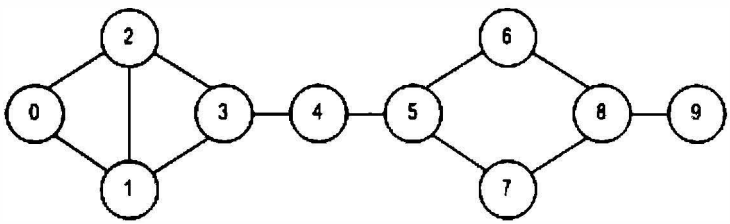


Рис. 1.1. Сеть DataSciencester

Представить дружеские отношения в виде списка пар не самый простой способ работы с ними. Для того чтобы найти все дружеские отношения для пользователя 1, вы должны перебрать каждую пару в поисках пар, содержащих 1. Если бы у вас было много пар, то на это ушло бы много времени.

Вместо этого давайте создадим словарь, в котором ключи — это идентификаторы пользователей, а значения — списки идентификаторов друзей. (Просмотр по словарю вверх будет очень быстрым.)

Вам не следует прямо сейчас погружаться в подробности реализации кода. В *главе 2* будет предложен интенсивный курс языка Python. Пока же следует всего лишь попытаться получить общее представление о том, что мы делаем.

Создавая словарь, нам все равно придется просмотреть каждую пару, но зато это надо будет сделать всего один раз, и после этого мы получим дешевые просмотры словаря:

```
# Инициализировать словарь пустым списком для идентификатора
# каждого пользователя
friendships = {user["id"]: [] for user in users}

# И перебрать все дружеские пары, заполняя их:
for i, j in friendships:
    friendships[i].append(j) # Добавить j как друга для i
    friendships[j].append(i) # Добавить i как друга для j
```

После того как в словаре каждого пользователя будет размещен список его друзей, получившийся граф можно легко опросить, например, на предмет среднего числа связей.

Сперва находим суммарное число связей, сложив длины всех списков друзей `friends`:

```
# число друзей
def number_of_friends(user):
    """Сколько друзей есть у пользователя user?"""
    user_id = user["id"]
    friend_ids = friendships[user_id]
    return len(friend_ids)

total_connections = sum(number_of_friends(user)  # Суммарное число
                        for user in users)       # связей - 24
```

А затем просто делим сумму на число пользователей:

```
num_users = len(users)                    # Длина списка пользователей
avg_connections = total_connections / num_users      # 24 / 10 = 2.4
```

Так же легко находим наиболее связанных людей, т. е. лиц, имеющих наибольшее число друзей.

Поскольку пользователей не слишком много, их можно упорядочить по убыванию числа друзей:

```
# Создать список в формате (id пользователя, число друзей)
num_friends_by_id = [(user["id"], number_of_friends(user))
                     for user in users]

num_friends_by_id.sort(                                  # Отсортировать список по полю
       key=lambda id_and_friends: id_and_friends[1], # num_friends
       reverse=True)                                     # в убывающем порядке

# Каждая пара представлена кортежем (user_id, num_friends),
# т. е. идентификтором пользователя и числом друзей
# [(1, 3), (2, 3), (3, 3), (5, 3), (8, 3),
#  (0, 2), (4, 2), (6, 2), (7, 2), (9, 1)]
```

Всю проделанную работу можно рассматривать как способ определить лиц, которые так или иначе занимают в сети центральное место. На самом деле то, что мы вычислили, представляет собой сетевую метрику связи под названием *центральность по степени* узлов (degree centrality) (рис. 1.2).

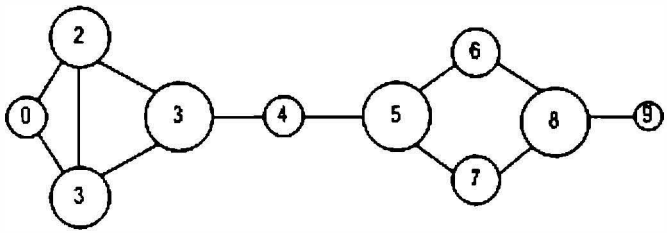


Рис. 1.2. Сеть DataSciencester, измеренная метрикой степени узлов

Достоинство этой метрики заключается в простоте вычислений, однако она не всегда дает нужные или ожидаемые результаты. Например, в DataSciencester пользователь Thor (`id 4`) имеет всего две связи, тогда как Dunn (`id 1`) — три. Несмотря на это, схема сети показывает, что Thor находится ближе к центру. В *главе 21* мы займемся изучением сетей подробнее и рассмотрим более сложные представления о центральности, которые могут лучше или хуже соответствовать интуиции.

Исследователи данных, которых вы должны знать

Вы еще не успели заполнить все страницы формуляра найма новых сотрудников, как директор по побратимским связям подходит к вашему столу. Она хочет простимулировать рост связей среди членов сети и просит вас разработать подсистему рекомендации новых друзей "Исследователи данных, которых вы должны знать".

Ваша первая реакция — предположить, что пользователь может знать друзей своих друзей. Их легко вычислить: надо лишь просмотреть друзей каждого пользователя и собрать результаты:

```
# Список id друзей пользователя user (плохой вариант)
def foaf_ids_bad(user):
    # "foaf" означает "товарищ товарища"
    return [foaf_id
            for friend_id in friendships[user["id"]]
            for foaf_id in friendships[friend_id]]
    return [foaf["id"]
            for friend in user["friends"]
            for foaf in friend["friends"]]
```

Если эту функцию вызвать с аргументом `users[0]` (Hero), она покажет:

```
[0, 2, 3, 0, 1, 3]
```

Список содержит пользователя 0 (два раза), т. к. пользователь Hero на самом деле дружит с обоими своими друзьями; содержит пользователей 1 и 2, хотя оба эти

пользователя уже дружат с Hero; и дважды содержит пользователя 3, поскольку Chi достижима через двух разных друзей:

```
print(friendships[0]  # [1, 2]
print(friendships[1]  # [0, 2, 3]
print(friendships[2]  # [0, 1, 3]
```

Информация о том, что некто может стать другом вашего друга несколькими путями, похоже, заслуживает внимания, поэтому, напротив, стоит добавить количество взаимных друзей, а для этих целей точно потребуется вспомогательная функция, которая будет исключать лиц, уже известных пользователю:

```
from collections import Counter  # Не загружается по умолчанию

def friends_of_friends(user):
    user_id = user["id"]
    return Counter(
        foaf_id
        for friend_id in friendships[user_id]   # По каждому моему другу
        for foaf_id in friendships[friend_id]   # отыскать его друзей,
        if foaf_id != user_id                   # которые не являются мной
        and foaf_id not in friendships[user_id] # и не являются моими друзьями
    )

print(friends_of_friends(users[3]))             # Counter({0: 2, 5: 1})
```

Такая реализация безошибочно сообщает Chi (`id 3`), что у нее с Hero (`id 0`) есть двое взаимных друзей, а с Clive (`id 5`) — всего один такой друг.

Будучи исследователем данных, вы понимаете, что вам было бы интересно встречаться с пользователями, которые имеют одинаковые интересы (кстати, это хороший пример аспекта "достаточной компетентности" из науки о данных). После того как вы поспрашивали вокруг, вам удалось получить на руки данные в виде списка пар (`идентификатор пользователя, интересующая тема`):

```
interests = [
     (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
     (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
     (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
     (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
     (2, "numpy"), (2, "statsmodels"), (2, "pandas"),
     (3, "R"), (3, "Python"),
     (3, "statistics"), (3, "regression"), (3, "probability"),
     (4, "machine learning"), (4, "regression"), (4, "decision trees"),
     (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
     (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
     (6, "probability"), (6, "mathematics"), (6, "theory"),
     (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
     (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
```

```
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

Например, у Него (`id 0`) нет общих друзей с Klein (`id 9`), но они оба разделяют интерес в машинном обучении.

Легко построить функцию, которая отыскивает пользователей, интересующихся определенной темой:

```
# Исследователи данных, которым нравится целевая тема target_interest
def data_scientists_who_like(target_interest):
    """Отыскать идентификаторы всех пользователей,
       которым интересна целевая тема."""
    return [user_id
            for user_id, user_interest in interests
            if user_interest == target_interest]
```

Эта функция работает, однако в такой реализации она просматривает весь список интересов при каждом поиске. Если пользователей и интересов много (либо планируется выполнять много запросов), то, по-видимому, лучше создать индекс пользователей, сгруппированный по интересу:

```
from collections import defaultdict

# Идентификаторы пользователей по идентификатору темы.
# Ключи - это интересующие темы,
# значения - списки идентификаторов пользователей с этой темой
user_ids_by_interest = defaultdict(list)

for user_id, interest in interests:
    user_ids_by_interest[interest].append(user_id)
```

И еще индекс тем, сгруппированный по пользователям:

```
# Идентификаторы тем по идентификатору пользователя.
# Ключи - это идетификаторы пользователей,
# значения - списки тем для конкретного идетификатора
interests_by_user_id = defaultdict(list)

for user_id, interest in interests:
    interests_by_user_id[user_id].append(interest)
```

Теперь легко найти лицо, у которого больше всего общих интересов с заданным пользователем. Для этого нужно:

- выполнить обход интересующих пользователя тем;
- по каждой теме выполнить обход других пользователей, интересующихся той же самой темой;
- подсчитать, сколько раз встретятся другие пользователи.

```
# Наиболее общие интересы с пользователем user
def most_common_interests_with(user):
    return Counter(
        interested_user_id
        for interest in interests_by_user_id[user["id"]]
        for interested_user_id in user_ids_by_interest[interest]
        if interested_user_id != user["id"])
```

Вы могли бы применить эту функцию для построения более богатого функционала рекомендательной подсистемы "Исследователи данных, которых вы должны знать", основываясь на сочетании взаимных друзей и общих интересов. Эти виды приложений мы разведаем в *главе 22*.

Зарплаты и опыт работы

В тот самый момент, когда вы собираетесь пойти пообедать, директор по связям с общественностью обращается к вам с вопросом: можете ли вы предоставить для веб-сайта какие-нибудь примечательные факты об уровне зарплат исследователей данных? Разумеется, данные о зарплатах имеют конфиденциальный характер, тем не менее ему удалось снабдить вас анонимным набором данных, содержащим зарплату каждого пользователя (в долларах) и его опыт работы исследователем данных (в годах):

```
# Зарплаты и стаж
salaries_and_tenures = [(83000, 8.7), (88000, 8.1),
                        (48000, 0.7), (76000, 6),
                        (69000, 6.5), (76000, 7.5),
                        (60000, 2.5), (83000, 10),
                        (48000, 1.9), (63000, 4.2)]
```

Первым делом вы выводите данные на диаграмму (в *главе 3* мы рассмотрим, как это делать). Результаты можно увидеть на рис. 1.3.

Совершенно очевидно, что лица с более продолжительным опытом, как правило, зарабатывают больше. Но как это превратить в примечательный факт? Ваша первая попытка — взять среднюю арифметическую зарплату по каждому стажу:

```
# Зарплата в зависимости от стажа.
# Ключи - это годы, значения - списки зарплат для каждого стажа
salary_by_tenure = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    salary_by_tenure[tenure].append(salary)

# Средняя зарплата в зависимости от стажа.
# Ключи - это годы, каждое значение - средняя зарплата по этому стажу
average_salary_by_tenure = {
    tenure : sum(salaries) / len(salaries)
    for tenure, salaries in salary_by_tenure.items()
}
```

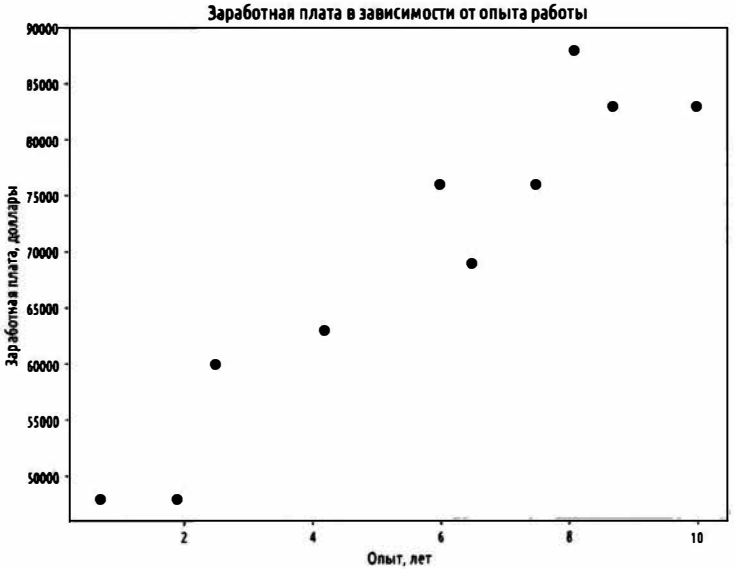


Рис. 1.3. Зависимость заработной платы от опыта работы

Но, как оказалось, это не несет какой-то практической пользы, т. к. у всех разный стаж, и значит, мы просто сообщаем о зарплатах отдельных пользователей:

```
{0.7: 48000.0,
 1.9: 48000.0,
 2.5: 60000.0,
 4.2: 63000.0,
 6:   76000.0,
 6.5: 69000.0,
 7.5: 76000.0,
 8.1: 88000.0,
 8.7: 83000.0,
 10:  83000.0}
```

Целесообразнее разбить продолжительности стажа на группы:

```
# Стажная группа
def tenure_bucket(tenure):
    if tenure < 2:
        return "менее двух"
    elif tenure < 5:
        return "между двумя и пятью"
    else:
        return "более пяти"
```

Затем сгруппировать все зарплаты, которые соответствуют каждой группе:

```
# Зарплата в зависимости от стажной группы.
# Ключи - это стажные группы, значения - списки зарплат в этой группе.
```

```
# Словарь содержит списки зарплат, соответствующие каждой стажной группе
salary_by_tenure_bucket = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    bucket = tenure_bucket(tenure)
    salary_by_tenure_bucket[bucket].append(salary)
```

И в конце вычислить среднюю арифметическую зарплату по каждой группе:

```
# Средняя зарплата по группе.
# Ключи - это стажные группы, значения - средняя зарплата по этой группе
average_salary_by_bucket = {
    tenure_bucket : sum(salaries) / len(salaries)
    for tenure_bucket, salaries in salary_by_tenure_bucket.iteritems()
}
```

Результат выглядит гораздо интереснее:

```
{'между двумя и пятью': 61500.0,
 'менее двух': 48000.0,
 'более пяти': 79166.66666666667}
```

К тому же вы получаете короткую и запоминающуюся "реплику"[8]: "Исследователи данных с опытом работы свыше 5 лет зарабатывают на 65% больше, чем исследователи с малым или отсутствующим опытом работы в этой области!"

Впрочем, мы выбрали группы произвольным образом. На самом деле нам хотелось бы получить некую констатацию степени воздействия на уровень зарплаты (в среднем) дополнительного года работы. Кроме получения более энергичного примечательного факта, это позволило бы строить предсказания о неизвестных зарплатах. Мы разведаем эту идею в *главе 14*.

Оплата аккаунтов

Когда вы возвращаетесь к своему рабочему столу, директор по доходам уже вас ожидает. Она желает получше разобраться в том, какие пользователи оплачивают свой аккаунт, а какие нет. (Она знает их имена, но эти сведения не имеют особой оперативной ценности.)

Вы обнаруживаете, что, по всей видимости, между опытом работы и оплатой аккаунта есть некое соответствие:

```
0.7 оплачено
1.9 не оплачено
2.5 оплачено
4.2 не оплачено
```

[8] Дословно "звуковой укус" от англ. *sound bite*. Согласно исследованиям, способность публики сосредоточиваться на том, что она слышит или видит, ограничена средним временем внимания, равным 30 секундам, поэтому вся реклама говорит эффектными репликами, или "звуковыми укусами", максимально пользуясь ограниченным сроком внимания публики. — *Прим. пер.*

```
6   не оплачено
6.5 не оплачено
7.5 не оплачено
8.1 не оплачено
8.7 оплачено
10  оплачено
```

Пользователи с малым и большим опытом работы, как правило, аккаунты оплачивают, а пользователи со средним опытом — нет.

Следовательно, если создать модель — хотя этих данных определенно недостаточно для того, чтобы строить на них модель, — то можно попытаться предсказать, что пользователи с малым и большим опытом работы оплатят свои акаунты, а середняки — нет:

```
# Предсказать оплату, исходя из стажа
def predict_paid_or_unpaid(years_experience):
    if years_experience < 3.0:
        return "оплачено"
    elif years_experience < 8.5:
        return "не оплачено"
    else:
        return "оплачено"
```

Разумеется, мы взяли здесь только точки отсечения.

Имея больше данных (и больше математики), мы могли бы построить модель, предсказывающую вероятность оплаты пользователем аккаунта, исходя из его опыта работы. Мы займемся исследованием этого типа задач в *главе 16*.

Интересующие темы

В тот момент, когда ваш первый рабочий день подходит к концу, директор по стратегии содержательного наполнения запрашивает у вас данные о том, какими темами пользователи интересуются больше всего, чтобы соответствующим образом распланировать свой календарь работы с блогом. У вас уже имеются сырые данные из проекта подсистемы рекомендации новых друзей:

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"),
    (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
```

```
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

Простой, если не самый впечатляющий, способ найти наиболее популярные темы, которые вызывают интерес, — просто подсчитать количества появлений слов:

1. Перевести каждую тему в нижний регистр (пользователи могут напечатать слова прописными).
2. Разбить ее на слова.
3. Подсчитать результаты.

В коде это выглядит так:

```
# Слова и количества появлений
words_and_counts = Counter(word
                           for user, interest in interests
                           for word in interest.lower().split())
```

Это позволяет легко перечислить слова, которые встречаются более одного раза:

```
for word, count in words_and_counts.most_common():
    if count > 1:
        print(word, count)
```

В итоге получаем ожидаемый результат (если только для разбиения строки на два слова вы не планируете использовать библиотеку scikit-learn, которая в этом случае не даст ожидаемых результатов):

```
learning 3
java 3
python 3
big 3
data 3
hbase 2
regression 2
cassandra 2
statistics 2
probability 2
hadoop 2
networks 2
machine 2
neural 2
scikit-learn 2
r 2
```

Мы обратимся к более продвинутым способам извлечения тематик из данных в *главе 20*.

Поехали!

Это был успешный день! Уставший, вы ускользаете из здания, прежде чем кому-нибудь еще удается задать вам вопрос по поводу чего-нибудь еще. Хорошенько отоспитесь, потому что на завтра намечена программа ориентации для новых сотрудников. (Да-да, вы проработали целый день, даже не пройдя программы ориентации! Поставьте этот вопрос перед кадровой службой.)

Интенсивный курс языка Python

> Уже 25 лет, как народ сходит с ума от Пайтона, чему я не перестаю удивляться.
> – *Майкл Пейлин*[1]

Все новые сотрудники DataSciencester обязаны проходить программу ориентации персонала, самой интересной частью которой является интенсивный курс языка программирования Python.

Это не всеобъемлющее руководство по языку. Напротив, интенсивный курс призван выделить те элементы языка, которые будут наиболее важны в дальнейшем (некоторым из них зачастую уделяют скромное внимание даже в учебниках). Если вы никогда не использовали Python прежде, то его, вероятно, стоит дополнить неким руководством для начинающих.

Дзен языка Python

В основе языка лежат несколько *кратких принципов конструирования программ*[2], именуемых "дзен языка Python". Их текст выводится на экран интерпретатором, если набрать команду `import this`.

Один из самых обсуждаемых следующий:

> "Должен существовать один — и, желательно, только один — очевидный способ сделать это".

Код, написанный в соответствии с этим "очевидным" способом (и, возможно, не совсем очевидным для новичка) часто характеризуется как "Python'овский". Несмотря на то, что темой данной книги не является программирование на Python, мы будем иногда противопоставлять Python- и не-Python-способы достигнуть одной и той же цели, но, как правило, предпочтение будет отдаваться Python'овским способам.

Другие касаются эстетики:

> "Красивое лучше, чем уродливое. Явное лучше, чем неявное. Простое лучше, чем сложное" —

---

[1] Майкл Пейлин (род. 1943) — британский актер, писатель, телеведущий, участник комик-группы, известной благодаря юмористическому телешоу "Летающий цирк Монти Пайтона", выходившему на BBC в 1970-х годах, в честь которого был назван язык Python — *Прим. пер.*

[2] См. **http://legacy.python.org/dev/peps/pep-0020/**.

и представляют идеалы, к которым мы будем стремиться в нашем программном коде.

Установка языка Python

Поскольку инструкции по установке могут измениться, а напечатанные книги — нет, обновленные инструкции по установке Python можно найти в репозитории книги на GitHub[3]. Если представленные здесь инструкции не работают, то обратитесь к репозиторию.

Python можно скачать с веб-сайта **python.org**[4]. Однако если он еще не установлен, то вместо этого источника рекомендуем дистрибутивный пакет Anaconda[5], который уже включает в себя большинство библиотек, необходимых для работы в области науки о данных.

Когда писалась первая версия настоящей книги, версия языка Python 2.7 была по-прежнему предпочтительной для большинства исследователей данных, и, соответственно, первая версия книги основывалась на Python 2.7[6].

Однако за последние несколько лет почти все, кому это важно, перешли на Python 3. В последних версиях Python есть много функциональных средств, которые облегчают написание чистого кода, и мы будем использовать возможности, доступные только в Python 3.6 или более поздней версии. Это означает, что вы должны установить Python 3.6 или более позднюю его версию. (Кроме того, многие полезные библиотеки заканчивают поддержку Python 2.7, что является еще одной причиной для перехода.)

Виртуальные среды

Начиная со следующей главы, мы будем использовать библиотеку matplotlib для создания графиков и диаграмм. Эта библиотека не является стержневой частью Python; вы должны установить ее самостоятельно. Для каждого выполняемого проекта в области науки о данных требуется определенная комбинация внешних библиотек, иногда конкретных версий, которые отличаются от версий, используемых для других проектов. Если у вас установлен Python для одиночного пользования, то эти библиотеки будут конфликтовать и вызывать всевозможные проблемы.

Стандартное решение — использовать виртуальные среды, т. е. изолированные среды Python, которые поддерживают собственные версии библиотек Python (и в зависимости от того, как вы настраиваете среду самого Python).

---

[3] См. **https://github.com/joelgrus/data-science-from-scratch/blob/master/INSTALL.md**.

[4] См. **https://www.python.org/**.

[5] См. **https://www.anaconda.com/download/**.

[6] Первое издание книги в русском переводе вышло уже адаптированным под версию Python 3.6 (издательство "БХВ-Петербург"). — *Прим. пер.*

Я рекомендовал вам установить дистрибутив Anaconda Python и поэтому ниже собираюсь объяснить, как работают среды Anaconda. Если вы не используете Anaconda, то можете использовать встроенный модуль `venv` или установить библиотеку virtualenv. В этом случае вы должны следовать их указаниям.

Для создания виртуальной среды (Anaconda) выполните следующие действия:

```
# Создать среду Python 3.6 с именем "dsfs"
conda create -n dsfs python=3.6
```

Следуйте подсказкам, и у вас будет виртуальная среда под названием "dsfs" с инструкциями:

```
# Для активации этой среды используйте:
# > source activate dsfs
#
# Для деактивации активной среды используйте:
# > source deactivate
#
```

Как предписано, затем вы активируете среду с помощью команды:

```
source activate dsfs
```

и в этом месте ваша командная строка должна измениться, указав активную среду. На моем MacBook подсказка теперь выглядит так:

```
(dsfs) ip-10-0-0-198:~ joelg$
```

До тех пор пока эта среда активна, все устанавливаемые библиотеки будут устанавливаться только в среде `dsfs`. После того как вы закончите читать эту книгу и перейдете к собственным проектам, вы должны создавать для них собственную среду.

Теперь, когда у вас есть ваша виртуальная среда, стоит установить полнофункциональную оболочку языка Python под названием IPython[7]:

```
python -m pip install ipython
```

Дистрибутив Anaconda поставляется со своим менеджером пакетов `conda`, но вы также можете использовать стандартный пакетный менеджер `pip`[8], что мы и будем делать.

В остальной части этой книги мы будем исходить из того, что вы создали и активировали указанную выше виртуальную среду Python 3.6 (хотя вы можете назвать ее как хотите), а последующие главы могут опираться на библиотеки, которые я порекомендовал вам установить в более ранних главах.

В качестве хорошей самодисциплины всегда следует работать в виртуальной среде и никогда не использовать "базовую" конфигурацию Python.

---

[7] См. **http://ipython.org/**.

[8] См. **https://pypi.python.org/pypi/pip**.

Пробельное форматирование

Во многих языках программирования для разграничения блоков кода применяются фигурные скобки. В Python используются отступы:

```
# Знак 'решетки' знаменует начало комментария.
# Сам Python игнорирует комментарии, но они полезны тем, кто читает код.
# Пример отступов во вложенных циклах for
for i in [1, 2, 3, 4, 5]:
    print(i)            # Первая строка в блоке for i
    for j in [1, 2, 3, 4, 5]:
        print(j)        # Первая строка в блоке for j
        print(i + j)    # Последняя строка в блоке for j
    print(i)            # Последняя строка в блоке for i
print("Циклы закончились")
```

Это делает код легко читаемым, но в то же время заставляет следить за форматированием.

Программисты часто спорят о том, что использовать лучше для отступов: знаки табуляции или знаки пробела. Для многих языков это не имеет большого значения; однако Python считает знаки табуляции и знаки пробела разными отступами и не сможет выполнить ваш код, если вы их перемешиваете. При написании на Python всегда следует использовать пробелы, а не табуляцию. (Если вы пишете код в редакторе, то вы можете настроить его так, чтобы клавиша <Tab> всегда вставляла пробелы.)

Пробел внутри круглых и квадратных скобок игнорируется, что облегчает написание многословных выражений:

```
# Пример многословного выражения
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 +
                           11 + 12 + 13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

и легко читаемого кода:

```
# Список списков
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Такой список список легче читается
easier_to_read_list_of_lists = [ [1, 2, 3],
                                 [4, 5, 6],
                                 [7, 8, 9] ]
```

Для продолжения инструкции на следующей строке используется обратная косая черта, впрочем такая запись будет применяться редко:

```
two_plus_three = 2 + \
                 3
```

Вследствие форматирования кода пробельными символами возникают трудности при копировании и вставке кода в оболочку Python. Например, попытка скопировать следующий код:

```
for i in [1, 2, 3, 4, 5]:

    # Обратите внимание на пустую строку
    print(i)
```

в стандартную оболочку Python вызовет ошибку:

```
# Ошибка нарушения отступа: ожидается блок с отступом
IndentationError: expected an indented block
```

потому что интерпретатор воспринимает пустую строку как сигнал о конце блока кода с циклом `for`.

Оболочка IPython располагает "волшебной" функцией `%paste`, которая правильно вставляет все то, что находится в буфере обмена, в том числе пробельные символы. Только одно это уже является веской причиной для того, чтобы использовать IPython.

Модули

Некоторые библиотеки среды программирования на основе Python не загружаются по умолчанию. Это касается как функционала, составляющего часть языка Python, так и сторонних инструментов, загружаемых самостоятельно. Для того чтобы их можно было использовать, необходимо *импортировать* модули, которые их содержат.

Один из подходов заключается в том, чтобы просто импортировать сам модуль:

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

Здесь `re` — это название модуля, содержащего функции и константы для работы с регулярными выражениями. Импортировав весь модуль таким способом, можно обращаться к функциям, предваряя их префиксом `re`.

Если в коде переменная с именем `re` уже есть, то можно воспользоваться псевдонимом модуля:

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

Псевдоним используют также в тех случаях, когда импортируемый модуль имеет громоздкое имя или когда в коде происходит частое обращение к модулю. Например, при визуализации данных на основе модуля `matplotlib` для него обычно используют следующий стандартный псевдоним:

```
import matplotlib.pyplot as plt


plt.plot(...)
```

Если из модуля необходимо получить несколько конкретных значений, то их можно импортировать в явном виде и использовать без ограничений:

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

Небрежный программист может импортировать все содержимое модуля в пространство имен, вследствие чего непреднамеренно будут перезаписаны уже объявленные переменные:

```
match = 10
from re import *    # Оппа, в модуле re есть функция match
print(match)        # "<function re.match>"
```

Следует избегать такого рода простых решений.

Функции

Функция — это правило, принимающее ноль или несколько аргументов на входе и возвращающее соответствующий результат на выходе. В Python функции обычно определяются при помощи инструкции `def`:

```
def double(x):
    """
    Когда требуется, здесь размещают
    многострочный документирующий комментарий docstring,
    который поясняет, что именно функция вычисляет.
    Например, указанная функция умножает входящее значение на 2
    """
    return x * 2
```

Функции в Python рассматриваются как объекты *первого класса*. Это означает, что их можно назначать переменным и передавать в другие функции так же, как любые другие аргументы:

```
# Применить функцию f к единице
def apply_to_one(f):
    """Вызывает функцию f с единицей в качестве аргумента"""
    return f(1)

my_double = double           # Ссылка на ранее определенную функцию
x = apply_to_one(my_double)  # равно 2
```

Кроме того, можно легко создавать короткие *анонимные функции* или *лямбда-выражения*:

```
y = apply_to_one(lambda x: x + 4)     # равно 5
```

Лямбда-выражения можно назначать переменным. Однако большинство программистов, напротив, рекомендуют пользоваться инструкцией `def`:

```
another_double = lambda x: 2 * x      # Так не делать
def another_double(x): return 2 * x   # Лучше так
```

Параметрам функции, помимо этого, можно передавать аргументы по умолчанию[9], которые следует указывать только тогда, когда ожидается значение, отличающееся от значения по умолчанию:

```
def my_print(message="мое сообщение по умолчанию"):
    print(message)

my_print("привет")  # Напечатает 'привет'
my_print()          # Напечатает 'мое сообщение по умолчанию'
```

Иногда полезно указывать аргументы по имени:

```
def full_name(first = "некто", last = "как-то там"):
    return first + " " + last

full_name("Джоэл", "Грас")  # "Джоэл Грас"
full_name("Джоэл")          # "Джоэл как-то там"
full_name(last="Грас")      # "некто Грас"
```

В дальнейшем функции будут использоваться очень часто.

Строки

Последовательности символов, или *строковые значения*, с обеих сторон ограничиваются одинарными или двойными кавычками (они должны совпадать):

```
single_quoted_string = 'наука о данных'   # Одинарные
double_quoted_string = "наука о данных"   # Двойные
```

Обратная косая черта используется для кодирования специальных символов. Например:

```
tab_string = "\t"         # Обозначает символ табуляции
len(tab_string)           # равно 1
```

Если требуется непосредственно сама обратная косая черта, которая встречается в именах каталогов в операционной системе Windows или в регулярных выражениях, то при помощи r"" можно создать *сырое строковое значение*:

```
not_tab_string = r"\t"    # Обозначает символы '\' и 't'
len(not_tab_string)       # равно 2
```

Многострочные последовательности символов можно создавать при помощи тройных одинарных (или двойных) кавычек:

```
multi_line_string = """Это первая строка.
это вторая строка
а это третья строка"""
```

[9] Параметр указывается в объявлении функции и описывает значение, которое необходимо передать при ее вызове. Аргумент — это конкретное значение, передаваемое при вызове функции. — *Прим. пер.*

Новым для Python 3.6 функционалом является форматированное строковое значение, или f-строка, которая обеспечивает простой способ подстановки значений в последовательности символов. Например, если имя и фамилия заданы по отдельности:

```
first_name = "Джоэл"
last_name = "Грас"
```

то мы бы, возможно, захотели объединить их в полное имя. Для конструирования строкового значения `full_name` с полным именем существует несколько способов:

```
full_name1 = first_name + " " + last_name   # Сложение строковых значений
full_name2 = "{0} {1}".format(first_name, last_name) # string.format
```

Но решение с f-строкой является гораздо менее громоздким:

```
full_name3 = f"{first_name} {last_name}"
```

и в последующих главах мы отдадим ему предпочтение.

Исключения

Когда что-то идет не так, Python вызывает *исключение*. Необработанные исключения приводят к непредвиденному останову программы. Исключения обрабатываются при помощи инструкций **`try`** и **`except`**:

```
try:
    print(0 / 0)
except ZeroDivisionError:
    print("Нельзя делить на ноль")
```

Хотя во многих языках программирования использование исключений считается плохим стилем программирования, в Python нет ничего страшного, если они применяются с целью сделать код чище, и мы будем иногда поступать именно так.

Списки

Наверное, наиважнейшей структурой данных в Python является *список*. Это просто упорядоченная совокупность (или коллекция), похожая на массив в других языках программирования, но с дополнительным функционалом.

```
integer_list = [1, 2, 3]                        # Список целых чисел
heterogeneous_list = ["строка", 0.1, True]      # Разнородный список
list_of_lists = [ integer_list, heterogeneous_list, [] ]  # Список списков

list_length = len(integer_list)  # Длина списка равна 3
list_sum    = sum(integer_list)  # Сумма значений в списке равна 6
```

Вы можете устанавливать значение и получать доступ к *n*-му элементу списка с помощью квадратных скобок:

```
x = list(range(10))  # Задает список [0, 1, ..., 9]
zero = x[0]    # равно 0, т. е. индекс 1-го элемента равен 0
one = x[1]     # равно 1
nine = x[-1]   # равно 9, по-Python'овски взять последний элемент
eight = x[-2]  # равно 8, по-Python'овски взять предпоследний элемент
x[0] = -1      # Теперь x равен [-1, 1, 2, 3, ..., 9]
```

Вы также можете применять квадратные скобки для "нарезки" списков. Срез `i:j` означает все элементы от `i` (включительно) до `j` (не включительно). Если опустить начало среза, то список будет нарезан с самого начала, а если опустить конец среза, то список будет нарезан до самого конца:

```
first_three  = x[:3]   # Первые три равны [-1, 1, 2]
three_to_end = x[3:]   # С третьего до конца равно [3, 4, ..., 9]
one_to_four  = x[1:5]  # С первого по четвертый равно [1, 2, 3, 4]
last_three   = x[-3:]  # Последние три равны [7, 8, 9]
without_first_and_last = x[1:-1]  # Без первого и последнего
                                  # равно [1, 2, ..., 8]
copy_of_x = x[:]       # Копия списка x равна [-1, 1, 2, ..., 9]
```

Квадратные скобки можно также использовать для нарезки строковых значений и других последовательно организованных типов.

Срез может принимать третий аргумент, обозначающий *сдвиг*, который может быть отрицательным:

```
every_third = x[::3]       # [-1, 3, 6, 9]
five_to_three = x[5:2:-1]  # [5, 4, 3]
```

В Python имеется оператор **in**, который проверяет принадлежность элемента списку:

```
1 in [1, 2, 3]    # True
0 in [1, 2, 3]    # False
```

Проверка заключается в поочередном просмотре всех элементов, поэтому пользоваться им стоит только тогда, когда точно известно, что список является небольшим или неважно, сколько времени уйдет на проверку.

Списки легко конкатенировать, т. е. сцеплять друг с другом. Если требуется модифицировать список прямо на месте, то используется метод `extend`, который добавляет элементы из другой коллекции:

```
x = [1, 2, 3]
x.extend([4, 5, 6])  # Теперь x равен [1, 2, 3, 4, 5, 6]
```

Если вы не хотите модифицировать список `x`, то можно воспользоваться сложением списков:

```
x = [1, 2, 3]
y = x + [4, 5, 6]    # y равен [1, 2, 3, 4, 5, 6]; x остался без изменений
```

Обычно к спискам добавляют по одному элементу за одну операцию:

```
x = [1, 2, 3]
x.append(0)          # Теперь x равен [1, 2, 3, 0]
y = x[-1]            # равно 0
z = len(x)           # равно 4
```

Нередко бывает удобно *распаковывать* списки, когда известно, сколько элементов в них содержится:

```
x, y = [1, 2]        # Теперь x равен 1, y равен 2
```

Однако, если с обеих сторон выражения число элементов неодинаково, то будет выдано сообщение об ошибке значения `ValueError`.

Для отбрасываемого значения обычно используется символ подчеркивания:

```
_, y = [1, 2]        # Теперь y равен 2, а первый элемент не нужен
```

Кортежи

Кортежи — это немутируемые двоюродные братья списков. Практически все, что можно делать со списком, не модифицируя его, можно делать и с кортежем. Вместо квадратных скобок кортеж оформляют круглыми скобками, или вообще обходятся без них:

```
my_list = [1, 2]    # Задать список
my_tuple = (1, 2)   # Задать кортеж
other_tuple = 3, 4  # Еще один кортеж
my_list[1] = 3      # Теперь список my_list равен [1, 3]

try:
    my_tuple[1] = 3
except TypeError:
    print("Кортеж нельзя модифицировать")
```

Кортежи обеспечивают удобный способ для возврата нескольких значений из функций:

```
# Функция возвращает сумму и произведение двух параметров
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3)      # равно (5, 6)
s, p = sum_and_product(5, 10)   # s равно 15, p равно 50
```

Кортежи (и списки) также используются во *множественном присваивании*:

```
x, y = 1, 2   # Теперь x равен 1, y равен 2
x, y = y, x   # Обмен значениями переменных по-Python'овски;
              # теперь x равен 2, y равен 1
```

Словари

Словарь или ассоциативный список — это еще одна фундаментальная структура данных. В нем *значения* ассоциированы с *ключами*, что позволяет быстро извлекать значение, соответствующее конкретному ключу:

```
empty_dict = {}                        # Задать словарь по-Python'овски
empty_dict2 = dict()                   # Не совсем по-Python'овски
grades = {"Joel" : 80, "Tim" : 95}     # Литерал словаря (оценки за экзамены)
```

Доступ к значению по ключу можно получить при помощи квадратных скобок:

```
joels_grade = grades["Joel"]           # равно 80
```

При попытке запросить значение, которое в словаре отсутствует, будет выдано сообщение об ошибке ключа `KeyError`:

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print("Оценки для Кэйт отсутствуют!")
```

Проверить наличие ключа можно при помощи оператора `in`:

```
joel_has_grade = "Joel" in grades      # True
kate_has_grade = "Kate" in grades      # False
```

Словари имеют метод `get`, который при поиске отсутствующего ключа вместо вызова исключения возвращает значение по умолчанию:

```
joels_grade = grades.get("Joel", 0)    # равно 80
kates_grade = grades.get("Kate", 0)    # равно 0
no_ones_grade = grades.get("Никто")    # Значение по умолчанию равно None
```

Присваивание значения по ключу выполняется при помощи тех же квадратных скобок:

```
grades["Tim"] = 99                     # Заменяет старое значение
grades["Kate"] = 100                   # Добавляет третью запись
num_students = len(grades)             # равно 3
```

Словари часто используются в качестве простого способа представить структурные данные:

```
tweet = {
  "user" : "joelgrus",
  "text" : "Наука о данных - потрясающая тема",
  "retweet_count" : 100,
  "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

хотя вскоре мы увидим подход получше.

Помимо поиска отдельных ключей можно обратиться ко всем сразу:

```
tweet_keys   = tweet.keys()    # Итерируемый объект для ключей
tweet_values = tweet.values()  # Итерируемый объект для значений
tweet_items  = tweet.items()   # Итерируемый объект для кортежей
                               # (ключ, значение)

"user" in tweet_keys         # Возвращает True, но не по-Python'овски
"user" in tweet              # Python'овский способ проверки ключа,
                             # используя быстрое in
"joelgrus" in tweet_values   # True (медленно, но единственный способ проверки)
```

Ключи словаря должны быть хешируемыми; в частности, в качестве ключей нельзя использовать списки. Если нужен составной ключ, то лучше воспользоваться кортежем или же найти способ преобразования ключа в строковое значение.

Словарь *defaultdict*

Представьте, что вы пытаетесь подсчитать количества появлений слов в документе. Очевидным решением задачи является создание словаря, в котором ключи — это слова, а значения — количества появлений слов. Во время проверки каждого слова количество появлений слов увеличивается, если текущее слово уже есть в словаре, и добавляется в словарь, если оно отсутствует:

```
# Частотности слов
word_counts = {}
document = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

Кроме этого, можно воспользоваться приемом под названием "лучше просить прощения, чем разрешения" и перехватывать ошибку при попытке обратиться к отсутствующему ключу:

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

Третий подход — использовать метод `get`, который изящно выходит из ситуации с отсутствующими ключами:

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

Все перечисленные приемы немного громоздкие, и по этой причине целесообразно использовать словарь `defaultdict`, т. е. словарь со значением по умолчанию. Он похож на обычный словарь за исключением одной особенности — при попытке обратиться к ключу, которого в нем нет, он сперва добавляет для него значение, используя функцию без аргументов, которая предоставляется при его создании. Для того чтобы воспользоваться словарями `defaultdict`, их необходимо импортировать из модуля `collections`:

```
from collections import defaultdict

word_counts = defaultdict(int)       # int() возвращает 0
for word in document:
    word_counts[word] += 1
```

Кроме того, использование словарей `defaultdict` имеет практическую пользу во время работы со списками, словарями и даже со собственными функциями:

```
dd_list = defaultdict(list)          # list() возвращает пустой список
dd_list[2].append(1)                 # Теперь dd_list содержит {2: [1]}

dd_dict = defaultdict(dict)          # dict() возвращает пустой словарь dict
dd_dict["Джоел"]["город"] = "Сиэтл"  # { "Джоел" : { "город" : Сиэтл"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                    # Теперь dd_pair содержит {2: [0,1]}
```

Эти возможности понадобятся, когда словари будут использоваться для "сбора" результатов по некоторому ключу и когда необходимо избежать повторяющихся проверок на присутствие ключа в словаре.

Счетчики

Словарь-счетчик `Counter` трансформирует последовательность значений в объект, похожий на словарь `defaultdict(int)`, где ключам поставлены в соответствие количества появлений. Он в основном будет применяться при создании гистограмм:

```
from collections import Counter
c = Counter([0, 1, 2, 0])  # В результате с равно { 0 : 2, 1 : 1, 2 : 1 }
```

Его функционал позволяет достаточно легко решать задачу подсчета количества появлений слов:

```
# Лучший вариант подсчета количества появлений слов
word_counts = Counter(document)
```

Словарь `Counter` располагает методом `most_common`, который нередко бывает полезен:

```
# Напечатать 10 наиболее встречаемых слов и их количество появлений
for word, count in word_counts.most_common(10):
    print(word, count)
```

Множества

Еще одна полезная структура данных — это множество `set`, которая представляет собой совокупность неупорядоченных элементов *без повторов*:

```
s = set()      # Задать пустое множество
s.add(1)       # Теперь s равно { 1 }
s.add(2)       # Теперь s равно { 1, 2 }
s.add(2)       # s как и прежде равно { 1, 2 }
x = len(s)     # равно 2
y = 2 in s     # равно True
z = 3 in s     # равно False
```

Множества будут использоваться по двум причинам. Во-первых, операция **`in`** на множествах очень быстрая. Если необходимо проверить большую совокупность элементов на принадлежность некоторой последовательности, то множество `set` подходит для этого лучше, чем список:

```
# Список стоп-слов
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "you"]
"zip" in stopwords_list    # False, но проверяется каждый элемент

# Множество стоп-слов
stopwords_set = set(stopwords_list)
"zip" in stopwords_set     # Очень быстрая проверка
```

Вторая причина — получение *уникальных* элементов в наборе данных:

```
item_list = [1, 2, 3, 1, 2, 3]          # Список
num_items = len(item_list)              # равно 6
item_set = set(item_list)               # Множество {1, 2, 3}
num_distinct_items = len(item_set)      # равно 3
distinct_item_list = list(item_set)     # Список [1, 2, 3]
```

Множества будут применяться намного реже словарей и списков.

Поток управления

Как и в большинстве других языков программирования, действия можно выполнять по условию, применяя инструкцию **`if`**:

```
if 1 > 2:
    message = "если 1 была бы больше 2..."
elif 1 > 3:
    message = "elif означает 'else if'"
else:
    message = "когда все предыдущие условия не выполняются, используется else"
```

Кроме того, можно воспользоваться однострочной *трехместной* инструкцией **`if`**-then-**`else`**, которая будет иногда применяться в дальнейшем:

```
parity = "четное" if x % 2 == 0 else "нечетное"
```

В Python имеется цикл `while`:

```
x = 0
while x < 10:
    print(x, "меньше 10")
    x += 1
```

Однако чаще будет использоваться цикл **`for`** совместно с оператором **`in`**:

```
for x in range(10):
    print(x, "меньше 10")
```

Если требуется более сложная логика управления циклом, то можно воспользоваться инструкциями `continue` и `break`:

```
for x in range(10):
    if x == 3:
        continue  # Сразу перейти к следующей итерации
    if x == 5:
        break     # Выйти из цикла
    print(x)
```

В результате будет напечатано `0`, `1`, `2` и `4`.

Истинность

Булевы, или логические, переменные в Python работают так же, как и в большинстве других языков программирования, лишь с одним исключением — они пишутся с заглавной буквы:

```
one_is_less_than_two = 1 < 2          # равно True
true_equals_false = True == False     # равно False
```

Для обозначения несуществующего значения применяется специальный объект `None`, который соответствует значению `null` в других языках:

```
x = None

assert x == None, "не Python'овский способ проверки наличия None"
assert x is None, "Python'овский способ проверки наличия None"
```

В Python может использоваться любое значение там, где ожидается логический тип `Boolean`. Все следующие элементы имеют логическое значение `False`:

- `False`;
- `None`;
- `[]` (пустой список);
- `{}` (пустой словарь);
- `""`;
- `set()` (множество);

- 0;
- 0.0.

Практически все остальное рассматривается как `True`. Это позволяет легко применять инструкции `if` для проверок на наличие пустых списков, пустых строковых значений, пустых словарей и т. д. Иногда, правда, это приводит к труднораспознаваемым дефектам, если не учитывать следующее:

```
s = some_function_that_returns_a_string()  # Возвращает строковое значение
if s:
    first_char = s[0]      # Первый символ в строковом значении
else:
    first_char = ""
```

Вот более простой способ сделать то же самое:

```
first_char = s and s[0]
```

т. к. логический оператор `and` возвращает второе значение, в случае если первое является истинным, и первое значение, в случае если оно является ложным. Аналогичным образом, если `x` в следующем ниже выражении является либо числом, либо, возможно, `None`, то результат так или иначе будет числом:

```
safe_x = x or 0      # Безопасный способ
```

Встроенная функция `all` языка Python берет список и возвращает `True` только тогда, когда каждый элемент списка истинен, а встроенная функция `any` возвращает `True`, когда истинен хотя бы один элемент:

```
all([True, 1, { 3 }])    # True
all([True, 1, {}])       # False, {} является ложным
any([True, 1, {}])       # True, True является истинным
all([])      # True, ложные элементы в списке отсутствуют
any([])      # False, истинные элементы в списке отсутствуют
```

Сортировка

Каждый список в Python располагает методом `sort()`, который упорядочивает его прямо на месте, т. е. внутри списка без выделения дополнительной памяти. Для того чтобы не загрязнять список, можно применить встроенную функцию `sorted`, которая возвращает новый список:

```
x = [4,1,2,3]
y = sorted(x)     # y равен [1, 2, 3, 4], x остался без изменений
x.sort()          # Теперь x равен [1, 2, 3, 4]
```

По умолчанию метод `sort()` и функция `sorted` сортируют список по возрастающему значению элементов, сравнивая элементы друг с другом.

Если необходимо, чтобы элементы были отсортированы по убывающему значению, надо указать аргумент направления сортировки `reverse=True`. А вместо сравнения

самих элементов можно сравнивать результаты функции, которую надо указать вместе с ключом `key`:

```
# Сортировать список по абсолютному значению в убывающем порядке
x = sorted([-4,1,-2,3], key=abs, reverse=True)    # равно [-4,3,-2,1]

# Сортировать слова и их количества появлений
# по убывающему значению количеств
wc = sorted(word_counts.items(),
            key=lambda word_and_count: word_and_count[1],
            reverse=True)
```

Включения в список

Нередко появляется необходимость преобразовать некий список в другой, выбирая только определенные элементы, или внося по ходу изменения в элементы исходного списка, или выполняя и то и другое одновременно. Python'овским решением является применение операции *включения в список*[10]:

```
# Четные числа
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4]
# Квадраты чисел
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]
# Квадраты четных чисел
even_squares = [x * x for x in even_numbers]        # [0, 4, 16]
```

Списки можно преобразовать в словари `dict` или множества `set` аналогичным образом:

```
# Словарь с квадратами чисел
square_dict = { x : x * x for x in range(5) } # {0:0, 1:1, 2:4, 3:9, 4:16}
# Множество с квадратами чисел
square_set  = { x * x for x in [1, -1] }      # { 1 }
```

Если значение в списке не нужно, то обычно в качестве переменной используется символ подчеркивания:

```
zeroes = [0 for _ in even_numbers] # Имеет ту же длину, что и even_numbers
```

Операция включения в список может содержать несколько инструкций `for`:

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]    # 100 пар (0,0) (0,1)... (9,8), (9,9)
```

[10] Операция включения (comprehension) аналогична математической записи для задания множества путем описания свойств, которыми должны обладать его члены, и представляет собой изящный способ преобразования одной последовательности в другую. Во время этого процесса элементы могут быть условно включены и преобразованы заданной функцией. В Python поддерживается включение в список, в словарь и в множество. — *Прим. пер.*

И последующие инструкции **for** могут использовать результаты предыдущих:

```
# Пары с возрастающим значением
increasing_pairs = [(x, y)                       # Только пары с x < y,
                    for x in range(10)           # range(мин, макс) равен
                    for y in range(x + 1, 10)]   # [мин, мин+1, ..., макс-1]
```

Включения в список будут применяться очень часто.

Автоматическое тестирование и инструкция *assert*

Как исследователи данных, мы будем создавать много кода. Как убедиться в том, что наш код верен? Один из способов — использовать типы (мы коснемся их вкратце), еще один способ — применять автоматизированные тесты.

Существуют сложные вычислительные каркасы, предназначенные для написания и запуска тестов, но в этой книге мы ограничимся использованием инструкций **assert**, которые заставят ваш код вызвать ошибку подтверждения `AssertionError`, если указанное вами условие не является истинным:

```
assert 1 + 1 == 2
assert 1 + 1 == 2, "1 + 1 должно равняться 2, но здесь это не так"
```

Как видно во втором случае, при необходимости можно добавлять сообщение для вывода на экран, если подтверждение правильности окажется неуспешным.

Подтвердить, что $1+1=2$, не вызывает особого интереса. Интереснее подтверждать, что функции, которые вы пишете, делают то, что вы от них ожидаете:

```
def smallest_item(xs):
    return min(xs)

assert smallest_item([10, 20, 5, 40]) == 5
assert smallest_item([1, 0, -1, 2]) == -1
```

На протяжении всей книги именно так мы и будем использовать инструкцию **assert**. Это хорошее практическое решение, и я настоятельно рекомендую вам широко применять его в собственном коде. (Если вы посмотрите на код книги в репозитории GitHub, то увидите, что он содержит гораздо больше операций подтверждения правильности, чем напечатано в книге. Это вселяет в меня уверенность, что код, который я написал для вас, является корректным.)

Еще одно менее распространенное применение — выполнять подтверждение правильности входов в функции:

```
def smallest_item(xs):
    assert xs, "пустой список не имеет наименьшего элемента"
    return min(xs)
```

Время от времени мы будем это делать, но чаще мы будем использовать инструкцию **assert** для подтерждения правильности нашего кода.

Объектно-ориентированное программирование

Как и во многих языках программирования, в Python имеется возможность определять *классы*, которые содержат данные и функции для работы с ними. Мы иногда будем ими пользоваться, делая код чище и проще. Наверное, самый простой способ объяснить, как они работают, — привести пример с подробным аннотированием.

Здесь мы построим класс, который описывает работу "счетчика нажатий", используемого в дверях для отслеживания количества людей, приходящих на собрания профессионалов "Продвинутые темы в науке о данных".

Он поддерживает счетчик, может быть нажат, увеличивая количество, позволяет читать количество посредством метода `read_count` и может обнуляться. (В реальной жизни один из них циклически сбрасывается с 9999 до 0000, но мы не будем этим утруждаться.)

Для определения класса используется ключевое слово `class` и ГорбатоеИмя:

```
class CountingClicker:
    """
    Класс может/должен иметь документирующее
    строковое значение docstring, как функцию
    """
```

Класс содержит ноль или более функций-членов, или компонентных функций. По соглашению каждый член принимает первый параметр `self`, который ссылается на конкретный экземпляр класса.

Как правило, у класса есть конструктор под названием `__init__`. Он принимает любые параметры, необходимые для конструирования экземпляра вашего класса, и выполняет все настроечные работы, которые вам требуются:

```
    def __init__(self, count = 0):
        self.count = count
```

Хотя конструктор имеет замысловатое имя, мы создаем экземпляры счетчика нажатий, используя только имя класса:

```
clicker1 = CountingClicker()           # Инициализируется нулем
clicker2 = CountingClicker(100)        # Стартует со счетчиком, равным 100
clicker3 = CountingClicker(count=100)  # Более явный способ сделать то же самое
```

Обратите внимание, что имя метода `__init__` начинается и заканчивается двойным подчеркиванием. Такие "магические" методы иногда называются "дандерными" методами от англ. *doubleUNDERscore* и реализуют "особые" линии поведения.

Классовые методы, имена которых начинаются с подчеркивания, по соглашению считаются "приватными", и пользователи класса не должны напрямую вызывать их. Однако Python не *остановит* пользователей от их вызова.

Еще одним таким методом является метод `__repr__`, который создает строковое представление экземпляра класса:

```
    def __repr__(self):
        return f"CountingClicker(count={self.count})"
```

И наконец, нам нужно имплементировать публичный API нашего класса:

```
    def click(self, num_times = 1):
        """Кликнуть на кликере несколько раз."""
        self.count += num_times

    def read(self):
        return self.count

    def reset(self):
        self.count = 0
```

Определив его, давайте применим инструкцию **assert** для написания нескольких тестовых случаев для нашего счетчика нажатий:

```
clicker = CountingClicker()
assert clicker.read() == 0, "счетчик должен начинаться со значения 0"
clicker.click()
clicker.click()
assert clicker.read() == 2, "после двух нажатий счетчик должен иметь значение 2"
clicker.reset()
assert clicker.read() == 0, "после сброса счетчик должен вернуться к 0"
```

Написание подобных тестов помогает нам быть уверенными в том, что наш код работает так, как он задумывался, и что он остается таким всякий раз, когда мы вносим в него изменения.

Мы также иногда создаем *подклассы*, которые наследуют некоторый свой функционал от родительского класса. Например, мы могли бы создать счетчик нажатий без сброса, используя класс `CountingClicker` в качестве базового класса и переопределив метод `reset`, который ничего не делает:

```
# Подкласс наследует все поведение от своего родительского класса.
class NoResetClicker(CountingClicker):
    # Этот класс имеет все те же самые методы, что и у CountingClicker

    # За исключением того, что его метод сброса reset ничего не делает.
    def reset(self):
        pass

clicker2 = NoResetClicker()
assert clicker2.read() == 0
clicker2.click()
assert clicker2.read() == 1
clicker2.reset()
assert clicker2.read() == 1, "функция reset не должна ничего делать"
```

Итерируемые объекты и генераторы

Прелесть списка в том, что из него можно извлекать конкретные элементы по их индексам. Но это не всегда требуется. Список из миллиарда чисел занимает много памяти. Если вам нужны элементы по одному за раз, то нет никакой причины держать их все под рукой. Если же в итоге вам требуются лишь первые несколько элементов, то генерирование всего миллиарда будет изрядной тратой ресурсов впустую.

Часто нам необходимо перебрать всю коллекцию, используя лишь инструкцию **for** или оператор **in**. В таких случаях мы можем создавать *генераторы*, которые можно итеративно перебирать точно так же, как списки, но при этом лениво генерировать их значения по требованию.

Один из способов создания генераторов заключается в использовании функций и инструкции `yield`:

```
def generate_range(n):
    i = 0
    while i < n:
        yield i    # Каждый вызов инструкции yield
        i += 1     # производит значение генератора
```

Приведенный ниже цикл будет потреблять предоставляемые инструкцией `yield` значения по одному за раз до тех пор, пока элементы не закончатся:

```
for i in generate_range(10):
    print(f"i: {i}")
```

(На самом деле функция `_range` сама является "ленивой", поэтому делать это нет смысла.)

С помощью генератора можно создать бесконечную последовательность:

```
# Натуральные числа
def natural_numbers():
    """Возвращает 1, 2, 3, ..."""
    n = 1
    while True:
        yield n
        n += 1
```

хотя, наверное, не стоит перебирать такую последовательность без применения какой-нибудь логики выхода из цикла.

Обратной стороной ленивого вычисления является то, что итеративный обход генератора можно выполнить всего один раз. Если вам нужно перебирать что-то многократно, то следует либо каждый раз создавать генератор заново, либо использовать список. Если генерирование значений является дорогостоящим, то это будет хорошей причиной вместо него использовать список.

Второй способ создания генераторов — использовать операции включения с инструкцией **for**, обернутые в круглые скобки:

```
evens_below_20 = (i for i in generate_range(20) if i % 2 == 0)
```

При таком включении генератор не выполняет никакой работы до тех пор, пока вы не выполните его обход (с использованием инструкции **for** или `next`). Его можно применять для наращивания изощренных конвейеров по обработки данных:

```
# Ни одно из этих вычислений *не делает* ничего
# до тех пор, пока мы не выполним обход
data = natural_numbers()
evens = (x for x in data if x % 2 == 0)
even_squares = (x ** 2 for x in evens)
even_squares_ending_in_six = (x for x in even_squares if x % 10 == 6)
# и т. д.
```

Нередко, когда мы перебираем список или генератор, нам нужны не только значения, но и их индексы. Для этого общего случая Python предоставляет функцию `enumerate`, которая превращает значения в пары `(индекс, значение)`:

```
names = ["Alice", "Bob", "Charlie", "Debbie"]

# Не по-Python'овски
for i in range(len(names)):
    print(f"name {i} is {names[i]}")

# Тоже не по-Python'овски
i = 0
for name in names:
    print(f"name {i} is {names[i]}")
    i += 1

# По-Python'овски
for i, name in enumerate(names):
    print(f"name {i} is {name}")
```

Мы будем использовать эту функцию очень много.

Случайность

По мере изучения науки о данных перед нами часто возникает необходимость генерировать случайные числа. Для этого предназначен модуль `random`:

```
import random
random.seed(10)     # Этим обеспечивается, что всякий раз
                    # мы получаем одинаковые результаты

# Четыре равномерные случайные величины
four_uniform_randoms = [random.random() for _ in range(4)]
```

```
# [0.5714025946899135, # Функция random.random() производит числа
# 0.4288890546751146,  # равномерно в интервале между 0 и 1
# 0.5780913011344704,  # Указанная функция будет применяться
# 0.20609823213950174] # наиболее часто.
```

Модуль `random` на самом деле генерирует *псевдо*случайные (т. е. детерминированные) числа на основе внутреннего состояния. Когда требуется получать воспроизводимые результаты, внутреннее состояние можно задавать при помощи посева начального значения для генератора псевдослучайных чисел `random.seed`:

```
random.seed(10)          # Задать начальное число для генератора
                         # случайных чисел равным 10
print(random.random())   # Получаем 0.57140259469
random.seed(10)          # Переустановить начальное значение, задав 10
print(random.random())   # Снова получаем 0.57140259469
```

Иногда будет применяться метод `random.randrange`, который принимает один или два аргумента и возвращает элемент, выбранный случайно из соответствующего интервала `range`:

```
random.randrange(10)     # Случайно выбрать из range(10) = [0, 1, ..., 9]
random.randrange(3, 6)   # Случайно выбрать из range(3, 6) = [3, 4, 5]
```

Есть еще ряд методов, которые пригодятся. Метод `random.shuffle`, например, перетасовывает элементы в списке в случайном порядке:

```
up_to_ten = range(10)    # Задать последовательность из 10 элементов
random.shuffle(up_to_ten)
print(up_to_ten)
# [2, 5, 1, 9, 7, 3, 8, 6, 4, 0] (фактические результаты могут отличаться)
```

Если вам нужно случайно выбрать один элемент из списка, то можно воспользоваться методом `random.choice`:

```
# Мой лучший друг
my_best_friend = random.choice(["Alice", "Bob", "Charlie"]) # у меня "Bob"
```

Если вам нужно произвести случайную выборку элементов *без возврата* их в последовательность (т. е. без повторяющихся элементов), то для этих целей служит метод `random.sample`:

```
# Лотерейные номера
lottery_numbers = range(60)
# Выигрышные номера (пример выборки без возврата)
winning_numbers = random.sample(lottery_numbers, 6)  # [16, 36, 10, 6, 25, 9]
```

Для получения выборки элементов *с возвратом* (т. е. допускающей повторы) вы можете делать многократные вызовы метода `random.choice`:

```
# Список из четырех элементов (пример выборки с возвратом)
four_with_replacement = [random.choice(range(10)) for _ in range(4)]
print(four_with_replacement)  # [9, 4, 4, 2]
```

Регулярные выражения

Регулярные выражения предоставляют средства для выполнения поиска в тексте. Они невероятно полезны и одновременно довольно сложны, причем настолько, что им посвящены целые книги. Мы рассмотрим их подробнее по мере того, как будем их встречать по ходу повествования. Вот несколько примеров их использования на языке Python:

```
import re

re_examples = [                    # Все они - истинные, т. к.
    not re.match("a", "cat"),      # слово 'cat' не начинается с 'a'
    re.search("a", "cat"),         # В слове 'cat' есть 'a'
    not re.search("c", "dog"),     # В слове 'dog' нет 'c'.
    3 == len(re.split("[ab]", "carbs")),   # Разбить по a или b
                                           # на ['c','r','s']
    "R-D-" == re.sub("[0-9]", "-", "R2D2") # Заменить цифры дефисами
]

assert all(re_examples), "все примеры регулярных выражений должны быть истинными"
```

Есть одна важная деталь, которую стоит отметить, — функция `re.match` проверяет, соответствует ли начало строкового значения регулярному выражению, в то время как функция `re.search` проверяет, соответствует ли какая-либо часть строкового значения регулярному выражению. В какой-то момент вы можете их перепутать, и это доставит вам массу неприятностей.

Официальная документация (**https://docs.python.org/3/library/re.html**) дает гораздо более подробную информацию.

Функциональное программирование

В первом издании этой книги были представлены функции `partial`, `map`, `reduce` и `filter` языка Python. На своем пути к просветлению я понял, что этих функций лучше избегать, и их использование в книге было заменено включениями в список, циклами и другими, более Python'овскими конструкциями.

Функция *zip* и распаковка аргументов

Часто возникает необходимость объединить два или более списков в один. Встроенная функция `zip` преобразовывает многочисленные итерируемые объекты в единый итерируемый объект кортежей, состоящий из соответствующих элементов:

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
```

```
# Функция zip является ленивой,
# поэтому следует сделать что-то вроде этого
[pair for pair in zip(list1, list2)]# равно [('a', 1), ('b', 2), ('c', 3)]
```

Если списки имеют разные длины, то функция прекратит работу, как только закончится первый из них. "Разъединить" список можно при помощи замысловатого трюка:

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

Звездочка `*` выполняет *распаковку аргументов*, которая использует элементы пар в качестве индивидуальных аргументов для функции `zip`. Это то же самое, как и вызвать:

```
letters, numbers = zip(('a', 1), ('b', 2), ('c', 3))
```

Результат в обоих случаях будет `[('a','b','c'), (1,2,3)]`.

Распаковка аргументов может применяться с любой функцией:

```
def add(a, b): return a + b

add(1, 2)       # Возвращает 3
try:
    add([1, 2])
except TypeError:
    print("Функция add ожидает два входа")
add(*[1, 2])    # Возвращает 3
```

Этот прием редко окажется для нас полезным, но когда мы его будем применять, то он будет выступать в качестве програмистского трюка.

Переменные *args* и *kwargs*

Предположим, нужно создать функцию высшего порядка, которая в качестве входа принимает некую функцию `f` и возвращает новую функцию, которая для любого входа возвращает удвоенное значение `f`:

```
# Удвоитель
def doubler(f):
    # Здесь мы определяем новую функцию, которая содержит ссылку на f
    def g(x):
        return 2 * f(x)

    # И возвращаем эту новую функцию
    return g
```

В некоторых случаях это работает:

```
def f1(x):
    return x + 1
```

```
g = doubler(f1)
assert g(3) == 8, "(3 + 1) * 2 должно быть равно 8"
assert g(-1) == 0, "(-1 + 1) * 2 должно быть равно 0"
```

Но терпит неудачу с функциями, которые принимают более одного аргумента:

```
def f2(x, y):
    return x + y

g = doubler(f2)
try:
    g(1, 2)
except TypeError:
    print("По определению g принимает только один аргумент")
```

Нам же нужен способ определения функции, которая принимает произвольное число аргументов. Это делается при помощи распаковки аргументов и небольшого волшебства:

```
def magic(*args, **kwargs):
    print("Безымянные аргументы:", args)
    print("Аргументы по ключу:", kwargs)

magic(1, 2, key="word", key2="word2")

# Напечатает
# Безымянные аргументы: (1, 2)
# Аргументы по ключу: {'key2': 'word2', 'key': 'word'}
```

То есть когда функция определена подобным образом, переменная `args` представляет собой кортеж из безымянных позиционных аргументов, а переменная `kwargs` — словарь из именованных аргументов. Она работает и по-другому, если необходимо использовать список (или кортеж) и словарь для *передачи* аргументов в функцию:

```
def other_way_magic(x, y, z):
    return x + y + z

x_y_list = [1, 2]
z_dict = {"z" : 3}
assert other_way_magic(*x_y_list, **z_dict) == 6, "1 + 2 + 3 должно быть равно 6"
```

Эти свойства переменных `args` и `kwargs` позволяют проделывать разного рода замысловатые трюки. Здесь этот прием будет использоваться исключительно для создания функций высшего порядка, которые могут принимать произвольное число входящих аргументов:

```
# Исправленный удвоитель
def doubler_correct(f):
    """Работает независимо от того, какого рода аргументы
       функция f ожидает"""
```

```
    def g(*args, **kwargs):
        """Какими бы ни были аргументы для g, передать их прямиком в f"""
        return 2 * f(*args, **kwargs)
    return g

g = doubler_correct(f2)
print(g(1, 2))    # равно 6
```

Как правило, ваш код будет правильнее и читабельнее, если вы четко указываете, какие аргументы требуются вашим функциям; соответственно, мы будем использовать `args` и `kwargs` только тогда, когда у нас нет другого варианта.

Аннотации типов

Python — это *динамически типизированный* язык. Это означает, что он вообще не заботится о типах объектов, которые мы используем, при условии, что мы применяем их приемлемыми способами:

```
def add(a, b):
    return a + b

assert add(10, 5) == 15, "+ является допустимым для чисел"
assert add([1, 2], [3]) == [1, 2, 3], "+ является допустимым для списков"
assert add("Эй, ", "привет!") == "Эй, привет", "+ является допустимым для строк"

try:
    add(10, "пять")
except TypeError:
    print("Невозможно прибавить целое число к строке")
```

тогда как в *статически типизированном* языке наши функции и объекты имели бы конкретные типы:

```
def add(a: int, b: int) -> int:
    return a + b

add(10, 5)           # Вы хотели бы, чтобы это работало
add("Эй, ", "там")   # Вы хотели бы, чтобы это не работало
```

На самом деле недавние версии Python (вроде) имеют этот функционал. Приведенная выше версия функции `add` с аннотацией типа `int` допустима в Python 3.6!

Однако эти аннотации типов в общем-то ничего не делают. Вы по-прежнему можете использовать аннотированную функцию `add` для добавления строк, и вызов `add(10, "пять")` по-прежнему будет поднимать ту же ошибку `TypeError`.

Тем не менее есть еще (по крайней мере) четыре веские причины использовать аннотации типов в коде Python.

- Типы являются важной формой документирования. Это вдвойне верно в книге, в которой используется исходный код для того, чтобы обучить вас теоретическим и математическим понятиям. Сравните следующие две заглушки функций:

```
def dot_product(x, y): ...

# Мы еще не определили Vector, но представим, что мы это сделали
def dot_product(x: Vector, y: Vector) -> float: ...
```

Я нахожу второй вариант чрезвычайно информативным; и надеюсь, вы тоже. (Я до такой степени привык к подсказкам типов, что теперь мне трудно читать нетипизированный Python.)

- Существуют внешние инструменты (самый популярный из них — библиотека mypy), которые будут читать ваш код, проверять аннотации типов и сообщать вам об ошибках типизации перед исполнением кода. Например, если выполнить mypy с файлом, содержащим `add("Эй", "привет")`, он предупредит вас (ошибка: аргумент 1 в `add` имеет несовместимый тип `str`; ожидается `int`):

```
error: Argument 1 to "add" has incompatible type "str"; expected "int"
```

Как и тестирование с использованием инструкции `assert`, это хороший способ отыскать несоотвествия в коде перед его выполнением. Изложение в книге не будет включать такую проверку типов; однако за кулисами я буду запускать подобную проверку, которая поможет убедиться, что сама книга является правильной.

- Необходимость думать о типах в коде заставляет вас разрабатывать более чистые функции и интерфейсы:

```
from typing import Union

def secretly_ugly_function(value, operation): ...

def ugly_function(value: int,
                  operation: Union[str, int, float, bool]) -> int:
```

Здесь мы имеем функцию, чей параметр `operation` может быть значением с типом `str`, либо `int`, либо `float`, либо `bool`. Весьма вероятно, что эта функция является хрупкой, и ее трудно использовать, но она становится гораздо яснее, когда типы становятся явными. Это заставит нас проектировать менее неуклюже, за что наши пользователи будут нам только благодарны.

- Использование типов дает возможность вашему редактору помогать вам с такими вещами, как автозаполнение (рис. 2.1) и сообщать о своем недовольстве при встрече ошибок.

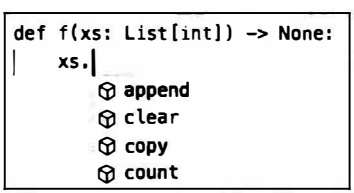


Рис. 2.1. Редактор VSCode, но, вероятно, ваш редактор делает то же самое

Иногда люди настаивают на том, что подсказки типов могут быть ценными для больших проектов, но не стоят времени для малых. Однако, поскольку подсказки типов почти не требуют дополнительного времени на ввод и позволяют редактору экономить ваше время, я утверждаю, что они действительно позволяют писать код быстрее, даже в случае небольших проектов.

По всем этим причинам во всем коде в остальной части книги будут использоваться аннотации типов. Я ожидаю, что некоторые читатели будут обескуражены использованием аннотаций типов; однако я подозреваю, что к концу книги они изменят свое мнение.

Как писать аннотации типов

Как мы видели, для встроенных типов, таких как `int`, `bool` и `float`, вы просто используете сам тип в качестве аннотации. Что, если у вас был (скажем) список?

```
def total(xs: list) -> float:
    return sum(total)
```

Это не является неправильным, но тип недостаточно специфичен. Совершенно очевидно, что мы на самом деле хотим, чтобы `xs` был списком из вещественных `float`, а не (скажем) списком строк.

Модуль **`typing`** предоставляет ряд параметризованных типов, которые мы можем для этого использовать:

```
from typing import List   # Обратите внимание на заглавную L

def total(xs: List[float]) -> float:
    return sum(total)
```

До сих пор мы указывали аннотации только для параметров функций и возвращаемых типов. Что касается самих переменных, то обычно ясно видно, какого они типа:

```
# Вот так аннотируются переменные при их определении.
# Но это не является необходимым; очевидно, что x имеет тип int.
x: int = 5
```

Однако иногда это не так очевидно:

```
values = []          # Какой у меня тип?
best_so_far = None   # Какой у меня тип?
```

В таких случаях мы предоставим внутристрочные (inline) подсказки типа:

```
from typing import Optional

values: List[int] = []
best_so_far: Optional[float] = None  # Допустимо иметь тип float либо None
```

Модуль `typing` содержит много других типов, только некоторые из которых мы когда-либо будем использовать:

```
# Все аннотации типов в этом фрагменте не являются необходимыми
from typing import Dict, Iterable, Tuple

# Ключи являются строками, значения - целыми числами
counts: Dict[str, int] = {'data': 1, 'science': 2}

# Списки и генераторы являются итерируемыми объектами
if lazy:
    evens: Iterable[int] = (x for x in range(10) if x % 2 == 0)
else:
    evens = [0, 2, 4, 6, 8]

# Кортежи конкретизируют тип каждого элемента
triple: Tuple[int, float, int] = (10, 2.3, 5)
```

Наконец, поскольку функции в Python используются как объекты первого класса, нам нужен тип для их представления. Вот довольно надуманный пример:

```
from typing import Callable

# Подсказка типа говорит, что repeater - это функция, которая принимает
# два аргумента с типами str и int и возвращает тип str.
def twice(repeater: Callable[[str, int], str], s: str) -> str:
    return repeater(s, 2)

def comma_repeater(s: str, n: int) -> str:
    n_copies = [s for _ in range(n)]
    return ', '.join(n_copies)

assert twice(comma_repeater, "подсказки типов") == "подсказки типов, подсказки
                                                                    типов"
```

Поскольку аннотации типов — это просто объекты Python, мы можем их назначать переменным для упрощения ссылки на них:

```
Number = int
Numbers = List[Number]

def total(xs: Numbers) -> Number:
    return sum(xs)
```

К концу книги вы уже будете хорошо знакомы с чтением и написанием аннотаций, и я надеюсь, что вы будете использовать их в своем коде.

Добро пожаловать в DataSciencester!

На этом программа ориентации новых сотрудников завершается. Да, и постарайтесь не растерять полученные сведения.

Для дальнейшего изучения

- Дефицит пособий по языку Python в мире отсутствует. Официальное руководство[11] будет неплохим подспорьем для начала.
- Официальное руководство по IPython[12] поможет вам начать работу с IPython, если вы решите использовать его. Очень рекомендуем.
- Документация по библиотеке mypy[13] расскажет вам больше, чем вы когда-либо хотели узнать об аннотациях типов и проверке типов в Python.

[11] См. **https://docs.python.org/3/tutorial/**.

[12] См. **http://ipython.readthedocs.io/en/stable/interactive/index.html**.

[13] См. (**https://mypy.readthedocs.io/en/stable/**.

ГЛАВА 3

Визуализация данных

Думаю, что визуализация — одно из самых мощных орудий достижения личных целей.
– *Харви Маккей*[1]

Фундаментальную часть в арсенале исследователя данных составляет визуализация данных. И если создать визуализацию достаточно просто, то производить *хорошие* визуализации гораздо сложнее.

Визуализация применяется прежде всего для:

- *разведывания* данных[2];
- *коммуницирования* данных.

В этой главе мы сконцентрируемся на развитии навыков, необходимых для того, чтобы вы могли начать разведывание своих собственных данных и создание на их основе визуализаций, которые мы будем использовать на протяжении всей книги. Как и большинство тем, вынесенных в названия глав этой книги, визуализация данных представляет собой обширную область исследования, которая заслуживает отдельной книги. Тем не менее мы постараемся дать вам почувствовать, что́ понимается под хорошей визуализацией, а что нет.

Библиотека matplotlib

Для визуализации данных существует большой набор разных инструментов. Мы будем пользоваться популярной библиотекой matplotlib[3] (которая, впрочем, уже показывает некоторые признаки устаревания). Если вас интересует создание детально проработанных интерактивных визуализаций для Всемирной паутины, то эта библиотека, скорее всего, вам не подойдет. Но с построением элементарных столбчатых и линейных графиков и диаграмм рассеяния она справляется легко.

---

[1] Харви Маккей (род. 1932) — предприниматель, писатель и обозреватель крупнейших американских новостных изданий. — *Прим. пер.*

[2] Разведывание (exploration) — это поиск новых идей или новых стратегий. Используется как антитеза для эксплуатации (exploitation), т. е. использованию существующих идей и стратегий, которые оказались успешными в прошлом. Интеллектуальная работа включает в себя надлежащий баланс между разведыванием, сопряженным с риском пустой траты ресурсов, и эксплуатацией проверенных временем решений. — *Прим. пер.*

[3] См. **http://matplotlib.org/**.

Как уже упоминалось ранее, библиотека matplotlib не является частью стандартной библиотеки Python. Поэтому следует активировать свою виртуальную среду (по поводу ее настройки вернитесь к *разд. "Виртуальные среды" главы 2* и следуйте инструкциям) и установить ее с помощью команды:

```
python -m pip install matplotlib
```

Мы будем пользоваться модулем `matplotlib.pyplot`. В самом простом случае `pyplot` поддерживает внутреннее состояние, в котором вы выстраиваете визуализацию шаг за шагом. Когда она завершена, изображение можно сохранить в файл при помощи метода `savefig` или вывести на экран методом `show`.

Например, достаточно легко создать простой график, такой как на рис. 3.1:

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]              # Годы
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]  # ВВП

# Создать линейную диаграмму: годы по оси x, ВВП по оси y
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# Добавить название диаграммы
plt.title("Номинальный ВВП")

# Добавить подпись к оси y
plt.ylabel("Млрд $")
plt.show()
```

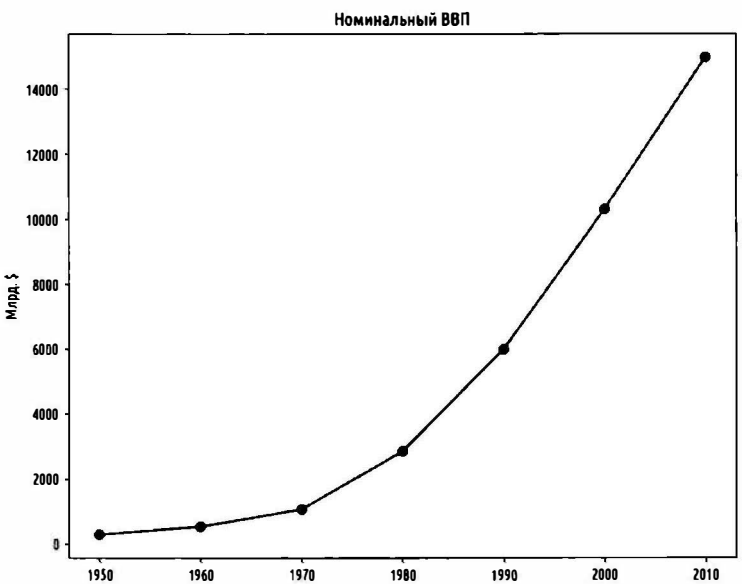


Рис. 3.1. Простой линейный график

Создавать графики типографского качества сложнее, и это выходит за рамки настоящей главы. Вы можете настраивать диаграммы самыми разнообразными способами, например, задавая подписи к осям, типы линий, маркеры точек и т. д. Вместо подробного рассмотрения всех этих вариантов настройки мы просто будем применять некоторые из них (и привлекать внимание к ним) в наших примерах.

Хотя мы не будем пользоваться значительной частью функционала библиотеки matplotlib, следует отметить, что она позволяет создавать вложенные диаграммы, выполнять сложное форматирование и делать интерактивные визуализации. Сверьтесь с документацией[4], если хотите копнуть глубже, чем это представлено в книге.

Столбчатые графики

Столбчатые графики хорошо подходят для тех случаев, когда требуется показать *варьируемость* некоторой величины среди некоего дискретного множества элементов. Например, на рис. 3.2 показано, сколько ежегодных премий "Оскар" Американской киноакадемии было завоевано каждым из перечисленных фильмов:

```
movies = ["Энни Холл", "Бен-Гур", "Касабланка", "Ганди", "Вестсайдская история"]
num_oscars = [5, 11, 3, 8, 10]

# Построить столбцы с левыми X-координатами [xs] и высотами [num_oscars]
plt.bar(range(len(movies)), num_oscars)
```

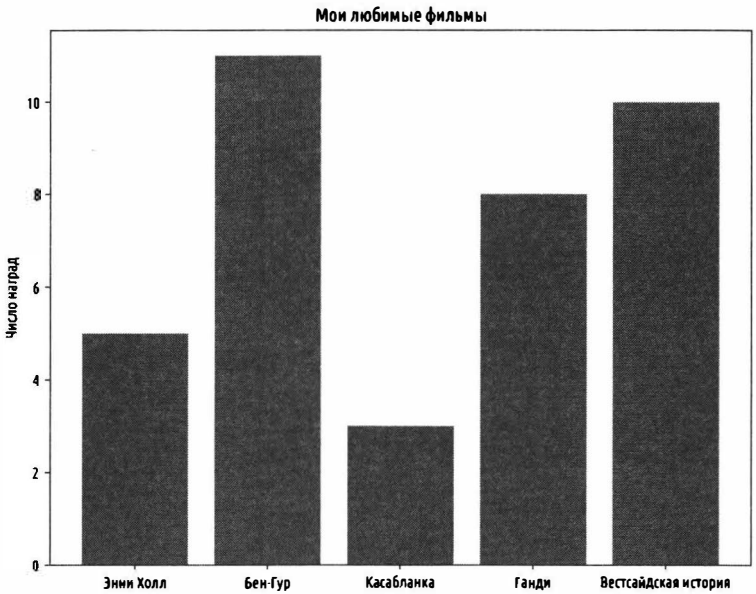


Рис. 3.2. Простой столбчатый график

[4] См. **https://matplotlib.org/users/index.html**.

```
plt.title("Мои любимые фильмы")     # Добавить заголовок
plt.ylabel("Количество наград")     # Разместить метку на оси y

# Пометить ось x названиями фильмов в центре каждого столбца
plt.xticks(range(len(movies)), movies)
plt.show()
```

Столбчатый график также является хорошим вариантом выбора для построения гистограмм сгруппированных числовых значений (рис. 3.3) с целью визуального разведывания характера *распределения* значений:

```
from collections import Counter
grades = [83, 95, 91, 87, 70, 0, 85, 82, 100, 67, 73, 77, 0]

# Сгруппировать оценки подецильно, но
# разместить 100 вместе с отметками 90 и выше
histogram = Counter(min(grade // 10 * 10, 90) for grade in grades)

plt.bar([x + 5 for x in histogram.keys()],  # Сдвинуть столбец влево на 5
        list(histogram.values()),           # Высота столбца
        10)                                 # Ширина каждого столбца 10

plt.axis([-5, 105, 0, 5])                   # Ось x от -5 до 105,
                                            # ось y от 0 до 5

plt.xticks([10 * i for i in range(11)])  # Метки по оси x: 0, 10,..., 100
plt.xlabel("Дециль")
plt.ylabel("Число студентов")
plt.title("Распределение оценок за экзамен № 1")
plt.show()
```

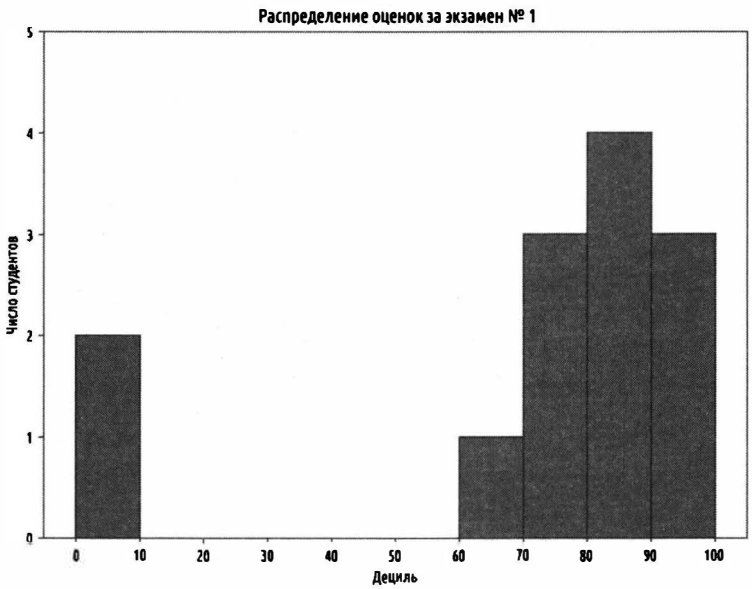


Рис. 3.3. Использование столбчатого графика для гистограммы

Третий аргумент в методе `plt.bar` задает ширину столбцов. Здесь мы выбрали ширину столбцов, равную 10, заполняя весь дециль. Мы также сдвинули столбцы на 5 так, чтобы, например, столбец "10" , который соответствует децилю 10–20, имел свой центр в 15 и, следовательно, занимал правильный диапазон. Мы также добавили в каждый столбец черную обводку, чтобы сделать каждый солбец четче.

Вызов метода `plt.axis` указывает на то, что мы хотим, чтобы значения меток на оси *x* варьировались в диапазоне от –5 до 105 (оставляя немного пространства слева и справа), а на оси *y* — в диапазоне от 0 до 5. Вызов метода `plt.xticks` размещает метки на оси *x* так: 0, 10, 20, ..., 100.

Следует разумно использовать метод `plt.axis`. Во время создания столбчатых графиков форма оси *y*, которая начинается не с нуля, считается плохой, поскольку легко дезориентирует людей (рис. 3.4):

```
mentions = [500, 505]      # Упоминания
years    = [2017, 2018]    # Годы

plt.bar(years, mentions, 0.8)
plt.xticks(years)
plt.ylabel("Число раз, когда я слышал, как упоминали науку о данных")

# Если этого не сделать, matplotlib подпишет ось x как 0, 1
# и добавит +2.013e3 в правом углу (недоработка matplotlib!)
plt.ticklabel_format(useOffset=False)

# Дезориентирующая ось y показывает только то, что выше 500
plt.axis([2016.5, 2018.5, 499, 506])
plt.title("Посмотрите, какой 'огромный' прирост!")
plt.show()
```

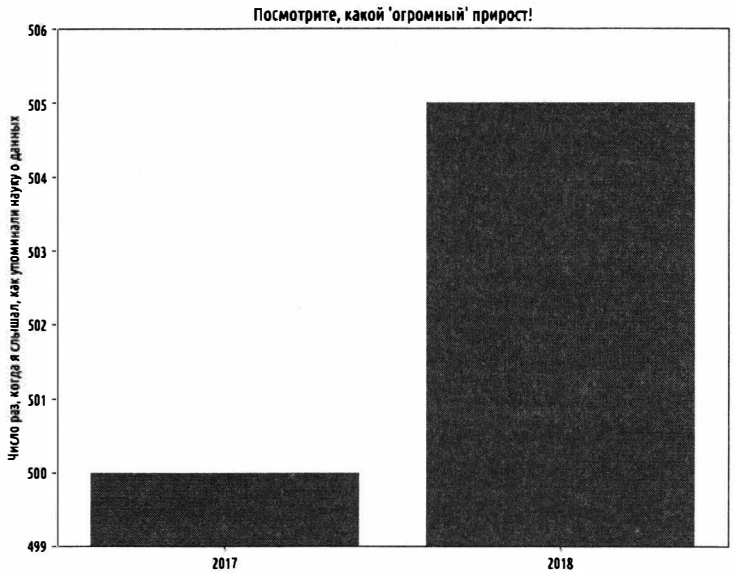


Рис. 3.4. Диаграмма с дезориентирующей осью *y*

На рис. 3.5 использованы более осмысленные оси, и график выглядит уже не таким впечатляющим:

```
plt.axis([2016.5, 2018.5, 499, 506])
plt.title("Теперь больше не такой огромный")
plt.show()
```

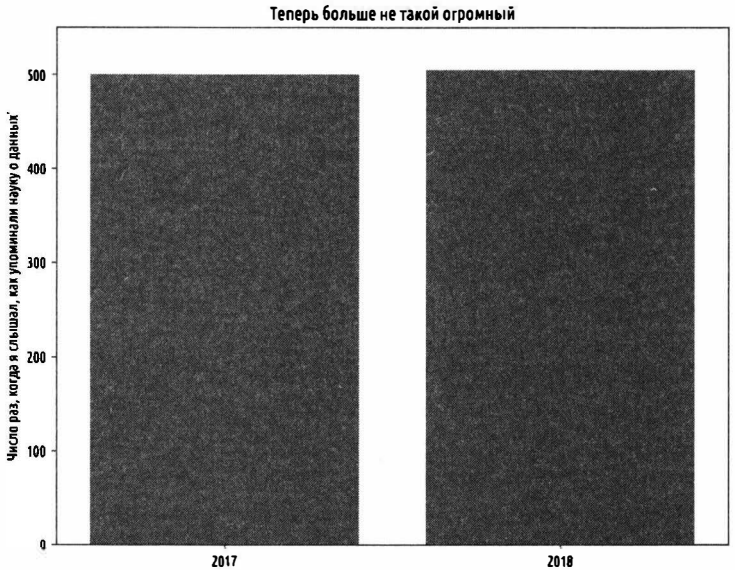


Рис. 3.5. Та же самая диаграмма с правильной осью *y*

Линейные графики

Как мы видели, линейные графики можно создавать при помощи метода `plt.plot`. Они хорошо подходят для изображения *трендов* (рис. 3.6):

```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]      # Дисперсия
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]      # Квадрат смещения
# Суммарная ошибка
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# Метод plt.plot можно вызывать много раз,
# чтобы показать несколько графиков на одной и той же диаграмме:
# зеленая сплошная линия
plt.plot(xs, variance,     'g-',  label='дисперсия')
# красная штрихпунктирная
plt.plot(xs, bias_squared, 'r-.', label='смещение^2')
# синяя пунктирная
plt.plot(xs, total_error,  'b:',  label='суммарная ошибка')
```

```
# Если для каждой линии задано название label,
# то легенда будет показана автоматически,
# loc=9 означает "наверху посередине"
plt.legend(loc=9)
plt.xlabel("Сложность модели")
plt.title("Компромисс между смещением и дисперсией")
plt.show()
```

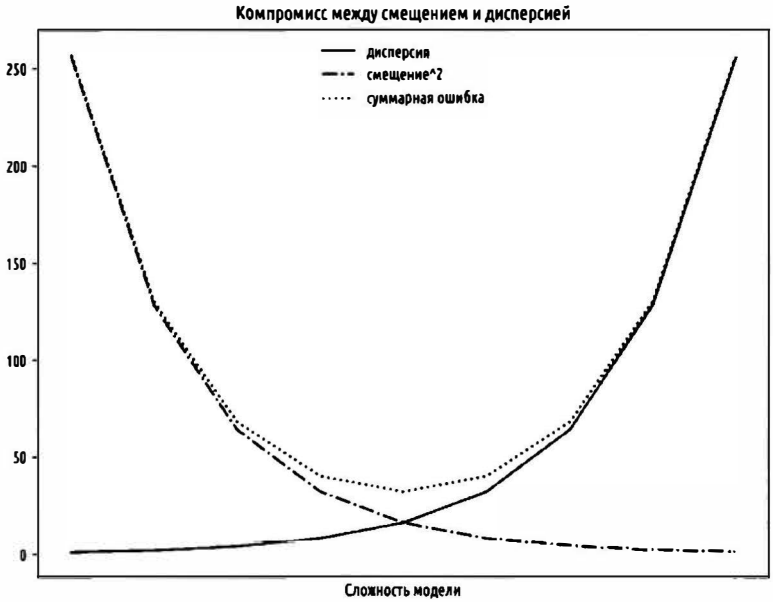


Рис. 3.6. Несколько линейных графиков с легендой

Диаграммы рассеяния

Диаграмма рассеяния лучше всего подходит для визуализации связи между двумя спаренными множествами данных. Например, на рис. 3.7 показана связь между числом друзей пользователя и числом минут, которые они проводят на веб-сайте каждый день:

```
friends = [ 70,  65,  72,  63,  71,  64,  60,  64,  67]  # Друзья
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]  # Минуты
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']   # Метки

plt.scatter(friends, minutes)

# Назначить метку для каждой точки
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
                 xy=(friend_count, minute_count), # Задать метку
                 xytext=(5, -5),                  # и немного сместить ее
                 textcoords='offset points')
```

```
plt.title("Число минут против числа друзей")
plt.xlabel("Число друзей")
plt.ylabel("Число минут, проводимых на сайте ежедневно")
plt.show()
```

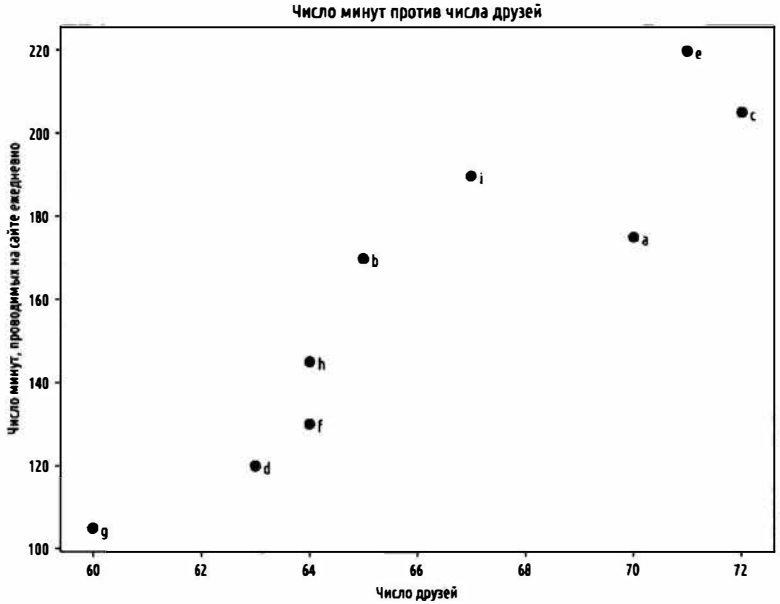


Рис. 3.7. Диаграмма рассеяния друзей и времени, проводимого на веб-сайте

При размещении на диаграмме рассеяния сопоставимых величин можно получить искаженную картину, если масштаб по осям будет определяться библиотекой matplotlib автоматически (рис. 3.8):

```
test_1_grades = [ 99, 90, 85, 97, 80]  # Оценки за экзамен 1
test_2_grades = [100, 85, 60, 90, 70]  # Оценки за экзамен 2

plt.scatter(test_1_grades, test_2_grades)
plt.title("Оси не сопоставимы")
plt.xlabel("Оценки за экзамен 1")
plt.ylabel("Оценки за экзамен 2")
plt.show()
```

Но если добавить вызов метода `plt.axis("equal")`, то диаграмма точнее покажет, что бóльшая часть вариации оценок приходится на экзамен 2 (рис. 3.9).

Этого будет достаточно для того, чтобы приступить к визуализации своих данных. Гораздо больше о визуализации мы узнаем в остальной части книги.

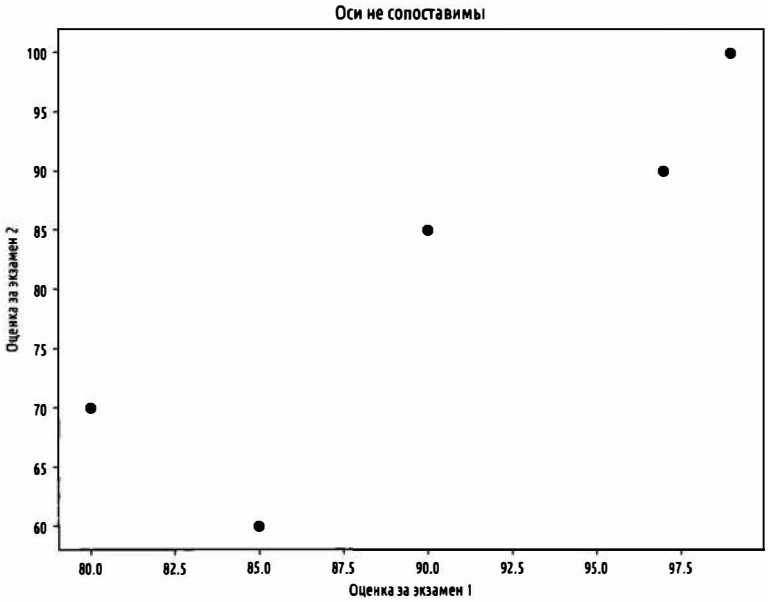


Рис. 3.8. Диаграмма рассеяния с несопоставимыми осями

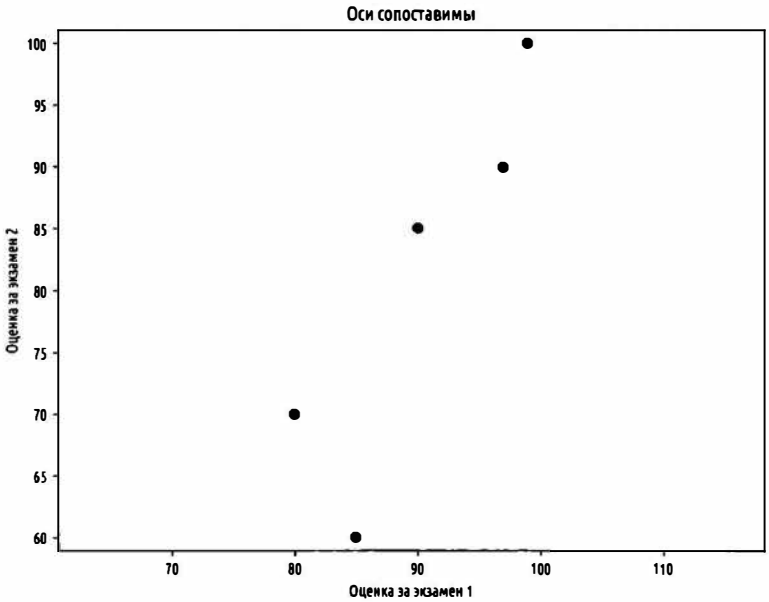


Рис. 3.9. Та же самая диаграмма рассеяния с сопоставимыми осями

Для дальнейшего изучения

- Галерея библиотеки matplotlib[5] даст вам хорошее представление о всем том, что можно делать с указанной библиотекой (и как это делать).
- Библиотека seaborn[6], разработанная поверх matplotlib, позволяет легко создавать более красивые и более сложные визуализации.
- Библиотека Altair[7] является более свежей библиотекой Python для создания декларативных визуализаций.
- Библиотека D3.js[8] на JavaScript пользуется популярностью при создании продвинутых интерактивных визуализаций для Интернета. Несмотря на то что она написана не на Python, она широко используется, и вам непременно стоит познакомиться с ней поближе.
- Библиотека Bokeh[9] позволяет создавать визуализации в стиле D3.js на Python.

[5] См. **https://matplotlib.org/gallery.html**.

[6] См. **https://seaborn.pydata.org/**.

[7] См. **https://altair-viz.github.io/**.

[8] См. **http://d3js.org/**.

[9] См. **http://bokeh.pydata.org/**.

ГЛАВА 4
Линейная алгебра

> Есть ли что-то более бесполезное или менее полезное, чем алгебра?
> – *Билли Коннолли*[1]

Линейная алгебра — это раздел математики, который занимается *векторными пространствами*. В этой короткой главе не стоит задача разобраться с целым разделом алгебры, но так как линейная алгебра лежит в основе большинства понятий и технических решений науки о данных, я буду признателен, если вы хотя бы попытаетесь. Все, чему мы научимся в этой главе, очень пригодится нам в остальной части книги.

Векторы

В абстрактном смысле *векторы* — это объекты, которые можно складывать между собой, формируя новые векторы, и умножать на *скалярные величины* (т. е. на числа), опять же формируя новые векторы.

Конкретно в нашем случае векторы — это точки в некотором конечноразмерном пространстве. Несмотря на то что вы, возможно, не рассматриваете свои данные как векторы, часто ими удобно представлять последовательности чисел.

Например, если у вас есть данные о росте, массе и возрасте большого числа людей, то эти данные можно трактовать как трехмерные векторы `[рост, масса, возраст]`. Если вы готовите группу учащихся к четырем экзаменам, то оценки студентов можно трактовать как четырехмерные векторы `[экзамен1, экзамен2, экзамен3, экзамен4]`.

Если подходить с нуля, то в простейшем случае векторы реализуются в виде списков чисел. Список из трех чисел соответствует вектору в трехмерном пространстве, и наоборот:

```
from typing import List

Vector = List[float]

height_weight_age = [175,    # Сантиметры
                     68,     # Килограммы
                     40 ]    # Годы
```

[1] Билл Коннолли (род. 1942) — шотландский комик, музыкант, ведущий и актер. — *Прим. пер.*

```
grades = [95,   # Экзамен1
          80,   # Экзамен2
          75,   # Экзамен3
          62 ]  # Экзамен4
```

Мы также захотим выполнять *арифметические* операции на векторах. Поскольку списки в Python не являются векторами (и, следовательно, не располагают средствами векторной арифметики), нам придется построить эти арифметические инструменты самим. Поэтому начнем с них.

Нередко приходится *складывать два вектора*. Векторы складывают *покомпонентно*, т. е. если два вектора `v` и `w` имеют одинаковую длину, то их сумма — это вектор, чей первый элемент равен `v[0] + w[0]`, второй — `v[1] + w[1]` и т. д. (Если длины векторов разные, то операция сложения не разрешена.)

Например, сложение векторов `[1, 2]` и `[2, 1]` даст `[1 + 2, 2 + 1]` или `[3, 3]` (рис. 4.1).

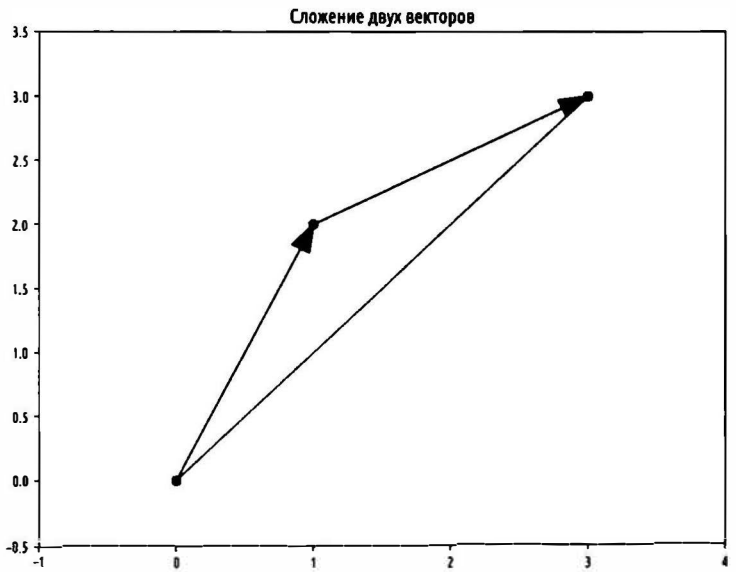


Рис. 4.1. Сложение двух векторов

Это легко можно имплементировать объединением двух векторов посредством функции `zip` и сложением соответствующих элементов с помощью операции включения в список:

```
def add(v: Vector, w: Vector) -> Vector:
    """Складывает соответствующие элементы"""
    assert len(v) == len(w), "векторы должны иметь одинаковую длину"

    return [v_i + w_i for v_i, w_i in zip(v, w)]

assert add([1, 2, 3], [4, 5, 6]) == [5, 7, 9]
```

Схожим образом для получения *разности двух векторов* мы вычитаем соответствующие элементы:

```
def subtract(v: Vector, w: Vector) -> Vector:
    """Вычитает соответствующие элементы"""
    assert len(v) == len(w), "векторы должны иметь одинаковую длину"

    return [v_i - w_i for v_i, w_i in zip(v, w)]

assert subtract([5, 7, 9], [4, 5, 6]) == [1, 2, 3]
```

Нам также иногда может понадобиться *покомпонентная сумма списка векторов*, т. е. нужно создать новый вектор, чей первый элемент равен сумме всех первых элементов, второй элемент — сумме всех вторых элементов и т. д.:

```
def vector_sum(vectors: List[Vector]) -> Vector:
    """Суммирует все соответствующие элементы"""
    # Проверить, что векторы не пустые
    assert vectors, "векторы не предоставлены!"

    # Проверить, что векторы имеют одинаковый размер
    num_elements = len(vectors[0])
    assert all(len(v) == num_elements for v in vectors), "разные размеры!"

    # i-й элемент результата является суммой каждого элемента vector[i]
    return [sum(vector[i] for vector in vectors)
            for i in range(num_elements)]

assert vector_sum([[1, 2], [3, 4], [5, 6], [7, 8]]) == [16, 20]
```

Нам еще понадобится *умножение вектора на скаляр*, которое реализуется простым умножением каждого элемента вектора на это число:

```
def scalar_multiply(c: float, v: Vector) -> Vector:
    """Умножает каждый элемент на c"""
    return [c * v_i for v_i in v]

assert scalar_multiply(2, [1, 2, 3]) == [2, 4, 6]
```

Эта функция позволяет нам вычислять покомпонентные средние значения списка векторов (одинакового размера):

```
def vector_mean(vectors: List[Vector]) -> Vector:
    """Вычисляет поэлементное среднее арифметическое"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))

assert vector_mean([[1, 2], [3, 4], [5, 6]]) == [3, 4]
```

Менее очевидный инструмент — это точечное, или *скалярное произведение* (т. е. производящее скалярный результат). Скалярное произведение двух векторов есть сумма их покомпонентных произведений:

```
def dot(v: Vector, w: Vector) -> float:
    """Вычисляет v_1 * w_1 + ... + v_n * w_n"""
    assert len(v) == len(w), "векторы должны иметь одинаковую длину"

    return sum(v_i * w_i for v_i, w_i in zip(v, w))

assert dot([1, 2, 3], [4, 5, 6]) == 32    # 1 * 4 + 2 * 5 + 3 * 6
```

Если `w` имеет магнитуду, равную 1, то скалярное произведение служит мерой того, насколько далеко вектор `v` простирается в направлении `w`. Например, если `w = [1, 0]`, тогда скалярное произведение `dot(v, w)` — это просто первый компонент вектора `v`. Выражаясь иначе, это длина вектора, которая получится, если спроецировать вектор `v` на вектор `w` (рис. 4.2).

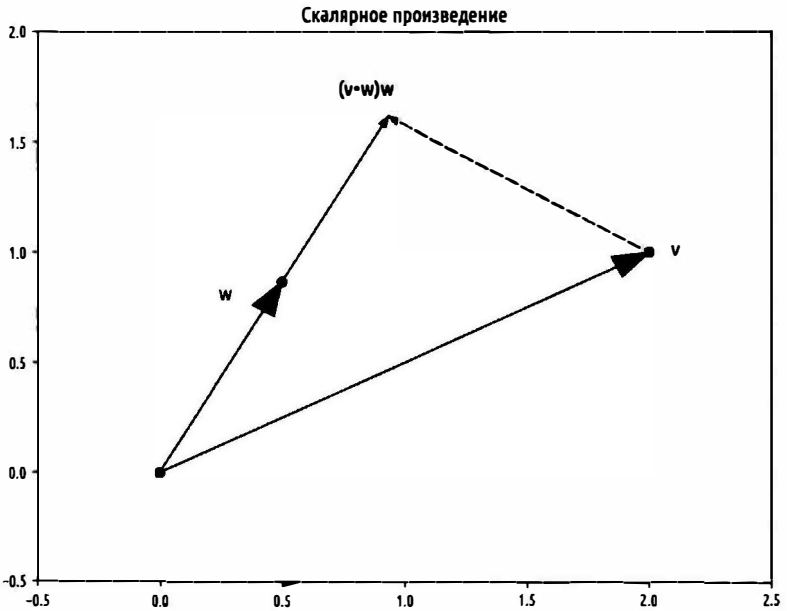


Рис. 4.2. Скалярное произведение как проекция вектора

Используя эту операцию, легко вычислить *сумму квадратов* вектора:

```
def sum_of_squares(v: Vector) -> float:
    """Возвращает v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)

assert sum_of_squares([1, 2, 3]) == 14     # 1 * 1 + 2 * 2 + 3 * 3
```

которую можно применить для вычисления его *магнитуды* (или длины):

```
import math

def magnitude(v: Vector) -> float:
    """Возвращает магнитуду (или длину) вектора v"""
```

```
    # math.sqrt - это функция квадратного корня
    return math.sqrt(sum_of_squares(v))
```

Теперь у нас есть все части, необходимые для вычисления (*евклидова*) расстояния между двумя векторами по формуле:

$$\sqrt{(v_1 - w_1)^2 + \ldots + (v_n - w_n)^2}.$$

В исходном коде оно выглядит так:

```
# Квадрат расстояния между двумя векторами
def squared_distance(v: Vector, w: Vector) -> float:
    """Вычисляет (v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
    return sum_of_squares(subtract(v, w))

# Расстояние между двумя векторами
def distance(v: Vector, w: Vector) -> float:
    """Вычисляет расстояние между v и w"""
    return math.sqrt(squared_distance(v, w))
```

Вероятно, будет понятнее, если мы запишем эту функцию как (эквивалент):

```
def distance(v: Vector, w: Vector) -> float:
    return magnitude(subtract(v, w))
```

Для начала этого вполне будет достаточно. Мы будем постоянно пользоваться этими функциями на протяжении всей книги.

Использование списков как векторов великолепно подходит в качестве иллюстрации, но является крайне неэффективным по производительности.

В рабочем коде следует пользоваться библиотекой NumPy, в которой есть класс, реализующий высокоэффективный массив с разнообразными арифметическими операциями.

Матрицы

Матрица — это двумерная коллекция чисел. Мы будем реализовывать матрицы как списки списков, где все внутренние списки имеют одинаковый размер и представляют *строку* матрицы. Если `A` — это матрица, то `A[i][j]` — это элемент в *i*-й строке и *j*-м столбце. В соответствии с правилами математической записи для обозначения матриц мы будем в основном использовать заглавные буквы. Например:

```
# Еще один псевдоним типа
Matrix = List[List[float]]

A = [[1, 2, 3],     # Матрица A имеет 2 строки и 3 столбца
     [4, 5, 6]]

B = [[1, 2],        # Матрица B имеет 3 строки и 2 столбца
     [3, 4],
     [5, 6]]
```

В математике, как правило, мы именуем первую строку матрицы "строкой 1" и первый столбец — "столбцом 1". Но поскольку мы реализуем матрицы в виде списков Python, которые являются нуль-индексными, то будем называть первую строку матрицы "строкой 0", а первый столбец — "столбцом 0".

Представленная в виде списка списков матрица `A` имеет `len(A)` строк и `len(A[0])` столбцов. Эти значения образуют *форму* матрицы:

```
from typing import Tuple

def shape(A: Matrix) -> Tuple[int, int]:
    """Возвращает (число строк A, число столбцов A)"""
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0  # Число элементов в первой строке
    return num_rows, num_cols

assert shape([[1, 2, 3], [4, 5, 6]]) == (2, 3) # 2 строки, 3 столбца
```

Если матрица имеет n строк и k столбцов, то мы будем говорить, что это $(n \times k)$-матрица, или матрица размера $n \times k$. Каждая строка $(n \times k)$-матрицы может быть представлена вектором длины k, а каждый столбец — вектором длины n:

```
def get_row(A: Matrix, i: int) -> Vector:
    """Возвращает i-ю строку A (как тип Vector)"""
    return A[i]             # A[i] является i-й строкой

def get_column(A: Matrix, j: int) -> Vector:
    """Возвращает j-й столбец A (как тип Vector)"""
    return [A_i[j]          # j-й элемент строки A_i
            for A_i in A]   # для каждой строки A_i
```

Нам также потребуется возможность создавать матрицу при наличии ее формы и функции, которая генерирует ее элементы. Это можно сделать на основе вложенного включения в список:

```
from typing import Callable

def make_matrix(num_rows: int,
                num_cols: int,
                entry_fn: Callable[[int, int], float]) -> Matrix:
    """
    Возвращает матрицу размера num_rows x num_cols,
    чей (i,j)-й элемент является функцией entry_fn(i, j)
    """
    return [[entry_fn(i, j)             # Создать список с учетом i
             for j in range(num_cols)]  # [entry_fn(i, 0), ... ]
            for i in range(num_rows)]   # Создать один список для каждого i
```

При наличии этой функции можно, например, создать (5×5)-*матрицу тождественности* (с единицами по диагонали и нулями в остальных элементах):

```
def identity_matrix(n: int) -> Matrix:
    """
    Возвращает (n x n)-матрицу тождественности,
    также именуемую единичной
    """
    return make_matrix(n, n, lambda i, j: 1 if i == j else 0)

assert identity_matrix(5) == [[1, 0, 0, 0, 0],
                              [0, 1, 0, 0, 0],
                              [0, 0, 1, 0, 0],
                              [0, 0, 0, 1, 0],
                              [0, 0, 0, 0, 1]]
```

Матрицы будут для нас важны по нескольким причинам.

Во-первых, мы можем использовать матрицу для представления набора данных, состоящего из нескольких векторов, рассматривая каждый вектор как строку матрицы. Например, если бы у вас были данные о росте, массе и возрасте 1000 человек, то их можно представить в виде (1000×3)-матрицы:

```
data = [[70, 170, 40],
        [65, 120, 26],
        [77, 250, 19],
        # ...
       ]
```

Во-вторых, как мы убедимся далее, $(n \times k)$-матрицу можно использовать в качестве линейной функции, которая отображает k-размерные векторы в n-размерные векторы. Некоторые наши технические решения и концепции будут привлекать такие функции.

И, в-третьих, матрицы можно использовать для представления бинарных связей. В *главе 1* мы представили ребра сети как коллекцию пар `(i, j)`. Альтернативная реализация состоит в создании *матрицы смежности*, т. е. матрицы `A`, такой, что ее элемент `A[i][j]` равен 1, если узлы `i` и `j` соединены между собой, и 0 — в противном случае.

Вспомните, что ранее у нас дружеские связи были списком кортежей:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

Мы можем представить то же самое следующим образом:

```
# Пользователь  0  1  2  3  4  5  6  7  8  9
#
friendships = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0],  # Пользователь  0
               [1, 0, 1, 1, 0, 0, 0, 0, 0, 0],  # Пользователь  1
               [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],  # Пользователь  2
               [0, 1, 1, 0, 1, 0, 0, 0, 0, 0],  # Пользователь  3
               [0, 0, 0, 1, 0, 1, 0, 0, 0, 0],  # Пользователь  4
               [0, 0, 0, 0, 1, 0, 1, 1, 0, 0],  # Пользователь  5
```

```
    [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],  # Пользователь 6
    [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],  # Пользователь 7
    [0, 0, 0, 0, 0, 0, 1, 1, 0, 1],  # Пользователь 8
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]]  # Пользователь 9
```

Если дружеских связей немного, то такое представление имеет гораздо меньшую эффективность, т. к. приходится хранить много нулей. Однако в таком матричном представлении проверка наличия связи между двумя узлами выполняется намного быстрее — вместо того, чтобы обследовать каждое ребро в списке, нам нужно всего лишь заглянуть в матрицу:

```
friendships[0][2] == 1    # True, 0 и 2 являются друзьями
friendships[0][8] == 1    # False, 0 и 8 не являются друзьями
```

Схожим образом, для того чтобы найти связи узла, вам нужно лишь обследовать столбец (или строку), соответствующий этому узлу:

```
# Нужно взглянуть лишь на одну строку
friends_of_five = [i
                   for i, is_friend in enumerate(friendships[5])
                   if is_friend]
```

С малым графом, ускоряя этот процесс, вы могли бы просто добавлять список связей в каждый узловой объект, но в случае крупного эволюционирующего графа это, вероятно, будет стоить слишком дорого и будет усложнять сопровождение.

Мы будем постоянно возвращаться к теме матриц на протяжении всей книги.

Для дальнейшего изучения

Линейная алгебра активно используется исследователями данных (часто неявным образом и нередко людьми, не вполне разбирающимися в ней). Поэтому неплохо было бы прочитать учебник по этой теме. В Интернете можно найти бесплатные пособия:

- "Линейная алгебра" Джими Хефферона (Jim Hefferon) из Вермонтского колледжа в Сан-Мишель[2] ;
- "Линейная алгебра" Дэвида Черны (David Cherney) и соавт. из Калифорнийского университета в Дэвисе[3];
- для тех, кто не боится сложностей, более продвинутое введение — "Неправильная линейная алгебра" Сергея Трейла (Sergei Treil)[4];
- весь механизм, который мы разработали в этой главе, можно получить бесплатно в библиотеке NumPy[5], и даже больше, включая более высокую производительность.

---

[2] См. **http://joshua.smcvt.edu/linearalgebra/**.

[3] См. **https://www.math.ucdavis.edu/~linear/linear-guest.pdf**.

[4] См. **http://www.math.brown.edu/~treil/papers/LADW/LADW.html**.

[5] См. **http://www.numpy.org/**.

ГЛАВА 5
Статистика

Факты — упрямая вещь, а статистика гораздо сговорчивее.
– *Марк Твен*[1]

Статистика опирается на математику и приемы, с помощью которых мы проникаем в сущность данных. Она представляет собой богатую и обширную область знаний, для которой больше подошла бы книжная полка или целая комната в библиотеке, а не глава в книге, поэтому наше изложение, конечно же, будет кратким. Тем не менее представленного в этой главе материала вам будет достаточно для того, чтобы стать опасным человеком, и пробудить в вас интерес к самостоятельному изучению статистики.

Описание одиночного набора данных

Благодаря полезному сочетанию живого слова и удачи социальная сеть DataSciencester выросла до нескольких десятков пользователей, и директор по привлечению финансовых ресурсов просит вас проанализировать, сколько друзей есть у пользователей сети, чтобы он мог включить эти данные в свои "презентации для лифта"[2].

Используя простые методы из *главы 1*, вы легко можете предъявить запрашиваемые данные. Однако сейчас вы столкнулись с задачей выполнения их *описательного анализа*.

Любой набор данных очевидным образом характеризует сам себя:

```
# Число друзей
num_friends = [100, 49, 41, 40, 25,
               # ... и еще много других
              ]
```

Для достаточно малого набора данных такое описание может даже оказаться наилучшим. Но для более крупного набора данных это будет выглядеть очень гро-

[1] Первая часть афоризма была популяризована Марком Твеном, вторая приписывается Лоренсу Питеру (1919–1990) — канадско-американскому педагогу и литератору. — *Прим. пер.*

[2] Презентация для лифта (elevator pitch) — короткий рассказ о концепции продукта, проекта или сервиса, который можно полностью представить во время поездки на лифте. Используется для краткого изложения концепции нового бизнеса в целях получения инвестиций. — *Прим. пер.*

моздко и, скорее всего, непрозрачно. (Представьте, что у вас перед глазами список из 1 млн чисел.) По этой причине пользуются статистикой, с помощью которой выжимают и коммуницируют информацию о существенных признаках, присутствующих в данных.

В качестве первоначального подхода вы помещаете число друзей на гистограмму, используя словарь `Counter` и метод `plt.bar` (рис. 5.1):

```
from collections import Counter
import matplotlib.pyplot as plt

friend_counts = Counter(num_friends)
xs = range(101)                         # Максимум равен 100
ys = [friend_counts[x] for x in xs]     # Высота - это число друзей

plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Гистограмма количеств друзей")
plt.xlabel("Число друзей")
plt.ylabel("Число людей")
plt.show()
```

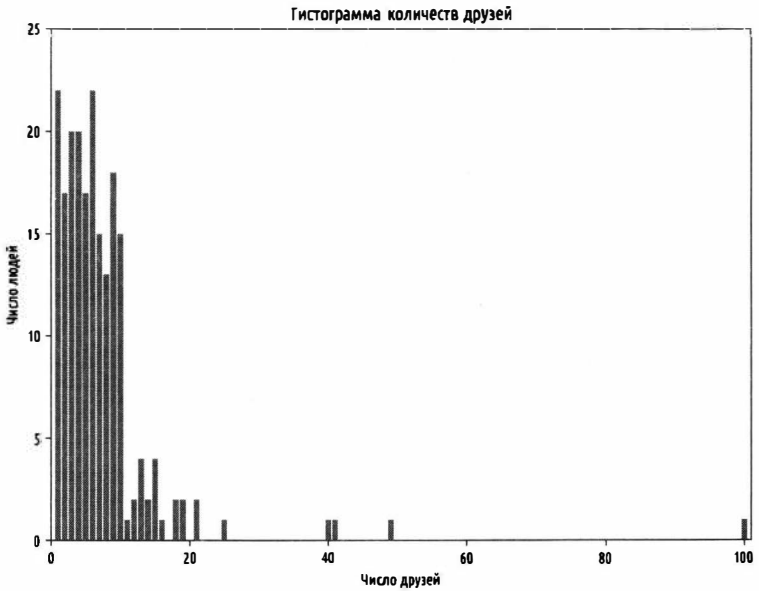


Рис. 5.1. Гистограмма количеств друзей

К сожалению, эту диаграмму крайне сложно упомянуть в разговоре, поэтому вы приступаете к генерированию некоторой статистики. Наверное, самым простым статистическим показателем является число точек данных:

```
num_points = len(num_friends)       # Число точек равно 204
```

Кроме этого, могут быть интересны наибольшие и наименьшие значения:

```
largest_value = max(num_friends)    # Наибольшее значение равно 100
smallest_value = min(num_friends)   # Наименьшее значение равно 1
```

которые являются особыми случаями, когда нужно узнать значения в определенных позициях:

```
sorted_values = sorted(num_friends)        # Отсортированные значения
smallest_value = sorted_values[0]          # Минимум равен 1
second_smallest_value = sorted_values[1]   # Следующий минимум равен 1
second_largest_value = sorted_values[-2]   # Следующий максимум равен 49
```

Но это только начало.

Центральные тенденции

Обычно мы хотим иметь некое представление о том, где наши данные центрированы. Чаще всего для этих целей используется *среднее* (или среднее арифметическое) значение, которое берется как сумма данных, деленная на их количество[3]:

```
# Среднее значение
def mean(xs: List[float]) -> float:
    return sum(x) / len(x)

mean(num_friends)     # 7.333333
```

Для двух точек средней является точка, лежащая посередине между ними. По мере добавления других точек среднее значение будет смещаться в разные стороны в зависимости от значения каждой новой точки. Например, если у вас 10 точек данных, и вы увеличиваете значение любой из них на 1, то вы увеличиваете среднее на 0.1.

Мы также иногда будем заинтересованы в *медиане*, которая является ближайшим к центру значением (если число точек данных нечетное) либо средним арифметическим, взятым как полусумма двух ближайших к середине значений (если число точек четное).

Например, если у нас есть отсортированный вектор `x` из пяти точек, то медианой будет `x[5 // 2]` или `x[2]`. Если в векторе шесть точек, то берется среднее арифметическое `x[2]` (третья точка) и `x[3]` (четвертая точка).

Обратите внимание, что медиана — в отличие от среднего — не зависит от каждого значения в наборе данных. Например, если сделать наибольшую точку еще больше (или наименьшую точку еще меньше), то срединные точки останутся неизменными, следовательно, и медиана тоже не изменится.

[3] Более общий вид имеет формула средней арифметической *взвешенной*, где каждое значение переменной сначала умножается на свой вес. В то время как средняя арифметическая *простая* — это частный случай, когда веса для всех значений равны 1. — *Прим. пер.*

Мы напишем разные функции для четого и нечетного случаев и объединим их:

```
# Символы подчеркивания указывают на то, что эти функции являются
# "приватными", т. к. они предназначены для вызова
# из нашей функции median,
# а не другими людьми, использующими нашу статитическую библиотеку
def _median_odd(xs: List[float]) -> float:
    """Если len(xs) является нечетной,
       то медиана - это срединный элемент"""
    return sorted(xs)[len(xs) // 2]

def _median_even(xs: List[float]) -> float:
    """Если len(xs) является четной, то она является средним значением
       двух срединных элементов"""
    sorted_xs = sorted(xs)
    hi_midpoint = len(xs) // 2  # напр. длина 4 => hi_midpoint 2
    return (sorted_xs[hi_midpoint - 1] + sorted_xs[hi_midpoint]) / 2

def median(v: List[float]) -> float:
    """Отыскивает 'ближайшее к середине' значение v"""
    return _median_even(v) if len(v) % 2 == 0 else _median_odd(v)

assert median([1, 10, 2, 9, 5]) == 5
assert median([1, 9, 2, 10]) == (2 + 9) / 2
```

И теперь мы можем вычислить медианное число друзей:

```
print(median(num_friends))   # 6
```

Ясно, что среднее значение вычисляется проще, и оно плавно варьирует по мере изменения данных. Если у нас есть n точек и одна из них увеличилась на любое малое число e, то среднее обязательно увеличится на e/n. (Этот факт делает его подверженным разного рода ухищрениям при калькуляции.) А для того чтобы найти медиану, данные нужно сперва отсортировать, и изменение одной из точек на любое малое число e может увеличить медиану на величину, равную e, меньшую чем e, либо не изменить совсем (в зависимости от остальных данных).

На самом деле существуют неочевидные приемы эффективного вычисления медиан[4] без сортировки данных. Поскольку их рассмотрение выходит за рамки этой книги, мы вынуждены будем сортировать данные.

Вместе с тем среднее значение очень чувствительно к *выбросам* в данных. Если бы самый дружелюбный пользователь имел 200 друзей (вместо 100), то среднее увеличилось бы до 7.82, а медиана осталась бы на прежнем уровне. Так как выбросы являются, скорее всего, плохими данными (или иначе — нерепрезентативными для ситуации, которую мы пытаемся понять), то среднее может иногда давать искажен-

[4] См. **https://en.wikipedia.org/wiki/Quickselect**.

ную картину. В качестве примера часто приводят историю, как в середине 1980-х годов в Университете Северной Каролины география стала специализацией с самой высокой стартовой среднестатистической зарплатой, что произошло в основном из-за звезды НБА (и выброса) Майкла Джордана.

Обобщением медианы является *квантиль*, который представляет значение, ниже которого располагается определенный процентиль данных (медиана представляет значение, ниже которого расположены 50% данных.)

```
def quantile(xs: List[float], p: float) -> float:
    """Возвращает значение p-го процентиля в x"""
    p_index = int(p * len(x))    # Преобразует % в индекс списка
    return sorted(x)[p_index]

assert quantile(num_friends, 0.10) == 1
assert quantile(num_friends, 0.25) == 3  # Нижний квантиль
assert quantile(num_friends, 0.75) == 9  # Верхний квантиль
assert quantile(num_friends, 0.90) == 13
```

Реже вам может понадобиться мода — значение или значения, которые встречаются наиболее часто:

```
def mode(x: List[float]) -> List[float]:
    """Возвращает список, т. к. может быть более одной моды"""
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.items()
            if count == max_count]

assert set(mode(num_friends)) == {1, 6}
```

Но чаще всего мы будем просто использовать среднее значение.

Вариация

Вариация служит мерой разброса наших данных. Как правило, это статистические показатели, у которых значения, близкие к нулю, означают полное *отсутствие разброса*, а большие значения (что бы это ни означало) — *очень большой разброс*. Например, самым простым показателем является *размах*, который определяется как разница между максимальным и минимальным значениями данных:

```
# Ключевое слово "range" (размах) в Python уже имеет
# свой смысл, поэтому берем другое
def data_range(xs: List[float]) -> float:
    return max(x) - min(x)

assert data_range(num_friends) == 99
```

Размах равен нулю, когда `max` и `min` эквивалентны, что происходит только тогда, когда все элементы `x` равны между собой, и значит, разбросанность в данных от-

сутствует. И наоборот, когда размах широкий, то `max` намного больше `min`, и разбросанность в данных высокая.

Как и медиана, размах не особо зависит от всего набора данных. Набор данных, все точки которого равны 0 или 100, имеет тот же размах, что и набор данных, чьи значения представлены числами 0, 100 и большого количества чисел 50, хотя кажется, что первый набор "должен" быть разбросан больше.

Более точным показателем вариации является *дисперсия*, вычисляемая как:

```
from scratch.linear_algebra import sum_of_squares

def de_mean(xs: List[float]) -> List[float]:
    """Транслировать xs путем вычитания его среднего
       (результат имеет нулевое среднее)"""
    x_bar = mean(xs)
    return [x - x_bar for x in xs]

def variance(xs: List[float]) -> float:
    """Почти среднеквадратическое отклонение от среднего"""
    assert len(xs) >= 2, "дисперсия требует наличия не менее двух элементов"

    n = len(xs)
    deviations = de_mean(xs)
    return sum_of_squares(deviations) / (n - 1)

assert 81.54 < variance(num_friends) < 81.55
```

Она выглядит почти как среднеквадратическое отклонение от среднего, но с одним исключением: в знаменателе не `n`, а на `n - 1`. В действительности, когда имеют дело с выборкой из более крупной популяции (генеральной совокупности), переменная `x_bar` является лишь *приближенной оценкой* среднего, где `(x_i - x_bar) ** 2` в среднем дает заниженную оценку квадрата отклонения от среднего для `x_i`. Поэтому делят не на `n`, а на `(n - 1)`[5]. См. Википедию[6].

При этом в каких бы единицах ни измерялись данные (в "друзьях", например), все меры центральной тенденции вычисляются в тех же самых единицах измерения. Аналогичная ситуация и с размахом. Дисперсия же измеряется в единицах, которые представляют собой *квадрат* исходных единиц ("друзья в квадрате"). Поскольку такие единицы измерения трудно интерпретировать, то вместо дисперсии мы будем чаще обращаться к *стандартному отклонению* (корню из дисперсии):

---

[5] Поскольку соотношение между выборочной и популяционной дисперсией составляет $n/(n-1)$, то с ростом n данное выражение стремится к 1, т. е. разница между значениями выборочной и популяционной дисперсиями уменьшается. На практике при $n > 30$ уже нет разницы, какое число стоит в знаменателе: n или $n-1$. — *Прим. пер.*

[6] См. **https://en.wikipedia.org/wiki/Unbiased_estimation_of_standard_deviation**.

```
import math

def standard_deviation(xs: List[float]) -> float:
    """Стандартное отклонение - это корень квадратный из дисперсии"""
    return math.sqrt(variance(xs))

assert 9.02 < standard_deviation(num_friends) < 9.043
```

Размах и стандартное отклонение имеют ту же проблему с выбросами, что и среднее. Используя тот же самый пример, отметим: если бы у самого дружелюбного пользователя было 200 друзей, то стандартное отклонение было бы 14.89, или на 60% выше!

Более надежной альтернативой является вычисление *интерквартильного размаха* или разности между значением, соответствующим 75% данных, и значением, соответствующим 25% данных:

```
def interquartile_range(xs: List[float]) -> float:
    """Возвращает разницу между 75%-ным и 25%-ным квартилями"""
    return quantile(xs, 0.75) - quantile(xs, 0.25)

assert interquartile_range(num_friends) == 6
```

который очевидным образом не находится под вдиянием небольшого числа выбросов.

Корреляция

У директора по развитию сети есть теория, что количество времени, которое люди проводят на веб-сайте, связано с числом их друзей (она же не просто так занимает должность директора), и она просит вас проверить это предположение.

Перелопатив журналы трафика, в итоге вы получили список `daily_minutes`, который показывает, сколько минут в день каждый пользователь тратит на DataSciencester, и упорядочили его так, чтобы его элементы соответствовали элементам нашего предыдущего списка `num_friends`. Мы хотели бы исследовать связь между этими двумя метриками.

Сперва обратимся к *ковариации* — парному аналогу дисперсии[7]. В отличие от дисперсии, которая измеряет отклонение одной-единственной переменной от ее среднего, ковариация измеряет отклонение двух переменных в тандеме от своих средних:

```
from scratch.linear_algebra import dot

def covariance(xs: List[float], ys: List[float]) -> float:
    assert len(xs) == len(ys), "xs и ys должны иметь одинаковое число элементов"
    return dot(de_mean(xs), de_mean(ys)) / (len(xs) - 1)
```

[7] Английский термин *covariance* дословно так и переводится — содисперсия или совместная дисперсия. — *Прим. пер.*

```
assert 22.42 < covariance(num_friends, daily_minutes) < 22.43
assert 22.42 / 60 < covariance(num_friends, daily_hours) < 22.43 / 60
```

Вспомните, что функция `dot` суммирует произведения соответствующих пар элементов *(см. главу 3)*. Когда соответствующие элементы обоих векторов `x` и `y` одновременно выше или ниже своих средних, то в сумму входит положительное число. Когда один из них находится выше своего среднего, а другой — ниже, то в сумму входит отрицательное число. Следовательно, "большая" положительная ковариация означает, что `x` стремится принимать большие значения при больших значениях `y` и малые значения — при малых значениях `y`. "Большая" отрицательная ковариация означает обратное — `x` стремится принимать малые значения при большом `y`, и наоборот. Ковариация, близкая к нулю, означает, что такой связи не существует.

Тем не менее этот показатель бывает трудно интерпретировать, и вот почему:

- единицами измерения ковариации являются произведения единиц входящих переменных (например, число друзей и минуты в день), которые трудно понять (что такое "друг в минуту в день"?);
- если бы у каждого пользователя было в 2 раза больше друзей (но такое же количество минут, проведенных на веб-сайте), то ковариация была бы в 2 раза больше. Однако в некотором смысле степень взаимосвязи между ними осталась бы на прежнем уровне. Говоря иначе, трудно определить, что считать "большой" ковариацией.

Поэтому чаще обращаются к *корреляции*, в которой ковариация распределяется между стандартными отклонениями обеих переменных:

```
def correlation(xs: List[float], ys: List[float]) -> float:
    """Измеряет степень, с которой xs и ys варьируются
       в тандеме вокруг своих средних"""
    stdev_x = standard_deviation(xs)
    stdev_y = standard_deviation(ys)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(xs, ys) / stdev_x / stdev_y
    else:
        return 0  # если вариации нет, то корреляция равна

assert 0.24 < correlation(num_friends, daily_minutes) < 0.25
assert 0.24 < correlation(num_friends, daily_hours) < 0.25
```

Корреляция является безразмерной величиной, ее значения всегда лежат между –1 (идеальная антикорреляция) и 1 (идеальная корреляция). Так, число 0.25 представляет собой относительно слабую положительную корреляцию.

Впрочем, мы забыли сделать одну вещь — проверить наши данные. Посмотрим на рис. 5.2.

Человек, у которого 100 друзей, проводит на веб-сайте всего одну минуту в день, и поэтому он представляет собой "дикий" выброс, а корреляция, как будет показано

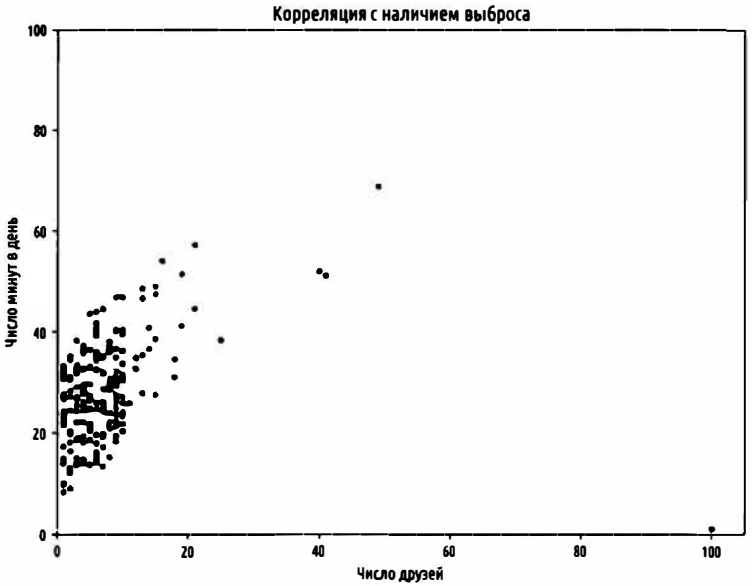


Рис. 5.2. Корреляция с наличием выброса

далее, может быть очень чувствительной к выбросам. Что произойдет, если мы проигнорируем этот выброс?

```
# Отфильтровать выброс
outlier = num_friends.index(100)  # Индекс выброса

num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]

daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if i != outlier]

daily_hours_good = [dm / 60 for dm in daily_minutes_good]

assert 0.57 < correlation(num_friends_good, daily_minutes_good) < 0.58
assert 0.57 < correlation(num_friends_good, daily_hours_good) < 0.58
```

Без выброса получается более сильная корреляция (рис. 5.3).

Вы исследуете данные дальше и обнаруживаете, что выброс на самом деле оказался техническим *тестовым* аккаунтом, который забыли удалить, и вы поступили совершенно правильно, исключив его из рассмотрения.

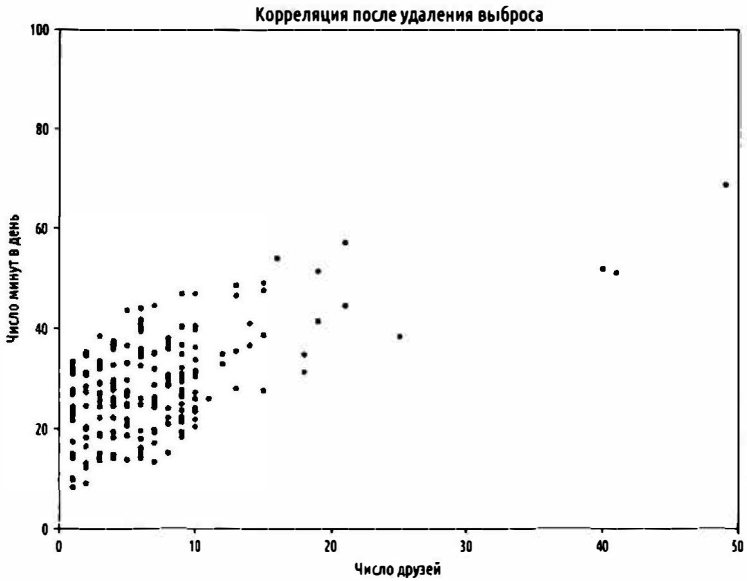


Рис. 5.3. Корреляция после удаления выброса

Парадокс Симпсона

Нередким сюрпризом при анализе данных является парадокс Симпсона[8], в котором корреляции могут быть обманчивыми, если игнорируются *спутывающие* переменные[9].

Например, представим, что вы можете отождествить всех членов сети как исследователей данных с Восточного побережья либо как исследователей данных с Западного побережья. Вы решаете выяснить, с какого побережья исследователи данных дружелюбнее (табл. 5.1).

***Таблица 5.1.** География членов сети*

Побережье	Количество пользователей	Среднее число друзей
Западное	101	8.2
Восточное	103	6.5

[8] *Парадокс Симпсона*, или парадокс объединения, — статистический эффект, когда при наличии двух групп данных, в каждой из которых наблюдается одинаково направленная зависимость, при объединении этих групп направление зависимости меняется на противоположное. Иллюстрирует неправомерность некоторых обобщений (см. **https://ru.wikipedia.org/wiki/Парадокс_Симпсона**). — *Прим. пер.*

[9] Спутывающая переменная (англ. *confounding variable*) — внешняя переменная статистической модели, которая коррелирует (положительно или отрицательно) как с зависимой, так и с независимой переменной. Ее игнорирование приводит к смещению модельной оценки. — *Прим. пер.*

Очевидно, что исследователи данных с Западного побережья дружелюбнее, чем исследователи данных с Восточного побережья. Ваши коллеги выдвигают разного рода предположения, почему так происходит: может быть, дело в солнце, или в кофе, или в органических продуктах, или в безмятежной атмосфере Тихого океана?

Продолжая изучать данные, вы обнаруживаете что-то очень необычное. Если учитывать только членов с ученой степенью, то у иследователей с Восточного побережья друзей в среднем больше (табл. 5.2), а если рассматривать только членов без ученой степени, то у исследователей с Восточного побережья друзей снова оказывается в среднем больше!

Таблица 5.2. *География и ученая степень членов сети*

Побережье	Степень	Количество пользователей	Среднее число друзей
Западное	Есть	35	3.1
Восточное	Есть	70	3.2
Западное	Нет	66	10.9
Восточное	Нет	33	13.4

Как только вы начинаете учитывать ученые степени, корреляция движется в обратную сторону! Группировка данных по признаку "восток — запад" скрыла тот факт, что среди исследователей с Восточного побережья имеется сильная асимметрия в сторону ученых степеней.

Подобный феномен возникает в реальном мире с определенной регулярностью. Главным моментом является то, что корреляция измеряет связь между двумя переменными *при прочих равных условиях*. Если данным назначать классы случайным образом, как и должно быть в хорошо поставленном эксперименте, то допущение "при прочих равных условиях" может быть и не плохим.

Единственный реальный способ избежать таких неприятностей — это *знать* свои данные и обеспечивать проверку всех возможных спутывающих факторов. Очевидно, это не всегда возможно. Если бы у вас не было данных об образовании 200 исследователей данных, то вы бы просто решили, что есть нечто, присущее людям с Западного побережья, делающее их общительнее.

Некоторые другие корреляционные ловушки

Корреляция, равная нулю, означает отсутствие линейной связи между двумя переменными. Однако могут быть совсем другие виды зависимостей. Например, если:

```
x = [-2, -1, 0, 1, 2]
y = [ 2,  1, 0, 1, 2]
```

то переменные `x` и `y` имеют нулевую корреляцию, но связь между ними определенно существует — каждый элемент `y` равен абсолютному значению соответствующего элемента `x`. Но в них отсутствует связь, при которой знание о том, как `x_i` со-

относится со средним `mean(x)`, дает нам информацию о том, как `y_i` соотносится со средним `mean(y)`. Это как раз тот тип связи, который корреляция пытается найти.

Кроме того, корреляция ничего не говорит о величине связи. Переменные:

```
x = [-2, 1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]
```

имеют очень хорошую корреляцию, но в зависимости от того, что вы измеряете, вполне возможно, что эта связь не представляет какого-то интереса.

Корреляция и причинно-следственная связь

Наверное, некоторым доводилось слышать утверждение, что "корреляция — это не каузация". Скорее всего, такая фраза принадлежит тому, кто, глядя на данные, которые противоречат какой-то части его мировоззрения, отказывается ставить его под сомнение. Тем не менее важно понимать, что если `x` и `y` сильно коррелированы, то это может означать, что либо `x` влечет за собой `y`, либо `y` влечет за собой `x`, либо они взаимообусловлены, либо они зависят от какого-то третьего фактора, либо это вовсе ничего не означает.

Рассмотрим связь между списками `num_friends` и `daily_minutes`. Возможно, что чем больше у пользователя друзей на веб-сайте, тем больше он проводит там времени. Это может быть из-за того, что каждый друг ежедневно публикует определенное число сообщений, и поэтому чем больше у вас друзей, тем больше вам требуется времени, чтобы оставаться в курсе последних сообщений.

Однако возможно и то, что чем больше времени пользователи проводят в спорах на форумах DataSciencester, тем чаще они знакомятся с единомышленниками и становятся друзьями. Иными словами, увеличение времени, проводимого на веб-сайте, *приводит* к росту числа друзей у пользователей.

Третий вариант заключается в том, что пользователи, которые очень увлечены наукой о данных, проводят на веб-сайте больше времени (потому что им это интересно) и более активно собирают вокруг себя друзей, специализирующихся в данной области (потому что они не хотят общаться с кем-либо еще).

Один из способов укрепить свою позицию в отношении причинно-следственной связи заключается в проведении рандомизированных испытаний. Если случайным образом распределить пользователей на две группы (экспериментальную и контрольную) с похожей демографией и предоставить экспериментальной группе несколько иной опыт взаимодействия, то нередко можно получить подтверждение, что разный опыт взаимодействия приводит к разным результатам.

Например, если вы не боитесь, что вас обвинят в экспериментах над вашими пользователями[10], то вы могли бы случайным образом выделить из числа пользователей

[10] См. **http://www.nytimes.com/2014/06/30/technology/facebook-tinkers-with-users-emotions-in-news-feed-experiment-stirring-outcry.html.**

сети подгруппу и показывать им только небольшую часть их друзей. Если эта подгруппа в дальнейшем проводила бы на веб-сайте меньше времени, то этот факт вселил бы в вас некоторую уверенность, что наличие большего числа друзей является *причиной* большего количества времени, проводимого на веб-сайте.

Для дальнейшего изучения

- Библиотеки SciPy[11], pandas[12] и StatsModels[13] поставляются с широким набором статистических функций.
- Статистика имеет *особое* значение (надо признать, и специалисты-статистики тоже). Если вы хотите стать хорошим исследователем данных, то стоит прочесть учебник по статистике. Многие из них доступны в Интернете.
  - "Вводная статистика" Дугласа Шейфера (Douglas Shafer) и соавт. (Saylor Foundation)[14].
  - "OnlineStatBook" Дэвида Лэйна (David Lane), Rice University[15].
  - "Вводная статистика", колледж OpenStax[16].

[11] См. **https://www.scipy.org/**.

[12] См. **http://pandas.pydata.org/**.

[13] См. **http://www.statsmodels.org/**.

[14] См. **https://open.umn.edu/opentextbooks/textbooks/introductory-statistics**.

[15] См. **http://onlinestatbook.com/**.

[16] См. **https://openstax.org/details/introductory-statistics**.

ГЛАВА 6
Вероятность

> Законы вероятности такие истинные в целом и такие ошибочные в частности.
> – *Эдвард Гиббон*[1]

Трудно заниматься наукой о данных, не имея представления о *вероятности* и ее математическом аппарате. Так же как и с обсуждением статистики в *главе 5*, здесь мы снова будем много размахивать руками, но обойдем стороной значительную часть технических подробностей.

Для наших целей будем представлять вероятность как способ количественной оценки неопределенности, ассоциированной с *событиями* из некоторого *вероятностного пространства*. Вместо того чтобы вдаваться в технические тонкости по поводу точного определения этих терминов, лучше представьте бросание кубика. Вероятностное пространство состоит из множества всех возможных элементарных исходов (в случае кубика их шесть — по числу граней), а любое подмножество этих исходов представляет собой случайное событие, например, такое как "выпала грань с единицей" или "выпала грань с четным числом".

Вероятность случайного события E мы будем обозначать через $P(E)$.

В дальнейшем мы будем применять теорию вероятностей как при построении моделей, так и при их вычислении. Иными словами, она будет использоваться постоянно.

Конечно, при желании можно погрузиться в философию и порассуждать о *сущности* теории вероятностей (этим неплохо заниматься за кружкой пива, например), но мы не будем этого делать.

Взаимная зависимость и независимость

Грубо говоря, два события E и F являются взаимно *зависимыми*, если какое-то знание о наступлении события E дает нам информацию о наступлении события F, и наоборот. В противном случае они являются взаимно *независимыми*.

Например, если мы дважды бросаем уравновешенную монету, то знание о том, что в первый раз выпадет орел, не дает никакой информации о том, что орел выпадет и

[1] Эдвард Гиббон (1737–1794) — знаменитый английский историк, автор "Истории упадка и разрушения Римской империи". — *Прим. пер.*

во второй раз. Эти события взаимно независимые. С другой стороны, знание о том, что в первый раз выпадет орел, определенно дает нам информацию о том, выпадут ли решки оба раза. (Если в первый раз выпадет орел, то, конечно же, исключаем случай, когда оба раза выпадают решки.) Оба этих события являются взаимно зависимыми.

С точки зрения математики говорят, что два события E и F являются взаимно *независимыми*, если вероятность их совместного наступления равна произведению вероятностей их наступления по отдельности:

$$P(E, F) = P(E)P(F).$$

В примере с бросанием монеты вероятность события, что в первый раз выпадет орел, равна 1/2, вероятность события, что оба раза выпадут решки, равна 1/4, а вероятность, что в первый раз выпадет орел *и* оба раза выпадут решки, равна 0.

Условная вероятность

Когда два события E и F являются взаимно независимыми, то по определению вероятность их совместного наступления равна:

$$P(E, F) = P(E)P(F).$$

Если же они не обязательно являются взаимно независимыми (и при этом вероятность F не равна 0), то *условная вероятность* события E при условии события F определяется так:

$$P(E \mid F) = \frac{P(E, F)}{P(F)}.$$

Под ней понимается вероятность наступления события E при условии, что известно о наступлении события F.

Эта формула часто приводится к следующему виду:

$$P(E, F) = P(E \mid F)P(F).$$

В случае когда E и F — взаимно независимые события, можно убедиться, что формула примет вид:

$$P(E \mid F) = P(E),$$

и на математическом языке будет выражать, что знание о наступлении события F не дает никакой дополнительной информации о наступлении события E.

Для демонстрации условной вероятности обычно приводят следующий замысловатый пример с семьей, где есть двое детей, чей пол нам неизвестен. При этом допустим, что:

- каждый ребенок равновероятно является либо мальчиком, либо девочкой;
- пол второго ребенка не зависит от пола первого.

Тогда событие, что оба ребенка не девочки, имеет вероятность 1/4; событие, что одна девочка и один мальчик, имеет вероятность 1/2; а событие, что обе — девочки, имеет вероятность 1/4.

Теперь мы можем задаться вопросом: какова вероятность события, когда оба ребенка — девочки (B), при условии, что старший ребенок — девочка (G)? Используя определение условной вероятности, мы получим:

$$P(B|G) = \frac{P(B,G)}{P(G)} = \frac{P(B)}{P(G)} = 1/2,$$

поскольку событие B *и* G (оба ребенка — девочки *и* старший ребенок — девочка) — это просто событие B. (Если вы знаете, что оба ребенка — девочки, то безусловно является истиной, что старший ребенок — девочка.)

И, скорее всего, такой результат будет в согласии с нашей интуицией.

Мы могли бы задаться вопросом о вероятности события, когда оба ребенка — девочки при условии, что как минимум один ребенок — девочка (L). И удивительная вещь — ответ будет отличаться от ранее полученного!

Как и раньше, событие B *и* L (оба ребенка — девочки *и* не менее одного ребенка — девочка) — это просто событие B. Поэтому имеем:

$$P(B|L) = \frac{P(B,L)}{P(L)} = \frac{P(B)}{P(L)} = 1/3.$$

Как такое может быть? Все дело в том, что если известно, что как минимум один ребенок — девочка, то вероятность, что в семье имеется один мальчик и одна девочка, в два раза выше, чем вероятность, что имеются две девочки[2].

Мы можем проверить это, "сгенерировав" большое число семей:

```
import enum, random

# Enum - это типизированное множество перечислимых значений.
# Мы можем их использовать для того, чтобы сделать наш код
# описательнее и читабельнее.
class Kid(enum.Enum):
    BOY = 0
    GIRL = 1

def random_kid() -> Kid:
    return random.choice([Kid.BOY, Kid.GIRL])
```

[2] Речь идет о парадоксе мальчика и девочки. Когда известно, что в семье как минимум одна девочка, мы автоматически отметаем вариант с двумя мальчиками. А из того, что оставшиеся три исхода равновероятны, делается вывод, что вероятность "девочка — девочка" равна 1/3.
См. **https://ru.wikipedia.org/wiki/Парадокс_мальчика_и_девочки**. — *Прим. пер.*

```
both_girls = 0
older_girl = 0
either_girl = 0

random.seed(0)

for _ in range(10000):
    younger = random_kid()
    older = random_kid()
    if older == Kid.GIRL:
        older_girl += 1
    if older == Kid.GIRL and younger == Kid.GIRL:
        both_girls += 1
    if older == Kid.GIRL or younger == Kid.GIRL:
        either_girl += 1

    print("P(both | older):", both_girls / older_girl)      # 0.514 ~ 1/2
    print("P(both | either): ", both_girls / either_girl)  # 0.342 ~ 1/3
```

Теорема Байеса

Лучшим другом исследователя данных является *теорема Байеса*[3], которая позволяет "переставлять" условные вероятности местами. Скажем, нам нужно узнать вероятность некоего события E, зависящего от наступления некоего другого события F, причем в наличии имеется лишь информация о вероятности события F, зависящего от наступления события E. Двукратное применение (в силу симметрии) определения условной вероятности даст формулу Байеса:

$$P(E \mid F) = \frac{P(E, F)}{P(F)} = \frac{P(F \mid E)P(E)}{P(F)}.$$

Если событие F разложить на два взаимоисключающих события — событие "F *и* E" и событие "F *и* не E" — и обозначить "не E" (т. е. E не наступает) как $\neg E$, тогда:

$$P(F) = P(F, E) + P(F, \neg E),$$

благодаря чему формула приводится к следующему виду:

$$P(E \mid F) = \frac{P(F \mid E)P(E)}{P(F \mid E)P(E) + P(F \mid \neg E)PP(\neg E)}.$$

Именно так формулируется теорема Байеса.

---

[3] Томас Байес (1702–1761) — английский математик и священник, который первым предложил использование теоремы для корректировки убеждений, основываясь на обновляемых данных. — *Прим. пер.*

Данную теорему часто используют для демонстрации того, почему исследователи данных умнее врачей[4]. Представим, что есть некая болезнь, которая поражает 1 из каждых 10 000 человек, и можно пройти обследование, выявляющее эту болезнь, которое в 99% случаев дает правильный результат ("болен", если заболевание имеется, и "не болен" — в противном случае).

Что означает положительный результат обследования? Пусть T — это событие, что "результат Вашего обследования положительный", а D — событие, что "у Вас имеется заболевание". Тогда, согласно теореме Байеса, вероятность наличия заболевания при положительном результате обследования равна:

$$P(D|T) = \frac{P(T|D)P(D)}{P(T|D)P(D) + P(T|\neg D)P(\neg D)}.$$

По условию задачи известно, что $P(T|D) = 0.99$ (вероятность, что заболевший получит положительный результат обследования), $P(D) = 1/10000 = 0.0001$ (вероятность, что любой человек имеет заболевание), $P(T|\neg D) = 0.01$ (вероятность, что здоровый человек получит положительный результат обследования) и $P(\neg D) = 0.9999$ (вероятность, что любое данное лицо не имеет заболевания). Если подставить эти числа в теорему Байеса, то:

$$P(D|T) = 0.98,$$

т. е. менее 1% людей, которые получат положительный результат обследования, имеют это заболевание на самом деле.

При этом считается, что люди проходят обследование более или менее случайно. Если же его проходят только те, у кого имеются определенные симптомы, то вместо этого пришлось бы обусловливать совместным событием "положительный результат обследования *и* симптомы", в результате чего, возможно, это число оказалось бы намного выше.

Интуитивно более понятный способ состоит в том, чтобы представить популяцию численностью 1 млн человек. 100 из них ожидаемо имеют заболевание, из которых 99 получили положительный результат обследования. С другой стороны, 999 900 из них ожидаемо не имеют заболевания, из которых 9999 получили положительный результат обследования, вследствие чего можно ожидать, что только 99 из (99 + 9999) получивших положительный результат обследования имеют это заболевание на самом деле[5].

---

[4] См. **https://ru.wikipedia.org/wiki/Теорема_Байеса**, пример 4. — *Прим. пер.*

[5] Приведенный пример называется *парадоксом теоремы Байеса*, который возникает, если берется редкое явление, такое как туберкулез, например. В этом случае возникает значительная разница в долях больных и здоровых, отсюда и удивительный результат. — *Прим. пер.*

Случайные величины

Случайная величина — это переменная, возможные значения которой ассоциированы с распределением вероятностей. Простая случайная величина равна 1, если подброшенная монета повернется орлом, и 0, если повернется решкой. Более сложная величина может измерять число орлов, наблюдаемых при 10 бросках, или значение, выбираемое из интервала `range(10)`, где каждое число является равновероятным.

Ассоциированное распределение предоставляет ей вероятности, с которыми она реализует каждое из своих возможных значений. Случайная величина броска монеты равна 0 с вероятностью 0.5 и 1 с той же вероятностью. Случайная величина `range(10)` имеет распределение, которое назначает вероятность 0.1 каждому числу от 0 до 9.

Мы иногда будем говорить о (среднем) ожидаемом значении или *математическом ожидании* случайной величины, которое представляет собой взвешенную сумму произведений каждого ее значения на его вероятность[6]. Среднее ожидаемое значение броска монеты равно 1/2 ($=0\cdot1/2+1\cdot1/2$); среднее ожидаемое значение `range(10)` равно 4.5.

Случайные величины могут *обусловливаться* событиями точно так же, как и другие события. Пользуясь приведенным выше примером с двумя детьми из *разд. "Условная вероятность"*, если X — это случайная величина, представляющая число девочек, то X равно 0 с вероятностью 1/4, 1 с вероятностью 1/2 и 2 с вероятностью 1/4.

Можно определить новую случайную величину Y, которая дает число девочек при условии, что как минимум один ребенок — девочка. Тогда Y равно 1 с вероятностью 2/3 и 2 с вероятностью 1/3. А также определить случайную величину Z, как число девочек при условии, что старший ребенок — девочка, равную 1 с вероятностью 1/2 и 2 с вероятностью 1/2.

В рассматриваемых задачах случайные величины будут использоваться по большей части неявным образом, без привлечения к ним особого внимания. Однако если копнуть глубже, то их можно обязательно обнаружить.

Непрерывные распределения

Бросание монеты соответствует *дискретному распределению*, т. е. такому, которое ассоциирует положительную вероятность с дискретными исходами. Однако нередко мы хотим моделировать распределения на непрерывном пространстве исходов. (Для целей изложения в книге эти исходы всегда будут вещественными числами, хотя в реальной жизни это не всегда так.) Например, *равномерное распределение* назначает *одинаковый вес* всем числам между 0 и 1.

Поскольку между 0 и 1 находится бесконечное количество чисел, то, значит, вес, который оно назначает индивидуальным точкам, должен с неизбежностью быть

[6] На практике при анализе выборок математическое ожидание, как правило, неизвестно. Поэтому вместо него используют его оценку — среднее арифметическое. — *Прим. пер.*

равен нулю. По этой причине мы представляем непрерывное распределение с помощью *функции плотности вероятности* (probability density function, PDF) — такой, что вероятность наблюдать значение в определенном интервале равна интегралу функции плотности над этим интервалом.

Если ваше интегральное исчисление хромает, то более простой способ понять это состоит в том, что если распределение имеет функцию плотности *f*, то вероятность наблюдать значение между *x* и *x* + *h* приближенно равна *h* · *f*(*x*) при малых значениях *h*.

Функция плотности равномерного распределения — это всего лишь:

```
def uniform_pdf(x: float) -> float:
    return 1 if x >= 0 and x < 1 else 0
```

Вероятность, что случайная величина, подчиняющаяся этому распределению, находится в интервале между 0.2 и 0.3, как и ожидалось, равна 1/10. Функция `random.random()` языка Python представляет собой (псевдо)случайную величину с равномерной плотностью.

Часто нас больше будет интересовать *кумулятивная функция распределения* (cumulative distribution function, CDF), которая дает вероятность, что случайная величина меньше или равна некоторому значению. Реализация указанной функции для равномерного распределения является элементарной (рис. 6.1):

```
def uniform_cdf(x: float) -> float:
    """Возвращает вероятность, что равномерно
       распределенная случайная величина <= x"""
    if x < 0:   return 0 # Равномерная величина никогда не бывает меньше 0
    elif x < 1: return x   # Например, P(X <= 0.4) = 0.4
    else:       return 1   # Равномерная величина всегда меньше 1
```

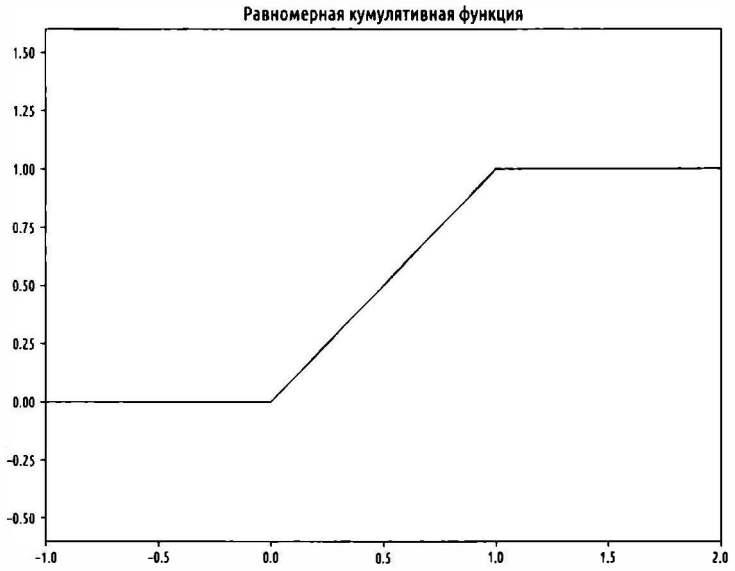


Рис. 6.1. Равномерная кумулятивная функция (CDF)

Нормальное распределение

Нормальное распределение — это классическое колоколообразное распределение. Оно полностью определяется двумя параметрами: его средним значением μ (мю) и его стандартным отклонением σ (сигмой). Среднее значение указывает, где колокол центрирован, а стандартное отклонение — насколько "широким" он является.

Его функция плотности распределения имеет следующий вид:

$$f(x|\mu,\sigma)=\frac{1}{\sqrt{2\pi\sigma}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right),$$

и ее можно имплементировать следующим образом:

```
import math
SQRT_TWO_PI = math.sqrt(2 * math.pi)

def normal_pdf(x: float, mu: float = 0, sigma: float = 1) -> float:
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (SQRT_TWO_PI * sigma))
```

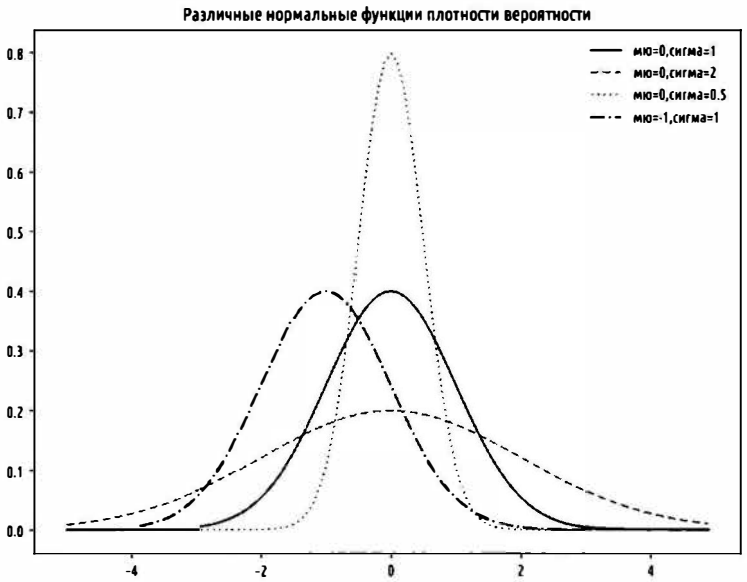


Рис. 6.2. Различные нормальные функции плотности вероятности (PDF)

На рис. 6.2 для наглядности приведены графики нескольких функций плотности вероятности (PDF):

```
import matplotlib.pyplot as plt

xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='мю=0, сигма=1')
```

```
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'--',label='мю=0, сигма=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='мю=0, сигма=0.5')
plt.plot(xs,[normal_pdf(x,mu=-1) for x in xs],'-.',label='мю=-1, сигма=1')
plt.legend()
plt.title("Различные нормальные функции плотности вероятности")
plt.show()
```

При $\mu = 0$ и $\sigma = 1$ распределение называется *стандартным нормальным распределением*. Если Z — это стандартная нормальная случайная величина, то оказывается, что

$$X = \sigma Z + \mu$$

тоже является нормальной, но со средним μ и стандартным отклонением σ. И наоборот, если X — нормальная случайная величина со средним μ и стандартным отклонением σ, то

$$Z = \frac{X - \mu}{\sigma}$$

есть стандартная нормальная случайная величина.

Кумулятивную функцию (CDF) для нормального распределения невозможно написать, пользуясь лишь "элементарными" средствами, однако это можно сделать при помощи функции интеграла вероятности `math.erf` языка Python:

```
def normal_cdf(x: float, mu: float = 0, sigma: float = 1) -> float:
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

И снова построим графики некоторых из кумулятивных функций (рис. 6.3):

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='мю=0, сигма=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='мю=0, сигма=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='мю=0, сигма=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='мю=-1, сигма=1')
plt.legend(loc=4)   # внизу справа
plt.title("Различные нормальные кумулятивные функции распределения")
plt.show()
```

Иногда нам нужно инвертировать кумулятивную функцию `normal_cdf`, чтобы отыскать значение, соответствующее указанной вероятности. Простой способ вычислить обратную функцию отсутствует, однако если учесть, что `normal_cdf` — непрерывная и монотонно возрастающая функция, то можно применить двоичный поиск[7]:

```
def inverse_normal_cdf(p: float,
                       mu: float = 0,
                       sigma: float = 1,
                       tolerance: float = 0.00001) -> float: # задать точность
    """Отыскать приближенную инверсию, используя бинарный поиск"""
```

[7] См. **https://ru.wikipedia.org/wiki/Двоичный_поиск**. — *Прим. пер.*

```
    # Если не стандарная, то вычислить стандартную и перешкалировать
    if mu != 0 or sigma != 1:
        return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)

    low_z = -10.0    # normal_cdf(-10) равно (находится очень близко к) 0
    hi_z = 10.0      # normal_cdf(10) равно (находится очень близко к) 1

    while hi_z - low_z > tolerance:
        mid_z = (low_z + hi_z) / 2   # Рассмотреть среднюю точку
        mid_p = normal_cdf(mid_z)    # и значение CDF
        if mid_p < p:
            low_z = mid_z  # Средняя точка слишком низкая, искать выше
        else:
            hi_z = mid_z   # Средняя точка слишком высокая, искать ниже
    return mid_z
```

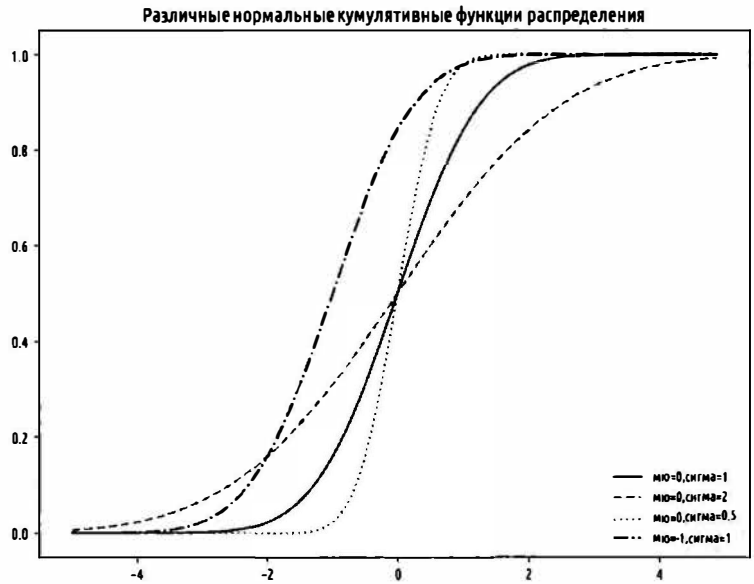


Рис. 6.3. Различные нормальные кумулятивные функции распределения

Функция многократно делит интервалы пополам, пока не выйдет на точку Z, которая достаточно близка к требуемой вероятности.

Центральная предельная теорема

Одна из причин распространенности нормального распределения заключается в *центральной предельной теореме* (ЦПТ), согласно которой (по существу) случайная величина, определенная как среднее большого числа взаимно независимых

и идентично распределенных случайных величин, сама является приближенно нормально распределенной.

В частности, если $x_1, ..., x_n$ — случайные величины со средним значением μ и стандартным отклонением σ и если n большое, то

$$\frac{1}{n}(x_1 + ... + x_n)$$

— это приближенно нормальная распределенная величина со средним значением μ и стандартным отклонением $\sigma/\sqrt{n}$. Эквивалентным образом (и чаще с большей практической пользой)

$$\frac{(x_1 + ... + x_n) - \mu n}{\sigma\sqrt{n}}$$

есть приближенно нормально распределенная величина с нулевым средним значением и стандартным отклонением, равным 1.

Это можно легко проиллюстрировать, обратившись к *биномиальным* случайным величинам, имеющим два параметра — n и p. Биномиальная случайная величина `binomial(n, p)` — это просто сумма n независимых случайных величин с распределением Бернулли `bernoulli(p)`, таких, что значение каждой из них равно 1 с вероятностью p и 0 с вероятностью $1 - p$:

```
def bernoulli_trial(p: float) -> int:
    """Возвращает 1 с вероятностью p и 0 с вероятностью 1-p"""
    return 1 if random.random() < p else 0

def binomial(n: int, p: float) -> int:
    """Возвращает сумму из n испытаний bernoulli(p)"""
    return sum(bernoulli_trial(p) for _ in range(n))
```

Среднее значение бернуллиевой величины `bernoulli(p)` равно p, ее стандартное отклонение равно $\sqrt{p(1-p)}$. Центральная предельная теорема констатирует, что по мере того, как n становится крупнее, биномиальная случайная величина `binomial(n,p)` является приближенно нормально распределенной со средним значением $\mu = np$ и стандартным отклонением $\sigma = \sqrt{np(1-p)}$. На диаграмме легко увидеть сходство обоих распределений:

```
from collections import Counter

def binomial_histogram(p: float, n: int, num_points: int) -> None:
    """Подбирает точки из binomial(n, p) и строит их гистограмму"""
    data = [binomial(n, p) for _ in range(num_points)]

    # Использовать столбчатый график
    # для показа фактических биномиальных выборок
```

```
    histogram = Counter(data)
    plt.bar([x - 0.4 for x in histogram.keys()],
            [v / num_points for v in histogram.values()],
            0.8,
            color='0.75')

    mu = p * n
    sigma = math.sqrt(n * p * (1 - p))

    # Использовать линейный график для показа нормальной аппроксимации
    xs = range(min(data), max(data) + 1)
    ys = [normal_cdf(i + 0.5, mu, sigma) - normal_cdf(i - 0.5, mu, sigma)
          for i in xs]

    plt.plot(xs,ys)
    plt.title("Биномиальное распределение и его нормальное приближение")
    plt.show()
```

Например, если вызвать приведенную выше функцию `binomial_histogram(0.75, 100, 10000)`, то мы получим диаграмму, как на рис. 6.4.

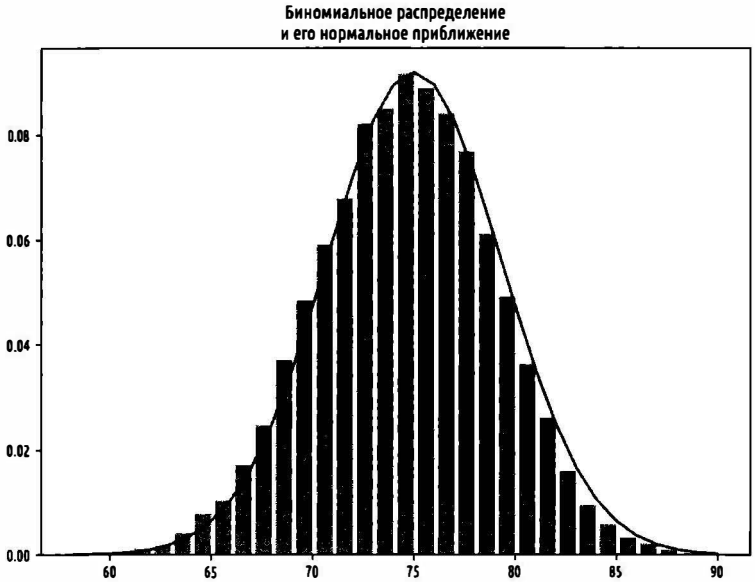


Рис. 6.4. Результат выполнения функции `binomial_histogram`

Мораль этого приближения заключается в том, что если (скажем) нужно узнать вероятность, что уравновешенная монета выпадет орлом более 60 раз из 100 бросков, то это можно вычислить как вероятность, что `normal(50,5)` больше 60, и это гораздо легче сделать, чем вычислять кумулятивную функцию для биномиального распределения `binomial(100,0.5)`. (Хотя в большинстве приложений вы, возможно, вос-

пользуетесь статистическим программным обеспечением, которое с готовностью вычислит любую вероятность, какую вы пожелаете.)

Для дальнейшего изучения

- Библиотека scipy.stats[8] содержит функции плотности вероятности и кумулятивные функции распределения для большинства популярных распределений вероятностей.
- Помните, как в конце *главы 5* мы отметили, что неплохо было бы изучить учебник по статистике? Это же касается и учебника по теории вероятностей. Лучший из имеющихся онлайн — "Введение в вероятность" Чарльза М. Гринстеда (Charles M. Grinstead) и Дж. Лори Снелла (J. Laurie Snell) (Американское математическое сообщество)[9].

[8] См. **https://docs.scipy.org/doc/scipy/reference/stats.html**.

[9] См. **http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book.html**.

ГЛАВА 7
Гипотеза и вывод

По-настоящему умного человека характеризует то, что им движет статистика.
– Джордж Бернард Шоу[1]

Для чего нужны все эти сведения из статистики и теории вероятностей? *Научная* сторона науки о данных часто предусматривает формулировку и проверку *статистических гипотез* о данных и процессах, которые их порождают.

Проверка статистической гипотезы

Нередко, будучи исследователями данных, мы хотим проверить, соответствует ли, по всей вероятности, истине определенная статистическая гипотеза. Для наших целей под *статистическими гипотезами* будем понимать утверждения типа "эта монета уравновешена" или "исследователи данных больше предпочитают Python, чем R", или "посетители, скорее всего, покинут страницу, не прочтя содержимого, если разместить на ней всплывающие окна с раздражающей рекламой и крошечной трудноразличимой кнопкой **Закрыть**". Все эти утверждения могут быть транслированы в статистические показатели о данных, или статистики. В условиях различных допущений эти статистики можно рассматривать в качестве наблюдений случайных величин из известных распределений, что позволяет делать утверждения о возможности, что эти допущения соблюдаются.

Классическая трактовка подразумевает наличие главной, или *нулевой, гипотезы* H_0, которая представляет некую позицию по умолчанию, и *альтернативной гипотезы* H_1, относительно которой мы хотим ее сопоставить. Для того чтобы принять решение, можно ли отклонить H_0 как ложную или принять ее как истинную, используют специальные статистики. По-видимому, будет разумнее показать это на примере.

Пример: бросание монеты

Представим, что у нас есть монета, которую требуется проверить, уравновешена ли она. Для этого делается допущение, что монета имеет некую вероятность p выпадения орла, и выдвигается нулевая гипотеза о том, что монета уравновешена, т. е. $p = 0.5$. Проверим ее, сопоставив с альтернативной гипотезой $p \neq 0.5$.

[1] Джордж Бернард Шоу (1856–1950) — ирландский драматург, писатель, романист. — *Прим. пер.*

В частности, наша проверка будет предусматривать бросание монеты *n* раз с подсчетом количества орлов *X*. Каждый бросок монеты — это бернуллиево испытание, где *X* — это биномиальная случайная величина `binomial(n, p)`, которую, как мы уже убедились в *главе 6*, можно аппроксимировать с помощью нормального распределения:

```
from typing import Tuple
import math

# Аппроксимация биномиальной случайной величины нормальным распределением
def normal_approximation_to_binomial(n: int, p: float) -> Tuple[float, float]:
    """Возвращает mu и sigma, соответствующие binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

Всякий раз, когда случайная величина подчиняется нормальному распределению, мы можем использовать функцию `normal_cdf` для выявления вероятности, что ее реализованное значение лежит в пределах или за пределами определенного интервала:

```
from scratch.probability import normal_cdf

# Нормальная функция CDF (normal_cdf) - это вероятность,
# что переменная лежит ниже порога
normal_probability_below = normal_cdf

# Она лежит выше порога, если она не ниже порога
def normal_probability_above(lo: float,
                             mu: float = 0,
                             sigma: float = 1) -> float:
    """Вероятность, что N(mu, sigma) выше, чем lo."""
    return 1 - normal_cdf(lo, mu, sigma)

# Она лежит между, если она меньше, чем hi, но не меньше, чем lo
def normal_probability_between(lo: float,
                               hi: float,
                               mu: float = 0,
                               sigma: float = 1) -> float:
    """Вероятность, что N(mu, sigma) между lo и hi."""
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

# Она лежит за пределами, если она не лежит между
def normal_probability_outside(lo: float,
                               hi: float,
                               mu: float = 0,
                               sigma: float = 1) -> float:
    """Вероятность, что N(mu, sigma) не лежит между lo и hi."""
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

Мы также можем сделать обратное — отыскать нехвостовой участок или же (симметричный) интервал вокруг среднего значения, на который приходится определенный уровень правдоподобия[2]. Например, если нам нужно отыскать интервал с центром в среднем значении, содержащий 60%-ную вероятность, то мы отыскиваем точки отсечения, где верхний и нижний "хвосты" содержат по 20% вероятности каждый (с остатком в 60%):

```
from scratch.probability import inverse_normal_cdf

# Верхняя граница
def normal_upper_bound(probability: float,
                       mu: float = 0,
                       sigma: float = 1) -> float:
    """Возвращает z, для которой P(Z <= z) = вероятность"""
    return inverse_normal_cdf(probability, mu, sigma)

# Нижняя граница
def normal_lower_bound(probability: float,
                       mu: float = 0,
                       sigma: float = 1) -> float:
    """Возвращает z, для которой P(Z >= z) = вероятность"""
    return inverse_normal_cdf(1 - probability, mu, sigma)

# Двусторонняя граница
def normal_two_sided_bounds(probability: float,
                            mu: float = 0,
                            sigma: float = 1) -> Tuple[float, float]:
    """Возвращает симметрические (вокруг среднего) границы,
       которые содержат указанную вероятность
    """
    tail_probability = (1 - probability) / 2

    # Верхняя граница должна иметь хвостовую tail_probability выше ее
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)

    # Нижняя граница должна иметь хвостовую tail_probability ниже ее
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)
    return lower_bound, upper_bound
```

[2] Значения понятий вероятности (probability) и правдоподобия (likelihood) различаются по роли параметра и результата. Вероятность используется для описания результата функции при наличии фиксированного значения параметра. (Если сделать 10 бросков уравновешенной монеты, какова вероятность, что она повернется 10 раз орлом?) Правдоподобие используется для описания параметра функции при наличии ее результата. (Если брошенная 10 раз монета повернулась 10 раз орлом, насколько правдоподобна уравновешенность монеты?) Кроме того, вероятность ограничена интервалом значений между 0 и 1, тогда как правдоподобие — нет. — *Прим. пер.*

В частности, будем считать, что мы решили сделать $n = 1000$ бросков. Если гипотеза об уравновешенности монеты является истинной, то X должна быть приближенно нормально распределена со средним значением, равным 500, и стандартным отклонением 15.8:

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

Нам нужно принять решение о *значимости*, т. е. насколько мы готовы совершить *ошибку 1-го рода* ("ложное утверждение"), при которой мы отклоняем H_0, даже если она является истинной. По причинам, затерявшимся в анналах истории, эта готовность часто задается на уровне 5% или 1%. Возьмем за основу 5%.

Рассмотрим проверку, которая отклоняет H_0, если X попадает за пределы границ, заданных следующим образом:

```
# (469, 531)
lower_bound, upper_bound = normal_two_sided_bounds(0.95, mu_0, sigma_0)
```

Допустив, что p действительно равно 0.5 (т. е. H_0 является истинной), существует всего 5%-ный шанс, что мы наблюдаем случайную величину X, которая лежит за пределами этого интервала, и это в точности соответствует тому уровню значимости, который мы хотели. Выражаясь иначе, если H_0 является истинной, то приближенно в 19 случаев из 20 такая проверка будет давать правильный результат.

Нередко мы также заинтересованы в *мощности* проверки, т. е. вероятности не совершить *ошибку 2-го рода* ("ложное отрицание"), когда нам не удается отклонить H_0, даже если она является ложной. Для того чтобы это измерить, следует конкретизировать, что́ в точности *означает* ложность гипотезы H_0. (Знание о том, что $p \neq 0.5$, не дает нам массы информации о распределении X.) В частности, давайте проверим, что произойдет, если на самом деле $p = 0.55$, вследствие чего монета слегка смещена в сторону орлов.

В этом случае мощность проверки можно вычислить так:

```
# 95%-ные границы, основыванные на допущении, что p равно 0.5
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)

# Фактические mu и sigma, основыванные на p, равном 0.55
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)

# Ошибка 2-го рода означает, что нам не удалось отклонить нулевую гипотезу,
# что произойдет, когда X все еще внутри нашего исходного интервала
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability     # 0.887
```

Вместо этого допустим, что наша нулевая гипотеза состояла в том, что монета не смещена в сторону орлов, или что $p \leq 0.5$. В этом случае нам нужна *односторонняя* проверка, которая отклоняет нулевую гипотезу, когда X больше 500, и не отклоняет, когда X меньше 500. Поэтому проверка на 5%-ном уровне значимости

предусматривает применение функции `normal_probability_below` для отыскания точки отсечения, ниже которой лежит 95% вероятности:

```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# равно 526 (< 531, т. к. нам нужно больше вероятности в верхнем хвосте)

type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
power = 1 - type_2_probability        # 0.936
```

Эта проверка мощнее, поскольку она больше не отклоняет H_0, когда X ниже 469 (что вряд ли вообще произойдет, если альтернативная гипотеза H_1 является истинной), и вместо этого отклоняет нулевую гипотезу H_0, когда X лежит между 526 и 531 (что вполне может произойти, если альтернативная гипотеза H_1 является истинной).

P-значения

Альтернативный подход к приведенной выше проверке предусматривает применение *p-значения*[3]. Вместо выбора границ на основе некоторой точки отсечения вероятности вычисляется вероятность (при условии, что H_0 является истинной) получения как минимум такого же предельного значения, как и то, которое фактически наблюдалось.

Для нашей двусторонней проверки, является ли монета уравновешенной, мы вычисляем:

```
# Двустороннее p-значение
def two_sided_p_value(x: float, mu: float = 0, sigma: float = 1) -> float:
    """
    Насколько правдоподобно увидеть значение, как минимум, такое же
    предельное, что и x (в любом направлении), если наши значения
    поступают из N(mu, sigma)?
    """
    if x >= mu:
        # x больше, чем среднее, поэтому хвост везде больше, чем x
        return 2 * normal_probability_above(x, mu, sigma)
    else:
        # x меньше, чем среднее, поэтому хвост везде меньше, чем x
        return 2 * normal_probability_below(x, mu, sigma)
```

Если бы мы увидели 530 орлов, то мы бы вычислили:

```
two_sided_p_value(529.5, mu_0, sigma_0)   # 0.062
```

[3] Фактически *p*-значение — это вероятность ошибки при отклонении нулевой гипотезы (ошибки 1-го рода). — *Прим. пер.*

Почему мы использовали значение 529,5, а не 530? Все дело в поправке на непрерывность[4]. Она отражает тот факт, что вызов функции `normal_probability_between(529.5, 530.5, mu_0, sigma_0)` является более точной оценкой вероятности наблюдать 530 орлов, чем `normal_probability_between(530, 531, mu_0, sigma_0)`.

Поэтому вызов функции `normal_probability_above(529.5, mu_0, sigma_0)` является более точной оценкой вероятности наблюдать как минимум 530 орлов. Отметим, что такой же прием использован в примере, результат выполнения которого показан на рис. 6.4.

Одним из способов убедиться, что этот результат является разумной оценкой, — провести симуляцию:

```
import random

extreme_value_count = 0
for _ in range(1000):
    num_heads = sum(1 if random.random() < 0.5 else 0   # Подсчитать число орлов
                    for _ in range(1000))               # в 1000 бросках
    if num_heads >= 530 or num_heads <= 470:            # и как часто это число
        extreme_value_count += 1                        # 'предельное'

# p-значение было 0.062 => ~62 предельных значений из 1000
assert 59 < extreme_value_count < 65, f"{extreme_value_count}"
```

Поскольку *p*-значение превышает заданный 5%-ный уровень значимости, то нулевая гипотеза не отклоняется. И напротив, при выпадении 532 орлов *p*-значение будет равно:

```
two_sided_p_value(531.5, mu_0, sigma_0)      # 0.0463
```

что меньше 5%-ного уровня значимости, и, следовательно, нулевая гипотеза будет отклонена. Это в точности такая же проверка, что и прежде. Разница лишь в подходе к статистикам.

Верхнее и нижнее *p*-значения можно получить аналогичным образом:

```
upper_p_value = normal_probability_above
lower_p_value = normal_probability_below
```

Для односторонней проверки при выпадении 525 орлов мы вычисляем:

```
upper_p_value(524.5, mu_0, sigma_0)          # 0.061
```

и, следовательно, нулевая гипотеза не будет отклонена. При выпадении 527 орлов вычисление будет:

```
upper_p_value(526.5, mu_0, sigma_0)          # 0.047
```

и нулевая гипотеза будет отклонена.

[4] См. **https://en.wikipedia.org/wiki/Continuity_correction**. — *Прим. пер.*

Следует удостовериться, что ваши данные являются приближенно нормально распределенными, и только после этого можно применять функцию `normal_probability_above` для вычисления *p*-значений. Анналы плохой науки о данных полны примеров людей, считающих, что шанс наступления некоего наблюдаемого события случайным образом равен одному на миллион, когда на самом деле они имеют в виду "шанс с учетом того, что данные являются нормально распределенными", что достаточно бессмысленно, если данные такими не являются.

Существуют разнообразные проверки на нормальность, но если просто вывести данные на график, то это будет шагом в правильном направлении.

Доверительные интервалы

Мы проверяли гипотезы о значении вероятности орлов *p*, т. е. *параметре* неизвестного распределения "орлов". В этом случае используют третий подход — строят *доверительный интервал* вокруг наблюдаемого значения параметра.

Например, мы можем оценить вероятность неуравновешенной монеты, обратившись к среднему значению бернуллиевых величин, соответствующих каждому броску монеты — 1 (орел) и 0 (решка). Если мы наблюдаем 525 орлов из 1000 бросков, то мы оцениваем, что *p* равно 0.525.

Насколько можно быть уверенными в этой оценке? Дело в том, что если бы имелось точное значение *p*, то согласно центральной предельной теореме (*см. разд. "Центральная предельная теорема" главы 6*) среднее этих бернуллиевых величин должно быть приближенно нормальным со средним *p* и стандартным отклонением:

```
math.sqrt(p * (1 - p) / 1000)
```

Здесь мы не знаем *p*, и поэтому вместо него мы используем оценку:

```
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000)    # 0.0158
```

Это не совсем оправданно, и тем не менее все равно поступают именно так. Используя нормальную аппроксимацию, мы делаем вывод, что с "уверенностью на 95%" следующий ниже интервал содержит истинный параметр *p*:

```
normal_two_sided_bounds(0.95, mu, sigma)          # [0.4940, 0.5560]
```

Это утверждение касается *интервала*, а не *p*. Следует понимать его как утверждение, что если бы пришлось повторять эксперимент много раз, то в 95% случаев "истинный" параметр (который каждый раз одинаков) будет лежать в пределах наблюдаемого доверительного интервала (который каждый раз может быть разным).

В частности, мы не делаем заключения о том, что монета не уравновешена, поскольку 0.5 попадает в пределы доверительного интервала.

И напротив, если бы выпало 540 орлов, то мы имели бы:

```
p_hat = 540 / 1000
mu = p_hat
```

```
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000)  # 0.0158
normal_two_sided_bounds(0.95, mu, sigma)       # [0.5091, 0.5709]
```

Здесь "уравновешенная монета" не лежит в доверительном интервале. (Гипотеза об уравновешенной монете не проходит проверки, которую, как ожидалось, она должна проходить в 95% случаев, если бы она была истинной.)

Взлом *p*-значения

Процедура, которая отклоняет нулевую гипотезу только в 5% случаев — по определению, — в 5% случаев будет отклонять нулевую гипотезу ошибочно:

```
from typing import List

def run_experiment() -> List[bool]:
    """Подбрасывает уравновешенную монету 1000 раз,
       Истина = орлы, Ложь = решки"""
    return [random.random() < 0.5 for _ in range(1000)]

def reject_fairness(experiment: List[bool]) -> bool:
    """Использование 5%-ных уровней значимости"""
    num_heads = len([flip for flip in experiment if flip])
    return num_heads < 469 or num_heads > 531

random.seed(0)
experiments = [run_experiment() for _ in range(1000)]

num_rejections = len([experiment
                      for experiment in experiments
                      if reject_fairness(experiment)])

assert num_rejections == 46
```

А это означает, что если задаться целью найти "значимые" результаты, то их обязательно найдешь. Проверь достаточное число гипотез относительно данных, и одна из них почти наверняка покажется значимой. Удали правильные выбросы, и в итоге вы вполне можете получить *p*-значение ниже 0.05. (Нечто отдаленно похожее было сделано в *разд. "Корреляция" главы 5*. Заметили?)

Это то, что называется взломом *p*-значения[5] (*p*-hacking) и в некоторой степени является следствием "вывода в рамках *p*-значений". По этой теме имеется замечательная статья "Земля круглая"[6], в которой критикуется такой подход.

Если вы хотите заниматься "хорошей" наукой о данных, то вам следует формулировать свои гипотезы до обращения к данным, вы должны очищать данные, не держа в уме гипотезы, и помнить, что *p*-значения — это не заменители здравого

[5] См. **http://www.nature.com/news/scientific-method-statistical-errors-1.14700.**

[6] См. **http://ist-socrates.berkeley.edu/~maccoun/PP279_Cohen1.pdf.**

смысла. (Альтернативный подход рассмотрен в *разд. "Байесов вывод" далее в этой главе*.)

Пример: проведение *A/B*-тестирования

Одна из ваших первостепенных обязанностей в DataSciencester — заниматься оптимизацией опыта взаимодействия. Под этим эвфемизмом скрываются усилия заставить пользователей щелкать на рекламных объявлениях. Один из рекламных агентов разработал рекламу нового энергетического напитка, предназначенного для исследователей данных, и директору отдела по рекламе нужна ваша помощь с выбором между двумя рекламными объявлениями: *A* ("Вкус отличный!") и *B* ("Меньше предвзятости!").

Будучи *исследователем*, вы решаете провести эксперимент, случайно показывая посетителям веб-сайта одно из двух рекламных сообщений, при этом отслеживая число нажатий на каждом из них.

Если из 1000 потребителей рекламы *A* ее выбрали 990 человек, а из 1000 потребителей рекламы *B* ее выбрали только 10 человек, то можно быть вполне уверенным, что реклама *A* лучше. Но что делать, если разница не такая большая? Как раз здесь и понадобится статистический вывод.

Скажем, N_A людей видят рекламу *A* и n_A из них нажимают на ней. Каждый просмотр рекламы можно представить в виде бернуллиева испытания, где p_A — это вероятность, что кто-то нажмет на рекламе *A*. Тогда (если N_A большое, что так и есть в данном случае) мы знаем, что n_A/N_A — это приближенно нормальная случайная величина со средним значением p_A и стандартным отклонением $\sigma_A = \sqrt{p_A(1-p_A)/N_A}$.

Схожим образом n_B/N_B — приближенно нормальная случайная величина со средним значением p_B и стандартным отклонением $\sigma_B = \sqrt{p_B(1-p_B)/N_B}$.

```
# Оценочные параметры
def estimated_parameters(N: int, n: int) -> Tuple[float, float]:
    p = n / N
    sigma = math.sqrt(p * (1 - p) / N)
    return p, sigma
```

Если допустить, что эти две нормальные случайные величины взаимно независимы (что кажется разумным, поскольку отдельные бернуллиевы испытания обязаны быть такими), то их разность тоже должна быть нормальной со средним значением $p_B - p_A$ и стандартным отклонением $\sqrt{\sigma_A^2 + \sigma_B^2}$.

Если быть точным, то математика сработает, только если стандартные отклонения *известны*. Здесь же речь идет об их оценках, исходя из выборочных данных, и, следовательно, на самом деле надо использовать *t*-распределение. Тем не менее для достаточно крупных наборов данных их значения настолько близки, что уже нет особой разницы.

Поэтому можно проверить *нулевую гипотезу* о том, что p_A и p_B являются одинаковыми (т. е. $p_A - p_B = 0$), используя следующую статистику:

```
def a_b_test_statistic(N_A: int, n_A: int, N_B: int, n_B: int) -> float:
    p_A, sigma_A = estimated_parameters(N_A, n_A)
    p_B, sigma_B = estimated_parameters(N_B, n_B)
    return (p_B - p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

которая должна быть приближенно стандартной нормальной.

Например, если реклама "Вкус отличный!" получает 200 кликов из 1000 просмотров, а реклама "Меньше предвзятости!" — 180 кликов из такого же числа, то статистика равна:

```
z = a_b_test_statistic(1000, 200, 1000, 180)   # -1.14
```

Вероятность наблюдать такую большую разницу, если средние значения были бы фактически одинаковыми, будет:

```
two_sided_p_value(z)                           # 0.254
```

которая большая настолько, что мы можем сделать вывод о наличии большой разницы. С другой стороны, если реклама "Меньше предвзятости!" получит только 150 откликов, то мы получим:

```
z = a_b_test_statistic(1000, 200, 1000, 150)   # -2.94
two_sided_p_value(z)                           # 0.003
```

и, следовательно, вероятность наблюдать такую большую разницу при одинаково эффективных рекламных объявлениях равна всего `0.003`.

Байесов вывод

Рассмотренные выше процедуры предусматривали выдвижение вероятностных утверждений в отношении *проверок* статистических гипотез типа "если нулевая гипотеза является истинной, то шанс обнаружить такую-то предельную (экстремальную) статистику равен всего 3%".

Альтернативный подход к статистическому выводу предусматривает трактовку самих неизвестных параметров как случайных величин. Аналитик (это вы) начинает с *априорного распределения* для параметров и затем использует наблюдаемые данные и теорему Байеса для получения обновленного *апостериорного распределения* для этих параметров. Вместо вероятностных суждений о проверках делаются вероятностные суждения о самих параметрах.

Например, если неизвестным параметром является вероятность (как в примере с бросанием монеты), то часто априорное распределение вероятности берут из *бета-распределения*[7], которое размещает всю свою вероятность между 0 и 1:

[7] Бета-распределение — двухпараметрическое (обычно с произвольными фиксированными параметрами α и β) непрерывное распределение. Используется для описания случайных величин, значения которых ограничены конечным интервалом (см. **https://ru.wikipedia.org/wiki/Бета-распределение**). — *Прим. пер.*

```
def B(alpha: float, beta: float) -> float:
    """Нормализующая константа, чтобы полная вероятность
       в сумме составляла 1"""
    return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)

def beta_pdf(x: float, alpha: float, beta: float) -> float:
    if x <= 0 or x >= 1:  # за пределами [0, 1] нет веса
        return 0
    return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(alpha, beta)
```

Вообще говоря, это распределение концентрирует свой вес в:

```
alpha / (alpha + beta)
```

и чем больше значение параметров `alpha` и `beta`, тем "плотнее" распределение.

Например, если и `alpha`, и `beta` оба равны 1, то это равномерное распределение (с центром в 0.5 и очень разбросанное). Если `alpha` намного больше `beta`, то бóльшая часть веса находится около 1, и если `alpha` намного меньшем `beta`, то бóльшая часть веса находится около 0. На рис. 7.1 показано несколько различных бета-распределений.

Скажем, мы допустили априорное распределение для p. Возможно, мы не настаиваем на том, что монета является уравновешенной и решаем, что оба параметра `alpha` и `beta` равны 1. Либо, возможно, мы абсолютно убеждены в том, что монета выпадает орлом в 55% случаев, и потому решаем, что `alpha` равен 55, `beta` равен 45.

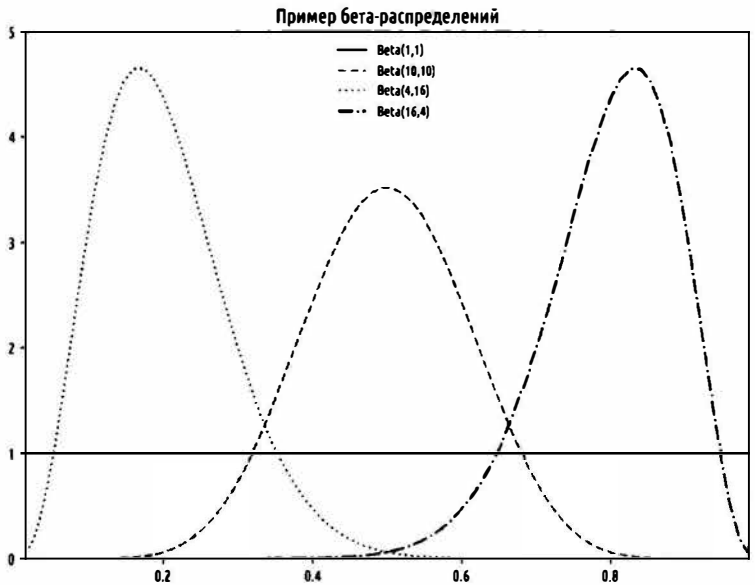


Рис. 7.1. Пример бета-распределений

Затем мы подбрасываем монету n раз и подсчитываем h орлов и t решек. Согласно теореме Байеса (и некоторым математикам, которых было бы утомительно тут перечислять) апостериорное распределение для p снова будет бета-распределением, но с параметрами `alpha + h` и `beta + t`.

Совсем не случайно, что апостериорное распределение снова подчиняется бета-распределению. Число орлов задано биномиальным распределением, а бета-распределение — это априорное распределение, сопряженное с биномиальным[8]. И поэтому, когда априорное бета-распределение обновляется, используя наблюдения из соответствующего биномиального, то в итоге получается апостериорное бета-распределение.

Скажем, вы подбрасываете монету 10 раз и видите выпадение только 3 орлов.

Если бы вы начали с равномерного априорного распределения (в некотором смысле, отказываясь отстаивать уравновешенность монеты), то апостериорное распределение было бы `beta(4,8)` с центром вокруг 0.33. Поскольку все вероятности рассматриваются равновозможными, то ваша наилучшая догадка будет находиться где-то поблизости от наблюдаемой вероятности.

Если бы вы начали с `beta(20,20)` (выражая уверенность, что монета является примерно уравновешеной), то апостериорное распределение было бы `beta(23,27)` с центром вокруг 0.46, свидетельствуя о пересмотре степени уверенности в сторону того, что, возможно, результаты подбрасывания монеты слегка смещены в сторону решек.

А если бы вы начали с `beta(30,10)` (выражая уверенность, что результаты подбрасывания монеты смещены в сторону выпадения орлов в 75% случаев), то апостериорное распределение было бы `beta(33,17)` с центром вокруг 0.66. В этом случае вы бы по-прежнему верили в смещенность в сторону орлов, но уже не так сильно, как первоначально. Эти три апостериорных распределения показаны на рис. 7.2.

Если бы вы продолжили подбросывать монету, то априорное распределение значило бы все меньше и меньше, пока в итоге у вас не получилось бы (почти) одинаковое апостериорное распределение, неважно, с какого априорного распределения оно начиналось.

Например, не имеет значения, насколько, по вашей первоначальной мысли, монета была смещена, поскольку будет трудно поддерживать эту уверенность после выпадения 1000 орлов из 2000 бросков (если, конечно, не быть экстремалом, который в качестве априорного бета-распределения решает выбрать, скажем, `beta(1000000,1)`).

Интересно, что на основе байесова вывода можно делать вероятностные утверждения о гипотезах, типа: "на основе априорного распределения и наблюдаемых данных имеется 5%-ное правдоподобие, что вероятность орлов монеты находится между 49 и 51%". С философской точки зрения такое утверждение сильно отличается от утверждений типа "если бы монета была уравновешенной, то мы могли бы ожидать, что будем наблюдать такие же предельные данные только в 5% случаев".

---

[8] См. **http://www.johndcook.com/blog/conjugate_prior_diagram/**.

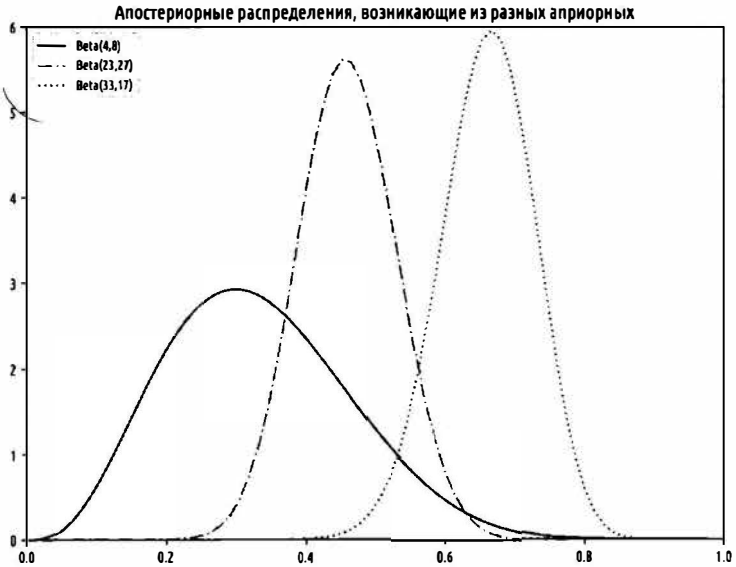


Рис. 7.2. Апостериорные распределения, возникающие из разных априорных

Применение байесова вывода для проверки статистических гипотез считается несколько противоречивым решением, частично потому, что его математический аппарат может становиться весьма запутанным, и частично из-за субъективной природы выбора априорного распределения. Мы больше не будем пользоваться им в этой книге, но знать о нем стоит.

Для дальнейшего изучения

- В данной главе мы лишь прикоснулись к тому, что вам необходимо знать о статистическом выводе. В книгах, рекомендованных для прочтения в конце *главы 5*, данная тема обсуждается гораздо глубже.
- Образовательная онлайновая платформа Coursera[9] предлагает курс по анализу данных и статистическому выводу[10], который охватывает многие из упомянутых тем.

[9] Coursera — проект в сфере массового онлайн-образования, основанный профессорами информатики Стэндфордского университета США, предлагающий бесплатные онлайн-курсы от специалистов со всего мира на разных языках. — *Прим. пер.*

[10] См. **https://www.coursera.org/course/statistics**.

ГЛАВА 8
Градиентный спуск

> Те, кто бахвалятся своим происхождением, хвастаются тем, что они задолжали другим.
> — *Сенека*[1]

Нередко, занимаясь наукой о данных, мы пытаемся отыскать модель, наилучшую для конкретной ситуации. При этом обычно под "наилучшей" подразумевается модель, которая "минимизирует ошибку ее предсказаний" либо "максимизирует правдоподобие данных". Другими словами, она будет представлять решение некой оптимизационной задачи.

Следовательно, нам придется решать ряд оптимизационных задач. И в частности, нам придется решать их с нуля. Наш подход будет основываться на техническом решении, именуемом *градиентным спуском*, который вполне укладывается в трактовку с нуля. Вы, возможно, не посчитаете его самим по себе каким-то суперзахватывающим, однако оно даст нам возможность заниматься увлекательными вещами на протяжении всей оставшейся книги. Так что потерпите.

Идея в основе градиентного спуска

Пусть имеется некая функция `f`, которая на входе принимает вектор из вещественных чисел и на выходе выдает одно-единственное вещественное число. Вот одна из таких простых функций:

```
from scratch.linear_algebra import Vector, dot

def sum_of_squares(v: Vector) -> float:
    """Вычисляет сумму возведенных в квадрат элементов в v"""
    return dot(v, v)
```

Нередко нам необходимо максимизировать или минимизировать такие функции. Другими словами, нам нужно отыскивать вход `v`, который производит наибольшее (или наименьшее) возможное значение.

Для таких функций, как наша, *градиент* (если вы помните дифференциальное исчисление, то это вектор, состоящий из частных производных) задает для входящего

[1] Луций Анней Сенека (IV в. до н. э.) — римский философ-стоик, поэт и государственный деятель. Здесь обыгрывается слово *descent*, которое можно понять, как происхождение и как спуск. — *Прим. пер.*

аргумента направление, в котором функция возврастает быстрее всего (если вы не помните дифференциальное исчисление, то просто поверьте мне на слово либо загляните в Интернет[2]).

Соответственно, один из подходов к максимизации функции состоит в том, чтобы подобрать случайную отправную точку, вычислить градиент, сделать небольшой шаг в направлении градиента (т. е. в направлении, которое побуждает функцию расти быстрее всего) и повторить итерацию с новой отправной точки. Схожим образом можно попытаться минимизировать функцию, делая небольшие шаги в *противоположном* направлении, как показано на рис. 8.1.

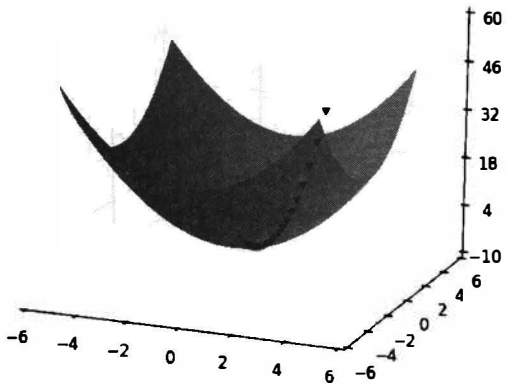


Рис. 8.1. Отыскание минимума с помощью градиентного спуска

Если функция имеет уникальный глобальный минимум, то этот алгоритм, вероятно, его отыщет. Если же (локальных) минимумов много, то он может "найти" какой-то неправильный из них, и в этом случае можно повторить процедуру, начиная с других отправных точек. Если функция не имеет минимума, то существует возможность, что эта процедура войдет в бесконечный цикл.

Оценивание градиента

Если `f` — это функция одной переменной, то ее производная в точке `x` служит мерой того, как `f(x)` изменяется, когда мы делаем очень малое изменение в `x`. Производная определяется как предел разностных отношений:

```
from typing import Callable

def difference_quotient(f: Callable[[float], float],
                        x: float,
                        h: float) -> float:
    return (f(x + h) - f(x)) / h
```

при стремлении `h` к нулю.

[2] См. **https://ru.wikipedia.org/wiki/Градиент**. — *Прим. пер.*

(Многие из тех, кто занимается дифференциальным исчислением, будут озадачены таким математическим определением предела, которое выглядит красиво, но кажется запретительным. Здесь мы слукавим и просто скажем, что "предел" означает именно то, что вы думаете.)

Производная — это наклон касательной в `(x, f(x))`, в то время как разностное отношение — это наклон "не совсем" касательной, которая проходит через `(x + h, f(x + h))`. По мере уменьшения `h` "не совсем" касательная становится все ближе и ближе к касательной (рис. 8.2).

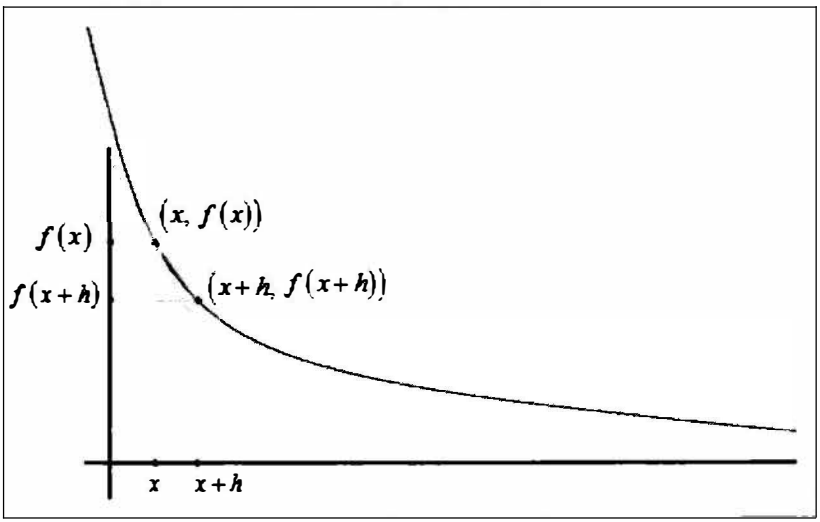


Рис. 8.2. Аппроксимация производной при помощи разностного отношения

Для многих функций достаточно легко выполнить точное вычисление производных. Например, функция возведения в степень `square`:

```
def square(x: float) -> float:
    return x * x
```

имеет производную:

```
def derivative(x: float) -> float:
    return 2 * x
```

в чем, если есть желание, можно убедиться, вычислив разностное отношение явным образом и взяв предел. (Для этого потребуется алгебра на уровне общеобразовательной школы.)

Что, если вы не смогли (или не захотели) отыскать градиент? Хотя мы не можем брать пределы на Python, мы можем оценить производные, вычислив разностное отношение для сколь угодно малого числа `e`. На рис. 8.3 показаны результаты одного такого оценивания:

```
xs = range(-10, 11)
actuals = [derivative(x) for x in xs]
estimates = [difference_quotient(square, x, h=0.001) for x in xs]
```

```
# Построить график, чтобы показать, что они в сущности одинаковые
import matplotlib.pyplot as plt
plt.title("Фактические производные и их оценки")
plt.plot(xs, actuals, 'rx', label='Actual')      # красный x
plt.plot(xs, estimates, 'b+', label='Estimate') # синий +
plt.legend(loc=9)
plt.show()
```

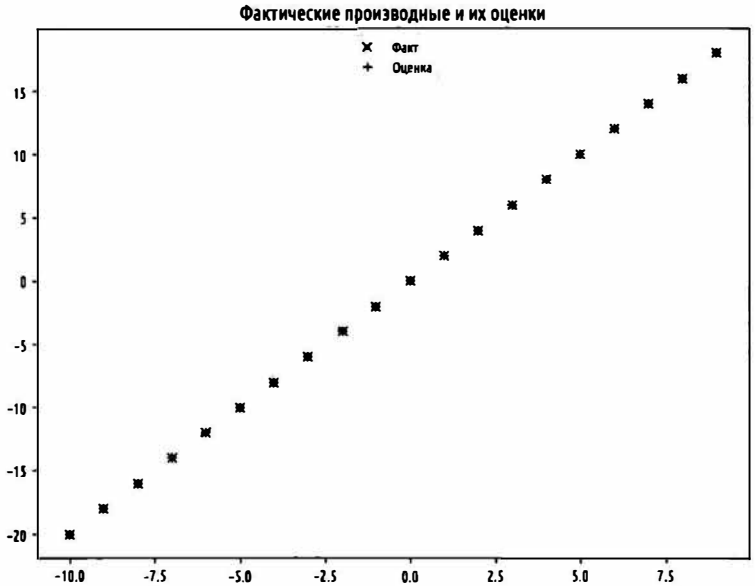


Рис. 8.3. Качество аппроксимации разностным отношением

Если `f` — это функция многих переменных, то она имеет многочисленные *частные производные*, каждая из которых показывает скорость изменения `f` при незначительном изменении лишь в одной из входных переменных.

Мы вычисляем ее *i*-ю частную производную, трактуя ее как функцию только одной *i*-й ее переменной, оставляя прочие переменные фиксированными:

```
# Частное разностное отношение
def partial_difference_quotient(f: Callable[[Vector], float],
                                v: Vector,
                                i: int,
                                h: float) -> float:
    """Возвращает i-е частное разностное отношение функции f в v"""
    w = [v_j + (h if j == i else 0)  # Добавить h только в i-й элемент v
         for j, v_j in enumerate(v)]

    return (f(w) - f(v)) / h
```

после чего можно таким же образом вычислить градиент:

```
def estimate_gradient(f: Callable[[Vector], float],
                      v: Vector,
                      h: float = 0.0001):
    return [partial_difference_quotient(f, v, i, h)
            for i in range(len(v))]
```

Главный недостаток оценки при помощи разностного отношения заключается в том, что она является вычислительно ресурсоемкой. Если `v` имеет длину *n*, то функция `estimate_gradient` должна вычислить `f` для *2n* разных входов. Если вы вычисляете градиенты многократно, то выполняете большой объем избыточной работы. Во всем, что мы делаем, мы будем использовать математику, вычисляя градиентные функции явным образом.

Использование градиента

Легко убедиться, что функция `sum_of_squares` является минимальной, когда ее вход `v` состоит из нулей. Притворимся, что мы не знаем об этом. Мы будем использовать градиенты для отыскания минимума среди всех трехмерных векторов. Выберем случайную отправную точку и крошечными шажками начнем двигаться в направлении, противоположном градиенту, до тех пор, пока не достигнем точки, где градиент будет очень мал:

```
import random
from scratch.linear_algebra import distance, add, scalar_multiply

def gradient_step(v: Vector, gradient: Vector, step_size: float) -> Vector:
    """Движется с шагом `step_size` в направлении
       градиента `gradient` от `v`"""
    assert len(v) == len(gradient)
    step = scalar_multiply(step_size, gradient)
    return add(v, step)

def sum_of_squares_gradient(v: Vector) -> Vector:
    return [2 * v_i for v_i in v]

# Подобрать случайную отправную точку
v = [random.uniform(-10, 10) for i in range(3)]

for epoch in range(1000):
    grad = sum_of_squares_gradient(v) # Вычислить градиент в v
    v = gradient_step(v, grad, -0.01) # Сделать отрицательный
                                      # градиентный шаг
    print(epoch, v)

assert distance(v, [0, 0, 0]) < 0.001 # v должен быть близким к 0
```

Если вы выполните этот фрагмент кода, то обнаружите, что он всегда завершается с вектором `v`, очень близким к `[0,0,0]`. Чем больше эпох его выполнять, тем ближе он будет к нулевому вектору.

Выбор правильного размера шага

Если причина движения против градиента понятна, то насколько далеко двигаться — не совсем. На самом деле выбор оптимального размера шага больше походит на искусство, чем на науку. Популярные варианты включают:

- использование постоянного размера шага;
- постепенное сжатие шага во времени;
- на каждом шаге выбор размера шага, который минимизирует значение целевой функции.

Последний вариант выглядит предпочтительнее, но на практике тоже требует дорогостоящих вычислений. Не усложняя вещи, в большинстве случаев мы будем пользоваться фиксированным размером шага. Размер шага, который "работает", зависит от задачи: слишком малый — и ваш градиентный спуск будет продолжаться вечно; слишком большой — и вы будете делать гигантские шаги, которые побудят интересующую вас функцию становиться больше или даже неопределенной. Поэтому придется экспериментировать.

Применение градиентного спуска для подгонки моделей

В этой книге мы будем использовать градиентный спуск для подгонки параметризованных моделей к данным. В обычном случае у нас будет некий набор данных и некая (гипотетическая) модель данных, которая зависит (дифференцируемым образом) от одного или нескольких параметров. У нас также будет функция *потери*, которая измеряет, насколько хорошо модель вписывается в наши данные. (Чем меньше потеря, тем лучше.)

Если мы думаем о наших данных как о фиксированных, то наша функция потери сообщает нам о том, насколько хороши или плохи те или иные параметры модели. Это означает, что мы можем использовать градиентный спуск для отыскания параметров модели, которые делают потерю как можно меньшей. Давайте рассмотрим простой пример:

```
# x изменяется в интервале от -50 до 49, y всегда равно 20 * x + 5
inputs = [(x, 20 * x + 5) for x in range(-50, 50)]
```

В этом случае мы знаем параметры линейной зависимости между `x` и `y`, но представьте, что мы хотели бы усвоить их из данных. Мы будем использовать градиентный спуск для отыскания углового коэффициента (наклона) и коэффициента сдвига (пересечения), которые минимизируют среднюю квадратическую ошибку.

Начнем с функции, которая определяет градиент на основе ошибки из одной точки данных:

```
# Линейный градиент
def linear_gradient(x: float, y: float, theta: Vector) -> Vector:
    slope, intercept = theta          # Наклон и пересечение
    predicted = slope * x + intercept # Модельное предсказание
    error = (predicted - y)           # Ошибка равна (предсказание - факт)
    squared_error = error ** 2        # Мы минимизируем квадрат ошибки,
    grad = [2 * error * x, 2 * error] # используя ее градиент
    return grad
```

Давайте подумаем, что́ означает этот градиент. Представьте, что для некоторого `x` наше предсказание является слишком большим. В этом случае ошибка `error` является положительной. Второй градиентный член, `2 * error`, является положительным, что отражает тот факт, что небольшое увеличение в пересечении будет увеличивать (уже слишком большое) предсказание еще больше, что приведет к тому, что квадратическая ошибка (для этого `x`) будет становиться еще больше.

Первый градиентный член, `2 * error * x`, имеет тот же знак, что и `x`. Конечно, если `x` является положительным, то небольшое увеличение в наклоне снова сделает предсказание (и, следовательно, ошибку) крупнее. Однако, если `x` является отрицательным, то небольшое увеличение в наклоне сделает предсказание (и, следовательно, ошибку) меньше.

Далее, это вычисление было для одной-единственной точки данных. Для всего набора данных мы рассмотрим *среднеквадратическую ошибку*. И градиент среднеквадратической ошибки — это всего-навсего среднее значение индивидуальных градиентов.

Итак, вот что мы намереваемся сделать:

1. Начать со случайного значения для `theta`.
2. Вычислить среднее значение градиентов.
3. Скорректировать `theta` в этом направлении.
4. Повторить.

После многочисленных эпох (эпохой мы называем каждое прохождение по набору данных) мы должны усвоить что-то вроде правильных параметров:

```
from scratch.linear_algebra import vector_mean

# Начать со случайных значений наклона и пересечения
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

learning_rate = 0.001       # Темп усвоения

for epoch in range(5000):
    # Вычислить средне значение градиентов
    grad = vector_mean([linear_gradient(x, y, theta) for x, y in inputs])
```

```
    # Сделать шаг в этом направлении
    theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1, "наклон должен быть равным примерно 20"
assert 4.9 < intercept < 5.1, "пересечение должно быть равным примерно 5"
```

Мини-пакетный и стохастический градиентный спуск

Один из недостатков предыдущего подхода заключается в том, что нам пришлось оценивать градиенты по всему набору данных и только потом делать градиентный шаг и обновлять наши параметры. В данном случае это было хорошо, потому что наш набор данных состоял всего из 100 пар, и вычисление градиента было дешевым.

Однако ваши модели часто будут иметь большие наборы данных и дорогостоящие градиентные вычисления. В таком случае вы захотите делать градиентные шаги чаще.

Мы можем сделать это, используя техническое решение под названием *мини-пакетный* градиентный спуск, в котором мы вычисляем градиент (и делаем градиентный шаг) на основе "мини-пакета", отбираемого из более крупного набора данных:

```
from typing import TypeVar, List, Iterator

T = TypeVar('T') # Это позволяет типизировать "обобщенные" функции

def minibatches(dataset: List[T],
                batch_size: int,
                shuffle: bool = True) -> Iterator[List[T]]:
    """Генерирует мини-пакеты в размере `batch_size` из набора данных"""
    # start индексируется с 0, batch_size, 2 * batch_size, ...
    batch_starts = [start for start in range(0, len(dataset), batch_size)]

    if shuffle: random.shuffle(batch_starts) # Перетасовать пакеты

    for start in batch_starts:
        end = start + batch_size
        yield dataset[start:end]
```

Конструкция `TypeVar(T)` позволяет создавать "обобщенную" функцию. Она говорит, что наш набор данных может быть списком любого типа — `str`, `int`, `list` и др., но независимо от типа выходы будут пакетами.

Мы можем решить нашу задачу снова, теперь с мини-пакетами:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

for epoch in range(1000):
    for batch in minibatches(inputs, batch_size=20):
        grad = vector_mean([linear_gradient(x, y, theta) for x, y in batch])
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1, "наклон должен быть равным примерно 20"
assert 4.9 < intercept < 5.1, "пересечение должно быть равным примерно 5"
```

Еще одним вариантом является *стохастичекий* градиентный спуск, в котором градиентные шаги делаются на основе одного тренировочного примера за раз:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

for epoch in range(100):
    for x, y in inputs:
        grad = linear_gradient(x, y, theta)
        theta = gradient_step(theta, grad, -learning_rate)
        print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1, "наклон должен быть равным примерно 20"
assert 4.9 < intercept < 5.1, "пересечение должно быть равным примерно 5"
```

В этой задаче стохастический градиентный спуск находит оптимальные параметры за гораздо меньшее число эпох. Но всегда есть компромиссы. Основывая градиентные шаги на небольших мини-пакетах (или на отдельных точках данных), вы можете взять большее их число, но градиент для одной-единственной точки может лежать в совершенно другом направлении от градиента для набора данных в целом.

Кроме того, если бы мы не занимались линейной алгеброй с нуля, то вместо вычисления градиента в одной точке за раз можно было бы получить прирост производительности от "векторизации" вычислений по всем пакетам.

На протяжении всей книги мы будем экспериментировать, отыскивая оптимальные размеры пакетов и размеры шагов.

Терминология для различных видов градиентного спуска характерна неоднородностью. Подход "вычислить градиент для всего набора данных" часто называют пакетным градиентным спуском, при этом некоторые люди используют термин "стохастический градиентный спуск", ссылаясь на мини-пактную версию (среди которых версия с одной точкой за один раз является частным случаем).

Для дальнейшего изучения

- Продолжайте читать! Мы будем пользоваться градиентным спуском для решения задач в остальной части книги.
- На данный момент некоторые, несомненно, уже почувствовали усталость от моих советов по поводу чтения учебников. Чтобы как-то утешить, вот книга "Active Calculus" ("Активный математический анализ")[3], которая, по-видимому, будет попонятнее учебников по математическому анализу, на которых учился я.
- Себастьян Рубер (Sebastian Ruder) написал эпический пост[4], в котором он сравнивает градиентный спуск и его многочисленные варианты.

[3] См. **https://scholarworks.gvsu.edu/books/10/**.

[4] См. **http://ruder.io/optimizing-gradient-descent/index.html**.

ГЛАВА 9
Получение данных

> Писал я ее ровно три месяца, обдумывал ее содержание три минуты, а материал для нее собирал всю жизнь.
>
> – *Ф. Скотт Фицджеральд*[1]

Для того чтобы быть исследователем данных, прежде всего нужны сами данные. По сути дела, как исследователь данных вы будете тратить ошеломляюще огромную часть времени на сбор, очистку и преобразование данных. Естественно, в случае крайней необходимости всегда можно набрать данные вручную (или поручить это своим миньонам, если они есть), но обычно такая работа не лучшее применение собственному времени. В этой главе мы обратимся к различным способам подачи данных в Python и форматам данных.

Объекты *stdin* и *stdout*

Если вы выполняете сценарии Python из командной строки, то можете передавать через них данные по конвейеру, пользуясь для этого объектами `sys.stdin` и `sys.stdout`. Например, вот сценарий, который считывает строки текста и выдает обратно те, которые соответствуют регулярному выражению:

```
# egrep.py
import sys, re

# sys.argv - список аргументов командной строки
# sys.argv[0] - имя самой программы
# sys.argv[1] - регулярное выражение, указываемое в командной строке
regex = sys.argv[1]

# Для каждой строки, переданной сценарию
for line in sys.stdin:
    # если она соответствует регулярному выражению regex,
    # то записать ее в stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

[1] Фрэнсис Скотт Фицджеральд (1896–1940) — американский писатель, крупнейший представитель так называемого "потерянного поколения" в литературе — *Прим. пер.*

А этот сценарий подсчитывает количество полученных строк и печатает итоговый результат:

```
# подсчет строк (line_count.py)
import sys

count = 0
for line in sys.stdin:
    count += 1

# Печать выводится на консоль sys.stdout
print(count)
```

Этот сценарий в дальнейшем можно использовать для подсчета строк файла, в которых содержатся числа. В Windows это выглядит следующим образом:

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

В UNIX-подобной системе это выглядит несколько иначе:

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

Символ конвейерной передачи "|" означает, что надо "использовать выход команды слева, как вход для команды справа". Таким способом можно создавать достаточно подробные конвейеры обработки данных.

Если вы используете Windows, то эту команду можно применять без ссылки на Python:

```
type SomeFile.txt | egrep.py "[0-9]" | line_count.py
```

Чтобы сделать то же самое в UNIX-подобной системе, потребуется немного больше работы[2]. Сначала следует добавить директиву шебанг (с последовательностью из двух символов) в качестве первой строки своего сценария `#!/usr/bin/env python`. Затем в командной строке использовать команду `chmod x egrep.py++` для того, чтобы сделать файл исполняемым.

Вот аналогичный сценарий, который подсчитывает количества появлений слов в своем входном потоке и выписывает наиболее распространенные из них:

```
# Наиболее распространенные слова (most_common_words.py)
import sys
from collections import Counter

# Передать число слов в качестве первого аргумента
try:
  num_words = int(sys.argv[1])
except:
  print("Применение: most_common_words.py num_words")
  sys.exit(1)         # Ненулевой код выхода сигнализирует об ошибке
```

[2] См. **http://stackoverflow.com/questions/15587877/run-a-python-script-in-terminal-without-the-python-command.**

```
counter = Counter(word.lower()                  # Перевести в нижний регистр
                  for line in sys.stdin
                  for word in line.strip().split()  # Разбить строку
                                                    # по пробелам
                  if word)                      # Пропустить 'пустые' слова

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

Затем можно сделать что-то вроде этого:

```
$ cat the_bible.txt | python most_common_words.py 10
36397 the
30031 and
20163 of
7154 to
6484 in
5856 that
5421 he
5226 his
5060 unto
4297 shall
```

(Если вы используете Windows, то вместо команды `cat` используйте команду `type`.)

Опытные UNIX-программисты, несомненно, знакомы с широким спектром встроенных инструментов командной строки, таких как `egrep`, например, которые, вероятно, будут предпочтительнее собственных, созданных с нуля. Тем не менее полезно знать, что при необходимости всегда есть возможность их реализовать.

Чтение файлов

Вы можете явным образом считывать и записывать файлы непосредственно из кода. Python значительно упрощает работу с файлами.

Основы текстовых файлов

Первым шагом в работе с текстовым файлом является получение *файлового объекта* с помощью метода `open()`:

```
# 'r' означает только для чтения = read-only
file_for_reading = open('reading_file.txt', 'r')
file_for_reading2 = open('reading_file.txt')

# 'w' пишет в файл - сотрет файл, если он уже существует!
file_for_writing = open('writing_file.txt', 'w')
```

```
# 'a' дополняет файл - для добавления в конец файла
file_for_appending = open('appending_file.txt', 'a')

# Не забудьте закрыть файл в конце работы
file_for_writing.close()
```

Поскольку легко можно забыть закрыть свои файлы, следует всегда использовать их в блоке `with`, в конце которого они будут закрыты автоматически:

```
with open(filename,'r') as f:
    data = function_that_gets_data_from(f)

# в этой точке f уже был закрыт, поэтому не пытайтесь использовать его
process(data)
```

Если требуется прочитать текстовый файл целиком, то вы можете итеративно перебрать строки файла в цикле при помощи инструкции `for`:

```
starts_with_hash = 0

with open('input.txt','r') as f:
    for line in file:        # Промотреть каждую строку файла, используя
        if re.match("^#",line):    # regex для проверки, начинается ли
                                   # она с '#'
            starts_with_hash += 1  # Если да, то добавить 1 в счетчик
```

Каждая текстовая строка, полученная таким образом, заканчивается символом новой строки, поэтому, прежде чем делать с ней что-либо еще, нередко бывает необходимо удалить этот символ при помощи метода `strip()`.

Например, представим, что у вас есть файл с адресами электронной почты, по одному в строке, и вам требуется сгенерировать гистограмму доменов. Правила корректного извлечения доменных имен несколько замысловаты (к примеру, в списке публичных суффиксов[3]), однако первым приближением будет взять части адреса электронной почты, которые идут после символа `@`. (При этом такие адреса, как `joel@mail.datasciencester.com`, будут обработаны неправильно; для целей данного примера мы оставим это как есть.)

```
def get_domain(email_address: str) -> str:
  """Разбить по '@' и вернуть остаток строки"""
  return email_address.lower().split("@")[-1]

# Пара проверок
assert get_domain('joelgrus@gmail.com') == 'gmail.com'
assert get_domain('joel@m.datasciencester.com') == 'm.datasciencester.com'

from collections import Counter
```

[3] См. **https://publicsuffix.org/**.

```
with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

Файлы с разделителями

В файле с фиктивными адресами электронной почты, который был только что обработан, адреса расположены по одному в строке. Однако чаще приходится работать с файлами, где в одной строке находится большое количество данных. Значения в таких файлах часто разделены либо *запятыми*, либо *символами табуляции*. Каждая строка содержит несколько полей данных, а запятая (или символ табуляции) указывают, где поле заканчивается и дальше начинается новое.

Все усложняется, когда появляются поля, в которых содержатся запятые, символы табуляции и новой строки (а такие неизбежно будут). По этой причине почти всегда будет ошибкой пытаться анализировать подобные файлы самостоятельно. Вместо этого следует использовать модуль `csv` (либо библиотеку pandas) либо какую-либо другую библиотеку, призванную читать файлы с разделителями полей в виде запятых или символов табуляции.

Никогда не делайте разбор файлов с разделителями самостоятельно. В предельных случаях вы непременно потерпите неудачу.

Если в файле нет заголовков (что, вероятно, означает, что вы хотите видеть каждую строку в виде списка, и что также обязывает помнить, какие данные находятся в каждом столбце), то вы можете воспользоваться читающим объектом `csv.reader`, который выполняет навигацию по строкам, надлежащим образом преобразуя каждую из них в список.

Например, если бы у нас был файл с курсами акций, в котором поля разделены символом табуляции:

```
6/20/2014	AAPL	90.91
6/20/2014	MSFT	41.68
6/20/2014	FB	64.5
6/19/2014	AAPL	91.86
6/19/2014	MSFT	41.51
6/19/2014	FB	64.34
```

то мы могли бы их обработать следующим образом:

```
import csv

with open('tab_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
```

```
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

Если же файл имеет заголовки:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

то вы можете либо пропустить строку заголовков, вызвав вначале метод `reader.next`, либо получить каждую строку в виде словаря (где ключами являются заголовки) с помощью читающего объекта `csv.DictReader`:

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
    colon_reader = csv.DictReader(f, delimiter=':')
    for dict_row in colon_reader:
        date = row["date"]
        symbol = row["symbol"]
        closing_price = float(dict_row["closing_price"])
        process(date, symbol, closing_price)
```

Даже если в файле нет заголовков, вы все равно можете использовать читающий объект `DictReader`, передав ему ключи в качестве параметра `fieldnames`.

Запись данных с разделителями в файл выполняется аналогичным образом при помощи пишущего объекта `csv.writer`:

```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }

with open('comma_delimited_stock_prices.txt','wb') as f:
    writer = csv.writer(f, delimiter=',')
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

Пишущий объект `csv.writer` справится, даже если внутри поля тоже есть запятые. Собственный модуль, собранный своими силами, наверняка на такое не будет способен. Например, если попытаться обработать самостоятельно:

```
results = [["test1", "success", "Monday"],
           ["test2", "success, kind of", "Tuesday"],
           ["test3", "failure, kind of", "Wednesday"],
           ["test4", "failure, utter", "Thursday"]]

# Не делайте этого!
with open('bad_csv.txt', 'wb') as f:
    for row in results:
        f.write(",".join(map(str, row)))# Внутри может быть много запятых!
        f.write("\n")  # Строка может также содержать символ новой строки!
```

то в итоге получится csv-файл, который выглядит следующим образом:

```
test1, success, Monday
test2, success, kind of, Tuesday
```

```
test3, failure, kind of, Wednesday
test4, failure, utter, Thursday
```

и который никто никогда не сможет разобрать.

Выскабливание Всемирной паутины

Еще один способ получения данных состоит в выскабливании их из веб-страниц. Выборку самих веб-страниц, как выясняется, сделать довольно легко, а вот получить из них значимую структурированную информацию не так просто.

HTML и его разбор

Веб-страницы написаны на HTML, где текст (в идеале) размечен на элементы и их атрибуты:

```
<html>
  <head>
    <title>Веб-страница</title>
  </head>
  <body>
     <p id="author">Джоэл Грас</p>
     <p id="subject">Наука о данных</p>
  </body>
</html>
```

В идеальном мире, где все веб-страницы ради нашей пользы размечены семантически, мы могли бы извлекать данные, используя правила типа "найти элемент `<p>`, чей `id` равен `subject`, и вернуть текст, который он содержит". В реальности же HTML обычно не только не аннотирован, он зачастую просто составлен не вполне правильно. И поэтому, чтобы в нем разобраться, придется прибегнуть к вспомогательным средствам.

Для получения данных из HTML мы применим библиотеку BeautifulSoup[4], которая строит дерево из разнообразных элементов, расположенных на странице, и предоставляет простой интерфейс для доступа к ним. На момент написания настоящей главы последняя версия была BeautifulSoup 4.6.0, и именно ее мы и будем использовать. Мы также воспользуемся библиотекой requests[5], которая предоставляет более удобный способ создания HTTP-запросов, чем соответствующие средства, встроенные в Python.

Встроенный в Python анализатор HTML слишком требователен к соблюдению правил и, как следствие, не всегда справляется с HTML, который не вполне хорошо сформирован. По этой причине мы также установим другой анализатор под названием html5lib.

---

[4] См. **http://www.crummy.com/software/BeautifulSoup/**.

[5] См. **http://docs.python-requests.org/en/latest/**.

Убедившись, что вы находитесь в правильной виртуальной среде, установите указанные бибилиотеки:

```
python -m pip install beautifulsoup4 requests html5lib
```

Для того чтобы воспользоваться библиотекой BeautifulSoup, нужно передать немного HTML-кода в функцию `BeautifulSoup`. В приводимых примерах это будет результатом вызова метода `requests.get`:

```
from bs4 import BeautifulSoup
import requests

# Я разместил соответствующий файл HTML в репозиторий GitHub,
# и чтобы URL-адрес уместился в книге, я разбил его на две строки.
# Напомним, что разделенные пробелами строковые значения конкатенируются
url = ("https://raw.githubusercontent.com/"
       "joelgrus/data/master/getting-data.html")

html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')
```

после чего можно добиться многого при помощи всего нескольких простых методов.

Как правило, мы будем работать с объектом `Tag`, который соответствует конструкциям разметки HTML или тегам, представляющим структуру HTML-страницы.

Например, чтобы отыскать первый тег `<p>` (и его содержимое), используется:

```
first_paragraph = soup.find('p')   # Первый тег <p>, можно просто soup.p
```

Текстовое содержимое объекта `Tag` можно получить, воспользовавшись его свойством `text`:

```
first_paragraph_text = soup.p.text            # Текст первого элемента <p>
first_paragraph_words = soup.p.text.split()   # Слова первого элемента
```

Атрибуты тега можно извлечь, обращаясь к ним, как к словарю:

```
first_paragraph_id = soup.p['id']             # Вызывает KeyError,
                                              # если 'id' отсутствует
first_paragraph_id2 = soup.p.get('id')        # Возвращает None,
                                              # если 'id' отсутствует
```

Можно получить несколько тегов сразу:

```
all_paragraphs = soup.find_all('p')           # или просто soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

Нередко нужно найти теги с конкретным классом `class` стилевой таблицы:

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
```

И вы можете объединить эти методы, имплементируя более детализированную логику. Например, если нужно отыскать каждый элемент , содержащийся внутри элемента , можно поступить следующим образом:

```
# Элементы <span> внутри элементов <div>.
# Предупреждение: вернет тот же span несколько раз,
# если он находится внутри нескольких элементов div.
# Нужно быть смышленее в этом случае
spans_inside_divs = [span
                     for div in soup('div')     # Для каждого <div>
                                                # на странице
                     for span in div('span')]   # отыскать каждый <span>
                                                # внутри него
```

Только этих нескольких функциональных средств будет достаточно для того, чтобы позволить нам достичь многого. Если же все-таки требуется больше для обработки более сложных вещей (или просто из любопытства), сверьтесь с документацией.

Естественно, какими бы ни были данные, как правило, их важность не обозначена в виде `class="important"`. С целью обеспечения правильности данных необходимо внимательно анализировать исходный код HTML, руководствоваться своей логикой отбора и учитывать крайние случаи. Рассмотрим пример.

Пример: слежение за Конгрессом

Директор по политике в DataSciencester обеспокоен потенциальным регулированием отрасли науки о данных и просит вас квантифицировать, что говорит Конгресс по этой теме. В частности, он хочет, чтобы вы нашли всех представителей, у которых есть пресс-релизы о "данных".

На момент публикации имеется страница со ссылками на все веб-сайты членов Палаты представителей по адресу: **https://www.house.gov/representatives**.

И если вы "просмотрите источник", то все ссылки на веб-сайты выглядят так:

```
<td>
  <a href="https://jayapal.house.gov">Jayapal, Pramila</a>
</td>
```

Начнем со сбора всех URL-адресов, которые привязаны к этой страницы:

```
from bs4 import BeautifulSoup
import requests

url = "https://www.house.gov/representatives"
text = requests.get(url).text
soup = BeautifulSoup(text, "html5lib")

all_urls = [a['href']
            for a in soup('a')
            if a.has_attr('href')]

print(len(all_urls))   # У меня оказалось слишком много, 965
```

Этот фрагмент кода вернет слишком много ссылок. Если вы посмотрите на них, то те, которые нам нужны, начинаются с http:// либо https://, имеют какое-то имя и заканчиваются либо на .house.gov либо на .house.gov/.

Это как раз то место, где следут применить регулярное выражение:

```
import re

# Должно начинаться с http:// либо https://
# Должно оканчиваться на .house.gov либо .house.gov/
regex = r"^https?://.*\.house\.gov/?$"

# Напишем несколько тестов!
assert re.match(regex, "http://joel.house.gov")
assert re.match(regex, "https://joel.house.gov")
assert re.match(regex, "http://joel.house.gov/")
assert re.match(regex, "https://joel.house.gov/")
assert not re.match(regex, "joel.house.gov")
assert not re.match(regex, "http://joel.house.com")
assert not re.match(regex, "https://joel.house.gov/biography")

# И теперь применим
good_urls = [url for url in all_urls if re.match(regex, url)]

print(len(good_urls))   # У меня по-прежнему много, 862
```

Это все еще слишком много, т. к. представителей всего 435. Если вы посмотрите на список, то там много дубликатов. Давайте воспользуемся структурой данных `set` для того, чтобы их устранить:

```
good_urls = list(set(good_urls))

print(len(good_urls))   # у меня только 431
```

Всегда есть пара свободных мест в палате, или, может быть, есть представитель без веб-сайта. В любом случае этого достаточно. Когда мы смотрим на веб-сайты, то видим, что большинство из них имеют ссылку на пресс-релизы. Например:

```
html = requests.get('https://jayapal.house.gov').text
soup = BeautifulSoup(html, 'html5lib')

# Мы используем множество set, т. к. ссылки могут появляться многократно
links = {a['href'] for a in soup('a') if 'press releases' in a.text.lower()}

print(links) # {'/media/press-releases'}
```

Обратите внимание, что эта ссылка является относительной, т. е. нам нужно помнить исходный веб-сайт. Давайте выполним его выскабливание:

```
from typing import Dict, Set

press_releases: Dict[str, Set[str]] = {}
```

```
for house_url in good_urls:
    html = requests.get(house_url).text
    soup = BeautifulSoup(html, 'html5lib')
    pr_links = {a['href'] for a in soup('a') if 'press releases'
                                              in a.text.lower()}
    print(f"{house_url}: {pr_links}")
    press_releases[house_url] = pr_links
```

Безудержное выскабливание веб-сайта, как тут, обычно считается невежливым поступком. Большинство веб-сайтов имеют файл robots.txt, который показывает, как часто вы можете выскабливать веб-сайт (и какие пути вы не должны выскабливать), но поскольку это Конгресс, то мы не обязаны быть особенно вежливыми.

Если вы понаблюдаете за тем, как они прокручиваются, то вы увидите много адресов /media/press-releases и media-center/press-releases, а также разнообразные другие адреса. Один из таких URL-адресов — **https://jayapal.house.gov/media/press-releases**.

Напомним, что наша цель — выяснить, какие конгрессмены имеют пресс-релизы с упоминанием "данных". Мы напишем несколько более общую функцию, которая проверяет, упоминает ли страница пресс-релизов какой-либо заданный термин.

Если вы посетите веб-сайт и просмотрите источник, то окажется, что внутри тега <p> есть фрагмент из каждого пресс-релиза, поэтому мы будем использовать его в качестве нашей первой попытки:

```
def paragraph_mentions(text: str, keyword: str) -> bool:
    """
    Возвращает True, если <p> находится внутри
    упоминаний {ключевого слова} в тексте
    """
    soup = BeautifulSoup(text, 'html5lib')
    paragraphs = [p.get_text() for p in soup('p')]

    return any(keyword.lower() in paragraph.lower()
               for paragraph in paragraphs)
```

Давайте напишем для этого быстрый тест:

```
text = """<body><h1>Facebook</h1><p>Twitter</p>"""
assert paragraph_mentions(text, "twitter")       # внутри <p>
assert not paragraph_mentions(text, "facebook")  # не внутри <p>
```

Наконец-то мы готовы найти соответствующих конгрессменов и назвать их имена директору:

```
for house_url, pr_links in press_releases.items():
    for pr_link in pr_links:
        url = f"{house_url}/{pr_link}"
        text = requests.get(url).text
```

```
        if paragraph_mentions(text, 'data'):
            print(f"{house_url}")
            break  # с этим адресом house_url закончено
```

Когда я выполняю этот фрагмент кода, я получаю список из приблизительно 20 представителей. Ваши результаты, вероятно, будут другими.

Если вы посмотрите на различные страницы "пресс-релизов", то большинство из них разбиты на страницы только по 5 или 10 пресс-релизов на страницу. Это означает, что мы получили лишь несколько последних пресс-релизов по каждому конгрессмену. Более тщательное решение обрабатывало бы страницы итеративно и извлекало полный текст каждого пресс-релиза.

Использование интерфейсов API

Многие веб-сайты и веб-службы предоставляют интерфейсы программирования приложений (application programming interfaces, API), которые позволяют явным образом запрашивать данные в структурированном формате. И это в значительной мере избавляет от необходимости заниматся выскабливанием данных самому!

Форматы JSON и XML

Поскольку протокол HTTP служит для передачи текста, то данные, которые запрашиваются через веб-API, должны быть *сериализованы* в строковый формат. Нередко при сериализации используется объектная нотация языка JavaScript (JavaScript Object Notation, JSON). Объекты в JavaScript очень похожи на словари Python, что облегчает интерпретацию их строковых представлений:

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2014,
  "topics" : [ "data", "science", "data science"] }
```

Текст в формате JSON анализируется при помощи модуля `json`. В частности, мы будем использовать его функцию `loads`, которая десериализует строкове значение, представляющее объект JSON, в объект Python:

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : "Joel Grus",
                  "publicationYear" : 2019,
                  "topics" : [ "data", "science", "data science"] }"""

# Разобрать JSON, создав Python'овский словарь
deserialized = json.loads(serialized)

assert deserialized["publicationYear"] == 2019
assert "data science" in deserialized["topics"]
```

Иногда поставщик API вас ненавидит и предоставляет ответы только в формате XML:

```
<Book>
   <Title>Data Science Book</Title>
   <Author>Joel Grus</Author>
   <PublicationYear>2014</PublicationYear>
   <Topics>
      <Topic>data</Topic>
      <Topic>science</Topic>
      <Topic>data science</Topic>
   </Topics>
</Book>
```

Для получения данных из XML вы можете воспользоваться библиотекой BeautifulSoup аналогично тому, как она применялась для получения данных из HTML. Подробности см. в документации.

Использование неаутентифицированного API

Большинство интерфейсов API в наши дни требует от пользователя сначала аутентифицировать себя прежде, чем начать ими пользоваться. Хотя никто не порицает их за такую политику, она создает много излишних шаблонов, которые затуманивают непосредственный контакт. Поэтому мы сначала обратимся к программному интерфейсу GitHub[6], с помощью которого можно делать некоторые простые вещи без аутентификации:

```
import requests, json

github_user = "joelgrus"
endpoint = f"https://api.github.com/users/{github_user}/repos"

repos = json.loads(requests.get(endpoint).text)
```

В этом месте переменная `repos` представляет собой список из словарей, каждый из которых обозначает публичное хранилище в моей учетной записи GitHub. (Можете подставить свое пользовательское имя и получить собственные данные из хранилища. Ведь у вас уже есть учетная запись GitHub?)

Эти данные можно использовать, например, для выявления того, в какие месяцы и по каким дням недели я чаще всего создаю хранилища. Единственная проблема заключается в том, что даты в ответе представлены строкой символов (Юникода):

```
"created_at": "2013-07-05T02:02:28Z"
```

Python поставляется без сносного анализатора дат, поэтому его придется установить:

```
pip install python-dateutil
```

[6] См. **http://developer.github.com/v3/**.

из которого, скорее всего, понадобится только функция `dateutil.parser.parse`:

```
from collections import Counter
from dateutil.parser import parse

dates = [parse(repo["created_at"]) for repo in repos]    # список дат
month_counts = Counter(date.month for date in dates)     # число месяцев
weekday_counts = Counter(date.weekday() for date in dates)  # число будних дней
```

Схожим образом вы можете получить языки программирования моих последних пяти хранилищ:

```
last_5_repositories = sorted(repos,                  # последние 5 хранилищ
                             key=lambda r: r["created_at"],
                             reverse=True)[:5]

last_5_languages = [repo["language"]                 # последние 5 языков
                    for repo in last_5_repositories]
```

Как правило, мы не будем работать с API на низком уровне в виде выполнения запросов и самостоятельного разбора ответов. Одно из преимуществ использования языка Python заключается в том, что кто-то уже разработал библиотеку практически для любого API, доступ к которому требуется получить. Когда библиотеки выполнены сносно, они помогут избежать уймы неприятностей при выяснении проблемных подробностей доступа к API. (Когда они выполнены кое-как или когда выясняется, что они основаны на несуществующей версии соответствующих API, они на самом деле становятся причиной головной боли.)

Так или иначе, периодически возникает необходимость разворачивать собственную библиотеку доступа к API (либо, что более вероятно, копаться в причинах, почему чужая библиотека не работает, как надо), поэтому хорошо бы познакомиться с некоторыми подробностями.

Отыскание API-интерфейсов

Если вам нужны данные с определенного веб-сайта, то все подробности следует искать в разделе для разработчиков или разделе API веб-сайта и попытаться отыскать библиотеку в Интернете по запросу "python <имя_сайта> api".

Существуют библиотеки для API Yelp, для API Instagram, для API Spotify и т. д.

Если нужен список API, которые имеют обертки Python, то неплохой список есть в репозитории Real Python на GitHub[7].

И если вы не можете отыскать то, что вам нужно, то всегда есть выскабливание, как последнее прибежище исследователя данных.

[7] См. **https://github.com/realpython/list-of-python-api-wrappers**.

Пример: использование API-интерфейсов Twitter

Twitter — это фантастический источник данных для работы. Его можно использовать для получения новостей в реальном времени, для измерения реакции на текущие события, для поиска ссылок, связанных с конкретными темами, практически для всего, что можно представить себе, пока имеется доступ к его данным, который можно получить через его программные интерфейсы API.

Для взаимодействия с интерфейсами программирования приложений, предоставленными разработчикам социальной сетью Twitter, будем использовать библиотеку Twython[8], `pip install twython`). Для работы с социальной сетью Twitter на языке Python имеется целый ряд других библиотек, однако работа именно с этой библиотекой у меня удалась лучше всего. Вы можете исследовать и другие!

Получение учетных данных

Для того чтобы воспользоваться API-интерфейсами соцсети Twitter, нужно получить некоторые полномочия (для которых необходимо иметь учетную запись Twitter, и которую так или иначе следует иметь, благодаря чему можно стать частью живого и дружелюбного сообщества Twitter `#datascience`).

Как и все остальные инструкции, касающиеся веб-сайтов, которые я не контролирую, в определенный момент они могут устареть, но все же есть надежда, что они будут актуальными в течение некоторого времени. (Хотя они уже менялись несколько раз с тех пор, как я начал писать эту книгу, так что удачи!)

Вот необходимые шаги:

1. Перейдите на **https://developer.twitter.com/**.
2. Если вы не вошли на веб-сайт, то щелкните на ссылке **Sign in** (Войти) и введите имя пользователя Twitter и пароль.
3. Нажмите кнопку **Apply** (Применить) для подачи заявки на аккаунт разработчика.
4. Запросите доступ для личного пользования.
5. Заполните заявку. Она потребует 300 слов (это так) относительно того, зачем вам нужен доступ, поэтому превысьте лимит и сообщите им об этой книге и о том, как она вам нравится.
6. Подождите некоторое неопределенное время.
7. Если вы знаете кого-то, кто работает в Twitter, то отправьте ему электронное письмо с просьбой ускорить вашу заявку. В противном случае просто продолжайте ждать.

[8] См. **https://github.com/ryanmcgrath/twython**.

8. После получения одобрения вернитесь на **https://developer.twitter.com/**, отыщите раздел приложений **Apps** и щелкните по **Create an app** (Создать приложение).

9. Заполните все необходимые поля (опять же, если потребуются дополнительные символы для описания, вы могли бы подумать об этой книге и о том, какой восхитительной вы ее находите).

10. Нажмите кнопку **Create** (Создать).

Теперь ваше приложение будет иметь вкладку **Keys and tokens** (Ключи и токены) с разделом ключей API потребителя, в котором перечислены "Ключ API" и "Секретный ключ API". Запишите их в свой блокнот; они вам понадобятся. (Также держите их в секрете! Это то же самое, что и пароли.)

Не делитесь ключами, не публикуйте их в своей книге и не помещайте их в свой публичный репозиторий GitHub. Одно из простых решений — хранить их в файле credentials.json, который не передается, а в своем коде использовать функцию `json.loads` для их извлечения. Еще одним решением является хранение их в переменных среды, а для их извлечения использовать функцию `os.environ`.

Использование библиотеки Twython

Самая сложная часть использования API Twitter — это аутентифицировать себя. (Действительно, это самая сложная часть использования большого числа API.) Поставщики API хотят убедиться, что вы авторизованы для доступа к их данным и не превышаете их лимиты использования. Они также хотят знать, кто получает доступ к их данным.

Аутентификация — это своего рода головная боль. Существует простой способ, OAuth 2, которого достаточно, когда вы просто хотите выполнить простой поиск. И есть сложный способ, OAuth 1, который требуется, когда вы хотите выполнять действия (например, отправление твитов) или (в частности, для нас) подключаться к потоку Twitter.

Поэтому мы застряли на более сложном пути, который постараемся автоматизировать как можно больше.

Во-первых, вам нужен ключ API и секретный ключ API (иногда соответственно именуемые ключом потребителя и секретом потребителя). Я буду получать свои из переменных среды, но не стесняйтесь поменять этот подход в своем случае так, как вы хотите:

```
import os

# Вы можете свободно встроить свой ключ и секрет прямиком в код
CONSUMER_KEY = os.environ.get("КЛЮЧ_ПОТРЕБИТЕЛЯ_TWITTER")
CONSUMER_SECRET = os.environ.get("СЕКРЕТ_ПОТРЕБИТЕЛЯ_TWITTER")
```

Теперь мы можем инстанцировать клиента:

```
import webbrowser
from twython import Twython

# Получить временного клиента для извлечения аутентификационного URL
temp_client = Twython(CONSUMER_KEY, CONSUMER_SECRET)
temp_creds = temp_client.get_authentication_tokens()
url = temp_creds['auth_url']

# Теперь посетить этот URL для авторизации приложения и получения PIN
print(f"перейдите на {url} и получите PIN-код и вставьте его ниже")
webbrowser.open(url)
PIN_CODE = input("пожалуйста, введите PIN-код: ")

# Теперь мы используем PIN_CODE для получения фактических токенов
auth_client = Twython(CONSUMER_KEY,
                      CONSUMER_SECRET,
                      temp_creds['oauth_token'],
                      temp_creds['oauth_token_secret'])

final_step = auth_client.get_authorized_tokens(PIN_CODE)

ACCESS_TOKEN = final_step['oauth_token']
ACCESS_TOKEN_SECRET = final_step['oauth_token_secret']

# И получим новый экземпляр Twython, используя их.
twitter = Twython(CONSUMER_KEY,
                  CONSUMER_SECRET,
                  ACCESS_TOKEN,
                  ACCESS_TOKEN_SECRET)
```

На этом этапе вы можете рассмотреть возможность сохранения `ACCESS_TOKEN` **и** `ACCESS_TOKEN_SECRET` **в безопасном месте, чтобы в следующий раз вам не пришлось проходить через этот вздор.**

После того как мы аутентифицировали экземпляр Twython, мы можем приступить к выполнению операций поиска:

```
# Найти твиты, содержащие фразу "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"]
    text = status["text"]
    print(f"{user}: {text}\n")
```

Если вы выполните этот фрагмент кода, то получите несколько твитов, как тут:

```
haithemnyc: Data scientists with the technical savvy &amp;amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM
```

```
spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization. So easy and Awesome.
```

Они не вызывают какого-то интереса, главным образом потому, что API поиска в соцсети Twitter Search API показывает всего несколько последних результатов, которые он считает нужным показать. Когда вы решаете задачи, связанные с наукой о данных, чаще требуется гораздо больше твитов. Для таких целей служит потоковый Streaming API[9]. Этот интерфейс позволяет подключаться к огромному потоку сообщений Twitter. Для того чтобы им воспользоваться, вам нужно аутентифицироваться с помощью личных токенов доступа.

Для того чтобы получить доступ к Streaming API при помощи Twython, необходимо определить класс, который наследует у класса `TwythonStreamer` и переопределяет его метод `on_success` и, возможно, его метод `on_error`:

```
from twython import TwythonStreamer

# Добавлять данные в глобальную переменную - это пример плохого стиля,
# но он намного упрощает пример
tweets = []

class MyStreamer(TwythonStreamer):
    """Наш собственный подкласс класса TwythonStreamer, который
       определяет, как взаимодействовать с потоком"""

    def on_success(self, data):
        """Что делать, когда Twitter присылает данные?
           Здесь данные будут в виде словаря, представляющего твит"""

        # Мы хотим собирать твиты только на английском
        if data['lang'] == 'en':
            tweets.append(data)
            print("received tweet #", len(tweets))

        # Остановиться, когда собрано достаточно
        if len(tweets) >= 1000:
            self.disconnect()

    def on_error(self, status_code, data):
        print(status_code, data)
        self.disconnect()
```

Подкласс `MyStreamer` подключится к потоку Twitter и будет ждать, когда Twitter подаст данные. Каждый раз, когда он получает некоторые данные (здесь твит представлен как объект языка Python), он передает этот твит в метод `on_success`, кото-

[9] Потоковые интерфейсы API соцсети Twitter Streaming API — это ряд компонентов API социальной сети Twitter, построенных на основе архитектуры REST, которые позволяют разработчикам получать доступ к глобальному потоку твит-данных с низкой задержкой доступа. — *Прим. пер.*

рый добавляет его в список полученных твитов `tweets` при условии, что они на английском языке, и в конце он отсоединяет стример, собрав 1000 твитов.

Осталось только инициализировать его и выполнить:

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)

# Начинает потреблять публичные новостные ленты,
# которые содержат ключевое слово 'data'
stream.statuses.filter(track='data')

# Если же, напротив, мы хотим начать потреблять
# ВСЕ публичные новостные ленты
# stream.statuses.sample()
```

Этот фрагмент кода будет выполняться до тех пор, пока он не соберет 1000 твитов (или пока не встретит ошибку), и остановится, после чего мы можем приступить к анализу твитов. Например, наиболее распространенные хештеги можно найти следующим образом:

```
# Ведущие хештеги
top_hashtags = Counter(hashtag['text'].lower()
                       for tweet in tweets
                       for hashtag in tweet["entities"]["hashtags"])

print(top_hashtags.most_common(5))
```

Каждый твит содержит большое количество данных. За подробностями можно обратиться к окружающим либо покопаться в документации Twitter API[10].

В реальном проекте, вероятно, вместо того чтобы полагаться на список, хранящийся в оперативной памяти, как на структуру данных для твитов, следует сохранить их в файл или базу данных, чтобы иметь их всегда под рукой.

Для дальнейшего изучения

- Библиотека pandas[11] — одна из первостепенных, используемых в науке о данных для работы с данными (в частности, для импорта данных).
- Библиотека Scrapy[12] более полнофункциональна и служит для построения сложных веб-анализаторов, которые обладают такими возможностями, как переход по неизвестным ссылкам.
- Интернет-сообщество исследователей данных Kaggle[13] содержит огромную коллекцию наборов данных.

[10] См. **https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object**.

[11] См. **http://pandas.pydata.org/**.

[12] См. **http://scrapy.org/**.

[13] См. **https://www.kaggle.com/datasets**.

ГЛАВА 10
Работа с данными

> Эксперты часто обладают, скорее, данными, чем здравым смыслом.
> – *Колин Пауэлл*[1]

Обработка данных — это одновременно и искусство, и наука. До сих пор главным образом обсуждалась научная сторона работы с данными. В этой главе будут рассмотрены некоторые аспекты, которые можно отнести к искусству.

Разведывательный анализ данных

После того как вы выявили вопросы, на которые вы пытаетесь получить ответ, и получили на руки некоторые данные, у вас может возникнуть соблазн сразу же погрузиться в работу, начав строить модели и получать ответы. Однако вам следует противостоять такому искушению. Вашим первым шагом должен быть *разведывательный анализ* данных.

Разведывание одномерных данных

Простейший случай — у вас есть одномерный набор данных, т. е. просто коллекция чисел. Например, это может быть среднее число минут, которые каждый пользователь проводит на веб-сайте каждый день, число просмотров учебных видеороликов из коллекции по науке о данных или число страниц в каждой книге из подборки по науке о данных.

Очевидным первым шагом будет вычисление нескольких сводных статистик. К примеру, можно выяснить количество точек данных, которые у вас есть, их минимальное, максимальное, среднее значение и стандартное отклонение.

Но даже эти показатели не обязательно обеспечат вам понимание. Неплохой следующий шаг — создать гистограмму, в которой данные разбиты на дискретные *интервалы* (корзины), и подсчитать число точек данных, попадающих в каждый из них:

---

[1] Колин Лютер Пауэлл (род. 1937) — генерал Вооруженных сил США, госсекретарь в период первого срока президентства Дж. Буша-младшего. Известен в связи с фальсификацией в Совете Безопасности ООН 5 февраля 2003 г., когда он демонстрировал ампулу, якобы с бациллами сибирской язвы, чтобы подтвердить факт наличия биологического оружия в Ираке. — *Прим. пер.*

```
from typing import List, Dict
from collections import Counter
import math
import matplotlib.pyplot as plt

def bucketize(point: float, bucket_size: float) -> float:
    """Округлить точку до следующего наименьшего кратного
       размера интервала bucket_size"""
    return bucket_size * math.floor(point / bucket_size)

def make_histogram(points: List[float], bucket_size: float) -> Dict[float, int]:
    """Разбивает точки на интервалы и подсчитывает
       их количество в каждом интервале"""
    return Counter(bucketize(point, bucket_size) for point in points)

def plot_histogram(points: List[float], bucket_size: float,
                   title: str = ""):
    histogram = make_histogram(points, bucket_size)
    plt.bar(list(histogram.keys()),
            list(histogram.values()), width=bucket_size)
    plt.title(title)
    plt.show()
```

Например, рассмотрим следующие два набора данных:

```
import random
from scratch.probability import inverse_normal_cdf

random.seed(0)

# Равномерное распределение между -100 и 100
uniform = [200 * random.random() - 100 for _ in range(10000)]

# Нормальное распределение со средним 0, стандартным отклонением 57
normal = [57 * inverse_normal_cdf(random.random())
          for _ in range(10000)]
```

У обоих средние значения близки к 0, а стандартные отклонения близки к 58. Тем не менее они имеют совершенно разные распределения. На рис. 10.1 показано равномерно распределенный набор данных `uniform`:

```
plot_histogram(uniform, 10, "Равномерная гистограмма")
```

тогда как на рис. 10.2 — нормально распределенный набор данных `normal`:

```
plot_histogram(normal, 10, "Нормальная гистограмма")
```

В данном случае у обоих распределений довольно разные показатели `max` и `min`, но даже этих сведений недостаточно, чтобы понять то, *насколько* они разные.

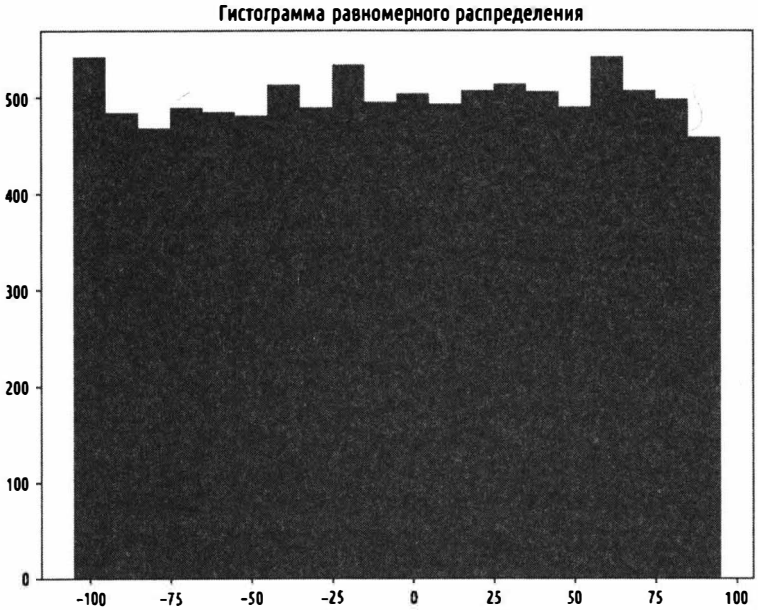


Рис. 10.1. Гистограмма равномерного распределения

Рис. 10.2. Гистограмма нормального распределения

Двумерные данные

Теперь в нашем распоряжении набор данных с двумя размерностями. Например, кроме ежедневного числа минут еще может учитываться опыт работы в науке о данных в годах. Разумеется, вы бы хотели разобраться в каждой размерности по отдельности. Но, вызможно, вы захотите разбросать данные на диаграмме рассеяния.

Для примера рассмотрим еще один фиктивный набор данных:

```
def random_normal():
    """Возвращает случайную выборку из стандартного
       нормального распределения"""
    return inverse_normal_cdf(random.random())

xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

Если бы вам нужно было выполнить функцию `plot_histogram` со списками `ys1` и `ys2` в качестве аргументов, то получились бы очень похожие диаграммы (действительно, обе нормально распределены с одинаковым средним значением и стандартным отклонением).

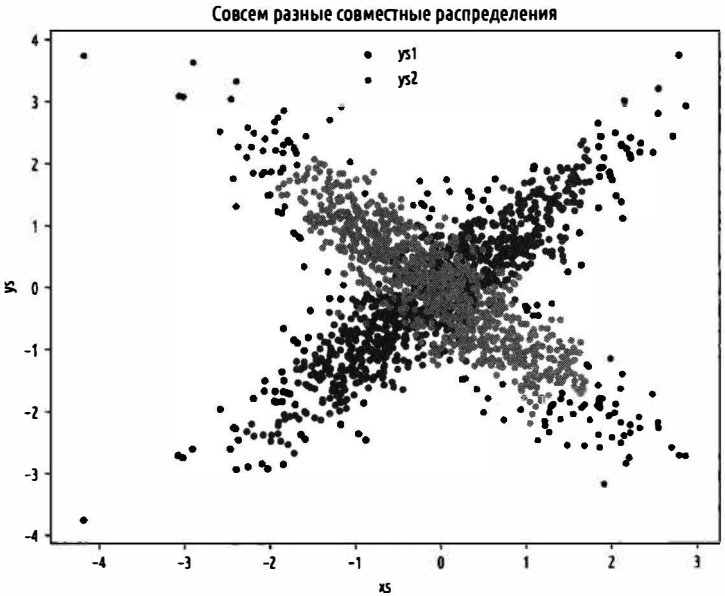


Рис. 10.3. Рассеяние двух разных наборов данных `ys`

Но у обоих, как видно из рис. 10.3, имеется совсем разное совместное распределение с `xs`:

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')
```

```
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Совсем разные совместные распределения")
plt.show()
```

Эта разница также будет очевидной, если вы обратитесь к корреляции:

```
from scratch.statistics import correlation

print(correlation(xs, ys1))   # примерно 0.9
print(correlation(xs, ys2))   # примерно -0.9
```

Многочисленные размерности

Имея многочисленные размерности, вы хотели бы знать, как все размерности соотносятся друг с другом. Простой подход состоит в том, чтобы посмотреть на *матрицу корреляций*, в которой элемент в строке *i* и столбце *j* означает корреляцию между *i*-й размерностью и *j*-й размерностью:

```
from scratch.linear_algebra import Matrix, Vector, make_matrix

def correlation_matrix(data: List[Vector]) -> Matrix:
    """
    Возвращает матрицу размера len(data) x len(data),
    (i, j)-й элемент которой является корреляцией между data[i] и data[j]
    """
    def correlation_ij(i: int, j: int) -> float:
        return correlation(data[i], data[j])

    return make_matrix(len(data), len(data), correlation_ij)
```

Более визуальный поход (если у вас не так много размерностей) — создать *матрицу рассеяний* (рис. 10.4), которая показывает все попарные диаграммы рассеяния. Для этого мы воспользуемся функцией `plt.subplots`, которая позволяет создавать подграфики. На вход функции передается число строк и столбцов, а та, в свою очередь, возвращает объект `figure` (который здесь не используется) и двумерный массив объектов `axes` (в каждом из которых мы построим график):

```
# Данные corr_data - это список из четырех 100-мерных векторов
num_vectors = len(corr_data)
fig, ax = plt.subplots(num_vectors, num_vectors)

for i in range(num_vectors):
    for j in range(num_vectors):

        # Разбросать столбец j по оси x напротив столбца i на оси y
        if i != j: ax[i][j].scatter(corr_data[j], corr_data[i])
```

```
        # Если не i == j, то в этом случае показать имя серии
        else: ax[i][j].annotate("Серия " + str(i), (0.5, 0.5),
                                xycoords='axes fraction',
                                ha="center", va="center")

        # Затем спрятать осевые метки, за исключением
        # левой и нижней диаграмм
        if i < num_vectors - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)

# Настроить правую нижнюю и левую верхнюю осевые метки,
# которые некорректны, потому что на их диаграммах выводится только текст
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())
plt.show()
```

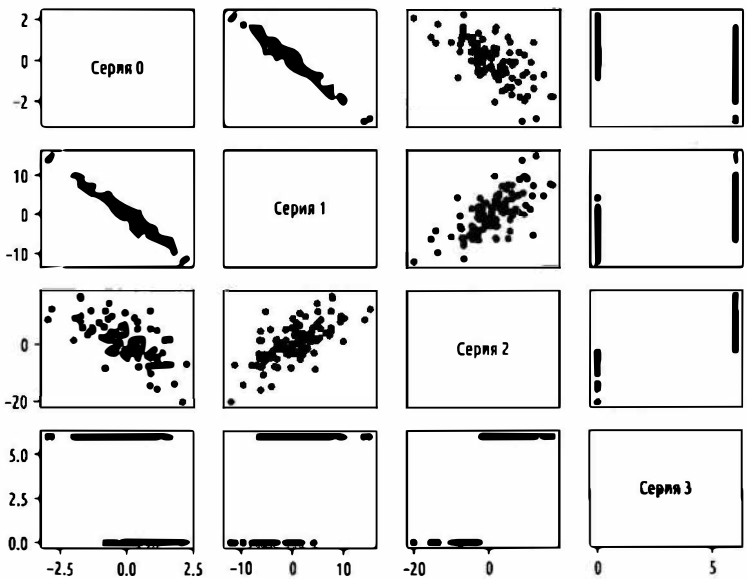


Рис. 10.4. Матрица рассеяний

Глядя на диаграммы рассеяния, можно увидеть, что серия 1 очень отрицательно коррелирует с серией 0, серия 2 положительно коррелирует с серией 1, серия 3 принимает только значения 0 и 6, причем 0 соответствует малым значениям серии 2, а 6 соответствует большим значениям.

Матрица рассеяний позволяет быстро получить грубое представление о том, какие переменные коррелируют (если, конечно, не сидеть часами, пытаясь точно настроить matplotlib, чтобы изобразить диаграмму именно так, как хочется, и тогда это не быстрый способ).

Применение типизированных именованных кортежей

Для представления данных общепринято использовать словари.

```
import datetime

stock_price = {'closing_price': 102.06,
               'date': datetime.date(2014, 8, 29),
               'symbol': 'AAPL'}
```

Однако есть несколько причин, почему это далеко не идеальный вариант. Такое представление является немного неэффективным (словарь сопровождается некоторыми накладными расходами); так что если у вас много цен на акции, то они займут больше памяти, чем нужно. По большей части это незначительное соображение.

Более крупная проблема заключается в том, что доступ к элементам по ключу словаря подвержен ошибкам. Следующий ниже фрагмент кода будет работать без ошибок и вести себя неправильно:

```
# Ой, опечатка
stock_price['cosing_price'] = 103.06
```

Наконец, в отличие от единообразных словарей, которые мы можем аннотировать типами:

```
prices: Dict[datetime.date, float] = {}
```

нет полезного способа аннотировать словари как данные, которые имеют множество разных типов значений. И поэтому мы также теряем мощь подсказок типов.

Язык Python в качестве альтернативы предлагает класс именованных кортежей `namedtuple`, который похож на кортеж, но с именованными слотами:

```
from collections import namedtuple

StockPrice = namedtuple('StockPrice', ['symbol', 'date', 'closing_price'])
price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)

assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
```

Как и обычные кортежи, именованные кортежи `namedtuple` немутируемы, т. е. вы не можете изменять их значения после их создания. Иногда это мешает, но в основном это хорошо.

Вы заметите, что мы все еще не решили проблему с аннотацией типов. Мы делаем это, используя типизированный вариант `NamedTuple`:

```
from typing import NamedTuple

class StockPrice(NamedTuple):
    symbol: str
```

```
    date: datetime.date
    closing_price: float

    def is_high_tech(self) -> bool:
        """Это класс, и поэтому мы также можем добавлять методы"""
        return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']

price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)

assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
assert price.is_high_tech()
```

И теперь ваш редактор способен вас выручить, как показано на рис. 10.5.

Рис. 10.5. Предупредительный редактор

Очень мало людей пользуется типизированными именованными кортежами `NamedTuple` таким образом. Но им следовало бы.

Классы данных *dataclasses*

Класс данных модуля `dataclasses` является (своего рода) мутируемой версией именованного кортежа модуля `namedtuple`. (Я говорю "вроде", потому что именованные кортежи, в частности его типизированный вариант `NamedTuple`, представляют свои данные компактно, как кортежи, тогда как классы данных — это регулярные классы Python, которые просто генерируют некоторые методы за вас автоматически.)

Модуль `dataclasses` появился в Python 3.7. Если вы используете старую версию языка, то этот раздел у вас работать не будет.

Синтаксис классов данных очень похож на именованные кортежи. Но вместо наследования от базового класса мы используем декоратор:

```
from dataclasses import dataclass

@dataclass
class StockPrice2:
    symbol: str
    date: datetime.date
    closing_price: float
```

```
    def is_high_tech(self) -> bool:
        """Это класс, и поэтому мы также можем добавлять методы"""
        return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']

price2 = StockPrice2('MSFT', datetime.date(2018, 12, 14), 106.03)

assert price2.symbol == 'MSFT'
assert price2.closing_price == 106.03
assert price2.is_high_tech()
```

Как уже упоминалось, главное различие заключается в том, что мы можем модифицировать значения экземпляра класса данных:

```
# Дробление акций
price2.closing_price /= 2
assert price2.closing_price == 51.03
```

Если мы попытаемся модифицировать поле версии `NamedTuple`, то мы получим ошибку атрибута `AttributeError`.

Это также подвергает нас таким ошибкам, которых мы надеялись избежать, не используя словарей:

```
# Это регулярный класс, и поэтому мы добавляем новые поля, как захотим!
price2.cosing_price = 75   # ой
```

Мы не будем использовать классы данных, но вы можете столкнуться с ними на практике.

Очистка и конвертирование

В реальном мире данные *загрязнены*. И прежде чем их использовать, нередко приходится проделывать немалую работу. Мы видели примеры тому в *главе 9*. Перед тем как начать пользоваться числами, мы должны конвертировать строковые значения в вещественные `float` и целые `int`. Мы должны проверить, имеются ли пропущенные, экстремальные (выбросы) или плохие данные.

Ранее мы делали это прямо перед их использованием:

```
# Цена закрытия, т. е. в конце периода
closing_price = float(row[2])
```

Однако, вероятно, менее подверженный ошибкам способ — делать разбор в функции, которую мы можем протестировать:

```
from dateutil.parser import parse

def parse_row(row: List[str]) -> StockPrice:
    symbol, date, closing_price = row
    return StockPrice(symbol=symbol,
                      date=parse(date).date(),
                      closing_price=float(closing_price))
```

```
# Теперь протестируем нашу функцию
stock = parse_row(["MSFT", "2018-12-14", "106.03"])

assert stock.symbol == "MSFT"
assert stock.date == datetime.date(2018, 12, 14)
assert stock.closing_price == 106.03
```

Что, если имеются плохие данные? "Вещественное" значение, которое фактически не представляет число? Возможно, вы бы предпочли получить `None`, чем сбой программы?

```
from typing import Optional
import re

def try_parse_row(row: List[str]) -> Optional[StockPrice]:
    symbol, date_, closing_price_ = row
    # Символ акции должен состоять только из прописных букв
    if not re.match(r"^[A-Z]+$", symbol):
        return None

    try:
        date = parse(date_).date()
    except ValueError:
        return None

    try:
        closing_price = float(closing_price_)
    except ValueError:
        return None

    return StockPrice(symbol, date, closing_price)

# Должно вернуть None в случае ошибок
assert try_parse_row(["MSFT0", "2018-12-14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12--14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12-14", "x"]) is None

# Но должно вернуть то же, что и раньше, если данные хорошие
assert try_parse_row(["MSFT", "2018-12-14", "106.03"]) == stock
```

Например, если у нас цены акций разделены запятыми и имеют плохие данные:

```
AAPL,6/20/2014,90.91
MSFT,6/20/2014,41.68
FB,6/20/3014,64.5
AAPL,6/19/2014,91.86
MSFT,6/19/2014,n/a
FB,6/19/2014,64.34
```

то теперь мы можем прочитать и вернуть только допустимые строки данных:

```
import csv

data: List[StockPrice] = []

with open("comma_delimited_stock_prices.csv") as f:
    reader = csv.reader(f)
    for row in reader:
        maybe_stock = try_parse_row(row)
        if maybe_stock is None:
            print(f"пропуск недопустимой строки данных: {row}")
        else:
            data.append(maybe_stock)
```

и решить, что нам делать с недопустимыми. Вообще говоря, существует три варианта — избавиться от них, вернуться к источнику и попытаться исправить плохие/пропущенные данные или ничего не делать и скрестить пальцы. Если имеется одна строка некорректных данных из миллионов, то, вероятно, нормальным решением будет ее проигнорировать. Но если половина ваших строк имеет плохие данные, то вам придется это исправить.

Хороший следующий шаг — проверить наличие выбросов, используя методы из *разд. "Разведывательный анализ данных"* либо путем конкретно обусловленного исследования. Например, заметили ли вы, что одна из дат в файле акций имела год 3014? Это не (обязательно) приведет вас к ошибке, но совершенно очевидно, что это неправильно, и вы получите странные результаты, если не отловите ее. В реальных наборах данных могут отсутствовать десятичные точки, иметься дополнительные нули, типографские ошибки и бесчисленное множество других проблем, которые вы должны отловить. (Может быть, это не входит в ваши обязанности, но кто еще это сделает?)

Оперирование данными

Одним из самых важных профессиональных навыков иследователя данных является умение *оперировать данными*. Это, скорее, общий подход, чем конкретный набор приемов, поэтому пройдемся по некоторым примерам для того, чтобы почувствовать его особенности.

Представим, что мы работаем со словарями, содержащими курсы акций в следующем формате:

```
data = [
    StockPrice(symbol='MSFT',
               date=datetime.date(2018, 12, 24),
               closing_price=106.03),
    # ...
]
```

Начнем задавать вопросы об этих данных, попутно пытаясь обнаружить регулярности в том, что делается, и абстрагируя для себя некоторые инструменты, которые облегчают оперирование данными.

Например, предположим, что нужно узнать самую высокую цену закрытия по акциям Apple (биржевой тикер AAPL). Разобьем задачу на конкретные шаги:

1. Ограничиться строками с финансовым активом AAPL.
2. Извлечь в каждой строке цену закрытия `closing_price`.
3. Взять из этих цен максимальную.

Все три шага можно выполнить одновременно при помощи операции включения:

```
# Максимальная цена на акции AAPL
max_aapl_price = max(row["closing_price"]
                     for row in data
                     if row["symbol"] == "AAPL")
```

В более общем плане, мы, возможно, захотим узнать самые высокие цены для каждой акции в нашем наборе данных. Один из способов сделать это:

1. Создать словарь для отслеживания самых высоких цен (мы будем использовать словарь `defaultdict`, который возвращает минус бесконечность для пропущенных значений, поскольку любая цена будет выше, чем она:
2. Перебрать наши данные, обоновляя их.

Вот соответствующий код:

```
from collections import defaultdict

max_prices: Dict[str, float] = defaultdict(lambda: float('-inf'))

for sp in data:
    symbol, closing_price = sp.symbol, sp.closing_price
    if closing_price > max_prices[symbol]:
        max_prices[symbol] = closing_price
```

Теперь мы можем начать задавать более сложные вопросы, например, каковы наибольшие и наименьшие однодневные процентные изменения в нашем наборе данных? Процентное изменение равно `price_today/price_yesterday - 1`, т. е. нам нужен какой-то способ связать сегодняшнюю цену и вчерашнюю цену. Один из подходов — сгруппировать цены по символу, а затем внутри каждой группы:

1. Упорядочить цены по дате.
2. Применить функцию `zip` для получения пар (предыдущая, текущая).
3. Превратить пары в новые строки с "процентным изменением".

Начнем с группировки цен по символу:

```
from typing import List
from collections import defaultdict
```

```
# Собрать цены по символу
prices: Dict[str, List[StockPrice]] = defaultdict(list)

for sp in data:
    prices[sp.symbol].append(sp)
```

Поскольку цены являются кортежами, они будут отсортированы по своим полям в следующем порядке: сначала по символу, затем по дате, затем по цене. Это означает, что если у нас есть несколько цен с одним и тем же символом, то функция `sort` будет сортировать их по дате (а затем по цене, где она ничего не будет делать, т. к. у нас только одна дата), что мы и хотим.

```
# Упорядочить цены по дате
prices = {symbol: sorted(symbol_prices)
          for symbol, symbol_prices in prices.items()}
```

Эти цены мы можем использовать для вычисления последовательности изменений день ко дню.

```
def pct_change(yesterday: StockPrice, today: StockPrice) -> float:
    return today.closing_price / yesterday.closing_price - 1

class DailyChange(NamedTuple):
    symbol: str
    date: datetime.date
    pct_change: float

def day_over_day_changes(prices: List[StockPrice]) -> List[DailyChange]:
    """Предполагает, что цены только для одной акции и упорядочены"""
    return [DailyChange(symbol=today.symbol,
                        date=today.date,
                        pct_change=pct_change(yesterday, today))
            for yesterday, today in zip(prices, prices[1:])]
```

И затем собрать их все:

```
all_changes = [change
               for symbol_prices in prices.values()
               for change in day_over_day_changes(symbol_prices)]
```

И в этом случае легко отыскать самую высокую и самую низкую цену:

```
    max_change = max(all_changes, key=lambda change: change.pct_change)
    # См. пример http://news.cnet.com/2100-1001-202143.html
    assert max_change.symbol == 'AAPL'
    assert max_change.date == datetime.date(1997, 8, 6)
    assert 0.33 < max_change.pct_change < 0.34

    min_change = min(all_changes, key=lambda change: change.pct_change)
    # См. пример http://money.cnn.com/2000/09/29/markets/techwrap/
    assert min_change.symbol == 'AAPL'
```

```
    assert min_change.date == datetime.date(2000, 9, 29)
    assert -0.52 < min_change.pct_change < -0.51
```

Теперь мы можем использовать этот новый набор данных `all_changes` для того, чтобы отыскать месяц, в котором лучше всего инвестировать в технологические акции. Мы просто посмотрим на среднедневное изменение по месяцу:

```
changes_by_month: List[DailyChange] = {month: [] for month in range(1, 13)}

for change in all_changes:
    changes_by_month[change.date.month].append(change)

avg_daily_change = {
    month: sum(change.pct_change for change in changes) / len(changes)
    for month, changes in changes_by_month.items()
}

# Октябрь - лучший месяц
assert avg_daily_change[10] == max(avg_daily_change.values())
```

Мы будем делать такие манипуляции на протяжении всей книги, правда, не привлекая к ним слишком много внимания.

Шкалирование

Многие технические решения чувствительны к *шкале* данных. Например, представим, что у нас набор данных, состоящий из информации о росте и массе сотен исследователей данных, и нужно выявить *кластеры* размеров тела.

Интуитивно мы хотели, чтобы кластеры представляли точки, расположенные рядом друг с другом, а значит, потребуется некоторое представление о расстоянии между точками. У нас уже есть функция евклидова расстояния `distance`, поэтому совершенно естественно рассматривать пары в формате `(рост, масса)` как точки в двумерном пространстве. Посмотрим на данные нескольких человек, представленные в табл. 10.1.

***Таблица 10.1.** Данные о росте и массе*

Лицо	Рост, дюймы	Рост, сантиметры	Масса, фунты
A	63	160	150
B	67	170.2	160
C	70	177.8	171

Если измерять рост в дюймах, то ближайшим соседом *B* будет *A*:

```
from scratch.linear_algebra import distance

a_to_b = distance([63, 150], [67, 160])        # 10.77
a_to_c = distance([63, 150], [70, 171])        # 22.14
b_to_c = distance([67, 160], [70, 171])        # 11.40
```

Однако если измерять в сантиметрах, то, напротив, ближайшим соседом *B* будет *C*:

```
a_to_b = distance([160, 150], [170.2, 160])    # 14.28
a_to_c = distance([160, 150], [177.8, 171])    # 27.53
b_to_c = distance([170.2, 160], [177.8, 171])  # 13.37
```

Очевидно, задача трудно разрешима, если смена единиц измерения приводит к таким результатам. По этой причине в тех случаях когда размерности друг с другом не сопоставимы, данные иногда *шкалируют*, в результате чего каждая размерность имеет нулевое среднее значение и стандартное отклонение, равное 1. Преобразование каждой размерности в "стандартные отклонения от среднего значения" позволяет фактически избавиться от единиц измерения.

Для начала нам нужно вычислить среднее `mean` и стандартное отклонение `standard_deviation`:

```
from typing import Tuple

from scratch.linear_algebra import vector_mean
from scratch.statistics import standard_deviation

def scale(data: List[Vector]) -> Tuple[Vector, Vector]:
    """Возвращает среднее значение и стандартное отклонение
       для каждой позиции"""
    dim = len(data[0])

    means = vector_mean(data)
    stdevs = [standard_deviation([vector[i] for vector in data])
              for i in range(dim)]

    return means, stdevs

vectors = [[-3, -1, 1], [-1, 0, 1], [1, 1, 1]]
means, stdevs = scale(vectors)
assert means == [-1, 0, 1]
assert stdevs == [2, 1, 0]
```

И затем применить эту функцию для создания нового набора данных:

```
def rescale(data: List[Vector]) -> List[Vector]:
    """
    Шкалирует входные данные так, чтобы каждый столбец
    имел нулевое среднее значение и стандартное отклонение, равное 1
    (оставляет позицию как есть, если ее стандартное отклонение равно 0)
    """
    dim = len(data[0])
    means, stdevs = scale(data)

    # Сделать копию каждого вектора
    rescaled = [v[:] for v in data]
```

```
    for v in rescaled:
        for i in range(dim):
            if stdevs[i] > 0:
                v[i] = (v[i] - means[i]) / stdevs[i]

    return rescaled
```

Конечно же, давайте напишем тест для подтверждения того, что функция `rescale` делает то, что мы запланировали:

```
means, stdevs = scale(rescale(vectors))
assert means == [0, 0, 1]
assert stdevs == [1, 1, 0]
```

Как всегда, нужно руководствоваться здравым смыслом. Если взять гигантский набор данных о росте и массе людей и отфильтровать его, сведя к росту между 69.5 и 70.5 дюймами, то вполне вероятно, что (в зависимости от вопроса, на который вы пытаетесь ответить) оставшаяся вариация будет *шумом*, и вы, возможно, захотите поставить его стандартное отклонение в равное положение с отклонениями других размерностей.

Ремарка: библиотека tqdm

Нередко мы в конечном итоге делаем вычисления, которые занимают много времени. При выполнении такой работы хорошо бы знать, что вы продвигаетесь вперед и как долго вы должны ожидать окончания процесса.

Один из способов сделать это — использовать библиотеку tqdm, которая генерирует пользовательские индикаторы выполнения. Мы будем применять ее в остальной части книги, поэтому давайте узнаем, как она работает.

Для начала вам нужно будет ее установить:

```
python -m pip install tqdm
```

У этой библиотеки имеется всего несколько функциональных особенностей, о которых вам нужно знать. Во-первых, что итерируемый объект, завернутый в `tqdm.tqdm`, будет производить индикатор выполнения:

```
import tqdm

for i in tqdm.tqdm(range(100)):
    # Делать что-то медленное
    _ = [random.random() for _ in range(1000000)]
```

который порождает результат, как показано ниже:

```
56%|                                        | 56/100 [00:08<00:06, 6.49it/s]
```

В частности, он демонстрирует вам, какая часть вашего цикла выполнена (хотя он не сможет этого сделать, если вы используете генератор), как долго он работает и как долго он продолжит выполняться.

В этом случае (где мы просто оборачиваем вызов в диапазон `range`) вы можете просто использовать `tqdm.trange`.

Вы также можете задать описание индикатора выполнения во время его работы. Для этого вам нужно захватить итератор `tqdm` в инструкции `with`:

```
from typing import List

def primes_up_to(n: int) -> List[int]:
    primes = [2]

    with tqdm.trange(3, n) as t:
        for i in t:
        # i является простым, если нет меньшего простого, которое делит его
            i_is_prime = not any(i % p == 0 for p in primes)
            if i_is_prime:
                primes.append(i)
            t.set_description(f"{len(primes)} простых")

    return primes

my_primes = primes_up_to(100_000)
```

Приведенный выше фрагмент кода добавляет описание, подобное приведенному ниже, со счетчиком, который обновляется по мере обнаружения новых простых чисел:

```
5116 primes: 50%|                    | 49529/99997 [00:03<00:03, 15905.90it/s]
```

Использование библиотеки tqdm может сделать ваш код капризным — иногда экран перерисовывается плохо, а иногда цикл просто зависает. И если вы случайно обернете цикл `tqdm` внутри другого цикла `tqdm`, то могут произойти странные вещи. Как правило, преимущества указанной библиотеки перевешивают эти недостатки, поэтому мы попробуем использовать ее всякий раз, когда у нас будут медленные вычисления.

Снижение размерности

Иногда "фактические" (или полезные) размерности данных могут не соответствовать имеющимся размерностям. Например, рассмотрим набор данных, изображенный на рис. 10.6.

Бóльшая часть вариации в данных, по всей видимости, проходит вдоль одной-единственной размерности, которая не соответствует ни оси x, ни оси y.

В таких случаях применяется техническое решение, именуемое анализом главных компонент (principal component analysis, PCA), который используется для извлечения одной или более размерностей, улавливающих как можно больше вариаций в данных.

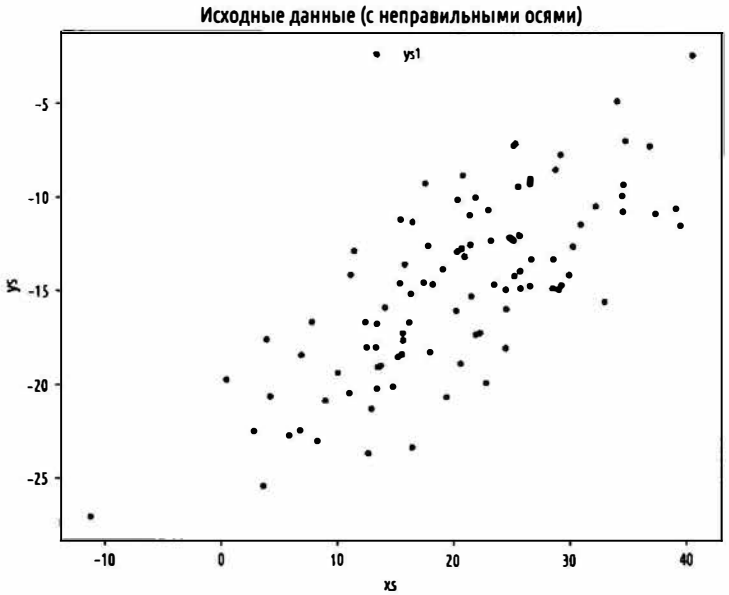


Рис. 10.6. Данные с "неправильными" осями

На практике вы не будете применять это техническое решение на наборе данных такой малой размерности. Снижение размерности в основном целесообразно применять, когда ваш набор данных имеет крупное число размерностей, и вы хотите отыскать небольшое подмножество, которое улавливает большинство вариаций. К сожалению, такие случаи трудно проиллюстрировать в двумерном формате книги.

В качестве первого шага нам нужно транслировать данные так, чтобы каждая размерность имела нулевое среднее:

```
from scratch.linear_algebra import subtract

def de_mean(data: List[Vector]) -> List[Vector]:
    """Перецентрировать данные, чтобы иметь среднее,
       равное 0, в каждой размерности"""
    mean = vector_mean(data)
    return [subtract(vector, mean) for vector in data]
```

(Если этого не сделать, то наше техническое решение, скорее всего, будет выявлять само среднее значение, а не вариацию в данных.)

На рис. 10.7 представлены данные из примера после вычитания среднего значения.

Теперь, имея в распоряжении центрированную матрицу `X`, можно задаться вопросом: какое направление улавливает наибольшее значение дисперсии в данных?

В частности, с учетом направления `d` (вектора магнитудой 1) каждая строка `x` в матрице простирается в размере `dot(x, d)` в направлении `d`. А каждый ненулевой вектор `w` определяет направление, если его прошкалировать так, чтобы он имел магнитуду, равную 1:

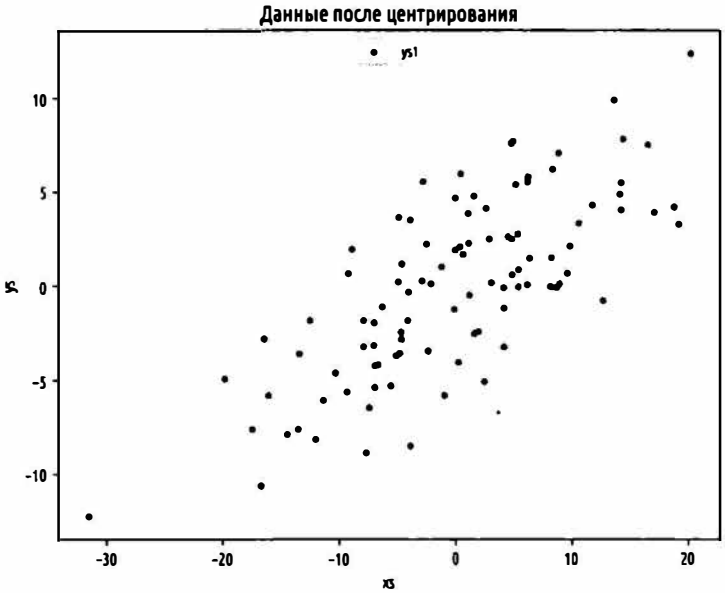


Рис. 10.7. Данные после вычитания среднего значения (центрирования)

```
from scratch.linear_algebra import magnitude

def direction(w: Vector) -> Vector:
    mag = magnitude(w)
    return [w_i / mag for w_i in w]
```

Следовательно, с учетом ненулевого вектора `w` мы можем вычислить дисперсию набора данных в направлении, определяемом вектором `w`:

```
from scratch.linear_algebra import dot

def directional_variance(data: List[Vector], w: Vector) -> float:
    """Возвращает дисперсию x в направлении w"""
    w_dir = direction(w)
    return sum(dot(v, w_dir) ** 2 for v in data)
```

Мы хотели бы отыскать направление, которое максимизирует эту дисперсию. Мы можем сделать это с помощью градиентного спуска, как только у нас будет функция градиента:

```
def directional_variance_gradient(data: List[Vector], w: Vector) -> Vector:
    """Градиент направленной дисперсии по отношению к w"""
    w_dir = direction(w)
    return [sum(2 * dot(v, w_dir) * v[i] for v in data)
            for i in range(len(w))]
```

И теперь первая главная компонента, которая у нас есть, будет направлением, которое максимизирует функцию `directional_variance`:

```
from scratch.gradient_descent import gradient_step

def first_principal_component(data: List[Vector],
                              n: int = 100,
                              step_size: float = 0.1) -> Vector:
    # Начать со случайной догадки
    guess = [1.0 for _ in data[0]]
    with tqdm.trange(n) as t:
        for _ in t:
            dv = directional_variance(data, guess)
            gradient = directional_variance_gradient(data, guess)
            guess = gradient_step(guess, gradient, step_size)
            t.set_description(f"dv: {dv:.3f}")

    return direction(guess)
```

На центрированном наборе данных эта функция возвращает направление `[0.924, 0.383]`, которое со всей очевидностью улавливает первичную ось, вдоль которой наши данные варьируют (рис. 10.8).

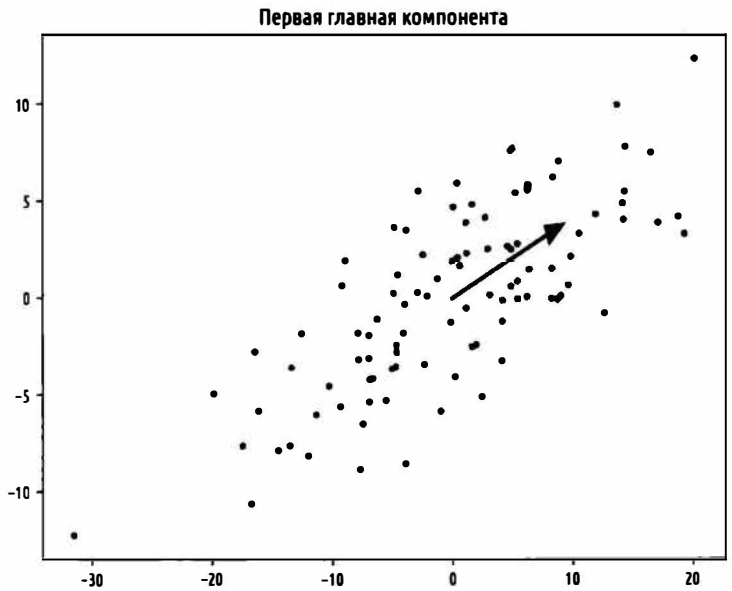


Рис. 10.8. Первая главная компонента

Найдя направление, т. е. первую главную компоненту, можно спроецировать на него данные, чтобы найти значения этой компоненты:

```
from scratch.linear_algebra import scalar_multiply

def project(v: Vector, w: Vector) -> Vector:
    """Вернуть проекцию v на направление w"""
```

```
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

Если вы хотите отыскать дальнейшие компоненты, то сначала следует удалить проекции из данных:

```
from scratch.linear_algebra import subtract

def remove_projection_from_vector(v: Vector, w: Vector) -> Vector:
    """Проецирует v на w и вычитает результат из v"""
    return subtract(v, project(v, w))

def remove_projection(data: List[Vector], w: Vector) -> List[Vector]:
    return [remove_projection_from_vector(v, w) for v in data]
```

Поскольку в этом примере использован двумерный набор данных, то после удаления первой компоненты фактически останутся одномерные данные (рис. 10.9).

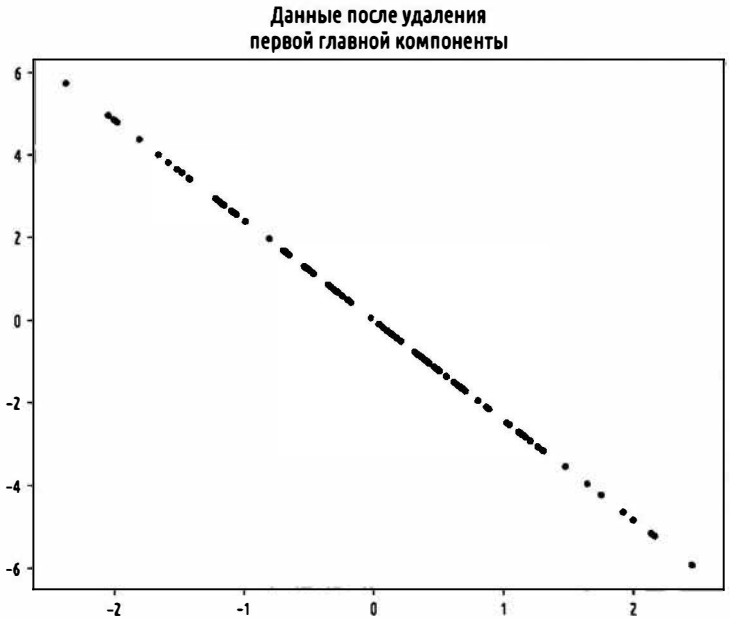


Рис. 10.9. Данные после удаления первой главной компоненты

В этом месте мы можем отыскать следующую главную компоненту, повторно применив процедуру к результату функции удаления проекции `remove_projection` (рис. 10.10).

На более высокоразмерных наборах данных мы можем итеративно отыскивать столько компонент, сколько потребуется:

```
# Анализ главных компонент
def pca(data: List[Vector], num_components: int) -> List[Vector]:
    components: List[Vector] = []
```

```
    for _ in range(num_components):
        component = first_principal_component(data)
        components.append(component)
        data = remove_projection(data, component)

    return components
```

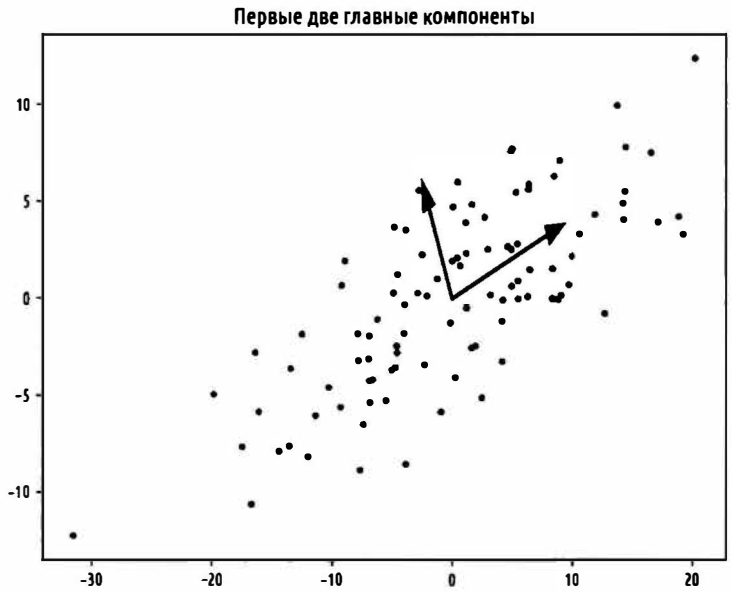


Рис. 10.10. Первые две главные компоненты

Затем мы можем *преобразовать* данные в более низкоразмерное пространство, охватываемое компонентами:

```
def transform_vector(v: Vector, components: List[Vector]) -> Vector:
    return [dot(v, w) for w in components]

def transform(data: List[Vector], components: List[Vector]) -> List[Vector]:
    return [transform_vector(v, components) for v in data]
```

Это техническое решение представляет ценность по нескольким причинам. Во-первых, оно помогает очистить данные за счет устранения шумовых размерностей и консолидации высококоррелированных размерностей.

Во-вторых, после извлечения низкоразмерного представления данных мы можем применить разнообразные технические решения, которые не работают на высокоразмерных данных. Мы увидим примеры таких решений в оставшейся части книги.

Хотя это техническое решение помогает строить более качественные модели, в то же время оно усложняет их интерпретацию. Выводы наподобие того, что "каждый дополнительный год стажа увеличивает зарплату в среднем на $10 000", абсолютно

понятны, что нельзя сказать о выводах типа "каждое увеличение на 0.1 единицу в третьей главной компоненте добавляет к зарплате в среднем $10 000".

Для дальнейшего изучения

- Как уже упоминалось в конце *главы 9*, библиотека pandas[2], вероятно, является самым важным инструментом Python для очистки, преобразования, оперирования и дальнейшей работы с данными. Все примеры, которые были созданы в этой главе вручную, можно реализовать гораздо проще с помощью pandas. Вероятно, наилучшим способом познакомиться с pandas является прочтение книги Уэса Маккинни "Python и анализ данных" (Python for Data Analysis)[3].
- Библиотека scikit-learn содержит целый ряд функций разложения матриц[4], включая анализ главных компонент (PCA).

[2] См. **http://pandas.pydata.org/**.

[3] См. **https://learning.oreilly.com/library/view/python-for-data/9781491957653/**.

[4] См. **https://scikit-learn.org/stable/modules/classes.html%23module-sklearn.decomposition**.

ГЛАВА 11
Машинное обучение

Всегда готов учиться, хотя не всегда нравится, когда меня учат.
– *Уинстон Черчилль*[1]

Многие полагают, что наука о данных в основном имеет дело с машинным обучением, и исследователи данных только и делают, что целыми днями строят, тренируют и настраивают автоматически обучающиеся модели. (Опять же, многие из этих людей *в действительности* не представляют, что такое машинное обучение.) По сути, наука о данных — это преимущественно сведение бизнес-проблем к проблемам в области данных, задачам сбора, понимания, очистки и форматирования данных, после чего машинное обучение идет как дополнение. Даже в этом виде оно является интересным и важным дополнением, в котором следует очень хорошо разбираться для того, чтобы решать задачи в области науки о данных.

Моделирование

Прежде чем начать обсуждать машинное обучение, следует поговорить о *моделях*.

Что такое модель? Это, в сущности, подробное описание математической (или вероятностной) связи, которая существует между различными величинами.

Например, если вы пытаетесь собрать денежные средства на социально-сетевой веб-сайт, то вы могли бы построить *бизнес-модель* (скорее всего, в электронной таблице), которая принимает такие входы, как "число пользователей", "выручка от рекламы в расчете на одного пользователя" и "число сотрудников", и возвращает годовую прибыль на ближайшие несколько лет. Рецепт из поваренной книги приводит к модели, которая соотносит такие входы, как "число едоков" и "аппетит", с количеством необходимых ингредиентов. А те, кто когда-либо смотрел по телевизору покер, знают, что "вероятность победы" каждого игрока оценивается в режиме реального времени на основе модели, которая учитывает карты, раскрытые на конкретный момент времени и распределение карт в колоде.

Бизнес-модель, вероятно, будет основана на простых математических соотношениях: доходы минус расходы, где доходы — это число проданных единиц, помножен-

[1] Сэр Уинстон Черчилль (1874–1965) — британский государственный и политический деятель, премьер-министр Великобритании в 1940–1945 и 1951–1955 гг.; военный (полковник), журналист, писатель. — *Прим. пер.*

ное на среднюю цену, и т. д. Модель поваренной книги, вероятно, будет основана на методе проб и ошибок — кто-то пошел на кухню и опробовал разные комбинации ингредиентов, пока не нашел то, что нравится. А модель карточной игры будет основана на теории вероятностей, правилах игры в покер и некоторых достаточно безобидных допущениях о случайностях в процессе раздачи карт.

Что такое машинное обучение?

У любого специалиста имеется собственное точное определение этого термина, но мы будем использовать термин *"машинное обучение"*, имея в виду создание и применение моделей, *усвоенных из данных*. В других контекстах это может называться *предсказательным моделированием* и *глубинным анализом данных*, но мы будем придерживаться термина "машинное обучение". Как правило, задача будет заключаться в том, чтобы использовать существующие данные для разработки моделей, которые можно применять для *предсказания* разных результатов в отношении новых данных, как, например, предсказание:

- является ли сообщение спамом или нет;
- является ли кредитно-карточная транзакция мошеннической;
- на каких рекламных сообщениях покупатель с наибольшей вероятностью будет кликать;
- какая футбольная команда выиграет Суперкубок.

Мы обратимся к контролируемым моделям (т. е. таким, где есть совокупность данных, помеченная правильными ответами, на которых проходит усвоение) и неконтролируемым моделям (где правильные ответы отсутствуют). Кроме них, существуют и другие типы моделей, такие как полуконтролируемые модели (где только некоторые данные помечены правильными ответами), онлайновые (в которых модель должна непрерывно самонастраиваться ко вновь поступающим данным) и подкрепляемые (где после серии предсказаний модель получает сигнал, указывающий, насколько хорошо она справилась), которые мы не будем рассматривать в этой книге.

При этом даже в самой простой ситуации существуют целые универсумы моделей, которые могли бы описывать связь, в которой мы заинтересованы. В большинстве случаев мы сами выбираем параметризованное семейство моделей, а затем используем данные для усвоения параметров, которые в определенной мере являются оптимальными.

Например, мы можем исходить из того, что рост человека (грубо) является линейной функцией его массы, и затем использовать данные для усвоения этой линейной функции. Либо мы можем исходить из того, что дерево решений является хорошим механизмом для диагностики заболеваний пациентов, и потом использовать данные для усвоения такого "оптимального" дерева. В остальной части книги мы займемся исследованием различных семейств моделей, которые мы можем усвоить.

Но прежде чем этим заняться, следует получше разобраться в основах машинного обучения. В оставшейся части главы мы обсудим несколько базовых понятий и только потом перейдем к самим моделям.

Переподгонка и недоподгонка

Общеизвестной ловушкой в машинном обучении является *переподгонка* (overfitting, т. е. излише плотное прилегание к тренировочным данным), производящая предсказательную модель, которая хорошо работает на данных из тренировочной выборки, но при этом плохо обобщает на любых новых данных. Она может быть связана с усвоением шума в данных. Либо она может быть связана с тем, что модель научилась выявлять специфические входы вместо любых факторов, которые на самом деле имеют предсказательный характер для желаемого выхода.

Другую сторону этого явления представляет *недоподгонка* (underfitting), которая производит предсказательную модель, не работающую даже на тренировочных данных. Впрочем, когда это случается, обычно делают вывод о том, что модель недостаточно хороша, и продолжают искать более подходящую[2].

На рис. 11.1 мы вписали три многочлена в выборку данных. (Не стоит сейчас озадачиваться тем, как это сделать; мы вернемся к этому в последующих главах.)

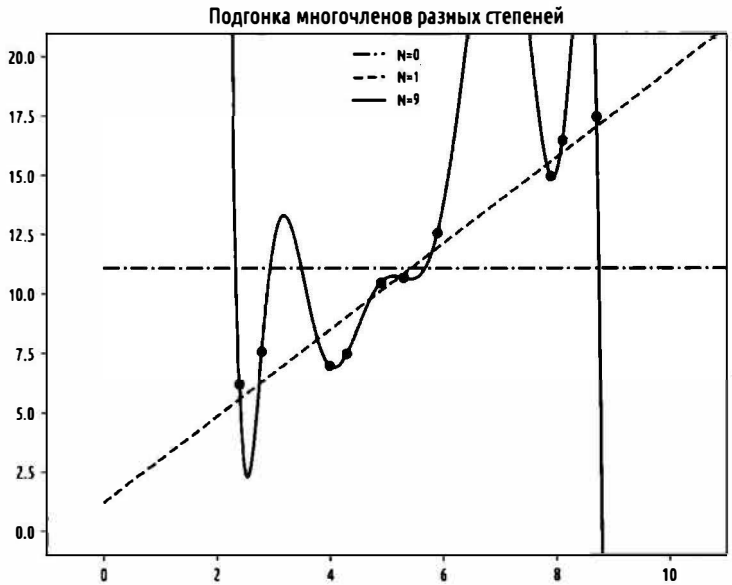


Рис. 11.1. Переподгонка и недооподгонка

[2] Простая аналогия поможет уяснить термины "переподгонка" и "недоподгонка". Представьте портного, шьющего один-единственный костюм (модель) для группы людей, которые имеют не сильно разнящиеся пропорции тела и рост (зачем он это делает, сейчас неважно). Слишком плотно прилегающий костюм (переподгонка), скорее всего, подойдет лишь нескольким из них, а слишком свободный (недоподгонка) не подойдет никому. — *Прим. пер.*

Горизонтальная прямая соответствует наилучшей подгонке многочлена 0-й степени (т. е. постоянного). Она крайне *недоподогнана*, т. е. недостаточно вписана в тренировочные данные. Наилучшая подгонка многочлена 9-й степени (т. е. из 10 параметров) проходит ровно через каждую точку тренировочных данных, но она крайне *переподогнана*, т. е. слишком плотно прилегает к тренировочным данным — если бы пришлось добавить еще несколько точек, то кривая, вполне вероятно, прошла бы мимо на большом от них удалении. А вот линия многочлена 1-й степени достигает хорошего равновесия — она находится довольно близко к каждой точке, и если данные являются репрезентативными, то прямая, скорее всего, будет находиться близко и к новым точкам данных.

Ясно, что слишком сложные модели приводят к переподгонке и не обобщают хорошо за пределами данных, на которых они тренировались. Тогда как сделать так, чтобы модели не были слишком сложными? Наиболее фундаментальный подход предусматривает использование разных данных для тренировки модели и для тестирования модели.

Простейший способ состоит в разбиении набора данных так, чтобы (например) две трети использовались для тренировки модели, после чего мы измеряем результативность модели на оставшейся трети:

```
import random
from typing import TypeVar, List, Tuple

X = TypeVar('X')  # Обобщенный тип для представления точки данных

def split_data(data: List[X], prob: float) -> Tuple[List[X], List[X]]:
    """Разбить данные на доли [prob, 1 - prob]"""
    data = data[:]                  # Сделать мелкую копию,
    random.shuffle(data)            # т. к. shuffle модифицирует список.
    cut = int(len(data) * prob)     # Применить prob для отыскания отсечения
    return data[:cut], data[cut:]  # и разбить там перетасованный список

data = [n for n in range(1000)]
train, test = split_data(data, 0.75)

# Пропорции должны быть правильными
assert len(train) == 750
assert len(test) == 250

# И исходные данные должны быть оставлены (в некоем порядке)
assert sorted(train + test) == data
```

Часто у нас имеются спаренные входные переменные. В этом случае необходимо обеспечить, чтобы соответствующие значения размещались вместе в тренировочных данных либо в тестовых данных:

```
Y = TypeVar('Y')   # Обобщенный тип для представления выходных переменных

def train_test_split(xs: List[X],
                     ys: List[Y],
                     test_pct: float) -> Tuple[List[X], List[X], List[Y],
                                               List[Y]]:
    # Сгенерировать индексы и разбить их
    idxs = [i for i in range(len(xs))]
    train_idxs, test_idxs = split_data(idxs, 1 - test_pct)

    return ([xs[i] for i in train_idxs],  # тренировка x_train
            [xs[i] for i in test_idxs],   # тест x_test
            [ys[i] for i in train_idxs],  # тренировка y_train
            [ys[i] for i in test_idxs])   # тест y_test
```

Как всегда, мы хотим удостовериться, что наш код работает, как надо:

```
xs = [x for x in range(1000)]      # xs - это 1 ... 1000
ys = [2 * x for x in xs]           # каждый y_i равен удвоенному x_i
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.25)

# Проверить, что пропорции являются правильными
assert len(x_train) == len(y_train) == 750
assert len(x_test) == len(y_test) == 250

# Проверить, что соответствующие точки данных спарены правильно
assert all(y == 2 * x for x, y in zip(x_train, y_train))
assert all(y == 2 * x for x, y in zip(x_test, y_test))
```

После чего вы можете сделать что-то вроде этого:

```
model = SomeKindOfModel()                    # Создать модель
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33) # Разбить данные
model.train(x_train, y_train)                # Натренировать модель
performance = model.test(x_test, y_test)     # Протестировать, получив
                                             # результативность
```

Если модель была переподогнана к тренировочным данным, то можно с уверенностью сказать, что она покажет очень слабые результаты на (совершенно отдельных) тестовых данных. Говоря иначе, если она хорошо работает на тестовых данных, то вы можете быть больше уверены в том, что она подогнана нежели *переподогнана*.

Тем не менее в ряде случаев всё может пойти не так.

Первый случай: если в тестовых и тренировочных данных имеются общие регулярности, которые не обобщаются на более крупный набор данных.

Например, представим, что набор данных состоит из действий пользователей, по одной строке на каждого пользователя в неделю. В таком случае большинство пользователей будут появляться как в тренировочных, так и в тестовых данных, и определенные модели могут начать *выявлять* пользователей, а не раскрывать свя-

зи с участием *атрибутов*. Эта ситуация не приводит к большим неприятностям, хотя однажды мне пришлось с ней столкнуться.

Второй случай представляет более значительную проблему, если вы используете разбивку на тестовые и тренировочные данные не только для того, чтобы судить о модели, но и для того, чтобы *выбирать* среди многих моделей. В этом случае, несмотря на то что каждая отдельная модель, возможно, не будет переподогнанной, "выбор модели, которая показывает наилучшие результаты на тестовом наборе данных", является метатренировкой, которая делает функцию тестового набора вторым тренировочным набором. (Разумеется, модель, которая показала наилучшие результаты на тестовом наборе, продолжит показывать наилучшие результаты на тестовом наборе.)

В такой ситуации необходимо разбить данные на три части: *тренировочный* набор для построения моделей, *перекрестно-контрольный* набор для выбора среди натренированных моделей и *тестовый* набор для оценивания окончательной модели.

Правильность, точность и прецизионность

Когда я не занимаюсь наукой о данных, то люблю копаться в медицине. В свободное время я придумал дешевый и неинвазивный тест, который можно проводить у новорожденных. Он предсказывает предрасположенность новорожденного к развитию лейкоза с точностью свыше 98%. Мой адвокат убедил меня, что указанный тест не является патентоспособным, и поэтому я поделюсь подробностями: модель предсказывает лейкоз, если и только если ребенка зовут Люк (звучит похоже).

Как мы увидим ниже, этот тест действительно более чем на 98% точен. И тем не менее он невероятно дурацкий, являясь хорошей иллюстрацией того, почему обычно критерий "точности" (accuracy) не используется для измерения качества (бинарно-классификационной) модели.

Представим, что строится модель, которая должна формулировать *бинарные* суждения. Является ли это письмо спамом? Следует ли нанять этого претендента? Является ли этот авиапассажир скрытым террористом?

При наличии набора помеченных данных и такой предсказательной модели каждая точка данных принадлежит одной из четырех категорий суждений:

- истинное утверждение: "это сообщение спамное, и его спамность была предсказана правильно";
- ложное утверждение (ошибка 1-го рода): "это сообщение не спамное, однако была предсказана его спамность";
- ложное отрицание (ошибка 2-го рода): "это сообщение спамное, однако была предсказана его неспамность";
- истинное отрицание: "это сообщение не спамное, и его неспамность была предсказана правильно".

Эти категории нередко представляют в качестве количеств в *матрице несоответствий* (табл. 11.1).

Таблица 11.1. Матрица несоответствий

	Спам	Не спам
Предсказано "спам"	Истинное утверждение	Ложное утверждение
Предсказано "не спам"	Ложное отрицание	Истинное отрицание

Давайте посмотрим, как мой тест на лейкоз вписывается в эти рамки. В наше время приблизительно 5% младенцам из 1000 дают имя Люк[3], а распространенность лейкоза на протяжении жизни составляет около 1,4%, или 14 человек на каждую 1000[4].

Если мы считаем, что эти два фактора являются взаимно независимыми и применим мой тест "Люк означает лейкоз" к 1 млн человек, то можно ожидать, что мы увидим матрицу несоответствий, которая выглядит так, как табл. 11.2.

Таблица 11.2. Матрица несоответствий для теста

	Лейкоз	Не лейкоз	Всего
"Люк"	70	4930	5000
"Не Люк"	13 930	981 070	995 000
Всего	14 000	986 000	1 000 000

Затем мы можем использовать эти данные для вычисления разнообразных статистик в отношении результативности модели. Например, показатель *точности* (accuracy) определяется как доля правильных предсказаний:

```
def accuracy(tp: int, fp: int, fn: int, tn: int) -> float:
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

assert accuracy(70, 4930, 13930, 981070) == 0.98114
```

И, как можно убедиться, он показывает впечатляющие результаты. Однако совершенно очевидно, что этот тест негодный, а значит, вероятно, нам совсем не стоит доверять такому грубому показателю.

В соответствии со стандартной практикой принято обращаться к сочетанию *прецизионности* (precision) и *полноты* (recall). При этом здесь прецизионность измеряет то, насколько точными были наши утвердительные предсказания:

```
def precision(tp: int, fp: int, fn: int, tn: int) -> float:
    return tp / (tp + fp)

assert precision(70, 4930, 13930, 981070) == 0.014
```

А полнота измеряет долю утвердительных предсказаний, которую наша модель выявила:

---

[3] См. **https://www.babycenter.com/baby-names-luke-2918.htm**.

[4] См. **https://seer.cancer.gov/statfacts/html/leuks.html**.

```
def recall(tp: int, fp: int, fn: int, tn: int) -> float:
    return tp / (tp + fn)

assert recall(70, 4930, 13930, 981070) == 0.005
```

Оба результата являются ужасными и отражают тот факт, что сама модель является ужасной.

В некоторых случаях прецизионность и полнота объединяются в отметку *F1*, которая определяется следующим образом:

```
def f1_score(tp: int, fp: int, fn: int, tn: int) -> float:
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)

    return 2 * p * r / (p + r)
```

Она представляет собой *гармоническое среднее значение*[5] прецизионности и полноты и с неизбежностью лежит между ними.

Обычно подбор модели предусматривает компромисс между прецизионностью и полнотой. Модель, которая предсказывает "да", когда она даже чуть-чуть уверена, будет иметь высокую полноту и низкую прецизионность, тогда как модель, которая предсказывает "да", только когда она крайне уверена, вероятно, будет иметь низкую полноту и высокую прецизионность.

С другой стороны, вы можете думать об этом как о компромиссе между ложными утверждениями и ложными отрицаниями. Ответ "да" слишком часто будет давать много ложных утверждений, а ответ "нет" — много ложных отрицаний.

Предположим, что имеется 10 факторов для лейкемии, и чем больше их выявлено у человека, тем больше шансов, что у него должен развиться лейкоз. В этом случае вы можете представить континуум тестов: "предсказать лейкоз при не менее одном рисковом факторе", "предсказать лейкоз при не менее двух рисковых факторах" и т. д. Если вы увеличиваете порог, то увеличиваете прецизионность теста (поскольку люди с большим числом рисковых факторов более склонны к развитию заболевания) и уменьшаете полноту теста (поскольку все меньше и меньше окончательно заболевших будут удовлетворять порогу). В таких случаях выбор правильного порога становится вопросом отыскания верного компромисса.

Компромисс между смещением и дисперсией

Еще один способ думать о проблеме переподгонки — рассматривать ее как компромисс между смещением и дисперсией.

Оба показателя измеряют то, что произойдет, если вы будете тренировать свою модель многократно на разных наборах тренировочных данных (из той же самой более крупной популяции).

---

[5] См. **https://en.wikipedia.org/wiki/Harmonic_mean, https://ru.wikipedia.org/wiki/Среднее_гармоническое.**

Например, модель с многочленом 0-й степени из *разд. "Переподгонка и недоподгонка" данной главы* сделает достаточно много ошибок практически для любого тренировочного набора (взятого из той же самой популяции), а значит, имеет высокое *смещение*. Однако два любых случайно выбранных тренировочных набора должны дать вполне похожие модели (поскольку два любых случайно выбранных тренировочных набора должны иметь вполне похожие средние значения). И поэтому мы говорим, что модель имеет низкую *дисперсию*. Высокое смещение и низкая дисперсия, как правило, соответствуют недоподгонке.

С другой стороны, модель с полиномом 9-й степени идеально вписывается в тренировочный набор. Она имеет очень низкое смещение, но очень высокую дисперсию (поскольку два любых тренировочных набора, скорее всего, породят очень разные модели). Это соответствует переподгонке.

Обдумывание модельных задач, таким образом, помогает выяснить, что делать, когда модель не работает, как надо.

Если ваша модель имеет высокое смещение (и значит, дает слабые результаты даже на тренировочных данных), можно попробовать *добавить* больше признаков. Переход от модели с многочленом 0-й степени к модели с многочленом 1-й степени в *разд. "Переподгонка и недоподгонка"* дал существенное улучшение.

Если же ваша модель имеет высокую дисперсию, то схожим образом вы можете *удалить* признаки. Впрочем, еще одно решение состоит в том, чтобы добыть больше данных (если это возможно).

На рис. 11.2 мы вписываем многочлен 9-й степени в выборки разных размеров. Подгонка модели на 10 точках данных разбросана по всей площади, как мы видели

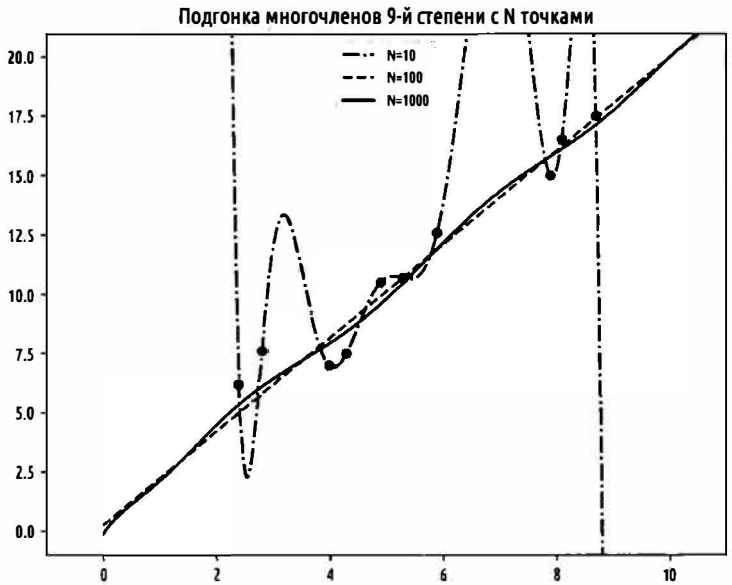


Рис. 11.2. Снижение дисперсии с помощью дополнительных данных

ранее. Если, напротив, мы тренируем модель на 100 точках данных, то имеется гораздо меньше переподгонки. И модель, натренированная на 1000 точках данных, будет выглядеть очень похожей на модель 1-й степени. Если удерживать сложность модели постоянной, то чем больше у вас данных, тем труднее достигнуть переподгонки. С другой стороны, дополнительные данные не помогут со смещением. Если ваша модель не использует достаточно признаков для улавливания регулярностей в данных, то вбрасывание дополнительных данных не поможет.

Извлечение и отбор признаков

Как уже упоминалось, когда данные не содержат достаточного числа признаков, ваша модель, скорее всего, будет недоподогнана. Когда же признаков слишком много, она легко достигает переподгонки. Но что такое признаки и откуда они берутся?

Признаки — это любые входные данные, которые передаются в модель.

В простейшем случае признаки просто вам даны. Если нужно предсказать зарплату на основе многолетнего опыта человека, то число лет будет единственным располагаемым признаком. (Впрочем, как мы видели в *разд. "Переподгонка и недоподгонка" данной главы*, помимо этого можно попробовать добавить опыт в квадрате, в кубе и т. д., если это помогает вам построить более качественную модель.)

Все становится интереснее по мере усложнения данных. Представим, что строится спам-фильтр, который должен предсказывать, является ли сообщение электронной почты нежелательным или нет. Большинство моделей не знают, что делать с сырым почтовым сообщением, которое представляет собой лишь коллекцию текста. Вам придется извлечь признаки. Например:

- Присутствует ли в почтовом сообщении слово "виагра"?
- Сколько раз появляется буква d?
- Каков домен отправителя?

Ответ на вопрос, похожий на первый из перечисленных выше, будет содержать только "да" или "нет", которые обычно кодируются как 1 или 0; второй будет числом, а третий — альтернативой из дискретного набора вариантов.

Почти всегда мы будем извлекать из наших данных признаки, которые будут попадать в одну из этих трех категорий. Более того, тип признаков накладывает ограничения на тип моделей, которые мы можем использовать.

- Наивный байесов классификатор, который будет построен в *главе 13*, подходит для бинарных признаков (типа да/нет), как и первый в предыдущем списке.
- Регрессионные модели, которые мы изучим в *главах 14* и *16*, требуют численных признаков (которые могут включать фиктивные переменные, равные нулям и единицам).
- Деревья решений, к которым мы обратимся в *главе 17*, могут иметь дело как с численными, так и с категориальными данными.

Хотя в примере со спам-фильтром мы искали способы создания признаков, иногда мы, напротив, ищем способы удаления признаков.

Например, входами могут являться векторы из нескольких сотен чисел. В зависимости от ситуации, возможно, будет целесообразнее выжать из них горстку важных размерностей (как в *разд. "Снижение размерности" главы 10*) и использовать только это малое число признаков. Либо применять техническое решение (подобное регуляризации, к которой мы обратимся в *разд. "Регуляризация" главы 15*), которое штрафует модели тем больше, чем больше признаков в них используется.

Как выполнять отбор признаков? Здесь в игру вступает сочетание *опыта* и *компетенции* в предметной области. Если вы получили много электронных писем, то, вероятно, у вас появится некое представление о том, что присутствие некоторых слов может быть хорошим индикатором спамности. А также о том, что число появлений буквы d в тексте, скорее всего, не является его хорошим индикатором. Но в целом необходимо опробовать разные подходы, что является частью игры.

Для дальнейшего изучения

- Продолжайте читать! Следующие несколько глав посвящены различным семействам автоматически обучающихся моделей.
- Курс по машинному обучению[6] в рамках онлайновой платформы Courser Стэндфордского университета является массовым открытым онлайновым курсом (МООК) и хорошей площадкой для более глубокого постижения основ машинного обучения.
- Учебник "Элементы статистического обучения" (The Elements of Statistical Learning) Джерома Х. Фридмана (Jerome H. Friedman) и соавт. является каноническим пособием, которое можно скачать бесплатно[7]. Но будьте осторожны: там *много* математики.

[6] См. **https://www.coursera.org/course/ml**.

[7] См. **http://stanford.io/1ycOXbo**.

ГЛАВА 12

k ближайших соседей

Если желаешь досадить соседям, расскажи им правду о них.
– *Пьетро Аретино*[1]

Представим, что вы пытаетесь предсказать, как я буду голосовать на следующих президентских выборах. Если вы больше ничего обо мне не знаете (кроме данных о месте проживания), то целесообразно посмотреть на то, как собираются голосовать мои *соседи*. Живя, как и я, в центре Сиэтла, они неизменно планируют голосовать за демократического кандидата, что позволяет с большой долей уверенности предположить, что я тоже сделаю свой выбор в пользу кандидата от демократов.

Теперь, предположим, обо мне известно больше, чем просто география — возможно, известны мой возраст, доход, сколько у меня детей и т. д. В той части, в которой мое поведение находится под влиянием этих вещей (или характеризуется ими), учет только моих соседей, которые близки мне среди всех этих размерностей, по-видимому, будет даже более качественным предсказанием, чем учет всех моих соседей.. В этом и заключается основная идея *классификации по ближайшим соседям*.

Модель

Ближайшие соседи — это одна из простейших предсказательных моделей, которые существуют. В ней не делается никаких математических допущений и не требуется какой-то тяжелый вычислительный механизм. Единственное, что нужно, — это:

- некоторая идея о расстоянии;
- допущение, что точки, расположенные друг к другу близко, подобны.

Большинство технических приемов, которые мы увидим в этой книге, обращаются к набору данных как целому для усвоения регулярностей. С другой стороны, метод ближайших соседей вполне осознанно пренебрегает значительной частью информации, поскольку предсказание для любой новой точки зависит лишь от горстки ближайших к ней точек.

---

[1] Пьетро Аретино (1492–1556) — итальянский писатель Позднего Ренессанса, сатирик, публицист и драматург, считающийся некоторыми исследователями предтечей и основателем европейской журналистики — *Прим. пер.*

Более того, ближайшие соседи, скорее всего, не помогут понять движущие силы любого явления, которое вы рассматриваете. Предсказывая мое голосование на выборах, основываясь на голосовании моих соседей, вы мало что получите о том, что́ заставляет меня голосовать именно таким образом, тогда как некая альтернативная модель, которая предсказывает голосование, основываясь (допустим) на моем доходе и семейном положении, очень даже может это понять.

В общей ситуации, у нас есть несколько точек данных и соответствующий набор меток. Метки могут быть `True` и `False`, указывая на то, удовлетворяет ли каждый вход некоему условию, такому как "спам?", или "ядовитый?", или "приятный для просмотра?". Либо они могут быть категориями, такими как рейтинги фильмов (G, PG, PG-13, R, NC-17), либо именами кандидатов в президенты. Либо они могут быть предпочтительными языками программирования.

В нашем случае точки данных будут векторами, а это означает, что мы можем воспользоваться функцией расстояния `distance` из *главы 4*.

Скажем, мы выбрали число k, равное 3 или 5. Когда мы хотим классифицировать некую новую точку данных, мы отыскиваем k ближайших помеченных точек и даем им проголосовать за новый результат.

Для этого потребуется функция, которая подсчитывает голоса. Одна из возможностей заключается в следующем:

```
from typing import List
from collections import Counter

def raw_majority_vote(labels: List[str]) -> str:
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner

assert raw_majority_vote(['a', 'b', 'c', 'b']) == 'b'
```

Однако она не делает ничего разумного в ситуации равного числа голосов. Например, допустим, мы назначаем фильмам рейтинги, и пять ближайших фильмов получают рейтинги G, G, PG, PG и R. Тогда G имеет два голоса, и PG тоже имеет два голоса. В таком случае у нас несколько вариантов:

- выбрать одного из победителей случайно;
- взвесить голоса по расстоянию и выбрать взвешенного победителя;
- уменьшать k до тех пор, пока не будет найден уникальный победитель.

Мы выполним реализацию третьего варианта:

```
def majority_vote(labels: List[str]) -> str:
    """Исходит из того, что метки упорядочены
       от ближайшей до самой удаленной"""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
```

```
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])
    if num_winners == 1:
        return winner  # уникальный победитель, поэтому вернуть его
    else:
        return majority_vote(labels[:-1]) # попытаться снова
                                          # без самой удаленной

# Равное число голосов, поэтому взять первые 4, затем 'b'
assert majority_vote(['a', 'b', 'c', 'b', 'a']) == 'b'
```

Такой подход обязательно в конце концов сработает, т. к. в худшем случае мы все равно придем только к одной метке, которая и будет победителем.

При помощи этой функции можно легко создать классификатор:

```
from typing import NamedTuple
from scratch.linear_algebra import Vector, distance

class LabeledPoint(NamedTuple):
    point: Vector
    label: str

def knn_classify(k: int,
                 labeled_points: List[LabeledPoint],
                 new_point: Vector) -> str:

    # Упорядочить помеченные точки от ближайшей до самой дальней
    by_distance = sorted(labeled_points,
                         key=lambda lp: distance(lp.point, new_point))

    # Отыскать метки для k ближайших
    k_nearest_labels = [lp.label for lp in by_distance[:k]]

    # И дать им проголосовать
    return majority_vote(k_nearest_labels)
```

Посмотрим, как этот классификатор работает.

Пример: набор данных о цветках ириса

Набор данных о цветках ириса (Iris) является основным ингредиентом машинного обучения. Он содержит горстку результатов измерений 150 цветков, представляющих три вида ириса. Для каждого цветка у нас есть длина лепестка, ширина лепестка, длина чашелистика и ширина чашелистика, а также его вид. Вы можете скачать этот набор данных с **https://archive.ics.uci.edu/ml/datasets/iris**:

```
import requests

data = requests.get(
  "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
)

with open('iris.data', 'w') as f:
    f.write(data.text)
```

Данные разделены запятыми, с полями "длина чашелистика", "ширина чашелистика", "длина лепестка", "ширина лепестка", "класс":

```
sepal_length, sepal_width, petal_length, petal_width, class
```

Например, первая строка выглядит так, обозначая ирис щетинистый:

```
5.1,3.5,1.4,0.2,Iris-setosa
```

В этом разделе мы попытаемся построить модель, которая может предсказывать класс (т. е. вид) из первых четырех результатов измерений.

Для начала давайте загрузим и разведаем данные. Наша функция ближайших соседей ожидает помеченную точку `LabeledPoint`, поэтому представим наши данные таким образом:

```
from typing import Dict
import csv
from collections import defaultdict

def parse_iris_row(row: List[str]) -> LabeledPoint:
    """длина чашелистика, ширина чашелистика, длина лепестка,
       ширина лепестка, класс"""
    measurements = [float(value) for value in row[:-1]]
    # Классом является, например, "Iris-virginica";
    # нам нужно просто "virginica" (виргинский)
    label = row[-1].split("-")[-1]

    return LabeledPoint(measurements, label)

with open('iris.data') as f:
    reader = csv.reader(f)
    iris_data = [parse_iris_row(row) for row in reader if len(row)>0]

# Мы также сгруппируем точки только по виду/метке,
# чтобы их можно было вывести на график
points_by_species: Dict[str, List[Vector]] = defaultdict(list)
for iris in iris_data:
    points_by_species[iris.label].append(iris.point)
```

Мы хотели бы вывести данные результатов измерений на график для того, чтобы можно было увидеть, как они различаются по видам. К сожалению, они являются

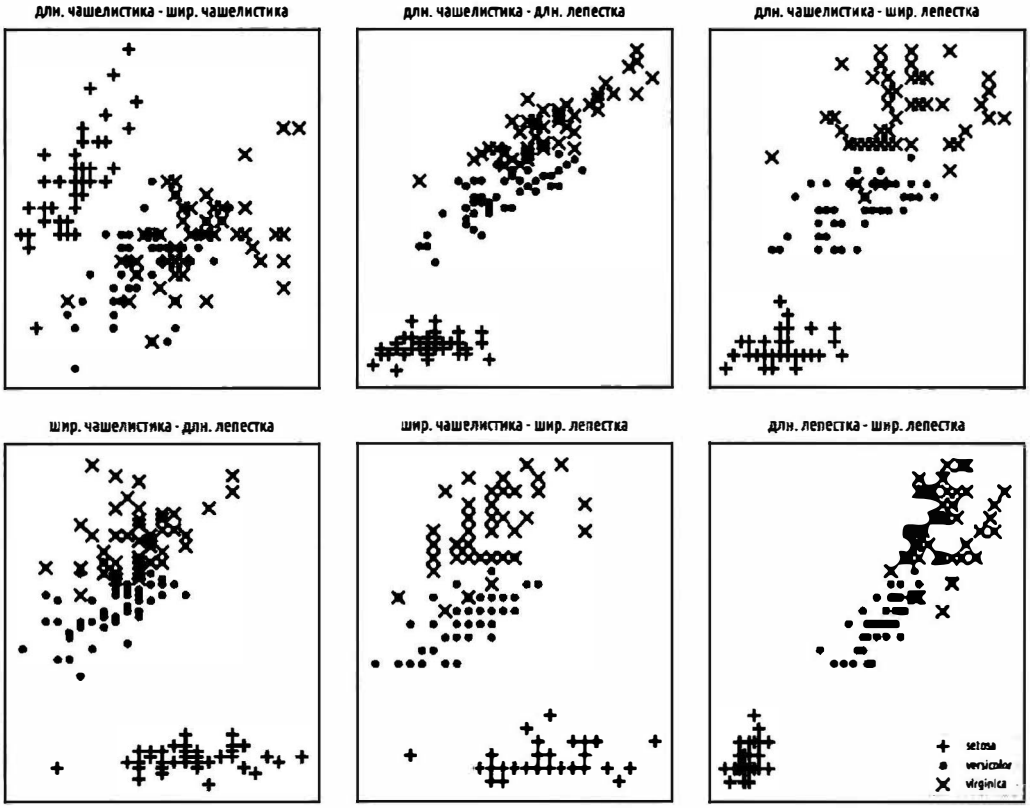


Рис. 12.1. Диаграммы рассеяния для цветков ириса

четырехмерными, что затрудняет их вывод на график. Единственное, что мы можем сделать, — это посмотреть на диаграммы рассеяния для каждой из шести пар данных результатов измерений (рис. 12.1). Я не буду объяснять все детали, но это является хорошей иллюстрацией более сложных вещей, которые вы можете проделывать с библиотекой matplotlib, поэтому стоит изучить ее:

```
from matplotlib import pyplot as plt
metrics = ['длн. чашелистика', 'шир. чашелистика', 'длн. лепестка', 'шир. лепестка']
pairs = [(i, j) for i in range(4) for j in range(4) if i < j]
marks = ['+', '.', 'x']   # У нас 3 класса, поэтому 3 метки

fig, ax = plt.subplots(2, 3)

for row in range(2):
    for col in range(3):
        i, j = pairs[3 * row + col]
        ax[row][col].set_title(f"{metrics[i]} против {metrics[j]}", fontsize=8)
        ax[row][col].set_xticks([])
        ax[row][col].set_yticks([])
```

```
        for mark, (species, points) in zip(marks, points_by_species.items()):
            xs = [point[i] for point in points]
            ys = [point[j] for point in points]
            ax[row][col].scatter(xs, ys, marker=mark, label=species)

ax[-1][-1].legend(loc='lower right', prop={'size': 6})
plt.show()
```

Если вы посмотрите на эти графики, то заметите, что, по всей видимости, данные результатов измерений действительно собираются в кластеры по видам. Например, глядя только на длину чашелистика и ширину чашелистика, вы, вероятно, не смогли бы отличить ирис разноцветный (versicolor) от ириса вергинского (virginica). Но как только вы добавите длину и ширину лепестка в эту смесь, то окажется, что вы сможете предсказать вид на основе ближайших соседей.

Для начала давайте разделим данные на тестовый и тренировочный наборы:

```
import random
from scratch.machine_learning import split_data

random.seed(12)
iris_train, iris_test = split_data(iris_data, 0.70)

assert len(iris_train) == 0.7 * 150
assert len(iris_test) == 0.3 * 150
```

Тренировочный набор будет "соседями", которых мы будем использовать для классифицирования точек в тестовом наборе. Нам просто нужно выбрать значение числа k, т. е. число соседей. Будь оно слишком малым (скажем, $k = 1$), и мы позволим выбросам иметь слишком большое влияние; будь оно слишком большим (скажем, $k = 105$), и мы просто будем предсказывать наиболее распространенный класс в наборе данных.

В реальном приложении (и с большим количеством данных) мы можем создать отдельный контрольный набор и использовать его для выбора k. Здесь мы просто используем $k = 5$:

```
from typing import Tuple

# Отследить число раз, когда мы будем видеть (предсказано, фактически)
confusion_matrix: Dict[Tuple[str, str], int] = defaultdict(int)
num_correct = 0

for iris in iris_test:
    predicted = knn_classify(5, iris_train, iris.point)
    actual = iris.label

    if predicted == actual:
        num_correct += 1

    confusion_matrix[(predicted, actual)] += 1
```

```
pct_correct = num_correct / len(iris_test)
print(pct_correct, confusion_matrix)
```

На этом простом наборе данных модель предсказывает почти идеально. Есть один ирис разноцветный, для которого она предсказывает виргинский, но в остальном модель все делает правильно.

Проклятие размерности

Алгоритм k ближайших соседей сталкивается с неприятностями в более высоких размерностях из-за "проклятия размерности", которое сводится к тому, что высокоразмерные пространства являются обширными. Точки в таких пространствах, как правило, не располагаются близко друг к другу. Для того чтобы это увидеть, надо сгенерировать пары точек в d-размерном "единичном кубе" в разных размерностях и вычислить расстояния между ними.

Генерирование случайных точек должно уже войти в привычку:

```
def random_point(dim: int) -> Vector:
    return [random.random() for _ in range(dim)]
```

как и написание функции, которая генерирует расстояния:

```
def random_distances(dim: int, num_pairs: int) -> List[float]:
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

Для каждой размерности от 1 до 100 мы вычислим 10 000 расстояний и воспользуемся ими для вычисления среднего расстояния между точками и минимального расстояния между точками в каждой размерности (рис. 12.2):

```
import tqdm
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in tqdm.tqdm(dimensions, desc="Проклятие размерности"):
    distances = random_distances(dim, 10000)  # 10 000 произвольных пар
    avg_distances.append(mean(distances))     # Отследить средние
    min_distances.append(min(distances))      # Отследить минимальные
```

По мере увеличения числа размерностей среднее расстояние между точками возрастает. Однако бóльшую проблему вызывает соотношение между минимальным и средним расстояниями (рис. 12.3):

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances,
                                               avg_distances)]
```

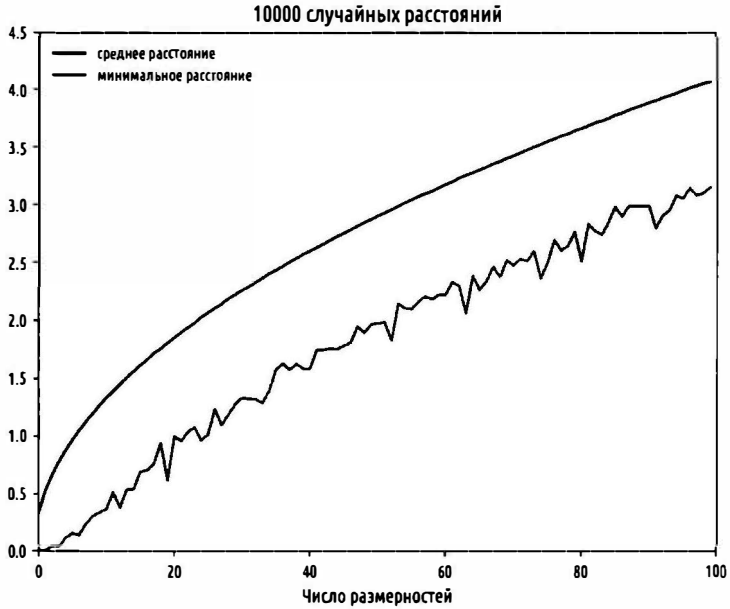


Рис. 12.2. Проклятие размерности

Рис. 12.3. Проклятие размерности (еще раз)

В низкоразмерных наборах данных ближайшие точки тяготеют к тому, что они лежат гораздо ближе среднего. Однако две точки лежат близко, только если они лежат близко в каждой размерности, а каждая дополнительная размерность — даже если это всего лишь шум — это еще одна возможность для того, чтобы любая точка стала еще дальше от любой другой точки. Когда у вас много размерностей, вполне возможно, что ближайшие точки будут лежать не намного ближе, чем среднее, вследствие чего близкая расположенность двух точек теряет особый смысл (если только в данных не присутствует большая структурированность, которая может побудить их вести себя так, как если бы они обладали значительно меньшей размерностью).

Другой способ взглянуть на эту проблему касается разреженности более высокоразмерных пространств.

Если вы выберите 50 случайных чисел между 0 и 1, то, скорее всего, вы получите довольно хорошую выборку единичного интервала (рис. 12.4).

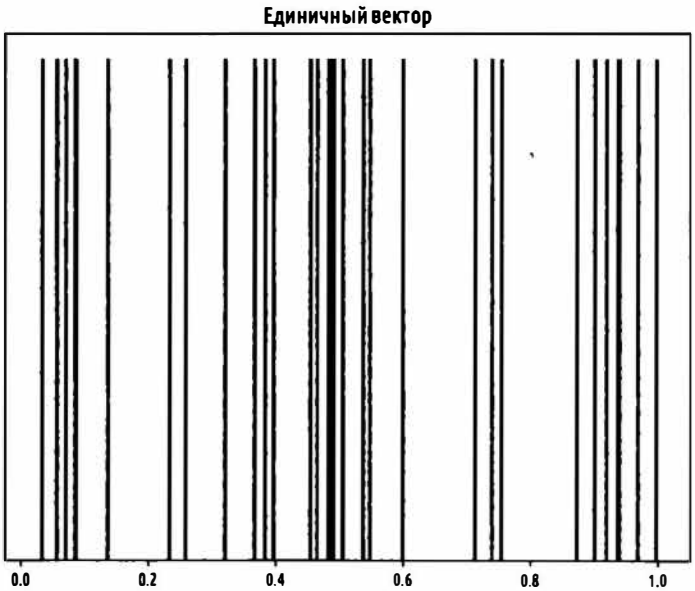


Рис. 12.4. Пятьдесят случайных точек в одной размерности

Если вы выберите 50 случайных точек в единичном квадрате, то получите меньше покрытия (рис. 12.5).

И еще меньше — в трех размерностях (рис. 12.6).

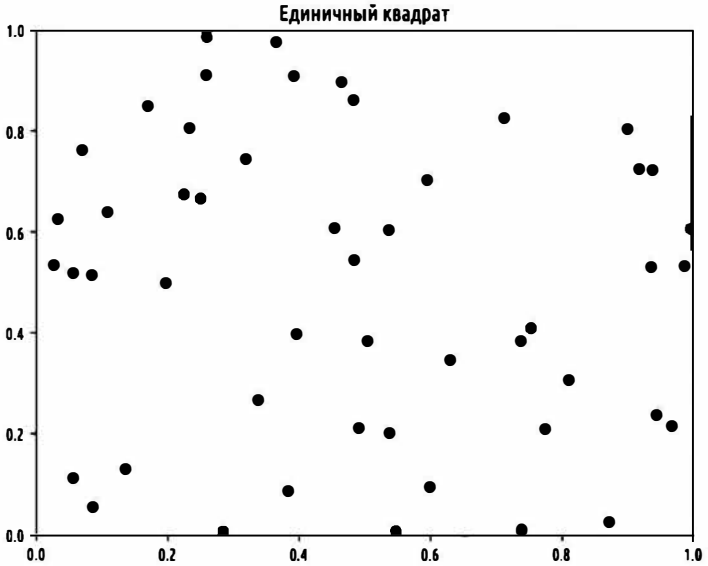


Рис. 12.5. Пятьдесят случайных точек в двух размерностях

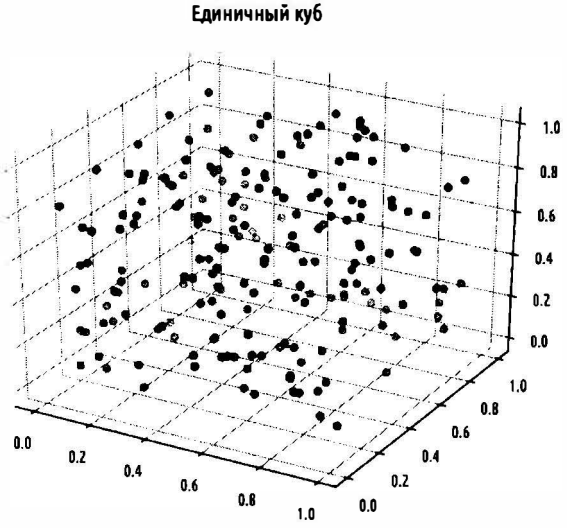


Рис. 12.6. Пятьдесят случайных точек в трех размерностях

Для дальнейшего изучения

Библиотека scikit-learn располагает многочисленными моделями на основе ближайших соседей[2].

[2] См. **https://scikit-learn.org/stable/modules/neighbors.html**.

ГЛАВА 13
Наивный Байес

> Хорошо наивным быть для сердца, но вредно для ума.
> – *Анатоль Франс*[1]

От социальной сети мало толка, если люди в ней не могут общаться. Поэтому DataSciencester предлагает популярную среди пользователей соцсети возможность, которая позволяет им отправлять сообщения другим пользователям. И хотя большинство из них являются ответственными гражданами, которые отправляют только хорошо принимаемые сообщения, типа "как дела?", несколько злоумышленников настойчиво спамят других членов, рассылая незапрошенные адресатами сообщения по поводу схем быстрого обогащения, безрецептурной фармацевтической продукции и платных программ аккредитации в области науки о данных. Пользователи начали жаловаться, и поэтому директор по связям попросил вас применить науку о данных для того, чтобы найти способ фильтрации спамных сообщений.

Реально глупый спам-фильтр

Представим "универсум", состоящий из множества всех возможных сообщений. Обозначим через S событие "сообщение является спамным" и через B — событие "сообщение содержит слово *биткойн*". Теорема Байеса сообщает нам, что вероятность спамного сообщения со словом *биткойн* равна

$$P(S \mid V) = \frac{P(V \mid S)P(S)}{P(V \mid S)P(S) + P(V \mid \neg S)P(\neg S)}.$$

Числитель обозначает вероятность, что сообщение является спамным и содержит слово *биткойн*, а знаменатель — это просто вероятность, что сообщение содержит слово *биткойн*. Следовательно, вы можете думать об этом вычислении, как просто представляющем долю биткойновых сообщений, которые являются спамными.

Если у нас есть крупная коллекция сообщений, о которых мы знаем, что они являются спамными, и крупная коллекция сообщений, о которых мы знаем, что они не являются спамными, то мы можем легко оценить $P(V \mid S)$ и $P(V \mid \neg S)$. Если мы

[1] Анатоль Франс (1844–1924) — французский писатель и литературный критик. Лауреат Нобелевской премии по литературе (1921), деньги которой он пожертвовал в пользу голодающих России — *Прим. пер.*

далее допустим, что любое сообщение равновероятно является спамным или неспамным (так, что $P(S) = P(\neg S) = 0.5$), тогда:

$$P(S \mid V) = \frac{P(V \mid S)}{P(V \mid S) + P(V \mid \neg S)}.$$

Например, если в 50% спамных сообщений имеется слово *биткойн*, но это слово есть только в 1% неспамных сообщений, то вероятность, что любое конкретное сообщение со словом *биткойн* является спамным, равна

$$0.5/(0.5 + 0.01) = 98\%.$$

Более изощренный спам-фильтр

Теперь представим, что у нас есть лексикон с многочисленными словами $w_1, ..., w_n$. Для того чтобы перенести его в область теории вероятностей, обозначим через X_i событие "сообщение содержит слово w_i". Также представим, что (благодаря некому неопределенному на данный момент процессу) мы получили оценку $P(X_i \mid S)$ для вероятности, что спамное сообщение содержит *i*-е слово, и схожую оценку $P(X_i \mid \neg S)$ для вероятности, что неспамное сообщение содержит *i*-е слово.

В основе *наивного байесова классификатора* лежит (серьезное) допущение о том, что присутствия (либо отсутствия) каждого слова не зависят от другого слова при условии, что сообщение является спамным или неспамным. Интуитивно это допущение означает, что знание о том, что некое спамное сообщение содержит слово "биткойн", не дает никакой информации о том, содержит ли то же самое сообщение слово "ролекс". В математических терминах это означает, что

$$P(X_1 = x_1, ..., X_n = x_n \mid S) = P(X_1 = x_1 \mid S) \cdots P(X_n = x_n \mid S).$$

Такое допущение является экстремальным. (Оно объясняет, почему это техническое решение называется наивным.) Представим, что наш лексикон состоит *только* из слов "биткойн" и "ролекс" и что половина всех спамных сообщений содержит выражение "заработать биткойн", а другая половина — "подлинные часы Rolex". В этом случае наивная байесова оценка, что спамное сообщение содержит оба слова: и "биткойн", и "ролекс", равна

$$P(X_1 = 1, X_2 = 1 \mid S) = P(X_1 = 1 \mid S) \cdot P(X_2 = 1 \mid S) = 0.5 \cdot 0.5 = 0.25,$$

т. к. мы изначально не принимали во внимание знание о том, что слова "биткойн" и "ролекс" фактически никогда не встречаются вместе. Несмотря на нереалистичность такого допущения, эта модель часто дает хорошие результаты и исторически используется в реальных спам-фильтрах.

То же самое рассуждение по теореме Байеса, которое мы использовали в спам-фильтре только для *одного* слова *биткойн*, говорит нам, что мы можем рассчитать вероятность того, что сообщение является спамным, используя уравнение:

$$P(S \mid X = x) = \frac{P(X = x \mid S)}{P(X = x \mid S) + P(X = x \mid \neg S)}.$$

Наивное байесово допущение позволяет вычислить каждую из вероятностей справа, просто перемножив между собой индивидуальные вероятностные оценки для каждого слова лексикона.

На практике обычно обходятся без перемножения большого числа вероятностей между собой во избежание проблемы арифметического *переполнения снизу* (приводящего к исчезновению разрядов), с которой компьютеры не справляются при работе с числами с плавающей точкой, находящимися слишком близко к нулю. Вспоминая свойства логарифмов из алгебры: $\log ab = \log a + \log b$ и $\exp(\log x) = x$, мы обычно вычисляем произведение $p_1 \cdots p_n$ как эквивалентную (и более дружелюбную к числам с плавающей точкой) сумму логарифмов:

$$\exp(\log(p_1) + \ldots + \log(p_n)).$$

Единственная сложность, которую осталось преодолеть, — это получить оценки для $P(X_i \mid S)$ и $P(X_i \mid \neg S)$, т. е. вероятности, что спамное (или неспамное) сообщение содержит слово w_i. При наличии значительного числа "тренировочных" сообщений, помеченных как спамное и неспамное, очевидная первая попытка заключается в оценке $P(X_i \mid S)$ просто как доли спамных сообщений со словом w_i.

Однако это приводит к большой проблеме. Предположим, что в тренировочном наборе слово "данные" появляется только в неспамных сообщениях. В результате наш наивный байесов классификатор будет всегда назначать нулевую вероятность спама любому сообщению со словом "данные", даже сообщению наподобие "данные по бесплатному биткойну и подлинным часам Rolex". Во избежание указанной проблемы обычно используют какое-нибудь сглаживание.

В частности, мы воспользуемся псевдосчетчиком k и оценим вероятность встретить i-е слово в спамном сообщении как:

$$P(X_i \mid S) = \frac{k + \text{число спамных сообщений с } w_i}{2k + \text{число спамных сообщений}}.$$

Мы поступим схожим образом для $P(X_i \mid \neg S)$. То есть во время вычисления вероятностей спама для i-го слова мы исходим из того, что мы встретили k дополнительных неспамных сообщений с этим словом и k дополнительных неспамных сообщений без этого слова.

Например, если слово "данные" встречается в 0/98 спамных сообщениях и если k равно 1, то мы оцениваем $P(\text{"данные"} \mid S)$ как $1/100 = 0.01$, что позволяет нашему классификатору все равно назначать ненулевые вероятности спама сообщениям со словом "данные".

Имплементация

Теперь у нас есть все составляющие, необходимые для построения классификатора. Прежде всего создадим простую функцию, которая лексемизирует сообщения на отдельные слова без повторов. Сначала мы конвертируем каждое сообщение в верхний регистр, затем применим функцию `re.findall` для извлечения "слов", состоящих из букв, цифр и апострофов, и, наконец, мы применим объект `Set` для получения уникальных слов:

```
from typing import Set
import re

def tokenize(text: str) -> Set[str]:
    text = text.lower()                 # Конвертировать в нижний регистр,
    all_words = re.findall("[a-z0-9']+", text)  # извлечь слова и
    return set(all_words)                       # удалить повторы.

assert tokenize("Data Science is science") == {"data", "science", "is"}
```

Мы также определяем тип для наших тренировочных данных:

```
from typing import NamedTuple

class Message(NamedTuple):
    text: str
    is_spam: bool
```

Поскольку наш классификатор должен отслеживать лексемы, количества и метки из тренировочных данных, то мы сделаем его классом. Следуя принятым в английском языке традициям, мы обозначим неспамные сообщения в коде как `ham_messages`.

Конструктор будет принимать только один параметр — псевдосчетчик для использования при вычислении вероятностей. Он также инициализирует пустое множество лексем — счетчики для отслеживания того, как часто каждая лексема встречается в спамных и неспамных сообщениях, — и подсчитывает число спамных и неспамных сообщений, на которых он был натренирован:

```
from typing import List, Tuple, Dict, Iterable
import math
from collections import defaultdict

class NaiveBayesClassifier:
    def __init__(self, k: float = 0.5) -> None:
        self.k = k  # Сглаживающий фактор

        self.tokens: Set[str] = set()
        self.token_spam_counts: Dict[str, int] = defaultdict(int)
        self.token_ham_counts: Dict[str, int] = defaultdict(int)
        self.spam_messages = self.ham_messages = 0
```

Далее мы дадим ему метод, который будет его тренировать на группе сообщений. Сначала мы увеличиваем количества спамных сообщений `spam_messages` и неспамных сообщений `ham_messages`. Затем мы лексемизируем текст каждого сообщения и для каждой лексемы увеличиваем спамные и неспамные количества `token_spam_counts` или `token_ham_counts`, основываясь на типе сообщения:

```
def train(self, messages: Iterable[Message]) -> None:
    for message in messages:
        # Увеличить количества сообщений
        if message.is_spam:
            self.spam_messages += 1
        else:
            self.ham_messages += 1

        # Увеличить количества появлений слов
        for token in tokenize(message.text):
            self.tokens.add(token)
            if message.is_spam:
                self.token_spam_counts[token] += 1
            else:
                self.token_ham_counts[token] += 1
```

В конечном итоге мы хотим предсказать $P(\text{спам} \mid \text{лексема})$. Как мы видели ранее, для того чтобы применить теорему Байеса, нам нужно знать $P(\text{лексема} \mid \text{спам})$ и $P(\text{лексема} \mid \text{неспам})$ для каждой лексемы в лексиконе. Поэтому для их вычисления мы создадим приватную вспомогательную функцию:

```
def _probabilities(self, token: str) -> Tuple[float, float]:
    """Возвращает P(лексема | спам) и P(лексема | неспам)"""
    spam = self.token_spam_counts[token]
    ham = self.token_ham_counts[token]

    p_token_spam = (spam + self.k) / (self.spam_messages + 2 * self.k)
    p_token_ham = (ham + self.k) / (self.ham_messages + 2 * self.k)

    return p_token_spam, p_token_ham
```

Наконец, мы готовы написать наш метод предсказания `predict`. Как упоминалось ранее, вместо перемножения многочисленных малых вероятностей мы просуммируем логарифмические вероятности:

```
def predict(self, text: str) -> float:
    text_tokens = tokenize(text)
    log_prob_if_spam = log_prob_if_ham = 0.0

    # Перебрать все слова в лексиконе
    for token in self.tokens:
        prob_if_spam, prob_if_ham = self._probabilities(token)
```

```
            # Если *лексема* появляется в сообщении,
            # то добавить логарифмическую вероятность ее встретить
            if token in text_tokens:
                log_prob_if_spam += math.log(prob_if_spam)
                log_prob_if_ham += math.log(prob_if_ham)

            # В противном случае добавить логарифмическую вероятность
            # ее НЕ встретить, т. е. log(1 - вероятность ее встретить)
            else:
                log_prob_if_spam += math.log(1.0 - prob_if_spam)
                log_prob_if_ham += math.log(1.0 - prob_if_ham)

        prob_if_spam = math.exp(log_prob_if_spam)
        prob_if_ham = math.exp(log_prob_if_ham)
        return prob_if_spam / (prob_if_spam + prob_if_ham)
```

И теперь у нас есть классификатор.

Тестирование модели

Давайте убедимся, что наша модель работает, написав для нее несколько модульных тестов.

```
messages = [Message("spam rules", is_spam=True),
            Message("ham rules", is_spam=False),
            Message("hello ham", is_spam=False)]

model = NaiveBayesClassifier(k=0.5)
model.train(messages)
```

Сперва проверим, что правильно рассчитаны количества:

```
assert model.tokens == {"spam", "ham", "rules", "hello"}
assert model.spam_messages == 1
assert model.ham_messages == 2
assert model.token_spam_counts == {"spam": 1, "rules": 1}
assert model.token_ham_counts == {"ham": 2, "rules": 1, "hello": 1}
```

Теперь сделаем предсказание. Мы так же (кропотливо) пройдемся по нашей логике наивного Байеса вручную и убедимся, что получим тот же результат:

```
text = "hello spam"

probs_if_spam = [
    (1 + 0.5) / (1 + 2 * 0.5),      # "spam" (присутствует)
    1 - (0 + 0.5) / (1 + 2 * 0.5),  # "ham" (не присутствует)
    1 - (1 + 0.5) / (1 + 2 * 0.5),  # "rules" (не присутствует)
    (0 + 0.5) / (1 + 2 * 0.5)       # "hello" (присутствует)
]
```

```
probs_if_ham = [
    (0 + 0.5) / (2 + 2 * 0.5),      # "spam" (присутствует)
    1 - (2 + 0.5) / (2 + 2 * 0.5),  # "ham" (не присутствует)
    1 - (1 + 0.5) / (2 + 2 * 0.5),  # "rules" (не присутствует)
    (1 + 0.5) / (2 + 2 * 0.5),      # "hello" (присутствует)
]

p_if_spam = math.exp(sum(math.log(p) for p in probs_if_spam))
p_if_ham = math.exp(sum(math.log(p) for p in probs_if_ham))

# Должно быть примерно 0.83
assert model.predict(text) == p_if_spam / (p_if_spam + p_if_ham)
```

Этот тест проходит, поэтому, похоже, наша модель делает то, что мы от нее ожидаем. Если посмотреть на фактические вероятности, то двумя большими движущими силами является то, что наше сообщение содержит спам (что и делало наше одинокое тренировочное спамное сообщение) и что оно не содержит неспама (что и делали оба наших тренировочных сообщения).

Теперь давайте опробуем модель на реальных данных.

Применение модели

Популярным (хотя и немного устаревшим) набором данных является публичный текстовый корпус под названием SpamAssassin (**https://spamassassin.apache.org/publiccorpus/**). Мы возьмем файлы с префиксом 20021010.

Вот сценарий, который скачает и распакует их в каталог по вашему выбору (либо вы можете сделать это вручную):

```
from io import BytesIO  # Необходимо трактовать байты как файл.
import requests         # Для скачивания файлов, которые
import tarfile          # находятся в формате .tar.bz

BASE_URL = "https://spamassassin.apache.org/old/publiccorpus"
FILES = ["20021010_easy_ham.tar.bz2",
         "20021010_hard_ham.tar.bz2",
         "20021010_spam.tar.bz2"]

# В этих папках данные окажутся после распаковки:
# /spam, /easy_ham и /hard_ham.
# Можете поменять каталог по своему усмотрению
OUTPUT_DIR = 'spam_data'

for filename in FILES:
    # Используем requests для получения
    # содержимого файлов в каждом URL
    content = requests.get(f"{BASE_URL}/{filename}").content
```

```
    # Обернуть байты в памяти, чтобы использовать их как "файл"
    fin = BytesIO(content)

    # И извлечь все файлы в указанный выходной каталог.
    with tarfile.open(fileobj=fin, mode='r:bz2') as tf:
        tf.extractall(OUTPUT_DIR)
```

Возможно, расположение файлов изменится (это произошло между первым и вторым изданиями этой книги), и в этом случае настройте сценарий соответствующим образом.

После скачивания данных у вас должно быть три папки: spam, easy_ham и hard_ham. Каждая папка содержит много писем, каждое из которых находится в одном файле. Для того чтобы все было действительно просто, мы будем просматривать тематические строки каждого письма.

Как вычленить строку с темой? Когда мы просматриваем файлы, то видим, что все они, похоже, начинаются со слова "Subject:". Поэтому мы будем искать именно эту цепочку символов:

```
import glob, re

# Замените этот путь на любой каталог, в который вы поместили файлы
path = 'spam_data/*/*'

data: List[Message] = []
# glob.glob возвращает каждое имя файла,
# которое соответствует поисковому шаблону пути
for filename in glob.glob(path):
    is_spam = "ham" not in filename

    # В письмах имеются несколько мусорных символов;
    # параметр errors='ignore' пропускает их вместо
    # вызова исключения
    with open(filename, errors='ignore') as email_file:
        for line in email_file:
            if line.startswith("Subject:"):
                subject = line.lstrip("Subject: ")
                data.append(Message(subject, is_spam))
                break   # С этим файлом работа закончена
```

Теперь мы можем разбить данные на тренировочные и тестовые, после чего всё готово для построения классификатора:

```
import random
from scratch.machine_learning import split_data

random.seed(0)   # Требуется для получения тех же ответов, что и у меня
train_messages, test_messages = split_data(data, 0.75)
```

```
model = NaiveBayesClassifier()
model.train(train_messages)
```

Давайте сгенерируем несколько предсказаний и проверим, как наша модель работает:

```
from collections import Counter

predictions = [(message, model.predict(message.text))
               for message in test_messages]

# Будем считать, что спамная вероятность spam_probability > 0.5
# соответствует предсказанию спама, и подсчитаем комбинации
# (фактический спам is_spam, предсказанный спам is_spam)
confusion_matrix = Counter((message.is_spam, spam_probability > 0.5)
                           for message, spam_probability in predictions)

print(confusion_matrix)
```

Это даст 84 истинных утверждения (спам классифицируется как "спам"), 25 ложных утверждений (неспам классифицируется как "спам"), 703 истинных отрицаний (неспам классифицируется как "неспам") и 44 ложных отрицания (спам классифицируется как "неспам"). Это означает, что прецизионность равна $84/(84+25) = 77\%$, а полнота равна $84/(84+44) = 65\%$, что совсем неплохо для такой простой модели. (По-видимому, было бы еще лучше, если бы мы просматривали не только темы.)

Мы также можем обследовать внутренности модели, чтобы увидеть, какие слова больше всего и меньше всего указывают на спам:

```
def p_spam_given_token(token: str, model: NaiveBayesClassifier) -> float:
    # Нам, вероятно, не следует вызывать приватные методы,
    # но это делается ради благой цели.
    prob_if_spam, prob_if_ham = model._probabilities(token)

    return prob_if_spam / (prob_if_spam + prob_if_ham)

words = sorted(model.tokens, key=lambda t: p_spam_given_token(t, model))

print("наиболее_спамные_слова", words[-10:])
print("наименее_спамные_слова", words[:10])
```

Наиболее спамные слова включают такие вещи, как sale (распродажа), mortgage (закладная), money (деньги) и rates (ставки, цены), тогда как наименее спамные слова включают такие вещи, как spambayes (байесов спам-фильтр), users (пользователи), apt (квартира) и perl (жемчуг). Это также дает нам некоторую интуитивную уверенность в том, что наша модель в основном поступает правильно.

Каким образом можно получить более высокую результативность? Один из очевидных способов — предоставить больше тренировочных данных. Кроме этого,

есть ряд других способов улучшения модели. Вот несколько возможностей, которые стоит попробовать.

- Просматривать не только темы, но и содержимое сообщений. Следует внимательно отнестись к обработке заголовков сообщений.
- Наш классификатор учитывает все слова, которые появляются в тренировочном наборе, даже те, которые появляются всего один раз. Следует модифицировать классификатор так, чтобы он принимал необязательный порог `min_count` и игнорировал слова, число появлений которых меньше порога.
- Лексемизатор не имеет понятия о похожих словах (к примеру, cheap и cheapest — дешевый, дешевейший). Можно модифицировать классификатор так, чтобы он включал необязательную функцию-стеммер для выделения основ слов, которая конвертирует слова в *классы эквивалентности*. Например, в качестве очень простого стеммера может выступать следующая ниже функция:

```
# Удалить окончание s
def drop_final_s(word):
    return re.sub("s$", "", word)
```

  Создание хорошей функции выделения основ слов — задача сложная. Для отыскания основы слова часто пользуются алгоритмом Портера[2].
- Хотя все наши признаки имеют форму "сообщение содержит слово w_i", нет никаких причин, почему это должно быть именно так. В нашей имплементации мы могли бы добавить дополнительные признаки, такие как "сообщение содержит число", путем создания фиктивных слов, таких как *содержит:число*, и модифицирования лексемизатора `tokenizer` так, чтобы он эмитировал их в случае необходимости.

Для дальнейшего изучения

- Статьи Пола Грэма "План для спама"[3] и "Улучшенная байесова фильтрация"[4] могут (быть интересными и) углубить понимание идеи, лежащей в основе построения спам-фильтров.
- Библиотека scikit-learn[5] содержит бернуллиеву модель `BernoulliNB`, имплементирующую тот же самый алгоритм наивной байесовой классификации, который имплементирован здесь, а также некоторые другие версии указанной модели.

---

[2] См. **http://tartarus.org/martin/PorterStemmer/, https://ru.wikipedia.org/wiki/Стеммер_Портера**.

[3] См. **http://www.paulgraham.com/spam.html**.

[4] См. **http://www.paulgraham.com/better.html**.

[5] См. **https://scikit-learn.org/stable/modules/naive_bayes.html**.

Простая линейная регрессия

> В морали, как в живописи, главное состоит в том, чтобы в нужном месте провести линию.
> – *Г. К. Честертон*[1]

В *главе 5* мы использовали функцию корреляции `correlation` для измерения силы линейной связи между двумя переменными. В большинстве приложений недостаточно знать, что такая линейная связь существует. Необходимо иметь возможность установить природу этой связи. Именно в этом случае мы будем использовать простую линейную регрессию.

Модель

Вспомним, что мы исследовали связь между числом друзей пользователя социальной сети DataSciencester и количеством времени, которое он проводит на ее веб-сайте каждый день. Будем считать, что мы убедили себя в том, что чем больше друзей, тем больше времени люди проводят на веб-сайте, в отличие от альтернативных объяснений, которые уже обсуждались.

Директор по взаимодействию просит вас построить модель, описывающую эту связь. Поскольку вы нашли довольно сильную линейную связь, то логично начать с линейной модели.

В частности, можно выдвинуть гипотезу, что существуют константы α (альфа) и β (бета), такие, что:

$$y_i = \beta x_i + \alpha + \varepsilon_i,$$

где y_i — это число минут, которое пользователь i ежедневно проводит на веб-сайте; x_i — это число друзей пользователя i; ε_i — это случайная (будем надеяться, небольшая) ошибка; данный член представляет тот факт, что существуют другие факторы, не учтенные этой простой моделью.

Допуская, что мы определили такие константы `alpha` и `beta`, сделать предсказание очень просто:

```
def predict(alpha: float, beta: float, x_i: float) -> float:
    return beta * x_i + alpha
```

[1] Гилберт Кит Честертон (1874–1936) — английский христианский мыслитель, журналист и писатель — *Прим. пер.*

Как выбрать alpha и beta? Начнем с того, что любой выбор alpha и beta дает нам предсказанный результат для каждого входного x_i. Поскольку мы знаем фактический выход y_i , мы можем вычислить ошибку для каждой пары:

```
def error(alpha: float, beta: float, x_i: float, y_i: float) -> float:
    """Ошибка предсказания beta * x_i + alpha,
       когда фактическое значение равно y_i"""
    return predict(alpha, beta, x_i) - y_i
```

На самом же деле мы хотели бы знать суммарную ошибку по всему набору данных. Но мы не хотим просто просуммировать ошибки: если предсказание для x_1 является слишком высоким, а для x_2 — слишком низким, то в результате ошибки могут нейтрализовать друг друга.

Вместо этого суммируются *квадраты* ошибки, т. е. квадратические:

```
from scratch.linear_algebra import Vector

def sum_of_sqerrors(alpha: float, beta: float, x: Vector, y: Vector) -> float:
    return sum(error(alpha, beta, x_i, y_i) ** 2
               for x_i, y_i in zip(x, y))
```

Решение наименьшими квадратами состоит в выборе alpha и beta такими, которые делают сумму квадратов ошибок sum_of_squared_errors как можно меньше.

Используя исчисление (или скучную алгебру), alpha и beta, которые минимизируют ошибку, задаются следующим образом:

```
from typing import Tuple
from scratch.linear_algebra import Vector
from scratch.statistics import correlation, standard_deviation, mean

def least_squares_fit(x: Vector, y: Vector) -> Tuple[float, float]:
    """Учитывая векторы x и y, отыскать
       значения alpha и beta по наименьшим квадратам"""
    beta = correlation(x, y) * standard_deviation(y) / standard_deviation(x)
    alpha = mean(y) - beta * mean(x)
    return alpha, beta
```

Не углубляясь в точную математику, подумаем, почему это решение может быть разумным. Выбор alpha просто говорит, что, когда мы видим среднее значение независимой переменной x, мы предсказываем среднее значение зависимой переменной y.

Выбор коэффициента beta означает, что, когда входное значение увеличивается на стандартное отклонение standard_deviation(x), предсказание увеличивается на correlation(x, y) * standard_deviation(y). В случае, когда x и y идеально коррелированны, увеличение на одно стандартное отклонение в x приводит к росту на одно стандартное отклонение в y в предсказании. Когда они идеально антикоррелированы, увеличение в x ведет к *снижению* в предсказании. И когда корреляция равна 0, то beta равен 0, а это означает, что изменения в x совершенно не влияют на предсказание.

Как обычно, давайте напишем для модели быстрый тест:

```
x = [i for i in range(-100, 110, 10)]
y = [3 * i - 5 for i in x]

# Должна отыскать, что y = 3x - 5
assert least_squares_fit(x, y) == (-5, 3)
```

Теперь легко применить ее к данным без выбросов из *главы 5*:

```
from scratch.statistics import num_friends_good, daily_minutes_good

alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)

assert 22.9 < alpha < 23.0
assert 0.9 < beta < 0.905
```

В результате получим значения `alpha = 22.95` и `beta = 0.903`. Таким образом, наша модель говорит, что пользователь с `n` друзьями из соцсети DataSciencester, согласно модели, будет проводить на веб-сайте `22.95 + n * 0.903` минут ежедневно. Другими словами, модель предсказывает, что пользователь без друзей все равно будет проводить на веб-сайте около 23 минут в день, а каждый дополнительный друг, как ожидается, будет увеличивать проводимое пользователем время почти на одну минуту в день.

На рис. 14.1 показана предсказательная линия, которая дает представление о подгонке модели, т. е. о том, насколько хорошо модель вписывается в данные.

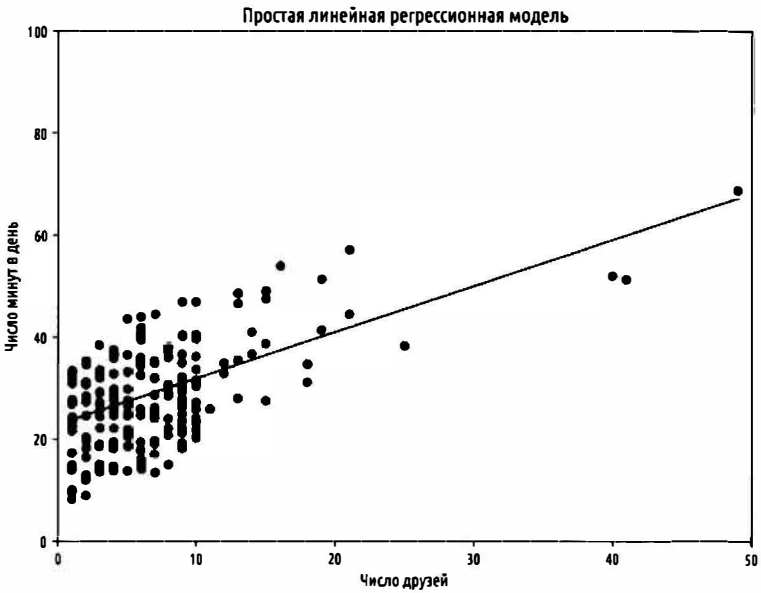


Рис. 14.1. Простая линейная модель

Разумеется, для того чтобы выяснить качество подгонки к данным, требуется более подходящий способ, чем просто рассматривать диаграмму. Распространенной мерой оценки качества подгонки является *коэффициент детерминации* (или *R-квадрат*, R^2), который измеряет долю суммарной вариации в зависимой переменной, улавливаемой моделью:

```
from scratch.statistics import de_mean

def total_sum_of_squares(y: Vector) -> float:
    """Полная сумма квдаратов отклонений y_i от их среднего"""
    return sum(v ** 2 for v in de_mean(y))

def r_squared(alpha: float, beta: float, x: Vector, y: Vector) -> float:
    """Доля отклонения в y, улавливаемая моделью, которая равна
       '1 - доля отклонения в y, не улавливаемая моделью'"""
    return 1.0 - (sum_of_sqerrors(alpha, beta, x, y) /
                  total_sum_of_squares(y))

rsq = r_squared(alpha, beta, num_friends_good, daily_minutes_good)
assert 0.328 < rsq < 0.330
```

Вспомним, что мы выбрали `alpha` и `beta`, которые минимизировали сумму квадратов ошибок предсказания. Линейная модель, которую мы могли бы выбрать, равна "всегда предсказывать среднее `mean(y)`" (что соответствует `alpha = mean(y)` и `beta = 0`), чья сумма квадратов ошибок в точности равна полной сумме квадратов. Это означает, что *R*-квадрат (коэффициент детерминации) равен нулю, указывая на модель, которая (в данном случае со всей очевидностью) работает не лучше, чем простое предсказание среднего значения.

Ясно, что модель на основе наименьших квадратов должна как минимум предсказывать не хуже, и поэтому сумма квадратов ошибок должна быть *не больше* полной суммы квадратов, а это значит, что *R*-квадрат должен быть не меньше 0. И сумма квадратов ошибок должна быть не меньше 0, следовательно, *R*-квадрат может быть не больше 1.

Чем выше число, тем лучше подгонка модели к данным. Здесь мы вычислили *R*-квадрат равным 0.329, что говорит о том, что наша модель вписывается в данные только частично и, очевидно, присутствуют дополнительные факторы.

Применение градиентного спуска

Если мы запишем `theta = [alpha, beta]`, то эту задачу можно решить с использованием градиентного спуска:

```
import random
import tqdm
from scratch.gradient_descent import gradient_step

num_epochs = 10000
random.seed(0)
```

```
guess = [random.random(), random.random()]    # Выбрать случайное число
                                              # для запуска.
learning_rate = 0.00001                       # Темп усвоения

with tqdm.trange(num_epochs) as t:
    for _ in t:
        alpha, beta = guess

        # Частная производная потери по отношению к alpha
        grad_a = sum(2 * error(alpha, beta, x_i, y_i)
                     for x_i, y_i in zip(num_friends_good,
                                         daily_minutes_good))

        # Частная производная потери по отношению к beta
        grad_b = sum(2 * error(alpha, beta, x_i, y_i) * x_i
                     for x_i, y_i in zip(num_friends_good,
                                         daily_minutes_good))

    # Вычислить потерю для вставки в описание tqdm
    loss = sum_of_sqerrors(alpha, beta,
                           num_friends_good, daily_minutes_good)
    t.set_description(f"потеря: {loss:.3f}")

    # В заключение обновить догадку
    guess = gradient_step(guess, [grad_a, grad_b], -learning_rate)

# Мы должны получить практически одинаковые результаты:
alpha, beta = guess
assert 22.9 < alpha < 23.0
assert 0.9 < beta < 0.905
```

Если вы выполните этот фрагмент кода, то получите для `alpha` и `beta` те же результаты, которые мы получили, используя точную формулу.

Оценивание максимального правдоподобия

Почему выбраны именно наименьшие квадраты? Одно из объяснений касается оценивания *максимального правдоподобия*[2]. Представим, что у нас есть выборка данных $v_1, ..., v_n$, которые поступают из распределения, зависящего от некоего неизвестного параметра θ (тета):

$$p(v_1, ..., v_n | \theta).$$

[2] Оценивание максимального правдоподобия — это метод оценивания неизвестного параметра путем максимизации функции вероятности, в результате чего приобретаются значения параметров модели, которые делают данные "ближе" к реальным. — *Прим. пер.*

Если параметр θ неизвестен, то можно поменять члены местами, чтобы представить эту величину, как правдоподобие параметра θ при наличии выборки:

$$L(\theta | v_1, ..., v_n).$$

В условиях такого подхода наиболее вероятным значением θ является то, которое максимизирует эту функцию правдоподобия, т. е. значение, которое делает наблюдаемые данные наиболее вероятными. В случае непрерывного распределения, где мы имеем функцию распределения вероятности вместо функции массы вероятности, мы можем сделать то же самое.

Но вернемся к регрессии. Одно из допущений, которое нередко принимается относительно простой регрессионной модели, заключается в том, что регрессионные ошибки нормально распределены с нулевым средним и неким (известным) стандартным отклонением σ. Если это так, то правдоподобие на основе наблюдаемой пары `(x_i, y_i)` равно:

$$L(\alpha, \beta | x_i, y_i, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(y_i - \alpha - \beta x_i)^2}{2\sigma^2} \right).$$

Правдоподобие на основе всего набора данных равно произведению индивидуальных правдоподобий, которое является максимальным именно тогда, когда выбираются такие `alpha` и `beta`, которые минимизируют сумму квадратов ошибок. То есть в этом случае (и с этими допущениями) минимизация суммы квадратов ошибок эквивалентна максимизации правдоподобия наблюдаемых данных.

Для дальнейшего изучения

Продолжение о множественной регрессии см. в *главе 15*.

Множественная регрессия

> Я не берусь за решение задачи, не вкладывая в нее переменные, которые не смогут на нее повлиять.
>
> – *Билл Парцеллс*[1]

Хотя директор порядком впечатлен вашей предсказательной моделью, он все же считает, что ее можно улучшить. С этой целью вы собрали дополнительные данные: по каждому пользователю теперь известно, сколько часов он работает каждый день и есть ли у него ученая степень. Вы собираетесь использовать эти данные для усовершенствования модели.

В соответствии с этим вы строите гипотезу о линейной модели, но теперь уже с бóльшим числом независимых переменных:

$$\text{минуты} = \alpha + \beta_1 \cdot \text{друзья} - \beta_2 \cdot \text{рабочие\_часы} + \beta_3 \cdot \text{степень} + \varepsilon .$$

Очевидно, наличие у пользователя ученой степени выражается не числом, но, как уже упоминалось в *главе 11*, мы можем ввести фиктивную переменную, которая равна 1 для пользователей с ученой степенью и 0 для пользователей без нее, после чего она будет такой же числовой, как и другие переменные.

Модель

Вспомните, что в *главе 14* мы выполняли подбор модели следующей формы:

$$y_i = \alpha + \beta x_i + \varepsilon_i .$$

Теперь представим, что каждый вход x_i — это не одно число, а вектор из k чисел $x_{i1}, \ldots, x_{ik}$. Множественная регрессионная модель исходит из того, что:

$$y_i = \alpha + \beta_1 x_{i1} + \ldots + \beta_k x_{ik} + \varepsilon_i .$$

Во множественной регрессии вектор параметров обычно называется бетой (β). Мы хотим, чтобы он также включал константный член, и мы достигаем этого путем добавления столбца из единиц в наши данные:

```
beta = [alpha, beta_1, ..., beta_k]
```

и:

```
x_i = [1, x_i1, ..., x_ik]
```

[1] Дуэйн Чарльз "Билл" Парцеллс (род. 1941) — бывший главный тренер американской национальной команды по американскому футболу — *Прим. пер.*

Тогда наша модель будет просто такой:

```
from scratch.linear_algebra import dot, Vector

def predict(x: Vector, beta: Vector) -> float:
    """Предполагается, что первый элемент каждого x_i равен 1"""
    return dot(x, beta)
```

В данном конкретном случае независимая переменная `x` будет списком векторов, каждый из которых выглядит следующим образом:

```
[1,     # константа
 49,    # число друзей
 4,     # рабочие часы в день
 0]     # не имеет ученой степени
```

Расширенные допущения модели наименьших квадратов

Имеется несколько расширенных допущений, необходимых для того, чтобы эта модель (и наше решение) имели смысл.

Первое заключается в том, что столбцы x являются *линейно независимыми*, т. е. невозможно записать любой из них как взвешенную сумму каких-то других столбцов. Если это допущение не соблюдается, то невозможно оценить `beta`. Для того чтобы увидеть это в экстремальном случае, представим, что в наших данных есть дополнительное поле для числа знакомых `num_acquaintances`, которое для каждого пользователя было равным числу друзей `num_friends`.

Затем, начиная с любого `beta`, если добавлять любое количество в коэффициент `num_friends` и вычитать это же количество из коэффициента `num_acquaintances`, то предсказания модели останутся без изменений. Это означает, что нет никакого способа отыскать коэффициент для `num_friends`. (Обычно нарушения этого допущения не столь очевидны.)

Второе важное допущение состоит в том, что все столбцы x являются некоррелированными с ошибками ε. Если это допущение не соблюдается, то наши оценки `beta` будут систематически неверными.

Например, в *главе 14* мы построили модель, которая предсказывала, что каждый дополнительный друг ассоциирован с дополнительными 0.90 минутами, проводимыми на веб-сайте ежедневно.

Представим, также, что:

- люди, которые работают больше часов, проводят на веб-сайте меньше времени;
- люди, у которых больше друзей, как правило, работают больше часов.

То есть представим, что "фактическая" модель равна:

$$\text{минуты} = \alpha + \beta_1 \cdot \text{друзья} - \beta_2 \cdot \text{рабочие\_часы} + \varepsilon,$$

где β_2 является отрицательным, и что рабочие часы и друзья положительно коррелируют. В этом случае, когда мы минимизируем ошибки однофакторной модели (с одной переменной):

$$\text{минуты} = \alpha + \beta_1 \cdot \text{друзья} + \varepsilon,$$

мы недооцениваем β_1.

Подумайте о том, что произойдет, если бы мы делали предсказания, используя однофакторную модель с "фактическим" значением β_1 (т. е. значением, которое возникает из минимизации ошибок того, что мы называем "фактической" моделью). Предсказания будут, как правило, чересчур большими для пользователей, которые работают много часов, и слегка большими для пользователей, которые работают всего несколько часов, потому что $\beta_2 < 0$, и мы "забыли" его включить. Поскольку рабочие часы положительно коррелирует с числом друзей, то это означает, что предсказания, как правило, будут чересчур большими для пользователей с многочисленными друзьями и лишь слегка большими для пользователей с малочисленными друзьями.

В результате мы можем снизить ошибки (в однофакторной модели), уменьшив нашу оценку β_1, а это означает, что минимизирующий ошибку β_1 является меньше "фактического" значения. То есть в данном случае однофакторное решение наименьшими квадратами смещено в сторону недооценки β_1. И в общем случае всегда, когда независимые переменные коррелируют с ошибками таким образом, наше решение наименьшими квадратами будет давать смещенную оценку β_1.

Подгонка модели

Как и в простой линейной модели, мы выберем параметр `beta`, минимизирующий сумму квадратов ошибок. Найти точное решение этой задачи вручную довольно трудно, а значит, нам потребуется градиентный спуск. Опять же мы хотели бы минимизировать сумму квадратов ошибок. Функция ошибки почти идентична той, которую мы использовали в *главе 14*, за исключением того, что вместо ожидаемых параметров `[alpha, beta]` она будет принимать вектор произвольной длины:

```
from typing import List

def error(x: Vector, y: float, beta: Vector) -> float:
    return predict(x, beta) - y

def squared_error(x: Vector, y: float, beta: Vector) -> float:
    return error(x, y, beta) ** 2

x = [1, 2, 3]
y = 30
beta = [4, 4, 4]   # поэтому предсказание равно 4 + 8 + 12 = 24

assert error(x, y, beta) == -6
assert squared_error(x, y, beta) == 36
```

Если вы знаете дифференциальное исчисление, то вычислить градиент будет легко:

```
def sqerror_gradient(x: Vector, y: float, beta: Vector) -> Vector:
    err = error(x, y, beta)
    return [2 * err * x_i for x_i in x]

assert sqerror_gradient(x, y, beta) == [-12, -24, -36]
```

В противном случае вам придется поверить мне на слово.

В этом месте мы готовы отыскать оптимальный `beta`, используя градиентный спуск. Давайте сначала напишем функцию подгонки наименьшими квадратами `least_squares_fit`, которая способна работать с любым набором данных:

```
import random
import tqdm
from scratch.linear_algebra import vector_mean
from scratch.gradient_descent import gradient_step

def least_squares_fit(xs: List[Vector],
                      ys: List[float],
                      learning_rate: float = 0.001,
                      num_steps: int = 1000,
                      batch_size: int = 1) -> Vector:
    """Отыскать beta, который минимизирует сумму квадратов ошибок,
       исходя из того, что модель y = dot(x, beta)."""

    # Начать со случайной догадки
    guess = [random.random() for _ in xs[0]]

    for _ in tqdm.trange(num_steps, desc="least squares fit"):
        for start in range(0, len(xs), batch_size):
            batch_xs = xs[start:start+batch_size]
            batch_ys = ys[start:start+batch_size]

            gradient = vector_mean([sqerror_gradient(x, y, guess)
                                    for x, y in zip(batch_xs, batch_ys)])
            guess = gradient_step(guess, gradient, -learning_rate)

    return guess
```

Затем мы можем применить ее к нашим данным:

```
from scratch.statistics import daily_minutes_good
from scratch.gradient_descent import gradient_step

random.seed(0)

# Параметры num_iters и step_size были выбраны
# мной путем проб и ошибок.
```

```
# Это заставит поработать некоторое время
learning_rate = 0.001

beta = least_squares_fit(inputs, daily_minutes_good,
                         learning_rate, 5000, 25)

assert 30.50 < beta[0] < 30.70  # константа
assert 0.96 < beta[1] < 1.00    # число друзей
assert -1.89 < beta[2] < -1.85  # число рабочих часов в день
assert 0.91 < beta[3] < 0.94    # имеет ученую степень
```

На практике вы не будете оценивать линейную регрессию с помощью градиентного спуска; вы бы получили точные коэффициенты, используя линейно-алгебраические технические приемы, которые выходят за рамки этой книги. Если бы вы это сделали, то пришли бы к уравнению:

$$\text{минуты} = 30.58 + 0.972 \cdot \text{друзей} - 1.87 \cdot \text{рабочие\_часы} + 0.923 \cdot \text{ученая\_степень},$$

что достаточно близко к тому, что мы обнаружили.

Интерпретация модели

Вы должны думать о коэффициентах модели как о величинах, представляющих оценки влияния каждого конкретного фактора при прочих равных. При прочих равных каждый дополнительный друг соответствует лишней минуте, проводимой на веб-сайте ежедневно. При прочих равных каждый дополнительный час в рабочем дне пользователя соответствует сокращению времени, проводимого на веб-сайте ежедневно, примерно на 2 минуты. При прочих равных наличие ученой степени связано с проведением на веб-сайте ежедневно одной лишней минуты.

Однако модель (непосредственно) нам ничего не говорит о взаимодействиях между переменными. Вполне возможно, что влияние фактора рабочих часов иное для людей с большим числом друзей, нежели его влияние для людей с малым числом друзей. Данная модель не улавливает это. Один из способов учесть такой случай состоит в введении новой переменной, которая является *произведением* "друзей" и "рабочих часов". Это фактически позволяет коэффициенту "рабочих часов" увеличиваться (или уменьшаться) по мере увеличения числа друзей.

Или возможно, что чем больше друзей у вас есть, тем больше времени вы проводите на веб-сайте *до определенного момента*, после которого дальнейшее увеличение числа друзей ведет к уменьшению проводимого вами времени на веб-сайте. (Может быть, при слишком большом числе друзей опыт взаимодействия станет слишком подавляющим?) Вы могли бы попробовать уловить это в нашей модели, добавив еще одну переменную в виде *квадрата* числа друзей.

Как только мы начинаем добавлять переменные, следует позаботиться о том, чтобы их коэффициенты имели "смысл". Добавлять же произведения, логарифмы, квадраты и более высокие степени можно без ограничений.

Качество подгонки

Снова обратимся к *R*-квадрату:

```
from scratch.simple_linear_regression import total_sum_of_squares

def multiple_r_squared(xs: List[Vector], ys: Vector, beta: Vector) -> float:
    sum_of_squared_errors = sum(error(x, y, beta) ** 2
                                for x, y in zip(xs, ys))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(ys)
```

который теперь вырос до 0.68:

```
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta) < 0.68
```

Однако имейте в виду, что добавление в регрессию новых переменных *неизбежно* увеличивает *R*-квадрат. В конце концов простая регрессионная модель является лишь частным случаем множественной регрессионной модели, где коэффициенты при "рабочих_часах" и "ученой_степени" равны 0. Оптимальная множественная регрессионная модель будет неизбежно иметь ошибку как минимум столь же малую, что и у той модели.

По этой причине в множественной регрессионной модели нам также необходимо рассматривать *стандартные ошибки* коэффициентов, которые измеряют уровень нашей уверенности в оценках каждого β_i. Регрессии в целом могут очень хорошо вписываться в наши данные, однако если некоторые независимые переменные коррелируют (или нерелевантны), то их коэффициенты могут не иметь большого *смысла*.

Типичный подход к измерению этих ошибок начинается с еще одного допущения — о том, что ошибки ε_i являются взаимно независимыми нормально распределенными случайными величинами с нулевым средним значением и неким совместным (неизвестным) стандартным отклонением σ. В этом случае мы (или, скорее всего, наше статистическое программное обеспечение) может использовать немного линейной алгебры для отыскания стандартной ошибки каждого коэффициента. Чем она больше, тем меньше наша модель уверена в коэффициенте. К сожалению, мы не подготвлены к тому, чтобы выполнять подобного рода линейно-алгебраические вычисления с нуля.

Отступление: размножение выборок

Представим, что у нас есть выборка из *n* точек данных, сгенерированная некоторым (неизвестным) распределением:

```
data = get_sample(num_points=n)  # Получить выборку размером n точек
```

В *главе 5* мы реализовали функцию, способную вычислять медиану выборки, которую мы можем использовать в качестве оценки медианы самого распределения.

Но насколько можно доверять нашей оценке? Если все значения в выборке очень близки к 100, то, скорее всего, фактическая медиана тоже будет близка к 100. Если

же приблизительно половина значений в выборке близка к 0, а другая половина близка к 200, то нельзя быть настолько же уверенным в отношении медианы.

Если бы мы могли многократно брать новые выборки, тогда можно было бы вычислить медианы многочисленных выборок и взглянуть на распределение этих медиан. Обычно это сделать нельзя. Но мы можем их размножить или, выражаясь технически, *бутстрапировать*, т. е. сгенерировать новые наборы данных, выбирая из наших данных *n* точек с *их возвратом* назад (такие выборки называются бутстраповскими). И затем можно вычислить медианы этих синтетических наборов данных[2]:

```
from typing import TypeVar, Callable

X = TypeVar('X')         # Обобщенный тип для данных
Stat = TypeVar('Stat')   # Обобщенный тип для "статистики"

def bootstrap_sample(data: List[X]) -> List[X]:
    """Случайно отбирает len(data) элементов с их возвратом назад"""
    return [random.choice(data) for _ in data]

def bootstrap_statistic(data: List[X],
                        stats_fn: Callable[[List[X]], Stat],
                        num_samples: int) -> List[Stat]:
    """Оценивает функцию stats_fn на num_samples числе
       бутстраповских выборок из данных"""
    return [stats_fn(bootstrap_sample(data)) for _ in range(num_samples)]
```

Например, рассмотрим следующие два набора данных:

```
# Значения всех 101 точки очень близки к 100
close_to_100 = [99.5 + random.random() for _ in range(101)]

# Значения 50 из 101 близки к 0, значения других 50 находятся рядом с 200
far_from_100 = ([99.5 + random.random()] +
                [random.random() for _ in range(50)] +
                [200 + random.random() for _ in range(50)])
```

Если вы вычислите медиану каждой выборки, то обе будут находиться очень близко к 100. Однако если вы взглянете на:

```
from scratch.statistics import median, standard_deviation

medians_close = bootstrap_statistic(close_to_100, median, 100)
```

то главным образом увидите числа, находящиеся действительно близко к 100, а если взгляните на:

```
medians_far = bootstrap_statistic(far_from_100, median, 100)
```

[2] Выборка с возвратом аналогична неоднократному вытягиванию случайной карты из колоды игральных карт, когда после каждого вытягивания карта возвращается назад в колоду. В результате время от времени обязательно будет попадаться карта, которая уже выбиралась. — *Прим. пер.*

то увидите много чисел, находящихся близко к 0, и много чисел, находящихся близко к 200.

Стандартное отклонение первого набора медиан находится близко к 0, тогда как для второго набора медиан оно находится близко к 100:

```
assert standard_deviation(medians_close) < 1
assert standard_deviation(medians_far) > 90
```

(Этот предельный случай было бы легко выявить, обследовав данные вручную, чего нельзя сказать в общем случае.)

Стандартные ошибки регрессионных коэффициентов

Мы можем принять тот же самый подход к оцениванию стандартных ошибок наших регрессионных коэффициентов. Мы многократно берем бутстраповскую выборку `bootstrap_sample` из данных и оцениваем `beta` на этой выборке. Если коэффициент, соответствующий одной из независимых переменных (скажем, `num_friends`), не сильно варьирует по всем выборкам, то мы можем быть уверенными, что наша оценка близка к оптимальной. Если же коэффициент сильно варьирует по всем выборкам, то мы абсолютно не можем быть уверенными в оценке.

Единственная тонкость состоит в том, что перед взятием выборки нам необходимо объединить наши данные `x` и данные `y` с помощью функции `zip` в целях обеспечения того, чтобы соответствующие значения независимой и зависимой переменных отбирались вместе. Это означает, что функция `bootstrap_sample` будет возвращать список пар `(x_i, y_i)`, которые потом нужно снова собрать в списки `x_sample` и `y_sample`:

```
from typing import Tuple

import datetime

def estimate_sample_beta(pairs: List[Tuple[Vector, float]]):
    x_sample = [x for x, _ in pairs]
    y_sample = [y for _, y in pairs]
    beta = least_squares_fit(x_sample, y_sample, learning_rate, 5000, 25)
    print("бутстраповская выборка", beta)
    return beta

random.seed(0) # чтобы вы получили те же самые результаты, что и я

# Это займет пару минут!
bootstrap_betas = bootstrap_statistic(list(zip(inputs,
                                               daily_minutes_good)),
                                      estimate_sample_beta,
                                      100)
```

После этого мы можем оценить стандартное отклонение каждого коэффициента:

```
bootstrap_standard_errors = [
    standard_deviation([beta[i] for beta in bootstrap_betas])
    for i in range(4)]

# [1.272,  # константный член,  фактическая ошибка = 1.19
#  0.103,  # число друзей,      фактическая ошибка = 0.080
#  0.155,  # рабочие часы,      фактическая ошибка = 0.127
#  1.249]  # ученая степень,    фактическая ошибка = 0.998
```

(Мы, вероятно, получили бы более точные оценки, если бы собрали более 100 выборок и использовали бы более 5000 итераций для оценивания каждого `beta`, но у нас нет целого дня свободного времени.)

Мы можем использовать их для проверки статистических гипотез, таких как "равняется ли β_1 нулю". При нулевой гипотезе $\beta_j = 0$ (и с другими допущениями о распределении ε_j) статистика:

$$t_j = \hat{\beta}_j / \hat{\sigma}_j,$$

которая представляет собой оценку β_1, деленную на оценку ее стандартной ошибки, подчиняется *t-распределению Стьюдента*[3] с "$n - k$ степенями свободы".

Если бы у нас была функция `students_t_cdf`, мы могли бы вычислить *p*-значения для каждого коэффициента наименьших квадратов, показав вероятность того, что мы будем наблюдать такое значение, если бы фактический коэффициент был равен 0. К сожалению, у нас нет такой функции. (Хотя ее можно было бы реализовать, если бы мы не работали с нуля.)

Тем не менее по мере увеличения степеней свободы *t*-распределение становится все ближе к стандартному нормальному распределению. В подобной ситуации, когда *n* намного больше *k*, мы можем применить функцию `normal_cdf` и при этом остаться довольными собой:

```
from scratch.probability import normal_cdf

def p_value(beta_hat_j: float, sigma_hat_j: float) -> float:
    if beta_hat_j > 0:
        # Если коэффициент является положительным, то
        # нам нужно вычислить удвоенную вероятность
        # наблюдать еще более *крупное* значение
        return 2 * (1 - normal_cdf(beta_hat_j / sigma_hat_j))
```

[3] Распределение Стьюдента — это однопараметрическое семейство абсолютно непрерывных распределений, которое по сути представляет собой сумму нескольких нормально распределенных случайных величин. Чем больше величин, тем больше вероятность, что их сумма будет иметь нормальное распределение. Таким образом, количество суммируемых величин определяет важнейший параметр формы данного распределения — число степеней свободы (см. **https://ru.wikipedia.org/wiki/Распределение_Стьюдента**). — *Прим. пер.*

```
    else:
        # В противном случае удвоенную вероятность
        # наблюдать *меньшее* значение
        return 2 * normal_cdf(beta_hat_j / sigma_hat_j)

assert p_value(30.58, 1.27) < 0.001    # константный член
assert p_value(0.972, 0.103) < 0.001   # число друзей
assert p_value(-1.865, 0.155) < 0.001  # рабочие часы
assert p_value(0.923, 1.249) > 0.4     # ученая степень
```

(В ситуации, не такой как эта, мы бы, вероятно, использовали статистическое программное обеспечение, которое знает, как вычислять *t*-распределение, а также как вычислять точные стандартные ошибки.)

В отличие от большинства коэффициентов, которые имеют очень малые *p*-значения (намекая на то, что они действительно ненулевые), коэффициент для "ученой_степени" не "значимо" отличается от нуля, тем самым повышая вероятность того, что он является скорее случайным, чем содержательным.

В сценариях более сложной регрессии иногда требуется проверить более сложные гипотезы о данных, такие как "по крайней мере один из β_j не равен нулю" или "β_1 равен β_2 и β_3 равен β_4". Вы можете сделать это с помощью *F-теста*[4], но этот вопрос, увы, выходит за рамки настоящей книги.

Регуляризация

На практике часто хочется применять линейную регрессию к наборам данных с большим числом переменных. Это создает пару лишних затруднений. Во-первых, чем больше переменных, тем больше шансов достичь переподгонки модели к тренировочным данным. И, во-вторых, чем больше ненулевых коэффициентов, тем труднее в них разобраться. Если задача заключается в *объяснении* некоторого явления, то разреженная модель с тремя факторами в практическом плане может оказаться полезнее, чем модель чуть получше, но с сотнями факторов.

Регуляризация — это подход, заключающийся в добавлении в член ошибки штрафа, который увеличивается по мере того, как `beta` становится крупнее. Затем мы минимизируем сочетание ошибки и штрафа. Чем большее значение мы придаем штрафу, тем больше мы противодействуем крупным коэффициентам.

Например, в *гребневой регрессии* мы добавляем штраф, пропорциональный сумме квадратов `beta_i` (за исключением того, что мы не штрафуем `beta_0`, константный член):

```
# alpha - это *гиперпараметр*, управляющий тем, каким грубым будет штраф.
# Иногда он называется "лямбдой", но это слово уже имеет
# свое применение в Python
```

[4] *F*-тестом называется любая проверка статистической гипотезы, статистика которой при соблюдении нулевой гипотезы имеет распределение Фишера (*F*-распределение). — *Прим. пер.*

```
def ridge_penalty(beta: Vector, alpha: float) -> float:
    return alpha * dot(beta[1:], beta[1:])

def squared_error_ridge(x: Vector,
                        y: float,
                        beta: Vector,
                        alpha: float) -> float:
    """Оценить ошибку плюс гребневый штраф на beta"""
    return error(x, y, beta) ** 2 + ridge_penalty(beta, alpha)
```

Затем мы можем подключить эту функцию к градиентному спуску обычным образом:

```
from scratch.linear_algebra import add

def ridge_penalty_gradient(beta: Vector, alpha: float) -> Vector:
    """Градиент только гребневого штрафа"""
    return [0.] + [2 * alpha * beta_j for beta_j in beta[1:]]

def sqerror_ridge_gradient(x: Vector,
                           y: float,
                           beta: Vector,
                           alpha: float) -> Vector:
    """Градиент, соответствующий i-му члену квадратической ошибки
       including the ridge penalty"""
    return add(sqerror_gradient(x, y, beta),
               ridge_penalty_gradient(beta, alpha))
```

И затем нам просто нужно модифицировать функцию `least_squares_fit`, применив вместо градиентной функции `sqerror_gradient` градиентную функцию `sqerror_ridge_gradient`. (Я не буду здесь повторять весь код.)

При `alpha = 0` штраф отсутствует совсем, и мы получаем те же самые результаты, что и раньше:

```
random.seed(0)
beta_0 = least_squares_fit_ridge(inputs, daily_minutes_good, 0.0, # alpha
                                 learning_rate, 5000, 25)
# [30.51, 0.97, -1.85, 0.91]
assert 5 < dot(beta_0[1:], beta_0[1:]) < 6
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_0) < 0.69
```

По мере увеличения `alpha` качество подгонки ухудшается, но размер `beta` становится меньше:

```
beta_0_1 = least_squares_fit_ridge(inputs, daily_minutes_good, 0.1, # alpha
                                   learning_rate, 5000, 25)

# [30.8, 0.95, -1.83, 0.54]
assert 4 < dot(beta_0_1[1:], beta_0_1[1:]) < 5
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_0_1) < 0.69
```

```
beta_1 = least_squares_fit_ridge(inputs, daily_minutes_good, 1, # alpha
                                 learning_rate, 5000, 25)

# [30.6, 0.90, -1.68, 0.10]
assert 3 < dot(beta_1[1:], beta_1[1:]) < 4
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_1) < 0.69

beta_10 = least_squares_fit_ridge(inputs, daily_minutes_good,10, # alpha
                                  learning_rate, 5000, 25)

# [28.3, 0.67, -0.90, -0.01]
assert 1 < dot(beta_10[1:], beta_10[1:]) < 2
assert 0.5 < multiple_r_squared(inputs, daily_minutes_good, beta_10) < 0.6
```

В частности, коэффициент к параметру "ученая_степень" исчезает по мере увеличения штрафа, что согласуется с нашим предыдущим результатом, который значимо не отличался от нуля.

Обычно перед применением этого подхода необходимо выполнить шкалирование данных. В конце концов, если вы изменили годы стажа на столетия, то соответствующий коэффициент наименьших квадратов увеличится в 100 раз и внезапно будет оштрафован намного больше, даже если модель та же самая.

Еще один подход — это *лассо-регрессия*, в которой используется штраф:

```
def lasso_penalty(beta, alpha):
    return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

В отличие от гребневого штрафа, который в совокупности уменьшает коэффициенты, лассо-штраф стремится обнулить их, что делает этот метод полезным для усвоения разреженных моделей. К сожалению, он не поддается градиентному спуску, и поэтому мы не сможем решить эту задачу с нуля.

Для дальнейшего изучения

- Регрессия в своей основе имеет богатую и обширную теорию. Это еще одна тема, при изучении которой вам следует подумать о прочтении учебника либо, по крайней мере, большого числа статей из Википедии.
- Библиотека scikit-learn включает модуль `linear_model`[5], который обеспечивает класс линейной регрессионной модели `LinearRegression`, аналогичной нашей, а также гребневую регрессию, лассо-регрессию и другие типы регуляризации.
- StatsModels[6] — это еще один модуль Python, который (среди всего прочего) содержит линейные регрессионные модели.

[5] См. **https://scikit-learn.org/stable/modules/linear_model.html**.

[6] См. **https://www.statsmodels.org/**.

Логистическая регрессия

> Многие говорят, что между гениальностью и сумасшествием прочерчена тонкая линия. Не думаю, что она тонкая. На самом деле она представляет собой зияющую пропасть.
>
> – *Билл Бэйли*[1]

В *главе 1* мы кратко рассмотрели задачу с попыткой предсказать, какие из пользователей DataSciencester оплачивают свои привилегированные аккаунты. В данной главе мы пересмотрим эту задачу.

Задача

У нас есть анонимизированный набор данных примерно на 200 пользователей, содержащий зарплату каждого пользователя, его опыт работы исследователем данных и состояние его платежей за учетную запись в соцсети (рис. 16.1). Как это

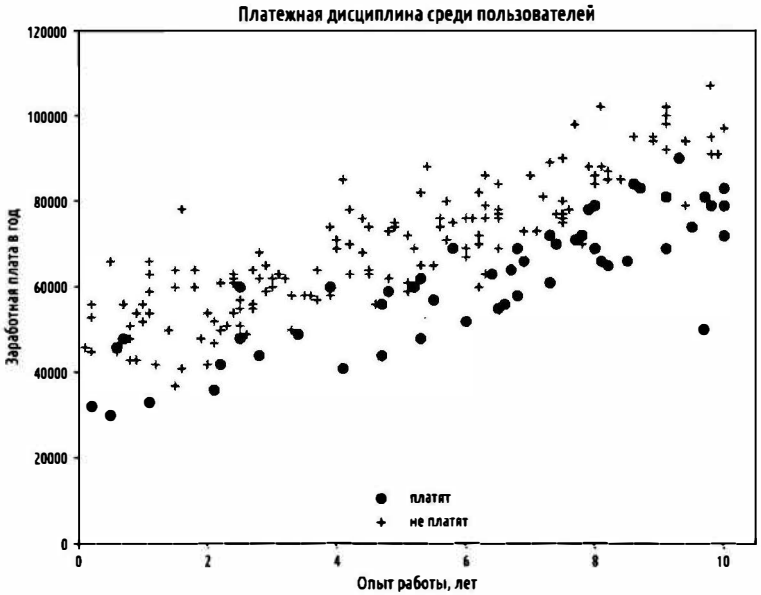


Рис. 16.1. Пользователи, оплатившие и не оплатившие аккаунты

[1] Билл Бэйли (род. 1964) — британский комик, музыкант и актер — *Прим. пер.*

обычно делается с категориальными переменными, мы представляем зависимую переменную равной либо 0 (привилегированный аккаунт отсутствует), либо 1 (привилегированный аккаунт есть).

Как обычно, наши данные представляют собой список строк `[опыт работы, зарплата, оплаченный_аккаунт]`. Давайте превратим их в формат, который нам нужен:

```
x = [[1.0] + row[:2] for row in data]  # [1, опыт, зарплата]
y = [row[2] for row in data]           # оплаченный_аккаунт
```

Очевидной первой попыткой будет применение линейной регрессии и отыскание наилучшей модели:

$$\text{оплаченный\_аккаунт} = \beta_0 + \beta_1 \cdot \text{опыт} + \beta_2 \cdot \text{зарплата} + \varepsilon$$

И конечно же, ничто не мешает смоделировать задачу таким образом:

```
from matplotlib import pyplot as plt
from scratch.working_with_data import rescale
from scratch.multiple_regression import least_squares_fit, predict
from scratch.gradient_descent import gradient_step

learning_rate = 0.001

rescaled_xs = rescale(xs)
beta = least_squares_fit(rescaled_xs, ys, learning_rate, 1000, 1)

# [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_xs]

plt.scatter(predictions, ys)
plt.xlabel("Предсказание")
plt.ylabel("Факт")
plt.show()
```

Результаты показаны на рис. 16.2.

Однако этот подход создает пару непосредственных проблем.

- Мы хотели бы, чтобы наши предсказанные результаты равнялись 0 либо 1, указывая на принадлежность к классу. Замечательно, если они будут в интервале между 0 и 1, поскольку их можно интерпретировать как вероятности — результат 0.25 мог бы означать 25%-ный шанс, что это пользователь с оплаченным аккаунтом. Однако результаты линейной модели могут быть огромными положительными числами или даже отрицательными числами. И тогда не ясно, как их интерпретировать. И действительно, здесь многие наши предсказания были отрицательными.
- Линейная регрессионная модель исходила из того, что ошибки не коррелируют со столбцами x. Однако здесь регрессионный коэффициент для опыта работы `experience` равен 0.43, указывая на то, что больший опыт работы ведет к большему правдоподобию привилегированного аккаунта. Это означает, что наша модель выдает очень большие значения для людей с очень большим опытом

работы. Но мы знаем, что фактические значения должны быть не выше 1, и значит, неизбежно очень большие результаты (и, следовательно, очень большие значения опыта работы `experience`) соответствуют очень большим отрицательным значениям члена ошибки. Поскольку это как раз тот случай, то наша оценка `beta` является смещенной.

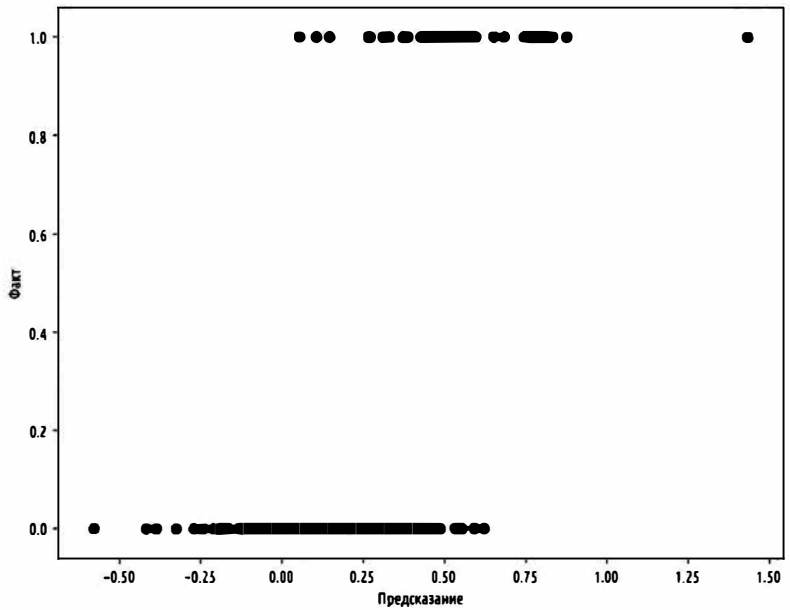


Рис. 16.2. Применение линейной регрессии для предсказания оплаты премиальных аккаунтов

Вместо этого мы хотели бы, чтобы крупные положительные значения функции `dot(x_i, beta)` соответствовали вероятностям, близким к 1, а крупные отрицательные значения — вероятностям, близким к 0. Мы можем добиться этого, применив к результату еще одну функцию.

Логистическая функция

В случае логистической регрессии мы используем *логистическую функцию*, изображенную на рис. 16.3:

```
def logistic(x: float) -> float:
    return 1.0 / (1 + math.exp(-x))
```

По мере того как данные на ее входе становятся большими и положительными, функция стремится к 1. По мере того как входные данные становятся большими и отрицательными, функция приближается к 0. В дополнение к этому у нее есть еще одно удобное свойство — ее производная задается функцией:

```
def logistic_prime(x: float) -> float:
    return logistic(x) * (1 - logistic(x))
```

к которой мы прибегнем ниже.

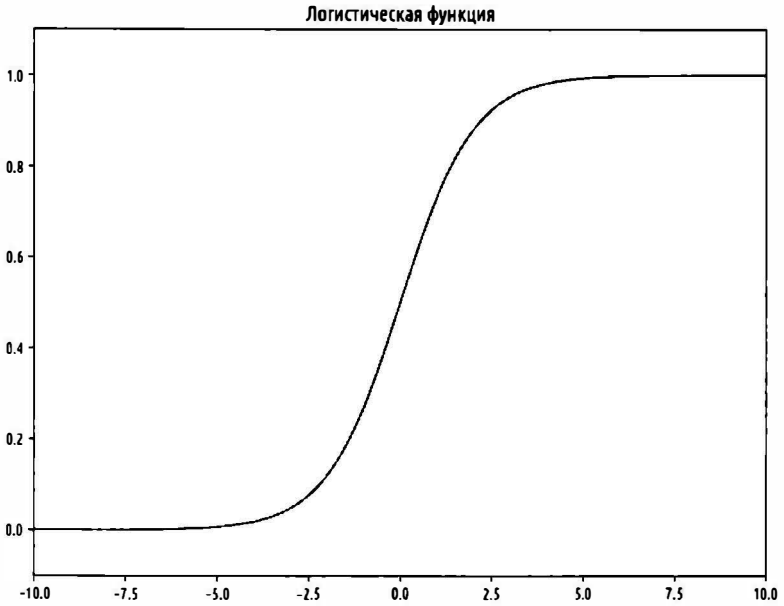


Рис. 16.3. Логистическая функция

Мы будем использовать ее для подгонки модели:

$$y_i = f(x_i\beta) + \varepsilon_i,$$

где f — это логистическая функция `logistic`.

Вспомните, что для линейной регрессии мы выполняем подгонку модели, минимизируя сумму квадратов ошибок, и эта подгонка завершалась выбором коэффициента β, который максимизировал правдоподобие данных.

Здесь эти операции не эквивалентны, поэтому мы будем применять градиентный спуск для непосредственной максимизации правдоподобия. Иными словами, нам необходимо вычислить функцию правдоподобия и ее градиент.

С учетом некоторого β наша модель сообщает нам, что каждый y_i должен быть равным 1 с вероятностью $f(x_i\beta)$ и равным 0 с вероятностью $1 - f(x_i\beta)$.

В частности, функция плотности вероятности для y_i может быть записана следующим образом:

$$p(y_i \mid x_i, \beta) = f(x_i\beta)^{y_i}\left(1 - f(x_i\beta)\right)^{1-y_i},$$

поскольку при $y_i = 0$ она равна:

$$1 - f(x_i\beta),$$

а при $y_i = 1$ она равна:

$$f(x_i\beta).$$

Оказывается, что на самом деле проще максимизировать *логарифмическое правдоподобие*:

$$\ln L(\beta | x_i, y_i) = y_i \ln f(x_i\beta) + (1 - y_i) \ln(1 - f(x_i\beta)).$$

Поскольку логарифм — это монотонно возрастающая функция, то любой `beta`, который максимизирует логарифмическое правдоподобие, также максимизирует правдоподобие, и наоборот. Поскольку градиентный спуск все минимизирует, мы на самом деле будем работать с отрицательным логарифмическим правдоподобием, т. к. максимизировать правдоподобие — это то же самое, что минимизировать его отрицание.

```
import math
from scratch.linear_algebra import Vector, dot

def _negative_log_likelihood(x: Vector, y: float, beta: Vector) -> float:
    """Отрицательное логарифмическое правдоподобие
       для одной точки данных"""
    if y == 1:
        return -math.log(logistic(dot(x, beta)))
    else:
        return -math.log(1 - logistic(dot(x, beta)))
```

Если допустить, что разные точки данных независимы друг от друга, то полное правдоподобие будет произведением индивидуальных правдоподобий. Это означает, что полное логарифмическое правдоподобие является суммой индивидуальных логарифмических правдоподобий:

```
def negative_log_likelihood(xs: List[Vector],
                            ys: List[float],
                            beta: Vector) -> float:
    return sum(_negative_log_likelihood(x, y, beta)
               for x, y in zip(xs, ys))
```

Немного дифференциального исчисления дает нам градиент:

```
from scratch.linear_algebra import vector_sum

def _negative_log_partial_j(x: Vector, y: float, beta: Vector, j: int) -> float:
    """j-я частная производная для одной точки данных.
       Здесь i является индексом точки данных"""
    return -(y - logistic(dot(x, beta))) * x[j]

def _negative_log_gradient(x: Vector, y: float, beta: Vector) -> Vector:
    """Градиент для одной точки данных"""
    return [_negative_log_partial_j(x, y, beta, j)
            for j in range(len(beta))]

def negative_log_gradient(xs: List[Vector],
                          ys: List[float],
                          beta: Vector) -> Vector:
```

```
    return vector_sum([_negative_log_gradient(x, y, beta)
                       for x, y in zip(xs, ys)])
```

И в этом месте у нас есть все, что нам необходимо.

Применение модели

Мы хотели бы разбить наши данные на тренировочный и тестовый наборы:

```
from scratch.machine_learning import train_test_split
import random
import tqdm

random.seed(0)

x_train, x_test, y_train, y_test = train_test_split(rescaled_xs, ys, 0.33)

learning_rate = 0.01            # Темп усвоения

# Подобрать случайную отправную точку
beta = [random.random() for _ in range(3)]

with tqdm.trange(5000) as t:
    for epoch in t:
        gradient = negative_log_gradient(x_train, y_train, beta)
        beta = gradient_step(beta, gradient, -learning_rate)
        loss = negative_log_likelihood(x_train, y_train, beta)
        t.set_description(f"потеря: {loss:.3f} beta: {beta}")
```

После чего мы обнаруживаем, что `beta` приближенно равен:

```
[-2.0, 4.7, -4.5]
```

Эти коэффициенты принадлежат шкалированным (`rescaled`) данным, но мы также можем преобразовать их назад в исходные данные:

```
from scratch.working_with_data import scale

means, stdevs = scale(xs)
beta_unscaled = [(beta[0]
                  - beta[1] * means[1] / stdevs[1]
                  - beta[2] * means[2] / stdevs[2]),
                 beta[1] / stdevs[1],
                 beta[2] / stdevs[2]]
# [8.9, 1.6, -0.000288]
```

К сожалению, эти данные не так просто интерпретировать по сравнению с коэффициентами линейной регрессии. При прочих равных дополнительный год опыта работы добавляет 1.6 во вход в логистическую функцию `logistic`. При прочих равных дополнительные $10 000 в зарплате вычитают 2.48 из входа в логистическую функцию.

Влияние на результат, однако, также зависит от других входов. Если результат `dot(beta, x_i)` уже является крупным (соответствуя вероятности, близкой к 1), то даже значительное его увеличение не сможет сильно повлиять на вероятность. Если же он близок к 0, то даже незначительное его увеличение может существенно увеличить вероятность.

Мы можем сказать, что — при прочих равных — люди с бóльшим опытом работы оплачивают аккаунты с большей вероятностью и что — при прочих равных — люди с более высокими зарплатами оплачивают аккаунты с меньшей вероятностью. (Это также было очевидно, когда мы вывели данные на график.)

Качество подгонки

Мы еще не использовали тестовые данные, которые отложили в сторону. Давайте посмотрим, что произойдет, если предсказывать *оплаченный аккаунт* всякий раз, когда вероятность превышает 0.5:

```
# Истинные утверждения, ложные утверждения,
# истинные отрицания, ложные отрицания
true_positives = false_positives = true_negatives = false_negatives = 0

for x_i, y_i in zip(x_test, y_test):
    prediction = logistic(dot(beta, x_i))

    if y_i == 1 and prediction >= 0.5: # TP: оплачен, и мы предсказали,
        true_positives += 1            # что оплачен
    elif y_i == 1:                     # FN: оплачен, и мы предсказали,
        false_negatives += 1           # что не оплачен
    elif prediction >= 0.5:            # FP: не оплачен, и мы предсказали,
        false_positives += 1           # что оплачен
    else:                              # TN: не оплачен, и мы предсказали,
        true_negatives += 1            # что не оплачен

precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
```

В результате мы получим прецизионность 75% ("когда мы предсказываем оплачиваемый аккаунт, мы правы в 75% случаях") и полноту 80% ("когда пользователь оплатил аккаунт, мы предсказываем оплаченный аккаунт в 80% случаях"), что не так ужасно, учитывая малое количество данных, которое у нас есть.

Мы также можем сопоставить предсказания и фактические данные на графике (рис. 16.4), который иллюстрирует хорошую результативность модели:

```
predictions = [logistic(dot(beta, x_i)) for x_i in x_test]
plt.scatter(predictions, y_test, marker='+')
plt.xlabel("Предсказанная вероятность")
plt.ylabel("Фактический результат")
plt.title("Логистическая регрессия: предсказания и факт")
plt.show()
```

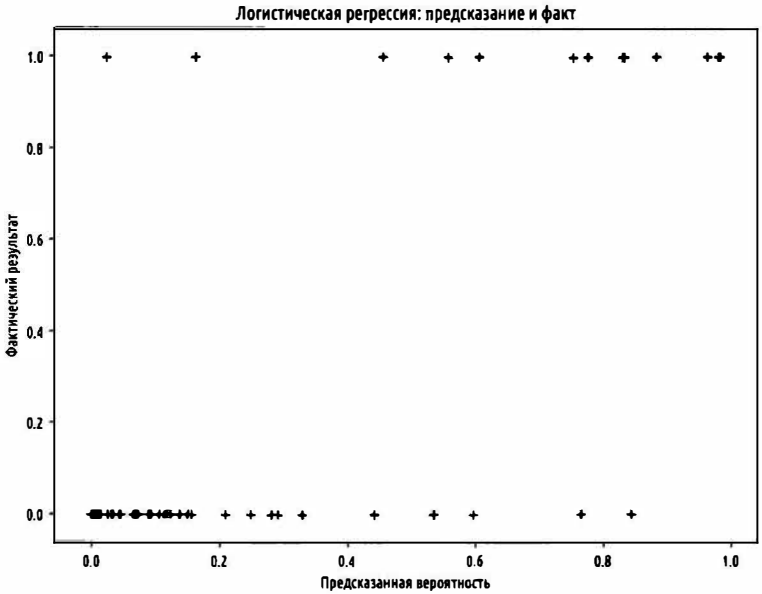


Рис. 16.4. Логистическая регрессия: предсказания в сопоставлении с фактическими данными

Машины опорных векторов

Множество точек, где `dot(beta_hat, x_i)` равняется 0, представляет границу между нашими классами. Мы можем построить график для того, чтобы увидеть, чтó конкретно модель делает (рис. 16.5).

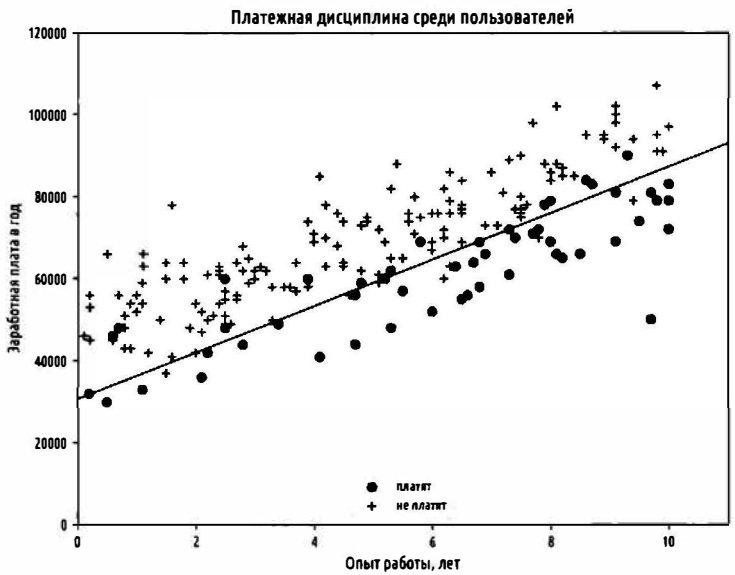


Рис. 16.5. Оплатившие и неоплатившие пользователи с границей принятия решения

Эта граница является *гиперплоскостью*, которая разделяет параметрическое пространство на два полупространства, соответствующие *предсказанию оплаты* и *предсказанию неоплаты*. Мы выявили ее в качестве побочного эффекта обнаружения наиболее правдоподобной логистической модели.

Альтернативный подход к классифицированию заключается в поиске гиперплоскости, которая "наилучшим образом" разделяет классы в тренировочных данных. Эта идея лежит в основе *машины опорных векторов*, которая отыскивает гиперплоскость, максимизирующую расстояние до ближайшей точки в каждом классе (рис. 16.6).

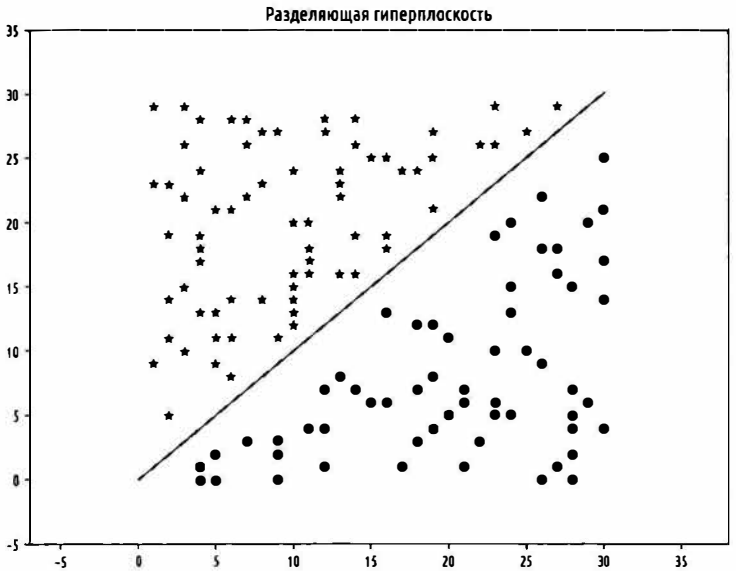


Рис. 16.6. Разделяющая гиперплоскость

Отыскание такой гиперплоскости является оптимизационной задачей, которая предусматривает использование технических решений, находящихся на более продвинутом уровне. Другая проблема заключается в том, что разделяющая гиперплоскость может не существовать вообще. В нашем наборе данных об оплате привилегированных аккаунтов просто нет линии, которая идеально отделяет оплативших пользователей от неоплативших.

Иногда мы можем обойти эту проблему, преобразовав данные в более высокоразмерное пространство. Например, рассмотрим простой одномерный набор данных, показанный на рис. 16.7.

Ясно, что здесь нет гиперплоскости, которая могла бы отделить положительные образцы от отрицательных. С другой стороны, посмотрим, что произойдет, если отобразить этот набор данных в две размерности, направив точку `x` в `(x, x**2)`. Неожиданно обнаружится, что можно найти гиперплоскость, разделяющую данные (рис. 16.8).

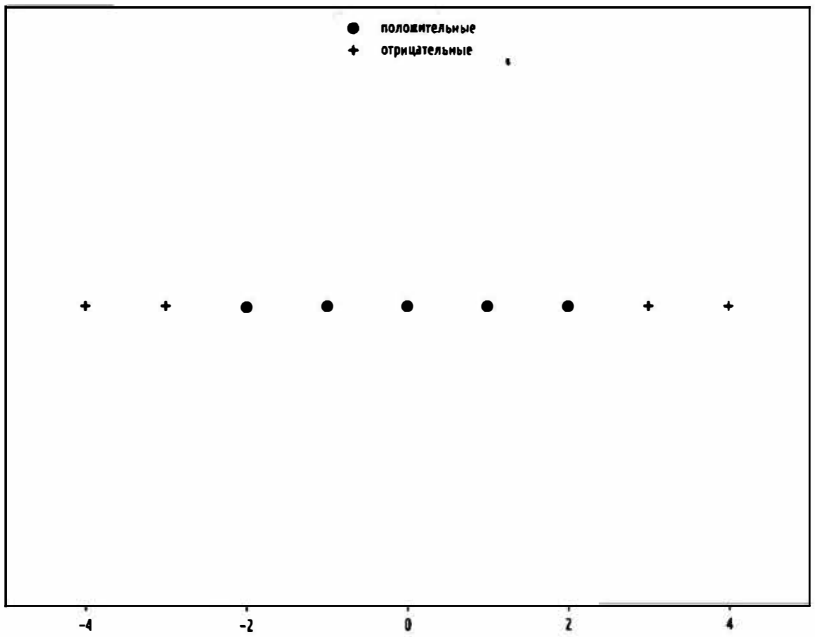


Рис. 16.7. Неразделимый одномерный набор данных

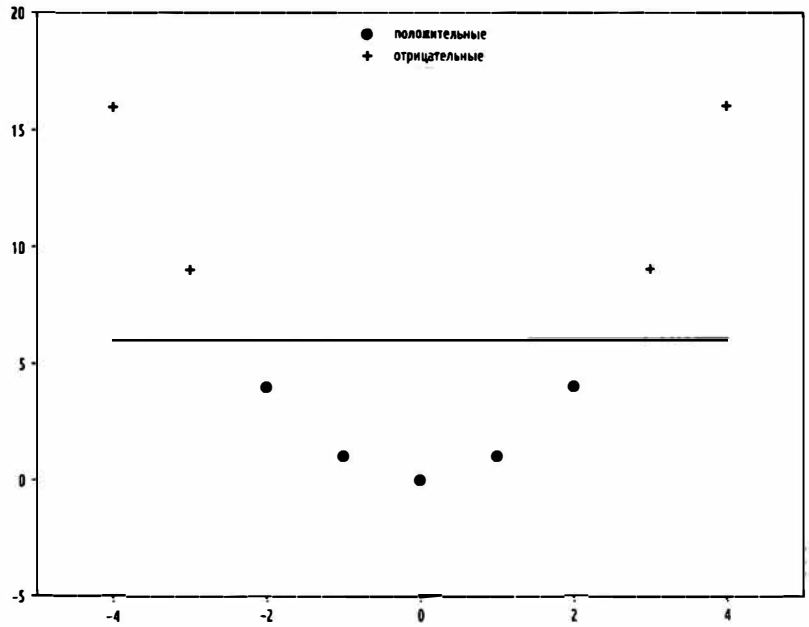


Рис. 16.8. Набор данных становится разделимым в более высоких размерностях

Обычно этот прием называется *ядерным трюком* (kernel trick), т. к. вместо фактического отображения точек в более высокоразмерное пространство (что может быть затратным, если точек много, и отображение осложнено) используется "ядерная" функция, которая вычисляет скалярные произведения в более высокоразмерном пространстве и использует их результаты для отыскания гиперплоскости.

Трудно (и, вероятно, неразумно) использовать опорно-векторные машины без специализированного оптимизационного программного обеспечения, написанного специалистами с соответствующим опытом, поэтому изложение данной темы придется на этом закончить.

Для дальнейшего изучения

- Библиотека scikit-learn имеет модули как для логистической регрессии[2], так и для машин опорных векторов[3].
- Библиотека LIBSVM[4] является имплементацией опорно-векторных машин, которая используется в scikit-learn за кулисами. Веб-сайт библиотеки располагает большим количеством разнообразной полезной документации по опорно-векторным машинам.

[2] См. **https://scikit-learn.org/stable/modules/linear_model.html%23logistic-regression**

[3] См. **https://scikit-learn.org/stable/modules/svm.html**

[4] См. **https://www.csie.ntu.edu.tw/~cjlin/libsvm/**.

ГЛАВА 17

Деревья решений

Дерево — это непостижимая загадка.
– Джим Вудринг[1]

Директор подразделения по поиску талантов с переменным успехом провел ряд собеседований с претендентами, чьи заявления поступили с веб-сайта DataSciencester, и в результате он собрал набор данных, состоящий из нескольких (качественных) признаков, описывающих каждого претендента, а также о том, прошел претендент собеседование успешно или нет. Он интересуется, можно ли использовать эти данные для построения модели, выявляющей, какие претенденты пройдут собеседование успешно, благодаря чему ему не придется тратить время на проведение собеседований.

Эта задача, судя по всему, хорошо подходит для модели на основе еще одного инструмента предсказательного моделирования в арсенале исследователя данных.

Что такое дерево решений?

В модели деревьев решений используется древесная структура для представления ряда возможных *путей принятия решения* и результата для каждого пути.

Тот, кто когда-либо играл в игру "20 вопросов"[2], уже знаком с деревьями решений, даже не подозревая об этом. Вот пример.

— *Я думаю о животном*.

— У него больше пяти лап?

— *Нет*.

— Он вкусный?

— *Нет*.

— Его можно увидеть на обратной стороне австралийского 5-центовика?

— *Да*.

— Это ехидна?

— *Да, ты угадал!*

---

[1] Джим Вудринг (род. 1952) — американский мультипликатор, художник изящных искусств, писатель и дизайнер игрушек — *Прим. пер.*

[2] См. **https://en.wikipedia.org/wiki/Twenty_Questions.**

Это соответствует пути:

"Не более 5 лап" → "Невкусный" → "На 5-центовой монете" → "Ехидна!"

на своеобразном (и не очень подробном) дереве решений "Угадать животное" (рис. 17.1).

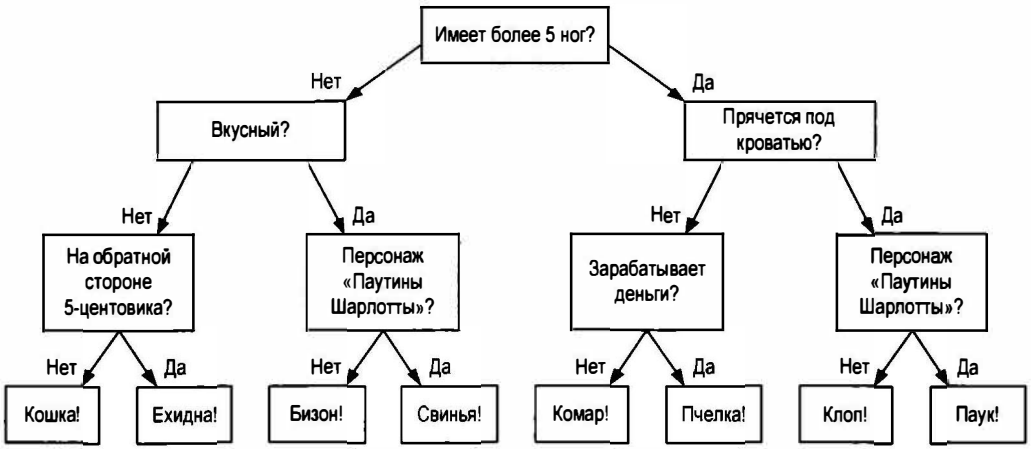


Рис. 17.1. Дерево решений "Угадать животное"

Деревья решений имеют много достоинств. Они понятны и легко интерпретируемы, а процесс, следуя которому они достигают предсказания, является абсолютно прозрачным. В отличие от других моделей, которые мы рассмотрели до этого, деревья решений легко справляются с сочетанием численных (количество ног) и категориальных (вкусный/невкусный) атрибутов и даже могут классифицировать данные, для которых атрибуты отсутствуют.

Вместе с тем задача отыскания "оптимального" дерева решений для набора тренировочных данных является вычислительно очень сложной. (Мы обойдем проблему путем построения достаточно хорошего дерева вместо оптимального, хотя для крупных наборов данных может потребоваться много работы.) Важнее то, что очень легко (и это замечательно) строить деревья решений, которые *переподогнаны* к тренировочным данным и не делают хороших обобщений на ранее не встречавшихся данных. Мы обратимся к способам решения этих сложностей.

Обычно деревья решений подразделяются на *классификационные деревья* (которые производят категорийные результаты) и *регрессионные деревья* (которые производят численные или непрерывные результаты). В этой главе мы сосредоточимся на классификационных деревьях и проанализируем алгоритм ID3 для усвоения дерева решений из набора помеченных данных, что должно помочь нам понять то, как деревья решений работают на самом деле. Не усложняя, мы ограничимся задачами с бинарными результатами типа "Следует ли нанять этого претендента?", или "Какое из двух рекламных сообщений следует показать этому посетителю веб-сайта: A или B?", или "Будет ли меня тошнить, если съесть эту пищу из офисного холодильника?".

Энтропия

Для того чтобы построить дерево решений, нам нужно определиться, какие вопросы задавать и в каком порядке. На каждом уровне дерева имеется несколько возможностей, которые мы исключили, и несколько возможностей, которые еще остались. Усвоив, что у животного не более пяти ног, мы исключаем возможность, что это кузнечик, но мы не исключили возможность, что это утка. Каждый возможный вопрос разделяет оставшиеся возможности в соответствии с полученным ответом.

В идеале мы хотели бы подобрать вопросы, ответы на которые дают много информации о том, что наше дерево должно предсказывать. Если есть единый общий вопрос ("да/нет"), для которого ответы "да" всегда соответствуют истинным результатам (`True`), а ответы "нет" — ложным (`False`), или наоборот, то этот вопрос — потрясающий. И с другой стороны, общий вопрос, для которого ни один из ответов не дает вам много новой информации о том, каким должно быть предсказание, вероятно, не является хорошим вариантом.

Мы объясняем идею "количества информации" *энтропией*. Вы, вероятно, слышали, что этот термин используется для обозначения беспорядка, хаоса. Здесь он применяется для обозначения неопределенности, ассоциированной с данными.

Представим, что у нас есть множество S данных, каждый помеченный элемент которого принадлежит одному из конечного числа классов $C_1, ..., C_n$. Если все точки данных принадлежат только одному классу, то неопределенность фактически отсутствует, и значит, мы хотели бы, чтобы там была низкая энтропия. Если же точки данных распределены по классам равномерно, то существует много неопределенности, и значит, мы хотели бы, чтобы там была высокая энтропия.

В математических терминах, если p_i — это пропорция данных, помеченная как класс c_i, то мы определяем энтропию как[3]:

$$H(S) = -p_1 \log_2 p_1 - ... - p_n \log_2 p_n$$

со (стандартным) соглашением, что $0 \cdot \log_2 0 = 0$.

Особо не вдаваясь в жуткие подробности, лишь отметим, что каждый член $-p_i \log_2 p_i$ принимает неотрицательное значение, близкое к нулю, именно тогда, когда параметр p_i близок к нулю или единице (рис. 17.2).

Это означает, что энтропия будет низкой, когда каждый p_i находится близко к 0 или 1 (т. е. когда подавляющая часть данных находится в одном-единственном классе); и напротив, она будет крупнее, когда многочисленные p_i не находятся близко к 0 (т. е. когда данные распределены по многочисленным классам). Это именно то поведение, которое нам требуется.

[3] Речь идет о вероятностной мере неопределенности Шеннона (или информационной энтропии). Данная формула означает, что прирост информации равен утраченной неопределенности. — *Прим. пер.*

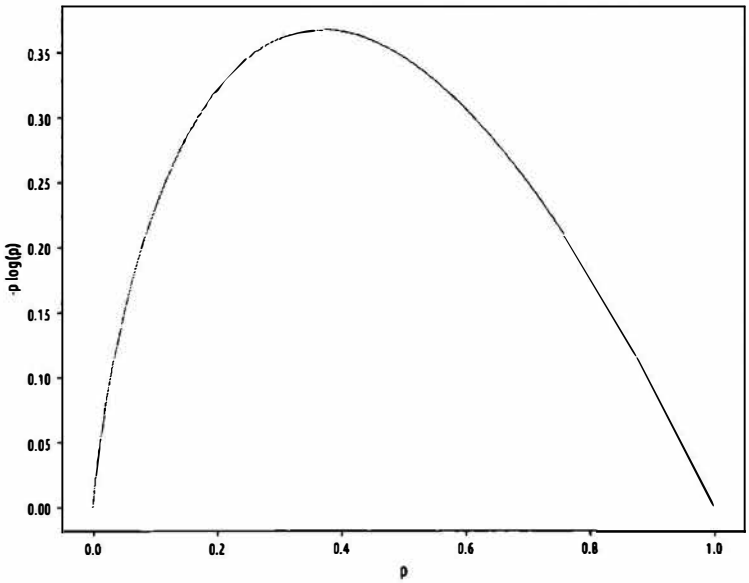


Рис. 17.2. График кривой $-p \log p$

Все это достаточно легко реализуется в функции:

```
from typing import List
import math

def entropy(class_probabilities: List[float]) -> float:
    """С учетом списка классовых вероятностей вычислить энтропию"""
    return sum(-p * math.log(p, 2)
               for p in class_probabilities
               if p > 0) # Игнорировать нулевые вероятности

assert entropy([1.0]) == 0
assert entropy([0.5, 0.5]) == 1
assert 0.81 < entropy([0.25, 0.75]) < 0.82
```

Наши данные будут состоять из пар `(вход, метка)`, а значит, нам придется вычислить классовые вероятности непосредственно. Обратите внимание, что нас на самом деле заботит не то, какая метка ассоциирована с каждой вероятностью, а только то, каковы вероятности:

```
from typing import Any
from collections import Counter

def class_probabilities(labels: List[Any]) -> List[float]:
     total_count = len(labels)
    return [count / total_count
            for count in Counter(labels).values()]
```

```
def data_entropy(labels: List[Any]) -> float:
    return entropy(class_probabilities(labels))

assert data_entropy(['a']) == 0
assert data_entropy([True, False]) == 1
assert data_entropy([3, 4, 4, 4]) == entropy([0.25, 0.75])
```

Энтропия подразделения

Пока мы сделали только то, что вычислили энтропию (читай "неопределенность") одиночного набора помеченных данных. Теперь каждый шаг дерева решений предусматривает постановку вопроса, ответ на который подразделяет данные на одно или (будем надеяться) более подмножеств. Например, вопрос "у него больше пяти ног?" подразделяет животных на тех, у которых их больше пяти (пауки), и остальных (ехидна).

Соответственно, мы хотели бы иметь некую идею энтропии, которая получается в результате подразделения набора данных определенным образом. Мы хотим, чтобы подразделение имело низкую энтропию, если оно разбивает данные на подмножества, которые сами имеют низкую энтропию (т. е. крайне определенные), и высокую энтропию, если оно содержит подмножества, которые (являются крупными и) имеют высокую энтропию (т. е. крайне неопределенные).

Например, мой вопрос об австралийской 5-центовой монете был довольно глупым (но весьма удачным), т. к. он подразделил оставшихся на тот момент животных на подмножества $S_1 = \{\text{ехидна}\}$ и $S_2 = \{\text{все остальные}\}$, где S_2 — большое и с высокой энтропией. (S_1 не имеет энтропии, но оно представляет малую часть оставшихся "классов.")

Математически, если мы подразделим данные S на подмножества $S_1, ..., S_m$, содержащие пропорции $q_1, ..., q_m$ данных, то мы вычислим энтропию подразделения (или ветвления) как взвешенную сумму:

$$H = q_1 H(S_1) + ... + q_m H(S_m),$$

которую мы можем имплементировать как:

```
def partition_entropy(subsets: List[List[Any]]) -> float:
    """Возвращает энтропию из этого подразделения данных на подмножества"""
    total_count = sum(len(subset) for subset in subsets)

    return sum(data_entropy(subset) * len(subset) / total_count
               for subset in subsets)
```

Проблема с этим подходом заключается в том, что подразделение по атрибуту с многочисленными разными значениями приведет к очень низкой энтропии из-за переподгонки. Например, представим, что вы работаете в банке и пытаетесь построить дерево решений для предсказания того, какие клиенты вашего банка, скорее всего, перестанут платить по ипотеке, используя в качестве тренировочного

набора исторические данные. Допустим далее, что набор данных содержит номер социального страхования (НСС) каждого клиента. Подразделение по НСС произведет подмножества из одного человека, и каждое неизбежно будет иметь нулевую энтропию. Однако модель, которая опирается на НСС, *определенно* не сможет обобщать *за пределами* тренировочного набора. По этой причине при создании дерева решений, очевидно, вам следует пытаться избегать (или при необходимости разбивать на интервалы) атрибуты с большим числом возможных значений.

Создание дерева решений

Директор предоставил данные о прошедших собеседование претендентах, которые состоят (согласно вашей спецификации) из типизированного именованного кортежа `NamedTuple` с релевантными атрибутами по каждому претенденту: уровень претендента, предпочитаемый язык программирования, является он активным пользователем соцсети Twitter, имеет ли он ученую степень и прошел ли он собеседование успешно.

```
from typing import NamedTuple, Optional

class Candidate(NamedTuple):
    level: str
    lang: str
    tweets: bool
    phd: bool
    did_well: Optional[bool] = None  # позволить непомеченные данные

                  # level     lang      tweets   phd      did_well
                  # уровень   язык      twitter  степень  прошел
inputs = [Candidate('Senior', 'Java',   False,   False,   False),
          Candidate('Senior', 'Java',   False,   True,    False),
          Candidate('Mid',    'Python', False,   False,   True),
          Candidate('Junior', 'Python', False,   False,   True),
          Candidate('Junior', 'R',      True,    False,   True),
          Candidate('Junior', 'R',      True,    True,    False),
          Candidate('Mid',    'R',      True,    True,    True),
          Candidate('Senior', 'Python', False,   False,   False),
          Candidate('Senior', 'R',      True,    False,   True),
          Candidate('Junior', 'Python', True,    False,   True),
          Candidate('Senior', 'Python', True,    True,    True),
          Candidate('Mid',    'Python', False,   True,    True),
          Candidate('Mid',    'Java',   True,    False,   True),
          Candidate('Junior', 'Python', False,   True,    False)
         ]
```

Наше дерево будет состоять из *решающих узлов* (которые задают вопрос и направляют нас в разные стороны в зависимости от ответа) и *листовых узлов* (которые дают нам предсказание). Дерево будет построено с помощью сравнительно простого *алгоритма ID3*, который работает следующим образом. Допустим, нам даны не-

которые помеченные данные и список атрибутов, которые будут рассматриваться при ветвлении.

1. Если все данные имеют одинаковую метку, то создать листовой узел, который предсказывает эту метку, и затем остановиться.
2. Если список атрибутов пуст (т. е. больше возможных вопросов не осталось), то создать листовой узел, который предсказывает наиболее общую метку, и затем остановиться.
3. В противном случае попытаться подразделить данные по каждому атрибуту.
4. Выбрать подразделение с наименьшей энтропией подразделения (ветвления).
5. Добавить решающий узел на основе выбранного атрибута.
6. Рекурсивно повторять на каждом подразделенном подмножестве, используя оставшиеся атрибуты.

Такой алгоритм называется "жадным", поскольку на каждом шаге он выбирает самым непосредственным образом наилучший вариант. С учетом набора данных может иметься более оптимальное дерево с внешне более худшим первым шагом. Если это так, то указанный алгоритм его не найдет. Тем не менее он сравнительно понятен и легко имплементируется, что делает его хорошей отправной точкой для разведывания деревьев решений.

Давайте пройдем эти шаги "в ручном режиме" на наборе данных о претендентах. Набор данных содержит метки `True` и `False`, и у нас есть четыре атрибута, по которым мы можем выполнять разбиение. Итак, первый шаг будет заключаться в отыскании подразделения с наименьшей энтропией. Мы начнем с написания функции, которая выполняет подразделение:

```
from typing import Dict, TypeVar
from collections import defaultdict

T = TypeVar('T')   # Обобщенный тип для входов

def partition_by(inputs: List[T], attribute: str) -> Dict[Any, List[T]]:
    """Подразделить входы на списки на основе заданного атрибута"""
    partitions: Dict[Any, List[T]] = defaultdict(list)
    for input in inputs:
        key = getattr(input, attribute)  # Значение заданного атрибута
        partitions[key].append(input)    # Добавить вход
                                         # в правильное подразделение
    return partitions
```

и функции, которая использует его для вычисления энтропии:

```
def partition_entropy_by(inputs: List[Any],
                         attribute: str,
                         label_attribute: str) -> float:
    """Вычислить энтропию, соответствующую заданному подразделу"""
    # Подразделы состоят из наших входов
    partitions = partition_by(inputs, attribute)
```

```
    # но энтропия partition_entropy нуждается только в классовых метках
    labels = [[getattr(input, label_attribute) for input in partition]
              for partition in partitions.values()]

    return partition_entropy(labels)
```

Затем нам нужно отыскать подразделение с минимальной энтропией для всего набора данных:

```
for key in ['level','lang','tweets','phd']:
    print(key, partition_entropy_by(inputs, key, 'did_well'))

assert 0.69 < partition_entropy_by(inputs, 'level', 'did_well') < 0.70
assert 0.86 < partition_entropy_by(inputs, 'lang', 'did_well') < 0.87
assert 0.78 < partition_entropy_by(inputs, 'tweets', 'did_well') < 0.79
assert 0.89 < partition_entropy_by(inputs, 'phd', 'did_well') < 0.90
```

Минимальная энтропия получается из разбиения по атрибуту `level` (уровень), поэтому нам нужно создать поддерево для каждого возможного значения уровня `level`. Каждый претендент с уровнем `Mid` (средний) помечен как `True`, а значит, поддерево `Mid` — это всего лишь листовой узел, предсказывающий `True`. У претендентов с уровнем `Senior` (высокий) у нас имеется смесь значений `True` и `False`, поэтому нам необходимо снова выполнить разбиение:

```
senior_inputs = [input for input in inputs if input.level == 'Senior']

assert 0.4 == partition_entropy_by(senior_inputs, 'lang', 'did_well')
assert 0.0 == partition_entropy_by(senior_inputs, 'tweets', 'did_well')
assert 0.95 < partition_entropy_by(senior_inputs, 'phd', 'did_well') < 0.96
```

Этот результат показывает, что наше следующее разбиение должно пройти по атрибуту `tweets`, которое дает разбиение с нулевой энтропией. Для претендентов с уровнем `Senior` атрибут `tweets` со значением "да" всегда дает `True`, а со значением "нет" — всегда `False`.

Наконец, если проделать то же самое для претендентов с уровнем `Junior` (начальный), то в конечном итоге разбиение пройдет по атрибуту `phd` (ученая степень),

Рис. 17.3. Дерево решений для найма сотрудников

после чего мы обнаружим, что отсутствие степени всегда дает `True`, а наличие степени — всегда `False`.

На рис. 17.3 показано завершенное дерево решений.

Собираем все вместе

Теперь, когда мы увидели то, как работает алгоритм, мы хотели бы имплементировать его в более общем плане. Иными словами, нам нужно решить, каким образом представлять деревья. Мы воспользуемся наиболее легким представлением из возможных. Определим *дерево* как одну из следующих альтернатив:

- либо как лист `Leaf` (который предсказывает одно-единственное значение);
- либо как разбиение `Split` (содержащее атрибут, по которому проводится разбиение, поддеревья для конкретных значений этого атрибута и, возможно, значение, принятое по умолчанию, которое будет использоваться, если мы встретим неизвестное значение).

```
from typing import NamedTuple, Union, Any

class Leaf(NamedTuple):
    value: Any

class Split(NamedTuple):
    attribute: str
    subtrees: dict
    default_value: Any = None

DecisionTree = Union[Leaf, Split]
```

При таком представлении наше дерево найма сотрудников будет выглядеть следующим образом:

```
hiring_tree = Split('level', { # Сперва рассмотреть уровень "level".
    'Junior': Split('phd', {   # Если уровень равен "Junior", обратиться
                               # к "phd".
        False: Leaf(True),     # Если "phd" равен False, предсказать True.
        True: Leaf(False)      # Если "phd" равен True, предсказать False.
    }),
    'Mid': Leaf(True),           # Если level равен "Mid", просто
                                 # предсказать True.
    'Senior': Split('tweets', {  # Если level равен "Senior", обратиться
                                 # к "tweets".
        False: Leaf(False),      # Если "tweets" равен False, предсказать
                                 # False.
        True: Leaf(True)         # Если "tweets" равен True, предсказать
                                 # True.
    })
})
```

Остался вопрос: что делать, если мы встретим неожиданное (или пропущенное) значение атрибута? Что должно делать наше дерево найма сотрудников, если оно встретит претендента, чей атрибут `level` равен "Intern" (стажер)? Мы обрабатываем этот случай, заполняя атрибут `default_value` наиболее общей меткой.

С учетом такого представления мы можем расклассифицировать вход с помощью:

```
def classify(tree: DecisionTree, input: Any) -> Any:
    """Классифицировать вход, используя заданное дерево решений"""
    # Если это листовой узел, то вернуть его значение
    if isinstance(tree, Leaf):
        return tree.value

    # В противном случае это дерево состоит из атрибута,
    # по которому проводится разбиение,
    # и словаря, ключи которого являются значениями этого атрибута
    # и значения которого являются поддеревьями, рассматриваемыми далее
    subtree_key = getattr(input, tree.attribute)

    if subtree_key not in tree.subtrees:  # Если для ключа нет поддерева,
        return tree.default_value         # то вернуть значение
                                          # по умолчанию

    subtree = tree.subtrees[subtree_key]  # Выбрать соответствующее
                                          # поддерево
    return classify(subtree, input)       # и использовать
                                          # для классификации входа
```

Осталось только построить древесное представление из тренировочных данных:

```
def build_tree_id3(inputs: List[Any],
                   split_attributes: List[str],
                   target_attribute: str) -> DecisionTree:

    # Подсчитать целевые метки
    label_counts = Counter(getattr(input, target_attribute)
                           for input in inputs)
    most_common_label = label_counts.most_common(1)[0][0]

    # Если имеется уникальная метка, то предсказать ее
    if len(label_counts) == 1:
        return Leaf(most_common_label)

    # Если больше не осталось атрибутов, по которым проводить
    # разбиение, то вернуть наиболее распространенную метку
    if not split_attributes:
        return Leaf(most_common_label)
```

```
    # В противном случае разбить по наилучшему атрибуту
    def split_entropy(attribute: str) -> float:
        """Вспомогательная функция для отыскания наилучшего атрибута"""
        return partition_entropy_by(inputs, attribute, target_attribute)

    best_attribute = min(split_attributes, key=split_entropy)

    partitions = partition_by(inputs, best_attribute)
    new_attributes = [a for a in split_attributes if a != best_attribute]

    # Рекурсивно строить поддеревья
    subtrees = {attribute_value : build_tree_id3(subset,
                                                 new_attributes,
                                                 target_attribute)
                for attribute_value, subset in partitions.items()}

    return Split(best_attribute, subtrees,
                 default_value=most_common_label)
```

В дереве, которое мы построили, каждый лист состоял целиком из входов `True` или входов `False`. Следовательно, дерево прекрасно предсказывает на тренировочном наборе данных. Но мы также можем его применить к новым данным, которые отсутствовали в тренировочном наборе:

```
tree = build_tree_id3(inputs,
                      ['level', 'lang', 'tweets', 'phd'],
                       'did_well')

# Должно предсказать True
assert classify(tree, Candidate("Junior", "Java", True, False))

# Должно предсказать False
assert not classify(tree, Candidate("Junior", "Java", True, True))
```

А также к данным с пропущенными или неожиданными значениями:

```
# Должно предсказать True
assert classify(tree, Candidate("Intern", "Java", True, True))
```

Поскольку наша задача в основном заключается в том, чтобы продемонстрировать процесс построения дерева, мы строим дерево, используя весь набор данных целиком. Как всегда, если бы стояла задача создать реально хорошую модель, то мы бы собрали больше данных и разбили бы их на тренировочное, контрольное и проверочное подмножества.

Случайные леса

Поскольку деревья решений могут вписывать себя в тренировочные данные, неудивительно, что они имеют тягу к переподгонке. Один из способов, который позволяет избежать этого, — техническое решение, именуемое *случайными лесами*,

в котором мы строим многочисленные деревья решений и совмещаем их результаты. Если эти деревья являются классификационными, то мы даем им возможность проголосовать; если же эти деревья являются регрессионными, то мы усредняем их предсказания.

Наш процесс построения дерева был детерминированным, поэтому возникает вопрос: как получить случайные леса?

Одна часть предусматривает бутстрапирование данных (вспомните *разд. "Отступление: размножение выборок" главы 15*). Вместо того чтобы тренировать каждое дерево на всех входах `inputs` в тренировочном наборе, мы тренируем каждое дерево на результате функции `bootstrap_sample(inputs)`. Поскольку каждое дерево строится на разных данных, эти деревья будут отличаться друг от друга. (Дополнительное преимущество заключается в том, что для проверки каждого дерева совершенно справедливо можно использовать невыборочные данные, а значит, в качестве тренировочного набора можно смело использовать все данные целиком, если умно подойти к измерению результативности.) Это техническое решение называется *агрегированием бутстраповских выборок* или *бэггингом*[4].

Второй источник случайности предусматривает изменение способа, которым мы выбираем наилучший атрибут `best_attribute`, и по этому атрибуту проводится разбиение. Вместо исчерпывающего просмотра всех оставшихся атрибутов мы сначала выбираем случайное их подмножество, а затем разбиваем по любому атрибуту, который является наилучшим:

```
# Если уже осталось мало кандидатов на разбиение, то обратиться ко всем
if len(split_candidates) <= self.num_split_candidates:
    sampled_split_candidates = split_candidates
# В противном случае подобрать случайный образец
else:
    sampled_split_candidates = random.sample(split_candidates,
                                             self.num_split_candidates)

# Теперь выбрать наилучший атрибут только из этих кандидатов
best_attribute = min(sampled_split_candidates, key=split_entropy)
partitions = partition_by(inputs, best_attribute)
```

Этот пример демонстрирует более широкое техническое решение, именуемое *ансамблевым обучением*, в котором мы объединяем несколько слабых учеников (как правило, это модели с высоким смещением и низкой дисперсией) с целью произвести совокупную сильную модель.

---

[4] Бутстрап-агрегирование или бэггинг — метод формирования ансамблей классификаторов с использованием случайной выборки с возвратом. При формировании выборок из исходного набора данных случайным образом отбирается несколько подмножеств такого же размера, что и исходный набор. Затем на основе каждой строится классификатор, и их выходы комбинируются путем голосования или простого усреднения. — *Прим. пер.*

Для дальнейшего изучения

- Библиотека scikit-learn имеет многочисленные модели деревьев решений[5]. Помимо этого, в ней есть ансамблевый модуль, который содержит классификатор `RandomForestClassifier`[6], а также другие ансамблевые методы.
- Библиотека XGBoost[7] служит для тренировки градиентно-бустированных деревьев решений, которые, как правило, побеждают на соревнованиях по машинному обучению в стиле Kaggle.
- В этой главе мы лишь слегка прикоснулись к теме деревьев решений и их алгоритмов. Википедия является хорошей отправной точкой для более широкого поиска информации[8].

[5] См. **https://scikit-learn.org/stable/modules/tree.html**.

[6] См. **https://scikit-learn.org/stable/modules/classes.html%23module-sklearn.ensemble**.

[7] См. **https://xgboost.ai/**.

[8] См. **https://en.wikipedia.org/wiki/Decision_tree_learning**.

ГЛАВА 18
Нейронные сети

Люблю бессмыслицы, они пробуждают мозговые клетки.
– Доктор Сьюз[1]

Искусственная нейронная сеть, или просто нейросеть, — это предсказательная модель, основанием для разработки которой послужил способ организации и принцип функционирования головного мозга. Думайте о мозге, как о коллекции нейронов, которые соединены между собой. Каждый нейрон смотрит на выходы других нейронов, которые входят в него, выполняет вычисление и затем либо активируется (если результат вычисления превышает некий порог), либо нет (если не превышает).

Соответственно, искусственные нейронные сети состоят из искусственных нейронов, которые производят аналогичные вычисления над своими входами. Нейронные сети могут решать широкий круг задач, среди которых такие как распознавание рукописного текста и распознавание лиц, и они интенсивно применяются в одной из самых сверхсовременных отраслей науки о данных — глубоком обучении. Однако большинство нейронных сетей представляют собой "черные ящики" — изучение подробностей их устройства не особо прибавит понимания того, *каким образом* они решают задачу. К тому же большие нейронные сети могут быть труднотренируемыми. Для большинства задач, с которыми вы будете сталкиваться в качестве начинающего исследователя данных, они, вероятно, вряд ли подойдут. Однако в один прекрасный день, когда вы попытаетесь построить искусственный интеллект, чтобы осуществить компьютерную сингулярность[2], они окажутся как нельзя кстати.

Персептроны

Абсолютно простейшая модель нейронной сети — это *персептрон*, который аппроксимирует один-единственный нейрон с n бинарными входами. Он вычисляет взвешенную сумму своих входов и "активизируется", если эта сумма больше или равна нулю:

[1] Теодор Сьюз Гейзель (1904–1991) — американский детский писатель и мультипликатор. Автор таких забавных персонажей, как Гринч, Лоракс и Хортон. — *Прим. пер.*

[2] Компьютерная сингулярность — точка во времени, с которой машины начинают совершенствовать сами себя без чьей-либо помощи. — *Прим. пер.*

```
from scratch.linear_algebra import Vector, dot

# Ступенчатая функция
def step_function(x):
    return 1 if x >= 0 else 0

def perceptron_output(weights: Vector, bias: float, x: Vector) -> float:
    """Возвращает 1, если персептрон 'активируется', и 0, если нет"""
    calculation = dot(weights, x) + bias
    return step_function(calculation)
```

Персептрон попросту проводит различие между полупространствами, разделенными гиперплоскостью точек x, для которых:

```
dot(weights,x) + bias == 0  # Скалярное произведение весов
                            # и точек плюс смещение
```

При правильно подобранных весах персептроны могут решать ряд простых задач (рис. 18.1). Например, можно создать *логический вентиль И* (который возвращает 1, если оба входа равны 1, и 0, если один из входов равен 0):

```
and_weights = [2., 2]
and_bias = -3.

assert perceptron_output(and_weights, and_bias, [1, 1]) == 1
assert perceptron_output(and_weights, and_bias, [0, 1]) == 0
assert perceptron_output(and_weights, and_bias, [1, 0]) == 0
assert perceptron_output(and_weights, and_bias, [0, 0]) == 0
```

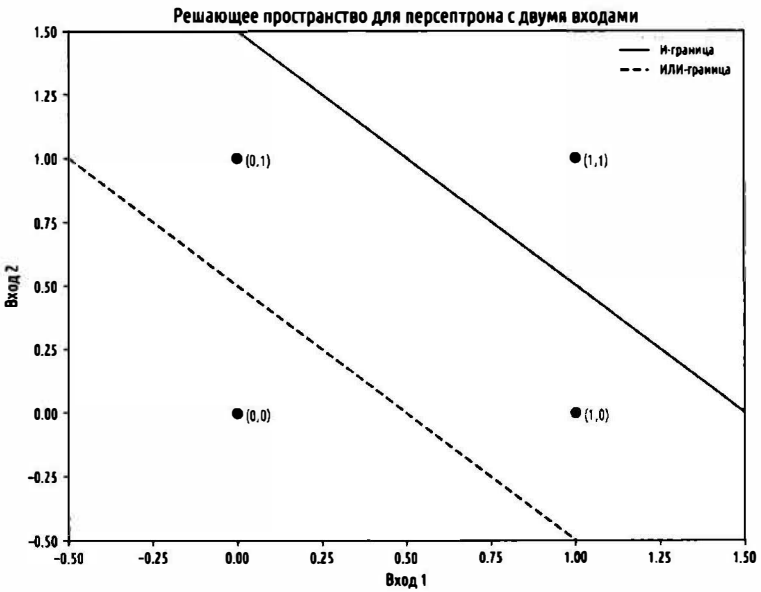


Рис. 18.1. Решающее пространство для персептрона с двумя входами

Если оба входа равны 1, то получим $2 + 2 - 3 = 1$, и выход равен 1. Если только один из входов равен 1, то получим $2 + 0 - 3 = -1$, и выход равен 0. И если оба входа равны 0, то получим -3, и выход равен 0.

Используя схожее рассуждение, мы могли бы построить *логический вентиль ИЛИ*:

```
or_weights = [2., 2]
or_bias = -1.

assert perceptron_output(or_weights, or_bias, [1, 1]) == 1
assert perceptron_output(or_weights, or_bias, [0, 1]) == 1
assert perceptron_output(or_weights, or_bias, [1, 0]) == 1
assert perceptron_output(or_weights, or_bias, [0, 0]) == 0
```

Мы также могли бы создать *логический вентиль НЕ* (который имеет всего один вход и преобразует 1 в 0, и наоборот) с помощью:

```
not_weights = [-2.]
not_bias = 1.

assert perceptron_output(not_weights, not_bias, [0]) == 1
assert perceptron_output(not_weights, not_bias, [1]) == 0
```

Однако некоторые задачи персептрон решить просто не способен. Например, как ни пытаться, но при помощи персептрона у вас не получится создать *логический вентиль "исключающее ИЛИ"* (XOR), который на выход выдает 1, если только один из его входов равен 1, и 0 — в противном случае. Отсюда возникает потребность в более сложных нейронных сетях.

Естественно, для того чтобы создать логический вентиль, вовсе не нужно аппроксимировать нейрон:

```
and_gate = min
or_gate = max
xor_gate = lambda x, y: 0 if x == y else 1
```

Как и настоящие нейроны, искусственные нейроны вызывают интерес, когда вы начинаете их соединять между собой.

Нейронные сети прямого распространения

Человеческий мозг имеет чрезвычайно сложную топологию, и поэтому принято аппроксимировать его с помощью идеализированной нейронной сети *прямого распространения*, состоящей из дискретных *слоев* нейронов, где каждый слой соединен со следующим. В такой сети, как правило, предусмотрен входной слой (который получает входы и передает их дальше без изменений), один или несколько "скрытых слоев" (каждый из которых состоит из нейронов, принимающих выходы предыдущего слоя, выполняет некое вычисление и передает результат в следующий слой) и выходной слой (который производит окончательные выходы).

Так же как и в персептроне, каждый (не входной) нейрон имеет вес, соответствующий каждому входу, и смещение. Упрощая наше представление, мы добавим

смещение в конец нашего вектора весов и дадим каждому нейрону *вход смещения*, всегда равный 1.

Как и у персептрона, по каждому нейрону мы будем суммировать произведения его входов и весов. Однако здесь вместо выдачи пороговой функции `step_function`, применяемой к этому произведению, мы будем выдавать ее гладкую аппроксимацию. В частности, мы воспользуемся сигмоидальной функцией `sigmoid` (рис. 18.2):

```
import math

def sigmoid(t: float) -> float:     # Сигмоида
    return 1 / (1 + math.exp(-t))
```

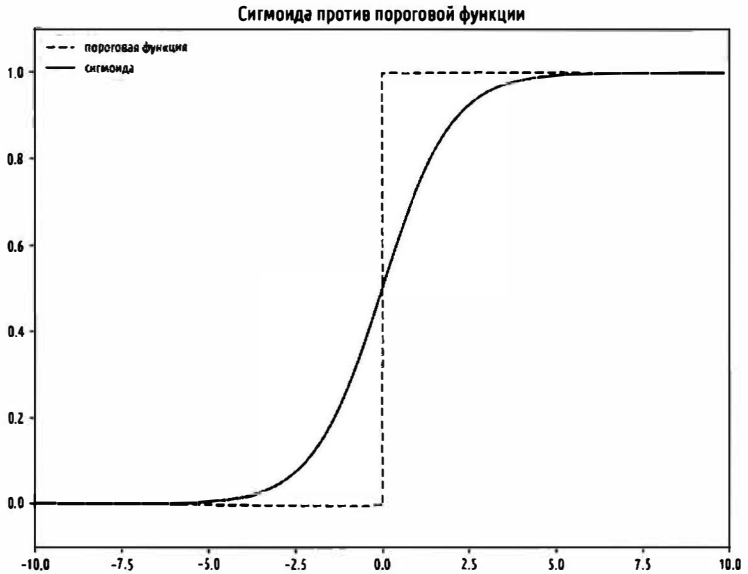


Рис. 18.2. Сигмоидальная и пороговая функции в сравнении

Зачем применять сигмоидальную функцию `sigmoid` вместо более простой пороговой `step_function`? Для тренировки нейронной сети мы используем дифференциальное исчисление, а для того чтобы использовать дифференциальное исчисление, нам требуются *гладкие* функции. Пороговая функция даже не является непрерывной, а сигмоида представляет собой ее хорошую гладкую аппроксимацию.

Вы, наверное, обратили внимание на функцию `sigmoid` в *главе 16*, где она носит название логистической `logistic`. Технически термин "сигмоида" относится к *форме* функции, а термин "логистический" — к этой конкретной функции, хотя часто эти термины используют как взаимозаменяемые.

Затем мы вычисляем выход следующим образом:

```
def neuron_output(weights: Vector, inputs: Vector) -> float:
    # weights включает член смещения, inputs включает единицу
    return sigmoid(dot(weights, inputs))
```

С учетом этой функции мы можем представить нейрон просто как вектор весов, длина которого на единицу больше числа входов в этот нейрон (из-за веса смещения). Тогда мы можем представить нейронную сеть как список (не входных) *слоев*, где каждый слой — это просто список нейронов в этом слое.

Иными словами, мы представим нейронную сеть как список (слоев) списков (нейронов) векторов (весов).

С учетом такого преставления использовать нейронную сеть достаточно просто:

```
from typing import List

def feed_forward(neural_network: List[List[Vector]],
                 input_vector: Vector) -> List[Vector]:
    """Пропускает входной вектор через нейронную сеть.
       Возвращает все слои (а не только последний)."""
    outputs: List[Vector] = []

    for layer in neural_network:
        input_with_bias = input_vector + [1] # Add a constant.
        output = [neuron_output(neuron, input_with_bias)  # Вычислить выход
                  for neuron in layer]                    # для каждого нейрона.
        outputs.append(output)                            # Сложить результаты

        # Затем вход в следующий слой является выходом этого слоя
        input_vector = output

    return outputs
```

Теперь можно легко построить логический вентиль исключающего ИЛИ (XOR), который нам не удалось реализовать с помощью персептрона. Нам нужно всего лишь прошкалировать веса так, чтобы выходы нейрона `neuron_outputs` находились очень близко к 0 либо к 1:

```
or_network = [# Скрытый слой
              [[20., 20, -30],   # Нейрон 'И'
               [20., 20, -10]],  # Нейрон 'ИЛИ'
              # Выходной слой
              [[-60., 60, -30]]] # Нейрон '2-й вход, но не 1-й вход'

# Функция feed_forward возвращает выходы всех слоев, поэтому
# [-1] получает окончательный выход и
# [0] получает значение из результирующего вектора
assert 0.000 < feed_forward(xor_network, [0, 0])[-1][0] < 0.001
assert 0.999 < feed_forward(xor_network, [1, 0])[-1][0] < 1.000
assert 0.999 < feed_forward(xor_network, [0, 1])[-1][0] < 1.000
assert 0.000 < feed_forward(xor_network, [1, 1])[-1][0] < 0.001
```

Для заданного входа (т. е. двумерного вектора) скрытый слой производит двумерный вектор, состоящий из "И" двух входных значений и "ИЛИ" двух входных значений.

Затем выходной слой принимает двумерный вектор и вычисляет "второй элемент, но не первый элемент". В результате получается сеть, которая выполняет операцию "или, но не и", т. е. в точности операцию "исключающее ИЛИ" (XOR) (рис. 18.3).

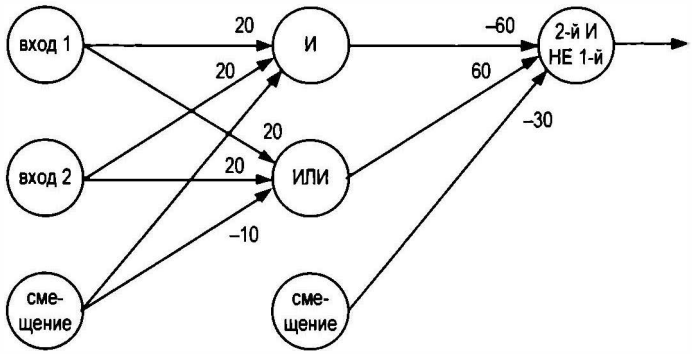


Рис. 18.3. Нейронная сеть для логического вентиля "исключающее ИЛИ"

Интерпретировать эту сеть можно так: скрытый слой вычисляет признаки входных данных (в этом случае "И" и "ИЛИ"), а выходной слой объединяет эти признаки таким образом, который генерирует желаемый результат.

Обратное распространение

Обычно нейронные сети вручную не строят. Отчасти потому, что их используют для решения гораздо более масштабных задач; например, задача распознавания изображений может предусматривать сотни или тысячи нейронов. А также потому, что у нас в общем случае не получится найти представление того, какими должны быть нейроны.

Вместо этого для тренировки нейронных сетей (обычно) используют данные. Типичным подохом является алгоритм обратного распространения, в котором применяется градиентный спуск либо один из его вариантов.

Представим, что у нас есть тренировочный набор, который состоит из векторов входов, и имеются соответствующие векторы целевых выходов. Например, в предыдущем примере нейросети XOR, `xor_network`, вектор входов `[1, 0]` соответствовал целевому выходу `[1]`. Допустим также, что наша нейросеть имеет некий набор весов. Тогда мы корректируем веса, используя следующий алгоритм:

1. Выполнить прямое прохождение по сети, применив функцию `feed_forward` на векторе входов, произведя выходы всех нейронов сети.
2. Мы знаем целевой выход, поэтому можем вычислить потерю, т. е. сумму квадратов ошибок.
3. Вычислить градиент этой ошибки как функцию весов выходного нейрона.
4. Распространить градиенты и ошибки назад по сети, вычислив градиенты по отношению к весам скрытых нейронов.
5. Сделать шаг градиентного спуска.

Как правило, этот алгоритм выполняют многократно для всего тренировочного набора до тех пор, пока сеть не сойдется.

Для начала давайте напишем функцию, которая вычисляет градиенты:

```
def sqerror_gradients(network: List[List[Vector]],
                      input_vector: Vector,
                      target_vector: Vector) -> List[List[Vector]]:
    """С учетом нейронной сети, вектора входов и вектора целей
       сделать предсказание и вычислить градиент потери, т. е.
       сумму квадратов ошибок по отношению к весам нейрона."""
    # Прямое прохождение
    hidden_outputs, outputs = feed_forward(network, input_vector)

    # Градиенты по отношению к преактивационным выходам выходного нейрона
    output_deltas = [output * (1 - output) * (output - target)
                     for output, target in zip(outputs, target_vector)]

    # Градиенты по отношению к весам выходного нейрона
    output_grads = [[output_deltas[i] * hidden_output
                     for hidden_output in hidden_outputs + [1]]
                    for i, output_neuron in enumerate(network[-1])]

    # Градиенты по отношению к преактивационным выходам скрытого нейрона
    hidden_deltas = [hidden_output * (1 - hidden_output) *
                         dot(output_deltas, [n[i] for n in network[-1]])
                     for i, hidden_output in enumerate(hidden_outputs)]

    # Градиенты по отношению к весам скрытого нейрона
    hidden_grads = [[hidden_deltas[i] *
                     input for input in input_vector + [1]]
                    for i, hidden_neuron in enumerate(network[0])]

    return [hidden_grads, output_grads]
```

Математика в основе приведенных выше расчетов не является какой-то невероятно сложной, но она сопряжена с утомительным исчислением и тщательным вниманием к деталям, поэтому я оставлю ее в качестве упражнения для вас.

Вооружившись способностью вычислять градиенты, мы теперь можем натренировать нейронную сеть. Давайте попробуем применить сеть XOR, которую мы ранее сконструировали вручную, для того, чтобы она усвоила эту операцию.

Мы начнем с генерирования тренировочных данных и инициализации нашей нейронной сети случайными весами.

```
import random
random.seed(0)

# Тренировочные данные
xs = [[0., 0], [0., 1], [1., 0], [1., 1]]
ys = [[0.], [1.], [1.], [0.]]
```

```
# Начать со случайных весов
network = [ # Скрытый слой: 2 входа -> 2 выхода
            [[random.random() for _ in range(2 + 1)], # 1-й скрытый нейрон
             [random.random() for _ in range(2 + 1)]],# 2-й скрытый нейрон
            # Выходной слой: 2 входа -> 1 выход
            [[random.random() for _ in range(2 + 1)]] # 1-й вых. нейрон
          ]
```

Как обычно, мы можем натренировать ее с помощью градиентного спуска. Одним из отличий от наших предыдущих примеров является то, что здесь у нас есть несколько векторов параметров, каждый со своим градиентом, а значит, нам придется вызвать функцию `gradient_step` для каждого из них.

```
from scratch.gradient_descent import gradient_step
import tqdm

learning_rate = 1.0       # Темп усвоения

for epoch in tqdm.trange(20000, desc="neural net for xor"):
    for x, y in zip(xs, ys):
        gradients = sqerror_gradients(network, x, y)

        # Сделать градиентный шаг для каждого нейрона в каждом слое
        network = [[gradient_step(neuron, grad, -learning_rate)
                    for neuron, grad in zip(layer, layer_grad)]
                   for layer, layer_grad in zip(network, gradients)]

# Проверить, что сеть усвоила операцию XOR
assert feed_forward(network, [0, 0])[-1][0] < 0.01
assert feed_forward(network, [0, 1])[-1][0] > 0.99
assert feed_forward(network, [1, 0])[-1][0] > 0.99
assert feed_forward(network, [1, 1])[-1][0] < 0.01
```

У меня результирующая сеть имеет веса, которые выглядят следующим образом:

```
[ # Скрытый слой
  [[7, 7, -3],     # Вычисляет ИЛИ
   [5, 5, -8]],    # Вычисляет И
  # Выходной слой
  [[11, -12, -5]]  # Вычисляет "первый, но не второй"
]
```

что концептуально очень похоже на нашу предыдущую сеть, "сшитую" на заказ.

Пример: задача Fizz Buzz

Директор по технологиям хочет проинтервьюировать технических кандидатов, предложив им решить следующую ниже хорошо поставленную задачу программирования под названием "Fizz Buzz"[3]:

[3] См. решение задачи Fizz Buzz на Хабре: **https://habr.com/ru/post/278867/**. — *Прим. пер.*

напечатать числа от 1 до 100, за исключением того, что если число делится на 3, то напечатать "fizz"; если число делится на 5, то напечатать "buzz"; и если число делится на 15, то напечатать "fizzbuzz".

Он считает, что умение решить эту задачу демонстрирует исключительные навыки программирования. Вы думаете, что эта задача является настолько простой, что нейронная сеть могла бы ее решить?

Нейронные сети принимают векторы на входе и производят векторы на выходе. Как уже говорилось, поставленная задача программирования состоит в том, чтобы превратить целое число в строку. Поэтому первая сложность состоит в том, чтобы придумать способ переформулировать задачу как векторную.

Для выходов это не сложно: в сущности есть четыре класса выходов, поэтому мы можем закодировать выход как вектор из четырех нулей и единиц:

```
def fizz_buzz_encode(x: int) -> Vector:
    if x % 15 == 0:
        return [0, 0, 0, 1]
    elif x % 5 == 0:
        return [0, 0, 1, 0]
    elif x % 3 == 0:
        return [0, 1, 0, 0]
    else:
        return [1, 0, 0, 0]

assert fizz_buzz_encode(2) == [1, 0, 0, 0]
assert fizz_buzz_encode(6) == [0, 1, 0, 0]
assert fizz_buzz_encode(10) == [0, 0, 1, 0]
assert fizz_buzz_encode(30) == [0, 0, 0, 1]
```

Мы будем использовать эту функцию для генерирования векторов целей. Векторы входов менее очевидны. По нескольким причинам вы не хотите использовать просто одномерный вектор, содержащий входное число. Одиночный вход улавливает "интенсивность", но тот факт, что 2 в два раза больше 1, а 4 снова в два раза больше, не имеет отношения к этой задаче. В дополнение к этому при наличии только одного входа скрытый слой не сможет вычислить очень интересные признаки, а значит, он, вероятно, не сможет решить задачу.

Оказывается, имеется кое-что, работающее достаточно хорошо, и это конвертирование каждого числа в его двоичное представление из единиц и нулей. (Не волнуйтесь, если это не очевидно — по крайней мере, для меня не было.)

```
def binary_encode(x: int) -> Vector:
    binary: List[float] = []

    for i in range(10):
        binary.append(x % 2)
        x = x // 2
    return binary
```

```
#                              1  2  4  8 16 32 64 128 256 512
assert binary_encode(0)    == [0, 0, 0, 0, 0, 0, 0,  0,  0,  0]
assert binary_encode(1)    == [1, 0, 0, 0, 0, 0, 0,  0,  0,  0]
assert binary_encode(10)   == [0, 1, 0, 1, 0, 0, 0,  0,  0,  0]
assert binary_encode(101)  == [1, 0, 1, 0, 0, 1, 1,  0,  0,  0]
assert binary_encode(999)  == [1, 1, 1, 0, 0, 1, 1,  1,  1,  1]
```

Поскольку цель состоит в том, чтобы сконструировать выходы для чисел от 1 до 100, было бы обманом тренироваться на этих числах. Поэтому мы будем тренироваться на числах от 101 до 1023 (это самое большое число, которое мы можем представить с помощью 10 двоичных цифр):

```
xs = [binary_encode(n) for n in range(101, 1024)]
ys = [fizz_buzz_encode(n) for n in range(101, 1024)]
```

Далее давайте создадим нейронную сеть со случайными начальными весами. Она будет иметь 10 входных нейронов (поскольку мы представляем наши входные данные как 10-мерные векторы) и 4 выходных нейрона (поскольку мы представляем наши цели как 4-мерные векторы). Мы дадим ему 25 скрытых элементов (блоков), но для этого мы будем использовать переменную, поэтому их число легко изменить:

```
NUM_HIDDEN = 25

network = [
    # Скрытый слой: 10 входов -> NUM_HIDDEN выходов
    [[random.random() for _ in range(10 + 1)] for _ in range(NUM_HIDDEN)],

    # Выходной слой: NUM_HIDDEN входов -> 4 выхода
    [[random.random() for _ in range(NUM_HIDDEN + 1)] for _ in range(4)]
]
```

Вот и все. Теперь мы готовы тренироваться. Поскольку эта задача сложнее (и есть еще много вещей, которые могут все испортить), мы хотели бы внимательно следить за тренировочным процессом. В частности, для каждой эпохи мы будем отслеживать сумму квадратов ошибок и печатать их. Мы хотим быть уверенными, что они уменьшаются:

```
from scratch.linear_algebra import squared_distance

learning_rate = 1.0        # Темп усвоения

with tqdm.trange(500) as t:
    for epoch in t:
        epoch_loss = 0.0

        for x, y in zip(xs, ys):
            predicted = feed_forward(network, x)[-1]
            epoch_loss += squared_distance(predicted, y)
            gradients = sqerror_gradients(network, x, y)
```

```
            # Сделать градиентный шаг для каждого нейрона в каждом слое
            network = [[gradient_step(neuron, grad, -learning_rate)
                        for neuron, grad in zip(layer, layer_grad)]
                       for layer, layer_grad in zip(network, gradients)]

        t.set_description(f"fizz buzz (потеря: {epoch_loss:.2f})")
```

Тренировка займет некоторое время, но в конечном итоге потеря должна начать опускаться.

Наконец-то мы готовы решить нашу исходную задачу. У нас остался один вопрос. Наша сеть будет производить четырехмерный вектор чисел, но мы хотим иметь одно-единственное предсказание. Мы сделаем это, взяв `argmax`, т. е. индекс наибольшего значения:

```
def argmax(xs: list) -> int:
    """Возвращает индекс наибольшего значения"""
    return max(range(len(xs)), key=lambda i: xs[i])

assert argmax([0, -1]) == 0               # items[0] является наибольшим
assert argmax([-1, 0]) == 1               # items[1] является наибольшим
assert argmax([-1, 10, 5, 20, -3]) == 3   # items[3] является наибольшим
```

Теперь мы можем, наконец, решить задачу "Fizz Buzz":

```
num_correct = 0

for n in range(1, 101):
    x = binary_encode(n)
    predicted = argmax(feed_forward(network, x)[-1])
    actual = argmax(fizz_buzz_encode(n))
    labels = [str(n), "fizz", "buzz", "fizzbuzz"]
    print(n, labels[predicted], labels[actual])

    if predicted == actual:
        num_correct += 1

print(num_correct, "/", 100)
```

У меня натренированная сеть отвечает правильно в 96 случаях из 100, что намного выше порога приема на работу, который установил директор по технологиям. Столкнувшись с доказательствами, он смягчается и меняет задачу интервью на "Инвертировать бинарное дерево".

Для дальнейшего изучения

- Продолжайте читать: в *главе 19* эти темы обсуждаются в гораздо больших подробностях.
- Читайте довольно хороший пост в моем блоге под названием "Задача Fizz Buzz в Tensorflow"[4].

[4] См. **http://joelgrus.com/2016/05/23/fizz-buzz-in-tensorflow/**.

ГЛАВА 19
Глубокое обучение

> Малая образованность — опасная вещь; пей сполна, иль не попробуешь источник вдохновенья.
> – *Александр Поуп*[1]

Глубокое обучение первоначально было связано с применением "глубоких" нейронных сетей (т. е. сетей с более чем одним скрытым слоем), однако на практике этот термин теперь охватывает широкий спектр нейронных архитектур (включая "простые" нейронные сети, которые мы разработали в *главе 18*).

В этой главе мы продолжим наши изыскания и рассмотрим более широкий спектр нейронных сетей. Для этого мы представим ряд абстракций, которые позволяют нам думать о нейронных сетях в более общем плане.

Тензор

Ранее мы проводили различие между векторами (одномерными массивами) и матрицами (двумерными массивами). Когда мы начнем работать с более сложными нейронными сетями, нам также придется использовать высокоразмерные массивы.

Во многих библиотеках нейронных сетей *n*-размерные массивы называются *тензорами*. Именно так мы и будем их называть. (Есть педантичные математические причины не называть *n*-размерные массивы тензорами; если вы один из таких педантов, то ваше возражение отмечено.)

Если бы я писал целую книгу о глубоком обучении, я бы имплементировал полнофункциональный тензорный класс, который перегружал бы арифметические операторы Python и мог бы обрабатывать целый ряд других операций. Такая имплементация сама по себе заняла бы целую главу. Здесь мы обманем и скажем, что тензор — это просто список. Это верно в одном направлении — все наши векторы и матрицы и более выскоразомерные аналоги являются списками. Это, конечно, неверно в другом направлении — большинство списков Python не являются *n*-размерными массивами в нашем смысле.

В идеале вы бы хотели сделать что-то вроде:

```
# Тензор является либо вещественным, либо списком тензоров
Tensor = Union[float, List[Tensor]]
```

[1] Александр Поуп (1688–1744) — английский поэт XVIII в., один из крупнейших авторов британского классицизма. — *Прим. пер.*

Однако Python не позволит вам определять рекурсивные типы, подобные этому. И даже если бы это было так, то это определение все равно является неправильным, поскольку оно допускает плохие "тензоры", такие как:

```
[[1.0, 2.0],
 [3.0]]
```

чьи строки имеют разные размеры, что делает тензор не *n*-размерным массивом.

Поэтому, как я сказал, мы просто обманем:

```
Tensor = list
```

И напишем вспомогательную функцию, отыскивающую форму тензора:

```
from typing import List

def shape(tensor: Tensor) -> List[int]:
    sizes: List[int] = []
    while isinstance(tensor, list):
        sizes.append(len(tensor))
        tensor = tensor[0]
    return sizes

assert shape([1, 2, 3]) == [3]
assert shape([[1, 2], [3, 4], [5, 6]]) == [3, 2]
```

Поскольку тензоры могут иметь любое число размерностей, нам обычно придется работать с ними рекурсивно. Мы будем выполнять следующие действия один раз в одномерном случае и рекурсивно в многомерном случае:

```
def is_1d(tensor: Tensor) -> bool:
    """
    Если tensor[0] является списком, то это тензор более высокого порядка.
    В противном случае tensor является одномерным (т. е. вектором).
    """
    return not isinstance(tensor[0], list)

assert is_1d([1, 2, 3])
assert not is_1d([[1, 2], [3, 4]])
```

И теперь эту функцию можно использовать для написания рекурсивной функции `tensor_sum`:

```
def tensor_sum(tensor: Tensor) -> float:
    """Суммирует все значения в тензоре"""
    if is_1d(tensor):
        return sum(tensor)  # Просто список вещественных,
                            # поэтому функция sum
    else:
        return sum(tensor_sum(tensor_i)      # Вызвать tensor_sum
                                             # на каждом ряду
```

```
                    for tensor_i in tensor)  # и просуммировать
                                             # эти результаты
assert tensor_sum([1, 2, 3]) == 6
assert tensor_sum([[1, 2], [3, 4]]) == 10
```

Если вы не привыкли мыслить рекурсивно, то вам следует обдумать эту функцию, пока она не обретет смысл, потому что мы будем использовать ту же логику на протяжении всей этой главы. Однако мы создадим пару вспомогательных функций для того, чтобы нам не пришлось переписывать эту логику повсюду. Первая применяет функцию поэлементно к одиночному тензору:

```
from typing import Callable

def tensor_apply(f: Callable[[float], float], tensor: Tensor) -> Tensor:
    """Применяет f поэлементно"""
    if is_1d(tensor):
        return [f(x) for x in tensor]
    else:
        return [tensor_apply(f, tensor_i) for tensor_i in tensor]

assert tensor_apply(lambda x: x + 1, [1, 2, 3]) == [2, 3, 4]
assert tensor_apply(lambda x: 2 * x, [[1, 2], [3, 4]]) == [[2, 4], [6, 8]]
```

Мы можем использовать ее для написания функции, которая создает нулевой тензор с той же формой, что и заданный тензор:

```
def zeros_like(tensor: Tensor) -> Tensor:
    return tensor_apply(lambda _: 0.0, tensor)

assert zeros_like([1, 2, 3]) == [0, 0, 0]
assert zeros_like([[1, 2], [3, 4]]) == [[0, 0], [0, 0]]
```

Нам также нужно будет применить функцию к соответствующим элементам из двух тензоров (которые должны быть точно такой же формы, хотя мы не будем проверять это):

```
def tensor_combine(f: Callable[[float, float], float],
                   t1: Tensor,
                   t2: Tensor) -> Tensor:
    """Применяет f к соответствующим элементам тензоров t1 и t2"""
    if is_1d(t1):
        return [f(x, y) for x, y in zip(t1, t2)]
    else:
        return [tensor_combine(f, t1_i, t2_i)
                for t1_i, t2_i in zip(t1, t2)]

import operator
assert tensor_combine(operator.add, [1, 2, 3], [4, 5, 6]) == [5, 7, 9]
assert tensor_combine(operator.mul, [1, 2, 3], [4, 5, 6]) == [4, 10, 18]
```

Абстракция слоя

В предыдущей главе мы построили простую нейронную сеть, которая позволила накладывать два слоя нейронов один на другой, каждый из которых вычислял сигмоиду `sigmoid(dot(weights, inputs))`.

Хотя это представление, возможно, идеализированное, демонстрирует действия реального нейрона, на практике хотелось бы обеспечить более широкий спектр действий. Пожалуй, было бы неплохо, чтобы нейроны помнили что-то о своих предыдущих входах. Возможно, мы хотели бы применять другую активационную функцию, чем сигмоидальная. И часто требуется использовать более двух слоев. (Наша функция `feed_forward` фактически обрабатывала любое число слоев, но наши вычисления градиента этого не делали.)

В этой главе мы построим механизм для имплементации такого разнообразия нейронных сетей. Нашей фундаментальной абстракцией будет слой `Layer`, т. е. объект, который знает, как применить некую функцию к своим входам и как распространять градиенты в обратном направлении.

Один из способов — думать о нейронных сетях, которые мы построили в *главе 18*, как о "линейном" слое, за которым следует "сигмоидальный" слой, затем еще один линейный слой и еще один сигмоидальный слой. Мы не различали их в этих терминах, но это позволит нам экспериментировать с гораздо более общими структурами:

```
from typing import Iterable, Tuple

class Layer:
    """Наши нейронные сети будут состоять из слоев, каждый из которых
       знает, как выполнять некоторые вычисления на своих входах
       в "прямом" направлении и распространять градиенты
       в "обратном" направлении."""
    def forward(self, input):
        """Обратите внимание на отсутствие типов.
           Мы не будем предписывать, какие типы входов слои могут
           принимать и какие виды выходов они могут возвращать."""
        raise NotImplementedError

    def backward(self, gradient):
        """Точно так же мы не будем предписывать, как выглядит градиент.
           В ваших обязанностях следить за тем, что вы все
           делаете разумно."""
        raise NotImplementedError

    def params(self) -> Iterable[Tensor]:
        """Возвращает параметры этого слоя. Дефолтная имплементация
           ничего не возвращает, так что если у вас есть слой без
           параметров, то вам не нужно его имплементировать."""
        return ()
```

```
    def grads(self) -> Iterable[Tensor]:
        """Возвращает градиенты в том же порядке, что и params()."""
        return ()
```

Методы `forward` и `backward` должны быть имплементированы в наших конкретных подклассах. Как только мы построим нейронную сеть, мы захотим натренировать ее с помощью градиентного спуска, т. е. захотим обновлять каждый параметр в сети, используя его градиент. Соответственно, мы настаиваем на том, чтобы каждый слой мог сообщать нам свои параметры и градиенты.

Некоторые слои (например, слой, который применяет функцию `sigmoid` к каждому из своих входов) не имеют параметров для обновления, поэтому мы предоставляем дефолтную имплементацию, которая обрабатывает этот случай.

Давайте посмотрим на этот слой:

```
from scratch.neural_networks import sigmoid

class Sigmoid(Layer):
    def forward(self, input: Tensor) -> Tensor:
        """Применить сигмоиду к каждому элементу входного тензора
           и сохранить результаты для использования в обратном
           распространении."""
        self.sigmoids = tensor_apply(sigmoid, input)
        return self.sigmoids

    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(lambda sig, grad: sig * (1 - sig) * grad,
                              self.sigmoids, gradient)
```

Здесь есть на что обратить внимание. Во-первых, во время прямого прохождения мы сохранили вычисленные сигмоиды для того, чтобы использовать их позже в обратном прохождении. Наши слои, как правило, должны делать такие вещи.

Во-вторых, вам может быть интересно, откуда взялось выражение `sig * (1 - sig) * grad`. Это всего-навсего цепное правило из дифференциального исчисления, и оно соответствует члену `output * (1 - output) * (output - target)` в наших предыдущих нейронных сетях.

Наконец, вы увидели, как можно применять функции `tensor_apply` и `tensor_combine`. В большинстве наших слоев эти функции будут использоваться схожим образом.

Линейный слой

Другой фрагмент, который нам понадобится для дублирования нейронных сетей из *главы 18*, — это "линейный" слой, представляющий ту часть нейронов, где выполняется скалярное умножение `dot(weights, inputs)`.

Этот слой будет иметь параметры, которые мы хотели бы инициализировать случайными значениями.

Оказывается, начальные значения параметров могут иметь огромную важность в том, как быстро сеть тренируется (а иногда сказываются на ее способности усваивать что-то вообще). Если веса слишком велики, то они могут производить большие выходы в интервале, где активационная функция имеет почти нулевые градиенты. И те части сети, которые имеют нулевые градиенты, неизбежно не смогут ничего усвоить с помощью градиентного спуска.

Соответственно, мы выполним имплементацию трех разных схем для случайного генерирования наших весовых тензоров. Первый — выбирать каждое значение из случайного равномерного распределения в интервале [0; 1], т. е. как `random.random()`. Второй (и по умолчанию) — выбирать каждое значение случайно из стандартного нормального распределения. И третий — использовать инициализацию Ксавье (Xavier), где каждый вес инициализируется случайным значением из нормального распределения со средним 0 и дисперсией `2/(num_inputs + num_outputs)`. Оказывается, для весов нейросети такая схема часто работает безупречно. Мы выполним имплементацию указанных схем с помощью функций `random_uniform` и `random_normal`:

```
import random
from scratch.probability import inverse_normal_cdf

def random_uniform(*dims: int) -> Tensor:
    if len(dims) == 1:
        return [random.random() for _ in range(dims[0])]
    else:
        return [random_uniform(*dims[1:]) for _ in range(dims[0])]

def random_normal(*dims: int,
                  mean: float = 0.0,
                  variance: float = 1.0) -> Tensor:
    if len(dims) == 1:
        return [mean + variance * inverse_normal_cdf(random.random())
                for _ in range(dims[0])]
    else:
        return [random_normal(*dims[1:], mean=mean, variance=variance)
                for _ in range(dims[0])]

assert shape(random_uniform(2, 3, 4)) == [2, 3, 4]
assert shape(random_normal(5, 6, mean=10)) == [5, 6]
```

А затем обернем их все в функцию `random_tensor`:

```
def random_tensor(*dims: int, init: str = 'normal') -> Tensor:
    if init == 'normal':
        return random_normal(*dims)
    elif init == 'uniform':
        return random_uniform(*dims)
    elif init == 'xavier':
        variance = len(dims) / sum(dims)
        return random_normal(*dims, variance=variance)
```

```
    else:
        raise ValueError(f"unknown init: {init}")
```

Теперь мы можем определить наш линейный слой. Нам нужно инициализировать его размерностью входов (которая говорит, сколько весов нужно каждому нейрону), размерностью выходов (которая говорит, сколько нейронов мы должны иметь) и схемой инициализации, которую мы хотим иметь:

```
from scratch.linear_algebra import dot

class Linear(Layer):
    def __init__(self,
                 input_dim: int,
                 output_dim: int,
                 init: str = 'xavier') -> None:
    """Слой из output_dim нейронов с input_dim весами каждый
      (and a bias)."""
    self.input_dim = input_dim
    self.output_dim = output_dim

    # self.w[o] - это веса для o-го нейрона
    self.w = random_tensor(output_dim, input_dim, init=init)

    # self.b[o] - это член смещения для o-го нейрона
    self.b = random_tensor(output_dim, init=init)
```

В случае если вы хотите знать, насколько важны схемы инициализации, то учтите, что некоторые сети в этой главе я вообще не смог натренировать с другими инициализациями, в отличие от тех, которые я использовал.

Метод `forward` имплементируется просто. Мы получим по одному выходу на каждый нейрон, который мы поместим в вектор. И выход каждого нейрона — это просто скалярное произведение (`dot`) его весов с входом плюс его смещение:

```
def forward(self, input: Tensor) -> Tensor:
    # Сохранить вход для использования в обратном прохождении
    self.input = input

    # Вернуть вектор выходов нейронов
    return [dot(input, self.w[o]) + self.b[o]
            for o in range(self.output_dim)]
```

Метод `backward` — изощреннее, но если вы знаете исчисление, то сложности не возникнут:

```
def backward(self, gradient: Tensor) -> Tensor:
    # Каждый b[o] добавляется в output[o], т. е.
    # градиент b тот же самый, что и градиент выхода
    self.b_grad = gradient
```

```
        # Каждый w[o][i] умножает input[i] и добавляется в output[o].
        # Поэтому его градиент равен input[i] * gradient[o]
        self.w_grad = [[self.input[i] * gradient[o]
                        for i in range(self.input_dim)]
                       for o in range(self.output_dim)]

        # Каждый input[i] умножает каждый w[o][i] и добавляется в каждый
        # output[o]. Поэтому его градиент равен сумме w[o][i] * gradient[o]
        # по всем выходам
        return [sum(self.w[o][i] * gradient[o] for o in range(self.output_dim))
                for i in range(self.input_dim)]
```

В "реальной" тензорной библиотеке эти (и многие другие) операции будут представлены как умножения матриц или тензоров, которые эти библиотеки призваны выполнять очень быстро. Наша библиотека очень медленная.

Наконец, здесь нам нужно имплементировать `params` и `grads`. У нас есть два параметра и два соответствующих градиента:

```
def params(self) -> Iterable[Tensor]:
    return [self.w, self.b]

def grads(self) -> Iterable[Tensor]:
    return [self.w_grad, self.b_grad]
```

Нейронные сети как последовательность слоев

Мы хотели бы думать о нейронных сетях как о последовательностях слоев, поэтому давайте придумаем способ объединять многочисленные слои в один. Результирующая нейронная сеть сама по себе является слоем, и она имплементирует методы класса `Layer` очевидными способами:

```
from typing import List

class Sequential(Layer):
    """Слой, состоящий из последовательности других слоев.
       Вы обязаны следить за тем, чтобы выход каждого слоя
       имел смысл в качестве входа в следующий слой."""
    def __init__(self, layers: List[Layer]) -> None:
        self.layers = layers

    def forward(self, input):
        """Распространить вход вперед через слои по порядку"""
        for layer in self.layers:
            input = layer.forward(input)
        return input
```

```
    def backward(self, gradient):
        """Распространить градиент назад через слои в универсуме"""
        for layer in reversed(self.layers):
            gradient = layer.backward(gradient)
        return gradient

    def params(self) -> Iterable[Tensor]:
        """Вернуть params из каждого слоя"""
        return (param for layer in self.layers for param in layer.params())

    def grads(self) -> Iterable[Tensor]:
        """Вернуть grads из каждого слоя"""
        return (grad for layer in self.layers for grad in layer.grads())
```

Таким образом, мы могли бы представить нейронную сеть, которую мы использовали для XOR, так:

```
xor_net = Sequential([
    Linear(input_dim=2, output_dim=2),
    Sigmoid(),
    Linear(input_dim=2, output_dim=1),
    Sigmoid()
])
```

Но нам по-прежнему требуется еще несколько инструментов, необходимых для ее тренировки.

Потеря и оптимизация

Ранее для наших моделей мы писали индивидуальные функции потери и функции градиента. Здесь мы хотим поэкспериментировать с разными функциями потери, поэтому (как обычно) введем новую абстракцию потери `Loss`, которая инкапсулирует вычисление потери и вычисление градиента:

```
class Loss:
    def loss(self, predicted: Tensor, actual: Tensor) -> float:
        """Насколько хорошим является предсказание?
           (Чем крупнее числа, тем хуже)"""
        raise NotImplementedError

    def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
        """Как изменяется потеря при изменении предсказаний?"""
        raise NotImplementedError
```

Мы уже много раз работали с потерей, которая является суммой квадратов ошибок, поэтому мы не должны испытывать трудности при ее имплементировании. Единственный трюк заключается в том, что нам нужно будет использовать функцию `tensor_combine`:

```
class SSE(Loss):
    """Функция потери, которая вычисляет сумму квадратов ошибок"""
    def loss(self, predicted: Tensor, actual: Tensor) -> float:
        # Вычислить тензор квадратических разностей
        squared_errors = tensor_combine(
            lambda predicted, actual: (predicted - actual) ** 2,
            predicted,
            actual)

        # И просто их сложить
        return tensor_sum(squared_errors)

    def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
        return tensor_combine(
            lambda predicted, actual: 2 * (predicted - actual),
            predicted,
            actual)
```

(Вскоре мы рассмотрим другую функцию потери.)

Последнее, что нужно выяснить, — это градиентный спуск. На протяжении всей книги мы осуществляли весь наш градиентный спуск вручную, имея тренировочный цикл, который включает в себя что-то вроде:

```
theta = gradient_step(theta, grad, -learning_rate)
```

Здесь это не совсем работает для нас по нескольким причинам. Во-первых, наши нейронные сети будут иметь много параметров, и нам нужно будет обновлять их все. Во-вторых, мы хотели бы иметь возможность использовать более умные варианты градиентного спуска, и мы не хотим переписывать их каждый раз.

Соответственно, мы введем (как вы уже догадались) абстракцию оптимизатора `Optimizer`, для которой градиентный спуск будет специфическим экземпляром:

```
class Optimizer:
    """Оптимизатор обновляет веса слоя (прямо на месте), используя
       информацию, известную либо слою, либо оптимизатору (либо обоим).
    """
    def step(self, layer: Layer) -> None:
        raise NotImplementedError
```

После этого легко имплементировать градиентный спуск, снова используя функцию `tensor_combine`:

```
class GradientDescent(Optimizer):
    def __init__(self, learning_rate: float = 0.1) -> None:
        self.lr = learning_rate

    def step(self, layer: Layer) -> None:
        for param, grad in zip(layer.params(), layer.grads()):
```

```
            # Обновить param, используя градиентный шаг
            param[:] = tensor_combine(
                lambda param, grad: param - grad * self.lr,
                param,
                grad)
```

Единственное, что может удивить, — это "срезовое присвоение", которое отражает тот факт, что переприсвоение списка не изменяет его первоначальное значение. То есть если вы только что выполнили `param = tensor_combine(...)`, то переопределили локальную переменную `param`, но не повлияли на исходный тензор параметров, хранящийся в слое. Однако если вы присваиваете срезу `[:]`, то это фактически изменяет значения внутри списка.

Вот простой пример, который это демонстрирует:

```
tensor = [[1, 2], [3, 4]]

for row in tensor:
    row = [0, 0]
assert tensor == [[1, 2], [3, 4]], "присвоение не обновляет список"

for row in tensor:
    row[:] = [0, 0]
assert tensor == [[0, 0], [0, 0]], "но присвоение срезу это делает"
```

Если вы несколько неопытны в Python, то такое поведение может быть удивительным, поэтому помедитируйте на нем и разбирайте примеры самостоятельно до тех пор, пока это не станет для вас понятным.

Для демонстрации ценности этой абстракции давайте имплементируем еще один оптимизатор, который использует *импульс*. Идея заключается в том, что мы не хотим чрезмерно реагировать на каждый новый градиент, и поэтому поддерживаем среднее значение встречаемых градиентов, обновляя его с каждым новым градиентом и делая шаг в направлении среднего:

```
class Momentum(Optimizer):
    def __init__(self,
                 learning_rate: float,
                 momentum: float = 0.9) -> None:
        self.lr = learning_rate
        self.mo = momentum
        self.updates: List[Tensor] = []  # Скользящее среднее

    def step(self, layer: Layer) -> None:
        # Если у нас нет предыдущих обновлений, то начать со всех нулей
        if not self.updates:
            self.updates = [zeros_like(grad) for grad in layer.grads()]

        for update, param, grad in zip(self.updates,
                                       layer.params(),
                                       layer.grads()):
```

```
            # Применить импульс
            update[:] = tensor_combine(
                lambda u, g: self.mo * u + (1 - self.mo) * g,
                update,
                grad)

        # Затем сделать градиентный шаг
        param[:] = tensor_combine(
            lambda p, u: p - self.lr * u,
            param,
            update)
```

Поскольку мы использовали абстракцию оптимизатора `Optimizer`, мы можем легко переключаться между нашими разными оптимизаторами.

Пример: сеть XOR еще раз

Давайте посмотрим, насколько легко использовать нашу новую структуру для тренировки сети, которая может вычислять XOR. Начнем с повторного создания тренировочных данных:

```
# Тренировочные данные
xs = [[0., 0], [0., 1], [1., 0], [1., 1]]
ys = [[0.], [1.], [1.], [0.]]
```

и затем зададим сеть, хотя теперь мы можем оставить последний слой сигмоидальным:

```
random.seed(0)

net = Sequential([
    Linear(input_dim=2, output_dim=2),
    Sigmoid(),
    Linear(input_dim=2, output_dim=1)
])
```

Далее мы можем написать простой тренировочный цикл, за исключением того, что теперь можно использовать абстракции оптимизатора `Optimizer` и потери `Loss`. Это позволяет нам легко попробовать разные варианты:

```
import tqdm

optimizer = GradientDescent(learning_rate=0.1)
loss = SSE()

with tqdm.trange(3000) as t:
    for epoch in t:
        epoch_loss = 0.0
```

```
        for x, y in zip(xs, ys):
            predicted = net.forward(x)
            epoch_loss += loss.loss(predicted, y)
            gradient = loss.gradient(predicted, y)
            net.backward(gradient)

            optimizer.step(net)

        t.set_description(f"xor потеря {epoch_loss:.3f}")
```

Этот фрагмент кода должен тренировать быстро, и вы должны увидеть, что потеря идет вниз. И теперь мы можем обследовать веса:

```
for param in net.params():
    print(param)
```

В моей сети получилось примерно:

```
hidden1 = -2.6 * x1 + -2.7 * x2 + 0.2  # NOR
hidden2 = 2.1 * x1 + 2.1 * x2 - 3.4    # AND
output = -3.1 * h1 + -2.6 * h2 + 1.8   # NOR
```

Таким образом, `hidden1` активируется, если ни один вход не равен 1; `hidden2` активируется, если оба входа равны 1. И `output` активируется, если ни один из скрытых выходов не равен 1, т. е. если ни один из входов не равен 1, а также если оба входа не равны 1. И действительно, это именно та самая логика XOR.

Обратите внимание, что эта сеть усвоила другие признаки, чем та, которую мы тренировали в *главе 18*, но ей все равно удается делать то же самое.

Другие активационные функции

Сигмоидальная функция `sigmoid` вышла из моды по нескольким причинам. Одна из причин заключается в том, что `sigmoid(0)` равно 1/2, т. е. нейрон, входы которого в сумме составляют 0, имеет положительный выход. Другая же состоит в том, что ее градиент очень близок к 0 для очень больших и очень малых входов, т. е. ее градиенты могут "насыщаться", а ее веса могут застревать.

Одной из популярных замен является функция `tanh` (гиперболический тангенс), т. е. другая сигмоидальная функция, которая варьируется в интервале от -1 до 1 и выводит 0, если ее вход равен 0. Производная `tanh(x)` — это просто `1 - tanh(x) ** 2`, что упрощает написание слоя:

```
import math

def tanh(x: float) -> float:
    # Если x является очень большим или очень малым,
    # то tanh (по существу) равен 1 или -1.
    # Мы делаем проверку этого, потому что,
    # например, math.exp(1000) вызывает ошибку
```

```
    if x < -100: return -1
    elif x > 100: return 1

    em2x = math.exp(-2 * x)
    return (1 - em2x) / (1 + em2x)

class Tanh(Layer):
    def forward(self, input: Tensor) -> Tensor:
        # Сохранить выход tanh для ипользования в обратном прохождении
        self.tanh = tensor_apply(tanh, input)
        return self.tanh

    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(
            lambda tanh, grad: (1 - tanh ** 2) * grad,
            self.tanh,
            gradient)
```

В более крупных сетях еще одной популярной заменой является активационная функция `Relu`, которая равна 0 для отрицательных входов и тождественна для положительных входов:

```
class Relu(Layer):
    def forward(self, input: Tensor) -> Tensor:
        self.input = input
        return tensor_apply(lambda x: max(x, 0), input)

    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(lambda x, grad: grad if x > 0 else 0,
                              self.input,
                              gradient)
```

Есть еще много других. Я призываю вас поиграть с ними в ваших сетях.

Пример: задача Fizz Buzz еще раз

Теперь мы можем применить наш вычислительнй каркас "глубокого обучения" для того, чтобы воспроизвести наше решение из *разд. "Пример: задача Fizz Buzz" главы 18*. Давайте настроим данные:

```
from scratch.neural_networks import binary_encode, fizz_buzz_encode, argmax

xs = [binary_encode(n) for n in range(101, 1024)]
ys = [fizz_buzz_encode(n) for n in range(101, 1024)]
```

и создадим сеть:

```
NUM_HIDDEN = 25

random.seed(0)
```

```
net = Sequential([
    Linear(input_dim=10, output_dim=NUM_HIDDEN, init='uniform'),
    Tanh(),
    Linear(input_dim=NUM_HIDDEN, output_dim=4, init='uniform'),
    Sigmoid()
])
```

Поскольку мы тренируем, давайте также отслеживать нашу точность на тренировочном наборе:

```
def fizzbuzz_accuracy(low: int, hi: int, net: Layer) -> float:
    num_correct = 0
    for n in range(low, hi):
        x = binary_encode(n)
        predicted = argmax(net.forward(x))
        actual = argmax(fizz_buzz_encode(n))
        if predicted == actual:
            num_correct += 1

    return num_correct / (hi - low)

optimizer = Momentum(learning_rate=0.1, momentum=0.9)
loss = SSE()

with tqdm.trange(1000) as t:
    for epoch in t:
        epoch_loss = 0.0

        for x, y in zip(xs, ys):
            predicted = net.forward(x)
            epoch_loss += loss.loss(predicted, y)
            gradient = loss.gradient(predicted, y)
            net.backward(gradient)

            optimizer.step(net)

        accuracy = fizzbuzz_accuracy(101, 1024, net)
        t.set_description(f"fb потеря: {epoch_loss:.2f} точн: {accuracy:.2f}")

# Теперь проверим результаты на тестовом наборе
print("тестовые результаты", fizzbuzz_accuracy(1, 101, net))
```

После 1000 тренировочных итераций модель получает точность 90% на тестовом наборе; если вы продолжите тренировать ее дольше, то она должна работать еще лучше. (Я не думаю, что можно натренировать ее с точностью до 100%, имея всего лишь 25 скрытых элементов, но это определенно возможно, если вы дойдете до 50 скрытых элементов.)

Функции *softmax* и перекрестная энтропия

Нейронная сеть, которую мы использовали в предыдущем разделе, заканчивалась сигмоидальным слоем `Sigmoid`, т. е. его выходом был вектор чисел между 0 и 1. В частности, он мог бы выводить вектор, целиком состоящий из нулей, либо вектор, целиком состоящий из единиц. С другой стороны, при решении классификационных задач удобно выводить 1 для правильного класса и 0 для всех неправильных классов. Как правило, наши предсказания не будут столь совершенными, но мы, по крайней мере, хотели бы предсказывать фактическое распределение вероятностей по классам.

Например, если у нас есть два класса и наша модель выводит `[0, 0]`, то трудно понять, что имеется в виду. Она не считает, что выход принадлежит какому-то из двух классов?

Но если наша модель выводит `[0.4, 0.6]`, то мы можем интерпретировать выход как предсказание, что с вероятностью 0.4 наш вход принадлежит первому классу, а с вероятностью 0.6 — второму классу.

Для того чтобы этого добиться, как правило, нужно отказаться от завершающего слоя `Sigmoid` и вместо него использовать функцию `softmax`, которая преобразовывает вектор действительных чисел в вектор вероятностей. Мы вычисляем `exp(x)` для каждого числа в векторе, что приводит к вектору положительных чисел. После этого мы просто делим каждое из этих положительных чисел на сумму, что дает нам кучу положительных чисел, которые при их сложении составляют 1, т. е. вектор вероятностей.

Если мы когда-нибудь попытаемся вычислить, скажем, `exp(1000)`, то получим ошибку Python, поэтому, прежде чем брать `exp`, мы вычитаем наибольшее значение. Это дает результат в тех же вероятностях, просто безопаснее вычисляется на Python:

```
def softmax(tensor: Tensor) -> Tensor:
    """Взять softmax вдоль последней размерности"""
    if is_1d(tensor):
        # Вычесть наибольшее значение в целях числовой стабильности.
        largest = max(tensor)
        exps = [math.exp(x - largest) for x in tensor]
        sum_of_exps = sum(exps)          # Это суммарный "вес".
        return [exp_i / sum_of_exps      # Вероятность - это доля
                for exp_i in exps]       # суммарного веса.
    else:
        return [softmax(tensor_i) for tensor_i in tensor]
```

Когда сеть производит вероятности, нередко используют другую функцию потери, именуемую *перекрестной энтропией* (или иногда "отрицательным логарифмическим правдоподобием").

Вы, возможно, помните, что в *разд. "Оценивание максимального правдоподобия" главы 14* мы оправдывали использование наименьших квадратов в линейной ре-

грессии, апеллируя к тому факту, что (при определенных допущениях) коэффициенты наименьших квадратов максимизируют правдоподобие наблюдаемых данных.

Здесь мы можем сделать что-то подобное: если наши выходы сети являются вероятностями, то перекрестно-энтропийная потеря представляет собой отрицательное логарифмическое правдоподобие наблюдаемых данных, и это означает, что минимизация такой потери равна максимизации логарифмического правдоподобия (и, следовательно, правдоподобия) тренировочных данных.

Как правило, мы не будем включать функцию `softmax` в состав самой нейронной сети. Причина, как оказалось, в том, что, если `softmax` является частью вашей функции потери, но не частью самой сети, то градиенты потери по отношению к выходам сети очень легко вычислить.

```
class SoftmaxCrossEntropy(Loss):
    """
    Отрицательное логарифмическое правдоподобие наблюдаемых значений
    при наличии нейросетевой модели. Поэтому, если мы выберем веса
    для ее минимизации, то наша модель будет максимизировать
    правдоподобие наблюдаемых данных
    """
    def loss(self, predicted: Tensor, actual: Tensor) -> float:
        # Применить softmax, чтобы получить вероятности
        probabilities = softmax(predicted)

        # Это будет логарифмом p_i для фактического класса i
        # и 0 для других классов. Мы добавляем крошечное значение
        # в p во избежание взятия log(0).
        likelihoods = tensor_combine(lambda p,
                                     act: math.log(p + 1e-30) * act,
                                     probabilities,
                                     actual)

        # И затем просто просуммировать отрицательные значения
        return -tensor_sum(likelihoods)

    def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
        probabilities = softmax(predicted)

        # Разве это не приятное уравнение?
        return tensor_combine(lambda p, actual: p - actual,
                              probabilities,
                              actual)
```

Если теперь натренировать ту же самую сеть Fizz Buzz, используя потерю `SoftmaxCrossEntropy`, то обнаружится, что она обычно тренируется намного быстрее (т. е. за гораздо меньшее число эпох). Предположительно, это происходит потому, что гораздо проще найти веса, которые приводятся в заданное распределение

функцией `softmax`, чем найти веса, которые приводятся в заданное распределение функцией `sigmoid`.

То есть если мне нужно предсказать класс 0 (вектор с 1 в первой позиции и 0 в остальных позициях), то в линейном + сигмоидальном случаях (слой `linear` + слой `sigmoid`) мне нужно, чтобы первый выход был большим положительным числом, а остальные выходы — большими отрицательными числами. Однако в случае `softmax` мне просто нужно, чтобы первый выход был больше остальных выходов. Очевидно, существует намного больше способов того, что произойдет второй случай, в предположении, что легче найти веса, которые делают это так:

```
random.seed(0)

net = Sequential([
    Linear(input_dim=10, output_dim=NUM_HIDDEN, init='uniform'),
    Tanh(),
    Linear(input_dim=NUM_HIDDEN, output_dim=4, init='uniform')
    # Теперь нет завершающего сигмоидального слоя
])

optimizer = Momentum(learning_rate=0.1, momentum=0.9)
loss = SoftmaxCrossEntropy()

with tqdm.trange(100) as t:
    for epoch in t:
        epoch_loss = 0.0

        for x, y in zip(xs, ys):
            predicted = net.forward(x)
            epoch_loss += loss.loss(predicted, y)
            gradient = loss.gradient(predicted, y)
            net.backward(gradient)

            optimizer.step(net)

        accuracy = fizzbuzz_accuracy(101, 1024, net)
        t.set_description(f"fb потеря: {epoch_loss:.3f} точн: {accuracy:.2f}")

# Снова проверим результаты на тестовом наборе
print("тестовые результаты", fizzbuzz_accuracy(1, 101, net))
```

Слой отсева

Как и большинство автоматически обучающихся моделей, нейронные сети склонны к переподгонке к своим тренировочным данным. Мы уже видели способы сгладить это; например, в *разд. "Регуляризация" главы 15* мы штрафовали большие веса, и это помогало предотвратить переподгонку.

Распространенным способом регуляризации нейронных сетей является использование отсева. Во время тренировки мы случайно отключаем каждый нейрон (т. е. заменяем его выход на 0) с некоторой фиксированной вероятностью. Это означает, что сеть не сможет научиться зависеть от какого-либо отдельного нейрона, что, похоже, помогает в предотвращении переподгонки.

Мы не хотим отсеивать нейроны во время оценивания, поэтому отсеивающий слой `Dropout` должен знать, тренируется он или нет. Во время тренировки слой `Dropout` пропускает только некоторую случайную часть своего входа. Для того чтобы сделать его выход сопоставимым во время оценивания, мы будем шкалировать выходы (равномерно), используя ту же долю:

```
class Dropout(Layer):
    def __init__(self, p: float) -> None:
        self.p = p
        self.train = True

    def forward(self, input: Tensor) -> Tensor:
        if self.train:
            # Создать маску из нулей и единиц с формой, что и вход,
            # используя указанную вероятность
            self.mask = tensor_apply(
                lambda _: 0 if random.random() < self.p else 1,
                input)
            # Умножить на маску для отсева входов
            return tensor_combine(operator.mul, input, self.mask)
        else:
            # Во время оценивания просто прошкалировать выходы равномерно.
            return tensor_apply(lambda x: x * (1 - self.p), input)

    def backward(self, gradient: Tensor) -> Tensor:
        if self.train:
            # Распространять градиенты только там, где mask == 1
            return tensor_combine(operator.mul, gradient, self.mask)
        else:
            raise RuntimeError("не вызывайте backward в тренировочном режиме")
```

Мы будем использовать этот класс для предотвращения переподгонки наших глубоко обучающихся моделей.

Пример: набор данных MNIST

MNIST (**http://yann.lecun.com/exdb/mnist/**) — это набор данных рукописных цифр, который все используют для проектов в области глубокого обучения.

Он доступен в несколько сложном двоичном формате, поэтому для работы с ним мы установим библиотеку mnist. (Вы правы, технически эта часть не относится к теме "с нуля".)

```
python -m pip install mnist
```

И затем мы можем загрузить данные:

```
import mnist
#
# Этот фрагмент скачает данные; поменяйте путь на тот, который вам нужен.
# (Да, это функция с 0 аргументами, именно то, что ожидает библиотека.)
# (И да, я назначаю лямбду переменной, хотя я советовал
# это никогда не делать.)
mnist.temporary_dir = lambda: '/tmp'    # в ОС Windows без слеша ('tmp')

# Обе функции сначала скачивают данные, а затем возвращают массив NumPy.
# Мы вызываем .tolist (), потому что "тензоры" - это просто списки.
train_images = mnist.train_images().tolist()
train_labels = mnist.train_labels().tolist()

assert shape(train_images) == [60000, 28, 28]
assert shape(train_labels) == [60000]
```

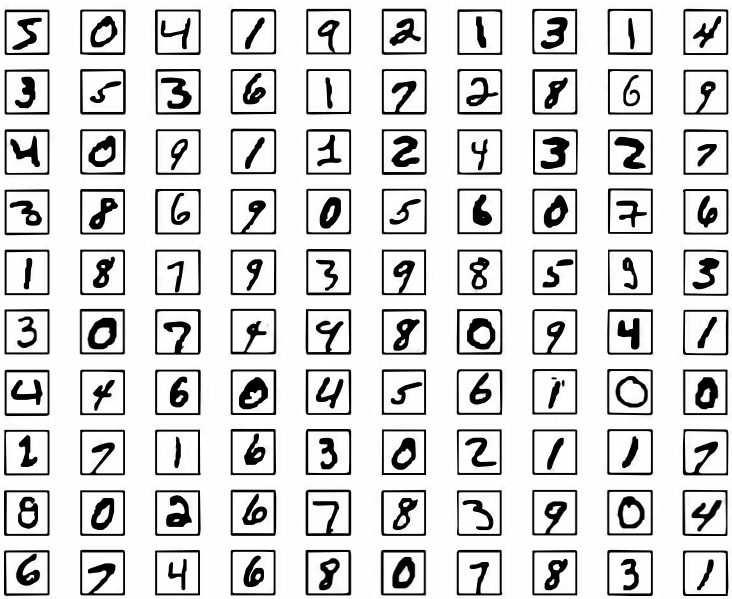


Рис. 19.1. Изображения набора данных MNIST

Давайте выведем первые 100 тренировочных изображений на график, чтобы увидеть, как они выглядят (рис. 19.1):

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(10, 10)

for i in range(10):
    for j in range(10):
```

```
        # Изобразить каждый снимок в черно-белом цвете и спрятать оси.
        ax[i][j].imshow(train_images[10 * i + j], cmap='Greys')
        ax[i][j].xaxis.set_visible(False)
        ax[i][j].yaxis.set_visible(False)

plt.show()
```

Вы видите, что они действительно выглядят как цифры, написанные от руки.

Моя первая попытка показать изображения привела к желтым цифрам на черном фоне. Я не настолько умен и проницателен, чтобы знать, что для получения черно-белых изображений мне нужно было добавить `cmap=Greys`; я погуглил и нашел решение на Stack Overflow. Будучи исследователем данных, вы станете довольно искусным в этом рабочем потоке.

Нам также нужно загрузить тестовые изображения:

```
test_images = mnist.test_images().tolist()
test_labels = mnist.test_labels().tolist()

assert shape(test_images) == [10000, 28, 28]
assert shape(test_labels) == [10000]
```

Каждое изображение имеет формат 28×28 пикселов, но наши линейные слои могут иметь дело только с одномерными входами, поэтому мы просто сгладим их (а также разделим на 256, получив их в интервале между 0 и 1). Кроме того, наша нейронная сеть будет тренироваться лучше, если наши входы в среднем равны 0, поэтому мы вычитаем среднее значение:

```
# Вычислить среднее пиксельное значение
avg = tensor_sum(train_images) / 60000 / 28 / 28

# Перецентрировать, перешкалировать и сгладить
train_images = [[(pixel - avg) / 256 for row in image for pixel in row]
                for image in train_images]
test_images = [[(pixel - avg) / 256 for row in image for pixel in row]
               for image in test_images]

assert shape(train_images) == [60000, 784], "снимки должны быть сглажены"
assert shape(test_images) == [10000, 784], "снимки должны быть сглажены"

# После центрирования средний пиксел должен варьироваться
# очень близко к 0
assert -0.0001 < tensor_sum(train_images) < 0.0001
```

Мы также хотим закодировать цели в формате с одним активным состоянием[2], т. к. у нас 10 выходов. Сначала давайте напишем функцию `one_hot_encode`:

[2] Словосочетание "кодирование с одним активным состоянием" (one-hot encoding) пришло из терминологии цифровых интегральных микросхем, где оно описывает конфигурацию микросхемы, в которой допускается, чтобы только один бит был положительным (активным — hot). — *Прим. пер.*

```
def one_hot_encode(i: int, num_labels: int = 10) -> List[float]:
    return [1.0 if j == i else 0.0 for j in range(num_labels)]

assert one_hot_encode(3) == [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
assert one_hot_encode(2, num_labels=5) == [0, 0, 1, 0, 0]
```

а затем применим ее к нашим данным:

```
train_labels = [one_hot_encode(label) for label in train_labels]
test_labels = [one_hot_encode(label) for label in test_labels]

assert shape(train_labels) == [60000, 10]
assert shape(test_labels) == [10000, 10]
```

Одной из сильных сторон наших абстракций является то, что мы можем использовать один и тот же цикл тренировки/оценивания с разными моделями. Давайте сначала напишем его. Мы передадим ему нашу модель, данные, функцию потери и (если мы выполняем тренировку, то) оптимизатор.

Цикл будет сканировать данные, отслеживать результативность и (если мы передали оптимизатор) обновлять параметры:

```
import tqdm

def loop(model: Layer,
         images: List[Tensor],
         labels: List[Tensor],
         loss: Loss,
         optimizer: Optimizer = None) -> None:
    correct = 0       # Отслеживать число правильных предсказаний
    total_loss = 0.0  # Отслеживать суммарную потерю

    with tqdm.trange(len(images)) as t:
        for i in t:
            predicted = model.forward(images[i])           # Предсказать
            if argmax(predicted) == argmax(labels[i]):     # Проверить на
                correct += 1                               # правильность
            total_loss += loss.loss(predicted, labels[i])  # Вычислить
                                                           # потерю
            # Мы тренируем, распространяем градиент по сети назад
            # и обновляем веса
            if optimizer is not None:
                gradient = loss.gradient(predicted, labels[i])
                model.backward(gradient)
                optimizer.step(model)

            # И обновить наши метрики в индикаторе выполнения
            avg_loss = total_loss / (i + 1)
            acc = correct / (i + 1)
            t.set_description(f"mnist потеря: {avg_loss:.3f} точн: {acc:.3f}")
```

В качестве базовой линии мы можем применить нашу библиотеку глубокого обучения для тренировки (многоклассовой) логистической регрессионной модели, которая представляет собой всего один линейный слой, за которым следует `softmax`. Эта модель (по сути) просто ищет 10 линейных функций, таких, что если вход представляет, скажем, 5, то пятая линейная функция производит наибольший выход.

Одного прохождения через наши 60 000 тренировочных примеров должно быть достаточно для того, чтобы усвоить модель:

```
random.seed(0)

# Логистическая регрессия - это просто линейный слой,
# за которым следует softmax
model = Linear(784, 10)
loss = SoftmaxCrossEntropy()

# Этот оптимизатор, похоже, справляется
optimizer = Momentum(learning_rate=0.01, momentum=0.99)

# Натренировать на тренировочных данных
loop(model, train_images, train_labels, loss, optimizer)

# Протестировать на тестовых данных
# (отсутствие оптимизатора означает просто оценивание)
loop(model, test_images, test_labels, loss)
```

В результате мы получим точность примерно 89%. Давайте посмотрим, сможем ли мы добиться лучшего с помощью глубокой нейронной сети. Мы будем использовать два скрытых слоя: первый — с 30 нейронами, второй — с 10 нейронами. И мы применим нашу активацию `Tanh`:

```
random.seed(0)

# Дадим им имена, чтобы можно было включать/выключать тренировку
dropout1 = Dropout(0.1)
dropout2 = Dropout(0.1)

model = Sequential([
    Linear(784, 30),  # Скрытый слой 1: размер 30
    dropout1,
    Tanh(),
    Linear(30, 10),   # Скрытый слой 2: размер 10
    dropout2,
    Tanh(),
    Linear(10, 10)    # Выходной слой: размер 10
])
```

И мы можем использовать тот же самый цикл!

```
optimizer = Momentum(learning_rate=0.01, momentum=0.99)
loss = SoftmaxCrossEntropy()
```

```
# Включить отсев и тренировать (занимает > 20 минут на моем ноуте!)
dropout1.train = dropout2.train = True
loop(model, train_images, train_labels, loss, optimizer)

# Отключить отсев и оценить
dropout1.train = dropout2.train = False
loop(model, test_images, test_labels, loss)
```

Наша глубокая модель получает точность лучше, чем 92% на тестовом наборе, что является солидным улучшением по сравнению с простой логистической моделью.

Веб-сайт набора данных MNIST[3] описывает различные модели, которые превосходят наши. Многие из них могут быть имплементированы с использованием механизма, разработанного нами выше, но тренировка в нашем вычислительном каркасе, где тензоры представлены в качестве списков, потребует очень много времени. Некоторые из лучших моделей включают *сверточные* слои, которые очень важны, но, к сожалению, совершенно не подходят для вводной книги по науке о данных.

Сохранение и загрузка моделей

Тренировка этих моделей занимает много времени, поэтому было бы хорошо, если бы мы могли сохранять их в файл, благодаря чему нам не придется тренировать их каждый раз заново. К счастью, мы можем использовать модуль `json`, с легкостью сериализуя модельные веса в файл.

Для экономии можно использовать `Layer.params` для сбора весов, вставлять их в список и применять функцию `json.dump` для сохранения этого списка в файле:

```
import json

def save_weights(model: Layer, filename: str) -> None:
    weights = list(model.params())
    with open(filename, 'w') as f:
        json.dump(weights, f)
```

Загрузка весов назад в программу добавляет лишь немного больше работы. Мы просто используем функцию `json.load`, получая список весов из файла и делая срезовое присвоение для установки весов нашей модели.

(В частности, это означает, что мы должны создать экземпляр модели сами, а затем загрузить веса. Альтернативный подход заключался бы в том, чтобы сохранять некоторое представление архитектуры модели и использовать его для создания экземпляра модели. Это не ужасная идея, но здесь потребуется намного больше кода и изменений во всех наших слоях, поэтому мы будем придерживаться более простого способа.)

Прежде чем загружать веса, мы хотели бы проверить, имеют ли они ту же форму, что и модели, в которые мы их загружаем. (Это гарантия, например, от попытки

[3] См. **http://yann.lecun.com/exdb/mnist/**.

загрузки весов сохраненной глубокой сети в мелкую сеть или аналогичных проблем.)

```
def load_weights(model: Layer, filename: str) -> None:
    with open(filename) as f:
        weights = json.load(f)

    # Проверить на непротиворечивость
    assert all(shape(param) == shape(weight)
               for param, weight in zip(model.params(), weights))

    # Затем загрузить, применив срезовое присвоение
    for param, weight in zip(model.params(), weights):
        param[:] = weight
```

В представлении JSON ваши данные хранятся в виде текста, что делает такое представление крайне неэффективным. В реальных приложениях вы, вероятно, используете библиотеку сериализации pickle, которая сериализует объекты в более эффективный двоичный формат. Здесь я решил оставить все простым и удобочитаемым.

Веса для различных сетей, которые мы тренируем, можно загрузить из репозитория книги на GitHub[4].

Для дальнейшего изучения

Глубокое обучение сейчас действительно является горячей темой, и в этой главе мы лишь слегка к ней прикоснулись. Существует много хороших книг и постов в блогах (и много-много плохих) почти о любом аспекте глубокого обучения, о котором вы хотели бы знать.

- Канонический учебник "Глубокое обучение"[5] (издательство MIT Press) Яна Гудфеллоу (Ian Goodfellow), Джошуа Бенджо (Yoshua Bengio) и Аарона Курвилля (Aaron Courville) находится в свободном доступе в Интернете. Учебник очень хорош, но содержит довольно много математики.
- Книга "Глубокое обучение с помощью Python"[6] (издательство "Manning") Франсуа Шолле (Francois Chollet) является отличным введением в библиотеку Keras, по образу которой была создана наша библиотека глубокого обучения.
- Для глубокого обучения я сам в основном использую библиотеку PyTorch[7]. На ее веб-сайте представлено много документации и учебных пособий.

---

[4] См. **https://github.com/joelgrus/data-science-from-scratch**.

[5] См. **https://www.deeplearningbook.org/**.

[6] См. **https://www.manning.com/books/deep-learning-with-python**.

[7] См. **https://pytorch.org/**.

ГЛАВА 20
Кластеризация

Где славный наш союз
Мятежно избавлял от уз.
– *Роберт Геррик*[1]

Большинство алгоритмов в этой книге известны как алгоритмы контролируемого обучения, т. е. они начинают работу с совокупности *помеченных* данных и используют их в качестве основы для выполнения предсказаний о новых, непомеченных данных. Кластеризация же — это пример неконтролируемого обучения, в которой мы работаем с совершенно непомеченными данными (или в которой наши данные помечены, но мы игнорируем метки).

Идея

Всякий раз, когда вы смотрите на какой-либо источник данных, то, вполне вероятно, предполагаете, что данные будут каким-то образом образовывать *кластеры*. Набор данных, показывающий, где живут миллионеры, возможно, имеет кластеры в таких местах, как Беверли-Хиллз и Манхэттен. Набор данных, показывающий, сколько времени люди работают каждую неделю, вероятно, имеет кластер около 40 часов (а если он взят из штата с законами, которые предусматривают специальные льготы для людей, работающих не более 20 часов в неделю, то он, скорее всего, создаст еще один кластер непосредственно возле 19). Набор данных о демографии зарегистрированных избирателей, скорее всего, образует несколько кластеров (например, "футбольные мамочки", "скучающие пенсионеры", "безработные представители поколения Y"), которых социологи и политические консультанты, по-видимому, посчитают актуальными.

В отличие от некоторых задач, которые мы рассмотрели ранее, обычно нет "правильной" кластеризации. Альтернативная схема кластеризации может сгруппировать некоторых "безработных поколения Y" с "аспирантами", другая — с "обитателями родительских подвалов"[2]. Ни одна схема не является обязательно правильнее

[1] Роберт Геррик (1591–1674) — английский поэт, представитель группы так называемых "поэтов-кавалеров", сторонников короля Карла I. Здесь обыгрывается слово "союз" (cluster). — *Прим. пер.*

[2] Обитатели родительских подвалов (**https://resurgencemedia.net/2015/10/19/millennial-basement-dwellers-and-their-pathetic-parents/**) — термин, чаще всего относящийся к иждивенцам у родителей. Это люди, которые проводят всю или большую часть времени в подвале дома. Проживание в подвале — это форма ухода от действительности и современный тренд в США, связанный с использованием последних достижений в области электроники и вычислительной техники, но который сродни тенденции

другой — напротив, каждая, скорее всего, является оптимальнее по отношению к своей собственной метрике, измеряющей качество кластеров.

Более того, кластеры не помечают сами себя. Вам придется делать это самостоятельно, глядя на данные, лежащие в основе каждого из них.

Модель

Для нас каждый вход `input` будет вектором в d-размерном пространстве, который, как обычно, мы представим в виде списка чисел. Наша задача — выявлять кластеры схожих входов и (иногда) отыскивать репрезентативное значение для каждого кластера.

Например, каждый вход может быть числовым вектором, который представляет заголовок поста в блоке; в таком случае наша цель могла бы заключаться в отыскании кластеров похожих постов, возможно, для того, чтобы понять, о чем пользователи веб-сайта пишут в своих блогах. Либо представим, что у нас есть фотография, содержащая тысячи оттенков (красного, зеленого и синего) цвета, и нам нужно сделать ее экранную копию в 10-цветном формате. Здесь кластеризация способна помочь нам выбрать 10 цветов, которые будут минимизировать суммарную "ошибку цветности"[3].

Одним из простейших кластерных методов является *k средних*, в котором число кластеров k выбирается заранее, после чего цель состоит в подразделении входов на подмножества $S_1, ..., S_k$ таким образом, чтобы минимизировать полную сумму квадратов расстояний от каждой точки до среднего значения назначенного ей кластера.

Существует целый ряд способов, которыми можно назначить n точек k кластерам, а это значит, что задача отыскания оптимальной кластеризации является очень трудной. Мы остановимся на итеративном алгоритме, который обычно отыскивает хорошую кластеризацию.

1. Начать с множества *k средних*, т. е. точек в d-размерном пространстве.
2. Назначить каждую точку среднему значению, к которому она находится ближе всего.
3. Если ни у одной точки ее назначение не изменилось, то остановиться и сохранить кластеры.
4. Если назначение одной из точек изменилось, то пересчитать средние и вернуться к шагу 2.

При помощи функции `vector_mean` из *главы 4* достаточно просто создать класс, который это выполняет. Мы будем использовать следующий ниже фрагмент кода для отслеживания тренировочного процесса:

предыдущих поколений, когда такой выбор делался большей частью из желания избежать очного контакта с другими людьми. — *Прим. пер.*

[3] См. **http://www.imagemagick.org/script/quantize.php#measure**.

```
from scratch.linear_algebra import Vector

def num_differences(v1: Vector, v2: Vector) -> int:
    assert len(v1) == len(v2)
    return len([x1 for x1, x2 in zip(v1, v2) if x1 != x2])

assert num_differences([1, 2, 3], [2, 1, 3]) == 2
assert num_differences([1, 2], [1, 2]) == 0
```

Нам также нужна функция, которая с учетом нескольких векторов и их назначений кластерам вычисляет средние кластеров. Может оказаться, что некий кластер не будет иметь назначенных ему точек. Мы не можем взять среднее значение пустой коллекции, поэтому мы просто случайно выберем одну из точек, которая будет служить "средним значением" этого кластера:

```
from typing import List
from scratch.linear_algebra import vector_mean

def cluster_means(k: int,
                  inputs: List[Vector],
                  assignments: List[int]) -> List[Vector]:
    # clusters[i] содержит входы, чье назначение равно i
    clusters = [[] for i in range(k)]
    for input, assignment in zip(inputs, assignments):
        clusters[assignment].append(input)

    # Если кластер пустой, то просто взять случайную точку
    return [vector_mean(cluster) if cluster else random.choice(inputs)
            for cluster in clusters]
```

И теперь мы готовы закодировать наш кластеризатор. Как обычно, мы будем использовать библиотеку tqdm, отслеживая продвижение работы, но здесь мы не знаем, сколько итераций потребуется, поэтому мы применяем функцию `itertools.count`, которая создает бесконечно итерируемый объект, и мы вернемся из него, когда закончим:

```
import itertools
import random
import tqdm
from scratch.linear_algebra import squared_distance

class KMeans:
    def __init__(self, k: int) -> None:
        self.k = k          # Число кластеров
        self.means = None

    def classify(self, input: Vector) -> int:
        """Вернуть индекс кластера, ближайшего к входному значению"""
        return min(range(self.k),
                   key=lambda i: squared_distance(input, self.means[i]))
```

```
    def train(self, inputs: List[Vector]) -> None:
        # Начать со случайных назначений
        assignments = [random.randrange(self.k) for _ in inputs]
        with tqdm.tqdm(itertools.count()) as t:
            for _ in t:
                # Вычислить средние и отыскать новые назначения
                self.means = cluster_means(self.k, inputs, assignments)
                new_assignments = [self.classify(input)
                                   for input in inputs]

                # Проверить, сколько назначений изменилось,
                # и если нисколько, то работа завершена
                num_changed = num_differences(assignments,
                                              new_assignments)
                if num_changed == 0:
                    return

                # В противном случае оставить новые назначения
                # и вычислить новые средние
                assignments = new_assignments
                self.means = cluster_means(self.k, inputs, assignments)
                t.set_description(f"changed: {num_changed} / {len(inputs)}")
```

Давайте посмотрим, как это работает.

Пример: встречи ИТ-специалистов

В ознаменование роста пользовательской базы социальной сети DataSciencester директор по премированию клиентуры хочет организовать для пользователей, проживающих в одном городе, несколько встреч ИТ-специалистов с пивом, пиццей и фирменными футболками DataSciencester. Вы знаете места проживания всех местных пользователей (рис. 20.1), и директор хотел бы, чтобы вы сами определили места встреч, которые были бы удобными для всех желающих приехать.

В зависимости от того, как смотреть, можно увидеть два или три кластера. (В этом легко убедиться визуально, поскольку данные представлены всего лишь в двух размерностях. В случае большего числа размерностей было бы намного труднее это заметить.)

Для начала представим, что директор имеет бюджет, достаточный для трех встреч. Вы идете к компьютеру и пробуете следующее:

```
random.seed(12)              # Для того чтобы иметь те же самые результаты
clusterer = KMeans(k=3)
clusterer.train(inputs)
means = sorted(clusterer.means)  # Отсортировать для модульного теста

assert len(means) == 3
```

```
# Проверить, что средние находятся близко к тому, что мы ожидаем
assert squared_distance(means[0], [-44, 5]) < 1
assert squared_distance(means[1], [-16, -10]) < 1
assert squared_distance(means[2], [18, 20]) < 1
```

Вы находите три кластера, центрированные в [–44, 5], [–16, –10] и [18, 20], и подбираете места встреч рядом с этими районами (рис. 20.2).

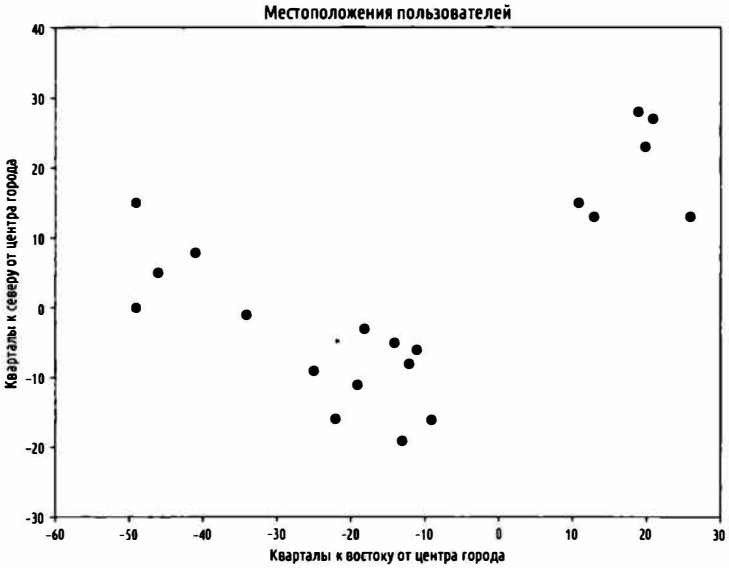


Рис. 20.1. Местоположения пользователей в городе

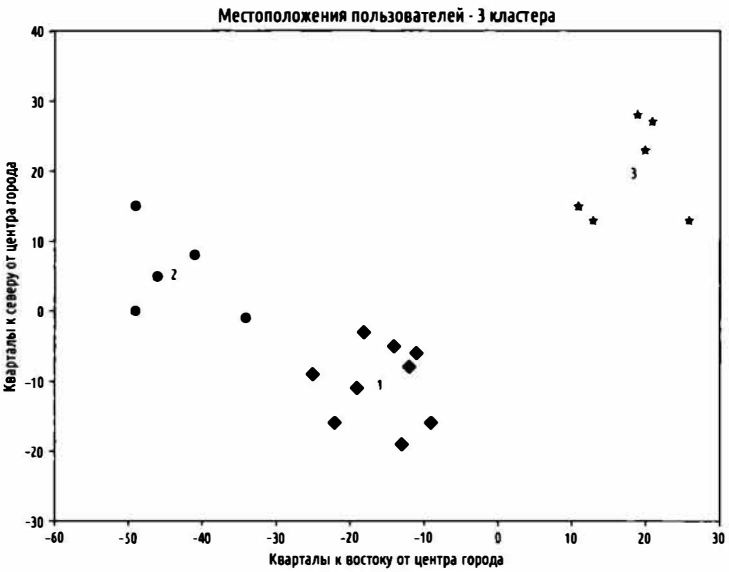


Рис. 20.2. Местоположения пользователей, сгруппированные в три кластера

Вы показываете свои результаты директору, который, правда, сообщает, что бюджета, к сожалению, хватит всего на *две* встречи.

"Нет проблем", — говорите вы.

```
random.seed(0)
clusterer = KMeans(k=2)
clusterer.train(inputs)
means = sorted(clusterer.means)

assert len(means) == 2
assert squared_distance(means[0], [-26, -5]) < 1
assert squared_distance(means[1], [18, 20]) < 1
```

Как показано на рис. 20.3, одна встреча по-прежнему должна быть близка с [18, 20], но теперь другая должна быть рядом с [–26, –5].

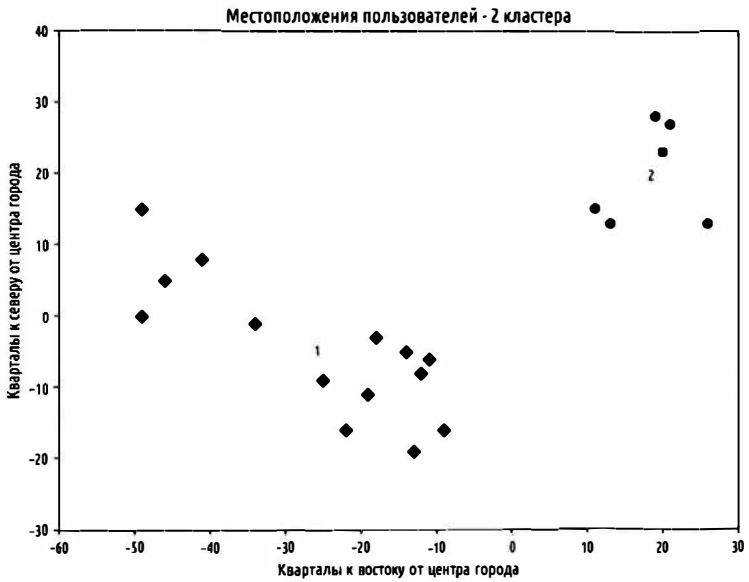


Рис. 20.3. Местоположения пользователей, сгруппированные в два кластера

Выбор числа *k*

В предыдущем примере выбор числа *k* был обусловлен факторами, находящимися вне нашего контроля. В общем случае так не бывает. Существует целый ряд способов выбора числа *k*. Один из самых легких для понимания предусматривает построение графика суммы квадратов ошибок (отклонений между каждой точкой и средним своей группы) как функции от *k* и нахождение места "излома" кривой (рис. 20.4):

```
from matplotlib import pyplot as plt

def squared_clustering_errors(inputs: List[Vector], k: int) -> float:
    """Отыскивает сумму квадратов ошибок, возникающих
       из кластеризации входов k средними"""
    clusterer = KMeans(k)
    clusterer.train(inputs)
    means = clusterer.means
    assignments = [clusterer.classify(input) for input in inputs]

    return sum(squared_distance(input, means[cluster])
               for input, cluster in zip(inputs, assignments))
```

которую можно применить к нашему предыдущему примеру:

```
ks = range(1, len(inputs) + 1)
errors = [squared_clustering_errors(inputs, k) for k in ks]
plt.plot(ks, errors)
plt.xticks(ks)
plt.xlabel("k")
plt.ylabel("Суммарная квадратическая ошибка")
plt.title("Суммарная ошибка против числа кластеров")
plt.show()
```

Рис. 20.4. Выбор числа *k*

Пример: кластеризация цвета

Директор по промоутерским материалам разработал привлекательные фирменные наклейки DataSciencester, которые он хотел бы раздать на дружеских встречах. К сожалению, офисный стикер-принтер может печатать только в пяти цветах, а поскольку директор по искусству находится в творческом отпуске, он интересуется, нельзя ли как-нибудь изменить дизайн наклеек так, чтобы они содержали только пять цветов.

Компьютерные изображения можно представить в виде двумерного массива пикселов, где каждый пиксел сам является трехмерным вектором (красный, зеленый, синий), обозначающим его цвет.

Тогда создание пятицветной версии изображения предусматривает:

- выбор пяти цветов;
- назначение одного из этих цветов каждому пикселу.

Оказывается, что эта задача как нельзя лучше подходит для кластеризации k средними, которая может подразделить пикселы на 5 кластеров в красно-зелено-синем пространстве. Если затем перекрасить пикселы в каждой группе в средний цвет, то задача будет решена.

Для начала нам нужно загрузить изображение в Python. Мы можем это сделать с помощью библиотеки matplotlib, если сперва установим библиотеку pillow:

```
python -m pip install pillow
```

Затем мы можем применить простую инструкцию `matplotlib.image.imread`:

```
image_path = r"girl_with_book.jpg"     # Любое место, где находится
                                       # изображение
import matplotlib.image as mpimg
img = mpimg.imread(image_path) / 256   # Прошкалировать между 0 и 1
```

За кулисами объект `img` является массивом NumPy, но для наших целей мы будем трактовать его как список списков списков.

`img[i][j]` — это пиксел в i-й строке и j-м столбце, причем каждый пиксел — это список `[red, green, blue]` чисел в диапазоне от 0 до 1, которые обозначают цвет этого пиксела[4]:

```
top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel
```

В частности, мы можем получить сглаженный список всех пикселов следующим образом:

```
# .tolist() конвертирует массив NumPy в список Python
pixels = [pixel.tolist() for row in img for pixel in row]
```

[4] См. **https://en.wikipedia.org/wiki/RGB_color_model**.

и затем передать его в наш кластеризатор:

```
clusterer = KMeans(5)
clusterer.train(pixels)  # это может потребовать некоторого времени
```

Когда он завершит работу, мы просто сконструируем новое изображение в том же формате:

```
def recolor(pixel: Vector) -> Vector:
    cluster = clusterer.classify(pixel)  # индекс ближайшего кластера
    return clusterer.means[cluster]      # среднее ближайшего кластера

new_img = [[recolor(pixel) for pixel in row]  # перекрасить эту строку
                                              # пикселов
           for row in img]                 # для каждой строки в изображении
```

и выведем его на экран при помощи `plt.imshow`:

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

Трудно изобразить цветные результаты в черно-белой книге, но рис. 20.5 показывает полутоновые версии полноцветной фотографии и результат после понижения шкалы до пяти цветов.

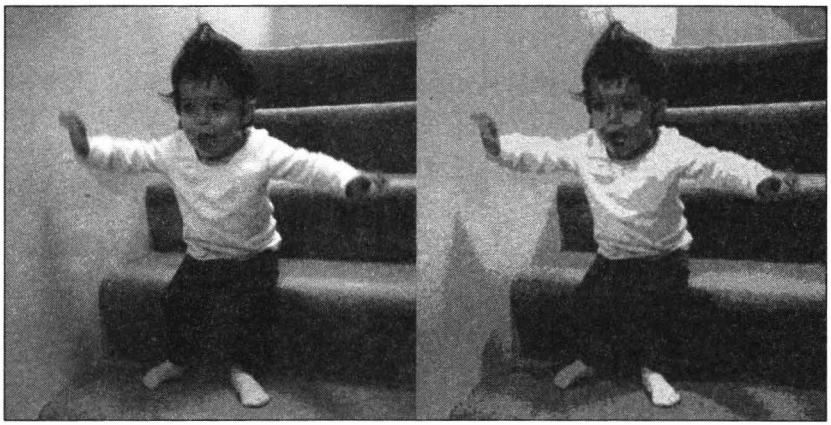


Рис. 20.5. Обесцвеченные оригинальная фотография и ее версия по методу 5 средних

Восходящая иерархическая кластеризация

Альтернативный подход к кластеризации заключается в "выращивании" кластеров снизу вверх. Это делается следующим образом:

1. Сделать каждый вход кластером из одного элемента.
2. До тех пор, пока остается больше одного кластера, отыскать два ближайших и объединить их.

В итоге у нас будет один гигантский кластер, содержащий все входы. Если мы будем отслеживать порядок объединения, то можно воссоздать любое число кластеров путем их разъединения. Например, если требуется три кластера, то надо отменить только последние два объединения.

Мы будем использовать совсем простое представление кластеров. Наши значения будут размещаться в *листовых* кластерах, которые мы представим как типизированные именованные кортежи `NamedTuple`.

```
from typing import NamedTuple, Union

class Leaf(NamedTuple):
    value: Vector

leaf1 = Leaf([10, 20])
leaf2 = Leaf([30, -15])
```

Будем их использовать для выращивания *объединенных* кластеров, которые мы также представим как `NamedTuple`:

```
class Merged(NamedTuple):
    children: tuple
    order: int

merged = Merged((leaf1, leaf2), order=1)

Cluster = Union[Leaf, Merged]
```

Это еще один случай, когда нас подвели аннотации типов Python. Вы хотели бы иметь подсказку типов для `Merged.children` как `Tuple[Cluster, Cluster]`, но библиотека `typing` не позволяет использовать такие рекурсивные типы.

Обсудим порядок объединения чуть позже, а пока создадим вспомогательную функцию, которая рекурсивно возвращает все значения, содержащиеся в (возможно, объединенном) кластере:

```
def get_values(cluster: Cluster) -> List[Vector]:
    if isinstance(cluster, Leaf):
        return [cluster.value]
    else:
        return [value
                for child in cluster.children
                for value in get_values(child)]

assert get_values(merged) == [[10, 20], [30, -15]]
```

Для объединения ближайших кластеров нам нужно некое представление о расстоянии между кластерами. Мы будем использовать *минимальное* расстояние между элементами двух кластеров, при котором объединяются два кластера, ближайших друг к другу (что иногда приводит к большим и не очень плотным цепеобразным

кластерам). Если бы мы хотели получить плотные сферические кластеры, то вместо этого можно воспользоваться *максимальным* расстоянием, поскольку оно объединяет два кластера, которые умещаются в самый маленький клубок. Оба варианта одинаково распространены наравне со *средним* расстоянием:

```
from typing import Callable
from scratch.linear_algebra import distance

def cluster_distance(cluster1: Cluster,
                     cluster2: Cluster,
                     distance_agg: Callable = min) -> float:
    """Вычислить все попарные расстояния между cluster1 и cluster2
       и применить агрегатную функцию _distance_agg_
       к результирующему списку"""
    return distance_agg([distance(v1, v2)
                         for v1 in get_values(cluster1)
                         for v2 in get_values(cluster2)])
```

Мы будем использовать порядковый слот для отслеживания порядка, в котором мы выполняли объединение. Меньшее число будет обозначать *более позднее* объединение. Это означает, что, когда мы хотим разъединить кластеры, мы это делаем от наименьшего значения порядка объединения к наибольшему. Поскольку листовые кластеры `Leaf` не объединяются, то мы назначим им литерал положительной бесконечности, т. е. самое высокое значение из возможных. И поскольку у них нет свойства упорядоченности `.order`, то мы создадим вспомогательную функцию:

```
def get_merge_order(cluster: Cluster) -> float:
    if isinstance(cluster, Leaf):
        return float('inf')   # Ни разу не объединялся
    else:
        return cluster.order
```

Поскольку листовые кластеры `Leaf` не имеют дочерних кластеров, мы создадим и добавим вспомогательную функцию:

```
from typing import Tuple

def get_children(cluster: Cluster):
    if isinstance(cluster, Leaf):
        raise TypeError("Лист не имеет дочерних элементов")
    else:
        return cluster.children
```

Теперь мы готовы создать кластерный алгоритм:

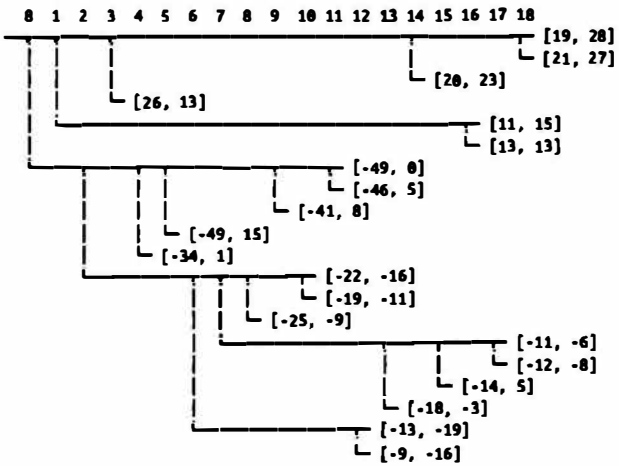
```
def bottom_up_cluster(inputs: List[Vector],
                      distance_agg: Callable = min) -> Cluster:
    # Начать с того, что все элементы являются листьями
    clusters: List[Cluster] = [Leaf(input) for input in inputs]
```

```
    def pair_distance(pair: Tuple[Cluster, Cluster]) -> float:
        return cluster_distance(pair[0], pair[1], distance_agg)

    # До тех пор, пока у нас осталось более одного кластера...
    while len(clusters) > 1:
        # Отыскать два ближайших кластера
        c1, c2 = min(((cluster1, cluster2)
                      for i, cluster1 in enumerate(clusters)
                      for cluster2 in clusters[:i]),
                     key=pair_distance)

        # Удалить их из списка кластеров
        clusters = [c for c in clusters if c != c1 and c != c2]

        # Объединить их, используя merge_order = число оставшихся
        # кластеров
        merged_cluster = Merged((c1, c2), order=len(clusters))

        # И добавить их объединение
        clusters.append(merged_cluster)

    # Когда остался всего один кластер, вернуть его
    return clusters[0]
```

Использовать его очень просто:

```
base_cluster = bottom_up_cluster(inputs)
```

Этот фрагмент кода производит кластеризацию, которая выглядит следующим образом:



Числа сверху показывают "очередность слияния". Поскольку у нас было 20 входов, получение одного кластера потребовало 19 объединений. Первое объединение создало кластер 18 путем совмещения листьев [19, 28] и [21, 27]. И последнее объединение создало кластер 0.

Если бы вам нужны были только два кластера, то вы бы разбили на первой развилке ("0"), создав один кластер с шестью точками и второй с остальными. Для трех кластеров вы бы продолжили до второй развилки ("1"), которая указывает на разбиение этого первого кластера на кластер с ([19, 28], [21, 27], [20, 23], [26, 13]) и кластер с ([11, 15], [13, 13]). И так далее.

Однако, как правило, мы стараемся избегать подобного рода заставляющих щуриться текстовых представлений. (Впрочем, создание удобной в использовании визуализации кластерной иерархии могло бы стать увлекательным упражнением.) Вместо этого давайте напишем функцию, которая генерирует любое число кластеров, выполняя надлежащее число разъединений:

```
def generate_clusters(base_cluster: Cluster,
                      num_clusters: int) -> List[Cluster]:
    # Начать со списка, состоящего только из базового кластера
    clusters = [base_cluster]

    # До тех пор, пока у нас недостаточно кластеров...
    while len(clusters) < num_clusters:

        # Выбрать из кластеров тот, который был объединен последним
        next_cluster = min(clusters, key=get_merge_order)

        # Удалить его из списка
        clusters = [c for c in clusters if c != next_cluster]

        # И добавить его дочерние элементы в список
        # (т. е. разъединить его)
        clusters.extend(get_children(next_cluster))

    # Как только у нас достаточно кластеров...
    return clusters
```

Так, например, если нужно сгенерировать три кластера, то мы можем сделать следующее:

```
three_clusters = [get_values(cluster)
                  for cluster in generate_clusters(base_cluster, 3)]
```

и затем можно легко вывести их на график:

```
for i, cluster, marker, color in zip([1, 2, 3],
                                     three_clusters,
                                     ['D','o','*'],
                                     ['r','g','b']):
    xs, ys = zip(*cluster)  # Волшебный трюк с распаковкой
    plt.scatter(xs, ys, color=color, marker=marker)

    # Установить число на среднем значении кластера
    x, y = vector_mean(cluster)
    plt.plot(x, y, marker='$' + str(i) + '$', color='black')
```

```
plt.title("Места проживания - 3 кластера снизу вверх, min")
plt.xlabel("Кварталы к востоку от центра города")
plt.ylabel("Кварталы к северу от центра города")
plt.show()
```

Этот фрагмент кода даст результаты, сильно отличающиеся от тех, которые были получены на основе *k* средних (рис. 20.6).

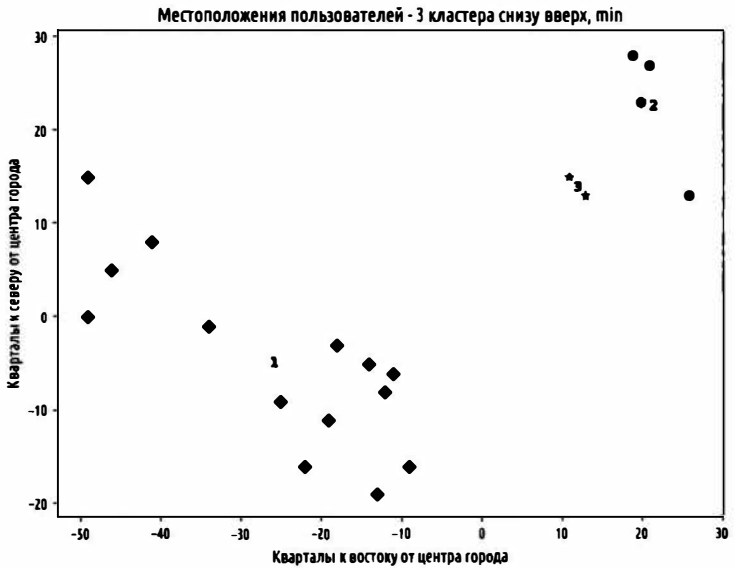


Рис. 20.6. Три кластера по восходящему методу на основе функции `min`

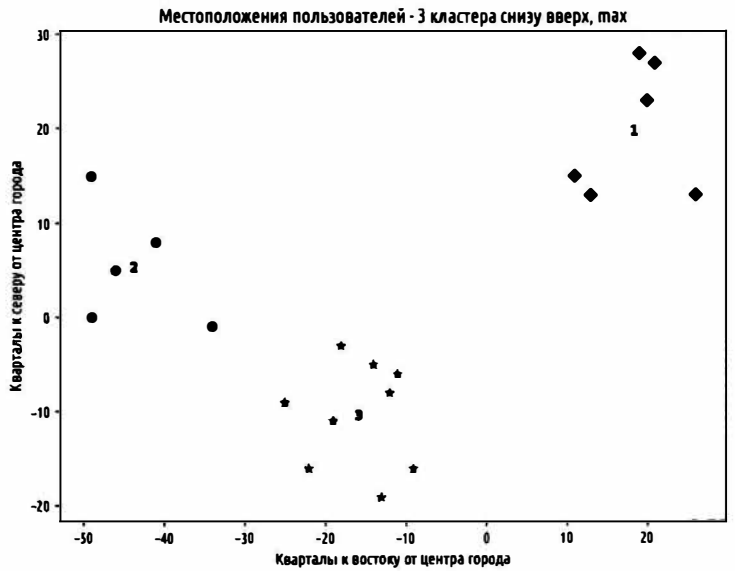


Рис. 20.7. Три кластера по восходящему методу на основе функции `max`

Как уже упоминалось ранее, это связано с тем, что применение функции `min` в функции `cluster_distance` имеет свойство порождать цепеобразные кластеры. Если вместо нее использовать функцию `max` (которая порождает плотные кластеры), то результат будет выглядеть точно так же, как и в случае с 3 средними (рис. 20.7).

Приведенная выше имплементация восходящего метода `bottom_up_clustering` является относительно простой и одновременно крайне неэффективной. В частности, здесь на каждом шаге заново вычисляется расстояние между каждой парой входов. В более эффективной имплементации попарные расстояния будут вычисляться заранее и затем будет выполняться поиск внутри списка, возвращаемого из функции `cluster_distance`. В действительно эффективной имплементации, скорее всего, также будут учитываться расстояния `cluster_distance` предыдущего шага.

Для дальнейшего изучения

- В библиотеке scikit-learn есть модуль `sklearn.cluster`[5], который содержит несколько кластерных алгоритмов, включая алгоритм *k* средних `KMeans` и алгоритм иерархической кластеризации Ворда `Ward` (в котором используется другой критерий объединения кластеров, в отличие от того, который мы рассмотрели в этой главе).
- Библиотека SciPy[6] имеет две кластерных модели: `scipy.cluster.vq` (которая выполяет *k* средних) и `scipy.cluster.hierarchy` (которая имеет целый ряд иерарархических кластерных алгоритмов).

[5] См. **http://scikit-learn.org/stable/modules/clustering.html**.

[6] См. **http://www.scipy.org/**.

ГЛАВА 21
Обработка естественного языка

> Они оба побывали на каком-то словесном пиру и наворовали там объедков.
> – *Уильям Шекспир*[1]

Обработка естественного языка (ЕЯ) имеет отношение к вычислительным техническим решениям с участием языка. Она представляет собой обширнейшую область, однако мы обратимся лишь к нескольким из приемов, простым и не очень.

Облака слов

В *главе 1* мы вычислили количества появлений слов, встречающихся в темах, которые интересуют пользователей. Одим из приемов визуализации слов и их количеств появлений являются облака слов (облака тегов), которые эффектно изображают слова в размерах, пропорциональных их количествам появлений.

В общем-то исследователи данных не высокого мнения об облаках слов во многом потому, что размещение слов не означает ничего, кроме того, что "вот тут есть место, где уместится слово".

Тем не менее, если когда-нибудь вам придется создавать облако слов, подумайте о том, сможете ли вы заставить осевые линии что-либо выразить. Например, представим, что для каждого слова из некоторой коллекции популярных терминов науки о данных имеется два числа между 0 и 100: первое представляет частотность его появлений в объявлениях о вакансиях, второе — частотность его появлений в резюме:

```
data = [("big data",100,15), ("Hadoop",95,25), ("Python",75,50),
        ("R",50,40), ("machine learning",80,20), ("statistics",20,60),
        ("data science",60,70), ("analytics",90,3),
        ("team player",85,85), ("dynamic",2,90), ("synergies",70,0),
        ("actionable insights",40,30), ("think out of the box",45,10),
        ("self-starter",30,50), ("customer focus",65,15),
        ("thought leadership",35,35)]
```

Подход на основе облака слов сводится к размещению на странице слов, набранных броским шрифтом (рис. 21.1).

---

[1] Реплика из комедии "Бесплодные усилия любви" (середина 1590-х гг.) Уильяма Шекспира (1564–1616) — знаменитого английского поэта и драматурга. — *Прим. пер.*

Рис. 21.1. Облако модных слов

Смотрится складно, но не говорит ни о чем. Более интересный подход может заключаться в том, чтобы разбросать слова так, что горизонтальное положение будет указывать на популярность среди объявлений, а вертикальное — на популярность среди резюме, что позволит производить визуализацию, которая будет выражать более глубокий смысл (рис. 21.2):

```
from matplotlib import pyplot as plt

def text_size(total: int) -> float:
    """Равняется 8, если итог равен 0, и 28, если итог равен 200"""
    return 8 + total / 200 * 20
```

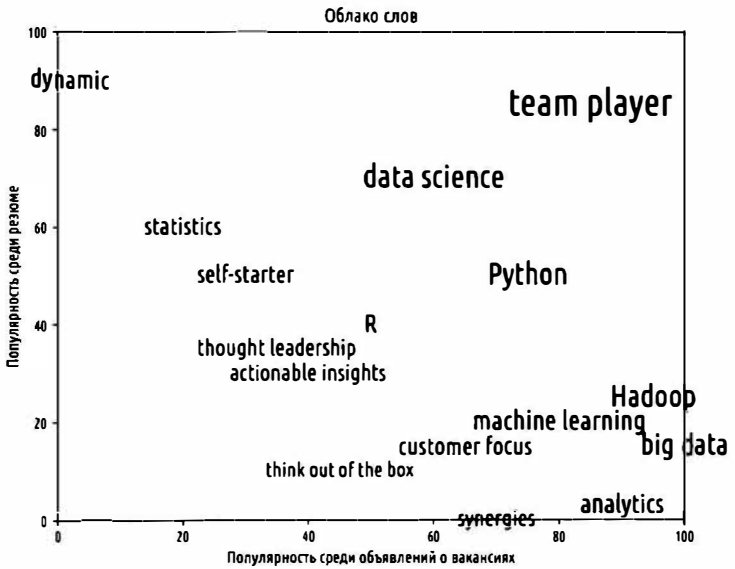


Рис. 21.2. Более осмысленное (и менее привлекательное) облако слов

```
for word, job_popularity, resume_popularity in data:
    plt.text(job_popularity, resume_popularity, word,
             ha='center', va='center',
             size=text_size(job_popularity + resume_popularity))
plt.xlabel("Популярность среди объявлений о вакансиях")
plt.ylabel("Популярность среди резюме")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

N-граммные языковые модели

Директор по поисковому маркетингу хочет создать тысячи веб-страниц, посвященных науке о данных для того, чтобы веб-сайт DataSciencester выше ранжировался в результатах поиска терминов, связанных с наукой о данных. (Вы пытаетесь объяснить, что алгоритмы поисковых систем достаточно умны и что это на самом деле не сработает, но он отказывается слушать.)

Естественно, он не хочет сам писать тысячи веб-страниц, но и не хочет платить орде "контент-стратегов" за выполнение этой работы. Вместо этого он спрашивает у вас, можно ли сгенерировать эти веб-страницы программно. Для этого нам понадобится каким-то образом смоделировать язык.

Один из подходов заключается в том, чтобы начать с корпуса документов и вычислить статистическую модель языка. В нашем случае мы начнем с очерка Майка Лукидеса "Что такое наука о данных?"[2].

Как и в *главе 9*, мы воспользуемся библиотеками requests и BeautifulSoup для извлечения данных. Есть, правда, пара вопросов, на которые стоит обратить внимание.

Во-первых, апострофы в тексте на самом деле представлены символом Юникода `u"\u2019"`. Мы создадим вспомогательную функцию, которая будет менять их на нормальные апострофы:

```
def fix_unicode(text: str) -> str:
    return text.replace(u"\u2019", "'")
```

Во-вторых, получив текст веб-страницы, мы захотим разбить его на последовательность слов и точек (благодаря чему мы сможем распознавать конец предложений). Это можно сделать при помощи метода `re.findall`:

```
import re
from bs4 import BeautifulSoup
import requests

url = "https://www.oreilly.com/ideas/what-is-data-science"
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')
```

[2] См. **https://www.oreilly.com/ideas/what-is-data-science**.

```
content = soup.find("div", "article-body")    # Отыскать div с именем
                                              # article-body
regex = r"[\w']+|[\.]"                        # Сочетает слово и точку

document = []

for paragraph in content("p"):
    words = re.findall(regex, fix_unicode(paragraph.text))
    document.extend(words)
```

Мы, конечно же, могли бы (и, вероятно, должны) продолжить очистку данных. В документе еще осталось некоторое количество постороннего текста (например, первое слово "Section") и несколько разбросанных повсюду заголовков и списков, и мы неправильно выполнили разбивку на точках внутри фраз (например, "Web 2.0"). Несмотря на сказанное, мы продолжим работать с документом как есть.

Теперь, когда у нас есть текст в виде последовательности слов, мы можем смоделировать язык следующим образом: имея некое стартовое слово (скажем, "book"), мы посмотрим на все слова, которые в исходном документе следуют за ним. Мы случайно выберем одно из них в качестве следующего слова и будем повторять процесс до тех пор, пока не получим точку, которая обозначает конец предложения. Такая процедура называется *биграммной моделью*, поскольку она полностью обусловливается частотностями биграмм (пар слов) в исходных данных.

Как быть со стартовым словом? Мы можем выбрать его случайно из тех слов, которые *следуют* за точкой. Для начала давайте предварительно рассчитаем возможные переходы от слова к слову. Вспомните, что функция `zip` останавливается, когда любой из ее входов заканчивается, вследствие чего выражение `zip(document, document[1:])` дает пары элементов, которые в документе `document` следуют строго один за другим:

```
from collections import defaultdict

transitions = defaultdict(list)
for prev, current in zip(document, document[1:]):
    transitions[prev].append(current)
```

Теперь все готово для генерирования предложений:

```
def generate_using_bigrams() -> str:
    current = "."   # Означает, что следующее слово начнет предложение
    result = []
    while True:
    next_word_candidates = transitions[current]    # Биграммы (current, _)
    current = random.choice(next_word_candidates)  # Выбрать одно случайно
    result.append(current)                         # Добавить в result
    if current == ".": return " ".join(result)     # Если ".", то завершаем
```

Предложения, которые она производит, выглядят полной тарабарщиной, но такой, что если разместить их на веб-сайте, то они вполне могут сойти за глубокомысленные высказывания на тему науки о данных. Например:

If you may know which are you want to data sort the data feeds web friend someone on trending topics as the data in Hadoop is the data science requires a book demonstrates why visualizations are but we do massive correlations across many commercial disk drives in Python language and creates more tractable form making connections then use and uses it to solve a data.

Биграммная модель.

Мы можем сделать предложения менее бредовыми, если обратиться к *триграммам*, т. е. триплетам идущих подряд слов. (В более общем случае вы могли бы обратиться к *n-граммам*, состоящим из *n* идущих подряд слов, однако трех для примера будет достаточно.) Теперь переходы будут зависеть от *двух* предыдущих слов:

```
trigram_transitions = defaultdict(list)
starts = []

for prev, current, next in zip(document, document[1:], document[2:]):

    if prev == ".":              # Если предыдущее "слово" было точкой,
        starts.append(current)   # то это стартовое слово

    trigram_transitions[(prev, current)].append(next)
```

Обратите внимание, что теперь мы должны отслеживать стартовые слова раздельно. Мы можем генерировать предложения практически тем же путем:

```
def generate_using_trigrams() -> str:
    current = random.choice(starts)  # Выбрать случайное стартовое слово
    prev = "."                       # и предварить его символом '.'
    result = [current]
    while True:
        next_word_candidates = trigram_transitions[(prev, current)]
        next_word = random.choice(next_word_candidates)

        prev, current = current, next_word
        result.append(current)

        if current == ".":
            return " ".join(result)
```

Эта функция производит более осмысленные предложения, например:

In hindsight MapReduce seems like an epidemic and if so does that give us new insights into how economies work That's not a question we could even have asked a few years there has been instrumented.

Триграммная модель.

Естественно, они выглядят лучше, потому что на каждом шаге у процесса генерирования остается меньше слов на выбор, а для многих шагов — всего один. Следовательно, модель часто будет генерировать предложения (или по крайней мере длинные словосочетания), которые существовали в исходных данных дословно.

Имея больше данных, мы могли бы сделать модель лучше; она также работала бы лучше, если бы мы собрали *n*-граммы из многочисленных очерков, посвященных науке о данных.

Грамматики

Другой способ моделирования языка основан на *грамматиках*, т. е. правилах генерирования допустимых предложений. Из начальной школы мы знаем о частях речи и о том, как они сочетаются. Например, если у вас был отвратительный учитель родного языка, то вы могли бы утверждать, что предложение обязательно должно состоять из *существительного*, за которым следует *глагол*. Тогда, если у вас есть список существительных и глаголов, то вы можете генерировать предложения согласно этому правилу.

Мы определим грамматику чуть посложнее:

```
from typing import List, Dict

# Псевдоним типов для ссылки на грамматики позже
Grammar = Dict[str, List[str]]

grammar = {
    "_S"  : ["_NP _VP"],
    "_NP" : ["_N",
             "_A _NP _P _A _N"],
    "_VP" : ["_V",
             "_V _NP"],
    "_N"  : ["data science", "Python", "regression"],
    "_A"  : ["big", "linear", "logistic"],
    "_P"  : ["about", "near"],
    "_V"  : ["learns", "trains", "tests", "is"]
}
```

Условимся, что имена, начинающиеся с символа подчеркивания, обозначают правила (нетерминальные лексемы), которые нуждаются в дальнейшем расширении, и что другие имена являются *терминальными лексемами*, которые не нуждаются в дальнейшей обработке.

Так, например, `"_S"` — это нетерминальная лексема, описывающая правило подстановки для "предложения", которое порождает `"_NP"` ("группу существительного"), за которой следует `"_VP"` ("группа глагола").

Нетерминальная лексема группы глагола `"_VP"` может порождать нетерминальную лексему глагола `"_V"` ("глагол") либо нетерминальную лексему глагола, за которой идет нетерминальная лексема группы существительного.

Обратите внимание, что нетерминальная лексема группы существительного `"_NP"` содержит саму себя в правой части одного из своих правил подстановки. Грамматики могут быть рекурсивными, что позволяет даже таким грамматикам с конеч-

ным числом состояний, как эта, генерировать бесконечное число разных предложений.

Как генерировать предложения на основе такой грамматики? Мы начнем со списка, содержащего лексему предложения "_S", и затем будем неоднократно расширять каждую лексему, подставляя вместо нее правую часть одного из правил подстановки, выбираемого случайно. Мы прекращаем обработку, когда у нас будет список, состоящий исключительно из терминалов.

Например, одна из таких последовательностей может выглядеть следующим образом:

```
['_S']
['_NP','_VP']
['_N','_VP']
['Python','_VP']
['Python','_V','_NP']
['Python','trains','_NP']
['Python','trains','_A','_NP','_P','_A','_N']
['Python','trains','logistic','_NP','_P','_A','_N']
['Python','trains','logistic','_N','_P','_A','_N']
['Python','trains','logistic','data science','_P','_A','_N']
['Python','trains','logistic','data science','about','_A', '_N']
['Python','trains','logistic','data science','about','logistic','_N']
['Python','trains','logistic','data science','about','logistic','Python']
```

Как это имплементировать? Для начала мы создадим простую вспомогательную функцию для выявления терминалов:

```
def is_terminal(token: str) -> bool:
    return token[0] != "_"
```

Далее нам нужно написать функцию, которая будет превращать список лексем в предложение. Мы будем искать первую нетерминальную лексему. Если такая отсутствует, то это значит, что мы имеем законченное предложение, и обработка прекращается.

Если же мы находим нетерминальную лексему, то мы случайно выбираем для нее одно из правил подстановки. Если в результате подстановки получится терминал (т. е. слово), то мы просто подставляем его вместо лексемы. В противном случае мы имеем дело с последовательностью разделенных пробелами нетерминальных лексем, которые нам нужно расширить, т. е. преобразовать в список (методом `split`), а затем присоединить к существующим лексемам. В любом случае мы повторяем этот процесс на новом набороме лексем.

Собрав все вместе, получим:

```
def expand(grammar: Grammar, tokens: List[str]) -> List[str]:
    for i, token in enumerate(tokens):
        # Если это терминальная лексема, то пропустить ее
        if is_terminal(token): continue
```

```
            # В противном случае это нетерминальная лексема,
            # поэтому нам нужно случайно выбрать подстановку
            replacement = random.choice(grammar[token])

            if is_terminal(replacement):
                tokens[i] = replacement
            else:
                # Подстановкой могло бы быть, например, "_NP _VP", поэтому нам
                # нужно разбить ее по пробелам и присоединить к списку
                tokens = tokens[:i] + replacement.split() + tokens[(i+1):]

            # Теперь вызвать expand с новым списком лексем.
            return expand(grammar, tokens)

    # Если мы тут, значит, у нас были терминалы. Закончить обработку
    return tokens
```

И теперь мы можем начать генерировать предложения:

```
def generate_sentence(grammar: Grammar) -> List[str]:
    return expand(grammar, ["_S"])
```

Попробуйте изменить грамматику — добавив больше слов, больше правил, другие части речи — до тех пор, пока вы не будете готовы генерировать столько веб-страниц, сколько потребуется для вашей компании.

Грамматики на самом деле становятся интереснее, когда они используются в обратном направлении. С учетом предложения мы можем применить грамматику для разбора предложения. Тогда это позволяет нам выявлять подлежащие и сказуемые и вычленять смысл предложения.

Уметь применять науку о данных для генерирования текста — это, конечно, классная вещь; однако ее применение для *понимания* смысла текста сродни волшебству. (*См. разд. "Для дальнейшего изучения" главы 16*, где указаны библиотеки, которыми можно воспользоваться для этих целей.)

Ремарка: генерирование выборок по Гиббсу

Генерировать выборки из некоторых распределений достаточно просто. Мы можем получить равномерные случайные величины с помощью:

```
random.random()
```

и нормальные случайные величины с помощью:

```
inverse_normal_cdf(random.random())
```

Но из некоторых выборок выполнять отбор труднее. *Генерирование выборок по Гиббсу* — это техническое решение для извлечения выборок из многомерных распределений, когда известны лишь некоторые условные распределения.

Например, представим бросание двух игральных кубиков. Пусть x будет значением первого кубика, а y — сумма кубиков. Допустим, нужно сгенерировать большое число пар (x, y). В этом случае легко сгенерировать выборки непосредственно:

```
from typing import Tuple
import random

def roll_a_die() -> int:
    return random.choice([1, 2, 3, 4, 5, 6])

def direct_sample() -> Tuple[int, int]:
    d1 = roll_a_die()
    d2 = roll_a_die()
    return d1, d1 + d2
```

Но, предположим, что известны только условные распределения. Распределение y в зависимости от x получить легко — если известно значение x, то y равновероятно будет $x+1$, $x+2$, $x+3$, $x+4$, $x+5$ или $x+6$:

```
def random_y_given_x(x: int) -> int:
    """Равновероятно составляет x + 1, x + 2, ... , x + 6"""
    return x + roll_a_die()
```

В обратную сторону уже сложнее. Например, если известно, что $y=2$, тогда неизбежно $x=1$ (поскольку единственный случай, когда сумма граней двух кубиков равна 2, только если обе равны 1). Если известно, что $y=3$, то x равновероятно составит 1 или 2. Схожим образом, если $y=11$, то x должно равняться 5 или 6:

```
def random_x_given_y(y: int) -> int:
    if y <= 7:
        # Если сумма <= 7, то первый кубик равновероятно будет
        # 1, 2, ..., (сумма - 1)
        return random.randrange(1, y)
    else:
        # Если сумма > 7, то первый кубик равновероятно будет
        # (сумма - 6), (сумма - 5), ..., 6
        return random.randrange(y - 6, 7)
```

Генерирование выборок по Гиббсу работает так: мы начинаем с любых (допустимых) значений x и y и затем многократно чередуем, подставляя вместо x случайное значение, выбранное в зависимости от y, и подставляя вместо y случайное значение, выбранное в зависимости от x. После ряда итераций полученные значения x и y будут представлять выборку из безусловного совместного распределения:

```
# Выборка по Гиббсу
def gibbs_sample(num_iters: int = 100) -> Tuple[int, int]:
    x, y = 1, 2 # doesn't really matter
    for _ in range(num_iters):
        x = random_x_given_y(y)
        y = random_y_given_x(x)
    return x, y
```

Вы можете проверить, что эта функция дает результаты, аналогичные непосредственной выборке:

```
# Сравнить распределения
def compare_distributions(num_samples: int = 1000) -> Dict[int, List[int]]:
    counts = defaultdict(lambda: [0, 0])
    for _ in range(num_samples):
        counts[gibbs_sample()][0] += 1
        counts[direct_sample()][1] += 1
    return counts
```

Мы будем использовать это техническое решение в следующем разделе.

Тематическое моделирование

Когда мы строили рекомендательную подсистему "Исследователи данных, которых вы должны знать" в *главе 1*, мы искали точные совпадения с темами, которыми пользователи интересуются.

Более изощренный поход к пониманию интересов пользователей заключается в попытке выявить более общие *тематики*, которые лежат в основе интересов пользователей. Для определения общих тематик в наборе документов обычно применяется *латентное размещение Дирихле*. Мы применим его к документам, состоящим из тем, интересующих каждого пользователя.

Латентное размещение Дирихле имеет некоторое сходство с наивным байесовым классификатором, разработанным в *главе 13*, в том, что в нем исходят из вероятностной модели документов. Мы опустим сложные математические подробности, но для наших целей эта модель исходит из того, что:

- существует некоторое фиксированное число K общих тематик;
- существует случайная величина, которая назначает каждой тематике ассоциированное с ней распределение вероятностей над множеством слов. Вы должны думать об этом распределении как о вероятности наблюдать слово w в заданной тематике k;
- существует еще одна случайная величина, которая назначает каждому документу распределение вероятностей над множеством тематик. Вы должны думать об этом распределении как о смеси тематик в документе d;
- каждое слово в документе было сгенерировано сначала путем случайного выбора тематики (из распределения тематик документа) и затем случайного выбора слова (из распределения слов тематики).

В частности, у нас есть коллекция документов `documents`, каждый из которых является списком слов. И у нас есть соответствующая коллекция тематик `document_topics`, которая назначает тематику (здесь число между 0 и $K-1$) каждому слову в каждом документе.

Так, пятое слово в четвертом документе равно:

```
documents[3][4]
```

а тематика, из которой это слово было выбрано, равна:

```
document_topics[3][4]
```

Это очень явно определяет распределение каждого документа над множеством тематик и неявно определяет распределение каждой тематики над множеством слов.

Мы можем оценить правдоподобие того, что тематика 1 производит определенное слово, сравнивая частотность, с которой тематика 1 производит это слово, с частотностью, с которой она производит *любое* слово. (Схожим образом, когда мы строили спам-фильтр в *главе 13*, мы сравнивали частотность появления каждого слова в спамном сообщении с суммарным числом слов, появляющихся в спамных сообщениях.)

Хотя эти тематики являются просто числами, мы можем дать им описательные имена, посмотрев на слова, которым они придают самый тяжелый вес. Нам осталось лишь сгенерировать список тематик `document_topics`. Именно тут в игру вступает генерирование выборок по Гиббсу.

Начнем с того, что абсолютно случайно назначим каждому слову в каждом документе тематику. Теперь мы пройдем по каждому документу слово за словом. Для этого слова и документа мы сконструируем веса каждой тематики, которые зависят от (текущего) распределения тематик в этом документе и (текущего) распределения слов в этой тематике. Затем мы применим эти веса при отборе новой тематики для этого слова. Если мы повторим этот процесс многократно, то в итоге получим совместную выборку из тематико-словарного распределения ("тематика-слово") и документно-тематического распределения ("документ-тематика").

Для начала нам потребуется функция, которая будет случайно выбирать индекс, основываясь на произвольном множестве весов:

```
def sample_from(weights: List[float]) -> int:
    """Возвращает i с вероятностью weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random()  # Равномерно между 0 и суммой
    for i, w in enumerate(weights):
        rnd -= w                   # Вернуть наименьший i, такой, что
        if rnd <= 0: return i      # weights[0] + ... + weights[i] >= rnd
```

Например, если передать ей веса `[1, 1, 3]`, то одну пятую случаев она будет возвращать 0, одну пятую — 1 и три пятых — 2. Давайте напишем тест:

```
from collections import Counter

# Извлечь 1000 раз и подсчитать
draws = Counter(sample_from([0.1, 0.1, 0.8]) for _ in range(1000))
assert 10 < draws[0] < 190   # Должно быть ~10%, это очень нестрогий тест
assert 10 < draws[1] < 190   # Должно быть ~10%, это очень нестрогий тест
assert 650 < draws[2] < 950  # Должно быть ~80%, это очень нестрогий тест
assert draws[0] + draws[1] + draws[2] == 1000
```

Наши документы представлены интересами пользователей. Эти документы имеют следующий вид:

```
documents = [
  ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
  ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
  ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
  ["R", "Python", "statistics", "regression", "probability"],
  ["machine learning", "regression", "decision trees", "libsvm"],
  ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
  ["statistics", "probability", "mathematics", "theory"],
  ["machine learning", "scikit-learn", "Mahout", "neural networks"],
  ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
  ["Hadoop", "Java", "MapReduce", "Big Data"],
  ["statistics", "R", "statsmodels"],
  ["C++", "deep learning", "artificial intelligence", "probability"],
  ["pandas", "R", "Python"],
  ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
  ["libsvm", "regression", "support vector machines"]
]
```

И мы попробуем отыскать тематики при:

```
K = 4
```

Для расчета весов выборки нам нужно отслеживать несколько количественных показателей. Давайте сначала создадим для них структуры данных.

Сколько раз каждая тематика назначается каждому документу:

```
# Список объектов Counter, один для каждого документа
document_topic_counts = [Counter() for _ in documents]
```

Сколько раз каждое слово назначается каждой тематике:

```
# Список объектов Counter, один для каждой тематики
topic_word_counts = [Counter() for _ in range(K)]
```

Суммарное число слов, назначенное каждой тематике:

```
# Список чисел, один для каждой тематики
topic_counts = [0 for _ in range(K)]
```

Суммарное число слов, содержащихся в каждом документе:

```
# Список чисел, один для каждого документа
document_lengths = [len(document) for document in documents]
```

Число различных слов:

```
distinct_words = set(word for document in documents for word in document)
W = len(distinct_words)
```

И число документов:

```
D = len(documents)
```

После того как они будут заполнены, мы можем, например, отыскать число слов в документе `documents[3]`, ассоциированных с тематикой 1, следующим образом:

```
document_topic_counts[3][1]
```

И мы можем отыскать, сколько раз слово *nlp* ассоциируется с тематикой 2, так:

```
topic_word_counts[2]["nlp"]
```

Теперь все готово для определения функций условной вероятности. Как и в *главе 13*, каждая из них имеет сглаживающий член, который обеспечивает, чтобы каждая тематика имела ненулевой шанс быть выбранной в любом документе и чтобы каждое слово имело ненулевой шанс быть выбранным в любой тематике:

```
# Вероятность тематики в зависимости от документа
def p_topic_given_document(topic: int, d: int,
                           alpha: float = 0.1) -> float:
    """Доля слов в документе 'd', которые назначаются
       тематике 'topic' (плюс некоторое сглаживание)"""

    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

# Вероятность слова в зависимости от тематики
def p_word_given_topic(word: str, topic: int,
                       beta: float = 0.1) -> float:
    """Доля слов, назначаемых тематике 'topic', которые
       равны 'word' (плюс некоторое сглаживание)"""

    return ((topic_word_counts[topic][word] + beta) /
            (topic_counts[topic] + W * beta))
```

Мы создадим веса для обновления тематик с использованием этих функций:

```
# Вес тематики
def topic_weight(d: int, word: str, k: int) -> float:
    """С учетом документа и слова в этом документе,
       вернуть вес k-й тематики"""
    return p_word_given_topic(word, k) * p_topic_given_document(k, d)

# Выбрать новую тематику
def choose_new_topic(d: int, word: str) -> int:
    return sample_from([topic_weight(d, word, k)
                        for k in range(K)])
```

Имеются веские математические причины, почему функция `topic_weight` определена именно таким образом, однако их детализация увела бы далеко вглубь специализированной области. Надеюсь, по крайней мере, интуитивно понятно, что с учетом данного слова и его документа правдоподобие выбора любой тематики зависит

одновременно от того, насколько правдоподобна эта тематика для документа и насколько правдоподобно это слово для этой тематики.

Вот и весь необходимый функционал. Мы начнем с того, что назначим каждому слову случайную тематику и заполним наши количественные показатели соответствующим образом:

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
                   for document in documents]

for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

Наша цель состоит в том, чтобы получить совместную выборку из тематико-словарного распределения ("тематика-слово") и документно-тематического распределения ("документ-тематика"). Мы достигнем этого, задействуя форму генерирования выборок по Гиббсу, в которой используются ранее определенные условные вероятности:

```
import tqdm

for iter in tqdm.trange(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):
            # Удалить это слово/тематику из показателя,
            # чтобы оно не влияло на веса
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1

            # Выбрать новую тематику, основываясь на весах
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic

            # И добавить его назад в показатель
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

Что собой представляют тематики? Это просто цифры 0, 1, 2 и 3. Если нам понадобятся для них имена, то мы должны сделать это сами. Давайте посмотрим на пять наиболее тяжеловесных слов по каждой (табл. 21.1):

```
for k, word_counts in enumerate(topic_word_counts):
    for word, count in word_counts.most_common():
        if count > 0:
            print(k, word, count)
```

***Таблица 21.1.** Наиболее распространенные слова по тематике*

Тематика 0	Тематика 1	Тематика 2	Тематика 3
Java	R	HBase	regression
Big Data	statistics	Postgres	libsvm
Hadoop	Python	MongoDB	scikit-learn
deep learning	probability	Cassandra	machine learning
artificial intelligence	pandas	NoSQL	neural networks

На их основе я бы, вероятно, назначил тематические имена:

```
topic_names = ["Большие данные и языки программирования",
               "Python и статистика",
               "Базы данных",
               "Машинное обучение"]
```

И в этом месте мы можем посмотреть, каким образом модель назначает тематики интересам пользователей:

```
for document, topic_counts in zip(documents, document_topic_counts):
    print(document)
    for topic, count in topic_counts.most_common():
        if count > 0:
            print(topic_names[topic], count)
    print()
```

В результате получится:

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Большие данные и языки программирования 4 Базы данных 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
Базы данных 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python и статистика 5 Машинное обучение 1
```

И так далее. С учетом союзов "и", которые были нужны в некоторых названиях тематик, возможно, нам следует использовать больше тематик, хотя, весьма вероятно, у нас недостаточно данных для того, чтобы успешно их усвоить.

Векторы слов

Многие последние достижения в обработке ЕЯ связаны с глубоким обучением. В остальной части этой главы мы рассмотрим пару из них, используя механизм, разработанный нами в *главе 19*.

Одним из важных нововведений является представление слов в виде низкоразмерных векторов[3]. Эти векторы можно сравнивать, складывать между собой, вводить в автоматически обучающиеся модели или делать с ними что угодно. Они обычно имеют хорошие свойства, например, схожие слова тяготеют к тому, что они имеют схожие векторы. То есть, как правило, вектор для слова "большой" находится довольно близко к вектору слова "крупный", вследствие чего модель, работающая на векторах слов, может (до некоторой степени) задаром обрабатывать такие вещи, как синонимия.

Часто векторы также будут проявлять восхитительные арифметические свойства. Например, в некоторых таких моделях, если вы возьмете вектор слова "король", вычтите вектор слова "мужчина" и добавите вектор слова "женщина", то в итоге получите вектор, очень близкий к вектору слова "королева". Наверно, будет интересно поразмыслить о том, что это значит в смысле того, что́ векторы слов на самом деле "усваивают", хотя мы не будем здесь тратить на это время.

Создание таких векторов для большого лексикона является трудным делом, поэтому обычно мы будем *усваивать* их из корпуса текста. Существует несколько разных схем, но на высоком уровне задача, как правило, выглядит следующим образом:

1. Получить фрагмент текста.
2. Создать набор данных, где цель состоит в том, чтобы предсказать слово с учетом близлежащих слов (или, как вариант, предсказать близлежащие слова с учетом заданного слова).
3. Натренировать нейронную сеть хорошо справляться с этой задачей.
4. Взять внутренние состояния натренированной нейронной сети как векторы слов.

В частности, поскольку задача состоит в том, чтобы предсказать слово с учетом близлежащих слов, слова, которые встречаются в схожих контекстах (и, следовательно, имеют схожие близлежащие слова), должны иметь аналогичные внутренние состояния и, следовательно, аналогичные векторы слов.

Здесь мы будем измерять "сходство", используя *косинусное сходство* (т. е. числом между −1 и 1), служащее мерой степени, в которой два вектора указывают в одном направлении:

```
from scratch.linear_algebra import dot, Vector
import math

def cosine_similarity(v1: Vector, v2: Vector) -> float:
    return dot(v1, v2) / math.sqrt(dot(v1, v1) * dot(v2, v2))
```

[3] Идея состоит в том, чтобы создать плотный вектор для каждого слова, выбранный таким образом, чтобы он был похож на векторы слов, которые появляются в схожем контексте. Данное техническое решение называется векторным вложением слов (word embedding), или вложением слов в векторное пространство, т. е. это векторное представление слова как часть распределенного представления текста в n-мерном пространстве. См. **https://hackernoon.com/word-embeddings-in-nlp-and-its-applications-fab15eaf7430**. — *Прим. пер.*

```
assert cosine_similarity([1., 1, 1], [2., 2, 2]) == 1, "то же направление"
assert cosine_similarity([-1., -1], [2., 2]) == -1, "противоположное направление"
assert cosine_similarity([1., 0], [0., 1]) == 0, "ортогональное"
```

Давайте усвоим несколько векторов слов, чтобы увидеть, как это работает.

Для начала нам понадобится набор игрушечных данных. Широко используемые векторы слов, как правило, выводятся в результате тренировки на миллионах или даже миллиардах слов. Поскольку наша игрушечная библиотека не способна справиться с таким количеством данных, мы создадим искусственный набор данных с некоторой структурой:

```
colors = ["red", "green", "blue", "yellow", "black", ""] # цвета
nouns = ["bed", "car", "boat", "cat"]                    # существительные
verbs = ["is", "was", "seems"]                           # глаголы
adverbs = ["very", "quite", "extremely", ""]             # наречия
adjectives = ["slow", "fast", "soft", "hard"]            # прилагательные

# Составить предложение
def make_sentence() -> str:
    return " ".join([
        "The",
        random.choice(colors),
        random.choice(nouns),
        random.choice(verbs),
        random.choice(adverbs),
        random.choice(adjectives),
        "."
    ])

NUM_SENTENCES = 50   # Число предложений

random.seed(0)
sentences = [make_sentence() for _ in range(NUM_SENTENCES)]
```

Этот фрагмент кода сгенерирует много предложений с похожей структурой, но с разными словами, например: "The green boat seems quite slow" (Зеленая лодка кажется довольно медленной). Учитывая эту настройку, цвета будут в основном отображаться в "похожих" контекстах, как и существительные, и т. д. Поэтому, если мы хорошо справимся с назначением векторов слов, цвета должны получать похожие векторы и т. д.

На практике у вас, вероятно, будет корпус из миллионов предложений, и в этом случае вы получите "достаточный" контекст прямо из предложений в том виде, в каком они есть. Здесь, имея всего 50 предложений, мы должны сделать их несколько искусственными.

Как упоминалось ранее, нам потребуется применить кодировку слов с одним активным состоянием, т. е. нам нужно будет преобразовать их в идентификаторы. Мы

введем класс `Vocabulary` **для лексикона, который будет отслеживать это отображение:**

```
from scratch.deep_learning import Tensor

class Vocabulary:
    def __init__(self, words: List[str] = None) -> None:
        self.w2i: Dict[str, int] = {}  # Отображение word -> word_id
        self.i2w: Dict[int, str] = {}  # Отображение word_id -> word

        for word in (words or []):     # Если слова были предоставлены,
            self.add(word)             # то добавить их

    @property
    def size(self) -> int:
        """Сколько слов в лексиконе"""
        return len(self.w2i)

    def add(self, word: str) -> None:
        if word not in self.w2i:       # Если слово является для нас новым:
            word_id = len(self.w2i)    # то отыскать следующий id
            self.w2i[word] = word_id   # Добавить в отображение word -> word_id
            self.i2w[word_id] = word   # Добавить в отображение word_id -> word

    def get_id(self, word: str) -> int:
        """Вернуть id слова (либо None)"""
        return self.w2i.get(word)

    def get_word(self, word_id: int) -> str:
        """Вернуть слово с заданным id (либо None)"""
        return self.i2w.get(word_id)

    def one_hot_encode(self, word: str) -> Tensor:
        word_id = self.get_id(word)
        assert word_id is not None, f"неизвестное слово {word}"

        return [1.0 if i == word_id else 0.0 for i in range(self.size)]
```

Все эти вещи мы могли бы сделать вручную, но это удобно иметь в классе. Вероятно, мы должны протестировать его:

```
vocab = Vocabulary(["a", "b", "c"])
assert vocab.size == 3, "в лексиконе есть 3 слова"
assert vocab.get_id("b") == 1, "b должно иметь word_id, равный 1"
assert vocab.one_hot_encode("b") == [0, 1, 0]
assert vocab.get_id("z") is None, "z отсутствует в лексиконе"
assert vocab.get_word(2) == "c", "word_id, равный 2, должно соответствовать c"
vocab.add("z")
```

```
assert vocab.size == 4, "теперь в лексиконе есть 4 слова"
assert vocab.get_id("z") == 3, "теперь z должно иметь id, равный 3"
assert vocab.one_hot_encode("z") == [0, 0, 0, 1]
```

Мы также должны написать простые вспомогательные функции сохранения и загрузки словаря, как и для наших глубоких автоматически обучающихся моделей:

```
import json

def save_vocab(vocab: Vocabulary, filename: str) -> None:
    with open(filename, 'w') as f:
        json.dump(vocab.w2i, f)  # Нужно сохранить только w2i

def load_vocab(filename: str) -> Vocabulary:
    vocab = Vocabulary()
    with open(filename) as f:
        # Загрузить w2i и сгенерировать из него i2w
        vocab.w2i = json.load(f)
        vocab.i2w = {id: word for word, id in vocab.w2i.items()}
    return vocab
```

Мы будем использовать модель векторов слов под названием Skip-Gram[4], которая на входе принимает слово и генерирует вероятности слов, которые могут встречаться рядом с ним. Мы будем подавать в нее тренировочные пары (слово, близлежащее_слово) и постараемся минимизировать перекрестно-энтропийную потерю `SoftmaxCrossEntropy`.

Еще одна распространенная модель, непрерывный мешок слов (continuous bag-of-words, CBOW), принимает на входе близлежащие слова и пытается предсказать исходное слово[5].

Давайте создадим нашу нейронную сеть. В ее основе будет слой вложения, который на входе принимает идентификатор слова и возвращает вектор слова. За кулисами мы можем использовать для этого простую просмотровую таблицу.

Затем мы передадим векторы слов в линейный слой `Linear` с тем же числом выходов, что и слова в нашем лексиконе. Как и раньше, мы будем использовать функцию `softmax` для конвертирования этих выходов в вероятности над близлежащими словами. Поскольку для тренировки модели мы используем градиентный спуск, мы будем обновлять векторы в просмотровой таблице. По завершении тренировки эта просмотровая таблица даст нам векторы слов.

---

[4] Модель Skip-Gram (SG) использует целевое слово для предсказания слов, отобранных из контекста. Она хорошо работает с малыми наборами данных и находит хорошие представления даже для редких слов или фраз. — *Прим. пер.*

[5] Модель непрерывного мешка слов (CBOW) предсказывает целевое слово, используя на входе среднее значение векторов контекстных слов, вследствие чего их порядок не имеет значения. Модель CBOW тренируется быстрее и демонстрирует тенденцию быть немного точнее для частых терминов, но уделяет меньше внимания нечастым словам. — *Прим. пер.*

Давайте создадим слой вложения. На практике мы, возможно, захотим вкладывать элементы, отличные от слов, поэтому мы построим более общий слой вложения `Embedding`. (Позже мы напишем подкласс `TextEmbedding`, который будет специально предназначен для векторов слов.)

В его конструкторе мы предоставим число и размерность наших векторов вложения, вследствие чего он может создавать вложения (которые изначально будут стандартными нормальными случайными величинами):

```
from typing import Iterable
from scratch.deep_learning import Layer, Tensor, random_tensor, zeros_like

class Embedding(Layer):
    def __init__(self, num_embeddings: int, embedding_dim: int) -> None:
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim

        # Один вектор размера embedding_dim для каждого желаемого вложения
        self.embeddings = random_tensor(num_embeddings, embedding_dim)
        self.grad = zeros_like(self.embeddings)

        # Сохранить id последнего входа
        self.last_input_id = None
```

В нашем случае мы будем вставлять всего одно слово за раз. Однако в других моделях мы можем захотеть вкладывать последовательность слов и получать последовательность векторов слов. (Например, если мы хотим натренировать описанную ранее модель CBOW.) И поэтому альтернативная конструкция будет принимать последовательности идентификаторов слов. Мы будем придерживаться одного слова, не усложняя вещи.

```
def forward(self, input_id: int) -> Tensor:
    """Просто выбрать вектор вложения, соответствующий входному id"""
    self.input_id = input_id    # запомнить для использования
                                # в обратном распространении
    return self.embeddings[input_id]
```

Для обратного прохождения мы получим градиент, соответствующий выбранному вектору вложения, и нам нужно будет построить соответствующий градиент для вложений `self.embeddings`, который равен нулю для каждого вложения, отличного от выбранного:

```
def backward(self, gradient: Tensor) -> None:
    # Обнулить градиент, соответствующий последнему входу.
    # Это гораздо дешевле, чем всякий раз создавать новый тензор нулей
    if self.last_input_id is not None:
        zero_row = [0 for _ in range(self.embedding_dim)]
        self.grad[self.last_input_id] = zero_row

    self.last_input_id = self.input_id
    self.grad[self.input_id] = gradient
```

Поскольку у нас есть параметры и градиенты, нам нужно переопределить эти методы:

```
def params(self) -> Iterable[Tensor]:
    return [self.embeddings]

def grads(self) -> Iterable[Tensor]:
    return [self.grad]
```

Как упоминалось ранее, нам понадобится подкласс специально для векторов слов. В таком случае наше число вложений определяется нашим лексиконом, поэтому давайте просто передадим его вместо этого:

```
class TextEmbedding(Embedding):
    def __init__(self, vocab: Vocabulary, embedding_dim: int) -> None:
        # Вызвать конструктор суперкласса
        super().__init__(vocab.size, embedding_dim)

        # И зафиксировать лексикон
        self.vocab = vocab
```

Другие встроенные методы будут работать как есть, но мы добавим еще несколько более специфических для работы с текстом. Например, мы хотели бы иметь возможность извлекать вектор для заданного слова. (Это не является частью интерфейса класса `Layer`, но мы всегда можем добавлять дополнительные методы к конкретным слоям, как нам нравится.)

```
def __getitem__(self, word: str) -> Tensor:
    word_id = self.vocab.get_id(word)
    if word_id is not None:
        return self.embeddings[word_id]
    else:
        return None
```

Этот магический (дандерный) метод позволит нам получать векторы слов с помощью индексирования:

```
word_vector = embedding["black"]
```

И мы также хотели бы, чтобы слой вложения сообщал нам самые близкие слова к заданному слову:

```
def closest(self, word: str, n: int = 5) -> List[Tuple[float, str]]:
    """Возвращает n ближайших слов на основе косинусного сходства"""
    vector = self[word]

    # Вычислить пары (сходство, другое_слово) и отсортировать по схожести
    # (самое схожее идет первым)
    scores = [(cosine_similarity(vector, self.embeddings[i]), other_word)
              for other_word, i in self.vocab.w2i.items()]
    scores.sort(reverse=True)

    return scores[:n]
```

Наш слой вложения просто выводит векторы, которые мы можем подавать в линейный слой `Linear`.

Теперь мы готовы собрать наши тренировочные данные. Для каждого входного слова мы выберем два слова слева и два слова справа в качестве целевых.

Давайте начнем с перевода предложений в нижний регистр и разбиения их на слова:

```
import re

# Это регулярное выражение не лучшее,
# но оно справляется с нашими данными
tokenized_sentences = [re.findall("[a-z]+|[.]", sentence.lower())
                       for sentence in sentences]
```

В этом месте мы можем построить лексикон:

```
# Создать лексикон (т. е. отображение word -> word_id)
# на основе нашего текста
vocab = Vocabulary(word
                   for sentence_words in tokenized_sentences
                   for word in sentence_words)
```

И теперь мы можем создать тренировочные данные:

```
from scratch.deep_learning import Tensor, one_hot_encode

inputs: List[int] = []
targets: List[Tensor] = []

for sentence in tokenized_sentences:
    for i, word in enumerate(sentence):          # Для каждого слова
        for j in [i - 2, i - 1, i + 1, i + 2]:   # взять близлежащие
                                                 # расположения,
            if 0 <= j < len(sentence):           # которые не выходят
                                                 # за границы,
                nearby_word = sentence[j]        # и получить эти слова

                # Добавить вход, т. е. исходный word_id
                inputs.append(vocab.get_id(word))

                # Добавить цель, т. е. близлежащее слово,
                # кодированное с одним активным состоянием
                targets.append(vocab.one_hot_encode(nearby_word))
```

С помощью механизма, который мы выстроили, теперь легко создать нашу модель:

```
from scratch.deep_learning import Sequential, Linear

random.seed(0)
EMBEDDING_DIM = 5 # seems like a good size
```

```
# Определить слой вложения отдельно,
# чтобы можно было на него ссылаться
embedding = TextEmbedding(vocab=vocab, embedding_dim=EMBEDDING_DIM)

model = Sequential([
    # С учетом заданного слова (как вектора идентификаторов word_id)
    # найти его вложение
    embedding,
    # И применить линейный слой для вычисления
    # балльных отметок для "близлежащих слов"
    Linear(input_dim=EMBEDDING_DIM, output_dim=vocab.size)
])
```

Используя функционал из *главы 19*, легко натренировать нашу модель:

```
from scratch.deep_learning import SoftmaxCrossEntropy, Momentum, GradientDescent

loss = SoftmaxCrossEntropy()
optimizer = GradientDescent(learning_rate=0.01)

for epoch in range(100):
    epoch_loss = 0.0
    for input, target in zip(inputs, targets):
        predicted = model.forward(input)
        epoch_loss += loss.loss(predicted, target)
        gradient = loss.gradient(predicted, target)
        model.backward(gradient)
        optimizer.step(model)
    print(epoch, epoch_loss)            # Напечатать потерю, а также
    print(embedding.closest("black"))   # несколько близлежащих слов,
    print(embedding.closest("slow"))    # чтобы увидеть, что было
    print(embedding.closest("car"))     # усвоено
```

Когда вы смотрите, как проходит процесс тренировки, вы видите, как цвета приближаются друг к другу, и то же самое происходит с прилагательными и существительными.

После того как модель будет натренирована, будет увлекательно разведать наиболее похожие слова:

```
pairs = [(cosine_similarity(embedding[w1], embedding[w2]), w1, w2)
         for w1 in vocab.w2i
         for w2 in vocab.w2i
         if w1 < w2]
pairs.sort(reverse=True)
print(pairs[:5])
```

Этот фрагмент кода (у меня) приводит к:

```
[(0.9980283554864815, 'boat', 'car'),
(0.9975147744587706, 'bed', 'cat'),
```

```
(0.9953153441218054, 'seems', 'was'),
(0.9927107440377975, 'extremely', 'quite'),
(0.9836183658415987, 'bed', 'car')]
```

(Очевидно, что "bed" (кровать) и "cat" (кот) на самом деле не похожи, но в наших тренировочных предложениях они таковыми кажутся, и это именно то, что данная модель улавливает.)

Мы также можем извлечь первые две главные компоненты и вывести их на график:

```
from scratch.working_with_data import pca, transform
import matplotlib.pyplot as plt

# Извлечь первые две главные компоненты и преобразовать в векторы слов
components = pca(embedding.embeddings, 2)
transformed = transform(embedding.embeddings, components)

# Рассеять точки (и сделать их белыми, чтобы они были "невидимыми")
fig, ax = plt.subplots()
ax.scatter(*zip(*transformed), marker='.', color='w')

# Добавить аннотации для каждого слова в его
# трансформированном расположении
for word, idx in vocab.w2i.items():
    ax.annotate(word, transformed[idx])

# И спрятать оси
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

plt.show()
```

График показывает, что схожие слова действительно группируются вместе (рис. 21.3).

Если вам интересно, натренировать векторы слов CBOW нетрудно. Вам придется немного поработать. Во-первых, вам нужно изменить слой вложения `Embedding` так, чтобы он принимал на входе список идентификаторов и выводил список векторов вложения. Затем вам нужно создать новый слой (`Sum`?), который берет список векторов и возвращает их сумму.

Каждое слово представляет собой тренировочный пример, где входами являются идентификаторы окружающих слов, а целью — закодированное с одним активным состоянием само слово.

Модифицированный слой `Embedding` превращает окружающие слова в список векторов, новый слой `Sum` сворачивает список векторов в один-единственный вектор, а затем линейный слой `Linear` может произвести отметки, которые могут быть обработаны функцией `softmax`, получая распределение, представляющее "наиболее вероятные слова с учетом этого контекста".

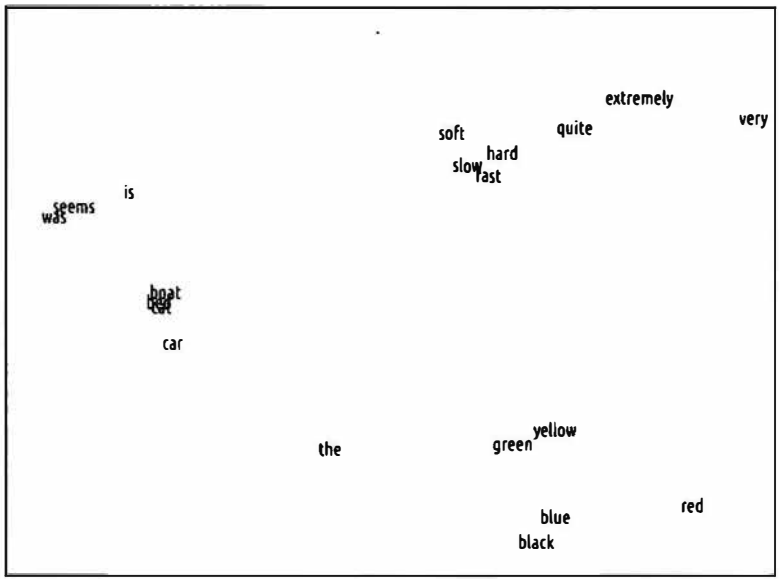


Рис. 21.3. Векторы слов

Я обнаружил, что модель CBOW сложнее тренируется, чем модель Skip-Gram, но я призываю вас поработать с ней.

Рекуррентные нейронные сети

Векторы слов, которые мы разработали в предыдущем разделе, часто используются в качестве входов в нейронные сети. Одна из проблем заключается в том, что предложения имеют разную длину: вы можете представить 3-словное предложение как тензор `[3, embedding_dim]`, а 10-словное предложение — как тензор `[10, embedding_dim]`. Скажем, для передачи их в линейный слой нам нужно что-то сделать с этой первой размерностью переменной длины.

Один из вариантов — использовать слой суммирования `Sum` (либо вариант, который берет среднее арифметическое); однако порядок слов в предложении обычно важен для его значения. Возьмем распространенный пример: "dog bites man" (собака кусает человека) и "man bites dog" (человек кусает собаку) — это две совершенно разные истории!

Другой способ — использовать рекуррентные нейронные сети (recurrent neural networks, RNN-сети), которые имеют скрытое состояние, поддерживаемое ими между входами. В простейшем случае каждый вход совмещается с текущим скрытым состоянием, производя выход, который затем используется как новое скрытое состояние. Это позволяет таким сетям (в некотором смысле) "запоминать" входы, которые они встречали, и создавать конечный выход, который зависит от всех входов и их порядка следования.

Мы создадим в значительной степени наипростейший из возможных слой RNN-сети, который будет принимать один-единственный вход (соответствующий, например, одному слову в предложении или одному символу в слове) и который будет поддерживать свое скрытое состояние между вызовами.

Напомним, что наш линейный слой `Linear` имел некие веса `w` и смещение `b`. Он брал вектор `input` и производил другой вектор `output` в качестве выхода, используя логику:

```
output[o] = dot(w[o], input) + b[o]
```

Здесь мы хотим встроить наше скрытое состояние, поэтому у нас будут два набора весов — один для применения ко входу `input` и один для применения к предыдущему скрытому состоянию `hidden`:

```
output[o] = dot(w[o], input) + dot(u[o], hidden) + b[o]
```

Далее мы будем использовать выходной вектор `output` как новое значение состояния `hidden`. Как видите, это — небольшое изменение, но оно позволит нашим сетям делать замечательные вещи.

```
from scratch.deep_learning import tensor_apply, tanh

class SimpleRnn(Layer):
    """Почти простейший из возможных рекуррентный слой"""
    def __init__(self, input_dim: int, hidden_dim: int) -> None:
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        self.w = random_tensor(hidden_dim, input_dim, init='xavier')
        self.u = random_tensor(hidden_dim, hidden_dim, init='xavier')
        self.b = random_tensor(hidden_dim)

        self.reset_hidden_state()

    def reset_hidden_state(self) -> None:
        self.hidden = [0 for _ in range(self.hidden_dim)]
```

Вы видите, что мы начинаем со скрытого состояния как вектора нулей и предоставляем функцию, которую люди, использующие сеть, могут вызвать для того, чтобы обнулять скрытое состояние.

С учетом этой настройки функция прямого распространения `forward` является достаточно простой (по крайней мере, если вы помните и понимаете, как работал наш линейный слой `Linear`):

```
    def forward(self, input: Tensor) -> Tensor:
        self.input = input              # Сохранить вход и предыдущее скрытое
        self.prev_hidden = self.hidden  # состояние для обратного
                                        # распространения
```

```
        a = [(dot(self.w[h], input) +          # веса @ вход
             dot(self.u[h], self.hidden) +     # веса @ скрытое состояние
             self.b[h])                        # смещение
            for h in range(self.hidden_dim)]

        self.hidden = tensor_apply(tanh, a)    # Применить активацию tanh
        return self.hidden                     # и вернуть результат
```

Обратное прохождение функцией `backward` аналогично тому, которое используется в нашем линейном слое `Linear`, за исключением того, что ему нужно вычислить дополнительный набор градиентов для весов `u`:

```
    def backward(self, gradient: Tensor):
        # Распространить назад через активацию tanh
        a_grad = [gradient[h] * (1 - self.hidden[h] ** 2)
                  for h in range(self.hidden_dim)]

        # b имеет тот же градиент, что и a
        self.b_grad = a_grad

        # Каждый w[h][i] умножается на input[i] и добавляется в a[h],
        # поэтому каждый w_grad[h][i] = a_grad[h] * input[i]
        self.w_grad = [[a_grad[h] * self.input[i]
                        for i in range(self.input_dim)]
                       for h in range(self.hidden_dim)]

        # Каждый u[h][h2] умножается на hidden[h2] и добавляется в a[h],
        # поэтому каждый u_grad[h][h2] = a_grad[h] * prev_hidden[h2]
        self.u_grad = [[a_grad[h] * self.prev_hidden[h2]
                        for h2 in range(self.hidden_dim)]
                       for h in range(self.hidden_dim)]

        # Каждый input[i] умножается на каждый w[h][i] и добавляется
        # в a[h], поэтому каждый
        # input_grad[i] = sum(a_grad[h] * w[h][i] for h in ...)
        return [sum(a_grad[h] * self.w[h][i] for h
                in range(self.hidden_dim))
                for i in range(self.input_dim)]
```

И наконец, нам нужно переопределить методы `params` и `grads`:

```
    def params(self) -> Iterable[Tensor]:
        return [self.w, self.u, self.b]

    def grads(self) -> Iterable[Tensor]:
        return [self.w_grad, self.u_grad, self.b_grad]
```

Эта "простая" RNN-сеть является настолько простой, что вам лучше не использовать ее на практике.

Наш простой слой `SimpleRnn` имеет несколько нежелательных особенностей. Во-первых, его скрытое состояние все целиком используется для обновления входа всякий раз, когда вы его вызываете. Во-вторых, скрытое состояние всё целиком перезаписывается при каждом вызове. Обе этих особенности затрудняют тренировку; в частности, они создают для модели сложности в усвоении долгосрочных зависимостей.

По этой причине почти никто не использует такой простой RNN-слой. Вместо этого применяются более сложные варианты, такие как элемент LSTM-слой (long short-term memory, долгая кратковременная память) или GRU-слой (gated recurrent unit, вентильный рекуррентный элемент), которые имеют гораздо больше параметров и используют параметризованные "вентили", разрешающие обновлять только некоторые состояния (и использовать только часть состояния) на каждом временном шаге.

В этих вариантах нет ничего особенно сложного; однако они задействуют гораздо больше исходного кода, который, на мой взгляд, не был бы нужным образом более поучительным при чтении. Исходный код для этой главы в репозитории GitHub включает имплементацию LSTM. Я призываю вас обратиться к нему, но он несколько утомителен, и поэтому здесь мы его не будем обсуждать.

Еще одна особенность нашей имплементации заключается в том, что она делает всего один "шаг" за раз и требует, чтобы мы вручную обнуляли скрытое состояние. Более практичная имплементация RNN-слоя может принимать последовательности входов, устанавливать свое скрытое состояние в нули в начале каждой последовательности и производить последовательности выходов. Наш слой, безусловно, можно модифицировать, чтобы он вел себя таким образом, и это потребует больше кода и сложности, но будет иметь малый выигрыш в понимании.

Пример: использование RNN-сети уровня букв

Недавно нанятый директор по брендингу не придумал название DataSciencester сам, и (соответственно) он подозревает, что к большему успеху компании может привести более подходящее название. Он просит вас применить науку о данных для того, чтобы предложить кандидатов на замену.

Одно "милое" применение RNN-сетей предусматривает использование в качестве входов не слов, а букв, тренировку с целью усвоения тонких языковых регулярностей в некоем наборе данных, а затем их использование для создания вымышленных экземпляров из этого набора данных.

Например, вы можете натренировать RNN-сеть названиям альтернативных рок-групп, применить натренированную модель для создания новых названий поддельных альтернативных рок-групп, а затем вручную выбрать самые смешные и поделиться ими в Twitter. Вот потеха!

Опробовав этот трюк достаточно много раз для того, чтобы больше не считать его умным, вы решаете дать ему шанс.

После некоторого копания во Всемирой паутине вы обнаруживаете, что ускоритель запуска Y Combinator опубликовал список своих 100 (на самом деле 101) самых успешных стартапов, что кажется хорошей отправной точкой. Проверяя страницу, вы обнаружите, что все названия компаний располагаются внутри тегов <b class="h4">. Это означает, что для их извлечения будет легко применить ваши навыки по веб-выскабливанию:

```
from bs4 import BeautifulSoup
import requests

url = "https://www.ycombinator.com/topcompanies/"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')

# Получаем компании дважды, поэтому применяем
# операцию включения во множество для удаления повторов
companies = list({b.text
                  for b in soup("b")
                  if "h4" in b.get("class", ())})

assert len(companies) == 101
```

Как всегда, веб-страница может измениться (или исчезнуть), и в таком случае указанный фрагмент кода работать не будет. Если это так, то вы можете применить свои недавно усвоенные навыки в области науки о данных, исправив это, либо просто получить список из репозитория книги на GitHub.

Так каков же наш план? Мы натренируем модель предсказывать следующую букву названия с учетом текущей буквы и скрытого состояния, представляющего все буквы, которые мы видели до сих пор.

Как обычно, мы фактически предсказываем распределение вероятности над буквами и тренируем нашу модель минимизировать перекрестно-энтропийную потерю `SoftmaxCrossEntropy`.

Когда наша модель будет натренирована, мы можем применить ее для того, чтобы генерировать некоторые вероятности, случайно отбирать букву в соответствии с этими вероятностями, а затем подавать эту букву в качестве следующего входа. Это позволит нам генерировать названия компаний, используя усвоенные веса.

Для начала мы должны построить лексикон `Vocabulary` из букв в названиях:

```
vocab = Vocabulary([c for company in companies for c in company])
```

В дополнение к этому мы будем использовать специальные лексемы, которые будут обозначать начало и конец названия компании. Это позволяет модели усваивать то, какие символы должны начинать название компании, а также усваивать, когда название компании заканчивается.

Мы просто будем использовать символы регулярных выражений для начала и конца, которые (к счастью) не отображаются в нашем списке компаний:

```
START = "^"
STOP = "$"

# Нам также потребуется добавить их в лексикон
vocab.add(START)
vocab.add(STOP)
```

Для нашей модели мы будем кодировать каждую букву в форме с одним активным состоянием, передавать ее через два слоя `SimpleRnn`, а затем использовать линейный слой `Linear` для генерирования балльных отметок для каждого возможного следующего символа:

```
HIDDEN_DIM = 32  # Вам следует поэкспериментировать с другими размерами!

rnn1 = SimpleRnn(input_dim=vocab.size, hidden_dim=HIDDEN_DIM)
rnn2 = SimpleRnn(input_dim=HIDDEN_DIM, hidden_dim=HIDDEN_DIM)
linear = Linear(input_dim=HIDDEN_DIM, output_dim=vocab.size)

model = Sequential([
    rnn1,
    rnn2,
    linear
])
```

Представьте на мгновение, что мы натренировали эту модель. Давайте напишем функцию, использующую ее для генерирования новых имен компаний, на основе функции `sample_from` из *разд. "Тематическое моделирование" этой главы*:

```
from scratch.deep_learning import softmax

def generate(seed: str = START, max_len: int = 50) -> str:
            rnn1.reset_hidden_state()  # Обнулить оба скрытых состояния
            rnn2.reset_hidden_state()
            output = [seed]            # Инициализировать выход заданным
                                       # начальным посевом

    # Продолжать до тех пор, пока мы не произведем букву STOP
    # либо не достигнем максимальной длины
    while output[-1] != STOP and len(output) < max_len:
        # Использовать последнюю букву как вход
        input = vocab.one_hot_encode(output[-1])

        # Сгенерировать отметки, используя модель
        predicted = model.forward(input)

        # Конвертировать их в вероятности и извлечь случайный char_id
        probabilities = softmax(predicted)
        next_char_id = sample_from(probabilities)
```

```
        # Добавить соответствующий символ char в выход
        output.append(vocab.get_word(next_char_id))

    # Отбросить буквы START и END и вернуть слово
    return ''.join(output[1:-1])
```

Наконец-то мы готовы натренировать нашу RNN-сеть уровня букв. Это займет некоторое время!

```
loss = SoftmaxCrossEntropy()
optimizer = Momentum(learning_rate=0.01, momentum=0.9)

for epoch in range(300):
    random.shuffle(companies)     # Тренировать в другом порядке
                                  # в каждой эпохе
    epoch_loss = 0                # Отслеживать потерю
    for company in tqdm.tqdm(companies):
        rnn1.reset_hidden_state()  # Обнулить оба скрытых состояния
        rnn2.reset_hidden_state()
        company = START + company + STOP  # Добавить буквы START и STOP

        # Остальное - это просто наш обычный тренировочный цикл обучения,
        # за исключением того, что входы и цель - это кодированные
        # в форме с одним активным состоянием предыдущая и следующая буквы
        for prev, next in zip(company, company[1:]):
            input = vocab.one_hot_encode(prev)
            target = vocab.one_hot_encode(next)
            predicted = model.forward(input)
            epoch_loss += loss.loss(predicted, target)
            gradient = loss.gradient(predicted, target)
            model.backward(gradient)
            optimizer.step(model)

        # Для каждой эпохи печатать потерю, а также генерировать название
        print(epoch, epoch_loss, generate())

        # Уменьшать темп усвоения для последних 100 эпох.
        # Для этого нет принципиальной причины, но, похоже, это работает
        if epoch == 200:
            optimizer.lr *= 0.1
```

После тренировки модель генерирует некоторые фактические названия из списка (что неудивительно, т. к. модель имеет достаточный объем возможностей и не так много тренировочных данных), а также названия, которые лишь немного отличаются от тренировочных названий (Scripe, Loinbare, Pozium); названия, которые кажутся по-настоящему изобретательными (Benuus, Cletpo, Equite, Vivest); и названия, которые относятся к мусору, но все-равно выглядят как слова (SFitreasy, Sint ocanelp, GliyOx, Doorboronelhav).

К сожалению, как и большинство выходов RNN-сети уровня букв, они лишь отдаленно толковы, и директор по брендингу не может их использовать.

Если я поднимаю скрытую размерность до 64, то получаю намного больше названий буквально из списка; если я опускаю его до 8, то получаю в основном мусор. Словарь и окончательные веса для всех этих модельных размеров имеются в репозитории книги на GitHub[6], и вы можете применить функции `load_weights` и `load_vocab` для того, чтобы использовать их самостоятельно.

Как упоминалось ранее, исходный код на GitHub для этой главы также содержит реализацию для LSTM-слоя, который вы можете свободно применить в качестве замены для слоев `SimpleRnn` в нашей модели названий компаний.

Для дальнейшего изучения

- Естественно-языковой инструментарий NLTK[7] (Natural Language Toolkit) — это популярная библиотека средств обработки ЕЯ на Python. Она имеет собственную книгу[8], которая доступна для чтения в сети.
- gensim[9] — это библиотека языка Python для тематического моделирования, которая предлагает больше возможностей, чем показанная здесь модель с нуля.
- spaCy[10] — это библиотека для "промышленной обработки естественного языка на Python", и она также довольно популярна.
- У Андрея Карпати есть известный пост "Необоснованная эффективность рекуррентных нейронных сетей"[11], который очень стоит прочитать.
- Моя дневная работа сопряжена с созданием AllenNLP[12], библиотеки языка Python для проведения исследований в области обработки ЕЯ. (По крайней мере, с того момента, когда эта книга попала в печать, так оно и было.) Указанная библиотека выходит совершенно за рамки этой книги, но вы все равно можете ее найти интересной, и в ней есть классная интерактивная демонстрация многих ультрасовременных моделей, используемых в обработке ЕЯ.

[6] См. **https://github.com/joelgrus/data-science-from-scratch**.

[7] См. **http://www.nltk.org/**.

[8] См. **http://www.nltk.org/book/**.

[9] См. **http://radimrehurek.com/gensim/**.

[10] См. **https://spacy.io/**.

[11] См. **http://karpathy.github.io/2015/05/21/rnn-effectiveness/**.

[12] См. **https://allennlp.org/**.

ГЛАВА 22
Сетевой анализ

> Ваши связи со всем, что вас окружает, в буквальном смысле определяют вас.
> – *Аарон О'Коннелл*[1]

Многие интересные задачи, связанные с анализом данных, могут быть плодотворно осмыслены в терминах *сетей*, состоящих из *узлов* определенного типа и *ребер*, которые их соединяют.

Например, друзья в Facebook образуют узлы сети, в которой ребра — это дружеские отношения. Менее очевидным примером является непосредственно сама Всемирная паутина, в которой веб-страницы являются узлами, а гиперссылки с одной страницы на другую — ребрами.

Дружба в Facebook взаимна: если я дружу с вами, то вы неизбежно дружите со мной. В этом случае говорят, что ребра являются *неориентированными*. А вот гиперссылки — нет: мой веб-сайт ссылается на веб-сайт Белого дома whitehouse.gov, однако (по необъяснимым для меня причинам) whitehouse.gov отказывается ссылаться на мой веб-сайт. Такие типы дуг называются *ориентированными*. Мы рассмотрим оба вида сетей.

Центральность по посредничеству

В *главе 1* были вычислены ключевые звенья в сети DataSciencester путем подсчета числа друзей каждого пользователя. Теперь мы имеем достаточно функционала для того, чтобы обратиться к другим подходам. Мы будем работать с той же сетью, но теперь будем использовать типизированные именованные кортежи `NamedTuple`.

Вспомните, что сеть (рис. 22.1) составляют пользователи:

```
from typing import NamedTuple

class User(NamedTuple):
    id: int
    name: str
```

[1] Аарон Д. О'Коннелл (род. 1981) — американский ученый в области экспериментальной квантовой физики, создатель первой в мире квантовой машины. — *Прим. пер.*

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

и дружеские отношения:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

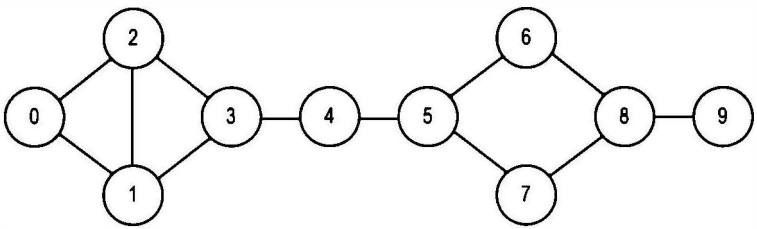


Рис. 22.1. Социальная сеть DataSciencester

С дружескими отношениями легче работать в качестве словаря:

```
from typing import Dict, List

# Псевдонимы типов для отслеживания дружеских отношений
Friendships = Dict[int, List[int]]

friendships: Friendships = {user.id: [] for user in users}

for i, j in friend_pairs:
    friendships[i].append(j)
    friendships[j].append(i)

assert friendships[4] == [3, 5]
assert friendships[8] == [6, 7, 9]
```

Когда мы оставили эту тему, мы были не удовлетворены понятием *центральности по степени узлов*, которое практически шло вразрез с интуитивным пониманием того, кто является ключевым звеном сети.

Альтернативной метрикой является *центральность по посредничеству*, выявляющая людей, которые часто расположены на кратчайшем пути между парами других

людей. В частности, центральность по посредничеству узла i вычисляется путем сложения для каждой другой пары узлов j и k доли кратчайших путей между узлом j и узлом k, которые проходят через i.

То есть для выяснения центральности по посредничеству пользователя Thor нам нужно вычислить все кратчайшие пути между всеми парами людей за исключением пользователя Thor. И затем подсчитать количество этих кратчайших путей, которые проходят через пользователя Thor. Например, через Thor проходит единственный кратчайший путь между Chi (`id` 3) и Clive (`id` 5) и ни один из двух кратчайших путей между Hero (`id` 0) и Chi (`id` 3).

Таким образом, в качестве первого шага нам необходимо выяснить кратчайшие пути между всеми парами людей. Эту задачу эффективно решают несколько довольно изощренных алгоритмов, однако (как и почти везде) мы воспользуемся алгоритмом, который менее эффективен, но легче для понимания.

Этот алгоритм (имплементация поиска сперва в ширину в графе) является одним из наиболее сложных в книге, так что остановимся на нем подробнее:

1. Нашей целью является функция, которая берет стартового пользователя `from_user` и находит *все* кратчайшие пути до остальных пользователей.
2. Мы представим путь как список идентификаторов пользователей. Поскольку каждый путь начинается с пользователя `from_user`, мы не будем включать его идентификатор в список. Это означает, что длина списка, представляющего путь, будет равна длине самого пути.
3. Мы будем поддерживать словарь кратчайших путей до пункта назначения, именуемый `shortest_paths_to`, в котором ключами будут идентификаторы пользователей, а значениями — списки путей, которые заканчиваются на пользователе с данным идентификатором. Если существует уникальный кратчайший путь, то список будет содержать этот один путь. Если же существуют многочисленные пути, то список будет содержать их все.
4. Мы также будем поддерживать очередь `frontier`, содержащую пользователей, которых мы хотим разведать в нужном нам порядке. Мы будем хранить их как пары `(предыдущий_пользователь, пользователь)` для того, чтобы знать о том, как мы добрались до каждого из них. Мы инициализируем очередь всеми соседями пользователя `from_user`. (Мы еще не говорили об очередях, т. е. структурах данных, оптимизированных для операций "добавить в конец очереди" и "удалить из головы очереди". На языке Python они имплементированы в виде очереди с двусторонним доступом `collections.deque`.)
5. По мере разведывания графа всякий раз, когда мы отыскиваем новых соседей, кратчайшие пути до которых еще нам неизвестны, мы добавляем их в конец очереди для того, чтобы разведать их позже, при этом текущий пользователь будет рассматриваться как предыдущий, `prev_user`.
6. Когда мы берем пользователя из очереди, и при этом мы никогда не встречали его раньше, то мы определенно нашли один или больше кратчайших путей к не-

му — кратчайший путь к пользователю `prev_user` с добавлением одного лишнего шага.

7. Когда мы берем пользователя из очереди, и мы *уже* встречали этого пользователя раньше, тогда либо мы нашли еще один кратчайший путь (в таком случае мы должны его добавить), либо мы нашли более длинный путь (в таком случае мы не должны этого делать).
8. Когда в очереди больше нет пользователей, значит, мы разведали весь граф целиком (либо, по крайней мере, те его части, которые достижимы из стартового пользователя) и завершаем работу.

Мы можем собрать все это в одну (большую) функцию:

```
from collections import deque

Path = List[int]

def shortest_paths_from(from_user_id: int,
                        friendships: Friendships) -> Dict[int, List[Path]]:
    # Словарь из user_id во *все* кратчайшие пути до этого пользователя
    shortest_paths_to: Dict[int, List[Path]] = {from_user_id: [[]]}

    # Очередь (предыдущий пользователь, следующий пользователь),
    # которую мы должны проверить, начинается со всех пар
    # (from_user, friend_of_from_user), т. е. пользователя и его друга
    frontier = deque((from_user_id, friend_id)
                     for friend_id in friendships[from_user_id])

    # Продолжать до тех пор, пока мы не опустошим очередь
    while frontier:
        # Удалить пару, которая является следующей в очереди
        prev_user_id, user_id = frontier.popleft()

        # Из-за способа, каким мы добавляем в очередь, мы
        # с неизбежностью уже знаем некоторые кратчайшие пути до prev_user
        paths_to_prev_user = shortest_paths_to[prev_user_id]
        new_paths_to_user = [path + [user_id]
                             for path in paths_to_prev_user]

        # Возможно, мы уже знаем кратчайший путь до user_id
        old_paths_to_user = shortest_paths_to.get(user_id, [])

        # Каков кратчайший путь сюда, который мы не видели до этого?
        if old_paths_to_user:
            min_path_length = len(old_paths_to_user[0])
        else:
            min_path_length = float('inf')
```

```
        # Оствить только те пути, которые
        # не слишком длинные и фактически новые
        new_paths_to_user = [path
                             for path in new_paths_to_user
                             if len(path) <= min_path_length
                             and path not in old_paths_to_user]

        shortest_paths_to[user_id] = old_paths_to_user + new_paths_to_user

        # Добавить никогда не встречавшихся соседей в frontier
        frontier.extend((user_id, friend_id)
                        for friend_id in friendships[user_id]
                        if friend_id not in shortest_paths_to)

    return shortest_paths_to
```

Теперь давайте вычислим все кратчайшие пути:

```
# Для каждого from_user и для каждого to_user список кратчайших путей
shortest_paths = {user.id: shortest_paths_from(user.id, friendships)
                  for user in users}
```

Наконец мы готовы вычислить центральность по посредничеству. Для каждой пары узлов i и j мы знаем n кратчайших путей из i в j. Тогда для каждого из этих путей мы просто добавляем $1/n$ в центральность каждого узла на этом пути:

```
betweenness_centrality = {user.id: 0.0 for user in users}

for source in users:
    for target_id, paths in shortest_paths[source.id].items():
        if source.id < target_id:     # Не дублировать подсчет
            num_paths = len(paths)    # Сколько кратчайших путей?
            contrib = 1 / num_paths   # Вклад в центральность
            for path in paths:
                for between_id in path:
                    if between_id not in [source.id, target_id]:
                        betweenness_centrality[between_id] += contrib
```

Как показано на рис. 22.2, пользователи 0 и 9 имеют нулевую центральность (т. к. ни один не находится на кратчайшем пути между другими пользователями), тогда как пользователи 3, 4 и 5 имеют высокие центральности (т. к. все трое находятся на нескольких кратчайших путях).

В общем случае показатели центральности сами по себе не так выразительны. А вот что нас действительно заботит, так это то, как числа для каждого узла соотносятся с числами для других узлов.

Еще одна мера, к которой мы можем обратиться, называется *центральностью по близости* (closeness centrality). Сначала для каждого пользователя мы вычисляем

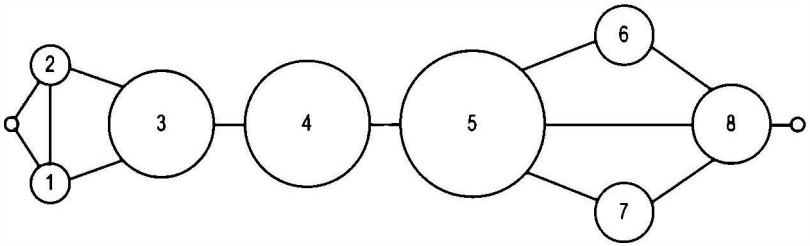


Рис. 22.2. Сеть, измеренная центральностью по посредничеству

его *удаленность*, т. е. сумму длин его кратчайших путей до каждого другого пользователя. Так как мы уже вычислили кратчайшие пути между каждой парой узлов, сложить их длины не представляет труда. (Если существуют многочисленные кратчайшие пути, то все они имеют одинаковую длину, поэтому мы можем просто взять первый из них.)

```
def user_id: int) -> float:
    """Сумма длин кратчайших путей до каждого другого пользователя"""
    return sum(len(paths[0])
               for paths in user["shortest_paths"].values())
```

После чего требуется очень мало работы для вычисления центральности по близости (рис. 22.3):

```
closeness_centrality = {user.id: 1 / farness(user.id) for user in users}
```

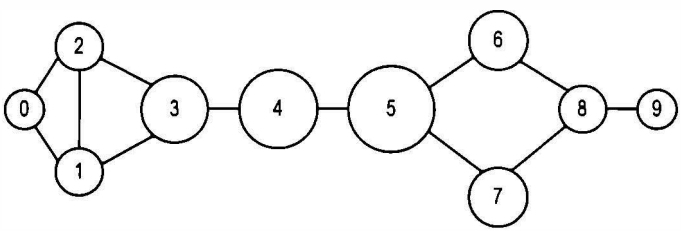


Рис. 22.3. Сеть, измеренная центральностью по близости

Здесь гораздо меньше вариаций — даже очень центральные узлы по-прежнему лежат довольно далеко от узлов на периферии.

Как мы убедились, вычисление кратчайших путей несколько досаждает. По этой причине центральность по посредничеству и центральность по близости используются в крупных сетях не так часто. А вот менее интуитивная (и в общем-то легче вычисляемая) *центральность по собственному вектору* применяется гораздо чаще.

Центральность по собственному вектору

Чтобы начать обсуждать *центральность по собственному вектору*, сначала мы должны поговорить о самих собственных векторах, а чтобы говорить о них, мы должны остановиться на умножении матриц.

Умножение матриц

Если **A** — это матрица $n \times m$, и **B** — это матрица $m \times k$ (обратите внимание, что вторая размерность **A** та же самая, что и первая резмерность **B**), тогда их произведение **AB** есть ($n \times k$)-матрица, (i, j)-й элемент которой равен:

$$a_{i1}b_{1j} + a_{i2}b_{2j} + ... + a_{im}b_{mj}, \qquad (*)$$

где a_{im} — элемент матрицы **A**, стоящий в i-й строке и m-м столбце; b_{mj} — элемент матрицы **B**, стоящий в m-й строке и j-м столбце.

Формула (*) в точности соответствует скалярному произведению i-й строки матрицы **A** (рассматриваемой как вектор) на j-й столбец матрицы **B** (тоже рассматриваемый как вектор).

Мы можем имплементировать это, используя функцию `make_matrix` из *главы 4*:

```
from scratch.linear_algebra import Matrix, make_matrix, shape

def matrix_times_matrix(m1: Matrix, m2: Matrix) -> Matrix:
    nr1, nc1 = shape(m1)
    nr2, nc2 = shape(m2)

    assert nc1 == nr2, "должно быть (число столбцов в m1) == (число строк в m2)"

    def entry_fn(i: int, j: int) -> float:
        """Скалярное произведение
        i-й строки матрицы m1 и j-го столбца матрицы m2"""
        return sum(m1[i][k] * m2[k][j] for k in range(nc1))

    return make_matrix(nr1, nc2, entry_fn)
```

Если мы будем рассматривать m-размерный вектор как матрицу `(m, 1)`, то можем умножить его на матрицу `(n, m)`, получив матрицу `(n, 1)`, которую мы можем рассматривать как n-размерный вектор.

Это означает, что существует еще один способ рассматривать матрицу `(n, m)` — рассматривать ее как линейное отображение, которое преобразовывает m-размерные векторы в n-размерные:

```
from scratch.linear_algebra import Vector, dot

def matrix_times_vector(m: Matrix, v: Vector) -> Vector:
    nr, nc = shape(m)
    n = len(v)
    assert nc == n, "должно быть (число столбцов в m) == (число элементов в v)"

    return [dot(row, v) for row in m]  # выход имеет длину nr
```

Когда **A** является квадратной матрицей, эта операция отображает n-размерные векторы в другие n-размерные векторы. Существует возможность, при которой для

некоторой матрицы **A** и вектора **v**, когда **A** работает на **v**, мы возвращаем скалярное кратное от **v**, т. е. результатом является вектор, который указывает в том же направлении, что и **v**. Когда это происходит (и когда вдобавок **v** не является вектором нулей), мы называем **v** *собственным вектором* матрицы **A**, а произведение — *собственным значением*.

Один из возможных способов найти собственный вектор матрицы **A** заключается в выборе начального вектора **v**, применении функции `matrix_times_vector`, перешкалировании результата до магнитуды 1 и повторении до тех пор, пока указанный процесс не сойдется:

```
from typing import Tuple
import random
from scratch.linear_algebra import magnitude, distance

def find_eigenvector(m: Matrix,
                     tolerance: float = 0.00001) -> Tuple[Vector, float]:
    guess = [random.random() for _ in m]

    while True:
        result = matrix_times_vector(m, guess)  # Преобразовать догадку
        norm = magnitude(result)                # Вычислить норму
        next_guess = [x / norm for x in result] # Перешкалировать

        if distance(guess, next_guess) < tolerance:
            # Схождение, поэтому вернуть
            # (собственный вектор, собственное значение).
            return next_guess, norm

        guess = next_guess
```

По конструкции возвращенная догадка `guess` является таким вектором, что, когда вы применяете к нему функцию `matrix_times_vector` и перешкалируете его до длины 1, вы возвращаете вектор, очень близкий к самому себе. А это означает, что это собственный вектор.

Не все матрицы вещественных чисел имеют собственные векторы и собственные значения. Например, матрица поворота:

```
rotate = [[ 0, 1],
          [-1, 0]]
```

поворачивает векторы на 90 градусов по часовой стрелке, т. е. единственным вектором, который она отображает в скалярное произведение самого себя, является вектор нулей. Если бы вы попытались отыскать собственный вектор функцией `find_eigenvector(rotate)`, то она работала бы вечно. Даже матрицы, которые имеют собственные векторы, иногда застревают в циклах. Рассмотрим матрицу отражения:

```
flip = [[0, 1],
        [1, 0]]
```

Эта матрица отображает любой вектор [x, y] в [y, x]. Это означает, что, например, [1, 1] — это собственный вектор с собственным значением, равным 1. Однако если мы начнем со случайного вектора с неравными координатами, то функция find_eigenvector будет бесконечно обмениваться координатами. (В библиотеках, написанных не с нуля, таких как NumPy, используются другие методы, которые в этом случае будут работать.) Тем не менее, когда функция find_eigenvector все-таки возвращает результат, этот результат действительно является собственным вектором.

Центральность

Как все это поможет понять социальную сеть DataSciencester? Для начала нам необходимо представить связи в нашей сети как матрицу смежности adjacency_matrix, чей (i, j)-й элемент равен либо 1 (если пользователи i и j — друзья), либо 0 (если нет):

```
def entry_fn(i: int, j: int):
    return 1 if (i, j) in friend_pairs or (j, i) in friend_pairs else 0

n = len(users)
adjacency_matrix = make_matrix(n, n, entry_fn)
```

Тогда центральность по собственному вектору для каждого пользователя — это элемент, соответствующий этому пользователю в собственном векторе, возвращаемом функцией find_eigenvector (рис. 22.4).

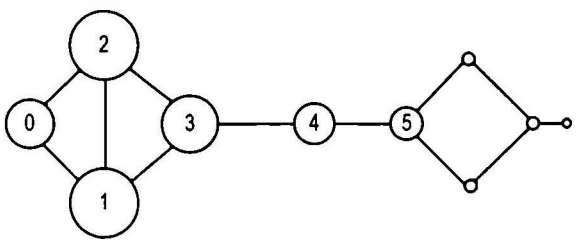


Рис. 22.4. Сеть, измеренная центральностью по собственному вектору

По техническим причинам, которые выходят за рамки этой книги, договоримся, что любая ненулевая матрица смежности обязательно имеет собственный вектор, все значения которого являются неотрицательными. И к счастью для нас, функция find_eigenvector находит его для матрицы смежности adjacency_matrix.

```
eigenvector_centralities, _ = find_eigenvector(adjacency_matrix)
```

Пользователями с высокой центральностью по собственному вектору должны быть те, у кого много связей с людьми, которые сами имеют высокую центральность.

Здесь пользователи 1 и 2 находятся к центру ближе всего, т. к. у обоих имеются три связи с людьми, которые сами находятся близко к центру. По мере того как мы от них отдаляемся, центральности людей неуклонно падают.

В такой малой сети, как эта, центральность по собственному вектору ведет себя несколько хаотично. Если вы попробуете добавлять или удалять связи, то обнаружите, что малые изменения в сети могут кардинально изменить показатели центральности. Для более крупных сетей это нехарактерно.

Мы до сих пор не обосновали причину, почему собственный вектор может приводить к разумному понятию центральности. Быть собственным вектором означает, что если вычислить:

```
matrix_times_vector(adjacency_matrix, eigenvector_centralities)
```

то результатом будет скалярное произведение для центральностей по собственному вектору `eigenvector_centralities`.

Если вы посмотрите на то, как работает умножение матриц, то функция `matrix_times_vector` возвращает вектор, чей *i*-й элемент равен:

```
dot(adjacency_matrix[i], eigenvector_centralities)
```

который является в точности суммой центральностей по собственному вектору пользователей, связанных с пользователем *i*.

Другими словами, центральности по собственному вектору являются числами, одно на каждого пользователя, такими, что значение каждого пользователя есть постоянное кратное суммы значений его соседей. В данном случае центральность означает наличие связи с людьми, которые сами являются центральными. Чем больше центральность, с которой вы непосредственно связаны, тем выше ваша собственная центральность. Такое определение, разумеется, имеет циклический характер — собственные векторы являются способом вырваться из этой цикличности.

Еще один способ понять это — думать о том, что здесь делает функция `find_eigenvector`. Она начинает с того, что назначает каждому узлу случайную центральность. Затем она повторяет следующие два шага до тех пор, пока процесс не сойдется:

1. Дать каждому узлу новую отметку центральности, которая в сумме составляет (старые) отметки центральности его соседей.
2. Перешкалировать вектор центральностей до магнитуды (длины), равной 1.

Хотя лежащая в основе математика может сначала показаться несколько затуманенной, сам расчет довольно прост (в отличие, скажем, от центральности по посредничеству) и легко выполняется даже на очень крупных графах. (По крайней мере, если вы используете настоящую линейно-алгебраическую библиотеку, то расчеты легко выполнять на больших графах. Если бы вы использовали нашу имплементацию матриц в виде списков, то вы бы сделали это с трудом.)

Ориентированные графы и алгоритм PageRank

Социальная сеть DataSciencester не получает достаточной поддержки, поэтому директор по доходам планирует развернуться от модели дружеских отношений в сторону модели *авторитетной поддержки*. Оказывается, никого особо не забо-

тит, какие исследователи данных *дружат* друг с другом, однако агенты по найму научно-технического персонала очень интересуются тем, какие исследователи пользуются в своей среде уважением.

В этой новой модели мы будем отслеживать авторитетную поддержку в формате `(источник, цель)`, которая больше не представляет взаимоотношения, но где теперь пользователь `source` поддерживает своим авторитетом пользователя `target` как потрясающего исследователя данных (рис. 22.5).

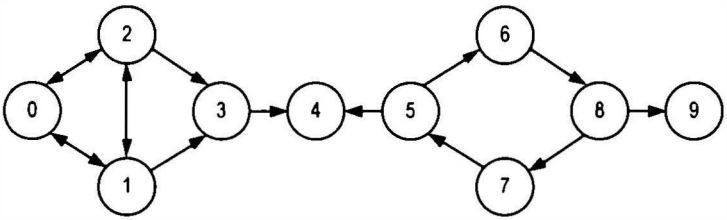


Рис. 22.5. Авторитетная поддержка в сети DataSciencester

Нам необходимо учесть эту асимметрию:

```
# Поддержка своим авторитетом
endorsements = [(0, 1), (1, 0), (0, 2), (2, 0), (1, 2),
                (2, 1), (1, 3), (2, 3), (3, 4), (5, 4),
                (5, 6), (7, 5), (6, 8), (8, 7), (8, 9)]
```

после чего мы можем легко отыскать наиболее поддерживаемых `most_endorsed` исследователей данных и продать эту информацию агентам по найму:

```
from collections import Counter

endorsement_counts = Counter(target for source, target in endorsements)
```

Тем не менее "число авторитетных поддерживаний" легко подделывается. Для этого надо всего лишь завести фальшивые аккаунты, чтобы те поддержали вас, или договориться с друзьями поддерживать друг друга (что пользователи 0, 1 и 2, кажется, сделали).

Усовершенствованная метрика могла бы учитывать того, *кто* именно вас поддерживает своим авторитетом. Авторитетные поддержки со стороны людей, которые сами имеют много авторитетной поддержки, должны каким-то образом значить больше, чем поддержки со стороны людей лишь с несколькими авторитетными поддержками. В этом суть алгоритма PageRank, используемого в Google для упорядочивания веб-сайтов по рангу страниц и основанного на том, какие веб-сайты на них ссылаются, какие другие веб-сайты ссылаются на эти последние и т. д.

(Если это несколько напоминает вам идею о центральности по собственному вектору, то так и должно быть.)

Упрощенная версия указанного алгоритма выглядит так:

1. В сети имеется суммарный ранг страниц PageRank, равный 1.0 (или 100%).
2. Первоначально этот ранг распределен между узлами равномерно.

3. На каждом шаге крупная доля ранга каждого узла равномерно распределяется среди его исходящих связей.
4. На каждом шаге оставшаяся часть ранга каждого узла равномерно распределяется среди всех узлов.

```
import tqdm

def page_rank(users: List[User],
              endorsements: List[Tuple[int, int]],
              damping: float = 0.85,
              num_iters: int = 100) -> Dict[int, float]:
    # Вычислить, сколько людей каждый пользователь
    # поддерживает своим авторитетом
    outgoing_counts = Counter(target for source, target in endorsements)

    # Первоначально распределить ранг PageRank равномерно
    num_users = len(users)
    pr = {user.id : 1 / num_users for user in users}

    # Малая доля ранга PageRank, которую каждый узел
    # получает на каждой итерации
    base_pr = (1 - damping) / num_users

    for iter in tqdm.trange(num_iters):
        next_pr = {user.id : base_pr for user in users} # начать с base_pr

        for source, target in endorsements:
            # Добавить демпфированную долю
            # исходного рейтинга source в цель target
            next_pr[target] += damping * pr[source] /
                               outgoing_counts[source]

        pr = next_pr

    return pr
```

Если мы вычислим ранги страниц:

```
pr = page_rank(users, endorsements)

# Пользователь Thor (user_id 4) имеет более высокий
# ранг страницы, чем кто-либо еще
assert pr[4] > max(page_rank
                   for user_id, page_rank in pr.items()
                   if user_id != 4)
```

то алгоритм PageRank (рис. 21.6) отождествит пользователя 4 (Thor) как исследователя данных с наивысшим рангом.

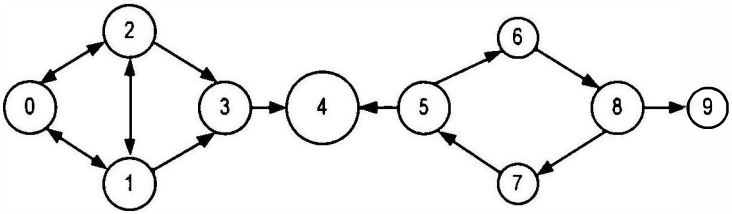


Рис. 21.6. Сеть, измеренная рангом страниц PageRank

Несмотря на то что у него меньше авторитетных поддержек (2), чем у пользователей 0, 1 и 2, его авторитетная поддержка несет вместе с собственной поддержкой ранг со стороны их поддержек. В дополнение к этому оба его сторонника поддерживали своим авторитетом только его, а это значит, что он не должен делить их ранг с кем бы то ни было еще.

Для дальнейшего изучения

- Существует много других понятий центральности[2] кроме тех, которые мы использовали (правда, те, которые мы рассмотрели, пользуются наибольшей популярностью).
- NetworkX[3] — это библиотека языка Python для сетевого анализа. Она имеет функции для вычисления центральностей и визуализации графов.
- Gephi[4] — это инструмент визуализации сетей из разряда "от любви до ненависти" на основе графического интерфейса пользователя.

[2] См. **http://en.wikipedia.org/wiki/Centrality**.

[3] См. **http://networkx.github.io/**.

[4] См. **https://gephi.org/**.

ГЛАВА 23
Рекомендательные системы

> О, природа, природа, и зачем же ты так нечестно поступаешь, когда посылаешь людей в этот мир с ложными рекомендациями!
>
> – *Генри Филдинг*[1]

Еще одна распространенная задача, связанная с аналитической обработкой данных, заключается в генерировании разного рода *рекомендаций*. Компания Netflix рекомендует фильмы, которые вы, возможно, захотите посмотреть, Amazon — товары, которые вы можете захотеть приобрести, Twitter — пользователей, за которыми вы могли бы последовать. В этой главе мы обратимся к нескольким способам применения данных для предоставления рекомендаций.

В частности, мы обратимся к набору данных `users_interests` об интересах пользователей, который применялся ранее:

```
users_interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

И мы подумаем над вопросом генерирования новых интересов для пользователя на основе интересов, указываемых им в настоящее время.

---

[1] Генри Филдинг (1707–1754) — знаменитый английский писатель и драматург XVIII в., известен своим житейским юмором и сатирическим мастерством. Один из основоположников реалистического романа — *Прим. пер.*

Неавтоматическое кураторство

До появления Интернета за советом по поводу книг обычно ходили в библиотеку. Там библиотекарь был готов предложить книги, которые имеют отношение к интересам читателя, либо аналогичные книги по желанию.

Учитывая ограниченное число пользователей соцсети DataSciencester и тем, вызывающих интерес, нетрудно провести полдня, вручную раздавая каждому пользователю рекомендации по поводу новых тем. Но этот метод не очень хорошо поддается масштабированию и ограничен личными познаниями и воображением. (Здесь нет ни капли намека на ограниченность чьих-то личных познаний и воображения.) Посмотрим, что можно сделать при помощи *данных*.

Рекомендация популярных тем

Простейший подход заключается в рекомендации того, что является популярным:

```
from collections import Counter

popular_interests = Counter(interest
                            for user_interests in users_interests
                            for interest in user_interests)
```

которое выглядит так:

```
[('Python', 4),
 ('R', 4),
 ('Java', 3),
 ('regression', 3),
 ('statistics', 3),
 ('probability', 3),
 # ...
]
```

Вычислив это, мы можем предложить пользователю наиболее популярные темы, которыми он еще не интересуется:

```
from typing import List, Tuple

def most_popular_new_interests(
        user_interests: List[str],
        max_results: int = 5) -> List[Tuple[str, int]]:
    suggestions = [(interest, frequency)
                   for interest, frequency
                   in popular_interests.most_common()
                   if interest not in user_interests]
    return suggestions[:max_results]
```

Так, если вы являетесь пользователем 1, который интересуется:

```
["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"]
```

то мы порекомендовали бы вам:

```
[('Python', 4), ('R', 4), ('Java', 3), ('regression', 3), ('statistics', 3)]
```

Если вы являетесь пользователем 3, кто уже интересуется многими из этих вещей, то вместо этого вы получили бы:

```
[('Java', 3), ('HBase', 3), ('Big Data', 3),
 ('neural networks', 2), ('Hadoop', 2)]
```

Естественно, маркетинговый ход в стиле "многие интересуются языком Python, так, может, и вам тоже стоит" не самый убедительный. Если на веб-сайт зашел новичок, о котором ничего не известно, то, возможно, это было бы лучшее, что можно сделать. Давайте посмотрим, каким образом мы могли бы усовершенствовать рекомендации каждому пользователю, если основываться на его интересах.

Коллаборативная фильтрация по схожести пользователей

Один из способов учитывать интересы пользователя заключается в том, чтобы отыскивать пользователей, которые на него как-то *похожи*, и затем предлагать вещи, в которых эти пользователи заинтересованы.

Для этого потребуется способ измерения степени сходства двух пользователей. Здесь будем применять косинусное сходство, которое мы задействовали в *главе 21* для измерения того, насколько два вектора слов похожи.

Мы применим его к векторам нулей и единиц, где каждый вектор `v` обозначает интересы одного пользователя. Элемент `v[i]` будет равен 1, если пользователь указал *i*-ю тему, и 0 — в противном случае. Соответственно, "схожие пользователи" будут означать "пользователей, чьи векторы интересов наиболее близко указывают в том же направлении". Пользователи с одинаковыми интересами будут иметь сходство, равное 1. Пользователи с отсутствующими идентичными интересами будут иметь сходство, равное 0. В противном случае сходство будет попадать между этими числами, где число, близкое к 1, будет указывать на "сильное сходство", а число, близкое к 0 — на "не сильное".

Хорошей отправной точкой является сбор известных интересов и (неявное) назначение им индексов. Мы можем это сделать, воспользовавшись операцией включения в множество, отыскав уникальные интересы, а затем отсортировав их в списке. Первый интерес в результирующем списке будет интересом с индексом 0 и т. д.:

```
# Уникальные интересы
unique_interests = sorted(list({ interest
                                 for user_interests in users_interests
                                 for interest in user_interests }))
```

Этот фрагмент кода даст нам список, который начинается так:

```
assert unique_interests[:6] == [
    'Big Data',
    'C++',
    'Cassandra',
    'HBase',
    'Hadoop',
    'Haskell',
    # ...
]
```

Далее мы хотим произвести вектор "интересов" нулей и единиц для каждого пользователя. Нам нужно только перебрать список уникальных интересов `unique_interests`, подставляя 1, если пользователь имеет соответствующий интерес, и 0 — если нет:

```
# Создать вектор интересов пользователя
def make_user_interest_vector(user_interests: List[str]) -> List[int]:
    """С учетом списка интересов произвести вектор, i-й элемент
       которого равен 1, если unique_interests[i] находится
       в списке, и 0 в противном случае"""
    return [1 if interest in user_interests else 0
            for interest in unique_interests]
```

И теперь мы можем создать список векторов интересов пользователя:

```
user_interest_vectors = [make_user_interest_vector(user_interests)
                         for user_interests in users_interests]
```

Теперь выражение `user_interest_vectors[i][j]` равно 1, если пользователь `i` указал интерес `j`, и 0 — в противном случае.

Так как у нас малый набор данных, не будет никаких проблем вычислить попарные сходства между всеми нашими пользователями:

```
from scratch.nlp import cosine_similarity

user_similarities = [[cosine_similarity(interest_vector_i,
                                        interest_vector_j)
                      for interest_vector_j in user_interest_vectors]
                     for interest_vector_i in user_interest_vectors]
```

после чего выражение `user_similarities[i][j]` дает нам сходство между пользователями `i` и `j`:

```
# Пользователи 0 и 9 делят интересы в Hadoop, Java и Big Data
assert 0.56 < user_similarities[0][9] < 0.58, "несколько общих интересов"

# Пользователи 0 и 8 делят только один интерес: Big Data
assert 0.18 < user_similarities[0][8] < 0.20, "только один общий интерес"
```

В частности, `user_similarities[i]` является вектором сходств пользователя `i` с каждым отдельным пользователем. Мы можем использовать его для написания функции, которая отыскивает пользователей, наиболее похожих на заданного пользователя. При этом мы обеспечиваем, чтобы она не включала ни самого пользователя, ни пользователей с нулевым сходством. И мы отсортируем результаты по убыванию от наиболее похожих до наименее похожих:

```
# Пользователи, наиболее похожие на пользователя user_id
def most_similar_users_to(user_id: int) -> List[Tuple[int, float]]:
    pairs = [(other_user_id, similarity)                # Отыскать других
             for other_user_id, similarity in           # пользователей
                 enumerate(user_similarities[user_id])  # с ненулевым
             if user_id != other_user_id and similarity > 0]  # сходством

    return sorted(pairs,                                # Отсортировать их
                  key=lambda pair: pair[-1],            # по убыванию
                  reverse=True)                         # сходства
```

К примеру, если мы вызовем `most_similar_users_to(0)`, то получим:

```
[(9, 0.5669467095138409),
 (1, 0.3380617018914066),
 (8, 0.1889822365046136),
 (13, 0.1690308509457033),
 (5, 0.1543033499620919)]
```

Как применить эту функцию для рекомендации пользователю новых интересов? По каждому интересу мы можем попросту сложить сходства пользователя с интересами, которые указали другие пользователи:

```
from collections import defaultdict

def user_based_suggestions(user_id: int,
                           include_current_interests: bool = False):
    # Просуммировать сходства
    suggestions: Dict[str, float] = defaultdict(float)
    for other_user_id, similarity in most_similar_users_to(user_id):
        for interest in users_interests[other_user_id]:
            suggestions[interest] += similarity

    # Конвертировать их в отсортированный список
    suggestions = sorted(suggestions.items(),
                         key=lambda pair: pair[-1],  # Вес
                         reverse=True)

    # И (возможно) исключить уже имеющиеся интересы
    if include_current_interests:
        return suggestions
```

```
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

Если мы вызовем `user_based_suggestions(0)`, то несколько первых рекомендуемых интересов будут:

```
[('MapReduce', 0.5669467095138409),
 ('MongoDB', 0.50709255283711),
 ('Postgres', 0.50709255283711),
 ('NoSQL', 0.3380617018914066),
 ('neural networks', 0.1889822365046136),
 ('deep learning', 0.1889822365046136),
 ('artificial intelligence', 0.1889822365046136),
 # ...
]
```

Эти рекомендации выглядят довольно приличными для того, чьи интересы лежат в области больших данных и связаны с базами данных. (Веса не имеют какого-то особого смысла; мы используем их только для упорядочивания.)

Этот подход перестает справляться так же хорошо, когда число элементов становится очень крупным. Вспомните о проклятии размерности из *главы 12* — в крупноразмерных векторных пространствах большинство векторов отдалены друг от друга на большом расстоянии (и также указывают в совершенно разных направлениях). Другими словами, когда имеется крупное число интересов, пользователи, "наиболее похожие" на заданного пользователя, могут оказаться совсем непохожими.

Представим веб-сайт наподобие Amazon.com, на котором за последнюю пару десятилетий я приобрел тысячи предметов. Вы могли бы попытаться выявить похожих на меня пользователей на основе шаблонов покупательского поведения, но, скорее всего, во всем мире нет никого, чья покупательская история выглядит даже отдаленно похожей на мою. Кто бы ни был этот "наиболее похожий" на меня покупатель, он, вероятно, совсем не похож на меня, и его покупки почти наверняка будут содействовать отвратительным рекомендациям.

Коллаборативная фильтрация по схожести предметов

Альтернативный подход заключается в вычислении сходств между интересами непосредственно. Мы можем затем сгенерировать рекомендации по каждому пользователю, агрегировав интересы, схожие с его текущими интересами.

Для начала нам потребуется *транспонировать* матрицу интересов пользователей, вследствие чего строки будут соответствовать интересам, а столбцы — пользователям:

```
interest_user_matrix = [[user_interest_vector[j]
                         for user_interest_vector in user_interest_vectors]
                        for j, _ in enumerate(unique_interests)]
```

Как она выглядит? Строка `j` матрицы `interest_user_matrix` — это столбец `j` матрицы `user_interest_matrix`. То есть она содержит 1 для пользователя с этим интересом и 0 для пользователя без него.

Например, `unique_interests[0]` содержит значение "большие данные" (Big Data), и поэтому `interest_user_matrix[0]` содержит:

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

потому что пользователи 0, 8 и 9 указали свою заинтересованность в больших данных.

Теперь мы можем снова воспользоваться косинусным сходством. Если эти же самые пользователи заинтересованы в двух темах, то их сходство будет равно 1. Если нет двух пользователей, которых интересуют обе темы, то их сходство будет равно 0.

```
interest_similarities = [[cosine_similarity(user_vector_i, user_vector_j)
                          for user_vector_j in interest_user_matrix]
                         for user_vector_i in interest_user_matrix]
```

Например, мы можем отыскать интересы, наиболее похожие на тему больших данных (интерес 0), используя:

```
def most_similar_interests_to(interest_id: int):
    similarities = interest_similarities[interest_id]
    pairs = [(unique_interests[other_interest_id], similarity)
             for other_interest_id, similarity in enumerate(similarities)
             if interest_id != other_interest_id and similarity > 0]
    return sorted(pairs,
                  key=lambda pair: pair[-1],
                  reverse=True)
```

которая рекомендует следующие ниже похожие интересы:

```
[('Hadoop', 0.8164965809277261),
 ('Java', 0.6666666666666666),
 ('MapReduce', 0.5773502691896258),
 ('Spark', 0.5773502691896258),
 ('Storm', 0.5773502691896258),
 ('Cassandra', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('neural networks', 0.4082482904638631),
 ('HBase', 0.3333333333333333)]
```

Теперь мы можем сгенерировать рекомендации для пользователя, просуммировав сходства интересов, похожих с его интересами:

```
def item_based_suggestions(user_id: int,
                           include_current_interests: bool = False):
    # Сложить похожие интересы
    suggestions = defaultdict(float)
    user_interest_vector = user_interest_vectors[user_id]
    for interest_id, is_interested in enumerate(user_interest_vector):
        if is_interested == 1:
            similar_interests = most_similar_interests_to(interest_id)
            for interest, similarity in similar_interests:
                suggestions[interest] += similarity

    # Отсортировать их по весу
    suggestions = sorted(suggestions.items(),
                         key=lambda pair: pair[-1],
                         reverse=True)

    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

Для пользователя 0 эта функция генерирует следующие ниже (на вид разумные) рекомендации:

```
[('MapReduce', 1.861807319565799),
 ('Postgres', 1.3164965809277263),
 ('MongoDB', 1.3164965809277263),
 ('NoSQL', 1.2844570503761732),
 ('programming languages', 0.5773502691896258),
 ('MySQL', 0.5773502691896258),
 ('Haskell', 0.5773502691896258),
 ('databases', 0.5773502691896258),
 ('neural networks', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('C++', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('Python', 0.2886751345948129),
 ('R', 0.2886751345948129)]
```

Разложение матрицы

Как мы видели, мы можем представлять предпочтения наших пользователей как матрицу `[число_пользователей, число_предметов]` нулей и единиц, где единицы обозначают понравившиеся предметы, а нули — не понравившиеся.

Иногда на самом деле у вас могут быть числовые рейтинги; например, когда вы пишете отзыв в Amazon, вы назначаете предмету отметку в интервале от 1 до 5 звезд. Вы по-прежнему могли бы представлять эти числа числами в матрице

[число_пользователей, число_предметов] (на время игнорируя проблему, связанную с тем, что делать с предметами без рейтинга).

В этом разделе мы будем считать, что у нас есть такие рейтинговые данные, и попытаемся усвоить модель, которая способна предсказывать рейтинг для заданного пользователя и предмета.

Один из подходов к решению указанной задачи — допустить, что каждый потребитель имеет какой-то скрытый "тип", который может быть представлен как вектор чисел, и что сходство каждого предмета так же имеет какой-то скрытый "тип".

Если типы пользователей представлены матрицей [число_пользователей, размерность] и транспонирование типов предметов представлено матрицей [размерность, число_предметов], то их произведением является матрица [число_пользователей, число_предметов]. Соответственно, одним из способов построения такой модели является "разложение" матрицы предпочтений на произведение матрицы пользователей и матрицы предметов.

(Возможно, эта идея латентных типов напоминает вам вложения слов в векторное пространство, над которым мы работали в *главе 21*. Ухватитесь за эту идею.)

Вместо того чтобы работать с нашим выдуманным набором данных из 10 пользователей, мы будем работать с набором данных MovieLens 100k, который содержит рейтинги в интервале от 0 до 5 многочисленных фильмов от многочисленных пользователей. Каждый пользователь назначил рейтинг лишь небольшому подмножеству фильмов. Мы воспользуемся этим для того, чтобы попытаться построить систему, которая будет способна предсказывать рейтинг для любой заданной пары (пользователь, фильм). Мы натренируем ее предсказывать на фильмах, которым каждый пользователь назначил рейтинг; будем надяться, что потом она будет обобщать на фильмы, которым пользователь не назначал рейтинг.

Для начала давайте заполучим набор данных. Вы можете скачать его по адресу **http://files.grouplens.org/datasets/movielens/ml-100k.zip**.

Распакуйте его и извлеките файлы; мы будем использовать только два из них:

```
# Текущий каталог; модифицируйте запись,
# если ваши файлы находятся в другом месте.
# Разметка вертикальной чертой: id_фильма|название|...
MOVIES = "u.item"
# Разметка символами табуляции:
# id_польз, id_фильма, рейтинг, временная метка
RATINGS = "u.data"
```

Как уже заведено, мы введем типизированный именованный кортеж NamedTuple, облегчив работу с задачей:

```
from typing import NamedTuple

class Rating(NamedTuple):
    user_id: str
    movie_id: str
    rating: float
```

Идентификатор фильма и идентификаторы пользователей на самом деле являются целыми числами, но они не следуют подряд, а значит, если бы мы работали с ними как целыми числами, то в конечном итоге у нас получилось бы много потраченных впустую размерностей (если только их не перенумеровать). Поэтому, не усложняя, мы будем трактовать их как строковые значения.

Теперь давайте их прочитаем и изучим. Файл фильмов, размеченный вертикальной чертой, имеет много столбцов. Нас интересуют только первые два, т. е. идентификатор и название:

```
import csv

# Мы указываем эту кодировку во избежание ошибки UnicodeDecodeError
# См. https://stackoverflow.com/a/53136168/1076346.
with open(MOVIES, encoding="iso-8859-1") as f:
    reader = csv.reader(f, delimiter="|")
    movies = {movie_id: title for movie_id, title, *_ in reader}
```

Файл рейтингов размечен символами табуляции и содержит четыре столбца для `идентификатор_пользователя`, `идентификатор_фильма`, `рейтинг` (от 1 до 5) и `временная_метка`. Мы будем игнорировать метку, т. к. она нам не потребуется:

```
# Создать список рейтингов [Rating]
with open(RATINGS, encoding="iso-8859-1") as f:
    reader = csv.reader(f, delimiter="\t")
    ratings = [Rating(user_id, movie_id, float(rating))
               for user_id, movie_id, rating, _ in reader]

# 1682 фильма с рейтингами, назначенными 943 пользователями
assert len(movies) == 1682
assert len(list({rating.user_id for rating in ratings})) == 943
```

На этих данных можно проделать большой разведывательный анализ; например, можно заинтересоваться средним рейтингом фильма "Звездные войны" (набор содержит данные с 1998 года, а значит, он предшествует "Скрытой угрозе" на год):

```
import re

# Структура данных для накопления рейтингов по id_фильма
star_wars_ratings = {movie_id: []
                     for movie_id, title in movies.items()
                     if re.search("Star Wars|Empire Strikes|Jedi", title)}

# Перебрать рейтинги, накапливая их для *Звездных войн*
for rating in ratings:
    if rating.movie_id in star_wars_ratings:
        star_wars_ratings[rating.movie_id].append(rating.rating)

# Вычислить средний рейтинг для каждого фильма
avg_ratings = [(sum(title_ratings) / len(title_ratings), movie_id)
               for movie_id, title_ratings in star_wars_ratings.items()]
```

```
# И затем напечатать их по порядку
for avg_rating, movie_id in sorted(avg_ratings, reverse=True):
    print(f"{avg_rating:.2f} {movies[movie_id]}")
```

Они все имеют довольно высокий рейтинг:

```
4.36 Star Wars (1977)
4.20 Empire Strikes Back, The (1980)
4.01 Return of the Jedi (1983)
```

Итак, давайте попробуем придумать модель, которая будет предсказывать эти рейтинги. В качестве первого шага разобьем рейтинговые данные на тренировочный, контрольный и тестовый наборы:

```
import random

random.seed(0)
random.shuffle(ratings)

split1 = int(len(ratings) * 0.7)
split2 = int(len(ratings) * 0.85)

train = ratings[:split1]              # 70% данных
validation = ratings[split1:split2]   # 15% данных
test = ratings[split2:]               # 15% данных
```

Всегда неплохо иметь простую базовую модель и обеспечивать, чтобы наша показывала более высокие результаты, чем базовая. Здесь простой базовой моделью может быть "предсказание среднего рейтинга". В качестве нашей метрики мы будем использовать среднеквадратическую ошибку, поэтому давайте посмотрим на то, как базовая модель работает на нашем тестовом наборе:

```
avg_rating = sum(rating.rating for rating in train) / len(train)
baseline_error = sum((rating.rating - avg_rating) ** 2
                     for rating in test) / len(test)

# Это то, что мы надеемся улучшить
assert 1.26 < baseline_error < 1.27
```

С учетом наших векторных вложений предсказанные рейтинги задаются матричным умножением вложений пользователя и вложений фильмов. Для заданного пользователя и фильма это значение является скалярным произведением соответствующих вложений.

Итак, давайте начнем с создания вложений. Мы будем представлять их как словарь, где ключами являются идентификаторы, а значениями — векторы, что позволит нам легко извлекать вложение по заданному идентификатору:

```
from scratch.deep_learning import random_tensor

EMBEDDING_DIM = 2
```

```
# Отыскать уникальные идентификаторы
user_ids = {rating.user_id for rating in ratings}
movie_ids = {rating.movie_id for rating in ratings}

# Затем создать случайный вектор в расчете на идентификатор
user_vectors = {user_id: random_tensor(EMBEDDING_DIM)
                for user_id in user_ids}
movie_vectors = {movie_id: random_tensor(EMBEDDING_DIM)
                 for movie_id in movie_ids}
```

К настоящему моменту мы должны уже стать экспертами в написании тренировочных циклов:

```
from typing import List
import tqdm
from scratch.linear_algebra import dot

def loop(dataset: List[Rating],
         learning_rate: float = None) -> None:
    with tqdm.tqdm(dataset) as t:
        loss = 0.0
        for i, rating in enumerate(t):
            movie_vector = movie_vectors[rating.movie_id]
            user_vector = user_vectors[rating.user_id]
            predicted = dot(user_vector, movie_vector)
            error = predicted - rating.rating
            loss += error ** 2

            if learning_rate is not None:
                #     предсказано = m_0 * u_0 + ... + m_k * u_k
                # Поэтому каждый u_j поступает во вход с коэффициентом m_j
                # и каждый m_j поступает во вход с коэффициентом u_j
                user_gradient = [error * m_j for m_j in movie_vector]
                movie_gradient = [error * u_j for u_j in user_vector]

                # Сделать градиентные шаги
                for j in range(EMBEDDING_DIM):
                    user_vector[j] -= learning_rate * user_gradient[j]
                    movie_vector[j] -= learning_rate * movie_gradient[j]

            t.set_description(f"avg loss: {loss / (i + 1)}")
```

И теперь мы можем натренировать нашу модель (т. е. отыскать оптимальные вложения). У меня лучше всего работало, если я немного снижал темп усвоения каждую эпоху:

```
learning_rate = 0.05
for epoch in range(20):
    learning_rate *= 0.9
```

```
    print(epoch, learning_rate)
    loop(train, learning_rate=learning_rate)
    loop(validation)
loop(test)
```

Эта модель довольно склонна к переподгонке к тренировочному набору. Я получил наилучшие результаты с EMBEDDING_DIM=2, что дало мне среднюю потерю на тестовом наборе, равную примерно 0.89.

Если вам нужны были высокоразмерные вложения, то вы можете попробовать регуляризацию *(см разд. "Регуляризация" главы 15)*. В частности, на каждом обновлении градиента вы могли бы ужимать веса в сторону 0. Я не смог получить улучшенных результатов таким образом.

Теперь обследуйте усвоенные векторы. Нет причин ожидать, что эти две компоненты будут особенно содержательными, поэтому мы воспользуемся анализом главных компонент:

```
from scratch.working_with_data import pca, transform

original_vectors = [vector for vector in movie_vectors.values()]
components = pca(original_vectors, 2)
```

Давайте преобразуем наши векторы так, чтобы они представляли главные компоненты, и внесем идентификаторы фильмов и средние рейтинги:

```
ratings_by_movie = defaultdict(list)

for rating in ratings:
    ratings_by_movie[rating.movie_id].append(rating.rating)

vectors = [
    (movie_id,
     sum(ratings_by_movie[movie_id]) / len(ratings_by_movie[movie_id]),
     movies[movie_id],
     vector)
    for movie_id, vector in zip(movie_vectors.keys(),
                                transform(original_vectors, components))
]

# Напечатать верхние 25 и нижние 25 по первой главной компоненте
print(sorted(vectors, key=lambda v: v[-1][0])[:25])
print(sorted(vectors, key=lambda v: v[-1][0])[-25:])
```

Все верхние 25 имеют высокий рейтинг, тогда как нижние 25 в основном имеют низкий рейтинг (либо не имеют ретинга в тренировочных данных). Это значит, что первая главная компонента преимущественно улавливает то, "насколько хорош этот фильм".

Мне трудно понять смысл второй компоненты; и действительно, двумерные вложения показали результаты лишь ненамного лучше, чем одномерные вложения,

говоря о том, что всё, что уловила вторая компонента, возможно, является едва уловимым. (Предположительно, один из более крупных наборов данных MovieLens будет показывать более интересные вещи.)

Для дальнейшего изучения

- Surprise[2] — это библиотека языка Python для "построения и анализа рекомендательных систем", которая, похоже, достаточно популярна и современна.
- Премия Netflix[3] — довольно известный конкурс сети потокового мультимедиа Netflix по созданию усовершенствованной системы рекомендаций фильмов пользователям.

[2] См. **http://surpriselib.com/**.

[3] См. **http://www.netflixprize.com/**.

ГЛАВА 24
Базы данных и SQL

> Память — лучший друг и худший враг человека.
> – *Гилберт Паркер*[1]

Данные, которые вам нужны, нередко размещаются в системах управления базами данных (СУБД), спроектированных для эффективного хранения и извлечения данных. Основная их масса является реляционными, такими как Oracle, MySQL и SQL Server, которые хранят данные в виде таблиц и, как правило, опрашиваются при помощи декларативного языка управления данными SQL (языка структурированных запросов, Structured Query Language).

Язык SQL — существенная часть инструментария исследователя данных. В этой главе мы создадим приложение NotQuiteABase, имплементацию на Python "не совсем базы данных". Мы также рассмотрим основы языка SQL, показывая то, как инструкции работают в нашей "не совсем" базе данных, являющейся простейшим способом демонстрации, который можно было придумать, чтобы помочь вам понять принцип их работы. Надеюсь, что решение задач с помощью NotQuiteABase даст вам хорошее представление о том, как решаются аналогичные задачи при помощи SQL.

Инструкции *CREATE TABLE* и *INSERT*

Реляционная база данных — это коллекция таблиц (и связей между ними). Таблица — это просто коллекция строк, почти как некоторые матрицы, с которыми мы работали ранее. Однако таблица имеет ассоциированную с ней закрепленную *схему*, состоящую из имен и типов столбцов.

Например, представим набор данных `users`, содержащий для каждого пользователя идентификатор `user_id`, имя `name` и число друзей `num_friends`:

```
users = [[0, "Hero", 0],
         [1, "Dunn", 2],
         [2, "Sue", 3],
         [3, "Chi", 3]]
```

На языке SQL такую таблицу можно создать при помощи следующей ниже инструкции:

[1] Гилберт Паркер (1862–1932) — канадский новеллист и английский политик. — *Прим. пер.*

```
CREATE TABLE users (
    user_id INT NOT NULL,
    name VARCHAR(200),
    num_friends INT);
```

Обратите внимание, мы указали, что поля `user_id` и `num_friends` должны быть целыми числами (и `user_id` не может быть `NULL`, которое указывает на отсутствующее значение и несколько похоже на `None` в языке Python) и что имя должно быть строкой длиной не более 200 символов. Мы будем использовать типы Python схожим образом.

Язык SQL практически полностью независим от регистра букв и отступов. Прописные буквы и отступы в данном случае являются моим предпочтительным стилем. Если вы начинаете изучение SQL, то наверняка встретите примеры, которые стилизованы по-другому.

Мы можем вставлять строки с помощью инструкции `INSERT`:

```
INSERT INTO users (user_id, name, num_friends) VALUES (0, 'Hero', 0);
```

Также обратите внимание на то, что инструкции SQL должны заканчиваться точкой с запятой и SQL требует одинарные кавычки для строковых значений.

В приложении NotQuiteABase вы создадите экземпляр класса `Table`, указав схожую схему. Затем для вставки строки вы будете использовать метод `insert` таблицы, принимающий список значений, которые должны быть в том же порядке, что и имена столбцов.

За кулисами мы будем хранить каждую строку как словарь отображений из имен столбцов в значения. Реальная СУБД никогда бы не использовала такое неэффективное по занимаемой памяти представление, однако поступая так, мы намного упрощаем работу с приложением NotQuiteABase.

Мы выполним имплементацию таблицы `Table` приложения NotQuiteABase как гигантский класс, в котором мы будем имплементировать один метод за раз. Давайте начнем с того, что разделаемся с несколькими импортами и псевдонимами типов:

```
from typing import Tuple, Sequence, List, Any, Callable, Dict, Iterator
from collections import defaultdict

# Несколько псевдонимов типов, которые будут использоваться позже
Row = Dict[str, Any]                        # Строка базы данных
WhereClause = Callable[[Row], bool]         # Предикат для единственной строки
HavingClause = Callable[[List[Row]], bool] # Предикат над многочисленными строками
```

Начнем с конструктора. Для создания таблицы приложения NotQuiteABase нам нужно передать список имен столбцов и список типов столбцов, как если бы вы создавали таблицу в СУБД SQL:

```
class Table:
    def __init__(self, columns: List[str], types: List[type]) -> None:
        assert len(columns) == len(types), "число столбцов должно == числу типов"
```

```
        self.columns = columns      # Имена столбцов
        self.types = types          # Типы данных в столбцах
        self.rows: List[Row] = []   # (данных пока нет)
```

Мы добавим вспомогательный метод, который будет получать тип столбца:

```
    def col2type(self, col: str) -> type:
        idx = self.columns.index(col)  # Отыскать индекс столбца
        return self.types[idx]         # и вернуть его тип
```

И мы добавим метод `insert`, который проверяет допустимость вставляемых значений. В частности, необходимо указать правильное число значений, и каждое должно иметь правильный тип (или `None`):

```
    def insert(self, values: list) -> None:
        # Проверить, правильное ли число значений
        if len(values) != len(self.types):
            raise ValueError(f"Требуется {len(self.types)} значений")

        # Проверить допуститмость типов значений
        for value, typ3 in zip(values, self.types):
            if not isinstance(value, typ3) and value is not None:
                raise TypeError(f"Ожидаемый тип {typ3}, но получено {value}")

        # Добавить соответствующий словарь как "строку"
        self.rows.append(dict(zip(self.columns, values)))
```

В фактической СУБД SQL вы бы явно указали, разрешено ли какому-либо столбцу содержать значения `null` (`None`); ради упрощения нашей жизни мы просто скажем, что любой столбец может содержать это значение.

Мы также представим несколько дандерных методов[2], которые позволяют трактовать таблицу как `List[Row]`. Мы их будем использовать в основном для тестирования нашего кода:

```
    def __getitem__(self, idx: int) -> Row:
        return self.rows[idx]

    def __iter__(self) -> Iterator[Row]:
        return iter(self.rows)

    def __len__(self) -> int:
        return len(self.rows)
```

И мы добавим метод структурированной печати нашей таблицы:

```
    def __repr__(self):
        """Структурированное представление таблицы:
           столбцы затем строки"""
```

[2] Дандерными называются все имена, которые в контексте класса начинаются с двух символов подчеркивания (double under = dunder). — *Прим. пер.*

```
        rows = "\n".join(str(row) for row in self.rows)
        return f"{self.columns}\n{rows}"
```

Например, мы можем создать нашу таблицу `Users`:

```
# Конструктор требует имена и типы столбцов
users = Table(['user_id', 'name', 'num_friends'], [int, str, int])
users.insert([0, "Hero", 0])
users.insert([1, "Dunn", 2])
users.insert([2, "Sue", 3])
users.insert([3, "Chi", 3])
users.insert([4, "Thor", 3])
users.insert([5, "Clive", 2])
users.insert([6, "Hicks", 3])
users.insert([7, "Devin", 2])
users.insert([8, "Kate", 2])
users.insert([9, "Klein", 3])
users.insert([10, "Jen", 1])
```

Если теперь вы напечаете `print(users)`, то получите:

```
['user_id', 'name', 'num_friends']
{'user_id': 0, 'name': 'Hero', 'num_friends': 0}
{'user_id': 1, 'name': 'Dunn', 'num_friends': 2}
{'user_id': 2, 'name': 'Sue', 'num_friends': 3}
...
```

Спископодобный API упрощает написание тестов:

```
assert len(users) == 11
assert users[1]['name'] == 'Dunn'
```

Нам предстоит добавить гораздо больше функционала.

Инструкция *UPDATE*

Иногда нужно обновить данные, которые уже есть в базе данных. Например, если Dunn приобретает еще одного друга, то вам, возможно, потребуется сделать следующее:

```
UPDATE users
SET num_friends = 3
WHERE user_id = 1;
```

Ключевые свойства следующие:

- какую таблицу обновлять;
- какие строки обновлять;
- какие поля обновлять;
- какими должны быть их новые значения.

Мы добавим в NotQuiteABase похожий метод `update`. Его первым аргументом будет словарь `dict`, ключами которого являются столбцы для обновления, а значениями — новые значения для этих полей. Вторым (необязательным) аргументом должна быть предикативная функция `predicate`, которая возвращает `True` для строк, которые должны быть обновлены, и `False` — в противном случае:

```
def update(self,
           updates: Dict[str, Any],
           predicate: WhereClause = lambda row: True):
    # Сначала убедиться, что обновления
    # имеют допустимые имена и типы
    for column, new_value in updates.items():
        if column not in self.columns:
            raise ValueError(f"недопустимый столбец: {column}")

        typ3 = self.col2type(column)
        if not isinstance(new_value, typ3) and new_value is not None:
            raise TypeError(f"ожидаемый тип {typ3}, но получено {new_value}")

    # Теперь обновить
    for row in self.rows:
        if predicate(row):
            for column, new_value in updates.items():
                row[column] = new_value
```

После чего мы можем просто сделать следующее:

```
assert users[1]['num_friends'] == 2          # Исходное значение

users.update({'num_friends' : 3},            # Назначить num_friends = 3
             lambda row: row['user_id'] == 1)  # в строках, где user_id == 1

assert users[1]['num_friends'] == 3          # Обновить значение
```

Инструкция *DELETE*

Есть два способа удаления строк из таблицы в SQL. Опасный способ удаляет все строки таблицы:

```
DELETE FROM users;
```

Менее опасный способ добавляет условное выражение `WHERE` и удаляет только те строки, которые удовлетворяют определенному условию:

```
DELETE FROM users WHERE user_id = 1;
```

Очень легко добавить этот функционал в нашу таблицу `Table`:

```
def delete(self, predicate: WhereClause = lambda row: True) -> None:
    """Удалить все строки, совпадающие с предикатом"""
    self.rows = [row for row in self.rows if not predicate(row)]
```

Если вы предоставите предикативную функцию (т. е. выражение WHERE), то будут удалены только те строки, которые удовлетворяют условию. Если вы не предоставите ее, то принятая по умолчанию предикативная функция всегда возвращает True, и вы удалите все строки.

Например:

```
# Мы на самом деле не собираемся использовать эти строки кода
users.delete(lambda row: row["user_id"] == 1)  # Удаляет строки с user_id == 1
users.delete()                                 # Удаляет все строки
```

Инструкция *SELECT*

Как правило, таблицы SQL целиком не инспектируются. Вместо этого из них выбирают данные с помощью инструкции SELECT:

```
SELECT * FROM users;           -- Получить все содержимое
SELECT * FROM users LIMIT 2;   -- Получить первые две строки
SELECT user_id FROM users;     -- Получить только конкретные столбцы
SELECT user_id FROM users WHERE name = 'Dunn';  -- Получить конкретные строки
```

Вы также можете использовать инструкцию SELECT для вычисления полей:

```
SELECT LENGTH(name) AS name_length FROM users;
```

Мы дадим нашему классу Table метод select, который возвращает новую таблицу. Указанный метод принимает два необязательных аргумента:

- keep_columns задает имена столбцов, которые вы хотите сохранить в результирующей таблице. Если он не указан, то результат будет содержать все столбцы;
- additional_columns — словарь, ключами которого являются новые имена столбцов, а значения — функциями, определяющими порядок вычисления значений новых столбцов.

Если вы не предоставите ни один из них, то просто получите копию таблицы:

```
def select(self,
           keep_columns: List[str] = None,
           additional_columns: Dict[str, Callable] = None) -> 'Table':

    if keep_columns is None:         # Если ни один столбец не указан,
        keep_columns = self.columns  # то вернуть все столбцы

    if additional_columns is None:
        additional_columns = {}

    # Имена и типы новых столбцов
    new_columns = keep_columns + list(additional_columns.keys())
    keep_types = [self.col2type(col) for col in keep_columns]
```

```
    # Вот как получить возвращаемый тип из аннотации типа.
    # Это даст сбой, если "вычисление" не имеет возвращаемого типа
    add_types = [calculation.__annotations__['return']
                 for calculation in additional_columns.values()]

    # Создать новую таблицу для результатов
    new_table = Table(new_columns, keep_types + add_types)

    for row in self.rows:
        new_row = [row[column] for column in keep_columns]
        for column_name, calculation in additional_columns.items():
            new_row.append(calculation(row))
        new_table.insert(new_row)

    return new_table
```

Помните, как в *главе 2* мы говорили, что аннотации типов на самом деле ничего не делают? Вот вам контрпример. Но взгляните на запутанную процедуру, которую мы должны пройти для того, чтобы добраться до них.

Наш метод `select` возвращает новую таблицу `Table`, тогда как типичная SQL-инструкция `SELECT` всего лишь производит некий промежуточный результирующий набор (если только вы явно не вставите результаты в таблицу).

Нам также потребуются методы `where` и `limit`. Оба достаточно просты:

```
def where(self, predicate: WhereClause = lambda row: True) -> 'Table':
    """Вернуть только строки, которые
       удовлетворяют переданному предикату"""
    where_table = Table(self.columns, self.types)
    for row in self.rows:
        if predicate(row):
            values = [row[column] for column in self.columns]
            where_table.insert(values)
    return where_table

def limit(self, num_rows: int) -> 'Table':
    """Вернуть только первые `num_rows` строк"""
    limit_table = Table(self.columns, self.types)
    for i, row in enumerate(self.rows):
        if i >= num_rows:
            break
        values = [row[column] for column in self.columns]
        limit_table.insert(values)
    return limit_table
```

После чего вы легко можете сконструировать эквиваленты приложения NotQuiteABase для приведенных выше инструкций SQL:

```
# SELECT * FROM users;
all_users = users.select()
assert len(all_users) == 11

# SELECT * FROM users LIMIT 2;
two_users = users.limit(2)
assert len(two_users) == 2

# SELECT user_id FROM users;
just_ids = users.select(keep_columns=["user_id"])
assert just_ids.columns == ['user_id']

# SELECT user_id FROM users WHERE name = 'Dunn';
dunn_ids = (
    users
    .where(lambda row: row["name"] == "Dunn")
    .select(keep_columns=["user_id"])
)
assert len(dunn_ids) == 1
assert dunn_ids[0] == {"user_id": 1}

# SELECT LENGTH(name) AS name_length FROM users;
def name_length(row) -> int: return len(row["name"])

name_lengths = users.select(keep_columns=[],
                            additional_columns = {"name_length": name_length})
assert name_lengths[0]['name_length'] == len("Hero")
```

Обратите внимание, что в случае многострочных "подвижных" запросов мы должны обрамлять весь запрос скобками.

Инструкция *GROUP BY*

Еще одной распространенной инструкцией SQL является `GROUP BY`, которая группирует строки с одинаковыми значениями в указанных полях и применяет агрегатные функции, такие как `MIN`, `MAX`, `COUNT` и `SUM`.

Например, вы, возможно, захотите отыскать число пользователей и наименьший идентификатор `user_id` для каждой возможной длины имени:

```
SELECT LENGTH(name) AS name_length,
       MIN(user_id) AS min_user_id,
       COUNT(*)     AS num_users
FROM users
GROUP BY LENGTH(name);
```

Каждое поле, которое мы выбираем инструкцией `SELECT`, должно находиться либо в выражении `GROUP BY` (каким является `name_length`), либо быть агрегатным вычислением (которыми являются `min_user_id` и `num_users`).

Язык SQL также поддерживает условное выражение HAVING, которое ведет себя аналогично выражению WHERE, за исключением того, что его фильтр применяется к агрегатам (тогда как выражение WHERE фильтрует строки до того, как состоялось агрегирование).

Вы, возможно, захотите узнать среднее число друзей для пользователей, чьи имена начинаются с определенных букв, но увидеть только результаты для тех букв, чье соответствующее среднее больше 1. (Признаюсь, что некоторые примеры надуманны.)

```
SELECT SUBSTR(name, 1, 1) AS first_letter,
       AVG(num_friends)   AS avg_num_friends
FROM users
GROUP BY SUBSTR(name, 1, 1)
HAVING AVG(num_friends) > 1;
```

Функции для работы со строковыми значениями разнятся от имплементации к имплементации SQL; вместо этого в некоторых СУБД может использоваться функция SUBSTRING либо что-то в этом роде.

Вы также можете вычислять совокупные агрегатные величины. В этом случае мы опускаем выражение GROUP BY:

```
SELECT SUM(user_id) AS user_id_sum
FROM users
WHERE user_id > 1;
```

Для того чтобы добавить этот функционал в таблицы Table приложения NotQuiteABase, мы напишем метод group_by. Он принимает имена столбцов, по которым нужна группировка, словарь агрегирующих функций, которые вы хотите выполнить на каждой группе, и необязательный предикат having, который работает на нескольких строках.

Затем он выполняет следующие шаги:

1. Создает словарь defaultdict для отображения кортежей tuple (со значениями group-by, по которым группировать) в строки (содержащие значения group-by, по которым группировать). Напомним, что использовать списки в качестве ключей словаря dict нельзя; следует использовать кортежи.
2. Перебирает строки таблицы, заполняя словарь defaultdict.
3. Создает новую таблицу с правильными столбцами на выходе.
4. Перебирает словарь defaultdict и заполняет результирующую таблицу, применяя фильтр having, если он имеется.

```
def group_by(self,
             group_by_columns: List[str],
             aggregates: Dict[str, Callable],
             having: HavingClause = lambda group: True) -> 'Table':

    grouped_rows = defaultdict(list)
```

```
        # Заполнить группы
        for row in self.rows:
            key = tuple(row[column] for column in group_by_columns)
            grouped_rows[key].append(row)

        # Результирующая таблица состоит
        # из столбцов group_by и агрегатов
        new_columns = group_by_columns + list(aggregates.keys())
        group_by_types = [self.col2type(col)
                          for col in group_by_columns]
        aggregate_types = [agg.__annotations__['return']
                           for agg in aggregates.values()]
        result_table = Table(new_columns,
                             group_by_types + aggregate_types)

        for key, rows in grouped_rows.items():
            if having(rows):
                new_row = list(key)
                for aggregate_name, aggregate_fn in aggregates.items():
                    new_row.append(aggregate_fn(rows))
                result_table.insert(new_row)

        return result_table
```

Фактическая СУБД почти наверняка сделает это более эффективным образом.

Опять-таки давайте посмотрим, как запрограммировать эквивалент приведенных выше инструкций SQL. Метрики длины `name_length` следующие:

```
def min_user_id(rows) -> int:
    return min(row["user_id"] for row in rows)

def length(rows) -> int:
    return len(rows)

stats_by_length = (
    users
    .select(additional_columns={"name_length" : name_length})
    .group_by(group_by_columns=["name_length"],
              aggregates={"min_user_id" : min_user_id,
                          "num_users" : length})
)
```

Метрики для первой буквы `first_letter`:

```
def first_letter_of_name(row: Row) -> str:
    return row["name"][0] if row["name"] else ""
```

```
def average_num_friends(rows: List[Row]) -> float:
    return sum(row["num_friends"] for row in rows) / len(rows)

def enough_friends(rows: List[Row]) -> bool:
    return average_num_friends(rows) > 1

avg_friends_by_letter = (
    users
    .select(additional_columns={'first_letter' : first_letter_of_name})
    .group_by(group_by_columns=['first_letter'],
              aggregates={"avg_num_friends" : average_num_friends},
              having=enough_friends)
)
```

и сумма `user_id_sum`:

```
def sum_user_ids(rows: List[Row]) -> int:
    return sum(row["user_id"] for row in rows)

user_id_sum = (
    users
    .where(lambda row: row["user_id"] > 1)
    .group_by(group_by_columns=[],
              aggregates={ "user_id_sum" : sum_user_ids })
)
```

Инструкция *ORDER BY*

Нередко требуется отсортировать результаты. Например, вы, возможно, захотите знать первые два имени ваших пользователей (в алфавитном порядке):

```
SELECT * FROM users
ORDER BY name
LIMIT 2;
```

Это легко имплементировать, предоставив нашему классу `Table` метод `order_by`, который принимает функцию `order`:

```
def order_by(self, order: Callable[[Row], Any]) -> 'Table':
    new_table = self.select() # make a copy
    new_table.rows.sort(key=order)
    return new_table
```

которую мы затем можем применить следующим образом:

```
friendliest_letters = (
    avg_friends_by_letter
    .order_by(lambda row: -row["avg_num_friends"])
    .limit(4)
)
```

SQL-инструкция ORDER BY позволяет указывать порядок сортировки ASC (в возрастающем порядке) или DESC (в убывающем порядке) для каждого сортируемого поля; здесь мы должны будем "запечь" все это в функции order.

Инструкция *JOIN*

Таблицы реляционных СУБД часто *нормализуются*, т. е. их организуют так, чтобы уменьшить избыточность. Например, когда мы работаем с интересами наших пользователей на языке Python, мы можем предоставить каждому пользователю список с его интересами.

Таблицы SQL, как правило, не могут содержать списки, поэтому типичным решением является создание второй таблицы user_interests, содержащей отношение "один-ко-многим" между идентификаторами пользователей user_id и интересами interests. На SQL вы могли бы сделать это так:

```
CREATE TABLE user_interests (
    user_id INT NOT NULL,
    interest VARCHAR(100) NOT NULL
);
```

Тогда как в приложении NotQuiteABase нам придется создать таблицу:

```
user_interests = Table(['user_id', 'interest'], [int, str])
user_interests.insert([0, "SQL"])
user_interests.insert([0, "NoSQL"])
user_interests.insert([2, "SQL"])
user_interests.insert([2, "MySQL"])
```

Много избыточности по-прежнему остается — интерес "SQL" хранится в двух разных местах. В реальной СУБД вы могли бы хранить идентификаторы user_id и interest_id в таблице user_interests, а затем создать третью таблицу interests, увязав поля interest_id с interest, чтобы хранить названия интересов только в одном экземпляре. Здесь это неоправданно усложнило бы наши примеры.

Когда наши данные разбросаны по разным таблицам, то как их анализировать? Это делается путем объединения таблиц на основе инструкции JOIN. Она сочетает строки в левой таблице с соответствующими строками в правой таблице, где значение слова "соответствующий" зависит от того, как задается объединение.

Например, чтобы отыскать пользователей, заинтересованных в SQL, вы бы сделали следующий ниже запрос:

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

Инструкция JOIN говорит, что для каждой строки в таблице users нам следует обратиться к идентификатору user_id и ассоциировать эту строку с каждой строкой в таблице user_interests, содержащей тот же самый идентификатор user_id.

Обратите внимание, что мы должны были указать, какие таблицы объединять (JOIN) и по каким столбцам (ON). Это так называемая *инструкция внутреннего объединения* (INNER JOIN), которая возвращает сочетания (и только те сочетания) строк, соответствующих заданным критериям объединининения.

Кроме того, есть инструкция левого объединения LEFT JOIN, которая в дополнение к сочетанию совпадающих строк возвращает строку для каждой строки левой таблицы с несовпадающими строками (в этом случае все поля, которые приходят из правой таблицы, равны NULL).

Используя инструкцию LEFT JOIN, легко подсчитать число интересов у каждого пользователя:

```
SELECT users.id, COUNT(user_interests.interest) AS num_interests
FROM users
LEFT JOIN user_interests
ON users.user_id = user_interests.user_id
```

Операция левого объединения LEFT JOIN обеспечивает, чтобы пользователи с отсутствием интересов по-прежнему имели строки в объединенном наборе данных (с пустыми значениями полей для полей из user_interests), и функция COUNT подсчитывает только значения, которые не являются пустыми, т. е. не равны NULL.

Имплементация функции join в приложении NotQuiteABase будет более ограниченной — она просто объединит две таблицы по любым общим столбцам. Но даже в таком виде написать ее — задача не из тривиальных:

```
def join(self, other_table: 'Table',
         left_join: bool = False) -> 'Table':

    join_on_columns = [c for c in self.columns        # Столбцы
                       if c in other_table.columns]   # в обеих
                                                      # таблицах

    additional_columns = [c for c in other_table.columns  # Столбцы
                                                          # только
                          if c not in join_on_columns]    # в правой
                                                          # таблице
    # Все столбцы из левой таблицы + дополнительные
    # additional_columns из правой
    new_columns = self.columns + additional_columns
    new_types = self.types + [other_table.col2type(col)
                              for col in additional_columns]

    join_table = Table(self.columns + additional_columns)
```

```
        for row in self.rows:
            def is_join(other_row):
                return all(other_row[c] == row[c]
                           for c in join_on_columns)

            other_rows = other_table.where(is_join).rows

            # Каждая строка, которая удовлетворяет предикату,
            # производит результирующую строку
            for other_row in other_rows:
                join_table.insert([row[c] for c in self.columns] +
                                  [other_row[c] for c
                                  in additional_columns])

            # Если ни одна строка не удовлетворяет условию и это
            # объединение left join, то выдать строку со значениями None
            if left_join and not other_rows:
                join_table.insert([row[c] for c in self.columns] +
                                  [None for c in additional_columns])
        return join_table
```

Таким образом, мы могли бы отыскать пользователей, заинтересованных в SQL, с помощью:

```
sql_users = (
    users
    .join(user_interests)
    .where(lambda row: row["interest"] == "SQL")
    .select(keep_columns=["name"])
)
```

И мы могли бы получить количества интересов с помощью:

```
def count_interests(rows):
    """Подсчитывает количество строк с ненулевыми интересами"""
    return len([row for row in rows if row["interest"] is not None])

user_interest_counts = (
    users
    .join(user_interests, left_join=True)
    .group_by(group_by_columns=["user_id"],
              aggregates={"num_interests" : count_interests })
)
```

В языке SQL есть еще *правое объединение* `RIGHT JOIN`, которое хранит строки из правой таблицы, не имеющей совпадений, и *полное внешнее объединение* `FULL OUTER JOIN`, которое хранит строки из обеих таблиц, не имеющих совпадений. Они останутся без имплементации.

Подзапросы

В SQL вы можете делать выборку (инструкцией `SELECT`) из (объединенных инструкцией `JOIN`) результатов запросов, как если бы они были таблицами. Так, если бы вы хотели найти наименьший идентификатор `user_id` из тех, кто заинтересован в SQL, то могли бы использовать *подзапрос*. (Естественно, вы могли бы сделать то же самое вычисление с помощью инструкции `JOIN`, но тогда не будут проиллюстрированы подзапросы.)

```
SELECT MIN(user_id) AS min_user_id FROM
(SELECT user_id FROM user_interests WHERE interest = 'SQL') sql_interests;
```

С учетом того, как нами разработано приложение NotQuiteABase, мы получаем это бесплатно. (Наши результаты запросов являются фактическими таблицами.)

```
likes_sql_user_ids = (
    user_interests
    .where(lambda row: row["interest"] == "SQL")
    .select(keep_columns=['user_id'])
)

likes_sql_user_ids.group_by(group_by_columns=[],
                            aggregates={ "min_user_id" : min_user_id })
```

Индексы

Для того чтобы отыскать строки, содержащие конкретное значение (допустим, где есть имя "Hero"), приложение NotQuiteABase должно обследовать каждую строку в таблице. Если таблица имеет много строк, то на это может уйти очень много времени.

Наш алгоритм объединения `join` схожим образом крайне неэффективен. Для каждой строки в левой таблице он обследует каждую строку в правой таблице в поисках совпадений. В случае двух крупных таблиц это могло бы занять вечность.

Нередко вам также требуется накладывать ограничения на некоторые ваши столбцы. Например, в таблице `users` вы, возможно, не захотите разрешить двум разным пользователям иметь одинаковый идентификатор `user_id`.

Индексы решают все эти задачи. Если бы таблица `user_interests` имела индекс по полю `user_id`, то умный алгоритм `join` отыскал бы совпадения непосредственно, а не в результате сканирования всей таблицы. Если бы таблица `users` имела "уникальный" индекс по полю `user_id`, то вы бы получили ошибку, если бы попытались вставить дубликат.

Каждая таблица в базе данных может иметь один или несколько индексов, которые позволяют выполнять быстрый просмотр таблиц по ключевым столбцам, эффективно объединять таблицы и накладывать уникальные ограничения на столбцы или их сочетания.

Конструирование и применение индексов часто относят скорее к черному искусству (которое разнится в зависимости от конкретной СУБД), но если вам придется в дальнейшем выполнять много работы с базой данных, то оно заслуживает того, чтобы изучить его как следует.

Оптимизация запросов

Вспомните запрос для отыскания всех пользователей, которые интересуются SQL:

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

В приложении NotQuiteABase существуют (по крайней мере) два разных способа написать этот запрос. Вы могли бы отфильтровать таблицу `user_interests` перед тем, как выполнить объединение:

```
(
    user_interests
    .where(lambda row: row["interest"] == "SQL")
    .join(users)
    .select(["name"])
)
```

Либо вы могли бы отфильтровать результаты объединения:

```
(
    user_interests
    .join(users)
    .where(lambda row: row["interest"] == "SQL")
    .select(["name"])
)
```

В обоих случаях вы придете к одинаковым результатам, но фильтрация до объединения эффективнее, поскольку в этом случае функция `join` имеет гораздо меньше строк, которыми она будет оперировать.

В SQL можно вообще не беспокоиться об этом. Вы "объявляете" результаты, которые хотите получить, и оставляете механизму выполнения запросов сделать всё остальное (и эффективно задействовать индексы).

Базы данных NoSQL

Новейший тренд в СУБД движется в сторону нереляционных (слабоструктурированных) СУБД "NoSQL", в которых данные не представлены таблицами. Например, MongoDB — это популярная бессхемная СУБД, элементами которой являются не строки таблиц, а произвольные многосложные документы в формате JSON.

Существуют столбцовые СУБД, которые хранят данные в столбцах вместо строк (хорошо подходят, когда данные имеют много столбцов, а запросы выполняются только к нескольким из них); хранилища данных в формате "ключ-значение", которые оптимальны для получения единого (комплексного) значения по ключу; графовые базы данных для хранения и обхода графов; базы данных, оптимизированные для работы между многочисленными центрами обработки данных; базы данных, предназначенные для работы в оперативной памяти; базы данных для хранения данных временных рядов и сотни других.

Характерное очертание на ближайшее будущее, возможно, еще не оформилось, поэтому я не могу сделать больше, кроме как сообщить, что СУБД NoSQL существуют. Так что теперь вы знаете, что это свершившийся факт.

Для дальнейшего изучения

- Если вы хотели бы скачать реляционную СУБД для экспериментирования, то для этого подойдет быстрая и миниатюрная СУБД SQLite[3]. Кроме того, есть более крупные и с бóльшим набором характеристик СУБД MySQL[4] и PostgreSQL[5]. Все они бесплатные и располагают обширной документацией.
- Если вы хотите разведать СУБД NoSQL, лучше всего начать с простой MongoDB[6], которая может стать как благословением, так и своего рода проклятием. Она также имеет достаточно хорошую документацию.
- Статья в Википедии, посвященная NoSQL[7], почти наверняка сейчас содержит ссылки на СУБД, которые даже не существовали на момент написания этой книги.

[3] См. **http://www.sqlite.org/**.

[4] См. **http://www.mysql.com/**.

[5] См. **http://www.postgresql.org/**.

[6] См. **http://www.mongodb.org/**.

[7] См. **http://en.wikipedia.org/wiki/NoSQL**.

ГЛАВА 25
Алгоритм MapReduce

> Будущее уже наступило. Просто оно еще не распределено равномерно.
> – *Уильям Гибсон*[1]

MapReduce — это вычислительная модель для выполнения параллельной обработки крупных наборов данных. Несмотря на то, что она представляет собой мощное техническое решение, база этой модели является достаточно простой.

Предположим, что у нас есть коллекция элементов, которые надо как-то обработать. Например, элементами могли бы быть журналы операций веб-сайта, тексты разнообразных книг, файлы фотоснимков или что угодно еще. Базовая версия алгоритма MapReduce состоит из следующих шагов:

1. Применить функцию трансформации `mapper`, которая превращает каждый элемент в ноль или более пар "ключ-значение". (Нередко эта функция называется `map`, однако в Python уже есть функция с таким названием, и нет никакой необходимости их смешивать.)
2. Собрать вместе все пары с идентичными ключами.
3. Применить функцию редукции `reducer` к каждой коллекции сгруппированных значений, произведя выходные значения для соответствующего ключа.

MapReduce — это своего рода прошлое, настолько, что сначала я решил удалить эту главу из второго издания. Но потом все-таки принял решение, что эта тема все еще вызывает интерес, поэтому в итоге ее оставил (что очевидно).

Все это выглядит абстрактно, поэтому обратимся к конкретному примеру. В науке о данных существует несколько абсолютных правил, и одно из них заключается в том, что первый пример работы алгоритма MapReduce должен быть связан с подсчетом количества появлений слов.

Пример: подсчет количества появлений слов

Социальная сеть DataSciencester выросла до миллионов пользователей! Великолепная гарантия обеспеченности работой; правда, это несколько осложняет выполнение рутинных обязанностей по анализу данных.

[1] Уильям Форд Гибсон (род. 1948) — американский писатель-фантаст. Считается основателем стиля киберпанк, определившего жанровое лицо литературы 1980-х гг. — *Прим. пер.*

К примеру, директор по контенту хотел бы знать, о чем люди пишут в своих постах, размещаемых в своей ленте новостей. В качестве первой попытки вы решаете подсчитать количества появлений слов, которые встречаются в постах, чтобы подготовить отчет о наиболее часто употребляемых словах.

Когда у вас несколько сотен пользователей, это сделать просто:

```
from typing import List
from collections import Counter

def tokenize(document: str) -> List[str]:
    """Просто разбить по пробелу"""
    return document.split()

def word_count_old(documents: List[str]):
    """Подсчет количеств появлений слов
       без использования алгоритма MapReduce"""
    return Counter(word
        for document in documents
        for word in tokenize(document))
```

Но когда пользователей — миллионы, набор документов `documents` (обновлений в ленте новостей) неожиданно становится слишком большим для того, чтобы поместиться на одном компьютере. Если вам удастся как-то разместить их в модели MapReduce, то вы сможете воспользоваться инфраструктурой "больших данных", которую имплементировали ваши инженеры.

Прежде всего нам нужна функция, которая превращает документ в последовательность пар "ключ-значение". Мы хотим, чтобы наш выход был сгруппирован по слову, а значит, ключи должны быть словами. И для каждого слова мы будем эмитировать значение 1, указывая на то, что эта пара соответствует одному появлению слова:

```
from typing import Iterator, Tuple

def wc_mapper(document: str) -> Iterator[Tuple[str, int]]:
    """Для каждого слова в документе эмитировать (слово, 1)"""
    for word in tokenize(document):
        yield (word, 1)
```

Временно пропуская "механику" реализации 2-го этапа, представьте, что для некоторого слова мы собрали список соответствующих количеств, которые мы эмитировали. Тогда для получения совокупного количества для этого слова нам нужно только выполнить редукцию:

```
from typing import Iterable

def wc_reducer(word: str,
               counts: Iterable[int]) -> Iterator[Tuple[str, int]]:
    """Просуммировать количества появлений для слова"""
    yield (word, sum(counts))
```

Вернемся ко 2-му этапу. Теперь нам нужно собрать результаты работы преобразователя `wc_mapper` и передать их в редуктор `wc_reducer`. Давайте подумаем, как это сделать на одном-единственном компьютере:

```
from collections import defaultdict

def word_count(documents: List[str]) -> List[Tuple[str, int]]:
    """Подсчитать количества появлений слов в выходных документах,
    используя алгоритм MapReduce"""
    collector = defaultdict(list) # Сохранить сгруппированные значения

    for document in documents:
        for word, count in wc_mapper(document):
            collector[word].append(count)

    return [output
            for word, counts in collector.items()
            for output in wc_reducer(word, counts)]
```

Представим, что у нас есть три документа: `["data science", "big data", "science fiction"]`.

Тогда преобразователь `wc_mapper`, примененный к первому документу, выдаст две пары `("data", 1)` и `("science", 1)`. После прохождения через все три документа переменная `collector` будет содержать:

```
{ "data" : [1, 1],
  "science" : [1, 1],
  "big" : [1],
  "fiction" : [1] }
```

Затем редуктор `wc_reducer` произведет количества появлений для каждого слова:

```
[("data", 2), ("science", 2), ("big", 1), ("fiction", 1)]
```

Почему алгоритм MapReduce?

Как уже упоминалось ранее, основное преимущество вычислительной модели MapReduce заключается в том, что она позволяет нам распределять вычисления, перемещая обработку в сторону данных. Представим, что мы хотим подсчитать количества появлений слов среди миллиардов документов.

Наш первоначальный подход (без алгоритма MapReduce) требует, чтобы выполняющая обработку машина имела доступ к любому документу. Это означает, что все документы должны либо располагаться на этой машине, либо передаваться ей во время обработки. И при этом, что представляется более важным, машина может обрабатывать только один документ за один раз.

Возможно даже, что она способна обрабатывать до нескольких документов одновременно, если у нее несколько ядер и если код переписан для того, чтобы воспользоваться ими. Но даже в этом случае все документы все равно должны *добраться* до этой машины.

Теперь представьте, что наши миллиарды документов разбросаны между 100 машинами. При наличии правильной инфраструктуры (и умалчивая о некоторых подробностях) мы можем сделать следующее:

1. Каждая машина пропускает через преобразователь свои документы, производя много пар "ключ-значение".
2. Нужно распределить эти пары "ключ-значение" среди определенного числа редуцирующих машин, обеспечивая, чтобы все пары, соответствующие любому заданному ключу, оставались на той же машине.
3. Каждая редуцирующая машина группирует пары по ключу и затем пропускает через редуктор каждый набор значений.
4. Вернуть каждую пару (ключ, выход).

Самое удивительное заключается в том, что это масштабируется по горизонтали. Если удвоить число машин, то (игнорируя определенные фиксированные затраты на работу системы MapReduce) наши вычисления должны выполняться примерно в два раза быстрее. Каждой преобразующей машине придется проделывать только половину работы, и (допустив, что имеется достаточно разных ключей для дальнейшего распределения работы редукторов) то же самое относится к редуцирующим машинам.

Алгоритм MapReduce в более общем плане

Если вы подумаете немного, то весь специфичный для подсчета количеств появлений слов исходный код в приведенном выше примере содержится в функциях `wc_mapper` и `wc_reducer`. Это означает, что внеся пару изменений, мы получим намного более общий каркас (который по-прежнему выполняется на одной-единственной машине).

Мы могли бы использовать обобщенные типы, полностью аннотировав нашу функцию `map_reduce`, но это создаст своего рода беспорядок в педагогическом плане, поэтому в этой главе мы гораздо небрежнее отнесемся к нашим аннотациям типов:

```
from typing import Callable, Iterable, Any, Tuple

# Пара ключ/значение - это просто 2-членный кортеж
KV = Tuple[Any, Any]

# Преобразователь - это функция, которая возвращает
# итерируемый объект Iterable, состоящий из пар ключ/значение
Mapper = Callable[..., Iterable[KV]]

# Редуктор - это функция, которая берет ключ
# и итерируемый объект со значениями
# и возвращает пару ключ/значение
Reducer = Callable[[Any, Iterable], KV]
```

Теперь мы можем написать обобщенную функцию map_reduce:

```
def map_reduce(inputs: Iterable,
               mapper: Mapper,
               reducer: Reducer) -> List[KV]:
    """Пропустить входы через MapReduce,
       используя преобразователь и редуктор"""
    collector = defaultdict(list)

    for input in inputs:
        for key, value in mapper(input):
            collector[key].append(value)

    return [output
            for key, values in collector.items()
            for output in reducer(key, values)]
```

Тогда мы можем подсчитать количества появлений слов, просто применив:

```
word_counts = map_reduce(documents, wc_mapper, wc_reducer)
```

Это предоставляет нам гибкость при решении широкого круга задач.

Прежде чем продолжить, обратите внимание, что функция wc_reducer просто суммирует значения, соответствующие каждому ключу. Этот вид агрегирования распространен настолько, что он заслуживает абстрагирования:

```
def values_reducer(values_fn: Callable) -> Reducer:
    """Вернуть редуктор, который просто применяет функцию
       values_fn к своим значениям"""
    def reduce(key, values: Iterable) -> KV:
        return (key, values_fn(values))

    return reduce
```

После чего мы можем легко создать:

```
sum_reducer = values_reducer(sum)
max_reducer = values_reducer(max)
min_reducer = values_reducer(min)
count_distinct_reducer = values_reducer(lambda values: len(set(values)))

assert sum_reducer("key", [1, 2, 3, 3]) == ("key", 9)
assert min_reducer("key", [1, 2, 3, 3]) == ("key", 1)
assert max_reducer("key", [1, 2, 3, 3]) == ("key", 3)
assert count_distinct_reducer("key", [1, 2, 3, 3]) == ("key", 3)
```

И так далее.

Пример: анализ обновлений новостной ленты

Директор по контенту был впечатлен отчетом о количествах употреблений слов и интересуется, что еще можно узнать из обновлений новостной ленты? Вам удалось извлечь набор обновлений, который выглядит примерно так:

```
status_updates = [
    {"id": 2,
     "username" : "joelgrus",
     "text" : "Should I write a second edition of my data science book?",
     "created_at" : datetime.datetime(2018, 2, 21, 11, 47, 0),
     "liked_by" : ["data_guy", "data_gal", "mike"] },
     # ...
]
```

Скажем, нам нужно выяснить, в какой день недели пользователи чаще всего говорят о науке о данных. Для того чтобы узнать это, мы просто подсчитаем, сколько обновлений по науке о данных делалось в каждый день недели. Иными словами, нам нужно сгруппировать данные по дню недели, который и будет ключом. И если мы эмитируем значение 1 для каждого обновления, которое содержит словосочетание "data science", то получаем совокупное число просто с помощью функции `sum`:

```
def data_science_day_mapper(status_update: dict) -> Iterable:
    """Выдает (день_недели, 1), если обновление ленты
       новостей содержит "data science" """
    if "data science" in status_update["text"].lower():
        day_of_week = status_update["created_at"].weekday()
        yield (day_of_week, 1)

data_science_days = map_reduce(status_updates,
                               data_science_day_mapper,
                               sum_reducer)
```

В качестве чуть более сложного примера представим, что по каждому пользователю нам нужно выяснить наиболее распространенное слово, которое он использует в своих обновлениях ленты новостей. На ум приходят три возможных подхода к реализации преобразователя `mapper`:

- поместить пользовательское имя в ключ; поместить слова и счетчики в значениях;
- поместить слово в ключ; поместить пользовательские имена и счетчики в значениях;
- поместить пользовательское имя и слово в ключ; поместить счетчики в значениях.

Если вы немного поразмыслите, то мы определенно хотим группировать по пользовательскому имени `username`, потому что предпочитаем рассматривать слова каждого человека отдельно. И мы не желаем группировать по слову, поскольку для того, чтобы выяснить, какие слова являются самыми популярными, нашему редук-

тору нужно видеть все слова по каждому человеку. Именно поэтому первый вариант является правильным:

```
def words_per_user_mapper(status_update: dict):
    user = status_update["username"]
    for word in tokenize(status_update["text"]):
        yield (user, (word, 1))

def most_popular_word_reducer(user: str,
                              words_and_counts: Iterable[KV]):
    """С учетом последовательности из пар (слово, количество)
       вернуть слово с наивысшим суммарным количеством появлений"""
    word_counts = Counter()
    for word, count in words_and_counts:
        word_counts[word] += count
    word, count = word_counts.most_common(1)[0]

    yield (user, (word, count))

user_words = map_reduce(status_updates,
                        words_per_user_mapper,
                        most_popular_word_reducer)
```

Либо мы могли бы выяснить число разных поклонников обновлений новостной ленты по каждому пользователю:

```
# Преобразователь поклонников
def liker_mapper(status_update: dict):
    user = status_update["username"]
    for liker in status_update["liked_by"]:
        yield (user, liker)

distinct_likers_per_user = map_reduce(status_updates,
                                      liker_mapper,
                                      count_distinct_reducer)
```

Пример: умножение матриц

Вспомните из *разд. "Умножение матриц" главы 22*, что с учетом $(n \times m)$-матрицы **A** и $(m \times k)$-матрицы **B** мы можем их перемножить, получив $(n \times k)$-матрицу **C**, где элемент матрицы **C** в строке i и столбце j задается так:

```
C[i][j] = sum(A[i][x] * B[x][j] for x in range(m))
```

Это работает, если мы представляем наши матрицы в виде списков списков, как мы и делали.

Однако крупные матрицы иногда бывают разреженными, и вследствие этого большинство их элементов равны нулю. В случае крупных разреженных матриц пред-

ставление в виде списка списков может стать очень расточительным. Более компактное представление хранит только позиции с ненулевыми значениями:

```
from typing import NamedTuple

# Элемент матрицы
class Entry(NamedTuple):
    name: str
    i: int
    j: int
    value: float
```

Например, матрица размера миллиард × миллиард имеет 1 *квинтиллион* элементов, которые было бы непросто хранить на компьютере. Но если в каждой строке имеется лишь несколько ненулевых элементов, то данное альтернативное представление займет на много порядков меньше места.

При таком представлении, оказывается, мы можем воспользоваться алгоритмом MapReduce для выполнения умножения матриц в распределенном стиле.

В обоснование нашего алгоритма обратите внимание, что каждый элемент `A[i][j]` используется для вычисления элементов матрицы **C** только в строке `i`, и каждый элемент `B[i][j]` используется для вычисления элементов матрицы **C** только в столбце `j`. Наша цель будет состоять в том, чтобы каждый выход из редуктора `reducer` являлся единственным элементом в матрице **C**. Это означает, что нам нужно, чтобы преобразователь эмитировал ключи, отождествляющие единственный элемент в **C**. Это предполагает следующую реализацию алгоритма:

```
def matrix_multiply_mapper(num_rows_a: int, num_cols_b: int) -> Mapper:
    # C[x][y] = A[x][0] * B[0][y] + ... + A[x][m] * B[m][y]
    #
    # поэтому элемент A[i][j] идет в каждый C[i][y] с коэфф. B[j][y]
    # и элемент B[i][j] идет в каждый C[x][j] с коэфф. A[x][i]
    def mapper(entry: Entry):
        if entry.name == "A":
            for y in range(num_cols_b):
                key = (entry.i, y)              # какой элемент C
                value = (entry.j, entry.value)  # какой элемент в сумме
                yield (key, value)
        else:
            for x in range(num_rows_a):
                key = (x, entry.j)              # какой элемент C
                value = (entry.i, entry.value)  # какой элемент в сумме
                yield (key, value)
    return mapper
```

И затем:

```
def matrix_multiply_reducer(key: Tuple[int, int],
                            indexed_values: Iterable[Tuple[int, int]]):
    results_by_index = defaultdict(list)
```

```
    for index, value in indexed_values:
        results_by_index[index].append(value)

    # Умножить значения для позиций с двумя значениями
    # (одно из A и одно из B) и суммировать их
    sumproduct = sum(values[0] * values[1]
                     for values in results_by_index.values()
                     if len(values) == 2)

    if sumproduct != 0.0:
        yield (key, sumproduct)
```

Например, если бы у нас были эти две матрицы:

```
A = [[3, 2, 0],
     [0, 0, 0]]

B = [[4, -1, 0],
     [10, 0, 0],
     [0, 0, 0]]
```

то мы могли бы их переписать как кортежи:

```
entries = [Entry("A", 0, 0, 3), Entry("A", 0, 1, 2), Entry("B", 0, 0, 4),
           Entry("B", 0, 1, -1), Entry("B", 1, 0, 10)]

mapper = matrix_multiply_mapper(num_rows_a=2, num_cols_b=3)
reducer = matrix_multiply_reducer

# Произведение должно равняться [[32, -3, 0], [0, 0, 0]],
# и поэтому должно быть два элемента
assert (set(map_reduce(entries, mapper, reducer)) ==
        {((0, 1), -3), ((0, 0), 32)})
```

Этот алгоритм не вызывает особого интереса на таких малых матрицах, но если бы у вас были миллионы строк и миллионы столбцов, то он очень вас выручил бы.

Ремарка: комбинаторы

Вы, вероятно, заметили, что многие наши преобразователи, похоже, содержат чересчур много лишней информации. Например, во время подсчета количеств появлений слов вместо эмитирования пар `(слово, 1)` и суммирования их значений мы могли бы эмитировать пары `(слово, None)` и просто брать длину.

Одна из причин, почему это не было сделано, заключается в том, что в распределенной среде мы иногда хотим использовать *комбинаторы* (combiner), которые редуцируют объем данных, подлежащих пересылке от машины к машине. Если одна из преобразующих машин видит слово "данные" 500 раз, то мы можем дать ей ука-

зание скомбинировать 500 экземпляров пар `("data", 1)` в одну-единственную пару `("data", 500)` перед пересылкой результата редуцирующей машине. Это приводит к гораздо меньшему объему перемещаемых данных, что может сделать наш алгоритм существенно быстрее.

То, как мы написали наш редуктор, позволяет ему обрабатывать эти скомбинированые данные правильно. (Если бы мы написали его с использованием функции `len`, то это уже не получится.)

Для дальнейшего изучения

- Как я уже сказал, вычислительная модель MapReduce сейчас гораздо менее популярна, чем тогда, когда я обдумывал первое издание. Наверное, не стоит тратить на нее массу времени.
- С учетом сказанного наиболее широко используемой системой на основе модели MapReduce является Hadoop[2]. Существуют различные коммерческие и некоммерческие дистрибутивы и огромная экосистема связанных с Hadoop инструментов.
- Amazon.com предлагает веб-службу Elastic MapReduce[3], которая, вероятно, проще, чем настройка собственного кластера.
- Пакетные задания Hadoop, как правило, имеют высокую латентность и поэтому плохо подходят для реально-временного анализа данных. Популярным вариантом для таких рабочих нагрузок является вычислительный каркас Spark[4], который может работать согласно вычислительной модели MapReduce.

[2] См. **http://hadoop.apache.org/**.

[3] См. **http://aws.amazon.com/elasticmapreduce/**.

[4] См. **http://spark.apache.org/**.

ГЛАВА 26

Этика данных

Сперва жратва, а этика — потом.
– *Бертольт Брехт*[1]

Что такое этика данных?

Использование данных сопровождается злоупотреблением данными. Это в значительной степени всегда было так, но в последнее время данная идея была овеществлена как "этика данных" и заняла видное место в новостях.

Например, на выборах 2016 года компания Cambridge Analytica ненадлежаще получила доступ к данным Facebook и использовала их для таргетинга политической рекламы[2].

В 2018 году автономный автомобиль, тестируемый Uber, сбил пешехода насмерть (в машине находился "страховочный водитель", но, видимо, в тот момент он отвлекся)[3].

Алгоритмы используются соответственно для предсказания риска того, что преступники будут повторно нарушать закон, и вынесения им приговоров[4]. Является ли это более или менее справедливым, чем давать судьям определять то же самое?

Некоторые авиакомпании выделяют семьям отдельные места, заставляя их доплачивать за то, чтобы сидеть вместе[5]. Должен ли исследователь данных вмешиваться для того, чтобы предотвращать такие вещи? (Многие исследователи данных в связанном потоке, похоже, так считают.)

---

[1] Бертольт Брехт (1898–1956) — немецкий драматург, поэт и прозаик, театральный деятель, теоретик искусства. Приводится выдержка из стихотворения "Чем жив человек" (из "Трехгрошовой оперы"):

Вы, господа, нас учите морали.
Но вот одно поймете вы едва ль:
Сперва бы лучше вы пожрать нам дали,
А уж потом читали нам мораль.
Вы, любящие свой живот и честность нашу,
Конечно, вы поймете лишь с трудом:
Сначала надо дать голодным хлеб да кашу, —
Сперва жратва, а нравственность — потом. — *Прим. пер.*

[2] См. **https://en.wikipedia.org/wiki/Facebook%E2%80%93Cambridge_Analytica_data_scandal**.

[3] См. **https://www.nytimes.com/2018/05/24/technology/uber-autonomous-car-ntsb-investigation.html**.

[4] См. **https://www.themarshallproject.org/2015/08/04/the-new-science-of-sentencing**.

[5] См. **https://twitter.com/ShelkeGaneshB/status/1066161967105216512**.

"Этика данных" призвана дать ответы на эти вопросы или, по крайней мере, очертить рамки для борьбы с такими нарушениями. Я не настолько самонадеян, чтобы говорить вам, как думать об этих вещах (а "эти вещи" быстро меняются), поэтому в данной главе мы просто проведем краткий обзор некоторых наиболее актуальных вопросов, и (надеюсь) вы получите вдохновение думать о них дальше. (Увы, я недостаточно хороший философ для того, чтобы заниматься этикой с нуля.)

Нет, ну правда, что же такое этика данных?

Хорошо, давайте начнем с того, "что такое этика?" Если вы возьмете среднее арифметическое каждого определения, которое вы можете найти, то получите что-то вроде: этика — это основа для размышлений о "правильном" и "неправильном" поведении. Тогда этика данных — это основа для размышлений о правильном и неправильном поведении, связанном с данными.

Некоторые говорят, что "этика данных" — это (возможно, неявно) набор заповедей о том, что вы можете и не можете сделать. Кое-кто из них усердно работает над созданием манифестов, другие — над обязательствами, которые они надеются заставить вас поклясться выполнять. Третьи выступают за то, чтобы этика данных стала обязательной частью учебной программы по информатике — отсюда и эта глава как средство хеджирования моих ставок в случае успеха.

Любопытно, что существует не так много данных, говорящих о том, что курсы этики приводят к нравственному поведению[6], и в этом случае, возможно, эта кампания сама по себе является неэтичной в смысле этики данных!

Отдельные индивидуумы (например, ваш покорный слуга) думают, что разумные люди часто расходятся во мнениях по тонким вопросам правильного и неправильного, и что важная часть этики данных обязывает учитывать этические последствия ваших линий поведения. Поэтому в спорах следует учитывать неодобрительное мнение многих сторонников "этики данных", но вас никто не заставляет соглашаться с их неодобрением.

Должен ли я заботиться об этике данных?

Вы обязаны заботиться об этике, какой бы ни была ваша работа. Если ваша деятельность связана с данными, то вы свободно можете охарактеризовать свою озабоченность как "этику данных", но вы должны заботиться об этике и в тех частях своей работы, которые не связаны с данными.

Пожалуй, технологическая работа отличается тем, что технология масштабируется, а решения, принимаемые людьми, работающими над технологическими задачами

[6] См. **https://www.washingtonpost.com/news/on-leadership/wp/2014/01/13/can-you-teach-businessmen-to-be-ethical**.

(связанными или не связанными с данными), потенциально имеют широкие последствия.

Крошечное изменение алгоритма обнаружения новостей может кардинально изменить многомиллионную аудиторию, читающую статью, на аудиторию людей, которые вообще ее не читают.

Единственный дефектный алгоритм предоставления условно-досрочного освобождения, который используется по всей стране, систематически затрагивает миллионы людей, в то время как ошибочная по-своему комиссия по условно-досрочному освобождению затрагивает только тех людей, которые предстают перед ней.

Так что да, в общем, вы должны заботиться о том, какое влияние ваша работа оказывает на мир. И чем шире последствия вашей работы, тем больше нужно беспокоиться об этих вещах.

К сожалению, некоторые рассуждения об этике данных привлекают людей, пытающихся навязать вам свои этические выводы. Должны ли вы заботиться о тех же вещах, о которых беспокоятся они, в действительности зависит от вас.

Создание плохих продуктов данных

Некоторые вопросы "этики данных" являются результатом создания плохих продуктов.

Например, компания Microsoft выпустила чат-бота под названием Tay[7], который как попугай твитил все, что ему писали в Twitter, и Интернет быстро обнаружил, что эта особенность позволяла любому побуждать Tay твитить всякого рода оскорбительные вещи. По всей видимости, вряд ли кто-либо в Microsoft обсуждал этичность выпуска "расистского" бота; скорее всего, разработчики просто построили бота и не додумались, что им можно легко злоупотребить. Этот пример, вероятно, устанавливает низкую планку, но давайте согласимся, что вы обязаны думать о возможности злоупотребления вещами, которые вы создаете.

Еще один пример касается Google-службы Фотографии, когда в какой-то момент там использовали алгоритм распознавания изображений, который иногда классифицировал фотографии чернокожих людей как "горилл"[8]. Опять же, крайне маловероятно, что кто-либо в Google явно решил поставлять этот функционал (не говоря уже о серьезной работе с "этикой"). Здесь, похоже, проблема заключается в некотором сочетании плохих тренировочных данных, неточности модели и грубой оскорбительности ошибки (если бы модель иногда классифицировала почтовые ящики как пожарные машины, вероятно, никто бы не озаботился).

В этом случае решение является менее очевидным: как обеспечить, чтобы ваша натренированная модель не делала предсказания, которые в некотором роде оскорби-

[7] См. **https://en.wikipedia.org/wiki/Tay_(bot)**.

[8] См. **https://www.theverge.com/2018/1/12/16882408/google-racist-gorillas-photo-recognition-algorithm-ai**.

тельны? Разумеется, вы должны тренировать (и тестировать) свою модель на различных входных данных, но можете ли вы быть уверены в том, что нигде нет никаких данных, которые заставят вашу модель вести себя так, что вам будет стыдно? Это сложная проблема. (В компании Google, похоже, ее "решили", просто отказавшись предсказывать "гориллу" вообще.)

Компромисс между точностью и справедливостью

Представьте, что вы строите модель, которая предсказывает, насколько вероятно, что люди предпримут какие-то действия. Вы делаете довольно хорошую работу (табл. 26.1).

Таблица 26.1. *Довольно хорошая работа*

Предсказание	Люди	Действия	%
Маловероятно	125	25	20
Вероятно	125	75	60

Из людей, в отношении которых вы делаете предсказание о маловероятном предпринимаемом ими действии, только 20% так и делают. А те, кто предпримут действие, 60% из них так и делают. Вроде не страшно.

Теперь представьте, что люди могут быть разделены на две группы: *A* и *B*. Некоторые из ваших коллег обеспокоены тем, что ваша модель несправедлива к одной из групп. Хотя модель не учитывает членство в группе, она учитывает другие разные факторы, которые сложным образом коррелируют с членством в группе.

И действительно, когда вы разбиваете предсказания на группы, вы обнаруживаете удивительную статистику (табл. 26.2).

Таблица 26.2. *Удивительная статистика*

Группа	Предсказание	Люди	Действия	%
A	Маловероятно	100	20	20
A	Вероятно	25	15	60
B	Маловероятно	25	5	20
B	Вероятно	100	60	60

Является ли ваша модель несправедливой? Исследователи данных в вашей команде приводят различные аргументы.

- *Аргумент 1.* Ваша модель классифицирует 80% группы *A* как "маловероятную", но 80% группы *B* как "вероятную". Этот исследователь данных жалуется, что

модель несправедливо относится к двум группам в том смысле, что она генерирует совершенно разные предсказания между двумя группами.

- *Аргумент 2.* Независимо от членства в группе, если мы предсказываем "маловероятно", то у вас есть 20%-ный шанс на действие, а если мы предсказываем "вероятно", то у вас есть 60%-й шанс на действие. Этот исследователь данных настаивает на том, что модель является "точной" в том смысле, что ее предсказания, похоже, означают одно и то же, независимо от того, к какой группе вы принадлежите.
- *Аргумент 3.* Ложно помеченными как "вероятные" были $40/125 = 32\%$ группы *B* и только $10/125 = 8\%$ группы *A*. Этот исследователь данных (который считает "вероятное" предсказание плохой идеей) настаивает на том, что модель несправедливо стигматизирует группу *B*.
- *Аргумент 4.* Ложно помеченными как "маловероятные" были $20/125 = 16\%$ группы *A* и только $5/125 = 4\%$ группы *B*. Этот исследователь данных (который считает "маловероятное" предсказание плохой идеей) настаивает на том, что модель несправедливо стигматизирует группу *A*.

Какие из этих исследователей данных не ошибаются? Есть ли среди них кто-нибудь, кто рассуждает правильно? Вероятно, это зависит от контекста.

Возможно, вы будете рассуждать одним путем, если две группы представлены "мужчинами" и "женщинами"; и другим путем, если две группы представлены "пользователями языка R" и "пользователями языка Python"; либо, возможно, не будете, если выяснится, что пользователи языка Python асимметрично представлены мужчинами, а пользователи языка R — женщинами?

Возможно, вы будете рассуждать одним путем, если модель предназначена для предсказания, будет ли пользователь социальной сети DataSciencester подавать заявку на работу через доску вакансий соцсети DataSciencester; и другим путем, если модель предсказывает, пройдет ли пользователь такое собеседование.

Возможно, ваше мнение зависит от самой модели, от того, какие признаки она учитывает и на каких данных она тренировалась.

В любом случае я хочу подчеркнуть, что между "точностью" и "справедливостью" может быть компромисс (в зависимости, конечно, от того, как вы их определяете) и что эти компромиссы не всегда имеют очевидные "правильные" решения.

Сотрудничество

Репрессивное (по вашим меркам) правительство страны наконец-то решило разрешить гражданам вступать в социальную сеть DataSciencester. Однако руководящие органы настаивают на том, чтобы пользователи из их страны не могли обсуждать глубокое обучение. Более того, они хотят, чтобы вы сообщали им имена любых пользователей, которые лишь пытаются искать информацию о глубоком обучении.

Будет ли лучше для исследователей данных этой страны иметь выход на ограниченную по тематике (и под надзором) социальную сеть DataSciencester, который

вам будет разрешено предложить? Или же предлагаемые ограничения настолько ужасны, что им лучше вообще не иметь доступа?

Интерпретируемость

Отдел кадров социальной сети DataSciencester просит вас разработать модель, предсказывающую риски увольнения сотрудников по собственному желанию, с тем, чтобы можно было вмешаться в ситуацию и попытаться удержать их. (Текучесть кадров — важный компонент рубрики журнала "10 самых счастливых рабочих мест", в которой ваш генеральный директор стремится появиться.)

Вы собрали набор исторических данных и рассматриваете три модели:

- дерево решений;
- нейронную сеть;
- дорогостоящего "эксперта по удержанию" кадров.

Один из ваших исследователей данных настаивает на том, что вы должны просто использовать ту модель, которая работает лучше всего.

Второй настаивает на том, чтобы вы не использовали нейросетевую модель, т. к. только две других могут объяснять свои предсказания, и что только объяснение предсказаний способно помочь агентству по трудоустройству широко распространять изменения (в отличие от одноразовых вмешательств).

Третий говорит, что, хотя "эксперт" может давать объяснение своим предсказаниям, нет причин верить ему на слово в том, что он описывает реальные причины, почему он предсказал так, а не иначе.

Как и в других наших примерах, здесь нет абсолютно лучшего варианта выбора. В некоторых обстоятельствах (допустим, по юридическим причинам либо если ваши предсказания каким-то образом меняют жизнь) вы предпочете модель, которая работает хуже, но чьи предсказания можно объяснять. В других случаях вам может понадобиться модель, которая предсказывает лучше всего. В третьих, возможно, вообще нет интерпретируемой модели, которая работает хорошо.

Рекомендации

Как мы обсуждали в *главе 23*, распространенное применение науки о данных связано с рекомендациями людям разнообразных вещей. Когда кто-то смотрит видеоролик на YouTube, этот видеохостинг рекомендует видеоролики, которые данному человеку следует посмотреть потом.

Видеохостинг YouTube зарабатывает деньги за счет рекламы и (предположительно) хочет рекомендовать видео, которые вы, скорее всего, будете смотреть, благодаря чему компания может показывать вам больше рекламы. Однако, оказывается, людям нравится смотреть видео о теориях заговора, которые, как правило, фигурируют в рекомендациях.

В то время, когда я писал эту главу, если бы вы поискали в YouTube слово "сатурн", то третьим результатом было бы "Что-то происходит на Сатурне... ОНИ от нас что-то скрывают?", что, возможно, дает вам представление о видеороликах, о которых я говорю.

Несет ли у YouTube на себе обязательство не рекомендовать видеоролики о заговорах? Даже если это то, что многие люди, похоже, хотят посмотреть?

Другой пример заключается в том, что если вы зайдете на google.com (или bing.com) и начнете набирать в поисковой строке, поисковая система выложит предложения, автоматически завершая вводимый текст. Эти предложения основаны (по крайней мере частично) на том, что искали другие люди; в частности, если другие люди ищут неприятные вещи, то это может быть отражено в предлагаемых вам вариантах.

Должна ли поисковая система пытаться положительно фильтровать предлагаемые варианты, которые ей не нравятся? Google (по какой-то причине), похоже, намерен не предлагать вещи, связанные с религией. Например, если вы наберете "mitt romney m" в Bing, то первым предложенным вариантом будет "mitt romney mormon" (это то, что я и ожидал), тогда как Google отказывается предоставлять этот вариант.

Действительно, Google явно отфильтровывает автозаполнения, которые он считает "оскорбительными или пренебрежительными"[9]. (Каким образом они там решают, что является оскорбительным или пренебрежительным, остается неясным.) И все же иногда правда является оскорбительной. Этично ли защищать людей от таких предлагаемых вариантов? Или же так поступать неэтично? Или же это вообще не вопрос этики?

Предвзятые данные

В *разд. "Векторы слов" главы 21* мы использовали корпус документов для усвоения векторных вложений слов. Эти векторы были разработаны с целью проявлять дистрибутивное сходство. То есть слова, которые появляются в похожих контекстах, должны иметь похожие векторы. В частности, любые предвзятости (смещения), которые существуют в тренировочных данных, будут отражены в самих векторах слов.

Например, если все наши документы посвящену тому, что пользователи R являются моральными негодяями и что пользователи Python являются образцами добродетели, то, скорее всего, модель усвоит такие ассоциации для слов "Python" и "R".

Чаще всего векторы слов основаны на некоторой комбинации статей из Google Новости, Википедии, книг и веб-страниц. Это означает, что они усвоят любые дистрибутивные регулярности, которые присутствуют в этих источниках.

---

[9] См. **https://blog.google/products/search/google-search-autocomplete/**.

Например, если большинство новостных статей об инженерах-программистах посвящено мужчинам-программистам, то усвоенный вектор для словосочетания "программное обеспечение" может лежать ближе к векторам для других "мужских" слов, чем к векторам для "женских" слов.

В этом месте любые последующие приложения, которые вы строите с использованием этих векторов, также могут проявлять эту близость. В зависимости от приложения, это может быть или не быть для вас проблемой. В таком случае существуют различные технические решения, которые вы можете попробовать для того, чтобы "удалять" определенные предвзятости, хотя вы, вероятно, никогда не уловите их все. Но это то, что вы должны знать.

Точно так же, как в примере с "фотографиями" в *разд. "Создание плохих продуктов данных"*, если вы тренируете модель на нерепрезентативных данных, то существует большая вероятность плохой работы модели в реальном мире, возможно даже с оскорблениями или смущением.

По иным направлениям также существует возможность, что ваши алгоритмы могут кодировать фактические предвзятости (смещения), которые существуют в мире. Например, ваша модель условно-досрочного освобождения может отлично предсказывать то, какие освобожденные преступники будут повторно арестованы, но если эти повторные аресты сами являются результатом предвзятых реальных процессов, то ваша модель может увековечить эту предвзятость.

Защита данных

Вы многое знаете о пользователях социальной сети DataSciencester: какие технологии им нравятся; что их друзья — исследователи данных; где они работают; сколько они зарабатывают; сколько времени они проводят на веб-сайте; по каким публикациям они кликают и т. д.

Директор по монетизации хочет продавать эти данные рекламодателям, которые, в свою очередь, хотят продавать свои различные решения "больших данных" вашим пользователям. Старший исследователь хочет поделиться этими данными с академическими исследователями, которые стремятся публиковать статьи о том, кто становится исследователем данных. Директор по предвыборной агитации планирует предоставить эти данные политическим кампаниям, большинство из которых стремятся рекрутировать собственные организации по исследованию данных. И директор по связям с государственными органами хотел бы использовать эти данные для того, чтобы отвечать на вопросы правоохранительных органов.

Благодаря дальновидному директору по контрактам ваши пользователи согласились с условиями обслуживания, которые гарантируют вам право делать с их данными всё, что вы захотите.

Однако (как вы теперь ожидаете) многие исследователи данных в вашей команде выдвигают различные возражения против такого рода применений. Один считает неправильным передавать данные рекламодателям; другой беспокоится, что академическим исследователям нельзя доверять ответственную защиту данных. Третий

уверен, что компания должна оставаться вне политики, в то время как последний настаивает на том, что полиции нельзя доверять и сотрудничество с правоохранительными органами повредит невинным людям.

Правы ли эти исследователи данных?

Резюме

Существует масса всего, что вызывает беспокойство! И есть бесчисленное множество других вещей, о которых мы не упомянули, и еще больше того, что появится в будущем, но о чем мы сегодня даже думать не смели.

Для дальнейшего изучения

- Нет недостатка в людях, исповедующих важные мысли об этике данных. Поиск в Twitter (или на вашем любимом новостном веб-сайте), вероятно, будет наилучшим способом узнать о самых последних дискуссиях по поводу этики данных.
- Если вы хотите что-то более практичное, то Майк Лукидес, Хилари Мейсон и Ди-Джей Патил написали короткую электронную книгу "Этика и наука о данных"[10] (Mike Loukides, Hilary Mason, DJ Patil. "Ethics and Data Science") о применении этики данных на практике, которую я с честью обязан рекомендовать, поскольку Майк согласился опубликовать книгу "Data Science с нуля" еще в 2014 году. (Упражнение: этично ли это с моей стороны?)

[10] См. **https://www.oreilly.com/library/view/ethics-and-data/9781492043898/**.

Идите вперед и займитесь наукой о данных

> И сейчас, в очередной раз, я приглашаю мое ужасное потомство идти вперед и процветать.
> – *Мэри Шелли*[1]

Куда двигаться дальше? Если допустить, что я не отпугнул вас от науки о данных, то существует ряд вещей, которые вам надлежит узнать в следующую очередь.

Программная оболочка IPython

Ранее в этой книге я упоминал оболочку IPython[2]. Она обеспечивает гораздо больший функционал, чем стандартная оболочка Python, а также добавляет "волшебные" функции, которые позволяют (помимо всего прочего) легко копировать и вставлять код (что в обычных условиях осложняется сочетанием пустых строк и пробельных символов форматирования) и выполнять сценарии из оболочки.

Освоение IPython намного упростит работу. (Этого можно добиться, зная всего лишь несколько элементов интерфейса IPython.)

В первом издании я также рекомендовал вам познакомиться с блокнотом IPython (теперь Jupyter), вычислительной средой, которая позволяет совмещать текст, живой код Python и визуализации.

С тех пор я стал скептиком в отношении блокнотов, поскольку обнаружил, что они запутывают новичков и поощряют плохие практики кодирования. (У меня есть много других причин.) Вы наверняка получите большую поддержку по их использованию со стороны людей, которые не являются мной, так что просто помните о моем несогласном голосе.

Математика

На протяжении всей книги мы пробовали себя в линейной алгебре *(см. главу 4)*, статистике *(см. главу 5)*, теории вероятностей *(см. главу 6)* и различных аспектах машинного обучения.

---

[1] Мэри Шелли (1797–1851) — английская писательница, автор романа "Франкенштейн, или современный Прометей" (см. **https://ru.wikipedia.org/wiki/Шелли,_Мэри**). — *Прим. пер.*

[2] См. **http://ipython.org/**.

Для того чтобы стать хорошим исследователем данных, требуется гораздо лучше разбираться в этих областях знаний, и я призываю вас заняться их более углубленным изучением, используя для этих целей учебники, рекомендованные в конце глав, учебники, которые вы предпочитаете сами, курсы дистанционного обучения и даже посещения реальных курсов.

Не с нуля

Имплементация технических решений "с нуля" позволяет лучше понять, как они работают. Но такой подход, как правило, сильно сказывается на их производительности (если только они не имплементируются специально для повышения производительности), простоте использования, быстроте прототипирования и обработке ошибок.

На практике лучше использовать хорошо продуманные библиотеки, прочно имплементирующие теоретические основы. Мое первоначальное предложение для этой книги предусматривало вторую часть "Теперь давайте займемся библиотеками", на которую издательство O'Reilly, к счастью, наложило вето. С тех пор как вышло первое издание, Джейк Вандерплас (Jake VanderPlas) написал книгу "Руководство по науке о данных на Python"[3] (Python Data Science Handbook), которая является удачным введением в соответствующие библиотеки и будет хорошей книгой, которую вы прочтете следующей.

Библиотека NumPy

Библиотека NumPy[4] (обозначает "численные вычисления на Python" от *англ.* Numeric Python) располагает функционалом для "реальных" научных вычислений. Она обеспечивает массивы, которые работают лучше, чем наши списковые векторы; матрицы, которые работают лучше, чем наши списко-списковые матрицы; и много числовых функций для работы с ними.

Данная библиотека является строительным блоком для многих других библиотек, что делает овладение ею особенно ценным.

Библиотека pandas

Библиотека pandas[5] предоставляет дополнительные структуры данных для работы с наборами данных на языке Python. Ее основная абстракция — это проиндексированный многомерный кадр данных `DataFrame`, который по сути похож на класс `Table` приложения NotQuiteABase, разработанного в *главе 24*, но с гораздо бóльшим функционалом и лучшей производительностью.

---

[3] См. **http://shop.oreilly.com/product/0636920034919.do**.

[4] См. **http://www.numpy.org/**.

[5] См. **http://pandas.pydata.org/**.

Если вы собираетесь использовать Python для преобразования, разбиения, группирования и оперирования наборами данных, то библиотека pandas — бесценный инструмент для этих целей[6].

Библиотека scikit-learn

Библиотека scikit-learn[7] — это, наверное, самая популярная библиотека для работы в области машинного обучения на языке Python. Она содержит все модели, которые были тут имплементированы, и многие другие. В реальной ситуации не следует строить дерево решений "с нуля"; всю тяжелую работу, связанную с решением этой задачи, должна делать библиотека scikit-learn. При решении реальной задачи вы никогда не будете писать оптимизационный алгоритм от руки; вы положитесь на библиотеку scikit-learn, где уже используется по-настоящему хороший алгоритм.

Документация по библиотеке содержит огромное число примеров[8] того, что она способна делать (и того, какие задачи решает машинное обучение).

Визуализация

Графики библиотеки matplotlib получились ясными и функциональными, но не особо стилизованными (и совсем не интерактивными). Если вы хотите углубиться в область визуализации данных, то вот несколько вариантов.

В первую очередь стоит глубже исследовать библиотеку matplotlib. То, что было показано, является лишь незначительной частью ее функциональных возможностей. На веб-сайте библиотеки содержится много примеров[9] ее функциональности и галерея[10] из некоторых наиболее интересных. Если вы хотите создавать статические визуализации (скажем, для книжных иллюстраций), то библиотека matplotlib, вероятно, будет вашим следующим шагом.

Помимо этого, вам стоит попробовать библиотеку seaborn[11], которая (среди прочего) делает библиотеку matplotlib привлекательнее.

Если же вы хотите создавать *интерактивные* визуализации, которыми можно делиться в сети, то, очевидно, для этого лучше всего подойдет библиотека D3.js[12], написанная на JavaScript и предназначенная для создания "документов, управляемых данными" (три буквы D от английского выражения Data Driven Documents). Даже если вы не знакомы с JavaScript, всегда можно взять примеры из галереи D3[13]

---

[6] В разы более высокую производительность показывает новая библиотека datatable (**https://pypi.org/project/datatable/**), правда, функционально она пока не дотягивает до pandas. — *Прим. пер.*

[7] См. **http://scikit-learn.org/**.

[8] См. **http://scikit-learn.org/stable/auto_examples/**.

[9] См. **http://matplotlib.org/examples/**.

[10] См. **http://matplotlib.org/gallery.html**.

[11] См. **https://seaborn.pydata.org/**.

[12] См. **http://d3js.org/**.

[13] См. **https://github.com/mbostock/d3/wiki/Gallery**.

и настроить их для работы со своими данными. (Хорошие исследователи данных копируют примеры из галереи D3; великие исследователи данных *крадут* их оттуда.)

Даже если вы не заинтересованы в библиотеке D3, то элементарный просмотр галереи сам по себе уже является невероятно поучительным занятием в области визуализации данных.

Библиотека Bokeh[14] является проектом, который привносит в Python функционал в стиле библиотеки D3.

Язык R

Хотя вы можете совершенно безнаказанно обойтись без овладения языком программирования R[15], многие исследователи данных и многие проекты в области исследования данных его используют, поэтому стоит по крайней мере с ним ознакомиться.

Отчасти это нужно для того, чтобы у вас появилась возможность понимать посты, примеры и код в блогах, ориентированных на пользователей языка R; отчасти потому, что знание языка поможет лучше оценить (сравнительно) чистую элегантность языка Python; а отчасти это поможет быть более информированным участником нескончаемых и горячих дискуссий по теме "R против Python".

Глубокое обучение

Вы можете быть исследователем данных, не занимаясь глубоким обучением, но вы не можете быть модным исследователем данных, не занимаясь им.

Двумя самыми популярными вычислительными каркасами глубокого обучения для Python являются библиотека TensorFlow[16] (созданная компанией Google) и PyTorch[17] (созданная компанией Facebook). Интернет полон для них учебников, от прекрасных до ужасных.

Библиотека TensorFlow старше и шире применяется, но библиотека PyTorch (на мой взгляд) намного проще в использовании и (в частности) гораздо удобнее для начинающих. Я предпочитаю (и рекомендую) библиотеку PyTorch, но, как говорится, никого никогда не увольняли за выбор библиотеки TensorFlow.

Отыщите данные

Если вы занимаетесь наукой о данных в рамках своей работы, то, скорее всего, вы получите данные в рамках своей работы (хотя и не обязательно). Но как быть, если вы занимаетесь наукой о данных ради интеллектуального удовольствия? Хотя данные есть везде, вот несколько отправных точек:

---

[14] См. **http://bokeh.pydata.org/**.

[15] См. **https://www.r-project.org/**.

[16] См. **https://www.tensorflow.org/**.

[17] См. **https://pytorch.org/**.

- Data.gov[18] — это портал открытых данных правительства США. Если требуются данные по всему, что связано с государством (которое в наши дни оккупирует собой почти все вокруг), то этот портал будет хорошей отправной точкой.[19]
- На социальном новостном веб-сайте reddit есть пара форумов: r/datasets[20] и r/data[21], где можно запросить и найти данные.
- Amazon.com поддерживает коллекцию общедоступных наборов данных[22], которые они хотели бы, чтобы пользователи применяли для анализа с помощью их продуктов (но которые вы можете анализировать с помощью любых продуктов по вашему желанию).
- У Робба Ситона в его блоге[23] есть причудливый список курируемых наборов данных.
- Kaggle[24] — это веб-сайт, который проводит состязания в области науки о данных. Мне ни разу не удавалось в них поучаствовать (ввиду нехватки соревновательного духа, когда дело доходит до науки о данных), но вы могли бы попробовать. Там размещено очень много наборов данных.
- У компании Google недавно появился новый функционал поиска наборов данных[25], который позволяет вам (да, вы угадали) производить поиск наборов данных.

Займитесь наукой о данных

Замечательно, когда ты просматриваешь каталоги данных, но лучшие проекты (и продукты) — те, которые были вызваны неким внутренним стремлением. Вот несколько из тех, которые я реализовал.

Новости хакера

Веб-сайт новостей хакера Hacker News[26] — это агрегатор новостей и дискуссионная площадка для новостей, связанных с технологиями. Указанный веб-сайт собирает огромное количество статей, многие из которых мне не интересны.

Поэтому несколько лет назад я задался целью создать классификатор статей этого сайта[27], который бы предсказывал, будет ли мне интересно то или иное повествова-

---

[18] См. **https://www.data.gov/**.

[19] В Российской Федерации тоже существует веб-сайт правительства РФ с открытыми данными — **http://government.ru**. — *Прим. пер.*

[20] См. **https://www.reddit.com/r/datasets**.

[21] См. **https://www.reddit.com/r/data**.

[22] См. **https://aws.amazon.com/ru/public-data-sets/**.

[23] См. **http://rs.io/100-interesting-data-sets-for-statistics/**.

[24] См. **https://www.kaggle.com/**.

[25] См. **https://toolbox.google.com/datasetsearch**.

[26] См. **https://news.ycombinator.com/news**.

[27] См. **https://github.com/joelgrus/hackernews**.

ние. С этим сайтом получилось не все так просто. Владельцы сайта отвергли саму идею о том, что кто-то может не быть заинтересован в любом из рассказов, выложенных на их сайте.

Работа включала в себя разметку в ручном режиме большого числа статей (с целью получения тренировочного набора), выбор характерных признаков статей (например, слов в заголовке и доменных имен в ссылках), тренировку наивного байесова классификатора, мало чем отличающегося от нашего спам-фильтра.

По причинам, ныне затерявшимся в истории, я написал его на языке Ruby. Учитесь на моих ошибках.

Пожарные машины

В течение многих лет я проживал на главной улице в центре Сиэтла, на полпути между пожарной частью и большинством городских пожаров (или мне так казалось). Соответственно, за те годы у меня развилось праздное любопытство к пожарной части моего города.

К счастью (с точки зрения данных), существует сайт Realtime 911[28], на котором выложены все вызовы по пожарной тревоге вместе с номерами участвовавших пожарных машин.

И вот, чтобы побаловать свое любопытство, я собрал с их сайта многолетние данные о вызовах и выполнил сетевой анализ[29] на примере пожарных машин. Среди прочего, мне пришлось специально для пожарных машин изобрести понятие центральности, которое я назвал рангом пожарной машины, Truck-Rank.

Футболки

У меня есть малолетняя дочь, и постоянным источником расстройства для меня на протяжении всех ее дошкольных лет было то, что большинство "девчачьих" футболок были довольно скучными, в то время как большинство "мальчишечьих" выглядели забавными.

В частности, было ясно, что между футболками для девочек и футболками для мальчиков дошкольного возраста существует четкое различие. И поэтому я задался вопросом, можно ли натренировать модель, которая распознавала бы эти различия?

Подсказка: ответ был положительным[30].

Для этого потребовалось скачать фотографии сотен футболок, сжать их все к единому размеру, превратить в векторы пиксельных цветных оттенков и на основе логистической регрессии построить классификатор.

Один подход просто смотрел, какие цвета присутствовали в каждой футболке; второй отыскал первые 10 главных компонент векторов снимков футболок и класси-

---

[28] См. **http://www2.seattle.gov/fire/realtime911/getDatePubTab.asp**.

[29] См. **https://github.com/joelgrus/fire**.

[30] См. **https://github.com/joelgrus/shirts**.

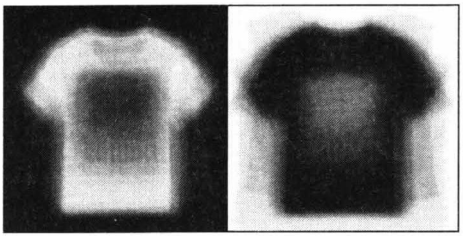


Рис. 27.1. Собственные футболки, соответствующие первой главной компоненте

фицировал каждую футболку, используя их проекции в 10-размерное пространство, занимаемое "собственными футболками" (рис. 27.1).

Твиты по всему глобусу

В течение многих лет я хотел построить визуализацию "вращающегося глобуса". Во время президентских выборов 2016 года я построил небольшое веб-приложение[31], которое слушало геотегированные твиты, соответствующие некоторому поисковому запросу (я использовал запрос "Trump", т. к. в то время он появлялся во многих твитах), отображал их на глобусе, вращая его в нужное местоположение, когда они появлялись.

Этот проект данных был полностью написан на языке JavaScript, поэтому, возможно, вы изучите немного JavaScript.

А вы?

А что интересует вас? Какие вопросы не дают вам спать по ночам? Попробуйте отыскать набор данных (или выскрести какой-нибудь веб-сайт) и займитесь наукой о данных.

Сообщите мне о своих находках! Шлите сообщения по электронному адресу **joelgrus@gmail.com** либо отыщите в Twitter по хештегу **@joelgrus**.

[31] См. **https://twitter.com/voxdotcom/status/1069427256580284416**.

Предметный указатель

T

V

W

X

Z

A

Б

В

Г

Д

З

И

К

Л

М

H

O

П

P

C

Я

O'REILLY®

Data Science
Наука о данных с нуля

Книга позволяет освоить науку о данных, начав «с чистого листа».

Она написана так, что способствует погружению в Data Science аналитика, фактически не обладающего глубокими знаниями в этой прикладной дисциплине.

При этом вы убедитесь, что описанные в книге программные библиотеки, платформы, модули и пакеты инструментов, предназначенные для работы в области науки о данных, великолепно справляются с задачами анализа данных.

А если у вас есть способности к математике и навыки программирования, то Джоэл Грас поможет вам почувствовать себя комфортно с математическим и статистическим аппаратом, лежащим в основе науки о данных, а также с приемами алгоритмизации, которые потребуются для работы в этой области.

Обновленное второе издание книги, использующее версию Python 3.6 и наполненное новыми материалами по глубокому обучению, статистике и обработке естественного языка, покажет вам, как найти драгоценные камни в сегодняшнем беспорядочном и избыточном потоке данных.

Вместе с Джоэлом Грасом и его книгой

- Пройдите интенсивный курс языка Python
- Изучите элементы линейной алгебры, математической статистики, теории вероятностей и их применение в науке о данных
- Займитесь сбором, очисткой, нормализацией и управлением данными
- Окунитесь в основы машинного обучения
- Познакомьтесь с различными математическими моделями и их реализацией по методу *k* ближайших соседей, наивной байесовой классификации, линейной и логистической регрессии, а также моделями на основе деревьев принятия решений, нейронных сетей и кластеризации
- Освойте работу с рекомендательными системами, приемы обработки естественного языка, методы анализа социальных сетей, технологии MapReduce и баз данных

«Джоэл проведет для вас экскурсию по науке о данных. В результате вы перейдете от простого любопытства к глубокому пониманию насущных алгоритмов, которые должен знать любой аналитик данных».

— Роит Шивапрасад

Специалист компании Amazon в области Data Science с 2014 г.

Джоэл Грас работает инженером-программистом в компании Google. До этого занимался аналитической работой в нескольких стартапах. Активно участвует в неформальных мероприятиях специалистов в области науки о данных. Всегда доступен в Twitter по хештегу @joelgrus.

ISBN 978-5-9775-6731-2

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®