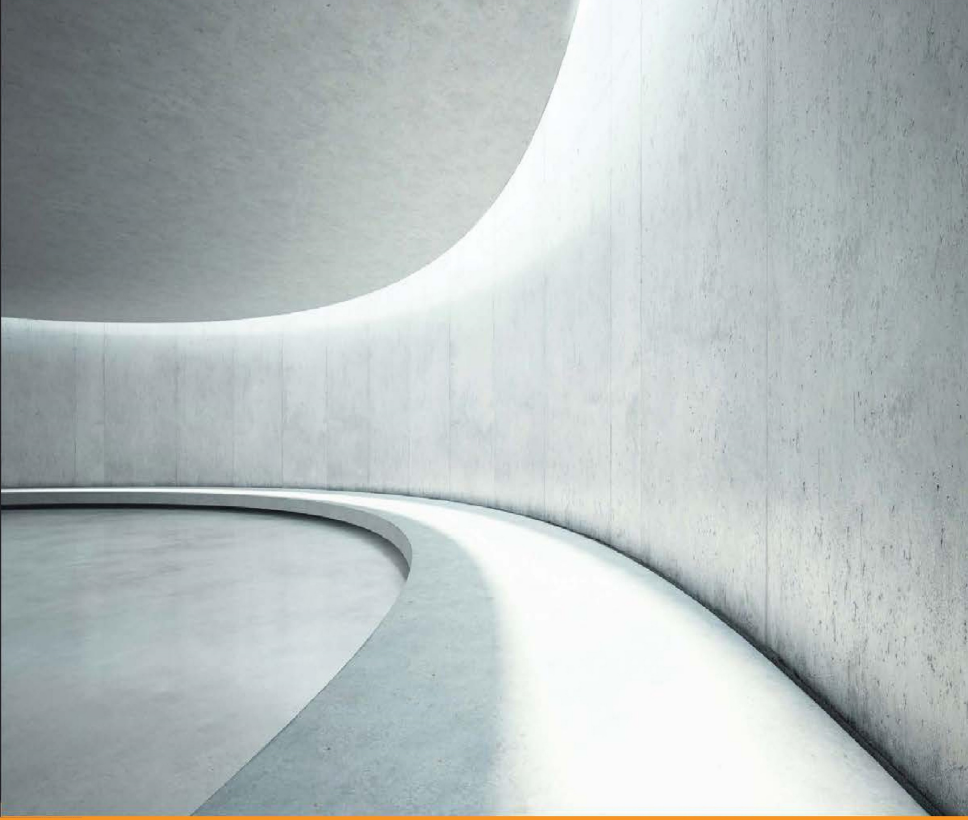


Построение систем машинного обучения на языке Python

Луис Педро Коэльо Вилли Ричарт

ДМК издательство

Building Machine Learning Systems with Python

Second Edition

Get more from your data through creating practical machine learning systems with Python

Luis Pedro Coelho
Willi Richert

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Построение систем машинного обучения на языке Python

Как извлечь больше информации из данных путем построения практичных систем машинного обучения на языке Python

Луис Педро Коэльо
Вилли Ричарт

Москва, 2016

УДК 004.438Python:004.6
ББК 32.973.22
К76

К76 Луис Педро Коэльо, Вилли Ричарт

Построение систем машинного обучения на языке Python. 2-е издание / пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2016. – 302 с.: ил.

ISBN 978-5-97060-330-7

Применение машинного обучения для лучшего понимания природы данных – умение, необходимое любому современному разработчику программ или аналитику. Python – замечательный язык для создания приложений машинного обучения. Благодаря своей динамичности он позволяет быстро производить разведочный анализ данных и экспериментировать с ними. Обладая первоклассным набором библиотек машинного обучения с открытым исходным кодом, Python дает возможность сосредоточиться на решаемой задаче и в то же время опробовать различные идеи.

Книга начинается с краткого введения в предмет машинного обучения и знакомства с библиотеками NumPy, SciPy, scikit-learn. Но довольно быстро авторы переходят к более серьезным проектам с реальными наборами данных, в частности, тематическому моделированию, анализу корзины покупок, облачным вычислениям и др.

Издание рассчитано на программистов, пишущих на Python и желающих узнать о построении систем машинного обучения и научиться извлекать из данных ценную информацию, необходимую для решения различных задач.

ISBN 978-1-78439-277-2 (англ.)
ISBN 978-5-97060-330-7 (рус.)

ОГЛАВЛЕНИЕ

ОБ АВТОРАХ

Луис Педро Коэльо (Luis Pedro Coelho) – специалист по компьютерной биологии, то есть человек, применяющий компьютеры для изучения биологических систем. Конкретно Луис анализирует ДНК микробных сообществ, чтобы охарактеризовать их поведение. Луис также много работал в области обработки биологических изображений – приложения методов машинного обучения к анализу изображений билогических образцов. Сфера его научных интересов – обработка и агрегирование больших наборов данных.

Луис защитил докторскую диссертацию в Университете Карнеги-Меллон, одном из ведущих учебных заведений в области машинного обучения. Он автор нескольких научных публикаций.

Разработкой программного обеспечения с открытым исходным кодом Луис начал заниматься в 1998 году, поставив целью применить на практике методы кодирования, которые он изучал на курсах информатики в Лиссабонском техническом университете. В 2004 году он начал программировать на Python и внес свой вклад в несколько открытых библиотек на этом языке. Он ведущий разработчик популярного пакета программ машинного зрения для Python mahotas, а также соавтор ряда программ машинного обучения.

В настоящее время Луис живет то в Люксембурге, то в Гейдельберге.

Благодарю свою жену Риту за любовь и поддержку и дочь Анну за то, что она лучше всех.

Вили Ричарт (Willi Richert) имеет степень доктора наук по машинному обучению и робототехнике. Он применял методы обучения с подкреплением, скрытые марковские модели и байесовские сети для обучения гетерогенных роботов путем подражания. В настоящее время работает в корпорации Microsoft в группе основ релевантности в Bing, где занимается различными приложениями машинного обучения, в том числе активным обучением, статистическим машинным переводом и решающими деревьями.

Эта книга не была бы написана без поддержки моей жены Натали и моих сыновей, Линуса и Морица. Я благодарю за полезные дискуссии моих предыдущих руководителей Андреаса Боде (Andreas Bode), Клеменса Маршнера (Clemens Marschner), Хон Янь Чжу (Hongyan Zhou) и Эрика Крестана (Eric Crestan), а также коллег и друзей Томаша Марциняка (Tomasz Marciniak), Кристиана Эйгеля (Cristian Eigel), Оливера Нихёрстера (Oliver Niehoerster) и Филиппа Эделта (Philipp Adelt). Интересные идеи, скорее всего, исходят от них, ошибки же целиком на моей совести.

О РЕЦЕНЗЕНТАХ

Мэтью Бруше получил диплом инженера во французской Высшей школе электрики (Ecole Supérieure d'Electricité) (факультет информации, обработки и измерения сигналов), а степень доктора наук по обучению без учителя на многообразиях – в Страсбургском университете, Франция. В настоящее время он занимается высокопроизводительными вычислениями в нефтяной компании и работает над программой моделирования резервуаров следующего поколения.

Морис НТ Линь программирует на Python с 2003 года. Он получил степень доктора наук по биоинформатике и диплом с отличием бакалавра по молекулярной и клеточной биологии в Университете Мельбурна. В настоящее время работает научным сотрудником в Наньянском технологическом университете, Сингапур, и почетным научным сотрудником в Университете Мельбурна, Австралия. Морис – главный редактор журнала «Computational and Mathematical Biology» и соредактор журнала «The Python Papers». Недавно Морис основал первый стартап по синтетической биологии в Сингапуре, AdvanceSyn Pte. Ltd., и занимает в нем должность директора и технического руководителя. Область его научных интересов – изучение различных аспектов жизни – биологической, искусственной и искусственного интеллекта – путем применения методов информатики и статистики. В свободное время Морис любит читать, смаковать чашечку кофе, писать в свой персональный журнал или размышлять о многообразных проявлениях жизни. Его сайт находится по адресу http://maurice.vodien.com, а страница в LinkedIn – по адресу http://www.linkedin.com/in/mauriceling.

Ражим Ржезоржек – технический спец и разработчик по призванию души. Он основал и возглавлял исследовательский отдел в компании Seznam.cz, ведущей поисковой системе в Центральной Европе. Защитив докторскую диссертацию, он решил не останавливаться на достигнутом, а создать для удовлетворения своей страсти к машин-

ному обучению собственную иследовательскую компанию RaRe Consulting Ltd. Компания специализируется на заказных решениях в области добычи данных и разработке передовых систем для клиентов самого разного ранга: от многонациональных корпораций до только что основанных стартапов.

Радим – также автор ряда популярных проектов с открытым исходным кодом, в том числе gensim и smart_open.

Большой любитель исследования разных культур, Радим в последние годы жил со своей женой в разных странах и намеревается перебраться к Южную Корею. Но где бы Радим ни находился, он вместе со своей командой стремится всячески популяризировать методы обработки данных и помогать компаниям по всему миру извлекать максимум пользы из машинного обучения.

ПРЕДИСЛОВИЕ

Кто-то скажет, что вы держите эту книгу в руках (или в своей читалке) лишь благодаря счастливому совпадению. В конце концов, каждый год выходят миллионы книг, и их читают миллионы людей. А эту книгу читаете вы. А кто-то возразит, что в вашей встрече с этой книгой – или книги с вами – сыграли роль кое-какие алгоритмы машинного обучения. И мы, авторы, очень рады, что вы захотите подробнее узнать, как и почему.

В основном, эта книга отвечает на вопрос «как». Как следует обрабатывать данные, чтобы алгоритмы машинного обучения могли извлечь из них максимум полезной информации? Как выбрать подходящий алгоритм для решения задачи?

Но иногда мы задаемся также вопросом «почему». Почему важно правильно производить измерения? Почему в конкретной ситуации один алгоритм оказывается лучше другого?

Мы знаем, что для превращения в специалиста в этой области нужно знать гораздо больше. Ведь мы смогли затронуть лишь несколько «как» и уж совсем немного «почему». Но мы надеемся, что эта малость поможет вам быстро встать на ноги и двигаться дальше самостоятельно.

О содержании книги

В *главе 1* «*Введение в машинное обучение на языке Python*» читатель знакомится с основной идеей машинного обучения на очень простом примере. Но, несмотря на простоту, в этом примере имеет место опасность переобучения.

В *главе 2* «*Классификация в реальной жизни*» мы используем реальные данные, чтобы продемонстрировать классификацию и научить компьютер различать различные классы цветов.

В *главе 3* «*Кластеризация – поиск взаимосвязанных сообщений*» мы узнаем об эффективности модели набора слов, с помощью которой сумеем найти похожие сообщения, не «понимая» их смысла.

В *главе 4* «*Тематическое моделирование*» мы не станем ограничиваться отнесением сообщения только к одному кластеру, а свяжем с ним несколько тем, поскольку политематичность характерна для реальных текстов.

В *главе 5* «*Классификация – выявление плохих ответов*» мы узнаем, как применить дилемму смещения-дисперсии к отладке моделей машинного обучения, хотя эта глава посвящена в основном использованию логистической регрессии для оценки того, хорош или плох ответ пользователя на заданный вопрос.

В *главе 6* «*Классификация II – анализ эмоциональной окраски*» объясняется принцип работы наивного байесовского классификатора и описывается, как с его помощью узнать, несет ли твит положительный или отрицательный эмоциональный заряд.

В *главе 7* «*Регрессия*» объясняется, как использовать классический, но не утративший актуальности метод – регрессию – при обработке данных. Вы узнаете и о более сложных методах регрессии, в частности Lasso и эластичных сетях.

В *главе 8* «*Рекомендование*» мы построим систему рекомендования на основе выставленных потребителями оценок. Мы также узнаем, как формировать рекомендации, имея только данные о покупках, безо всяких оценок (которые пользователи выставляют далеко не всегда).

В *главе 9* «*Классификация по музыкальным жанрам*» мы предположим, что кто-то сознательно внес хаос в нашу огромную коллекцию музыкальных произведений, и единственная надежда навести порядок – поручить машине их классификацию. Как выяснится, иногда лучше довериться чужому опыту, чем создавать признаки самостоятельно.

В *главе 10* «*Машинное зрение*» мы применим методы классификации к обработке изображений, выделяя признаки из данных. Мы также увидим, как с помощью этих методов можно находить похожие изображения в наборе.

Из *главы 11* «*Понижение размерности*» мы узнаем о методах, позволяющих уменьшить объем данных, чтобы алгоритмы машинного обучения могли с ними справиться.

В *главе 12* «*Когда данных больше*» мы рассмотрим некоторые подходы, позволяющие успешно обрабатывать большие наборы данных, задействуя несколько ядер или вычислительные кластеры. Мы также познакомимся с основами облачных вычислений (на примере служб Amazon Web Services).

В *приложении «Где получить дополнительные сведения о машинном обучении»* перечислены многочисленные полезные ресурсы, посвященные этой теме.

Что необходимо для чтения этой книги

Предполагается, что вы знаете язык Python и умеете устанавливать библиотеки с помощью программы easy_install или pip. Мы не прибегаем к сложной математике типа математического анализа или матричной алгебры.

В этой книге используются следующие версии программ, хотя более поздние тоже должны подойти:

- Python 2.7 (весь код совместим также с версиями 3.3 и 3.4);
- NumPy 1.8.1;
- SciPy 0.13;
- scikit-learn 0.14.0.

На кого рассчитана эта книга

Книга рассчитана на программистов, пишущих на Python и желающих узнать о построении систем машинного обучения с помощью библиотек с открытым исходным кодом. Мы рассматриваем основные модели машинного обучения на примерах, взятых из реальной жизни.

Эта книга будет полезна также специалистам по машинному обучению, желающим использовать Python для создания своих систем. Python – гибкий язык для быстрой разработки прототипов, сами же базовые алгоритмы написаны на C или C++ и хорошо оптимизированы. Поэтому получается быстрый и надежный код, пригодный для производственных систем.

Графические выделения

В этой книге тип информации обозначается шрифтом. Ниже приведено несколько примеров с пояснениями.

Фрагменты кода внутри абзаца, имена таблиц базы данных, папок и файлов, URL-адреса, данные, которые вводит пользователь, и

адреса в Твиттере выделяются следующим образом: «Далее мы с помощью метода `poly1d()` строим модельную функцию по параметрам модели».

Кусок кода выглядит так:

```
[aws info]
AWS_ACCESS_KEY_ID = AAKIIT7HHF6IUSN3OCAA
AWS_SECRET_ACCESS_KEY = <ваш секретный ключ>
```

Входная и выходная информация командных утилит выглядит так:

```
>>> import numpy
>>> numpy.version.full_version
1.8.1
```

Новые термины и **важные фрагменты** выделяются полужирным шрифтом. Например, элементы графического интерфейса в меню или диалоговых окнах выглядят в книге так: «После остановки машины становится доступной кнопка **Change instance type**».

Предупреждения и важные примечания выглядят так.

Советы и рекомендации выглядят так.

Отзывы

Мы всегда рады отзывам читателей. Расскажите нам, что вы думаете об этой книге – что вам понравилось или, быть может, не понравилось. Читательские отзывы важны для нас, так как помогают выпускать книги, из которых вы черпаете максимум полезного для себя.

Чтобы отправить обычный отзыв, просто пошлите письмо на адрес `feedback@packtpub.com`, указав название книги в качестве темы. Если вы являетесь специалистом в некоторой области и хотели бы стать автором или соавтором книги, познакомьтесь с инструкциями для авторов по адресу `www.packtpub.com/authors`.

Поддержка клиентов

Счастливым обладателям книг Packt мы можем предложить ряд услуг, которые позволят извлечь из своего приобретения максимум пользы.

Загрузка кода примеров

Вы можете скачать код примеров ко всем книгам издательства Packt, купленным на сайте `http://www.packtpub.com`. Если книга была куплена в другом месте, зайдите на страницу `http://www.packtpub.com/support`, зарегистрируйтесь, и мы отправим файлы по электронной почте.

Код к этой книге имеется также на странице сайта GitHub по адресу `https://github.com/luispedro/BuildingMachineLearningSystemsWithPython`. Этот репозиторий поддерживается в актуальном состоянии, то есть включены исправления замеченных ошибок и обновления для новых версий Python и использованных в книге пакетов.

Опечатки

Мы проверяли содержимое книги со всем тщанием, но какие-то ошибки все же могли проскользнуть. Если вы найдете в нашей книге ошибку, в тексте или в коде, пожалуйста, сообщите нам о ней. Так вы избавите других читателей от разочарования и поможете нам сделать следующие издания книги лучше. При обнаружении опечатки просьба зайти на страницу `http://www.packtpub.com/support`, выбрать книгу, щелкнуть по ссылке **Errata Submission Form** и ввести информацию об опечатке. Проверив ваше сообщение, мы поместим информацию об опечатке на нашем сайте или добавим ее в список замеченных опечаток в разделе Errata для данной книги.

Список ранее отправленных опечаток можно просмотреть, выбрав название книги на странице `http://www.packtpub.com/books/content/support`. Запрошенная информация появится в разделе **Errata**.

Есть еще один отличный способ – зайти на сайт `www.TwoToReal.com`, где авторы стараются предоставлять техническую поддержку читателям и отвечать на их вопросы.

Нарушение авторских прав

Незаконное размещение защищенного авторским правом материала в Интернете – проблема для всех носителей информации. В издательстве Packt мы относимся к защите прав интеллектуальной собственности и лицензированию очень серьезно. Если вы обнаружите незаконные копии наших изданий в любой форме в Интернете, пожалуйста, незамедлительно сообщите нам адрес или название веб-сайта, чтобы мы могли предпринять соответствующие меры.

Просим отправить ссылку на вызывающий подозрение в пиратстве материал по адресу `copyright@packtpub.com`.

Мы будем признательны за помощь в защите прав наших авторов и содействие в наших стараниях предоставлять читателям полезные сведения.

Вопросы

Если вас смущает что-то в этой книге, вы можете связаться с нами по адресу `questions@packtpub.com`, и мы сделаем все возможное для решения проблемы.

ГЛАВА 1. Введение в машинное обучение на языке Python

Машинное обучение – это наука о том, как научить машину самостоятельно решать задачи. Вот так все просто. Но дьявол кроется в деталях, и именно поэтому вы читаете эту книгу.

Быть может, данных слишком много, а априорных знаний о них слишком мало. И вы надеетесь, что алгоритмы машинного обучения помогут справиться с этой проблемой, а потому начинаете в них разбираться. Но через некоторое время впадаете в ступор: какой же из мириада алгоритмов выбрать?

Или, быть может, вас заинтересовали общие вопросы машинного обучения, и вы начали читать блоги и статьи на эту тему. Все это показалось вам таким крутым и волшебным, что вы приступили к собственным исследованиям и загрузили простенькие данные в решающее дерево или в машину опорных векторов. Но вот эксперимент завершился, и возникает вопрос: а правильно ли он был поставлен? Насколько оптимальны полученные результаты? И как узнать, не существует ли алгоритма получше? Да и вообще – правильно ли были подобраны данные?

Вы не одиноки! Мы оба (авторы) проходили через это – искали сведения о том, как на практике выглядит то, о чем пишут в учебниках по машинному обучению. Оказалось, что очень многое – «черная магия», о которой авторы стандартных учебников забывают упомянуть. Так что в некотором роде эта книга написана нами для нас же, только на несколько лет моложе. Здесь вы найдете не только краткое введение в методы машинного обучения, но и уроки, которые мы извлекли, идя по этому пути. Мы надеемся, что благодаря этой книге дорога, ведущая в самые захватывающие области Информатики, станет чуть более гладкой.

Машинное обучение и Python – команда мечты

Цель машинного обучения – научить машину (точнее, программу) решать задачу, предъявив ей несколько примеров (с правильными и неправильными решениями). Предположим, что каждое утро, включив компьютер, вы делаете одно и то же: сортируете пришедшую почту, раскладывая письма в папки по темам. Через какое-то время эта работа вам надоест, и вы захотите ее автоматизировать. Один из возможных подходов – проанализировать собственное поведение и выписать правила, которыми вы руководствуетесь при сортировке писем. Однако это громоздкое и отнюдь не совершенное решение. Некоторые правила вы упустите из виду, другие сформулируете излишне детально. Гораздо лучше выбрать какой-то набор метаданных о письмах, задать пары (тело письма, имя папки) и поручить алгоритму вывести наилучший набор правил. Такие пары называются обучающими данными, а получившийся набор правил можно будет применить к будущим письмам, которых алгоритм еще не видел. Это и есть машинное обучение в простейшем виде.

Разумеется, машинное обучение (его еще часто называют добычей данных или прогностическим анализом) – не новая дисциплина. Наоборот, своими успехами в последние годы она обязана практическому применению проверенных временем методов и идей из других областей знания, в частности математической статистики. Цель человека – извлечь полезную информацию из данных, например, выявить скрытые закономерности и взаимосвязи. Читая об успешных применениях машинного обучения (Вы ведь уже открыли для себя сайт `www.kaggle.com`, правда?), вы убедитесь, что прикладная статистика вовсю используется специалистами.

Ниже вы увидите, что процесс поиска подходящего подхода к машинному обучению вовсе не линеен. Напротив, приходится многократно возвращаться назад, пробуя другие сочетания исходных данных и алгоритмов. Именно изыскательская природа этого процесса делает применение Python чрезвычайно уместным. Ведь Python, будучи интерпретируемым высокоуровневым языком программирования, как будто специально придуман для опробования разных вариантов. К тому же, он работает быстро. Конечно, он медленнее C и подобных ему статически типизированных языков. Но при наличии огромного числа простых в использовании библиотек, зачастую написанных на C, вам не придется жертвовать скоростью ради гибкости.

Что вы узнаете (и чего не узнаете из этой книги)

В этой книге вы найдете общий обзор обучающих алгоритмов, которые чаще всего применяются в различных отраслях машинного обучения, и узнаете, на что обращать внимание при их применении. Но по своему опыту мы знаем, что такие «интересные» вещи, как использование и настройка отдельных алгоритмов, например метода опорных векторов, классификации по ближайшим соседям, или их ансамблей, – занимают лишь малую часть рабочего времени специалиста по машинному обучению. А основное время тратится на довольно скучную работу:

- чтение и очистка данных;
- изучение исходных данные и попытки понять их;
- размышления о том, как лучше подать данные на вход алгоритма обучения;
- выбор подходящей модели и алгоритма;
- правильное измерение качества работы алгоритма.

В процессе изучения и осмысления исходных данных нам понадобится статистика и не очень сложная математика. И, как вы убедитесь, методы, казавшиеся такими скучными на занятиях по математике, могут стать по-настоящему увлекательными, когда применяются для анализа интересных данных.

Наше путешествие начинается с чтения данных. Вы поймете, что поиск ответа на вопрос, как быть с некорректными или отсутствующими данными, – скорее искусство, чем точная наука. Правильное решение воздастся сторицей, потому что позволит применить к данным больше обучающих алгоритмов, а, значит, повысит шансы на успех.

Когда данные будут представлены в виде структур данных в программе, вы захотите понять, с чем же все-таки работаете. Достаточно ли данных для ответа на интересующие вас вопросы? Если нет, то как добыть дополнительные данные? А, быть может, данных слишком много? Тогда нужно подумать, как лучше всего произвести выборку из них.

Часто бывает, что данные не стоит подавать сразу на вход алгоритма машинного обучения, а надо предварительно улучшить их. Алгоритм с благодарностью ответит на это более качественными результатами. Иногда даже оказывается, что простой алгоритм на предварительно обработанных данных работает лучше, чем очень изощренный алгоритм на данных в первозданном виде. Эта часть

работы, называемая **подготовкой признаков** (feature engineering), чаще всего оказывается безумно увлекательной задачей, приносящей немедленные плоды. Вы сразу же видите результаты творческого нестандартного подхода.

Таким образом, чтобы выбрать подходящий обучающий алгоритм, недостаточно просто ткнуть наугад в один из трех-четырех имеющихся в вашем арсенале (как вы скоро увидите, их вообще-то больше). Это вдумчивый процесс взвешивания различных критериев – от качества работы до функциональных требований. Вы хотите получить результат быстро, даже в ущерб качеству? Или предпочитаете потратить больше времени, но получить наилучший возможный результат? У вас есть отчетливое представление о будущих данных или лучше бы не делать слишком ограничительных предположений на этот счет?

Наконец, измерение качества работы – то место, где начинающий изучать машинное обучение может наделать больше всего ошибок. Есть ошибки простые, например, контроль результатов на тех же данных, на которых производилось обучение. А есть и посложнее, например, подача на вход несбалансированного обучающего набора. Как и раньше, именно данные определяют успех или неудачу всего предприятия.

Как мы видим, лишь четвертый пункт в нашем списке относится собственно к возне с алгоритмами. И тем не менее, мы надеемся убедить вас в том, что и остальные четыре задачи могут быть не менее увлекательными. Нам хотелось бы, чтобы, прочитав книгу до конца, вы влюбились не столько в алгоритмы обучения, сколько в сами данные.

Поэтому мы не станем нагружать вас теоретическими вопросами разнообразных алгоритмов машинного обучения, поскольку на эту тему существует немало отличных книг (их перечень вы найдете в приложении), а вместо этого попытаемся развить интуицию настолько, чтобы вы поняли идею и смогли сделать первые шаги. Так что эту книгу ни в коем случае нельзя считать *авторитетным руководством* по машинному обучению. Это скорее учебник для начинающих. Мы надеемся возбудить в вас любопытство настолько, чтобы вам захотелось продолжить изучение этой интереснейшей области знаний.

Далее в этой главе мы познакомимся с основами библиотек NumPy и SciPy для Python и обучим наш первый алгоритм с помощью библиотеки scikit-learn. По ходу дела мы введем основные понятия машинного обучения, которые будут использоваться на протяжении всей книги. В последующих главах мы подробно рассмотрим все

пять вышеперечисленных шагов, демонстрируя различные аспекты машинного обучения на примерах из разных областей.

Что делать, если вы застряли

Мы старались как можно понятнее изложить все идеи, необходимые, чтобы воспроизвести описанные в книге примеры. Тем не менее, не исключено, что вы зайдете в тупик. Причины могут быть различными: опечатки, непредвиденные комбинации версий пакетов или недопонимание.

В таком случае обратиться за помощью можно в разные места. Скорее всего, кто-то уже сталкивался с вашей проблемой, и решение имеется на одном из следующих сайтов вопросов и ответов.

- `http://metaoptimize.com/qa`: этот сайт посвящен исключительно машинному обучению. Почти на каждый вопрос имеются квалифицированные ответы от специалистов. Даже если у вас нет никаких вопросов, полезно взять за правило время от времени заходить сюда и читать ответы.
- `http://stats.stackexchange.com`: этот сайт носит название Cross Validated, он похож на MetaOptimize, но посвящен больше вопросам из области статистики.
- `http://stackoverflow.com`: этот сайт во многом напоминает предыдущий, но диапазон рассматриваемых вопросов программирования гораздо шире. Так, на нем задается больше вопросов о некоторых используемых в этой книге пакетах, в частности, SciPy и matplotlib.
- Канал `#machinelearning` на сайте `https://freenode.net/`: это IRC-канал, посвященный проблемам машинного обучения. Здесь собирается небольшое, но очень активное и всегда готовое оказать помощь сообщество специалистов по машинному обучению.
- `http://www.TwoToReal.com`: это сайт мгновенных вопросов и ответов, созданный авторами книги для оказания помощи по вопросам, не попадающим ни в одну из описанных выше категорий. Если вы зададите здесь вопрос, то один из авторов немедленно получит сообщение, если находится в сети, и будет готов вступить с вами в беседу.

Как уже было сказано в начале книги, наша цель – помочь вам быстро освоить азы машинного обучения и дальше двигаться уже самостоятельно. Поэтому мы горячо рекомендуем составить свой список

блогов, относящихся к машинному обучению, и регулярно заглядывать в них. Это лучший способ узнать, что работает, а что – нет.

Единственный блог, о котором мы хотели бы упомянуть прямо сейчас (и подробнее рассказать в приложении) – `http://blog.kaggle.com`, блог компании Kaggle, которая проводит конкурсы по машинному обучению. Обычно победителям предлагается рассказать о своих подходах к конкурсным задачам, о том, какие стратегии не сработали, и как они пришли к победной стратегии. Этот блог обязателен к прочтению, пусть даже он окажется единственным.

Приступая к работе

В предположении, что Python уже установлен (годится любая версия, начиная с 2.7), нам нужно еще установить пакеты NumPy и SciPy для численных расчетов и matplotlib для визуализации.

Введение в NumPy, SciPy и matplotlib

Прежде чем говорить о конкретных алгоритмах машинного обучения, следует определиться с тем, как лучше хранить данные, которые мы будем обрабатывать. Это важно, потому что даже самый изощренный обучающий алгоритм ничем не поможет, если он не завершается. А так может случиться просто потому, что доступ к данным слишком медленный. Или потому, что выбранное представление данных заставляет операционную систему постоянно пробуксовывать. Добавьте сюда тот факт, что Python все-таки интерпретируемый язык (хотя и хорошо оптимизированный), слишком медленный для численных расчетов по сравнению с C или FORTRAN. Но тогда возникает вопрос – почему же так много ученых и компаний ставят свое благополучие на Python, даже в задачах, требующих очень интенсивных вычислений?

Дело в том, что, работая на Python, очень просто перепоручить численные расчеты низкоуровневым расширениям, реализованным на C или FORTRAN. Именно так устроены библиотеки NumPy и SciPy (`http://scipy.org/Download`). В этом тандеме NumPy отвечает за высоко оптимизированные многомерные массивы – основную структуру данных для большинства современных алгоритмов. А SciPy на базе этих массивов реализует быстрые численные алгоритмы. Наконец, matplotlib (`http://matplotlib.org/`) – пожалуй, самая удобная и функционально развитая библиотека для построения высококачественных графиков на Python.

Установка Python

По счастью, для всех популярных операционных систем, то есть Windows, Mac и Linux, существуют готовые инсталляторы NumPy, SciPy и matplotlib. Если вы не уверены, сможете ли справиться с их установкой, то установите дистрибутив Anaconda Python (его можно скачать с сайта `https://store.continuum.io/cshop/anaconda/`), созданный Трэвисом Олифантом (Travis Oliphant), основателем и активным автором проекта SciPy. От других дистрибутивов, например Enthought Canopy (`https://www.enthought.com/downloads/`) или Python(x,y) (`http://code.google.com/p/pythonxy/wiki/Downloads`), Anaconda отличается полной совместимостью с Python 3 – версией Python, которой мы будем пользоваться в этой книге.

NumPy как средство эффективной и SciPy как средство интеллектуальной обработки данных

Давайте разберем несколько простых примеров применения NumPy, а затем посмотрим, что SciPy дает сверх того. Попутно мы сделаем первую попытку построить графики с помощью чудесного пакета Matplotlib.

Желающие углубиться в тему могут ознакомиться с интересными примерами применения NumPy, приведенными в пособии по адресу `http://www.scipy.org/Tentative_NumPy_Tutorial`.

Весьма полезна также книга Ivan Idris «NumPy Beginner's Guide» (второе издание), вышедшая в издательстве Packt Publishing. Дополнительные учебные материалы можно найти на сайте `http://scipy-lectures.github.com`, а также в официальном руководстве по SciPy по адресу `http://docs.scipy.org/doc/scipy/reference/tutorial`.

В этой книге мы пользуемся версиями NumPy 1.8.1 и SciPy 0.14.0.

Изучаем NumPy

Итак, импортируем NumPy и немного поэкспериментируем. Для этого нужно запустить интерактивную оболочку Python:

```
>>> import numpy
>>> numpy.version.full_version
1.8.1
```

Поскольку мы не хотим загрязнять пространство имен, то, конечно же, не станем писать такой код:

```
>>> from numpy import *
```

Так как, к примеру, пакет `numpy.array` мог бы замаскировать пакет `array` из стандартной библиотеки Python. Вместо этого мы будем пользоваться следующим удобным сокращением:

```
>>> import numpy as np
>>> a = np.array([0,1,2,3,4,5])
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.ndim
1
>>> a.shape
(6,)
```

Только что мы создали массив – точно так же, как список в Python. Однако в массивах NumPy хранится дополнительная информация о форме. В данном случае мы имеем одномерный массив с шестью элементами. Пока никаких сюрпризов.

Теперь можно преобразовать этот массив в двумерную матрицу:

```
>>> b = a.reshape((3,2))
>>> b
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> b.ndim
2
>>> b.shape
(3, 2)
```

Интересные вещи начинаются с осознанием того, как сильно оптимизирован пакет NumPy. Например, в следующих операциях копирование по возможности не производится:

```
>>> b[1][0] = 77
>>> b
array([[ 0, 1],
       [77, 3],
       [ 4, 5]])
>>> a
array([ 0, 1, 77, 3, 4, 5])
```

В данном случае мы изменили один элемент `b` с `2` на `77` и, как видим, это изменение отразилось и на массиве `a`. Но если копирование таки необходимо, то это можно организовать:

```
>>> c = a.reshape((3,2)).copy()
>>> c
array([[ 0, 1],
       [77, 3],
       [ 4, 5]])
>>> c[0][0] = -99
>>> a
array([ 0, 1, 77, 3, 4, 5])
>>> c
array([[-99, 1],
       [ 77, 3],
       [ 4, 5]])
```

Теперь `c` и `a` – совершенно независимые копии.

Еще одно преимущество массивов NumPy – распространение операций на отдельные элементы. Например, умножение массива NumPy на число порождает массив такого же размера, в котором все элементы умножены на это число:

```
>>> d = np.array([1,2,3,4,5])
>>> d*2
array([ 2, 4, 6, 8, 10])
```

То же справедливо и для других операций:

```
>>> d**2
array([ 1, 4, 9, 16, 25])
```

Сравните с обычным списком Python:

```
>>> [1,2,3,4,5]*2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> [1,2,3,4,5]**2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

Разумеется, прибегая к массивам NumPy, мы приносим в жертву гибкость списков Python. Такие простые операции, как добавление или удаление элементов, для массивов NumPy реализуются сложнее. Но, к счастью, в нашем распоряжении есть и то, и другое, так что можно выбирать подходящий инструмент по ситуации.

Индексирование

Своей эффективностью библиотека NumPy отчасти обязана разнообразием способов доступа к массивам. Помимо знакомого по спискам индексирования, можно использовать в качестве индексов сами массивы:

```
>>> a[np.array([2,3,4])]
array([77, 3, 4])
```

Учитывая, что логические условия также распространяются на отдельные элементы, мы получаем очень удобный способ доступа к данным:

```
>>> a>4
array([False, False, True, False, False, True], dtype=bool)
>>> a[a>4]
array([77, 5])
```

Следующая команда позволяет ограничить выбросы:

```
>>> a[a>4] = 4
>>> a
array([0, 1, 4, 3, 4, 4])
```

Поскольку эта бывает необходимо очень часто, существует специальная функция `clip`, которая позволяет задать обе границы с помощью одного вызова:

```
>>> a.clip(0,4)
array([0, 1, 4, 3, 4, 4])
```

Обработка отсутствующих значений

Средства индексирования NumPy оказываются особенно полезны, когда нужно произвести предварительную обработку данных, прочитанных из текстового файла. Чаще всего данные содержат некорректные значения, которые мы можем пометить как не-числа с помощью константы `numpy.NAN`:

```
>>> c = np.array([1, 2, np.NAN, 3, 4]) # допустим, что мы прочли
                                       # это из текстового файла
>>> c
array([ 1., 2., nan, 3., 4.])
>>> np.isnan(c)
array([False, False, True, False, False], dtype=bool)
>>> c[~np.isnan(c)]
array([ 1., 2., 3., 4.])
>>> np.mean(c[~np.isnan(c)])
2.5
```

Сравнение времени работы

Давайте сравним производительность NumPy и обычных списков Python. Следующая программа вычисляет сумму квадратов чисел от 1 до 1000 и замеряет время работы. Для большей точности мы прогоним ее 10 000 раз и напечатаем общее время.

```
import timeit
normal_py_sec = timeit.timeit('sum(x*x for x in range(1000))',
                              number=10000)
naive_np_sec = timeit.timeit(
                'sum(na*na)',
                setup="import numpy as np; na=np.arange(1000)",
                number=10000)
good_np_sec = timeit.timeit(
                'na.dot(na)',
                setup="import numpy as np; na=np.arange(1000)",
                number=10000)

print("Normal Python: %f sec" % normal_py_sec)
print("Naive NumPy: %f sec" % naive_np_sec)
print("Good NumPy: %f sec" % good_np_sec)

Normal Python: 1.050749 sec
Naive NumPy: 3.962259 sec
Good NumPy: 0.040481 sec
```

Сделаем два любопытных наблюдения. Во-первых, если использовать NumPy только для хранения данных (Naive NumPy), то программа работает в 3,5 раза дольше. Это странно, потому что мы ожидали, что написанное на C расширение будет работать гораздо быстрее. Одна из причин заключается в том, что доступ к отдельным элементам массива из Python обходится довольно дорого. Ускорение мы получим лишь в том случае, если сумеем применить алгоритм, не выходя за пределы оптимизированного кода расширения. Другое наблюдение поражает: использование входящей в NumPy функции `dot()`, которая делает в точности то же самое, приводит к 25-кратному ускорению. Вывод – реализуя любой алгоритм, нужно стремиться к тому, чтобы вместо цикла по отдельным элементам массива на Python воспользоваться какой-нибудь оптимизированной функцией, входящей в состав NumPy или SciPy.

Однако за быстродействие приходиться расплачиваться. Работая с массивами NumPy, мы утрачиваем невероятную гибкость списков Python, в которых можно хранить практически всё. Все элементы массива NumPy должны иметь одинаковый тип.

```
>>> a = np.array([1,2,3])
>>> a.dtype
dtype('int64')
```

Если мы попытаемся использовать элементы разных типов, как в следующем примере, то NumPy постарается привести их к одному и тому же, наиболее разумному в данных обстоятельствах, типу:

```
>>> np.array([1, "stringy"])
array(['1', 'stringy'], dtype='<U7')
>>> np.array([1, "stringy", set([1,2,3])])
array([1, stringy, {1, 2, 3}], dtype=object)
```

Изучаем SciPy

Поверх эффективных структур данных NumPy библиотека SciPy надстраивает многочисленные алгоритмы, работающие с массивами. Какой бы численный алгоритм из описываемых в современных учебниках ни взять, с большой вероятностью он будет так или иначе поддержан в SciPy. Это относится к операциям над матрицами, линейной алгебре, оптимизации, кластеризации, пространственным операциям и даже быстрому преобразованию Фурье. Поэтому прежде чем приступать к самостоятельной реализации численного алгоритма, обязательно посмотрите, нет ли его в модуле `scipy`.

Для удобства полное пространство имен NumPy доступно также через SciPy. Поэтому, начиная с этого места, мы будем обращаться ко всем средствам NumPy с помощью пространства имен SciPy. В корректности этого подхода легко убедиться, сравнив ссылки на любую функцию:

```
>>> import scipy, numpy
>>> scipy.version.full_version
0.14.0
>>> scipy.dot is numpy.dot
True
```

Алгоритмы разбиты на следующие группы.

SciPy packages	Functionalities
`cluster`	• Иерархическая кластеризация (`cluster.hierarchy`) • Векторное квантование / метод *k*-средних (`cluster.vq`)
`constants`	• Физические и математические константы • Методы преобразования
`fftpack`	Алгоритм дискретного преобразования Фурье
`integrate`	Процедуры интегрирования
`interpolate`	Интерполяция (линейная, кубическая и т. п.)
`io`	Ввод-вывод данных
`linalg`	Процедуры линейной алгебры на основе оптимизированных библиотек BLAS и LAPACK
`ndimage`	Обработка многомерных изображений

SciPy packages	Functionalities
odr	Ортогональная регрессия
optimize	Оптимизация (нахождение минимумов и корней)
signal	Обработка сигналов
sparse	Разреженные матрицы
spatial	Пространственные структуры данных и алгоритмы
special	Специальные математические функции, например, функция Бесселя и якобиан
stats	Математическая статистика

Для нас наибольший интерес представляют пакеты `scipy.stats`, `scipy.interpolate`, `scipy.cluster` и `scipy.signal`. Сейчас мы вкратце рассмотрим возможности пакета stats, а изучение остальных отложим до тех глав, где они понадобятся.

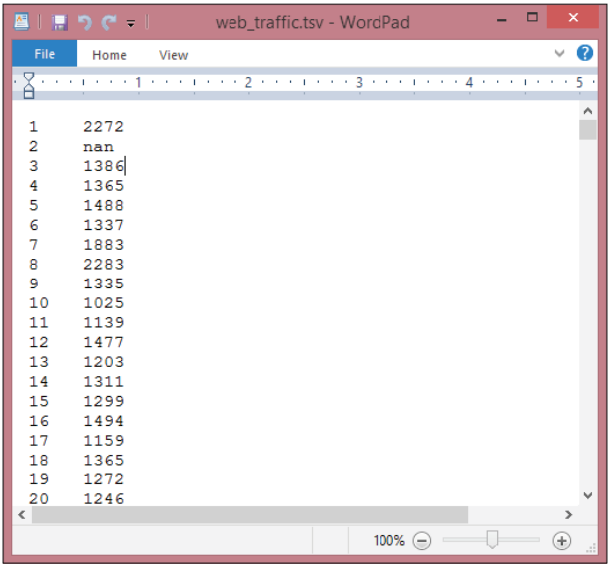
Наше первое (простенькое) приложение машинного обучения

Хватит слов, возьмем для примера гипотетическую недавно образованную компанию MLaaS, которая предоставляет платные услуги машинного обучения через Интернет. Наша компания растет, и ей понадобилось улучшить инфраструктуру для обслуживания поступающих запросов. Мы не хотим выделять слишком много ресурсов, потому что это будет дорого стоить. С другой стороны, если не зарезервировать достаточно ресурсов, то мы потеряем деньги из-за невозможности обслужить все запросы. Вопрос заключается в том, когда мы достигнем предельной пропускной способности инфраструктуры, которую оценили в 100 000 запросов в час. Хотелось бы заранее знать, когда запросить дополнительные серверы в облаке, чтобы обслужить все входящие запросы, но не платить за неиспользованные ресурсы.

Чтение данных

Мы собрали и агрегировали статистику веб за последний месяц, эти данные находятся в файле `ch01/data/web_traffic.tsv` (расширение `tsv` означает, что значения разделены знаками табуляции). Данные представляют собой количество запросов в час. В каждой строке указан час (по порядку) и количество запросов за этот час.

Ниже показаны первые несколько строк:



Метод `genfromtxt()` из библиотеки SciPy позволяет легко прочитать эти данные:

```
>>> import scipy as sp
>>> data = sp.genfromtxt("web_traffic.tsv", delimiter="\t")
```

Чтобы столбцы правильно распознавались, необходимо указать, что разделителем служит знак табуляции.

Простая проверка показывает, что данные прочитаны верно:

```
>>> print(data[:10])
[[ 1.00000000e+00 2.27200000e+03]
 [ 2.00000000e+00 nan]
 [ 3.00000000e+00 1.38600000e+03]
 [ 4.00000000e+00 1.36500000e+03]
 [ 5.00000000e+00 1.48800000e+03]
 [ 6.00000000e+00 1.33700000e+03]
 [ 7.00000000e+00 1.88300000e+03]
 [ 8.00000000e+00 2.28300000e+03]
 [ 9.00000000e+00 1.33500000e+03]
 [ 1.00000000e+01 1.02500000e+03]]
>>> print(data.shape)
(743, 2)
```

Как видим, создан двумерный массив, содержащий 743 результата измерений.

Предварительная обработка и очистка данных

Для SciPy удобнее представить данные в виде двух векторов длиной 743 каждый. Первый вектор, x, будет содержать часы, второй, y, – количество запросов в течение соответствующего часа. Для такого разделения массива в SciPy предусмотрена специальная форма индексов, позволяющая выбирать отдельные столбцы:

```
x = data[:,0]
y = data[:,1]
```

В SciPy есть и много других способов выбрать данные из массива. Дополнительные сведения об индексировании, вырезании и итерировании см. в руководстве по адресу `http://www.scipy.org/Tentative_NumPy_Tutorial`.

Проблема в том, что в векторе y встречаются недопустимые значения – nan. И что с ними делать? Посмотрим, сколько раз встречается такое значение:

```
>>> sp.sum(sp.isnan(y))
8
```

Как видим, отсутствуют всего 8 из 743 значений, так что можно спокойно удалить их. Напомним, что массив SciPy можно индексировать другим массивом. Метод `sp.isnan(y)` возвращает массив булевых величин, показывающих, является элемент числом или нет. Оператор ~ вычисляет логическое отрицание этого массива, то есть позволяет выбрать из векторов x и y только элементы, содержащие число:

```
>>> x = x[~sp.isnan(y)]
>>> y = y[~sp.isnan(y)]
```

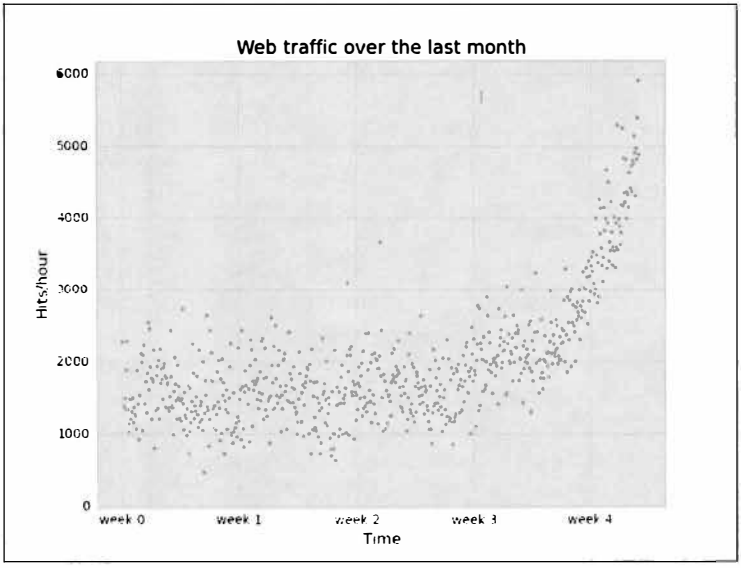
Чтобы составить первое представление о данных, нанесем на график диаграмму рассеяния, воспользовавшись библиотекой matplotlib. В этой библиотеке есть пакет pyplot, имитирующий интерфейс MATLAB – очень удобный и простой в использовании:

```
>>> import matplotlib.pyplot as plt
>>> # представляем точки (x,y) кружочками диаметра 10
>>> plt.scatter(x, y, s=10)
>>> plt.title("Web traffic over the last month")
>>> plt.xlabel("Time")
>>> plt.ylabel("Hits/hour")
>>> plt.xticks([w*7*24 for w in range(10)],
               ['week %i' % w for w in range(10)])
>>> plt.autoscale(tight=True)
```

```
>>> # рисуем полупрозрачную сетку пунктирными линиями
>>> plt.grid(True, linestyle='-', color='0.75')
>>> plt.show()
```

Дополнительные сведения о построении графиков можно найти на странице `http://matplotlib.org/users/pyplot_tutorial.html`.

Из графика видно, что в первые несколько недель трафик был более-менее стабильным, но в последнюю неделю наблюдался резкий рост.



Выбор подходящей модели и обучающего алгоритма

Составив представление о данных, вернемся к исходному вопросу: как долго наш сервер сможет обслуживать входящий трафик? Для ответа на него нужно сделать следующее:

1. Построить модель, отражающую зашумленные данные.
2. С помощью этой модели экстраполировать данные на будущее и определить, в какой момент следует расширить инфраструктуру.

Прежде чем строить модель...

Можно считать, что модель – это упрощенное теоретическое приближение к сложной реальности. Поэтому в модели всегда существует неточность, которую называют погрешностью аппроксимации. Именно величина погрешности и позволяет выбрать подходящую модель из множества возможных. Погрешность вычисляется как квадрат расстояния между реальными и предсказанными моделью данными; если `f` – обученная модельная функция, то погрешность равна:

```
def error(f, x, y):
  return sp.sum((f(x)-y)**2)
```

Векторы `x` и `y` содержат подготовленные ранее статистические данных о запросах. Тут мы наглядно видим удобство векторных функций в SciPy. Предполагается, что обученная модель принимает вектор и возвращает результаты в виде вектора того же размера, так что разность между результатом и вектором `y` корректно определена.

Начнем с прямой линии

Предположим ненадолго, что данные можно смоделировать прямой линией. Тогда наша задача – найти такую прямую, для которой погрешность аппроксимации минимальна. Именно это и делает функция `polyfit()` из SciPy. Получив на входе векторы `x`, `y` и требуемую степень полинома (для прямой линии степень равна 1), она находит модельную функцию, которая минимизирует определенную выше функцию погрешности:

```
fp1, residuals, rank, sv, rcond = sp.polyfit(x, y, 1, full=True)
```

Функция `polyfit()` возвращает параметры подобранной модельной функции `fp1`. Если задать `full=True`, то мы получим дополнительную информацию. Нас из нее интересуют только невязки, которые и описывают погрешность аппроксимации:

```
>>> print("Параметры модели: %s" % fp1)
Параметры модели: [ 2.59619213 989.02487106]
>>> print(residuals)
[  3.17389767e+08]
```

Это означает, что наилучшую линейную аппроксимацию дает следующая функция:

```
f(x) = 2.59619213 * x + 989.02487106.
```

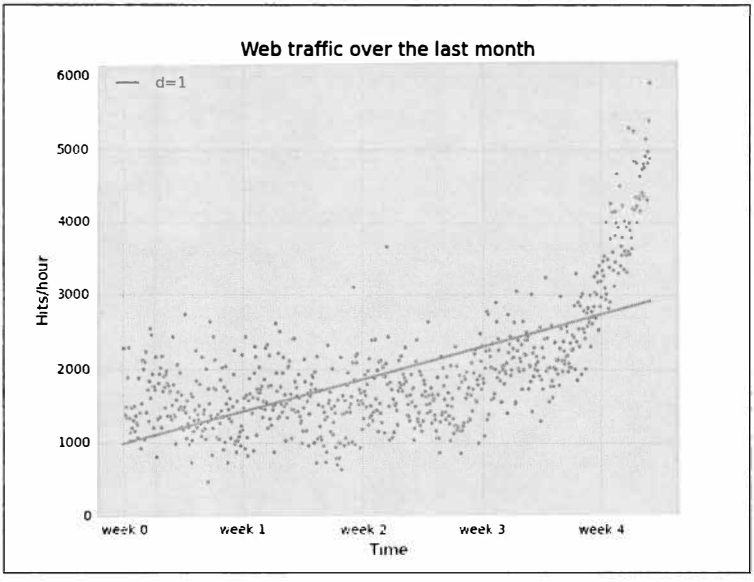
Затем мы воспользуемся функцией `poly1d()` для построения модельной функции по параметрам модели:

```
>>> f1 = sp.poly1d(fp1)
>>> print(error(f1, x, y))
317389767.34
```

Мы задали `full=True`, чтобы получить детали процесса поиска аппроксимации. Обычно этого не делают, и тогда возвращаются только параметры модели. Теперь мы можем нанести на график функцию `f1()` и посмотреть на свою первую обученную модель. К приведенным ранее командам нужно всего лишь добавить следующий код:

```
fx = sp.linspace(0,x[-1], 1000) # сгенерировать значения X для графика
plt.plot(fx, f1(fx), linewidth=4)
plt.legend(["d=%i" % f1.order], loc="upper left")
```

В результате получится такой график:



Сразу видно, что наше исходное предположение о линейности модели плохо вяжется с реальной картиной. И потом: погрешность 317 389 767.34 – это хорошо или плохо?

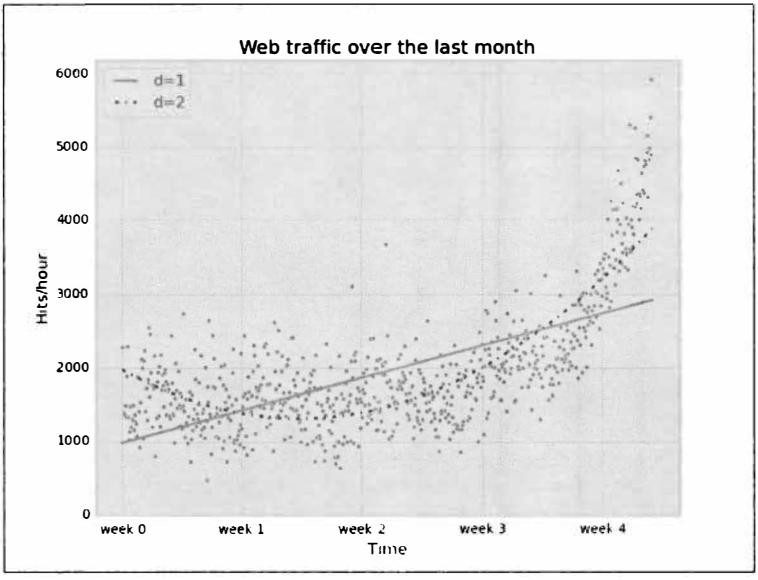
Абсолютная величина погрешности редко бывает полезна сама по себе. Но если сравнить две конкурирующие модели, то на основании погрешности можно судить, какая из них лучше. И хотя мы безусловно не станем пользоваться своей первой моделью, в общей последовательности действий она играет важную роль. Она станет эталоном, с которым будут сравниваться другие модели.

В поисках лучшего решения

Попробуем теперь усложнить модель, взяв полином степени 2. Быть может, она сможет лучше отразить наши данные.

```
>>> f2p = sp.polyfit(x, y, 2)
>>> print(f2p)
array([ 1.05322215e-02, -5.26545650e+00, 1.97476082e+03])
>>> f2 = sp.poly1d(f2p)
>>> print(error(f2, x, y))
179983507.878
```

Получается такой график.



Погрешность составляет 179 983 507.878, почти вдвое меньше погрешности линейной модели. Это хорошо, но дается не даром: функция стала сложнее, а значит, `polyfit()` должна вычислять один дополнительный параметр. Аппроксимирующий полином теперь выглядит так:

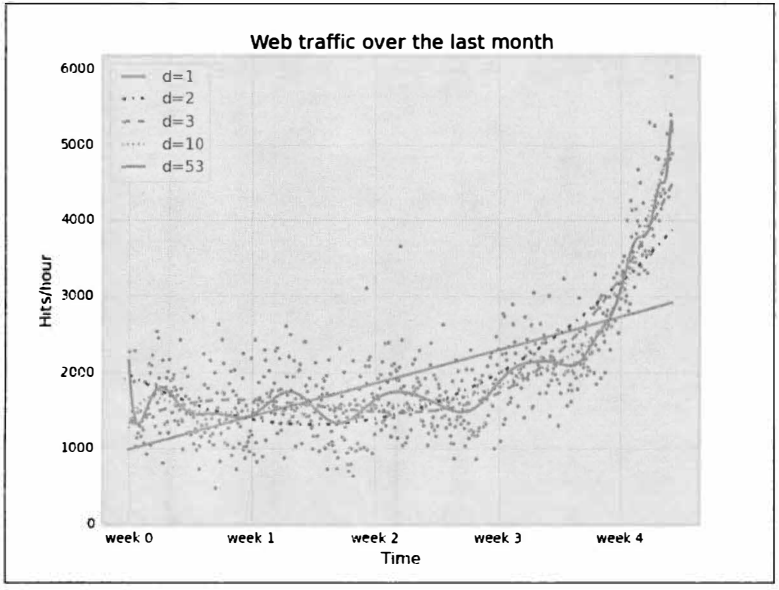
```
f(x) = 0.0105322215 * x**2 - 5.26545650 * x + 1974.76082
```

Но если усложнение функции улучшает результаты, то почему не пойти по этому пути дальше? Попробуем степени 3, 10 и 100.

Интересно, что вместо полинома степени 100 мы видим на консоли многочисленные сообщения

```
RankWarning: Polyfit may be poorly conditioned
```

Это означает, что из-за погрешностей при вычислениях `polyfit` не смогла найти аппроксимирующий полином степени 100 и сочла, что степени 53 вполне достаточно.



Похоже, что чем лучше кривая аппроксимирует данные, тем она сложнее. О точности аппроксимации говорят вычисленные погрешности:

```
Error d=1: 317,389,767.339778
Error d=2: 179,983,507.878179
Error d=3: 139,350,144.031725
Error d=10: 121,942,326.363461
Error d=53: 109,318,004.475556
```

Но внимательно приглядевшись к аппроксимирующим кривым, мы начинаем сомневаться в том, что они действительно описывают процесс, порождающий данные. Иначе говоря, правильно ли наша модель представляет групповое поведение пользователей, посещающих наш сайт? Для полиномов степени 10 и 53 характерны резкие колебания. Складывается впечатление, что модель слишком уж близко повторяет наблюдаемые данные. Настолько близко, что улавливает не только истинный процесс, но и шум. Этот феномен называется **переобучением**.

Придя к такому выводу, мы можем выбрать один из следующих путей:

- выбрать какую-то аппроксимирующую полиномиальную модель;
- перейти на более сложные модели, например сплайновые;
- переосмыслить данные и начать все сначала.

Из пяти исследованных моделей линейная слишком проста, а модели степени 10 и 53 переобучены. Лишь полиномы второй и третьей степени более-менее соответствуют данным. Но если экстраполировать их в обе стороны, то поведение становится ни с чем не сообразным.

Переход на другой класс моделей – более сложных – тоже не кажется шагом в правильном направлении. Какими аргументами мы могли бы подкрепить выбор класса? Похоже, мы не до конца понимаем природу наших данных.

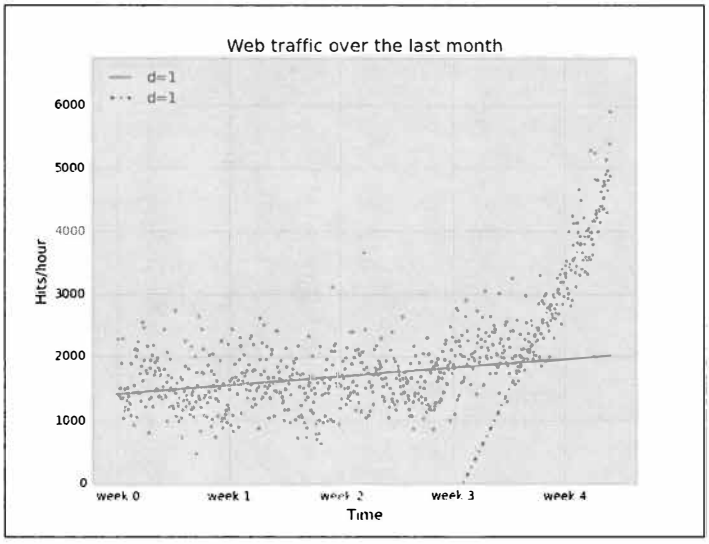
Отступить, чтобы двинуться вперед – другой взгляд на данные

Итак, отступим на шаг назад и взглянем на данные еще раз. Между третьей и четвертой неделей наблюдается явный изгиб. Так давайте разделим данные и обучим две прямые линии, взяв в качестве точки раздела неделю 3,5:

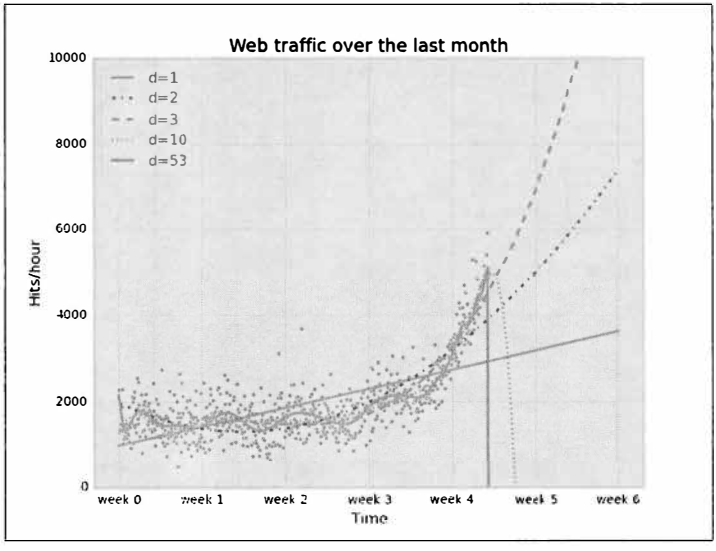
```
inflection = 3.5*7*24   # вычислить положение точки изгиба в часах
xa = x[:inflection]     # данные до точки изгиба
ya = y[:inflection]
xb = x[inflection:]     # данные после нее
yb = y[inflection:]

fa = sp.polyld(sp.polyfit(xa, ya, 1))
fb = sp.polyld(sp.polyfit(xb, yb, 1))

fa_error = error(fa, xa, ya)
fb_error = error(fb, xb, yb)
print("Error inflection=%f" % (fa_error + fb_error))
Error inflection=132950348.197616
```

Первую прямую мы обучаем на данных до точки изгиба, а вторую – на оставшихся.

Очевидно, что такая комбинация двух прямых аппроксимирует данные гораздо лучше, чем любая из рассмотренных выше моделей. Тем не менее, погрешность больше, чем в случае полиномов высокой степени. Так можно ли вообще доверять погрешности?

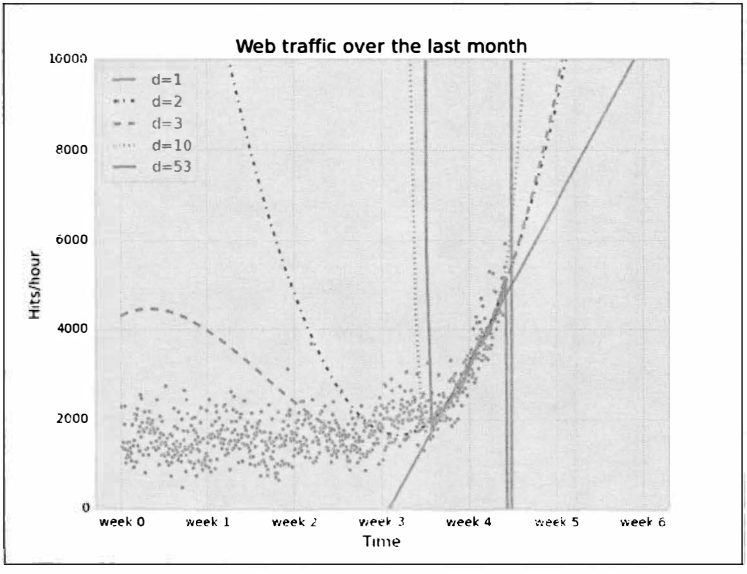


Иначе говоря, почему прямой, аппроксимирующей только данные за последние полторы недели, мы доверяем больше, чем более сложным моделям? Потому что полагаем, что она лучше аппроксимирует будущие данные. Если продолжить график в будущее, то будет видно, правы ли мы (**d=1** соответствует самой первой прямой).



Модели степени 10 и 53, похоже, не обещают нашему стартапу радужное будущее. Они так стремились как можно лучше соответствовать уже имеющимся данным, что оказались совершенно непригодны для экстраполяции. Это переобучение. С другой стороны, модели небольших степеней недостаточно хорошо улавливают характер данных. Это называется **недообучением**.

Но давайте будем справедливы к моделям степени 2 и выше и посмотрим, как они поведут себя, если обучить их только на данных за последние полторы недели. Ведь мы исходим из того, что данные за последний период больше говорят о будущем, чем предшествующие данные. Результат показан на следующем бредовом графике, который еще раз подтверждает, насколько серьезной может оказаться проблема переобучения.



И все же, опираясь на погрешности моделей обученных только на данных за последние полторы недели, мы должны были бы выбрать самую сложную (отметим, что и погрешность вычисляется только за период времени после точки изгиба):

```
Error d=1:  22,143,941.107618
Error d=2:  19,768,846.989176
Error d=3:  19,766,452.361027
Error d=10: 18,949,339.348539
Error d=53: 18,300,702.038119
```

Обучение и тестирование

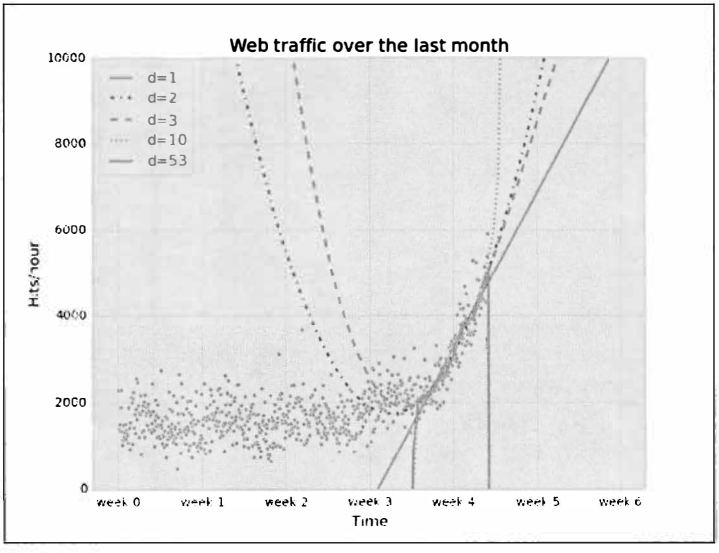
Если бы у нас были какие-то будущие данные, на которых можно было бы проверить модель, то мы смогли бы оценить выбор модели, опираясь на одну лишь результирующую погрешность аппроксимации.

Но хотя нам не дано заглянуть в будущее, мы можем и даже должны имитировать его, зарезервировав часть имеющихся данных. К примеру, удалим какой-то процент данных и обучим модель на оставшихся. А затем воспользуемся отложенными в сторонку данными для вычисления погрешности. Поскольку обученная модель ничего не знает о зарезервированных данных, то мы сможем составить более реалистичную картину ее поведения в будущем.

Погрешности для моделей, обученных только на данных после точки изгиба, дают совершенно иную картину:

```
Error d=1: 6397694.386394
Error d=2: 6010775.401243
Error d=3: 6047678.658525
Error d=10: 7037551.009519
Error d=53: 7052400.001761
```

Взгляните на график ниже:



Вот теперь, похоже, победитель окончательно определился: у модели степени 2 наименьшая погрешность на тестовых данных, то есть на

тех данных, которые не предъявлялись во время обучения. И это дает надежду, что ее поведение на будущих данных не станет сюрпризом.

Ответ на первоначальный вопрос

Наконец мы нашли модель, которая, как нам кажется, наилучшим образом описывает истинный процесс; теперь нетрудно определить, когда мы выйдем на уровень 100 000 запросов в час. Нужно лишь вычислить, когда модельная функция принимает значение 100 000.

Для полинома степени 2 мы можем просто найти обратную функцию и вычислить ее значение в точке 100 000. Но, конечно, хотелось бы уметь решать эту задачу для любой модельной функции.

Для этого достаточно вычесть из полинома 100 000 и найти корни нового полинома. В модуле SciPy `optimize` имеется функция `fsolve`, решающая эту задачу, ей нужно только указать начальную позицию с помощью параметра `x0`. Поскольку любая точка в нашем файле данных соответствует одному часу, а всего имеется 743 точки, в качестве начальной позиции зададим какое-то значение, большее максимума. Пусть `fbt2` – полином-победитель степени 2.

```
>>> fbt2 = sp.poly1d(sp.polyfit(xb[train], yb[train], 2))
>>> print("fbt2(x)= \n%s" % fbt2)
fbt2(x)=
        2
0.086 x - 94.02 x + 2.744e+04
>>> print("fbt2(x)-100,000= \n%s" % (fbt2-100000))
fbt2(x)-100,000=
        2
0.086 x - 94.02 x - 7.256e+04
>>> from scipy.optimize import fsolve
>>> reached_max = fsolve(fbt2-100000, x0=800)/(7*24)
>>> print("100,000 hits/hour expected at week %f" % reached_max[0])
```

Таким образом, мы ожидаем выхода на уровень 100 000 запросов/час в неделю 9.616071, то есть наша модель предсказывает, что если нынешнее поведение пользователей сохранится и компания не утратит привлекательности, то до насыщения текущей пропускной способности остается еще месяц.

Конечно, этот прогноз не вполне достоверен. Для получения более реалистичной картины можно было бы собрать дополнительные статистические данные, которые позволили бы вычислить ожидаемую дисперсию при заглядывании во все более отдаленное будущее.

Ну и кроме того, мы не можем построить точную модель динамики поведения пользователей. Однако на данный момент прогноз нас

удовлетворяет. В конце концов, он дает возможность заранее предпринять требующие времени действия. Если еще и пристально следить за трафиком, то мы сможем вовремя понять, когда заказывать дополнительные ресурсы.

Резюме

Принимайте поздравления! Вы только что узнали две важные вещи, из которых главная – что при решении типичных задач машинного обучения основное время уходит на то, чтобы понять и улучшить данные, именно этим мы и занимались в первом нашем простеньком примере. Надеемся, что он помог вам мысленно перенести акцент с алгоритмов на данные. Еще вы поняли, как важно правильно поставить эксперимент и не смешивать обучающие и тестовые данные.

Конечно, полиномиальная аппроксимация – не самая интересная вещь в мире машинного обучения. Мы выбрали ее просто для того, чтобы довести до вас две важные мысли, не отвлекаясь на красоту какого-нибудь изысканного алгоритма.

А в следующей главе мы с головой погрузимся в scikit-learn, замечательный пакет программ машинного обучения, опишем различные типы обучения и продемонстрируем, каким интересным делом может быть подготовка признаков.

ГЛАВА 2. Классификация в реальной жизни

Тема этой главы – **классификация**. Вероятно, вы уже встречались с этим видом машинного обучения как пользователь, даже не подозревая об этом. Любая современная система электронной почты умеет автоматически распознавать спам. Это означает, что система анализирует все входящие сообщения и помечает их как спам или неспам. Часто конечному пользователю предоставляется возможность вручную помечать сообщения и тем самым улучшать способность системы к распознаванию спама. В этом варианте машинного обучения системе предъявляются примеры сообщений двух типов: спам и неспам, и на этих примерах она учится автоматически классифицировать входящие сообщения.

Общая методика классификации заключается в том, чтобы с помощью набора примеров из каждого класса отыскать правила, которые могут быть применены к новым примерам. Этот – один из самых важных режимов машинного обучения – и является темой данной главы.

Для работы с текстами, в частности почтовыми сообщениями, требуются особые приемы и навыки, которые мы обсудим в следующей главе. А пока будем иметь дело с небольшим набором данных, удобным для обработки. В этой главе мы поставим такой вопрос: «Может ли машина различить виды цветов, опираясь на их изображения?». Мы будем использовать два набора данных, в которых сведения о морфологии цветов сопровождаются образцами нескольких видов.

Для исследования этих небольших наборов данных мы применим несколько простых алгоритмов. Сначала напишем код классификации самостоятельно, чтобы понять основные идеи, а затем перейдем к использованию scikit-learn. Наша цель – уяснить базовые принципы классификации, после чего познакомиться с современными реализациями.

Набор данных Iris

Iris – классический набор данных, созданный еще в 1930-х годах; это один из первых современных примеров статистической классификации.

Он представляет собой собрание морфологических измерений нескольких сортов ирисов. Эти измерения позволяют различить разные сорта цветов. В наши дни образцы идентифицируются генетическими отпечатками ДНК, но в 1930-е годы роль ДНК в генетике еще не была открыта.

Для каждого растения измерялись четыре характеристики:

- длина чашелистика;
- ширина чашелистика;
- длина лепестка;
- ширина лепестка.

В общем случае числовые характеристики, используемые для описания данных, называются **признаками**. Признаки можно измерять непосредственно или вычислять на основе промежуточных данных.

В этом числовом наборе четыре признака. Дополнительно для каждого растения записывался его вид. Задача ставится так: «Имея такие примеры, сможем ли мы уверенно определить вид нового цветка по измерениям его признаков?».

Это так называемая проблема **обучения с учителем**, или классификации: располагая помеченными примерами, вывести правило, которое можно было бы применить к другим примерам. Современным читателям, далеким от ботаники, более знаком другой пример: фильтрация спама, когда пользователь может пометить почтовые сообщения как спам, а система использует эти и другие – неспамные – сообщения, чтобы определить, является новое сообщение спамом или нет.

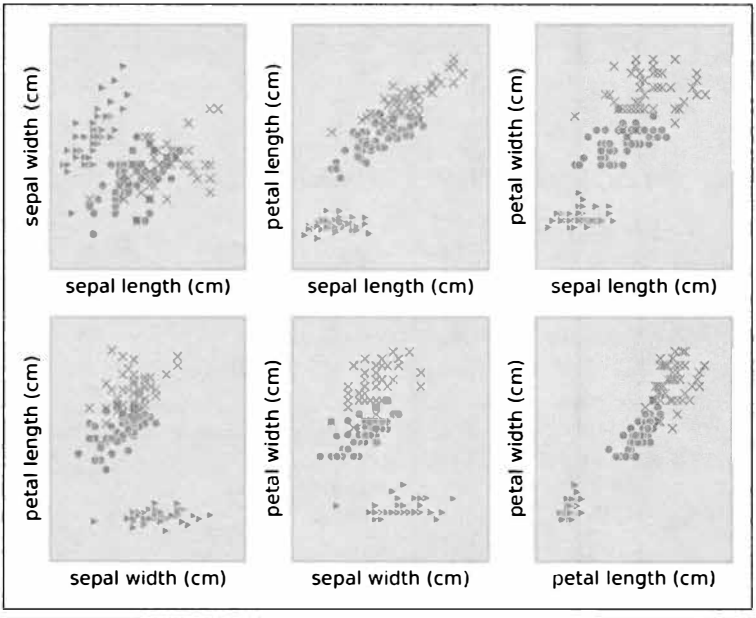
Далее в этой книге (прямо со следующей главы) мы рассмотрим задачи обработки текста. А пока набор данных Iris нас вполне устраивает. Он небольшой (150 примеров, по четыре признака в каждом), легко поддается визуализации и манипулированию.

Визуализация – первый шаг к цели

В наборах данных, которые встретятся нам далее, количество признаков измеряется тысячами. А все двумерные проекции набора с

четырьмя признаками легко умещаются на одной странице. На этом небольшом примере мы разовьем интуицию, которую потом можно будет распространить и на такие наборы данных, где признаков куда больше. В предыдущей главе мы видели, что визуализация – отличный инструмент начального исследовательского этапа анализа, поскольку дает возможность уяснить общий характер задачи и на ранних стадиях выявить возможные проблемы со сбором данных.

На каждом подграфике ниже показаны все точки в проекции на двумерную плоскость. Стоящая особняком группа (треугольники) – растения вида Iris Setosa, а в центре находятся Iris Versicolor plants (кружочки) и Iris Virginica (крестики). Как видим, выделяются две крупные группы: Iris Setosa и Iris Versicolor совместно с Iris Virginica.



Ниже показан код загрузки данных и построения графика:

```
>>> from matplotlib import pyplot as plt
>>> import numpy as np

>>> # Загружаем данные функцией load_iris из sklearn
>>> from sklearn.datasets import load_iris
>>> data = load_iris()

>>> # load_iris возвращает объект с несколькими полями
```

```
>>> features = data.data
>>> feature_names = data.feature_names
>>> target = data.target
>>> target_names = data.target_names

>>> for t in range(3):
...   if t == 0:
...      c = 'r'
...      marker = '>'
...   elif t == 1:
...      c = 'g'
...      marker = 'o'
...   elif t == 2:
...      c = 'b'
...      marker = 'x'
...   plt.scatter(features[target == t,0],
...               features[target == t,1],
...               marker=marker,
...               c=c)
```

Построение первой модели классификации

Если цель – различить три вида цветов, то можно сразу же предложить несколько подходов, стоит только взглянуть на данные. Например, отличить Iris Setosa от двух других видов можно по длине лепестка. Следующий код определяет, где проходит граница:

```
>>> # Используем прихотливое индексирование NumPy, чтобы получить
    # массив строк
>>> labels = target_names[target]
>>> # Длина лепестка - признак в позиции 2
>>> plength = features[:, 2]
>>> # Строим массив булевых значений
>>> is_setosa = (labels == 'setosa')
>>> # Это важный шаг
>>> max_setosa = plength[is_setosa].max()
>>> min_non_setosa = plength[~is_setosa].min()
>>> print('Maximum of setosa: {0}.'.format(max_setosa))
Maximum of setosa: 1.9.

>>> print('Minimum of others: {0}.'.format(min_non_setosa))
Minimum of others: 3.0.
```

Итак, мы можем построить простую модель: если длина лепестка меньше 2, то это Iris Setosa, иначе либо Iris Virginica, либо Iris Versicolor. Наша первая модель отлично справляется с отделением

Iris Setosa от двух других видов, не делая ни одной ошибки. В данном случае мы вообще не прибегали к машинному обучению, а просто изучили данные самостоятельно и нашли, как разделить классы. Машинное обучение начинается, когда мы пишем код, способный произвести такое разделение автоматически.

Задача распознавания Iris Setosa оказалась очень легкой. Но вот как выбрать наилучший порог, позволяющий отличить Iris Virginica от Iris Versicolor, сразу не ясно. Более того, мы видим, что идеальное разделение по этим признакам вообще невозможно. Но можно попытаться найти лучшее из возможного: такое разделение, при котором количество ошибок минимально. Для этого надо будет кое-что посчитать.

Сначала выберем только признаки и метки, не относящиеся к Setosa:

```
>>> # ~ - оператор логического отрицания
>>> features = features[~is_setosa]
>>> labels = labels[~is_setosa]
>>> # Создаем новую целевую переменную is_virginica
>>> is_virginica = (labels == 'virginica')
```

Здесь мы вовсю используем операции над массивами NumPy. Массив `is_setosa` – это массив булевых величин, с его помощью мы выбираем подмножества двух других массивов, `features` и `labels`. Наконец, мы создаем новый булев массив, сравнивая метки на равенство.

Далее в цикле перебираем все возможные признаки и пороги, чтобы узнать, при каком достигается наилучшая верность. Верность – это просто доля правильно классифицированных примеров.

```
>>> # Инициализируем best_acc значением, меньшим минимально возможного
>>> best_acc = -1.0
>>> for fi in range(features.shape[1]):
...   # Мы будем проверять все возможные пороговые значения
...   thresh = features[:,fi]
...   for t in thresh:
...     # Получить вектор для признака 'fi'
...     feature_i = features[:, fi]
...     # применить порог 't'
...     pred = (feature_i > t)
...     acc = (pred == is_virginica).mean()
...     rev_acc = (pred == ~is_virginica).mean()
...     if rev_acc > acc:
...        reverse = True
...        acc = rev_acc
...     else:
```

```
            reverse = False

        if acc > best_acc:
            best_acc = acc
            best_fi = fi
            best_t = t
            best_reverse = reverse
```

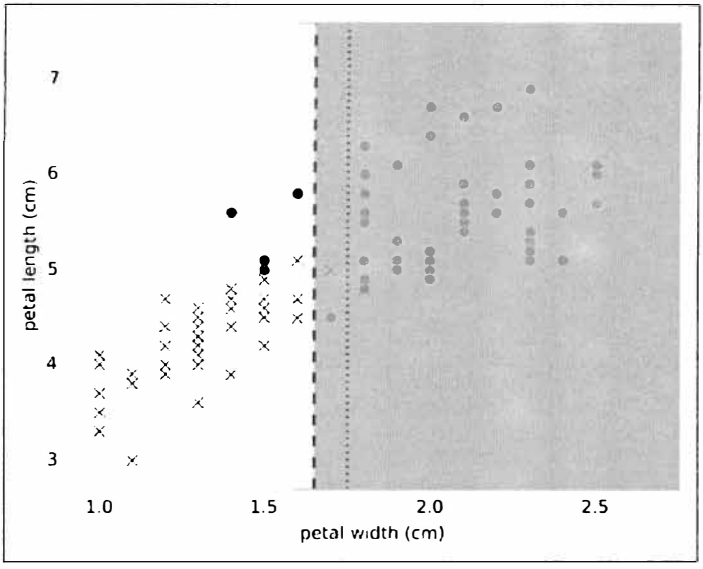
Необходимо проверить два порога для каждого признака и значения: мы проверяем условие «больше порога» и противоположное ему. Именно поэтому в показанном выше коде нужна переменная `rev_acc`, в ней хранится верность обращенного сравнения.

В последних строках выбирается наилучшая модель. Сначала мы сравниваем предсказанные метки `pred` с фактическими, `is_virginica`. Вычисление среднего значения результатов сравнения дает долю правильных результатов, то есть верность. В конце цикла все возможные пороги для всех признаков проверены, и переменные `best_fi`, `best_t` и `best_reverse` описывают модель. Этой информации достаточно для классификации неизвестного ранее объекта, то есть отнесения его к определенному классу. Что и делается в следующем методе:

```
def is_virginica_test(fi, t, reverse, example):
  "Apply threshold model to a new example"
  test = example[fi] > t
  if reverse:
     test = not test
  return test
```

Как устроена эта модель? Если прогнать этот код для всего набора данных, то модель, признанная лучшей, принимает решения путем разделения по ширине лепестка. Чтобы интуитивно представить, как это работает, полезно построить **решающую границу**. То есть мы видим, при каких значениях признака принимается одно решение, а при каких – другое, и где проходит граница. На рисунке ниже показаны две области: белая и серая. Точки, попадающие в белую область, классифицируются как Iris Virginica, а попадающие в серую область – как Iris Versicolor.

В пороговой модели решающая граница всегда будет прямой, параллельной одной из осей. На графике выше показана решающая граница и области по обе стороны от нее. Здесь же показан альтернативный порог (пунктирная линия), при котором достигается точно такая же верность. Наш метод выбрал первый порог, но это было абсолютно произвольное решение.



Оценка качества – резервирование данных и перекрестная проверка

В предыдущем разделе была рассмотрена простая модель, она позволяет добиться верности 94 % на всем наборе данных. Однако такая оценка может оказаться излишне оптимистичной. Мы сначала воспользовались данными для определения порогового значения, а затем теми же данными для оценки качества модели. Разумеется, такая модель будет вести себя лучше, чем любая другая. Мы попали в порочный круг.

А в действительности нам нужно оценить способность модели к обобщению на новые примеры. Мы должны измерить ее качество на примерах, которые не предъявлялись алгоритму во время обучения. Следовательно, нужно более строгое оценивание и использование зарезервированных данных. Для этого разобьем данные на две группы: на одной будем обучать модель, а на другой тестировать. Полный код, основанный на приведенном выше, имеется в сетевом репозитории. Он выводит такие результаты:

```
Training accuracy was 96.0%.
Testing accuracy was 90.0% (N = 50).
```

Результат, показанный на обучающих данных (подмножестве всего набора данных) оказался даже лучше, чем раньше. Но важно отметить, что результат на тестовых данных хуже. Начинающему это может показаться странным, но вообще нет ничего неожиданного в том, что верность на тестовых данных ниже, чем на обучающих. Чтобы понять, почему это так, взгляните на график, где показана решающая граница. Подумайте, что произошло бы, если бы примеров, расположенных близко к границе, там не было или если бы какой-то пример, оказавшийся между двумя линиями, отсутствовал бы. Легко понять, что граница сдвинулась бы немного вправо или влево, так что они оказались бы *не по ту* сторону от нее.

Верность на обучающих данных почти всегда дает чрезмерно оптимистичную оценку качества алгоритма. Измерять следует **верность на тестовых данных**, то есть на примерах, которые не предъявлялись на этапе обучения.

Важность этих понятий возрастает с увеличением сложности модели. В данном примере различие между верностью на обучающих и на тестовых данных невелико. Но если модель сложна, то вполне возможно на обучающих данных получить стопроцентную верность, а на тестовых – не большую, чем при случайном угадывании!

Выше, резервируя данные, мы решили использовать для обучения только половину имеющихся данных. Быть может, было бы лучше увеличить эту долю. С другой стороны, если оставить для тестирования слишком мало данных, то может не хватить примеров для оценивания погрешности. В идеале было бы хорошо использовать все данные как для обучения, так и для тестирования, но это невозможно.

Приемлемое приближение к недостижимому идеалу дает метод **перекрестной проверки**. Одна из его форм называется *перекрестной проверкой с исключением по одному*. Мы выбираем какой-нибудь пример из имеющихся данных, обучаем модель на всех данных, кроме этого примера, а затем проверяем, правильно ли модель классифицирует этот пример. Этот процесс повторяется для всех элементов набора данных.

В показанном ниже коде реализована эта идея:

```
>>> correct = 0.0
>>> for ei in range(len(features)):
    # оставить все элементы, кроме находящегося в позиции 'ei':
    training = np.ones(len(features), bool)
    training[ei] = False
```

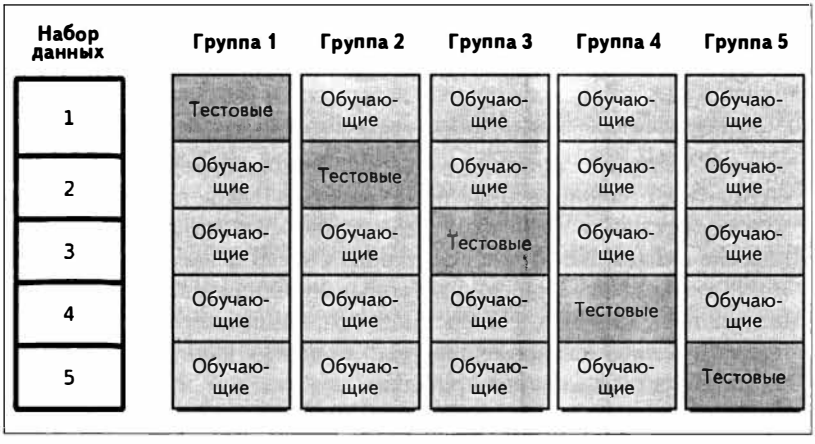
```
        testing = ~training
        model = fit_model(features[training], is_virginica[training])
        predictions = predict(model, features[testing])
        correct += np.sum(predictions == is_virginica[testing])
>>> acc = correct/float(len(features))
>>> print('Верность: {0:.1%}'.format(acc))
Верность: 87.0%
```

В этом цикле мы тестируем последовательность моделей на всех примерах, а по его завершении печатаем усредненный результат. При использовании перекрестной проверки порочный круг не возникает, потому что каждая модель тестируется на примере, который она не видела при обучении. Поэтому полученную таким образом оценку можно считать надежным показателем обобщаемости модели на новые данные.

Основная проблема перекрестной проверки с исключением по одному состоит в том, что нам приходится проделывать гораздо больше работы. По существу, мы должны обучать новую модель для каждого примера, и с ростом набора данных затраты оказываются очень велики.

Получить преимущества исключения по одному с гораздо меньшими издержками позволяет x-проходная перекрестная проверка, где x – небольшое целое число. Например, в случае пятипроходной проверки мы разбиваем данные на 5 групп.

Затем мы обучаем пять моделей, каждый раз исключая из обучающего набора одну группу. Получающийся код похож на приведенный выше, но резервируем мы сразу 20 процентов данных, а не один элемент. Все обученные модели тестируются на данных, входящих в исключенную группу, и результаты усредняются.



На этом рисунке показана пятипроходная проверка: набор данных разбит на пять групп. На каждом проходе мы резервируем одну группу для тестирования и обучаем модель на остальных четырех. Количество проходов может быть произвольным. Чем больше проходов, тем выше верность (поскольку мы приближаемся к идеалу – обучению на всех имеющихся данных), но и тем больше вычислительные затраты. Пять проходов – разумный компромисс. В этом случае мы используем для обучения 80 процентов данных, результат должен получиться довольно близким к тому, что получился бы при обучении на всех данных. Если данных мало, то можно остановиться на 10 или 20 проходах. В предельном случае, когда проходов столько же, сколько элементов данных, мы получаем просто перекрестную проверку с исключением по одному. С другой стороны, если время вычислений оказывается недопустимо большим, а данных достаточно, можно ограничиться 2 или 3 проходами.

При отборе групп нужно следить за их сбалансированностью. Так, если в какой-то группе все примеры принадлежат одному классу, то результат окажется непрезентативным. Мы не будем вдаваться в детали этого процесса, потому что пакет программ машинного обучения scikit-learn все это умеет.

Итак, мы сгенерировали не одну, а несколько моделей. Возникает вопрос – какую из них использовать для классификации новых данных? Простейшее решение – обучить одну модель на всех обучающих данных. Проведенная ранее перекрестная проверка уже показала, насколько хорошо эта модель будет обобщаться.

Перекрестная проверка позволяет использовать имеющиеся данные для оценки качества работы выбранного алгоритма. По завершении цикла можно обучить окончательную модель на всех данных.

Хотя не вполне понятно, когда машинное обучение впервые выделилось в отдельную дисциплину, сегодня считается очень дурным тоном обсуждать даже саму возможность тестирования верности классификации на обучающих данных. Дело в том, что результаты могут оказаться совершенно неверными и одно лишь их упоминание выдает в вас новичка. Измерение и сравнение погрешности следует проводить либо на зарезервированном наборе данных, либо с помощью метода перекрестной проверки.

Построение более сложных классификаторов

В предыдущем разделе мы использовали очень простую модель: порог по одному признаку. Существуют ли другие системы? А как же! И в изобилии. В этой книге мы увидим много разных моделей, но при этом даже не пытаемся охватить все разнообразие.

Зададимся более абстрактным вопросом: «Как вообще можно охарактеризовать модель классификации?». Можно выделить три части.

- **Структура модели.** Как именно модель принимает решения? В данном случае решение зависело только от того, будет значение конкретного признака больше или меньше порогового. Это годится только для самых простых задач.
- **Процедура поиска.** Как мы находим подходящую модель? В нашем случае мы опробовали все комбинации признака и порогового значения. Нетрудно сообразить, что с увеличением сложности модели и размера набора данных полный перебор всех комбинаций становится невозможным и приходится использовать приближенные решения. Поэтому для поиска хорошего решения часто используются изощренные методы оптимизации (к счастью, в scikit-learn они уже реализованы, так что пользоваться ими просто, даже если внутренний код очень сложен).
- **Функция выигрыша или функция потерь.** Как мы решаем, какую из рассмотренных моделей возвращать? Редко удается найти идеальную модель, которая вообще не делает ошибок, поэтому нужно решить, какую использовать. Мы брали в качестве критерия верность, но иногда полезнее выбирать модель, которая делает наименьшее количество ошибок определенного вида. Например, при фильтрации спама удалить хорошее сообщение хуже, чем по ошибке оставить плохое. В таком случае нужно выбирать модель, которая с осторожностью отбрасывает почтовые сообщения, а не такую, которая просто минимизирует общее число ошибок. Эти проблемы можно обсуждать в терминах выигрыша (который мы хотим максимизировать) или потерь (которые мы хотим вести к минимуму). Оба взгляда эквивалентны, но иногда удобнее один, а иногда другой.

Варьируя эти три аспекта, мы получаем разные системы классификации. Пороговые модели – одни из самых простых в машинном

обучении, они годятся только для совсем тривиальных задач типа классификации цветов в наборе данных Iris. В следующем разделе мы рассмотрим более трудную задачу классификации, для которой понадобится структура посложнее.

В нашем примере порог выбирался, так чтобы минимизировать число ошибок. Но можно было бы взять и другую функцию потерь. Не исключено, что одни ошибки обходятся гораздо дороже других. В медицинских приложениях ложноположительные и ложноотрицательные результаты не эквивалентны. **Ложноотрицательный** результат (когда модель возвращает отрицательное заключение, хотя это неверно) может означать, что пациент не получит лечение от серьезной болезни. **Ложноположительный** результат (когда модель возвращает положительное заключение, хотя у пациента нет такой болезни) может означать, что нужны дополнительные анализы или что пациента будут лечить без необходимости, чтобы подтвердить или опровергнуть диагноз (тут тоже есть свои неприятности, в том числе побочные эффекты от лечения, но они все же менее серьезны, чем непоставленный диагноз). Таким образом, подход зависит от конкретной ситуации. В одном предельном случае, когда болезнь смертельна, а лечение дешево и почти не имеет побочных эффектов, следует стремиться к минимизации ложноотрицательных результатов.

Выбор функции **потерь** (или **выигрыша**) всегда зависит от конкретной задачи. Представляя универсальный алгоритм, мы часто стремимся к минимизации числа ошибок, то есть к достижению максимальной верности. Но если одни ошибки обходятся дороже других, то, быть может, стоит смириться с уменьшением общей верности в пользу минимизации общих издержек.

Более сложный набор данных и более сложный классификатор

Рассмотрим теперь несколько более сложный набор данных. На его примере мы познакомимся еще с одним алгоритмом классификации и кое-какими новыми идеями.

Набор данных Seeds

Возьмем еще один набор данных из области ботаники, тоже небольшой, но уже не такой маленький, чтобы его можно было целиком

изобразить на печатной страницы, как в случае набора Iris. В этот набор входят данные об измерениях семян пшеницы. Всего есть семь признаков:

- площадь A;
- периметр P;
- компактность $C = 4\pi A/P^2$;
- длина зерна;
- ширина зерна;
- коэффициент асимметрии;
- длина желобка зерна.

Существует три класса, соответствующие трем сортам пшеницы: Canadian, Koma и Rosa. Как и раньше, наша цель – научиться классифицировать образцы по этим морфологическим измерениям. В отличие от набора данных Iris, который был создан в 1930-х годах, это совсем недавний набор, для которого признаки автоматически вычислялись путем обработки цифровых изображений.

Вот как можно реализовать автоматическое распознавание образов: берем изображения, представленные в цифровом виде, вычисляем по ним интересующие нас признаки и применяем общую систему классификации. В главе 10 мы рассмотрим ту часть этой задачи, которая относится к машинному зрению и вычислению признаков по изображению. А пока будем работать с признаками, которые уже имеются.

Репозиторий наборов данных для машинного обучения UCI

В Калифорнийском университете в Ирвайне (UCI) ведется сетевой репозиторий наборов данных для машинного обучения (на момент написания этой книги там было 223 набора). Наборы Iris и Seeds взяты именно оттуда. Адрес репозитория `http://archive.ics.uci.edu/ml/`.

Признаки и подготовка признаков

У этих признаков есть интересная особенность: компактность – не результат независимого измерения, а функция двух предыдущих признаков, площади и периметра. Такое выведение новых признаков из существующих часто бывает очень полезно. Деятельность по созданию новых признаков называют **подготовкой признаков** (feature engineering). Она не такая эффектная, как применение алгоритмов, но нередко оказывает на качество работы даже большее влияние

(простой алгоритм на хорошо подобранных признаках работает лучше, чем изощренный алгоритм на плохих признаках).

В данном случае авторы набора вычисляли **компактность** – типичный признак геометрических фигур. Иногда его еще называют **округленностью**. Этот признак будет одинаков для двух зерен одинаковой формы, из которых одно вдвое больше другого. Но для круглых и продолговатых зерен от различен (в первом случае компактность близка к единице, во втором – к нулю).

Хороший признак должен изменяться при изменении того, что существенно (желательный результат), и быть инвариантным относительно того, что несущественно. Например, компактность не зависит от размера, но изменяется при изменении формы. На практике достичь обеих целей одновременно трудно, но стремиться к этому надо.

Для проектирования хороших признаков нужны знания в предметной области. К счастью, для многих предметных областей уже имеется обширная литература по признакам и их типам, так что есть, что взять за основу. Если говорить об изображениях, то все вышеупомянутые признаки типичны, и библиотеки машинного зрения вычисляют их автоматически. Для задач обработки текста также имеются стандартные решения, которые можно комбинировать (мы познакомимся с ними в следующей главе). Когда проектируете признаки (или берете те, что упоминаются в литературе), старайтесь применять свои знания о задаче; ваша цель – выбрать признаки, максимально соответствующие данным.

Еще до того как вы приступите к сбору данных, решите, какие данные стоит собирать. Затем подайте признаки машине, и пусть она найдет наилучший классификатор.

Возникает естественный вопрос – нельзя ли отбирать признаки автоматически. Эта задача называется **отбором признаков**. Предложено много методов ее решения, но на практике лучше всего работают самые простые идеи. Для небольших задач, которыми мы сейчас занимаемся, применять отбор признаков вообще не имеет смысла, но если признаков тысячи, то, отбросив большинство из них, мы существенно ускорим последующие этапы процесса.

Классификация по ближайшему соседу

Для работы с этим набором данных мы рассмотрим новый классификатор – **по ближайшему соседу**. Это очень простой классификатор. Когда требуется классифицировать новый образец, он просматривает обучающие данные и находит ближайший к нему объект.

В качестве ответа возвращается метка этого объекта. Отметим, что на самих обучающих данных эта модель работает идеально! Для любой точки ближайшей является она сама, поэтому метка всегда будет правильной (если только не существует двух примеров с одинаковыми наборами признаков и разными метками, но это означало бы, что описательная способность признаков оставляет желать лучшего). Поэтому для тестирования этого алгоритма классификации применять перекрестную проверку просто необходимо.

Метод ближайшего соседа можно обобщить – искать не одного, а несколько ближайших соседей и проводить между ними голосование. Это повышает устойчивость к выбросам и неправильно помеченным примерам.

Классификация с помощью scikit-learn

Выше мы писали код классификации вручную, но Python – очень подходящий язык для машинного обучения, потому что располагает отличными библиотеками. В частности, scikit-learn стала стандартной библиотекой для многих задач машинного обучения, включая классификацию. В этом разделе мы воспользуемся имеющейся в ней реализацией классификации по ближайшим соседям.

API классификации в scikit-learn построен на основе объектов-классификаторов. У такого объекта есть два основных метода:

- `fit(features, labels)`: это этап обучения и подгонки параметров модели;
- `predict(features)`: вызывается только после `fit` и возвращает предсказание для одного или нескольких входных образцов.

Вот как можно было бы применить эту реализацию метода *k* ближайших соседей к нашим данным. Начнем с импорта класса `KneighborsClassifier` из подмодуля `sklearn.neighbors`:

```
>>> from sklearn.neighbors import KNeighborsClassifier
```

Модуль scikit-learn импортируется как sklearn (иногда для ссылки на scikit-learn употребляется именно это сокращение). Вся функциональность sklearn распределена по подмодулям – таким, как `sklearn.neighbors`.

Теперь можно создать объект классификатора. Конструктору мы передаем количество соседей:

```
>>> classifier = KNeighborsClassifier(n_neighbors=1)
```

Если количество соседей не задано, то по умолчанию подразумевается 5, что обычно является неплохим выбором.

Мы хотим воспользоваться перекрестной проверкой (а как же иначе?). Модуль scikit-learn легко позволяет это сделать:

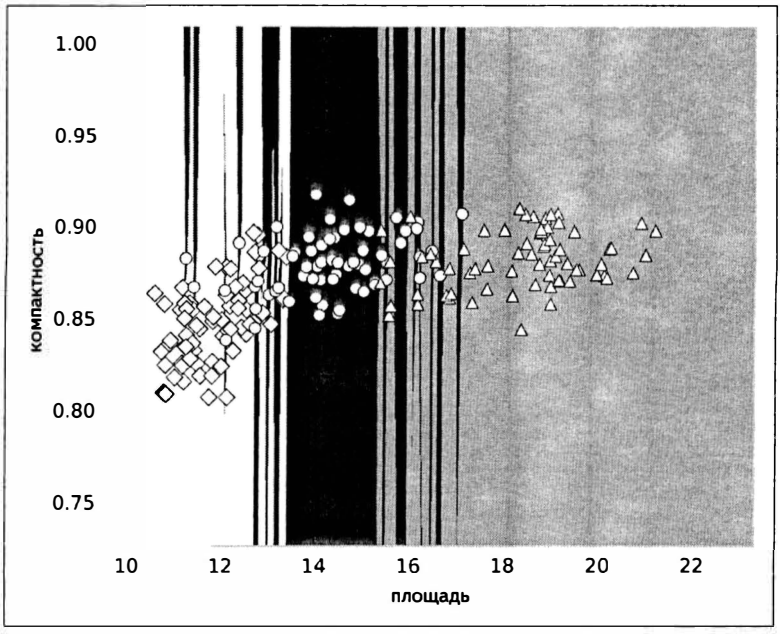
```
>>> from sklearn.cross_validation import KFold
>>> kf = KFold(len(features), n_folds=5, shuffle=True)
>>> # 'means' - список усредненных верностей (по одной для каждой группы)
>>> means = []
>>> for training,testing in kf:
...  # Обучаем модель на этой группе, затем применяем ее к
...  # тестовым данным с помощью метода 'predict':
...  classifier.fit(features[training], labels[training])
...  prediction = classifier.predict(features[testing])
...
...  # метод np.mean, примененный к массиву булевых величин,
...  # возвращает долю правильных решений для этой группы
...  curmean = np.mean(prediction == labels[testing])
...  means.append(curmean)
>>> print("Средняя верность: {:.1%}".format(np.mean(means)))
Средняя верность: 90.5%
```

Использование пятипроходной перекрестной проверки для этого набора данных и этого алгоритма дает верность 90.5 %. Выше мы уже говорили, что верность перекрестной проверки меньше, чем верность на обучающих данных, но при этом заслуживает гораздо больше доверия в качестве показателя качества модели.

Решающие границы

Рассмотрим теперь решающие границы. Чтобы нарисовать их на бумаге, мы упростим задачу, оставив только два измерения. Взгляните на следующую диаграмму:

Образцы сорта Canadian представлены ромбами, сорта Koma – кружочками, а сорта Rosa – треугольниками. Соответственные им области окрашены белым, черным и серым цветом. Возникает вопрос, почему области до странности вертикальны. Дело в том, что по оси x (площадь) отложен диапазона от 10 до 22, а по оси y (компактность) – от 0,75 до 1,0. Это означает, что малое изменение x на графике оказывается гораздо значимее малого изменения y. Поэтому в расстояние между точками наибольший вклад вносит x. Вот вам неплохая иллюстрация того, почему всегда имеет смысл визуализировать данные и смотреть, не будет ли какого-нибудь сюрприза.



Если вы изучали физику (и еще не все забыли), то, наверное, обратили внимание, что мы суммируем длины, площади и безразмерные величины, не обращая внимания на единицы измерения (чего в реальной физической системе ни в коем случае нельзя делать). Необходимо привести все признаки к единой шкале. Решать эту проблему можно по-разному; проще всего воспользоваться *нормированными z-оценками*. Z-оценка некоторого значения показывает, насколько далеко оно отстоит от среднего в единицах, равных стандартному отклонению. Вычисляется она следующим образом:

$$f' = \frac{f - \mu}{\sigma}$$

Здесь f – исходное значение признака, f' – нормированное значение, μ – среднее значение признака, а σ – стандартное отклонение. Величины μ и σ вычисляются по обучающим данным. Каковы бы ни были исходные данные, после нормировки нулевое значение z-оценки соответствует среднему значению обучающих данных, положительные значения – значениям больше среднего, а отрицательные – значениям меньше среднего.

С помощью модуля scikit-learn ничего не стоит произвести такую нормировку на этапе предобработки. Мы применим конвейер операций: первый его элемент выполняет преобразование, второй – классификацию. Сначала импортируем классы конвейера и шкалирования признаков:

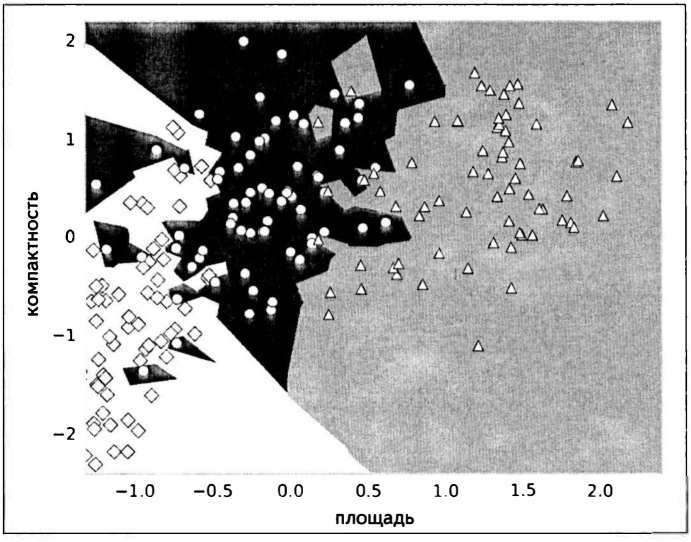
```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.preprocessing import StandardScaler
```

Теперь можно построить конвейер:

```
>>> classifier = KNeighborsClassifier(n_neighbors=1)
>>> classifier = Pipeline([('norm', StandardScaler()),
...        ('knn', classifier)])
```

Конструктор класса Pipeline принимает список пар `(str, clf)`. Каждой паре соответствует один этап конвейера; первый элемент – строка, именующая этап, второй – объект, выполняющий операцию. В более сложных приложениях по именам можно ссылаться на этапы конвейера.

После нормировки все признаки измеряются в одних и тех же единицах (строго говоря, они теперь безразмерны, то есть говорить о единицах измерения бессмысленно), и над ними можно беспрепятственно производить математические операции. Если теперь прогнать классификатор по ближайшим соседям, то получится верность 93% при той же самой пятипроходной перекрестной проверке!



Теперь границы выглядят совсем по-другому, и мы видим, что на результат влияют оба измерения. Полному набору данных соответствует семимерное пространство, изобразить которое затруднительно, но принцип тот же: если в исходных данных некоторые измерения доминируют, то после нормировки все становятся равноправны.

Бинарная и многоклассовая классификация

Наш первый пороговый классификатор был простым бинарным классификатором. На выходе он дает один из двух классов, поскольку любое значение либо больше порога, либо меньше. Второй наш классификатор – по ближайшим соседям – естественно оказался многоклассовым, его результатом может быть один из нескольких классов.

Часто бывает легче определить простой бинарный метод, чем метод, применяемый к многоклассовым задачам. Однако любую многоклассовую задачу можно свести к последовательности бинарных решений. Именно так мы поступили с набором данных Iris, хотя это получилось случайно: мы заметили, что один класс легко отделяется, и сосредоточили внимание на двух других, сведя задачу к двум бинарным решениям:

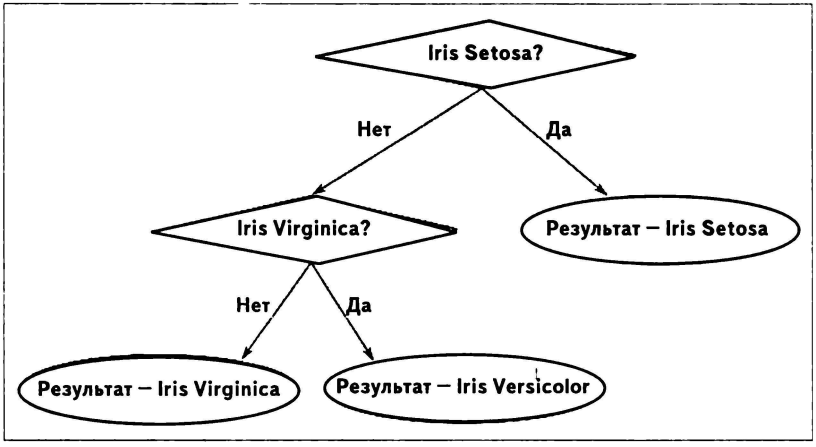
1. Это Iris Setosa (да или нет)?
2. Если нет, проверить, что это Iris Virginica (да или нет).

Разумеется, мы хотели бы поручить такого рода рассуждения компьютеру. Как обычно, есть несколько подходов к решению задачи о многоклассовой редукции.

Проще всего воспользоваться последовательностью классификаторов *один против остальных*. Для каждой возможной метки ℓ строим классификатор вида *это ℓ или что-то другое*? Если в ходе применения этого правила ровно один классификатор отвечает *да*, то решение получено. К сожалению, так бывает не всегда, и приходится решать, как быть, когда есть несколько положительных ответов или нет ни одного.

Можно вместо этого построить дерево классификации. Разобьем множество всех меток на два подмножества и построим классификатор, который спрашивает: «Куда отправить этот пример – налево или направо?». Такое разбиение можно продолжать рекурсивно, пока не останется одна метка. На рисунке выше изображено дерево рассужде-

ний для набора данных Iris. Каждый ромб – это один бинарный классификатор. Понятно, что дерево могло бы быть больше и охватывать много решений. Таким образом, любой классификатор, который годится для бинарной классификации, можно легко обобщить на любое число классов.



Есть много других способов превратить бинарный алгоритм в многоклассовый. Ни один из них не является однозначно предпочтительным во всех случаях. В модуле scikit-learn реализовано несколько таких алгоритмов с подмодуле `sklearn.multiclass`.

Некоторые классификаторы являются бинарными, тогда как многие реальные задачи естественно оказываются многоклассовыми. Существует несколько простых протоколов сведения многоклассовой задачи к последовательности бинарных решений, что позволяет применить бинарные модели к многоклассовым задачам. Это означает, что методы, которые, на первый взгляд, работают только для бинарных данных, можно без особых усилий обобщить на многоклассовые данные.

Резюме

Под классификацией понимается обобщение примеров для построения модели (то есть набор правил, которые могут быть применены к новым, ранее не классифицированным объектам). Это один из важнейших инструментов машинного обучения, и мы еще неоднократно с ним встретимся на страницах этой книги.

Эта глава была в каком-то смысле теоретической, поскольку мы ввели общие понятия на простых примерах. Мы проделали несколько операций с набором данных Iris. Это небольшой набор, но он обладает тем преимуществом, что его можно изобразить и во всех деталях понять, что мы делаем. Это преимущество теряется при переходе к многомерным задачам с тысячами примеров. Однако приобретенная интуиция останется с нами.

Мы также узнали, что погрешность на обучающих данных – вещь обманчивая, поскольку дает излишне оптимистическую оценку поведения модели. Модель следует оценивать на тестовых данных, которые не использовались для обучения. Чтобы не расходовать слишком много примеров на тестирование, применяется метод перекрестной проверки, благодаря которому мы получаем лучшее из обоих миров (ценой увеличения объема вычислений).

Мы также рассмотрели проблему подготовки признаков. Признаки вовсе не являются предопределенными, их выбор и проектирование – неотъемлемая часть конструирования конвейера машинного обучения. На самом деле, именно в этой области часто можно добиться наибольшего увеличения верности, потому что качественные данные лучше изощренных алгоритмов. В главах, посвященных классификации текстов, распознаванию музыкальных жанров и машинному зрению, мы приведем дополнительные примеры подготовки признаков.

В следующей главе мы поговорим о том, что делать, когда в данных нет бросающихся в глаза классов.

ГЛАВА 3. Кластеризация – поиск взаимосвязанных сообщений

В предыдущей главе мы узнали, как находить классы, или категории отдельных образцов. Имея набор обучающих образцов, которым уже сопоставлены классы, мы обучили модель и затем можем воспользоваться ей для классификации новых образцов. Мы назвали этот процесс обучением с учителем, в нашем случае роль учителя сводилась к правильной классификации примеров.

Теперь допустим, что мы не располагаем метками, с помощью которых можно было бы обучить модель классификации, например, потому что разметка обошлась бы слишком дорого. Что, если единственный способ получить миллионы меток – попросить, чтобы их вручную проставил человек? Как быть в таком случае?

Конечно, обучить модель классификации мы не сможем. Но тем не менее можно попытаться найти какие-то закономерности в самих данных. То есть возложить на данные задачу самоописания. Именно этим мы и займемся в этой главе, а в качестве примера рассмотрим вопросно-ответный сайт. Когда пользователь будет искать на нашем сайте какую-то информацию, поисковая система, скорее всего, сразу покажет нужный ему ответ. Если имеющиеся ответы пользователя не устраивают, то сайт должен хотя бы показать близкие ответы, чтобы пользователь быстро понял, какие ответы существуют, и не ушел с сайта.

Наивный подход – просто взять сообщение, вычислить его схожесть со всеми остальными сообщениями и показать первые n самых похожих сообщений в виде ссылок. Но очень скоро такое решение станет слишком накладным. Нужен метод, который быстро находит все взаимосвязанные сообщения.

Для достижения этой цели мы воспользуемся кластеризацией. Это метод такой организации данных, когда похожие элементы оказываются в одном кластере, а непохожие – в разных. Первая проблема, которую нам предстоит решить, – как превратить текст в нечто такое, для чего можно вычислить сходство. Располагая способом измерения похожести, мы далее можем подумать, как с его помощью быстро построить кластер, содержащий похожие сообщения. Ну а потом останется только проверить, какие еще документы принадлежат этому кластеру. Всю эту программу реализует замечательная библиотека SciKit, в которой представлены различные алгоритмы машинного обучения, с которыми мы будем работать в последующих главах.

Измерение сходства сообщений

С точки зрения машинного обучения «голый» текст абсолютно бесполезен. Лишь преобразовав его в осмысленные числа, мы сможем подать их на вход алгоритмам машинного обучения, например кластеризации. Это относится и к более приземленным операциям с текстом, в частности, измерению сходства.

Как не надо делать

Одной из мер сходства является расстояние Левенштейна, или редакционное расстояние. Пусть есть два слова: «machine» и «mchiene». Их сходство можно определить как минимальное число операций редактирования, необходимых для перехода от одного слова к другому. В данном случае нужно всего две операции: добавить «a» после «m» и удалить первое «e». Однако это весьма дорогой алгоритм, потому что время его работы определяется произведением длин обоих слов.

Возвращаясь к сообщениям, мы могли бы схитрить: рассматривать слова целиком как символы и выполнять операции редактирования на уровне слов. Пусть есть два сообщения (для простоты ограничимся только заголовками): «How to format my hard disk» («Как мне отформатировать жесткий диск») и «Hard disk format problems» («Проблемы с форматированием жесткого диска»). Редакционное расстояние между ними равно 5, потому что нужно удалить слова «how», «to», «format», «my», а затем добавить в конец слова «format» и «problems». Следовательно, можно было бы определить различие между двумя сообщениями, как количество слов, которые следует добавить или удалить, чтобы преобразовать один текст в другой. Эту идею мож-

но было бы немного усовершенствовать, но по существу временная сложность остается той же самой.

Но даже если бы мы могли добиться достаточного быстродействия, существует еще одна проблема. В нашем примере слово «format» дает вклад 2 в расстояние, потому что мы сначала удалили его, а потом снова добавили. Следовательно, такое расстояние неустойчиво относительно изменения порядка слов.

Как надо делать

Более надежное редакционное расстояние дает так называемый **набор слов**. При таком подходе порядок слов полностью игнорируется, а в основу кладутся просто счетчики вхождений слов. Каждому встречающемуся в сообщении слову сопоставляется количество его вхождений, и эти пары сохраняются в векторе. Неудивительно, что эта операция называется векторизацией. Обычно вектор получается очень большим, потому что содержит столько элементов, сколько есть слов во всем наборе данных. Возьмем, к примеру, два сообщения с такими счетчиками слов:

Слово	Вхождений в сообщение 1	Вхождений в сообщение 2
disk	1	1
format	1	1
how	1	0
hard	1	1
my	1	0
problems	0	1
to	1	0

Столбцы «Вхождений в сообщение 1» и «Вхождений в сообщение 2» можно рассматривать как простые векторы. Можно вычислить евклидово расстояние между вектором вопроса и векторами всех сообщений и взять ближайшее сообщение (правда, как мы уже выяснили, это слишком медленно). А, кроме того, мы можем использовать их как векторы признаков на этапе кластеризации, применяя следующую процедуру:

1. Выделить характерные признаки из каждого сообщения и сохранить их в виде ассоциированного с сообщением вектора.

2. Произвести кластеризацию этих векторов.
3. Определить кластер, в который входит сообщение-вопрос.
4. Выбрать из этого кластера сколько-то сообщений, имеющих различное сходство с сообщением-вопросом. Это повышает разнообразие.

Но для реализации этой программы нужно проделать еще кое-какую работу. А для этого нам понадобятся данные.

Предварительная обработка – количество общих слов как мера сходства

Как мы уже видели, подход на основе набора слов является и быстрым, и надежным. Однако и он не без минусов. Давайте разберемся.

Преобразование простого текста в набор слов

Нам нет нужды писать свой код подсчета слов и представления набора слов в виде вектора. Метод `CountVectorizer` из библиотеки SciKit не только умеет делать это эффективно, но и обладает очень удобным интерфейсом. Функции и классы из SciKit импортируются посредством пакета `sklearn`:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(min_df=1)
```

Параметр `min_df` (минимальная частота в документе) определяет, как `CountVectorizer` должен обходиться с редко встречающимися словами. Если его значением является целое число, то слова с меньшим числом вхождений, отбрасываются. Если значение – дробное число, то отбрасываются слова, доля которых во всем документе меньше этого числа. Параметр `max_df` интерпретируется аналогично. Распечатав объект, мы увидим все остальные параметры, которым SciKit присвоила значения по умолчанию:

```
>>> print(vectorizer)
CountVectorizer(analyzer='word', binary=False, charset=None,
     charset_error=None, decode_error='strict',
     dtype=<class 'numpy.int64'>, encoding='utf-8',
```

```
        input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

Как и следовало ожидать, подсчитываются именно слова (`analyzer=word`), а что считать словом, определяется регулярным выражением `token_pattern`. Например, строка «cross-validated» будет разбита на два слова: «cross» и «validated». Пока не будем обращать внимания на прочие параметры и рассмотрим две строки из нашего примера, содержащие темы сообщений:

```
>>> content = ["How to format my hard disk", " Hard disk format
problems "]
```

Этот список строк можно подать на вход метода векторизатора `fit_transform()`, который и проделает всю работу:

```
>>> X = vectorizer.fit_transform(content)
>>> vectorizer.get_feature_names()
[u'disk', u'format', u'hard', u'how', u'my', u'problems', u'to']
```

Векторизатор распознал семь слов, для каждого которых мы можем получить счетчики:

```
>>> print(X.toarray().transpose())
[[1 1]
 [1 1]
 [1 1]
 [1 0]
 [1 0]
 [0 1]
 [1 0]]
```

Это означает, что первое предложение содержит все слова, кроме «problems», а второе – все слова, кроме «how», «my» и «to». На самом деле, это как раз те столбцы, которые присутствовали в предыдущей таблице. Из `X` мы можем выделить вектор признаков, которым воспользуемся для сравнения документов.

Сначала применим наивный подход, чтобы показать проблемы, с которыми придется столкнуться на этапе предварительной обработки. Случайно выберем какое-то сообщение и создадим для него вектор счетчиков. Затем вычислим расстояния от него до всех векторов счетчиков и выберем сообщение, для которого расстояние минимально.

Подсчет слов

Для экспериментов возьмем игрушечный набор данных, содержащий такие сообщения.

Имя файла	Текст сообщения
01.txt	This is a toy post about machine learning. Actually, it contains not much interesting stuff.
02.txt	Imaging databases can get huge.
03.txt	Most imaging databases save images permanently.
04.txt	Imaging databases store images.
05.txt	Imaging databases store images. Imaging databases store images. Imaging databases store images.

В этом наборе мы хотим найти сообщение, которое больше других похоже на сообщение «imaging databases».

В предположении, что сообщения находятся в каталоге `DIR`, мы можем задействовать для этой цели `CountVectorizer`:

```
>>> posts = [open(os.path.join(DIR, f)).read() for f in
os.listdir(DIR)]
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(min_df=1)
```

Нам нужно уведомить векторизатор о полном наборе данных, чтобы он заранее знал, каких слов ожидать:

```
>>> X_train = vectorizer.fit_transform(posts)
>>> num_samples, num_features = X_train.shape
>>> print(«#samples: %d, #features: %d» % (num_samples,
num_features))
#samples: 5, #features: 25
```

Получилось 5 сообщений и 25 различных слов – всё правильно. Подсчитаны следующие выделенные из текста слова:

```
>>> print(vectorizer.get_feature_names())
[u'about', u'actually', u'capabilities', u'contains', u'data',
u'databases', u'images', u'imaging', u'interesting', u'is', u'it',
u'learning', u'machine', u'most', u'much', u'not', u'permanently',
u'post', u'provide', u'save', u'storage', u'store', u'stuff',
u'this', u'toy']
```

Теперь векторизуем новое сообщение:

```
>>> new_post = «imaging databases»
>>> new_post_vec = vectorizer.transform([new_post])
```

Отметим, что метод `transform` возвращает разреженные векторы счетчиков, то есть в векторе не хранятся счетчики для каждого слова, потому что большая их часть равна нулю (в сообщении такое слово не встречается). Вместо этого используется потребляющая меньше памяти структура данных `coo_matrix` (от слова «COOrdinate»). Так, для нашего сообщения вектор содержит всего два элемента:

```
>>> print(new_post_vec)
(0, 7) 1
(0, 5) 1
```

Воспользовавшись методом `toarray()`, мы можем восстановить весь массив `ndarray`:

```
>>> print(new_post_vec.toarray())
[[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

Весь массив нам понадобится, если мы захотим использовать вектор для вычисления сходства. Чтобы измерить сходство (при наивном подходе), мы вычисляем евклидово расстояние между векторами счетчиков нового и всех старых сообщений:

```
>>> import scipy as sp
>>> def dist_raw(v1, v2):
...     delta = v1-v2
...     return sp.linalg.norm(delta.toarray())
```

Метод `norm()` вычисляет евклидову норму (кратчайшее расстояние). Это самая очевидная метрика, но есть и много других определений расстояния. Почитайте статью «Distance Coefficients between Two Lists or Sets» на сайте The Python Papers Source Codes, где Морис Линь (Maurice Ling) описывает 35 разных расстояний.

Имея функцию `dist_raw`, мы теперь должны перебрать все сообщения и запомнить самое близкое:

```
>>> import sys
>>> best_doc = None
>>> best_dist = sys.maxint
>>> best_i = None
>>> for i, post in enumerate(num_samples):
...  if post == new_post:
...     continue
...  post_vec = X_train.getrow(i)
...  d = dist_raw(post_vec, new_post_vec)
...  print("=== Post %i with dist=%.2f: %s"%(i, d, post))
...  if d<best_dist:
...     best_dist = d
...     best_i = i
```

```
>>> print("Best post is %i with dist=%.2f"%(best_i, best_dist))

=== Post 0 with dist=4.00: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.

=== Post 1 with dist=1.73: Imaging databases provide storage
capabilities.

=== Post 2 with dist=2.00: Most imaging databases save images
permanently.

=== Post 3 with dist=1.41: Imaging databases store data.

=== Post 4 with dist=5.10: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 3 with dist=1.41
```

Вот мы и произвели первое измерение сходства, примите поздравления. Сообщение 0 сильнее всего отличается от нового. Что и понятно, ведь в них нет ни одного общего слова. Понятно также, что сообщение 1 очень похоже на новое, но не является лучшим, т. к. содержит на одно отсутствующее в новом сообщении слово больше, чем сообщение 3.

Но при сравнении сообщений 3 и 4 все уже не так очевидно. Сообщение 4 – это просто сообщение 3, повторенное трижды. Поэтому его сходство с новым сообщением должно быть точно таким же, как у сообщения 3.

Распечатка соответствующих векторов признаков объясняет, почему это не так:

```
>>> print(X_train.getrow(3).toarray())
[[0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]]
>>> print(X_train.getrow(4).toarray())
[[0 0 0 0 3 3 0 3 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0]]
```

Как видим, одних лишь счетчиков слов недостаточно. Необходимо нормировать векторы на единичную длину.

Нормировка векторов счетчиков слов

В функции `dist_raw` мы будем вычислять расстояние не между исходными, а между нормированными векторами:

```
>>> def dist_norm(v1, v2):
...   v1_normalized = v1/sp.linalg.norm(v1.toarray())
...   v2_normalized = v2/sp.linalg.norm(v2.toarray())
...   delta = v1_normalized - v2_normalized
...   return sp.linalg.norm(delta.toarray())
```

Тогда результаты измерения сходства изменятся следующим образом:

```
=== Post 0 with dist=1.41: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.

=== Post 1 with dist=0.86: Imaging databases provide storage
capabilities.

=== Post 2 with dist=0.92: Most imaging databases save images
permanently.

=== Post 3 with dist=0.77: Imaging databases store data.

=== Post 4 with dist=0.77: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 3 with dist=0.77
```

Так уже лучше. Теперь сходство сообщений 3 и 4 в точности одинаково. Кто-то, возможно, сочтет, что многократное повторение вызовет восторг у читателя, но с точки зрения подсчета слов в сообщениях этот результат представляется правильным.

Удаление малозначимых слов

Взглянем еще раз на сообщение 2. В нем встречаются следующие слова, отсутствующие в новом сообщении: «most» (большинство), «save» (сохранять), «images» (изображения) и «permanently» (постоянно). Но их значимость в сообщении совершенно различна. Слова типа «most», встречающиеся в самых разных контекстах, называются стоп-словами. Они несут мало информации и потому должны весить меньше слов типа «images», которые встречаются отнюдь не во всех контекстах. Лучше всего вообще удалить слова, которые употребляются настолько широко, что не помогают выявить различия между текстами.

Этот шаг весьма типичен для обработки текста, поэтому в `CountVectorizer` для него предусмотрен специальный параметр:

```
>>> vectorizer = CountVectorizer(min_df=1, stop_words='english')
```

Если вы точно знаете, какие стоп-слова хотели бы удалить, то можете передать их полный список. Если параметр `stop_words` равен `english`, то список будет состоять из 318 английских слов. Каких именно, покажет метод `get_stop_words()`:

```
>>> sorted(vectorizer.get_stop_words())[0:20]
['a', 'about', 'above', 'across', 'after', 'afterwards', 'again',
```

```
'against', 'all', 'almost', 'alone', 'along', 'already', 'also',
'although', 'always', 'am', 'among', 'amongst', 'amoungst']
```

Новый список содержит на семь слов меньше:

```
[u'actually', u'capabilities', u'contains', u'data', u'databases',
u'images', u'imaging', u'interesting', u'learning', u'machine',
u'permanently', u'post', u'provide', u'save', u'storage', u'store',
u'stuff', u'toy']
```

После исключения стоп-слов получаем такие результаты измерения сходства:

```
=== Post 0 with dist=1.41: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.

=== Post 1 with dist=0.86: Imaging databases provide storage
capabilities.

=== Post 2 with dist=0.86: Most imaging databases save images
permanently.

=== Post 3 with dist=0.77: Imaging databases store data.

=== Post 4 with dist=0.77: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 3 with dist=0.77
```

Теперь сообщения 1 и 2 сравнялись. Но расстояния изменились несущественно, потому что наши демонстрационные сообщения очень короткие. Картина будет совершенно другой, если взять реальные тексты.

Стемминг

Но одну вещь мы упустили. Одно и то же слово в разных грамматических формах мы считаем разными словами. Например, в сообщении 2 есть слова «imaging» и «images». Имеет смысл считать их, как одно слово, ведь они обозначают одно и то же понятие.

Нам нужна функция, которая производит стемминг, то есть выделяет из слова его основу. В библиотеке SciKit стеммера нет. Но можно скачать бесплатную библиотеку `Natural Language Toolkit (NLTK)`, где имеется стеммер, который легко подключить к `CountVectorizer`.

Установка и использование NLTK

Порядок установки NLTK в конкретную операционную систему подробно описан на странице `http://nltk.org/install.html`. К сожалению, для Python 3 эта библиотека пока официально не поддержи-

вается, а, значит, и pip для нее работать не будет. Но можно скачать пакет со страницы `http://www.nltk.org/nltk3-alpha/`, распаковать и установить вручную с помощью скрипта `setup.py`.

Для проверки успешности установки запустите интерпретатор Python и введите команду:

```
>>> import nltk
```

Очень неплохое пособие по NLTK имеется в книге Jacob Perkins «Python 3 Text Processing with NLTK 3 Cookbook», изданной Packt Publishing. Если хотите поэкспериментировать со стеммером, зайдите на страницу `http://text-processing.com/demo/stem/`.

В NLTK есть несколько стеммеров. И это необходимо, потому что в каждом языке свои правила стемминга. Для английского языка возьмем класс `SnowballStemmer`.

```
>>> import nltk.stem
>>> s = nltk.stem.SnowballStemmer('english')
>>> s.stem("graphics")
u'graphic'
>>> s.stem("imaging")
u'imag'
>>> s.stem("image")
u'imag'
>>> s.stem("imagination")
u'imagin'
>>> s.stem("imagine")
u'imagin'
```

Отметим, что результатом стемминга вовсе необязательно являются допустимые английские слова.

Стеммер работает и для глаголов:

```
>>> s.stem("buys")
u'buy'
>>> s.stem("buying")
u'buy'
```

В большинстве случаев, но не всегда[1]:

```
>>> s.stem("bought")
u'bought'
```

[1] bought – форма прошедшего времени неправильного глагола buy (покупать). Как видим, в этом случае стеммер ошибается. – *Прим. перев.*

Совместное использование векторизатора и стеммера из библиотеки NLTK

Нам нужно произвести стемминг сообщений перед их подачей классу `CountVectorizer`. В этом классе есть несколько точек подключения, позволяющих настроить этапы предварительной обработки и лексического анализа. Препроцессор и лексический анализатор могут быть переданы конструктору в качестве параметров. Мы не хотим помещать стеммер ни туда, ни сюда, потому что тогда нам пришлось бы заниматься лексическим анализом и нормировкой самостоятельно. Вместо этого мы переопределим метод `build_analyzer`:

```
>>> import nltk.stem
>>> english_stemmer = nltk.stem.SnowballStemmer('english')
>>> class StemmedCountVectorizer(CountVectorizer):
...  def build_analyzer(self):
...    analyzer = super(StemmedCountVectorizer, self).build_analyzer()
...    return lambda doc: (english_stemmer.stem(w) for w in analyzer(doc))
>>> vectorizer = StemmedCountVectorizer(min_df=1, stop_words='english')
```

При этом каждое сообщение будет подвергнуто следующей обработке.

1. Сначала на шаге предварительной обработки (в родительском классе) все буквы сообщения будут переведены в нижний регистр.
2. На шаге лексического анализа выделяются отдельные слова (в родительском классе).
3. И в завершение из каждого слова будет выделена основа.

В результате у нас получится на один признак меньше, потому что слова «images» и «imaging» сольются. Останется такой список признаков:

```
[u'actual', u'capabl', u'contain', u'data', u'databas', u'imag',
u'interest', u'learn', u'machin', u'perman', u'post', u'provid',
u'save', u'storag', u'store', u'stuff', u'toy']
```

Если после объединения слов «images» и «imaging» прогнать новый векторизатор со стеммингом для всех сообщений, то выяснится, что теперь на новое сообщение больше всего похоже сообщение 2, поскольку оно дважды содержит понятие «imag»:

```
=== Post 0 with dist=1.41: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.

=== Post 1 with dist=0.86: Imaging databases provide storage
capabilities.

=== Post 2 with dist=0.63: Most imaging databases save images
```

```
permanently.

=== Post 3 with dist=0.77: Imaging databases store data.

=== Post 4 with dist=0.77: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 2 with dist=0.63
```

Развитие концепции стоп-слов

Теперь, когда у нас есть разумный способ построить компактный вектор по зашумленному текстовому сообщению, вернемся назад и подумаем, в чем на самом деле смысл значений признаков.

Значения признаков – это просто счетчики вхождения термов в сообщение. Мы молчаливо предполагали, что чем больше это значение, тем важнее терм для данного сообщения. Но как быть, например, со словом «subject» (тема), которое естественно встречается в каждом сообщении? Можно, конечно, попросить `CountVectorizer` удалить его, воспользовавшись параметром `max_df`. Например, если задать для него значение `0.9`, то слова, встречающиеся в 90 и более процентах сообщений, будут игнорироваться. А если слово встречается в 89 процентах сообщений? Как выбрать правильную величину `max_df`? Проблема в том, что какое бы значение ни выбрать, всегда какие-то термы будут важнее для различения документов, чем другие.

Решить эту проблему можно только одним способом – подсчитав частоты термов для каждого сообщения и «оштрафовав» те, которые встречаются во многих сообщениях. Иными словами, мы хотим, чтобы данному терму в данном сообщении было сопоставлено большое значение, если он встречается в этом сообщении и мало где еще.

Именно в этом состоит смысл характеристики «частота терма – обратная частота документа» (term frequency – inverse document frequency, или TF-IDF). Здесь TF относится к подсчету, а IDF – к «штрафу». Наивная реализация могла бы выглядеть так:

```
>>> import scipy as sp
>>> def tfidf(term, doc, corpus):
...     tf = doc.count(term) / len(doc)
...     num_docs_with_term = len([d for d in corpus if term in d])
...     idf = sp.log(len(corpus) / num_docs_with_term)
...     return tf * idf
```

Как видите, мы не просто подсчитали термы, но и нормировали счетчики на длину документа. Поэтому длинные документы не получат несправедливого преимущества перед короткими.

Если взять показанный ниже список D уже разбитых на лексемы документов, то мы увидим, что термы обрабатываются по-разному, хотя в каждом документе встречаются с одинаковой частотой:

```
>>> a, abb, abc = ["a"], ["a", "b", "b"], ["a", "b", "c"]
>>> D = [a, abb, abc]
>>> print(tfidf("a", a, D))
0.0
>>> print(tfidf("a", abb, D))
0.0
>>> print(tfidf("a", abc, D))
0.0
>>> print(tfidf("b", abb, D))
0.270310072072
>>> print(tfidf("a", abc, D))
0.0
>>> print(tfidf("b", abc, D))
0.135155036036
>>> print(tfidf("c", abc, D))
0.366204096223
```

Видно, что терм a не значим ни для одного документа, потому что встречается во всех. Терм b важнее для документа abb, чем для abc, потому что встречается там дважды.

На практике граничных случаев больше, чем показано в этом примере. Но благодаря SciKit мы можем о них не думать, потому что все они учтены в классе TfidfVectorizer, наследующем CountVectorizer. Разумеется, не нужно забывать про наш стеммер:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> class StemmedTfidfVectorizer(TfidfVectorizer):
...  def build_analyzer(self):
...   analyzer = super(TfidfVectorizer, self).build_analyzer()
...   return lambda doc: (
       english_stemmer.stem(w) for w in analyzer(doc))
>>> vectorizer = StemmedTfidfVectorizer(min_df=1,
               stop_words='english', decode_error='ignore')
```

Теперь векторы документов вообще не содержат счетчиков. А содержат они значения TF-IDF для каждого терма.

Чего мы достигли и к чему стремимся

Пока что этап предварительной обработки включает следующие шаги:

1. Лексический анализ текста и разбиение его на лексемы.
2. Отбрасывание слов, которые встречаются слишком часто и потому не помогают находить релевантные сообщения.

3. Отбрасывание слов, которые встречаются так редко, что вряд ли встретятся в будущих сообщениях.
4. Подсчет оставшихся слов.
5. Вычисление TF-IDF по счетчикам с учетом всего корпуса текстов.

Можем снова поздравить себя. Этот процесс позволяет преобразовать исходный зашумленный текст в компактное представление в виде значений признаков.

Но при всей простоте и эффективности подхода на основе набора слов с дополнительными расширениями у него имеется ряд недостатков, о которых следует знать.

- **Не учитываются связи между словами.** Если принять описанный подход к векторизации, то у фраз «Car hits wall» (Машина врезалась в стену) и «Wall hits car» (Стена врезалась в машину) будет один и тот же набор признаков.
- **Не улавливается отрицание.** Например, фразы «I will eat ice cream» (Я стану есть мороженое) и «I will not eat ice cream» (Я не стану есть мороженое) с точки зрения векторов признаков очень похожи, но имеют противоположный смысл. Впрочем, эту проблему легко решить, если подсчитывать не только отдельные слова (униграммы), но также пары слов (биграммы) и тройки слов (триграммы).
- **Никак не обрабатываются ошибки в правописании.** Хотя человеку совершенно понятно, что слова «database» и «databas» означают одно и то же, в принятом подходе они считаются различными.

Но, чтобы не отвлекаться, мы все же останемся верны описанному подходу и воспользуемся им для эффективного построения кластеров.

Кластеризация

Итак, у нас имеются векторы, которые, как нам кажется, достаточно адекватно отражают содержание сообщений. Не должно вызывать удивления, что есть много способов сгруппировать эти векторы. Большинство алгоритмов кластеризации относятся к одной из двух категорий: плоские и иерархические.

Алгоритмы плоской кластеризации разбивают множество сообщений на несколько кластеров, никак не связанных между собой. Цель

проста – найти такое разбиение, чтобы все сообщения, попавшие в один кластер, были похожи друг на друга и в то же время отличались от сообщений из других кластеров. Во многих алгоритмах плоской кластеризации требуется задавать число кластеров заранее.

В алгоритмах иерархической кластеризации число кластеров задавать не нужно – алгоритм сам строит иерархию кластеров. Похожие сообщения собираются в одном кластере, а затем похожие кластеры объединяются в *суперкластер*. Это делается рекурсивно, пока не останется один кластер, содержащий все сообщения. По завершении процесса можно выбрать нужное число кластеров из построенной иерархии. Однако эффективность такого процесса невысока.

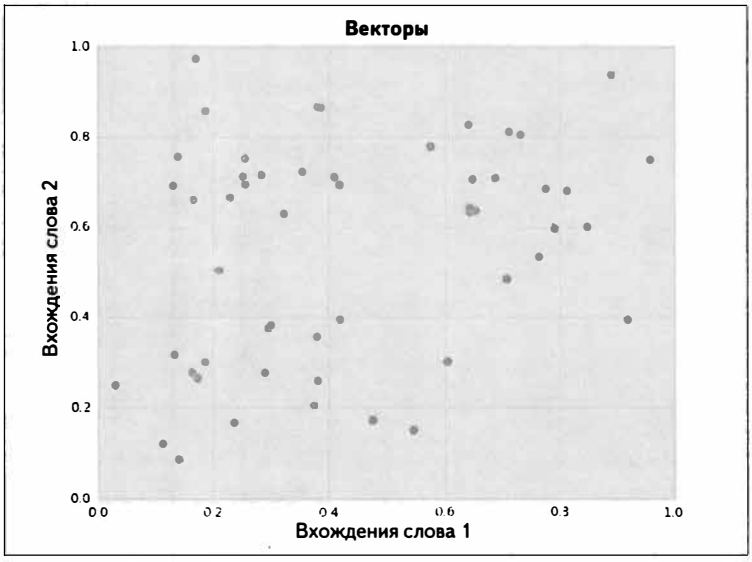
В пакете `sklearn.cluster` из библиотеки SciKit реализованы различные подходы к кластеризации. Прочитать о достоинствах и недостатках каждого из них можно на странице `http://scikit-learn.org/dev/modules/clustering.html`.

В следующих разделах мы будем пользоваться методом K средних – одним из алгоритмов плоской кластеризации – и поэкспериментируем с числом кластеров.

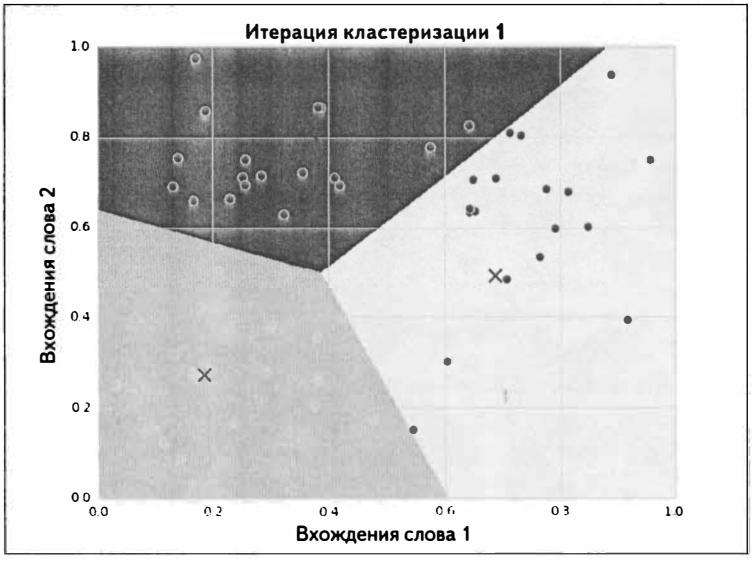
Метод K средних

Метод *K* средних – самый распространенный алгоритм плоской кластеризации. Получив требуемое число кластеров, `num_clusters`, он создает именно столько так называемых центроидов кластеров. Сначала алгоритм произвольно выбирает `num_clusters` сообщений и в качестве центроидов берет их векторы признаков. Затем каждое из оставшихся сообщений соотносится с ближайшим к нему центроидом, и так образуются кластеры. После этого вычисляются новые положения центроидов – путем усреднения по всем векторам каждого кластера. Разумеется, при этом изменяется распределение сообщений по кластерам. Некоторые сообщения оказываются ближе к другому кластеру. Поэтому происходит перемещение таких сообщений из одного кластера в другой. Это делается до тех пор, пока положение центроидов не стабилизируется. После нескольких итераций расстояния между старым и новым положением каждого центроида оказывается меньше порогового значения, и мы считаем, что процесс построения кластеров сошелся.

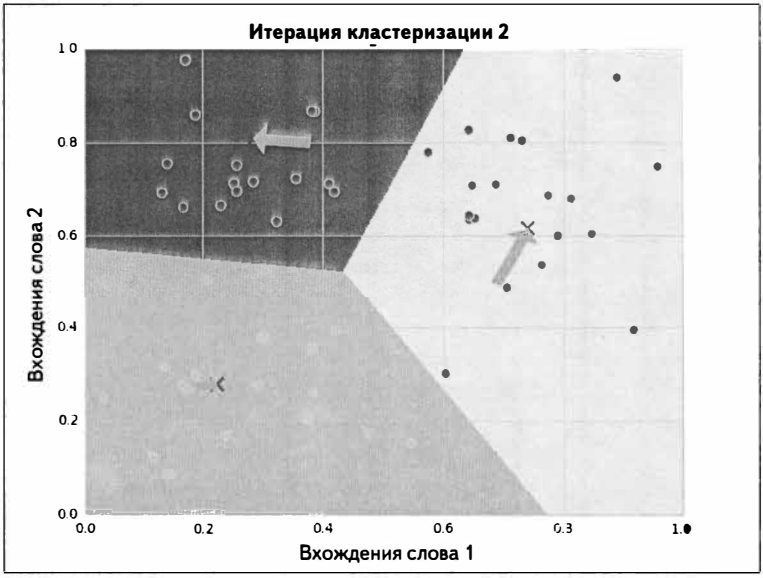
Рассмотрим простенький пример, когда сообщения состоят всего из двух слов. Каждая точка на следующем рисунке представляет один документ.



После одной итерации метода K средних, то есть выбора двух произвольных векторов в качестве начальных точек, распределения остальных по кластерам и изменения положения центроидов кластеров получается такая картина:



Поскольку центроиды кластеров сместились, мы должны переместить некоторые точки в другие кластеры и пересчитать положения центроидов. После второй итерации получаются такие кластеры:



Стрелками показано перемещение центроидов. После пяти итераций центроиды практически перестают смещаться (в SciKit пороговое отклонение по умолчанию равно 0.0001).

После стабилизации кластеров нам остается только запомнить положения центроидов и то, к каким кластерам они относятся. Когда поступает новый документ, мы векторизуем его, вычисляем расстояние до каждого центроида и относим документ к тому кластеру, центроид которого окажется ближе всего.

Тестовые данные для проверки наших идей

Чтобы протестировать алгоритм кластеризации, мы расстанемся с игрушечными примерами и подыщем данные, напоминающие те, с которыми ожидаем столкнуться в реальности. Нам нужны уже сгруппированные документы на технические темы, чтобы мы могли проверить, будет ли алгоритм работать, как мы предполагаем, когда ему предъявят новые документы.

Одним из стандартных для машинного обучения является набор данных `20newsgroup`, содержащий 18 826 сообщений из 20 новостных групп по разным тематикам – как техническим, например `comp.sys.mac.hardware` или `sci.crypt`, так и относящимся к политике и религии, например `talk.politics.guns` или `soc.religion`. Мы ограничимся только техническими группами. Если считать каждую новостную группу кластером, то нам будет очень легко проверить, какой подход к поиску взаимосвязанных сообщений окажется наилучшим.

Скачать набор данных можно со страницы `http://people.csail.mit.edu/jrennie/20Newsgroups`. Но гораздо удобнее скачать его с сайта MLComp по адресу `http://mlcomp.org/datasets/379` (обязательна бесплатная регистрация). В библиотеке SciKit уже есть специальный загрузчик для этого набора с очень удобными параметрами.

Этот набор поставляется в формате ZIP-файла `dataset-379-20news-18828_WJQIG.zip`, который мы распакуем в каталог `379`. Нужно будет еще сообщить SciKit путь к каталогу данных. Там находится файл метаданных и три подкаталога: `test`, `train` и `raw`. В каталоге `train` находится 60 % данных, предназначенных для обучения, а в каталоге `test` – 40 %, предназначенных для тестирования. Путь к каталогу указывается либо в переменной окружения `MLCOMP_DATASETS_HOME`, либо непосредственно в параметре `mlcomp_root` команды загрузки набора.

Сайт `http://mlcomp.org` организован для сравнения программ машинного обучения на разнообразных наборах данных. У него две цели: помочь в поиске подходящего набора для настройки программы и узнать, как другие использовали некий набор. Например, можно посмотреть, насколько хорошо работали придуманные другими алгоритмы на том или ином наборе, и сравнить с ними свой собственный.

Для удобства модуль `sklearn.datasets` включает функцию `fetch_20newsgroups`, которая автоматически скачивает данные:

```
>>> import sklearn.datasets
>>> all_data = sklearn.datasets.fetch_20newsgroups(subset='all')
>>> print(len(all_data.filenames))
18846
>>> print(all_data.target_names)
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware',
'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles',
'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt',
'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.christian',
```

```
'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.misc',
'talk.religion.misc']
```

Мы можем выбрать обучающий или тестовый набор:

```
>>> train_data = sklearn.datasets.fetch_20newsgroups(subset='train',
categories=groups)
>>> print(len(train_data.filenames))
11314
>>> test_data = sklearn.datasets.fetch_20newsgroups(subset='test')
>>> print(len(test_data.filenames))
7532
```

Для простоты возьмем не все группы, чтобы цикл эксперимента был короче. Для этого служит параметр `categories`:

```
>>> groups = ['comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware',
'comp.windows.x', 'sci.space']
>>> train_data = sklearn.datasets.fetch_20newsgroups(subset='train',
categories=groups)
>>> print(len(train_data.filenames))
3529

>>> test_data = sklearn.datasets.fetch_20newsgroups(subset='test',
categories=groups)
>>> print(len(test_data.filenames))
2349
```

Кластеризация сообщений

Вы, наверное, уже обратили внимание, что реальные данные обычно зашумлены. И набор данных из новостных групп – не исключение. Он даже содержит недопустимые символы, приводящие к ошибке `UnicodeDecodeError`.

Мы должны сказать векторизатору, что такие ошибки следует игнорировать:

```
>>> vectorizer = StemmedTfidfVectorizer(min_df=10, max_df=0.5,
...                 stop_words='english', decode_error='ignore')
>>> vectorized = vectorizer.fit_transform(train_data.data)
>>> num_samples, num_features = vectorized.shape
>>> print(«#samples: %d, #features: %d» % (num_samples, num_features))
#samples: 3529, #features: 4712
```

Теперь у нас есть пул из 3529 сообщений и построенный по ним 4172-мерный вектор признаков. Это и будет подано на вход алгоритма K средних. В этой главе мы будем использовать 50 кластеров, но надеемся, что любознательные читатели попробуют и другие значения.

```
>>> num_clusters = 50
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=num_clusters, init='random', n_init=1,
verbose=1, random_state=3)
>>> km.fit(vectorized)
```

Вот и все. Мы зафиксировали параметр `random_state`, чтобы вы могли получить точно такие же результаты. В реальных приложениях этого делать не нужно. После обучения модели информацию о кластерах можно получить из полей объекта `km`. Каждому векторизованному сообщению, участвовавшему в обучении, соответствует целочисленная метка в массиве `km.labels_`:

```
>>> print(km.labels_)
[48 23 31 ..., 6 2 22]
>>> print(km.labels_.shape)
3529
```

Центроиды кластеров можно получить с помощью `km.cluster_centers_`. В следующем разделе мы увидим, как соотнести с кластером новое сообщение с помощью метода `km.predict`.

Решение исходной задачи

А теперь соберем все вместе и продемонстрируем работу системы на примере следующего сообщения, которое мы запишем в переменную `new_post`:

> *Disk drive problems. Hi, I have a problem with my hard disk.*
> *After 1 year it is working only sporadically now.*
> *I tried to format it, but now it doesn't boot any more.*
> *Any ideas? Thanks.*

Мы уже знаем, что перед тем как вычислять кластер сообщения, его нужно векторизовать:

```
>>> new_post_vec = vectorizer.transform([new_post])
>>> new_post_label = km.predict(new_post_vec)[0]
```

Теперь, когда у нас имеются кластеры, уже нет необходимости сравнивать `new_post_vec` с векторами всех сообщений. Можно ограничиться лишь сообщениями в одном с ним кластерами. Давайте выберем индексы этих сообщений в исходном наборе данных:

```
>>> similar_indices = (km.labels_==new_post_label).nonzero()[0]
```

Результатом сравнения в круглых скобках является булев массив,

а метод `nonzero` возвращает меньший массив, содержащий индексы элементов, равных `True`.

С помощью метода `similar_indices` мы затем строим список сообщений вместе с оценками сходства:

```
>>> similar = []
>>> for i in similar_indices:
...     dist = sp.linalg.norm((new_post_vec - vectorized[i]).toarray())
...     similar.append((dist, dataset.data[i]))
>>> similar = sorted(similar)
>>> print(len(similar))
131
```

В том кластере, куда попало наше сообщение, находится 131 сообщение. Чтобы пользователь мог составить представление о том, какие есть похожие сообщения, мы покажем одно самое похожее (`show_at_1`) и два не так похожих, но все-таки не совсем посторонних – все берутся из одного и того же кластера.

```
>>> show_at_1 = similar[0]
>>> show_at_2 = similar[int(len(similar)/10)]
>>> show_at_3 = similar[int(len(similar)/2)]
```

В следующей таблице показаны эти сообщения и их оценки сходства.

Позиция	Сходство	Выдержка из сообщения
1	1.038	BOOT PROBLEM with IDE controller Hi, I've got a Multi I/O card (IDE controller + serial/parallel interface) and two floppy drives (5 1/4, 3 1/2) and a Quantum ProDrive 80AT connected to it. I was able to format the hard disk, but I could not boot from it. I can boot from drive A: (which disk drive does not matter) but if I remove the disk from drive A and press the reset switch, the LED of drive A: continues to glow, and the hard disk is not accessed at all. I guess this must be a problem of either the Multi I/o card or floppy disk drive settings (jumper configuration?) Does someone have any hint what could be the reason for it. [...]
2	1.150	Booting from B drive I have a 5 1/4" drive as drive A. How can I make the system boot from my 3 1/2" B drive? (Optimally, the computer would be able to boot: from either A or B, checking them in order for a bootable disk. But: if I have to switch cables around and simply switch the drives so that: it can't boot 5 1/4" disks, that's OK. Also, boot_b won't do the trick for me. [...] [...]

Позиция	Сходство	Выдержка из сообщения
3	1.280	IBM PS/1 vs TEAC FD Hello, I already tried our national news group without success. I tried to replace a friend s original IBM floppy disk in his PS/1-PC with a normal TEAC drive. I already identified the power supply on pins 3 (5V) and 6 (12V), shorted pin 6 (5.25"/3.5" switch) and inserted pullup resistors (2K2) on pins 8, 26, 28, 30, and 34. The computer doesn't complain about a missing FD, but the FD s light stays on all the time. The drive spins up o.k. when I insert a disk, but I can't access it. The TEAC works fine in a normal PC. Are there any points I missed? [...] [...]

Интересно, как оценка схожести зависит от сообщения. Первое сообщение содержит все характерные слова, встречающиеся в новом сообщении. Второе относится к проблемам загрузки (booting), но с гибких, а не с жестких дисков. Наконец, третье вообще не про жесткие диски и не про загрузку. И тем не менее, все они без сомнения написаны на ту же тему, что и новое сообщение.

Другой взгляд на шум

Не следует ожидать, что кластеризация окажется идеальной, то есть что все сообщения из одной новостной группы (например, `comp.graphics`) попадут в один кластер. Ниже приведен пример, показывающий, какой бывает шум. Для простоты ограничимся одним коротким сообщением.

```
>>> post_group = zip(train_data.data, train_data.target)
>>> all = [(len(post[0]), post[0], train_data.target_names[post[1]])
for post in post_group]
>>> graphics = sorted([post for post in all if
post[2]=='comp.graphics'])
>>> print(graphics[5])
(245, 'From: SITUNAYA@IBM3090.BHAM.AC.UK\nSubject:
test....(sorry)\nOrganization: The University of Birmingham, United
Kingdom\nLines: 1\nNNTP-Posting-Host: ibm3090.bham.ac.uk<...snip...>',
'comp.graphics')
```

Здесь нет никаких указаний на принадлежность к группе `comp.graphics`, если рассматривать только слова, оставшиеся после этапа предварительной обработки:

```
>>> noise_post = graphics[5][1]
>>> analyzer = vectorizer.build_analyzer()
```

```
>>> print(list(analyzer(noise_post)))
['situnaya', 'ibm3090', 'bham', 'ac', 'uk', 'subject', 'test',
'sorri', 'organ', 'univers', 'birmingham', 'unit', 'kingdom', 'line',
'nntp', 'post', 'host', 'ibm3090', 'bham', 'ac', 'uk']
```

И это мы произвели только разбиение на лексемы, приведение к нижнему регистру и удаление стоп-слов. А если к тому же исключить слова, которые на следующем этапе – при вызове метода `fit_transform` – будут отфильтрованы за счет параметров `min_df` и `max_df`, то ситуация станет еще хуже:

```
>>> useful = set(analyzer(noise_post)).intersection
(vectorizer.get_feature_names())
>>> print(sorted(useful))
['ac', 'birmingham', 'host', 'kingdom', 'nntp', 'sorri', 'test',
'uk', 'unit', 'univers']
```

Но и этого мало. Большинство этих слов часто встречаются и в других сообщениях, о чем свидетельствуют оценки IDF. Напомним, что чем выше показатель TF-IDF, тем лучше данный терм характеризует именно это сообщение. IDF является мультипликативным фактором: если его значение мало, значит, ценность терма невелика.

```
>>> for term in sorted(useful):
...     print('IDF(%s)=%.2f'%(term,
vectorizer._tfidf.idf_[vectorizer.vocabulary_[term]]))
IDF(ac)=3.51
IDF(birmingham)=6.77
IDF(host)=1.74
IDF(kingdom)=6.68
IDF(nntp)=1.77
IDF(sorri)=4.14
IDF(test)=3.83
IDF(uk)=3.70
IDF(unit)=4.42
IDF(univers)=1.91
```

Таким образом, термы с наивысшей различающей способностью, `birmingham` и `kingdom`, очевидно, не относятся к компьютерной графике, как и термы с низкими оценками IDF. Поэтому понятно, почему сообщения из разных новостных групп могут попасть в один кластер.

Нам, впрочем, это не так уж важно, потому что наша цель – сократить количество сообщений, с которыми нужно сравнивать новое. А то, из какой новостной группы взяты обучающие данные, большого интереса не представляет.

Настройка параметров

Ну а как на счет остальных параметров? Нельзя ли их подкорректировать, чтобы улучшить результаты?

А как же! Конечно, мы можем подобрать количество кластеров или попробовать разные значения параметра `max_features` векторизатора (обязательно попробуйте!). Кроме того, можно поиграть с разными начальными положениями центроидов. Ну и, наконец, на самом алгоритме K средних свет клином не сошелся. Например, можно использовать для кластеризации другие меры сходства: коэффициент Отиаи, коэффициент корреляции Пирсона, меру Жаккара. Есть простор для экспериментов.

Но прежде чем двигаться в этом направлении, нужно решить, что значит «лучше». В SciKit есть целый пакет, посвященный этому вопросу. Он называется `sklearn.metrics` и содержит полный спектр метрик для измерения качества кластеризации. Быть может, сначала стоит заглянуть туда – прямо в исходный код этого пакета.

Резюме

Это было непростое путешествие, начавшееся с предварительной обработки, продолжившееся обсуждением кластеризации и завершившееся решением, которое позволяет преобразовать зашумленный текст в компактное векторное представление, допускающее кластеризацию. Собственно на кластеризацию было потрачено больше половины усилий. Но попутно мы кое-что узнали об обработке текстов и о том, как далеко может завести простой подсчет в мире реальных данных.

Наше путешествие оказалось бы куда труднее, не будь библиотеки SciKit и входящих в нее замечательных пакетов. Но области для исследования остались. В этой главе мы лишь скользнули по поверхности, возможности библиотеки гораздо шире. В последующих главах мы познакомимся с ними получше.

ГЛАВА 4. Тематическое моделирование

В предыдущей главе мы занимались группировкой текстовых документов с применением методов кластеризации. Это средство весьма полезное, но не всегда наилучшее. Оно приводит к тому, что каждый текст попадает в один и только один кластер. Но вот эта книга посвящена машинному обучению и языку Python. Куда ее отнести – к работам по теме «Python» или по теме «машинное обучение»? В реальном книжном магазине книгу нужно поместить на какую-то одну полку. Но в Интернет-магазине она должна присутствовать в обеих рубриках. Это, конечно, не значит, что ее следует включать во все вообще рубрики, скажем, в раздел, посвященный кулинарии.

В этой главе мы познакомимся с методами, которые позволяют относить каждый документ к нескольким **темам**, а не помещать в один-единственный кластер. Темы будут определяться автоматически по имеющемуся набору документов. Документами могут быть как книги, так и более короткие тексты, например, сообщение в блоге, новость или электронное письмо.

Хотелось бы уметь определять центральную и второстепенные темы документа. В этой книге часто упоминается построение графиков, но центральной темой является все же не оно, а машинное обучение. Отрасль машинного обучения, в которой рассматриваются подобные проблемы и которой посвящена эта глава, называется **тематическим моделированием**.

Латентное размещение Дирихле

Так получилось, что в машинном обучении есть два метода с аббревиатурой **LDA**: латентное размещение Дирихле (latent Dirichlet allocation), один из алгоритмов тематического моделирования, и линейный дискриминантный анализ (linear discriminant analysis) – ал-

горитм классификации. Кроме аббревиатуры, у них нет ничего общего, и иногда это приводит к недоразумениям. В модуле scikit-learn есть подмодуль `sklearn.lda`, в котором реализован линейный дискриминантный анализ. В настоящее время метод латентного размещения Дирихле в scikit-learn не реализован.

Математические идеи, лежащие в основе метода латентного размещения Дирихле, довольно сложны, и мы не будем вдаваться в детали. Интересующиеся читатели с авантюрным складом ума найдут все относящиеся к делу уравнения в википедии по адресу `http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation`.

Однако можно составить общее представление о LDA на интуитивном уровне. LDA принадлежит к классу так называемых порождающих моделей, поскольку они сопровождаются пояснением, описывающим, как были сгенерированы данные. Разумеется, это пояснение – всего лишь приближение к реальности, призванное упростить машинное обучение. В LDA мы сначала создаем темы, назначая словам веса в форме вероятностей. В каждой теме одному и тому же слову назначены разные веса. Так, в теме «Python» у слова «variable» (переменная) будет высокая вероятность, а у слова «inebriated» (подвыпивший) – низкая. Желая сгенерировать новый документ, мы сначала выбираем соответствующие ему темы, а затем комбинируем слова, относящиеся к этим темам.

Пусть, например, в книгах обсуждаются всего три темы:

- машинное обучение;
- язык Python;
- кулинария.

С каждой темой ассоциирован список слов. В той книге, что вы сейчас читаете, будут встречаться слова из первых двух тем, положим, в пропорции 50/50. Доли необязательно должны быть одинаковыми, возможно и соотношение 70/30. Порождая новый текст, мы выбираем слово за словом. Сначала решаем, из какой темы брать слово; решение принимается случайно, но в соответствии с весами тем. Определившись с темой, мы выбираем слово из списка английских слов, ассоциированных с этой темой.

В этой модели порядок слов не играет роли. То есть это модель *набора слов*, с которой мы уже встречались в предыдущей главе. Это очень грубое приближение к реальному языку, но часто оно оказывается достаточным, потому что одно лишь знание частот слов, встречающихся в документе, позволяет принимать решения в алгоритмах машинного обучения.

На практике темы нам заранее неизвестны. Наша задача – получить набор текстов и восстановить по нему пояснение, то есть выяснить, какие темы вообще представлены и к каким из них относится каждый документ.

Построение тематической модели

К сожалению, scikit-learn не поддерживает метод латентного размещения Дирихле. Поэтому мы воспользуемся написанным на Python пакетом gensim. Этот пакет разработан Радимом Ржехоржеком – исследователем в области машинного обучения и консультантом из Великобритании. Сначала установим пакет командой

```
pip install gensim
```

В качестве данных мы возьмем собрание новостей агентства **Associated Press (AP)**. Это стандартный набор данных для исследований по машинному обучению, который использовался в некоторых ранних работах по тематическому моделированию. Скачав данные, загрузим их в память:

```
>>> from gensim import corpora, models
>>> corpus = corpora.BleiCorpus('./data/ap/ap.dat', './data/ap/vocab.txt')
```

В переменной `corpus` хранятся все текстовые документы в формате, удобном для обработки. Теперь мы можем построить на основе этого объекта тематическую модель:

```
>>> model = models.ldamodel.LdaModel(
            corpus,
            num_topics=100,
            id2word=corpus.id2word)
```

В результате этого вызова конструктора производится статистическая обработка корпуса текстов и выявляется, какие темы в нем представлены. Получившуюся модель можно исследовать с разных точек зрения. Можно с помощью конструкции `model[doc]` вывести список тем, ассоциированных с документом:

```
>>> doc = corpus.docbyoffset(0)
>>> topics = model[doc]
>>> print(topics)
[(3, 0.023607255776894751),
 (13, 0.11679936618551275),
 (19, 0.075935855202707139),
```

```
....
 (92, 0.10781541687001292)]
```

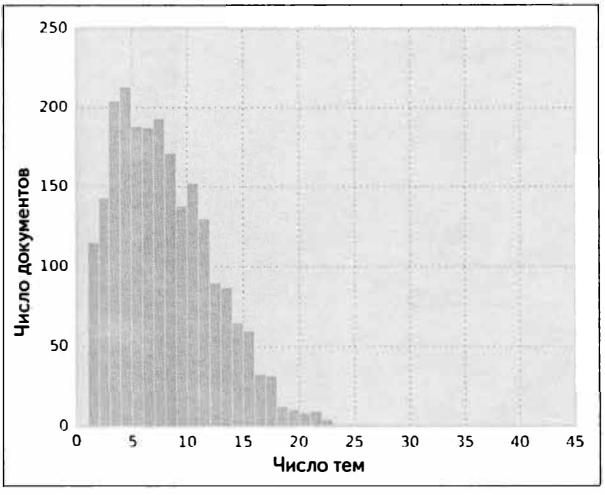
На вашем компьютере результат почти наверняка будет другим! В обучающем алгоритме используются случайные величины, поэтому при многократном обучении тематической модели на одних и тех же данных всякий раз получается новый результат. Но если данные ведут себя хорошо, то некоторые качественные свойства модели будут неизменны. Например, если использовать темы для сравнения документов, что мы и делаем, то сходство будет устойчивым, слабо изменяющимся свойством. С другой стороны, порядок тем будет совершенно различным.

Результат выдается в виде списка пар `(topic_index, topic_weight)`. Мы видим, что с каждым документом ассоциирована лишь часть всех тем (в примере выше для тем 0, 1 и 2 вес не указан, то есть равен нулю). Тематическая модель разрежена, то есть всего тем много, но каждый отдельный документ принадлежит лишь немногим. Строго говоря, это не совсем так, поскольку в модели LDA у любой темы ненулевая вероятность, но для некоторых она настолько мала, что можно считать ее равной нулю, не жертвуя качеством аппроксимации.

Можно продолжить исследование и построить гистограмму тем, ассоциированных с документом:

```
>>> num_topics_used = [len(model[doc]) for doc in corpus]
>>> plt.hist(num_topics_used)
```

Получится такой график:



Вектор или матрица называют **разреженными**, если большая часть элементов равна нулю (или настолько мала, что можно без ущерба для точности решения считать их нулевыми). Поэтому релевантны лишь немногие значения.

Часто задачу, кажущуюся неподъемно большой, удается решить, потому что данные разрежены. Например, любая веб-страница теоретически может ссылаться на любую другую, но на практике граф ссылок очень сильно разрежен, потому что имеются ссылки лишь на небольшое число страниц.

На рисунке выше видно, что примерно 150 документов относятся к 5 темам, а с большинством ассоциировано от 10 до 12 тем. Нет ни одного документа, в котором бы шла речь более чем о 20 разных темах.

Это в значительной степени связано с выбором параметра `alpha`. Его точный смысл выражается в очень абстрактных терминах, но чем больше значение `alpha`, тем больше тем будет у каждого документа.

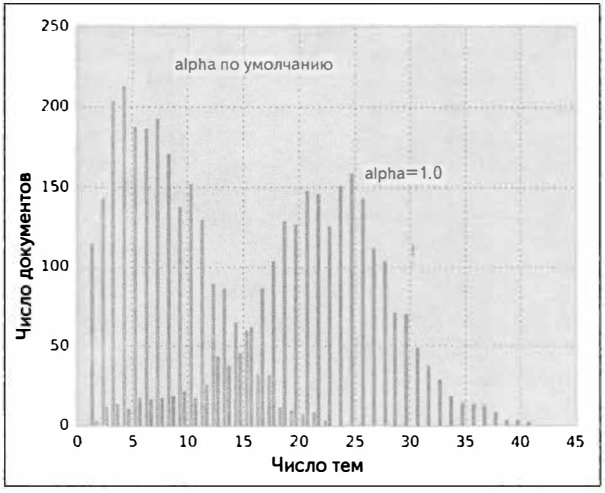
Параметр `alpha` должен быть больше нуля, и обычно берется значение меньше 1. Чем меньше `alpha`, тем меньше ожидаемое количество тем документа. По умолчанию в gensim `alpha` принимается равным `1/num_topics`, но можно задать значение явно, передав конструктору `LdaModel` одноименный параметр:

```
>>> model = models.ldamodel.LdaModel(
            corpus,
            num_topics=100,
            id2word=corpus.id2word,
            alpha=1)
```

Это значение `alpha` больше подразумеваемого по умолчанию, поэтому у каждого документа должно быть больше тем. На комбинированной гистограмме ниже мы видим, что gensim ведет себя в соответствии с ожиданиями – с каждым документом теперь ассоциировано больше тем.

Сейчас для многих документов количество тем варьируется от 20 до 25. Если уменьшить значение, то будет наблюдаться противоположная тенденция (скачав код из сетевого репозитория, вы сможете сами поэкспериментировать с заданием параметров).

Что это за темы? Технически это мультиномиальное распределение по словам, то есть каждому слову из словаря назначается вероятность относительно некоторой темы. Чем выше вероятность, тем больше шансов, что слово связано с данной темой.



Наш мозг плохо приспособлен для рассуждений о распределениях вероятности, но список слов мы воспринимаем легко. Поэтому обычно темы описываются списком слова с наибольшими вероятностями.

В таблице ниже перечислены первые десять тем:

Номер темы	Тема
1	dress military soviet president new state capt carlucci states leader stance government
2	koch zambia lusaka oneparty orange kochs party i government mayor new political
3	human turkey rights abuses royal thompson threats new state wrote garden president
4	bill employees experiments levin taxation federal measure legislation senate president whistleblowers sponsor
5	ohio july drought jesus disaster percent hartford mississippi crops northern valley virginia
6	united percent billion year president world years states people i bush news
7	b hughes affidavit states united ounces squarefoot care delaying charged unrealistic bush
8	yeutter dukakis bush convention farm subsidies uruguay percent secretary general i told
9	kashmir government people srinagar india dumps city two jammukashmir group moslem pakistan
10	workers vietnamese irish wage immigrants percent bargaining last island police hutton I

На первый взгляд, полная неразбериха, но, вчитавшись в список слов, мы начинаем понимать, что темы – не просто произвольно выхваченные слова, а образуют логические группы. Мы также видим, что темы относятся к старым новостям – к тому времени, когда еще существовал Советский Союз, а Горбачев был Генеральным секретарем. Темы можно представить также в виде облака слов, в котором слова с большей вероятностью набраны более крупным шрифтом. Вот, например, как выглядит тема, относящаяся к политике на Ближнем Востоке:

Видно также, что некоторые слова имело бы смысл удалить (например, слово «I»), потому что они малоинформативны – являются стоп-словами. При построении тематической модели полезно отфильтровывать стоп-слова, иначе могут появиться темы, целиком составленные из таких слов. Кроме того, желательно произвести предварительную обработку – стемминг для нормализации форм множественного числа и глагольных форм. Этот процесс подробно освещался в предыдущей главе. Интересующийся читатель может скачать код с сайта книги и поэкспериментировать с различными вариациями на эту тему.

Для построения облака слов имеются различные инструменты. Рисунки в этой главе были созданы с помощью пакета pytagcloud. Он зависит от нескольких других пакетов и не связан с машинным обучением как таковым, поэтому в основном тексте не рассматривается. Однако в сетевом репозитории имеется весь код, необходимый для построения рисунков.

Сравнение документов по темам

Темы могут быть полезны сами по себе для построения своего рода виньеток из слов, как на рисунке выше. Такие наглядные представления можно использовать для навигации по большим наборам документов. Например, на сайте разные темы можно показать в виде разных облаков слов, позволяющих пользователю с помощью нескольких щелчков мышью добраться до документов. На самом деле, именно такая методика реально применялась для анализа больших наборов документов.

Однако чаще темы служат промежуточным средством для достижения другой цели. Теперь, когда у нас для каждого документа имеется оценка соотнесенности его с разными темами, мы можем сравнивать документы в пространстве тем. Это означает, что мы сравниваем документы не по отдельным словам, а по тому, насколько близки затрагиваемые в них темы.

И это очень эффективный подход, потому что два текстовых документа, почти не имеющие общих слов могут, тем не менее, относиться к одной и той же теме! Возможно, в них просто используются разные словесные конструкции (скажем, в одном документе говорится о «президенте США», а в другом – о «Бараке Обаме»).

Тематические модели хороши сами по себе для визуализации и исследования данных. Но они полезны и в качестве промежуточного шага при решении многих других задач.

Сейчас мы можем по-другому подойти к упражнению из предыдущей главы и поискать сообщение, больше всего похожее на заданное в вопросе, исходя из сходства по темам. Если раньше при сравнении двух документов мы использовали их векторы слов, то теперь сравним векторы тем.

Для этого спроецируем документы в пространство тем. Иначе говоря, мы хотим построить вектор тем, описывающий документ. Как решается такая задача **понижения размерности** в общем случае – вопрос сам по себе важный и интересный, которому мы посвятим целую главу. А пока просто покажем, как использовать для этой цели тематические модели; вычислив для каждого документа вектор тем, мы сможем производить над этими векторами различные операции, забыв об исходных словах. Если темы значимы, то они потенциально могут оказаться более информативными, чем сами слова. Кроме того,

так можно и сэкономить на вычислениях, потому что сравнить векторы с вероятностями сотни тем гораздо быстрее, чем векторы, размер которых сравним с размером словаря (тысячи термов).

Мы видели, как с помощью пакета gensim вычислить темы всех документов из корпуса. Сделаем это, сохраним результаты в массивах NumPy и вычислим попарные расстояния:

```
>>> from gensim import matutils
>>> topics = matutils.corpus2dense(model[corpus],
    num_terms=model.num_topics)
```

Переменная `topics` содержит матрицу тем. Для вычисления попарных расстояний можно воспользоваться функцией `pdist` из библиотеки SciPy. Один ее вызов вычисляет все значения `sum((topics[ti] - topics[tj])**2)`:

```
>>> from scipy.spatial import distance
>>> pairwise = distance.squareform(distance.pdist(topics))
```

Далее применим еще один трюк; присвоим диагональным элементам матрицы `distance` некое большое значение (оно должно быть больше всех остальных элементов матрицы):

```
>>> largest = pairwise.max()
>>> for ti in range(len(topics)):
...     pairwise[ti,ti] = largest+1
```

Вот и всё! Для каждого документа можно легко найти ближайший к нему (получился вариант классификатора по ближайшему соседу):

```
>>> def closest_to(doc_id):
...     return pairwise[doc_id].argmin()
```

Отметим, что это решение не сработало бы, если бы мы не присвоили диагональным элементам большое значение: функция всегда возвращала бы сам документ, потому что он больше всего похож на себя самого (за исключением маловероятного случая, когда у двух элементов в точности одинаковое распределение вероятностей тем, что практически невозможно, если только они не совпадают буквально).

Приведем для примера такой документ-запрос (это второй документ в нашем наборе):

```
From: geb@cs.pitt.edu (Gordon Banks)
Subject: Re: request for information on "essential tremor" and
```

```
Indrol?

In article <1q1tbnINNnfn@life.ai.mit.edu> sundar@ai.mit.edu
writes:

Essential tremor is a progressive hereditary tremor that gets
worse
when the patient tries to use the effected member. All limbs,
vocal
cords, and head can be involved. Inderal is a beta-blocker and
is usually effective in diminishing the tremor. Alcohol and
mysoline
are also effective, but alcohol is too toxic to use as a
treatment.
--
----------------------------------------------------------------
----------
Gordon Banks N3JXP | "Skepticism is the chastity of the
intellect, and
geb@cadre.dsl.pitt.edu | it is shameful to surrender it too
soon."
----------------------------------------------------------------
------------
```

Если мы попросим найти наиболее похожий документ – `closest_to(1)` – то получим такой результат:

```
From: geb@cs.pitt.edu (Gordon Banks)
Subject: Re: High Prolactin
In article <93088.112203JER4@psuvm.psu.edu> JER4@psuvm.psu.edu
(John E. Rodway) writes:
>Any comments on the use of the drug Parlodel for high prolactin
in the blood?

>

It can suppress secretion of prolactin. Is useful in cases of
galactorrhea.
Some adenomas of the pituitary secret too much.

--
----------------------------------------------------------------
----------
Gordon Banks N3JXP | "Skepticism is the chastity of the
intellect, and
geb@cadre.dsl.pitt.edu | it is shameful to surrender it too
soon."
```

Система возвращает сообщение того же автора, в котором обсуждаются побочные действия лекарства.

Моделирование всей википедии

Первые реализации LDA были медленными, поэтому их применение ограничивалось лишь небольшими корпусами документов, но современные алгоритмы отлично работают и с очень большими наборами данных. Следуя документации по gensim, мы построим тематическую модель всего англоязычного сегмента википедии. На это уйдет несколько часов, но задача решаема даже на ноутбуке! Имея кластер компьютеров, это можно было бы сделать гораздо быстрее, но рассмотрение такой конфигурации мы отложим на потом.

Сначала скачаем все данные википедии с сайта `http://dumps.wikimedia.org`. Это большой файл (больше 10 ГБ), так что на его скачивание уйдет некоторое время, если только вы не являетесь счастливым пользователем очень быстрого подключения к Интернету. Затем проиндексируем данные:

```
python -m gensim.scripts.make_wiki \
enwiki-latest-pages-articles.xml.bz2 wiki_en_output
```

Эту команду следует запускать из оболочки ОС, а не из оболочки Python. Спустя несколько часов в том же каталоге, что исходный файл, появится индекс. Теперь можно приступать к построению собственно тематической модели. Процесс выглядит точно так же, как для небольшого набора данных AP. Сначала импортируем несколько пакетов:

```
>>> import logging, gensim
```

Теперь настроим протоколирование, воспользовавшись стандартным модулем Python (с помощью которого gensim выводит сообщения о состоянии). Этот шаг необязателен, но хорошо бы все-таки видеть, что происходит:

```
>>> logging.basicConfig(
    format='%(asctime)s : %(levelname)s : %(message)s',
    level=logging.INFO)
```

Далее загрузим предварительно обработанные данные:

```
>>> id2word = gensim.corpora.Dictionary.load_from_text(
              'wiki_en_output_wordids.txt')
>>> mm = gensim.corpora.MmCorpus('wiki_en_output_tfidf.mm')
```

Наконец, построим LDA-модель, как и раньше:

```
>>> model = gensim.models.ldamodel.LdaModel(
            corpus=mm,
            id2word=id2word,
```

```
num_topics=100,
update_every=1,
chunksize=10000,
passes=1)
```

Это тоже займет пару часов. Сообщения о ходе работы выводятся на консоль, так что вы будете знать, сколько еще ждать.

Созданную тематическую модель можно сохранить в файле, чтобы не пришлось строить ее заново:

```
>>> model.save('wiki_lda.pkl')
```

Если вы завершите сеанс и возобновите работу позже, то сможете загрузить модель такой командой (предварительно нужно, конечно, импортировать необходимые пакеты):

```
>>> model = gensim.models.ldamodel.LdaModel.load('wiki_lda.pkl')
```

Объект `model` позволяет исследовать набор документов и построить матрицу `topics`, как мы уже делали раньше.

Видно, что и эта модель является разреженной, хотя документов гораздо больше, чем раньше (на момент написания книги свыше 4 миллионов):

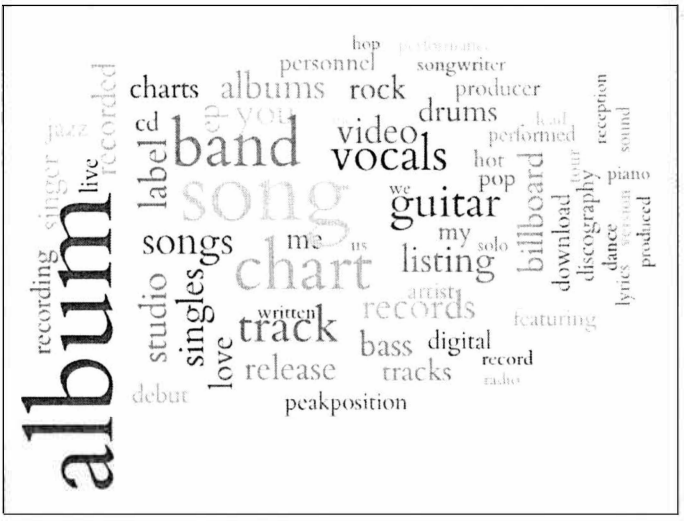
```
>>> lens = (topics > 0).sum(axis=0)
>>> print(np.mean(lens))
6.41
>>> print(np.mean(lens <= 10))
0.941
```

Таким образом, в среднем в документе упоминается 6.4 тем и в 94% документов упоминается не более 10 тем.

Можно поинтересоваться, какая самая обсуждаемая тема в википедии. Сначала вычислим суммарный вес каждой темы (просуммировав веса по всем документам), а затем выберем слова, относящиеся к теме с наибольшим весом. Для этого нужно выполнить такой код:

```
>>> weights = topics.sum(axis=0)
>>> words = model.show_topic(weights.argmax(), 64)
```

С помощью тех же инструментов визуализации, что и выше, мы видим, что больше всего статей посвящено музыке, причем это очень компактная тема. Целых 18% страниц в википедии так или иначе связаны с музыкой (с этой темой ассоциировано 5,5% всех слов в википедии). Взгляните на рисунок:



Рисунки и числа получены на момент написания книги. Поскольку википедия не стоит на месте, у вас могут получиться другие результаты. Мы полагаем, что общие тенденции сохранятся, но детали могут измениться.

Можно также найти наименее обсуждаемую тему:

```
>>> words = model.show_topic(weights.argmin(), 64)
```

Понять, что это за тема, труднее, но многие наиболее употребительные слова касаются аэропортов в восточных странах. Тема затрагивается лишь в 1,6 % документов, и к ней относится 0,1 % слов.

Выбор числа тем

До сих пор в этой главе мы задавали фиксированное число тем – 100. Это число выбрано совершенно произвольно, с тем же успехом можно было взять 10 или 200. По счастью, для многих применений число тем не слишком существенно. Если вы собираетесь использовать тематическую модель только в качестве промежуточного шага, как мы делали при поиске похожих сообщений, то поведение конечной системы редко оказывается сильно зависящим от точного числа тем в модели. Это означает, что при наличии достаточно большого числа тем, неважно будет их 100 или 200, рекомендации не слишком различаются; обычно 100 – вполне приемлемая величина (а 20 для набора текстовых документов общего вида слишком мало). То же самое относится к выбору параметра `alpha`. Конечно, распределение по темам зависит от него, но окончательные результаты устойчивы к его изменению.

Тематическое моделирование часто является лишь одним из этапов на пути к конечной цели. В таком случае не слишком важно, какие конкретно параметры используются. Вне зависимости от числа тем или значения параметра alpha конечная система выдает практически идентичные результаты.

С другой стороны, если вы планируете исследовать сами темы или построить систему их визуализации, то следует попробовать несколько значений и посмотреть, при каком получаются наиболее полезные или привлекающие внимание результаты.

Существуют также методы, которые автоматически определяют число тем в зависимости от набора данных. Одна из наиболее популярных моделей такого рода называется **иерархический процесс Дирихле**. Ее полное математическое описание слишком сложно для этой книги, но в пояснении мы можем сказать, что вместо выбора фиксированного числа тем, как в случае LDA, нужно генерировать темы вместе с данными – поочередно. Когда генератор приступает к новому документу, он может либо выбрать уже существующую тему, либо создать новую. Чем больше создано тем, тем меньше вероятность создания новой, хотя она всегда ненулевая.

Это означает, что чем больше имеется документов, тем больше будет тем. Это утверждение, на первый взгляд, противоречит интуиции, но, немного поразмыслив, мы придем к выводу, что оно совершенно правильно. Мы группируем документы и чем больше у нас примеров, тем к большему числу рубрик мы сможем их отнести. Если у нас есть всего несколько новостей, то все их можно отнести к рубрике «Спорт». Но по мере увеличения числа примеров выделяются более точные рубрики: «Хоккей», «Футбол» и т. д. Имея еще больше данных, мы можем различить такие нюансы, как отдельные команды или даже отдельные игроки. То же относится и к людям. В группе людей с разным образованием, где мало «компьютерщиков», можно их всех поместить в одну категорию. Из группы большего размера можно выделить программистов и системных администраторов, а в реальном мире есть даже разные категории для программистов на Python и на Ruby.

В пакете gensim реализован **иерархический процесс Дирихле (HDP)**. Воспользоваться им очень просто. В написанном ранее коде для LDA достаточно лишь заменить обращение к `gensim.models.ldamodel.LdaModel` на обращение к конструктору `HdpModel`:

```
>>> hdp = gensim.models.hdpmodel.HdpModel(mm, id2word)
```

И это всё (правда, вычисление займет больше времени – бесплатных завтраков не бывает). Использовать получившуюся модель можно точно так же, как LDA-модель, только задавать количество тем заранее уже не нужно.

Резюме

В этой главе мы обсудили тематическое моделирование. Это методика, более гибкая, чем кластеризация, потому что позволяет относить документ сразу к нескольким группам. Для исследования этих методов мы воспользовались новым пакетом, gensim.

Первоначально тематическое моделирование было разработано для анализа текстов, и в этом контексте понять его проще всего, но в главе о машинном зрении мы увидим, что некоторые методы можно применить и к изображениям. Тематические модели играют важную роль в современных исследованиях по машинному зрению. Вообще, эта глава, в отличие от предыдущих, очень близка к передовым рубежам исследований в области алгоритмов машинного обучения. Первоначальный вариант алгоритма LDA был опубликован

в научном журнале за 2003 год, метод, используемый в пакете gensim для обработки данных википедии, разработан только в 2010 году, а алгоритм HDP – в 2011. Исследования на эту тему продолжаются, так что вы можете найти много разных вариаций и моделей с такими причудливыми названиями, как *индийский буфет* (не путать с *китайским рестораном*, это другая модель) или *размещение патинко* (патинко – японская игра, нечто среднее между игорным автоматом и пинболлом).

Мы рассмотрели несколько основных моделей машинного обучения: классификация, кластеризация и тематическое моделирование.

В следующей главе мы вернемся к классификации, но на этот раз будем изучать передовые алгоритмы и подходы.

ГЛАВА 5. Классификация – выявление плохих ответов

Научившись выделять полезные признаки из текста, мы можем взяться за построение классификатора на основе реальных данных. Вернемся к введенному в главе 3 воображаемому сайту, на котором пользователь может задавать вопросы и получать на них ответы.

Перед владельцами таких вопросно-ответных сайтов стоит вечная проблема: как обеспечить достойное качество получаемого от пользователей материала. Такие сайты, как StackOverflow, прилагают значительные усилия, чтобы побудить пользователей с помощью различных средств оценивать содержимое. Они предлагают знаки отличия и бонусные очки за шлифовку текста вопроса и за предложенные ответы.

Особенно удачной инициативой следует признать имеющуюся у автора вопроса возможность пометить один ответ как принятый (за это также полагаются поощрения). Автор помеченного ответа получает дополнительные очки.

А не правда ли, было бы здорово, если бы пользователь, еще только набирая текст ответа, видел, насколько этот ответ хорош? Это означает, что сайт постоянно оценивает ответ по мере его ввода и сообщает о наличии признаков плохо го ответа. Тогда у пользователя будет стимул усерднее поработать над ответом (быть может, добавить пример кода или картинку) и тем самым улучшить систему в целом.

В этой главе мы создадим такой механизм.

План действий

Поскольку мы строим систему на основе сильно зашумленных реальных данных, эта глава не для слабых духом. Мы не найдем идеальный классификатор, дающий стопроцентную верность, поскольку часто

даже люди не могут прийти к согласию относительно качества ответа (достаточно посмотреть на комментарии на сайте StackOverflow). Напротив, мы обнаружим, что некоторые задачи и, в частности, эта настолько трудны, что по ходу дела приходится корректировать исходные цели. Мы начнем с классификации по ближайшим соседям, выясним, что этот подход не слишком пригоден для данной задачи, перейдем на логистическую регрессию и в конце концов отыщем решение, которое даст прогноз достаточно высокого качества, но не на всех ответах. Напоследок мы посмотрим, как выявить победителя и развернуть его на целевой системе.

Учимся классифицировать классные ответы

Задача классификации – поставить в соответствие образцам данных подходящие **классы**, или **метки**. Для этого нужно ответить на два вопроса:

- как представлять образцы данных?
- какая модель или структура лежит в основе классификатора?

Подготовка образца

В простейшем варианте образец – это текст ответа, а метка – бинарное значение, показывающее принял автор вопроса этот ответ или нет. Но простой текст – крайне неудобное представление входных данных для большинства алгоритмов машинного обучения. Алгоритмам нужны числа. Поэтому наша задача – извлечь из текста полезные признаки, которые алгоритм сможет использовать для выработки метки.

Настройка классификатора

Подобрав достаточное количество пар (текст, метка), мы можем обучить **классификатор**. Вариантов существует множество, у каждого свои плюсы и минусы. Назовем лишь самые употребительные алгоритмы: логистическая регрессия, решающие деревья, метод опорных векторов (SVM), наивный байесовский классификатор. В этой главе мы сравним рассмотренный ранее метод, основанный на образцах, – классификацию по ближайшим соседям – и метод логистической регрессии, основанный на модели.

Получение данных

К счастью для нас, разработчики сайта StackOverflow предоставляют большую часть данных, обитающих во вселенной StackExchange, частью которой является и StackOverflow, на условиях лицензии cc-wiki. На момент написания этой книги последнюю выгрузку данных можно было скачать по адресу `https://archive.org/details/stackexchange`. В нее входят данные со всех вопросно-ответных сайтов из семейства StackExchange. Для StackOverflow имеется несколько файлов, нам из них понадобится только `stackoverflow.com-Posts.7z` размером 5,2 ГБ.

После распаковки мы получим примерно 26 ГБ данных в формате XML, содержащих все вопросы и ответы в виде элементов `row`, расположенных внутри корневого элемента `posts`:

```
<?xml version='1.0' encoding='utf-8'?>
<posts>
...
<row Id="4572748" PostTypeId="2" ParentId="4568987"
CreationDate="2011-01-01T00:01:03.387" Score="4" ViewCount=""
Body="&lt;p&gt;IANAL, but &lt;a
href=&quot;http://support.apple.com/kb/HT2931&quot;
rel=&quot;nofollow&quot;&gt;this&lt;/a&gt; indicates to me that you
cannot use the loops in your
application:&lt;/p&gt;&#xA;&#xA;&lt;blockquote&gt;&#xA;
&lt;p&gt;...however, individual audio loops may&#xA; not be
commercially or otherwise&#xA; distributed on a standalone basis,
nor&#xA; may they be repackaged in whole or in&#xA; part as audio
samples, sound effects&#xA; or music beds.&quot;&lt;/p&gt;&#xA;
&#xA; &lt;p&gt;So don't worry, you can make&#xA; commercial music
with GarageBand, you&#xA; just can't distribute the loops as&#xA;
loops.&lt;/p&gt;&#xA;&lt;/blockquote&gt;&#xA;" OwnerUserId="203568"
LastActivityDate="2011-01-01T00:01:03.387" CommentCount="1" />
...
</posts>
```

Имя	Тип	Описание
`Id`	`Integer`	Уникальный идентификатор
`PostTypeId`	`Integer`	Тип сообщения. Нам интересны только значения: • вопрос; • ответ. Остальные значения игнорируются.

Имя	Тип	Описание
ParentId	Integer	Идентификатор вопроса, к которому относится ответ (для вопросов отсутствует)
CreationDate	DateTime	Дата отправки
Score	Integer	Оценка сообщения
ViewCount	Integer или Empty	Количество просмотров сообщения
Body	String	Полный текст сообщения в HTML-кодированном виде
OwnerUserId	Id	Уникальный идентификатор автора сообщения. Если 1, то это вопрос из вики
Title	String	Заголовок вопроса (для ответов отсутствует)
AcceptedAnswerId	Id	Идентификатор принятого ответа (для ответов отсутствует)
CommentCount	Integer	Количество комментариев к данному сообщению

Сокращение объема данных

Для ускорения экспериментов не стоит проверять идеи на всем гигантском XML-файле. Нужно подумать, как извлечь из него репрезентативную выборку, позволяющую все же решить, хороша идея или нет. Если оставить только элементы `row` с датой создания в 2012 году, то получится больше 6 миллионов сообщений (2 323 184 вопросов и 4 055 999 ответов), этого достаточно для выборки обучающих данных. Кроме того, мы не хотим работать с форматом XML, поскольку он тоже замедляет обработку. Чем формат проще, тем лучше. Поэтому мы разберем оставшуюся часть XML-файла с помощью класса Python `cElementTree` и создадим файл с полями, разделенными знаками табуляции.

Предварительная выборка и обработка атрибутов

Чтобы еще больше сократить объем данных, мы можем просто опустить атрибуты, которые, как нам кажется, помогут классификатору

отличить хорошие ответы от плохих. Но тут важно не переусердствовать. Некоторые атрибуты напрямую не влияют на классификацию, но сохранить их тем не менее необходимо.

Так, атрибут `PostTypeId` нужен, чтобы отличить вопрос от ответа. Как признак он не представляет ценности, но необходим для фильтрации данных.

Атрибут `CreationDate` может представлять интерес для определения промежутка времени между вопросом и ответом, поэтому оставим его. Атрибут `Score`, безусловно, важен – это индикатор мнения сообщества.

А вот атрибут `ViewCount` для нас, скорее всего, бесполезен. Даже если бы его и можно было использовать для различения хороших и плохих ответов, эта информация недоступна в момент отправки ответа. В топку его!

Понятно, что атрибут `Body` содержит самую ценную информацию. Поскольку он представлен в HTML-кодированном виде, нам нужно будет декодировать его и получить простой текст.

Атрибут `OwnerUserId` был бы полезен, только если бы мы собирались принимать в расчет признаки, зависящие от пользователя, чего мы делать не станем. Сейчас мы его отбросим, но рекомендуем поискать способы его использования для создания более качественного классификатора (быть может, совместно с данными из файла `stackoverflow.com-Users.7z`).

Атрибут `Title` также игнорируется, хотя он мог бы сообщить дополнительные сведения о вопросе.

Игнорируется и атрибут `CommentCount`. Как и `ViewCount`, он мог бы помочь при классификации сообщений, существующих уже достаточно долго (больше комментариев = менее понятный ответ?). Но в момент отправки ответа он ничем не поможет.

Атрибут `AcceptedAnswerId` аналогичен `Score` в том смысле, что является показателем качества сообщения. Поскольку он будет нужен нам в каждом ответе, то вместо сохранения этого атрибута мы заведем новый – `IsAccepted`, равный 0 или 1 для ответов и игнорируемый для вопросов (если `ParentId=-1`).

В итоге получается такой формат:

```
Id <TAB> ParentId <TAB> IsAccepted <TAB> TimeToAnswer <TAB> Score
<TAB> Text
```

Конкретные детали разбора смотрите в файлах `so_xml_to_tsv.py` и `choose_instance.py`. Скажем лишь, что для ускорения обработки

мы распределяем данные по двум файлам. В файле `meta.json` хранится словарь в формате JSON, отображающий идентификатор сообщения на прочие данные, кроме `Text`. Например, оценку сообщения можно получить в виде `meta[Id]['Score']`. В файле `data.tsv` хранятся значения атрибутов `Id` и `Text`, который легко прочитать таким методом:

```
def fetch_posts():
  for line in open("data.tsv", "r"):
     post_id, text = line.split("\t")
     yield int(post_id), text.strip()
```

Что считать хорошим ответом?

Прежде чем обучить классификатор отличать хорошие ответы от плохих, нужно создать обучающие данные. Пока что у нас есть только сами данные, а предстоит еще снабдить их метками.

Можно, конечно, просто использовать в качестве метки атрибут `IsAccepted`. Ведь именно так помечается правильный ответ на вопрос. Однако это всего лишь мнение автора вопроса. Естественно, спрашивающему нужен ответ как можно быстрее, и он принимает первый понравившийся ему ответ. Но со временем могут быть поданы и другие ответы, причем некоторые из них могут быть лучше уже принятого. Но автор вопроса редко возвращается к своему вопросу, чтобы изменить свое мнение. Поэтому образуется много вопросов, для которых принятым является вовсе не ответ с наивысшей оценкой.

Другая крайность – взять ответы с самой высокой и самой низкой оценкой как положительный и отрицательный примеры соответственно. Но что делать с вопросами, на которые есть только хорошие ответы, скажем один, заслуживший два балла, и другой с четырьмя баллами? Действительно ли следует считать ответ с двумя баллами отрицательным примером только потому, что у него оказалась самая низкая оценка?

Нужно найти что-то среднее. Если считать все ответы с оценками, большими нуля, положительными примерами, а ответы с оценками, меньшими или равными нулю, – отрицательными, то получаются вполне разумные метки:

```
>>> all_answers = [q for q,v in meta.items() if v['ParentId']!=-1]
>>> Y = np.asarray([meta[answerId]['Score']>0 for answerId in
all_answers])
```

Создание первого классификатора

Начнем с простого и элегантного метода ближайших соседей, рассмотренного в предыдущей главе. Хотя и не такой продвинутый, как другие методы, он все же весьма эффективен: поскольку он не основан на модели, обучить его можно практически по любым данным. Но у этой элегантности есть и оборотная сторона, с которой мы очень скоро познакомимся.

Метод k ближайших соседей

На этот раз мы не станем реализовывать его сами, а возьмем из библиотеки `sklearn`. Классификатор находится в пакете `sklearn.neighbors`. Начнем с простого классификатора по двум ближайшим соседям:

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=2)
>>> print(knn)
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski', n_neighbors=2, p=2, weights='uniform')
```

Интерфейс этого классификатора такой же, как у всех других прогностических моделей в `sklearn`: обучаем с помощью метода `fit()`, после чего можем предсказывать класс новых образцов методом `predict()`:

```
>>> knn.fit([[1],[2],[3],[4],[5],[6]], [0,0,0,1,1,1])
>>> knn.predict(1.5)
array([0])
>>> knn.predict(37)
array([1])
>>> knn.predict(3)
array([0])
```

Для получения вероятностей классов служит метод `predict_proba()`. В случае двух классов, 0 и 1, он вернет массив из двух элементов:

```
>>> knn.predict_proba(1.5)
array([[ 1., 0.]])
>>> knn.predict_proba(37)
array([[ 0., 1.]])
>>> knn.predict_proba(3.5)
array([[ 0.5, 0.5]])
```

Подготовка признаков

Так какими признаками мы можем снабдить наш классификатор? У каких предположительно будет максимальная различающая способность?

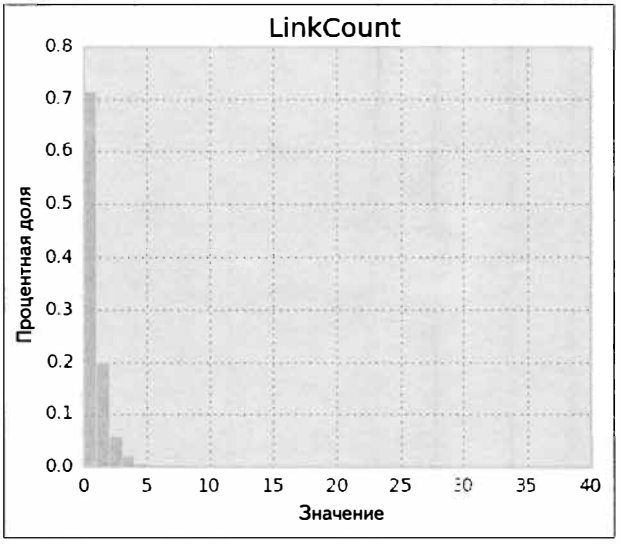
Атрибут `TimeToAnswer` уже имеется в нашем словаре `meta`, но маловероятно, что сам по себе он представляет большую ценность. Есть еще атрибут `Text`, но просто так мы не можем передать его классификатору, потому что признаки должны быть числовыми. Необходимо проделать грязную (и увлекательную!) работу по выделению признаков из текста.

В качестве одной из характеристик качества можно было бы взять число гиперссылок в ответе. В основе этого решения лежит предположение о том, что чем больше гиперссылок, тем тщательнее продуман ответ и тем выше шансы, что за него проголосуют. Конечно, учитывать нужно только ссылки в обычном тексте, а не в примерах кода:

```
import re

code_match = re.compile('<pre>(.*?)</pre>',
                        re.MULTILINE | re.DOTALL)
link_match = re.compile('<a href="http://.*?".*?>(.*?)</a>',
                        re.MULTILINE | re.DOTALL)
tag_match = re.compile('<[^>]*>',
                        re.MULTILINE | re.DOTALL)

def extract_features_from_body(s):
  link_count_in_code = 0
  # подсчитываем число ссылок в коде, чтобы потом вычесть его
  for match_str in code_match.findall(s):
     link_count_in_code += len(link_match.findall(match_str))

  return len(link_match.findall(s)) - link_count_in_code
```

В производственной системе не следует разбирать HTML-код с помощью регулярных выражений. Есть отличные библиотеки, например BeautifulSoup, которые прекрасно справляются с различными странностями, которыми изобилует типичная HTML-разметка.

Это позволяет сгенерировать один признак для каждого ответа. Но прежде чем приступать к обучению классификатора, разберемся, на каких данных мы будем его обучать. Первое представление можно составить на основании распределения частот нашего признака. Для этого нужно построить график процентной встречаемости каждого значения в данных. Вот он:



Оказывается, что в большинстве сообщений нет ни одной ссылки, так что, имея только этот признак, хороший классификатор не обучить. Тем не менее, попробуем – и посмотрим, что получится.

Обучение классификатора

Чтобы получить классификатор, мы должны передать обучающему алгоритму *kNN* (*k* ближайших соседей) массивы признаков *X* и сопоставленных им меток Y:

```
X = np.asarray([extract_features_from_body(text) for post_id, text in
                fetch_posts() if post_id in all_answers])
knn = neighbors.KNeighborsClassifier()
knn.fit(X, Y)
```

Поскольку использовались параметры по умолчанию, то получился обученный классификатор 5NN (то есть по 5 ближайшим соседям). Почему именно 5NN? Потому что при нынешнем состоянии знаний о данных мы понятия не имеем, каким должно быть *k*. Когда узнаем побольше, тогда и решим, какое взять значение k.

Измерение качества классификатора

Нужно точно решить, что именно мы собираемся измерять. Наивный, но самый простой способ – измерить среднее качество пред-

сказания по всему тестовому набору. Получится число между 0 (все предсказания неверны) и 1 (все предсказания правильны). Верность можно получить с помощью метода `knn.score()`.

Но, как было сказано ранее, недостаточно сделать это один раз, нужно применить перекрестную проверку, воспользовавшись готовым классом `KFold` из модуля `sklearn.cross_validation`. Затем мы смотрим, насколько расходятся средние оценки, полученные на каждом проходе, в терминах стандартного отклонения.

```
from sklearn.cross_validation import KFold
scores = []

cv = KFold(n=len(X), k=10, indices=True)

for train, test in cv:
  X_train, y_train = X[train], Y[train]
  X_test, y_test = X[test], Y[test]
  clf = neighbors.KNeighborsClassifier()
  clf.fit(X, Y)
  scores.append(clf.score(X_test, y_test))

print("Mean(scores)=%.5f\tStddev(scores)=%.5f"\
      %(np.mean(scores), np.std(scores)))
```

Получается такой результат:

```
Mean(scores)=0.50250 Stddev(scores)=0.055591
```

Никуда не годится. Верность 55% немногим лучше подбрасывания монетки. Очевидно, количество ссылок в сообщении – не лучший индикатор качества ответа. То есть различающая способность этого признака мала – по крайней мере, для классификатора *kNN* с `k=5`.

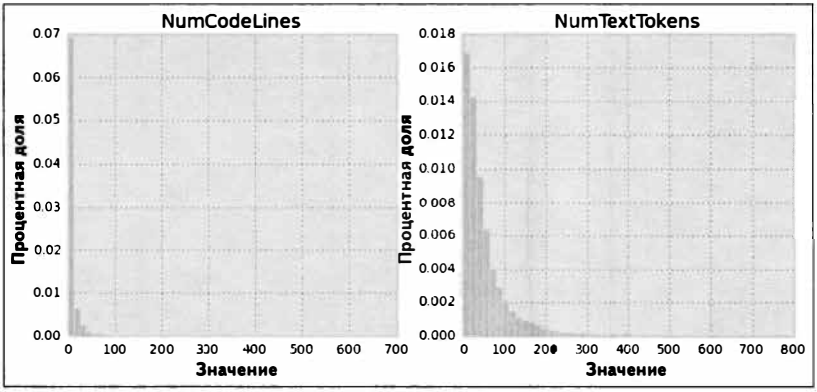
Проектирование дополнительных признаков

В дополнение к числу гиперссылок неплохим показателем может оказаться число строк кода в ответе. По крайней мере, для автора вопроса это, скорее всего, важно. Фрагменты кода находятся между тегами `<pre>` и `</pre>`. А после выделения кода подсчитаем число слов в сообщении без учета слов в коде.

```
def extract_features_from_body(s):
  num_code_lines = 0
  link_count_in_code = 0
  code_free_s = s

  # извлечь исходный код и подсчитать число строк
```

```
    for match_str in code_match.findall(s):
        num_code_lines += match_str.count('\n')
        code_free_s = code_match.sub("", code_free_s)

        # Иногда в исходном коде встречаются ссылки, которые
        # мы не хотим учитывать
        link_count_in_code += len(link_match.findall(match_str))

    links = link_match.findall(s)
    link_count = len(links)
    link_count -= link_count_in_code
    html_free_s = re.sub(" +", " ",
                         tag_match.sub('', code_free_s)).replace("\n", "")
    link_free_s = html_free_s

    # перед подсчетом слов удалить ссылки из текста
    for link in links:
        if link.lower().startswith("http://"):
            link_free_s = link_free_s.replace(link,'')

    num_text_tokens = html_free_s.count(" ")

    return num_text_tokens, num_code_lines, link_count
```

Из графика видно, что, по крайней мере, число слов в сообщении варьируется в более широких пределах:



Обучение на большем пространстве признаков несколько повышает верность:

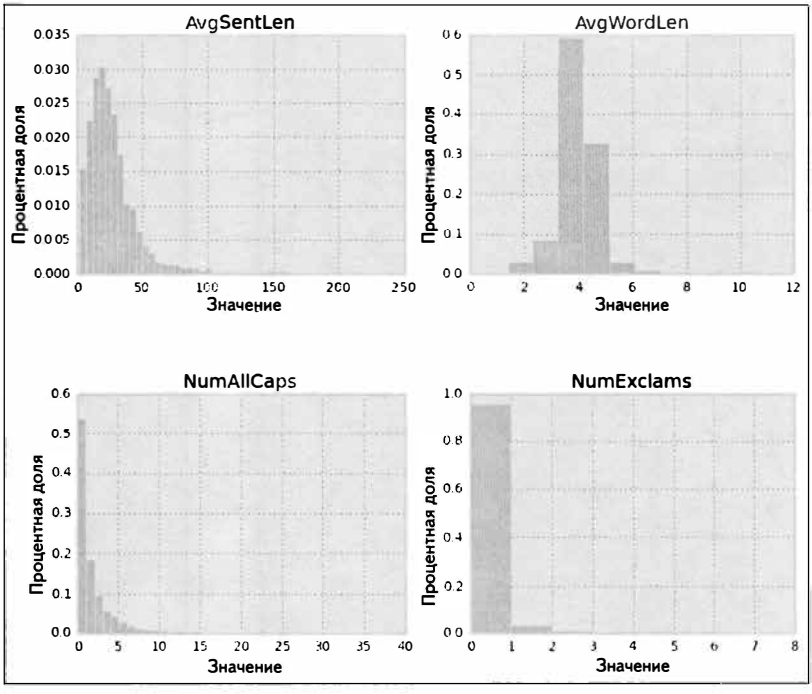
```
Mean(scores)=0.59800 Stddev(scores)=0.02600
```

Конечно, это означает, что 4 из 10 предсказаний будут неверны. Но мы хотя бы движемся в правильном направлении. Чем больше при-

знаков, тем выше верность, а, значит, надо добавить новые признаки. Возьмем такие признаки.

- `AvgSentLen`: среднее число слов в предложении. Быть может, особенно хорошими считаются ответы, которые не перегружают мозги читателя слишком длинными предложениями?
- `AvgWordLen`: среднее число символов в словах, составляющих сообщение.
- `NumAllCaps`: число слов, записанных только заглавными буквами; это считается дурным тоном.
- `NumExclams`: число восклицательных знаков.

На графиках ниже показаны распределения средних длин предложения и слова, количества слов, записанных заглавными буквами, и количества восклицательных знаков.



Теперь каждое сообщение характеризуется семью признаками. Посмотрим, чего мы добились:

```
Mean(scores)=0.61400 Stddev(scores)= 0.02154
```

Любопытно. Мы добавили четыре признака и ничего не получили взамен. С чего бы это?

Чтобы понять, в чем дело, нужно вспомнить, как работает классификатор *kNN*. Наш классификатор 5NN определяет класс нового сообщения, вычисляя семь признаков: `LinkCount`, `NumTextTokens`, `NumCodeLines`, `AvgSentLen`, `AvgWordLen`, `NumAllCaps` и `NumExclams`, после чего находит пять ближайших сообщений. В качестве класса нового сообщения берется класс, к которому относится большинство ближайших соседей. А близость определяется путем вычисления евклидова расстояния (поскольку мы не сказали ничего другого, принято значение по умолчанию `p=2`, где `p` – параметр метрики Минковского). Это означает, что все семь признаков считаются равноправными. В частности, *kNN* не понял, что признак `NumTextTokens` хорош, но гораздо менее важен, чем `NumLinks`. Рассмотрим два сообщения *A* и *B*, отличающиеся только показанными ниже признаками, и сравним их с новым сообщением:

Сообщение	NumLinks	NumTextTokens
A	2	20
B	0	25
новое	1	23

Мы считаем, что ссылки ценнее обычного текста, однако классификатор думает, что сообщение *B* больше похоже на новое, чем *A*.

Очевидно, у классификатора *kNN* имеются трудности в части правильного использования доступных данных.

Как поправить дело?

Для улучшения результатов у нас есть следующие пути.

- **Добавить еще данные.** Быть может, данных просто недостаточно для обучения алгоритма и нужно добавить еще?
- **Поэкспериментировать со сложностью модели.** Быть может, модель недостаточно сложна? Или наоборот слишком сложна? В таком случае следует уменьшить *k*, чтобы модель принимать в расчет меньше соседей и, следовательно, давала лучшие предсказания на неравномерно распределенных данных. Или увеличить *k* для достижения противоположного результата.
- **Изменить пространство признаков.** Быть может, мы выбрали не те признаки? Можно было бы, к примеру, изменить шка-

лу уже отобранных признаков или придумать какие-то новые. А, возможно, следует исключить часть текущих признаков, потому что они, по существу, характеризуют одно и то же.

- **Изменить модель.** Быть может, модель *kNN* не годится для нашего случая, и с ее помощью никогда не удастся достичь хорошего качества предсказаний, вне зависимости от сложности и изощренности пространства признаков?

На практике, дойдя до этой развилки, исследователь зачастую пытается улучшить качество наугад, без какой-то системы выбирая тот или иной вариант в надежде случайно наткнуться на идеальную конфигурацию. Мы тоже могли бы поступить таким образом, но это, без сомнения, заняло бы больше времени, чем взвешенные действия на основе знаний. Поэтому пойдем по второму пути, для чего придется сначала познакомиться с дилеммой смещения-дисперсии.

Дилемма смещения-дисперсии

В главе 1 мы пытались подобрать полиномы различной сложности, определяемой степенью d, для аппроксимации данных. Мы поняли, что линейная функция не годится, потому что сама природа данных нелинейна. И как бы мы ни изощрялись, двумерная модель неизменно будет видеть прямую линию. Мы говорим, что у модели слишком высокое смещение в применении к имеющимся данным. Она недообучена.

Мы продолжили экспериментировать с размерностью и обнаружили, что 100-мерный полином слишком близко подгоняется к данным, на которых был обучен (тогда мы еще не знали о разделении данных на обучающие и тестовые). Мы поняли, что модель так сильно переобучена, что, выбирая разные подмножества имеющихся данных, мы будем получать совершенно разные полиномы. Мы говорим, что у модели слишком высокая дисперсия в применении к имеющимся данным. Она переобучена.

Это две крайности, между которыми располагается большинство задач машинного обучения. В идеале хотелось бы иметь одновременно низкое смещение и низкую дисперсию. Но мир далек от совершенства, поэтому приходится искать компромисс. Улучшая одно, мы обычно ухудшаем другое.

Устранение высокого смещения

Предположим, что модель страдает от высокого смещения. В таком случае добавление новых обучающих данных не поможет. И уж

точно не поможет исключение признаков, потому что модель и так чрезмерно проста.

Остается только три возможности: увеличить число признаков, усложнить модель или изменить ее.

Устранение высокой дисперсии

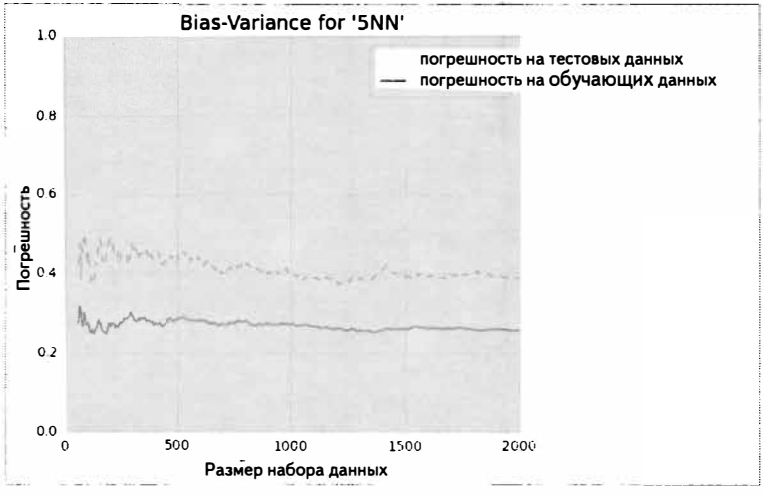
Напротив, если для модели характерна высокая дисперсия, значит, она слишком сложна для имеющихся данных. В таком случае нужно либо добыть дополнительные данные, либо уменьшить сложность. Иными словами, увеличить k, приняв во внимание больше соседей, или исключить часть признаков.

Низкое или высокое смещение?

Чтобы понять, в чем на самом деле заключается проблема, достаточно построить графики зависимости погрешностей на обучающих и тестовых данных от размера набора.

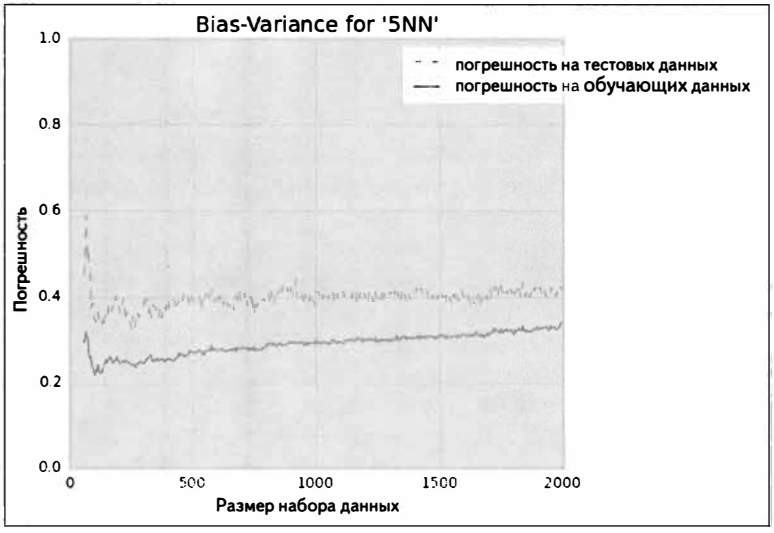
Для высокого смещения характерно небольшое убывание погрешности на тестовых данных в начале с дальнейшим выходом на очень высокое плато, приближающееся к погрешности на обучающих данных по мере роста размера набора. Для высокой дисперсии характерен большой разрыв между двумя кривыми.

Построив графики погрешностей для классификатора 5NN, мы увидим значительный разрыв между погрешностями на обучающих и тестовых данных, указывающий на наличие высокой дисперсии:



Взглянув на график, мы сразу понимаем, что добавление обучающих данных не поможет, поскольку пунктирная линия, соответствующая погрешности на тестовых данных, похоже, устойчиво проходит выше 0,4. Единственный шанс – уменьшить сложность – либо увеличив *k*, либо сократив пространство признаков.

Но здесь сокращение пространства признаков не помогает. В этом легко убедиться, построив графики для пространства признаков, состоящего только из признаков `LinkCount` и `NumTextTokens`:



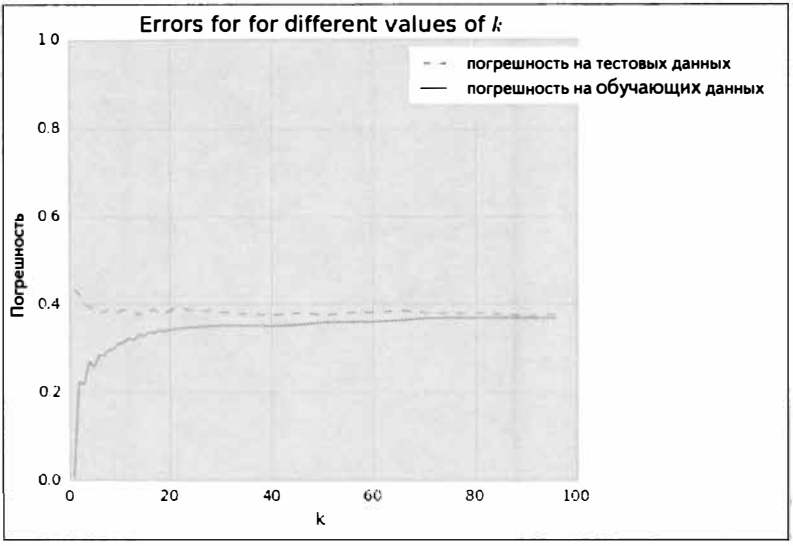
Похожие графики получаются и при исключении других признаков. Какое бы подмножество признаков ни взять, характер графика не меняется.

Но хотя бы уменьшение сложности модели за счет увеличения *k* дает какой-то положительный эффект:

k	mean(scores)	stddev(scores)
40	0.62800	0.03750
10	0.62000	0.04111
5	0.61400	0.02154

Однако он недостаточен и за него приходится расплачиваться увеличением времени классификации. Возьмем, к примеру, значение `k=40`, при котором погрешность на тестовых данных очень низкая.

Для классификации нового сообщения нам пришлось бы найти 40 ближайших сообщений:



Очевидно, в нашем случае классификация по ближайшим соседям оказывается проблематичной. И у нее есть еще один недостаток. Со временем количество сообщений в системе будет только возрастать. А поскольку метод ближайших соседей основан на образцах, то все сообщения придется хранить в системе. Чем их больше, тем медленнее будет работать система. В этом отличие от систем на основе моделей, в которых мы пытаемся построить модель, исходя из данных.

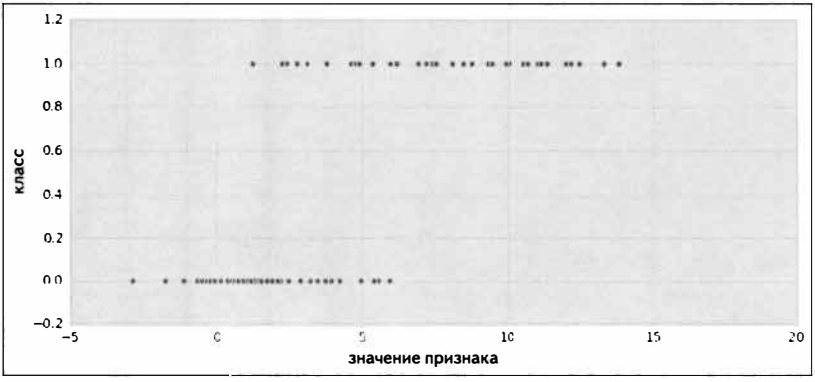
Итак, у нас набралось достаточно причин отказаться от метода ближайших соседей и поискать какой-нибудь другой классификатор. Конечно, не исключено, что существует какой-то чудо-признак, о котором мы не догадались. Но знать этого нам не дано, поэтому перейдем к другому алгоритму, который заведомо хорошо справляется с классификацией текстов.

Логистическая регрессия

Несмотря на название, логистическая регрессия – это метод классификации. Очень эффективный метод, когда требуется классифицировать тексты; а достигается это посредством выполнения регрессии для логистической функции – отсюда и название.

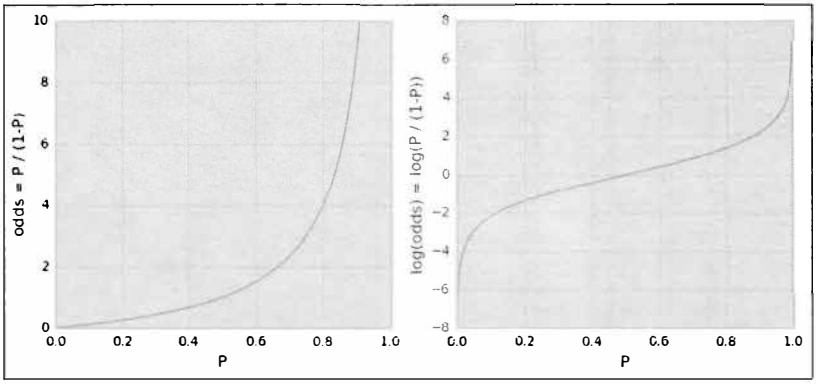
Немного математики на простом примере

Чтобы понять, как работает логистическая регрессия, рассмотрим следующий пример, в котором на график нанесены значения гипотетического признака X и соответствующие классы, 0 и 1. Как видим, данные зашумлены, так что классы перекрываются в диапазоне значений признака от 1 до 6. Поэтому лучше моделировать не сами дискретные классы, а вероятность того, что значение признака принадлежит классу 1, – $P(X)$. Имея такую модель, мы сможем сказать, что образец принадлежит классу 1, если $P(X) > 0.5$, и классу 0 в противном случае.



С точки зрения математики, всегда трудно моделировать величину, имеющую конечную область значений, например, дискретные метки 0 и 1. Но можно ввести в рассмотрение вероятности, которые всегда находятся между 0 и 1. Для этого нам потребуется отношение шансов и его логарифм.

Допустим, вероятность того, что признак принадлежит классу 1, равна 0.9 – $P(y{=}1) = 0.9$. Тогда отношение шансов определяется как $P(y{=}1)/P(y{=}0) = 0.9/0.1 = 9$. Можно сказать, что шансы признака попасть в класс 1 равны 9:1. Если $P(y{=}0.5)$, то шансы попасть в класс 1 равны 1:1. Отношение шансов снизу ограничено 0, а сверху бесконечностью (левый из двух графиков ниже). Если взять его логарифм, то можно будет отобразить множество вероятностей в диапазоне от 0 до 1 на всю вещественную ось от минус до плюс бесконечности (правый из двух графиков). Важно, что это отображение монотонно – чем больше вероятность, тем больше логарифм отношения шансов, – однако мы уже не ограничены только значениями 0 или 1.



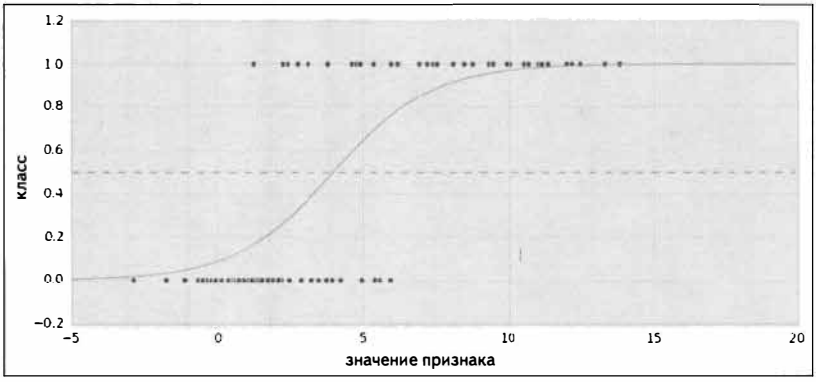
Это означает, что теперь мы можем подогнать линейные комбинации наших признаков (да, сейчас признак только один, к тому же константный, но это скоро изменится) под значения *log(odds)*. То есть мы заменяем линейную функцию $y_i = c_0 + c_1 x_i$ функцией $log\left(\frac{p_i}{1-p_i}\right) = c_0 + c_1 x$ (подставляя *log(odds)* вместо y). Решая это уравнение относительно p_i, получаем $p_i = \frac{1}{1+e^{-(c_0+c_1 x_i)}}$.

Нам нужно только подобрать коэффициенты, так чтобы эта формула давала наименьшие погрешности для всех пар (x_i, p_i) из нашего набора. Но как раз это умеет делать библиотека scikit-learn. После того как коэффициенты найдены, мы можем по этой формуле вычислить вероятность принадлежности любого нового образца классу 1.

```
>>> from sklearn.linear_model import LogisticRegression
>>> clf = LogisticRegression()
>>> print(clf)
LogisticRegression(C=1.0, class_weight=None, dual=False,
fit_intercept=True, intercept_scaling=1, penalty=l2, tol=0.0001)
>>> clf.fit(X, y)
>>> print(np.exp(clf.intercept_), np.exp(clf.coef_.ravel()))
[ 0.09437188] [ 1.80094112]
>>> def lr_model(clf, X):
...   return 1 / (1 + np.exp(-(clf.intercept_ + clf.coef_*X)))
>>> print("P(x=-1)=%.2f\tP(x=7)=%.2f"%(lr_model(clf, -1),
lr_model(clf, 7)))
P(x=-1)=0.05 P(x=7)=0.85
```

Вероятно, вы обратили внимание, что scikit-learn возвращает первый коэффициент – свободный член – в специальном поле `intercept_`.

Изобразив найденную модель на графике, мы увидим, что она отлично согласуется с данными.



Применение логистической регрессии к задаче классификации

Признаемся, что пример из предыдущего раздела специально выбран для демонстрации красоты логистической регрессии. Но как она поведет себя на реальных зашумленных данных?

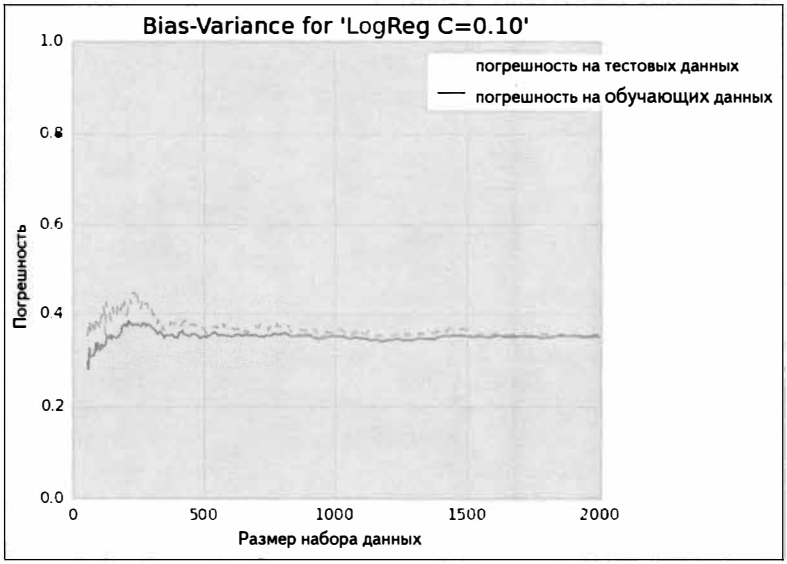
В сравнении с классификатором по ближайшим соседям (`k=40`) немного лучше, но кардинально ситуация не изменилась.

Метод	mean(scores)	stddev(scores)
LogReg C=0.1	0.64650	0.03139
LogReg C=1.00	0.64650	0.03155
LogReg C=10.00	0.64550	0.03102
LogReg C=0.01	0.63850	0.01950
40NN	0.62800	0.03750

Здесь показана верность для различных значений параметра регуляризации `C`. Он позволяет управлять сложностью модели, как параметр `k` для метода ближайших соседей. Чем меньше `C`, тем выше штраф за сложность модели.

Беглый взгляд на диаграмму смещения-дисперсии для варианта с `C=0.1` показывает, что у нашей модели высокое смещение – кривые

погрешностей на обучающих и тестовых данных близки, но остаются на недопустимо высоком уровне. Это означает, что алгоритм логистической регрессии при текущем пространстве признаков недообучен, поэтому модель не полностью отражает особенности данных.



И что теперь? Мы сменили модель, сделали для ее настройки все, что могли при нынешнем уровне знаний, но удовлетворительного классификатора так и не получили.

Все сильнее подозрение, что данные слишком зашумлены для этой задачи или что наш набор признаков не позволяет различать классы с достаточной уверенностью.

Не верностью единой – точность и полнота

Отступим назад и подумаем, чего мы хотим достичь. На самом деле, нам не нужен классификатор, который точно определяет хорошие и плохие ответы в терминах верности. Если бы мы смогли настроить классификатор, так чтобы он особенно хорошо предсказывал какой-то один класс, то сумели бы соответственно организовать обратную связь с пользователем. Если бы, например, классификатор всегда

правильно определял, что ответ плохой, то мы ничего не сообщали бы пользователю, пока классификатор не скажет: «Это плохой ответ». Наоборот, если бы классификатор гениально предсказывал хорошие ответы, то мы могли бы показывать полезные подсказки в начале и убирать их, когда классификатор признает ответ хорошим.

Чтобы узнать, в какой ситуации мы находимся, понадобится умение измерять точность и полноту. А понять, что это такое, нам поможет следующая таблица, в которой приведены четыре результата классификации.

		Классифицирован как	
		Положительный	**Отрицательный**
На самом деле	Положительный	Истинно положительный (TP)	**Ложноотрицательный (FN)**
	Отрицательный	Ложноположительный (FP)	**Истинно отрицательный (TN)**

Например, если классификатор определяет, что образец положительный, и он действительно положительный, то имеет место истинно положительный результат. С другой стороны, если классификатор определяет, что образец отрицательный, хотя на самом деле он положительный, то имеет место ложноотрицательный результат.

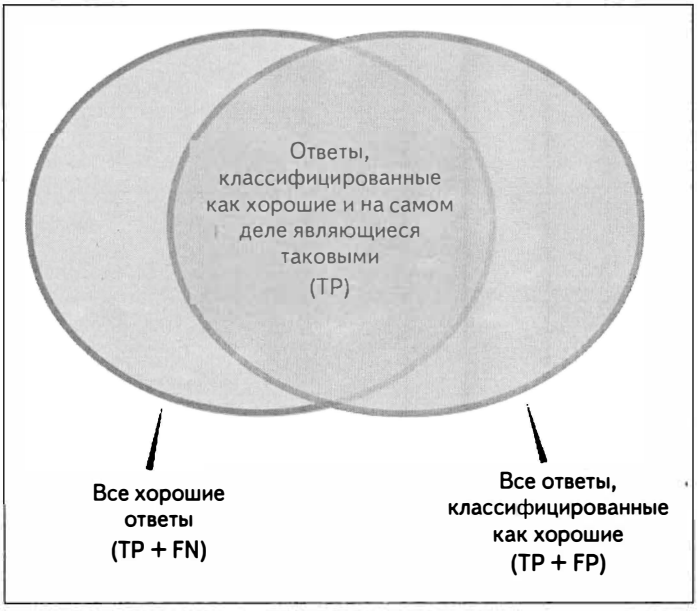
Нам нужно, чтобы была высока частота правильного предсказания хотя бы одного класса, необязательно обоих. То есть требуется, чтобы количество истинно положительных результатов было максимально. Это свойство характеризуется точностью:

$$Точность = \frac{TP}{TP + FP}$$

Если бы, напротив, мы хотели правильно определять как можно больше плохих или хороших ответов, то нас интересовала бы полнота:

$$Полнота = \frac{TP}{TP + FN}$$

На рисунке ниже точность – это отношение площади пересечения к площади правого эллипса, а полнота – это отношение площади пересечения к площади левого эллипса.

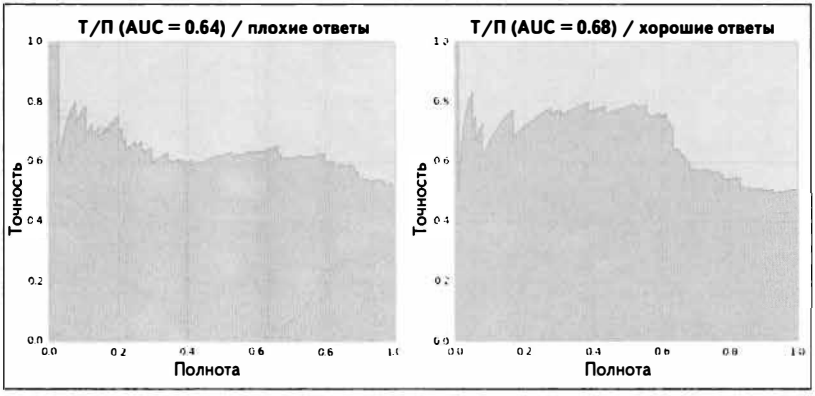


Ну и как же нам максимизировать точность? До сих пор в качестве порога, отличающего хороший ответ от плохого, мы брали 0.5. А теперь мы можем вычислить TP, FP и FN, меняя порог в диапазоне от 0 до 1. Зная эти величины, мы затем сможем построить график зависимости точности от полноты.

Все вычисления за нас проделает функция `precision_recall_curve()` из модуля metrics:

```
>>> from sklearn.metrics import precision_recall_curve
>>> precision, recall, thresholds = precision_recall_curve(y_test,
    clf.predict(X_test))
```

Приемлемое качество предсказания одного класса не означает, что и другой класс предсказывается столь же хорошо. Это видно из следующих двух графиков, где изображены кривые зависимости точности от полноты для классификации плохих (левый график) и хороших (правый график) ответов.



В эти графики мы включили также гораздо лучшую характеристику качества классификатора – **площадь под кривой** (AUC). Ее можно интерпретировать как среднюю точность классификатора. Для сравнения разных классификаторов она очень удобна.

Как видно, мы можем забыть о предсказании плохих ответов (левый график). Точность высока только при очень низкой полноте, а затем остается на неприемлемо низком уровне 60 процентов.

Что же касается хороших ответов, то мы можем получить точность выше 80% при полноте почти 40%. Найдем, какое для этого нужно пороговое значение. Во время обучения нескольких классификаторов на разных группах (вспомните о функции `KFold()`) нам нужно отобрать классификатор не слишком хороший и не слишком плохой, дающий реалистичный взгляд на вещи. Назовем его средним клоном:

```
>>> medium = np.argsort(scores)[int(len(scores) / 2)]
>>> thresholds = np.hstack(([0],thresholds[medium]))
>>> idx80 = precisions>=0.8
>>> print("P=%.2f R=%.2f thresh=%.2f" % (precision[idx80][0],
          recall[idx80][0], threshold[idx80][0]))
P=0.80 R=0.37 thresh=0.59
```

Задав порог `0.59`, мы сможем достичь точности определения хороших ответов 80% при низкой полноте 37%. Это означает, что мы будем распознавать только один из трех хороших ответов. Но уж эта треть будет действительно хорошей. Что до остальных, то мы можем вежливо предложить общие советы о том, как улучшить качество ответа.

Чтобы применить этот порог к процессу предсказания, нужно воспользоваться функцией `predict_proba()`, которая возвращает вероятности классов, а не функцией `predict()`, возвращающей сами классы.

```
>>> thresh80 = threshold[idx80][0]
>>> probs_for_good = clf.predict_proba(answer_features)[:,1]
>>> answer_class = probs_for_good>thresh80
```

Убедиться в том, что мы попадаем в желаемый диапазон точности и полноты, позволит функция `classification_report`:

```
>>> from sklearn.metrics import classification_report
>>> print(classification_report(y_test, clf.predict_proba [:,1]>0.63,
    target_names=['not accepted', 'accepted']))


              precision    recall  f1-score  support
not accepted       0.59      0.85      0.70      101
accepted           0.73      0.40      0.52       99
avg / total        0.66      0.63      0.61      200
```

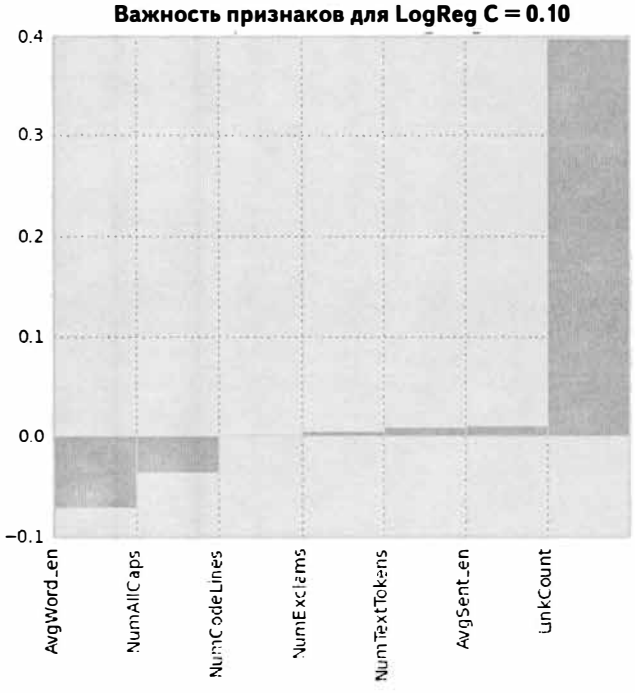
Отметим, что использование порога не гарантирует, что мы никогда не опустимся ниже той точности и полноты, для которых был определен порог.

Упрощение классификатора

Всегда имеет смысл посмотреть на фактический вклад отдельных признаков. В случае логистической регрессии мы можем получить представление о влиянии признаков прямо по найденным в процессе обучения коэффициентам (`clf.coef_`). Чем выше коэффициент при признаке, тем большую роль этот признак играет в определении качества сообщения. А отрицательные коэффициенты означают, что высокое значение соответствующего признака свидетельствует о том, что ответ плохой.

Как видно на рисунке ниже, наибольшее влияние на решение классификатора оказывают признаки `LinkCount`, `AvgWordLen`, `NumAllCaps` и `NumExclams`, тогда как `NumImages` (признак, который мы ввели в последнюю секунду просто для демонстрации) и `AvgSentLen` играют куда меньшую роль. В целом важность признаков согласуется с интуицией, но тот факт, что `NumImages` практически игнорируется, вызывает

удивление. Обычно ответы, содержащие картинки, котируются высоко. Но на практике ответов, содержащих картинки, очень мало. То есть в принципе признак очень полезный, но настолько редкий, что реальной ценностью не обладает. Его можно спокойно опустить – на качестве классификации это не отразится.



К поставке готов!

Допустим, мы хотим интегрировать этот классификатор с сайтом. Чего нам точно не надо, так это повторного обучения всякий раз, как запускается служба классификации. А надо сериализовать классификатор после обучения и затем десериализовать в процессе развертывания на сайте:

```
>>> import pickle
>>> pickle.dump(clf, open("logreg.dat", "w"))
>>> clf = pickle.load(open("logreg.dat", "r"))
```

Ну вот, теперь классификатор готов к работе, как будто его только обучили. Принимайте поздравления!

Резюме

Мы сделали это! Для очень зашумленного набора данных мы построили классификатор, который решает поставленную задачу хотя бы частично. Правда, нам пришлось пойти на поводу у прагматики и немного изменить первоначальную цель, иначе она осталась бы недостижимой. Но попутно мы узнали о сильных и слабых сторонах алгоритма ближайших соседей и логистической регрессии. Мы научились выделять такие признаки, как `LinkCount`, `NumTextTokens`, `NumCodeLines`, `AvgSentLen`, `AvgWordLen`, `NumAllCaps`, `NumExclams` и `NumImages`, и анализировать их влияние на качество классификации.

Но еще ценнее тот факт, что мы поняли, как целенаправленно отлаживать плохо работающие классификаторы. В будущем этом позволит доводить системы до работоспособного состояния гораздо быстрее.

Познакомившись с методом ближайших соседей и логистической регрессией, в следующей главе мы рассмотрим еще один простой, но эффективный алгоритм классификации: наивный байесовский. И по ходу дела изучим дополнительные средства из библиотеки scikit-learn.

ГЛАВА 6. Классификация II – анализ эмоциональной окраски

Для компаний жизненно важно внимательно следить за тем, как общество реагирует на такие значимые события, как вывод на рынок нового продукта или выпуск пресс-релиза. В наши дни, когда так легко в реальном времени читать сообщения пользователей в Твиттере, стало возможно классифицировать твиты по эмоциональной окраске. Эта технология, которую иногда называют также «добычей мнений», находится на переднем крае исследований, и несколько компаний уже продают подобные услуги. Ну а раз существует спрос, то у нас есть все основания потренировать подросшие в предыдущей главе мускулы, разработав собственный классификатор эмоций.

План действий

Анализировать эмоциональную окраску твитов особенно сложно из-за ограничения на размер твита – всего 140 символов. Отсюда и специальный синтаксис, и нестандартные аббревиатуры и неправильно построенные предложения. Типичный подход – грамматический анализ предложений, агрегирование эмоциональной информации по абзацам и, наконец, вычисление общей эмоциональной окраски документа – здесь не годится.

Понятно, что мы не ставим целью создание высококачественного классификатора эмоций. Наша задача скромнее:

- воспользоваться этим примером для знакомства еще с одним алгоритмом классификации, **наивным байесовским**;
- объяснить, как работает **частеречная разметка** и чем она нам поможет;
- продемонстрировать новые приемы работы с библиотекой scikit-learn, которые иногда оказываются полезными.

Чтение данных из Твиттера

Естественно, нам понадобятся твиты и ассоциированные с ними метки, описывающие эмоциональную окраску: положительную, отрицательную или нейтральную. В этой главе мы будем работать с корпусом из более 5000 твитов, который вручную разметил Ниек Сандерс (Niek Sanders); большое ему спасибо за любезное разрешение воспользоваться плодами этого гигантского труда.

Чтобы не нарушать условия обслуживания, мы не станем в книге приводить ни сами твиты, ни какие-либо данные, полученные из Твиттера, а будем пользоваться размеченными Сандерсом данными, в которых указаны идентификатор твита и проставленная вручную эмоциональная окраска, а также показанным ниже скриптом `install.py`, который загружает соответствующие твиты. Поскольку скрипт стремится не перегружать серверы Твиттера, для скачивания более 5000 твитов потребуется некоторое время. Так что запускайте его прямо сейчас.

Данные сопровождаются одной из четырех меток эмоциональной окраски:

```
>>> X, Y = load_sanders_data()
>>> classes = np.unique(Y)
>>> for c in classes: print("#%s: %i" % (c, sum(Y==c)))
#irrelevant: 490
#negative: 487
#neutral: 1952
#positive: 433
```

В функции `load_sanders_data()` мы объединяем нерелевантные и нейтральные метки в одну – нейтральную – и отбрасываем твиты на любых языках, кроме английского, так что в итоге получается 3362 твита. Если у вас получились другие счетчики, то, быть может, за прошедшее время некоторые твиты были удалены или сделаны закрытыми. В таком случае числа и графики в последующих разделах будут выглядеть немного иначе.

Введение в наивный байесовский классификатор

Наивная байесовская классификация – пожалуй, один из самых элегантных практически используемых алгоритмов машинного обучения. Вопреки своему названию, он отнюдь не так наивен, если судить

по качеству результата. Он устойчив к нерелевантным признакам, которые попросту игнорирует. Он быстро обучается и быстро возвращает предсказание. Он потребляет не очень много памяти. Так почему же его называют наивным?

Слово «наивный» относится к предположению, необходимому для оптимальной работы байесовского классификатора. Состоит оно в том, что признаки не оказывают влияния друг на друга. В реальных приложениях так бывает редко. И, тем не менее, на практике верность этого алгоритма достаточно высока, даже если предположение о независимости признаков не оправдывается.

О теореме Байеса

По сути своей, наивная байесовская классификация – не что иное, как отслеживание того, какой признак о каком классе свидетельствует. Способ проектирования признаков определяет модель, используемую для обучения. В модели Бернулли допускаются только булевы признаки; встречается слово в твите один раз или несколько, не имеет значения. Напротив, в мультиномиальной модели признаками являются счетчики слов. Для простоты мы воспользуемся моделью Бернулли, чтобы объяснить, как наивный байесовский классификатор применяется для анализа эмоциональной окраски. А затем – для обучения и настройки реальных классификаторов – перейдем на мультиномиальную модель.

Введем следующие переменные:

Переменная	Назначение
C	Класс твита (положительный или отрицательный)
F_1	В твите хотя бы один раз встречается слово «awesome» (восхитительный)
F_2	В твите хотя бы один раз встречается слово «crazy» (чумовой или дурацкий)

В ходе обучения мы построили наивную байесовскую модель, которая возвращает вероятность класса C, если известны признаки F_1 и F_2. Эта вероятность записывается в виде $P(C|F_1, F_2)$.

Поскольку мы не можем оценить $P(C|F_1, F_2)$ непосредственно, то применяем формулу, изобретенную Байесом:

$$P(A) \cdot P(B|A) = P(B) \cdot P(A|B)$$

Если считать, что A – событие, состоящее во вхождении обоих слов «awesome» и «crazy», а B – принадлежность твита классу C, то получится формула, которая впоследствии поможет нам вычислить вероятность принадлежности образца указанному классу:

$$P(F_1, F_2) \cdot P(C|F_1, F_2) = P(C) \cdot P(F_1, F_2|C)$$

Это позволяет выразить $P(C|F_1, F_2)$ через другие вероятности:

$$P(C|F_1, F_2) = \frac{P(C) \cdot P(F_1, F_2|C)}{P(F_1, F_2)}$$

Можно записать это и в таком виде:

$$prior = \frac{posterior \cdot likelihood}{evidence}$$

prior и *evidence* найти легко:

- $P(C)$ – априорная вероятность класса без каких-либо знаний о данных. Оценить ее можно, напрямую подсчитав долю обучающих примеров, принадлежащих данному классу.
- $P(F_1, F_2)$ – свидетельство, или вероятность одновременного наличия признаков F_1 и F_2.

Нетривиальная часть – вычисление правдоподобия (likelihood) $P(F_1, F_2|C)$. Эта величина говорит о том, насколько вероятно увидеть признаки F_1 и F_2, если мы знаем, что образец принадлежит классу C. Для ее оценки нужно немного подумать.

Что значит быть наивным

Из теории вероятности известна следующая формула:

$$P(F_1, F_2|C) = P(F_1|C) \cdot P(F_2|C, F_1)$$

Но сама по себе она мало что дает, потому что мы заменяем одну трудную задачу (оценка $P(F_1, F_2|C)$) другой, не менее трудной (оценка $P(F_2|C, F_1)$).

Однако если наивно предположить, что F_1 и F_2 независимы, то $P(F_2|C, F_1)$ сводится к $P(F_2|C)$, и мы можем записать:

$$P(F_1, F_2|C) = P(F_1|C) \cdot P(F_2|C)$$

Собирая все вместе, получаем вполне удобоваримую формулу:

$$P(C|F_1, F_2) = \frac{P(C) \cdot P(F_1|C) \cdot P(F_2|C)}{P(F_1, F_2)}$$

Любопытная вещь: хотя теоретически неправильно выдвигать произвольные предположения под настроение, в данном случае такой подход на удивление хорошо работает в реальных задачах.

Использование наивного байесовского алгоритма для классификации

Получив новый твит, мы должны только вычислить вероятности:

$$P(C = \text{"pos"}|F_1, F_2) = \frac{P(C = \text{"pos"}) \cdot P(F_1|C = \text{"pos"}) \cdot P(F_2|C = \text{"pos"})}{P(F_1, F_2)}$$

$$P(C = \text{"neg"}|F_1, F_2) = \frac{P(C = \text{"neg"}) \cdot P(F_1|C = \text{"neg"}) \cdot P(F_2|C = \text{"neg"})}{P(F_1, F_2)}$$

А затем выбрать класс C_{best} с наибольшей вероятностью.

Поскольку для обоих классов знаменатель $P(F_1, F_2)$ один и тот же, мы можем его попросту игнорировать – предсказанный класс от этого не изменится.

Отметим, однако, что реальные вероятности больше не вычисляются. Вместо этого мы оцениваем, какой класс более правдоподобен, по имеющимся свидетельствам. Это еще одна причина устойчивости наивного байесовского классификатора: его интересуют не столько истинные вероятности, сколько информация о том, какой класс правдоподобнее. Короче говоря, можно написать:

$$C_{best} = \underset{c \in C}{\operatorname{argmax}} P(C = c) \cdot P(F_1|C = c) \cdot P(F_2|C = c)$$

Здесь говорится, что мы вычисляем часть после *argmax* для всех классов (в нашем случае *pos* и *neg*) и возвращаем тот класс, для которого получилось наибольшее значение.

Но в следующем примере давайте вернемся к истинным вероятностям и проделаем кое-какие вычисления, чтобы понаблюдать за работой наивного байесовского алгоритма. Для простоты будем предполагать, что Твиттер разрешает употреблять только два слова: «awesome» и «crazy» и что мы уже классифицировали вручную несколько твитов:

Твит	Класс
awesome	Положительный твит
awesome	Положительный твит
awesome crazy	Положительный твит
crazy	Положительный твит
crazy	Отрицательный твит
crazy	Отрицательный твит

В этом примере твит «crazy» получил как положительную, так и отрицательную оценку с целью промоделировать неоднозначность реальной речи (например, «балдеть от футбола» и «дурацкий идиот»). Всего у нас есть шесть твитов – четыре положительных и два отрицательных, поэтому получаются такие априорные вероятности:

$$P(C = "pos") = \frac{4}{6} \cong 0.67$$

$$P(C = "neg") = \frac{2}{6} \cong 0.33$$

Это означает, что, ничего не зная о самом твите, разумно будет предположить, что он положительный.

Пока что отсутствует вычисление $P(F_1|C)$ и $P(F_2|C)$ – вероятностей признаков F_1 и F_2 при условии класса C.

Они вычисляются как количество твитов, в которых встречался конкретный признак, поделенное на количество твитов, помеченных классом C. Вероятность встретить в твите слово «awesome», если известно, что класс положительный, вычисляется по формуле:

$$P(F_1=1|C="pos") = \frac{\textit{число положительных твитов, содержащих «awesome»}}{\textit{число всех положительных твитов}} = \frac{3}{4}$$

Поскольку из четырех положительных твитов три содержали слово «awesome».

Очевидно, что вероятность не встретить слово «awesome» в положительном твите равна:

$$P(F_1 = 0|C = "pos") = 1 - P(F_1 = 1|C = "pos") = 0.25$$

Точно так же производятся остальные вычисления (опущен случай, когда в твите не встречается слово):

$$P(F_2 = 1 | C = "pos") = \frac{2}{4} = 0.5$$

$$P(F_1 = 1 | C = "neg") = \frac{0}{2} = 0$$

$$P(F_2 = 1 | C = "neg") = \frac{2}{2} = 1$$

Для полноты картины вычислим также свидетельство, чтобы узнать истинные вероятности. Для двух конкретных значений F_1 и F_2 свидетельство вычисляется так:

$$P(F_1, F_2) = P(F_1, F_2 | C = "pos") \cdot P(C = "pos") + P(F_1, F_2 | C = "neg") \cdot P(C = "neg")$$

Это дает следующие значения:

$$P(F_1 = 1, F_2 = 1) = \frac{3}{4} \cdot \frac{2}{4} \cdot \frac{4}{6} + 0 \cdot 1 \cdot \frac{2}{6} \cdot = \frac{1}{4}$$

$$P(F_1 = 1, F_2 = 0) = \frac{3}{4} \cdot \frac{2}{4} \cdot \frac{4}{6} + 0 \cdot 0 \cdot \frac{2}{6} \cdot = \frac{1}{4}$$

$$P(F_1 = 0, F_2 = 1) = \frac{1}{4} \cdot \frac{2}{4} \cdot \frac{4}{6} + \frac{2}{2} \cdot \frac{2}{2} \cdot \frac{2}{6} \cdot = \frac{5}{15}$$

Теперь у нас есть все данные, необходимые для классификации новых твитов. Осталось лишь разобрать твит и породить на его основе признаки:

Твит	F_1	F_2	Вероятности классов	Классификация
«awesome»	1	0	$P(C = "pos" \mid F_1 = 1, F_2 = 0) = \frac{\frac{3}{4} \cdot \frac{2}{4} \cdot \frac{4}{6}}{\frac{1}{4}} = 1$ $P(C = "neg" \mid F_1 = 1, F_2 = 0) = \frac{\frac{0}{2} \cdot \frac{2}{2} \cdot \frac{2}{6}}{\frac{1}{4}} = 0$	Положительный

Твит	F_1	F_2	Вероятности классов	Классификация
«crazy»	0	1	$P(C="pos" \mid F_1=0, F_2=1)=\frac{\frac{1}{4}\cdot\frac{2}{4}\cdot\frac{4}{6}}{\frac{5}{12}}=\frac{1}{5}$ $P(C="neg" \mid F_1=0, F_2=1)=\frac{\frac{2}{2}\cdot\frac{2}{2}\cdot\frac{2}{6}}{\frac{5}{12}}=\frac{4}{5}$	Отрицательный
«awesome crazy»	1	1	$P(C="pos" \mid F_1=1, F_2=1)=\frac{\frac{3}{4}\cdot\frac{2}{4}\cdot\frac{4}{6}}{\frac{1}{4}}=1$ $P(C="neg" \mid F_1=1, F_2=1)=\frac{\frac{0}{2}\cdot\frac{2}{2}\cdot\frac{2}{6}}{\frac{1}{4}}=0$	Положительный

Пока всё хорошо. При классификации тривиальных твитов метки, похоже, назначаются правильно. Но остается вопрос, как быть со словами, которые не встречались в обучающем корпусе. Ведь по формуле выше всем новым словам будет присвоена нулевая вероятность.

Учет ранее не встречавшихся слов и другие тонкости

Вычисляя вероятности в предыдущем разделе, мы себя обманывали. Вычислялись не истинные вероятности, а лишь грубые приближения к ним. Мы предполагали, что обучающий корпус расскажет нам правду об истинных вероятностях. Но это не так. Очевидно, что корпус из шести твитов не может дать всю информацию о каждом когда-либо написанном твите. Например, без сомнения существуют твиты, содержащие слово «text». Просто мы их ни разу не видели. Следовательно, наше приближение очень грубое, и это нужно учитывать. На практике для этого часто применяется **сглаживание с прибавлением единицы** (add-one smoothing).

Сглаживание с прибавлением единицы иногда называют **аддитивным сглаживанием** или **сглаживанием Лапласа**. Отметим, что сглаживание Лапласа не имеет ничего общего с лапласовским сглаживанием, которое относится к сглаживанию полигональных сеток. Если прибавляется не 1, а настраиваемый параметр `alpha<0`, то говорят о сглаживании Лидстона.

Это очень простой прием, заключающийся в прибавлении единицы ко всем вхождениям признака. В его основе лежит предположение, что даже если мы не видели данного слова во всем корпусе, все равно есть шанс, что это только потому, что в нашей выборке твитов такого слова не оказалось. Поэтому, применяя сглаживание с прибавлением единицы, мы делаем вид, что видели каждое слово на один раз больше, чем в действительности. То есть вместо вычисления $P(F_1 = 1 | C = "pos") = \frac{3}{4} = 0.75$ мы вычисляем $P(F_1 = 1 | C = "pos") = \frac{3+1}{4+2} = 0.67$.

Почему в знаменателе прибавлено 2? Потому что всего у нас есть два признака: вхождения слов «awesome» и «crazy». Поскольку мы прибавляем 1 для каждого признака, надо позаботиться о том, чтобы в результате получились все-таки вероятности. И действительно, сумма оказывается равной 1:

$$P(F_1 = 1 | C = "pos") + P(F_1 = 0 | C = "pos") = \frac{3+1}{4+2} + \frac{1+1}{4+2} = 1.$$

Борьба с потерей точности при вычислениях

Есть еще один подводный камень. На практике мы имеем дело с вероятностями, гораздо меньшими тех, что встретились в этом игрушечном примере. Как правило, и число признаков намного больше двух, и их вероятности приходится перемножать. В результате очень быстро возникает ситуация, когда точности NumPy перестает хватать:

```
>>> import numpy as np
>>> np.set_printoptions(precision=20) # просим numpy выводить больше
                                      # цифр (по умолчанию 8)
>>> np.array([2.48E-324])
array([ 4.94065645841246544177e-324])
>>> np.array([2.47E-324])
array([ 0.])
```

Насколько вероятно, что нам придется столкнуться с числом вроде `2.47E-324`? Чтобы ответить на этот вопрос, давайте представим, что условные вероятности равны 0.0001, и нужно вычислить произведение 65 таких вероятностей (то есть у нас есть 65 маловероятных признаков). Вот вам и потеря точности:

```
>>> x = 0.00001
>>> x**64 # все еще нормально
1e-320
>>> x**65 # ой!
0.0
```

Числа с плавающей точкой в Python обычно реализуются на основе типа double в C. Узнать, так ли это на вашей платформе, просто:

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16,
radix=2, rounds=1)
```

Чтобы решить эту проблему, можно перейти на математические библиотеки типа `mpmath` (`http://code.google.com/p/mpmath/`), обеспечивающие произвольную точность. Однако они недостаточно быстрые, чтобы служить в качестве замены NumPy.

По счастью, существует более разумный способ, основанный на известной из школьной программы формуле:

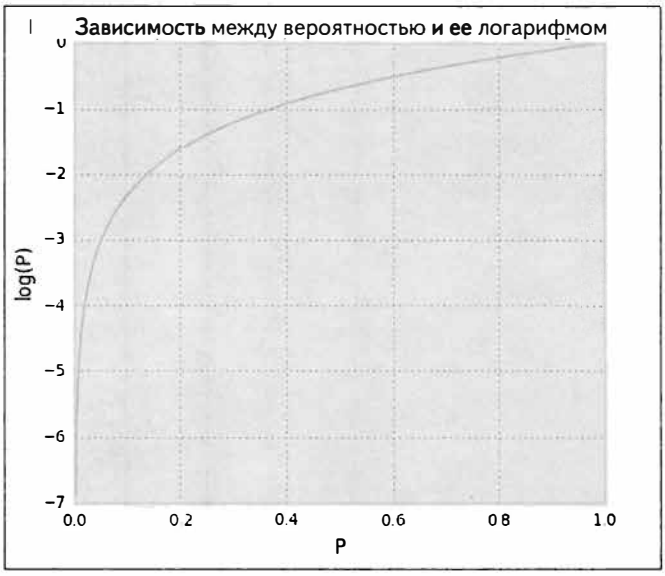
$$\log(x \cdot y) = \log(x) + \log(y)$$

В применении к нашему случаю получаем:

$$\log P(C) \cdot P(F_1|C) \cdot P(F_2|C) = \log P(C) + \log P(F_1|C) + \log P(F_2|C)$$

Поскольку вероятность лежит в интервале от 0 до 1, то ее логарифм находится в интервале от $-\infty$ до 0. Но пусть вас это не беспокоит. Попрежнему, чем больше число, тем точнее определен класс, только сами числа теперь отрицательны.

Но один подвох все же остается: в числителе дроби нет никакого логарифма, есть только произведение вероятностей. Но, к счастью, фактические значения вероятностей нам не интересны, а нужно лишь знать, у какого класса наибольшая апостериорная вероятность. И тут нам повезло, потому что если верно, что $P(C="pos"|F_1, F_2) > P(C="neg"|F_1, F_2)$, то верно и то, что $\log P(C="pos"|F_1, F_2) > \log P(C="neg"|F_1, F_2)$



Из графика на рисунке выше видно, что кривая монотонно возрастает, поэтому можно воспользоваться приведенной ранее формулой

$$C_{best} = \underset{c \in C}{\operatorname{argmax}} P(C = c) \cdot P(F_1 | C = c) \cdot P(F_2 | C = c)$$

И вот, наконец, мы получаем формулу для двух признаков, которая дает наилучший класс даже для образцов, которые мы раньше не видели:

$$C_{best} = \underset{c \in C}{\operatorname{argmax}} (\log P(C = c) + \log P(F_1 | C = c) + \log P(F_2 | C = c)$$

Разумеется, двух признаков маловато, поэтому обобщим ее на произвольное число признаков:

$$C_{best} = \underset{c \in N}{\operatorname{argmax}} \left(\log P(C = c) + \sum_k \log P(F_k | C = c) \right)$$

Теперь мы готовы применить построить наивный байесовский классификатор, пользуясь библиотекой scikit-learn.

Как уже отмечалось, мы рассматривали лишь модель Бернулли для наивного байесовского классификатора. Но вместо булевых признаков мы можем использовать счетчики вхождения слов, то есть с

мультиномиальную модель. Она дает больше информации и часто показывает лучшие результаты, поэтому к реальным данным применим именно ее. Отметим, однако, что приведенные выше формулы немного меняются. Впрочем, ничего страшного, общий принцип работы наивного байесовского классификатора остается тем же самым.

Создание и настройка классификатора

Все наивные байесовские классификаторы – а их несколько – находятся в пакете `sklearn.naive_bayes`.

- `GaussianNB`: предполагается, что признаки имеют нормальное (гауссово) распределение. Возможное применение – определение пола по росту и ширине плеч человека. Но у нас имеются тексты твитов, из которых мы извлекаем счетчики слов. Очевидно, что это не гауссово распределение.
- `MultinomialNB`: предполагается, что признаками являются счетчики вхождений, то есть как раз наш случай. На практике этот классификатор хорошо работает и с векторами TF-IDF.
- `BernoulliNB`: классификатор похож на `MultinomialNB`, но больше подходит для случая, когда признаками являются логические флаги «входит–не входит», а не счетчики слов.

Поскольку нас интересуют, прежде всего, вхождения слов, то лучше взять `MultinomialNB`.

Сначала решим простую задачу

Как мы уже видели, эмоциональная окраска твитов может быть не только положительной или отрицательной. На самом деле, большинство твитов вообще никак не окрашены, они нейтральны или нерелевантны и просто содержат информацию (например, «Вышла новая книга: Построение систем машинного обучения... http://link»). Получается, стало быть, четыре класса. Чтобы не усложнять задачу, сосредоточимся пока только на положительных и отрицательных твитах.

```
>>> # сначала создаем список, в котором элемент равен true, если твит
>>> # имеет положительную или отрицательную эмоциональную окраску
>>> pos_neg_idx = np.logical_or(Y=="positive", Y=="negative")

>>> # теперь с помощью этого индекса фильтруем данные и метки
>>> X = X[pos_neg_idx]
```

```
>>> Y = Y[pos_neg_idx]

>>> # и наконец, преобразуем сами метки в булевы величины
>>> Y = Y=="positive"
```

Теперь в массиве `X` находятся исходные тексты твитов, а в `Y` – их бинарные метки: 0 для отрицательных твитов и 1 для положительных.

Выше мы сказали, что будем использовать в качестве признаков счетчики вхождений. Да, но только не напрямую, а воспользуемся уже знакомым классом `TfidfVectorizer` для преобразования исходного текста твита в значения TF-IDF, которые наряду с метками применим для обучения классификатора. Для удобства будем использовать класс `Pipeline`, который позволяет связать векторизатор и классификатор, предоставляя тот же интерфейс:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

def create_ngram_model():
  tfidf_ngrams = TfidfVectorizer(ngram_range=(1, 3),
                                 analyzer="word", binary=False)
  clf = MultinomialNB()
  return Pipeline([('vect', tfidf_ngrams), ('clf', clf)])
```

Объект `Pipeline`, возвращенный функцией `create_ngram_model()`, можно использовать для обучения и предсказания, как обычный классификатор.

Поскольку данных у нас не так уж много, придется прибегнуть к перекрестной проверке. Но на этот раз мы воспользуемся не классом `KFold`, который последовательно выделяет из данных одну группу для тестирования, а классом `ShuffleSplit`. Он случайно перетасовывает данные, но не препятствует появлению одного и того же образца в нескольких группах. Для каждой группы мы будем вычислять и запоминать площадь под кривой точность-полнота и верность.

Для повышения гибкости эксперимента мы обернем всю процедуру функцией train_model(), которая принимает в качестве параметра функцию создания классификатора.

```
from sklearn.metrics import precision_recall_curve, auc
from sklearn.cross_validation import ShuffleSplit

def train_model(clf_factory, X, Y):
  # задаем random_state, чтобы получить детерминированное поведение
  cv = ShuffleSplit(n=len(X), n_iter=10, test_size=0.3,
```

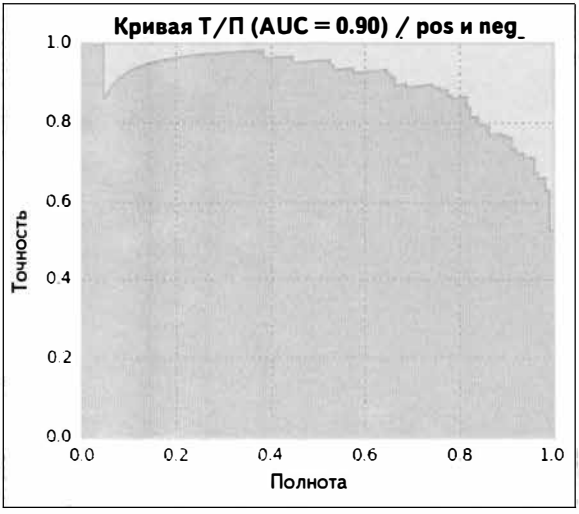
```
    random_state=0)

    scores = []
    pr_scores = []

    for train, test in cv:
        X_train, y_train = X[train], Y[train]
        X_test, y_test = X[test], Y[test]

        clf = clf_factory()
        clf.fit(X_train, y_train)

        train_score = clf.score(X_train, y_train)
        test_score = clf.score(X_test, y_test)

        scores.append(test_score)
        proba = clf.predict_proba(X_test)

        precision, recall, pr_thresholds =
        precision_recall_curve(y_test, proba[:,1])

        pr_scores.append(auc(recall, precision))

        summary = (np.mean(scores), np.std(scores),
                   np.mean(pr_scores), np.std(pr_scores))
        print("%.3f\t%.3f\t%.3f\t%.3f" % summary)
```

Собрав все вместе, мы можем обучить свою первую модель:

```
>>> X, Y = load_sanders_data()
>>> pos_neg_idx = np.logical_or(Y=="positive", Y=="negative")
>>> X = X[pos_neg_idx]
>>> Y = Y[pos_neg_idx]
>>> Y = Y=="positive"
>>> train_model(create_ngram_model, X, Y)
0.788   0.024   0.882   0.036
```

Первая попытка построить наивный байесовский классификатор с векторными триграммными признаками TF-IDF дала верность 78.8% и среднюю площадь под кривой Т/П 88.2%. На графике Т/П медианы (то разбиение данных на обучающие и тестовые, при котором наблюдается поведение, максимально близкое к среднему) видно, что поведение классификатора внушает куда больше оптимизма, чем в предыдущей главе.

Для первой попытки результаты вполне обнадеживающие. Тем более что верность 100 % в задаче классификации эмоциональной окраски вообще недостижима. Относительно окраски некоторых твитов даже люди не могут прийти к единому мнению.



Использование всех классов

Мы снова упростили задачу, оставив только положительные и отрицательные твиты. То есть мы предположили, что существует некий идеальный классификатор, который предварительно решает, есть ли у твита эмоциональная окраска, и, если да, то передает его нашему наивному байесовскому классификатору.

А что получится, если мы попытаемся также определять, содержит ли твит вообще какие-то эмоции? Чтобы ответить на этот вопрос, напишем сначала вспомогательную функцию, которая возвращает модифицированный массив классов, содержащий список эмоций, которые мы хотели бы интерпретировать как положительные:

```
def tweak_labels(Y, pos_sent_list):
 pos = Y==pos_sent_list[0]
 for sent_label in pos_sent_list[1:]:
    pos |= Y==sent_label

 Y = np.zeros(Y.shape[0])
 Y[pos] = 1
 Y = Y.astype(int)

 return Y
```

Обратите внимание, что теперь у нас два понятия «положительности». Эмоциональная окраска твита может быть положительной, но

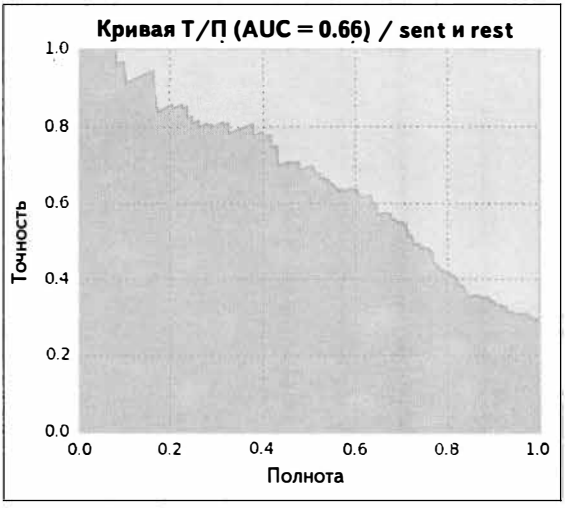
ее надо отличать от класса обучающих данных. Если, например, мы хотим узнать, насколько хорошо способны отличать эмоционально окрашенные твиты от нейтральных, то можем написать:

```
>>> Y = tweak_labels(Y, ["positive", "negative"])
```

Теперь в массиве `Y` значение `1` (положительный класс) проставлено для всех твитов, имеющих положительную или отрицательную окраску, и 0 для нейтральных и нерелевантных твитов.

```
>>> train_model(create_ngram_model, X, Y, plot=True)
0.750   0.012   0.659   0.023
```

Взгляните на следующий график.

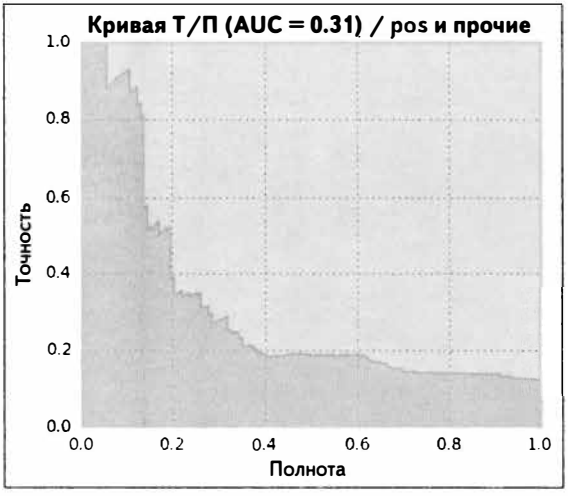


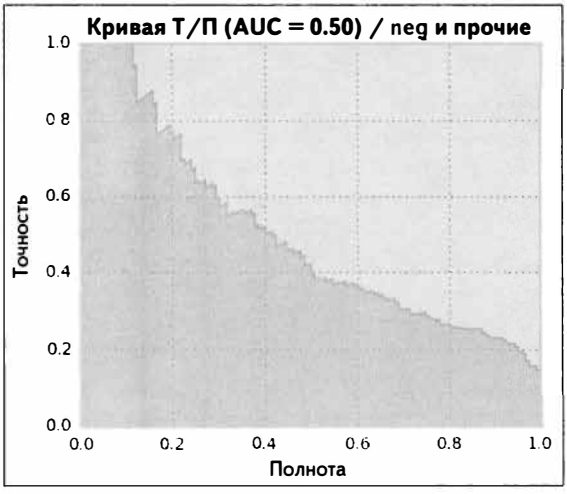
Как и следовало ожидать, площадь под кривой Т/П заметно упала и теперь составляет только 66%. Верность по-прежнему высокая, но лишь потому, что наш набор данных далек от сбалансированности. Из 3362 твитов только 920, то есть всего около 27 процентов, имеют положительную или отрицательную эмоциональную окраску. Это означает, что даже классификатор, который для любого твита говорит, что тот не имеет эмоциональной окраски, будет иметь верность 73%. Следовательно, ситуация, в которой обучающий и тестовый наборы не сбалансированы, – еще один пример, когда анализ точности и полноты обязателен.

Так как же наивный байесовский классификатор отличает положительные (отрицательные) твиты от прочих? Плохо отличает.

```
== Pos vs. rest ==
0.873 0.009 0.305 0.026

== Neg vs. rest ==
0.861 0.006 0.497 0.026
```

На мой взгляд, это никуда не годится. Кривые Т/П на графиках ниже тоже не показывают сколько-нибудь приемлемого компромисса между точностью и полнотой – в отличие от классификатора из предыдущей главы.





Настройка параметров классификатора

Конечно, мы недостаточно подробно исследовали возможности текущего решения, и этим нужно заняться. Есть две части, где можно «покрутить ручки»: `TfidfVectorizer` и `MultinomialNB`. Поскольку никаких интуитивных предположений, в какую сторону смотреть, нет, попробуем разложить значения параметров по полочкам.

Сначала рассмотрим параметры `TfidfVectorizer`:

- различные настройки *n*-грамм:
  - униграммы (1,1);
  - униграммы и биграммы (1,2);
  - униграммы, биграммы и триграммы (1,3).
- варьирование `min_df`: 1 или 2;
- изучение влияния IDF в паре TF-IDF с помощью параметров `use_idf` и `smooth_idf`: `False` или `True`;
- удалять или нет стоп-слова – параметр `stop_words` равен `english` или `None`;
- использовать ли логарифмы счетчиков слов (`sublinear_tf`);
- использовать счетчики слова или просто отслеживать, встречается слово или нет – параметр `binary` равен `True` или `False`.

Теперь обратимся к параметрам классификатора `MultinomialNB`:

- параметр `alpha`, определяющий какой использовать метод сглаживания:
  - с прибавлением 1, или сглаживание Лапласа: 1;
  - сглаживание Лидстона: 0.01, 0.05, 0.1 или 0.5;
  - без сглаживания: 0.

Можно подойти просто: обучить классификаторы для всех разумных значений одного параметра, зафиксировав остальные, и посмотреть на результаты. Поскольку мы не знаем, влияют ли параметры друг на друга, то придется обучать классификаторы для всех возможных комбинаций значений параметров. Очевидно, это слишком трудоемко.

Поскольку такого рода исследование параметров встречается в машинном обучении довольно часто, в scikit-learn имеется специальный класс для этой цели: `GridSearchCV`. Он принимает объект-оценщик (реализующий такой же интерфейс, как у классификатора), которым в нашем случае будет объект `Pipeline` и словарь параметров с потенциальными значениями.

GridSearchCV ожидает, что ключи словаря представлены в определенном формате, чтобы можно было задать параметры подходящего оценщика. Формат таков:

```
<estimator>__<subestimator>__...__<param_name>
```

Например, если мы хотим задать значения для исследования параметра ngram_range объекта TfidfVectorizer (который в описании Pipeline называется vect), то должны написать:

```
param_grid={"vect__ngram_range"=[(1, 1), (1, 2), (1, 3)]}
```

Тогда GridSearchCV будет пробовать униграммы, биграммы и триграммы в качестве значений параметра ngram_range объекта vect.

Затем эта функция обучает оценщик на всех возможных комбинациях значений параметров. Причем обучение производится на случайных выборках из обучающих данных, для чего мы используем объект ShuffleSplit, который порождает итератор случайных разбиений на обучающий и тестовый наборы. И напоследок функция возвращает лучший оценщик в переменной-члене best_estimator_.

Поскольку мы хотим сравнивать возвращенный лучший классификатор с текущим лучшим, необходимо каким-то образом оценить его качество. Для этого можно передать объект ShuffleSplit в параметре cv (отсюда и буквы CV в имени GridSearchCV).

Последнее, что осталось, – решить, как GridSearchCV должен выбирать лучший оценщик. Для этого можно передать функцию рейтингования score в параметре score_func. Ее можно написать самостоятельно или взять готовую из пакета sklearn.metrics. Только не надо брать функцию metric.accuracy из-за дисбаланса классов (эмоционально окрашенных твитов гораздо меньше, чем нейтральных). Нам же нужна хорошая точность и полнота на обоих классах – эмоционально окрашенных и лишенных окраски. Примером метрики, сочетающей точность и полноту, может служить **F-мера**, реализованная в функции metrics.f1_score:

$$F = \frac{2 \cdot точность \cdot полнота}{точность + полнота}$$

Собрав все вместе, получаем такой код:

```
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import f1_score

def grid_search_model(clf_factory, X, Y):
```

```
    cv = ShuffleSplit(
        n=len(X), n_iter=10, test_size=0.3,random_state=0)

    param_grid = dict(vect__ngram_range=[(1, 1), (1, 2), (1, 3)],
        vect__min_df=[1, 2],
        vect__stop_words=[None, "english"],
        vect__smooth_idf=[False, True],
        vect__use_idf=[False, True],
        vect__sublinear_tf=[False, True],
        vect__binary=[False, True],
        clf__alpha=[0, 0.01, 0.05, 0.1, 0.5, 1],
    )

    grid_search = GridSearchCV(clf_factory(),
        param_grid=param_grid,
        cv=cv,
        score_func=f1_score,
        verbose=10)
    grid_search.fit(X, Y)

    return grid_search.best_estimator_
```

После запуска этой функции нужно запастись терпением:

```
clf = grid_search_model(create_ngram_model, X, Y)
print(clf)
```

Перебор всех $3 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 6 = 1152$ комбинаций, да еще с десятипроходным обучением каждого классификатора требует времени.

```
... спустя несколько часов ...
Pipeline(clf=MultinomialNB(
alpha=0.01, class_weight=None, fit_prior=True),
clf__alpha=0.01,
clf__class_weight=None,
clf__fit_prior=True,
vect=TfidfVectorizer(
analyzer=word, binary=False,
charset=utf-8, charset_error=strict,
dtype=<type 'long'>,input=content,
lowercase=True, max_df=1.0,
max_features=None, max_n=None,
min_df=1, min_n=None, ngram_range=(1, 2),
norm=l2, preprocessor=None, smooth_idf=False,
stop_words=None,strip_accents=None,
sublinear_tf=True,token_pattern=(?u)\b\w\w+\b,
token_processor=None, tokenizer=None,
use_idf=False, vocabulary=None),
vect__analyzer=word, vect__binary=False,
vect__charset=utf-8,
```

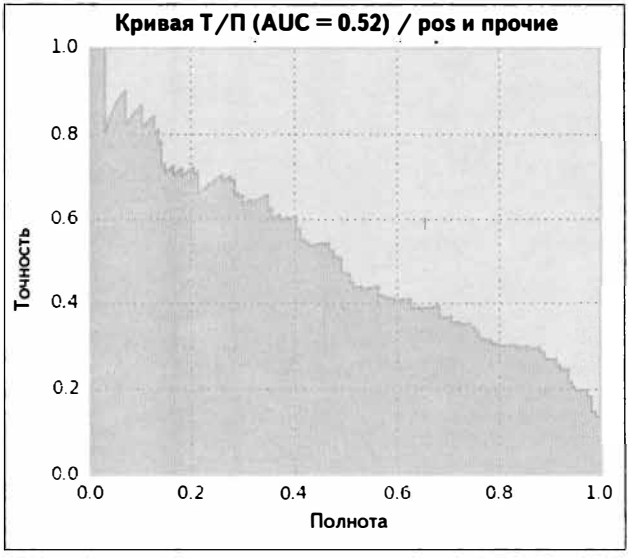
```
vect__charset_error=strict,
vect__dtype=<type 'long'>,
vect__input=content, vect__lowercase=True,
vect__max_df=1.0,vect__max_features=None,
vect__max_n=None, vect__min_df=1,
vect__min_n=None, vect__ngram_range=(1, 2),
vect__norm=l2, vect__preprocessor=None,
vect__smooth_idf=False, vect__stop_words=None,
vect__strip_accents=None, vect__sublinear_tf=True,
vect__token_pattern=(?u)\b\w\w+\b,
vect__token_processor=None, vect__tokenizer=None,
vect__use_idf=False, vect__vocabulary=None)
0.795   0.007   0.702   0.028
```

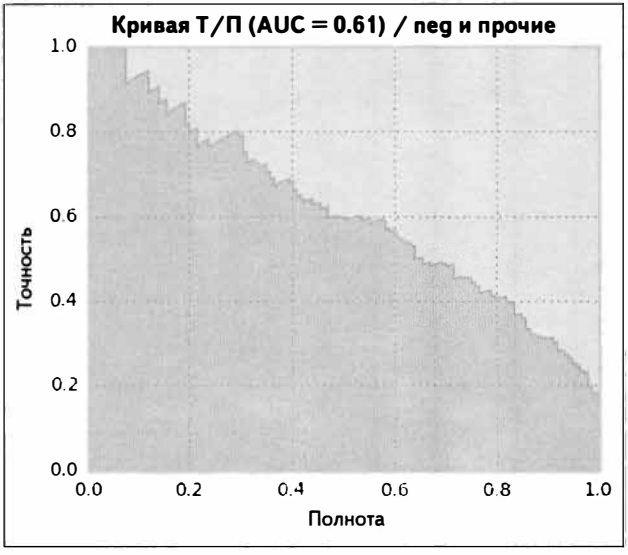
Наилучший оценщик действительно увеличивает площадь под кривой Т/П почти на 3.3%, доведя ее до уровня 70.2. Значения параметров, при которых это достигается, показаны выше.

Улучшаются и кошмарные результаты различения положительных (отрицательных) и прочих твитов, если присвоить параметрам векторизатора и классификатора найденные значения:

```
== Pos vs. rest ==
0.889 0.010 0.509 0.041
== Neg vs. rest ==
0.886 0.007 0.615 0.035
```

Взгляните на графики ниже.





Действительно, кривые Т/П выглядят гораздо лучше (отметим, что графики построены для медианного классификатора, прошедшего многопроходное обучение, поэтому значения AUC немного отличаются). И все же мы бы не стали использовать такие классификаторы на практике. Пора подойти к задаче с другого конца...

Очистка твитов

Новые ограничения рождают новые формы. И Твиттер – не исключение. Поскольку нужно уложиться в 140 символов, пользователи придумали новые сокращения, позволяющие выразить мысль меньшим числом символов. До сих пор мы игнорировали всяческие смайлики и аббревиатуры. Посмотрим, что удастся сделать, приняв их во внимание. Для этого понадобится написать собственную функции `preprocessor()` и подсунуть ее объекту `TfidfVectorizer`.

Прежде всего, определим словарь, в котором каждому смайлику сопоставляется текстовая замена. Можно было бы брать разные заменяющие слова, но мы ограничимся очевидными словами, несущими положительную и отрицательную окраску:

```
emo_repl = {
  # положительные смайлики
  "&lt;3": " good ",
```

```
    ":d": " good ", # :D в нижнем регистре
    ":dd": " good ", # :DD в нижнем регистре
    "8)": " good ",
    ":-)": " good ",
    ":)": " good ",
    ";)": " good ",
    "(-:": " good ",
    "(:": " good ",

    # отрицательные смайлики:
    ":/": " bad ",
    ":&gt;": " sad ",
    ":')": " sad ",
    ":-(": " bad ",
    ":(": " bad ",
    ":S": " bad ",
    ":-S": " bad ",
  }

  # нужно, чтобы :dd заменялось раньше :d (к примеру)
  emo_repl_order = [k for (k_len,k) in reversed(sorted([(len(k),k)
for k in
  emo_repl.keys()]))]
```

Затем определяем регулярные выражения для выделения аббревиатур и соответствующие им расширения (`\b` обозначает границу слова):

```
  re_repl = {
    r"\br\b": "are",
    r"\bu\b": "you",
    r"\bhaha\b": "ha",
    r"\bhahaha\b": "ha",
    r"\bdon't\b": "do not",
    r"\bdoesn't\b": "does not",
    r"\bdidn't\b": "did not",
    r"\bhasn't\b": "has not",
    r"\bhaven't\b": "have not",
    r"\bhadn't\b": "had not",
    r"\bwon't\b": "will not",
    r"\bwouldn't\b": "would not",
    r"\bcan't\b": "can not",
    r"\bcannot\b": "can not",
  }

  def create_ngram_model(params=None):
    def preprocessor(tweet):
       tweet = tweet.lower()
       for k in emo_repl_order:
          tweet = tweet.replace(k, emo_repl[k])
```

```
    for r, repl in re_repl.items():
        tweet = re.sub(r, repl, tweet)

    return tweet

tfidf_ngrams = TfidfVectorizer(preprocessor=preprocessor,
                               analyzer="word")
# ...
```

Конечно, аббревиатур гораздо больше, чем здесь перечислено. Но даже с таким ограниченным набором качество различения эмоционально окрашенных и нейтральных твитов повысилось на полпроцента, до 70.7%:

```
== Pos vs. neg ==
0.808   0.024   0.885   0.029
== Pos/neg vs. irrelevant/neutral ==
0.793   0.010   0.685   0.024
== Pos vs. rest ==
0.890   0.011   0.517   0.041
== Neg vs. rest ==
0.886   0.006   0.624   0.033
```

Учет типов слов

До сих пор мы надеялись, что будет достаточно подхода на основе набора слов, в котором предполагается, что все слова независимы. Однако интуиция подсказывает, что нейтральные твиты, скорее всего, содержат больше существительных, тогда как для красочного выражения эмоций потребуются прилагательные и глаголы. А что, если использовать и такую лингвистическую информацию? Если бы мы могли определить, сколько в тексте твита существительных, глаголов, прилагательных и т. д., то, возможно, классификатор смог бы этим воспользоваться.

Определение типов слов

В этом и состоит задача частеречной разметки (POS). Разметчик анализирует предложение целиком, пытаясь построить дерево зависимостей, в котором каждый узел соответствует слову, а связи типа родитель-потомок определяют, от какого слова оно зависит. Имея такое дерево, мы сможем принимать обоснованные решения, например о том, какой частью речи является слово «book» – именем существительным («This is a good book» – «Это хорошая книга») или глаголом

(«Could you please book the flight?» – «Не могли бы вы забронировать билет на самолет?»).

Вы, наверное, уже догадались, что NLTK причастен и к этой задаче. Действительно, в его дистрибутив входят разнообразные анализаторы и разметчики. Частеречный разметчик `nltk.pos_tag()` на самом деле является полноценным классификатором, обученным на вручную аннотированных предложениях из проекта Penn Treebank (`http://www.cis.upenn.edu/~treebank`). Он принимает на входе список лексем и возвращает список кортежей, в котором каждый элемент содержит часть исходного предложения и метку части речи.

```
>>> import nltk

>>> nltk.pos_tag(nltk.word_tokenize(«This is a good book.»))
[('This', 'DT'), ('is', 'VBZ'), ('a', 'DT'), ('good', 'JJ'), ('book',
'NN'), ('.', '.')]

>>> nltk.pos_tag(nltk.word_tokenize("Could you please book the flight?"))
[('Could', 'MD'), ('you', 'PRP'), ('please', 'VB'), ('book', 'NN'),
('the', 'DT'), ('flight', 'NN'), ('?', '.')]
```

Сокращенные названия меток взяты из проекта Penn Treebank (см. `http://www.anc.org/OANC/penn.html`):

Метка части речи	Описание	Пример
CC	соединительный союз	or
CD	количественное числительное	2, second
DT	определительное слово	the
EX	*there* в предложении существования	*there* are
FW	иностранное слово	kindergarten
IN	предлог, подчинительный союз	on, of, like
JJ	имя прилагательное	cool
JJR	имя прилагательное в сравнительной степени	cooler
JJS	имя прилагательное в превосходной степени	coolest
LS	маркер списка	1)
MD	модальный глагол	could, will
NN	имя существительное в единственном числе или неисчисляемое	book

Метка части речи	Описание	Пример
NNS	имя существительное во множественном числе	books
NNP	имя собственное в единственном числе	Sean
NNPS	имя собственное во множественном числе	Vikings
PDT	предетерминатив	both the boys
POS	притяжательный падеж	friend's
PRP	личное местоимение	I, he, it
PRP$	притяжательное местоимение	my, his
RB	наречие	however, usually, naturally, here, good
RBR	наречие в сравнительной степени	better
RBS	наречие в превосходной степени	best
RP	частица	give *up*
TO	*to*	*to* go, *to* him
UH	междометие	uhhuhhuhh
VB	глагол в основной форме	take
VBD	глагол в форме прошедшего времени	took
VBG	глагол в форме герундия или причастия настоящего времени	taking
VBN	глагол в форме причастия прошедшего времени	taken
VBP	глагол в форме единственного числа настоящего времени, не в третьем лице	take
VBZ	глагол в форме единственного числа настоящего времени, в третьем лице	takes
WDT	определительное слово на *wh*	which
WP	местоимение на *wh*	who, what
WP$	притяжательное местоимение на *wh*	whose
WRB	наречие на *wh*	where, when

Теперь несложно отобрать нужные метки из результата, возвращенного функцией `pos_tag()`. Чтобы узнать число существительных, нуж-

но просто подсчитать слова, метки которых начинаются с NN, для глаголов соответственно VB, для прилагательных – JJ, а для наречий – RB.

Удачный обмен с помощью SentiWordNet

Как отмечалось в предыдущем разделе, лингвистическая информация, скорее всего, нам поможет, но для ее сбора есть инструмент получше: SentiWordNet (`http://sentiwordnet.isti.cnr.it`). Попросту говоря, это файл размером 13 МБ, который приписывает большинству английских слов положительную или отрицательную коннотацию. Более точно, каждому набору синонимов сопоставляется положительная или отрицательная эмоциональная окраска. Вот несколько примеров.

Часть речи	Ид	PosScore	NegScore	SynsetTerms	Описание
a	00311354	0.25	0.125	studious#1	Требующий старания и усилий; «усердно пытался починить телевизор»
a	00311663	0	0.5	careless#1	Отличающийся недостатком внимания, обдумывания, тщательности; противоположность careful
n	03563710	0	0	implant#1	Протез, постоянно размещаемый в тканях тела
v	00362128	0	0	kink#2 curve#5 curl#1	Образующий завиток, колечко или петлю; «сигарный дым кольцами поднимался к потолку»

Благодаря информации в столбце «Часть речи» мы можем отличить существительное «book» (книга) от глагола «book» (бронировать). Столбцы `PosScore` и `NegScore` в совокупности позволяют определить степень нейтральности слова, равную 1-PosScore-NegScore. В столбце `SynsetTerms` перечислены все слова, составляющие данные набор синонимов. Столбцы **Ид** и **Описание** нам не интересны.

К словам из набора синонимов добавлено число, поскольку некоторые из них встречаются в разных наборах синонимов. Например, у слова «fantasize» два значения, а, значит, и две оценки:

Часть речи	Ид	PosScore	NegScore	SynsetTerms	Описание
v	01636859	0.375	0	fantasize#2 fantasize#2	Мысленно представлять; «он нарисовал мысленный портрет идеальной жены»
v	01637368	0	0.125	fantasy#1 fantasize#1 fantasize#1	Фантазировать; «он фантазирует, говоря о планах создать свою компанию»

Чтобы решить, какой набор синонимов относится к делу, мы должны понимать смысл твитов, но эта тема выходит за рамки данной главы. Область исследований, в центре внимания которой находится эта задача, называется «разрешение лексической многозначности». Мы пойдем по легкому пути и просто усредним оценки по всем наборам синонимов, в которых встречается терм. Для слова «fantasize» `PosScore` будет равно 0.1875, а `NegScore` – 0.0625.

Приведенная ниже функция `load_sent_word_net()` делает все вышеописанное и возвращает словарь, ключами которого являются строки вида *тип слова/слово*, например n/implant, а значениями – положительная и отрицательная оценки.

```
import csv, collections

def load_sent_word_net():
  # упростим себе жизнь, воспользовавшись словарем, который
  # автоматически создает пустой список при попытке обратиться
  # к еще не существующему ключу
  sent_scores = collections.defaultdict(list)

  with open(os.path.join(DATA_DIR, SentiWordNet_3.0.0_20130122.txt"),
"r") as csvfile:
      reader = csv.reader(csvfile, delimiter='\t', quotechar='"')
      for line in reader:
         if line[0].startswith("#"):
            continue
         if len(line)==1:
            continue

         POS, ID, PosScore, NegScore, SynsetTerms, Gloss = line
         if len(POS)==0 or len(ID)==0:
            continue
         for term in SynsetTerms.split(" "):
            # отбросить число в конце терма
```

```
            term = term.split("#")[0]
            term = term.replace("-", " ").replace("_", " ")
            key = "%s/%s"%(POS, term.split("#")[0])
            sent_scores[key].append((float(PosScore),float(NegScore)))

    for key, value in sent_scores.items():
        sent_scores[key] = np.mean(value, axis=0)

    return sent_scores
```

Наш первый оценщик

Теперь у нас есть все необходимое для создания своего первого векторизатора. Самый удобный способ унаследовать классу `BaseEstimator`, для чего требуется реализовать следующие три метода:

- `get_feature_names()`: возвращает список имен признаков, которые мы будет возвращать из `transform()`;
- `fit(document, y=None)`: поскольку мы не собираемся обучать классификатор, можем здесь ничего не делать, а просто вернуть `self`;
- `transform(documents)`: возвращает массив `numpy.array()`, имеющий форму `(len(documents), len(get_feature_names))`. Это означает, что для каждого документа из списка `documents` возвращаются значения всех признаков, имена которых получены от функции `get_feature_names()`.

Реализация показана ниже.

```
sent_word_net = load_sent_word_net()

class LinguisticVectorizer(BaseEstimator):
  def get_feature_names(self):
      return np.array(['sent_neut', 'sent_pos', 'sent_neg',
        'nouns', 'adjectives', 'verbs', 'adverbs',
        'allcaps', 'exclamation', 'question', 'hashtag',
        'mentioning'])

  # мы не занимаемся обучением, но должны вернуть ссылку, чтобы
  # можно было вызвать метод так: fit(d).transform(d)
  def fit(self, documents, y=None):
      return self

  def _get_sentiments(self, d):
      sent = tuple(d.split())
      tagged = nltk.pos_tag(sent)

      pos_vals = []
```

```
    neg_vals = []

    nouns = 0.
    adjectives = 0.
    verbs = 0.
    adverbs = 0.

    for w,t in tagged:
       p, n = 0,0
       sent_pos_type = None
       if t.startswith("NN"):
          sent_pos_type = "n"
          nouns += 1
       elif t.startswith("JJ"):
          sent_pos_type = "a"
          adjectives += 1
       elif t.startswith("VB"):
          sent_pos_type = "v"
          verbs += 1
       elif t.startswith("RB"):
          sent_pos_type = "r"
          adverbs += 1

       if sent_pos_type is not None:
          sent_word = "%s/%s" % (sent_pos_type, w)

          if sent_word in sent_word_net:
            p,n = sent_word_net[sent_word]

       pos_vals.append(p)
       neg_vals.append(n)

    l = len(sent)
    avg_pos_val = np.mean(pos_vals)
    avg_neg_val = np.mean(neg_vals)
    return [1-avg_pos_val-avg_neg_val, avg_pos_val, avg_neg_val,
           nouns/l, adjectives/l, verbs/l, adverbs/l]

def transform(self, documents):
    obj_val, pos_val, neg_val, nouns, adjectives, \
    verbs, adverbs = np.array([self._get_sentiments(d) \
    for d in documents]).T

    allcaps = []
    exclamation = []
    question = []
    hashtag = []
    mentioning = []

    for d in documents:
       allcaps.append(np.sum([t.isupper() \
```

```
        for t in d.split() if len(t)>2]))

    exclamation.append(d.count("!"))
    question.append(d.count("?"))
    hashtag.append(d.count("#"))
    mentioning.append(d.count("@"))
    result = np.array([obj_val, pos_val, neg_val, nouns, adjectives,
                       verbs, adverbs, allcaps, exclamation, question,
                       hashtag, mentioning]).T

    return result
```

Соберем все вместе

Однако одно лишь использование лингвистических признаков без самих слов мало что дает. Поэтому нужно объединить параметр `TfidfVectorizer` с лингвистическими признаками. Для этого служит класс `FeatureUnion` из библиотеки scikit-learn. Инициализируется он так же, как `Pipeline`, но вместо последовательного вызова оценщиков, каждый из которых подает свой выход на вход следующего, `FeatureUnion` вызывает их параллельно и потом объединяет результирующие векторы.

```
def create_union_model(params=None):
  def preprocessor(tweet):
     tweet = tweet.lower()
     for k in emo_repl_order:
        tweet = tweet.replace(k, emo_repl[k])
     for r, repl in re_repl.items():
        tweet = re.sub(r, repl, tweet)

     return tweet.replace("-", " ").replace("_", " ")

  tfidf_ngrams = TfidfVectorizer(preprocessor=preprocessor,
  analyzer="word")

  ling_stats = LinguisticVectorizer()

  all_features = FeatureUnion([('ling', ling_stats),
                               ('tfidf', tfidf_ngrams)])

  clf = MultinomialNB()
  pipeline = Pipeline([('all', all_features), ('clf', clf)])

  if params:
     pipeline.set_params(**params)

  return pipeline
```

После обучения и тестирования комбинированных выделителей признаков мы улучшаем площадь под кривой Т/П для различения положительно и отрицательно окрашенных твитов еще на 0.4%.

```
== Pos vs. neg ==
0.810   0.023   0.890   0.025

== Pos/neg vs. irrelevant/neutral ==
0.791   0.007   0.691   0.022

== Pos vs. rest ==
0.890   0.011   0.529   0.035

== Neg vs. rest ==
0.883   0.007   0.617   0.033

time spent: 214.12578797340393
```

С такими результатами мы вряд ли захотим использовать классификаторы для различения положительных (или отрицательных) и прочих твитов, но сначала воспользуемся классификатором, который определяет, несет ли твит хоть какую-то эмоциональную окраску (pos/neg versus irrelevant/neutral), и, если несет, то применим классификатор, различающий положительные и отрицательные эмоции.

Резюме

Поздравляем тех, кто дошел с нами до самого конца! Мы изучили, как работает наивный байесовский классификатор, и объяснили, почему он вовсе не является таким уж наивным. Особенно хорошо этот классификатор справляется с задачей обобщения, когда обучающий набор содержит недостаточно данных, чтобы покрыть все закоулки пространства вероятностей. Мы видели, как применить его к классификации твитов и поняли, что очистка текста твитов очень помогает. Наконец, мы уяснили, что небольшой «обман» (но после того, как основная часть работы уже проделана) не повредит. Тем более, когда он улучшает качество работы классификатора, как то произошло в результате использования `SentiWordNet`.

В следующей главе мы познакомимся с регрессией.

ГЛАВА 7. Регрессия

Вы, наверное, изучали регрессию в вузовском курсе высшей математики. А конкретно – регрессию **обычным методом наименьших квадратов** (OLS). Этому очень быстрому методу уже 200 лет, но он по-прежнему применяется для решения многих практических задач. Мы начнем с его описания и покажем, как он реализован в scikit-learn.

Но есть задачи, для которых этого метода недостаточно. Особенно это относится к случаю, когда признаков много, а если их больше, чем образцов, то метод и вовсе непригоден. В таких ситуациях нужны более сложные методы – появившиеся совсем недавно, в последние десять лет. Это методы Lasso, гребневой регрессии и эластичных сетей. Все они есть в scikit-learn, и мы подробно рассмотрим их.

Прогнозирование стоимости домов с помощью регрессии

Начнем с простой задачи – прогнозирования стоимости домов в Бостоне. Для нее мы сможем воспользоваться общедоступным набором данных. Нам известно несколько демографических и географических характеристик, например: уровень преступности в округе или среднее число учащихся на одного преподавателя. Цель – предсказать медианную стоимость дома в определенном районе. Как обычно, имеются обучающие данные, в которых ответ уже известен.

Это один из наборов, входящих в дистрибутив scikit-learn, поэтому загрузить его в память совсем просто:

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
```

Объект `boston` имеет несколько атрибутов; в частности, `boston.data` содержит исходные данные, а `boston.target` – цены домов.

Начнем с простой одномерной регрессии, то есть попытаемся предсказать цену всего по одному атрибуту, среднему числу комнат в жилом помещении в данном районе. Этот атрибут хранится в пятом элементе записи (подробную информацию о данных можно почерпнуть из атрибутов `boston.DESCR` и `boston.feature_names`):

```
>>> from matplotlib import pyplot as plt
>>> plt.scatter(boston.data[:,5], boston.target, color='r')
```

Атрибут `boston.target` содержит среднюю цену дома (наша целевая переменная). Можно применить стандартную регрессию методом наименьших квадратов, с которой вы, скорее всего, знакомы. Первая попытка выглядит так:

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression()
```

Мы импортируем класс `LinearRegression` из модуля `sklearn.linear_model` и конструируем объект этого класса. Он ведет себя так же, как рассмотренные выше объекты классификаторов из библиотеки scikit-learn.

```
>>> x = boston.data[:,5]
>>> y = boston.target
>>> x = np.transpose(np.atleast_2d(x))
>>> lr.fit(x, y)
>>> y_predicted = lr.predict(x)
```

Единственная неочевидная строка в этом коде – обращение к методу `np.atleast_2d`, который преобразует одномерный массив `x` в двумерный. Это преобразование необходимо, потому что метод `fit` ожидает, что в первом аргументе будет передан двумерный массив. Затем, чтобы расположить измерения в нужном порядке, мы транспонируем этот массив.

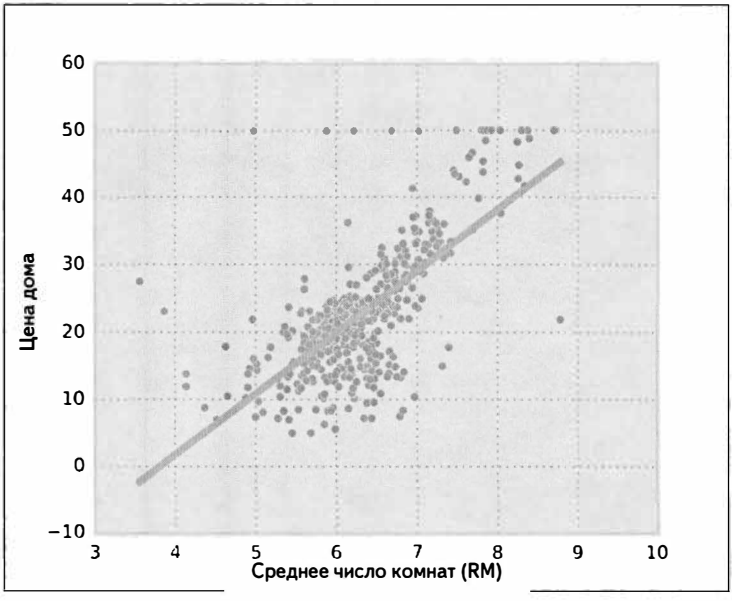
Отметим, что методы `fit` и `predict` вызываются от имени объекта `LinearRegression` – так же, как в случае объектов классификаторов, – хотя сейчас мы производим регрессию. Это единообразие API – одно из достоинств scikit-learn.

На графике ниже показаны все образцы (в виде точек) и наша аппроксимация (сплошная прямая линия). Выглядит неплохо, если не считать нескольких выбросов.

Но в идеале хотелось бы количественно измерить качество аппроксимации, иначе будет трудно сравнивать различные методы. Для этого можно измерить близость прогноза к истинным значениям, вос-

пользовавшись функцией `mean_squared_error` из модуля `sklearn.metrics`:

```
>>> from sklearn.metrics import mean_squared_error
```



Эта функция принимает два аргумента: истинные и предсказанные значения:

```
>>> mse = mean_squared_error(y, lr.predict(x))
>>> print("Квадрат среднеквадратичной погрешности (на обучающих данных): {:.3}".format(mse))
Квадрат среднеквадратичной погрешности (на обучающих данных): 58.4
```

Это значение трудно интерпретировать, лучше извлечь из него квадратный корень, получив **среднеквадратичную погрешность** (RMSE):

```
>>> rmse = np.sqrt(mse)
>>> print("RMSE (на обучающих данных): {:.3}".format(rmse))
RMSE (на обучающих данных): 6.6
```

Преимущество RMSE состоит в том, что можно быстро получить грубую оценку погрешности, умножив эту величину на два. В нашем случае можно ожидать, что оценочная стоимость будет отличаться от реальной не более, чем на 13 тысяч долларов.

Среднеквадратичная погрешность и прогноз

Среднеквадратичная погрешность приблизительно соответствует оценке стандартного отклонения. Поскольку в большинстве случаев данные отличаются от среднего не более, чем на два стандартных отклонения, мы можем умножить RMSE на два для получения грубой оценки доверительного интервала. Эта оценка теоретически правильна, только если ошибки имеют нормальное распределение, но зачастую она годится и в других случаях.

Любое конкретное число, например 6.6, нашей интуиции ничего не говорит. Этот прогноз хорош или нет? Для ответа на этот вопрос можно провести сравнение с эталоном – постоянной моделью. Если мы ничего не знаем о характере входных данных, то и спрогнозировать нельзя ничего лучше среднего значения `y`. Затем можно сравнить среднеквадратичную погрешность нашей и нулевой модели. Эту идею формализует **коэффициент детерминации**:

$$1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \approx 1 - \frac{\text{MSE}}{\text{VAR}(y)}$$

Здесь y_i – значение i-ого элемента, а $\hat{y}_i$ – оценка этого элемента, полученная с помощью регрессионной модели. Наконец, $\bar{y}$ – среднее значение y, представляющее *нулевую модель*, которая всегда возвращает одно и то же значение. Приблизительно то же самое мы получим, вычислив разность между единицей и отношением квадрата среднеквадратичной ошибки к дисперсии результата. Следовательно, для идеальной модели получится оценка 1, а для нулевой – оценка 0. Отметим, что оценка может получиться и отрицательной; это означает, что модель настолько плоха, что лучше уж брать в качестве прогноза среднее значение.

Коэффициент детерминации вычисляет функция `r2_score` из модуля `sklearn.metrics`:

```
>>> from sklearn.metrics import r2_score
>>> r2 = r2_score(y, lr.predict(x))
>>> print("R2 (на обучающих данных): {:.2}".format(r2))
R2 (на обучающих данных): 0.31
```

Эту метрику еще называют оценкой R^2. Если используется линейная регрессия, а погрешность вычисляется на обучающих данных, то она соответствует квадрату коэффициента корреляции R. Однако эта

метрика более общая и, как уже было сказано, может принимать отрицательные значения.

По-другому коэффициент детерминации можно вычислить, вызвав метод `score` объекта `LinearRegression`:

```
>>> r2 = lr.score(x,y)
```

Многомерная регрессия

До сих пор мы использовали для прогнозирования только одну переменную – среднее число комнат в доме. Теперь воспользуемся для обучения модели всеми имеющимися данными, применив многомерную регрессию. Попробуем предсказать один результат (среднюю цену дома), принимая во внимание несколько характеристик.

Код выглядит почти так же, как раньше. Он даже проще, потому что атрибут `boston.data` можно передать методу `fit` непосредственно:

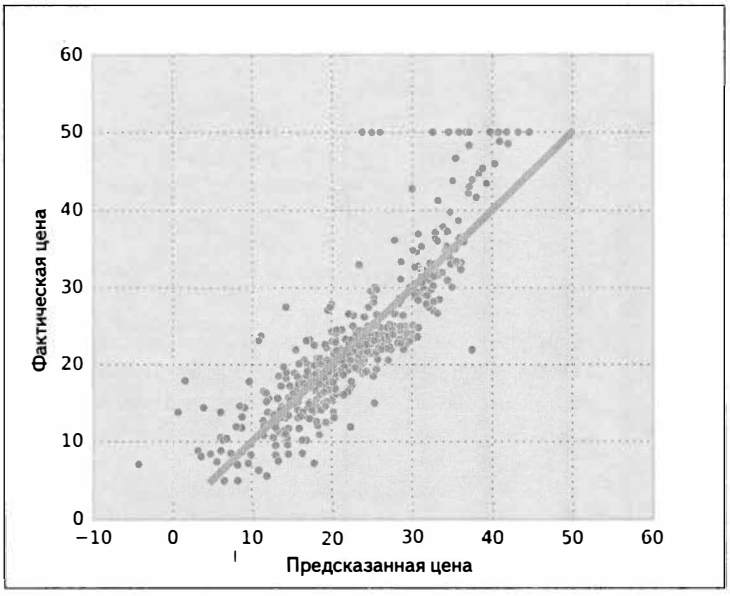
```
>>> x = boston.data
>>> y = boston.target
>>> lr.fit(x, y)
```

При использовании всех входных данных среднеквадратичная погрешность составляет всего 4.7, что соответствует коэффициенту детерминации 0.74. Это лучше, чем раньше, а, значит, дополнительные переменные помогли. Правда, теперь результат не удастся показать наглядно, потому что это 14-мерная гиперплоскость регрессии, а не прямая линия на плоскости.

Но можно построить график зависимости предсказанного значения от фактического. Ниже приведен код:

```
>>> p = lr.predict(x)
>>> plt.scatter(p, y)
>>> plt.xlabel('Предсказанная цена')
>>> plt.ylabel('Фактическая цена')
>>> plt.plot([y.min(), y.max()], [[y.min()], [y.max()]])
```

В последней строке проводится диагональная прямая, соответствующая идеальному согласию. Это помогает наглядно представить картину. Результаты показаны на графике ниже, где сплошной линией показана диагональ (на которой лежали бы все точки, если бы существовало идеальное согласие между истинными и предсказанными значениями).



Перекрестная проверка для регрессии

Знакомясь с классификацией, мы подчеркивали важность перекрестной проверки для проверки качества прогнозов. Для регрессии это делается не всегда. На самом деле, до сих пор мы обсуждали только погрешность на обучающих данных. Это неправильный подход, если мы хотим сделать достойные доверия выводы о способности к обобщению. Поскольку обычный метод наименьших квадратов – очень простая модель, ошибка обычно не слишком серьезна. Иными словами, степень переобучения низкая. Но все равно нужно протестировать ее хотя бы эмпирически, что легко позволяет сделать scikit-learn.

Воспользуемся классом `Kfold` для реализации пятипроходной проверки способности линейной регрессии к обобщению:

```
>>> from sklearn.cross_validation import Kfold
>>> kf = KFold(len(x), n_folds=5)
>>> p = np.zeros_like(y)
>>> for train,test in kf:
...  lr.fit(x[train], y[train])
...  p[test] = lr.predict(x[test])
```

```
>>> rmse_cv = np.sqrt(mean_squared_error(p, y))
>>> print('RMSE на 5-проходной ПП: {:.2}'.format(rmse_cv))
'RMSE на 5-проходной ПП: 5.6
```

С применением перекрестной проверки получается более консервативная оценка (то есть с большей погрешностью): `5.6`. Как и в случае классификации, оценка, полученная с помощью перекрестной проверки, точнее показывает, чего можно ожидать от обобщения модели на ранее не предъявлявшиеся данные.

Обычный метод наименьших квадратов быстро обучается и возвращает простую модель, которая быстро дает прогноз. Поэтому именно его рекомендуется использовать в качестве первой попытки решить любую задачу регрессии. Но далее мы познакомимся с более сложными методами и объясним, почему они иногда являются предпочтительными.

Регрессия со штрафом, или регуляризованная регрессия

В этом разделе описывается важный класс регрессионных моделей – регрессия со штрафом, которую также называют **регуляризованной регрессией**.

При обычной регрессии возвращается наилучшая аппроксимация обучающих данных. Это может привести к переобучению. Штрафование означает, что мы включаем штраф за чрезмерное доверие к параметрическим данным. Иначе говоря, мы соглашаемся на несколько худшую аппроксимацию ради более простой модели.

Можно взглянуть на это иначе – считать, что по умолчанию нет никакой связи между входными переменными и выходным прогнозом. Получая данные, мы изменяем это мнение, а добавление штрафа означает, что требуется больше данных, чтобы убедить нас в наличии сильной связи.

Регрессия со штрафом как компромисс

Регрессия со штрафом – еще один пример компромисса между смещением и дисперсией. Налагая штраф, мы получаем худшую аппроксимацию данных, потому что увеличиваем смещение. С другой стороны, дисперсия уменьшается, а вместе с ней и шансы получить переобученную модель. Поэтому получившийся результат, скорее всего, будет лучше обобщаться на новые (тестовые) данные.

Штрафы L1 и L2

Теперь перейдем к деталям. Те, кому математические соображения неинтересны, могут перейти сразу к следующему разделу, где описывается использование регуляризованной регрессии в scikit-learn.

В общем случае мы имеем матрицу X обучающих данных (каждая строка в ней – результат наблюдений, а каждый столбец – один признак) и вектор y выходных значений. Задача состоит в том, чтобы найти вектор весов b^*. В случае регрессии методом обычных наименьших квадратов он выражается следующей формулой:

$$\vec{b}^* = \arg\min_{\vec{b}} \mid \vec{y} - X\vec{b} \mid^2$$

То есть требуется найти вектор b, который обращает в минимум квадрат расстояния до вектора y. В этих формулах свободный член игнорируется, потому что мы предполагаем, что обучающие данные были предварительно обработаны, так что среднее значение y равно нулю.

Добавление штрафа, или регуляризация означает, что мы хотим не только найти наилучшую аппроксимацию, но и принимаем во внимание строение вектора. Существуют два основных штрафа, применяемых в случае регрессии: L1 и L2. Штраф L1 означает, что к регрессии прибавляется сумма абсолютных значений коэффициентов, а штраф L2 – что прибавляется сумма их квадратов.

Если налагается штраф L1, то вместо предыдущего выражения мы оптимизируем такое:

$$\vec{b}^* = \arg\min_{\vec{b}} \mid \vec{y} - X\vec{b} \mid^2 + \alpha \sum_i \mid b_i \mid$$

Здесь мы одновременно пытаемся уменьшить как саму погрешность, так и значения коэффициентов (по абсолютной величине). В случае штрафа L2 оптимизируется следующее выражение:

$$\vec{b}^* = \arg\min_{\vec{b}} \mid \vec{y} - X\vec{b} \mid^2 + \alpha \sum_i b_i^2$$

Различие, на первый взгляд, несущественное: штраф составляет не сумму абсолютных величин коэффициентов, а сумму их квадратов. Однако результаты могут отличаться очень сильно.

Гребневая регрессия, Lasso и эластичные сети

У моделей регуляризации довольно любопытные названия. L1-регуляризованную регрессию часто называют **Lasso**[1], а L2-регуляризованную – **гребневой** регрессией. При использовании того и другого получается модель **эластичной сети**.

Как Lasso, так и гребневая регрессия дают меньшие (по абсолютной величине, без учета знака) значения коэффициентов, чем регрессия без штрафа. Однако у метода Lasso есть дополнительное свойство – многие коэффициенты оказываются в точности равны нулю! Это означает, что в окончательной модели некоторые признаки вообще не используются, то есть модель получается **разреженной**. Часто это свойство весьма желательно, поскольку модель осуществляет не только регрессию, но и селекцию признаков.

Обратите внимание, что штраф сопровождается весовым множителем α, который контролирует величину штрафа. Если значение α близко к нулю, то получается модель, мало отличающаяся от нерегуляризованной регрессии (при $\alpha = 0$ это не что иное, как обычный метод наименьших квадратов), а при большом α результирующая модель очень сильно отличается от нерегуляризованной.

Гребневая модель появилась раньше, потому что Lasso очень трудно рассчитать с помощью карандаша и бумаги. Но на современных компьютерах Lasso обсчитывается так же просто, как гребневая регрессия, а при желании их можно даже объединить в алгоритме эластичной сети, в котором применяются два штрафа: сумма абсолютных величин и сумма квадратов, то есть требуется решить такое уравнение:

$$\vec{b}^* = \arg\min_{\vec{b}} \mid \vec{y} - X\vec{b} \mid^2 + \alpha_1 \sum_i |b_i| + \alpha_2 \sum_i b_i^2$$

Это выражение – комбинация двух предыдущих, с двумя параметрами α_1 и α_2. Ниже мы обсудим, как следует выбирать значения этих параметров.

Lasso и эластичная сеть в библиотеке scikit-learn

Модифицируем предыдущий пример для использования эластичной сети. При работе с библиотекой scikit-learn достаточно просто заменить класс регрессии на `ElasticNet`:

[1] Слово Lasso не имеет никакого отношения к аркану и потому не транслитерируется, это аббревиатура least absolute shrinkage and selection operator. – *Прим. перев.*

```
>>> from sklearn.linear_model import ElasticNet, Lasso
>>> en = ElasticNet(alpha=0.5)
```

Единственное изменение заключается в названии переменной: `en` вместо `lr`. Результаты вполне согласуются с ожиданиями. Погрешность на обучающих данных возросла до 5.0 (а была 4.6), но погрешность после перекрестной проверки снизилась до 5.4 (а была 5.6). Мы согласились на большую погрешность на обучающих данных в обмен на лучшую обобщаемость. Почти не меняя код, можно было попробовать также штраф L1, взяв класс `Lasso`, и штраф L2, взяв класс `Ridge`.

Визуализация пути в Lasso

Библиотека scikit-learn позволяет наглядно представить, что происходит при изменении параметра регуляризации (альфа). Снова обратимся к данным о домах в Бостоне, но теперь для регрессии возьмем класс `Lasso`:

```
>>> las = Lasso(normalize=1)
>>> alphas = np.logspace(-5, 2, 1000)
>>> alphas, coefs, _= las.path(x, y, alphas=alphas)
```

Для каждого значения из массива `alphas` метод `path` объекта `Lasso` возвращает коэффициенты, составляющие решение задачи при данном параметре. Поскольку зависимость результата от альфы гладкая, вычисления весьма эффективны.

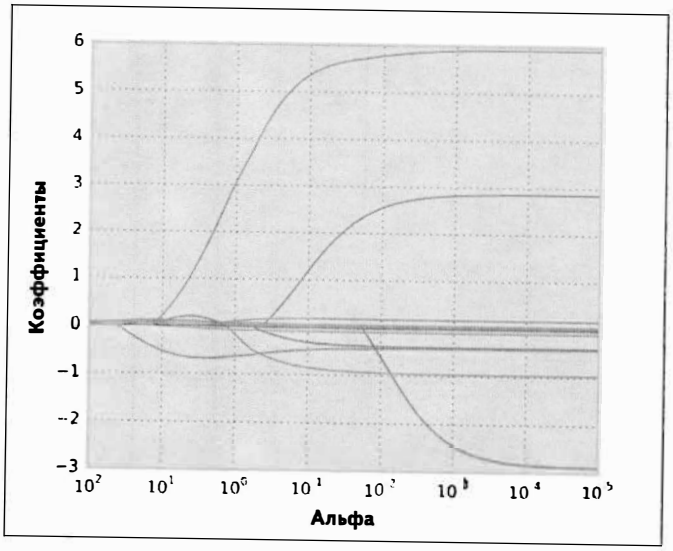
Чтобы визуализировать этот путь, обычно строят график зависимости коэффициентов от альфы:

```
>>> fig,ax = plt.subplots()
>>> ax.plot(alphas, coefs.T)
>>> # Задаем логарифмическую шкалу
>>> ax.set_xscale('log')
>>> # Уменьшаем alpha при движении слева направо
>>> ax.set_xlim(alphas.max(), alphas.min())
```

Получается график, который показан ниже (мы опустили тривиальный код рисования названий осей и меток на них):

Здесь по оси x отложены уменьшающиеся слева направо значения коэффициента регуляризации α. Каждая кривая описывает изменение одного коэффициента. Из графика видно, что когда регуляризация очень сильная (левая часть рисунка, большие значения α), наилучшее решение – положить все коэффициенты равными нулю. По мере ослабления регуляризации то один, то другой коэффициент сначала начинает резко изменяться, а потом стабилизируется. В какой-

то момент все они выходят на плато, потому что мы, вероятно, уже близки к нерегуляризованному решению.



Сценарии P-больше-N

Название этого раздела – специальный жаргонизм, с которым мы сейчас познакомимся. Начиная с 1990-х годов, сначала в медико-биологических исследованиях, а затем и в веб стали появляться задачи, в которых P больше N, то есть количество признаков P больше количества примеров N (этими буквами принято обозначать соответствующие понятия в статистике). Такие задачи получили название задач типа *P-больше-N*.

Например, если на входе имеется набор письменных документов, то проще всего считать признаком каждое слово из словаря и выполнить для них регрессию (ниже мы займемся такой задачей). Для английского языка слов получится больше 20 000 (если предварительно выполнить стемминг и учитывать только употребительные слова; в противном случае слов будет в 10 раз больше). Если документов-примеров всего несколько сотен или тысяч, то возникает ситуация, когда признаков больше, чем примеров.

Раз так, то можно точно подогнать модель по обучающие данные. Это математический факт, который от данных не зависит. По существу, мы решаем систему линейных уравнений, в которой уравнений

больше, чем переменных, и можно подобрать коэффициенты регрессии, так что погрешность на обучающих данных будет нулевой (таких идеальных решений даже не одно, а бесконечно много).

Однако – и это серьезная проблема – нулевая погрешность на обучающих данных не означает, что решение хорошо обобщается. Более того, его способность к обобщению может быть очень низкой. И хотя регуляризация может несколько замедлить скорость обучения, она становится абсолютно необходимой для получения осмысленного результата.

Пример, основанный на текстовых документах

Обратимся к примеру, который изучался группой профессора Ноя Смита в университете Карнеги-Меллон. Исследованию были подвергнуты так называемые формы 10-K, которые компании ежегодно отправляют в **Комиссию по ценным бумагам и биржам США (SEC)**. Это обязательное требование для всех компаний, зарегистрированных на бирже. Цель исследования формулировалась так: на основе имеющейся открытой информации спрогнозировать будущую волатильность акций компании. В качестве обучающих данных используются исторические данные, для которых ответ уже известен.

Всего имеется 16 087 примеров. Признаки, уже предварительно обработанные, – это различные слова, всего их 150 360. Таким образом, признаков гораздо больше, чем примеров, почти в десять раз. Во введении мы говорили, что в таких случаях обычный метод наименьших квадратов не годится, а теперь покажем, во что выливается попытка слепо применить его.

Набор данных в формате SVMLight можно скачать из разных источников, в том числе с сопроводительного сайта этой книги. Библиотека scikit-learn понимает этот формат. Как следует из названия, SVMLight – реализация метода опорных векторов, также поддерживаемого scikit-learn; но сейчас нас интересует только формат файла:

```
>>> from sklearn.datasets import load_svmlight_file
>>> data,target = load_svmlight_file('E2006.train')
```

Здесь `data` – разреженная матрица (большая часть ее элементов равна нулю, а в памяти хранятся только ненулевые элементы), а `target` – простой одномерный вектор. Начнем с изучения некоторых атрибутов `target`:

```
>>> print('Мин значение target: {}'.format(target.min()))
Мин значение target: -7.89957807347
>>> print('Макс значение target: {}'.format(target.max()))
Макс значение target: -0.51940952694
>>> print('Среднее значение target: {}'.format(target.mean()))
Среднее значение target: -3.51405313669
>>> print('Стандартное отклонение target: {}'.format(target.std()))
Стандартное отклонение target: 0.632278353911
```

Как видим, данные лежат в диапазоне от –7.9 до –0.5. Имея некоторое представление о данных, посмотрим, что произойдет при попытке использовать обычный метод наименьших квадратов для прогнозирования. При этом используются те же классы и методы, что и раньше.

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression()
>>> lr.fit(data,target)
>>> pred = lr.predict(data)
>>> rmse_train = np.sqrt(mean_squared_error(target, pred))
>>> print('RMSE на обучающих данных: {:.2}'.format(rmse_train))
RMSE на обучающих данных: 0.0025
>>> print('R2 на обучающих данных: {:.2}'.format(r2_score(target, pred)))
R2 на обучающих данных: 1.0
```

Среднеквадратичная погрешность не точно равна нулю из-за ошибок округления, но очень близка к нему. Коэффициент детерминации равен 1.0. То есть мы получаем линейную модель, которая дает точный прогноз на обучающих данных.

Но при использовании перекрестной проверки (код очень похож на тот, что был написан ранее для примера с домами в Бостоне) получается нечто совершенно иное: RMSE равно 0.75, что соответствует отрицательному коэффициенту детерминации –0.42. То есть даже если мы всегда будем «предсказывать» среднее значение –3.5, то все равно это будет лучше прогноза регрессионной модели!

Погрешность на обучающих данных и способность к обобщению

Если количество признаков больше количества примеров, то метод наименьших квадратов всегда дает нулевую погрешность на обучающих данных, если пренебречь ошибками округления. Но это редко означает, что модель способна к обобщению. На самом деле, можно получить нулевую погрешность на обучающих данных и абсолютно бесполезную модель.

Естественное решение – воспользоваться регуляризацией для противодействия переобучению. Мы можем воспользоваться тем же самым циклом перекрестной проверки, взяв объект класса `ElasticNet` и установив коэффициент регуляризации 0.1:

```
>>> from sklearn.linear_model import ElasticNet
>>> met = ElasticNet(alpha=0.1)

>>> kf = KFold(len(target), n_folds=5)
>>> pred = np.zeros_like(target)
>>> for train, test in kf:
...    met.fit(data[train], target[train])
...    pred[test] = met.predict(data[test])

>>> # Вычислить RMSE
>>> rmse = np.sqrt(mean_squared_error(target, pred))
>>> print('[EN 0.1] RMSE на обучающих данных (5 проходов): {:.2}'.format(rmse))
[EN 0.1] RMSE на обучающих данных (5 проходов): 0.4

>>> # Вычислить коэффициент детерминации
>>> r2 = r2_score(target, pred)
>>> print('[EN 0.1] R2 на обучающих данных (5 проходов): {:.2}'.format(r2))
[EN 0.1] R2 на обучающих данных (5 проходов): 0.61
```

Теперь RMSE равно `0.4`, а R2 `0.61` – гораздо лучше, чем выбор среднего значения в качестве прогноза. Но в этом решении есть одна проблема – выбор значения альфа. Если взять значение по умолчанию (`1.0`), то результат будет другим (хуже).

В данном случае мы немного смухлевали, потому что автор попробовал несколько значений, чтобы выбрать лучшее. Это неэффективно и может стать причиной ни на чем не основанной уверенности (поскольку мы анализируем тестовые данные, чтобы понять, какие значения параметров хороши, а какие никуда не годны). В следующем разделе объясняется, как сделать это правильно и как этот подход поддержан в scikit-learn.

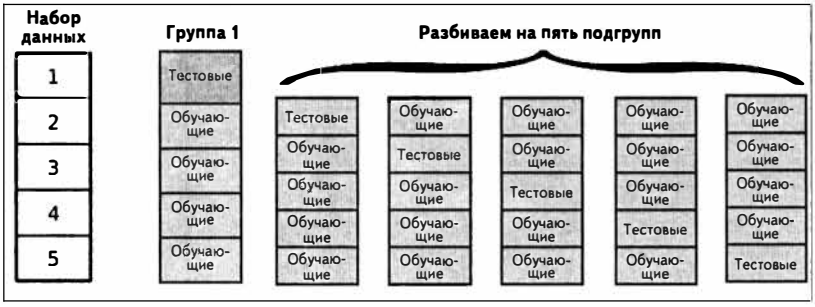
Объективный подход к заданию гиперпараметров

В предыдущем примере параметр регуляризации был взят равным 0.1. А можно было бы взять 0.7 или 23.9. Конечно, результаты будут меняться. Взяв чересчур большое значение, мы получим недообученную модель. В самом крайнем случае после обучения получится модель, в которой все коэффициенты нулевые. Напротив, если

выбрать слишком малое значение, то модель будет очень близка к полученной методом наименьших квадратов, то есть переобучена и неспособна к обобщению (что мы и видели выше).

Как же выбрать хорошее значение? Это общая проблема машинного обучения: задание параметров алгоритмов обучения. Решение – применить перекрестную проверку. Выбираем несколько возможных значений и с помощью перекрестной проверки оставляем лучшее. Конечно, требуются дополнительные вычисления (в пять раз больше, если применяется пятипроходная проверка), но метод всегда работает и дает объективные результаты.

Однако нужна предусмотрительность. Чтобы получить оценку обобщаемости, нам понадобится **два уровня перекрестной проверки**: на первом уровне оценивается обобщаемость, на втором выбираются хорошие параметры. Допустим, данные разбиты на пять групп. Сначала резервируем первую группу и обучаем на четырех оставшихся. Затем еще раз разбиваем оставшиеся данные на пять групп, чтобы выбрать параметры. Подобрав параметры, проводим тестирование на первой группе. И повторяем эту процедуру еще четыре раза.



На рисунке выше показано, как одна обучающая группа разбивается на подгруппы. То же самое нужно повторить для остальных групп. В данном случае получится пять внешних групп и пять внутренних, но нет никаких причин, почему внутренних и внешних групп должно быть поровну; можно разбивать как угодно, лишь бы группы не пересекались.

Конечно, объем вычислений резко возрастает, но это плата за корректное решение. Проблема в том, что если использовать часть данных для принятия любых решений о модели (в том числе о значениях параметров), то данные становятся «запачканными» и уже не могут использоваться для проверки способности модели к обобщению. Это

тонкий момент, который сразу не очевиден. И до сих пор многие исследователи, практикующие машинное обучение, совершают эту ошибку, а, значит, переоценивают качество своих систем – все из-за неправильного применения перекрестной проверки.

По счастью, библиотека scikit-learn позволяет сделать всё правильно; в ней есть классы `LassoCV`, `RidgeCV` и `ElasticNetCV`, в которых инкапсулирован внутренний цикл перекрестной проверки для оптимизации нужного параметра. Код почти не отличается от предыдущего, только для альфа не нужно задавать никакого значения:

```
>>> from sklearn.linear_model import ElasticNetCV
>>> met = ElasticNetCV()
>>> kf = KFold(len(target), n_folds=5)
>>> p = np.zeros_like(target)
>>> for train,test in kf:
...    met.fit(data[train],target[train])
...    p[test] = met.predict(data[test])
>>> r2_cv = r2_score(target, p)
>>> print(«R2 ElasticNetCV: {:.2}».format(r2_cv))
R2 ElasticNetCV: 0.65
```

Считаться будет долго, так что вы вполне успеете выпить чашечку кофе (конечно, все зависит от быстродействия вашего компьютера). Улучшить производительность можно, задействовав несколько процессоров. Эта возможность встроена в scikit-learn; чтобы воспользоваться ей, достаточно задать параметр `n_jobs` конструктора `ElasticNetCV`. Например, чтобы задействовать 4 процессора, напишите:

```
>>> met = ElasticNetCV(n_jobs=4)
```

Если положить `n_jobs` равным –1, то будут задействованы все имеющиеся процессоры:

```
>>> met = ElasticNetCV(n_jobs=-1)
```

Возможно, вы недоумеваете, почему мы задаем только одно значение альфа, если в алгоритме `ElasticNet` два штрафа, L1 и L2. На самом деле, оба значения становятся известны в результате задания альфа и переменной `l1_ratio`. Значения α_1 и α_2 вычисляются следующим образом (здесь буквой ρ обозначено `l1_ratio`):

$$\alpha_1 = \rho\,\alpha$$
$$\alpha_2 = (1-\rho)\,\alpha$$

Интуитивно понятно, что альфа задает общую величину регуляризации, а `l1_ratio` – соотношение между различными типами регуляризации, L1 и L2.

Можно потребовать, чтобы объект `ElasticNetCV` проверял различные значения `l1_ratio`, как показано в коде ниже:

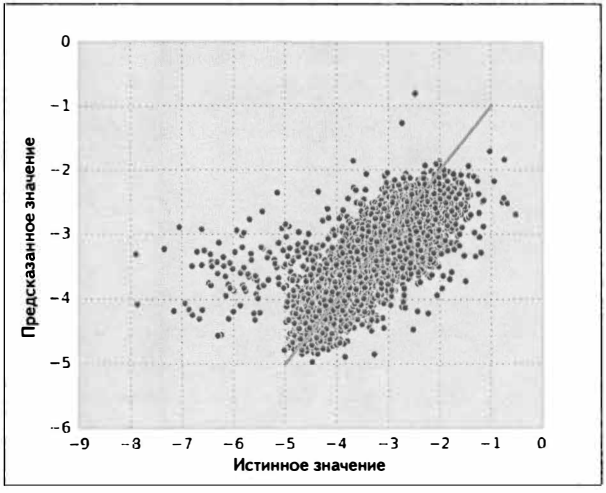
```
>>> l1_ratio=[.01, .05, .25, .5, .75, .95, .99]
>>> met = ElasticNetCV(l1_ratio=l1_ratio, n_jobs=-1)
```

Такой набор значений `l1_ratio` рекомендован в документации. При этом тестируются как модели, близкие к гребневой регрессии (когда `l1_ratio` равно 0.01 или 0.05), так и модели, близкие к Lasso (когда `l1_ratio` равно 0.95 или 0.99). Следовательно, будет рассмотрен весь диапазон вариантов.

Благодаря своей гибкости и умению задействовать несколько процессоров `ElasticNetCV` – отличное решение по умолчанию для задач регрессии в случае, когда нет причин предпочесть какую-то одну модель. Собрав все вместе, мы можем наглядно представить соотношение прогноза и реальных данных для этого большого набора:

```
>>> l1_ratio = [.01, .05, .25, .5, .75, .95, .99]
>>> met = ElasticNetCV(l1_ratio=l1_ratio, n_jobs=-1)
>>> p = np.zeros_like(target)
>>> for train,test in kf:
...    met.fit(data[train],target[train])
...    p[test] = met.predict(data[test])
>>> plt.scatter(p, y)
>>> # Добавить диагональ для сравнения)
>>> # (описывает точное согласие)
>>> plt.plot([p.min(), p.max()], [p.min(), p.max()])
```

Получается такой график:



Как видим, прогноз не очень хорош в нижнем левом углу. Быть может, это связано с тем, что в этой области примеров гораздо меньше (а, значит, этот недостаток негативно отразится лишь на небольшой доле данных).

И последнее замечание: использование внутреннего цикла перекрестной проверки для задания параметров поддерживается в scikit-learn и с применением поиска на сетке. Мы уже видели это в предыдущей главе.

Резюме

Мы начали эту главу с самого старого из всех рассмотренных в этой книге алгоритмов – регрессии обычным методом наименьших квадратов. Ему уже две сотни лет, но по-прежнему он зачастую является наилучшим для решения задачи регрессии. Но видели мы и более современные подходы, которые позволяют избежать переобучения и дают более точные результаты, особенно если количество признаков очень велико. В качестве примеров мы привели методы гребневой регрессии, Lasso и эластичных сетей.

Мы еще раз продемонстрировали, как опасно полагаться на погрешность на обучающих данных для оценки способности модели к обобщению: может оказаться, что погрешность нулевая, а модель, тем не менее, абсолютно бесполезна. Размышления над этими проблемами привели нас к двухуровневой перекрестной проверке – важной методике, которую еще не все работающие в области машинного обучения исследователи осознали и приняли на вооружение.

В этой главе мы смогли воспользоваться scikit-learn для реализации всего задуманного, в том числе и правильной перекрестной проверки. Мы рекомендуем по умолчанию использовать для регрессии эластичные сети с циклом внутренней перекрестной проверки (в scikit-learn это класс `ElasticNetCV`).

Что-то другое имеет смысл использовать, в частности, тогда, когда вам интересно разреженное решение. В таком случаем чистый алгоритм Lasso подходит лучше, потому что в результате его применения многие коэффициенты оказываются равны нулю. Он также позволяет определить по данным небольшое число переменных, оказывающих наибольшее влияние на результат. Это знание интересно и само по себе, помимо получения хорошей регрессионной модели.

В следующей главе мы займемся рекомендованием, еще одной задачей машинного обучения. Сначала мы воспользуемся регрессией для прогнозирования рейтингов потребительского продукта. А затем рассмотрим альтернативные модели выработки рекомендаций.

ГЛАВА 8. Рекомендации

Рекомендование стало одним из столпов услуг и торговли в Интернете. Такая автоматизированная система может предложить пользователю персонализированный список рекомендаций (например, перечень товаров, возможных признаков или новых социальных связей). В этой главе мы рассмотрим принципы работы автоматизированных систем генерации рекомендаций. Та часть этой дисциплины, которая изучает рекомендование на основе данных, введенных пользователями, называется коллаборативной фильтрацией, потому что пользователи при посредстве системы совместно (коллаборативно) ищут интересующие друг друга объекты.

В первой части главы мы увидим, как воспользоваться оценками, которые были поставлены пользователями ранее, для предсказания новых оценок. Начнем с нескольких полезных идей, а затем объединим их вместе. При этом мы будем использовать регрессию, чтобы найти наилучший способ объединения. И заодно изучим еще одну общую концепцию машинного обучения: обучение ансамбля моделей.

Во второй части главы мы познакомимся с другим подходом к обучению системы рекомендования: анализом корзины. В этом случае у нас не будет числовых оценок, а все, чем мы располагаем, – информация о содержимом корзин покупателей, то есть предметов, которые покупали вместе. А цель, как и раньше, – вывести отсюда рекомендации. Вы, наверное, встречали в разных Интернетах-магазинах фразу типа «купившие *X* покупали также *Y*». Мы разработаем аналогичную функциональность.

Прогноз и рекомендование оценок

Если в последние 10 лет вы хоть раз покупали что-нибудь в Интернете, то наверняка видели подобные рекомендации. В одних магазинах,

например в Amazon, пишут «покупатели, купившие *X*, купили также *Y*». Такие рекомендации мы рассмотрим ниже в разделе «Анализ корзины». Другие рекомендации основаны на прогнозировании оценки объекта, например кинофильма.

Задача обучения системы рекомендования по оценкам объектов получила известность благодаря призу в миллион долларов, учрежденному компанией Netflix. Компания Netflix (хорошо известная в США и Великобритании и осуществляющая экспансию на международный рынок) занимается прокатом кинофильмов. Раньше заказчик получал по почте DVD-диск, но в последнее время Netflix концентрируется на потоковом показе фильмов через Интернет и по телевизору. С самого начала служба отличалась тем, что позволяла пользователям оценивать просмотренные фильмы. Затем Netflix использовала эти оценки, чтобы рекомендовать клиентам другие фильмы. В этой задаче машинного обучения в нашем распоряжении имеется не только информация о том, какие фильмы смотрели пользователи, но и о том, как они их оценили.

В 2006 году Netflix собрала в своей базе данных большое число оценок фильмов, поставленных пользователями, и, сделав их общедоступными, объявила конкурс. Была поставлена задача улучшить применяемый компанией алгоритм прогнозирования оценок. Участник, которому удалось бы повысить его качество не менее чем на 10 процентов, получал приз в один миллион долларов. В 2009 году международный коллектив под названием BellKor's Pragmatic Chaos сумел превзойти алгоритм Netflix и получил приз. Они всего на 20 минут опередили другую команду, The Ensemble, которая также преодолела 10-процентный рубеж, – потрясающий финал соревнования, длившегося несколько лет.

Машинное обучение в реальном мире

О призе Netflix немало писали, и, читая эти материалы, можно многому научиться. Победителями оказались методики, в которых применялись как сложные алгоритмы машинного обучения, так и разнообразные действия по предварительной обработке данных. Например, одни пользователи ставят всем очень высокие оценки, тогда как другие постоянно их занижают. Если не принять такие вещи во внимание на этапе предварительной обработки, то пострадает качество модели. Для достижения хорошего результата необходимо было учитывать и другие факторы: как давно снят фильм и сколько всего оценок ему поставлено. Хорошие алгоритмы – вещь важная, но нельзя пренебрегать «грязной работой»: нужно обязательно настраивать алгоритмы под свойства имеющихся данных.

Предобработка и нормировка данных часто занимают львиную долю времени в процессе машинного обучения. Однако именно эти операции оказывают наибольший эффект на качество конечной системы.

Говоря о призе Netflix Prize, нужно, прежде всего, отметить сложность задачи. Грубо говоря, внутренняя система Netflix была всего на 10% лучше отсутствия всяких рекомендаций (когда каждому фильм присваивается оценка, равная среднему значению по всем пользователям). Цель состояла в том, чтобы улучшить эту рекомендацию еще на 10%. Победившая система оказалась примерно на 20% лучше отсутствия рекомендаций. А сколько времени и сил было положено на достижение этой цели! И хотя 20 процентов вроде бы не так много, получившаяся система практически полезна.

К сожалению, по причинам юридического характера этот набор данных уже недоступен. Несмотря на анонимность данных, возникли опасения, что на их основе можно определить личности клиентов и узнать конфиденциальные сведения об аренде фильмов. Однако мы можем воспользоваться академическим набором данных с аналогичными характеристиками. Этот набор подготовлен исследовательской группой GroupLens из Миннесотского университета.

Как ответить на поставленный Netflix вопрос о прогнозировании оценок? Мы рассмотрим два подхода: на основе ближайших соседей и регрессионный. Мы также покажем, как объединить оба подхода для получения единого прогноза.

Разделение данных на обучающие и тестовые

Принципиально разбиение набора данных на обучающие и тестовые для получения объективной оценки качества системы производится так же, как в предыдущих главах: резервируем какую-то часть данных (скажем, 10 процентов) для тестирования, а остальные используем для обучения. Но поскольку в этом контексте структура данных отличается, код тоже будет другим. Первым делом загрузим данные с диска, воспользовавшись следующей функцией:

```
def load():
  import numpy as np
  from scipy import sparse

  data = np.loadtxt('data/ml-100k/u.data')
```

```
    ij = data[:, :2]
    ij -= 1  # индексация исходных данных начинается с 1
    values = data[:, 2]
    reviews = sparse.csc_matrix((values, ij.T)).astype(float)
    return reviews.toarray()
```

Отметим, что нулевые элементы этой матрицы соответствуют отсутствующим оценкам.

```
>>> reviews = load()
>>> U,M = np.where(reviews)
```

Далее с помощью стандартного модуля `random` выберем индексы, резервируемые для тестирования:

```
>>> import random
>>> test_idxs = np.array(random.sample(range(len(U)), len(U)//10))
```

Теперь построим матрицу `train`, которая отличается от `reviews` тем, что элементы, соответствующие тестовым данным, равны нулю:

```
>>> train = reviews.copy()
>>> train[U[test_idxs], M[test_idxs]] = 0
```

И матрицу `test`, содержащую только тестовые данные:

```
>>> test = np.zeros_like(reviews)
>>> test[U[test_idxs], M[test_idxs]] = reviews[U[test_idxs],
M[test_idxs]]
```

Начиная с этого момента, мы будем работать только с обучающими данными и попытаемся предсказать значения, отсутствующие в наборе данных. То есть напишем код, который сопоставляет каждой паре (пользователь, фильм) рекомендацию.

Нормировка обучающих данных

Как мы уже говорили, лучше нормировать данные, чтобы устранить очевидные особенности, характерные для фильмов или пользователей. Мы применим только очень простой вид нормировки, которым пользовались и раньше: преобразование к z-оценкам.

К сожалению, мы не можем просто воспользоваться нормировочными объектами scikit-learn, так как должны учитывать возможность отсутствия некоторых значений (из-за того, что не каждый пользователь оценивает все фильмы). Поэтому требуется нормировать с учетом среднего и стандартного отклонения тех значений, которые присутствуют.

Напишем собственный класс, который будет игнорировать отсутствующие значения, следуя определенному в scikit-learn API предварительной обработки:

```
class NormalizePositive(object):
```

Нужно выбрать ось нормировки. По умолчанию нормировка производится по первой оси, но иногда полезнее нормировать по второй. Это соглашение, принятое во многих функциях, относящихся к NumPy:

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

Самый важный метод – fit. Мы будем вычислять среднее и стандартное отклонение по значениям, отличным от нуля. Напомним, что нулями представлены «отсутствующие значения»:

```
def fit(self, features, y=None):
```

Если параметр `axis` равен 1, то мы будем работать с транспонированным массивом:

```
if self.axis == 1:
  features = features.T
# подсчитать, сколько признаков больше 0 на оси 0
binary = (features > 0)
count0 = binary.sum(axis=0)

# во избежание деления на 0 установить нулевые счетчики в единицу
count0[count0 == 0] = 1.

# вычислить среднее нетрудно
self.mean = features.sum(axis=0)/count0

# рассматриваем только разности, для которых binary равно True
diff = (features - self.mean) * binary
diff **= 2

# регуляризировать оценку стандартного отклонения, прибавив 0.1
self.std = np.sqrt(0.1 + diff.sum(axis=0)/count0)
return self
```

Мы прибавили 0.1 к вычисленной оценке стандартного отклонения, чтобы предотвратить недооценку его значения в случае, когда есть всего несколько образцов и все они равны. Точное значение не играет существенной роли для конечного результата, но необходимо избежать деления на нуль.

Метод `transform` должен позаботиться о корректности структуры `binary`:

```
def transform(self, features):
  if self.axis == 1:
     features = features.T
  binary = (features > 0)
  features = features - self.mean
  features /= self.std
  features *= binary
  if self.axis == 1:
     features = features.T
  return features
```

Обратите внимание, что в случае, когда `axis` равно 1, мы транспонируем входную матрицу, а затем транспонируем ее еще раз, чтобы возвращаемое значение имело ту же форму, что входное. Метод `inverse_transform` выполняет обратное транспонирование:

```
def inverse_transform(self, features, copy=True):
  if copy:
     features = features.copy()
  if self.axis == 1:
     features = features.T
  features *= self.std
  features += self.mean
  if self.axis == 1:
     features = features.T
  return features
```

Наконец, метод `fit_transform` объединяет операции `fit` и `transform`:

```
def fit_transform(self, features):
  return self.fit(features).transform(features)
```

Определенные нами методы (`fit`, `transform`, `transform_inverse` и `fit_transform`) – те же самые, что в классах из модуля `sklearn.preprocessing`. В следующих разделах мы сначала нормируем входные данные, затем сгенерируем нормированные предсказания и, наконец, применим обратное преобразование, чтобы получить окончательные предсказания.

Рекомендование на основе ближайших соседей

Понятие соседства можно трактовать двумя способами: соседние пользователи и соседние фильмы. Для пользователей понятие соседства реализуется очень просто: чтобы узнать, как некий пользователь оценит фильм, нужно найти самых похожих на него пользователей и посмотреть, какие оценки поставили они. Мы сейчас займемся этой

идеей, а в конце этого раздела обсудим, как модифицировать код, чтобы вычислять соседние фильмы.

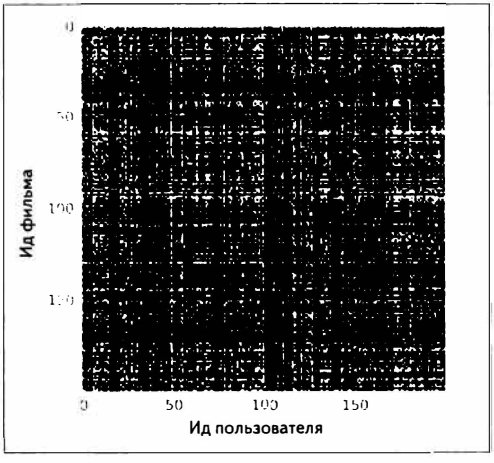
Интересная мысль – посмотреть, какие фильмы оценивал каждый пользователь, причем сама оценка нас мало интересует. Даже имея лишь двоичную матрицу, в которой элемент равен 1, если пользователь оценивал фильм, и 0, если не оценивал, мы можем сделать полезные предсказания. И если подумать, то это вполне логично: мы же решаем, какой фильм посмотреть, не совсем случайно, а стараемся выбирать те, что могут нам понравиться. И решение о том, какие фильмы оценивать, не произвольно – обычно мы оцениваем те фильмы, которые вызвали у нас сильные эмоции (конечно, бывают исключения, но в среднем это, пожалуй, так).

Мы можем визуализировать матрицу, изобразив каждую оценку в виде квадратика. Черным цветом обозначим отсутствие оценки, а градациями серого – значение оценки.

Код такой визуализации данных очень прост (можете модифицировать его, чтобы показать большую часть матрицы, чем поместилось на печатной странице).

```
>>> from matplotlib import pyplot as plt
>>> # Строим объект, определенный выше
>>> norm = NormalizePositive(axis=1)
>>> binary = (train > 0)
>>> train = norm.fit_transform(train)
>>> # из-за ограниченности места рисуем только область размером 200x200
>>> plt.imshow(binary[:200, :200], interpolation='nearest')
```

В результате получается такая картина:



Как видим, матрица разреженная – большинство квадратиков черные. Видно также, что некоторые пользователи оценивают гораздо больше фильмов, чем остальные, и что некоторые фильмы получают больше оценок.

Теперь мы воспользуемся этой двоичной матрицей, чтобы спрогнозировать оценки фильма. Алгоритм выглядит следующим образом.

1. Для каждого пользователя ранжировать всех остальных по близости к нему. На этом шаге мы воспользуемся двоичной матрицей и в качестве меры близости возьмем корреляцию (интерпретируя матрицу как состоящую из нулей и единиц, мы сможем выполнить это вычисление).
2. Чтобы предсказать оценку для пары (пользователь, фильм), мы возьмем всех пользователей, оценивших данный фильм, и разделим их на две равные группы: самые похожие и самые непохожие. Предсказанием будет средняя оценка по первой группе.

Для построения матрицы расстояний между пользователями воспользуемся функцией `scipy.spatial.distance.pdist`. Она возвращает корреляционное расстояние, инвертируя величину корреляции таким образом, что чем менее похожи числа, тем больше расстояние между ними. Математически корреляционное расстояние равно `1 - r`, где `r` – величина корреляции. Код приведен ниже.

```
>>> from scipy.spatial import distance
>>> # вычислить попарные расстояния
>>> dists = distance.pdist(binary, 'correlation')
>>> # Преобразовать в квадратную матрицу, в которой dists[i,j] -
>>> # расстояние между binary[i] и binary[j]
>>> dists = distance.squareform(dists)
```

Теперь с помощью этой матрицы можно выбрать ближайших соседей каждого пользователя:

```
>>> neighbors = dists.argsort(axis=1)
```

Затем перебираем всех пользователей, чтобы предсказать оценки каждого фильма:

```
>>> # Заполняем матрицу результатов
>>> filled = train.copy()
>>> for u in range(filled.shape[0]):
      # n_u - соседи пользователя
        n_u = neighbors[u, 1:]
        # t_u - обучающие данные

        for m in range(filled.shape[1]):
```

```
...         # получить релевантные оценки по порядку!
...         revs = [train[neigh, m]
...             for neigh in n_u
...                 if binary [neigh, m]]
...         if len(revs):
...             # n - количество оценок для данного фильма
...             n = len(revs)
...             # рассматриваем половину оценок плюс 1
...             n //= 2
...             n += 1
...             revs = revs[:n]
...             filled[u,m] = np.mean(revs )
```

Нетривиальная часть этого кода – индексирование подходящими значениями, чтобы выбрать только тех соседей, которые оценили данный фильм. Затем мы выбираем половину, ближайшую к пользователю (строка `rev[:n]`) и по ней производим усреднение. Поскольку для одних фильмов оценок много, а для других мало, не всегда возможно отобрать одинаковое число пользователей для всех случаев. Выбор половины имеющихся данных – более общий подход.

Для получения конечного результата нужно денормировать предсказания:

```
>>> predicted = norm.inverse_transform(filled)
```

Можно использовать те же метрики, что в предыдущих главах:

```
>>> from sklearn import metrics
>>> r2 = metrics.r2_score(test[test > 0], predicted[test > 0])
>>> print('Оценка R2 (двоичные соседи): {:.1%}'.format(r2))
Оценка R2 (двоичные соседи): 29.5%
```

Выше мы вычисляли результат на основе идеи о соседних пользователях, но можно рассматривать и соседние фильмы, для этого достаточно просто транспонировать входную матрицу. На самом деле, программа вычисляет соседей для объектов, расположенных в строках входной матрицы, чем бы они ни были.

Перезапустим эту программу, добавив в начало следующую строку:

```
>>> reviews = reviews.T
>>> # тот же код, что и раньше ...
>>> r2 = metrics.r2_score(test[test > 0], predicted[test > 0])
>>> print('Оценка R2 (двоичные соседи-фильмы): {:.1%}'.format(r2))
Оценка R2 (двоичные соседи-фильмы): 29.8%
```

Как видим, результаты мало отличаются. В репозитории кода из книги вычисление соседей обернуто простой функцией, чтобы его было проще использовать повторно.

Регрессионный подход к рекомендованию

Можно сформулировать задачу рекомендования и как задачу регрессии и применить методы из предыдущей главы.

Заодно обсудим, почему эту задачу не следует рассматривать как задачу классификации. Конечно, можно было бы попытаться обучить модель с пятью классами, по одному для каждой возможной оценки. Но возникают две проблемы.

- Возможные ошибки далеко не равноценны. Например, если по ошибке оценить 5-звездочный фильм на 4 звезды, то это не так страшно, как присвоить ему одну звезду.
- Промежуточные значения также осмыслены. Даже если на входе допустимы только целые числа, то предсказание 4.3 имеет смысл. Мы понимаем, что оно отлично от 3.5, хотя округление в обоих случаях дает 4.

В совокупности эти два соображения показывают, что классификация – не лучший подход к решению этой задачи. А вот регрессия подходит.

Как и раньше, у нас есть выбор: построить модель, ориентированную на фильмы или на пользователей. Сначала построим модель, ориентированную на пользователей. Это означает, что для каждого пользователя в качестве целевой переменной мы возьмем оцененные им фильмы. Входами будут оценки, поставленные другими пользователями. Наша гипотеза состоит в том, что таким образом мы получим высокие значения для пользователей, похожих на выбранного (и отрицательное значение для тех пользователей, которым нравятся именно те фильмы, которые выбранному не нравятся).

Матрица `train` и `test` создаются, как и раньше (включая нормировку). Поэтому сразу перейдем к этапу обучения. Сначала создадим объект для выполнения регрессии:

```
>>> reg = ElasticNetCV(alphas=[
            0.0125, 0.025, 0.05, .125, .25, .5, 1., 2., 4.])
```

Мы хотим построить матрицу, содержащую оценки для каждой пары (пользователь, фильм). Инициализируем ее копией обучающих данных:

```
>>> filled = train.copy()
```

Теперь перебираем всех пользователей и для каждого обучаем регрессионную модель на тех данных, которые получены от других пользователей:

```
>>> for u in range(train.shape[0]):
...     curtrain = np.delete(train, u, axis=0)
...     # В binary хранится информация о присутствии оценки
...     bu = binary[u]
...     # Обучим текущего пользователя на данных обо всех остальных
...     reg.fit(curtrain[:,bu].T, train[u, bu])
...     # Добавим отсутствующие оценки
...     filled[u, ~bu] = reg.predict(curtrain[:,~bu].T)
```

Оценивание качества методики производится, как и раньше:

```
>>> predicted = norm.inverse_transform(filled)
>>> r2 = metrics.r2_score(test[test > 0], predicted[test > 0])
>>> print('Оценка R2 (регрессия по пользователям): {:.1%}'.format(r2))
Оценка R2 (регрессия по пользователям): 32.3%
```

Этот код также можно модифицировать, чтобы регрессия производилась по фильмам, для этого достаточно транспонировать матрицу.

Комбинирование нескольких методов

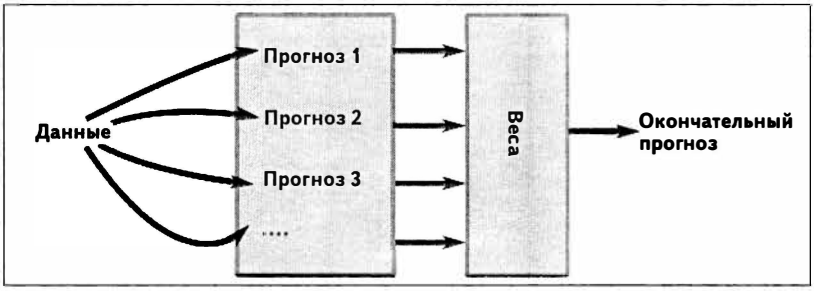
Теперь объединим оба описанных выше метода в один. Интуитивно идея представляется заманчивой, но как это сделать практически? Первое, что приходит на ум, – усреднить оба прогноза. Возможно, и получится приличный результат, но нет никаких оснований полагать, что все прогнозы должны считаться равноценными. Один может быть лучше, другой – хуже.

Можно попробовать взвешенное среднее, когда каждый прогноз умножается на какой-то вес, а потом результаты суммируются. Но как найти наилучшие веса? Ну, разумеется, путем обучения на данных!

Обучение ансамбля

Речь здесь идет об общей технике машинного обучения, применимой не только к регрессии: обучении ансамбля моделей. Мы обучаем ансамбль (то есть набор) прогностических моделей – предикторов, а затем комбинируем их для получения одного результата. Интересно, что каждый прогноз можно рассматривать как новый признак, а задача заключается в том, чтобы как-то скомбинировать признаки на основе обучающих данных, но именно этим мы всю дорогу и занимаемся. И хотя сейчас мы выполняем регрессию, те же рассуждения применимы и к классификации: обучаем несколько классификаторов, а затем главный классификатор, который получает частичные результаты и возвращает окончательный прогноз. Существуют разные формы обучения ансамбля, определяемые способом комбинирования предикторов.

Для комбинирования методов мы воспользуемся техникой **послойного обучения** (stacked learning). Идея заключается в том, чтобы обучить набор предикторов, а затем использовать их результаты как признаки для еще одного предиктора. Можно даже завести несколько слоев и обучать каждый следующий слой на результатах предыдущего. Взгляните на рисунок ниже:



Чтобы обучить эту комбинированную модель, мы разделим обучающий набор на два. Можно было бы вместо этого использовать перекрестную проверку (первоначально модель послойного обучения так и работала). Но в данном случае данных достаточно для получения хороших оценок, даже при условии, что часть данных зарезервирована для тестирования.

Но, как и при подборе гиперпараметров, нам понадобится два уровня разделения на обучающие и тестовые данные: верхний и нижний, на котором мы разбиваем обучающие данные на две части, чтобы обучить вложенный предиктор:

```
>>> train,test = load_ml100k.get_train_test(random_state=12)
>>> # Now split the training again into two subgroups
>>> tr_train,tr_test = load_ml100k.get_train_test(train,
    random_state=34)
>>> # Вызываем все ранее определенные методы:
>>> # они реализованы в виде функций
>>> tr_predicted0 = regression.predict(tr_train)
>>> tr_predicted1 = regression.predict(tr_train.T).T
>>> tr_predicted2 = corrneighbours.predict(tr_train)
>>> tr_predicted3 = corrneighbours.predict(tr_train.T).T
>>> tr_predicted4 = norm.predict(tr_train)
>>> tr_predicted5 = norm.predict(tr_train.T).T

>>> # Теперь собираем их прогнозы в один массив
>>> stack_tr = np.array([
...    tr_predicted0[tr_test > 0],
...    tr_predicted1[tr_test > 0],
```

```
...    tr_predicted2[tr_test > 0],
...    tr_predicted3[tr_test > 0],
...    tr_predicted4[tr_test > 0],
...    tr_predicted5[tr_test > 0],
...    ]).T
>>> # Обучаем простую модель линейной регрессии
>>> lr = linear_model.LinearRegression()
>>> lr.fit(stack_tr, tr_test[tr_test > 0])
```

Теперь применим модель к тестовым данным и оценим качество:

```
>>> stack_te = np.array([
...    tr_predicted0.ravel(),
...    tr_predicted1.ravel(),
...    tr_predicted2.ravel(),
...    tr_predicted3.ravel(),
...    tr_predicted4.ravel(),
...    tr_predicted5.ravel(),
...    ]).T
>>> predicted = lr.predict(stack_te).reshape(train.shape)
```

Качество оцениваем, как и раньше:

```
>>> r2 = metrics.r2_score(test[test > 0], predicted[test > 0])
>>> print('R2 послойной модели: {:.2%}'.format(r2))
R2 послойной модели: 33.15%
```

Результат послойного обучения лучше, чем у любого отдельного метода. Это типичное явление: комбинирование методов дает небольшое повышение качества, но каких-то потрясающих прорывов ждать не стоит.

Имея гибкий способ объединения нескольких методов, мы можем апробировать любую пришедшую в голову идею, добавив новый предиктор и поручив системе использовать его для прогнозирования. Можно, например, заменить критерий близости в алгоритме ближайших соседей.

Однако нужно следить за тем, чтобы не получить переобученную модель. Если случайным образом пробовать слишком много разных подходов, то некоторые будут хорошо работать на имеющемся наборе данных, но не будут обобщаться. Хотя мы и разделяем данные на части, строгой перекрестной проверке принятые решения не подвергаются. Если вы хотите получить хорошую модель и располагаете достаточным количеством примеров, то отложите часть данных в сторонку, чтобы протестировать на них окончательную модель перед внедрением ее в эксплуатацию. Тестирование на зарезервированных данных даст объективную оценку ожидаемого качества работы модели на реальных данных.

Анализ корзины

Выше мы обсуждали методы, которые хорошо работают, если имеются числовые оценки объекта пользователями. Но такая информация не всегда доступна, поскольку ее сбор требует от пользователей активных действий.

Анализ корзины – альтернативный способ обучения системы рекомендования. В этом режиме нам известно лишь, какие товары покупали вместе, но мы ничего не знаем о том, понравились они покупателю или нет. Даже если покупатель иногда сожалеет о покупке, в среднем, зная, что люди покупают, мы уже имеем достаточно информации для выработки хороших рекомендаций. Часто собрать такие данные легче, чем оценки, поскольку многие пользователи оценок не ставят, тогда как создание корзины – непременный побочный эффект акта покупки. На рисунке ниже показана веб-страница сайта Amazon.com для романа Толстого «Война и мир», из которой видно, как такие данные обычно используются:

Естественно, этот способ обучения применим не к одним лишь корзинам покупок, а в любой ситуации, когда имеется группа объектов и требуется рекомендовать еще один. Например, Gmail рекомендует дополнительных адресатов пользователю, который пишет письмо, и это можно было бы реализовать похожим образом (мы не знаем, как это делает Gmail; возможно, комбинируются несколько методов, как в предыдущем разделе). Или можно было бы написать приложение, которое рекомендует, какие страницы посетить, исходя из вашей истории просмотра. А если речь идет о покупках, то, возможно, имеет смысл объединить все приобретенные покупателем товары в одной корзине вне зависимости от того, покупались ли они за один или не-

сколько раз. Все зависит от ситуации, главное – помните, что это гибкая методика, которую можно с пользой применить в разных задачах.

Пиво и подгузники

Говоря об анализе корзины, часто вспоминают историю о пиве и подгузниках. Известно, что когда супермаркеты только начинали анализировать собранные ими данные, оказалось, что подгузники часто покупают вместе с пивом. Предположительно так поступали папаши, которые заходили за подгузниками, а заодно прихватывали бутылочку-другую пива. Много спорили, правда это или просто городской фольклор. В данном случае похоже на правду. В начале 1990-х годов Оско Драг обнаружил, что ранним вечером пиво и подгузники действительно покупают вместе, чем сильно удивил менеджеров, до тех пор не видевших между этими товарами никакого сходства. А неправда, что это открытие побудило владельцев магазинов приблизить пиво к секции подгузников. И, кроме того, мы так и не знаем, только ли отцы покупают пиво и подгузники вместе, или это характерно и для матерей (а равно дедушек и бабушек).

Получение полезных прогнозов

Речь идет не о том, что «купившие *X* часто покупают также *Y*», хотя именно эту фразу часто можно встретить на сайтах Интернет-магазинов (см. пример сайта Amazon.com выше); реальная система так работать не может. Почему? Потому что такую систему вводили бы в заблуждение любые часто покупаемые товары, и она просто рекомендовала бы популярные товары без какой бы то ни было персонализации.

Например, многие покупают в супермаркете хлеб всякий раз, когда оказываются поблизости (для определенности предположим, что хлеб входит в 50% покупок). Поэтому если взять любой другой товар, скажем средство для мытья посуды, и посмотреть, что часто покупают вместе с ним, то окажется, что это хлеб. Действительно, в среднем в половину случаев человек, купивший средство для мытья посуды, покупает и хлеб. Но хлеб так же часто покупают с любым другим товаром, просто потому что его вообще покупают часто.

В действительности же нас интересует такое утверждение: «покупатели, купившие *X*, статистически с большей вероятностью купят *Y*, чем средний покупатель, не купивший *X*». Если вы покупаете средство для мытья посуды, то вы, вероятно, купите и хлеб, но не с большей вероятностью, чем любой другой покупатель. Точно так же,

книжный магазин, который просто рекомендует бестселлеры вне зависимости от того, что вы покупали раньше, вряд ли можно назвать хорошим персональным рекомендателем.

Анализ корзин покупок в супермаркете

В качестве примера рассмотрим набор данных, состоящий из анонимных покупок в одном бельгийском супермаркете. Этот набор подготовил Том Брийс (Tom Brijs) из Хассельтского университета. Для сохранения конфиденциальности все персональные данные исключены, то есть имеется только купленное количество каждого товара, а корзина представляет собой набор чисел. Этот файл данных можно скачать из разных источников, в том числе с сопроводительного сайта книги.

Для начала загрузим набор данных и немного познакомимся с его статистическими характеристиками (это всегда полезно).

```
>>> from collections import defaultdict
>>> from itertools import chain

>>> # Скачанный файл сжат
>>> import gzip
>>> # Каждой покупке отведена в файле одна строка
>>> # вида '12 34 342 5...'
>>> dataset = [[int(tok) for tok in line.strip().split()]
...     for line in gzip.open('retail.dat.gz')]
>>> # Удобнее работать с множествами
>>> dataset = [set(d) for d in dataset]
>>> # Подсчитаем, сколько раз покупали каждый товар
>>> counts = defaultdict(int)
>>> for elem in chain(*dataset):
...     counts[elem] += 1
```

Результаты сведены в следующую таблицу:

Сколько раз куплено	Количество товаров
1 раз	2224
2 – 3	2438
4 – 7	2508
8 – 15	2251
16 – 31	2182
32 – 63	1940
64 – 127	1523
128 – 511	1225
512 и более	179

Многие товары покупали всего несколько раз. Например, 33% товаров покупали не более четырех раз. Однако они составляют всего 1 процент покупок. Явление, заключающееся в том, что многие товары покупают малое число раз, называют *длинным хвостом*, оно стало встречаться чаще, после того как Интернет удешевил складирование и продажу нишевых товаров. Чтобы дать рекомендации по таким товарам, нам нужно гораздо больше данных.

Существует несколько открытых реализаций алгоритмов анализа корзины, но ни один не интегрирован в библиотеку scikit-learn и другие используемые нами пакеты. Поэтому мы сами реализуем классический алгоритм Apriori, немного устаревший (он был опубликован в 1994 году в работе Ракеша Агравала (Rakesh Agrawal) и Рамакришнана Сриканта (Ramakrishnan Srikant)), но все еще работающий (разумеется, алгоритмы никогда не перестают работать, просто со временем им на смену приходят более удачные).

Формально говоря, алгоритм Apriori принимает набор множеств (корзин покупок) и возвращает множества, которые часто встречаются в качестве подмножеств (то есть предметы, которые входят вместе во многие корзины).

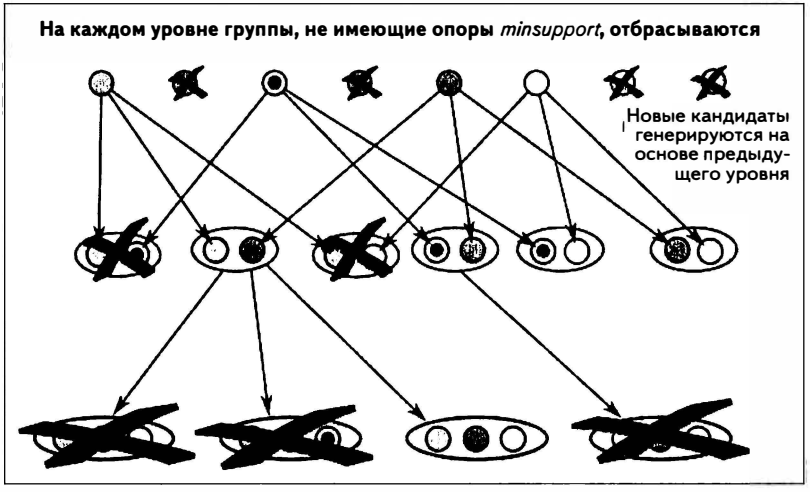
Алгоритм работает снизу вверх: начав с наименьших кандидатов (состоящих из одного элемента), он добавляет по одному элементу за раз. Формально алгоритм принимает набор корзин и минимальный допустимый размер (параметр, который мы назовем *минимальной опорой* – minsupport). На первом шаге рассматриваются все корзины, содержащие всего один элемент с минимальной опорой. Затем они комбинируются всеми возможными способами для построения двухэлементных корзин. Из этих корзин мы оставляем только те, что имеют минимальную опору. Далее рассматриваются всевозможные трехэлементные корзины и оставляются только имеющие минимальную опору. И так далее. Идея Apriori заключается в том, что при построении большей корзины *нужно рассматривать только те, что построены из меньших множеств*.

На рисунке ниже схематически изображена работа алгоритма.

Теперь напишем код, реализующий этот алгоритм. Определим минимальную опору:

```
>>> minsupport = 80
```

Опорой набора предметов называется количество раз, когда этот набор участвовал в одной покупке. Цель алгоритма Apriori – найти наборы с высокой опорой. При этом соблюдается такое логическое



свойство: любой набор предметов с опорой больше минимальной может быть составлен только из предметов, опора которых не меньше минимальной:

```
>>> valid = set(k for k,v in counts.items()
...          if (v >= minsupport))
```

Первоначально все наборы предметов одноэлементные. А одноэлементные наборы с опорой не ниже минимальной – это частые предметные наборы:

```
>>> itemsets = [frozenset([v]) for v in valid]
```

Описанный выше цикл можно закодировать следующим образом:

```
>>> freqsets = []
>>> for i in range(16):
...  nextsets = []
...    tested = set()
...    for it in itemsets:
...       for v in valid:
...          if v not in it:
...            # Создаем нового кандидата, добавляя в него v
...            c = (it | frozenset([v]))
...            # смотрим, проверялся ли он раньше
...            if c in tested:
...                  continue
...            tested.add(c)
...
...            # Вычисляем опору, обходя набор данных в цикле
```

```
...             # Этот шаг работает медленно.
...             # В файле 'apriori.py' приведена более быстрая реализация.
...             support_c = sum(1 for d in dataset if d.issuperset(c))
...             if support_c > minsupport:
...                     nextsets.append(c)
...     freqsets.extend(nextsets)
...     itemsets = nextsets
...     if not len(itemsets):
...         break
>>> print("Готово!")
Готово!
```

Этот код работает, но медленно. Его можно улучшить, избежав перебора в цикле всех наборов данных для вычисления опоры `support_c`. В частности, можно запоминать корзины с частыми предметными наборами. Это ускорит цикл, но код станет менее понятным. Поэтому здесь мы его не приводим, но на сайте книги имеются обе реализации. Мы также обернули код функцией, чтобы его можно было применить к другим наборам данных.

Алгоритм Apriori возвращает частые предметные наборы, то есть корзины, встречающиеся чаще заданного порогового значения (переменная `minsupport` в коде).

Поиск ассоциативных правил

Сами по себе частые предметные наборы не особенно полезны. Следующий шаг – построить **ассоциативные правила**. Имея в виду эту цель, всю задачу анализа корзины часто называют *выявлением ассоциативных правил*.

Ассоциативным правилом называется утверждение вида «если X, то Y», например: «если покупатель купил "Войну и мир", то он купит и "Анну Каренину"». Отметим, что правило не является детерминированным (не все покупатели, купившие X, покупают и Y), но каждый раз говорить «если покупатель купил X, то с вероятностью, превышающей базовую, он купит и Y» долго, поэтому мы говорим просто «если X, то Y», подразумевая вероятностную интерпретацию.

Кстати, посылка и заключение могут содержать и несколько объектов: покупатели, купившие X, Y и Z, покупают также A, B и C. Множественная посылка иногда позволяет давать более точные прогнозы, чем единичная.

Получить правило из частого набора можно, перепробовав все возможные комбинации X и Y, для которых из X следует Y. Нетрудно сгенерировать много таких правил. Но нас интересуют только полезные,

а, значит, нужно научиться измерить ценность правила. Часто в качестве меры используют **подъем**. Так называется отношение вероятности при условии применения данного правила к базовой вероятности:

$$\text{lift}(X \rightarrow Y) = \frac{P(Y \mid X)}{P(Y)}$$

В этой формуле $P(Y)$ – доля покупок, включающих предмет Y, а $P(Y|X)$ – доля покупок, включающих Y при условии, что они включают также и X. Применение подъема помогает избежать проблемы рекомендования бестселлеров. Для бестселлера и $P(Y)$, и $P(Y|X)$ будут велики, следовательно, подъем близок к единице, и правило следует считать нерелевантным. На практике желательно, что подъем был не меньше 10, а лучше ближе к 100.

Взгляните на следующий код:

```
>>> minlift = 5.0
>>> nr_transactions = float(len(dataset))
>>> for itemset in freqsets:
...  for item in itemset:
...       consequent = frozenset([item])
...       antecedent = itemset-consequent
...       base = 0.0
...       # acount: счетчик посылок
...       acount = 0.0
...
...       # ccount : счетчик следствий
...       ccount = 0.0
...       for d in dataset:
...          if item in d: base += 1
...          if d.issuperset(itemset): ccount += 1
...          if d.issuperset(antecedent): acount += 1
...       base /= nr_transactions
...       p_y_given_x = ccount/acount
...       lift = p_y_given_x / base
...       if lift > minlift:
...          print('У правила {0} -> {1} подъем равен {2}'
...            .format(antecedent, consequent,lift))
```

В следующей таблице приведены некоторые результаты. Счетчики – это количество покупок, которые включают **только предметы, перечисленные в следствии**, то есть в правой части импликации (базовая частота покупки данного предмета), **все предметы, перечисленные в посылке** (левой части импликации), и **все предметы, встречающиеся в посылке и следствии**.

Посылка	Следствие	Счетчик следствий	Счетчик посылок	Счетчик посылок и следствий	Подъем
1378, 1379, 1380	1269	279 (0.3%)	80	57	225
48, 41, 976	117	1026 (1.1%)	122	51	35
48, 41, 1, 6011	16010	1316 (1.5%)	165	159	64

Например, мы видим, что в 80 случаях предметы 1378, 1379 и 1380 были куплены вместе. Из них, в 57 покупках присутствовал еще предмет 1269, поэтому условная вероятность равна $57/80 \approx 71\%$. Учитывая, что только 0.3% всех покупок содержали предмет 1269, получается подъем 255.

Сколько-нибудь обоснованные выводы можно сделать, только если эти счетчики достаточно велики, поэтому мы сначала должны отобрать частые предметные наборы. Если бы мы начали генерировать правила по нечастому набору, то счетчики оказались бы малы, поэтому относительные значения были бы бессмысленны или недостоверны.

Отметим, что по этому набору можно найти много других ассоциативных правил; алгоритм находит 1030 правил (при минимальной опоре 80 и подъеме, не меньшем 5). Это небольшой набор данных по сравнению с теми, что ныне встречаются в веб. Когда набор содержит миллионы покупок, количество правил вполне может исчисляться тысячами, а то и миллионами.

Однако для каждого пользователя или товара релевантны будут лишь несколько правил, поэтому пользователь получит на так уж много рекомендаций.

Более сложный анализ корзины

Сегодня существуют и другие алгоритмы анализа корзины, работающие быстрее Apriori. Показанный выше код прост, и нам его вполне хватало, потому что в нашем распоряжении были сведения только о 100 тысячах покупок. Если бы покупок были миллионы, то стоило бы взять алгоритм побыстрее. Отметим, впрочем, что искать ассоциативные правила можно и автономно, тогда эффективность не столь критична.

Существуют также методы работы с хронологической информацией, вырабатывающие правила, в которых учитывается порядок со-

вершения покупок. Допустим, к примеру, что человек, закупивший продукты для большого приема, затем возвращается, чтобы купить мешки для мусора. Тогда имеет смысл предложить ему мешки для мусора при первом же посещении. Однако предлагать припасы для вечеринки всякому, кто покупает мешки для мусора, вряд ли разумно.

Резюме

Мы начали эту главу с использования регрессии для прогнозирования оценок. Мы видели, что это можно сделать двумя способами, а затем объединили их, применив машинное обучение для нахождения весов. Эта техника обучения ансамбля, и в особенности послойное обучение, применяется в разных ситуациях и не только для регрессии. Она позволяет комбинировать различные идеи, даже если внутренние механизмы их реализации совершенно различны; комбинируются конечные результаты.

Во второй части главы мы переключили передачу и рассмотрели другой способ выработки рекомендаций: анализ корзины покупок и поиск ассоциативных правил. В этом случае мы пытаемся выявить (вероятностные) ассоциативные правила вида «купивших *X*, вероятно, заинтересует также *Y*». Такая методика обладает тем преимуществом, что нужны только данные о самих покупках, а не проставленные пользователями оценки. В настоящее время в scikit-learn еще нет реализаций соответствующих алгоритмов, поэтому мы написали свою собственную.

Искать ассоциативные правила следует осторожно, чтобы дело не свелось к рекомендованию бестселлеров каждому пользователю (где же тут персонализация?). Мы избежали этой опасности, научившись измерять относительную ценность правила с помощью так называемого подъема.

На данный момент мы рассмотрели основную отрасль машинного обучения: классификацию. В следующих двух главах мы познакомимся с двумя частными случаями данных: музыкой и изображениями. Наша первая цель – построить классификатор музыки по жанрам.

ГЛАВА 9. Классификация по музыкальным жанрам

До сих пор мы существовали в весьма комфортных условиях: любой обучающий набор легко было описать вектором значений признаков. Например, в наборе данных Iris цветы представлялись векторами, содержащими длину и ширину некоторых частей цветка. В примерах, относящихся к текстам, нам удавалось преобразовать текст в набор представлений слов и вручную подобрать признаки, улавливающие определенные характеристики текста.

В этой главе все будет по-другому. Мы постараемся классифицировать музыкальные произведения по жанрам. А как описать трехминутную песню? Брать отдельные биты MP3-представления? Наверное, нет, потому что попытка обращаться с музыкой, как с текстом и создавать что-то вроде «набора звуковых фрагментов» оказалась бы слишком сложным делом. Но тем не менее, мы должны как-то преобразовать музыкальное произведение в последовательность значений, адекватно описывающую его.

План действий

В этой главе мы покажем, как построить приличный классификатор для предметной области, не столь комфортной, как раньше. Для начала нам придется иметь дело со звуковыми признаками, которые гораздо сложнее текстовых. А затем нужно будет научиться работать с несколькими классами, тогда как до сих пор мы встречались лишь с бинарной классификацией. Ну и попутно мы познакомимся с новыми способами измерения качества классификации.

Допустим, мы обнаружили на диске кучу MP3-файлов со случайными именами, которые предположительно содержат музыку. Наша задача – разложить их по разным папкам в соответствии с жанром: джаз, классическая музыка, кантри, поп, рок, металл и т. д.

Получение музыкальных данных

Мы будем работать с набором данных GTZAN, который часто используют для оценки алгоритмов классификации по музыкальным жанрам. В нем выделены 10 жанров, из которых мы для простоты возьмем только шесть: классика, джаз, кантри, поп, рок и металл. В набор данных включены первые 30 секунд звучания 100 произведений каждого жанра. Скачать набор можно по адресу `http://opihi.cs.uvic.ca/sound/genres.tar.gz`.

Музыкальные произведения записаны с частотой дискретизации 22 050 Гц (22 050 отсчетов в секунду) в монофоническом звучании и представлены в формате WAV.

Преобразование в формат WAV

Конечно, если бы нам предстояло протестировать свой классификатор на собственной коллекции MP3-файлов, то потерпели бы фиаско. Дело в том, что MP3 – формат сжатия с потерей информации, в котором обрезаются частоты, не воспринимаемые человеческим ухом. Это хорошо с точки зрения хранения, потому что мы сможем сохранить на устройстве примерно в 10 раз больше музыки. Но вот для нашего предприятия это печально. Классификацию проще производить на WAV-файлах, потому что их можно считывать непосредственно с помощью пакета `scipy.io.wavfile`. Поэтому нам придется конвертировать MP3-файлы, чтобы подать на вход нашему классификатору.

Если у вас нет под рукой подходящего конвертера, попробуйте взять SoX с сайта `http://sox.sourceforge.net`. Он рекламируется как швейцарский армейский нож в части обработки звука, и мы согласны с этим смелым заявлением.

Одно из достоинств формата WAV – возможность непосредственного чтения файла с помощью комплекта инструментов SciPy:

```
>>> sample_rate, X = scipy.io.wavfile.read(wave_filename)
```

Теперь `X` содержит отсчеты, а `sample_rate` – частота, с которой они берутся. Воспользуемся этой информацией, чтобы заглянуть внутрь музыкального файла и получить представление о характере данных.

Взгляд на музыку

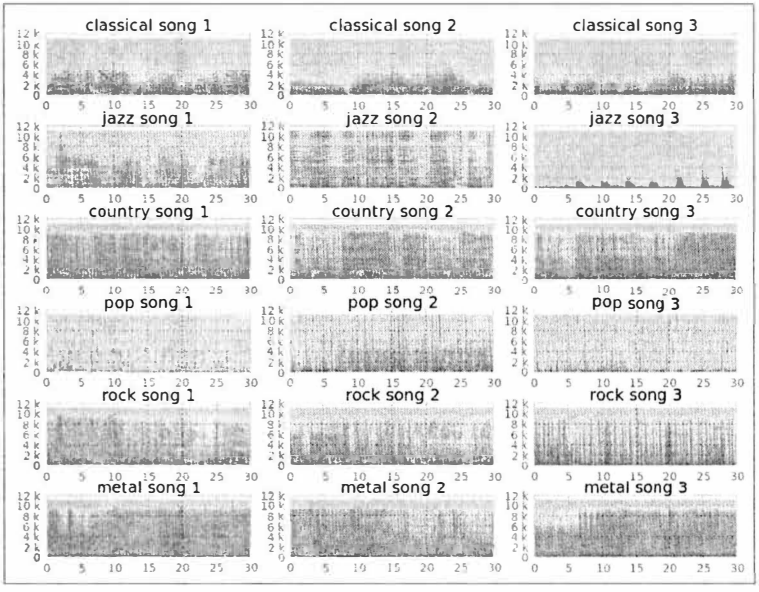
Очень удобный способ составить первое впечатление о том, как «выглядят» музыкальные произведения разных жанров, – нарисовать спектрограмму произведения. Спектрограмма – это наглядное представление встречающихся частот. По оси y откладывается интенсивность частот, а по оси x – временные интервалы. Чем темнее цвет, тем интенсивнее частота в данном интервале звучания.

В библиотеке matplotlib есть функция `specgram()`, которая производит вычисления и строит спектрограмму:

```
>>> import scipy
>>> from matplotlib.pyplot import specgram
>>> sample_rate, X = scipy.io.wavfile.read(wave_filename)
>>> print sample_rate, X.shape
22050, (661794,)
>>> specgram(X, Fs=sample_rate, xextent=(0,30))
```

Прочитанный WAV-файл был дискретизирован с частотой 22 050 Гц и содержал 661 794 отсчета.

Построив спектрограммы первых 30 секунд звучания для разных WAV-файлов, мы увидим, что у произведений одного жанра есть общие черты.



Достаточно бросить беглый взгляд на рисунок, как мы сразу заметим разницу между спектром, скажем, металла и классики. Если в «металлических» песнях наблюдается высокая интенсивность частот в большей части спектра на протяжении всего времени (еще бы, ведь они энергичные!), то для классических произведений характерно большее разнообразие.

Вероятно, не составило бы большого труда обучить классификатор, который довольно точно различал бы металл и классику. Но другие пары жанров, например кантри и рок, далеко не так очевидны. Это проблема, потому что нам нужно различать на два, а все шесть классов. И притом с достаточной надежностью.

Разложение на синусоидальные волны

Наш план состоит в том, чтобы извлечь интенсивности отдельных частот из массива отсчетов (ранее сохраненного в переменной x) и подать их на вход классификатора. Для этой цели служит **быстрое преобразование Фурье (БПФ)**. Теория, лежащая в основе БПФ, выходит за рамки этой книги, мы просто на примере покажем, что получается в результате. А затем будем использовать БПФ как черный ящик, вырабатывающий признаки.

Сгенерируем, к примеру, два WAV-файла, `sine_a.wav` и `sine_b.wav`, содержащие синусоидальные звуковые волны частотой 400 и 3000 Гц. Сделать это позволяет вышеупомянутый «швейцарский армейский нож» SoX:

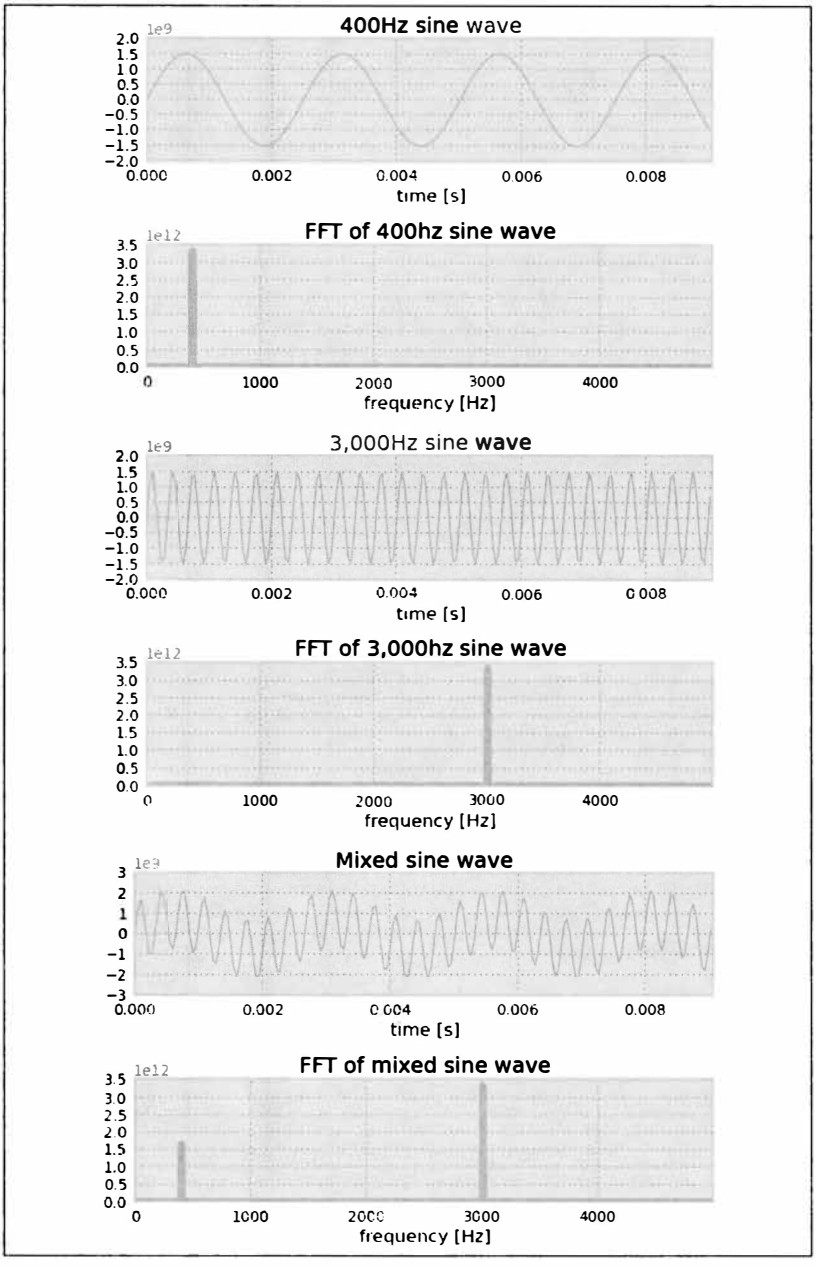
```
$ sox --null -r 22050 sine_a.wav synth 0.2 sine 400
$ sox --null -r 22050 sine_b.wav synth 0.2 sine 3000
```

На рисунках ниже показаны первые 0.008 секунд звучания. А под ними – БПФ синусоидальных волн. Как и следовало ожидать, на частотах 400 и 3000 Гц наблюдаются пики.

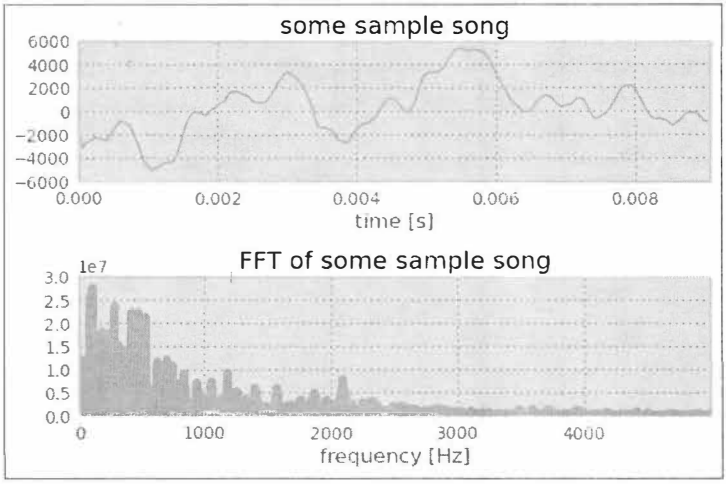
Теперь смешаем оба файла, назначив звуку частотой 400 Гц половинную громкость, а звуку 3000 Гц – единичную:

```
$ sox --combine mix --volume 1 sine_b.wav --volume 0.5 sine_a.wav
sine_mix.wav
```

На графике комбинированного звучания мы увидим два пика, причем тот, что соответствует частоте 3000 Гц, в два раза выше соответствующего частоте 400 Гц.


400Hz sine wave
1e9
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.000
0.002
0.004
0.006
0.008
time [s]
FFT of 400hz sine wave
1e12
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1000
2000
3000
4000
frequency [Hz]
3,000Hz sine wave
1e9
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.000
0.002
0.004
0.006
0.008
time [s]
FFT of 3,000hz sine wave
1e12
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1000
2000
3000
4000
frequency [Hz]
Mixed sine wave
1e9
3
2
1
0
−1
−2
−3
0.000
0.002
0.004
0.006
0.008
time [s]
FFT of mixed sine wave
1e12
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1000
2000
3000
4000
frequency [Hz]

Конечно, для реальной музыки БПФ выглядит не так красиво, как для искусственного примера.



Применение БПФ для построения первого классификатора

И тем не менее, с помощью БПФ мы можем создать своего рода цифровой отпечаток музыкального произведения. Если сделать это для нескольких произведений и вручную сопоставить им метки – жанры, то получатся обучающие данные, которые можно подать классификатору.

Повышение гибкости эксперимента

Прежде чем с головой погрузиться в обучение классификатора, подумаем о том, как повысить гибкость эксперимента. Хотя первое слово в аббревиатуре БПФ – «быстрое», на самом деле это происходит куда медленнее, чем выделение признаков из текста. А поскольку мы пока экспериментируем, то можем задуматься об ускорении процесса выработки признаков.

Разумеется, БПФ для каждого файла создается одинаково при каждом запуске классификатора. Поэтому можно было бы кэши-

ровать результат БПФ и читать из кэша только его, а не весь WAV-файл. Мы так и поступаем в функции `create_fft()`, которая вызывает `scipy.fft()` для создания БПФ. Для простоты (и скорости!) ограничимся первыми 1000 компонентов БПФ. При том, что нам сейчас известно, мы не можем решить, будут ли именно эти компоненты самыми важными для классификации по жанрам; мы лишь видим, что в показанном выше примере БПФ у них самые интенсивные частоты. Если впоследствии мы захотим увеличить или уменьшить число компонент, то должны будем пересоздать БПФ для каждого файла.

```
import os
import scipy

def create_fft(fn):
  sample_rate, X = scipy.io.wavfile.read(fn)
  fft_features = abs(scipy.fft(X)[:1000])
  base_fn, ext = os.path.splitext(fn)
  data_fn = base_fn + ".fft"
  scipy.save(data_fn, fft_features)
```

Мы сохраняем данные с помощью функции NumPy `save()`, которая всегда создает файл с расширением `.npy`. Это нужно сделать только один раз для каждого WAV-файла, необходимого для обучения или прогнозирования.

Для чтения БПФ служит функция `read_fft()`:

```
import glob

def read_fft(genre_list, base_dir=GENRE_DIR):
  X = []
  y = []

  for label, genre in enumerate(genre_list):
     genre_dir = os.path.join(base_dir, genre, "*.fft.npy")
     file_list = glob.glob(genre_dir)

     for fn in file_list:
        fft_features = scipy.load(fn)

        X.append(fft_features[:1000])
        y.append(label)

  return np.array(X), np.array(y)
```

Предполагается, что в нашем перепутанном каталоге есть произведения таких жанров:

```
genre_list = ["classical", "jazz", "country", "pop", "rock", "metal"]
```

Обучение классификатора

Воспользуемся классификатором на основе логистической регрессии, который сослужил нам хорошую службу в главе 6. Правда, возникает дополнительная трудность – теперь мы имеем дело с задачей многоклассовой классификации, тогда как раньше нужно было различать всего два класса.

Отметим, что при переходе от бинарной к многоклассовой классификации оценки верности претерпевают удивительную метаморфозу. При бинарной классификации мы считали, что верность 50% – худший случай, потому что такого результата можно достичь и случайным угадыванием. Но в случае многоклассовой классификации 50% может быть очень хорошей верностью. Если имеется 6 жанров, то случайное угадывание дало бы только 16.7% (в предположении, что размеры классов одинаковы).

Применение матрицы неточностей для измерения верности в многоклассовых задачах

В многоклассовых задачах нас интересует не только то, насколько правильно мы определяем жанр произведения. Важно также знать, какие жанры мы путаем. Это можно сделать с помощью матрицы неточностей, как показано ниже:

```
>>> from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_test, y_pred)
>>> print(cm)
[[26  1  2  0  0  2]
 [ 4  7  5  0  5  3]
 [ 1  2 14  2  8  3]
 [ 5  4  7  3  7  5]
 [ 0  0 10  2 10 12]
 [ 1  0  4  0 13 12]]
```

Печатается распределение меток, предсказанных классификатором на тестовом наборе для каждого жанра. На диагонали находятся правильные результаты. Поскольку всего жанров шесть, то матрица имеет размер 6 × 6. Первая строка означает, что из 31 классического произведения (сумма всех чисел в первой строке) 26 были классифицированы как классика, 1 – как джаз, 2 – как кантри и 2 – как металл. Правильный результат классификации лежит на диагонали. Следовательно, 26 произведений классифицированы правильно,

а 5 – неправильно. Не так уж плохо. Вторая строка отрезвляет: из 24 джазовых композиций лишь 7 классифицированы правильно – всего 29%.

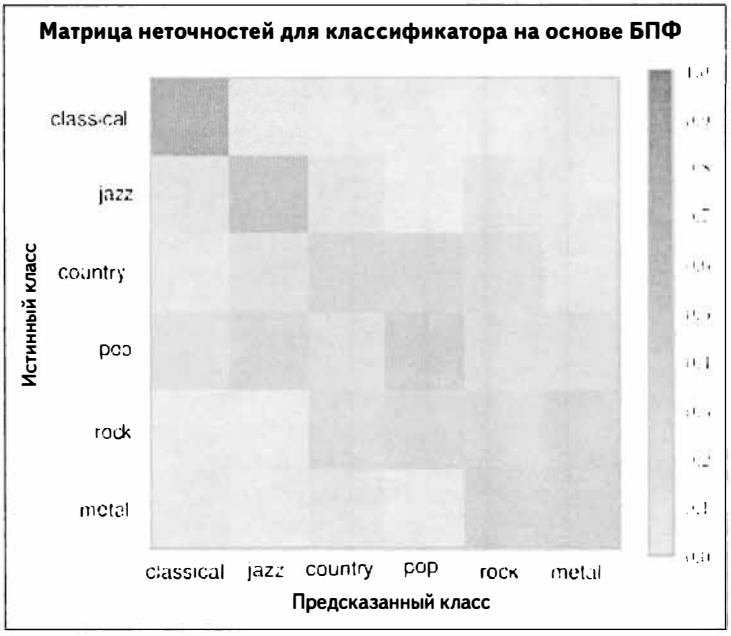
Разумеется, мы разбиваем весь набор данных на обучающие и тестовые, как было описано в предыдущих главах, и должны составлять матрицы неточностей для каждого прохода перекрестной проверки. Затем нужно будет произвести усреднение и нормировку, чтобы величины находились в диапазоне от 0 (полное фиаско) до 1 (все образцы классифицированы правильно).

Наглядное представление часто воспринимается гораздо легче, чем массивы NumPy. На помощь приходит функция `matshow()` из библиотеки matplotlib:

```
from matplotlib import pylab

def plot_confusion_matrix(cm, genre_list, name, title):
  pylab.clf()
  pylab.matshow(cm, fignum=False, cmap='Blues',
                vmin=0, vmax=1.0)

  ax = pylab.axes()
  ax.set_xticks(range(len(genre_list)))
  ax.set_xticklabels(genre_list)
  ax.xaxis.set_ticks_position("bottom")
  ax.set_yticks(range(len(genre_list)))
  ax.set_yticklabels(genre_list)

  pylab.title(title)
  pylab.colorbar()
  pylab.grid(False)
  pylab.xlabel('Predicted class')
  pylab.ylabel('True class')
  pylab.grid(False)

  pylab.show()
```

При создании матрицы неточностей не забудьте выбрать цветовую схему (параметр `cmap` функции `matshow()`), указав подходящий порядок цветов, чтобы с первого взгляда было понятно, что означает более светлый или более темный цвет. Для такого рода диаграммы противопоказано раскрашивание в цвета радуги, как в подразумеваемой по умолчанию схеме `jet` или даже `Paired`.

Готовая диаграмма выглядит так:



От идеального классификатора мы ожидаем появления темных квадратиков на диагонали, идущей из левого верхнего в правый нижний угол, и светлых – во всех остальных местах. Из диаграммы выше сразу видно, что классификатор на основе БПФ далек от совершенства. Он правильно определяет лишь классические музыкальные произведения (темный квадратик). А, скажем, року он, как правило, предпочитает металл.

Очевидно, что использование БПФ – шаг в правильном направлении (на классике-то все выглядит неплохо), но этого недостаточно для получения достойного классификатора. Конечно, можно поэкспериментировать с количеством компонент БПФ (сейчас оно равно 1000). Но перед тем как приниматься за настройку параметров, надо проделать изыскания. Пока что мы знаем, что БПФ дает приемлемые признаки для классификации по жанрам, но нуждается в уточнении. Вскоре мы увидим, как можно повысить качество классификации за счет применения варианта с предварительной обработкой.

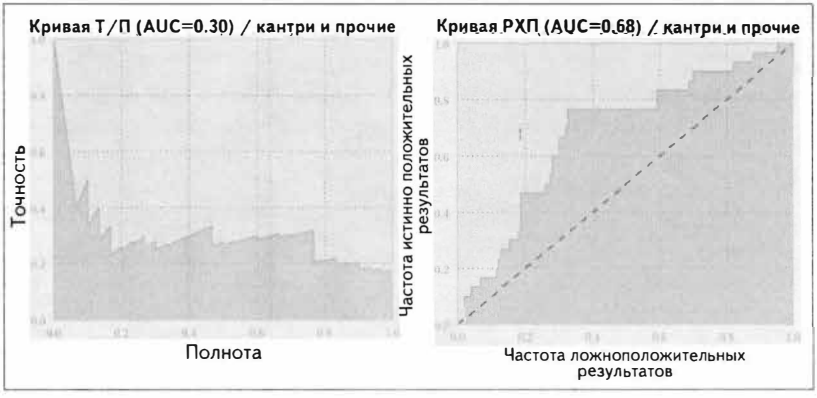
Но прежде рассмотрим еще один метод измерения качества классификации.

Альтернативный способ измерения качества классификатора с помощью рабочей характеристики приемника

Мы уже знаем, что измерения верности недостаточно для оценки классификатора. Чтобы лучше понять, как работает классификатор, мы использовали кривые точности-полноты (Т/П).

Существуют также похожие на Т/П кривые рабочей характеристики приемника (РХП), которые измеряют сходные аспекты работы классификатора, но в другом виде представляют его качество. Ключевое различие заключается в том, что кривые Т/П больше подходят для задач, в которых положительный класс гораздо интереснее отрицательного, а также тех, где положительных примеров намного меньше, чем отрицательных. Типичные области применения – информационный поиск и обнаружение мошенничества. С другой стороны, кривые РХП дают более полную картину поведения классификатора в целом.

Чтобы лучше разобраться в различиях, рассмотрим качество ранее обученного классификатора в части правильности определения песен в стиле кантри (см. рисунок ниже).



На левом рисунке показана кривая Т/П. Для идеального классификатора это были бы два отрезка: из левого верхнего в правый верхний, а затем в правый нижний угол, так что площадь под кривой (AUC) оказалась бы равна 1.0.

На правом рисунке показана соответствующая кривая РХП. Она отражает зависимость частоты истинно положительных от частоты

ложноположительных результатов. Для идеального классификатора мы получили бы два отрезка: из левого нижнего в левый верхний, а затем в правый верхний угол. Классификатор, определяющий класс наугад, был бы представлен прямой, идущей из левого нижнего в правый верхний угол (изображена на рисунке штриховой линией), для которой AUC равна 0.5. Поэтому напрямую сравнивать AUC кривых Т/П и РХП нельзя.

Но в любом случае гарантируется, что при сравнении двух классификаторов на одном наборе данных большая площадь под кривой Т/П означает, что и площадь под кривой РХП будет больше, и наоборот. Поэтому генерировать обе кривые необязательно. Подробнее об этом можно прочитать в весьма содержательной статье Davis, Goadrich «The Relationship Between Precision-Recall and ROC Curves» (ICML, 2006).

В таблице ниже сведены различия между кривыми Т/П и РХП.

	Ось *x*	Ось *y*
Т/П	$Полнота = \frac{TP}{TP + FN}$	$Точность = \frac{TP}{TP + FP}$
РХП	$FPR = \frac{FP}{FP + TN}$	$TPR = \frac{TP}{TP + TN}$

Из определений осей x и y для обеих кривых видно, что частота истинно положительных результатов (TPR) по оси y кривой РХП в точности совпадает с полнотой по оси x кривой Т/П.

Частота ложноположительных результатов (FPR) измеряет долю истинных отрицательных примеров, ошибочно признанных положительных, и в идеальном случае равна 0 (нет ни одного ложноположительного результата), а в самом худшем – единице. Сравните с точностью, которая измеряет прямо противоположную величину, а именно долю истинных положительных примеров, ошибочно признанных отрицательными.

Пойдем дальше и применим кривые РХП для измерения качества классификаторов, это позволит лучше прочувствовать особенности инструмента. Единственная трудность состоит в том, что обе кривые – РХП и Т/П – относятся к задаче бинарной классификации, тогда как перед нами стоит многоклассовая задача. Поэтому давайте создадим по одной кривой на каждый жанр, которая будет показы-

вать, насколько хорошо классификатор отличает произведения этого жанра от всех прочих.

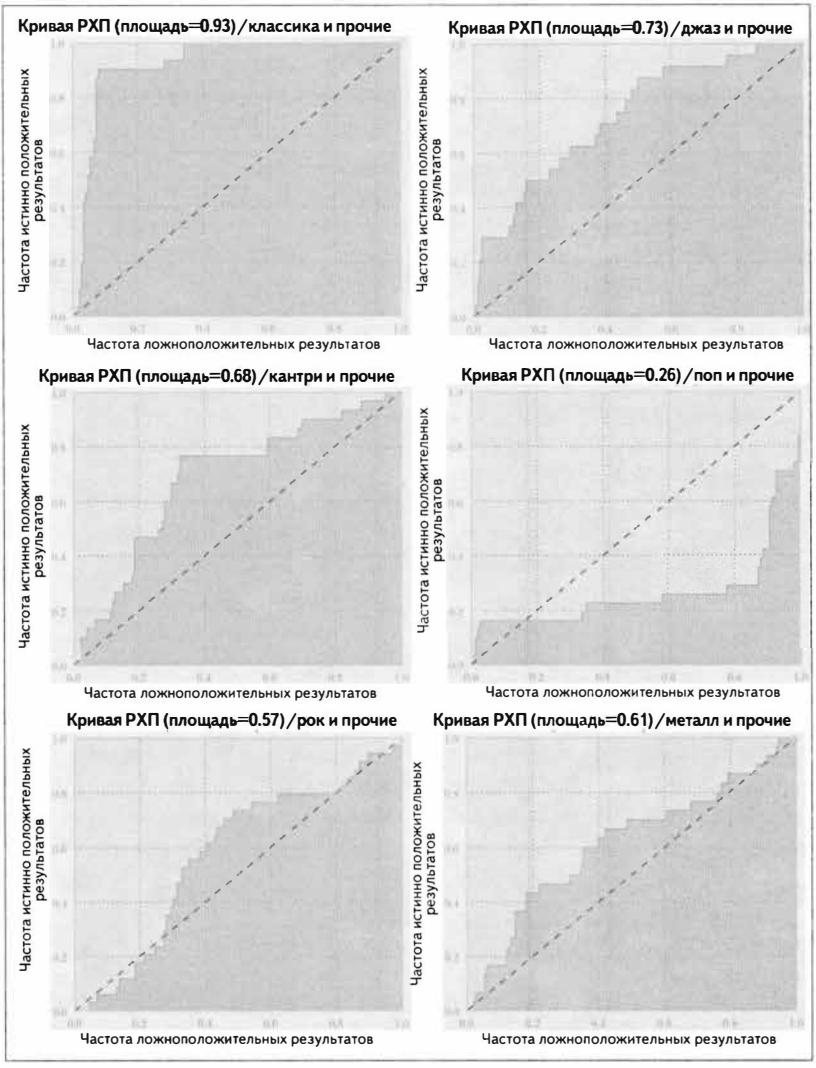
```
from sklearn.metrics import roc_curve

y_pred = clf.predict(X_test)

for label in labels:
  y_label_test = scipy.asarray(y_test==label, dtype=int)
  proba = clf.predict_proba(X_test)
  proba_label = proba[:,label]

  # вычислить частоты ложно и истинно положительных результатов,
  # а также пороги РХП
  fpr, tpr, roc_thres = roc_curve(y_label_test, proba_label)

  # построить график зависимости tpr от fpr ...
```

На выходе получаются графики РХП, показанные на следующей странице. Как мы уже выяснили, первая версия нашего классификатора хорошо работает только для классических произведений. Кривые РХП показывают, что на самом деле качество оставляет желать лучшего на большинстве остальных жанров. Лишь джаз и кантри вселяют какую-то надежду, классификация прочих вообще никуда не годится.

Повышение качества классификации с помощью мел-частотных кепстральных коэффициентов

Мы уже поняли, что БПФ – шаг в правильном направлении, но его одного недостаточно для построения классификатора, который рассортировал бы наш каталог с музыкальными произведениями разных жанров. Нужно что-то более точное.

Сейчас будет правильно признать, что нужны дополнительные исследования. Возможно, кто-то уже сталкивался с подобной проблемой в прошлом и нашел решение, которое нам поможет. И действительно, даже существует ежегодная конференция, посвященная классификации по музыкальным жанрам, которую проводит **Международное общество по музыкальному информационному поиску**



(International Society for Music Information Retrieval, ISMIR). Похоже, **автоматическая классификация по музыкальным жанрам** (Automatic Music Genre Classification, AMGC) – уже сложившаяся подобласть музыкального информационного поиска. Изучая список литературы по AMGC, мы найдем немало работ, которые могут нам помочь.

В частности, во многих работах на эту тему успешно применяется метод мел-частотных кепстральных коэффициентов. **Мел-частотный кепстр** (Mel Frequency Cepstrum, MFC) – это способ кодирования энергетического спектра звука, то есть распределения энергии, содержащейся в каждой частоте. Он вычисляется как преобразование Фурье логарифма спектра сигнала. Если это кажется слишком сложным, имейте в виду, что название «кепстр» – просто анаграмма слова «спектр». Метод MFC успешно используется при распознавании речи и говорящего. Посмотрим, поможет ли он в нашем случае.

Нам повезло в том смысле, что кто-то уже занимался именно этой задачей и опубликовал реализацию в виде библиотеки Talkbox SciKit. Мы можем скачать ее со страницы `https://pypi.python.org/pypi/scikits.talkbox`. Затем вызовем функцию `mfcc()`, которая вычислит коэффициента MFC:

```
>>> from scikits.talkbox.features import mfcc
>>> sample_rate, X = scipy.io.wavfile.read(fn)
>>> ceps, mspec, spec = mfcc(X)
>>> print(ceps.shape)
(4135, 13)
```

Подать на вход классификатору мы хотели бы массив `ceps`, содержащий 13 коэффициентов (это подразумеваемое по умолчанию значение параметра `nceps` функции `mfcc()`) для каждого из 4135 кадров музыкального произведения в файле с именем `fn`. Но такой объем данных наш классификатор не переварит. Вместо этого можно усреднить коэффициенты по всем кадрам. В предположении, что начало и конец каждого произведения в меньшей степени определяют жанр, чем середина, мы также проигнорируем первые и последние 10 процентов звучания.

```
x = np.mean(ceps[int(num_ceps*0.1):int(num_ceps*0.9)], axis=0)
```

Правда, набор данных, на котором мы ставим эксперименты, содержит только первые 30 секунд звучания каждого произведения, поэтому можно было бы и не отсекать последние 10 процентов. Но мы все-таки это сделаем, чтобы наш код корректно работал с реальными данными.

Так же, как и в случае БПФ, мы будем кэшировать однажды сгенерированные признаки MFC, а не создавать их при каждом обучении классификатора.

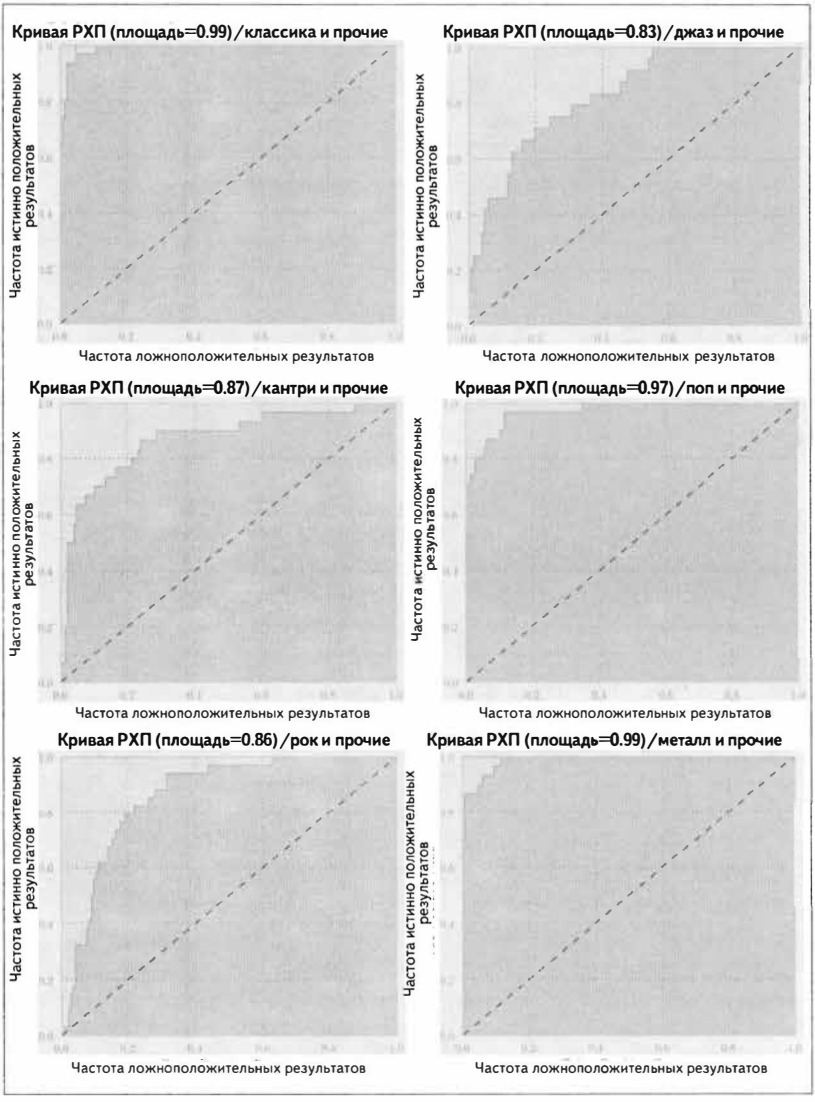
Получившийся код приведен ниже:

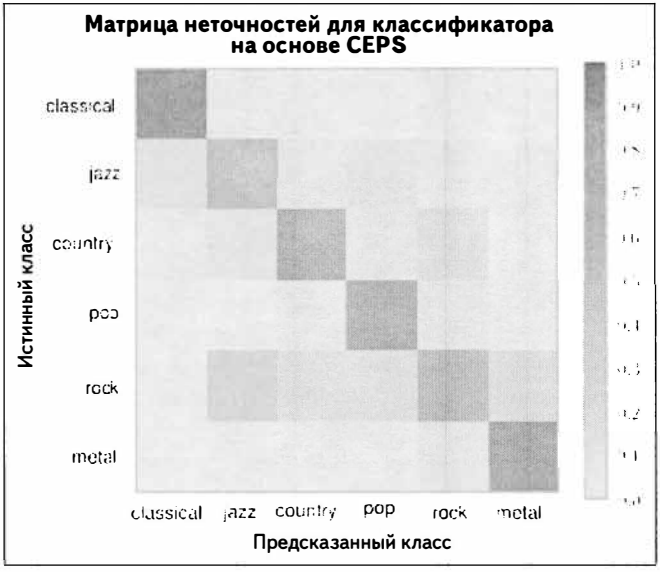
```
def write_ceps(ceps, fn):
  base_fn, ext = os.path.splitext(fn)
  data_fn = base_fn + «.ceps»
  np.save(data_fn, ceps)
  print(«Written to %s» % data_fn)

def create_ceps(fn):
  sample_rate, X = scipy.io.wavfile.read(fn)
  ceps, mspec, spec = mfcc(X)
  write_ceps(ceps, fn)

def read_ceps(genre_list, base_dir=GENRE_DIR):
  X, y = [], []
  for label, genre in enumerate(genre_list):
     for fn in glob.glob(os.path.join(
           base_dir, genre, «*.ceps.npy»)):
        ceps = np.load(fn)
        num_ceps = len(ceps)
        X.append(np.mean(
              ceps[int(num_ceps*0.1):int(num_ceps*0.9)], axis=0))
        y.append(label)

  return np.array(X), np.array(y)
```

Получаем следующие многообещающие результаты для классификатора, в котором используется всего 13 признаков на каждое музыкальное произведение (см. графики на следующей странице).

Качество классификации улучшилось для всех жанров. Более того, для классики и металла площадь под кривой лишь чуть-чуть недотягивает до 1.0. Да и матрица неточностей на диаграмме ниже выглядит гораздо лучше. Отчетливо выделяется диагональ, а это свидетельствует о том, что классификатор правильно определяет жанры в большинстве случаев. Такой классификатор вполне пригоден для решения поставленной задачи (см. графики на следующей странице).

Если бы мы захотели и дальше улучшать классификатор, то матрица неточностей сразу показала бы, на что обратить внимание: небелые места вне диагонали. Например, имеется темное место там, где мы с довольно большой вероятностью определили роковые композиции как джазовые. Чтобы устранить проблему, нужно было бы глубже проанализировать произведение и выделить в качестве признаков такие вещи, как барабанная дробь и прочие жанровые характеристики. А кроме того – просматривая статьи ISMIR – мы прочитали про признаки на основе темпоральной огибающей акустического банка фильтров (Auditory Filterbank Temporal Envelope, AFTE), которые, похоже, превосходят признаки MFC в некоторых ситуациях. Может быть, стоит присмотреться к ним?


Кривая РХП (площадь=0.99) / классика и прочие
Кривая РХП (площадь=0.83) / джаз и прочие
Кривая РХП (площадь=0.87) / кантри и прочие
Кривая РХП (площадь=0.97) / поп и прочие
Кривая РХП (площадь=0.86) / рок и прочие
Кривая РХП (площадь=0.99) / металл и прочие
Частота истинно положительных результатов
Частота ложноположительных результатов



Важно, что, имея в арсенале кривые РХП и матрицы неточностей, мы можем экспериментировать с экстракторами признаков, придуманными другими экспертами, даже не понимая, как они работают. Наши инструменты измерения скажут, в правильном ли направлении мы движемся, или надо сменить курс. Разумеется, у человека, постоянно стремящегося к новым знаниям, останется смутное ощущение, что где-то в глубинах черного ящика скрыт интереснейший алгоритм выделения признаков, который только и ждет, чтобы в нем разобрались.

Резюме

В этой главе мы покинули комфортную зону и построили классификатор музыки по жанрам. Плохо разбираясь в теории музыки, мы сначала потерпели неудачу, попытавшись обучить классификатор распознавать жанры, исходя из результатов БПФ. Но затем мы воспользовались признаками MFC и смогли построить классификатор вполне приличного качества.

В обоих случаях мы применяли признаки, которых не понимали, но знали, как их выделить и передать классификатору. Одни привели

к неудаче, другие – к успеху. Разница в том, что во втором случае мы брали признаки, предложенные специалистами в предметной области.

И это совершенно нормально. Если нас интересует в основном результат, то иногда приходится срезать углы – нужно только, чтобы короткая дорожка была проложена специалистами. А поскольку мы научились правильно измерять качество решения этой многоклассовой задачи классификации, то можем срезать углы достаточно уверенно.

В следующей главе мы посмотрим, как изученные нами методы применяются к новому виду данных. Мы научимся пользоваться пакетом машинного зрения mahotas для предварительной обработки изображений традиционными способами.

ГЛАВА 10. Машинное зрение

Анализ изображений и машинное зрение всегда играли важную роль в промышленных и научных приложениях. После того как мобильные телефоны стали оснащаться мощными камерами и подключением к Интернету, потребители начали генерировать изображения во все возрастающих количествах. Поэтому открываются возможности использовать машинное зрение, чтобы пользователю было удобнее работать.

В этой главе мы поговорим о том, как применять изученные в процессе чтения этой книги методы к данным такого специального вида. В частности, мы научимся применять пакет машинного зрения mahotas для выделения признаков из изображений. Эти признаки можно подать на вход рассмотренных выше алгоритмов классификации. Разработанную методику мы применим к обработке общедоступных наборов фотографий. Далее мы увидим, как те же самые признаки можно применить к другой задаче – поиску похожих изображений.

Наконец, в конце главы обсудим использование локальных признаков. Это сравнительно новые алгоритмы; первый из них, достигший промышленного качества, SIFT (scale-invariant feature transform – масштабно-инвариантное преобразование признаков), был предложен в 1999 году. Сейчас они показывают хорошие результаты в разных задачах.

Введение в обработку изображений

С точки зрения компьютера, изображение – это большой прямоугольный массив значений пикселей. Наша цель – обработать изображение и принять решение, представляющее интерес для разрабатываемого приложения.

Первый шаг – загрузить изображение с диска, где оно обычно хранится в одном из графических форматов, например PNG или JPEG. Отметим, что в формате PNG изображение сжимается без потери информации, а в формате JPEG – с потерей, но он оптимизирован для визуального восприятия фотографий. Затем мы можем выполнить предварительную обработку изображения (например, нормировать с учетом разных условий освещения).

Вся эта глава построена вокруг задачи классификации. Мы хотим обучить классификатор на основе метода опорных векторов (или какого-нибудь другого алгоритма) на изображениях. Но перед тем как приступать к машинному обучению, нужно придумать промежуточное представление и выделить из изображений числовые признаки.

Загрузка и показ изображения

Для манипулирования изображениями мы будем использовать пакет mahotas. Скачать его можно со страницы `https://pypi.python.org/pypi/`, а прочитать руководство – на странице – `http://mahotas.readthedocs.org`. Mahotas – пакет с открытым исходным кодом (распространяемый по лицензии MIT, то есть его можно использовать в любом проекте), разработанный одним из авторов этой книги. По счастью, он основан на NumPy. Знания о NumPy, полученные к этому моменту, можно применить к обработке изображений. Существуют и другие пакеты на эту тему, например scikit-image (skimage), модуль ndimage (*n*-мерные изображения) в SciPy и интерфейс из Python к библиотеке OpenCV. Все они работают с массивами NumPy, поэтому можно без опаски комбинировать функции, реализованные в разных пакетах, для построения конвейера.

Для начала импортируем mahotas, назначив ему сокращенное имя `mh`, которым и будем пользоваться в этой главе:

```
>>> import mahotas as mh
```

Теперь можно загрузить файл изображения методом `imread`:

```
>>> image = mh.imread('scene00.jpg')
```

Файл `scene00.jpg` (имеется в составе набора данных в репозитории на сопроводительном сайте книги) содержит цветное изображение высотой `h` и шириной `w`; это изображение будет храниться в массиве

формы (h, w, 3). Первое измерение – высота, второе – ширина, а третье – цвет в виде трех компонент: красной, зеленой и синей. В других системах первое измерение соответствует ширине, но во всех пакетах на основе NumPy принято именно такое соглашение. Тип данных в массиве обычно np.uint8 (8-разрядное целое без знака). Именно так представлены изображения, которые вы снимаете камерой или отображаете на экране монитора.

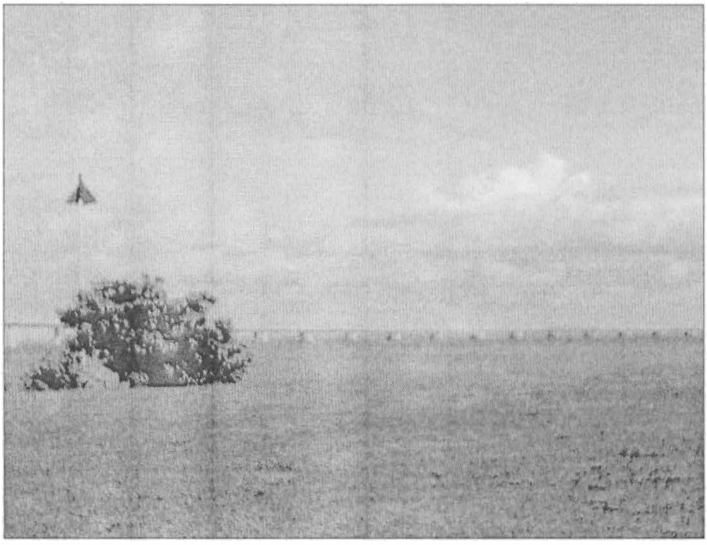
Специальное оборудование, применяемое в научно-технических приложениях, умеет работать с изображениями более высокой разрядности (то есть с большей чувствительностью к малым изменениям яркости), обычно 12- или 16-разрядными. Mahotas умеет работать и с такими типами, в том числе с числами с плавающей точкой. Во многих расчетах удобно перейти от целых чисел без знака к числам с плавающей точкой, чтобы упростить обработку округления и переполнения.

Mahotas может работать с различными системами ввода-вывода. К сожалению, ни одна из них не понимает все существующие форматы (их сотни, и у многих по несколько вариантов). Однако загрузку изображений в форматах PNG и JPEG поддерживают все. Мы ограничимся только ими, а за сведениями о том, как работать с менее распространенными форматами, отсылаем читателя к документации по mahotas.

Чтобы показать изображение на экране, воспользуемся библиотекой matplotlib, к которой уже неоднократно прибегали:

```
>>> from matplotlib import pyplot as plt
>>> plt.imshow(image)
>>> plt.show()
```

Как показано на рисунке ниже, этот код предполагает, что первое измерение – высота, а второе – ширина. Цветные изображения он также обрабатывает правильно. При использовании Python для числовых расчетов нам помогает тщательно спроектированная экосистема: mahotas работает с массивами NumPy, которые умеет отображать matplotlib; впоследствии мы извлечем из изображений признаки, которые обработаем посредством scikit-learn.



Бинаризация

Бинаризация – очень простая операция: все пиксели, большие некоторого порогового значения, заменяются единицей, а меньшие этого значения – нулем (или, если использовать булевы величины, преобразуются в `True` и `False`). Важно решить, как именно выбрать хорошее пороговое значение. В mahotas реализовано несколько способов выбора порога по изображению. Один из них называется **Otsu**, по имени своего изобретателя Оцу. Прежде всего, нужно перейти к полутоновому изображению с помощью функции `rgb2gray` из подмодуля `mahotas.colors`.

Вместо использования `rgb2gray` можно было бы вычислить среднее значение красного, зеленого и синего каналов, вызвав `image.mean(2)`. Но результат получился бы другой, потому что `rgb2gray` назначает цветам разные веса для получения более приятного для глаза изображения. Чувствительность наших глаз к трем основным цветам неодинакова.

```
>>> image = mh.colors.rgb2grey(image, dtype=np.uint8)
>>> plt.imshow(image) # Вывести изображение на экрана
```

По умолчанию matplotlib отображает такое одноканальное изображение как псевдоцветное: для больших значений берется красный

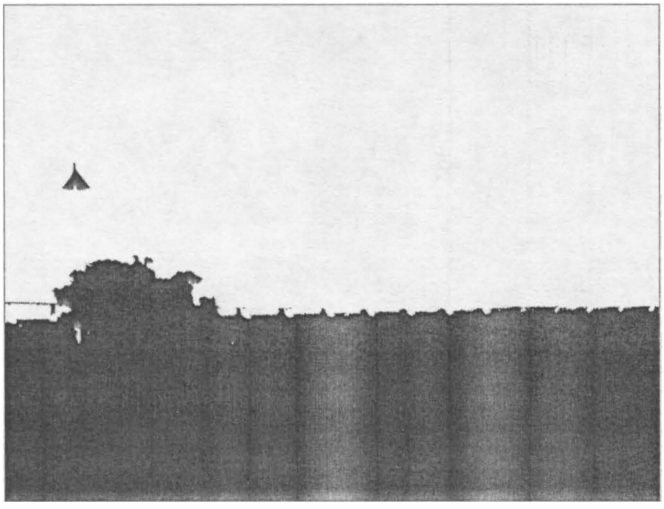
цвет, а для малых – синий. Но для естественных изображений предпочтительнее полутоновая гамма. Этот режим задается так:

```
>>> plt.gray()
```

Теперь получается полутоновое изображение. Отметим, что изменились только интерпретация и способ показа пикселей, сами данные изображения остались в неприкосновенности. Продолжим обработку и вычислим пороговое значение:

```
>>> thresh = mh.thresholding.otsu(image)
>>> print('Порог Оцу равен {}.'.format(thresh))
Порог Оцу равен 138.
>>> plt.imshow(image > thresh)
```

Для показанного выше изображения этот метод вычисляет пороговое значение 138, при этом разделяются земля и небо:



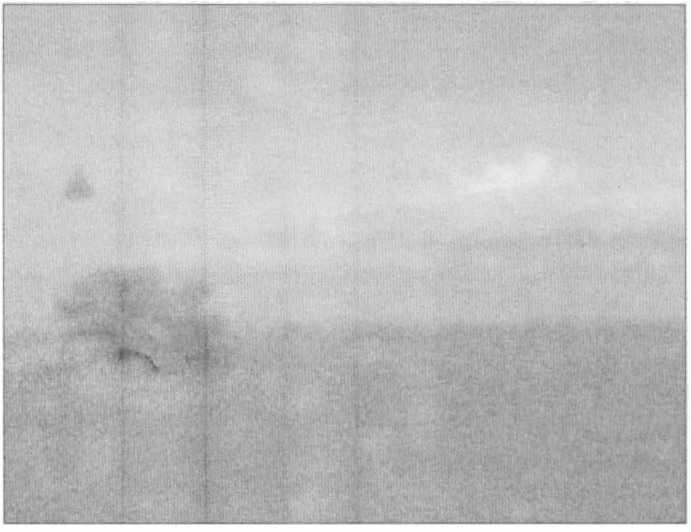
Гауссово размывание

На первый взгляд, непонятно, зачем размывать изображение, но часто это позволяет уменьшить шум и тем самым упростить последующую обработку. В mahotas для этого нужен всего один вызов:

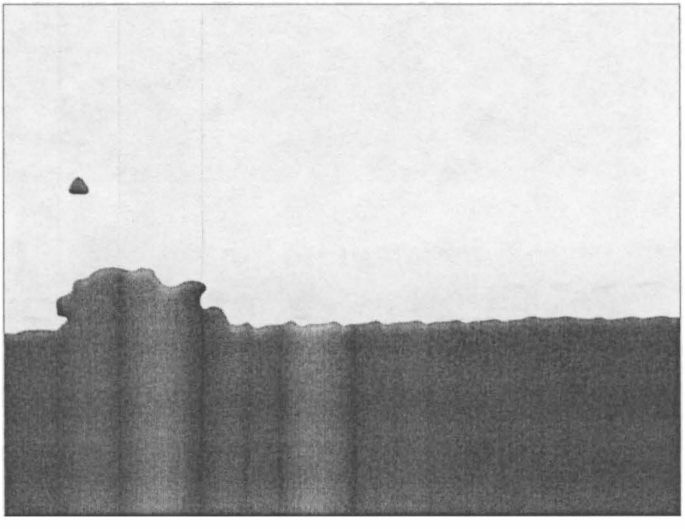
```
>>> im16 = mh.gaussian_filter(image, 16)
```

Отметим, что мы не преобразовывали значения пикселей полутонового изображения в целые числа без знака, а воспользовались

полученным результатом, где пиксели представлены числами с плавающей точкой. Второй аргумент функции `gaussian_filter` – размер фильтра (его стандартное отклонение). Чем он больше, тем сильнее размывание (см. рисунок ниже):



Если применить бинаризацию Оцу к размытому изображению, то границы будут более плавными:



Помещение центра в фокус

В следующем примере показано, как, объединив операторы NumPy с фильтрацией, можно добиться интересного результата. Начнем с фотографии Лены и выделим из него цветовые каналы:

```
>>> im = mh.demos.load('lena')
```

Эта фотография девушки часто используется для демонстрации обработки изображений:

Для выделения красного, зеленого и синего каналов напишем такой код:

```
>>> r,g,b = im.transpose(2,0,1)
```

Теперь профильтруем все три канала порознь и построим из них новое изображение с помощью функции `mh.as_rgb`. Она принимает три двумерных массива, производит растяжение контрастности, преобразуя каждый в массив 8-разрядных целых, а затем накладывает их друг на друга и возвращает цветное RGB-изображение:

```
>>> r12 = mh.gaussian_filter(r, 12.)
>>> g12 = mh.gaussian_filter(g, 12.)
>>> b12 = mh.gaussian_filter(b, 12.)
>>> im12 = mh.as_rgb(r12, g12, b12)
```

Теперь сошьем оба изображения от центра к краям. Сначала построим массив весов `w`, который будет содержать в позиции каждого пикселя нормированное значение – расстояние пикселя от центра:

```
>>> h, w = r.shape # высота и ширина
>>> Y, X = np.mgrid[:h,:w]
```

Мы воспользовались объектом `np.mgrid`, который возвращает массив размера `(h, w)`, значения в котором соответствую координатам y и x. Далее проделаем следующие операции:

```
>>> Y = Y - h/2. # центрировать на h/2
>>> Y = Y / Y.max() # привести к диапазону -1 .. +1

>>> X = X - w/2.
>>> X = X / X.max()
```

Теперь воспользуемся гауссовой функцией, чтобы увеличить значения пикселей в центральной области:

```
>>> C = np.exp(-2.*(X**2+ Y**2))
>>> # Снова нормируем: приводим к диапазону 0..1
>>> C = C - C.min()
>>> C = C / C.ptp()
>>> C = C[:,:,None] # Добавляем в массив C третье измерение
```

Отметим, что всё это – операции с массивами NumPy, никакой специфики mahotas здесь нет. Наконец, объединим оба массива, так чтобы центральная часть изображения оказалась в фокусе, а края были более расплывчатыми:

```
>>> ringed = mh.stretch(im*C + (1-C)*im12)
```

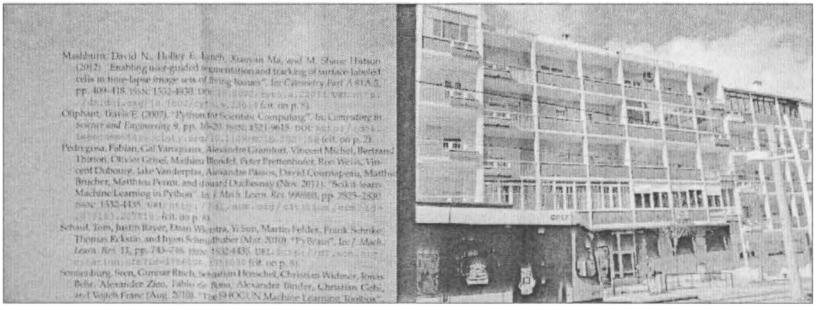
Простая классификация изображений

Начнем с небольшого набора данных, подготовленного специально для этой книги. В нем есть три класса: здания, природные ландшафты и фотографии текста. В каждой категории представлено 30 изображений, все они были сделаны камерой сотового телефона с минимальной композицией. Фотографии похожи на те, что закачивают на современные сайты пользователи, не учившиеся фотографировать. Набор данных можно скачать с сайта книги или из репозитория кода на GitHub. Позже мы рассмотрим более трудный набор данных, в котором и изображений, и категорий больше.

Для классификаций изображений у нас имеется большой прямоугольный массив чисел (значений пикселей). В наши дни типичный размер массив – порядка нескольких миллионов пикселей. Можно, конечно, попробовать загрузить все эти числа в алгоритм обучения в качестве признаков, но это не очень хорошая мысль. Дело в том, что связь каждого отдельного пикселя (и даже небольшой группы пикселей) с изображением очень отдаленная. Кроме того, наличие миллионов пикселей при очень небольшом количестве изображений приводит к чрезвычайно трудной задаче статистического обучения. Это крайнее проявление задач типа «*P* больше *N*», которые мы обсуждали в главе 7 «Регрессия». Гораздо разумнее вычислить признаки по изображению и использовать их для классификации.

И тем не менее, хочу отметить, что существуют методы, работающие непосредственно со значениями пикселей. В них встроены подпрограммы вычисления признаков. Они даже могут попытаться вывести хорошие признаки автоматически. В этой области сейчас ведутся активные исследования. Как правило, такие методы хорошо работают с очень большими наборами данных (содержащими миллионы изображений).

Выше мы уже видели пример изображения ландшафта. А вот ниже показаны примеры изображений текста и здания.



Вычисление признаков по изображению

Mahotas позволяет легко вычислять признаки по изображению. Соответствующие функции находятся в подмодуле `mahotas.features`.

Хорошо известен набор текстурных признаков Haralick. Как и многие алгоритмы обработки изображений, он назван по имени своего изобретателя. Признаки основаны на текстурах, то есть различают структурированные и неструктурированные изображения, а также различные повторяющиеся структуры. С помощью mahotas эти признаки вычисляются очень просто:

```
>>> haralick_features = mh.features.haralick(image)
>>> haralick_features_mean = np.mean(haralick_features, axis=0)
>>> haralick_features_all = np.ravel(haralick_features)
```

Функция `mh.features.haralick` возвращает массив 4×13. Первое измерение – четыре возможных направления, по которым вычисляются признаки (вертикаль, горизонталь и две диагонали). Если никакое конкретное направление нас не интересует, то можно усреднить признаки по всем направлениям (в коде выше эта переменная названа `haralick_features_mean`). Или же можно использовать все признаки по отдельности (переменная `haralick_features_all`). Решение зависит от свойств конкретного набора данных. Мы сочли, что в нашем случае признаки по вертикали и по горизонтали нужно хранить порознь, поэтому используем `haralick_features_all`.

В mahotas реализовано еще несколько наборов признаков. В частности, локальные бинарные шаблоны весьма устойчивы к изменению освещенности. Есть и другие типы признаков, в том числе локальных, которые мы обсудим ниже в этой главе.

Имея признаки, мы можем воспользоваться каким-нибудь стандартным методом классификации, например логистической регрессией:

```
>>> from glob import glob
>>> images = glob('SimpleImageDataset/*.jpg')
>>> features = []
>>> labels = []
>>> for im in images:
...     labels.append(im[:-len('00.jpg')])
...     im = mh.imread(im)
...     im = mh.colors.rgb2gray(im, dtype=np.uint8)
...     features.append(mh.features.haralick(im).ravel())

>>> features = np.array(features)
>>> labels = np.array(labels)
```

У трех наших классов текстуры сильно различаются. Для зданий характерны резкие края и крупные области, в которых цвета очень близки (значения пикселей редко бывают одинаковыми, но вариации невелики). Для текста характерно много резких переходов от темного к светлому и маленькие островки черного в море белого. Природные ландшафты характеризуются более плавными переходами фрактального типа. Поэтому можно ожидать, что классификатор, основанный на текстурах, будет работать хорошо.

Мы построим классификатор на базе логистической регрессии с предварительной обработкой признаков:

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.linear_model import LogisticRegression
>>> clf = Pipeline([('preproc', StandardScaler()),
                    ('classifier', LogisticRegression())])
```

Поскольку наш набор данных мал, можно воспользоваться регрессией с исключением по одному:

```
>>> from sklearn import cross_validation
>>> cv = cross_validation.LeaveOneOut(len(images))
>>> scores = cross_validation.cross_val_score(
...    clf, features, labels, cv=cv)
>>> print('Верность: {:.1%}'.format(scores.mean()))
Верность: 81.1%
```

Верность 81% – неплохо для трех классов (случайное угадывание дало бы только 33%). Но можно улучшить этот результат, создав собственные признаки.

Создание собственных признаков

В признаках нет ничего сверхъестественного. Это просто числа, вычисляемые по изображению. В литературе описано несколько наборов признаков. У них зачастую есть то преимущество, что они специально проектировались, чтобы не зависеть от малосущественных факторов. Например, локальные бинарные шаблоны инвариантны относительно умножения значений всех пикселей на одно число или прибавления к ним константы. Поэтому такой набор признаков устойчив к изменению освещенности.

Но не исключено, что в конкретной ситуации лучше будет работать какой-то специальный набор признаков.

Простой признак, не включенный в дистрибутив mahotas, – гистограмма цветов. К счастью, реализовать его совсем нетрудно. Идея

гистограммы цветов состоит в том, чтобы разбить пространство цветов на несколько интервалов, а затем подсчитать, сколько пикселей оказалось в каждом интервале.

Изображения хранятся в формате RGB, то есть каждый пиксель представлен тремя значениями: R (красный), G (зеленый) и B (синий). Каждое значение – это 8-разрядное число, что в сумме дает 17 миллионов цветов. Мы сократим количество цветов до 64, разнеся их по интервалам. Напишем функцию, инкапсулирующую этот алгоритм:

```
def chist(im):
```

Чтобы вычислить интервал цвета, сначала разделим весь массив на 64 с округлением значений:

```
    im = im // 64
```

Теперь значения каждого пикселя лежат в диапазоне от 0 до 3, то есть всего получается 64 цвета.

Выделим красный, зеленый и синий канал:

```
    r,g,b = im.transpose((2,0,1))
    pixels = 1 * r + 4 * b + 16 * g
    hist = np.bincount(pixels.ravel(), minlength=64)
    hist = hist.astype(float)
```

Приведем к логарифмической шкале, как показано в следующем фрагменте. Строго говоря, это необязательно, но признаки получаются более качественными. Мы пользуемся функцией `np.log1p`, которая вычисляет $\log(h+1)$. При этом нулевые значения остаются нулевыми (логарифм нуля не определен, и NumPy напечатает предупреждение при попытке вычислить его).

```
    hist = np.log1p(hist)
    return hist
```

Эту функцию несложно включить в написанный ранее код:

```
>>> features = []
>>> for im in images:
...   image = mh.imread(im)
...   features.append(chist(im))
```

Применяя тот же код перекрестной проверки, что и раньше, мы получаем верность 90%. Но самые лучшие результаты получаются, если объединить все признаки:

```
>>> features = []
>>> for im in images:
```

```
...   imcolor = mh.imread(im)
...   im = mh.colors.rgb2gray(imcolor, dtype=np.uint8)
...   features.append(np.concatenate([
...           mh.features.haralick(im).ravel(),
...            chist(imcolor),
...   ]))
```

При использовании всех этих признаков верность составит 95.6%:

```
>>> scores = cross_validation.cross_val_score(
...     clf, features, labels, cv=cv)
>>> print('Верность: {:.1%}'.format(scores.mean()))
Верность: 95.6%
```

Это еще одно подтверждение того, что хорошие алгоритмы – простая часть работы. Мы всегда можем воспользоваться реализациями самых передовых алгоритмов классификации, имеющимися в библиотеке scikit-learn. А самая-то соль и трудность – проектирование и подготовка признаков. Именно тут оказывается полезно знание о характере набора данных.

Использование признаков для поиска похожих изображений

Принципиальную идею о том, чтобы представить изображение сравнительно небольшим числом признаков, можно применить не только к классификации. Например, с ее помощью можно искать изображения, похожие на предъявленный в запросе образец (как с текстовыми документами).

Мы будем вычислять те же признаки, что и раньше, но с одним важным отличием: приграничные участки изображения игнорируются. Причина в том, что из-за любительской композиции на краях фотографии часто оказываются несущественные детали. Если вычислять признаки по всему изображению, то эти детали вносят вклад. А игнорируя их, удается получить чуть более качественные признаки. В случае обучения с учителем это не важно, потому что алгоритм обучения сможет определить, какие признаки более информативны, и назначить им соответственный вес. В режиме обучения без учителя нужно более внимательно следить за тем, чтобы признаки улавливали важные особенности данных. Эта идея реализована в следующем цикле:

```
>>> features = []
>>> for im in images:
...   imcolor = mh.imread(im)
```

```
...    # игнорировать все, что находится не дальше чем в 200
...    # пикселях от границы
...     imcolor = imcolor[200:-200, 200:-200]
...   im = mh.colors.rgb2gray(imcolor, dtype=np.uint8)
...   features.append(np.concatenate([
...       mh.features.haralick(im).ravel(),
...       chist(imcolor),
...   ]))
```

Теперь нормируем признаки и вычислим матрицу расстояний:

```
>>> sc = StandardScaler()
>>> features = sc.fit_transform(features)
>>> from scipy.spatial import distance
>>> dists = distance.squareform(distance.pdist(features))
```

Выведем только подмножество данных (каждый десятый элемент), расположив образец сверху, а возвращенный «ближайший сосед» снизу:

```
>>> fig, axes = plt.subplots(2, 9)
>>> for ci,i in enumerate(range(0,90,10)):
...   left = images[i]
...   dists_left = dists[i]
...   right = dists_left.argsort()
...   # right[0] - то же, что left[i], поэтому выберем следующий ближайший
...   right = right[1]
...   right = images[right]
...   left = mh.imread(left)
...   right = mh.imread(right)
...   axes[0, ci].imshow(left)
...   axes[1, ci].imshow(right)
```

Результат показан на рисунке ниже:

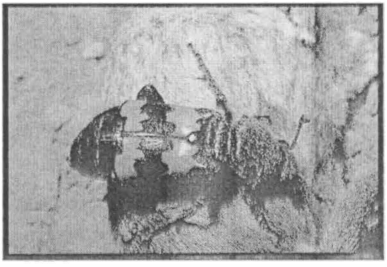
Видно, что система не совершенна, но все же способна находить изображения, которые, по крайней мере, зрительно похожи на предъявленный образец. Во всех случаях, кроме одного, найдены изображения из того же класса, что образец.

Классификация на более трудном наборе данных

Предыдущий набор оказалось легко классифицировать с помощью текстурных признаков. На самом деле, многие задачи, представляющие интерес для бизнеса, сравнительно просты. Но иногда попадаются проблемы потруднее, и тогда для получения приемлемого результата приходится применять более развитые современные методы.

Поэкспериментируем с общедоступным набором данных, имеющим такую же структуру: несколько фотографий, отнесенных к небольшому числу классов: животные, автомобили, транспортные средства и природные ландшафты.

Эти классы труднее различить, чем те три, что мы рассматривали выше. Природные ландшафты, здания и тексты характеризуются совершенно разными текстурами. А теперь текстуры и цвет не могут служить столь же очевидными маркерами класса. Вот пример изображения животного:



А вот – автомобиля:

Оба объекта сняты на фоне природы, а сами не имеют четко выраженной повторяющейся структуры. Это задача потруднее, поэтому понадобятся более сложные методы. Во-первых, мы воспользуемся более мощным классификатором. В библиотеке scikit-learn реализована регуляризованная форма логистической регрессии с настраиваемым параметром C. По умолчанию `C = 1.0`, но это значение не всегда оптимально. Для нахождения хорошего значения параметра можно использовать сеточный поиск:

```
>>> from sklearn.grid_search import GridSearchCV
>>> C_range = 10.0 ** np.arange(-4, 3)
>>> grid = GridSearchCV(LogisticRegression(), param_grid={'C' : C_range})
>>> clf = Pipeline([('preproc', StandardScaler()),
...                 ('classifier', grid)])
```

Данные в наборе данных расположены не в случайном порядке: похожие изображения находятся рядом. Поэтому при перекрестной проверке будем перетасовывать данные, чтобы данные в каждой группе были более репрезентативны:

```
>>> cv = cross_validation.KFold(len(features), 5,
...                         shuffle=True, random_state=123)
>>> scores = cross_validation.cross_val_score(
...    clf, features, labels, cv=cv)
>>> print('Верность: {:.1%}'.format(scores.mean()))
Верность: 72.1%
```

Для четырех классов совсем неплохо, но ниже мы покажем, как улучшить результат, взяв другой набор признаков. На самом деле, мы увидим, что для получения оптимального результата необходимо сочетать эти признаки с другими методами.

Локальные представления признаков

Сравнительно недавно в области машинного зрения были разработаны методы на основе локальных признаков. Локальные признаки вычисляются по небольшому участку изображения – в отличие от рассмотренных ранее признаков, вычисляемых по всему изображению. В mahotas поддерживается вычисление признаков типа **SURF** (Speeded Up Robust Features – ускоренно вычисляемые устойчивые признаки). Есть и несколько других, самый известный – первоначально предложенный SIFT. Эти признаки спроектированы с учетом

устойчивости относительно вращения и освещенности (то есть они мало изменяются при изменении освещения).

Нам нужно решить, где вычислять эти признаки. Обычно выбирают одну из трех возможностей:

- в случайных областях;
- на сетке;
- выявить интересные участки изображения (эта техника известна под названием «определение ключевых точек»).

Каждый из этих подходов при определенных обстоятельствах дает хорошие результаты. В mahotas поддерживаются все три. Определение ключевых точек лучше всего работает, если есть основания полагать, что ключевые точки действительно соответствуют наиболее важным участкам изображения.

Мы применим метод ключевых точек. Вычислить признаки в mahotas легко: нужно импортировать подходящий подмодуль и вызвать функцию `surf.surf`:

```
>>> from mahotas.features import surf
>>> image = mh.demos.load('lena')
>>> image = mh.colors.rgb2gray(im, dtype=np.uint8)
>>> descriptors = surf.surf(image, descriptor_only=True)
```

Флаг `descriptors_only=True` означает, что нас интересуют только сами дескрипторы, а не положение пикселей, размер или ориентация. Можно было бы вместо этого воспользоваться методом частой выборки, вызвав функцию `surf.dense`:

```
>>> from mahotas.features import surf
>>> descriptors = surf.dense(image, spacing=16)
```

При этом возвращаются значения дескрипторов, вычисленные в точках, находящихся на расстоянии 16 пикселей друг от друга. Поскольку положения точек фиксированы, метаинформация о ключевых точках не очень интересна и по умолчанию не возвращается. В любом случае результат (дескрипторы) – это массив $n \times 64$, где n – количество точек в выборке. Оно зависит от размера изображения, его содержания и параметров функции. В данном случае мы взяли значения по умолчанию, это дает несколько сотен дескрипторов на одно изображение.

Мы не можем непосредственно загрузить эти дескрипторы в алгоритм опорных векторов, логистической регрессии или еще какой-нибудь алгоритм классификации. Существует несколько способов использовать дескрипторы изображения. Можно их просто усреднить,

но результаты получатся плохонькие, потому что при таком подходе отбрасывается важная информация о местоположении. В нашем примере мы просто получили бы еще один глобальный набор признаков, основанных на измерениях у границ.

Решение – воспользоваться моделью **набора слов** – появилось совсем недавно. Впервые в такой форме оно было опубликовано в 2004 году. Его можно отнести к идеям типа «как же мы раньше-то проглядели»: реализовать очень легко, а результаты получаются замечательные.

На первый взгляд, странно говорить о *словах* применительно к изображениям. Понять эту идею будет проще, если представлять себе не написанные слова, которые легко разделить, а произносимые. Произнесенное слово каждый раз звучит немного иначе, а разные люди произносят одно и то же слово по-разному. Поэтому волновые формы слова не будут повторяться. Тем не менее, применив к волновым формам кластеризацию, мы надеемся восстановить большую часть структуры, так что все варианты произношения данного слова окажутся в одном кластере. Даже если этот процесс не совершенен (а он таки не совершенен), все равно можно говорить о группировке волновых форм в слова.

Ту же самую операцию можно проделать и с данными изображения: кластеризовать похожие участки всех изображений и назвать кластеры **визуальными словами**.

Количество используемых слов не оказывает существенного влияния на качество алгоритма. Естественно, если их количество совсем уж мало (10–20 при нескольких тысячах изображений), то хорошей работы от системы ожидать не приходится. Но и при чрезмерно большом количестве слов (например, если их намного больше, чем изображений) система тоже будет работать не оптимально. Между этими крайностями часто лежит протяженное плато, на котором качество результата слабо зависит от количества слов. Значения 256, 512 или, – при очень большом числе изображений, – 1024 должны дать приемлемый результат.

Начнем с вычисления признаков:

```
>>> alldescriptors = []
>>> for im in images:
...   im = mh.imread(im, as_grey=True)
...   im = im.astype(np.uint8)
...   alldescriptors.append(surf.dense(image, spacing=16))
```

```
>>> # получить все дескрипторы в одном массиве
>>> concatenated = np.concatenate(alldescriptors)
>>> print('Количество дескрипторов: {}'.format(len(concatenated)))
Количество дескрипторов: 2489031
```

Мы получили свыше 2 миллионов локальных дескрипторов. Теперь с помощью кластеризации методом K средних найдем центроиды. Можно было бы использовать все дескрипторы, но для быстроты мы ограничимся только малой частью:

```
>>> # использовать каждый 64-ый вектор
>>> concatenated = concatenated[::64]
>>> from sklearn.cluster import KMeans
>>> k = 256
>>> km = KMeans(k)
>>> km.fit(concatenated)
```

Это займет некоторое время, но по завершении объект `km` будет содержать сведения о центроидах. Теперь вернемся к дескрипторам и построим векторы признаков:

```
>>> sfeatures = []
>>> for d in alldescriptors:
...  c = km.predict(d)
...  sfeatures.append(
...     np.array([np.sum(c == ci) for ci in range(k)])
...  )
>>> # строим один массив и преобразовываем в тип float
>>> sfeatures = np.array(sfeatures, dtype=float)
```

По выходе из цикла элемент `sfeatures[fi, fj]` массива показывает, сколько раз элемент `fj` встречается в изображении `fi`. Этот массив можно было бы вычислить и быстрее, воспользовавшись функцией `np.histogram`, но подготовка аргументов для нее – не вполне тривиальное занятие. Мы преобразуем результат к типу с плавающей точкой, поскольку не хотим возиться с целочисленной арифметикой и семантикой округления.

Теперь каждое изображение представлено одним массивом признаков, и размеры всех массивов одинаковы (равны количеству кластеров, в нашем случае 256). Таким образом, можно применить стандартные методы классификации:

```
>>> scores = cross_validation.cross_val_score(
...    clf, sfeatures, labels, cv=cv)
>>> print('Верность: {:.1%}'.format(scores.mean()))
Верность: 62.6%
```

Хуже, чем раньше! Мы что же, ничего не выиграли?

На самом деле, выиграли, потому что можем объединить все признаки и получить верность 76.1%:

```
>>> combined = np.hstack([features, features])
>>> scores = cross_validation.cross_val_score(
...    clf, combined, labels, cv=cv)
>>> print('Верность: {:.1%}'.format(scores.mean()))
Верность: 76.1%
```

Это лучший из достигнутых результатов – лучше, чем при любом наборе признаков по отдельности. Объясняется это тем, что локальные признаки SURF существенно отличаются от глобальных признаков изображения и потому привносят новую информацию в окончательный результат.

Резюме

Мы познакомились с классическим основанным на признаках подходом к обработке изображений в контексте машинного обучения: перейдя от миллионов пикселей к немногим числовым признакам, мы смогли воспользоваться классификатором на базе логистической регрессии. Все технологии, изученные в предыдущих главах, волшебным образом оказались применимы к задачам машинного зрения. Один из примеров – поиск похожих изображений в наборе данных.

Мы также научились использовать для классификации локальные признаки в виде модели набора слов. Это очень современный подход к машинному зрению, который, с одной стороны, дает отличные результаты, а, с другой, нечувствителен к несущественным деталям изображения, например освещению и даже неравномерному освещению одного и того же изображения. Мы также воспользовались кластеризацией не ради нее самой, а как полезным промежуточным шагом классификации.

Мы работали с mahotas, одной из основных библиотек машинного зрения на Python. Но есть и другие, поддерживаемые ничуть не хуже. Skimage (scikit-image) близка по духу, но строит другой набор признаков. OpenCV – отличная библиотека, написанная на C++ и имеющая интерфейс к Python. Все они могут работать с массивами NumPy, поэтому можно свободно пользоваться функциями из разных библиотек для построения сложных конвейеров машинного зрения.

В следующей главе мы займемся другим видом машинного обучения: понижением размерности. В предыдущих главах мы видели,

что с вычислительной точки зрения сгенерировать много признаков совсем нетрудно. Но часто желательно уменьшить число признаков ради повышения быстродействия, наглядности или улучшения качества результатов. Далее мы узнаем, как это делается.

ГЛАВА 11.
Понижение размерности

Мусор на входе, мусор на выходе – на страницах этой книги мы не раз видели, как справедливость этого высказывания подтверждается, когда мы применяем методы машинного обучения к имеющимся данным. Оглядываясь назад, мы понимаем, что самые интересные вопросы возникают на этапе подготовки признаков, когда мы стараемся использовать знания о существе проблемы для тщательного отбора признаков, которые, хочется надеяться, алгоритм сумеет переварить.

В этой главе мы двинемся в обратном направлении: смысл задачи понижения размерности состоит в том, чтобы отбросить нерелевантные или избыточные признаки. На первый взгляд, избавление от признаков противоречит интуиции, ведь чем больше информации, тем лучше, разве не так? Да и потом, даже если в нашем наборе данных и имеются избыточные признаки, разве алгоритм обучения не сможет быстро выявить их и назначить им нулевой вес? Приведенные ниже аргументы показывают, почему на практике до сих пор следует всеми силами стремиться к понижению размерности задачи.

- Лишние признаки могут ввести алгоритм обучения в заблуждение. Это относится не ко всем алгоритмам (например, метод опорных векторов работает тем лучше, чем выше размерность), но некоторые модели предпочитают поменьше измерений.
- Чем больше признаков, тем больше параметров нужно настраивать и тем выше риск переобучения.
- Возможно, что у данных, собранных для решения задачи, размерность завышена искусственно, а истинная размерность невелика.
- Меньше размерность = более быстрое обучение = больше вариантов параметров можно рассмотреть за фиксированное время = лучше конечный результат.
- Если мы хотим визуализировать данные, то ограничены двумя или тремя измерениями.

Итак, в этой главе мы покажем, как избавиться от мусора в данных, сохранив в то же время всё ценное.

План действий

Методы понижения размерности можно разбить на две большие группы: отбор признаков и выделение признаков. В предыдущих главах мы уже сталкивались с отбором признаков, когда придумывали, анализировали, а иногда и отбрасывали некоторые признаки. В этой главе мы продемонстрируем, как использование статистических методов, а именно вычисление корреляции и взаимной информации, помогает решать эту задачу в многомерных пространствах признаков. Выделение признаков – это попытка преобразовать исходное пространство признаков в пространство более низкой размерности. Это особенно полезно, когда мы не в состоянии избавиться от признаков, применяя методы отбора, но и работать с таким большим числом признаков невозможно. Мы продемонстрируем **анализ главных компонент** (principal component analysis, PCA), **линейный дискриминантный анализ** (linear discriminant analysis, LDA) и **многомерное шкалирование** (multidimensional scaling, MDS).

Отбор признаков

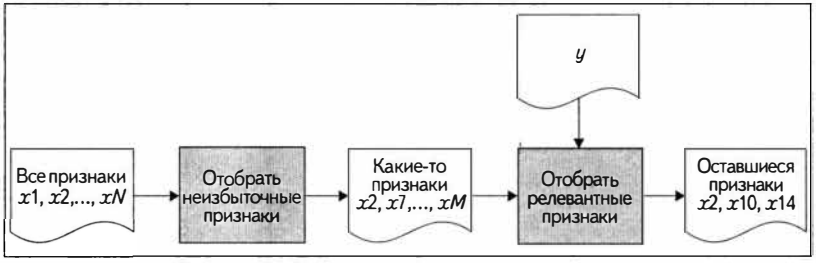
Алгоритм машинного обучения работает оптимально, если поданные ему на вход признаки не зависят друг от друга, но сильно зависят от прогнозируемого значения. Это означает, что каждый новый признак добавляет существенную информацию. А удаление любого признака приводит к снижению качества.

Если признаков раз-два и обчелся, то можно нарисовать матрицу диаграмм разброса (по одной для каждой пары признаков). С ее помощью легко выявить связи между признаками. Если для какой-то пары имеется очевидная зависимость, то можно либо отказаться от какого-то признака, либо подумать над выработкой нового, более качественного признака из этих двух.

Но обычно признаков гораздо больше. Вспомните классификацию качества ответа на основе модели набора слов – тут нам пришлось бы нарисовать 1000 × 1000 диаграмм разброса. Ниже мы опишем два общих способа решения задачи: фильтры и обертки.

Выявление избыточных признаков с помощью фильтров

Задача фильтра – расчистить «лес» признаков независимо от того, какие методы машинного обучения будут применены впоследствии. Для нахождения избыточных или нерелевантных признаков используются статистические методы. Если найдены избыточные признаки, то из каждой группы мы оставляем только один, а нерелевантные признаки просто удаляются. Общая схема работы фильтра изображена ниже.



Корреляция

Корреляция выявляет линейные связи между парами признаков. На рисунках ниже мы видим разные степени корреляции, а также потенциальную линейную зависимость, изображенную красной пунктирной линией (аппроксимированной линейным полиномом). Коэффициент корреляции Пирсона (r) $Cor(X_1, X_2)$ над каждым графиком вычислен с помощью функции `pearsonr()` из модуля `scipy.stat`.

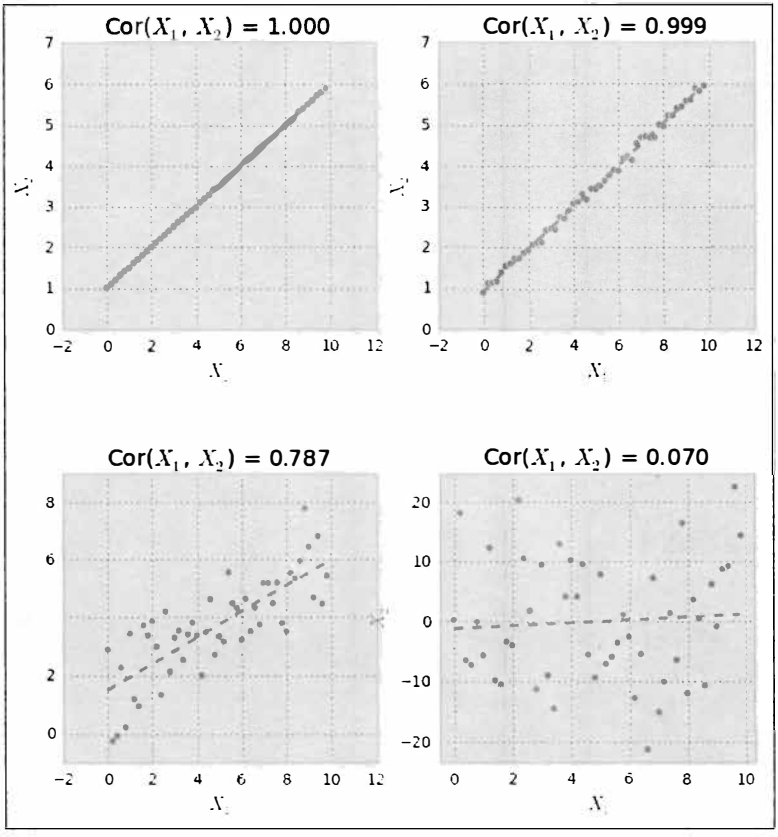
Получив два ряда данных одинакового размера, эта функция возвращает кортеж, содержащий коэффициент корреляции и p-значение. P-значение описывает вероятность того, что эти ряды данных были сгенерированы некореллированной системой. Иными словами, чем выше p-значение, тем меньше доверия коэффициенту корреляции.

```
>>> from scipy.stats import pearsonr
>>> pearsonr([1,2,3], [1,2,3.1])
>>> (0.99962228516121843, 0.017498096813278487)
>>> pearsonr([1,2,3], [1,20,6])
>>> (0.25383654128340477, 0.83661493668227405)
```

В первом случае есть четкое указание на то, что оба ряда коррелируют. Во втором значение r также отнюдь не нулевое, но поскольку

p-значение равно 0.84, то мы делаем вывод, что коэффициент корреляции не обладает статистической значимостью и обращать на него внимания не стоит.

Взгляните на следующие графики.

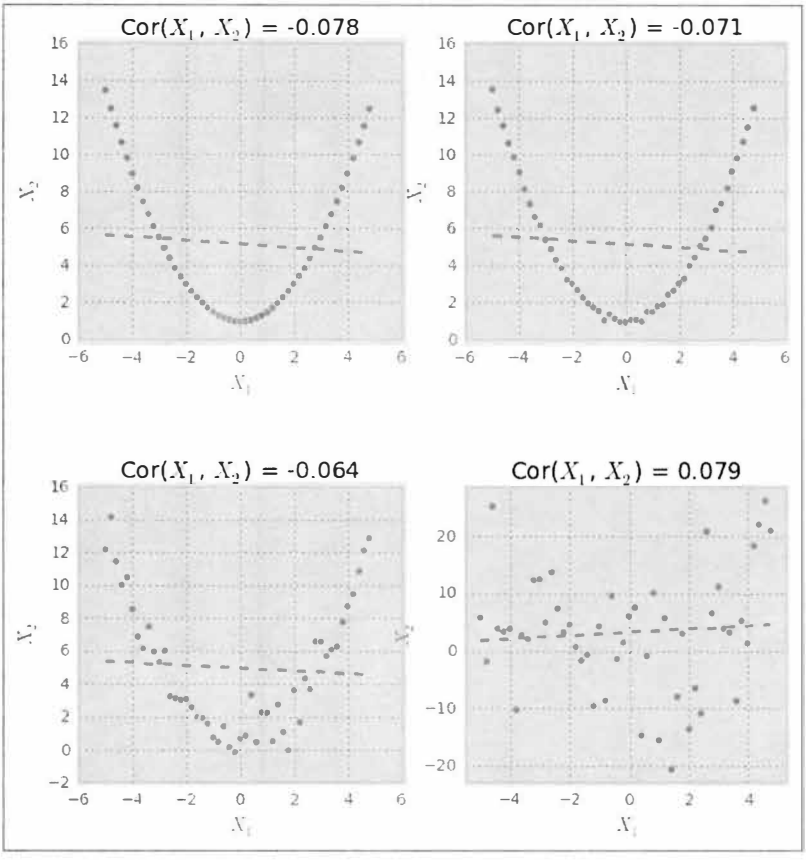


В первых трех случаях коэффициенты корреляции высокие, поэтому, наверное, стоит отбросить X_1 или X_2, потому что они несут очень похожую, а, может быть, и вовсе одну и ту же информацию.

Но в последнем случае оба признаки нужно оставить. Решение, разумеется, определяется p-значением.

В примере все было замечательно, но реальность редко бывает столь благосклонна к нам. Крупный недостаток отбора признаков на основе корреляции заключается в том, что таким образом можно выявить лишь линейные зависимости (такие, которые можно описать

прямой линией). Попытавшись применить корреляцию к нелинейным данным, мы увидим, в чем проблема. В следующем примере зависимость квадратичная.



Человек сразу видит, что между X_1 и X_2 существует зависимость на всех графиках, кроме последнего, однако коэффициент корреляции ее не отражает. Очевидно, что корреляция полезна для нахождения линейных зависимостей, но перестает работать для любых других. Иногда удается получить линейную зависимость, применив простое преобразование. Например, на предыдущем графике мы получили бы высокий коэффициент корреляции, если бы построили зависимость X_2 от квадрата X_1. Но на практике такая возможность предоставляется нечасто.

По счастью, в случае нелинейных зависимостей на помощь приходит взаимная информация.

Взаимная информация

Говоря об отборе признаков, мы не должны сосредотачиваться на типе зависимости, как в предыдущем разделе (линейная зависимость). Вместо этого нужно думать о том, сколько информации привносит признак (при условии, что уже есть другой признак).

Чтобы разобраться в этой идее, давайте предположим, что мы хотим использовать признаки `house_size` (размер дома), `number_of_levels` (количество этажей) и `avg_rent_price` (средняя стоимость аренды), чтобы обучить классификатор, который определяет, есть в доме лифт или нет. Интуитивно очевидно, что если мы знаем `house_size`, то знать `number_of_levels` необязательно, поскольку это в некотором роде избыточная информация. Но `avg_rent_price` – другое дело, потому что вывести стоимость аренды из одного лишь размера дома или количества этажей невозможно. Поэтому было бы разумно оставить среднюю стоимость аренды и один из двух других признаков.

Понятие взаимной информации формализует это рассуждение путем вычисления количества информации, общей для двух признаков. Но в отличие от корреляции, в основу кладется не последовательность данных, а распределение. Чтобы понять, как это устроено, придется познакомиться с понятием энтропии информации.

Пусть имеется правильная монета. До того как мы ее подбросим, неопределенность исхода – выпадет орел или решка – максимальная, т. к. вероятность обоих исходов равна 50%. Эту неопределенность можно измерить с помощью энтропии информации, введенной Клодом Шенноном:

$$H(X) = \sum_{i=1}^{n} p(X_i) \log_2 p(X_i)$$

В случае правильной монеты есть два исхода: X_0 – выпадение орла и X_1 – выпадение решки, причем $p(X_0) = p(X_1) = 0.5$.

Отсюда

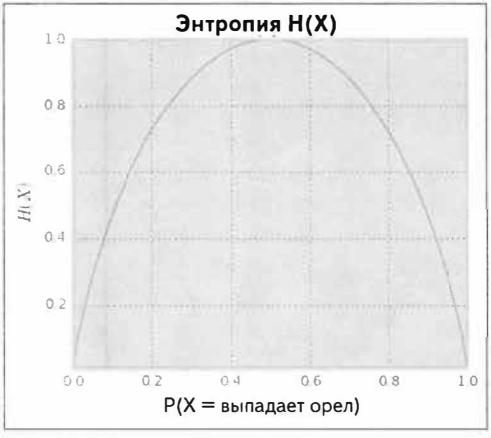
$$H(X) = -p(X_0) \log_2 p(X_0) - p(X_1) \log_2 p(X_1) =$$
$$= -0.5 \log_2 (0.5) - 0.5 \log_2 (0.5) = 1.0$$

Для удобства можно воспользоваться также функцией `scipy.stats.entropy([0.5,0.5], base=2)`. Мы задали параметр `base` (основание логарифма) равным 2, чтобы получить тот же результата, что и выше. По умолчанию в этой функции используется натуральный логарифм, `np.log()`. Вообще говоря, основание несущественно (если только во всех расчетах оно одно и то же).

Теперь представим, что нам заранее известно, что монета неправильная: орел выпадает с вероятностью 60%:

$$H(X) = -0.6 \cdot \log_2 0.6 - 0.4 \cdot \log_2 0.4 = 0.97$$

Как видим, неопределенность уменьшилась. Уменьшение продолжится по мере удаления от значения 0.5 и достигнет минимума, когда вероятность выпадение орла составит 0 или 100%. Это показано на рисунке ниже.



Модифицируем формулу энтропии $H(X)$, применив ее к двум признакам вместо одного, так чтобы она измеряла, насколько уменьшается неопределенность X, когда мы узнаем о Y. Таким образом, мы сможем узнать, как один признак снижает неопределенность другого.

Например, в отсутствие какой-либо информации о погоде, мы не можем сказать, идет на улице дождь или нет – полная неопределенность. Но если мы знаем, что трава мокрая, то неопределенность уменьшается (правда, нужно еще проверить, не включена ли поливальная система).

Формально взаимная информация определяется следующим образом:

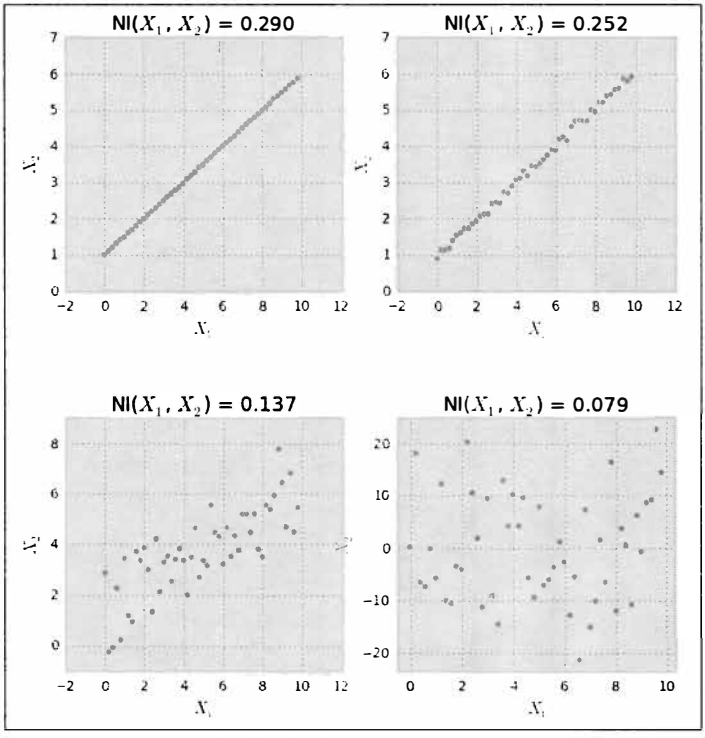
$$I(X;Y)=\sum_{i=1}^{m}\sum_{j=1}^{n}P(X_i,Y_j)\log_2\frac{P(X_i,Y_j)}{P(X_i)P(Y_j)}$$

Выглядит устрашающе, но на самом деле тут ничего нет, кроме сумм и произведений. Например, чтобы вычислить $P()$, нужно распределить значения признаков по интервалам, а затем подсчитать, сколько значений оказалось в каждом интервале. На графиках ниже мы взяли 10 интервалов.

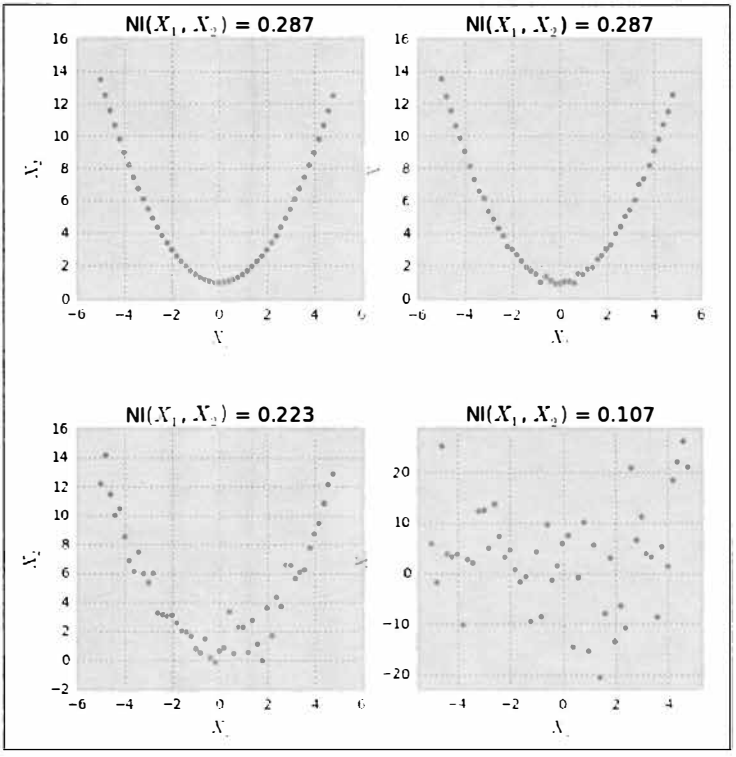
Чтобы привести взаимную информацию к диапазону [0,1], нужно разделить ее на сумму энтропий отдельных признаков. В результате получаем нормированную взаимную информацию:

$$NI(X;Y)=\frac{I(X;Y)}{H(X)+H(Y)}$$

Взаимная информация, в отличие от корреляции, находит не только линейные зависимости, как видно из следующих графиков.



Мы видим, что взаимная информация может показать силу линейной связи. А на следующем рисунке показано, что она работает и для квадратичных зависимостей.



Таким образом, мы должны вычислить нормированную взаимную информацию для всех пар признаков. Если для какой-то пары значение оказалось слишком большим (еще надо определить, что это значит), то один их признаков следует отбросить. В случае регрессии мы могли бы отбросить признак, для которого слишком мала взаимная информация с желаемым результирующим значением.

Этот подход годится, когда набор признаков не слишком велик. Но, начиная с какого-то момента, процедура становится слишком дорогой, поскольку объем вычислений возрастает квадратично (т. к. вычисляется взаимная информация для каждой пары признаков).

Еще один существенный недостаток фильтрации заключается в отбрасывании признаков, которые кажутся бесполезными по отдельно-

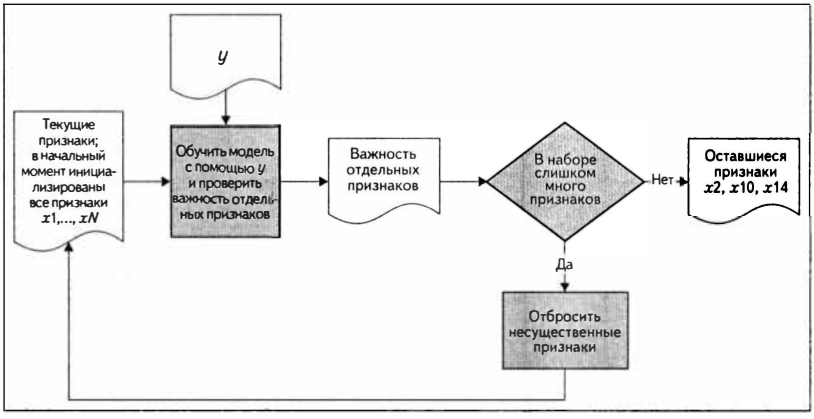
сти. Однако часто существует группа признаков, каждый из которых вроде бы абсолютно не зависит от целевой переменной, но вместе они «играют». Чтобы оставить такие признаки, нужны обертки.

Применение оберток для задания модели вопросов о признаках

Фильтры, конечно, могут оказать огромную помощь в избавлении от бесполезных признаков, но у них есть пределы. После фильтрации могут остаться признаки, которые не зависят друг от друга и, похоже, как-то связаны с целевой переменной, но с точки зрения модели абсолютно бесполезны. Рассмотрим следующие данные, которые описывают функцию XOR. По отдельности ни A, ни B не показывают никакой зависимости от Y, но, если взять их в совокупности, то такая зависимость, несомненно, присутствует:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Так почему бы не попросить саму модель высказаться по поводу отдельных признаков? Именно в этом и состоит смысл оберток, показанный на следующей блок-схеме.



Здесь мы перенесли вычисление важности признаков в процесс обучения модели. К сожалению (хотя это и понятно), важность признака теперь описывается не бинарным значением да-нет, а рангом. Поэтому нам все равно придется самостоятельно выбрать пороговое значение, то есть решить, какие признаки оставить, а какие отбросить.

В библиотеке scikit-learn, а точнее в пакете `sklearn.feature_selection`, имеется целый ряд отличных классов-оберток. Рабочей лошадкой является класс `RFE`, его имя расшифровывается как «recursive feature elimination» (рекурсивное исключение признаков). Он принимает объект-оценщик и количество признаков, которые нужно оставить, а затем обучает оценщик на различных наборах признаков, пока не найдет достаточно малый поднабор. Сам объект `RFE` предоставляет интерфейс оценщика, то есть обертывает настоящий оценщик, переданный в качестве параметра.

В примере ниже мы создаем искусственную задачу классификации с 100 образцами, применяя вспомогательную функцию `make_classification()`. Она позволяет сказать, что мы хотим создать 10 признаков, из которых только три представляют реальную ценность для решения задачи классификации:

```
>>> from sklearn.feature_selection import RFE
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.datasets import make_classification
>>> X,y = make_classification(n_samples=100, n_features=10,
n_informative=3, random_state=0)
>>> clf = LogisticRegression()
>>> clf.fit(X, y)
>>> selector = RFE(clf, n_features_to_select=3)
>>> selector = selector.fit(X, y)
>>> print(selector.support_)
[False True False True False False False False True False]
>>> print(selector.ranking_)
[4 1 3 1 8 5 7 6 1 2]
```

На практике проблема, конечно, состоит в том, как узнать правильное значение параметра `n_features_to_select`. По правде говоря, никак. Но обычно мы можем взять выборку данных и, поэкспериментировав с разными значениями параметров, прикинуть, как обстоят дела.

Впрочем, при использовании оберток особая точность и не нужна. Давайте попробуем разные значения `n_features_to_select` и посмотрим, как изменяются `support_` и `ranking_`.

n_features_to_select	support_	ranking_
1	[False False False True False False False False False]	[6 3 5 1 10 7 9 2 4]
2	[False False False True False False False False True False]	[5 2 4 1 9 6 8 7 1 3]
3	[False True False True False False False False True False]	[4 1 3 1 8 5 7 6 1 2]
4	[False True False True False False False False True True]	[3 1 2 1 7 4 6 5 1 1]
5	[False True True True False False False False True True]	[2 1 1 1 6 3 5 4 1 1]
6	[True True True True False False False False True True]	[1 1 1 1 5 2 4 3 1 1]
7	[True True True True False True False False True True]	[1 1 1 1 4 1 3 2 1 1]
8	[True True True True False True False True True True]	[1 1 1 1 3 1 2 1 1 1]
9	[True True True True False True True True True True]	[1 1 1 1 2 1 1 1 1 1]
10	[True True True True True True True True True True]	[1 1 1 1 1 1 1 1 1 1]

Как видим, результат очень устойчив. Признаки, использованные, когда мы запрашивали небольшой набор, выбираются и при увеличении размера набора. И, кроме того, мы можем положиться на то, что разделение набора данных на обучающие и тестовые вовремя предупредит нас, если мы двинемся в неверном направлении.

Другие методы отбора признаков

В литературе по машинному обучению описано еще несколько методов отбора признаков. Некоторые даже не выглядят как таковые, потому что встроены в процесс обучения (не путайте с описанными выше обертками). Например, в решающих деревьях механизм отбора признаков неотделим от самой идеи алгоритма. В других методах применяется тот или иной вид регуляризации, штрафующий за сложность модели, так что процесс обучения направлен на получения хороших и в то же время «простых» моделей. Для этого важность признаков, не вносящих существенного вклада, понижается до нуля, после чего они отбрасываются (L1-регуляризация).

Так что будьте начеку! Зачастую эффективность алгоритма машинного обучения в немалой степени обусловлена встроенным в него методом отбора признаков.

Выделение признаков

Бывает так, что даже после исключения избыточных признаков и отбрасывания нерелевантных количество оставшихся признаков слишком велико. И тогда любой метод обучения работает плохо, а, учитывая размер пространства признаков, мы понимаем, что поделать ничего нельзя. Тогда мы приходим к выводу, что придется «резать по живому» – избавляться от некоторых признаков, хотя здравый смысл говорит, что они полезны. Другая ситуация, в которой необходимо понизить размерность, а отбор признаков не помогает, – желание визуализировать данные. В этом случае мы не можем оставить больше трех признаков.

На помощь приходят методы выделения признаков. Они изменяют структуру пространства признаков, приспосабливая его к модели, или просто уменьшают размерность до двух или трех, чтобы можно было наглядно представить зависимости.

Как и раньше, методы выделения признаков бывают линейными и нелинейными. Мы представим по одному методу из каждой категории: анализ главных компонент (линейный) и многомерное шкалирование (нелинейный). Они хорошо известны и часто применяются, но есть и много других интересных и мощных методов выделения признаков.

Об анализе главных компонент

К анализу главных компонент (principal component analysis, PCA) обычно первым делом прибегают, когда нужно сократить количество признаков, но неизвестно, какой метод выделения признаков подойдет лучше. Ограничением PCA является линейность, но вполне возможно, его окажется достаточно для обучения хорошей модели. Добавьте сюда полезные математические свойства, которыми обладает этот алгоритм, и скорость, с которой он находит преобразованное пространство признаков и затем осуществляет преобразование между исходным и новым пространством, – и можете быть уверены, что он станет одним из ваших любимых инструментов машинного обучения.

Итак, имея исходное пространство признаков, алгоритм PCA находит его линейную проекцию на пространство меньшей размерности со следующими свойствами:

- достигается максимум остаточной дисперсии;
- достигается минимум ошибки реконструкции (при попытке возврата от преобразованных признаков к исходным).

Поскольку PCA просто преобразует исходные данные, его можно применять к задачам классификации и регрессии. В этом разделе мы обсудим этот метод на примере задачи классификации.

Принцип работы PCA

Алгоритм PCA основан на линейной алгебре, в которую мы вдаваться не станем. Однако принцип его работы описать несложно.

1. Центрировать данные, вычтя из каждого элемент среднее.
2. Вычислить ковариационную матрицу.
3. Вычислить собственные векторы ковариационной матрицы.

Если начать с N признаков, то алгоритм вернет преобразованное пространство, тоже N-мерное (пока что мы ничего не выиграли). Важно, однако, то, что собственные значения показывают, какая часть дисперсии приходится на соответственный собственный вектор.

Пусть в начале было $N = 1000$ признаков, и мы знаем, что модель хорошо работает, только если признаков не больше 20. Тогда мы просто выбираем 20 собственных векторов с наибольшими собственными значениями.

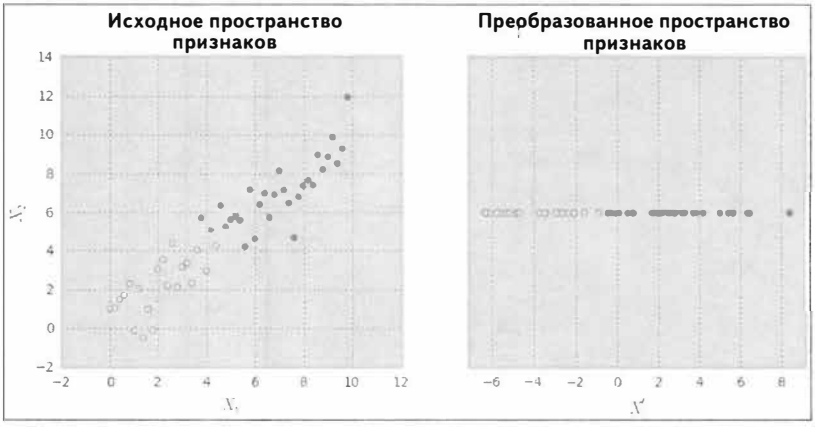
Применение PCA

Рассмотрим следующий искусственный набор данных, показанный ниже на левом рисунке:

```
>>> x1 = np.arange(0, 10, .2)
>>> x2 = x1+np.random.normal(loc=0, scale=1, size=len(x1))
>>> X = np.c_[(x1, x2)]
>>> good = (x1>5) | (x2>5) # какие-то произвольные классы
>>> bad = ~good # чтобы пример хорошо выглядел
```

В библиотеке scikit-learn есть класс `PCA`, входящий в пакет `decomposition`. В примере выше отчетливо видно, что для описания данных вполне хватило бы одного измерения. Это можно указать, задав параметр `n_components`:

```
>>> from sklearn import linear_model, decomposition, datasets
>>> pca = decomposition.PCA(n_components=1)
```



Кроме того, можно воспользоваться методами `fit()` и `transform()` объекта `pca` (или их комбинацией `fit_transform()`), чтобы проанализировать данные и спроецировать их на преобразованное пространство признаков:

```
>>> Xtrans = pca.fit_transform(X)
```

Как мы и просили, размерность `Xtrans` равна 1. Результат показан на правом рисунке выше. В данном случае результат даже является линейно разделимым, так что для различения классов сложный классификатор не понадобится.

Чтобы понять, что такое ошибка реконструкции, взглянем на дисперсию данных, сохраненную в результате преобразования:

```
>>> print(pca.explained_variance_ratio_)
>>> [ 0.96393127]
```

Это означает, что после перехода от двух измерений к одному дисперсия по-прежнему равна 96%.

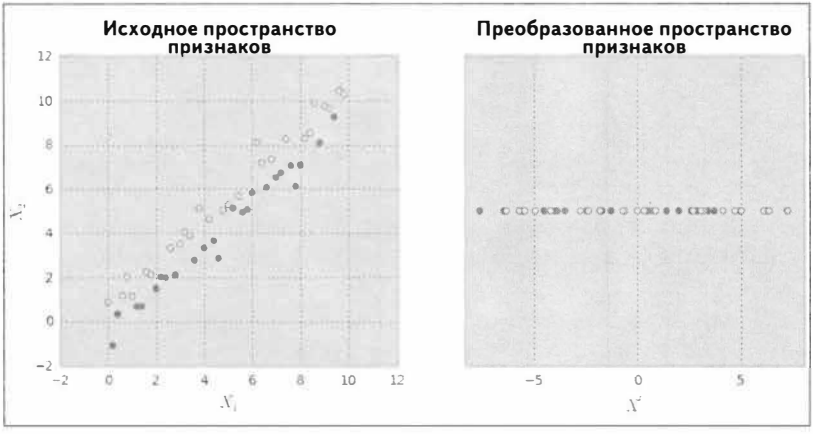
Разумеется, не всегда все так просто. Часто желательное число измерений заранее неизвестно. В таком случае мы не задаем параметр `n_components`, предлагая объекту `PCA` вычислить полное преобразование. После аппроксимации данных поле `explained_variance_ratio_` содержит массив отношений в порядке убывания. Первый член – это отношение базисного вектора, описывающего направление с наибольшей дисперсией, второй – отношение в направлении второй по величине дисперсии и т. д. Построив график, мы сразу видим, сколько компонент потребуется: хорошей гипотезой будет число компонент непосредственно до изгиба кривой.

График, на котором показана зависимость объясненной дисперсии от количества компонент, называется «графиком каменистой осыпи», или графиком Кеттела. Интересный пример комбинирования графика каменистой осыпи с сеточным поиском с целью нахождения оптимальных параметров для задачи классификации приведен на странице `http://scikit-learn.sourceforge.net/stable/auto_examples/plot_digits_pipe.html`.

Ограничения PCA и чем может помочь LDA

У PCA, как у всякого линейного алгоритма, имеются ограничения при работе с нелинейными зависимостями. Не вдаваясь в детали, скажем, что существуют обобщения PCA, например Kernel PCA, в которых благодаря нелинейному преобразованию все-таки удается применить метод PCA.

Еще одно любопытное слабое место PCA, которое мы здесь рассмотрим, обнаруживается при попытке применить его к задачам классификации особого вида. Заменив выражение `good = (x1 > 5) | (x2 > 5)` выражением `good = x1 > x2`, мы сразу же увидим, в чем проблема.

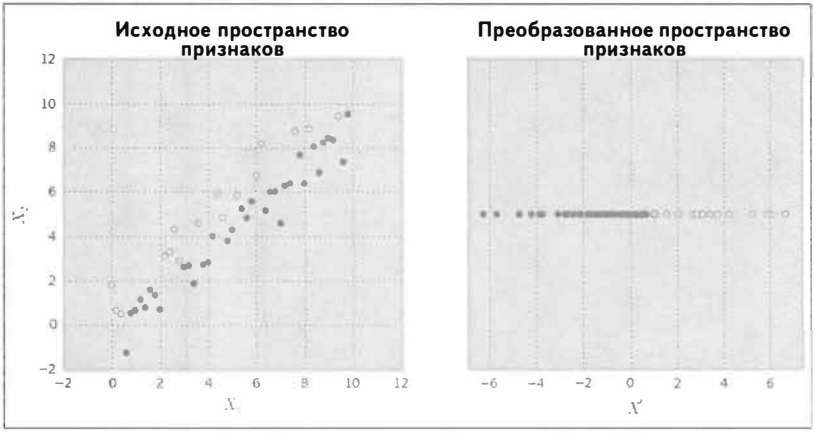


Здесь классы распределены вдоль оси не с наибольшей, а со второй по величине дисперсией. Очевидно, с такой задачей PCA не справляется. Поскольку мы не сообщили PCA никакой информации о метках классов, ждать от него чего-то лучшего не приходится.

В таких случаях на помощь приходит линейный дискриминантный анализ (LDA). Идея этого метода заключается в том, чтобы максимизировать расстояния между точками, принадлежащими разным классам, но минимизировать расстояния между точками из одного класса. Не углубляясь в теоретические основы, просто покажем, как этот метод используется:

```
>>> from sklearn import lda
>>> lda_inst = lda.LDA(n_components=1)
>>> Xtrans = lda_inst.fit_transform(X, good)
```

Вот и все. Обратите внимание, что в отличие от примера PCA мы передали методу `fit_transform()` метки классов. Таким образом, PCA – алгоритм выделения признаков без учителя, а LDA – с учителем. Результаты выглядит, как и следовало ожидать:



Но тогда зачем вообще применять PCA, почему бы просто не пользоваться LDA всегда? Да потому что не все так просто. Когда число классов растет, а количество примеров каждого класса уменьшается, LDA перестает быть такой уж заманчивой альтернативой. Кроме того, PCA менее чувствителен к различиям в обучающих данных, чем LDA. Поэтому, когда нас спрашивают, какой метод мы рекомендуем, мы можем ответить лишь «зависит от ситуации».

Многомерное шкалирование

В то время как PCA пытается выполнить оптимизацию с сохранением дисперсии, метод многомерного шкалирования (MDS) стремит-

ся по возможности сохранить относительные расстояния, уменьшив число измерений. Это полезно, когда требуется наглядно представить многомерный набор данных.

Для алгоритма MDS сами точки не представляют интереса, для него важны расхождения между парами точек, интерпретируемые как расстояния. Поэтому первым делом MDS по N точкам в k-мерном пространстве вычисляет матрицу расстояний, пользуясь функцией метрики d_0, которая измеряется расстояние в исходном пространстве признаков (как правило, обычное евклидово расстояние):

$$\begin{pmatrix} X_{11} & X_{N1} \\ \vdots & \vdots \\ X_{1k} & X_{Nk} \end{pmatrix} \rightarrow \begin{pmatrix} d_o(X_1, X_1) & \cdots & d_o(X_N, X_1) \\ \vdots & \ddots & \vdots \\ d_o(X_1, X_N) & \cdots & d_o(X_N, X_N) \end{pmatrix}$$

Затем MDS пытается расположить точки в пространстве меньшей размерности, так чтобы новые расстояния между ними как можно более походили на исходные. Поскольку MDS чаще всего применяется для визуализации, новая размерность обычно равна 2 или 3.

Рассмотрим следующий простой набор данных из трех точек в пятимерном пространстве. Две точки расположены близко друг к другу, а третья отдалена от них, и мы хотим проиллюстрировать это в двумерном и трехмерном пространстве.

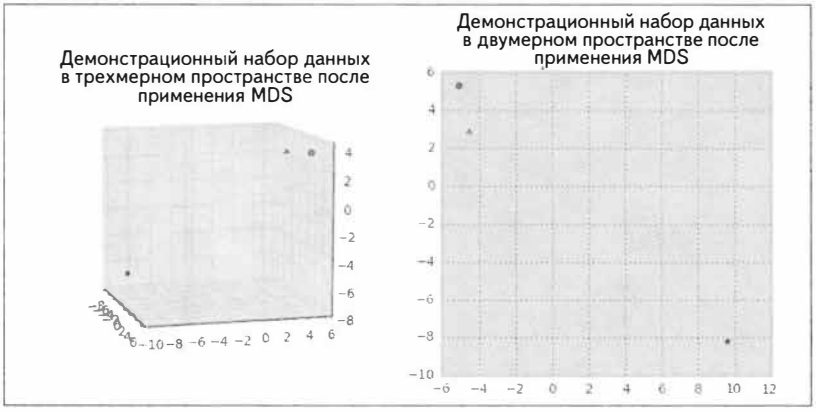
```
>>> X = np.c_[np.ones(5), 2 * np.ones(5), 10 * np.ones(5)].T
>>> print(X)
[[  1.   1.   1.   1.   1.]
 [  2.   2.   2.   2.   2.]
 [ 10.  10.  10.  10.  10.]]
```

С помощью класса `MDS` из пакета `manifold` в библиотеке scikit-learn мы сначала говорим, что хотели бы преобразовать `X` в трехмерное евклидово пространство:

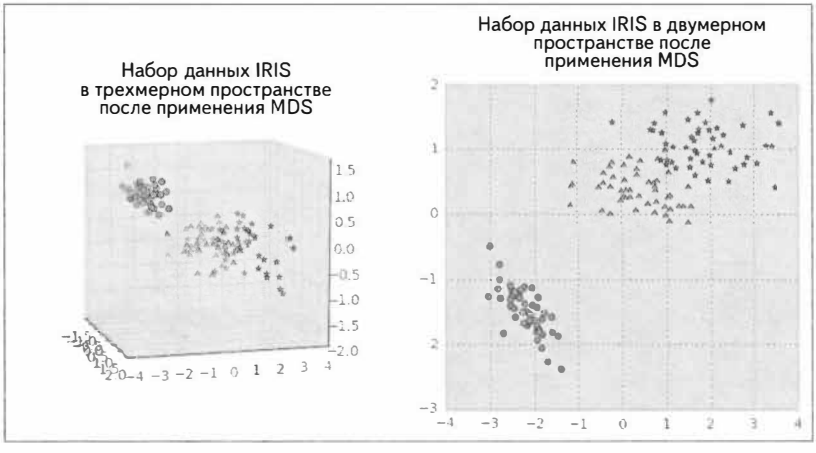
```
>>> from sklearn import manifold
>>> mds = manifold.MDS(n_components=3)
>>> Xtrans = mds.fit_transform(X)
```

Для визуализации в двумерном пространстве нужно соответственно задать параметр `n_components`.

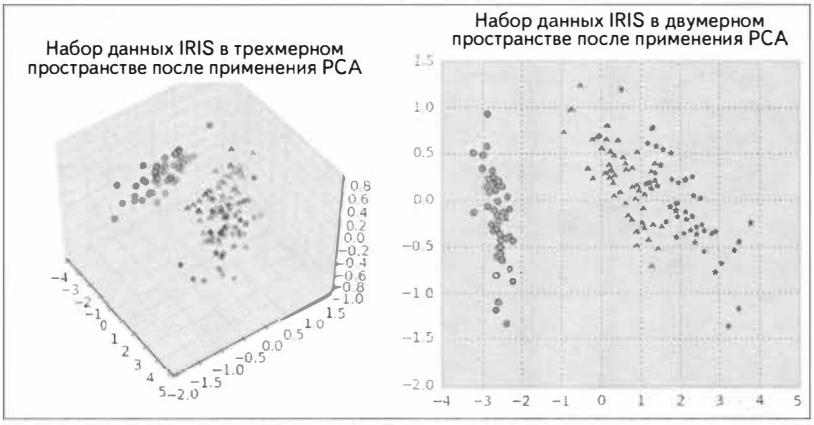
Результаты показаны на рисунках ниже. Треугольник и кружочек находятся рядом, а звездочка – далеко от них.



Рассмотрим чуть более сложный набор данных Iris. Позже мы воспользуемся им для сравнения LDA с PCA. В наборе данных Iris каждый цветок представлен четырьмя атрибутами. Пользуясь приведенным выше кодом, мы могли бы спроецировать его на трехмерное пространство с сохранением (по возможности) относительных расстояний между отдельными видами цветов. Выше мы не задавали метрику, поэтому по умолчанию подразумевалось евклидово расстояние. Это означает, что цветы, которые в достаточной мере «различаются» по четырем атрибутам, должны находиться далеко друг от друга и в трехмерном MDS-шкалированном пространстве, тогда как «близкие» цветы должны располагаться рядом, как на рисунке ниже:



Если же понижать размерность до трех или двух измерений с помощью алгоритма PCA, то мы, как и следовало ожидать, увидим большой разброс цветов, принадлежащих одному классу.



Разумеется, для применения MDS нужно понимать, в каких единицах измеряются отдельные признаки; не исключено, что признаки вообще нельзя сравнивать с помощью евклидовой метрики. Например, категориальные переменные, даже закодированные целыми числами (0 = круг, 1 = звездочка, 2 = треугольник и т. д.), несравнимы (круг ближе к звездочке, чем треугольник?).

Однако коль скоро мы помним об этой проблеме, MDS является полезным инструментом, который выявляет такие виды сходства внутри данных, которые было бы трудно распознать в исходном пространстве признаков.

Копнув чуть глубже, мы поймем, что MDS – не единственный алгоритм, а семейство алгоритмов, из которого мы использовали лишь один. То же справедливо и для PCA. Кроме того, если вы пришли к выводу, что ни PCA, ни MDS не годится для решения вашей задачи, то в модуле `manifold` есть и много других алгоритмов обучения.

Но прежде чем очертя голову кидаться в море алгоритмов, всегда лучше начать с простейшего и посмотреть, как далеко удастся продвинуться. А затем продолжить с этой точки, взяв следующий по сложности алгоритм.

Резюме

Мы узнали, что иногда можно отказаться от использования всех признаков, воспользовавшись методами отбора признаков. Мы также

видели, что в некоторых случаях этого недостаточно и приходится прибегать к методам выделения признаков, которые вскрывают истинную структуру данных, имеющую более низкую размерность, в надежде, что с такими данными модели будет проще справиться.

Конечно, мы смогли дать лишь беглый обзор огромного массива методов понижения размерности. И надеемся, что эта тема вас увлекла, поскольку другие методы только и ждут, когда вы уделите им внимание. Напоследок хотим отметить, что отбор и выделение признаков – это искусство, точно так же, как выбор правильного алгоритма обучения или модели.

В следующей главе мы познакомимся с Jug, небольшим написанным на Python каркасе, который позволяет задействовать для вычислений несколько процессорных ядер или машин. Также мы узнаем о системе AWS, облаке компании Amazon.

ГЛАВА 12.
Когда данных больше

Трудно однозначно сказать, что такое большие данные. Мы примем рабочее определение: если данных так много, что с ними становится неудобно работать, то будем говорить о **больших данных**. Иногда это означает петабайты данных или триллионы транзакций, так что данные не помещаются на один жесткий диск. А иногда данных в сотни раз меньше, но работать с ними все равно трудно.

Почему объем данных превратился в проблему? По мере того как компьютеры становились быстрее, а размер памяти больше, рос и объем данных. На самом деле, рост данных даже опережал рост быстродействия компьютеров, а лишь немногие алгоритмы линейно масштабируются с ростом входных данных. Короче говоря, данные растут быстрее, чем наша способность их обрабатывать.

Для начала мы, опираясь на фундамент, заложенный в предыдущих главах, попробуем поработать со «средними» данными (не очень большими, но уже и не маленькими). Для этого нам понадобится пакет **jug**, умеющий делать следующее:

- строить из задач конвейер;
- кэшировать (сохранять в памяти) промежуточные результаты;
- задействовать несколько процессорных ядер, в том числе машин в grid-сети.

Следующий шаг – переход к по-настоящему *большим данным*; мы увидим, как можно использовать для вычислений облако. Конкретно, мы познакомимся с инфраструктурой Amazon Web Services и с Python-пакетом StarCluster для управления кластером.

Что такое большие данные

Выражение «большие данные» не означает какой-то конкретный объем данных. Это понятие не выражается ни в количестве примеров, ни в количестве гигабайтов, терабайтов или петабайтов, занятых данны-

ми. А означает оно, что объем данных растет быстрее, чем обрабатывающие мощности. Отсюда вытекает ряд следствий.

- Некоторые методы и приемы, хорошо зарекомендовавшие себя в прошлом, теперь нуждаются в пересмотре или замене, потому что не масштабируются на современный объем данных.
- Алгоритмы не могут предполагать, что все исходные данные умещаются в оперативной памяти.
- Управление данными само по себе становится нетривиальной задачей.
- Применение кластеров или многоядерных процессоров становится необходимостью, а не роскошью.

В этой главе мы займемся последней частью головоломки: как задействовать несколько процессорных ядер (на одной или на нескольких машинах) во имя ускорения и организации вычислений. Это будет полезно и для других задач умеренного размера.

Использование jug для построения конвейера задач

Часто у нас имеется простой конвейер: предварительная обработка исходных данных, вычисление признаков и подача признаков на вход алгоритма машинного обучения.

Пакет jug разработал Луис Педро Коэльо, один из авторов этой книги. Это ПО с открытым исходным кодом (распространяется по либеральной лицензии MIT), которое может найти применение во многих областях, но разработано специально для анализа данных. Пакет решает сразу несколько задач.

Он умеет *запоминать* (memoize) результаты вычислений на диске (или в базе данных), то есть если вы попросите повторно произвести вычисления, которые уже выполнялись раньше, то результат будет прочитан с диска, а не вычислен заново.

Он может задействовать несколько процессорных ядер или даже компьютеров, объединенных в кластер. При проектировании jug закладывалась также возможность работы в пакетной среде, где используются системы очередей, например **PBS** (Portable Batch System), **LSF** (Load Sharing Facility) и **Grid Engine**. Этой возможностью мы воспользуемся во второй части этой главы, где построим оперативные кластеры и поручим им выполнение заданий.

Введение в задачи jug

Задачи – это простейший структурный элемент в jug. Задача состоит из функции и значений ее аргументов. Рассмотрим простой пример:

```
def double(x):
  return 2*x
```

В этой главе примеры кода следует размещать в файлах скриптов, поэтому маркера >>> не будет. Командам оболочки предшествует знак $.

Задачей мог бы быть «вызов double с аргументом 3». Другая задача – «вызов double с аргументом 642.34». С помощью jug эти задачи создаются следующим образом:

```
from jug import Task
t1 = Task(double, 3)
t2 = Task(double, 642.34)
```

Сохраните этот код в файле `jugfile.py` (обычный Python-файл). После этого для запуска задач нужно выполнить команду `jug execute`, которая вводится в оболочке операционной системы, а не Python, поэтому ей предшествует знак $:

```
$ jug execute
```

Вы увидите сообщения о том, что происходит (jug скажет, что запущены две задачи с именем `double`). Если еще раз выполнить `jug execute`, то jug сообщит, что не сделал ничего! Да ему и не нужно ничего делать. В данном случае мы почти ничего не выиграли, но если бы вычисления занимали много времени, то такое поведение оказалось бы весьма кстати.

Возможно, вы заметили, что на диске появился новый каталог `jugfile.jugdata`, в котором находятся файлы со странными именами. Это кэш запоминания. Если его удалить, что `jug execute` будет запускать задачи заново.

Часто полезно отличать чистые функции, которые только принимают входные данные и возвращают результат, от общих функций, которые могут выполнять произвольные действия (читать файлы, записывать в файлы, обращаться к глобальным переменным, модифицировать свои аргументы и вообще делать всё, что позволяет язык). В некоторых языках программирования, например в Haskell, даже есть синтаксические средства, чтобы отличать чистые функции от нечистых.

В jug задачи необязательно должны быть чистыми в строгом смысле слова. Даже рекомендуется создавать задачи, которые читают данные или записывают результаты. Однако доступ к глобальным переменным и их модификация не приветствуются: задачи могут исполняться на разных процессорах в разном порядке. Исключение составляют глобальные константы, но даже они способны сбить с толку систему запоминания (если значение константы меняется от запуска к запуску). По тем же причинам не следует модифицировать входные данные. В jug есть отладочный режим (`jug execute --debug`), в котором вычисления производятся медленнее, но зато выдаются полезные сообщения о такого рода ошибках.

Показанный выше код работает, но он несколько громоздкий. Приходится каждый раз повторять конструкцию `Task(function, argument)`. С помощью магии Python можно записать тот же код более естественно:

```
from jug import TaskGenerator
from time import sleep

@TaskGenerator
def double(x):
  sleep(4)
  return 2*x

@TaskGenerator
def add(a, b):
  return a + b

@TaskGenerator
def print_final_result(oname, value):
  with open(oname, 'w') as output:
     output.write('Final result: {}\n'.format(value))


y = double(2)
z = double(y)

y2 = double(7)
z2 = double(y2)
print_final_result('output.txt', add(z,z2))
```

Если не считать использования `TaskGenerator`, то этот код ничем не отличается от стандартного Python-файла! А благодаря декоратору `TaskGenerator` он создает ряд задач, которые теперь можно запустить таким образом, что будут задействованы все имеющиеся процессоры. За кулисами этот декоратор преобразует функции так, что в момент

обращения они не выполняются, а создают объект `Task`. Кроме того, мы пользуемся тем фактом, что одни задачи можно передавать другим, создавая тем самым зависимости.

Вероятно, вы обратили внимание на вызов `sleep(4)`. Он моделирует длительное вычисление. Иначе этот пример работал бы так быстро, что никакого смысла в использовании нескольких процессоров не было бы.

Сначала выполним команду `jug status`, ее результат показан на снимке экрана ниже:

```
Waiting    Ready    Finished    Running  Task name
-------------------------------------------------------------------
      1        0           0          0  jugfile.print_final_result
      1        0           0          0  jugfile.add
      2        2           0          0  jugfile.double
...................................................................
      4        2           0          0  Total
```

Затем запустим два процесса одновременно (знак `&` означает выполнение в фоновом режиме):

```
$ jug execute &
$ jug execute &
```

Снова выполним `jug status`:

```
Waiting    Ready    Finished    Running  Task name
-------------------------------------------------------------------
      1        0           0          0  jugfile.print_final_result
      2        0           0          2  jugfile.double
      1        0           0          0  jugfile.add
...................................................................
      4        2           0          0  Total
```

Как видим, обе функции `double` работают параллельно. Спустя 8 секунд весь процесс завершается и создается файл `output.txt`.

Кстати, если назвать файл не `jugfile.py`, а как-то иначе, то его имя придется явно указать в командной строке. Например, если бы файл назывался `analysis.py`, то надо было бы выполнить такую команду:

```
$ jug execute analysis.py
```

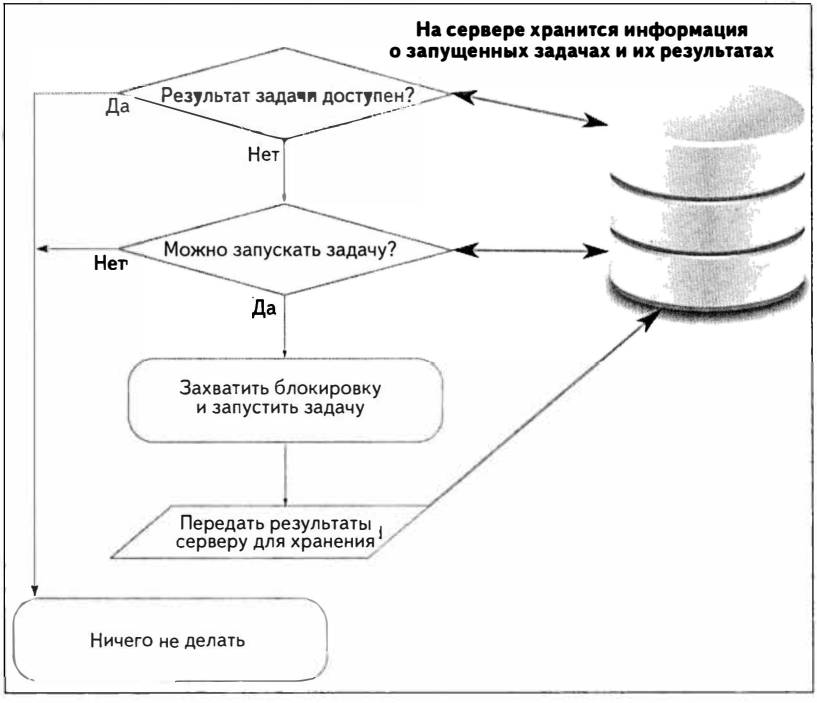
Это единственное неудобство, связанное с выбором имени, отличного от `jugfile.py`, так что можете называть свои файлы более осмысленными именами.

Заглянем под капот

Как работает jug? С принципиальной точки зрения, очень просто. Объект `Task` состоит из функции и ее аргументов. Аргументами мо-

гут быть как значения, так и другие задачи. Если задача принимает другую задачу в качестве аргумента, то между ними образуется зависимость (и вторая задача не может начаться, пока не станут доступны результаты первой).

Исходя из этого, jug рекурсивно вычисляет свертку для каждой задачи. В значении свертки закодировано все вычисление, необходимое для получения результата. При выполнении команды `jug execute` для каждой задачи запускается цикл, в котором исполняется логика, показанная на следующей блок-схеме.



Подразумеваемый по умолчанию сервер записывает данные на диск (в каталог `jugfile.jugdata/`). Существует также сервер, сохраняющие результаты в базе данных Redis. При надлежащем управлении блокировками, о котором позаботится jug, это позволяет выполнять задачи на нескольких процессорах; каждый процесс независимо просматривает список задач, выбирает еще не запущенные, а затем передает их результаты общему серверу. Этот механизм может работать как на одной машине (с несколькими процессорами), так и на не-

скольких при условии, что все они имеют доступ к общему серверу (например, через сетевой диск или базу данных Redis). Во второй части этой главы мы обсудим кластеры компьютеров, а пока сосредоточимся на одном компьютере с несколькими ядрами.

Понятен также механизм запоминания промежуточных результатов. Если сервер уже имеет результат выполнения задачи, то он не запускает ее повторно. С другой стороны, если задача хоть как-то изменилась (пусть даже отличается только значением одного параметра), то изменится ее свертка, поэтому она будет перезапущена. Кроме того, изменятся свертки всех зависящих от нее задач, поэтому они также будут перезапущены.

Применение jug для анализа данных

Jug – каркас общего назначения, но он идеально подходит для анализа данных среднего объема. При разработке конвейера анализа желательно, чтобы промежуточные данные автоматически сохранялись. Если этап предварительной обработки однажды уже был выполнен и с тех пор изменился только способ вычисления признаков, то не хотелось бы повторять все сначала. Если признаки уже вычислены, но требуется добавить новые, то зачем заново вычислять старые признаки?

Кроме того, jug оптимизирован для работы с массивами NumPy. Если задачи принимает или возвращает массив NumPy, то этой оптимизацией можно воспользоваться. Jug – еще одна часть экосистемы совместно работающих компонентов.

Вернемся к главе 10 «Машинное зрение». Там мы обсуждали вычисление признаков для изображений. Напомним, что конвейер состоял из следующих шагов:

- загрузка файлов изображений;
- вычисление признаков;
- комбинирование признаков;
- нормировка признаков;
- создание классификатора.

Повторим это упражнение, но на этот раз воспользуемся jug. Эта версия лучше тем, что позволяет добавлять новый признак или классификатор, не вычисляя заново весь конвейер.

Для начала импортируем несколько модулей:

```
from jug import TaskGenerator
import mahotas as mh
from glob import glob
```

Теперь определим генераторы задач и функции вычисления признаков:

```
@TaskGenerator
def compute_texture(im):
  from features import texture
  imc = mh.imread(im)
  return texture(mh.colors.rgb2gray(imc))

@TaskGenerator
def chist_file(fname):
  from features import chist
  im = mh.imread(fname)
  return chist(im)
```

Импортируемый модуль `features` разработан в главе 10.

Наши функции принимают имя файла, а не массив изображений. Конечно, передавать изображения тоже можно, но это небольшая оптимизация. Имя файла – короткая строка, поэтому ее проще передать серверу, да и свертка вычисляется очень быстро. Кроме того, изображения загружаются только тогда, когда это необходимо процессу.

Декоратор `TaskGenerator` можно применить к любой функции, даже к той, что написана не нами, например `np.array`, `np.hstack` или к следующей команде:

```
import numpy as np
to_array = TaskGenerator(np.array)
hstack = TaskGenerator(np.hstack)

haralicks = []
chists = []
labels = []

# Измените эту переменную, так чтобы она указывала на
# набор данных на диске
basedir = '../SimpleImageDataset/'
# Используем glob для получения всех изображений
images = glob('{}/*.jpg'.format(basedir))

for fname in sorted(images):
  haralicks.append(compute_texture(fname))
  chists.append(chist_file(fname))
  # Класс закодирован в имени файла, например xxxx00.jpg
  labels.append(fname[:-len('00.jpg')])

haralicks = to_array(haralicks)
```

```
chists = to_array(chists)
labels = to_array(labels)
```

Мелкое неудобство jug состоит в том, что мы всегда должны писать функции для вывода результатов в файл, как в примерах выше. Но это небольшая плата за те преимущества, которые дает jug.

```
@TaskGenerator
def accuracy(features, labels):
  from sklearn.linear_model import LogisticRegression
  from sklearn.pipeline import Pipeline
  from sklearn.preprocessing import StandardScaler
  from sklearn import cross_validation

  clf = Pipeline([('preproc', StandardScaler()),
                  ('classifier', LogisticRegression())])
  cv = cross_validation.LeaveOneOut(len(features))
  scores = cross_validation.cross_val_score(
     clf, features, labels, cv=cv)
  return scores.mean()
```

Отметим, что `sklearn` импортируется только внутри этой функции. Это небольшая оптимизация, позволяющая импортировать модули только там и тогда, где и когда это действительно необходимо.

```
scores_base = accuracy(haralicks, labels)
scores_chist = accuracy(chists, labels)

combined = hstack([chists, haralicks])
scores_combined = accuracy(combined, labels)
```

Осталось лишь написать и вызвать функцию для вывода результатов. Она ожидает получить список пар, содержащих название алгоритма и результаты:

```
@TaskGenerator
def print_results(scores):
  with open('results.image.txt', 'w') as output:
     for k,v in scores:
        output.write('Accuracy [{}]: {:.1%}\n'.format(
           k, v.mean()))

print_results([
 ('base', scores_base),
 ('chists', scores_chist),
('combined' , scores_combined),
])
```

Вот и всё. Теперь выполните в оболочке следующую команду, которая запустит конвейер jug:

```
$ jug execute image-classification.py
```

Повторное использование частичных результатов

Допустим, мы хотим добавить новый признак (или даже целый набор признаков). В главе 10 «Машинное зрение» мы видели, что это легко сделать, изменив код вычисления признаков. Но тогда нам пришлось бы заново вычислять все вообще признаки, а это расточительно, особенно если требуется быстро протестировать новые признаки и методы.

Добавим набор признаков, а точнее еще один вид текстурных признаков – линейные бинарные шаблоны. Они уже реализованы в mahotas, так что нам нужно только вызвать функцию, предварительно обернув ее декоратором `TaskGenerator`:

```
@TaskGenerator
def compute_lbp(fname):
  from mahotas.features import lbp
  imc = mh.imread(fname)
  im = mh.colors.rgb2grey(imc)
  # Параметрам 'radius' и 'points' присваиваются типичные значения.
  # Что они означают, смотрите в документации.
  return lbp(im, radius=8, points=6)
```

Заменим написанный ранее цикл дополнительным вызовом функции:

```
lbps = []
for fname in sorted(images):
  # все остальное, как и раньше
  lbps.append(compute_lbp(fname))
lbps = to_array(lbps)
```

Вызовем функцию `accuracy`, передав ей новые признаки:

```
scores_lbps = accuracy(lbps, labels)
combined_all = hstack([chists, haralicks, lbps])
scores_combined_all = accuracy(combined_all, labels)
print_results([
  ('base', scores_base),
  ('chists', scores_chist),
  ('lbps', scores_lbps),
  ('combined' , scores_combined),
  ('combined_all' , scores_combined_all),
])
```

Если теперь снова выполнить `jug execute`, то новые признаки будут вычислены, а старые загружены из кэша. Вот в таких ситуациях эффективность jug проявляется особенно наглядно. Гарантируется,

что вы всегда получите нужные результаты, не вычисляя заново то, что уже хранится в кэше. Кстати, добавление новых признаков улучшает качество работы рассмотренных выше методов.

В этой главе невозможно рассказать обо всех возможностях jug, но все же перечислим наиболее интересные из тех, что не вошли в основной текст.

- `jug invalidate`: объявляет, что все результаты указанной функции следует считать недействительными и вычислить заново. При этом также производятся все вычисления, прямо или косвенно зависящие от более не действительных результатов.
- `jug status --cache`: если команда `jug status` занимает слишком много времени, можно задать флаг `--cache`, чтобы кэшировать состояние и ускорить работу. Отметим, что при этом не обнаруживаются изменения в файле jugfile, но всегда можно задать комбинацию флагов `--cache --clear`, чтобы удалить кэш и начать все заново.
- `jug cleanup`: удаляет лишние файлы из кэша запоминания. Это операция сборки мусора.

Существуют также более продвинутые функции, позволяющие исследовать значения, вычисленные в ходе выполнения jugfile. В документации по адресу `http://jug.rtfd.org` описаны и другие возможности, например барьеры.

Работа с Amazon Web Services

Если данных очень много и с ними необходимо произвести сложные вычисления, то может возникнуть непреодолимое желание обзавестись дополнительными вычислительными мощностями. Компания Amazon (`http://aws.amazon.com`) позволяет арендовать машинное время с почасовой оплатой. Таким образом, вы можете получить гигантские вычислительные мощности, не приобретая кучу машин (и не неся расходов по управлению инфраструктурой). На этом рынке есть и другие конкуренты, но Amazon – самый крупный игрок, поэтому коротко расскажем о нем.

Amazon Web Services (AWS) – это большой набор служб. Мы остановимся только на службе **Elastic Compute Cloud (EC2)**, позволяющей быстро выделять и освобождать виртуальные машины и дисковое пространство.

Существует три режима работы. Во-первых, режим с резервированием, когда вы платите авансом, зато дешевле, почасовой тариф и переменный тариф, который зависит от складывающейся обстановки (когда спрос меньше, цена ниже, а когда больше – выше).

На этом общем фундаменте надстроено несколько типов машин, предлагаемых по различной цене: от одноядерных до многоядерных с большим объемом оперативной памяти и даже с графическими процессорами (GPU). Ниже мы увидим, что можно арендовать несколько более дешевых машин и самостоятельно построить из них кластер. Можно также выбрать тип сервера: Linux или Windows (Linux обходится немного дешевле). В этой главе все примеры рассчитаны на Linux, но почти все, о чем пойдет речь, справедливо и для Windows.

Для тестирования можно воспользоваться одной машиной из **бесплатного яруса** (free tier). Это позволяет поэкспериментировать с системой, привыкнуть к пользовательскому интерфейсу и т. д. Отметим, однако, что такая машина оснащена медленным процессором.

Ресурсами можно управлять с помощью веб-интерфейса. Но можно делать это и программно, писать скрипты, которые выделяют виртуальные машины, размечают жесткие диски и вообще выполняют все операции, которые доступны в интерфейсе. На самом деле, веб-интерфейс изменяется очень часто (и некоторые снимки экрана, приведенные в книге, к моменту ее выхода из печати, возможно, уже устареют), тогда как программный интерфейс более стабилен, а общая архитектура не менялась с момента появления службы.

Доступ к службам AWS производится традиционно: по имени и паролю пользователя. Правда, Amazon называет имя пользователя *ключом доступа*, а пароль – *секретным ключом*. Наверное, это сделано для того, чтобы отличить их от имени и пароля для доступа к веб-интерфейсу. Можно создать сколько угодно пар (ключ доступа, секретный ключ) и назначить им разные права. Это полезно в большой команде, где старший пользователь, имеющий доступ ко всей веб-панели, может создавать ключи для разработчиков с меньшими привилегиями.

Amazon.com обслуживает несколько регионов, соответствующих географическим областям: западное побережье США, восточное побережье США, несколько регионов в Азии, один в Южной Америке и два в Европе. Если вы собираетесь перемещать данные, то лучше выбирать регион, ближайший к источнику и получателю. Кроме того, имейте в виду, что могут действовать законы, требующие хранить информацию о пользователях в определенной юрисдик-

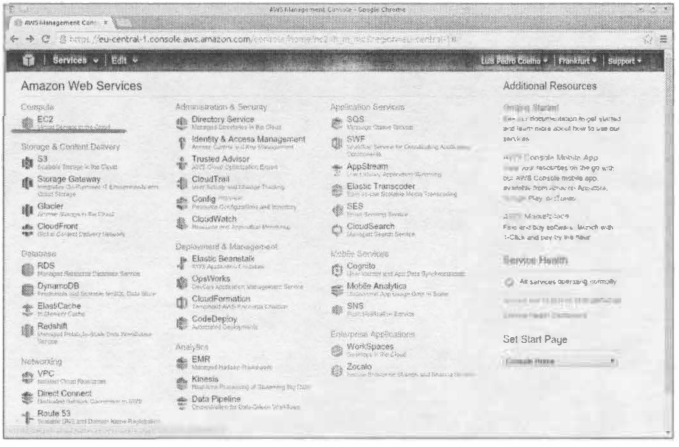
ции. В таком случае проконсультируйтесь со знающим юристом о последствиях переноса данных о европейских пользователях в США и т. п.

Amazon Web Services – очень обширная тема, ей посвящены целые книги. Цель этой главы – дать общее представление о возможностях AWS. Оставаясь верны практической направленности этой книги, мы рассмотрим ряд примеров, которые, конечно, не покрывают все многообразие функциональности.

Создание виртуальной машины

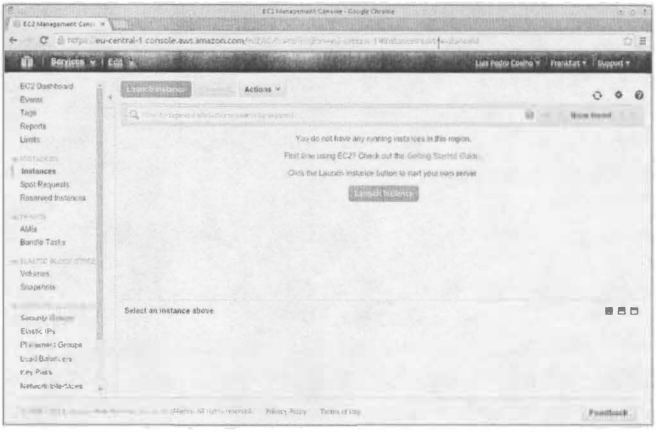
Прежде всего, нужно зайти на сайт `http://aws.amazon.com/` и создать учетную запись. Действия такие же, как для любой другой онлайновой службы. Одна машина предоставляется бесплатно, но, чтобы получить больше, понадобится кредитная карта. В примере ниже мы будем работать с несколькими машинами, так что если вы хотите проработать их вместе с нами, придется раскошелиться на несколько долларов. Если вы пока не готовы доставать из бумажника кредитную карту, то просто прочитайте эту главу, чтобы познакомиться с AWS, не выполняя примеров. Тогда у вас будет больше информации, чтобы решить, стоит ли регистрироваться.

После регистрации на AWS и входа в систему вы увидите консоль. На ней перечислены различные службы, предоставляемые AWS:



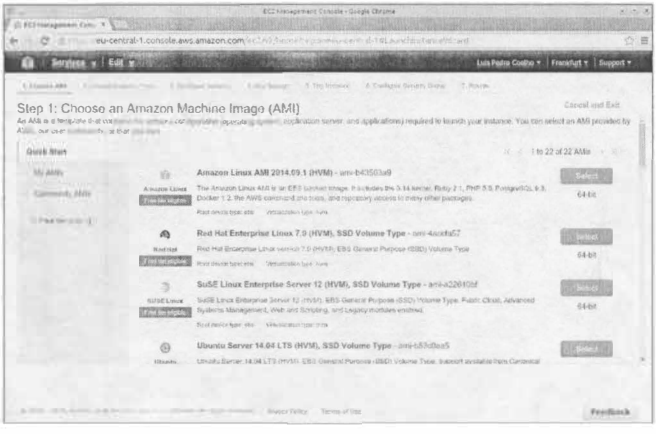
Мы выберем службу **EC2** (верхний пункт в левой колонке – так панель выглядела на момент написания этой книги. Amazon регулярно

вносит мелкие изменения, поэтому, когда вы зайдете, панель, возможно, будет выглядеть по-другому). Появится консоль управления EC2:

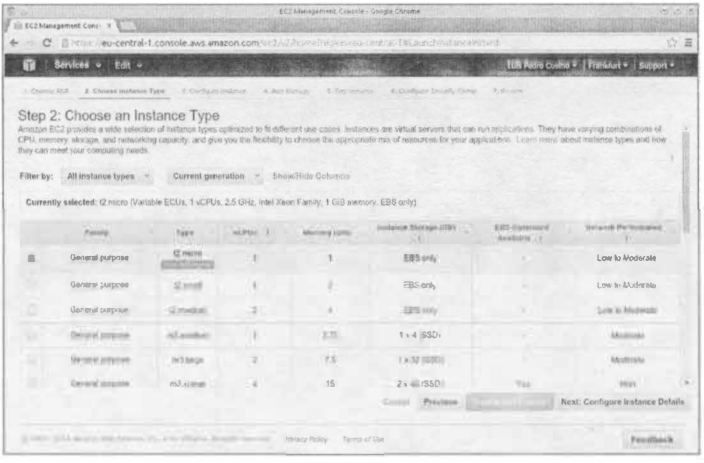


В правом верхнем углу можно выбрать регион (см. врезку о регионах Amazon). Отметим, что *показывается только информация о регионе, который выбран в данный момент*. Поэтому, если вы по ошибке выберете не тот регион (или создадите машины в разных регионах), то можете не увидеть свои машины (это типичная ловушка, в которую попадают пользователи веб-консоли управления).

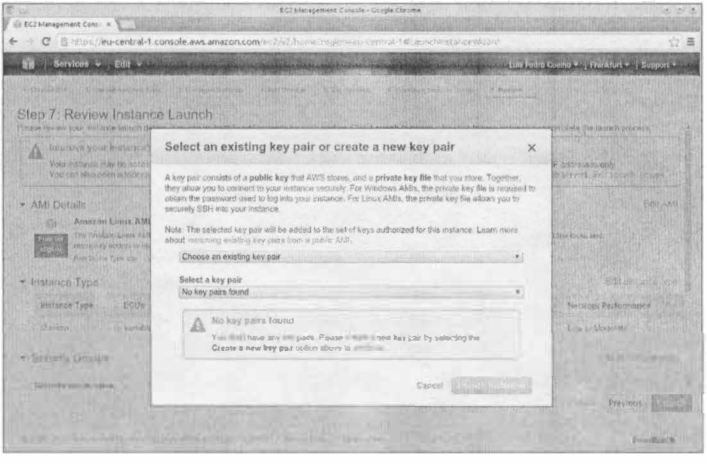
В терминологии EC2 работающий сервер называется **экземпляром**. Нажав кнопку **Launch Instance** (Запустить экземпляр), мы перейдем на следующий экран, где предлагается выбрать операционную систему:



Выберем вариант **Amazon Linux** (если вы знакомы с другими дистрибутивами, например Red Hat, SUSE или Ubuntu, то можете выбрать любой из них, но конфигурации будут немного различаться). Итак, программное обеспечение выбрано, теперь надо определиться с оборудованием. На следующем экране предлагается указать тип машины:

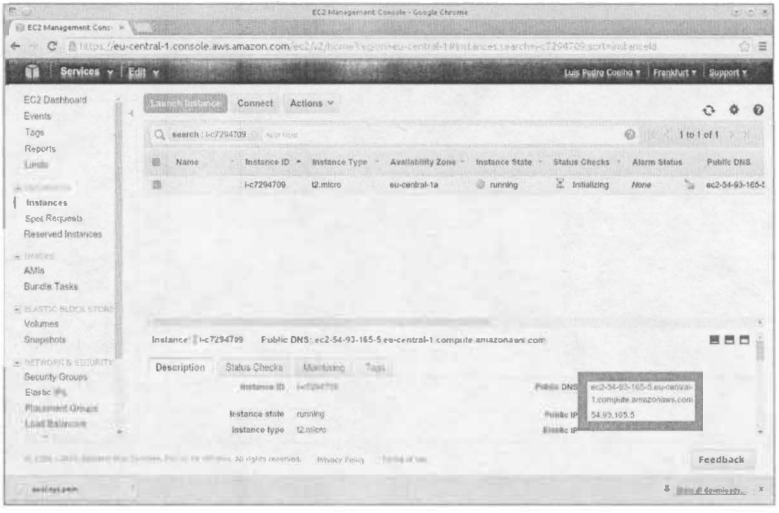


Начнем с одного экземпляра типа **t2.micro** (раньше существовал тип **t1.micro** – еще менее мощный). Это минимальная конфигурация, и предоставляется она бесплатно. Нажимайте кнопку **Next**, соглашаясь со всеми умолчаниями, пока не дойдете до экрана, где упоминается пара ключей:



Мы назовем эту пару ключей `awskeys`. Затем выберите из списка пункт **Create a new key pair** (Создать новую пару ключей). Назовите файл, содержащий ключи, `awskeys.pem`. Скачайте этот файл и сохраните его в безопасном месте. Это файл с ключами SSH (Secure Shell), который позволит заходить на облачную машину. После того как вы согласитесь с остальными умолчаниями, экземпляр запустится.

Теперь надо несколько минут подождать завершения инициализации экземпляра. В конечном итоге появится зеленый кружочек, и состояние сменится на **running** (работает):



На рисунке выше виден открытый IP-адрес (Public IP), по которому можно зайти на экземпляр, выполнив следующую команду:

```
$ ssh -i awskeys.pem ec2-user@54.93.165.5
```

Таким образом, команде `ssh` с помощью флага `-i` передается файл ключей, который мы ранее скачали и который играет роль удостоверения. Мы входим от имени пользователя `ec2-user` на машину с IP-адресом 54.93.165.5. Разумеется, у вас адрес будет другой. Если выбрать другой дистрибутив операционной системы, то имя пользователя может измениться. В таком случае попробуйте `root`, `ubuntu` (для дистрибутива Ubuntu) или `fedora` (для дистрибутива Fedora).

Наконец, в Unix-подобных ОС (включая Mac OS) необходимо изменить права доступа к файлу следующей командой:

```
$ chmod 600 awskeys.pem
```

Тем самым мы разрешаем чтение и запись в файл только его владельцу. Если этого не сделать, SSH выдаст угрожающее предупреждение.

Теперь можете заходить на свою машину. Если все пройдет успешно, вы увидите извещение, показанное на рисунке ниже.

```
$ ssh -i awskeys.pem ec2-user@ec2-54-93-165-5.eu-central-1.compute.amazonaws.com
Last login: Thu Nov 13 07:43:33 2014 from embln.embl.de

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2014.09-release-notes/
7 package(s) needed for security, out of 18 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-26-129 ~]$
```

Это обычная Linux-машина, где вы имеете права `sudo`, то есть можете запускать любую команду от имени суперпользователя, предварив ее командой `sudo`. Имеет смысл последовать рекомендации и выполнить команду `update`, чтобы ускорить работу компьютера.

Установка Python-пакетов на Amazon Linux

Если вы лучше знакомы с другим дистрибутивом, то можете использовать свои знания о нем для установки Python, NumPy и других пакетов. Ниже показано, как это делается в стандартном дистрибутиве Amazon. Для начала установим несколько базовых Python-пакетов:

```
$ sudo yum -y install python-devel \
python-pip numpy scipy python-matplotlib
```

Для компиляции mahotas понадобится компилятор C++:

```
$ sudo yum -y install gcc-c++
```

Наконец, установим `git`, чтобы получить последнюю версию кода из книги:

```
$ sudo yum -y install git
```

В этой системе пакет `pip` установлен под именем `pip-python`. Для удобства попросим pip обновить себя самого, а затем с его помощью установим необходимые пакеты:

```
$ sudo pip-python install -U pip
$ sudo pip install scikit-learn jug mahotas
```

Теперь вы можете с помощью pip установить любой другой пакет.

Запуск jug на облачной машине

Чтобы скачать данные и код, сопровождающие эту книгу, нужно выполнить такие команды:

```
$ git clone \
https://github.com/luispedro/BuildingMachineLearningSystemsWithPython
$ cd BuildingMachineLearningSystemsWithPython
$ cd ch12
```

И потом:

```
$ jug execute
```

Все должно отработать на ура, но результатов придется ждать долго. Наша машина из бесплатного яруса (типа `t2.micro`) не отличается высоким быстродействием и имеет всего один процессор. Поэтому *модернизируем ее*!

Перейдем на консоль EC2 и щелкнем правой кнопкой мыши по работающему экземпляру – появится контекстное меню. Сначала нужно остановить экземпляр. Для виртуальной машины это эквивалентно выключению питания. Свои машины вы можете останавливать в любое время. За остановленную машину плата не начисляется. Но вы по-прежнему используете место на диске, за что платите отдельно. Можно уничтожить экземпляр, тогда освободится и занятое вами место на диске, но будет потеряна вся хранящаяся на этой машине информация.

После остановки машины становится доступна команда **Change instance type** (Изменить тип экземпляра). Теперь мы можем выбрать более мощный экземпляр, например **c1.xlarge** с восьмью ядрами. Машина все еще остановлена, поэтому ее надо запустить (эквивалент загрузки для виртуальной машины).

AWS предлагает несколько типов экземпляров, отличающихся ценой. Поскольку условия постоянно пересматриваются по мере появления более мощных конфигураций и изменения цен (как правило, в сторону понижения), мы не можем здесь приводить детали, но актуальную информацию всегда можно найти на сайте Amazon.

Нужно подождать, пока экземпляр поднимется, а затем узнать его IP-адрес, как и раньше. При смене типа экземпляр получает новый IP-адрес.

Подсистема Elastic IP (слева на консоли EC2) позволяет назначить экземпляру постоянный IP-адрес. Это полезно, если вы часто создаете и модифицируете экземпляры. Услуга платная, хотя и недорогая.

Имея восемь ядер, мы можем одновременно запустить восемь процессов jug, как в следующем коде:

```
$ # в следующем цикле 8 итераций
$ for counter in $(seq 8); do
>   jug execute &
> done
```

С помощью команды `jug status` проверьте, что действительно запущено восемь задач. После того как все они завершатся (это произойдет очень скоро), можете остановить машину и понизить тип до **t2.micro**, чтобы не платить деньги. Экземпляром типа micro можно пользоваться бесплатно (в определенных пределах), а экземпляр **c1.xlarge** стоит 0,064 доллара в час (по расценкам на февраль 2015, актуальную информацию смотрите на сайте AWS).

Автоматизированная генерация кластеров с помощью StarCluster

Мы уже знаем, как создавать машины с помощью веб-интерфейса, но это утомительно и чревато ошибками. По счастью, Amazon предоставляет API. Это означает, что мы можем писать скрипты, которые будут автоматически выполнять все описанные выше операции. Более того, другие люди уже разработали инструменты для автоматизации многих процессов, обычно выполняемых в AWS.

В частности, группа разработчиков из MIT написала инструмент StarCluster. Поскольку это Python-пакет, его можно установить, воспользовавшись средствами Python:

```
$ sudo pip install starcluster
```

Эту команду можно запустить на виртуальной машине Amazon или на своей локальной машине.

Нужно указать, как должен быть устроен кластер. Для этого отредактируем конфигурационный файл. Чтобы узнать, как генерируется шаблонный конфигурационный файл, выполните такую команду:

```
$ starcluster help
```

Затем выберите режим генерации конфигурационного файла `~/.starcluster/config`. Сгенерированный файл нужно будет отредактировать вручную.

Ключи, ключи и еще раз ключи

При работе с AWS используются ключи трех разных видов. Во-первых, есть стандартная пара имя пользователя/пароль для входа на сайт. Во-вторых, открытый и закрытый ключ для работы с системой SSH, они хранятся в файле и предназначены для захода на удаленную машину. В-третьих, ключ доступа и секретный ключ AWS, представляющие собой разновидность имени и пароля пользователя для входа разных пользователей в одну и ту же учетную запись (каждому пользователю можно назначать отдельные права, но эти детали мы рассматривать не будем).

Чтобы узнать свой ключ доступа, зайдите на консоль AWS, щелкните по своему имени в правом верхнем углу и выберите команду Security Credentials. В нижней части экрана должен появиться ключ доступа вида AAKIIT7HHF6IUSN3OCAA. Мы воспользуемся им в примере ниже.

Теперь отредактируем конфигурационный файл. Это обычный ini-файл: текстовый файл с разделами, начинающимися названием в квадратных скобках, за которым следуют параметры в формате `имя=значение`. Первый раздел называется `aws info`, в него нужно скопировать ключи:

```
[aws info]
AWS_ACCESS_KEY_ID = AAKIIT7HHF6IUSN3OCAA
AWS_SECRET_ACCESS_KEY = <ваш секретный ключ>
```

Далее содержательная часть работы – определение кластера. StarCluster позволяет определить сколько угодно кластеров. Первоначально в конфигурационном файле определен единственный кластер с именем smallcluster – в разделе `cluster smallcluster`. Измените его следующим образом:

```
[cluster smallcluster]
KEYNAME = mykey
CLUSTER_SIZE = 16
```

Мы задали 16 узлов вместо двух по умолчанию. Можно также указать тип каждого и начальный образ узла кластера (напомним, что образ используется для инициализации виртуального жесткого диска и определяет операционную систему и установленное программное

обеспечение). В StarCluster включено несколько готовых образов, но можно создать и свой собственный.

Мы должны создать новый ключ SSH такой командой:

```
$ starcluster createkey mykey -o ~/.ssh/mykey.rsa
```

Теперь, сконфигурировав кластер с 16 узлами и подготовив ключи, попробуем запустить его:

```
$ starcluster start smallcluster
```

На выделение семнадцати машин уйдет несколько минут. Почему семнадцати, если в кластере только 16 узлов? Потому что StarCluster всегда создает главный узел. Все узлы разделяют общую файловую систему, поэтому любые файлы, созданные на главном узле, видны всем рабочим узлам. Это означает также, что мы можем использовать jug в таких кластерах.

Вы можете использовать кластеры по собственному усмотрению, но они уже оснащены механизмом очереди работ, идеальным для пакетной обработки. Процедура проста:

1. Зайти на главный узел.
2. Подготовить скрипты на главном узле (лучше бы сделать это заранее).
3. Поместить работы в очередь. Работой может быть любая команда Unix. Планировщик найдет свободные узлы и запустит ваши работы.
4. Дождаться завершения работ.
5. Прочитать результаты на главном узле. Для экономии денег можно также уничтожить рабочие узлы. В любом случае не оставляйте систему в работающем состоянии, если не пользуетесь ей! В противном случае вам придется за нее платить.

Перед тем как зайти на кластер, скопируем на него наши данные (напомним, что раньше мы клонировали репозиторий в каталог `BuildingMachineLearningSystemsWithPython`):

```
$ dir=BuildingMachineLearningSystemsWithPython
$ starcluster put smallcluster $dir $dir
```

Мы завели переменную `$dir`, чтобы команда поместилась на одной строке. Для захода на главный узел нужно выполнить такую команду:

```
$ starcluster sshmaster smallcluster
```

Можно было бы вместо этого узнать IP-адрес сгенерированной машины и воспользоваться командой `ssh`, как раньше, но для пока-

занной выше команды IP-адрес неважен, поскольку StarCluster сам позаботится о таких деталях.

Как уже было сказано, StarCluster предоставляет для кластеров очередь пакетных работ; вы пишете скрипт, выполняющий какие-то действия, ставите его в очередь, и в свое время он будет запущен на любом доступном узле.

Сейчас нам снова необходимо установить дополнительные пакеты. К счастью, StarCluster половину работы уже сделал. Если бы это был реальный проект, то мы написали бы скрипт, выполняющий всю инициализацию. StarCluster умеет это делать. Но поскольку это всего лишь пособие, просто выполним еще раз команду установки:

```
$ pip install jug mahotas scikit-learn
```

Можно использовать тот же jugfile, что и раньше, только запускать его нужно не на самом главном узле, а в кластере.

Сначала напишем простой обертывающий скрипт:

```
#!/usr/bin/env bash
jug execute jugfile.py
```

Назовите его `run-jugfile.sh` и с помощью команды `chmod +x run-jugfile.sh` разрешите его выполнять. Теперь мы можем запланировать 16 работ на кластере следующей командой:

```
$ for c in $(seq 16); do
>     qsub -cwd run-jugfile.sh
> done
```

Она создает 16 работ, каждая из которых будет выполнять скрипт `run-jugfile.sh`, откуда просто вызывается jug. Главный узел при этом свободен, на нем можно запускать любые команды. В частности, в любой момент можно выполнить команду `jug status` и узнать, в каком состоянии находятся вычисления. На самом деле, jug проектировался для работы именно в такой среде, так что чувствует себя в ней, как рыба в воде.

В конце концов вычисление завершится. После этого нужно сначала сохранить результаты, а затем уничтожить узлы. Создадим каталог `~/results` и скопируем в него результаты:

```
# mkdir ~/results
# cp results.image.txt ~/results
```

Теперь выйдем из кластера и вернемся на свою рабочую машину:

```
# exit
```

Мы снова находимся на машине AWS (обратите внимание на знак $ в следующих фрагментах кода). Скопируем результаты на этот компьютер, воспользовавшись командой `starcluster get` (это зеркальное отражение уже встречавшейся нам команды `put`):

```
$ starcluster get smallcluster results results
```

И напоследок нужно уничтожить все узлы, чтобы не тратить зря деньги.

```
$ starcluster stop smallcluster
$ starcluster terminate smallcluster
```

Отметим, что уничтожение кластера стирает файловую систему вместе со всеми результатами. Мы предварительно вручную скопировали результаты в безопасное место. Есть и другая возможность: поручить кластеру писать в файловую систему, которая не создается и не уничтожается StarCluster, а существует на обычном экземпляре. Эти инструменты в действительности обладают колоссальной гибкостью, но разбираться во всех тонкостях у нас не хватит места.

На сайте `http://star.mit.edu/cluster/` имеется отличная документация по StarCluster, в которой можно прочитать обо всех возможностях продукта. Мы познакомились лишь с малой их толикой и притом использовали только параметры по умолчанию.

Резюме

Мы видели, как работать с jug, небольшим написанным на Python каркасом, который умеет выполнять вычисления на нескольких процессорных ядрах или машинах. Каркас универсальный, но проектировался специально для удовлетворения потребностей автора (одновременно одного из соавторов этой книги) в анализе данных. Поэтому в некоторых отношениях он отлично стыкуется с общей инфраструктурой машинного обучения на Python.

Мы также познакомились с облаком Amazon – AWS. Облачные вычисления зачастую позволяют более эффективно использовать ресурсы, чем возможно в локальном вычислительном центре, в особенности если потребности постоянно изменяются. Пакет StarCluster даже позволяет создавать кластеры, которые автоматически расширяются при запуске дополнительных работ и уменьшаются, когда работы завершаются.

Вот мы и добрались до конца книги. Мы прошли длинный путь. Вы узнали, как выполнять классификацию, когда имеются размеченные данные, и кластеризацию, когда таких данных нет. Вы научились понижать размерность задачи и производить тематическое моделирование, чтобы извлечь знания из больших наборов данных. В конце книги мы рассмотрели кое-какие специализированные приложения (в частности, классификацию музыкальных произведений по жанрам и машинное зрение). Для реализации мы использовали язык Python, для которого создана и постоянно расширяется экосистема пакетов для численных расчетов, основанная на NumPy. Всюду, где возможно, мы полагались на библиотеку scikit-learn, но при необходимости не чурались и других пакетов. Поскольку во всех них используется одна и та же базовая структура данных (многомерный массив NumPy), предоставляемые ими средства можно свободно комбинировать. Все упоминаемые в этой книге пакеты поставляются в исходных кодах и могут использоваться в любых проектах.

Естественно, мы не смогли охватить все многообразие машинного обучения. В приложении приведены ссылки на другие ресурсы, где интересующийся читатель сможет расширить свои знания о машинном обучении.

Где получить дополнительные сведения о машинном обучении

Дописав книгу до конца, мы еще немного задержимся, чтобы подсказать читателям, где найти дополнительные материалы.

Машинному обучению посвящено немало замечательных ресурсов – гораздо больше, чем мы в состоянии здесь упомянуть. Приведенный ниже перечень – лишь малая и к тому же не вполне объективная выборка ресурсов, которые авторы считали наилучшими на момент написания книги.

Онлайновые курсы

Профессор Стэнфордского университета Эндрю Нг (Andrew Ng) ведет большой открытый онлайновый курс по машинному обучению на сайте Coursera (`http://www.coursera.org`). Курс бесплатный, но требует много времени.

Книги

Эта книга ориентирована на практические аспекты машинного обучения. Мы не рассказывали ни об идейной основе алгоритмов, ни о стоящей за ними теории. Если вас интересует эта сторона машинного обучения, рекомендуем книгу Christopher Bishop «Pattern Recognition and Machine Learning». Это классический учебник начального уровня. В нем освещены технические детали большинства использованных в книге алгоритмов.

Для тех, кому введения недостаточно и кто хочет изучить математическую теорию, советуем обратиться к замечательной книге Kevin P. Murphy «Machine Learning: A Probabilistic Perspective» (`www.cs.ubc.ca/~murphyk/MLbook`). Она вышла совсем недавно (в 2012 году) и со-

держит сведения о самых передовых исследованиях в области машинного обучения. Этот фундаментальный труд на 1100 страницах может служить и справочником, поскольку охватывает практически всю проблематику машинного обучения.

Вопросно-ответные сайты

MetaOptimize (`http://metaoptimize.com/qa`) – вопросно-ответный сайт по машинному обучению, на котором общаются весьма квалифицированные исследователи и практики.

Cross Validated (`http://stats.stackexchange.com`) – сайт, посвященный общим проблемам статистики, но часто в нем задают и вопросы по машинному обучению.

В начале книги мы уже говорили, что вопросы, касающиеся этой книги, можно задавать на сайте TwoToReal (`http://www.twotoreal.com`). Мы постараемся откликнуться как можно быстрее и поможем всем, чем сможем.

Блоги

Ниже приведен далеко не полный список блогов, представляющих интерес для всех, кто занимается машинным обучением.

- Теория машинного обучения: `http://hunch.net`
- Статьи появляются примерно раз в месяц и носят в основном теоретический характер. В качестве бонуса предлагаются головоломные задачки.
- Практические вопросы обработки текста и добычи данных: `http://textanddatamining.blogspot.de`
- Средняя периодичность появления статей – раз в месяц. Описываются практические и всегда неожиданные подходы.
- Блог Эдвина Чена (Edwin Chen): `http://blog.echen.me`
- Средняя периодичность появления статей – раз в месяц. Тематика прикладного характера.
- Заметки по машинному обучению: `http://www.machinedlearnings.com`
- Средняя периодичность – одна статья в месяц. Тематика прикладного характера.
- FlowingData: `http://flowingdata.com`
- Средняя периодичность – одна статья в день. Рассматриваются преимущественно вопросы математической статистики.

- Simply Statistics: `http://simplystatistics.org`
- Несколько статей в месяц, посвященных в основном статистике и большим данным.
- Статистическое моделирование, методы причинно-следственного вывода и социальные науки: `http://andrewgelman.com`
- Одна статья в день, бывает смешно, когда автор, пользуясь статистическими методами, указывает на ошибки в популярных СМИ.

Источники данных

Желающие поэкспериментировать с алгоритмами, могут найти много наборов данных в репозитории машинного обучения в Калифорнийском университете в Ирвайне (UCI). Адрес репозитория `http://archive.ics.uci.edu/ml`.

Участие в конкурсах

Отличный способ больше узнать о машинном обучении – посоревноваться с кем-то! Сайт Kaggle (`http://www.kaggle.com`) – место, где проводятся такие конкурсы, мы упоминали о нем во введении. Здесь вы найдете конкурсы с различной структурой и зачастую денежными призами.

Конкурсы по машинному обучению с учителем почти всегда устроены однотипно: участникам предоставляется доступ к размеченным обучающим данным и тестовым данным (без меток). Цель – классифицировать тестовые данные. Побеждает тот, кто продемонстрировал наилучшую верность. Призы разные – от славы до наличных денег.

Разумеется, что-то выиграть приятно, но и простое участие дает бесценный опыт. Так что не уходите и после завершения конкурса, потому что участники рассказывают о своих подходах на форуме. Как правило, победу приносит не изобретение нового алгоритма, а продуманная предварительная обработка, нормировка и сочетание различных методов.

Что не вошло в книгу

Мы не смогли охватить все пакеты машинного обучения, доступные из Python. Будучи ограничены объемом книги, мы решили сосредо-

точиться на библиотеке scikit-learn. Но есть и другие возможности, некоторые из них перечислены ниже.

- Комплект инструментов MDP (`http://mdp-toolkit.sourceforge.net`): модульный инструментарий для обработки данных.
- PyBrain (`http://pybrain.org`): написанная на Python библиотека для обучения с подкреплением, искусственного интеллекта и построения нейронных сетей.
- Machine Learning Toolkit (Milk) (`http://luispedro.org/software/milk`): пакет разработан одним из авторов этой книги, включает некоторые алгоритмы и методы, не вошедшие в scikit-learn.
- Pattern (`http://www.clips.ua.ac.be/pattern`): пакет объединяет добычу данных в веб, обработку естественных языков и машинное обучение, включает обертывающие API для Google, Twitter и википедии.

Более общим является сайт `http://mloss.org`, на котором размещен репозиторий программ машинного обучения с открытым исходным кодом. Как всегда бывает с такими репозиториями, качество варьируется в широких пределах: от отменных, хорошо сопровождаемых проектов до когда-то начатых и заброшенных. Сюда стоит заглянуть, если перед вами стоит очень специфичная проблема, которую не решает ни один из универсальных пакетов.

Резюме

Вот теперь точно конец. Надеемся, что книга вам понравилась и что вы готовы к собственным приключениям в мире машинного обучения.

Надеемся также, что вы в полной мере осознали важность скрупулезного тестирования применяемых методов. Особое внимание обращайте на правильное использование метода перекрестной проверки и не рапортуйте о результатах тестирования на обучающих данных, потому что полученная таким образом оценка излишне оптимистична.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ