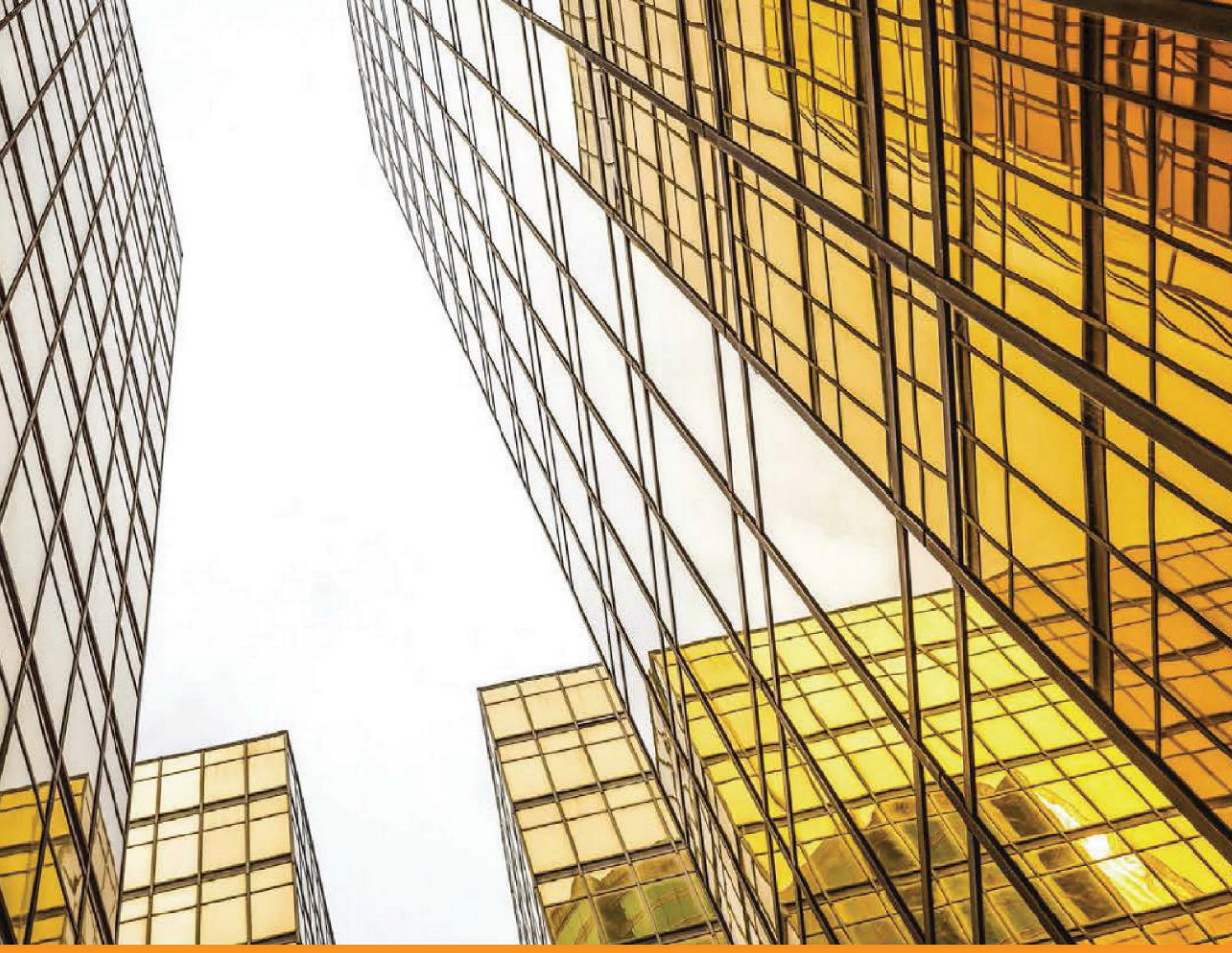


Крупномасштабное машинное обучение вместе с Python

Бастиан Шарден
Лука Массарон
Альберто Боскетти

[PACKT] PUBLISHING

ДМК издательство

Бастиан Шарден, Лука Массарон, Альберто Боскетти

Крупномасштабное машинное обучение вместе с Python

Bastiaan Sjardin, Luca Massaron, Alberto Boschetti

Large Scale Machine Learning with Python

Learn to build powerful machine learning models quickly and deploy large-scale predictive applications

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM – MUMBAI

Бастиан Шарден, Лука Массарон, Альберто Боскетти

Крупномасштабное машинное обучение вместе с Python

Учитесь быстро создавать мощные модели машинного обучения
и развертывать крупномасштабные приложения прогнозирования

Москва, 2018

УДК 004.438Python:004.6
ББК 32.973.22
С33

Шарден Б., Массарон Л., Боскетти А.

Ш25 Крупномасштабное машинное обучение вместе с Python / пер. с анг. А. В. Логунова. – М.: ДМК Пресс, 2018. – 358 с.: ил.

ISBN978-5-97060-506-6

Главная задача настоящей книги состоит в том, чтобы предоставить способы применения мощных методов машинного обучения с открытым исходным кодом в крупномасштабных проектах без привлечения дорогостоящих корпоративных решений или больших вычислительных кластеров. Описаны масштабируемое обучение в Scikit-learn, нейронные сети и глубокое обучение с использованием Theano, H2O и TensorFlow. Рассмотрены классификационные и регрессионные деревья, а также обучение без учителя. Охвачены эффективные методы машинного обучения в вычислительной среде MapReduce на платформах Hadoop и Spark на языке Python.

УДК 004.438Python:004.6
ББК 32.973.22

First published in the English language under the title 'Large Scale Machine Learning with Python – (9781785887215).

ISBN 978-1-78588-721-5 (анг.)
ISBN 978-5-97060-506-6 (рус.)

Содержание

Об авторах

Бастиан Шарден – исследователь-аналитик и создатель с опытом работы в области искусственного интеллекта и математики. Имеет степень магистра по когнитивистике, полученную им в университете Лейдена вместе с очными курсами в Массачусетском технологическом институте (MIT). В течение прошедших 5 лет он работал над широким спектром проектов в области науки о данных и искусственного интеллекта. Часто принимает участие как непостоянный член преподавательского состава в онлайн-курсах Coursera в области анализа социальных сетей Мичиганского университета и практического машинного обучения Университета Джонса Хопкинса. Его предпочтительными языками программирования являются Python и R. В настоящее время является соучредителем компании Quandbee (http://www.quandbee.com/), предлагающей масштабируемые приложения машинного обучения и искусственного интеллекта.

Лука Массарон – исследователь-аналитик и директор по анализу рынка, специализирующийся на многомерном статистическом анализе, машинном обучении и потребительском понимании, с более чем десятилетним опытом в решении реальных задач и генерировании стоимости для заинтересованных сторон путем применения логического обоснования, статистики, анализа данных и алгоритмов. Являясь первопроходцем в сфере исследования веб-аудитории в Италии, сегодня входит в лучшую десятку аналитиков по версии профессионального аналитического веб-сайта Kaggle. Всегда очень увлекался всем, что связано с данными и их анализом, а также демонстрацией потенциала, управляемого данными обнаружения знаний для экспертов и неспециалистов. Отдавая предпочтение простоте над ненужной сложностью, он полагает, что в науке о данных можно достичь многого, всего лишь руководствуясь основами.

Хотел бы благодарить Юкико и Амелию за их постоянную поддержку, помощь и любвеобильное терпение.

Альберто Боскетти – исследователь-аналитик с профессиональным опытом в области обработки сигналов и математической статистики. Имеет научную степень доктора наук по инженерному делу в области телекоммуникаций, в настоящее время живет и работает в Лондоне. В своих рабочих проектах он занимается решением сложных задач, которые включают в себя широкий круг областей – от обработки естественного языка (NLP) вплоть до машинного обучения и распределенной обработки. Очень увлечен своей работой и всегда пытается оставаться в курсе технологических новинок в области науки о данных, участвуя во встречах профессионалов, конференциях и других событиях.

О рецензентах

Олег Окунь – эксперт по машинному обучению, является автором/редактором четырех книг, многочисленных журнальных статей и докладов на конференциях. Работает в отрасли уже более четверти века, проработав в высшей школе и промышленности на своей родине в Белоруссии и за границей (в Финляндии, Швеции и Германии). Его опыт работы включает анализ цифровых образов документов, биометрию цифрового отпечатка, биоинформатику, онлайновую и внесетевую маркетинговую аналитику и аналитику рейтинга кредитоспособности. Интересуется всеми аспектами распределенного машинного обучения и Интернетом вещей. Олег в настоящее время живет и работает в Гамбурге, Германия, и собирается приступить к новой работе в качестве главного архитектора интеллектуальных систем. Его предпочтительными языками программирования являются Python, R и Scala.

Хотел бы выразить свою самую глубокую благодарность моим родителям за все, что они для меня сделали.

Кай Лонденберг – исследователь-аналитик и эксперт в области больших данных с многолетним профессиональным опытом. В настоящее время работает исследователем-аналитиком в компании Volkswagen Data Lab. До этого имел удовольствие работать ведущим исследователем-аналитиком в компании Searchmetrics, где Лука Массарон был членом его команды. Кай любит работать с передовыми технологиями, и, являясь прагматично настроенным практиком машинного обучения и действующим разработчиком программного обеспечения, он любит всегда оставаться в курсе последних технологических и научных достижений в области машинного обучения, ИИ и смежных областях. Его можно найти в социальной сети LinkedIn по адресу https://www.linkedin.com/in/kailondenberg.

Предисловие

> «Самое приятное в мозге – то, что можно узнать, что невежество вытесняется знанием и что крохотные частички знания постепенно образуют солидные пирамиды».
>
> *– Дуглас Хофштадтер*

Машинное обучение часто называют *подобластью искусственного интеллекта, которая реально работает*. Его цель состоит в том, чтобы, основываясь на существующем подмножестве данных (тренировочном наборе), с максимально возможной точностью найти функцию для предсказания исходов подмножества ранее не наблюдавшихся данных (тестового набора). Это происходит либо в форме меток и классов (задачи классификации), либо в форме непрерывного значения (задачи регрессии). Материальные примеры машинного обучения в реальных приложениях простираются от предсказания будущих курсов акций до идентификации половой принадлежности автора, исходя из серии документов. В этой книге читатель получит объяснение самых важных принципов работы машинного обучения, а также методов, подходящих для обработки крупных наборов данных, благодаря практическим примерам на языке Python. Мы рассмотрим обучение с учителем (классификация и регрессия) и обучение без учителя (в т. ч. анализ главных компонент, кластеризацию данных и тематическое моделирование), которые оказались применимыми к большим наборам данных.

Работающие в области ИТ крупные корпорации, такие как Google, Facebook и Uber, подняли нешуточный ажиотаж, утверждая, что они успешно применили подобного рода методы машинного обучения в крупном масштабе. С появлением и распространением больших данных спрос на масштабируемые решения в области машинного обучения рос экспоненциально, и множество других компаний и отдельных специалистов устремилось на поиски зрелых плодов скрытых корреляций в больших наборах данных. К сожалению, большинство обучающихся алгоритмов не очень хорошо масштабируется, перегружая центральные процессоры и память настольного компьютера или большого вычислительного кластера. В нынешние времена, несмотря на то что *большие данные* преодолели пик ажиотажа, масштабируемых решений для машинного обучения до сих пор не так уж и много.

Откровенно говоря, нам по-прежнему приходится обходить немало узких мест даже с наборами данных, которые едва ли можно классифицировать как *большие данные* (речь идет о наборах данных объемом до 2 Гб или меньше). Главная задача настоящей книги состоит в том, чтобы предоставить способы (иногда нестандартные) для применения самых мощных методов машинного обучения с открытым исходным кодом в более крупном масштабе без привлечения дорогостоящих корпоративных решений или больших вычислительных кластеров. На протяжении всей книги мы будем использовать Python и некоторые другие легкодоступные решения, которые хорошо интегрируются в масштабируемых конвейерах машинного обучения. Чтение книги станет путешествием, которое переопределит все то,

что вы знали о машинном обучении, позволив вам сделать значительный рывок вперед в анализе по-настоящему больших данных.

О чем эта книга рассказывает

Глава 1 «*Первые шаги на пути к масштабируемости*» ставит задачу масштабируемого машинного обучения с правильной перспективой и знакомит с инструментами, которые мы будем использовать в этой книге.

Глава 2 «*Масштабируемое обучение в Scikit-learn*» обсуждает тему стратегии стохастического градиентного спуска с ограничением потребления оперативной памяти; этот прием основывается на теме *внеядерного* обучения. Мы также охватим технические приемы подготовки данных, включая хэширование признаков, способные справляться с самыми разнообразными данными.

Глава 3 «*Быстрообучающиеся реализации машин SVM*» посвящена потоковым алгоритмам, которые способны обнаруживать нелинейности в форме машин опорных векторов. Мы представим альтернативы программной библиотеке Scikit-learn, в частности программы LIBLINEAR и Vowpal Wabbit, которые, несмотря на то что выполняются как команды внешней оболочки, с легкостью обертываться в сценарии Python и ими управляются.

Глава 4 «*Нейронные сети и глубокое обучение*» охватывает полезную методику применения глубоких нейронных сетей в платформе Theano и крупномасштабные приложения на основе платформы H2O. Хотя эта тема сегодня является актуальной, их успешное применение может представлять серьезную трудность, уже не говоря о предложении масштабируемых решений. Мы также обратимся к предварительной тренировке без учителя с автокодировщиками на основе библиотеки theanets.

Глава 5 «*Глубокое обучение с библиотекой TensorFlow*» охватывает интересную методику глубокого обучения нейронных сетей и онлайновые методы. Хотя платформа TensorFlow находится в стадии становления, она предоставляет изящные решения для машинного обучения. Мы также воспользуемся возможностями библиотеки Keras по работе со сверточными нейронными сетями в среде TensorFlow.

Глава 6 «*Классификационные и регрессионные деревья в крупном масштабе*» посвящена объяснению масштабируемых решений с использованием алгоритмов случайного леса, градиентного бустинга и экстремального градиентного бустинга XGboost. Алгоритм машинного обучения CART, аббревиатура для классификационных и регрессионных деревьев, обычно применяется в рамках ансамблевых методов. Мы также предоставим примеры крупномасштабного приложения в среде H2O.

Глава 7 «*Обучение без учителя в крупном масштабе*» вплотную посвящена обучению без учителя, а именно будут рассмотрены алгоритмы PCA, кластерного анализа и тематического моделирования с использованием соответствующего подхода, позволяющего масштабировать их вертикально.

Глава 8 «*Распределенные среды – Hadoop и Spark*» научит настраивать систему Spark в среде виртуальной машины с переходом от одиночной машины к парадигме вычислительной сети. Поскольку Python способен с легкостью связывать индивидуальные наработки и подключать их к кластеру машин, привлечение возможностей кластера Hadoop становится простым делом.

Глава 9 «*Практическое машинное обучение на платформе Spark*» посвящена непосредственной работе с платформой Spark и обучает необходимым основам, для того чтобы начать напрямую управлять данными и создавать прогнозные модели на больших наборах данных.

Приложение «Введение в графические процессоры и библиотеку Theano» коснется основ работы в среде Theano и вычислений на GPU. Это пособие поможет установить и подготовить свою собственную среду для использования Theano на GPU, если, конечно, располагаемая система это позволяет сделать.

Что требуется для этой книги

Выполнение примеров кода, прилагаемых к данной книге, требует установки Python 2.7 или старших версий в Mac OS, Linux или Microsoft Windows.

В примерах по всей книге будут часто использоваться важные программные библиотеки Python для научных и статистических вычислений, в частности SciPy, NumPy, Scikit-learn, StatsModels и реже matplotlib и pandas. Мы также задействуем приложение для внеядерных облачных вычислений под названием H2O.

Эта книга в значительной мере зависит от интерактивной среды программирования Jupyter и ее так называемых блокнотов, или записных книжек, приводимых в действие ядром Python. Для этой книги мы воспользуемся ее последней версией 4.1 (на момент публикации перевода – 5.0).

Первая глава предоставит все пошаговые инструкции и некоторые полезные советы по настройке среды Python, упомянутых выше рабочих программных библиотек и всего необходимого инструментария.

Для кого эта книга

Эта книга предназначена для начинающих и действующих практиков в области науки о данных, разработчиков и всех тех, кто намеревается работать с большими и сложными наборами данных. Мы постарались сделать эту книгу максимально доступной для самой широкой аудитории. И все же, учитывая, что темы в этой книге достаточно продвинутые, читателям рекомендуется знать основные понятия из области машинного обучения, в частности классификацию и регрессию, функции минимизации ошибки и перекрестную проверку, однако это условие не является обязательным.

Мы также предполагаем, что читатель обладает некоторым опытом программирования на Python, работы с блокнотами Jupyter и выполнением команд из командной строки, а также познаниями в математике на разумном уровне, чтобы усвоить концепции, лежащие в основе разных больших решений, которые мы здесь предлагаем. Текст написан в стиле, который поймут программисты на других языках (R, Java и MATLAB). В идеальном случае он очень подходит для исследователя-аналитика (но не ограничен лишь им), знакомого с машинным обучением и заинтересованного в мобилизации возможностей языка Python, в отличие от других языков, таких как R или MATLAB, из-за его широких возможностей выполнять вычислительные задачи, работать с оперативной памятью и выполнять ввод-вывод данных.

Условные обозначения

В этой книге вы найдете несколько текстовых стилей, которые выделяют различные виды информации. Вот некоторые примеры этих стилей и объяснение их значения.

Ключевые слова в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, пути файловой системы, фиктивные URL, ввод данных пользователем и дескрипторы Twitter показаны следующим образом: «Во время обследования линейной модели сначала проверьте атрибут `coef_`».

Фрагмент исходного кода оформляется следующим образом:

```
from sklearn import datasets
iris = datasets.load_iris()
```

Поскольку в большинстве примеров мы будем использовать блокноты Jupyter, фрагменты исходного кода будут оформляться в соответствии с тем, как он выглядит в ячейках блокнотов: входные данные будут всегда помечены как `In:`, а выходные данные – часто как `Out:`. На компьютере требуется только ввести исходный код после `In:` и проверить, соответствуют ли результаты содержимому `Out:`:

```
In:
clf.fit(X, y)

Out:
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
degree=3, gamma=0.0, kernel='rbf', max_iter=-1, probability=False,
random_state=None, shrinking=True, tol=0.001, verbose=False)
```

Когда команда предназначена для выполнения в командной строке терминала, такая команда будет предваряться префиксом `$>`, в противном случае, если она предназначена для интепретатора Python REPL, ей будет предшествовать приглашение `>>>`:

```
$ python
>>> import sys
>>> print sys.version_info
```

Новые термины и **важные слова** показаны полужирным шрифтом. Слова, которые выводятся на экран, например в меню или диалоговых окнах, выглядят в тексте следующим образом: «Как правило, необходимо просто ввести в ячейки после **In:** исходный код и его выполнить».


Предупреждения или важные примечания появляются в этом поле.


Подсказки и приемы появляются тут.


Дополнения к тексту оригинала книги.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или вебсайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Комментарий переводчика

Весь материал книги приведен в соответствие с последними действующими версиями программных библиотек (время перевода книги – май-июнь 2017 г.) и протестирован в среде Windows 10. При тестировании исходного кода за основу взят Python версии 2.7.13.

Большинство содержащихся в книге технических терминов и аббревиатур для удобства кратко определено в сносках, а для некоторых терминов в силу отсутствия единой терминологии приведены соответствующие варианты наименований или пояснения.

Прилагаемый к книге адаптированный и скорректированный исходный код примеров лучше всего разместить в подпапке домашней папки пользователя (`/home/Ваши_проекты_Python` или `C:\Users\[ИМЯ_ПОЛЬЗОВАТЕЛЯ]\Ваши_проекты_Python`). Ниже приведена структура папки с прилагаемыми примерами:

`Chapter 01-09`	Исходный код примеров в виде записных книжек Jupyter
`py_scripts`	Исходный код примеров в виде сценариев Python
`vowpal_wabbit_for_windows`	Исполнимые файлы программного продукта Vowpal Wabbit для 32- и 64-разрядных ОС Windows

Для просмотра исходного кода примеров лучше всего пользоваться блокнотами Jupyter. Они более читабельны, содержат графики, цветные рисунки и расширенные пояснения.

Далее приведены особенности инсталляции некоторых используемых программных библиотек Python.

Особенности программного обеспечения

В обычных условиях библиотеки Python можно скачать из каталога библиотек Python `PyPi` (https://pypi.python.org/). Однако следует учесть, что для работы библиотек SciPy и Scikit-learn в Windows требуется, чтобы в системе была установлена библиотека Numpy+MKL. Библиотека **Numpy+MKL** привязана к библиотеке Intel® Math Kernel Library и включает в свой состав необходимые динамические библиотеки (DLL) в каталоге numpy.core. Ее следует скачать с репозитория whl-файлов (http://www.lfd.uci.edu/~gohlke/pythonlibs/) и установить (например, `pip install numpy-1.13.0+mkl-cp27-cp27m-win_amd64.whl` для 64-разрядной операционной системы Windows и среды Python 2.7) как whl (соответствующая процедура установки описана ниже).

Далее приведены сведения о других основных библиотеках:

- **NumPy** – основополагающая библиотека, необходимая для научных вычислений на Python;
- **Matplotlib** – библиотека для работы с двумерными графиками. Требует наличия numpy и некоторых других;
- **Pandas** – инструмент для анализа структурных данных и временных рядов. Требует наличия numpy и некоторых других;
- **Scikit-learn** – интегратор классических алгоритмов машинного обучения. Требует наличия numpy+mkl;
- **SciPy** – библиотека, используемая в математике, естественных науках и инженерном деле. Требует наличия numpy+mkl;
- **Jupyter** – интерактивная вычислительная среда.

Факультативно:

- **PyQt5** – библиотека инструментов для программирования визуального интерфейса, требуется для работы инструментальной среды программирования Spyder;
- **Spyder** – инструментальная среда программирования на Python.

Протокол установки программного обеспечения

```
python -m pip install --upgrade pip
pip install numpy
```

либо как whl:

```
pip install numpy-1.13.0+mkl-cp27-cp27m-win_amd64.whl
pip install scipy
```

либо как whl:

```
pip install scipy-0.19.0-cp27-cp27m-win_amd64.whl
pip install Scikit-learn
```

либо как whl:

```
pip3 install scikit_learn-0.18.1-cp27-cp27m-win_amd64.whl
pip install matplotlib
pip install pandas
pip install gensim
pip install jupyter
pip install neurolab
pip install theano
pip3 install tensorflow
```

Следует отметить, что библиотека TensorFlow НЕ работает с Python 2. Для работы с библиотекой TensorFlow необходимо инсталлировать Python 3. Блокнот Jupyter (гл. 5) с примерами применения библиотеки TensorFlow использует ядро Python 3.

```
pip install scikit-neuralnetwork
pip install h2o
pip install keras
```

Факультативно:

```
pip install pyqt5
pip install spyder
```

Примечание: в зависимости от базовой ОС, версий языка Python и версий программных библиотек устанавливаемые вами версии whl-файлов могут отличаться от приведенных выше, где показаны последние на июнь 2017 г. версии для 64-разрядной ОС Windows и Python версии 2.7.

Работа со связкой Hadoop/Vagrant/Jupyter

Начало работы:

```
vagrant up (в папке с файлом Vagrantfile)
set PATH=%PATH%;C:\Program Files\Git\usr\bin (в командной строке cmd)
vagrant ssh
vagrant@sparkbox:~$ ./start_hadoop.sh (в консоли vagrant)
vagrant@sparkbox:~$ ./start_jupyter_yarn.sh
http://localhost:8888 (в браузере)
```

Запуск автономного режима без поддержки менеджера YARN:

```
vagrant up (в папке с файлом Vagrantfile)
set PATH=%PATH%;C:\Program Files\Git\usr\bin (в командной строке cmd)
vagrant ssh
vagrant@sparkbox:~$ ./start_jupyter_yarn.sh
http://localhost:8888 (в браузере)
```

Конец работы с Hadoop/Vagrant/Jupyter:

```
Ctrl + C (в командной строке блокнота)
vagrant@sparkbox:~$ ./stop_hadoop.sh (в консоли vagrant)
vagrant@sparkbox:~$ exit
vagrant halt (в командной строке)
```

Добавления в системную переменную Path в ОС Windows

После установки соответствующего программного обеспечения следует проверить наличие следующих значений в системной переменной Path:

```
%HADOOP_HOME%\bin
C:\HashiCorp\Vagrant\bin
C:\Program Files\Git\cmd
C:\Program Files\Git\usr\bin
C:\Program Files\mingw-w64\x86_64-7.1.0-win32-seh-rt_v5-rev0\mingw64\bin
```

Установка библиотек Python из whl-файла

Библиотеки для Python можно разрабатывать не только на чистом Python. Довольно часто библиотеки пишутся на C (динамические библиотеки), и для них пишется обертка Python, или же библиотека пишется на Python, но для оптимизации узких мест часть кода пишется на C. Такие библиотеки получаются очень быстрыми, однако библиотеки с вкраплениями кода на C программисту на Python тяжелее установить ввиду банального отсутствия соответствующих знаний либо необходимых компонентов и настроек в рабочей среде (в особенности в Windows). Для решения описанных проблем разработан специальный формат (файлы с расширением .whl) для распространения библиотек, который содержит заранее скомпилированную версию библиотеки со всеми ее зависимостями. Формат whl поддерживается всеми основными платформами (Mac OS X, Linux, Windows).

Установка производится с помощью менеджера библиотек pip. В отличие от обычной установки командой `pip install <имя_библиотеки>`, вместо имени библиотеки указывается путь к whl-файлу `pip install <путь/к/whl_файлу>`. Например:

```
pip install C:\temp\scipy-0.19.0-cp27-cp27m-win_amd64.whl
```

Откройте окно командой строки и при помощи команды `cd` перейдите в каталог, где размещен ваш whl-файл. Просто скопируйте туда имя вашего whl-файла. В этом случае полный путь указывать не понадобится. Например:

```
pip install scipy-0.19.0-cp27-cp27m-win_amd64.whl
```

При выборе библиотеки важно, чтобы разрядность устанавливаемой библиотеки и разрядность интерпретатора совпадали. Пользователи Windows могут брать whl-файлы на веб-странице http://www.lfd.uci.edu/~gohlke/pythonlibs/ Кристофа Голька из Лаборатории динамики флуоресценции Калифорнийского университета в г. Ирвайн. Библиотеки там постоянно обновляются, и в архиве содержатся все, какие только могут понадобиться.

Установка и настройка инструментальной среды Spyder

Spyder – это инструментальная среда для научных вычислений для языка Python (Scientific PYthon Development EnviRonment) для Windows, Mac OS X и Linux. Это

простая, легковесная и бесплатная интерактивная среда разработки на Python, которая предлагает функционал, аналогичный среде разработки на MATLAB, включая готовые к использованию виджеты `PyQt5` и `PySide`: редактор исходного кода, редактор массивов данных `NumPy`, редактор словарей, консоли Python и IPython и многое другое.

Чтобы установить среду Spyder в Ubuntu Linux, используя официальный менеджер библиотек, нужна всего одна команда:

```
sudo apt-get install spyder
```

Чтобы установить с использованием менеджера библиотек pip:

```
sudo apt-get install python-qt5 python-sphinx
sudo pip install spyder
```

И чтобы обновить:

```
sudo pip install -U spyder
```

Установка среды Spyder в Fedora 25:

```
dnf install python-spyder
```

Установка среды Spyder в Windows:

```
pip install spyder
```

Примечание: среда Spyder требует обязательной установки библиотеки PyQt5.

Глава 1

Первые шаги к масштабируемости

Добро пожаловать в книгу по масштабируемому машинному обучению на Python.

В этой главе мы обсудим способы эффективного обучения на больших данных в среде Python и как это можно осуществить, используя всего одну машину или кластер из других машин, который, к примеру, можно получить в веб-службах облачных вычислений **Amazon Web Services** (AWS) или в веб-службах платформы Google-облако.

В настоящей книге мы будем использовать реализацию масштабируемых алгоритмов машинного обучения на языке Python. Иными словами, они смогут работать с большим объемом данных и не дадут сбоя из-за нехватки оперативной памяти. Кроме того, работа таких алгоритмов будет занимать разумное количество времени, достаточно приемлемое для прототипа в области науки о данных и для развертывания проекта в эксплуатационной среде. Главы книги организованы вокруг решений (таких как потоковая передача данных), алгоритмов (таких как нейронные сети или ансамбль деревьев) и платформ (Hadoop или Spark). Мы также предложим небольшой справочник по алгоритмам машинного обучения и объясним, как сделать их масштабируемыми и пригодными для решения задач с крупными наборами данных.

С учетом таких стартовых предпосылок вам потребуется изучить основы (чтобы уяснить перспективу, с которой эта книга была написана), а также установить и настроить все основные инструменты, которые позволят незамедлительно приступить к чтению глав.

В этой главе мы представим следующие темы:

- что в действительности означает термин «масштабируемость»;
- на какие узкие места необходимо обратить внимание во время работы с данными;
- какие задачи эта книга поможет решать;
- как использовать Python для эффективного анализа наборов данных в крупном масштабе;
- каким образом быстро настроить свою машину для выполнения представленных в этой книге примеров.

Давайте же начнем наше совместное путешествие по масштабируемым решениям в среде Python!

Подробное объяснение термина масштабируемости

Несмотря на весь нынешний ажиотаж вокруг больших данных, большие наборы данных существовали задолго до того, как был введен сам термин. Большое количество текстовых данных, последовательностей ДНК и огромное количество данных с радиотелескопов всегда представляли проблему для ученых и исследователей-аналитиков. Поскольку большинство алгоритмов машинного обучения имеет вычислительную сложность $O(n^2)$ или даже $O(n^3)$, где n – это число тренировочных прецедентов, исследователи и аналитики прежде решали поставленные крупными наборами данных сложные задачи, привлекая более эффективные алгоритмы обработки данных. Алгоритм машинного обучения считается масштабируемым, когда после соответствующей настройки он может работать в условиях больших наборов данных. Набор данных может быть большим в силу большого количества прецедентов либо переменных, либо в силу обоих причин, а масштабируемый алгоритм может справляться с ними эффективно, поскольку его время выполнения увеличивается почти линейно в соответствии с размером задачи. Следовательно, это просто вопрос обмена в соотношении 1:1 большего количества времени (или большей вычислительной мощи) на большее количество данных. Между тем обычный алгоритм машинного обучения, когда сталкивается с большими объемами данных, не масштабируется; он попросту прекращает работать либо работает со временем выполнения, которое увеличивается нелинейно, например экспоненциально, тем самым делая обучение неосуществимым.

Внедрение дешевых систем хранения данных, больших RAM и многоядерных CPU кардинально все изменило, увеличив возможности одиночных ноутбуков по анализу больших объемов данных. Появление в недавнем прошлом еще одного игрока стало переломным моментом, переориентировав внимание с одиночных мощных машин на кластеры серийных компьютеров (более дешевых и легкодоступных). Эта серьезная перемена обусловила внедрение вычислительной сетевой парадигмы **MapReduce** и платформы с открытым исходным кодом Apache Hadoop с ее **распределенной файловой системой Hadoop** HDFS (Hadoop distributed file system) и в целом параллельных вычислений в компьютерных сетях.

Чтобы выяснить, каким образом обе эти перемены глубоко и положительно повлияли на возможности решения крупномасштабных задач, прежде всего следует выяснить, что на самом деле мешало (и по-прежнему мешает, в зависимости от того, насколько крупной является решаемая задача) выполнять анализ крупных наборов данных.

Независимо от того, какую задачу вы решаете, в конечном счете вы обнаружите, что не можете выполнить анализа своих данных в силу следующих ниже ограничений:

- вычислительная емкость оказывает влияние на время, затрачиваемое на выполнение анализа;
- емкость каналов ввода-вывода данных влияет на то, сколько данных может быть передано за единицу времени из хранилища данных в оперативную память машины;
- емкость оперативной памяти влияет на то, насколько большими будут данные, которые можно обработать за один раз.

Ваш компьютер имеет ограничения, которые будут определять, сможете ли вы обучиться на данных и сколько потребуется времени, прежде чем вы упретесь в стену. Вычислительные ограничения бывают во многих вычислительно емких расчетах, проблемы, связанные с вводом-выводом, образуют узкое место для быстрого доступа к данным, и, наконец, ограничения по памяти могут вынудить принимать лишь часть данных, тем самым ограничивая возможности матричных вычислений, к которым можно было бы обратиться, либо прецизионность или даже строгость получаемых оценок.

Каждое из этих аппаратных ограничений будет также оказывать разное влияние по степени серьезности относительно анализируемых данных:

- высокие данные, отличительная особенность которых состоит в том, что они имеют большое количество прецедентов;
- широкие данные, которые характерны тем, что имеют большое количество признаков;
- высокие и широкие данные, которые имеют одновременно большое количество прецедентов и признаков;
- разреженные данные, которые отличаются тем, что имеют большое количество нулевых записей или записей, которые можно преобразовать в нули (т. е. матрица данных может быть высокой и/или широкой, но информативной, при этом не все записи в матрице имеют информационное наполнение).

И в конце дело сводится к алгоритму, который вы собираетесь использовать, чтобы обучиться на данных. Каждый алгоритм имеет свои собственные свойства и способен преобразовывать данные, используя решение, на которое по-разному воздействует смещение или дисперсия. Следовательно, относительно задачи, которую вы до сих пор решали при помощи машинного обучения, вы рассчитывали, что определенные алгоритмы могут работать лучше других, основываясь при этом на своем опыте или эмпирической проверке. В случае с крупномасштабными задачами при выборе алгоритма необходимо добавить еще несколько совсем других соображений:

- какова вычислительная сложность алгоритма, т. е. влияет ли число строк и столбцов в данных на число вычислений линейным или нелинейным образом. Большинство решений в области машинного обучения основывается на алгоритмах квадратичной или кубической сложности, тем самым строго ограничивая их применимость к большим данным;
- сколько в модели параметров; здесь дело не просто в проблеме дисперсии оценок (переподгонке), а во времени, которое может потребоваться на их вычисление;
- можно ли параллелизировать процессы оптимизации, т. е. можно ли легко разделить вычисления между многочисленными узлами или ядрами CPU, или же приходится опираться на одиночный последовательный процесс оптимизации;
- должен ли алгоритм обучаться сразу на всех данных, или же вместо этого можно использовать одиночные примеры либо небольшие пакеты данных.

Если перекрестно оценить аппаратные ограничения и свойства данных, с одной стороны, и подобного рода алгоритмы – с другой, то можно получить множество возможных проблемных сочетаний, которые могут стать препятствием для полу-

чения результатов от проведения крупномасштабного анализа. С практической точки зрения все проблемные сочетания можно решить на основе трех подходов:

- вертикального масштабирования, т. е. улучшения производительности одиночной машины путем модификации программного обеспечения и/или оборудования (больше оперативной памяти, более быстрые CPU и дисковая память, а также использование модулей GPU);
- горизонтального масштабирования, т. е. распределения вычислений (и производительности) по многочисленным машинам с привлечением внешних ресурсов, а именно другой дисковой памяти и других модулей CPU (либо GPU);
- вертикального и горизонтального масштабирования, т. е. взятия лучшего решения из вертикальных и горизонтальных решений вместе взятых.

Приведение крупномасштабных примеров

Несколько мотивирующих примеров прояснит ситуацию и сделает ее запоминающейся.

Возьмем два простых примера:

- способность предсказывать **кликабельность** (click-through-rate, CTR), т. е. отношение числа щелчков к числу показов. В наши дни, когда интернет-реклама настолько распространилась вширь и вглубь, что отъедает значительные куски у традиционных СМИ, она помогает довольно много зарабатывать;
- способность предложить правильную информацию своим клиентам, когда они ищут предлагаемые вашим сайтом продукты и услуги, может по-настоящему улучшить ваши возможности их продавать, в случае если вы сможете угадывать, что помещать во главу результатов их поискового запроса.

В обоих случаях мы располагаем довольно большими наборами данных, которые продуцируются пользователями в результате их взаимодействия в Интернете.

В зависимости от бизнеса, который мы имеем в виду (тут можно вообразить некоторых крупных игроков), в обоих указанных случаях мы, очевидно, говорим о миллионах точек данных в день. В случае рекламы данные, разумеется, являются высокими, потому что они представляют собой непрерывный поток информации с заменой старых данных на новые, в большей мере отражающих рынки и потребителей. В случае поисковой системы данные являются широкими, дополняясь компонентом, предоставляемым результатами, которые вы предложили своим клиентам: например, если вы занимаетесь туристическим бизнесом, то у вас будет довольно много признаков об отелях, местах посещений и предлагаемых услугах.

Безусловно, масштабируемость создает трудности в обеих этих задачах:

- необходимо обучаться на данных, которые растут каждый день, и причем обучаться быстро, потому что, пока вы обучаетесь, продолжают поступать новые данные. При этом вам приходится иметь дело с данными, которые, очевидно, не смогут уместиться в оперативной памяти, потому что матрица слишком высокая или слишком большая;
- необходимо часто выполнять обновления модели машинного обучения с целью размещения новых данных. Для этого потребуется алгоритм, который может обрабатывать информацию в нужные сроки. Вычислительную сложность $O(n^2)$ или $O(n^3)$ практически невозможно обработать ввиду ко-

личества данных; потребуется такой алгоритм, который сможет работать с пониженной сложностью (такой как $O(n)$) или путем разделения данных, в результате которого n будет гораздо меньше;
- необходимо уметь предсказывать быстро, потому что предсказания должны предоставляться только новым клиентам. И опять-таки, здесь важную роль играет вычислительная сложность используемого алгоритма.

Проблема масштабируемости может быть решена одним или несколькими способами:
- вертикальное масштабирование путем снижения размерности задачи; например, в случае поисковой системы, путем эффективного отбора релевантных признаков для использования;
- вертикальное масштабирование с использованием приемлемого алгоритма; например, в случае рекламных данных, существуют соответствующие алгоритмы для эффективного обучения на потоках данных;
- горизонтальное масштабирование процесса обучения путем привлечения многочисленных машин;
- вертикальное масштабирование процесса внедрения с эффективным использованием многопроцессорной обработки и векторизации на одиночном сервере.

В настоящей книге мы покажем, какие практические задачи могут решаться каждым из этих предложенных решений или алгоритмов. В результате вы научитесь автоматически связывать отдельное ограничение по времени и исполнению (CPU, оперативная память или операции ввода-вывода) с самым подходящим решением среди тех, которые мы предлагаем.

Введение в язык Python

Поскольку наше исследование будет зависеть от языка Python – общедоступного языка программирования, выбранного в качестве базового для настоящей книги, – мы должны сделать короткую остановку, чтобы познакомиться с языком, прежде чем приступим к выяснению того, каким образом Python способен помочь легко масштабировать решение задачи с массивными данными вертикально и горизонтально.

Созданный в 1991 г. как общецелевой, интерпретируемый, объектно-ориентированный язык программирования, Python медленно, но верно завоевывал научное сообщество и в конечном итоге превратился в зрелую экосистему, состоящую из специализированных библиотек для обработки и анализа данных. Он позволяет проводить бесчисленные и быстрые эксперименты, легко разрабатывать программы, воплощающие теоретические предпосылки, и быстро развертывать научные приложения.

Как практик машинного обучения вы найдете использование Python интересным по нескольким причинам:
- он предлагает большую, зрелую систему библиотек для анализа данных и машинного обучения. Это гарантирует, что вы получите все, в чем вы, возможно, нуждаетесь в ходе анализа данных, и иногда даже больше;
- он очень универсален. Независимо от опыта или стиля программирования (объектно-ориентированный, функциональный или процедурный), программирование на Python доставит вам удовольствие;

- если вы еще с ним незнакомы, но хорошо знаете другие языки, такие как C/C++ или Java, то он очень прост в освоении и использовании. После того как вы уясните основы, лучший способ узнать больше – сразу же приступить к программированию;
- он является кросс-платформенным; ваши решения будут работать отлично и гладко в операционных системах Windows, Linux и Mac OS. Вам не придется беспокоиться о переносимости исходного кода;
- хотя он является интерпретируемым, его быстродействие, несомненно, выше, по сравнению с другими основными языками для анализа данных, такими как R и MATLAB (хотя он не сопоставим с C, Java и недавно появившимся языком Julia);
- он может работать с большими данными в оперативной памяти в силу минимального объема потребляемой оперативной памяти и превосходного управления ею. Сборщик «мусора» в оперативной памяти будет часто экономить уйму времени, когда вы будете загружать, преобразовывать, нарезать, группировать, сохранять или отбрасывать данные, используя различные циклы и повторные операции по подготовке данных к их анализу.

Если вы еще не являетесь экспертом в этом языке (на самом деле, чтобы иметь возможность использовать эту книгу максимальным образом, мы требуем лишь элементарных знаний языка), то всю информацию о языке можно почерпнуть (а также найти основные установочные файлы) непосредственно на главном сайте языка Python по адресу https://www.python.org/.

Вертикальное масштабирование средствами Python

Python – это интерпретируемый язык; он выполняет чтение вашего сценария из оперативной памяти и выполняет его динамически, тем самым получая доступ к необходимым ресурсам (файлам, объектам в памяти и т. п.). Помимо того что он – интерпретируемый, следует учитывать еще один важный аспект при использовании Python для анализа данных и машинного обучения – Python является однопоточным языком. Однопоточность означает, что любая программа на Python выполняется последовательно от начала до конца сценария и что Python не может использовать в своих интересах дополнительных вычислительных возможностей, предлагаемых многочисленными процессами и процессорами, которые могут иметься у компьютера (большинство компьютеров в наше время многоядерные).

Учитывая такую ситуацию, вертикальное масштабирование на основе Python может достигаться на основе разных стратегий:

- компиляция сценариев Python для достижения большей скорости исполнения. Несмотря на то что она легко осуществима, например при помощи **PyPy – динамического** (JIT) компилятора, который можно найти на http://pypy.org/, мы в нашей книге на самом деле к такому решению не обращались, потому что оно требует написания алгоритмов на Python с нуля;
- использование Python в качестве оберточного языка, тем самым связывая выполняемые на Python операции с выполнением внешних библиотек и программ, некоторые из которых приспособлены к многоядерной обработке. В нашей книге вы найдете много примеров, где вызываются специализированные библиотеки, в частности **библиотека для машин опорных векторов** LIBSVM (Library for Support Vector Machines), или такие програм-

мы, как **Vowpal Wabbit** (VW), **XGBoost** или **H2O**, для выполнения операций по машинному обучению;

- эффективное использование методов векторизации, т. е. специальных библиотек для выполнения матричных вычислений. Это может быть достигнуто при помощи библиотек **NumPy** или **pandas**, в которых задействуются вычисления с использованием модулей GPU. Модули GPU точно так же, как и многоядерные CPU, имеют свою собственную оперативную память и способны выполнять вычисления в параллельном режиме (как вы можете догадаться, они имеют многочисленные крошечные ядра). Методы векторизации на основе модулей GPU могут невероятно ускорить вычисления, в особенности во время работы с нейронными сетями. Однако GPU имеют свои собственные ограничения. Прежде всего располагаемая память имеет определенный канал ввода-вывода для передачи данных в память GPU и возврата результатов назад в CPU, и они требуют параллельного программирования посредством специального программного интерфейса (API), такого как **CUDA** для модулей GPU производства NVIDIA (отсюда следует, что требуется установка надлежащих драйверов и программ);
- сведение большой задачи к порциям и решение каждой порции поочередно в оперативной памяти (алгоритмы парадигмы «разделяй и властвуй»). Это влечет за собой разбиение данных из оперативной памяти или диска на части либо извлечение из них подвыборок и управление приближенными решениями задачи машинного обучения, что позволяет добиваться довольно эффективных результатов. Важно отметить, что разбиение на части и извлечение подвыборок может действовать как для прецедентов, так и для признаков (либо для обоих одновременно). Если исходные данные хранятся в дисковой памяти, то для итоговой производительности определяющими, безусловно, станут ограничения канала ввода-вывода;
- эффективное привлечение как многопроцессорной, так и многопоточной обработки в зависимости от используемого обучающегося алгоритма. Некоторые алгоритмы естественным образом могут разбивать свои операции на параллельные. В таких случаях единственным ограничением будут CPU и оперативная память (так как данные должны быть реплицированы для каждого параллельного рабочего процесса, который будет использоваться). Некоторые другие алгоритмы вместо этого используют преимущества многопоточной обработки, тем самым одновременно управляя бóльшим числом операций на тех же блоках памяти.

Горизонтальное масштабирование средствами Python

Горизонтальное масштабирование решений состоит в объединении многочисленных машин в кластер. При подключении машин (т. е. масштабируя вширь) можно также масштабировать каждую из них вертикально (т. е. вглубь), используя более мощные конфигурации (тем самым усиливая CPU, оперативную память и каналы ввода-вывода), применяя методы, упомянутые нами в предыдущем абзаце, и улучшая их производительность.

Подключая многочисленные машины, можно мобилизовать их вычислительную мощь в параллельном виде. Ваши данные будут распределены по многочисленным дисковым хранилищам/устройствам памяти, тем самым ограничивая

число операций передачи данных по каналу ввода-вывода и заставляя каждую машину обрабатывать только имеющиеся у нее данные (т. е. свое собственное дисковое хранилище или оперативную память).

В нашей книге это транслируется в эффективное использование внешних ресурсов посредством следующих платформ:

- платформы H2O;
- платформы Hadoop и ее компонентов, таких как распределенная файловая система HDFS, вычислительная парадигма MapReduce и менеджер ресурсов **YARN**;
- платформы Spark поверх Hadoop.

Python будет управлять всеми этими платформами (например, платформа Spark управляется ее Python'овским интерфейсом под названием pySpark).

Python для крупномасштабного машинного обучения

С учетом наличия многих удобных библиотек для машинного обучения и того факта, что этот язык программирования довольно популярен среди исследователей-аналитиков, Python был выбран в качестве базового языка для написания всего представленного в настоящей книге исходного кода.

В этой книге, по мере необходимости, мы будем предоставлять дальнейшие инструкции по установке других необходимых библиотек или инструментов. Здесь же вместо этого мы приступим к установке самых главных компонентов, т. е. языка Python и наиболее часто используемых библиотек, применяемых для вычислений и машинного обучения.

Выбор между Python 2 и Python 3

Перед тем как начинать, важно уяснить, что существуют две основные ветви Python: версии 2 и 3. Поскольку в обеих версиях много базовой функциональности изменилось, сценарии, созданные для одной версии, иногда несовместимы с другой (они будут работать, вызывая сообщения об ошибках и предупреждения). Несмотря на то что третья версия является новейшей, более старая по-прежнему широко используется в научной среде и является версией по умолчанию для многих действующих систем (в основном для совместимости при обновлениях). Когда версия 3 была выпущена (в 2008 г.), большинство научных библиотек еще не было готово, и поэтому научное сообщество продолжило использовать предыдущую версию. К счастью, с тех пор почти все библиотеки были обновлены, за исключением лишь некоторых (см. http://py3readiness.org по поводу обзора совместимости), оставшихся несовместимыми с Python 3.

Несмотря на недавний рост популярности Python 3 (который продолжит свое развитие в будущем, и мы не должны это забывать), Python 2 все еще широко используется среди исследователей и аналитиков. Кроме того, в течение длительного времени Python 2 устанавливался по умолчанию (например, в Ubuntu), таким образом, эта версия будет наиболее вероятной, которую большинство читателей будет иметь под рукой. Исходя из всех этих причин, для этой книги мы примем за основу Python 2. Это решение вызвано не какой-то *любовью к старым технологиям*, а попросту является практичным выбором в пользу того, чтобы

сделать книгу «*Крупномасштабное машинное обучение на Python*» доступной для большей аудитории:

- исходный код на Python 2 сразу охватит всю существующую аудиторию экспертов в области данных;
- пользователи Python 3 очень легко смогут преобразовать наши сценарии для работы в версии Python, которую они предпочитают, потому что написанный нами исходный код легко конвертируем.

В случае если есть необходимость подробно разобраться в различиях между Python 2 и Python 3, мы предлагаем прочитать эту веб-страницу о написании совместимого между Python 2–3 исходного кода: http://python-future.org/compatible_idioms.html. Кроме того, на веб-сайте **Python-Future** можно найти полезные сведения о том, как преобразовать исходный код Python 2 в Python 3: http://python-future.org/automatic_conversion.html.

Инсталляция среды Python

В качестве первого шага мы создадим рабочую среду для науки о данных, которую можно использовать для репликации и тестирования примеров из книги и создания прототипов собственных больших решений.

Независимо от того, на каком языке вы собираетесь разрабатывать свое приложение, с Python вы не будете испытывать трудностей при получении своих данных, создании из них модели и извлечении правильных параметров, которые необходимы для выполнения прогнозов в эксплуатационной среде.

Python – это общедоступный объектно-ориентированный кросс-платформенный язык программирования, который, по сравнению с его прямыми конкурентами (например, C/C++ и Java), производит очень сжатый и читаемый код. Это позволяет создавать действующий прототип программного обеспечения в очень короткие сроки и тестировать, поддерживать и масштабировать его в будущем. Он завоевал статус наиболее используемого языка среди программного инструментария исследователя-аналитика, потому что, будучи языком общего назначения, он в конечном итоге приобрел гибкость благодаря большому разнообразию имеющихся библиотек, которые способны легко и быстро помочь в решении самого широкого спектра как распространенных, так и нишевых задач.

Пошаговая установка

Если вы никогда не использовали Python (однако это не исключает того, что он на вашей машине может быть уже установлен), то сначала необходимо скачать установщик с основного веб-сайта проекта https://www.python.org/downloads/ (напомним, что мы используем версию 2) и затем установить его на локальной машине.

Этот раздел предоставляет вам полный контроль над тем, что может быть установлено на вашей машине. Это очень удобно, если вы собираетесь использовать Python одновременно как язык для прототипирования и программирования промышленного кода. Кроме того, этот раздел поможет отслеживать используемые версии библиотек. Так или иначе, предупреждаем, что пошаговая установка в действительности потребует некоторого времени и усилий. С другой стороны, установка готового научного дистрибутива облегчит бремя инсталляционной процедуры и хорошо подойдет для первого раза и обучения, потому что экономит довольно много времени, хотя и установит на ваш компьютер сразу большое ко-

личество библиотек (которые вы по большей части, возможно, никогда не будете использовать). Поэтому если вы хотите начать немедленно и не желаете слишком уж беспокоиться по поводу контроля над инсталляцией, то просто пропустите эту часть и перейдите к следующему разделу «*Научные дистрибутивы*».

Поскольку Python является мультиплатформенным языком программирования, вы найдете установщики для компьютеров, которые работают под управлением операционной системы Windows либо Linux-/Unix-подобных ОС. Напомним, что в некоторых дистрибутивах Linux (таких как Ubuntu) язык Python 2 уже включен в репозиторий по умолчанию, что делает процесс установки еще проще.

1. Откройте оболочку Python, введите python в терминале или щелкните по значку Python.
2. Затем для тестирования результата инсталляции выполните следующие команды в интерактивной оболочке Python, или ее стандартной интерактивной среде программирования **REPL** (от англ. Read-Eval-Print Loop, работающей в цикле чтения-вычисления-печати результата), или других решениях, таких как Spyder или PyCharm:

```
>>> import sys
>>> print(sys.version)
```

Если была поднята синтаксическая ошибка, то, значит, вы выполняете Python 3 вместо Python 2. Если же ошибка отсутствует и вы можете прочитать, что используемой версией является Python 2.7.x (во время написания книги последней была версия 2.7.12), то поздравляем с установкой версии Python, которую мы выбрали для этой книги. Для тех же читателей, кто работает с Python 3, подойдут все версии начиная с Python 3.4 (во время написания последней была версия 3.5.2).

Напоминаем, что, когда команда выдается в командой строке терминала, она предваряется префиксом `$`. В противном случае, если речь о стандартной среде Python REPL, ей предшествует подсказка `>>>`.

Установка библиотек

В зависимости от вашей системы и предыдущих инсталляций среда Python может не оказаться укомплектованной всем тем, в чем вы нуждаетесь, если не установлен дистрибутив (который, с другой стороны, обычно укомплектован гораздо шире, чем вам, возможно, понадобится).

Чтобы установить любую нужную библиотеку, можно применить команду `pip` или `easy_install`; однако в будущем поддержка команды `easy_install` будет прекращена, и **pip** имеет перед ней важные преимущества.

Инструмент установки библиотек Python **pip** непосредственно получает доступ к Интернету и выбирает их из каталога библиотек Python **PyPI** (https://pypi.python.org/pypi). PyPI представляет собой репозиторий, содержащий сторонние библиотеки с открытым исходным кодом, которые постоянно поддерживаются в работоспособном состоянии и сохраняются в репозитории их авторами.

Устанавливать библиотеки лучше всего при помощи `pip` по следующим причинам:

- он является предпочтительным диспетчером библиотек Python и начиная с Python 2.7.9 и Python 3.4 по умолчанию включен в двоичные установщики Python;

- он обеспечивает функциональность по деинсталляции библиотек;
- он возвращает вашу систему в исходное состояние и оставляет ее чистой, если по какой-либо причине установленная библиотека перестала работать.

Команда `pip` работает в командной строке и ускоряет весь процесс установки, обновления и удаления библиотек Python.

Как уже упоминалось, если вы работаете как минимум с Python 2.7.9 или Python 3.4, то команда `pip` должна уже иметься в наличии. Чтобы удостовериться в том, какие инструменты были установлены на локальной машине, выполните прямую проверку на возможные ошибки при помощи следующей команды:

```
$ pip -V
```

В некоторых инсталляциях в Linux и Mac OS устанавливается Python 3, а не Python 2, в результате чего эта команда может присутствовать как `pip3`, поэтому при получении ошибки при поиске `pip` попробуйте выполнить следующую команду:

```
$ pip3 -V
```

Если это так, то напомним, что `pip3` подходит только для установки библиотек в Python 3. Поскольку в книге мы работаем с Python 2 (если, разумеется, вы не решили использовать новейшую версию Python 3), то вашим основным установщиком библиотек всегда будет `pip`.

Как вариант можно также проверить доступность старой команды `easy_install`:

```
$ easy_install --version
```

Использовать `easy_install` вместо `pip` целесообразно, если вы работаете под Windows, потому что `pip` не устанавливает двоичных библиотек; поэтому если при установке библиотеки вы испытываете неожиданные трудности, то `easy_install` может сэкономить вам уйму времени.

Если проверка закончилась ошибкой, то вам действительно необходимо установить `pip` с нуля (и при этом также `easy_install`).

Для установки `pip` просто следуйте инструкциям на https://pip.pypa.io/en/stable/installing/. Самый безопасный путь состоит в том, чтобы скачать сценарий `get-pip.py` по прямой ссылке с https://bootstrap.pypa.io/get-pip.py и затем выполнить его при помощи следующей команды:

```
$ python get-pip.py
```

Между прочим, этот сценарий также установит настроечный инструмент `setuptools` с https://pypi.python.org/pypi/setuptools, который содержит `easy_install`.

В качестве альтернативы, если вы работаете под управлением Unix-подобной операционной системы Debian/Ubuntu, укороченный способ установки всего, что нужно, будет состоять в команде `apt-get`:

```
$ sudo apt-get install python3-pip
```

После проверки этого основного требования теперь все готово к установке всех библиотек, которые потребуются для выполнения примеров, прилагаемых к настоящей книге. Чтобы установить типовую библиотеку `<lib>`, нужно просто выполнить следующую команду:

```
$ pip install <lib>
```

Как вариант, если вы предпочитаете использовать команду easy_install, можно также выполнить следующую команду:

```
$ easy_install <lib>
```

После этого библиотека <lib> и все библиотеки, от которых она зависит, будут скачаны и установлены.

Если вы не уверены в том, была библиотека установлена или нет, просто попробуйте импортировать из нее модуль. Если интерпретатор Python поднимает ошибку импорта **ImportError**, то можно сделать вывод, что библиотека не была установлена.

Посмотрим на примере. Вот что происходит, когда библиотека NumPy была установлена:

```
>>> import numpy
>>>
```

А вот что – если она не установлена:

```
>>> import numpy
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ImportError: No module named numpy
```

В последнем случае, прежде чем ее импортировать, надо установить библиотеку посредством pip или easy_install.

Позаботьтесь о том, чтобы не перепутать библиотеки с модулями. При помощи pip вы устанавливаете библиотеку, а в среду Python вы импортируете модуль. Иногда библиотека и модуль имеют одинаковое название, но во многих случаях они не совпадают. Например, модуль **sklearn** включен в библиотеку **Scikit-learn**.

Способы обновления библиотек

Как правило, вы окажетесь в ситуации, когда необходимо обновить библиотеку, потому что некая связанная с ней другая библиотека, т. н. зависимость, требует наличия новой версии, либо имеется дополнительный функционал, который требуется задействовать. Для этого сначала нужно проверить версию установленной библиотеки, обратившись к атрибуту __version__, как показано на примере с библиотекой NumPy ниже:

```
>>> import numpy
>>> numpy.__version__ # 2 символа подчеркивания перед ним и после него
'1.9.0'
```

Далее, если нужно обновить ее до более нового выпуска, скажем, в точности до версии 1.9.2, то из командной строки можно выполнить следующую ниже команду:

```
$ pip install -U numpy==1.9.2
```

Как вариант (но мы его не рекомендуем, только в случае, если это оказывается необходимым), можно также использовать следующую команду:

```
$ easy_install --upgrade numpy==1.9.2
```

Наконец, если вы попросту заинтересованы в ее обновлении до последней доступной версии, то просто выполните следующую команду:

```
$ pip install -U numpy
```

Можно также выполнить альтернативную команду `easy_install`:

```
$ easy_install --upgrade numpy
```

Научные дистрибутивы

Из того, что вы прочли до сих пор, следует, что работа по созданию рабочей среды занимает у исследователя-аналитика много времени. Сначала нужно установить Python и затем, одну за другой, можно установить все библиотеки, в которых вы будете нуждаться. (Иногда инсталляционная процедура может пойти не так гладко, как вы надеялись.)

Если вы хотите сэкономить время и усилия и гарантированно получить полностью рабочую, готовую к использованию среду Python, то можно просто скачать, установить и использовать научный дистрибутив Python. Помимо языка Python, такие дистрибутивы также содержат множество предварительно установленных библиотек, и иногда в них даже имеются для использования дополнительные инструменты и интегрированные среды разработки (IDE). Несколько из них хорошо известно среди исследователей-аналитиков, и в последующих разделах вы познакомитесь с некоторыми главными особенности двух таких комплектов программного обеспечения, которые мы сочли самыми полезными и практичными.

Чтобы немедленно сосредоточиться на содержимом настоящей книги, мы предлагаем сначала быстро скачать и установить научный дистрибутив, в частности **Anaconda** (который, по нашему мнению, является самым полным из всех), и затем после выполнения примеров из этой книги полностью его деинсталлировать, чтобы установить только один Python, дополнив его лишь теми библиотеками, в которых вы нуждаетесь для разработки своих проектов.

И опять-таки, если это возможно, скачайте и установите версию дистрибутива, содержащую Python 2.

В качестве первого комплекта программного обеспечения мы рекомендуем попробовать дистрибутив Python Anaconda (https://www.continuum.io/downloads), предлагаемый компанией Continuum Analytics, который включает почти 200 библиотек, в т. ч. NumPy, SciPy, pandas, IPython, matplotlib, Scikit-learn и StatsModels. Это кросс-платформенный дистрибутив, который можно установить на машины с другими существующими дистрибутивами и версиями Python, а его базовая версия бесплатна. Дополнительные надстройки, которые содержат расширенные возможности, оплачиваются отдельно. Anaconda представляет двоичный диспетчер библиотек **conda** как инструмент командной строки для управления установкой библиотек. Как утверждается на веб-сайте дистрибутива, задача Anaconda состоит в том, чтобы обеспечить дистрибутив Python, готовый к работе на уровне предприятия, для крупномасштабной обработки данных, прогнозной аналитики и научных вычислений. Что касается версии 2.7 Python, то мы рекомендуем как минимум дистрибутив Anaconda 4.0.0. (Чтобы взглянуть на устанавливаемые вместе с Anaconda библиотеки, следует обратиться к списку на https://docs.continuum.io/anaconda/pkg-docs.)

Второе предложение – дистрибутив **WinPython** (http://winpython.sourceforge.net/). Этот дистрибутив может оказаться довольно интересной альтернативой, если вы работаете под Windows и желаете, чтобы ваш дистрибутив был переносимым (извините, но версии под Linux и Mac OS отсутствуют). WinPython – это тоже бесплатный дистрибутив Python с открытым исходным кодом, который поддерживается сообществом. И он тоже разработан, имея в виду исследователей-аналитиков, и включает в себя комплект основных библиотек, таких как NumPy, SciPy, matplotlib и IPython (в основном те же, что и в Anaconda). Он так же, как и Anaconda, содержит инструментальную среду разработки **Spyder**, которая может быть полезной, если у вас есть опыт работы в интерфейсе языка MATLAB. Решающее преимущество данного дистрибутива состоит в том, что он переносим (его можно поместить в любом каталоге или даже на флеш-накопитель USB), тем самым на компьютере могут существовать различные версии, их можно перемещать с одного компьютера Windows на другой, и более старая версия дистрибутива легко заменяется на более новую всего лишь путем смены каталога. При выполнении WinPython или его оболочки он автоматически установит все необходимые для выполнения Python переменные окружения так, как будто он был инсталлирован в регулярном режиме и зарегистрирован в системе.

Во время написания настоящей книги последним по времени дистрибутивом WinPython для версии Python 2.7 был созданный в октябре 2015 г. релиз 2.7.10; с тех пор публиковались обновления дистрибутива WinPython только для версии Python 3. После установки дистрибутива в операционной системе, возможно, потребуется обновить некоторые ключевые библиотеки, необходимые для выполнения примеров из этой книги.

Введение в Jupyter

Бесплатный проект IPython был запущен Фернандо Пересом в 2001 г. с целью решения проблемы отсутствия в Python стека научных исследований с использованием пользовательского программного интерфейса, который мог бы включать в процесс разработки программного обеспечения научный подход (главным образом экспериментирование и интерактивный поиск).

Научный подход подразумевает проведение быстрых экспериментов в отношении разных гипотез в воспроизводимой форме (подобно задаче разведочного анализа в науке о данных). Используя IPython во время написания своего исходного кода, вы сможете реализовывать разведочные, итеративные и эмпирические (путем проб и ошибок) исследования более естественным образом.

В конце 2015 г. значительная часть проекта IPython переместилась в новый проект под названием **Jupyter** (http://jupyter.org/). Этот новый проект расширяет потенциальное удобство использования исходного интерфейса IPython до широкого спектра языков программирования[1]. (Для получения полного списка языков программирования посетите https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages).

Благодаря мощной идее ядер языков программирования специальный внешний интерфейс позволяет передавать в них программы, которые выполняют

[1] Далее в книге все упоминания интерактивной среды IPython заменены на Jupyter, являющуюся более общей и более современной интерактивной средой программирования, куда IPython входит в качестве главной составной части (как нулевое ядро). – *Прим. перев.*

пользовательский программный код, и в качестве отклика получать результат исполненного исходного кода; вы можете пользоваться единым интерфейсом и интерактивным стилем программирования независимо от того, на каком языке пишется программа.

Jupyter (IPython является нулевым ядром, запускаемым с самого начала) можно описать как инструмент для выполнения интерактивных задач, пригодных для работы в консоли или веб-ориентированном блокноте, который предлагает специальные команды, помогающие разработчикам лучше понимать и создавать исходный код, пишущийся в настоящий момент.

В отличие от интегрированной среды разработки (IDE), которая строится вокруг идеи написания сценария, его последующего выполнения и оценки его результатов, среда программирования Jupyter позволяет писать свой исходный код порциями, именуемыми **ячейками**, последовательно выполнять каждую из них и оценивать результаты каждой ячейки отдельно, исследуя одновременно текстовые и графические результаты. Помимо графической интеграции, эта среда предоставляет дополнительную помощь благодаря настраиваемым командам, богатой истории (в формате JSON) и вычислительному параллелизму для повышения производительности во время работы с тяжелыми численными вычислениями.

Такой подход также в особенности продуктивен для заданий, связанных с разработкой исходного кода на основе данных, поскольку он автоматически реализует часто пренебрегаемую обязанность по документированию и иллюстрированию того, каким образом анализ данных был проведен, его предпосылок и допущений и его промежуточных и конечных результатов. Если часть вашего задания, кроме того, состоит в презентации работы с целью убедить в проекте внутренние или внешние заинтересованные стороны, то Jupyter может действительно совершать волшебство, выполняя презентацию за вас, при этом требуя лишь небольших дополнительных усилий. На https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks имеется множество примеров использования блокнотов, некоторые из которых могут вдохновить вас в вашей работе, как это вышло у нас.

На самом деле мы должны признаться, что поддержание чистого, актуального блокнота Jupyter сэкономило нам неисчислимое количество времени, когда неожиданно всплыли незапланированные встречи с менеджерами/заинтересованными сторонами, которые потребовали от нас торопливо представить состояние нашей работы.

Одним словом, Jupyter предлагает вам следующие возможности:

- видеть промежуточные результаты (и их отладку) для каждого шага анализа;
- выполнять только некоторые разделы (или ячейки) с исходным кодом;
- хранить промежуточные результаты в формате JSON и иметь возможность выполнять на них версионный контроль;
- выполнять презентацию работы (в виде комбинации текста, исходного кода и изображений), делясь ею при помощи средства просмотра блокнотов Jupyter Notebook Viewer (http://nbviewer.jupyter.org/) и легко экспортируя его в PY, HTML, PDF или даже в слайдовые презентации.

Блокноты Jupyter являются нашим предпочтительным выбором на протяжении всей книги. Они используются для ясной и эффективной иллюстрации операций, связанных с изложением материала, на основе сценариев и данных, и последующих результатов.

Хотя мы рекомендуем использовать исключительно Jupyter, если вы используете среду интерпретатора REPL или IDE, то вы можете использовать те же самые инструкции и ожидать идентичных результатов (за исключением форматов и расширений оператора печати для возвращаемых результатов).

Если в вашей операционной системе Jupyter не установлен, то его можно быстро установить, воспользовавшись следующей командой:

```
$ pip install jupyter
```

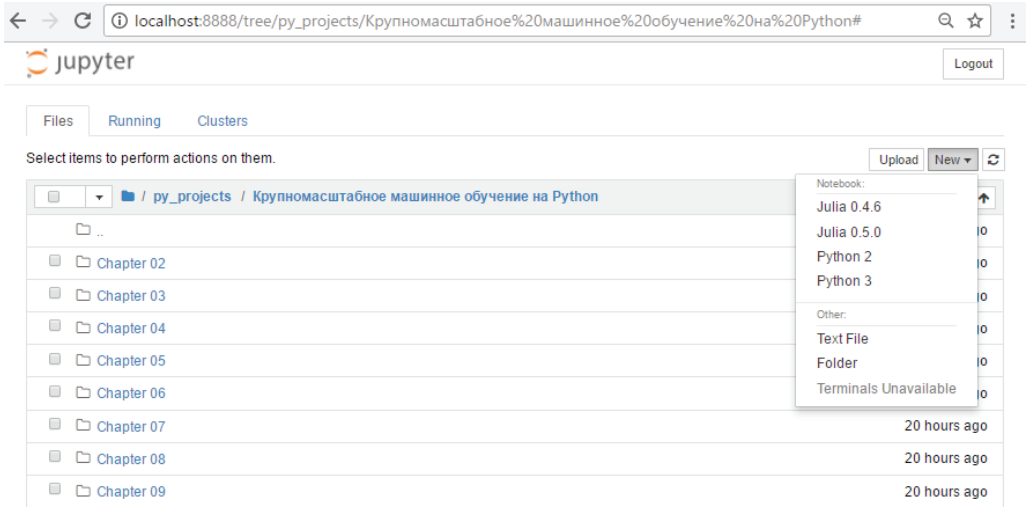
Полные инструкции по установке Jupyter (с охватом разных операционных систем) можно найти на http://jupyter.readthedocs.io/en/latest/install.html.

Если Jupyter уже установлен, то он должен быть обновлен как минимум до версии 4.1.

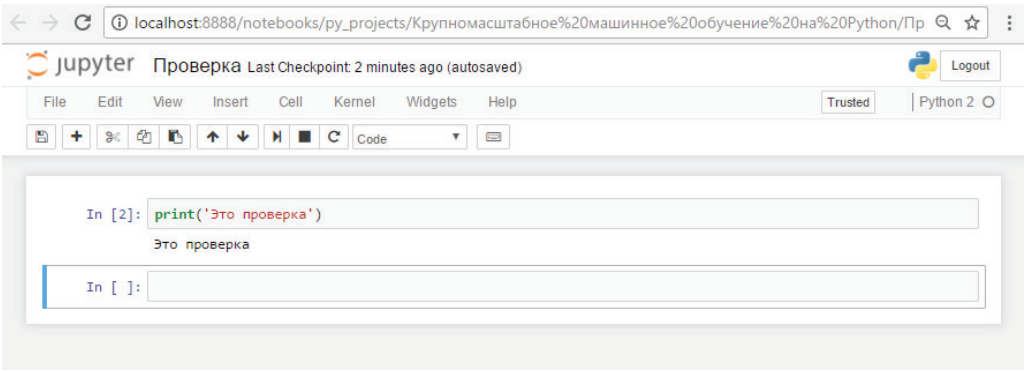
После установки можно сразу начать использовать Jupyter, вызвав его из командной строки:

```
$ jupyter notebook
```

Открыв экземпляр Jupyter в браузере, щелкните по кнопке **New** и в разделе блокнотов Notebooks выберите **Python 2** (в разделе могут присутствовать другие ядра, в зависимости от того, что установлено):



В этом месте новый пустой блокнот будет похож на следующий ниже снимок экрана, и можно приступать к вводу команд в ячейки:



Например, можно набрать в ячейку следующее:

```
In:
print («Это проверка»)
```

Набрав в ячейку исходный код, просто нажмите кнопку воспроизведения (под вкладкой **Cell**), чтобы его выполнить и получить результат. Ниже под ячейкой с введенным кодом появится еще одна ячейка. Если при заполнении ячеек нажать кнопку с плюсом в верхней строке меню, то вы получите новую ячейку, при этом перемещаться от ячейки к ячейке можно при помощи стрелок в меню.

Большинство других функций вполне интуитивно понятно, и мы приглашаем вас их попробовать. Чтобы лучше познакомиться с тем, как работает Jupyter, можно воспользоваться кратким практическим руководством, в частности http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/, либо получить специализированную книгу с описанием функционала Jupyter.

Для полного изложения всего спектра функциональных возможностей Jupyter при выполнении ядра IPython обратитесь к следующим двум книгам издательства Packt Publishing:

- *IPython Interactive Computing and Visualization Cookbook* («Книга рецептов по интерактивным вычислениям и визуализации в IPython»), автор Сирилл Россан (Cyrille Rossant), сентябрь 2014;
- *Learning IPython for Interactive Computing and Data Visualization* («Изучение IPython для интерактивных вычислений и визуализации данных»), автор Сирилл Россан, апрель 2013.

Следует учитывать, что для большей наглядности каждый блок инструкций Jupyter имеет пронумерованный оператор ввода и оператор вывода, поэтому в этой книге вы увидите, что исходный код структурирован в двух блоках, за исключением случаев, когда результат на выходе тривиален, – в этом случае ожидайте увидеть всего один входной оператор:

```
In: <вводимый исходный код>
Out: <полученный итоговый результат>
```

Как правило, вам придется лишь набирать исходный код в ячейки после оператора **In:** и выполнять его. Затем вы будете сравнивать полученные вами выходные данные с результатом в книге после оператора **Out:**, который мы фактически получили на наших компьютерах, когда тестировали приводимый исходный код.

Библиотеки Python

В настоящем разделе мы представим библиотеки, которые в этой книге будут применяться часто. Если вы не используете научного дистрибутива, то мы предлагаем пошаговое краткое описание того, на каких версиях необходимо остановиться и каким образом быстро и успешно их установить[1].

NumPy

Библиотека NumPy, разработанная Трэвисом Олифантом (Travis Oliphant), находится в сердцевине каждого аналитического решения на языке Python. Он предлагает пользователю многомерные массивы вместе с большим набором функций для работы с многочисленными математическими операциями на этих массивах. Массивы – это блоки данных, расположенные вдоль многочисленных размерностей, программно реализующие математические векторы и матрицы. Массивы полезны не только для хранения данных, но и для быстрых матричных операций (векторизации), которые необходимы, когда вы хотите решать оперативные задачи науки о данных.

- Веб-сайт: http://www.numpy.org/.
- Версия во время написания: 1.11.1 (1.13.0).
- Предлагаемая команда установки:

```
$ pip install numpy
```

В сообществе разработчиков на Python негласно принято при импорте библиотеки NumPy использовать псевдоним `np`:

```
import numpy as np
```

SciPy

Библиотека SciPy, исходный проект нескольких авторов (Travis Oliphant, Pearu Peterson и Eric Jones), дополняет функциональность библиотеки NumPy, предлагая более широкое разнообразие научных алгоритмов для линейной алгебры, разреженных матриц, обработки сигналов и изображений, оптимизации, быстрого преобразования Фурье и многих других.

- Веб-сайт: http://www.scipy.org/.
- Версия во время написания: 0.17.1 (0.19.0).
- Предлагаемая команда установки:

```
$ pip install scipy
```

Pandas

Библиотека pandas имеет дело со всем, что не могут делать NumPy и SciPy. В частности, благодаря своим специальным объектным структурам данных DataFrame и Series она позволяет обрабатывать сложные таблицы данных различных типов (то, что массивы NumPy не могут делать) и временные ряды. Благодаря создателю Уэс Маккинни (Wes McKinney) вы получите возможность легко и быстро загру-

[1] В строке с указанием версии библиотеки в скобках приведена версия на момент публикации перевода настоящей книги. – *Прим. перев.*

жать данные из разнообразных источников, а затем нарезать, группировать, обрабатывать пропущенные элементы, добавлять, переименовывать, агрегировать, видоизменять и, наконец, визуализировать их по своему усмотрению.

- Веб-сайт: http://pandas.pydata.org/.
- Версия во время написания: 0.18.0 (0.20.2).
- Предлагаемая команда установки:

```
$ pip install pandas
```


Традиционно библиотека pandas импортируется как `pd`:

```
import pandas as pd
```

Scikit-learn

Появившись в составе SciKits (SciPy Toolkits, т. е. инструментария библиотеки SciPy), сегодня библиотека Scikit-learn является ключевым элементом операций в области науки о данных на Python. Он предлагает все, в чем вы, возможно, будете нуждаться с точки зрения предобработки данных, обучения с учителем и без учителя, отбора модели, перекрестной проверки и метрик ошибочности. Мы будем подробно говорить об этой библиотеке на протяжении всей книги.

Разработка библиотеки Scikit-learn была начата Дэвидом Курнапо (David Cournapeau) в 2007 г. в рамках ежегодного проекта Google Summer of Code. С 2013 г. она была передана исследователям в Inria (во французском Исследовательском институте в области информатики и автоматизации).

Библиотека Scikit-learn предлагает модули для обработки данных (`sklearn.preprocessing` и `sklearn.feature_extraction`), отбора и перекрестной проверки моделей (`sklearn.model_selection` и `sklearn.metrics`) и полный комплект методов (`sklearn.linear_model`), в котором предполагается, что целевое значение, число или вероятность, будет линейной комбинацией входных переменных.

- Веб-сайт: http://scikit-learn.org/stable/.
- Версия во время написания: 0.17.1 (0.18.1).
- Предлагаемая команда установки:

```
$ pip install scikit-learn
```


Отметим, что импортированный модуль называется `sklearn`.

Matplotlib

Библиотека matplotlib первоначально была разработана Джоном Хартером (John Hunter). Она содержит все стандартные компоненты, необходимые для создания качественных графиков из массивов и их визуализации в интерактивном режиме.

Внутри модуля PyLab можно найти все MATLAB-подобные компоненты для построения графиков и диаграмм.

- Веб-сайт: http://matplotlib.org/.
- Версия во время написания: 1.5.1 (2.0.2).
- Предлагаемая команда установки:

```
$ pip install matplotlib
```

Имеется возможность импортировать только то, что требуется для целей визуализации:

```
import matplotlib as mpl
from matplotlib import pyplot as plt
```

Gensim

Библиотека с открытым исходным кодом Gensim, разработанная Радимом Жехужеком (Radim Řehůřek), предназначена для анализа больших текстовых наборов с использованием параллельно распределяемых онлайновых алгоритмов. Среди прочего продвинутого функционала в ней реализованы **латентно-семантический анализ** (LSA), тематическое моделирование на основе **латентного распределения Дирихле (LDA)** и мощный алгоритм Google **word2vec** для преобразования текстов в векторные признаки для использования в машинном обучении с учителем и без учителя.

- Веб-сайт: http://radimrehurek.com/gensim/.
- Версия во время написания: 0.13.1 (2.1.0).
- Предлагаемая команда установки:

```
$ pip install gensim
```

H2O

Платформа с открытым исходным кодом H2O была создана стартап-компанией **H2O.ai** (ранее называвшейся 0xdata) и служит для анализа больших данных. Она работает с языками программирования R, Python, Scala и Java. Платформа H2O позволяет легко использовать автономную машину (с привлечением многопроцессорной обработки) или кластер Hadoop (например, кластер в среде AWS), тем самым помогая достигать вертикальной и горизонтальной масштабируемости.

- Веб-сайт: http://www.h2o.ai.
- Версия во время написания: 3.8.3.3 (3.10.4.8).

Чтобы установить эту библиотеку, сначала необходимо скачать и установить в операционную систему среду Java (вам понадобится комплект разработчика приложений на Java **Java Development Kit** (сокращенно JDK) версии 1.8, поскольку H2O основывается на Java), затем можно обратиться к онлайновым инструкциям, предоставленным на http://www.h2o.ai/download/h2o/python.

Все шаги по инсталляции платформы можно свести к следующим строкам.

При помощи следующих ниже инструкций можно установить одновременно платформу H2O и ее программный интерфейс для Python, который мы использовали в нашей книге:

```
$ pip install -U requests
$ pip install -U tabulate
$ pip install -U future
$ pip install -U six
```

Эти команды установят необходимые библиотеки, и затем можно установить платформу, позаботившись об удалении любой предыдущей установки:

```
$ pip uninstall h2o
$ pip install h2o
```

Для того чтобы установить ту же самую версию, которую мы применяли в нашей книге, можно поменять последнюю команду `pip install` на следующую:

```
$ pip install http://h2o-release.s3.amazonaws.com/h2o/rel-turin/3/Python/
h2o-3.8.3.3-py2.py3-none-any.whl
```

Если возникнут какие-либо проблемы, приглашаем посетить страницу платформы H2O в Google-группах, где можно получить справку по вашей проблеме: https://groups.google.com/forum/#!forum/h2ostream.

XGBoost

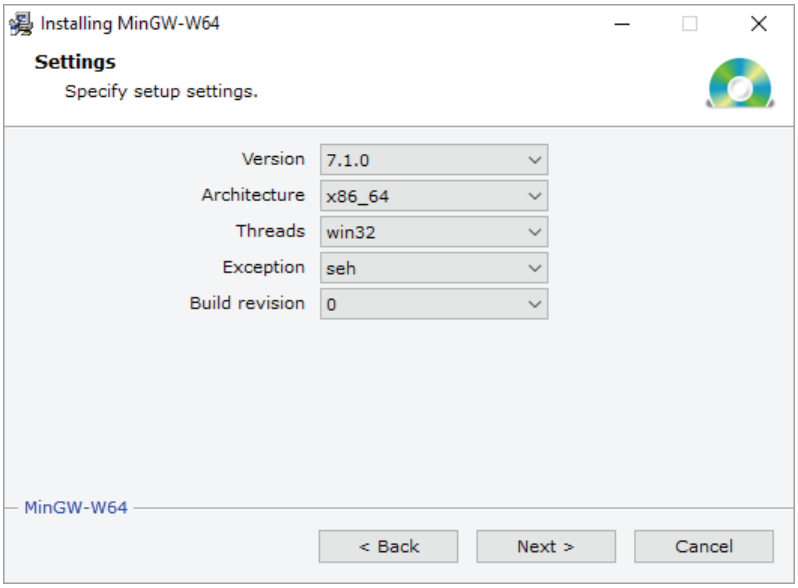
Масштабируемая, переносимая и распределенная библиотека градиентного бустинга **XGBoost** (древовидный ансамблевый алгоритм машинного обучения) имеется для языков Python, R, Java, Scala, Julia и C++; она может работать на одиночной машине (с привлечением многопоточной обработки) одновременно в кластерах Hadoop и Spark.

- Веб-сайт: https://xgboost.readthedocs.io/en/latest/.
- Версия во время написания: 0.4 (0.6a2).

Подробные инструкции по установке библиотеки XGBoost в вашей операционной системе можно найти на https://github.com/dmlc/xgboost/blob/master/doc/build.md.

Установка XGBoost в Linux и Mac OS довольно прямолинейная, но немного более заковыристая для пользователей Windows. По этой причине ниже мы в пошаговом режиме приводим процедуру ее установки, чтобы получить работающую под Windows версию этой библиотеки:

1. Прежде всего скачать и установить распределенную систему контроля версий **Git для Windows** (https://git-for-windows.github.io/).
2. Затем потребуется наличие в операционной системе компилятора **MinGW** (минималистской среды разработки GNU для Windows). Его можно скачать с https://sourceforge.net/projects/mingw-w64/ либо https://wiki.qt.io/MinGW-64-bit или http://www.mingw.org/ в соответствии с характеристиками вашей системы. Во время инсталляции можно выбрать следующие настройки (на рисунке показано окно настроек среды MinGW из первой ссылки):



3. Из командной строки выполнить следующее:

```
$ git clone --recursive https://github.com/dmlc/xgboost
$ cd xgboost
$ git submodule init
$ git submodule update
```

4. Затем из командной строки, находясь в каталоге xgboost, скопировать конфигурацию для 64-разрядных систем, сделав ее конфигурацией по умолчанию:

```
$ copy make\mingw64.mk config.mk
```

   Как вариант можно скопировать простую 32-разрядную версию:

```
$ copy make\mingw.mk config.mk
```

5. После копирования конфигурационного файла запустить компилятор, установив число потоков равным 4 для ускорения процедуры компиляции[1]:

```
$ make -j4
```

6. В заключение, если компилятор завершил свою работу без ошибок, установить библиотеку в свою среду Python, выполнив следующие команды:

```
$ cd python-package
$ python setup.py install
```

Theano

Библиотека Python **Theano** позволяет эффективным образом определять, оптимизировать и вычислять математические выражения, в т. ч. многомерные массивы. В сущности, она предоставляет все структурные компоненты, необходимые для создания глубоких нейронных сетей.

- Веб-сайт: http://deeplearning.net/software/theano/.
- Версия во время написания: 0.8.2 (0.10.0.dev1).

Установка Theano должна быть прямолинейной, поскольку теперь она находится в каталоге библиотек PyPI:

```
$ pip install Theano
```

Если нужно установить самую последнюю обновленную версию библиотеки, то ее можно склонировать с GitHub:

```
$ git clone git://github.com/Theano/Theano.git
```

Затем можно перейти непосредственно к установке в среде Python:

```
$ cd Theano
$ python setup.py install
```

Чтобы проверить результат установки, можно выполнить следующие команды в терминале/командной оболочке и проверить отчеты по установке:

```
$ pip install nose
$ pip install parameterized
$ nosetests theano
```

---

[1] Компоновщик make в MinGW имеет имя `mingw32-make.exe`, поэтому следующая команда должна быть: `mingw32-make -j4`. – *Прим. перев.*

Если вы работаете в Windows и предыдущие инструкции не срабатывают, то можно попробовать следующее:

1. Установить 64-разрядный комплект компиляторов TDM-GCC x64 (http://tdm-gcc.tdragon.net/).
2. Открыть командную строку Anaconda и выполнить следующее:

```
$ conda update conda
$ conda update -all
$ conda install mingw libpython
$ pip install git+git://github.com/Theano/Theano.git
```

Библиотека Theano требует наличия библиотеки libpython, которая пока не совместима с версией Python 3.5, поэтому, будучи установленным в Windows, он не работает, и этот факт может оказаться наиболее вероятной причиной.

Кроме того, веб-сайт Theano предоставляет пользователям Windows некоторую информацию, которая может вас поддерживать, когда все остальное не получается: http://deeplearning.net/software/theano/install_windows.html.

Для обеспечения горизонтального масштабирования при помощи Theano с использованием модулей GPU крайне необходима установка драйверов программно-аппаратной архитектуры параллельных вычислений **CUDA** от компании NVIDIA, а также SDK для генерирования исходного кода и его выполнения на CPU. Если вы не располагаете достаточными сведениями об инструментарии CUDA Toolkit, то, собственно, вы можете начать с приведенной ниже веб-страницы, чтобы глубже разобраться в используемой технологии: https://developer.nvidia.com/cuda-toolkit.

Таким образом, если ваш компьютер обладает графическим процессором NVIDIA, то вы сможете найти все необходимые инструкции по установке CUDA, используя приведенную ниже учебную страницу непосредственно от самой компании NVIDIA: http://docs.nvidia.com/cuda/cuda-quick-start-guide/index.html#axzz4A8augxYy.

TensorFlow

Еще одна программная библиотека с открытым исходным кодом **TensorFlow** так же, как и Theano, предназначена для численных вычислений. В ней вместо простых массивов используются графы потока данных. Узлы в таком графе представлены математическими операциями, тогда как дуги графа – многомерными массивами данных (так называемыми тензорами), перемещаемыми между узлами. Первоначально разработчики Google развивали проект TensorFlow в рамках команды Google Brain Team, но совсем недавно сделали эту библиотеку общедоступной.

- Веб-сайт: https://github.com/tensorflow/tensorflow.
- Версия во время написания: 0.8.0 (1.2.0rc2).

Чтобы установить TensorFlow на своем компьютере, следуйте инструкциям, находящимся по следующей ссылке: https://www.tensorflow.org/install/.

Имеется поддержка Windows; соответствующие инструкции по установке можно найти, перейдя по ссылке: https://www.tensorflow.org/install/install_windows.

В отличие от других используемых в книге библиотек, TensorFlow работает только с Python 3.5+. Для пользователей Windows хорошим компромиссом может

быть выполнение данной библиотеки в Linux в рамках виртуальной машины или машины Docker. (Приведенная выше веб-страница с информацией по установке библиотеки содержит указания относительно того, как это сделать.)

sknn

Библиотека **sknn** (расширение `scikit-neuralnetwork` библиотеки scikit для работы с нейронными сетями) представляет собой обертку вокруг библиотеки Pylearn2 и служит в качестве вспомогательного средства для реализации глубоких нейронных сетей, не требуя от вас становиться экспертом по Theano. В качестве бонуса настоящая библиотека совместима с программным интерфейсом библиотеки Scikit-learn.

- Веб-сайт: https://scikit-neuralnetwork.readthedocs.io/en/latest/.
- Версия во время написания: 0.7 (0.7).
- Предлагаемая команда установки:

```
$ pip install scikit-neuralnetwork
```

В факультативном порядке, если есть потребность воспользоваться преимуществами наиболее продвинутого функционала, в частности сверткой (конволюцией), объединением или вертикальным масштабированием, необходимо завершить установку следующим образом:

```
$ pip install -r pip install -r https://raw.githubusercontent.com/aigamedev/scikit-
neuralnetwork/master/requirements.txt
```

После установки также необходимо выполнить следующее:

```
$ git clone https://github.com/aigamedev/scikit-neuralnetwork.git
$ cd scikit-neuralnetwork
$ python setup.py develop
```

Как и в случае с библиотекой XGBoost, это сделает библиотеку `sknn` доступной в вашей среде Python.

Theanets

Библиотека **theanets** представляет собой написанный на Python комплект инструментов по глубокому обучению и нейронным сетям, в котором для ускорения вычислений используется библиотека Theano. Как и в случае с библиотекой sknn, она разработана с целью упростить взаимодействие с функционалом библиотеки Theano для создания моделей глубокого обучения.

- Веб-сайт: https://github.com/lmjohns3/theanets.
- Версия во время написания: 0.7.3 (0.7.3).
- Предлагаемая команда установки:

```
$ pip install theanets
```

Текущую версию данной библиотеки можно также загрузить с GitHub и установить непосредственно в Python:

```
$ git clone https://github.com/lmjohns3/theanets
$ cd theanets
$ python setup.py develop
```

Keras

Библиотека Python **Keras** – это минималистская, мультимодульная библиотека для программирования нейронных сетей, которая может выполняться поверх библиотек TensorFlow и Theano.

- Веб-сайт: http://keras.io/.
- Версия во время написания: 1.0.5 (2.0.5).
- Предлагаемая команда установки:

```
$ pip install keras
```

Последнюю версию библиотеки (в виде версии, находящей в процессе доработки) можно установить при помощи следующей команды:

```
$ pip install git+git://github.com/fchollet/keras.git
```

Другие вспомогательные библиотеки

Завершая этот длинный обзор целого ряда библиотек, которые вы увидите в действии на последующих страницах настоящей книги, мы закончим тремя простыми и одновременно довольно полезными библиотеками, которые не нуждаются в подробных описаниях, но которые нужно обязательно установить в вашей операционной системе: **memory_profiler**, **climate** и **NeuroLab**.

Профилировщик оперативной памяти `memory_profiler` – это библиотека, которая управляет тем, как вычислительный процесс использует память. Она также помогает анализировать потребление памяти конкретным сценарием Python в построчном режиме. Данную библиотеку можно установить следующим образом:

```
$ pip install -U memory_profiler
```

Библиотека climate состоит всего из нескольких основных утилит командной строки, предназначенных для Python. Ее можно быстро установить следующим образом:

```
$ pip install climate
```

Наконец, библиотека NeuroLab – это чрезвычайно простая нейросетевая библиотека, которая отдаленно основана на комплекте инструментов **Neural Network Toolbox** (NNT) языка MATLAB. В ее основе лежат библиотеки NumPy и SciPy, но не Theano, и поэтому от нее не следует ожидать какой-то удивительной производительности. Однако стоит отметить, что она предлагает хороший набор инструментов для обучения. Эту библиотеку можно легко установить следующим образом:

```
$ pip install neurolab
```

Резюме

В этой вводной главе мы проиллюстрировали различные способы, которыми можно сделать алгоритмы машинного обучения масштабируемыми, используя язык Python (методы вертикального и горизонтального масштабирования). Мы также предложили несколько мотивирующих примеров и заложили основу для

книги, проиллюстрировав, каким образом устанавливать на машину среду Python. В частности, мы познакомили вас со средой интерактивного программирования Jupyter и охватили все самые важные библиотеки, которые будут использоваться в настоящей книге.

В следующей главе мы займемся обсуждением того, каким образом алгоритм стохастического градиентного спуска способен помочь справляться с крупными наборами данных с привлечением стандартного потока ввода-вывода на одиночной машине. В целом мы охватим различные способы потоковой передачи данных из больших файлов либо репозиториев данных и их подачи на вход базового обучаемого алгоритма. Вы будете поражены тем, как простые решения могут оказаться эффективными, и обнаружите, что даже настольный компьютер может легко перемалывать большие данные.

Глава 2

Масштабируемое обучение в Scikit-learn

Зачастую при наличии довольно мощных и одновременно доступных компьютеров последних лет не представляет особого труда загрузить набор данных в оперативную память, подготовить матрицы данных, натренировать алгоритм машинного обучения и протестировать его обобщающие способности с использованием вневыборочных наблюдений. Однако все чаще и чаще масштаб подлежащих переработке данных настолько огромен, что их загрузка в основную память компьютера становится невозможной, и даже если с этим удастся справиться, получить результат не получится как с точки зрения управления данными, так и с точки зрения машинного обучения.

Приемлемые альтернативные стратегии, помимо обработки данных в основной памяти, вполне возможны: разбивка данных на выборки, использование параллелизма и, наконец, обучение малыми пакетами данных либо на одиночных прецедентах. В центре внимания настоящей главы будет готовое решение, которое предлагает библиотека Scikit-learn: потоковая передача мини-пакетов, состоящих из прецедентов (т. е. наблюдений), из хранилища и при инкрементном обучении на их основе. Подобного рода решение называется внеядерным обучением (out-of-core learning), или обучением вне основной памяти компьютера.

Обработка данных посильными порциями и инкрементное обучение – замечательная идея. Однако когда вы попытаетесь ее реализовать, она может оказаться гораздо сложнее из-за ограничений в обучающихся алгоритмах. К тому же передача данных в потоке требует, чтобы вы думали иначе с точки зрения управления данными и извлечения признаков. Помимо знакомства с функционалом библиотеки Scikit-learn для внеядерного обучения, мы также попытаемся представить решения на Python вне всякого сомнения чрезвычайно сложных задач, с которыми можно столкнуться, когда придется в каждый момент времени наблюдать только малые порции своих данных.

В этой главе мы затронем следующие темы:

- реализация внеядерного обучения в библиотеке Scikit-learn;
- эффективное управление потоками данных при помощи хэширования признаков;
- составные части механизма стохастического обучения;
- реализация науки о данных на основе онлайнового обучения;
- преобразования потоков данных в рамках обучения без учителя.

Внеядерное обучение

Как было упомянуто во вступлении к настоящей главе, внеядерное обучение относится к разряду алгоритмов, работающих с данными, которые не умещаются в оперативной памяти одиночного компьютера, но которые легко могут уместиться в хранилище данных, таком как локальный жесткий диск или веб-репозиторий. Объем имеющейся в распоряжении RAM, т. е. основной, памяти на одиночной машине может в действительности колебаться от нескольких гигабайт (иногда 2 Гб, чаще 4 Гб, но мы возьмем за основу, что вы располагаете максимум 2 Гб) до 256 Гб на больших серверных машинах. Крупные серверы можно получить в облачных вычислительных веб-службах, таких как **Эластичное вычислительное облако** компании Amazon (Elastic Compute Cloud, EC2), тогда как ваши возможности по хранению данных могут легко превышать терабайтные объемы, используя при этом всего лишь внешний привод (скорее всего, объемом порядка 1 Тб, но который может достигать 4 Тб).

Поскольку машинное обучение основывается на глобальном уменьшении функции стоимости, многие алгоритмы первоначально задумывались для работы на каждой итерации процесса оптимизации с использованием абсолютно всех имеющихся данных. Это в особенности верно для всех алгоритмов, основанных на статистическом обучении, в которых используется матричное исчисление, к примеру инвертирование матриц, к тому же алгоритмы, основанные на жадном поиске, нуждаются в оценивании как можно большего объема данных, перед тем как сделать следующий шаг. Следовательно, наиболее распространенные готовые к работе алгоритмы наподобие регрессионных (взвешенные линейные комбинации признаков) обновляют свои коэффициенты, пытаясь минимизировать объединенную ошибку всего набора данных целиком. Деревья решений, будучи очень чувствительными к шуму в наборе данных, аналогичным образом вынуждены для отыскания наилучшего решения выбирать оптимальное расщепление, основываясь на всех имеющихся данных.

Если в такой ситуации данные не умещаются в основной памяти компьютера, то остается не так уж и много возможных решений. Можно увеличить имеющуюся оперативную память (в зависимости от ограничений системной платы, после чего придется обратиться к распределенным системам, таким как Hadoop и Spark, и это решение мы упомянем в последних главах настоящей книги), либо попросту уменьшить набор данных, чтобы он уместился в оперативной памяти.

Если данные разреженные, т. е. в наборе данных есть много нулевых значений, то можно преобразовать плотную матрицу в разреженную. Это, как правило, происходит с текстовыми данными со многими столбцами, потому что каждый из них – это слово, но с малым числом значений, которые обозначают количества словоупотреблений, потому что одиночные документы обычно показывают ограниченный набор слов. Иногда решить проблему может использование разреженных матриц, позволяя загружать и обрабатывать другие довольно крупные наборы данных, но это не панацея (извините, никаких бесплатных обедов, т. е. отсутствует решение, которое подходило бы для абсолютно всех задач), потому что некоторые матрицы данных, хотя и разреженные, могут иметь пугающие размеры.

В такой ситуации можно всегда попытаться уменьшить набор данных путем сокращения числа прецедентов (строк) либо ограничения числа признаков (столб-

цов), тем самым снижая размерности матрицы набора данных и занятой ею области в оперативной памяти. Сокращение набора данных посредством извлечения только части наблюдений называется подвыборкой (частичным отбором либо просто выборкой). Подвыборка, по сути, не является неправильным решением, но она имеет серьезные недостатки, и о них необходимо помнить прежде, чем принимать решение о дальнейшем ходе анализа данных.

Подвыборка как приемлемый вариант

При извлечении подвыборки в действительности отбрасывается часть имеющегося информационного многообразия, при этом невозможно быть абсолютно уверенным, что отбрасываются избыточные, не совсем полезные данные. В действительности же некоторые неизвестные ранее «жемчужины» можно обнаружить только в результате рассмотрения всех данных. Несмотря на то что в вычислительном плане это решение выглядит привлекательным – потому что извлечение подвыборки требует лишь того, чтобы генератор случайных чисел сообщал, следует ли брать некий прецедент или нет, – извлекая отобранный набор данных, вы на самом деле рискуете ограничить возможности своего алгоритма тем, что он будет обучаться правилам и ассоциациям в данных не в полном объеме. В компромиссе между дисперсией и смещением извлечение подвыборки вызывает инфляцию дисперсии в предсказаниях, потому что оценки будут неопределеннее вследствие случайного шума в данных либо выбросов, т. е. периферийных наблюдений.

В мире больших данных побеждает алгоритм с более качественными данными, потому что он может обучиться большему количеству способов соотнесения предсказания с предикторами, чем другие модели с меньшими по объему (либо более шумными) данными. Следовательно, подвыборка, хотя и является приемлемым решением, может наложить ограничение на результаты вашей деятельности по машинному обучению из-за менее точных предсказаний и большей дисперсии оценок.

Недостатки подвыборки так или иначе могут быть преодолены, если обучаться многочисленным моделям на многочисленных подвыборках данных и затем собирать все решения в ансамбль либо накладывать результаты работы всех моделей поверх друг друга в каскад, тем самым создавая уменьшенную матрицу данных для проведения дальнейшей тренировки. Эта процедура называется агрегацией бутстрапированных выборок, или бэггингом. (Благодаря этому вы на самом деле сжимаете признаки, тем самым сокращая пространство данных в оперативной памяти.) Мы займемся исследованием способов составления ансамблей и каскадов в одной из глав книги и узнаем, как они в действительности снижают дисперсию оценок, накачиваемую за счет подвыборки.

Как вариант вместо сокращения прецедентов можно сократить признаки, но опять же мы получим проблему, которая состоит в том, что сперва нужно построить модель из данных, чтобы проверить, какие признаки можно отобрать, так что нам по-прежнему нужно построить модель с данными, которые не умещаются в оперативной памяти.

Оптимизация по одному прецеденту за раз

Разобравшись, что подвыборка данных является хоть и всегда приемлемым, но не оптимальным решением, мы должны оценить другой подход, и внеядерный подход на самом деле не требует отказываться от наблюдений (строк) или призна-

ков (столбцов). Просто на тренировку модели уходит чуть больше времени, требуя большего количества итераций и пересылки данных из хранилища в оперативную память компьютера. Прямо сейчас мы дадим первое интуитивно понятное объяснение того, как работает внеядерное обучение.

Начнем с обучения, т. е. процесса, в котором мы пытаемся соотнести неизвестную функцию, демонстрирующую отклик (число либо исход, имея в виду задачу регрессии либо классификации) относительно имеющихся данных. Обучение осуществляется путем подгонки внутренних коэффициентов обучающегося алгоритма в попытке достичь оптимального соответствия на имеющихся данных, а именно минимизируя функцию стоимости, т. е. меру, которая говорит, насколько хорошей является наша аппроксимация. Если резюмировать, мы говорим о процессе оптимизации.

Различные алгоритмы оптимизации, аналогично градиентному спуску, представляют собой процессы, которые в состоянии работать на любых объемах данных. Они занимаются получением градиента оптимизации (направления в ходе оптимизации) и заставляют обучающийся алгоритм адаптировать свои параметры, следуя вдоль градиента.

Если говорить конкретно о градиентном спуске, то после определенного числа итераций при условии решаемости задачи и отсутствия других проблем, таких как слишком высокий темп обучения, градиент должен стать настолько малым, что процесс оптимизации можно остановить. В конце процесса можно с уверенностью сказать, что было найдено решение, которое является оптимальным (потому что это глобальный оптимум, хотя иногда он может оказаться локальным, в случае если аппроксимируемая функция не выпуклая).

Поскольку направленность, продиктованная градиентом, может быть взята, основываясь на любом количестве примеров, она может быть взята и на одиночном прецеденте. Взятие градиента на одиночном прецеденте требует небольшого темпа обучения, но в конце процесс может достичь такой же оптимизации, что и градиентный спуск, взятый на полных данных. В конечном счете алгоритму требуется только направление, которое правильно сориентирует процесс обучения по подгонке под имеющиеся данные. Следовательно, обучение такому направлению на случайном одиночном спуске из данных является полностью выполнимым:

- можно получить одинаковые результаты, как если бы мы обрабатывали все данные одним разом, хотя ход оптимизации может стать чуть более шероховатым; если большинство наблюдений указывает на оптимальное направление, то алгоритм пойдет по нему. Единственная трудность будет состоять в правильной доводке нужных параметров процесса обучения и многократном прохождении по данным, для того чтобы оптимизация гарантированно завершилась, так как такая процедура обучения намного медленнее, чем работа со всеми имеющимися данными;
- не будет никаких особых трудностей с тем, как справиться с хранением одиночного прецедента в основной памяти машины, оставляя весь массив данных вне ее. Другие трудности могут быть связаны с перемещением данных в размере одиночных примеров из репозитория в основную память машины. Масштабируемость гарантируется, потому что требующееся для обработки данных время линейное; времязатраты использования еще одного

прецедента всегда одинаковые, независимо от общего числа прецедентов, которые предстоит обработать.

Метод подгонки обучающегося алгоритма каждый раз всего на одном прецеденте либо подмножестве данных, который умещается в оперативной памяти, называется онлайновым (или динамическим) обучением, а градиентный спуск, выполняемый на основе таких одиночных наблюдений, называется стохастическим градиентным спуском (stochastic gradient descent, SGD). Как уже указывалось ранее, онлайновое обучение – это внеядерная методика, и она принята за основу в библиотеке Scikit-learn во многих обучающихся алгоритмах.

Создание системы внеядерного обучения

Мы проиллюстрируем внутреннее устройство алгоритма стохастического градиентного спуска чуть позже в следующих нескольких абзацах, предложив на этот счет больше подробностей и рассуждений. Сейчас же знание того, каким образом можно обучаться вне ядра (благодаря алгоритму стохастического градиентного спуска), позволяет нарисовать более четкую картину того, что мы должны сделать, чтобы заставить его работать на своем компьютере.

Можно подразделить свои действия на несколько разных задач:

1. Подготовить доступ к своему репозиторию данных для потоковой передачи по одному прецеденту. Это действие может потребовать рандомизации порядка следования строк данных перед доставкой данных на компьютер, чтобы удалить любую информацию, которая может быть вызвана упорядоченностью данных.
2. Сначала выполнить небольшое обследование данных, возможно, на небольшой части всех данных (к примеру, взяв первые десять тысяч строк) в попытке выяснить, согласуются ли поступающие прецеденты между собой по числу признаков, типу данных, присутствию или отсутствию значений, минимальным и максимальным значениям каждой переменной и их средним и медианным значениям. Уточнить размах или класс целевой переменной.
3. Преобразовать каждую полученную строку данных в фиксированный формат, который может быть принят обучающимся алгоритмом (плотный или разреженный вектор). На данном этапе можно выполнить любое элементарное преобразование, к примеру превратить категориальные признаки в числовые или дать числовым признакам взаимодействовать без посторонней помощи, взяв векторное произведение непосредственно самих признаков.
4. После рандомизации порядка следования примеров (как упомянуто в первом пункте) определиться с процедурой перекрестной проверки, используя регулярно отложенные выборки (holdout) либо выборки, отложенные после определенного числа наблюдений.
5. Выполнить доводку гиперпараметров путем повторной потоковой передачи данных либо обрабатывая небольшие выборки из них. На этом этапе самое подходящее время, чтобы заняться конструированием признаков (с использованием обучения без учителя и специальных функций преобразования, таких как ядерные аппроксимации) и задействовать регуляризацию и отбор признаков.
6. Построить свою итоговую модель с использованием данных, которые были зарезервированы для тренировки, и идеальным образом протестировать эффективность модели на новых данных.

В качестве первого шага мы обсудим способы подготовки данных и затем легко создадим поток, подходящий для онлайнового обучения, с привлечением вспомогательных функций из библиотек Python, в частности из pandas и Scikit-learn.

Потоковая передача данных из источников

Некоторые данные в полном смысле этого слова текут потоком через ваш компьютер, когда имеется порождающий процесс, который передает данные; эти данные можно обрабатывать на лету либо просто отбрасывать без возможности их последующего восстановления, если, конечно, их не сохранить где-нибудь в репозитории архивации данных. Это все равно, что таскать воду из речного потока, – река продолжает течь, однако вы можете фильтровать и обрабатывать всю воду, покуда она течет. Это совершенно другая стратегия, радикально отличающаяся от обработки всех данных сразу, которая больше походит на ситуацию, когда вся вода помещена в отстойник, ограниченный плотиной (по аналогии с обработкой всех данных в оперативной памяти).

В качестве примера потоковой передачи можно привести поток данных, который через определенные промежутки продуцируется датчиком, или, еще проще, поток сообщений социальной сети Twitter. Как правило, основными источниками потоков данных являются:

- датчики окружающей среды, измеряющие температуру, давление и влажность;
- следящие датчики GPS, записывающие географическое положение (широту-долготу);
- спутники, записывающие данные съемки поверхности Земли;
- данные видеонаблюдения и звукозаписи;
- интернет-трафик, или информационный обмен в Интернете.

Однако вам нечасто придется обрабатывать реальные потоки данных, за исключением разве что оставленных ими статических записей, которые хранятся в репозитории или файле. В таких случаях поток может быть воссоздан согласно определенным критериям, например путем извлечения каждый раз последовательно или случайным образом по одной записи. Если, например, данные содержатся в текстовом файле либо файле данных с разделением полей запятыми (CSV), то нам остается лишь поставлять по одной строке файла и передавать строку на вход обучающегося алгоритма.

Что касается примеров в настоящей и следующей главах, то мы будем обрабатывать файлы, хранящиеся на локальном жестком диске, и подготовим исходный код на Python, необходимый для их извлечения и потоковой передачи. Мы не будем пользоваться миниатюрным набором данных, но и не перегрузим локальный жесткий диск слишком большим объемом данных для проведения проверок и демонстраций.

Наборы данных для реальных дел

С 1987 г. в **Калифорнийском университете в Ирвайне** (UCI), США, размещен **репозиторий машинного обучения** UCI Machine Learning Repository. Это большое хранилище наборов данных для эмпирического тестирования алгоритмов машинного обучения сообществом, занимающимся исследованием и разработкой

в области обучения машин. Во время написания настоящей книги этот репозиторий содержал порядка 350 наборов данных из совершенно разных предметных областей и для совершенно разных целей от задач регрессии и классификации, т. е. методов обучения с учителем, до задач без учителя. Все имеющиеся наборы данных можно просмотреть на https://archive.ics.uci.edu/ml/.

С нашей стороны мы отобрали несколько наборов данных, которые на протяжении всей книги окажутся полезными в качестве исходного материала с большим числом строк или столбцов для решения сложных задач на основе необычного, по нашим временам, но все еще приемлемого компьютера с 2 Гб RAM:

Название набора данных	URL-адрес набора данных	Тип задачи	Строки и столбцы
Набор данных Bike Sharing	https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset	Регрессия	17389, 16
Набор данных BlogFeedback	https://archive.ics.uci.edu/ml/datasets/BlogFeedback	Регрессия	60021, 281
Набор данных Buzz in social meadia	https://archive.ics.uci.edu/ml/datasets/Buzz+in+social+media+	Регрессия и классификация	140000, 77
Набор данных Census Income (KDD)	https://archive.ics.uci.edu/ml/datasets/Census-Income + % 28KDD%29	Классификация с пропущенными данными	299285, 40
Набор данных Covertype	https://archive.ics.uci.edu/ml/datasets/Covertype	Классификация	581012, 54
Данные Cup 1999	https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data	Классификация	4000000, 42

Чтобы скачать и использовать набор данных из репозитория UCI, необходимо попасть на страницу, посвященную набору данных, и перейти по ссылке под заголовком **Download:Data Folder** (Скачать:Папка данных). Мы приготовили несколько сценариев для автоматического скачивания данных, которые будут помещаться именно в тот каталог, в котором вы работаете на Python, тем самым упрощая доступ к данным.

Ниже приведено несколько подготовленных нами функций, к которым мы будем обращаться регулярно на протяжении всех глав, когда возникнет потребность скачать любой набор данных из UCI:

```
In:
import urllib2
import requests, io, os, StringIO
import numpy as np
import tarfile, zipfile, gzip

def unzip_from_UCI(UCI_url, dest=''):
    """
    Скачивает и распаковывает наборы данных из UCI в формате zip
    """
    response = requests.get(UCI_url)
    compressed_file = io.BytesIO(response.content)
    z = zipfile.ZipFile(compressed_file)
    print(u'Извлечение в {0}'.format(os.getcwd().decode('cp1251') +'\\'+dest))
    for name in z.namelist():
```

```
        if '.csv' in name:
            print(u'\tраспакован {0}'.format(name))
            z.extract(name, path=os.getcwd().decode('cp1251')+'\\'+dest)

def gzip_from_UCI(UCI_url, dest=''):
    """
    Скачивает и распаковывает наборы данных из UCI в формате gzip
    """
    response = urllib2.urlopen(UCI_url)
    compressed_file = io.BytesIO(response.read())
    decompressed_file = gzip.GzipFile(fileobj=compressed_file)
    filename = UCI_url.split('/')[-1][:-3]
    with open(os.getcwd().decode('cp1251')+'\\'+filename, 'wb') as outfile:
        outfile.write(decompressed_file.read())
    print(u'Файл {0} разархивирован'.format(filename))

def targzip_from_UCI(UCI_url, dest='.'):
    """
    Скачивает и распаковывает наборы данных из UCI в формате tar.gz
    """
    response = urllib2.urlopen(UCI_url)
    compressed_file = StringIO.StringIO(response.read())
    tar = tarfile.open(mode="r:gz", fileobj = compressed_file)
    tar.extractall(path=dest)
    datasets = tar.getnames()
    for dataset in datasets:
        size = os.path.getsize(dest+'\\'+dataset)
        print(u'Файл {0} составляет {1} байт'.format(dataset,size))
    tar.close()

def load_matrix(UCI_url):
    """
    Скачивает наборы данных из UCI в матричной форме
    """
    return np.loadtxt(urllib2.urlopen(UCI_url))
```

Скачивание исходного кода примеров

Подробные шаги по скачиванию комплекта с исходным кодом примеров упомянуты в предисловии настоящей книги. Пожалуйста, ознакомьтесь с ними.

Комплект исходного кода для настоящей книги также размещен на веб-сайте GitHub на https://github.com/PacktPublishing/Large-Scale-Machine-Learning-With-Python. Мы также располагаем другими комплектами исходного кода примеров, которые можно выбрать из нашего обширного каталога книг и видео, предлагаемого на странице https://github.com/PacktPublishing/. Можете убедиться сами!

Эти функции представляют собой вспомогательные обертки вокруг разных библиотек, в частности `tarfile`, `zipfile` и `gzip`, предназначенных для работы со сжатыми данными. Файл открывается при помощью модуля `urllib2`, который генерирует дескриптор для удаленной системы и предусматривает последовательную передачу данных с их сохранением в памяти в виде последовательности символов либо в двоичном режиме, используя для этого соответственно классы `StringIO` и `BytesIO` из модуля `io` – этот модуль посвящен управлению потоками (https://docs.python.org/2/library/io.html). После помещения файла в оперативную память он

восстанавливается подобно обычному файлу, находящемуся на диске, благодаря функциям, специально предназначенным для распаковки сжатых файлов.

Четыре предоставленные выше функции должны помочь быстро скачать наборы данных, независимо от того, в каком формате они упакованы: zip, tar, gzip или же просто как текст в матричной форме, устраняя хлопоты, связанные с ручным скачиванием и операциями по извлечению из архива.

Первый пример – потоковая передача набора данных Bike-sharing

В качестве первого примера мы обратимся к набору данных Bike-sharing об аренде велосипедов. Этот набор данных содержит два CSV-файла с данными о почасовом и ежедневном количестве велосипедов, арендованных между 2011 и 2012 г. в системе Capital Bikeshare, расположенной в г. Вашингтон, округ Колумбия, США (https://www.capitalbikeshare.com/). Данные характеризуются соответствующей погодной и сезонной информацией относительно дня сдачи в аренду. Набор данных связан со следующей публикацией: *Fanaee-T, Hadi, and Gama, Joao, Event labeling combining ensemble detectors and background knowledge, Progress in Artificial Intelligence (2013): pp. 1–15, Springer Berlin Heidelberg* («*Маркировка событий с совмещением ансамблевых детекторов и фонового знания*»).

Первая задача состоит в том, чтобы сохранить набор данных на локальном жестком диске при помощи вспомогательных оберточных функций, которые были определены всего несколько абзацев ранее:

```
In:
UCI_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00275/Bike-Sharing-
Dataset.zip'
unzip_from_UCI(UCI_url, dest='data\\bikesharing')

Out:
Извлечение в C:\scisoft\WinPython-64bit-2.7.9.4\notebooks\data\bikesharing
    распакован day.csv
    распакован hour.csv
```

Если все выполнено успешно, то программный код укажет, в каком каталоге были сохранены CSV-файлы, и распечатает имена обоих разархивированных файлов.

Сохранив информацию на физическом устройстве, в этом месте мы напишем сценарий, составляющий сердцевину нашей системы внеядерного обучения, которое обеспечит потоковую передачу данных из файла. Сначала мы воспользуемся библиотекой `csv`, предлагающей двойной выбор: восстановить данные в виде списка либо словаря Python. Начнем со списка:

```
In:
import os, csv

local_path = os.getcwd()
source = 'data\\bikesharing\\hour.csv'
SEP = str(',') # определяется с целью его замены в дальнейшем
               # для файла с другим разделителем

with open(local_path.decode('cp1251')+'\\'+source, 'rb') as R:
```

```
    iterator = csv.reader(R, delimiter=SEP)

    for n, row in enumerate(iterator):
        if n==0:
            header = row
        else:
            # заглушка для ОБРАБОТКИ ДАННЫХ
            # заглушка для МАШИННОГО ОБУЧЕНИЯ
            pass

    print('Всего строк: {0}'.format(n+1))
    print('Заголовок: {0}'.format(', '.join(header)))
    print('Примеры значений: {0}'.format(', '.join(row)))
```

```
Out:
Всего строк: 17380
Заголовок: instant, dteday, season, yr, mnth, hr, holiday, weekday, workingday, weathersit,
temp, atemp, hum, windspeed, casual, registered, cnt
Примеры значений: 17379, 2012-12-31, 1, 1, 12, 23, 0, 1, 1, 1, 0.26, 0.2727, 0.65, 0.1343, 12,
37, 49
```

Полученный результат сообщит, сколько было прочитано строк, содержание заголовка – первая строка CSV-файла (сохраненного в списке) – и содержимое строк (для удобства мы распечатали последнюю строку). Читающий объект `csv.reader` создает итератор, который благодаря циклу `for` будет раз за разом выдавать по одной строке файла. Отметим, что внутрь фрагмента кода мы поместили два комментария, тем самым указав, куда на протяжении этой главы мы будем помещать другой исходный код для предобработки данных и машинного обучения.

В данном случае признаки должны обрабатываться с использованием позиционного подхода, который применяет индексную позицию метки в заголовке. Это может вызывать небольшое неудобство, в случае если вам придется активно управлять признаками. Это неудобство преодолевается использованием читающего объекта `csv.DictReader`, который на выходе производит словарь Python (словарь не упорядочен, но признаки могут быть легко восстановлены по их меткам):

```
In:
with open(local_path.decode('cp1251')+'\\'+source, 'rb') as R:
    iterator = csv.DictReader(R, delimiter=SEP)

    for n, row in enumerate(iterator):
        # заглушка для ОБРАБОТКИ ДАННЫХ
        # заглушка для МАШИННОГО ОБУЧЕНИЯ
        pass

    print('Всего строк: {0}'.format(n+1))
    print('Примеры значений: {0}'.format(row))
```

```
Out:
Всего строк: 17379
Примеры значений: {'mnth': '12', 'cnt': '49', 'holiday': '0', 'instant': '17379', 'temp':
'0.26', 'dteday': '2012-12-31', 'hr': '23', 'season': '1', 'registered': '37', 'windspeed':
'0.1343', 'atemp': '0.2727', 'workingday': '1', 'weathersit': '1', 'weekday': '1', 'hum':
'0.65', 'yr': '1', 'casual': '12'}
```

Использование инструментов ввода-вывода библиотеки pandas

В качестве альтернативы модулю `csv` можно воспользоваться функцией библиотеки pandas `read_csv`. Данная функция, как указано в документации к pandas на http://pandas.pydata.org/pandas-docs/stable/io.html, специально предназначена для закачивания CSV-файлов и является составной частью целого ряда функций, посвященных операциям ввода-вывода на различных файловых форматах.

Большие преимущества использования функций ввода-вывода библиотеки pandas заключаются в следующем:

- можно обеспечить совместимость исходного кода при изменении типа источника, т. е. требуется всего лишь переопределить потоковый итератор;
- можно получить доступ к большому числу разных форматов, таких как CSV, неформатированный текст, HDF, JSON и SQL-запрос к конкретной базе данных;
- данные поступают потоком в порции требуемого размера, реализованные как структуры данных DataFrame, в результате чего можно обращаться к признакам позиционным способом либо путем восстановления их метки благодаря методам loc, iloc, ix, которые, как правило, используются для взятия срезов и подмножеств в таблице данных DataFrame.

Приведем пример с использованием того же самого подхода, что и ранее, но на этот раз построенный вокруг функции pandas `read_csv`:

```
In:
import pandas as pd

CHUNK_SIZE = 1000

with open(local_path.decode('cp1251')+'\\'+source, 'rb') as R:
    iterator = pd.read_csv(R, chunksize=CHUNK_SIZE)

    for n, data_chunk in enumerate(iterator):
        print('Размер закачанной порции данных: %i прецедентов, %i признаков'
              % (data_chunk.shape))
        # заглушка для ОБРАБОТКИ ДАННЫХ
        # заглушка для МАШИННОГО ОБУЧЕНИЯ
        pass

    print('Примеры значений: \n{0}'.format(str(data_chunk.iloc[0])))

Out:
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
```

```
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 1000 прецедентов, 17 признаков
Размер закачанной порции данных: 379 прецедентов, 17 признаков
Примеры значений:
instant            17001
dteday        2012-12-16
season                 4
yr                     1
mnth                  12
hr                     3
holiday                0
weekday                0
workingday             0
weathersit             2
temp                0.34
atemp             0.3333
hum                 0.87
windspeed          0.194
casual                 1
registered            37
cnt                   38
Name: 17000, dtype: object
```

Здесь очень важно отметить, что итератор инстанцирован с указанием размера порции данных, т. е. числа строк, которые итератор должен возвращать во время каждой итерации. Параметр размера порции `chunksize` может принимать значения от 1 до любого значения, хотя ясно, что размер мини-пакета данных (извлеченной порции данных) строго привязан к размеру доступной памяти, в которой он будет храниться и обрабатываться в следующей фазе предобработки.

Передача в оперативную память более крупных порций дает преимущество только с точки зрения доступа к диску. Порции меньших размеров требуют многократного доступа к диску и, в зависимости от характеристик физического хранилища данных, более продолжительного времени прохождения данных. Тем не менее, с точки зрения машинного обучения, порции меньшего или бóльшего размеров имеют для внеядерных функций библиотеки Scikit-learn незначительную разницу, так как они обучаются, учитывая всякий раз всего один прецедент, что делает их по-настоящему линейными в части вычислительных затрат.

Работа с базами данных

В качестве примера гибкости инструментов ввода-вывода библиотеки pandas мы покажем еще один пример, теперь с использованием СУБД SQLite3 и потоковой передачей данных из простого запроса поочередными порциями. Этот пример предлагается исключительно для дидактического применения. Работа с большим хранилищем в базах данных может действительно принести преимущества с точки зрения времени обработки и дискового пространства.

Данные, сгруппированные в таблицы базы данных SQL, могут быть нормализованы, тем самым удаляя избыточность и повторения и экономя дисковое пространство. Нормализация базы данных позволяет расположить столбцы и табли-

цы в базе данных таким образом, чтобы уменьшить их размерности без потери информации. Нередко это осуществляется путем разбивки таблиц и перекодирования повторяющихся данных в ключи. Кроме того, работу оптимизированной с точки зрения памяти, операций и многопроцессорной обработки реляционной базы данных можно ускорить и предвосхитить часть тех предварительных работ, которые выполняются в сценариях Python.

В среде программирования на языке Python СУБД SQLite (http://www.sqlite.org) поставляется по умолчанию и является хорошим выбором благодаря следующим причинам:

- имеет открытый исходный код;
- может обрабатывать большие объемы данных (теоретически до 140 Тб в расчете на каждую базу данных, хотя вряд ли мы увидим приложение SQLite, которое будет иметь дело с такими объемами данных);
- работает в Mac OS и одновременно в 32- и 64-разрядных средах под управлением Linux и Windows;
- не требует какой-то серверной инфраструктуры или отдельной инсталляции (нулевой конфигурации), поскольку все данные хранятся в одном-единственном дисковом файле;
- может быть легко расширена при помощи исходного кода Python, который становится хранимой процедурой.

Кроме того, стандартная библиотека Python содержит модуль `sqlite3`, обеспечивающий все функции для создания базы данных с нуля и дальнейшей работы с ней.

В нашем примере мы сначала закачаем в базу данных SQLite наш CSV-файл с набором данных Bike-sharing об аренде велосипедов на ежедневной и почасовой основе и затем организуем из нее поточную передачу, точно так же, как мы делали это из CSV-файла. Предоставленный исходный код закачки базы данных будет использоваться повторно на протяжении всей книги и допускает повторное использование в ваших собственных приложениях, поскольку он не привязан к какому-то определенному примеру, который мы предлагаем (просто нужно поменять параметры ввода и вывода, и это все):

```
In:
import os, sys
import sqlite3, csv,glob

SEP = str(',')

def define_field(s):
    try:
        int(s)
        return 'integer'
    except ValueError:
        try:
            float(s)
            return 'real'
        except:
            return 'text'

def create_sqlite_db(db='data\\database.sqlite', file_pattern=''):
```

```
    conn = sqlite3.connect(db)
    conn.text_factory = str  # позволяет хранить данные в кодировке utf-8

    c = conn.cursor()

    # обойти каталог и обработать все файлы .csv,
    # которые подходят для построения БД
    target_files = glob.glob(file_pattern)

    print('Создание {0} таблиц(ы) в {1} из файла(ов): {2}'.format(len(target_files), db, ',
'.join(target_files)))

    for k,csvfile in enumerate(target_files):
        # удалить путь и расширение и использовать все остальное,
        # как имя таблицы
        tablename = os.path.splitext(os.path.basename(csvfile))[0]

        with open(csvfile, "rb") as f:
            reader = csv.reader(f, delimiter=SEP)

            f.seek(0)
            for n,row in enumerate(reader):
                if n==11:
                    types = map(define_field,row)
                else:
                    if n>11:
                        break

            f.seek(0)
            for n,row in enumerate(reader):
                if n==0:

                    sql = "DROP TABLE IF EXISTS %s" % tablename
                    c.execute(sql)
                    sql = "CREATE TABLE %s (%s)" % (tablename,
                              ", ".join(["%s %s" % (col, ct)
                                         for col, ct in zip(row, types)]))
                    print('{0}) {1}'.format(k+1,sql))
                    c.execute(sql)

                    # создание индексов для ускоренных соединений
                    # на длинных строках
                    for column in row:
                        if column.endswith("_ID_hash"):
                            index = "%s__%s" % ( tablename, column )
                            sql = "CREATE INDEX %s on %s (%s)"
                                   % ( index, tablename, column )
                            c.execute(sql)

                    insertsql = "INSERT INTO %s VALUES (%s)" % (tablename,
                                ", ".join(["?" for column in row]))
                    rowlen = len(row)
                else:
                    # поднять ошибку, если есть строки
                    # с неправильным числом полей
                    if len(row) == rowlen:
                        c.execute(insertsql, row)
```

```
                else:
                    print('Ошибка в строке {0} в файле {1}'
                          .format(n,csvfile))
                    raise ValueError(
                       'Хьюстон, у нас проблемы в строке {0}'.format(n))

        conn.commit()
        print('* Вставлено {0} строк'.format(n))

    c.close()
    conn.close()
```

Следующий ниже сценарий передает допустимое имя базы данных и шаблон для определения файлов, которые вы хотите импортировать (подстановочные знаки, такие как *, допускаются), и с нуля создает необходимую новую базу данных и таблицы, впоследствии заполняя их всеми имеющимися данными:

```
In:
create_sqlite_db(db='data\\bikesharing.sqlite',
                 file_pattern='data\\bikesharing\\*.csv')

Out:
Создание 2 таблиц(ы) в data\bikesharing.sqlite из файла(ов): data\bikesharing\day.csv, data\
bikesharing\hour.csv
1) CREATE TABLE day (instant integer, dteday text, season integer, yr integer, mnth integer,
holiday integer, weekday integer, workingday integer, weathersit integer, temp real, atemp
real, hum real, windspeed real, casual integer, registered integer, cnt integer)
* Вставлено 731 строк
2) CREATE TABLE hour (instant integer, dteday text, season integer, yr integer, mnth integer,
hr integer, holiday integer, weekday integer, workingday integer, weathersit integer, temp
real, atemp real, hum real, windspeed real, casual integer, registered integer, cnt integer)
* Вставлено 17379 строк
```

Следующий ниже сценарий сообщает о типах данных для создаваемых полей и количестве строк, поэтому довольно просто удостовериться, что во время импорта все прошло гладко. Теперь из базы данных можно легко организовать потоковую передачу. В нашем примере мы создадим внутреннее объединение между таблицами с почасовыми и ежедневными данными и извлечем данные на почасовой основе с информацией об арендных платежах по дням:

```
In:
import os, sqlite3
import pandas as pd

DIR_PATH = os.getcwd()
DB_NAME = 'data\\bikesharing.sqlite'
CHUNK_SIZE = 2500

conn = sqlite3.connect(DB_NAME)
conn.text_factory = str   # позволяет хранить данные в кодировке utf-8
sql = "SELECT H.*, D.cnt AS day_cnt FROM hour AS H INNER JOIN day as D ON (H.dteday =
D.dteday)"
DB_stream = pd.io.sql.read_sql(sql, conn, chunksize=CHUNK_SIZE)

for j,data_chunk in enumerate(DB_stream):
    x, y = data_chunk.shape
```

```
    print('Порция {0} -'.format(j+1)),
    print('Размер закачанной порции данных: {0} прецедентов, {1} признаков'
          .format(x,y))
    # заглушка для ОБРАБОТКИ ДАННЫХ
    # заглушка для МАШИННОГО ОБУЧЕНИЯ
```

```
Out:
Порция 1 - Размер закачанной порции данных: 2500 прецедентов, 18 признаков
Порция 2 - Размер закачанной порции данных: 2500 прецедентов, 18 признаков
Порция 3 - Размер закачанной порции данных: 2500 прецедентов, 18 признаков
Порция 4 - Размер закачанной порции данных: 2500 прецедентов, 18 признаков
Порция 5 - Размер закачанной порции данных: 2500 прецедентов, 18 признаков
Порция 6 - Размер закачанной порции данных: 2500 прецедентов, 18 признаков
Порция 7 - Размер закачанной порции данных: 2379 прецедентов, 18 признаков
```

Если потоковую передачу нужно ускорить, то необходимо оптимизировать базу данных, прежде всего создав индексы для реляционного запроса, который вы намереваетесь использовать.

Строка `conn.text_factory = str` составляет очень важную часть сценария; позволяет хранить данные в кодировке UTF-8. Если эту команду проигнорировать, то можно столкнуться с непривычными ошибками при вводе данных.

Особое внимание упорядочению прецедентов

В качестве заключительного комментария по теме потоковой передачи данных мы должны предупредить, что во время потоковой передачи в процесс обучения в действительности вносится скрытая информация в силу порядка следования примеров, на которых базируется процесс машинного обучения.

По сути дела, онлайновые ученики оптимизируют свои параметры на основе каждого прецедента, который они оценивают. В процессе оптимизации каждый прецедент направляет ученика в определенном направлении. В глобальном плане ученик должен взять правильное направление оптимизации, если дано достаточно большое число оцененных прецедентов. Однако если ученик вместо этого тренируется на смещенных наблюдениях (например, упорядоченных по времени либо сгруппированных осмысленным образом), то алгоритм тоже будет обучаться смещению. Во время обучения что-то еще можно сделать для того, чтобы не запоминать ранее наблюдавшихся прецедентов, но так или иначе некоторое смещение все равно будет допущено. Если в процессе обучения извлекается временной ряд – отклик на течение времени часто является частью модели, – такое смещение оказывается довольно полезным, но в большинстве других случаев оно действует как некая переподгонка, или излишняя аппроксимация, и приобретает форму дефицита обобщающей способности в итоговой модели.

Если в данных имеется некая упорядоченность (как, например, порядок следования идентификаторов) и вы не хотите, чтобы алгоритм машинного обучения ей обучился, то, прежде чем начать потоковую передачу, в качестве предупредительной меры можно перемешать строки данных и получить произвольный порядок следования, более подходящий для онлайнового стохастического обучения.

Самый быстрый способ, который к тому же занимает меньше места на диске, состоит в том, чтобы передать набор данных потоком непосредственно в оперативную память и уменьшить его путем сжатия. В большинстве случаев, но не

во всех, это будет работать благодаря применяемому алгоритму сжатия и относительной разреженности и избыточности данных, которые вы используете для тренировки. В случаях, когда это не работает, необходимо перемешать данные непосредственно на диске, что влечет за собой большее потребление дискового пространства.

Ниже мы сначала представим быстрый способ перемешивания данных в оперативной памяти благодаря библиотеке `zlib`, которая может быстро сжимать строки в оперативной памяти, и функции для перемешивания `shuffle` из стандартного модуля `random`:

```
In:
import zlib
from random import shuffle

def ram_shuffle(filename_in, filename_out, header=True):
    with open(filename_in, 'rb') as f:
        zlines = [zlib.compress(line, 9) for line in f]
        if header:
            first_row = zlines.pop(0)
    shuffle(zlines)
    with open(filename_out, 'wb') as f:
        if header:
            f.write(zlib.decompress(first_row))
        for zline in zlines:
            f.write(zlib.decompress(zline))

import os

local_path = os.getcwd().decode('cp1251')
source = 'data\\bikesharing\\hour.csv'

ram_shuffle(filename_in=local_path+'\\'+source,
            filename_out=local_path+'\\data\\bikesharing\\shuffled_hour.csv',
            header=True)
```

Для пользователей UNIX одиночный вызов команды `sort` (с параметром `-R`) очень легко перемешивает огромные количества текстовых файлов, делая это намного эффективнее, чем любая реализация на Python. Ее можно комбинировать с шагами распаковки и сжатия, используя для этого конвейерные операторы.

Так, следующая ниже команда должна добиться поставленной цели:

```
zcat sorted.gz | sort -R | gzip - > shuffled.gz
```

В случае если объема RAM для хранения всех сжатых данных не хватает, то единственное эффективное решение состоит в том, чтобы выполнить операции непосредственно на самом файле в том виде, в котором он размещен на диске. Следующий фрагмент исходного кода определяет функцию, которая будет систематически разбивать файл на меньшие по размеру файлы, перемешивать их содержимое и размещать их снова в случайном порядке в большем по размеру файле. В результате получится не совсем полная случайная перестановка, но строки файла разбросаны в достаточной мере, чтобы уничтожить любой предыдущий порядок следования строк, который мог бы повлиять на онлайновое обучение:

```
In:
import os
from random import shuffle
import numpy as np
import pandas as pd

def disk_shuffle(filename_in, filename_out, header=True,
                 iterations = 3, CHUNK_SIZE = 2500, SEP=str(',')):
    for i in range(iterations):
        with open(filename_in, 'rb') as R:
            iterator = pd.read_csv(R, chunksize=CHUNK_SIZE)
            for n, df in enumerate(iterator):
                if n==0 and header:
                    header_cols =SEP.join(df.columns)+'\n'
                df.iloc[np.random.permutation(len(df))
                       ].to_csv(str(n)+'_chunk.csv',
                                index=False,
                                header=False,
                                sep=SEP)
        ordering = list(range(0,n+1))
        shuffle(ordering)
        with open(filename_out, 'wb') as W:
            if header:
                W.write(header_cols)
            for f in ordering:
                with open(str(f)+'_chunk.csv', 'r') as R:
                    for line in R:
                        W.write(line)
                os.remove(str(f)+'_chunk.csv')

        filename_in = filename_out
        CHUNK_SIZE = int(CHUNK_SIZE / 2)

import os

local_path = os.getcwd().decode('cp1251')
source = 'data\\bikesharing\\hour.csv'

disk_shuffle(filename_in=local_path+'\\'+source,
             filename_out=local_path+'\\data\\bikesharing\\shuffled_hour.csv',
             header=True)
```

Стохастическое обучение

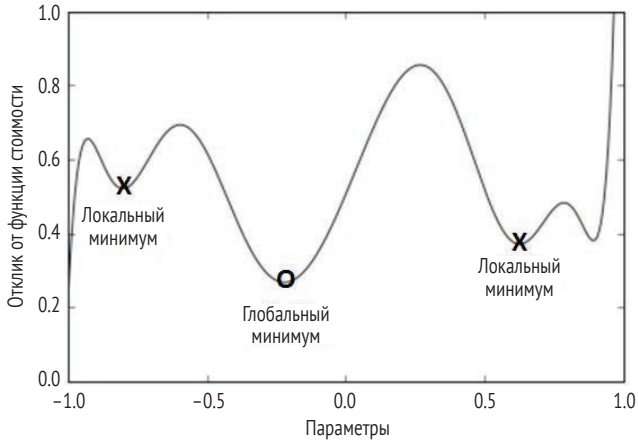
Разработав процесс потоковой передачи данных, теперь пора взглянуть на процесс обучения, так как именно обучение и его специфические потребности определяют выбор наилучшего способа управления данными и их преобразования на этапе предобработки.

Онлайновое обучение, в противоположность пакетному, работает с бóльшим числом итераций и получает направление движения поочередно от каждого отдельного прецедента, тем самым обеспечивая более блуждающую процедуру обучения, чем оптимизация, выполняемая на пакете данных, которая сразу же стремится указать правильное направление, исходя из всех данных целиком.

Пакетный градиентный спуск

В машинном обучении алгоритм градиентного спуска является стержневым, и поэтому он пересмотрен, чтобы его адаптировать под онлайновое обучение. Во время обработки пакетных данных градиентный спуск может минимизировать функцию стоимости линейного регрессионного анализа, используя намного меньше вычислений, чем статистические алгоритмы. Сложность вычисления градиентного спуска занимает порядка $O(n*p)$, делая возможным обучение коэффициентам регрессии даже в случае больших значений n (которое обозначает число наблюдений) и p (число переменных). Он работает также идеально, когда в тренировочных данных присутствуют высококоррелированные или даже идентичные признаки.

Все основывается на простом методе оптимизации: набор параметров изменяется в результате многочисленных итераций таким образом, что он постепенно сходится к оптимальному решению, начиная со случайного. Градиентный спуск – это теоретически хорошо проработанный метод оптимизации с известными гарантиями сходимости для определенных задач, в частности регрессионных. Тем не менее давайте начнем со следующего ниже рисунка, на котором показано сложное соответствие (характерное для нейронных сетей) между значениями, которые параметры могут принять (представляя пространство гипотезы), и результатом с точки зрения минимизации функции стоимости:



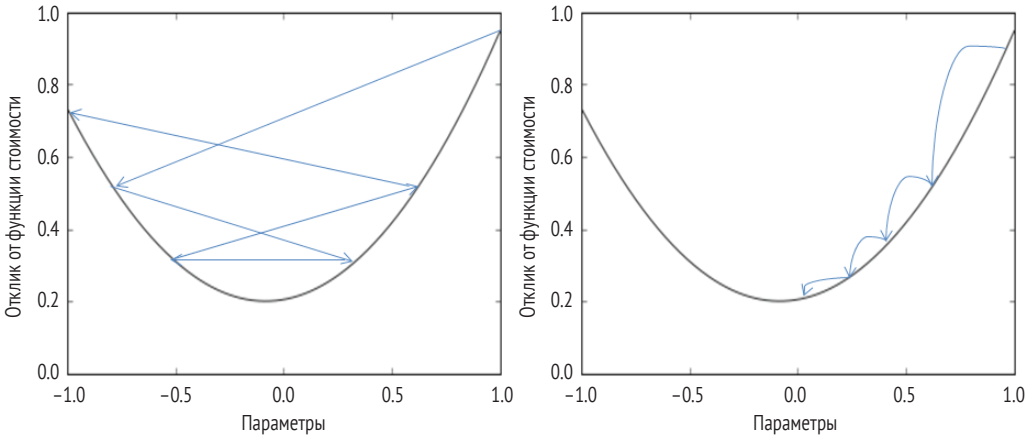
Используя образный пример, градиентный спуск напоминает прогулку по холмам с закрытыми глазами. Если вы хотите спуститься в самую низкую долину, не видя дороги, то можно продолжить движение, направляясь исключительно по пути, который, похоже, ведет вниз; надо чуть-чуть по нему пройти, затем остановиться, снова нащупать местность и затем продолжить путь в том направлении, которое, похоже, ведет вниз, и т. д., снова и снова. Если продолжить двигаться туда, где поверхность уходит вниз, то вы, наконец, прибудете в точку, откуда больше не сможете спуститься, потому что местность там будет плоской. Будем надеяться, что в той точке вы достигнете места назначения.

Используя этот подход, нам нужно выполнить следующие действия:

- принять решение об исходной точке. Обычно это достигается за счет первоначального случайного угадывания параметров функции (многократные

перезапуски обеспечат, что инициализация параметров не вынудит алгоритм достичь локального оптимума из-за неудачных начальных условий);

- уметь нащупывать местность, т. е. быть в состоянии различать, когда она уходит вниз. С математической точки зрения это означает, что вы должны суметь взять производную фактической параметризованной функции относительно целевой переменной, т. е. частную производную оптимизируемой функции стоимости. Отметим, что обычный градиентный спуск работает на всех данных, пытаясь оптимизировать предсказания, исходя из всех прецедентов одновременно;
- решить, как долго следует двигаться по направлению, продиктованному производной. С математической точки зрения это соответствует весу (обычно обозначаемому греческой буквой α), решающему, на какую величину надо изменять параметры на каждом шаге оптимизации. Этот аспект можно рассматривать как фактор обучения, потому что он показывает величину обучения на данных на каждом шаге оптимизации. Как и в случае с любым другим гиперпараметром, оптимальное значение α можно определить оценкой результативности на перекрестно-проверочном наборе;
- определить, когда следует остановиться, если имеется слишком незначительное улучшение функции стоимости относительно предыдущего шага. В этом смысле вы также должны суметь заметить, когда что-то пойдет не так, и вы движетесь в неправильном направлении, возможно, потому, что используете слишком большое значение α для обучения. Это в действительности связано с *инерцией*, т. е. скоростью, с которой алгоритм сходится к оптимуму. Это все равно, что бросить шар вниз по склону холма: он попросту будет перекатываться через маленькие вмятины на поверхности, однако если его скорость слишком высока, то в нужной точке он не остановится. Таким образом, если значение α будет установлено неправильно, то шар просто перепрыгнет через глобальный оптимум и продолжит сообщать об ошибках, которые подлежат минимизации, как показано на следующем ниже изображении на левой панели, когда процесс оптимизации заставит параметры перепрыгивать через разные значения, не достигая требуемой минимизации ошибки. Однако если значение α установлено правильно, то инерция естественным образом замедлится по мере приближения алгоритма к оптимуму, как показано на следующем ниже изображении в правой панели:



Чтобы точнее показать, что происходит с градиентным спуском, возьмем пример из линейной регрессии, чьи параметры оптимизируются благодаря такого рода процедуре. Мы начинаем с функции стоимости *J* на весовом векторе *w*:

$$J(w) = \frac{1}{2n}\sum(Xw - y)^2.$$

Матрично-векторное произведение *Xw* матрицы тренировочных данных *X* на вектор весовых коэффициентов *w* представляет собой предсказания из линейной модели, отклонения которых от отклика *y* возводятся в квадрат, затем суммируются и, наконец, делятся на число прецедентов *n*, умноженное на 2.

Первоначально вектор *w* может быть инстанцирован случайными числами, взятыми из стандартизированного нормального распределения с нулевым средним значением и единичным стандартным отклонением. (На самом деле инициализация может быть выполнена большим количеством различных способов, при этом все работают одинаково хорошо, аппроксимируя линейную регрессию, чья функция стоимости в форме чаши имеет уникальный минимум.) Это позволяет запускать наш алгоритм где-нибудь вдоль траектории оптимизации и эффективным образом ускоряет сходимость процесса. При оптимизации линейной регрессии инициализация не причиняет алгоритму каких-то неприятностей (в худшем случае неправильный старт попросту его замедлит). Напротив, когда мы используем градиентный спуск для оптимизации других алгоритмов машинного обучения, таких как нейронные сети, мы рискуем застрять из-за неправильной инициализации. Это произойдет, если, например, начальный вектор *w* будет заполнен нулевыми значениями (риск застрять на совершенно симметричной вершине, где никакое направление не может сразу же повести оптимизацию по лучшему пути, чем какое-либо другое). Это также может произойти с процессами оптимизации, которые имеют многочисленные локальные минимумы.

При наличии исходного случайного вектора коэффициентов *w* можно непосредственно вычислить функцию стоимости *J*(*w*) и определить начальное направление для каждого отдельного коэффициента, вычтя из каждого долю α (темпа обучения) частной производной функции стоимости, как эксплицируется в следующей формуле:

$$w_j = w_j - \alpha\frac{\delta}{\delta w}J(w).$$

Ее смысл можно лучше понять, если решить частную производную следующим образом:

$$w_j = w_j - \alpha\frac{1}{n}\sum(Xw - y)x_j.$$

Стоит отметить, что обновление выполняется на каждом отдельном коэффициенте (w_j) при наличии его вектора признаков x_j, но на основе всех предсказаний сразу (отсюда и суммирование).

После итерации по всем коэффициентам в *w* обновление коэффициентов будет завершено, и процесс оптимизации может быть перезапущен снова, вычислением частной производной и обновлением вектора *w*.

Интересное свойство этого процесса заключается в том, что величина обновления будет все меньше и меньше по мере приближения вектора *w* к оптималь-

ной конфигурации. Поэтому процесс может остановиться, когда вызванное в w изменение относительно предыдущей операции будет небольшим. Так или иначе, мы действительно имеем уменьшающуюся величину обновления, когда задан правильный размер темпа обучения α. По сути дела, если его значение слишком большое, то он может заставить процесс оптимизации отклониться и завершиться неудачей, приведя – в некоторых случаях – к полному расхождению процесса и невозможности сойтись к итоговому решению. В сущности, процесс оптимизации продемонстрирует тенденцию промахиваться по цели и в действительности еще больше от него удалится.

С другой стороны, слишком малое значение α не только будет двигать процесс оптимизации к своей цели слишком медленно, но, помимо этого, он легко может застрять где-нибудь в локальном минимуме. Это в особенности верно для более сложных алгоритмов, в частности которые используются в нейронных сетях. Что касается линейной регрессии и ее классификационного аналога, логистической регрессии, то поскольку в них кривая оптимизации имеет чашеобразную форму, подобную вогнутой кривой, она характеризуется единственным (глобальным) минимумом и полным отсутствием локальных минимумов.

В проиллюстрированной нами реализации алгоритма темп обучения α – это константа (градиентный спуск с фиксированным темпом обучения). Поскольку темп обучения α играет в схождении к оптимальному решению особо важную роль, для него были разработаны различные стратегии, в которых он вначале получает более крупное значение и сжимается по мере выполнения оптимизации. Мы обсудим такие подходы, когда займемся исследованием реализации алгоритма градиентного спуска в библиотеке Scikit-learn.

Стохастический градиентный спуск

Рассмотренная нами к настоящему моменту версия градиентного спуска называется полным пакетным градиентным спуском. Она работает за счет оптимизации ошибки всего набора данных и тем самым нуждается в его наличии в оперативной памяти. Внеядерные версии называются **стохастическим градиентным спуском** и мини-пакетным градиентным спуском.

Здесь формула остается точно такой же, за исключением обновления; за один раз обновляется всего один прецедент, тем самым позволяя оставлять базовые данные в своем хранилище и работать всего с одним наблюдением в оперативной памяти:

$$w_j = w_j - \alpha(x_j w - y)x_j.$$

Центральная идея этого алгоритма состоит в том, что если прецеденты выбираются случайным образом без особых смещений, то процесс оптимизации будет двигаться в среднем по направлению к целевой минимизации стоимости. Этим как раз и объясняется, почему мы обсудили вопрос, как из потока удалять любое упорядочение и генерировать его максимально случайным образом. В частности, если в примере с арендой велосипедов алгоритм стохастического градиентного спуска сначала обучается закономерностям начала сезона, потом сосредоточивается на лете, потом на осени и т. д., то в зависимости от сезона, когда процесс оптимизации будет остановлен, модель будет настроена на предсказании последнего сезона лучше остальных, потому что большинство недавних примеров взя-

то из него. В оптимизации методом стохастического градиентного спуска, когда данные **независимы и одинаково распределены** (independent and identically distributed, i.i.d.), сходимость к глобальному минимуму гарантируется. В практическом плане i.i.d. означает, что примеры не должны быть последовательно упорядочены или распределены, а предложены на вход алгоритма, будучи отобранными из данных случайным образом.

Реализация алгоритма SGD в библиотеке Scikit-learn

В библиотеке Scikit-learn можно найти большое количество онлайновых обучающихся алгоритмов. Не все алгоритмы машинного обучения имеют свой онлайновый аналог, но этот список интересен и постоянно растет. Что касается обучения с учителем, то имеющихся учеников можно разделить на классификаторы и регрессоры.

В качестве классификаторов можно упомянуть следующие[1]:

- `sklearn.naive_bayes.MultinomialNB`;
- `sklearn.naive_bayes.BernoulliNB`;
- `sklearn.linear_model.Perceptron`;
- `sklearn.linear_model.PassiveAggressiveClassifier`;
- `sklearn.linear_model.SGDClassifier`.

В качестве регрессоров имеются два варианта:

- `sklearn.linear_model.PassiveAggressiveRegressor`;
- `sklearn.linear_model.SGDRegressor`.

Они все могут обучаться инкрементно, обновляясь прецедент за прецедентом. Правда, из них только реализации `SGDClassifier` и `SGDRegressor` основаны на описанной ранее оптимизации методом стохастического градиентного спуска и являются основными темами этой главы. Ученики SGD оптимальны для всех крупномасштабных задач, поскольку их вычислительная сложность связана с $O(k*n*p)$, где k – это число прохождений по данным, n – число прецедентов и p – число признаков (естественно, ненулевых, если мы работаем с разреженными матрицами). Это полностью линейно-временные ученики, требующие большего времени в строгой пропорции с числом показанных примеров.

Другие онлайновые алгоритмы будут использоваться для проведения сравнительного тестирования. Более того, все алгоритмы пользуются одним общим программным интерфейсом, основанным на методе `partial_fit` для онлайнового и мини-пакетного обучения (в последнем случае при потоковой передаче больших порций, а не одиночного прецедента). Предоставление общего API делает все эти методы обучения взаимозаменяемыми в рамках задачи машинного обучения.

В противоположность методу подгонки `fit`, который использует все имеющиеся данные для их непосредственной оптимизации, метод `partial_fit` управляет частичной оптимизацией, основываясь на каждом отдельно передаваемом ему прецеденте. Даже если на вход метода `partial_fit` будет передан набор данных, то алгоритм не будет обрабатывать весь пакет данных, а только по одному элементу, приводя к по-настоящему линейной вычислительной сложности операций

[1] Полный перечень классификаторов и регрессоров можно найти на следующей странице библиотеки, посвященной методам обучения с учителем: http://scikit-learn.org/stable/supervised_learning.html. – *Прим. перев.*

обучения. Более того, после вызова метода `partial_fit` ученик может постоянно обновляться последующими вызовами `partial_fit`, что делает его идеальным для онлайнового обучения на непрерывных потоках данных.

При выполнении задачи классификации единственная трудность состоит в том, что при первой инициализации необходимо знать, скольким классам мы собираемся обучиться и как они промаркированы. Это можно сделать при помощи параметра `classes`, указав список меток с числовыми значениями. При этом требуется предварительно их обследовать, просмотрев данные в потоке, чтобы записать метки задачи, и принять к сведению их распределения, в случае если они не сбалансированы – т. е. когда некий класс по численности слишком велик либо слишком мал относительно других (правда, реализация в Scikit-learn предлагает способ автоматической обработки такой ситуации). Если целевая переменная числовая, то по-прежнему полезно знать ее распределение, но это не является необходимым условием для успешной работы ученика.

В библиотеке Scikit-learn есть две реализации алгоритма SGD – одна для задач классификации (`SGDClassifier`) и другая для регрессии (`SGDRegressor`). Классификационная реализация может справляться с многоклассовыми задачами с использованием схемы «**один против всех**» (one-vs-all, OVA). Эта стратегия подразумевает, что если дано k классов, то строятся k моделей, одна для каждого класса против всех прецедентов из других классов, следовательно, создавая k бинарных классификаций. Это приводит к созданию k наборов коэффициентов и k векторов предсказаний и их вероятностей. В конце, основываясь на эмитированной вероятности каждого класса по сравнению с другим, распознанным считается класс с самой высокой вероятностью. Если нужно получить фактические вероятности для мультиномиального распределения, то можно нормализовать результаты, разделив на их сумму. (Это то, что происходит в нейронной сети в слое с функцией мягкого максимума, в чем мы убедимся в последующих главах.)

Как классификационная, так и регрессионная реализации алгоритма SGD в библиотеке Scikit-learn характерны разными функциями потерь (функция стоимости – это ядро оптимизации на основе стохастического градиентного спуска).

Для классификации данных, выраженной параметром `loss`, задающим функцию потерь, можно опираться на следующие функции:

- `loss='log'`: классическая логистическая регрессия;
- `loss='hinge'`: мягкий зазор, т. е. линейная машина опорных векторов;
- `loss='modified_huber'`: сглаженная кусочно-линейная функция потерь.

Для задачи регрессии имеются три функции потерь:

- `loss='squared_loss'`: **обычный метод наименьших квадратов** (обычный МНК, ordinary least squares, OLS) для линейной регрессии;
- `loss='huber'`: функция потерь Хьюбера для регрессии, устойчивой к выбросам;
- `loss='epsilon_insensitive'`: линейная машина опорных векторов для регрессии.

Мы представим несколько примеров с использованием классических статистических функций потерь, т. е. логистической функции потерей и обычного МНК. Кусочно-линейная функция потерь и **машины опорных векторов** (SVM) будут обсуждаться в следующей главе, так как необходима подробная вводная информация об их функционировании.

В качестве напоминания (чтобы не пришлось обращаться за консультацией к другой книге по машинному обучению), если задать функцию регрессии как h, а ее предсказания выразить как $h(X)$, потому что X – это матрица признаков, тогда приемлемой формулой будет следующая:

$$y \approx h(X) = \beta X + \beta_0.$$

Следовательно, подлежащая минимизации функция стоимости по обычному методу МНК выглядит так:

$$\frac{1}{2n} * \sum (h(X) - y)^2.$$

В логистической регрессии, в условиях преобразования бинарного исхода 0/1 в отношение шансов, где y – это вероятность положительного исхода, формула будет следующей:

$$y \approx h(X) = \frac{1}{1 + e^{\beta X + \beta_0}}.$$

Логистическая функция стоимости, следовательно, определяется следующим образом:

$$-\frac{1}{n} * \sum [y * \ln(h(X)) + (1 - y) * \ln(1 - h(X))].$$

Определение параметров обучения алгоритма SGD

Чтобы задать параметры алгоритма SGD в библиотеке Scikit-learn, как в задачах классификации, так и задачах регрессии (чтобы они были применимы одновременно для алгоритмов `SGDClassifier` и `SGDRegressor`), необходимо выяснить, как вести себя с несколькими важными параметрами, которые требуются для правильного обучения, когда невозможно оценить все данные сразу.

Первый параметр `n_iter` определяет число итераций по данным. Исходно его значение равно 5, однако опытным путем было показано, что он должен быть подстроен под ученика, с тем чтобы тот, при условии что другие параметры заданы по умолчанию, видел порядка `10^6` примеров; поэтому хорошим решением будет его установка в `n_iter = np.ceil (10 ** 6 / n)`, где `n` – это число прецедентов. Стоит отметить, что `n_iter` работает только с наборами данных, расположенными в оперативной памяти, и поэтому он действует только тогда, когда вы используете метод `fit` вместо `partial_fit`. В действительности метод `partial_fit` выполнит повторный итеративный обход тех же данных, если их повторно «прокачать» в своей процедуре, при этом нужное число итераций повторных прокачек должно проверяться по ходу самой обучающей процедуры, так как оно зависит от типа данных. В следующей главе мы проиллюстрируем гиперпараметрическую оптимизацию и обсудим вопрос надлежащего числа прохождений по данным.

Во время выполнения мини-макетного обучения имеет смысл повторно перемешивать данные после каждого полного прохождения по всем данным.

Параметр `shuffle` требуется на случай, если вы хотите перемешать свои данные. Он касается мини-пакетов, находящихся в оперативной памяти, и не ак-

туален для внеядерного упорядочения данных. Он также работает с методом `partial_fit`, но его эффект в таком случае сильно ограничен. Всегда устанавливайте его в `True`, за исключением тех случаев, когда данные передаются порциями; что касается перемешивания данных вне ядра, то поступайте так, как мы описали ранее.

Параметр `warm_start` также работает с методом `fit`, потому что он помнит предыдущие коэффициенты подгонки (но не темп обучения, в случае если он был динамически изменен). Если используется метод `partial_fit`, то алгоритм запомнит ранее усвоенные коэффициенты и состояние схемы темпа обучения.

Параметр `average` инициирует вычислительный прием, который на определенном прецеденте запускает усреднение новых коэффициентов с предыдущими, позволяя быстрее достигать схождения. Его можно установить в `True` либо в целочисленное значение, тем самым отметив, с какого прецедента он начнет усреднение.

Наконец, но не в последнюю очередь, имеются параметр темпа обучения `learning_rate` и связанные с ним параметры `eta0` и `power_t`. Параметр `learning_rate` означает, насколько каждый наблюдаемый прецедент влияет на процесс оптимизации. При описании алгоритма SGD с теоретической точки зрения мы представили темп обучения как константу, которая может реплицироваться путем установки `learning_rate='constant'`.

Однако есть и другие варианты, позволяющие η (греч. буква «эта», обозначающая в Scikit-learn темп обучения, определяемый во время t) постепенно уменьшаться. В задаче классификации предлагается решение с темпом обучения `learning_rate='optimal'`, который выражается следующей ниже формулой:

$$\eta^t = \frac{1}{\alpha_{t0} + \alpha_t}.$$

Здесь t – это временной шаг, задаваемый числом прецедентов, умноженных на итерации, и $t0$ – выбираемое эвристическим способом значение на основании исследований Леона Боту (Léon Bottou), чья версия *SVM на основе стохастических градиентных аппроксимаций* в большой степени повлияла на реализацию алгоритма SGD в библиотеке Scikit-learn (http://leon.bottou.org/projects/sgd). Явное преимущество такой стратегии обучения состоит в том, что обучение уменьшается вместе с ростом числа наблюдаемых примеров, не допуская внезапных возмущений в процессе оптимизации, задаваемых необычными значениями. Разумеется, эта стратегия тоже предоставлена в готовом к использованию виде, имея в виду, что у вас не так уж много вариантов для работы с ней. В регрессии предлагаемое затухание обучения выражается следующей ниже формулой, что соответствует темпу обучения `learning_rate='invscaling'`:

$$\eta^t = \frac{eta_0}{t^{power\_t}}.$$

Здесь `eta0` и `power_t` – это гиперпараметры (они первоначально установлены в 0 и 0.5) и подлежат оптимизации оптимизационным поиском. Стоит отметить, что, используя темп обучения `invscaling`, алгоритм SGD начинает с более низкого, менее оптимального темпа обучения, и он будет уменьшаться медленнее, при этом являясь чуть более адаптируемым во время обучения.

Управление признаками на потоках данных

Потоки данных ставят задачу, которую невозможно решить теми же средствами, когда работают с полным набором данных в оперативной памяти. Правильный и оптимальный подход к подаче данных во внеядерный обучающийся алгоритм SGD требует сначала обследовать данные (к примеру, взяв несколько пробных прецедентов из исходного файла) и узнать тип поступающих данных.

Мы различаем следующие типы данных:

- количественные значения;
- категориальные значения, представленные целыми числами;
- неструктурированные категориальные значения, выраженные в текстовой форме.

В случае количественных данных их можно сразу передать ученику SGD, за исключением того, что этот алгоритм довольно чувствителен к шкалированию признаков; иными словами, необходимо привести все количественные признаки к одному и тому же интервалу значений, иначе процесс обучения не сойдется легко и правильно. Возможные стратегии шкалирования конвертируют все значения в интервалы [0,1], [−1,1] или стандартизируют переменную, центрируя ее среднее на нуле и конвертируя ее дисперсию в единицу. У нас нет каких-то особых предложений по поводу выбора стратегии шкалирования, но конвертирование в интервал [0,1] работает особенно хорошо, в случае если речь идет о разреженной матрице, и при этом большинство значений нулевые.

Что касается обучения в оперативной памяти, то во время преобразования переменных на тренировочном наборе следует обращать внимание на используемые значения (в основном нужно получить минимальные, максимальные, средние значения и стандартные отклонения каждого признака) и применить их повторно в тестовом наборе для достижения согласованных результатов.

Учитывая, что данные передаются потоком и отсутствует возможность закачать все данные в оперативную память, необходимо их подсчитать, пройдя по всем данным либо как минимум их части (выборка всегда является эффективным решением). Ситуация, когда вы работаете со скоротечным эфемерным потоком (потоком, который невозможно реплицировать), ставит еще более сложные задачи; фактически необходимо постоянно вести учет значений, которые продолжают поступать.

Если выборка просто сводится к расчету необходимой статистики на порции из n прецедентов (при допущении, что поток не имеет какого-то определенного порядка следования), то вычисление статистик на лету требует регистрировать соответствующие количественные показатели.

Для получения минимума и максимума по каждому количественному признаку следует хранить отдельную переменную. Начиная с самого первого значения, которое берется как исходный минимум и максимум, затем следует сопоставлять каждое новое значение, которое поступает из потока, с ранее зарегистрированными минимальным и максимальным значениями. Если новый прецедент выходит за пределы предыдущего интервала значений, то переменная соответствующим образом обновляется.

Среднее арифметическое значение тоже не вызывает каких-то особых проблем, потому что требуется хранить только сумму просмотренных значений и ко-

личество прецедентов. Что касается дисперсии, то напомним ее хрестоматийную формулу:

$$\sigma^2 = \frac{1}{n}\sum(x-\mu)^2.$$

Следует отметить, что здесь требуется знать среднее значение μ, которое также извлекается из потока инкрементно. Однако формула дисперсии для потоковой передачи может быть эксплицирована следующим образом:

$$\sigma^2 = \frac{1}{n}\left(\sum x^2 - \frac{(\sum x)^2}{n}\right).$$

Поскольку число прецедентов *n* и сумма значений *x* уже регистрируются, требуется хранить всего одну дополнительную переменную – сумму значений *x* в квадрате, и в результате будут получены все ингредиенты для рецепта.

В качестве примера возьмем набор данных Bike-sharing об аренде велосипедов, вычислим текущее среднее значение, стандартное отклонение и размах значений и построим график, показывающий, как эти статистики изменялись по мере того, как данные потоком передавались из диска:

```
In:
import os, csv

local_path = os.getcwd().decode('cp1251')
source = 'data\\bikesharing\\hour.csv'
SEP = str(',')

running_mean = list()
running_std = list()

with open(local_path+'\\'+source, 'rb') as R:
    iterator = csv.DictReader(R, delimiter=SEP)
    x = 0.0
    x_squared = 0.0
    for n, row in enumerate(iterator):
        temp = float(row['temp'])
        if n == 0:
            max_x, min_x = temp, temp
        else:
            max_x, min_x = max(temp, max_x),min(temp, min_x)
        x += temp
        x_squared += temp**2
        running_mean.append(x / (n+1))
        running_std.append(((x_squared - (x**2)/(n+1))/(n+1))**0.5)
        # заглушка для ОБРАБОТКИ ДАННЫХ
        # заглушка для МАШИННОГО ОБУЧЕНИЯ
        pass
    print('Всего строк: {0}'.format(n+1))
    print('Признак \'temp\': среднее=%0.3f, макс=%0.3f, мин=%0.3f, стд.откл.=%0.3f'
          % (running_mean[-1], max_x, min_x, running_std[-1]))

Out:
Всего строк: 17379
Признак 'temp': среднее=0.497, макс=1.000, мин=0.020, стд.откл.=0.193
```

Через несколько мгновений данные пойдут потоком из источника, и ключевые цифры относительно признака `temp` начнут регистрироваться по мере того, как вычисляется текущая оценка среднего и стандартного отклонений и сохраняется в двух отдельных списках.

Выведя на график списковые значения, можно узнать, насколько оценки колебались относительно итоговых цифр, и получить представление о том, сколько требуется прецедентов, прежде чем получить стабильную оценку среднего и стандартного отклонений[1]:

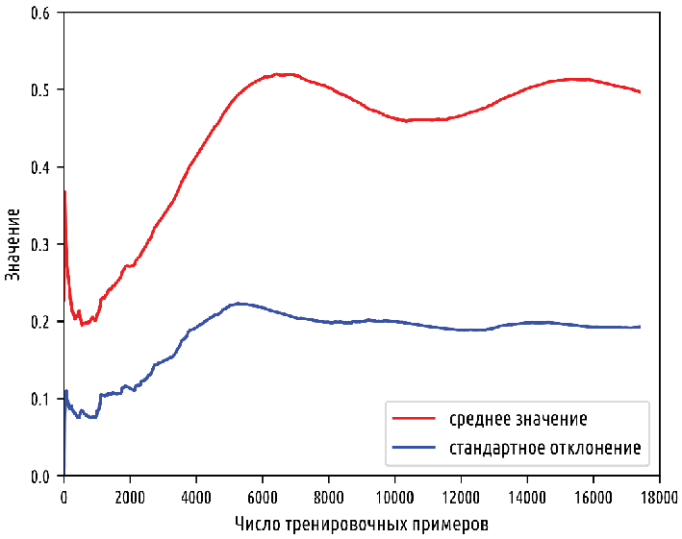
```
In:
# См. инициализацию графических настроек в начале блокнота
plt.plot(running_mean,'r-', label=u'среднее значение')
plt.plot(running_std,'b-', label=u'стандартное отклонение')
plt.xlim(0,18000)
plt.ylim(0.0,0.6)
plt.xlabel(u'Число тренировочных примеров')
plt.ylabel(u'Значение')
plt.legend(loc='lower right', numpoints= 1)
plt.show()
```

Если предварительно обработать исходный набор данных об аренде велосипедов, то получится график, на котором в данных отчетливо видна тенденция (благодаря упорядочению во времени, потому что температура естественно меняется в зависимости от времен года):

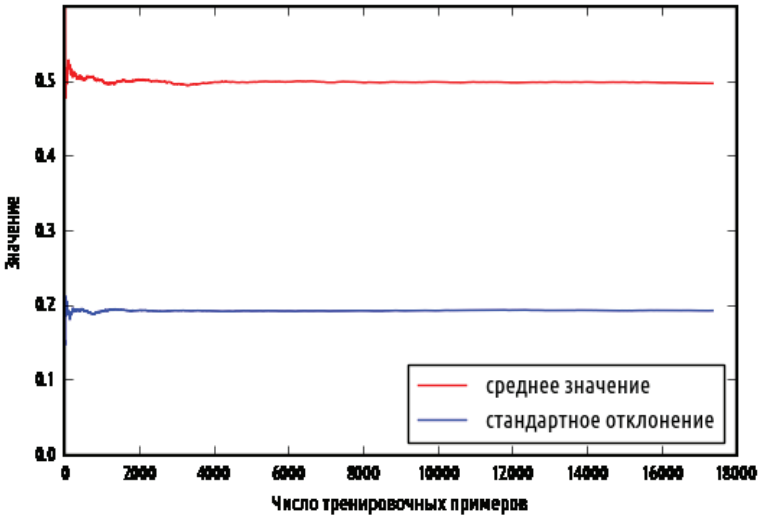
---

[1] В начале каждого блокнота, где присутствуют графики, имеется следующая ниже ячейка с графическими настройками для графиков:

```
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import rcParams
rcParams['font.family']     = 'sans-serif'
rcParams['font.sans-serif'] = ['Ubuntu Condensed']
rcParams['figure.figsize']  = (5, 4)
rcParams['legend.fontsize'] = 10
rcParams['xtick.labelsize'] = 9
rcParams['ytick.labelsize'] = 9
def saveplot(dest):
    plt.tight_layout()
    plt.savefig('images/' + dest, dpi=600)
```



И наоборот, если в качестве источника использовать перемешанную версию набора данных в файле `shuffled_hour.csv`, то мы, скорее всего, получим пару намного более стабильных и быстро сходящихся оценок. Следовательно, мы приобретем приблизительную, но более надежную оценку среднего значения и стандартного отклонения, регистрируя меньшее число прецедентов из потока:



Расхождение в двух диаграммах напоминает нам о важности рандомизации порядка следования наблюдений. Тенденции в данных могут оказывать сильное влияние даже на усвоение простых описательных статистик; следовательно, мы должны уделять этому особое внимание во время обучения сложных моделей алгоритмом SGD.

Описание целевой переменной

В дополнение перед началом следует также проанализировать целевую переменную. Мы, по сути дела, должны быть уверены в том, какие она принимает значения, является ли категориальной, и выяснить ее несбалансированность, когда речь идет о классах, или скошенность распределения, когда она представлена числом.

Если в процессе обучения извлекается числовой отклик, то можно принять ту же самую стратегию, показанную ранее для признаков, тогда как для классов будет достаточно словаря Python, который будет регистрировать количество классов (ключи) и их частоты (значения).

В качестве примера скачаем набор данных **Forest Covertype** с данными лесного покрова для задачи классификации.

Чтобы быстро скачать и подготовить данные, мы воспользуемся функцией `gzip_from_UCI` в том виде, в каком она определена в разделе «*Наборы данных для реальных дел*» данной главы:

```
In:
UCI_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/covtype.data.gz'
gzip_from_UCI(UCI_url, dest='data\\')
```

В случае проблем при выполнении программного кода либо если вы предпочитаете подготовить файл самостоятельно, просто перейдите на веб-сайт UCI, скачайте набор данных и распакуйте его в каталог, где работает Python: https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/.

Когда данные будут на диске, можно просканировать все 581 012 прецедентов, конвертируя последнее значение каждой строки, которое представляет оцениваемый класс, в соответствующий ему тип лесного покрова:

```
In:
import os, csv

local_path = os.getcwd().decode('cp1251')
source = 'data\\covtype.data'
SEP = str(',')

forest_type = {1:"ель/хвойное", 2:"сосна красная",
               3:"сосна желтая", 4:"тополь/ива",
               5:"осина", 6:"пихта", 7:"криволесье"}
forest_type_count = {value:0 for value in forest_type.values()}
forest_type_count['другое'] = 0

lodgepole_pine = 0
spruce = 0
proportions = list()

with open(local_path+'\\'+source, 'rb') as R:
    iterator = csv.reader(R, delimiter=SEP)
```

```
    for n, row in enumerate(iterator):
        response = int(row[-1]) # последнее значение - это отклик
        try:
            forest_type_count[forest_type[response]] +=1
            if response == 1:
                spruce += 1
            elif response == 2:
                lodgepole_pine +=1
            if n % 10000 == 0:
                proportions.append([spruce/float(n+1),
                                    lodgepole_pine/float(n+1)])
        except:
            forest_type_count['другое'] += 1

    print('Всего строк: %i' % (n+1))
    print('Частота классов:')
    for ftype, freq in sorted([(t,v)
                               for t,v in forest_type_count.iteritems()],
                              key = lambda x: x[1], reverse=True):
        print("%-18s: %6i %04.1f%%" % (ftype, freq, freq*100/float(n+1)))
```

```
Out:
Всего строк: 581012
Частота классов:
сосна красная     : 283301 48.8%
ель/хвойное       : 211840 36.5%
сосна желтая      :  35754 06.2%
криволесье        :  20510 03.5%
пихта             :  17367 03.0%
осина             :   9493 01.6%
тополь/ива        :   2747 00.5%
другое            :      0 00.0%
```

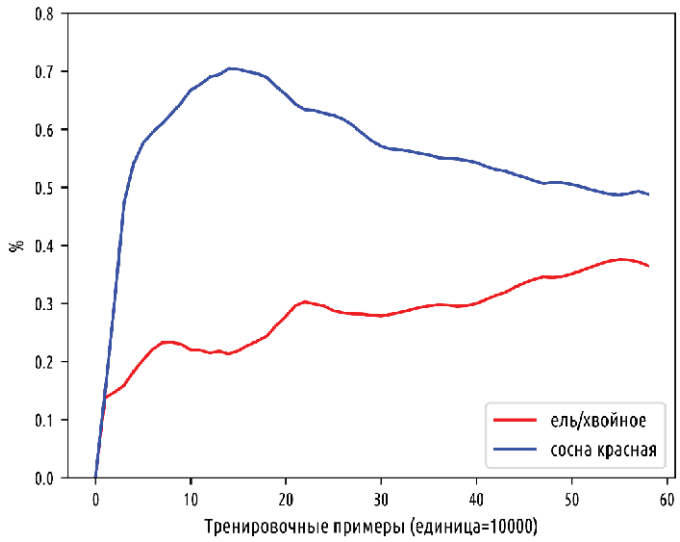
Результат показывает, что два класса, «сосна красная» и «ель/хвойное», составляют бóльшую часть наблюдений. Если примеры надлежаще перемешиваются в потоке, то алгоритм SGD соответствующим образом извлечет правильное априорное распределение и, следовательно, скорректирует свою эмиссию вероятности (апостериорную вероятность).

Если вопреки нашему нынешнему случаю целью является не точность классификации, а увеличение **площади под ROC-кривой** (AUC) либо отметка F1 (функции ошибки, которые могут использоваться для оценивания модели), то для получения обзорной информации можно непосредственно проконсультироваться с документацией по библиотеке Scikit-learn на http://scikit-learn.org/stable/modules/model_evaluation.html относительно классификационной модели, натренированной на несбалансированных данных, и затем предоставленная информация поможет вам сбалансировать веса с использованием параметра `class_weight` при задании классификатора `SGDClassifier` либо параметра `sample_weight` при частичной подгонке модели. Оба параметра изменяют влияние наблюдаемого прецедента путем присвоения положительного или отрицательного весового коэффициента. В обоих случаях оперирование этими двумя параметрами может изменить априорное распределение. Взвешивание классов и прецедентов будет обсуждаться в следующей главе.

Графики характеристической кривой, или графики ROC-кривой, иногда также кривой ошибок, от англ. Receiver Operator Characteristic (ROC), т. е. график рабочей характеристики получателя, – это широко используемый инструмент отбора моделей для классификации данных. Диагональ ROC-графика можно интерпретировать как случайное гадание, и классификационные модели, которые попадают ниже диагонали, считаются хуже случайного гадания. Основываясь на ROC-кривой, затем можно вычислить показатель так называемой площади под кривой AUC, от англ. area under the curve, чтобы охарактеризовать результативность классификационной модели.

Прежде чем перейти к тренировке модели и работе с классами, можно проверить, имеет ли соотношение классов всегда систематический характер, с тем чтобы подать на вход алгоритма SGD правильную априорную вероятность:

```
In:
import numpy as np

proportions = np.array(proportions)

plt.plot(proportions[:,0],'r-', label=u'ель/хвойное')
plt.plot(proportions[:,1],'b-', label=u'сосна красная')
plt.ylim(0.0,0.8)
plt.xlabel(u'Тренировочные примеры (единица=10000)')
plt.ylabel('%')
plt.legend(loc='lower right', numpoints= 1)
plt.show()
```



На приведенном выше графике заметно, как процент примеров изменяется по мере того, как мы продвигаемся вперед, передавая данные потоком в существующем порядке следования. В этом случае данные действительно необходимо перемешать, если мы на самом деле хотим, чтобы стохастический онлайновый алгоритм правильно обучился на данных.

Собственно говоря, соотношения классов можно изменить; этот набор данных имеет некую упорядоченность, возможно, географическую, которую следует исправить путем перемешивания данных, иначе мы рискуем переоценить либо недооценивать некоторые классы относительно других.

Хэширование признаков

Если среди признаков имеются категории (представленные числовыми значениями или оставленные в текстовой форме), то задача может стать немного сложнее. В пакетном обучении категории обычно записываются с использованием прямого унитарного кодирования, в результате чего получается столько новых двоичных категорий, сколько имеется категорий. К сожалению, в потоке невозможно узнать заранее, с каким количеством категорий имеешь дело, и даже в результате выборки нельзя быть уверенным в их числе, потому что редко встречающиеся категории могут появиться в потоке существенно позже либо потребуют найти слишком большую выборку. Вам придется сначала прокачать все данные и зарегистрировать каждую появившуюся категорию. Так или иначе, потоки могут быть эфемерными, и при этом число классов иногда может быть настолько большим, что их не получится сохранить в оперативной памяти. Онлайновые рекламные данные являются таким примером в силу своих высоких объемов, которые трудно сохранить в памяти, и потому, что по потоку нельзя пройти более одного раза. Более того, рекламные данные в значительной степени варьируются, а их признаки постоянно изменяются.

Прямое унитарное кодирование, от англ. one-hot encoding (OHE), – это представление конечного множества значений признака в виде двоичного кода фиксированной длины, содержащего только одну 1, т. е. где *i*-е значение признака равно 1, а все остальные равны 0. При обратном кодировании имеется только один 0. Длина кода определяется количеством кодируемых значений признака, или объектов.

Работа с текстами делает проблему значительно еще более отчетливой, потому что невозможно предвидеть, какое слово может быть частью анализируемого текста. В модели «мешка слов», где в каждом тексте подсчитываются встретившиеся в нем слова и их частотное значение вставляется в соответствующий каждому слову элемент вектора признаков, необходимо суметь заранее сопоставить каждому слову номер индекса. И даже если с этим удастся справиться, всегда придется разбираться с ситуацией, когда неожиданно во время тестирования либо во время использования предиктора в эксплуатационной среде появится незнакомое слово (которому поэтому индексный номер никогда заранее не сопоставлен). Косвенно следует также добавить, что, будучи отражением разговорного языка, совсем не являются необычными тематические словари, состоящие из сотен тысяч или даже миллионов разных терминов.

Резюмируя сказанное, если вам удастся заранее узнать классы в признаках, то вы сможете справиться с ними при помощи прямого кодировщика из библиотеки Scikit-learn (http://Scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html). Мы в действительности не будем здесь иллюстрировать его работу, но, в принципе, этот подход нисколько не отличается от того, который применялся бы во время работы с пакетным обучением. Однако мы хотим проиллюстрировать ситуацию, когда применить прямое кодирование действительно невозможно.

Существует опирающееся на хэш-функцию решение под названием «**хэширование признаков**», или хэш-трюк (hashing trick), которое может работать как с текстом, так и с категориальными переменными в целочисленной либо строковой форме. Это решение также работает с категориальными переменными, смешанными с числовыми значениями из количественных признаков. Ключевая проблема с прямым унитарным кодированием состоит в том, что оно присваивает позицию значению в векторе признаков, только после того как его признаку сопоставлена эта позиция. Хэширование признаков, с другой стороны, способно недвусмысленно сопоставить значению его позицию без априорной необходимости выполнить оценку признака, потому что эта операция задействует ключевое свойство хэш-функции – детерминированное преобразование величины либо строкового значения в целочисленное значение.

Поэтому единственная необходимая подготовительная работа перед применением хэширования признаков состоит в создании разреженного вектора, который будет достаточно крупным, чтобы представлять вычислительную сложность данных (потенциально содержащий от *2**19* до *2**30* элементов в зависимости от доступной оперативной памяти, шинной архитектуры компьютера и типа используемой хэш-функции). Если будет обрабатываться текст, то, помимо этого, понадобится лексемизатор, т. е. функция, которая разобьет текст на отдельные слова и удалит пунктуацию.

Простой миниатюрный пример внесет ясность. Мы воспользуемся двумя специализированными функциями из библиотеки Scikit-learn: преобразователем `HashingVectorizer`, опирающимся на хэширование признаков, который обрабатывает текстовые данные, и еще одним преобразователем `FeatureHasher`, который заточен на преобразовании строки данных в виде словаря Python в разреженный вектор признаков.

В качестве первого примера превратим фразу в вектор:

```
In:
from sklearn.feature_extraction.text import HashingVectorizer

h = HashingVectorizer(n_features=1000, binary=True, norm=None)

# Перевод: простой миниатюрный пример внесет ясность в то, как он работает
sentence = 'A simple toy example will make clear how it works.'
sparse_vector = h.transform([sentence])

print(sparse_vector)

Out:
  (0, 61)     1.0
  (0, 271)    1.0
  (0, 287)    1.0
  (0, 452)    1.0
  (0, 462)    1.0
  (0, 539)    1.0
  (0, 605)    1.0
  (0, 726)    1.0
  (0, 918)    1.0
```

Итоговый вектор имеет единичные значения только с определенным индексом, указывая на связь между маркером во фразе (словом) и определенной позицией в векторе. К сожалению, эту связь нельзя инвертировать, если, конечно, не сопоставить значение хэш-функции каждому маркеру во внешнем словаре Python. Хотя такое сопоставление возможно, оно в действительности будет потреблять много памяти, потому что словари бывают большими, в диапазоне миллионов элементов и даже больше, в зависимости от языка и тематической направленности. На самом деле нам не нужно вести такой учет, потому что хэш-функции всегда гарантированно из одинакового маркера продуцируют одинаковый индексный указатель.

Реальную проблему для хэширования признаков в конечном счете представляет возможность конфликта, который случается, когда два разных маркера ассоциируются с одной и той же позицией. Это редкое, но возможное явление возникает во время работы с большими словарями. С другой стороны, в скомпонованной из миллионов коэффициентов модели очень немногие из них имеют влияние. Следовательно, если конфликт произойдет, то весьма вероятно, что он будет связан с двумя неважными маркерами. При использовании хэширования признаков вероятность на вашей стороне, потому что весьма маловероятно, что в достаточно большом выходном векторе (например, с числом элементов свыше *2^24*) конфликты, хотя и являясь всегда возможными, будут содержать важные элементы модели.

Хэширование признаков может применяться к нормальным векторам признаков, в особенности когда имеются категориальные переменные. Ниже приведен пример с преобразователем `FeatureHasher`:

```
In:
from sklearn.feature_extraction import FeatureHasher

h = FeatureHasher(n_features=1000, non_negative=True)
example_row = {'numeric feature':3,         # числовой признак
               'another numeric feature':2, # еще один числовой признак
               'Categorical feature = 3':1, # категориальный признак
               'f1*f2*f3':1*2*3}

print(example_row))

Out:
{'another numeric feature': 2, 'f1*f2*f3': 6, 'numeric feature': 3, 'Categorical feature = 3': 1}
```

Если словарь Python для числовых значений будет содержать имена признаков и для любой категориальной переменной – сочетание имени признака и значения, то значения словаря будут отображаться посредством хэшированного индексного номера ключей, создавая прямокодированный вектор признаков, готовый к тому, чтобы быть заученным алгоритмом SGD:

```
In:
print(h.transform([example_row]))

Out:
  (0, 16)  2.0
  (0, 373) 1.0
  (0, 884) 6.0
  (0, 945) 3.0
```

Другие элементарные преобразования

После того как пример взят из хранилища данных, помимо превращения категориальных переменных в числовые, к нему можно применить еще одно преобразование, с тем чтобы увеличить прогнозирующую способность обучающегося алгоритма. Преобразования могут применяться к признакам посредством функции (путем применения квадратного корня, логарифма или другой функции преобразования) либо посредством операций на группах признаков.

В следующей главе мы предложим подробные примеры с полиномиальным разложением и методами случайного неизбирательного отбора признаков (kitchen-sink). В настоящей главе мы, забегая вперед, создадим квадратичные признаки путем вложенных итераций. Квадратичные признаки обычно создаются при вычислении полиномиальных разложений, и их цель состоит в том, чтобы уловить, каким образом прогнозные признаки взаимодействуют; они могут неожиданным образом повлиять на отклик в целевой переменной.

В качестве примера, дающего интуитивно понятное разъяснение того, почему квадратичные признаки могут иметь важное значение в моделировании целевого отклика, объясним случай воздействия двух лекарств на пациента. По сути дела, может оказаться, что оба лекарства в большей или меньшей степени являются эффективными в лечении болезни, против которой мы боремся. Как бы там ни было, но, когда пациент принимает оба этих препарата, состоящих из разных компонентов, вместе, они проявляют тенденцию аннулировать влияние друг друга. В данном случае, хотя оба лекарства эффективны, вместе они ввиду своего отрицательного взаимодействия не работают вообще.

В этом смысле взаимодействия между признаками можно найти среди широкого круга признаков, и не только в медицине. При этом критически важно выяснить наиболее значимый из них, чтобы модель работала лучше при выполнении предсказания своей целевой переменной. Если неизвестно, что определенные признаки взаимодействуют относительно задачи, то единственный выбор состоит в том, чтобы все их систематически протестировать и позволить модели оставить те признаки, которые работают лучше.

В следующем ниже простом примере представим, что вектор *v* только что в результате потоковой передачи был помещен в оперативную память с целью его заучивания, затем преобразован в другой вектор *vv*, в котором исходные признаки вектора *v* дополнены результатами их мультипликативных взаимодействий (каждый признак помножен один раз на все остальные). При наличии большего числа признаков в обучающийся алгоритм поступит вектор *vv* вместо исходного вектора *v* с целью достичь наилучшей аппроксимации данных:

```
In:
import numpy as np

v = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
vv = np.hstack((v, [v[i]*v[j]
                    for i in range(len(v))
                    for j in range(i+1, len(v))]))

Out:
[ 1  2  3  4  5  6  7  8  9 10  2  3  4  5  6  7  8  9 10  6  8 10 12 14 16 18 20 12 15 18 21
24 27 30 20 24 28 32 36 40 30 35 40 45 50 42 48 54 60 56 63 70 72 80 90]
```

Аналогичные или еще более сложные преобразования могут генерироваться на лету по ходу потоковой передачи примеров в обучающийся алгоритм, пользуясь тем, что пакет данных небольшой (иногда сводимый к одиночным примерам), и расширение числа признаков такого небольшого количества примеров может быть реально достигнуто в оперативной памяти. В следующей главе мы рассмотрим еще больше примеров таких преобразований и их успешной интеграции в конвейер обучения.

Тестирование и перекрестная проверка в потоке

Мы воздержались от показа полных примеров тренировки после ознакомления с алгоритмом SGD, потому что необходимо сначала познакомиться с тем, каким образом в потоке выполняются тестирование и перекрестная проверка (контроль). При использовании пакетного обучения тестирование и перекрестная проверка являются вопросом рандомизации порядка следования наблюдений, нарезки набора данных на блоки и взятия конкретного блока в качестве тестового набора либо систематического взятия по очереди одного из блоков для тестирования способностей алгоритма к обучению.

Потоки не могут оставаться в оперативной памяти, поэтому, исходя из того, что следующие прецеденты будут уже рандомизированы, лучшее решение состоит в том, чтобы брать проверочные прецеденты после некоторого времени с начала поступления данных потока либо систематически использовать конкретный воспроизводимый шаблон в потоке данных.

Вневыборочный подход на части потока фактически сопоставим с тестовой выборкой и может быть успешно реализован, только если заранее известна длина потока. Для непрерывных потоков этот подход по-прежнему возможен, но означает безусловную остановку процесса обучения, как только начинаются тестовые прецеденты. Этот метод называется стратегией с периодическим отложением данных (holdout) после *n* прецедентов.

Проверки эффективности прогнозирования модели обычно проводятся путем разбиения заданного набора данных на *внутривыборочный* (in-sample) период, который используется для первоначальной оценки параметров и отбора модели, и *вневыборочный* (out-of-sample) период, используемый для оценки эффективности прогнозирования на ранее не встречавшихся данных.

Перекрестно-проверочный тип подхода возможен при использовании систематического и воспроизводимого отбора проверочных прецедентов. После определения стартового буфера прецедент отбирается для проверки с интервалом *n*. Такой прецедент используется не для тренировки, а исключительно для тестирования. Этот метод называется стратегией с отложением данных с периодичностью *n*.

Поскольку проверка выполняется на основе одиночного прецедента, вычисляется глобальная мера результативности, усредняющая все меры ошибки, собранные на данный момент в пределах одного и того же прохождения по данным либо в оконном режиме с использованием самого последнего набора из *k* мер, где *k* – это число тестов, которое, как вы полагаете, является обоснованно репрезентативным.

Собственно говоря, во время первого прохождения обучающийся алгоритм в действительности не знаком ни с одним прецедентом. Поэтому целесообразно

тестировать алгоритм по мере получения им случаев для обучения, проверяя его отклик на поступивший случай, перед тем как ему обучиться. Этот подход называется прогрессивной проверкой.

Применение алгоритма SGD в деле

В заключение настоящей главы мы реализуем два примера: один – для задачи классификации на основе данных Covertype о лесном покрове, и другой – для регрессии на основе набора данных Bike-sharing об аренде велосипедов. Мы увидим, как реализовать на практике приведенные выше соображения об отклике и распределениях признаков и для каждой из этих задач применить оптимальную стратегию проверки.

Если начать с задачи классификации, то следует отметить два аспекта. Говоря о мультиклассовой задаче, мы в первую очередь заметили, что в базе данных и распределении классов по потоку прецедентов существует некая упорядоченность. В качестве исходного шага мы перемешаем данные при помощи функции `ram_shuffle`, определенной в разделе «*Особое внимание упорядочению прецедентов*» настоящей главы:

```
In:
import os

local_path = os.getcwd().decode('cp1251')
source = 'data\\covtype.data'

ram_shuffle(filename_in=local_path+'\\'+source,
            filename_out=local_path+'\\data\\shuffled_covtype.data',
            header=False)
```

Поскольку мы распаковываем строки в оперативной памяти и перемешиваем их без особого использования дискового пространства, можно быстро получить новый рабочий файл. В следующем фрагменте кода мы тренируем классификатор `SGDClassifier` с логистической функцией потерь (эквивалентной логистической регрессии), с тем чтобы он задействовал наши предыдущие знания о присутствующих в наборе данных классах. Список `forest_type` содержит все коды классов, и он передается каждый раз (хотя всего одного, первого раза было бы достаточно) в метод `partial_fit` ученика SGD.

Для целей проверки мы определяем «холодный» запуск из 200 000 наблюдавшихся случаев. На каждом десятом прецеденте один будет изыматься из процесса тренировки и использоваться для проверки. Эта схема допускает воспроизводимость, даже если мы собираемся выполнять многократные прохождения по данным; при каждом прохождении те же самые прецеденты будут откладываться в сторону в качестве вневыборочного теста, что позволяет проверить эффект многократных прохождений по тем же данным, создав кривую проверки.

Схема с отложением данных также сопровождается прогрессивной проверкой. Поэтому каждый случай после «холодного» запуска оценивается до того, как подаваться на вход этапа тренировки. Хотя прогрессивная проверка обеспечивает интересную обратную связь, такой способ будет работать только для первого прохождения; на деле после первоначального прохождения все наблюдения (кроме тех, которые отложены согласно схеме отложенной выборки) станут внутривы-

борочными прецедентами. В нашем примере мы собираемся сделать всего одно прохождение.

Напоминаем, что набор данных имеет 581 012 прецедентов, так что его потоковая передача и моделирование алгоритмом SGD могут занять достаточно продолжительное время (это вполне крупномасштабная задача для одиночного компьютера). Хотя мы установили ограничитель на количество наблюдений в размере всего 250 000 прецедентов, все же дайте компьютеру поработать в течение порядка 15–20 мин., чтобы получить конечные результаты:

```
In:
import csv, time
import numpy as np
from sklearn.linear_model import SGDClassifier

source = 'data\\shuffled_covtype.data'
SEP = str(',')
forest_type = [t+1 for t in range(7)]

SGD = SGDClassifier(loss='log',
                    penalty=None,
                    random_state=1,
                    average=True)
accuracy = 0
holdout_count = 0
prog_accuracy = 0
prog_count = 0
cold_start = 200000
k_holdout  = 10

with open(local_path+'\\'+source, 'rb') as R:
    iterator = csv.reader(R, delimiter=SEP)
    for n, row in enumerate(iterator):
        if n > 250000: # сокращаем время выполнения эксперимента
            break
        # ОБРАБОТКА ДАННЫХ
        response = np.array([int(row[-1])]) # последнее значение - отклик
        features = np.array(map(float,row[:-1])).reshape(1,-1)
        # МАШИННОЕ ОБУЧЕНИЕ
        if (n+1) >= cold_start and (n+1-cold_start) % k_holdout==0:
            if int(SGD.predict(features))==response[0]:
                accuracy += 1
            holdout_count += 1
            if (n+1-cold_start) % 25000 == 0 and (n+1) > cold_start:
                    print('%s Точность на отложенных данных: %0.3f'
                          % (time.strftime('%X'),
                             accuracy / float(holdout_count)))
        else:
            # ПРОГРЕССИВНАЯ ПРОВЕРКА
            if (n+1) >= cold_start:
                if int(SGD.predict(features)) == response[0]:
                    prog_accuracy += 1
                prog_count += 1
                if n % 25000 == 0 and n > cold_start:
```

```
                    print('%s Прогрессивная точность: %0.3f'
                          % (time.strftime('%X'),
                             prog_accuracy / float(prog_count)))
            # ФАЗА ОБУЧЕНИЯ
            SGD.partial_fit(features, response, classes=forest_type)
print('%s ИТОГОВАЯ точность на отложенных данных: %0.3f'
      % (time.strftime('%X'),
         accuracy / ((n+1-cold_start) / float(k_holdout))))
print('%s ИТОГОВАЯ прогрессивная точность: %0.3f'
      % (time.strftime('%X'),
         prog_accuracy / float(prog_count)))
```

```
Out:
13:39:59 Точность на отложенных данных: 0.637
13:39:59 Прогрессивная точность: 0.616
13:41:22 Точность на отложенных данных: 0.628
13:41:22 Прогрессивная точность: 0.614
13:41:22 ИТОГОВАЯ точность на отложенных данных: 0.629
13:41:22 ИТОГОВАЯ прогрессивная точность: 0.614
```

В качестве второго примера мы попытаемся предсказать число сданных в аренду велосипедов в Вашингтоне на основе серии данных о погоде и времени. Учитывая исторический порядок следования данных в наборе, мы его не перемешиваем и рассматриваем задачу как временной ряд. Наша стратегия проверки состоит в тестировании результатов после просмотра определенного числа примеров, для того чтобы, начиная с этого момента, реплицировать неопределенности с целью выполнения прогноза.

Кроме того, интересно отметить, что некоторые признаки являются категориальными, и поэтому мы применили класс `FeatureHasher` из библиотеки Scikit-learn, чтобы записать категории как ключи в словаре в виде комбинированной строки, состоящей из кода категории и имени переменной. Назначаемое каждому из этих ключей в словаре значение аналогично двоичной переменной в разреженном векторе, который будет создаваться хэшированием признаков:

```
In:
import csv, time, os
import numpy as np
from sklearn.linear_model import SGDRegressor
from sklearn.feature_extraction import FeatureHasher

local_path = os.getcwd().decode('cp1251')
source = 'data\\bikesharing\\hour.csv'
SEP = str(',')

def apply_log(x): return np.log(float(x)+1)
def apply_exp(x): return np.exp(float(x))-1

SGD = SGDRegressor(loss='squared_loss',
                   penalty=None,
                   random_state=1,
                   average=True)
h = FeatureHasher(non_negative=True)
```

```
val_rmse  = 0
val_rmsle = 0
predictions_start = 16000

with open(local_path+'\\'+source, 'rb') as R:
    iterator = csv.DictReader(R, delimiter=SEP)
    for n, row in enumerate(iterator):
        # ОБРАБОТКА ДАННЫХ
        target = np.array([apply_log(row['cnt'])])
        features = {k+'_'+v:1 for k,v in row.iteritems()
                    if k in ['holiday','hr','mnth','season',
                             'weathersit','weekday','workingday','yr']}
        numeric_features = {k:float(v)
                            for k,v in row.iteritems()
                            if k in ['hum', 'temp', 'atemp', 'windspeed']}
        features.update(numeric_features)
        hashed_features = h.transform([features])
        # МАШИННОЕ ОБУЧЕНИЕ
        if (n+1) >= predictions_start:
            # ФАЗА С ОТКЛАДЫВАНИЕМ ПОСЛЕ N
            predicted  = SGD.predict(hashed_features)
            val_rmse  += (apply_exp(predicted) - apply_exp(target))**2
            val_rmsle += (predicted - target)**2
            if (n-predictions_start+1) % 250 == 0
            and (n+1) > predictions_start:
                print('%s RMSE на отложенных данных: %0.3f,'
                      % (time.strftime('%X'),
                        (val_rmse / float(n-predictions_start+1))**0.5)),
                print('то же самое RMSLE: %0.3f'
                      % ((val_rmsle / float(n-predictions_start+1))**0.5))
        else:
            # ФАЗА ОБУЧЕНИЯ
            SGD.partial_fit(hashed_features, target)

print('%s ИТОГОВАЯ RMSE на отложенных данных: %0.3f'
      % (time.strftime('%X'),
        (val_rmse / float(n-predictions_start+1))**0.5))
print('%s ИТОГОВАЯ RMSLE на отложенных данных: %0.3f'
      % (time.strftime('%X'),
        (val_rmsle / float(n-predictions_start+1))**0.5))

Out:
15:42:20 RMSE на отложенных данных: 281.214, то же самое RMSLE: 1.901
15:42:20 RMSE на отложенных данных: 255.092, то же самое RMSLE: 1.803
15:42:20 RMSE на отложенных данных: 255.578, то же самое RMSLE: 1.799
15:42:21 RMSE на отложенных данных: 254.598, то же самое RMSLE: 1.816
15:42:21 RMSE на отложенных данных: 239.728, то же самое RMSLE: 1.738
15:42:21 ИТОГОВАЯ RMSE на отложенных данных: 229.154
15:42:21 ИТОГОВАЯ RMSLE на отложенных данных: 1.679
```

Аббревиатуры RMSE (root mean squared error) и RMSLE (root mean squared logarithmic error) обозначают соответственно корень из среднеквадратической ошибки и корень из среднеквадратической логарифмической ошибки.

Резюме

В этой главе мы выяснили, как организуется процесс обучения вне ядра путем потоковой передачи данных из текстового файла или базы данных на жестком диске, независимо от того, насколько большими эти данные являются. Эти методы, конечно же, применяются к гораздо более крупным наборам данных, чем те примеры, которые мы использовали для их демонстрации (которые в действительности могут быть решены в оперативной памяти при помощи нестандартных и мощных аппаратных средств).

Мы также дали объяснение ключевого алгоритма, который делает возможным внеядерное обучение, – алгоритма стохастического градиентного спуска SGD – и исследовали его достоинства и недостатки, подчеркнув необходимость настоящей стохастичности потоков (т. е. обеспечения произвольного порядка следования прецедентов), чтобы стать по-настоящему эффективными, если, конечно, порядок следования не является частью задачи обучения. В частности, мы представили реализацию алгоритма SGD в библиотеке Scikit-learn, ограничившись линейной регрессионной и логистической регрессионной функциями потерь.

Наконец, мы обсудили подготовку данных, представили вводную информацию о хэшировании признаков и стратегии проверки потоков и воплотили полученные знания об алгоритме SGD, выполнив подгонку двух разных моделей – классификации и регрессии.

В следующей главе мы продолжим обогащать наши возможности по внеядерному обучению, выяснив, каким образом подключить в нашу схему обучения нелинейность и кусочно-линейную функцию потерь для метода опорных векторов. Мы также представим альтернативные библиотеке Scikit-learn программные инструменты, такие как **Liblinear**, **Vowpal Wabbit** и **StreamSVM**. Хотя все они выполняются на основе внешних команд оболочки, они легко обертываются и контролируются сценариями Python.

Глава 3

Быстрообучающиеся реализации машин SVM

Поэкспериментировав с онлайновым стилем обучения в предыдущей главе, вы, возможно, были удивлены его простотой и вместе с тем эффективностью и масштабируемостью по сравнению с пакетным обучением. Несмотря на то что в каждый момент времени обучение протекает на основе всего одного примера, алгоритм SGD может аппроксимировать результаты так же, как при обучении, когда все данные находятся в оперативной памяти и используется пакетный алгоритм. Требуется всего лишь обеспечить, чтобы поток был по-настоящему стохастическим (данные не содержат никаких тенденций) и чтобы ученик был хорошо настроен на задачу (темп обучения часто является ключевым и фиксированным параметром).

Так или иначе, внимательно изучая подобного рода достижения, совершенно очевидно, что полученные результаты все же сопоставимы только с пакетными линейными моделями, но не с более сложными учениками, отличающимися более высокой дисперсией, чем смещение, такими как машины опорных векторов, нейронные сети или ансамбли деревьев решений с бэггингом и бустингом.

Для определенных задач, в частности с высокими и широкими, но разреженными данными, можно обойтись и линейными комбинациями, если исходить из наблюдения, что простой алгоритм с большим количеством данных зачастую побеждает более сложные, натренированные на меньшем количестве данных. Но если же линейные модели применять, прибегая к явному отображению существующих признаков на признаки с более высокой размерностью (используя другой порядок взаимодействий, полиномиальные разложения и ядерные аппроксимации), то они могут ускорить и улучшить процесс усвоения сложных нелинейных связей между откликом и признаками.

В настоящей главе мы поэтому сначала представим линейные машины SVM как альтернативный линейным моделям алгоритм машинного обучения, который опирается на другой подход к задаче обучения на данных. Затем мы продемонстрируем, как из существующих признаков лучше всего создавать более полные признаки для решения задач машинного обучения, когда приходится иметь дело с крупномасштабными данными, в особенности высокими данными (т. е. наборами данных, имеющими много прецедентов, на которых будет выполняться обучение).

Таким образом, в этой главе мы затронем следующие темы:

- ознакомление с машинами SVM и предоставление лежащих в их основе понятий и математических формул, объясняющих принцип работы этого алгоритма;

- представление алгоритма SGD с кусочно-линейной функцией потерь как эффективного решения крупномасштабных задач, в котором используется такой же подход к оптимизации, что и в пакетном алгоритме SVM;
- рассмотрение сопутствующих алгоритму SGD нелинейных приближений;
- обзор других крупномасштабных онлайновых решений, представленных в библиотеке Scikit-learn помимо алгоритма SGD.

Наборы данных для самостоятельного экспериментирования

Как и в предыдущей главе, мы будем использовать наборы данных из репозитория машинного обучения UCI, в частности наборы данных Bike-sharing об аренде велосипедов (задача регрессии) и Covertype о лесном покрове (задача многоклассовой классификации).

Если вы еще их не скачали либо если необходимо скачать оба набора данных снова, то вам потребуется пара функций, определенных в разделе «*Наборы данных для реальных дел*» *главы 2* «*Масштабируемое обучение в Scikit-learn*». Вам потребуются функции `unzip_from_UCI` и `gzip_from_UCI`. Обе эти функции задействуют Python'овское подключение к репозиторию UCI; скачайте сжатый файл и разархивируйте его в рабочем каталоге Python. Если вызвать функции из ячейки Jupyter, то вы найдете соответствующие новые каталоги и файлы именно в том месте, где Jupyter будет их искать.

В случае если эти функции не работают, не переживайте; мы предоставим ссылку для прямого скачивания. После этого необходимо только распаковать данные в текущем рабочем каталоге Python, который можно узнать, выполнив следующую команду в интерфейсе Python (в Jupyter либо любом другом IDE):

```
In:
import os
print(u'Текущий каталог: "{0}"'.format(os.getcwd().decode('cp1251')))

Out:
Текущий каталог: "C:\WinPython-64bit-2.7.9.4\notebooks\Packt-Large Scale"
```

Набор данных Bike-sharing

Этот набор данных включает в себя два файла в формате CSV, содержащих почасовое и ежедневное количество велосипедов, взятых в аренду за период с 2011 по 2012 г. в системе Capital Bike-share в г. Вашингтон, округ Колумбия, США. Как напоминание, в данных отражена соответствующая погодная и сезонная информация относительно дня сдачи в аренду.

Следующий фрагмент кода сохранит набор данных на локальном жестком диске при помощи вспомогательной оберточной функции `unzip_from_UCI`:

```
In:
UCI_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00275/Bike-Sharing-
Dataset.zip'
unzip_from_UCI(UCI_url, dest='data\\bikesharing')

Out:
Извлечение в C:\scisoft\WinPython-64bit-2.7.9.4\notebooks\data\bikesharing
```

```
распакован day.csv
распакован hour.csv
```

В случае успешного выполнения программный код укажет, в каком каталоге CSV-файлы были сохранены, и распечатает имена обоих разархивированных файлов. При неуспешном выполнении просто скачайте файл, используя следующую прямую ссылку https://archive.ics.uci.edu/ml/machine-learning-databases/00275/Bike-Sharing-Dataset.zip, и разархивируйте оба файла, `day.csv` и `hour.csv`, в каталоге `bikesharing`, который вы ранее создали в своем рабочем каталоге Python.

Набор данных Covertype

Набор данных Covertype, безвозмездно предоставленный Джоком А. Блэкардом, Дэнисом Дж. Дином, Чарлзом У. Андерсеном (Jock A. Blackard, Dr. Denis J. Dean, Dr. Charles W. Anderson) и Университетом штата Колорадо, содержит 581 012 примеров и серию из 54 картографических переменных, включая рельеф местности и тип почвы, предназначенных для предсказания типа лесного покрова, который состоит из 7 видов (таким образом, эта задача является мультиклассовой). Для того чтобы обеспечить сравнимость с академическими исследованиями на тех же данных, согласно инструкции, рекомендуется использовать первые 11 340 записей для обучения, следующие 3780 записей для проверки и, наконец, оставшиеся 565 892 записи как тестовые примеры:

```
In:
UCI_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/ covtype/covtype.data.gz'
gzip_from_UCI (UCI_url, dest='data\\')
```

В случае проблем при выполнении программного кода либо если вы предпочитаете подготовить файл самостоятельно, просто перейдите на веб-сайт UCI, скачайте набор данных по прямой ссылке https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/covtype.data.gz и распакуйте его в каталог, в котором в настоящее время работает Python.

Машины опорных векторов

Машины опорных векторов (support vector machines, SVMs) – это набор методов обучения с учителем для задач классификации и регрессии (а также для обнаружения выбросов), который довольно универсален, поскольку он может подбирать как линейные, так и нелинейные модели благодаря специальным ядерным функциям. Особенность этих ядерных функций состоит в том, что они способны отображать входные признаки на новый, более сложный вектор признаков, используя для этого ограниченный объем вычислений. Ядерные функции нелинейно рекомбинируют исходные признаки, делая возможным отображение отклика очень сложными функциями. В этом смысле машины SVM сравнимы с нейронными сетями как универсальными аппроксиматорами и тем самым во многих задачах могут демонстрировать аналогичную прогнозирующую способность.

Вопреки рассмотренным в предыдущей главе линейным моделям, машины опорных векторов начинались как метод для решения не регрессионных, а классификационных задач.

Алгоритм SVM был разработан в лабораториях AT&T в 90-х годах математиком Владимиром Вапником (Vladimir Vapnik) и программистом Коринна Кортес (Corinna Cortes) (но над алгоритмом вместе с Вапником также работали многие другие участники). В сущности, алгоритм SVM стремится решить задачу классификации путем нахождения особой гиперплоскости, разделяющей классы в пространстве признаков. Такого рода гиперплоскость должна характеризоваться наличием самого большого зазора между границами классов (зазор призван служить в качестве промежутка или пространства без единого примера непосредственно между самими классами).

Такое интуитивно понятное объяснение подразумевает два последствия:

- эмпирически машины SVM пытаются минимизировать ошибку на тестовом наборе путем нахождения решения в тренировочном наборе, которое находится точно посередине наблюдаемых классов, тем самым решение однозначно имеет вычислительный характер (оно представляет собой оптимизацию на основе квадратичного программирования – https://en.wikipedia.org/wiki/Quadratic_programming;
- поскольку решение основывается только на границах классов, задаваемых смежными примерами (именуемыми опорными векторами), другие примеры могут быть проигнорированы, что делает метод нечувствительным к выбросам и менее интенсивным относительно потребления оперативной памяти, чем методы, основанные на обращении матрицы, в частности линейные модели.

Предоставив общие сведения о машинах SVM, далее мы посвятим несколько страниц, делая акцент на ключевых формулах, которые характеризуют этот метод. Несмотря на то что полное и подробное объяснение этого метода выходит за рамки настоящей книги, схематичное описание того, как он работает, может действительно помочь выяснить его внутреннюю механику и обеспечит основу для понимания, каким образом его можно сделать масштабируемым до больших данных.

Исторически машины SVM рассматривались как классификаторы с *жестким зазором*, точно так же, как и персептрон. В сущности, машины SVM были первоначально сформулированы в попытке найти две гиперплоскости, разделяющие классы, взаимное расстояние которых было максимально возможным. Такой подход отлично работал с линейно разделимыми синтетическими данными. Так или иначе, в версии с жестким зазором, когда машина SVM сталкивалась с нелинейно разделимыми данными, она могла быть успешной только с использованием нелинейных преобразований признаков. Однако они всегда квалифицировались как неуспешные, когда ошибки неправильной классификации происходили не из-за нелинейности, а из-за шума в данных.

По этой причине были введены мягкие зазоры с использованием функции стоимости, которая принимала в расчет серьезность ошибки (так как жесткие зазоры срабатывали, только если ошибка происходила), тем самым позволяя иметь определенную терпимость к неправильно классифицированным случаям, ошибка которых не была слишком большой из-за того, что они были размещены рядом с разделяющей гиперплоскостью.

С введением мягких зазоров машины SVM стали способны противостоять неразделимости, вызванной шумом. Мягкие зазоры вводились путем создания функции стоимости вокруг слабой переменной, которая аппроксимирует число неправильно классифицированных примеров. Подобного рода слабая перемен-

ная также называется эмпирическим риском (риском построения неправильной классификации, если рассматривать его с точки зрения тренировочных данных).

Математически, если дана матрица набора данных X из n примеров и m признаков и вектор отклика, выражающий принадлежность к классу, где `+1` (принадлежит) и `-1` (не принадлежит), то алгоритм SVM для бинарной классификации стремится минимизировать следующую функцию стоимости:

$$\frac{\lambda}{2}||w^2|| + \left[\frac{1}{n}\sum_{i=1}^{n}\max(0,1 - y(wX + b))\right].$$

В приведенной выше функции w – это вектор коэффициентов, который выражает разделяющую гиперплоскость вместе со смещением b, которое обозначает смещение от источника; также имеется лямбда-параметр ($\lambda \geqslant 0$), который является параметром регуляризации.

Для лучшего понимания того, как работает функция стоимости, необходимо разделить ее на две части. Первая часть – это член регуляризации:

$$\frac{\lambda}{2}||w^2||.$$

Член регуляризации делает процесс минимизации более явным, когда вектор w принимает высокие значения. Второй член называется составляющей потерь, или слабой переменной, и в действительности является ядром процедуры минимизации в алгоритме SVM:

$$\frac{1}{n}\sum_{i=1}^{n}\max(0,1 - y(wX + b)).$$

Составляющая потерь выдает приближенное значение ошибок классификации. По сути дела, суммирование будет стремиться добавлять единичную величину каждой ошибки классификации, общее количество которых, деленное на число примеров n, обеспечит приблизительную долю ошибки классификации.

Очень часто, подобно тому, как это реализовано в Scikit-learn, параметр λ убирается из члена регуляризации и заменяется параметром неправильной классификации C, который умножается на составляющую потерь:

$$\frac{1}{2}||w^2|| + C\left[\sum_{i=1}^{n}\max(0,1 - y(wX + b))\right].$$

Связь между приведенным выше параметром λ и новым параметром C в следующем:

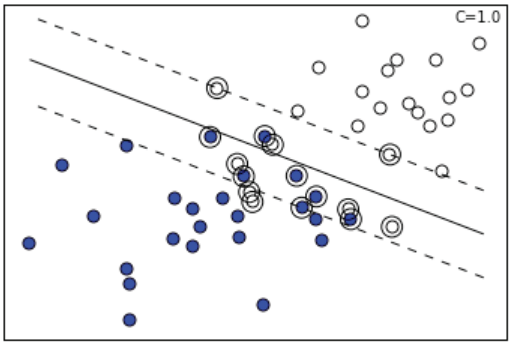
$$\lambda = \frac{1}{nC}.$$

Это просто вопрос принятых правил, так как переход от параметра λ к параметру C в формуле оптимизации не означает разных результатов.

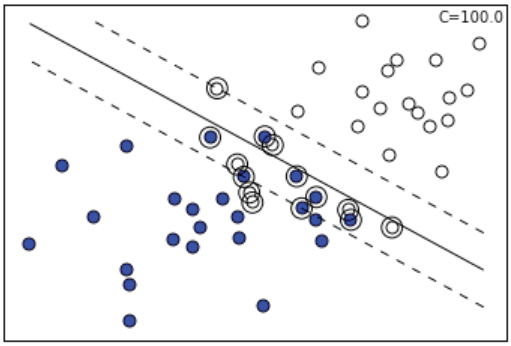
Влияние составляющей потерь опосредуется гиперпараметром C. Высокие значения C налагают высокий штраф на ошибки, тем самым вынуждая алгоритм SVM пытаться правильно классифицировать все тренировочные примеры. Следовательно, большие значения C вынуждают зазор становиться более жестким и рас-

сматривать меньшее число опорных векторов. Такого рода сокращение зазора транслируется в увеличенное смещение и уменьшенную дисперсию.

Это приводит к необходимости конкретизации роли определенных наблюдений относительно других; фактически в качестве опорных векторов мы задаем те примеры, которые неправильно классифицированы или не классифицированы с уверенностью, что они лежат в зазоре (шумные наблюдения, которые делают разделение классов невозможным). Оптимизация становится возможной, принимая в расчет только такого рода примеры, что делает алгоритм SVM по-настоящему эффективным методом использования оперативной памяти:

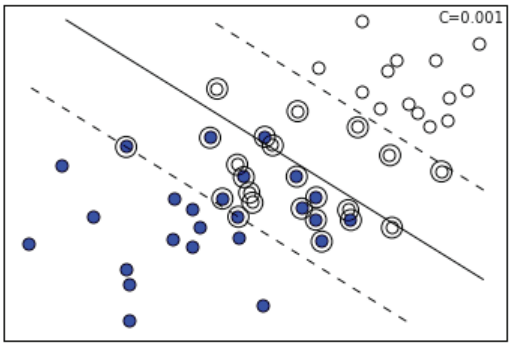


На приведенной выше визуализации можно заметить проекцию двух групп точек (синих и белых) на две размерности признаков. Вычислительное решение на основе алгоритма SVM с гиперпараметром *C*, установленным в 1.0, способно легко обнаруживать линию разделения (на графике она представлена как сплошная линия), несмотря на то что с обеих сторон имеется несколько неправильно классифицированных случаев. Кроме того, зазор можно визуализировать (он задан двумя пунктирными линиями); он идентифицируется благодаря опорным векторам соответствующего класса, лежащим дальше от разделяющей линии. На графике опорные векторы отмечены внешним кругом, при этом можно заметить, что некоторые опорные векторы в действительности лежат вне зазора; это вызвано тем, что они являются неправильно классифицированными случаями, и алгоритм SVM должен отслеживать их в целях оптимизации, поскольку их ошибка учитывается в составляющей потерь:



При увеличении значения *C* зазор стремится сузиться, поскольку в процессе оптимизации алгоритм SVM принимает в расчет меньше опорных векторов. Следовательно, наклон разделяющей линии также изменяется.

И наоборот, меньшие значения *C* имеют тенденцию к ослаблению зазора и тем самым к увеличению дисперсии. Крайне малые значения *C* могут даже привести к тому, что алгоритм SVM будет рассматривать в зазоре все прецедентные точки. Меньшие значения *C* идеальны, когда имеется много шумных примеров. Такие условия вынуждают алгоритм SVM игнорировать большое число неправильно классифицированных примеров в определении зазора (ошибки получают меньший вес, и поэтому при поиске максимального зазора терпимость к ним больше):



Продолжая с предыдущим визуальным примером, если уменьшить гиперпараметр *C*, то зазор в действительности расширяется, потому что число опорных векторов увеличивается. Следовательно, раз зазор другой, алгоритм SVM принимает решение в пользу другой разделяющей линии. Какое-либо значение *C*, которое можно расценивать как правильное до его проверки на данных, отсутствует; правильное значение нужно всегда находить эмпирически путем перекрестной проверки. Вне всякого сомнения, в алгоритме SVM гиперпараметр *C* считается самым важным, и его настройка выполняется после решения о том, какую ядерную функцию использовать.

Вместе с тем ядерные функции отображают исходные признаки в более высокоразмерное пространство путем их нелинейного комбинирования, в результате чего явно неразделимые группы в исходном пространстве признаков могут стать разделимыми в более высокоразмерном представлении. Такая проекция не требует слишком сложных вычислений, несмотря на то что процесс явного преобразования исходных значений признаков в новые может генерировать потенциальный взрыв в числе признаков при проецировании в высокие размерности. Вместо того чтобы делать такого рода громоздкие вычисления, в решающую функцию просто подключаются ядерные функции, тем самым заменяя исходное скалярное произведение между признаками и вектором коэффициентов и получая тот же самый результат оптимизации, который имело бы явное отображение. (Такого рода подключение называется ядерным трюком, потому что на самом деле он представляет собой математический прием.)

Стандартные ядерные функции – это линейные функции (не предполагающие никаких преобразований), полиномиальные функции, **радиально-базисные**

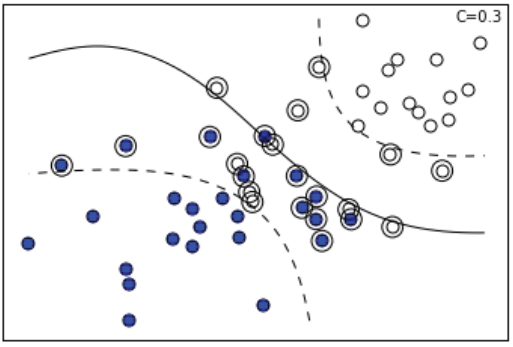
функции (radial basis function, RBF) и сигмоидальные функции. Чтобы получить представление, функцию RBF можно выразить следующим образом:

$K(x_i, x) = \exp(-||x_i - x||^2/2\sigma)$.

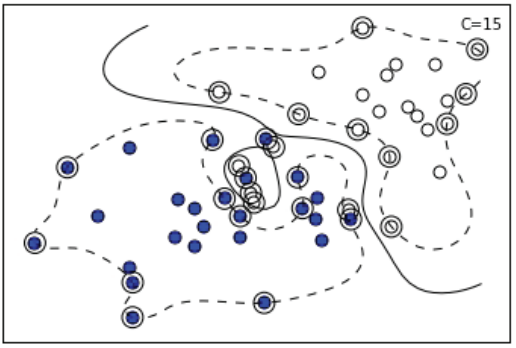
В сущности, ядро RBF и другие ядра подключаются непосредственно в вариант ранее встречавшейся минимизируемой функции. Ранее встречавшаяся функция оптимизации называется первичной формулировкой, тогда как аналогичное преобразованное выражение называется двойственной формулировкой:

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b.$$

Хотя без математической демонстрации переход от первичной к двойственной формулировке довольно сложен, важно понимать, что если дана ядерная функция, которая сравнивает примеры парами, то ядерный трюк является просто вопросом ограниченного количества вычислений относительно бесконечномерного пространства признаков, которое она может развернуть. Такого рода ядерный трюк выражает алгоритм с наибольшей эффективностью (сравнимой с нейронными сетями) относительно довольно сложных задач, таких как распознавание изображений или текстовая классификация:



Например, приведенное выше решение алгоритма SVM возможно благодаря сигмоидальному ядру, тогда как следующее является результатом ядра RBF:



Как видно из графика, ядро RBF делает возможным довольно сложные определения зазора, и в том числе с его расщеплением на многочисленные части (в приведенном выше примере можно заметить анклав).

Формула ядра RBF следующая:

$k(x_i, x_j) = \exp(-\gamma||x_i - x_j||^2)$.

Гиперпараметр гамма γ задается априорно. Ядерное преобразование создает своего рода классификационные пузыри вокруг опорных векторов, тем самым позволяя определять очень сложные граничные фигуры путем слияния самих пузырей.

Формула сигмоидального ядра следующая:

$k(x_i, x_j) = \tanh(\gamma\langle x_i, x_j\rangle^2 + r)$.

Здесь, кроме параметра гамма γ, для получения оптимального результата нужно также подобрать r.

Безусловно, решения на основе сигмоидального, RBF- и полиномиального ядер (последнее неявно выполняет полиномиальное разложение, которое мы обсудим в следующих параграфах), обеспечивают дисперсию оценок больше, чем их смещение, тем самым требуя строгой проверки при принятии решения об их использовании. Хотя алгоритм SVM устойчив к переподгонке, он, разумеется, от нее не застрахован.

Опорно-векторная регрессия связана с опорно-векторной классификацией. Она отличается только математической записью (более похожа на линейную регрессию, используя коэффициенты β вместо вектора коэффициентов w) и функцией потерь:

$$\sum_{j=1}^{m}\beta_j^2 + C\sum_{i=1}^{n}L_\varepsilon(y_i - \hat{y}_l).$$

Стоит отметить, что единственное значительное отличие – это функция потерь L_ε, которая нечувствительна к ошибкам (и тем самым их не вычисляет), если примеры лежат на определенном расстоянии эпсилон ε от гиперплоскости регрессии. Минимизация такой функции стоимости оптимизирует результат задачи регрессии, выдавая на выходе значения, а не классы.

Кусочно-линейная функция потерь и ее варианты

В качестве заключительных комментариев по поводу составных частей алгоритма SVM стоит напомнить, что лежащая в основе алгоритма функция стоимости – это кусочно-линейная функция потерь (hinge loss):

$loss(y, \hat{y}) = \max(0, 1 - y\hat{y})$.

Как уже отмечалось ранее, $\hat{y}$ выражается как сумма скалярного произведения X и вектора коэффициентов w со смещением b:

$\hat{y} = wX + b$.

Такая функция потерь напоминает *персептрон* и штрафует ошибки линейно, показывая ошибку, когда пример классифицирован на неправильной стороне

зазора, пропорционально его расстоянию от самого зазора. Хотя и являясь выпуклой, она имеет недостаток, который состоит в том, что она не может быть повсюду дифференцируемой, и поэтому она иногда заменяется на всегда дифференцируемые варианты, в частности квадратичную кусочно-линейную функцию (также именуемую функцией потерь L2, при этом функция потерь L1 – это кусочно-линейная функция потерь):

$$L2\_loss(y, \hat{y}) = \max(0,1 - y\hat{y})^2.$$

Другой вариант – функция потерь Хьюбера, т. е. квадратичная функция, когда ошибка равна определенному пороговому значению h, или ниже его, и линейная функция в противном случае. Такой подход смешивает варианты L1 и L2 кусочно-линейной функции потерь, основываясь на ошибке, и представляет собой достаточно стойкую к выбросам альтернативу, поскольку более крупные значения ошибок не возводятся в квадрат, и тем самым требует от обучающегося алгоритма SVM меньшего количества корректировок. Функция потерь Хьюбера является также альтернативой для логарифмической функции потерь (линейные модели), потому что ее вычисление быстрее, и она способна обеспечить оценки вероятностей классов (кусочно-линейная функция не обладает такой способностью).

С практической точки зрения нет каких-либо конкретных сообщений о том, что функция потерь Хьюбера либо квадратичная кусочно-линейная функция L2 может систематически выполняться лучше, чем просто кусочно-линейная функция. В конечном итоге процесс выбора функции стоимости просто сводится к тестированию имеющихся функций относительно каждой конкретной задачи обучения. (Согласно принципу теоремы об отсутствии бесплатных обедов no-free-lunch, в машинном обучении отсутствуют какие-либо решения, которые были бы приемлемы для всех задач.)

Объяснение реализации алгоритма SVM в Scikit-learn

Библиотека Scikit-learn предлагает реализацию алгоритма SVM с использованием двух библиотек C++, разработанных в Национальном Тайваньском университете (с программным интерфейсом на C для взаимодействия с другими языками): **библиотеки для машин опорных векторов** LIBSVM (library for support vector machines) для классификации и регрессии на основе алгоритма SVM (http://www.csie.ntu.edu.tw/~cjlin/libsvm/) и LIBLINEAR для задач классификации с использованием линейных методов для больших и разреженных наборов данных (http://www.csie.ntu.edu.tw/~cjlin/liblinear/). Обе библиотеки общедоступны, довольно быстры в расчетах и уже протестированы в большом количестве других решений, и все реализации Scikit-learn в модуле `sklearn.svm` опираются на одно или другое (кстати, классы `Perceptron` и `LogisticRegression` тоже их используют), превращая Python просто в удобную обертку.

С другой стороны, классы `SGDClassifier` и `SGDRegressor` используют иную реализацию, поскольку ни LIBSVM, ни LIBLINEAR не обладают онлайновой версией, являясь инструментами пакетного обучения. По сути дела, во время работы LIBSVM и LIBLINEAR показывают наилучшую производительность, когда посредством параметра `cache_size` для ядерных операций выделяется надлежащая память.

Реализации для задачи классификации следующие:

Класс	Цель	Гиперпараметры
`sklearn.svm.SVC`	LIBSVM-реализация для бинарной и мультиклассовой линейной и ядерной классификаций	C, kernel, degree, gamma
`sklearn.svm.NuSVC`	То же, что и выше	nu, kernel, degree, gamma
`sklearn.svm. OneClassSVM`	Неконтролируемое (без учителя) обнаружение выбросов	nu, kernel, degree, gamma
`sklearn.svm.LinearSVC`	Бинарный и мультиклассовый линейный классификатор на основе LIBLINEAR	Penalty, loss, C

Что касается задачи регрессии, решения такие:

Класс	Цель	Гиперпараметры
`sklearn.svm.SVR`	LIBSVM-реализация для регрессии	C, kernel, degree, gamma, epsilon
`sklearn.svm.NuSVR`	То же, что и выше	nu, C, kernel, degree, gamma

Как видно, имеется немалое количество гиперпараметров, которые требуют доводки для каждой версии алгоритма, что делает машины SVM хорошими учениками, когда эти параметры используются со значениями, заданными по умолчанию, и отличными учениками, когда эти параметры соответствующим образом отстроены благодаря перекрестной проверке с использованием класса `GridSearchCV` из модуля `grid_search`[1].

Золотое правило гласит, что некоторые параметры влияют на результат больше, и поэтому их следует зафиксировать заранее, а другие зависят от их значений. Согласно такому эмпирическому правилу, следует правильно установить следующие ниже параметры (упорядочены по важности):

- `C`: значение штрафа, который мы обсуждали ранее. Его уменьшение делает зазор больше, тем самым позволяя игнорировать больше шума и одновременно способствуя увеличению объема вычислений. Наилучшее значение, как правило, можно отыскать в интервале `np.logspace(-3, 3, 7)`;
- `kernel`: наиболее интенсивно используемый параметр, «рабочая лошадка» нелинейности, потому что алгоритм SVM может быть установлен в `linear`, `poly`, `rbf`, `sigmoid`, либо собственное ядро (только для экспертов!). Широко используемым является параметр `rbf`;
- `degree`: работает с `kernel='poly'`, сигнализируя о размерности полиномиального разложения. Игнорируется другими ядрами. Значения 2–5 обычно работают лучше всего;
- `gamma`: этот коэффициент предназначен для `'rbf'`, `'poly'` и `'sigmoid'`; высокие значения стремятся аппроксимировать данные лучше всего. Сеточный поиск[2] предлагается выполнять в интервале `np.logspace(-3, 3, 7)`;

---

[1] Начиная с версии 0.18.0 библиотеки Scikit-learn класс `GridSearchCV` из модуля `grid_search` перемещен в модуль `model_selection`. – *Прим. перев.*

[2] Сеточный поиск (от англ. grid search) – это исчерпывающий поиск оптимальной параметрической конфигурации в массиве, заданном списком значений параметра либо словарем, ключом которого является имя параметра, а значением – серия проверяемых значений. – *Прим. перев.*

- `nu`: этот параметр для регрессии и классификации используется с классами nuSVR и nuSVC; он аппроксимирует тренировочные точки, которые не были классифицированы с уверенностью, неправильно классифицированные точки и правильные точки внутри или на зазоре. Параметр должен быть выражен числом в интервале [0,1], поскольку этот параметр представляет собой долю относительно тренировочного набора. В конце концов, он действует как параметр *C*, когда высокие доли увеличивают зазор;
- `epsilon`: этот параметр определяет, сколько ошибок класс SVR собирается принять, и делается это путем определения интервала в размере эпсилон ε, где никакой штраф не ассоциируется относительно истинного значения точки. Предлагается выполнять его поиск в интервале `np.insert(np.logspace(-4, 2, 7), 0, [0])`;
- `penalty`, `loss` и `dual`: для LinearSVC эти параметры принимают сочетания `('l1', 'squared_hinge', False)`, `('l2', 'hinge', True)`, `('l2', 'squared_hinge', True)` и `('l2', 'squared_hinge', False)`. Сочетание `('l2', 'hinge', True)` эквивалентно ученику `SVC(kernel='linear')`.

В качестве примера базовой классификации и регрессии с использованием классов `SVC` и `SVR` из модуля `sklearn.svm` библиотеки Scikit-learn мы возьмем пару популярных миниатюрных наборов данных – Iris и Boston (http://scikit-learn.org/stable/datasets/).

Сначала загрузим набор данных Iris:

```
In:
from sklearn import datasets

iris = datasets.load_iris()
X_i, y_i = iris.data, iris.target
```

Затем выполним подгонку `SVC` ядром RBF (параметры *C* и γ были выбраны на основе других известных примеров в Scikit-learn) и протестируем результаты при помощи функции `cross_val_score` для расчета отметки точности после перекрестной проверки:

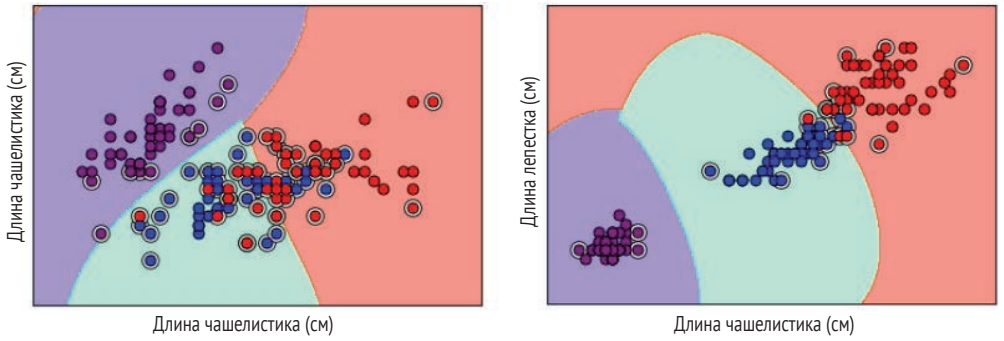
```
In:
import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score

h_class = SVC(kernel='rbf', C=1.0, gamma=0.7, random_state=101)
scores = cross_val_score(h_class, X_i, y_i, cv=20, scoring='accuracy')

print('Точность: %0.3f' % np.mean(scores))

Out:
Точность: 0.969
```

Подобранная модель предоставит массив индексов, указывающий на опорные векторы среди тренировочных примеров:

```
In:
h_class.fit(X_i, y_i)
print(h_class.support_)

Out:
[ 13 14 15 22 24 41 44 50 52 56 60 62 63 66 68 70
```

```
72 76 77 83 84 85 98 100 106 110 114 117 118 119 121 123 126 127
129 131 133 134 138 141 146 149]
```

Ниже приведено графическое представление опорных векторов, которые модель `SVC` выбрала для набора данных Iris. Оно представлено цветными границами решения (мы протестировали дискретный массив значений, чтобы для каждой области диаграммы иметь возможность спроецировать, какой класс модель предскажет):



Если вы интересуетесь тем, как воспроизвести приведенные выше графики, то можете взглянуть на фрагмент программного кода на http://scikit-learn.org/stable/auto_examples/svm/plot_iris.html и настроить его под свои потребности.

Чтобы протестировать регрессор алгоритма SVM, мы решили попробовать модель `SVR` с набором данных Boston. Сначала мы закачиваем набор данных в оперативную память и затем рандомизируем порядок следования примеров, так как можно заметить, что подобный набор данных, если присмотреться повнимательнее, в действительности упорядочен, тем самым делая результаты перекрестной проверки с нерандомизированным порядком следования недействительными:

```
In:
import numpy as np
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
boston = load_boston()
shuffled = np.random.permutation(boston.target.size)
X_b = scaler.fit_transform(boston.data[shuffled,:])
y_b = boston.target[shuffled]
```

Ввиду того, что мы использовали функцию `permutation` из модуля random библиотеки NumPy, вы можете получить набор данных, который будет перемешан по-другому, и поэтому при выполнении следующего ниже теста может оказаться немного другая отметка после перекрестной проверки. Кроме того, если признаки находятся в разных шкалах, то их рекомендуется стандартизировать, чтобы они имели центрованное на нуле среднее и единичную дисперсию. В особенности при использовании алгоритма SVM с ядрами. В этом случае стандартизация действительно крайне важна.

Наконец, можно выполнить подбор модели `SVR` (мы выбрали несколько параметров – C, гамма и эпсилон, которые, по нашим сведениям, отлично работают),

и, воспользовавшись перекрестной проверкой, оценили ее при помощи квадратного корня из среднеквадратической ошибки (RMSE):

```
In:
from sklearn.svm import SVR
from sklearn.model_selection import cross_val_score

h_regr = SVR(kernel='rbf', C=20.0, gamma=0.001, epsilon=1.0)
scores = cross_val_score(h_regr, X_b, y_b, cv=20,
                         scoring='neg_mean_squared_error')

print('Среднеквадратическая ошибка: %0.3f' % abs(np.mean(scores)))

Out:
Среднеквадратическая ошибка: 28.087
```

Поиск нелинейных SVM с привлечением подвыборки

Машины SVM имеют довольно много преимуществ перед другими алгоритмами машинного обучения:

- могут обрабатывать большинство задач с учителем, таких как регрессия, классификация и обнаружение аномалий, хотя в действительности они лучше всего проявляют себя в бинарной классификации;
- обеспечивают хорошую обработку шумных данных и выбросов и менее склонны к переподгонке, так как работают только с опорными векторами;
- хорошо работают с широкими наборами данных (больше признаков, чем примеров); хотя, как и с другими алгоритмами машинного обучения, алгоритм SVM извлекает выгоду как из снижения размерности, так и из отбора признаков.

Среди недостатков необходимо упомянуть следующие:

- предлагают лишь оценочные значения, а не вероятности, если только не выполнить некую времязатратную и в вычислительном отношении интенсивную калибровку вероятности посредством шкалирования Платта;
- масштабируются суперлинейно вместе с числом примеров.

В частности, последний недостаток накладывает сильное ограничение на использование машин SVM для больших наборов данных. Метод оптимизации в ядре этого обучающегося алгоритма – квадратичное программирование – масштабируется в реализации Scikit-learn между вычислительной сложностью *O*(число признаков*число прецедентов^2) и *O*(число признаков*число прецедентов^3), серьезно ограничивая применимость алгоритма к наборам данных, которые имеют объем до `10^4` случаев.

И опять-таки, как отмечено в предыдущей главе, когда имеется пакетный алгоритм и слишком много данных, существует всего несколько вариантов для выбора: подвыборка, параллелизация и внеядерное обучение путем потоковой передачи данных. Подвыборка и параллелизация редко упоминаются как самые лучшие решения, тогда как потоковой передаче отдается предпочтение при реализации машин SVM для решения крупномасштабных задач.

Вместе с тем подвыборка, хотя и реже используется, достаточно просто реализуется с привлечением резервуарного отбора, посредством которого можно быстро продуцировать случайные выборки из потоков, получаемых из наборов данных, и бесконечных онлайновых потоков. Благодаря подвыборке можно производить многочисленные модели SVM, чьи результаты могут усредняться

для получения более высоких общих результатов. Полученные из многочисленных моделей SVM прогнозы могут даже каскадироваться (или накладываться – stacked), тем самым создавая новый набор данных, и использоваться для создания новой модели с объединением прогнозных способностей каждой отдельной модели. Этот механизм будет описан в *главе 6* «*Классификационные и регрессионные деревья в крупном масштабе*».

Резервуарный отбор (reservoir sampling) – это алгоритм для случайного выделения выборок из потока без предварительной информации о длине потока. По сути дела, каждое наблюдение в потоке имеет одинаковую вероятность быть выбранным. Он инициализируется выборкой, извлекаемой из первых наблюдений в потоке, и далее каждый элемент в выборке в любой момент может быть заменен примером из потока согласно вероятности, пропорциональной числу элементов, переданных к настоящему моменту. Так, например, когда поступает *i*-й элемент потока, он имеет вероятность быть вставленным вместо случайного элемента в выборке. Такая вероятность вставки эквивалентна размерности выборки, деленной на *i*; поэтому она прогрессивно уменьшается относительно длины потока. Если поток бесконечен, то остановка в любое время гарантирует, что выборка является представительной для элементов, которые были просмотрены к настоящему моменту.

В нашем примере мы берем две случайные взаимоисключающие выборки из потока – одна для тренировки, другая для тестирования. Мы извлечем такого рода выборки из базы данных Covertype, используя исходный упорядоченный файл. (Поскольку мы будем передавать все данные потоком до извлечения выборки, случайный отбор не будет затронут упорядочением.) Мы остановились на тренировочной выборке в размере 5000 примеров – это количество должно хорошо масштабироваться на большинстве настольных компьютеров. Что касается тестового набора, то мы будем использовать 20 000 примеров:

```
In:
from random import seed, randint

SAMPLE_COUNT = 5000
TEST_COUNT = 20000

seed(0) # обеспечивает воспроизводимость результатов

sample = list()
test_sample = list()

for index, line in enumerate(open('data\\covtype.data','rb')):
    if index < SAMPLE_COUNT:
        sample.append(line)
    else:
        r = randint(0, index)
        if r < SAMPLE_COUNT:
            sample[r] = line
        else:
            k = randint(0, index)
            if k < TEST_COUNT:
                if len(test_sample) < TEST_COUNT:
                    test_sample.append(line)
                else:
                    test_sample[k] = line
```

Алгоритм должен обеспечивать достаточно быструю потоковую передачу порядка на 500 000 строках матрицы данных. Для того чтобы обеспечить максимальное быстродействие потока, во время потоковой передачи мы, в сущности, не делали никакой предобработки. Следовательно, теперь мы должны преобразовать данные в массив NumPy и стандартизировать признаки:

```
In:
import numpy as np
from sklearn.preprocessing import StandardScaler

for n,line in enumerate(sample):
    sample[n] = map(float,line.strip().split(','))

y = np.array(sample)[:,-1]
scaling = StandardScaler()
X = scaling.fit_transform(np.array(sample)[:,:-1])
```

Закончив работу с тренировочными данными *X*, *y*, необходимо таким же образом обработать тестовые данные; в частности, стандартизировать признаки, используя такие же параметры стандартизации (средние значения и стандартные отклонения), что и в тренировочной выборке:

```
In:
for n,line in enumerate(test_sample):
    test_sample[n] = map(float,line.strip().split(','))
yt = np.array(test_sample)[:,-1]
Xt = scaling.transform(np.array(test_sample)[:,:-1])
```

Когда оба набора – тренировочный и тестовый – готовы, можно построить модель SVC и предсказать результаты:

```
In:
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

h = SVC(kernel='rbf', C=250.0, gamma=0.0025, random_state=101)
h.fit(X,y)
prediction = h.predict(Xt)

print(accuracy_score(yt, prediction))

Out:
0.75205
```

Реализация SVM в крупном масштабе на основе SGD

Учитывая недостатки подвыборки (в первую очередь недоподгонку относительно моделей, тренируемых на крупных наборах данных), единственным вариантом при использовании Scikit-learn для применяемых к крупномасштабным потокам линейных машин SVM остается использование методов `SGDClassifier` и `SGDRegressor`, оба из которых доступны в модуле `linear_model`. Давайте посмотрим, как их можно применить оптимальным образом и как улучшить результаты на демонстрационных наборах данных.

Для этой цели мы задействуем примеры, рассмотренные ранее в главе, посвященной линейной и логистической регрессиям, и преобразуем их в эффективную

модель SVM. Если говорить о классификации, то при помощи гиперпараметра `loss` требуется задать тип функции потерь. Значениями этого параметра могут быть `'hinge'`, `'squared_hinge'` и `'modified_huber'`. Все эти функции потерь уже были представлены и обсуждались в настоящей главе при рассмотрении формулы алгоритма SVM.

Все они предполагают применение линейного алгоритма SVM с мягким зазором (без ядер), тем самым приводя к модели SVM, которая невосприимчива к неправильным классификациям и шумным данным. Впрочем, можно также попробовать использовать функцию потерь `'perceptron'`; этот тип функции потерь ведет к кусочно-линейной функции без зазора. Такое решение приемлемо, когда необходимо обратиться к модели с бóльшим смещением, чем у других возможных вариантов функций потерь.

Для получения оптимальных результатов при использовании такого набора кусочно-линейных функций потерь следует принять во внимание два аспекта:

- когда используется любая функция потерь, стохастический градиентный спуск становится ленивым, причем вектор коэффициентов обновляется, только когда пример нарушает ранее определенные зазоры. Это совершенно противоречит логарифмической либо квадратической функции потерь, когда для обновления вектора коэффициентов в действительности рассматривается каждый пример. В случае если в обучении участвует много признаков, такой ленивый подход в результате приводит к более разреженному вектору коэффициентов, тем самым уменьшая переподгонку (более плотный вектор влечет за собой бóльшую переподгонку, потому что некоторые коэффициенты, скорее всего, захватят больше шума, чем сигналы из данных);
- функция `'modified_huber'` – это единственная функция потерь, которая допускает вероятностное оценивание, что (как показывает стохастическая логистическая регрессия) делает ее приемлемой альтернативой для логарифмической функции потерь. Модифицированная функция Хьюбера также показывает более совершенные результаты при обработке мультиклассовых предсказаний по схеме «**один против всех**», поскольку вероятностные результаты множественных моделей лучше, чем свойство стандартных решающих функций кусочно-линейных функций потерь (вероятности работают лучше, чем необработанный результат решающих функций, так как они находятся в одинаковой шкале в интервале от 0 до 1). Эта функция потерь работает путем получения оценочного значения вероятности непосредственно из решающей функции: `(clip(decision_function(X), -1, 1) + 1) / 2`.

Что касается задач регрессии, то класс `SGDRegressor` предлагает два варианта функций потерь для SVM:

```
'epsilon_insensitive'
'squared_epsilon_insensitive'
```

Обе активируют регрессию с линейными опорными векторами, где ошибки (остаток из прогноза) в значении эпсилон игнорируются. Помимо значения эпсилон, функция потерь `epsilon_insensitive` рассматривает ошибку как есть. Функция потерь `squared_epsilon_insensitive` работает схожим образом, хотя здесь ошибка штрафуется больше, так как она возводится в квадрат, причем бóльшие ошибки влияют на построение модели больше.

В обоих случаях критически важно задание правильного гиперпараметра эпсилон ε. В качестве значения по умолчанию Scikit-learn предлагает `epsilon=0.1`, но оптимальное для вашей задачи значение должно быть найдено в результате сеточного поиска, поддерживаемого перекрестной проверкой, как мы убедимся в следующих абзацах.

Отметим, что среди регрессионных функций потерь также имеется функция `'huber'`, которая не активирует свойственную для алгоритма SVM оптимизацию, а просто модифицирует обычную квадратическую функцию потерь `'squared_loss'`, чтобы стать нечувствительной к выбросам, переключившись с квадратичной на линейную меру потерь за пределами значения расстояния параметра эпсилон.

Если говорить о наших примерах, то мы повторим процесс потоковой передачи определенное число раз, чтобы продемонстрировать, как устанавливать разные гиперпараметры и преобразовывать признаки, а для сокращения числа повторяющихся строк программного кода воспользуемся вспомогательными функциями. Кроме того, чтобы ускорить выполнение примеров, мы установим ограничение на число случаев или значения терпимости (допуска), на которые алгоритм ссылается. Таким способом показатели времени тренировки и проверки будут оставаться на минимуме, и любой пример займет не больше, чем время на чашку чая или кофе.

Что касается вспомогательных оберточных функций, то первая из них будет предназначаться для первоначальной потоковой передачи части либо всех данных сразу (предел устанавливается при помощи параметра `max_rows`). После завершения потоковой передачи функция сможет выяснить все уровни категориальных признаков и записать различные интервалы числовых признаков. Как напоминание запись интервалов является важным аспектом, которому следует уделить внимание. Алгоритмы SGD и SVM чувствительны к шкалам разных интервалов, и они работают хуже с числом, лежащим за пределами интервала [–1,1].

На выходе наша функция будет возвращать два натренированных объекта Scikit-learn: `DictVectorizer` (способный преобразовывать присутствующие в словаре интервалы признаков в вектор признаков) и `MinMaxScaler`, который перешкалирует числовые переменные в интервал [0,1] (пригодный для хранения значений в наборе данных в разреженном виде, тем самым поддерживая использование памяти на низком уровне и достигая быстрых вычислений, когда большинство значений равно 0). В качестве уникального ограничения необходимо знать имена признаков числовых и категориальных переменных, которые вы хотите использовать для прогнозной модели. Признаки, не включенные в список для передачи параметрам `binary_features` или `numeric_features`, будут в действительности проигнорированы. Когда поток не имеет имен признаков, вам необходимо дать им имена, используя для этого параметр `fieldnames`:

```
In:
import csv, time, os
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.linear_model import SGDRegressor
from sklearn.feature_extraction import DictVectorizer
from sklearn.preprocessing import MinMaxScaler

def explore(target_file, separator=',', fieldnames= None,
            binary_features=list(), numeric_features=list(),
```

```
        max_rows=20000):
"""
Генерировать из онлайнового потока DictVectorizer и MinMaxScaler.
Параметры
---------
target file = файл, из которого организовать потоковую передачу
separator = символ разграничения полей
fieldnames = метки полей (можно пропустить и читать из файла)
binary_features = список качественных признаков для рассмотрения
numeric_features = список числовых признаков для рассмотрения
max_rows = число строк, читаемых из потока (может быть пустым)
"""
features = dict()
min_max = dict()
vectorizer = DictVectorizer(sparse=False)
scaler = MinMaxScaler()
with open(target_file, 'rb') as R:
    iterator = csv.DictReader(R, fieldnames, delimiter=separator)
    for n, row in enumerate(iterator):
        # ИССЛЕДОВАНИЕ ДАННЫХ
        for k,v in row.iteritems():
            if k in binary_features:
                if k+'_'+v not in features:
                    features[k+'_'+v]=0
            elif k in numeric_features:
                v = float(v)
                if k not in features:
                    features[k]=0
                    min_max[k] = [v,v]
                else:
                    if v < min_max[k][0]:
                        min_max[k][0]= v
                    elif v > min_max[k][1]:
                        min_max[k][1]= v
            else:
                pass # игнорировать признак
        if max_rows and n > max_rows:
            break
vectorizer.fit([features])
A = vectorizer.transform([{f:0
                           if f not in min_max else min_max[f][0]
                           for f in vectorizer.feature_names_},
                          {f:1
                           if f not in min_max else min_max[f][1]
                           for f in vectorizer.feature_names_}])
scaler.fit(A)
return vectorizer, scaler
```

Этот фрагмент исходного кода можно легко использовать повторно для своих собственных приложений машинного обучения на крупномасштабных данных. В случае если поток онлайновый (непрерывная потоковая передача) или слишком длинный, то путем установки параметра `max_rows` можно применить другой предел на число наблюдаемых примеров.

Вторая функция, напротив, будет просто извлекать данные из потока и преобразовывать их в вектор признаков, нормализуя числовые признаки, если вместо значения `None` будет предоставлен подходящий объект `MinMaxScaler`:

```
In:
def pull_examples(target_file, vectorizer,
                  binary_features, numeric_features,
                  target, min_max=None, separator=',',
                  fieldnames=None, sparse=True):
    """
    Читает онлайновый поток и возвращает генератор нормализованных
    векторов признаков
    Параметры
    ----------
    target file = файл, из которого организовать потоковую передачу
    vectorizer = объект DictVectorizer
    binary_features = список качественных признаков для рассмотрения
    numeric_features = список числовых признаков для рассмотрения
    target = метка переменной отклика
    min_max = объект MinMaxScaler, можно пропустить, установив в None
    separator = символ разделения полей
    fieldnames = метки полей (можно пропустить и читать из файла)
    sparse = будет ли возвращен из генератора разреженный вектор
    """
    with open(target_file, 'rb') as R:
        iterator = csv.DictReader(R, fieldnames, delimiter=separator)
        for n, row in enumerate(iterator):
            # ОБРАБОТКА ДАННЫХ
            stream_row = {}
            response = np.array([float(row[target])])
            for k,v in row.iteritems():
                if k in binary_features:
                    stream_row[k+'_'+v]=1.0
                else:
                    if k in numeric_features:
                        stream_row[k]=float(v)
            if min_max:
                features = min_max.transform(
                    vectorizer.transform( [stream_row]))
            else:
                features = vectorizer.transform([stream_row])
            if sparse:
                yield(csr_matrix(features), response, n)
            else:
                yield(features, response, n)
```

Располагая этими двумя функциями, теперь попробуем еще раз смоделировать первую задачу регрессии из предыдущей главы с набором данных Bike-sharing об аренде велосипедов, но на этот раз с использованием кусочно-линейной функции вместо функции среднеквадратических ошибок модели, которую мы использовали ранее.

В качестве первого шага мы передаем в поток имя файла и список качественных и числовых переменных (взятых из заголовка файла и в результате его перво-

начального исследования). Программный код нашей оберточной функции вернет информацию о прямокодированных переменных и интервалах значений. В этом случае большинство переменных будет двоичными единицами, что является идеальной ситуацией для разреженного представления, поскольку большинство значений в наборе данных нулевые:

```
In:
local_path = os.getcwd().decode('cp1251')
source = '\\data\\bikesharing\\hour.csv'

b_vars = ['holiday','hr','mnth', 'season',
          'weathersit','weekday','workingday','yr']
n_vars = ['hum', 'temp', 'atemp', 'windspeed']

std_row, min_max = explore(target_file=local_path+'\\'+source,
                           binary_features=b_vars,
                           numeric_features=n_vars)

print('Признаки: ')
for f,mv,mx in zip(std_row.feature_names_,
                   min_max.data_min_,
                   min_max.data_max_):
    print '%s:[%0.2f,%0.2f] ' % (f,mv,mx)

Out:
Признаки:
atemp:[0.00,1.00]
holiday_0:[0.00,1.00]
holiday_1:[0.00,1.00]
...
workingday_1:[0.00,1.00]
yr_0:[0.00,1.00]
yr_1:[0.00,1.00]
```

Как можно заметить по итоговому результату, качественные переменные были закодированы с использованием их имен и добавления их значения после символа подчеркивания и затем преобразованы в двоичные признаки (которые имеют значение 1, когда признак присутствует, и 0 в противном случае). Отметим, что мы всегда используем модели SGD с параметром `average=True`, чтобы обеспечить более быструю сходимость (это соответствует использованию модели **усредненного стохастического градиентного спуска** (averaged stochastic gradient descent, ASGD), как уже обсуждалось в предыдущей главе):

```
In:
from sklearn.linear_model import SGDRegressor

SGD = SGDRegressor(loss='epsilon_insensitive',
                   epsilon=0.001,
                   penalty=None,
                   random_state=1,
                   average=True)
val_rmse = 0
val_rmsle = 0
predictions_start = 16000

def apply_log(x): return np.log(x + 1.0)
```

```
def apply_exp(x): return np.exp(x) - 1.0

for x,y,n in pull_examples(target_file=local_path+'\\'+source,
                           vectorizer=std_row, min_max=min_max,
                           binary_features=b_vars,
                           numeric_features=n_vars,
                           target='cnt'):
    y_log = apply_log(y)
    # МАШИННОЕ ОБУЧЕНИЕ
    if (n+1) >= predictions_start:
        # ФАЗА С ОТКЛАДЫВАНИЕМ ПОСЛЕ N
        predicted = SGD.predict(x)
        val_rmse += (apply_exp(predicted) - y)**2
        val_rmsle += (predicted - y_log)**2
        if (n-predictions_start+1) % 250 == 0
        and (n+1) > predictions_start:
            print(n),
            print('%s RMSE на отложенных данных: %0.3f,' %
                  (time.strftime('%X'),
                  (val_rmse / float(n-predictions_start+1))**0.5)),
            print('то же самое RMSLE: %0.3f' %
                  ((val_rmsle / float(n-predictions_start+1))**0.5))
    else:
        # ФАЗА ОБУЧЕНИЯ
        SGD.partial_fit(x, y_log)
print('%s ИТОГОВАЯ RMSE на отложенных данных: %0.3f' %
      (time.strftime('%X'),
      (val_rmse / float(n-predictions_start+1))**0.5))
print('%s ИТОГОВАЯ RMSLE на отложенных данных: %0.3f' %
      (time.strftime('%X'),
      (val_rmsle / float(n-predictions_start+1))**0.5))
```

```
Out:
16249 22:11:39 RMSE на отложенных данных: 276.604, то же самое RMSLE: 1.796
16499 22:11:39 RMSE на отложенных данных: 250.419, то же самое RMSLE: 1.706
16749 22:11:39 RMSE на отложенных данных: 250.639, то же самое RMSLE: 1.694
16999 22:11:40 RMSE на отложенных данных: 249.561, то же самое RMSLE: 1.702
17249 22:11:40 RMSE на отложенных данных: 234.840, то же самое RMSLE: 1.640
22:11:40 ИТОГОВАЯ RMSE на отложенных данных: 224.404
22:11:40 ИТОГОВАЯ RMSLE на отложенных данных: 1.594
```

А теперь попробуем решить задачу классификации с набором данных Covertype о лесном покрове:

```
In:
local_path = os.getcwd().decode('cp1251')
source = 'data\\shuffled_covtype.data'

n_vars = ['var_'+'0'*int(j<10)+str(j)
          for j in range(54)]

std_row, min_max = explore(target_file=local_path+'\\'+source,
                           binary_features=list(),
                           fieldnames=n_vars+['covertype'],
                           numeric_features=n_vars,
                           max_rows=50000)
```

```
print('Признаки: ')

for f,mv,mx in zip(std_row.feature_names_,
                   min_max.data_min_,
                   min_max.data_max_):
    print('%s:[%0.2f,%0.2f] ' % (f,mv,mx))
```

```
Out:
Признаки:
var_00:[1871.00,3853.00]
var_01:[0.00,360.00]
var_02:[0.00,61.00]
var_03:[0.00,1397.00]
var_04:[-164.00,588.00]
var_05:[0.00,7116.00]
var_06:[58.00,254.00]
var_07:[0.00,254.00]
var_08:[0.00,254.00]
var_09:[0.00,7168.00]
...
```

После выборки из потока и настройки объектов `DictVectorizer` и `MinMaxScaler` можно запустить процесс обучения, используя на этот раз прогрессивную проверку (меру ошибки получают путем тестирования модели на прецедентах, перед тем как они будут использованы для тренировки), при условии что имеется большое количество примеров. Через каждое определенное число примеров, которое программно установлено в переменной `sample`, сценарий сообщает о ситуации показателем средней точности на самых последних примерах:

```
In:
from sklearn.linear_model import SGDClassifier

SGD = SGDClassifier(loss='hinge',
                    penalty=None,
                    random_state=1,
                    average=True)

accuracy = 0
accuracy_record = list()
predictions_start = 50
sample = 5000
early_stop = 50000

for x,y,n in pull_examples(target_file=local_path+'\\'+source,
                           vectorizer=std_row,
                           min_max=min_max,
                           binary_features=list(),
                           numeric_features=n_vars,
                           fieldnames=n_vars+['covertype'],
                           target='covertype'):
    # ФАЗА ОБУЧЕНИЯ
    if n > predictions_start:
        SGD.partial_fit(x, y, classes=range(1,8))
        accuracy += int(int(SGD.predict(x))==y[0])
        if n % sample == 0:
            accuracy_record.append(accuracy / float(sample))
```

```
            print('%s Прогрессивная точность в примере %i: %0.3f'
                  % (time.strftime('%X'), n,
                     np.mean(accuracy_record[-sample:])))
            accuracy = 0
        if early_stop and n >= early_stop:
            break
```

```
Out:
...
19:23:49 Прогрессивная точность в примере 50000: 0.699
```

Необходимо обработать свыше 575 000 примеров, однако после 50 000 примеров мы делаем в процессе обучения раннюю остановку. Вы свободны изменить эти параметры, исходя из мощности вашего компьютера и наличия времени. Предупреждаем, что выполнение программного кода может занять достаточно продолжительное время. По нашему опыту на процессоре Intel Core i3 с тактовой частотой 2,20 ГГц вычисление заняло приблизительно 30 мин.

Отбор признаков посредством регуляризации

В пакетном контексте общепринято управлять отбором признаков следующим образом:

- предварительная фильтрация на основе полноты данных (распространенность пропущенных значений), дисперсия и высокая мультиколлинеарность между переменными, для того чтобы иметь более чистый набор данных с соответствующими и операбельными признаками;
- еще одна начальная фильтрация на основе одномерной связи (статистическая проверка хи-квадрат, F-значение и простая линейная регрессия) между признаками и переменной отклика для немедленного удаления бесполезных для задачи прогнозирования признаков, так как они мало связаны с откликом или не связаны вообще;
- рекурсивный подход во время моделирования со вставкой и/или исключением признаков, исходя из их способности улучшить прогнозирующую способность тестируемого на отложенной выборке алгоритма. Использование меньшего подмножества только релевантных признаков позволяет алгоритму машинного обучения быть менее подверженным переподгонке из-за шумных переменных и избыточных параметров вследствие высокой размерности признаков.

Применение такого рода подходов в онлайновых условиях, разумеется, по-прежнему возможно, но довольно дорогостояще с точки зрения требующегося времени из-за количества потоковых данных для завершения одиночной модели. Рекурсивные подходы, основанные на большом количестве итераций и проверок, требуют гибкого набора данных, который может уместиться в памяти. Как упоминалось ранее, в этом случае хорошим вариантом была бы подвыборка, что позволит выяснить признаки и модели для применения в более крупном масштабе в дальнейшем.

Если придерживаться внеядерного подхода, то регуляризация является идеальным решением, как способ отбора переменных во время потоковой передачи и фильтрации шумных или избыточных признаков. Регуляризация хорошо работает с онлайновыми алгоритмами, поскольку она действует так, как работает онлайновый алгоритм машинного обучения, и выполняет подгонку своих ко-

эффициентов на примерах без какой-либо потребности запускать для целей отбора другие потоки. Собственно, регуляризация – это просто значение штрафа, которое добавляется к оптимизации процесса обучения. Его значение зависит от коэффициента признака и параметра альфа α, задающего влияние регуляризации. Регуляризационное выравнивание вступает в силу, когда модель обновляет веса коэффициентов. В это время регуляризация действует путем сокращения результирующих весов, если значение обновления недостаточно большое. Эффект исключения или ослабления избыточных переменных достигается за счет параметра регуляризации альфа, который должен быть установлен эмпирически в правильную величину для получения наилучшего результата относительно каждого конкретного элемента данных, который будет усвоен.

Алгоритм SGD реализует те же самые стратегии регуляризации, которые можно найти в пакетных алгоритмах:

- подталкивание к нулю избыточных и не столь существенных переменных за счет штрафа L1-регуляризации;
- уменьшение веса менее важных признаков за счет L2-регуляризации;
- смешивание эффекта от регуляризации L1 и L2 методом эластичной сети.

Регуляризация L1 является идеальной стратегией, когда имеются необычные и избыточные переменные, так как она подталкивает коэффициенты таких признаков к нулю, делая их нерелевантными при вычислении прогноза.

Регуляризация L2 подходит, когда между переменными имеется много корреляций, так как ее стратегия состоит лишь в уменьшении весов тех признаков, вариация которых менее важна для минимизации функции потерь. С L2 все переменные продолжают вносить свой вклад в прогноз, правда, некоторые немного меньше.

Эластичная сеть смешивает L1 и L2, используя взвешенную сумму. Это решение интересно тем, что иногда регуляризация L1 нестабильна во время работы с очень коррелированными переменными, выбирая одну или другую относительно наблюдаемых примеров. Используя `ElasticNet`, многие необычные признаки будут по-прежнему подталкиваться к нулю, как при регуляризации L1, но коррелируемые будут ослабляться, как в регуляризации L2.

Классы `SGDClassifier` и `SGDRegressor` могут реализовывать регуляризацию L1, L2 и методом эластичной сети при помощи параметров `penalty`, `alpha` и `l1_ratio`.

Параметр `alpha` является самым критическим параметром после решения о виде штрафа или смеси из обоих. В идеальном случае подходящие значения можно протестировать в интервале от `0.1` до `10^-7`, воспользовавшись списком значений, создаваемых при помощи `10.0**-np.arange(1,7)`.

Если параметр `penalty` определяет, какой вид регуляризации выбран, то параметр `alpha`, как уже упомянуто, будет определять ее силу. Поскольку `alpha` – это константа, которая умножается на штрафную составляющую, низкие значения этого параметра повлияют на итоговый коэффициент незначительно, тогда как его высокие значения окажут на него сильное влияние. Наконец, коэффициент `l1_ratio` показывает, при `penalty='elasticnet'`, какой процент приходится на L1-штраф относительно L2-штрафа.

Настройка регуляризации с алгоритмом SGD очень простая. Например, можно попробовать изменить предыдущий пример исходного кода, вставив в `SGDClassifier` штраф L2:

```
SGD = SGDClassifier(loss='hinge', penalty='l2', alpha= 0.0001,
                    random_state=1, average=True)
```

Если вы желаете протестировать смешивание эффектов двух подходов регуляризации методом эластичной сети, то вам нужно явно указать только соотношение между L1 и L2, задав коэффициент `l1_ratio`:

```
SGD = SGDClassifier(loss=''hinge'', penalty=''elasticnet'',
                    alpha= 0.001, l1_ratio=0.5, random_state=1,
                    average=True)
```

Поскольку успех регуляризации зависит от включения правильного вида штрафа и наилучшего значения параметра `alpha`, мы посмотрим на регуляризацию в действии на примерах во время решения задачи гиперпараметрической оптимизации.

Добавление нелинейности в алгоритм SGD

Самый быстрый (и, в сущности, не такой уж и сложный) способ добавить нелинейность к линейному SGD-ученику состоит в том, чтобы преобразовать полученный из потока вектор примера в новый вектор, включая степенные преобразования и сочетание признаков вплоть до определенной степени.

Сочетания могут представлять взаимодействия между признаками (объясняя, когда два признака согласованно оказывают особое воздействие на отклик), тем самым помогая включать в линейную модель SVM определенную величину нелинейности. Например, двухстороннее взаимодействие создается в результате перемножения двух признаков, трехстороннее – в результате перемножения трех признаков и т. д., создавая еще более сложные взаимодействия для разложений более высокой степени.

В библиотеке Scikit-learn модуль предобработки содержит класс `PolynomialFeatures`, который может автоматически преобразовывать вектор признаков полиномиальным разложением в нужной степени:

```
In:
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import PolynomialFeatures

local_path = os.getcwd()
source = '\\data\\bikesharing\\hour.csv'

b_vars = ['holiday','hr','mnth', 'season',
          'weathersit','weekday','workingday','yr']
n_vars = ['hum', 'temp', 'atemp', 'windspeed']

std_row, min_max = explore(target_file=local_path+'\\'+source,
                           binary_features=b_vars,
                           numeric_features=n_vars)

poly = PolynomialFeatures(degree=2, interaction_only=False,
                          include_bias=False)
SGD = SGDRegressor(loss='epsilon_insensitive', epsilon=0.001,
                   penalty=None, random_state=1, average=True)

val_rmse = 0
val_rmsle = 0
predictions_start = 16000
```

```
def apply_log(x): return np.log(x + 1.0)
def apply_exp(x): return np.exp(x) - 1.0

for x,y,n in pull_examples(target_file=local_path+'\\'
                           +source,vectorizer=std_row, min_max=min_max,
                           sparse = False, binary_features=b_vars,
                           numeric_features=n_vars, target='cnt'):
    y_log = apply_log(y)

    # Извлечь только качественные признаки и разложить их
    num_index = [j
                 for j, i in enumerate(std_row.feature_names_)
                 if i in n_vars]
    x_poly = poly.fit_transform(x[:,num_index])[:,len(num_index):]
    new_x = np.concatenate((x, x_poly), axis=1)

    # МАШИННОЕ ОБУЧЕНИЕ
    if (n+1) >= predictions_start:
        # ФАЗА С ОТКЛАДЫВАНИЕМ ПОСЛЕ N
        predicted = SGD.predict(new_x)
        val_rmse += (apply_exp(predicted) - y)**2
        val_rmsle += (predicted - y_log)**2
        if (n-predictions_start+1) % 250 == 0
        and (n+1) > predictions_start:
            print(n),
            print('%s RMSE на отложенных данных: %0.3f,'
                  % (time.strftime('%X'),
                    (val_rmse / float(n-predictions_start+1))**0.5)),
            print('то же самое RMSLE: %0.3f'
                  % ((val_rmsle / float(n-predictions_start+1))**0.5))
    else:
        # ФАЗА ОБУЧЕНИЯ
        SGD.partial_fit(new_x, y_log)
print('%s ИТОГОВАЯ RMSE на отложенных данных: %0.3f'
      % (time.strftime('%X'),
        (val_rmse / float(n-predictions_start+1))**0.5))
print('%s ИТОГОВАЯ RMSLE на отложенных данных: %0.3f'
      % (time.strftime('%X'),
        (val_rmsle / float(n-predictions_start+1))**0.5))

Out:
...
21:49:24 ИТОГОВАЯ RMSE на отложенных данных: 219.191
21:49:24 ИТОГОВАЯ RMSLE на отложенных данных: 1.480
```

`PolynomialFeatures` ожидает на входе плотную матрицу, неразреженную. Функция `pull_examples` принимает параметр `sparse`, который обычно устанавливается в `True`, но который может быть установлен в `False`, тем самым возвращая плотную матрицу.

Испытание явных высокоразмерных отображений

Несмотря на то что полиномиальные разложения являются довольно мощным преобразованием, они могут оказаться дорогостоящими в вычислительном плане, когда мы пытаемся выполнить разложение с высокими степенями, и быстро нивелируют положительные эффекты захвата важной нелинейности из-за переподгонки, вызванный сверхпараметризацией (когда имеется слишком много из-

быточных и неполезных признаков). Как уже было отмечено при рассмотрении классов `SVC` и `SVR` библиотеки Scikit-learn, в этих случаях на помощь приходят ядерные преобразования. Ядерные преобразования алгоритма SVM, будучи неявными, для своей работы требуют наличия в оперативной памяти матрицы данных. В библиотеке Scikit-learn имеется класс преобразований, основанный на случайных приближениях, которые в контексте линейной модели могут достигать очень похожих результатов, что и ядерный алгоритм SVM.

Модуль `sklearn.kernel_approximation` содержит несколько таких алгоритмов:

- `RBFSampler`: аппроксимирует карту признаков ядра RBF;
- `Nystroem`: аппроксимирует карту ядра, используя подмножество тренировочных данных;
- `AdditiveChi2Sampler`: аппроксимирует карту признаков для аддитивного ядра хи-квадрат; это ядро используется в компьютерном зрении;
- `SkewedChi2Sampler`: аппроксимирует карту признаков аналогично скошенному ядру хи-квадрат; также используется в компьютерном зрении.

Кроме метода Nystroem, ни один из упомянутых выше классов не требует обучения из выборки данных, что делает их идеально подходящими для онлайнового обучения. Единственное, что они должны знать, – это каким образом вектор с примером сформирован (сколько в нем признаков), и затем они производят большое число случайных нелинейностей, которые, возможно, хорошо впишутся в вашу задачу обработки данных.

В этих алгоритмах аппроксимации нет каких-то сложных оптимизационных алгоритмов, которые требуют объяснения; фактически сама оптимизация заменена рандомизацией, и результаты во многом зависят от числа выходных признаков, на которые указывают параметры `n_components`. Чем больше выходных признаков, тем выше вероятность, что случайно вы получите нужную нелинейность, идеально работающую с вашей задачей.

Важно отметить, что если случай действительно играет очень существенную роль в создании нужных признаков, которые улучшают предсказание, то воспроизводимость результатов приобретает крайне важное значение, и вы должны стремиться ее получить; в противном случае вы не сможете последовательно перетренировывать и тонко настраивать алгоритм одинаковым образом. Примечательно, что каждый класс обеспечен параметром `random_state`, тем самым позволяя управлять генерированием случайного признака и давая возможность воссоздавать его позже на том же компьютере, как и на других.

Теоретические основы таких методов создания признаков объяснены в научных статьях *Random Features for Large-Scale Kernel Machines* («Случайные признаки крупномасштабных ядерных машин»), A. Rahimi и Benjamin Recht (http://www.eecs.berkeley.edu/~brecht/papers/07.rah.rec.nips.pdf) и *Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning* («Взвешенные суммы случайно генерируемых признаков: замена минимизации рандомизацией в обучении»), A. Rahimi и Benjamin Recht (http://www.eecs.berkeley.edu/~brecht/papers/08.rah.rec.nips.pdf).

Для наших целей этого будет достаточно, чтобы знать, как реализовать и применить этот метод для улучшения моделей SGD, как линейных, так и на основе SVM:

```
In:
from sklearn.linear_model import SGDClassifier
from sklearn.kernel_approximation import RBFSampler
```

```
local_path = os.getcwd()
source = 'data\\shuffled_covtype.data'

n_vars = ['var_'+str(j) for j in range(54)]

std_row, min_max = explore(target_file=local_path+'\\'+source,
                           binary_features=list(),
                           fieldnames= n_vars+['covertype'],
                           numeric_features=n_vars,
                           max_rows=50000)

SGD = SGDClassifier(loss='hinge', penalty=None,
                    random_state=1, average=True)
rbf_feature = RBFSampler(gamma=0.5, n_components=300,
                         random_state=0)

accuracy = 0
accuracy_record = list()
predictions_start = 50
sample = 5000
early_stop = 50000

for x,y,n in pull_examples(target_file=local_path+'\\'+source,
                           vectorizer=std_row,
                           min_max=min_max,
                           binary_features=list(),
                           numeric_features=n_vars,
                           fieldnames= n_vars+['covertype'],
                           target='covertype', sparse=False):
    rbf_x = rbf_feature.fit_transform(x)
    # ФАЗА ОБУЧЕНИЯ
    if n > predictions_start:
        accuracy += int(int(SGD.predict(rbf_x))==y[0])
        if n % sample == 0:
            accuracy_record.append(accuracy / float(sample))
            print('%s Прогрессивная точность в примере %i: %0.3f'
                  % (time.strftime('%X'), n,
                     np.mean(accuracy_record[-sample:])))
            accuracy = 0
    if early_stop and n >= early_stop:
        break
    SGD.partial_fit(rbf_x, y, classes=range(1,8))

Out:
...
09:57:45 Прогрессивная точность в примере 50000: 0.707
```

Доводка гиперпараметров

Как и в пакетном обучении, короткие пути при тестировании оптимальных сочетаний гиперпараметров во внеядерных алгоритмах отсутствуют; придется опробовать определенное число их сочетаний, чтобы выяснить возможное оптимальное решение и провести вневыборочный анализ ошибок для оценки результативности сочетаний.

Поскольку в действительности неизвестно, имеет ли задача прогнозирования простую гладкую выпуклую функцию потерь или же более сложную, и неизвестно точно, как гиперпараметры взаимодействуют друг с другом, очень просто застрять в субоптимальном локальном минимуме, в случае если было опробовано недостаточное число сочетаний. К сожалению, на данный момент в Scikit-learn не предлагается никаких специализированных процедур оптимизации для внеядерных алгоритмов. Учитывая с неизбежностью продолжительное время обучения алгоритма SGD на длинном потоке, доводка гиперпараметров с использованием таких методов может фактически стать узким местом во время построения модели на собственных данных.

Здесь мы представим несколько эмпирических правил, которые могут помочь сэкономить время и усилия и достигнуть наилучших результатов.

Прежде всего можно настроить параметры в окне или выборке из данных, которые смогут уместиться в памяти. Как мы убедились на примере с ядерными машинами SVM, работа с резервуарной выборкой довольно быстрая, даже если поток огромен. Затем можно выполнить оптимизацию в оперативной памяти и использовать найденные в потоке оптимальные параметры.

Леон Ботту (Léon Bottou) из Microsoft Research в своем техническом отчете *Stochastic Gradient Descent Tricks* («Приемы стохастического градиентного спуска») отметил:

> «*Математика стохастического градиентного спуска удивительно независима от размера тренировочного набора*».

Это относится ко всем основным параметрам, и в особенности к темпу обучения; темп обучения, который хорошо работает с выборкой, будет лучше работать с полными данными. Кроме того, идеальное число прохождений по данным большей частью можно угадать, попробовав выполнить схождение на небольшом выборочном наборе данных. В качестве практического ориентира мы предлагаем показательное число `10**6` исследуемых алгоритмом примеров – как указывается в документации Scikit-learn, это число мы часто находили точным, хотя идеальное число итераций может меняться в зависимости от параметров регуляризации.

Хотя при использовании алгоритма SGD бóльшая часть работы может быть сделана в относительно малом масштабе, мы должны определиться с тем, как решать проблему фиксации многочисленных параметров. Традиционно чаще всего использовались подходы на основе ручного и сеточного поисков, при этом сеточный поиск решал проблему, систематически тестируя все сочетания возможных параметров в значительных значениях (используя, к примеру, проверку в логарифмической шкале с возведением в разную степень: 10 или 2).

Совсем недавно Джеймс Бергстра (James Bergstra) и Джошуа Бенджио (Yoshua Bengio) в своей статье *Random Search for Hyper-Parameter Optimization* («Случайный поиск для гиперпараметрической оптимизации») продемонстрировали другой подход, основанный на случайной выборке значений гиперпараметров. Такой подход, хотя и опирается на случайный выбор, по результатам часто эквивалентен сеточному поиску (но при этом требует меньшего количества прогонов), когда число гиперпараметров низкое и может превысить производительность систематического поиска, когда параметров много и не все они релевантны для эффективности алгоритма.

Мы предоставляем читателю право найти дополнительные причины, почему этот простой и привлекательный подход работает так хорошо в теории, отослав к ранее упомянутой статье Беркстры и Бенджио. Испытав на практике его превосходство относительно других подходов, в следующем фрагменте программного кода мы предлагаем подход, который в Scikit-learn работает хорошо для потоков на основе функции `ParameterSampler`. Функция `ParameterSampler` способна случайным образом отбирать (из функций распределения и списков дискретных значений) разные наборы гиперпараметров, которые позже применяются к обучающемуся алгоритму SGD посредством метода `set_params`:

```
In:
from sklearn.linear_model import SGDRegressor
from sklearn.model_selection import ParameterSampler

local_path = os.getcwd().decode('cp1251')
source = '\\data\\bikesharing\\hour.csv'

b_vars = ['holiday','hr','mnth', 'season',
          'weathersit','weekday','workingday','yr']
n_vars = ['hum', 'temp', 'atemp', 'windspeed']

std_row, min_max = explore(target_file=local_path+'\\'+source,
                           binary_features=b_vars,
                           numeric_features=n_vars)

val_rmse = 0
val_rmsle = 0
predictions_start = 16000
tmp_rsmle = 10**6

def apply_log(x): return np.log(x + 1.0)
def apply_exp(x): return np.exp(x) - 1.0

param_grid = {'penalty':['l1', 'l2'],
              'alpha': 10.0**-np.arange(2,5)}

random_tests = 3
search_schedule = list(ParameterSampler(param_grid,
                                        n_iter=random_tests,
                                        random_state=5))
results = dict()

for search in search_schedule:
    SGD = SGDRegressor(loss='epsilon_insensitive',
                       epsilon=0.001,
                       penalty=None,
                       random_state=1,
                       average=True)
    params = SGD.get_params()
    new_params = {p:params[p]
                  if p not in search else search[p]
                  for p in params}
    SGD.set_params(**new_params)
    print str(search)[1:-1]
    for iterations in range(200):
```

```
        for x,y,n in pull_examples(target_file=local_path+'\\'+source,
                                   vectorizer=std_row,
                                   min_max=min_max,
                                   sparse = False,
                                   binary_features=b_vars,
                                   numeric_features=n_vars,
                                   target='cnt'):
            y_log = apply_log(y)

            # МАШИННОЕ ОБУЧЕНИЕ
            if (n+1) >= predictions_start:
                # ФАЗА С ОТКЛАДЫВАНИЕМ ПОСЛЕ N
                predicted = SGD.predict(x)
                val_rmse += (apply_exp(predicted) - y)**2
                val_rmsle += (predicted - y_log)**2
            else:
                # ФАЗА ОБУЧЕНИЯ
                SGD.partial_fit(x, y_log)

        examples = float(n-predictions_start+1) * (iterations+1)
        print_rmse = (val_rmse / examples)**0.5
        print_rmsle = (val_rmsle / examples)**0.5
        if iterations == 0:
            print('Итерация %i - RMSE: %0.3f - RMSE: %0.3f'
                  % (iterations+1, print_rmse, print_rmsle))
        if iterations > 0:
            if tmp_rmsle / print_rmsle <= 1.01:
                print('Итерация %i - RMSE: %0.3f - RMSE: %0.3f\n'
                      % (iterations+1, print_rmse, print_rmsle))
                results[str(search)]= {'rmse':float(print_rmse),
                                       'rmsle':float(print_rmsle)}
                break
        tmp_rmsle = print_rmsle
```

```
Out:
'penalty': 'l2', 'alpha': 0.0001
Итерация 1 - RMSE: 216.217 - RMSE: 1.440
Итерация 20 - RMSE: 151.872 - RMSE: 0.856

'penalty': 'l1', 'alpha': 0.001
Итерация 1 - RMSE: 712.781 - RMSE: 4.090
Итерация 31 - RMSE: 184.582 - RMSE: 1.053

'penalty': 'l1', 'alpha': 0.0001
Итерация 1 - RMSE: 1050.206 - RMSE: 6.039
Итерация 36 - RMSE: 217.732 - RMSE: 1.252
```

В приведенном выше примере используется тот факт, что набор данных Bike-sharing об аренде велосипедов довольно небольшой и не требует выборки. В других контекстах целесообразно сначала ограничить число рассматриваемых строк либо создать меньшую выборку посредством резервуарного отбора или других потоковых методов отбора, встречавшихся до сих пор. Если требуется исследовать оптимизацию более углубленно, то можно изменить переменную `random_tests`, зафиксировав число тестируемых сочетаний отобранных гиперпараметров. Затем следует модифицировать условие `if tmp_rmsle / print_rmsle <= 1.01`, используя число ближе

к `1.0` – может, даже само число `1.0`, – тем самым давая алгоритму полностью сходиться, пока не будет получен возможный прирост в прогнозирующей способности.

Хотя рекомендуется вместо взятия значений из списков использовать функции распределения, вы по-прежнему можете соответствующим образом использовать предложенные ранее интервалы гиперпараметров, просто увеличив число значений, которые, возможно, будут отобраны из списков. Например, для параметра `alpha` в регуляризации L1 и L2 можно использовать функцию библиотеки NumPy `arrange` с малым шагом, как, например, `10.0**-np.arange(1, 7, step=0.1)`, либо `logspace` с высоким числом для параметра `num`: `1.0/np.logspace(1,7, num=50)`.

Другие альтернативы быстро обучающихся реализаций SVM

Хотя библиотека Scikit-learn предлагает достаточно много инструментов и алгоритмов для внеядерного обучения, среди бесплатного программного обеспечения существуют другие интересные альтернативы. Некоторые опираются на те же самые библиотеки, которые используются в самом Scikit-learn, в частности Liblinear/SBM; другие же абсолютно новые, такие как Sofia-ml, LASVM и Vowpal Wabbit. Например, Liblinear/SBM основана на выборочно-блочной минимизации и реализована как ветвление `liblinear-cdblock` исходной библиотеки (https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/#large_linear_classification_when_data_cannot_fit_in_memory). Библиотека Liblinear/SBM способна подбирать нелинейные модели SVM на больших объемах данных, которые не умещаются в оперативной памяти, задействуя прием, когда для тренировки ученика используются новые выборки из данных и каждая из них затем смешивается с предыдущими выборками, уже используемыми для минимизации (отсюда и термин *блочный*, от англ. blocked, в названии алгоритма).

Еще одна альтернативная библиотека SofiaML (https://code.google.com/archive/p/sofia-ml/) основывается на онлайновом алгоритме оптимизации модели SVM под названием Pegasos SVM. Этот алгоритм является онлайновой аппроксимацией SVM, так же, как и другой программный продукт под названием LaSVM, созданный Леоном Ботту (http://leon.bottou.org/projects/lasvm). Все эти решения могут работать с разреженными данными, в особенности текстовыми, и решать задачи регрессии, классификации и ранжирования. До настоящего времени никакое протестированное нами альтернативное решение не оказалось столь же быстрым и универсальным, как Vowpal Wabbit. Этот программный продукт будет представлен в следующих разделах и будет использоваться для демонстрации того, каким образом можно интегрировать внешние программы в среде Python.

Нелинейность и быстродействие с Vowpal Wabbit

Проект с открытым исходным кодом **Vowpal Wabbit (VW)**, предлагающий быстродействующего онлайнового ученика, был выпущен в 2007 г. Джоном Лэнгфордом (John Langford), Лихонг Ли (Lihong Li) и Алексом Штрелем (Alex Strehl) из Yahoo! Research (http://hunch.net/?p=309) и затем последовательно спонсировался Microsoft Research, так как Джон Лэнгфорд стал главным исследователем в Microsoft. За эти годы проект получил дальнейшее развитие, достигнув сегодня версии 8.1.0[1] с почти сотней работающих над ним участников. (Имеется интерес-

[1] Ко времени работы над переводом настоящей главы (июнь 2017 г.) последней была версия 8.3.0.9. – *Прим. перев.*

ное видео, в котором процесс разработки и внесенные в проект усилия разработчиков визуализируются во времени, используя программу **Gource**, – https://www.youtube.com/watch?v=-aXelGLMMgk). По сей день проект VW по-прежнему находится в стадии постоянной доработки, и его обучающие способности продолжают наращиваться после каждой итерации разработки.

Поразительная характеристика ученика VW состоит в том, что он работает очень быстро, по сравнению с другими имеющимися решениями (LIBLINEAR, Sofia-ml, svmsgd и Scikit-learn). Его секрет прост и одновременно чрезвычайно эффективен: он может загружать данные и обучаться им одновременно. Асинхронный процесс (thread) выполняет разбор прибывающих примеров, в то время как несколько обучающихся процессов работает над непересекающимся набором признаков, тем самым гарантируя высокую вычислительную эффективность, несмотря на то что в результате разбора создаются высокоразмерные признаки (как, например, квадратическое или кубическое полиномиальное разложение). В большинстве случаев реальным узким местом процесса обучения является пропускная способность передачи поступающих в ученика VW данных с диска или из Сети.

Обучающийся алгоритм VW может решать задачу классификации (даже мультиклассовую и многозначную), задачу регрессии (регрессию на основе обычного МНК и квантильную регрессию) и активные задачи обучения, предлагая обширный диапазон сопровождающих инструментов обучения (называемых редукциями), в частности матричную факторизацию, **латентное размещение Дирихле** (LDA), нейронные сети, *n*-граммы для языковых моделей и бутстрапирование.

Инсталляция программы VW

Программу VW можно получить из онлайнового версионного репозитория GitHub (https://github.com/JohnLangford/vowpal_wabbit), где ее можно клонировать при помощи Git или же просто скачать в форме упакованного zip-файла. Будучи разработанной в системах Linux, ее легко скомпилировать в любой среде POSIX простой последовательностью команд make и make install. Подробные инструкции по инсталляции приводятся непосредственно на странице инсталляции программы, откуда можно скачать прекомпилированные бинарники Linux непосредственно от автора (https://github.com/JohnLangford/vowpal_wabbit/wiki/Download).

Версию программы VW, работающую под операционными системами Windows, к сожалению, будет получить немного труднее. Чтобы ее создать, прежде всего нужно обратиться непосредственно к самой документации по библиотеке VW, где процедура компиляции подробно объяснена (https://github.com/JohnLangford/vowpal_wabbit/blob/master/README.windows.txt).

На сопутствующем книге веб-сайте и среди прилагаемых к книге исходных кодов примеров мы предлагаем 32- и 64-разрядные бинарники Windows библиотеки VW версии 8.1.0, которые мы использовали во время работы над книгой.

Анализ формата данных программы VW

Программа VW может работать со специальным форматом данных и вызываться из оболочки. Джон Лэнгфорд использует следующий ниже демонстрационный набор данных для своего онлайнового учебного руководства (https://github.com/JohnLangford/vowpal_wabbit/wiki/Tutorial), который описывает три дома, чьи крыши могут быть заменены. Мы считаем этот пример интересным и предлагаем его с комментариями:

```
In:
with open('data\\house_dataset','wb') as W:
    W.write("0 | price:.23 sqft:.25 age:.05 2006\n")
    W.write("1 2 'second_house | price:.18 sqft:.15 age:.35 1976\n")
    W.write("0 1 0.5 'third_house | price:.53 sqft:.32 age:.87 1924\n")

with open('data\\house_dataset','rb') as R:
    for line in R:
        print(line.strip())

Out:
0 | price:.23 sqft:.25 age:.05 2006
1 2 'second_house | price:.18 sqft:.15 age:.35 1976
0 1 0.5 'third_house | price:.53 sqft:.32 age:.87 1924
```

Первый заметный аспект этого формата файла состоит в том, что он не имеет строки заголовка. Это объясняется тем, что в проекте VW для размещения признаков в разреженном векторе используется хэширование признаков, и, следовательно, ненужные признаки известны заранее. Блоки данных разделены конвейерным символом («|») на пространства имен, которые задуманы как различные признаковые кластеры, каждый из которых содержит один или несколько признаков.

Первое пространство имен всегда содержит переменную отклика. Отклик может быть вещественным (или целым) числом, указывающим на числовое значение, которое будет регрессироваться, бинарный класс либо один из множества классов. Отклик – это всегда первое число в строке. Бинарный класс кодируется при помощи `1` для положительного и `-1` для отрицательного класса (использование 0 в качестве отклика разрешено только для регрессии). Многочисленные классы должны быть последовательно пронумерованы, начиная с `1`, причем наличие разрывов в числах не желательно, потому что VW запрашивает последний класс и рассматривает все целые числа между `1` и последним.

Число сразу после значения отклика является весом (которое говорит, следует ли рассматривать пример как множественный пример или как составную часть примера), затем идет базис, который играет роль первоначального прогноза (своего рода смещение). Наконец, идет метка, предваряемая символом апострофа (`'`); она может быть числом или текстом, которые позже можно найти в результатах VW на выходе (в прогнозе имеется идентификатор для каждого оценочного значения). Вес, базис и метки не обязательны: если их нет, то в качестве веса будет подставлена `1`, а базис и метка не будут иметь значения.

После первого пространства имен можно добавить столько пространств имен, сколько нужно, маркируя каждое числом или строковым значением. Чтобы считаться меткой пространства имен, она должна идти сразу за конвейерным символом, например `|label`.

После метки пространства имен можно поименно добавлять любой признак. Имя признака может быть чем угодно, но должно содержать конвейерный символ или двоеточие. В пространстве имен можно разместить целые тексты, и каждое слово будет рассматриваться как признак. Каждый признак будут считаться равным `1`. Если нужно присвоить другое число, то следует добавить двоеточие в конец имени признака и разместить его значение после него.

Например, вот допустимая в Vowpal Wabbit строка:

```
0 1 0.5 'third_house | price:.53 sqft:.32 age:.87 1924
```

В первом пространстве имен отклик равен `0`, вес примера равен `1`, базис – `0.5` и его метка – `third_house`. Пространство имен является безымянным и состоит из четырех признаков, а именно `price` (`.53`), `sqft` (`.32`), `age` (`.87`) и `1924` (значение равно `1`).

Если в примере имеется признак, который в другом примере отсутствует, алгоритм решит, что значение признака во втором примере равно нулю. Поэтому такой признак, как `1924`, в приведенном выше примере может служить как бинарная переменная, которой, когда она присутствует, автоматически присваивается `1`, а когда отсутствует – `0`. Это также говорит о том, как программа VW обрабатывает пропущенные значения, – она автоматически рассматривает их как `0`.

Вы можете легко обрабатывать пропущенные значения, ставя новый признак, когда значение отсутствует. К примеру, если признак является возрастом (age), то можно добавить новый признак age_missing, который будет бинарной переменной со значением 1. При оценке коэффициентов эта переменная будет действовать как оценщик отсутствующего значения. На веб-сайте автора можно также найти валидатор входных данных, проверяющий, что входные данные отформатированы правильно для передачи в VW, и отображающий процесс их интерпретации этим программным продуктом: http://hunch.net/~vw/validate.html.

Интеграция с Python

Существует несколько библиотек, которые интегрируют программу VW в Python (**vowpal_porpoise**, **Wabbit Wappa** либо **pyvw**). Их инсталляция в системах Linux выполняется просто, но намного тяжелее в Windows. Не важно, работаете вы с Jupyter или IDE, самый простой способ интегрировать VW со сценариями Python состоит в том, чтобы задействовать функцию `Popen` из библиотеки `subprocess`. Она заставляет VW выполняться параллельно с Python. Python просто ожидает, когда программа VW завершит свою работу, захватывает результат и печатает его на экране:

```
In:
import subprocess

def execute_vw(parameters):
    execution = subprocess.Popen('data\\vw '+parameters,
                                 shell=True,
                                 stderr=subprocess.PIPE)
    line = ""
    history = ""
    while True:
        out = execution.stderr.read(1)
        history += out
        if out == '' and execution.poll() != None:
            print('------------ РАБОТА ЗАВЕРШЕНА ------------\n')
            break
        if out != '':
            line += out
            if '\n' in line[-2:]:
                print(line[:-2])
                line = ''
    return history.split('\r\n')
```

Эта функция возвращает список результатов процесса обучения, упрощая его обработку, при этом извлекается соответствующая информация, допускающая повторное использование (как, например, мера ошибки). В качестве предвари-

тельного условия для его корректного функционирования поместите исполняемый файл VW (файл `vw.exe`) в рабочий каталог Python или системный путь, где он может быть найден.

Вызвав функцию на ранее записанном наборе данных о домах, можно взглянуть на то, как все работает и какие результаты получаются на выходе:

```
In:
params = "data\\house_dataset"
results = execute_vw(params)

Out:
Num weight bits = 18
learning rate = 0.5
initial_t = 0
power_t = 0.5
using no cache
Reading datafile = house_dataset
num sources = 1
average  since         example        example  current  current  current
loss     last          counter         weight    label  predict features
0.000000 0.000000            1            1.0   0.0000   0.0000        5
0.666667 1.000000            2            3.0   1.0000   0.0000        5

finished run
number of examples per pass = 3
passes used = 1
weighted example sum = 4.000000
weighted label sum = 2.000000
average loss = 0.750000
best constant = 0.500000
best constant's loss = 0.250000
total feature number = 15
------------ РАБОТА ЗАВЕРШЕНА ------------
```

Начальные строки результата просто напоминают об использованных параметрах и служат в качестве подтверждения о том, какой файл данных использовался. Самая интересная часть – это прогрессивный отчет с числом переданных потоком примеров (выраженным числом в степени 2, т. е. пример 1, 2, 4, 8, 16 и т. д.). Относительно функции потерь в отчете показана средняя мера потерь, прогрессивный отчет для первой итерации на основе отложенных данных, мера потерь которого обозначается отнесенной в конец буквой h (если откладывание данных исключено, то имеется возможность только сообщать о внутривыборочной мере). В столбце `example weight` сообщается о весе примера, и затем пример описывается как `current label`, `current predict` (т. е. текущая метка и текущее предсказание), и на экран выводится число найденных на этой строке признаков (`current features`). Вся эта информация должна помочь следить за потоком и процессом обучения.

После того как обучение завершено, выводится несколько итоговых мер. Средняя потеря является самой важной, в частности когда используется отложенная выборка. Этот показатель полезнее всего использовать для сравнения, так как его можно сразу сравнить с показателем `best constant's loss` (мера потерь или базовая прогнозирующая способность простой константы) и в разных прогонах с разными параметрическими конфигурациями.

Еще одна очень полезная приготовленная нами функция для интеграции программы VW и Python связана с автоматическим преобразованием CSV-файлов в файлы данных VW. Ее можно увидеть в следующем ниже фрагменте исходного кода. Она поможет воспроизвести рассмотренные ранее задачи на основе наборов данных Bike-sharing и Covertype, на этот раз при помощи программы VW. Этой функцией можно легко воспользоваться повторно для своих собственных проектов:

```
In:
import csv

def vw_convert(origin_file, target_file, binary_features,
               numeric_features, target, transform_target=lambda(x):x,
               separator=',', classification=True, multiclass=False,
               fieldnames= None, header=True, sparse=True):
    """
    Читает онлайновый поток и возвращает генератор нормализованных векторов признаков
    Параметры
    ----------
    target file = файл, из которого организовать потоковую передачу
    vectorizer = объект DictVectorizer
    binary_features = список качественных признаков для рассмотрения
    numeric_features = список числовых признаков для рассмотрения
    target = метка переменной отклика
    min_max = объект MinMaxScaler, можно пропустить, установив в None
    separator = символ разделения полей
    fieldnames = метки полей (можно пропустить и читать из файла)
    sparse = будет ли возвращен из генератора разреженный вектор
    """
    with open(target_file, 'wb') as W:
        with open(origin_file, 'rb') as R:
            iterator = csv.DictReader(R, fieldnames,
                                      delimiter=separator)
            for n, row in enumerate(iterator):
                if not header or n>0:
                # ОБРАБОТКА ДАННЫХ
                    response = transform_target(float(row[target]))
                    if classification and not multiclass:
                            if response == 0:
                                stream_row = '-1 '
                            else:
                                stream_row = '1 '
                    else:
                        stream_row = str(response)+' '
                    quantitative = list()
                    qualitative  = list()
                    for k,v in row.iteritems():
                        if k in binary_features:
                            qualitative.append(str(k)+'_'+str(v)+':1')
                        else:
                            if k in numeric_features
                            and (float(v)!=0 or not sparse):
                                quantitative.append(str(k)+':'+str(v))
                    if quantitative:
                        stream_row += '|n '+' '.join(quantitative)
```

```
    if qualitative:
        stream_row += '|q ' + ' '.join(qualitative)
    W.write(stream_row+'\n')
```

Несколько примеров с использованием редукций для алгоритма SVM и нейронных сетей

Программа VW работает по принципу минимизации общей функции стоимости, которая выглядит следующим образом:

$$\lambda_1||w||_1 + \frac{\lambda_2}{2}||w||_2^2 + \sum_{i=1}^{n}\text{loss}(x_i, y_i, w).$$

Как и в других встречавшихся ранее формулах, w – это вектор коэффициентов, при этом оптимизация достигается отдельно для каждого x_i и y_i согласно выбранной функции потерь (обычный МНК, логистическая или кусочно-линейная). λ_1 и λ_2 – это параметры регуляризации, которые по умолчанию равны нулю, но которые можно задать при помощи опций `--l1` и `--l2` в командной строке VW.

С учетом такой базовой структуры VW со временем становился сложнее и совершеннее благодаря парадигме редукций. Редукция – это просто способ снова использовать существующий алгоритм для решения новых задач, не программируя новых алгоритмов решений с нуля. Другими словами, если есть сложная задача машинного обучения A, то она просто сводится к задаче B. Решение B наводит на мысль о решении A. Внедрение этого способа также обусловлено растущим интересом к машинному обучению и взрывному числу задач, которые невозможно решить созданием огромного числа новых алгоритмов. Этот интересный подход, усиливающий существующие возможности, предлагаемые основными алгоритмами, стал одной из главных причин, почему за прошедшие годы приложимость VW получила свое развитие, хотя программа осталась довольно компактной. Если вы интересуетесь этим подходом, то можно взглянуть на следующие два учебных руководства от Джона Лэнгфорда: http://hunch.net/~reductions_tutorial/ и http://hunch.net/~jl/projects/reductions/reductions.html.

В качестве другой иллюстрации мы кратко представим пару редукций для реализации SVM с ядром `RBF` и мелкой нейронной сети при помощи программы VW в чисто внеядерном стиле, т. е. вне основной памяти компьютера. Для этого мы воспользуемся несколькими миниатюрными наборами данных.

Ниже приведен набор данных Iris, модифицированный для задачи бинарной классификации с целью распознавания цветков ириса Iris Versicolor, Iris Setosa и Iris Virginica (ирис разноцветный, щетинистый и виргинский):

```
In:
from random import seed
import numpy as np
from sklearn.datasets import load_iris, load_boston

iris = load_iris()
seed(2)
re_order = np.random.permutation(len(iris.target))

with open('data\\iris_versicolor.vw','wb') as W1:
    for k in re_order:
        y = iris.target[k]
```

```
        X = iris.values()[1][k,:]
        features = ' |f '+' '.join([a+':'+str(b)
                                    for a,b in zip(map(lambda(a):
                                                       a[:-5].replace(' ','_'),
                                                       iris.feature_names),X)])
        target = '1' if y==1 else '-1'
        W1.write(target+features+'\n')
```

Затем для задачи регрессии мы воспользуемся набором данных Boston с ценами на жилую недвижимость:

```
In:
boston = load_boston()
seed(2)
re_order = np.random.permutation(len(boston.target))

with open('data\\boston.vw','wb') as W1:
     for k in re_order:
        y = boston.target[k]
        X = boston.data[k,:]
        features = ' |f '+' '.join([a+':'+str(b)
                                    for a,b in zip(map(lambda(a):
                                                       a[:-5].replace(' ','_'),
                                                       iris.feature_names),X)])
        W1.write(str(y)+features+'\n')
```

Сначала мы опробуем алгоритм SVM. Редукция `kvsm` основана на алгоритме LaSVM (*Быстрые ядерные классификаторы с онлайновым и активным обучением* – http://www.jmlr.org/papers/volume6/bordes05a/bordes05a.pdf) без постоянного смещения. Версия VW, как правило, выполняется всего за одно прохождение и с 1–2 переработками (reprocessing) произвольно отбираемого опорного вектора (хотя некоторые задачи, возможно, потребуют многократных прохождений и переработок). В нашем случае мы используем всего одно прохождение и пару переработок, чтобы аппроксимировать ядро RBF на нашей бинарной задаче (опция `kvsm` работает только для задачи классификации). Реализованы ядра линейное, радиально-базисная функция и полиномиальное. Для того чтобы все работало, используйте опцию `--ksvm`, задайте при помощи `--reprocess` число переработок (по умолчанию оно равно 1), выберите ядро при помощи `--kernel` (варианты `linear`, `poly` и `rbf`). Затем, если ядро полиномиальное, задайте целое число для степени `--degree` или число с плавающей точкой (по умолчанию 1.0) для пропускной способности `--bandwidth`, если используется RBF. Кроме того, необходимо обязательно определить регуляризацию L2, в противном случае редукция не будет работать должным образом. В нашем примере мы задаем ядро `RBF` с пропускной способностью 0.1:

```
In:
params = '--ksvm --l2 0.000001 --reprocess 2 -b 18 --kernel rbf
--bandwidth=0.1 -p data\\iris_bin.test -d data\\iris_versicolor.vw'
results = execute_vw(params)

import numpy as np

def sigmoid(x):
    return 1. / (1. + np.exp(-x))
```

```
accuracy = 0

with open('data\\iris_bin.test', 'rb') as R:
    with open('data\\iris_versicolor.vw', 'rb') as TRAIN:
        holdouts = 0.0
        for n,(line, example) in enumerate(zip(R,TRAIN)):
            if (n+1) % 10==0:
                predicted = float(line.strip())
                y = float(example.split('|')[0])
                accuracy += np.sign(predicted)==np.sign(y)
                holdouts += 1
print('Точность на отложенных данных: %0.3f'
      % ((accuracy / holdouts)**0.5))

Out:
Точность на отложенных данных: 0.966
```

Нейронные сети – это еще одно замечательное дополнение к возможностям проекта VW; благодаря работе Пола Минейро (Paul Mineiro) (http://www.machinedlearnings.com/2012/11/unpimp-your-sigmoid.html) программа VW может реализовывать одноуровневую нейронную сеть с гиперболической тангенсной (*tanh*) функцией активации и, факультативно, с прореживанием весов (используя опцию `--dropout`). Несмотря на то что имеется возможность определять только число нейронов, нейронная редукция отлично справляется как с задачами регрессии, так и классификации и может гладко принимать другие преобразования VW в виде входных данных (такие как квадратичные переменные и *n*-граммы), делая этот программный продукт очень хорошо интегрированным, универсальным (нейронные сети могут решать довольно много задач) и быстрым решением. В нашем примере мы применяем редукцию к набору данных Boston, используя пять нейронов и прореживание:

```
In:
params = 'data\\boston.vw -f data\\boston.model --loss_function squared -k
--cache_file data\\cache_train.vw --passes=20 --nn 5 --dropout'
results = execute_vw(params)

params = '-t data\\boston.vw -i data\\boston.model -k --cache_file data\\cache_test.vw -p
data\\boston.test'
results = execute_vw(params)

val_rmse = 0

with open('data\\boston.test', 'rb') as R:
    with open('data\\boston.vw', 'rb') as TRAIN:
        holdouts = 0.0
        for n,(line, example) in enumerate(zip(R,TRAIN)):
            if (n+1) % 10==0:
                predicted = float(line.strip())
                y = float(example.split('|')[0])
                val_rmse += (predicted - y)**2
                holdouts += 1
print('RMSE на отложенных данных: %0.3f'
      % ((val_rmse / holdouts)**0.5))

Out:
RMSE на отложенных данных: 7.010
```

Ускоренный набор данных Bike-sharing

Давайте испытаем программу VW на ранее созданном демонстрационном файле Bike-sharing с данными об аренде велосипедов для объяснения выходных компонентов. В качестве первого шага необходимо преобразовать CSV-файл в VW-файл, и для этого ранее приведенная функция `vw_convert` окажется весьма кстати. Как и прежде, мы применим к числовому отклику логарифмическое преобразование, воспользовавшись функцией `apply_log`, передаваемой параметром `transform_target` функции `vw_convert`:

```
In:
import os
import numpy as np

def apply_log(x):
    return np.log(x + 1.0)

def apply_exp(x):
    return np.exp(x) - 1.0

local_path = os.getcwd().decode('cp1251')
source = '\\data\\bikesharing\\hour.csv'

origin = target_file=local_path+'\\'+source
target = target_file=local_path+'\\'+'data\\bike.vw'

b_vars = ['holiday','hr','mnth', 'season',
          'weathersit','weekday','workingday','yr']
n_vars = ['hum', 'temp', 'atemp', 'windspeed']

vw_convert(origin, target,
           binary_features=b_vars,
           numeric_features=n_vars,
           target = 'cnt',
           transform_target=apply_log,
           separator=',',
           classification=False,
           multiclass=False,
           fieldnames= None,
           header=True)
```

Через несколько секунд должен быть готов новый файл. Мы можем сразу выполнить расчеты, которые представляют собой простую линейную регрессию (эта опция задана в VW по умолчанию). Обучение предположительно будет выполняться в течение 100 прохождений и контролироваться вневыборочной проверкой, которая в VW реализована автоматически (систематически и непрерывно выбирая в качестве проверочных данных одно наблюдение из каждых 10). В этом случае мы решаем задать отложенную выборку после 16 000 примеров (используя опцию `--holdout_after`). Когда проверочная ошибка на перекрестной проверке увеличивается (а не уменьшается), программа VW останавливается после нескольких итераций (по умолчанию три, но это число может быть изменено при помощи опции `--early_terminate`), избегая переподгонки (избыточной аппроксимации данных):

```
In:
params = 'data\\bike.vw -f data\\regression.model -k --cache_file data\\cache_train.vw
--passes=1000 --hash strings --holdout_after 16000'
```

```
results = execute_vw(params)
```

```
Out:
...
finished run
number of examples per pass = 15999
passes used = 6
weighted example sum = 95994.000000
weighted label sum = 439183.191893
average loss = 0.427485 h
best constant = 4.575111
total feature number = 1235898
------------ РАБОТА ЗАВЕРШЕНА ------------
```

Итоговый отчет показывает, что были завершены шесть прохождений (из 100 возможных), и вневыборочная средняя потеря составила 0.428. Поскольку нас интересует квадратный корень из среднеквадратической ошибки `RMSE` и квадратный корень из среднеквадратической логарифмической ошибки `RMSLE`, мы должны вычислить их сами.

Затем мы записываем результаты предсказания в файл (`pred.test`), чтобы потом их прочесть и вычислить меру ошибки, используя ту же схему с откладыванием данных, что и в тренировочном наборе. Результаты действительно намного лучше (в долях времени), чем те, которые мы получили ранее с алгоритмом SGD библиотеки Scikit-learn:

```
In:
params = '-t data\\bike.vw -i data\\regression.model -k --cache_file data\\cache_test.vw -p
data\\pred.test'
results = execute_vw(params)
```

```
val_rmse = 0
val_rmsle = 0
```

```
with open('data\\pred.test', 'rb') as R:
    with open('data\\bike.vw', 'rb') as TRAIN:
        holdouts = 0.0
        for n,(line, example) in enumerate(zip(R,TRAIN)):
            if n > 16000:
                predicted = float(line.strip())
                y_log = float(example.split('|')[0])
                y = apply_exp(y_log)
                val_rmse += (apply_exp(predicted) - y)**2
                val_rmsle += (predicted - y_log)**2
                holdouts += 1
print('RMSE на отложенных данных: %0.3f' % ((val_rmse / holdouts)**0.5))
print('RMSLE на отложенных данных: %0.3f' % ((val_rmsle / holdouts)**0.5))
```

```
Out:
RMSE на отложенных данных: 135.306
RMSLE на отложенных данных: 0.845
```

Переработка набора данных Covertype в VW

Задача с набором данных Covertype в программе VW тоже решается лучше и легче, чем получалось ранее. На этот раз следует настроить несколько параметров и принять решение на **турнире корректировки ошибки** (error correcting tournament,

ECT), запускаемом в программе VW параметром `--ect`, где каждый класс соревнуется на турнире с выбыванием, чтобы стать меткой для примера. Во многих примерах ECT может превзойти схему «**один против всех**», но это не общее правило, и ECT является лишь одним из подходов, который требует проверки во время работы с мультиклассовыми задачами. (Другая возможная опция `--log_multi` использует онлайновые деревья решений для расщепления выборки на меньшие по объему множества, к которым затем применяются одиночные прогнозные модели.) Мы также устанавливаем темп обучения в 1.0 и создаем полиномиальное разложение третьей степени, используя для этого параметр `--cubic` с указанием, какие пространства имен должны быть перемножены друг с другом (в этом случае пространство имен `f` для трех раз обозначается строкой `nnn`, которая идет после опции `--cubic`):

```
In:
import os

local_path = os.getcwd().decode('cp1251')
source = 'data\\shuffled_covtype.data'

origin = target_file=local_path+'\\'+source
target = target_file=local_path+'\\'+'data\\covtype.vw'

n_vars = ['var_'+'0'*int(j<10)+str(j) for j in range(54)]

vw_convert(origin, target, binary_features=list(),
           fieldnames= n_vars+['covertype'], numeric_features=n_vars,
           target = 'covertype', separator=',', classification=True,
           multiclass=True, header=False, sparse=False)

params = 'data\\covtype.vw --ect 7 -f data\\multiclass.model -k --cache_file data\\cache_
train.vw --passes=2 -l 1.0 --cubic nnn'
results = execute_vw(params)

Out:
finished run
number of examples per pass = 522911
passes used = 2
weighted example sum = 1045822.000000
weighted label sum = 0.000000
average loss = 0.235538 h
total feature number = 384838154
------------ РАБОТА ЗАВЕРШЕНА ------------
```


Для того чтобы пример был быстрым, мы ограничиваем число прохождений по данным всего двумя. Если у вас есть время, увеличьте число до 100, и вы увидите, что полученная точность может быть улучшена еще больше.

Здесь нам не придется обследовать меру ошибки дальше, поскольку сообщаемая средняя потеря является дополнением 1.0 меры точности; мы просто вычисляем ее для полноты, подтверждая, что наша точность на отложенных данных равна именно `0.769`:

```
In:
params = '-t data\\covtype.vw -i data\\multiclass.model -k --cache_file data\\cache_test.vw -p
data\\covertype.test'
```

```
results = execute_vw(params)

accuracy = 0

with open('data\\covertype.test', 'rb') as R:
    with open('data\\covtype.vw', 'rb') as TRAIN:
        holdouts = 0.0
        for n,(line, example) in enumerate(zip(R,TRAIN)):
            if (n+1) % 10==0:
                predicted = float(line.strip())
                y = float(example.split('|')[0])
                accuracy += predicted ==y
                holdouts += 1
print('Точность на отложенных данных: %0.3f' % (accuracy / holdouts))

Out:
Точность на отложенных данных: 0.769
```

Резюме

В этой главе мы подробно остановились на начальном обсуждении внеядерных алгоритмов путем добавления машин SVM к простым регрессионным линейным моделям. Большую часть времени мы уделили реализациям этих алгоритмов в библиотеке Scikit-learn – главным образом алгоритма SGD – и завершили обзором внешних инструментов, которые могут быть интегрированы сценариями Python, такими как программа Vowpal Wabbit Джона Лэнгфорда. По пути мы завершили обзор методов совершенствования и проверки моделей во время работы вне ядра, обсудив алгоритм резервуарного отбора, регуляризацию, явные и неявные нелинейные преобразования и гиперпараметрическую оптимизацию.

В следующей главе мы займемся еще более сложными и мощными подходами к обучению и представим глубокое обучение и нейронные сети в крупномасштабных задачах. Если ваши проекты вращаются вокруг анализа изображений и звуков, то все увиденное пока может не оказаться тем волшебным решением, которое вы искали. Следующая глава предоставит вам все искомые решения.

Глава 4

Искусственные нейронные сети и глубокое обучение

В этой главе мы осветим одну из самых захватывающих областей в искусственном интеллекте и машинном обучении – глубокое обучение – и пройдемся по самым важным понятиям, необходимым для его эффективного применения. При этом мы затронем следующие темы:

- важнейшие элементы теории искусственных нейронных сетей;
- реализация нейронных сетей на GPU или CPU;
- доводка параметров нейронных сетей;
- крупномасштабное глубокое обучение в среде H2O;
- глубокое обучение с автокодировщиками (предварительная тренировка).

Глубокое обучение развилось из подобласти искусственного интеллекта, где как раз и были разработаны искусственные нейронные сети. Строго говоря, любую крупную искусственную нейронную сеть можно рассматривать как *глубоко обучаемую*. Однако последние разработки в глубоких архитектурах (или топологиях) требуют большего, чем просто создание крупных нейронных сетей. Разница между глубокими архитектурами и нормальными многослойными сетями заключается в том, что глубокая архитектура состоит из множественной предобработки и шагов, выполняющихся без учителя, которые обнаруживают латентную размерность в данных, передаваемую позже на дальнейшие этапы обработки в сети. Самое важное, что нужно знать о глубоком обучении, состоит в том, что новые признаки усваиваются и трансформируются, проходя через эти глубокие архитектуры, с целью улучшения общей точности обучения. Поэтому важное различие между текущим поколением методов глубокого обучения и другими подходами машинного обучения – в том, что с глубоким обучением работа по конструированию признаков частично автоматизирована. Не слишком переживайте, если эти понятия будут звучать абстрактно; позже в этой главе они получат свое разъяснение вместе с практическими примерами. Эти методы глубокого обучения привносят новые сложности, которые делают их эффективное применение довольно серьезным испытанием.

Самое серьезное испытание заключается в том, что они с трудом поддаются тренировке, а также требуют продолжительных вычислений и доводки (тонкой

настройки) параметров. В этой главе будут рассмотрены некоторые решения этих трудностей.

За прошедшее десятилетие появились интересные применения глубокого обучения в машинном зрении, обработке естественного языка и аудиоданных, а также приложения, такие как проект Deep Face, созданный исследовательской группой компании Facebook, частично возглавляемый известным исследователем в области глубокого обучения Яном Лекуном (Yann LeCun). Проект Deep Face ставит своей целью извлечение и идентификацию человеческих лиц из цифровых изображений. Компания Google имеет свой собственный проект DeepMind во главе с Джеффри Хинтоном (Geoffrey Hinton). Недавно Google представил библиотеку с открытым исходным кодом TensorFlow, предлагающую применения глубокого обучения, которая будет подробно освещена в следующей главе.

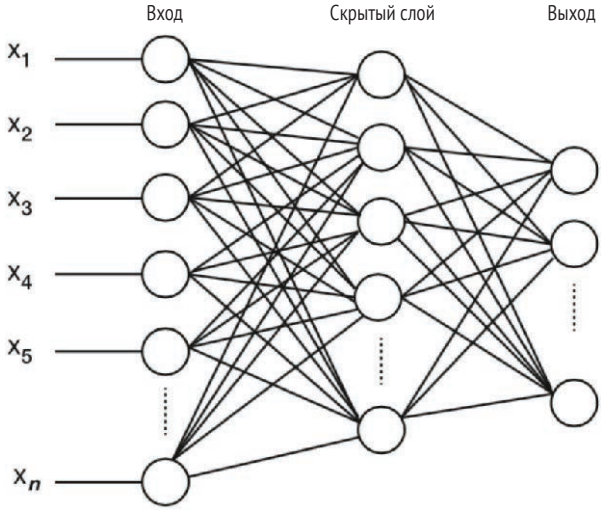
Прежде чем мы начнем запускать автономных интеллектуальных агентов, успешно проходящих тесты Тьюринга и выступающих на математических конкурсах, давайте сделаем шаг назад и пробежимся по основам.

Архитектура нейронной сети

Итак, сперва сосредоточимся на том, каким образом искусственные нейронные сети организованы, и начнем с их архитектуры и нескольких определений.

Сеть, в которой поток обучения за одно прохождение передается вперед вплоть до выходов, называется **нейронной сетью прямого распространения сигнала**.

Элементарную искусственную нейронную сеть прямого распространения можно легко изобразить на сетевом графике, как показано ниже:

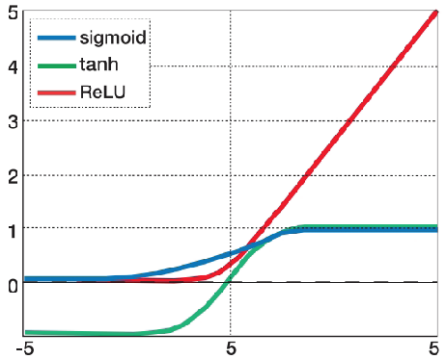


На сетевом графике видно, что эта архитектура состоит из входного, скрытого и выходного слоев. Входной слой содержит векторы признаков (где каждое наблюдение имеет n признаков), а выходной слой состоит из отдельных узлов для каждого класса выходного вектора в случае классификации и единственного числового вектора в случае регрессии.

Сила связей между узлами выражается весами, которые позже передаются функции активации. Цель функции активации состоит в том, чтобы преобразовывать свой вход в выход, который делает бинарные решения более разделимыми.

Функции активации по приемуществу дифференцируемы, и поэтому они могут использоваться для обучения.

Самыми популярными функциями активации являются **сигмоидальная** и **гиперболическая тангенсная** (tanh), а совсем недавно получила распространение функция **выпрямленный линейный узел** (rectified linear unit, ReLU)[1]. Давайте сравним самые важные функции активации, чтобы разобраться в их преимуществах и недостатках. Отметим, что мы говорим о диапазонах выходных и активных значений функции. Диапазон выходных значений – это просто фактические значения на выходе из самой функции. Диапазон активных значений, напротив, немного сложнее; это диапазон, где градиент имеет наибольшую дисперсию в заключительных обновлениях веса. Иными словами, за пределами этого диапазона градиент лежит близко к нулю и не вносит вклада в обновления параметров во время обучения. Проблему близкого к нулю градиента также называют **проблемой исчезающего градиента**, которая решается функцией активации ReLU. Данная функция активации в настоящее время является самой популярной и используется в больших нейронных сетях:



Важно отметить, что признаки должны быть прошкалированы в *диапазон активных значений* выбранной функции активации. В большинстве современных библиотек данная процедура предобработки является стандартной, и поэтому ее не придется делать самим:

Сигмоид	$\frac{1}{(1+e^{-t})}$	Диапазон активных значений: [sqrt(3), sqrt(3)] Диапазон выходных значений: (0, 1)

[1] Эта функция аналогична однополупериодному выпрямителю в электротехнике. – *Прим. перев.*

Сигмоидальные функции часто используются для удобства математических расчетов, потому что их производные очень просто вычисляются, чем мы и воспользуемся для вычисления обновлений весовых коэффициентов во время тренировки алгоритмов:

Функция tanh	$\frac{e^t - e^{-t}}{e^t + e^{-t}}$	Диапазон активных значений: [–2, 2] Диапазон выходных значений: (–1, +1)

Интересно отметить, что гиперболическая тангенсная и логистическая сигмоидальная функции связаны линейно, и tanh может рассматриваться как перешкалированная версия сигмоиды, для того чтобы ее диапазон лежал в интервале между `-1` и `1`.

Функция ReLU	$f(x) = \max(0, x)$	Диапазон активных значений: [0, inf]

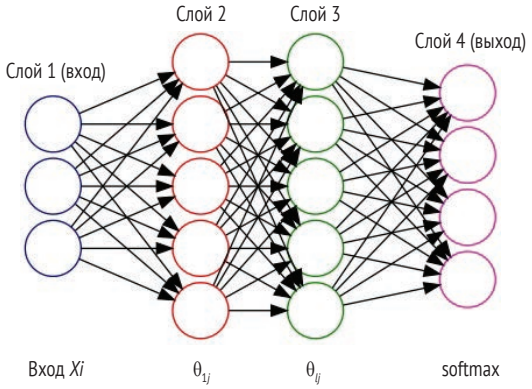
Функция ReLU является наилучшим выбором для более глубоких архитектур. Ее можно рассматривать как линейно нарастающую функцию, диапазон которой находится выше `0` и до бесконечности. Хорошо видно, что ее намного проще вычислить, чем сигмоидальную функцию. Самое большое преимущество этой функции состоит в том, что она обходит проблему исчезающего градиента. Если ReLU выглядит как вариант для проекта с глубоким обучением, то непременно им воспользуйтесь.

Функция мягкого максимума для классификации До сих пор мы видели, что функции активации, после того как они перемножены с весовыми векторами, преобразуют значения в определенном диапазоне. Мы также должны преобразовать выходы последнего скрытого слоя, прежде чем предоставить сбалансированные классы либо вероятностные выходы (значения логарифмического правдоподобия).

Данная функция будет преобразовывать выход предыдущего слоя в вероятностные значения, чтобы выполнить итоговое предсказание класса. Возведение в степень в этом случае будет возвращать почти нулевое значение каждый раз, когда выход значительно меньше, чем максимум всех значений; таким способом усиливаются различия:

$$\text{softmax}(k, x_1, \ldots, x_n) = \frac{e^{xk}}{\sum_{i=1}^{n} e^{xi}}.$$

Прямое распространение сигнала Разобравшись в функциях активации и конечных выходах из сети, теперь посмотрим, как входные признаки проходят по сети для предоставления итогового предсказания. Вычисления с участием огромных порций узлов и связей, возможно, выглядят сложной задачей, но, к счастью, процесс прямого распространения сигнала в нейронной сети сводится к последовательности векторных вычислений:



Мы приходим к итоговому прогнозу, выполнив следующие шаги:

1. Скалярное произведение на входах с весами между первым и вторым слоями и преобразование результата функцией активации.
2. Скалярное произведение на выходах из первого скрытого слоя с весами между вторым и третьим слоями. Эти результаты затем преобразуются функцией активации в каждом узле второго скрытого слоя.
3. Наконец, мы приходим к прогнозу, умножив вектор на функцию активации (функцию мягкого максимума softmax для классификации).

Каждый слой в сети можно рассматривать как вектор и применять простые векторные умножения. Более формально это будет выглядеть следующим образом:

$z_{(2)} = \theta_{(1)}x + b_{(1)}$

$a_{(2)} = f(z_{(2)})$ трансформация после слоя 1

$z_{(3)} = \theta_{(2)}a_{(2)} + b_{(2)}$

$h = a_{(3)} = f_{\text{softmax}}(z_{(3)})$ конечный выход с трансформацией функцией softmax

где θ – это вектор весов слоя x, $b_{(1)}$ и $b_{(2)}$ – узлы смещения и f – функция активации.

Отметим, что этот пример основывается на сетевой архитектуре с единственным скрытым слоем.

Давайте выполним простое прямое прохождение по нейронной сети с двумя скрытыми слоями, воспользовавшись элементарным функционалом библиотеки NumPy, и применим к конечному результату функцию мягкого максимума `softmax`:

```
In:
import numpy as np
import math

b1 = 0 # узел смещения 1
b2 = 0 # узел смещения 2

def sigmoid(x):      # сигмоидальная функция
    return 1 / (1+(math.e**-x))

def softmax(x):     # функция мягкого максимума
    l_exp = np.exp(x)
    sm = l_exp/np.sum(l_exp, axis=0)
    return sm
```

```
# входной набор данных с 3 признаками
X = np.array([ [.35,.21,.33],
               [.2,.4,.3],
               [.4,.34,.5],
               [.18,.21,16] ])
len_X = len(X) # размер тренировочного набора
input_dim = 3  # размерность входного слоя
output_dim = 1 # размерность выходного слоя
hidden_units = 4
np.random.seed(22)

# создать векторы случайных весовых коэффициентов
theta0 = 2*np.random.random((input_dim, hidden_units))
theta1 = 2*np.random.random((hidden_units, output_dim))

# проход с прямым распространением сигнала
d1 = X.dot(theta0)+b1
l1 = sigmoid(d1)
l2 = l1.dot(theta1)+b2

# применить функцию softmax к выходу из конечного слоя
output = softmax(l2)

Out:
array([[ 0.1635402 ],
       [ 0.15942338],
       [ 0.19455826],
       [ 0.48247816]])
```

Отметим, что узел смещения позволяет функции двигаться вверх-вниз и помогает плотнее подобрать целевые значения. Каждый скрытый слой содержит один узел смещения.

Обратное распространение ошибки С простым примером использования прямого распространения сигнала мы сделали первые шаги в тренировке модели. Тренировка нейронных сетей выполняется вполне аналогично методам градиентного спуска, которые мы видели с другими алгоритмами машинного обучения. А именно мы обновляем параметры модели для нахождения глобального минимума функции ошибки. Важное отличие нейронных сетей заключается в том, что теперь мы должны иметь дело с многочисленными узлами по всей сети, которые мы должны натренировать независимо. Мы делаем это, используя частную производную функции стоимости и вычисляя, насколько кривая ошибки падает, когда мы изменяем отдельно взятый признаковый вектор на определенную величину (темп обучения). Мы начинаем с самого близкого к выходу слоя и вычисляем градиент относительно производной нашей функции потерь. Если существуют скрытые слои, то мы переходим во второй скрытый слой и обновляем веса, пока не будет достигнут первый слой сети прямого распространения сигнала.

Центральная идея обратного распространения ошибки вполне аналогична другим алгоритмам машинного обучения с одним важным усложнением – мы имеем дело с многочисленными слоями и узлами. Мы видели, что каждый слой в сети представлен весовым вектором θ_{ij}. Тогда каким образом решить эту задачу? Необходимость независимо натренировать большое количество весов может выглядеть устрашающе. Однако ради нашего удобства для этих целей мы можем задействовать векторизованные операции. Аналогично тому, как мы делали во время

прямого прохождения по сети, мы точно так же вычисляем градиенты и обновляем веса, принадлежащие весовым векторам (θ_{ij}).

В алгоритме обратного распространения ошибки можно резюмировать следующие шаги:

1. Прямое прохождение: случайным образом инициализировать весовые векторы и перемножить вход с последующими весовыми векторами вплоть до конечного результата.
2. Вычисление ошибки: вычислить меру ошибки/потери на выходе из этапа прямого прохождения. Случайным образом инициализировать весовые векторы.
3. Обратное распространение вплоть до последнего скрытого слоя (относительно выхода). Вычислить градиент этой ошибки и изменить веса по направлению градиента. Это делается путем перемножения весового вектора θ_j с полученными градиентами.
4. Обновление весов, пока не будет достигнут критерий останова (минимальная ошибка или число раундов тренировки):

$$\theta_{ij} := \theta_{ij} - \eta * \Delta_{\theta} J(\theta_{ij}).$$

Мы показали прохождение по произвольной двуслойной сети с прямым распространением сигнала; теперь давайте применим к тому же самому входу, который мы использовали в приведенном ранее примере, метод обратного распространения ошибки, используя для этого алгоритм SGD в библиотеке NumPy. Обратите особое внимание на то, как мы обновляем весовые параметры:

```
In:
import numpy as np
import math

def sigmoid(x):      # сигмоидальная функция
    return 1 / (1+(math.e**-x))

def deriv_sigmoid(y): # производная сигмоидальной функции
    return y * (1.0 - y)
alpha = .1     # темп обучения
X = np.array([ [.35,.21,.33],
               [.2,.4,.3],
               [.4,.34,.5],
               [.18,.21,16] ])
y = np.array([ [0],
               [1],
               [1],
               [0] ])

np.random.seed(1)

# произвольно инициализировать слои
theta0 = 2*np.random.random((3,4)) - 1
theta1 = 2*np.random.random((4,1)) - 1

for iter in range(205000): # указываем количество тренировочных раундов
    # распространить вход вперед так же,
    # как было сделано в предыдущем упражнении
    input_layer = X
    l1 = sigmoid(np.dot(input_layer, theta0))
```

```
    l2 = sigmoid(np.dot(l1, theta1))

    # рассчитать ошибку
    l2_error = y - l2
    if (iter% 1000) == 0:
        print("Точность нейросети:" + str(np.mean(1-(np.abs(l2_error)))))
    # рассчитать градиенты в векторизованной форме;
    # функция softmax и узлы смещения пропущены в учебных целях
    l2_delta = alpha*(l2_error * deriv_sigmoid(l2))
    l1_error = l2_delta.dot(theta1.T)
    l1_delta = alpha*(l1_error * deriv_sigmoid(l1))

    theta1 += l1.T.dot(l2_delta)
    theta0 += input_layer.T.dot(l1_delta)
```

Теперь убедимся, как с каждым прохождением по сети точность увеличивается:

```
Out:
...
Точность нейросети:0.983345051044
Точность нейросети:0.983404936523
Точность нейросети:0.983464255273
Точность нейросети:0.983523015841
Точность нейросети:0.983581226603
Точность нейросети:0.983638895759
Точность нейросети:0.983696031345
Точность нейросети:0.983752641234
Точность нейросети:0.983808733139
Точность нейросети:0.98386431462
Точность нейросети:0.983919393086
Точность нейросети:0.983973975799
Точность нейросети:0.984028069878
Точность нейросети:0.984081682304
Точность нейросети:0.984134819919
```

Типичные проблемы обратного распространения ошибки Известная проблема с нейронными сетями состоит в том, что во время оптимизации с обратным распространением ошибки градиент может застревать в локальных минимумах. Это происходит, когда процесс минимизации ошибки попал в ловушку и видит минимум (точка S на рисунке) там, где это в действительности просто локальная неровность на пути прохождения пика S:

Еще одна типичная проблема появляется, когда градиентный спуск не попадает по глобальному минимуму, что иногда может приводить к удивительно плохо работающим моделям. Данная проблема еще называется **промахом**.

Обе эти проблемы можно решить путем выбора более низкого темпа обучения, когда модель промахивается, либо выбора более высокого темпа обучения при застревании в локальных минимумах. Иногда эта корректировка по-прежнему не приводит к удовлетворительному и быстрому схождению. В последнее время был найден целый ряд решений, которые смягчают эти проблемы. Были разработаны обучающиеся алгоритмы с доработками классических алгоритмов `SGD`, которые мы недавно рассматривали. Важно их понимать, чтобы можно было выбрать правильный алгоритм, подходящий для любой конкретной задачи. Давайте рассмотрим эти алгоритмы обучения более подробно.

Обратное распространение в мини-пакетном режиме Пакетный градиентный спуск вычисляет градиент, используя весь набор данных, но алгоритм SGD обратного распространения ошибки может также работать с так называемыми **мини-пакетами**, где выборка из набора данных размером k (пакеты) используется для обновления параметра обучения. Величина нерегулярности ошибки между каждым обновлением может быть сглажена мини-пакетом, который способен избегать застревания и промаха по локальным минимумам. В большинстве нейросетевых пакетов можно поменять размер пакета алгоритма (мы рассмотрим это позже). В зависимости от количества тренировочных примеров используется размер пакета данных где угодно между 10 и 300.

Тренировка с использованием инерции Инерция (momentum) – это метод, который добавляет долю предыдущего обновления веса к текущему[1]:

$$v_{t+1} = \mu v_t - \eta \nabla \mathcal{L}(\theta_t);$$
$$\theta_{t+1} = \theta_t + v_{t+1}.$$

Здесь доля предыдущего обновления веса добавляется к текущему. Высокая величина параметра инерции помогает увеличивать скорость сходимости и в результате быстрее достигать глобального минимума. Глядя на формулу, вы видите параметр v. Это эквивалент скорости обновлений градиента с темпом обучения η. Чтобы разобраться в том, как он работает, надо просто понять, что, когда градиент продолжает указывать в одинаковом направлении на протяжении многочисленных прецедентов, скорость схождения увеличивается с каждым шагом к глобальному минимуму. Он также до определенного предела удаляет нерегулярности между градиентами. Параметр инерции имеется в большинстве библиотек (как мы убедимся в более позднем примере). Когда мы задаем слишком высокое значение этого параметра, мы должны иметь в виду, что существует риск промаха по глобальному минимуму. С другой стороны, когда мы устанавливаем параметр инерции слишком низким, коэффициент может застрять в локальных минимумах, а также замедлить обучение. Идеальные значения коэффициента инерции обычно находятся в интервале между .5 и .99.

---

[1] Этот метод помогает процессу схождения принять инерционную скорость, которая сделает его устойчивым к шуму, появляющемуся в первоначальной переходной фазе тренировки. – *Прим. перев.*

Инерция Нестерова Инерция Нестерова – это более новая и улучшенная версия классической инерции. В дополнение к классической инерции он *заглядывает вперед* в направлении градиента. Другими словами, инерция Нестерова делает простой шаг, идущий из *x* в *y*, и перемещается немного далее в том же направлении таким образом, что *x* в *y* принимает вид *x* в $\{y(v_1 + 1)\}$ в направлении, заданном предыдущей точкой. Мы обойдем технические подробности этого метода, но стоит напомнить, что он систематически превосходит нормальную инерцию с точки зрения сходимости. Если существует возможность применения инерции Нестерова, то ею следует воспользоваться.

Метод ADAGRAD Метод адаптивного градиента ADAGRAD обеспечивает характерный для признака темп обучения, привлекая информацию из предыдущих обновлений:

$$g_{t+1} = g_t - \nabla\mathcal{L}(\theta_t)^2;$$

$$\theta_{t+1} = \theta_t - \frac{\eta\nabla\mathcal{L}(\theta_t)}{\sqrt{g_{t+1} + \varepsilon}}.$$

Метод ADAGRAD обновляет темп обучения для каждого параметра согласно информации из ранее выполненных итерированных градиентов для этого параметра. Это делается путем деления каждой составляющей на квадратный корень суммы квадратов его предыдущего градиента. Благодаря этому темп обучения уменьшается в течение продолжительного времени, потому что сумма квадратов будет продолжать увеличиваться с каждой итерацией. Уменьшающийся темп обучения имеет вполне существенное преимущество в том, что риск промаха по глобальному минимуму сокращается.

Метод RPROP Метод устойчивого обратного распространения RPROP (resilient backpropagation) – это адаптивный метод, в котором историческая информация не используется, а вместо этого просто учитывается знак частной производной тренировочного прецедента, и веса обновляются соответствующим образом.

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}^{(t-1)}, \text{если } \dfrac{\delta E^{(t-1)}}{\delta w_{ij}} * \dfrac{\delta E^{(t)}}{\delta w_{ij}} > 0 \\ \eta^- * \Delta_{ij}^{(t-1)}, \text{если } \dfrac{\delta E^{(t-1)}}{\delta w_{ij}} * \dfrac{\delta E^{(t)}}{\delta w_{ij}} < 0 \\ \Delta_{ij}^{(t-1)}, \text{иначе} \end{cases}$$

где $0 < \eta^- < 1 < \eta^+$.

Прямой адаптивный метод для более быстрого обучения методом обратного распространения: Алгоритм RPROP. Мартин Ридмиллер (Martin Riedmiller) 1993

Детально обследуя приведенные выше формулы, мы видим, что, как только частная производная ошибки изменяет свой знак (> 0 или < 0), градиент начинает двигаться в противоположном направлении, которое ведет к глобальному минимуму, тем самым исправляя промах. Однако если знак не меняется вообще, то де-

лаются более крупные шаги к глобальному минимуму. Во множестве статей было доказано превосходство метода RPROP над методом ADAGRAD, но на практике это не подтверждается систематически. Следует иметь в виду еще одну важную деталь – метод RPROP не работает надлежащим образом с мини-пакетами.

Метод RMSProp Метод адаптивного скользящего среднего градиентов RMSProp – это метод адаптивного обучения без сжатия темпа обучения:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g_2]t + e}} g_t .$$

В этом методе также задействуется идея инерции и метода ADAGRAD, но с одним важным дополнением – он позволяет избегать уменьшения темпа обучения на протяжении всего времени. С этим методом сжатие контролируется функцией экспоненциального затухания на среднем числе значений градиентов.

Ниже приведен список алгоритмов оптимизации по методу градиентного спуска:

	Приложения	Типичные проблемы	Практические подсказки
Регулярный SGD	Широко применим	Промах, попадание в локальные минимумы	Использовать с инерцией и мини-пакетом
ADAGRAD	Наборы данных меньшего объема < 10k	Медленная сходимость	Использовать темп обучения между .01 и.1. Широко применим. Работает с разреженными данными
RPROP	Более крупные наборы данных > 10k	Не эффективен с мини-пакетами	Использовать RMSProp, если это возможно
RMSProp	Более крупные наборы данных > 10k	Не эффективен с широкими и мелкими сетями	Особенно полезен для широких разреженных данных

Чему и как нейронные сети обучаются

Теперь, когда у нас есть основное понимание метода обратного распространения ошибки во всех его формах, пора обратиться к наиболее трудной задаче в нейросетевых проектах: как выбрать правильную архитектуру? Ключевая способность нейронных сетей состоит в том, что веса внутри архитектуры могут преобразовывать вход в нелинейное пространство признаков и таким образом решать нелинейную классификацию (границы решения) и задачи регрессии. Давайте выполним простое и одновременно показательное упражнение с целью продемонстрировать эту идею с использованием библиотеки `neurolab`. Мы воспользуемся библиотекой `neurolab` только для короткого упражнения; для масштабируемых задач обучения мы предложим другие методы.

Прежде всего установите библиотеку `neurolab` при помощи менеджера библиотек `pip`, используя для этого терминал:

```
$ pip install neurolab
```

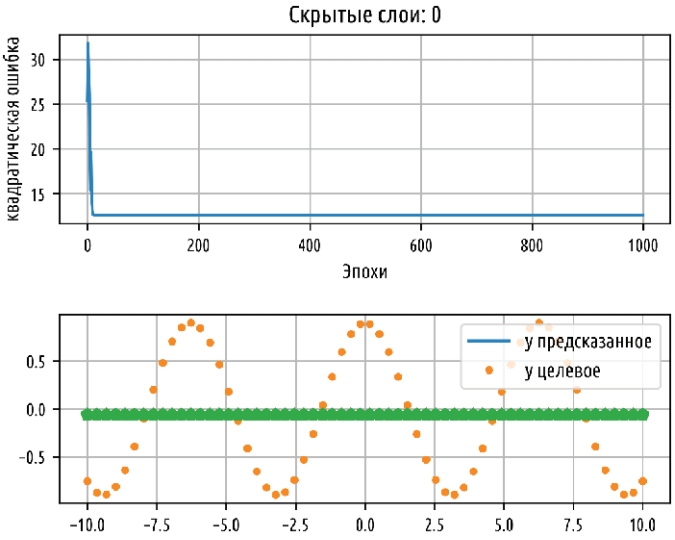
В этом примере мы сгенерируем простую нелинейную косинусную функцию при помощи NumPy и натренируем нейронную сеть, чтобы предсказывать коси-

нусную функцию из переменной. Мы настроим несколько нейросетевых архитектур, чтобы увидеть, насколько хорошо каждая архитектура способна предсказывать косинусную целевую переменную:

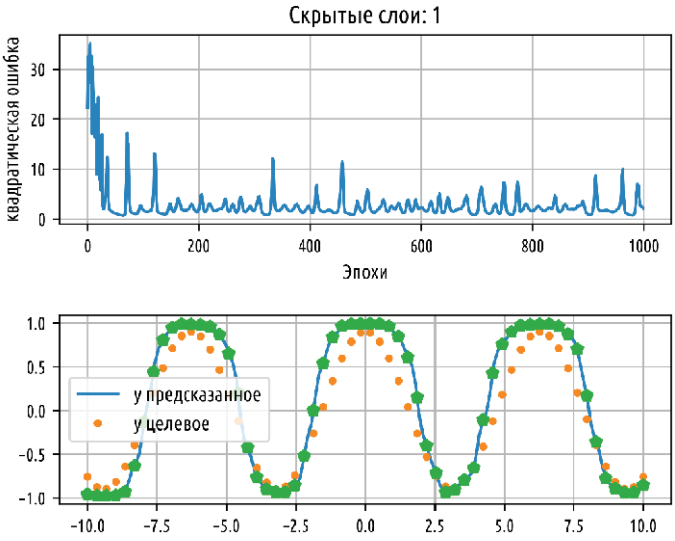
```
In:
import neurolab as nl
import numpy as np
from sklearn import preprocessing

# Создать тренировочные выборки
x = np.linspace(-10, 10, 60)
y = np.cos(x) * 0.9
size = len(x)
x_train = x.reshape(size, 1)
y_train = y.reshape(size, 1)

# Создать сеть с 4 слоями и инициализировать случайным образом
# Поэкспериментировать с количеством слоев
d = [[1,1], [45,1], [45,45,1], [45,45,45,1]]

for i in range(4):
    net = nl.net.newff([[-10, 10]], d[i])
    train_net = nl.train.train_gd(net, x_train, y_train,
                                  epochs=1000, show=100)
    outp = net.sim(x_train)

    # Построить график результатов
    # (двойной график с кривой ошибки и предсказанными значениями)
    plt.subplot(2, 1, 1)
    plt.grid(True)
    plt.plot(train_net)
    plt.title(u'Скрытые слои: ' + str(i))
    plt.xlabel(u'Эпохи')
    plt.ylabel(u'квадратическая ошибка')
    x2 = np.linspace(-10.0,10.0,150)
    y2 = net.sim(x2.reshape(x2.size, 1)).reshape(x2.size)
    y3 = outp.reshape(size)
    plt.subplot(2, 1, 2)
    plt.grid(True)
    plt.plot(x2, y2, '-',x , y, '.', x, y3, 'p')
    plt.legend([u'y предсказанное', u'y целевое'])
    saveplot('ex_4_' + str(i) + '.png')
    plt.show()
```

Теперь приглядимся, как ведет себя кривая ошибки и как ожидаемые значения начинают аппроксимировать целевые значения по мере добавления к нейросети новых слоев.

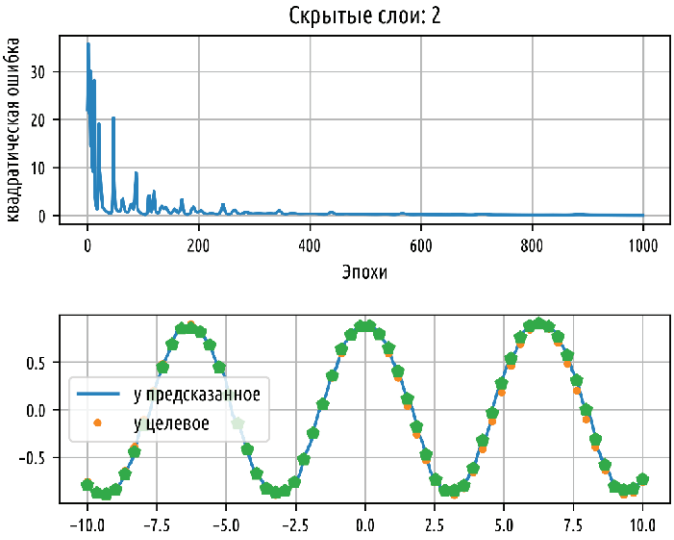
С нулевыми скрытыми слоями нейросеть проецирует прямую линию через целевые значения. Кривая ошибки быстро падает до минимума с плохой подгонкой:



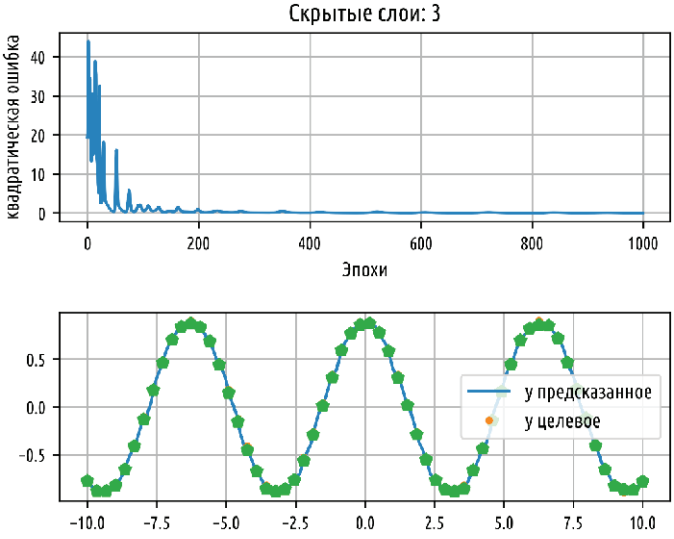
С одним скрытым слоем сеть начинает аппроксимировать целевой выход. Обратите внимание, насколько нерегулярна кривая ошибки:



С двумя скрытыми слоями нейронная сеть аппроксимирует целевое значение еще ближе. Кривая ошибки падает быстрее и ведет себя менее нерегулярно:



Почти совершенная подгонка с тремя скрытыми слоями. Кривая ошибки падает намного быстрее (приблизительно 220 итераций).



Синяя линия в верхнем графике представляет собой визуализацию того, как ошибка падает с каждой эпохой (полное прохождение через тренировочный набор). Она показывает, что для достижения глобального минимума нужно определенное число прохождений через тренировочный набор. Если обследовать эту кривую ошибки внимательнее, то вы увидите, что кривая ошибки с каждой архитектурой ведет себя по-разному. Нижний график (пунктирная линия) показывает, как ожидаемые значения начинают аппроксимировать целевые значения. Без скрытого слоя нейронная сеть неспособна обнаруживать нелинейные функции,

но как только мы добавляем скрытые слои, сеть начинает усваивать нелинейные функции и все более и более сложные функции. По сути дела, нейронные сети могут усваивать *любую* возможную функцию. Такая способность обучаться любой возможной функции называется **универсальной теоремой аппроксимации** (теорема Цыбенко). Мы можем модифицировать эту аппроксимацию, добавляя в нейронную сеть скрытые нейроны (узлы и слои). При этом, однако, нужно быть осторожными, чтобы не вызвать переподгонку; добавление большого количества слоев и узлов приведет к запоминанию тренировочных данных, вместо того чтобы подобрать оптимальную обобщаемую функцию. Довольно часто слишком много слоев в сети может оказаться вредным для точности прогнозирования.

Выбор правильной архитектуры

Как мы убедились, комбинаторное пространство возможных нейросетевых архитектур почти бесконечно. Тогда как узнать заранее, какая архитектура подойдет для нашего проекта? Хотелось бы иметь своего рода эвристическое или эмпирическое правило, для того чтобы сконструировать архитектуру для конкретной задачи. В последнем разделе мы использовали простой пример всего с одним выходом и одной функцией. Однако недавняя волна нейросетевых архитектур, которую мы называем *глубоким обучением*, весьма многосложна, и крайне важно уметь создавать правильную архитектуру нейронной сети для любой задачи. Как мы упомянули ранее, типичная нейронная сеть состоит из входного слоя, одного или более скрытых слоев и выходного слоя. Давайте подробно рассмотрим каждый слой архитектуры, с тем чтобы можно было ориентироваться при формировании нужной архитектуры для той или иной задачи.

Входной слой

Когда мы упоминаем входной слой, мы в основном говорим о признаках, которые будут использоваться как вход в нейронную сеть. Требуемые шаги предобработки очень зависят от формы и содержания данных. Если имеются признаки, которые измеряются в разных шкалах, то нам нужно перешкалировать и нормализовать данные. В случаях, когда имеется большое количество признаков, рекомендуется использовать один из методов снижения размерности, в частности методы PCA или SVD.

Следующие ниже методы предобработки могут быть применены ко входам перед процессом обучения:

- нормализация, шкалирование и обнаружение выбросов;
- снижение размерности (метод SVD и факторный анализ);
- предварительная тренировка (автокодировщики и машины Больцмана).

Каждый из этих методов будет рассмотрен в нижеследующих примерах.

Скрытый слой

Как выбрать количество узлов в скрытых слоях? Сколько скрытых слоев добавить к сети? В предыдущем примере мы увидели, что нейронная сеть без скрытого слоя неспособна обучиться нелинейной функции (это касается и подбора кривых для задачи регрессии, и границ решения для задачи классификации). Таким образом, если будет иметься нелинейная закономерность или граница решения для проецирования, то нам понадобятся скрытые слои. Когда дело касается выбора количества узлов в скрытом слое, обычно стремятся к тому, чтобы в скрытом слое

количество узлов было меньше, чем количество узлов во входном слое, и больше, чем количество выходных узлов. Желательно иметь:

- меньше скрытых узлов, чем количество входных признаков;
- больше узлов, чем количество выходных узлов (классы для классификации).

Иногда, когда целевая функция очень сложна по форме, существует исключение. В случае когда мы добавляем больше узлов, чем входных размерностей, мы добавляем **расширение** пространства признаков. Сети с такими слоями обычно называются **широкими**.

Многосложные сети могут обучаться большему количеству сложных функций, но это не означает, что можно просто продолжать накладывать слои один на другой. Рекомендуется держать количество слоев под контролем, потому что слишком много слоев вызовет проблемы с переподгонкой, более высокой загрузкой CPU и даже недоподгонкой. Обычно достаточно иметь от одного до четырех скрытых слоев.

В качестве отправной точки рекомендуется использовать от одного до четырех слоев.

Выходной слой

Каждая нейронная сеть имеет один выходной слой, который, аналогично входному слою, очень зависит от структуры используемых данных. Для задачи классификации мы обычно будем использовать функцию мягкого максимума `softmax`. В этом случае следует использовать такое же количество узлов, что и количество предсказываемых классов.

Нейронные сети в действии

Давайте получим небольшой практический опыт работы с тренировкой нейронных сетей для классификации. Мы воспользуемся библиотекой sknn, которая представляет собой обертку Scikit-learn для библиотек lasagne и Pylearn2. Дополнительную информацию об этой библиотеке можно узнать на https://github.com/aigamedev/scikit-neuralnetwork/.

Мы воспользуемся этим инструментом из-за его практичного и Python'овского интерфейса, а также потому, что он является великолепным введением в более сложные платформы, такие как библиотека Keras.

Библиотека sknn может по вашему выбору выполняться как на CPU, так и на GPU. Отметим, что если вы решите использовать GPU, то sknn будет работать на Theano.

Для CPU (наиболее стабильная версия):

```
# Использовать CPU в 32-разрядном режиме, from sknn.platform import gpu32
from sknn.platform import cpu32, threading

# Использовать CPU в 64-разрядном режиме, from sknn.platform import cpu64
from sknn.platform import cpu64, threading
```

Для GPU:

```
# Использовать GPU в 32-разрядном режиме
from sknn.platform import gpu32

# Использовать GPU в 64-разрядном режиме
from sknn.platform import gpu64
```

Параллелизация для библиотеки sknn

Параллельная обработка может быть задействована следующим образом, но сразу нужно предупредить – это не самый стабильный метод:

```
from sknn.platform import cpu64, threading
```

Можно указать sknn использовать определенное количество процессов:

```
from sknn.platform import cpu64, threads2 # любое нужное число потоков
```

После того как задано соответствующее число процессов, программный код можно параллелизировать, добавив в перекрестную проверку `n_jobs=nthreads`, т. е. число заданий равно числу процессов.

После того как были рассмотрены самые важные понятия и подготовлена среда, теперь приступим к реализации нейронной сети. Для этого примера мы применим удобный, но довольно скучный набор данных Iris.

Загрузив данные, мы выполним их предобработку в форме нормализации и шкалирования и начнем строить модель:

```
In:
import numpy as np
from sknn.mlp import Classifier, Layer
from sklearn import model_selection, datasets
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn import preprocessing
from sklearn.metrics import accuracy_score

# импортировать уже знакомый набор данных Iris
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = \
    train_test_split(iris.data, iris.target,
                     test_size=0.2, random_state=0)
```

Ниже применим ко входам предобработку, нормализацию и шкалирование:

```
X_trainn = preprocessing.normalize(X_train, norm='l2')
X_testn = preprocessing.normalize(X_test, norm='l2')

X_trainn = preprocessing.scale(X_trainn)
X_testn = preprocessing.scale(X_testn)
```

Теперь сформируем нейросетевую архитектуру и параметры. Начнем с двухслойной нейронной сети. В компоненте `Layer` мы независимо определяем настройки каждого слоя. (Мы снова встретим этот метод, когда будем работать с Tensorflow и Keras.) Набор данных цветков ириса Iris состоит из четырех признаков, но поскольку в данном конкретном случае *широкая* нейронная сеть работает вполне хорошо, мы будем использовать 13 узлов в каждом скрытом слое. Отметим, что sknn применяет алгоритм SGD по умолчанию:

```
clf = Classifier(
    layers=[Layer("Rectifier", units=13),
            Layer("Rectifier", units=13),
            Layer("Softmax")],
    learning_rate=0.001,
    learning_rule='sgd',
```

```
    random_state=201,
    n_iter=200)

model1 = clf.fit(X_trainn, y_train)
y_hat  = clf.predict(X_testn)
scores = model_selection.cross_val_score(clf,
                                         X_trainn, y_train,
                                         cv=5)

print('Средняя точность на тренировочных данных %s' % np.mean(scores))
print('Точность на тестовых данных с классическим sgd %s'
       % accuracy_score(y_hat,y_test))

Out:
Средняя точность на тренировочных данных 0.949909090909
Точность на тестовых данных с классическим sgd 0.933333333333
```

На тренировочном наборе получен порядочный результат, но можно было бы добиться большего успеха.

Как уже обсуждалось, инерция Нестерова может сократить длину пути к глобальному минимуму; давайте выполним этот алгоритм, задав `rule='nesterov'`, чтобы посмотреть, удастся ли увеличить точность и улучшить сходимость:

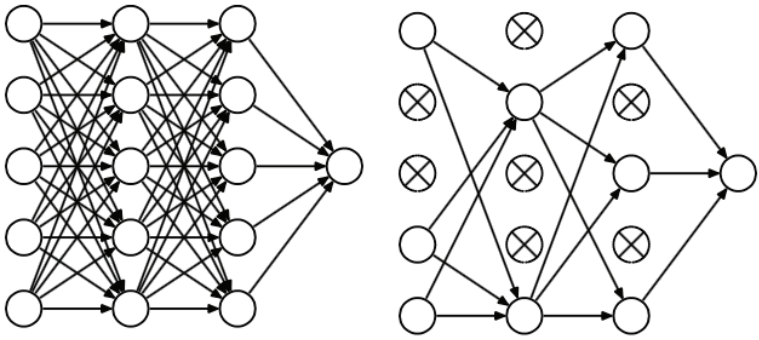
```
In:
clf = Classifier(
    layers=[Layer("Rectifier", units=13),
            Layer("Rectifier", units=13),
            Layer("Softmax")],
    learning_rate=0.001,
    learning_rule='nesterov',
    random_state=101,
    n_iter=1000)

model1 = clf.fit(X_trainn, y_train)
y_hat  = clf.predict(X_testn)
scores = model_selection.cross_val_score(clf,
                                         X_trainn, y_train,
                                         cv=5)

print('Средняя точность на тренировочных данных с инерцией Нестерова %s'
      % np.mean(scores))
print('Точность на тестовых данных с инерцией Нестерова %s'
      % accuracy_score(y_hat,y_test))

Out:
Средняя точность на тренировочных данных с инерцией Нестерова, 0.966575757576
Точность на тестовых данных с инерцией Нестерова 0.966666666667
```

В данном случае модель с инерцией Нестерова стала лучше.

Нейронные сети и регуляризация

Несмотря на то что предыдущий пример не привел к переподгонке модели, следует предусмотреть стратегии регуляризации нейронных сетей. Приведем три наиболее широко используемых способа, при помощи которых к нейронной сети можно применить регуляризацию:

- регуляризация **L1** и **L2** с затуханием весов в качестве параметра для силы регуляризации;
- **прореживание** (dropout) – означает, что деактивирование узлов нейронной сети методом случайного отбора может заставить другие узлы взять работу на себя;



Слева находится обычная нейронная сеть.
Справа – архитектура с применением прореживания,
где узлы сети случайным образом деактивированы (помечены как X)

- **усреднение**, или ансамблирование, многочисленных нейронных сетей (каждый с разными настройками).

Теперь попробуем применить к этой модели прореживание и посмотрим, будет ли оно работать:

```
In:
clf = Classifier(
    layers = [Layer("Rectifier", units=13),
              Layer("Rectifier", units=13),
              Layer("Softmax")],
    learning_rate=0.01,
    n_iter=2000,
    learning_rule='nesterov',
    regularize='dropout',   # задаем процедуру прореживания
    dropout_rate=.1,        # доля прореживания нейронных узлов во всей сети
    random_state=0)

model1 = clf.fit(X_trainn, y_train)
scores = model_selection.cross_val_score(clf, X_trainn, y_train, cv=5)
y_hat  = clf.predict(X_testn)

print('Средняя точность на тренировочных данных с прореживанием %s'
      % np.mean(scores))
print('Точность на тестовых данных с прореживанием %s'
      % accuracy_score(y_hat,y_test))

Out:
Средняя точность на тренировочных данных с прореживанием 0.958242424242
Точность на тестовых данных с прореживанием 0.833333333333
```

В данном случае прореживание не привело к удовлетворительным результатам, и поэтому его следует полностью исключить. Не стесняйтесь эксперименти-

ровать и с другими методами. Просто измените параметр `learning_rule` и посмотрите, что он делает с общей точностью. Можно попробовать модели `sgd`, `momentum`, `nesterov`, `adagrad` и `rmsprop`. Из этого примера вы узнали, что инерция Нестерова может увеличить общую точность, а прореживание `dropout` не стало подходящим методом регуляризации и оказалось вредным для результативности модели. Учитывая, что все эти многочисленные параметры взаимодействуют и приводят к непредсказуемым результатам, нам действительно нужен метод тонкой настройки. Это именно то, чем мы займемся в следующем разделе.

Нейронные сети и гиперпараметрическая оптимизация

Поскольку параметрическое пространство нейронных сетей и моделей глубокого обучения очень широкое, оптимизация является задачей трудной и в вычислительном отношении очень дорогостоящей. Неправильная архитектура нейронной сети может стать самым лучшим рецептом для неудачи. Эти модели могут быть точными, только если применены правильные параметры и выбрана правильная архитектура для решаемой задачи. К сожалению, существует всего несколько приложений, которые предоставляют методы тонкой настройки. Мы обнаружили, что наилучший метод тонкой настройки параметров на данный момент – это алгоритм **рандомизированного поиска**, который выполняет итеративный обход параметрического пространства в случайном порядке, экономя вычислительные ресурсы. Библиотека sknn в действительности единственная, которая имеет эту возможность. Давайте пройдемся по методам тонкой настройки (или доводки) параметров вместе со следующим ниже примером с набором данных Wine о качестве вин.

В этом примере мы сначала загружаем набор данных Wine, их преобразуем и затем строим модель, основываясь на выбранных параметрах. Отметим, что этот набор данных имеет 13 признаков, и мы задаем узлы в каждом слое в количестве между 4 и 20. В данном случае мини-пакет не используется; набор данных просто слишком мал:

```
In:
import numpy as np
import scipy as sp
import pandas as pd
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from scipy import stats
from sklearn.model_selection import train_test_split
from sknn.mlp import  Layer, Regressor, Classifier as skClassifier

# загрузить данные
df = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/
winequality-red.csv', sep = ';')
X  = df.drop('quality', 1).values   # отбросить целевую переменную
y1 = df['quality'].values           # исходная целевая переменная
y  = y1 <= 5             # новая целевая переменная: рейтинг <= 5?

# разбить данные на тренировочный и тестовый наборы
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.2, random_state=42)

print(X_train.shape)
```

```
max_net = skClassifier(
    layers = [ Layer("Rectifier",units=1),
               Layer("Rectifier",units=1),
               Layer("Rectifier",units=1),
               Layer("Softmax") ])
params = {'learning_rate' : [.002],
          'hidden0__units': sp.stats.randint(8, 20),
          'hidden0__type' : ["Rectifier"],
          'hidden1__units': sp.stats.randint(8, 20),
          'hidden1__type' : ["Rectifier"],
          'learning_rule' :["adam","rmsprop","sgd"]}
max_net2 = RandomizedSearchCV(max_net, param_distributions=params,
                              n_iter=10,cv=3,random_state=101,
                              scoring='accuracy',verbose=10,
                              pre_dispatch=None)
model_tuning=max_net2.fit(X_train,y_train)
```

```
print("Лучшая отметка %s" % model_tuning.best_score_)
print("Лучшие параметры %s" % model_tuning.best_params_)
```

```
Out:
[CV] hidden0__units=11, learning_rate=0.100932183167, hidden2__
units=4, hidden2__type=Rectifier, batch_size=30, hidden1__
units=11, learning_rule=adagrad, hidden1__type=Rectifier, hidden0__
type=Rectifier, score=0.655914 - 3.0s
[Parallel(n_jobs=1)]: Done 74 tasks | elapsed: 3.0min
[CV] hidden0__units=11, learning_rate=0.100932183167, hidden2__
units=4, hidden2__type=Rectifier, batch_size=30, hidden1__
units=11, learning_rule=adagrad, hidden1__type=Rectifier, hidden0__
type=Rectifier
[CV] hidden0__units=11, learning_rate=0.100932183167, hidden2__
units=4, hidden2__type=Rectifier, batch_size=30, hidden1__
units=11, learning_rule=adagrad, hidden1__type=Rectifier, hidden0__
type=Rectifier, score=0.750000 - 3.3s
[Parallel(n_jobs=1)]: Done 75 tasks | elapsed: 3.0min
[Parallel(n_jobs=1)]: Done 75 out of 75 | elapsed: 3.0min finished
Лучшая отметка 0.721366278222
```

```
Лучшие параметры {'hidden0__units': 14, 'learning_rate':
0.03202394348494512, 'hidden2__units': 19, 'hidden2__type':
'Rectifier', 'batch_size': 30, 'hidden1__units': 17, 'learning_rule':
'adagrad', 'hidden1__type': 'Rectifier', 'hidden0__type': 'Rectifier'}
```

Предупреждение: ввиду того что поиск в параметрическом пространстве осуществляется в произвольном порядке, результаты могут быть непоследовательными.

Мы видим, что лучшими параметрами для нашей модели являются прежде всего первый слой с 14 узлами, второй слой с 17 узлами и третий слой с 19 узлами. Это достаточно сложная архитектура, которую мы никогда, возможно, не смогли бы вывести сами, что демонстрирует важность гиперпараметрической оптимизации.

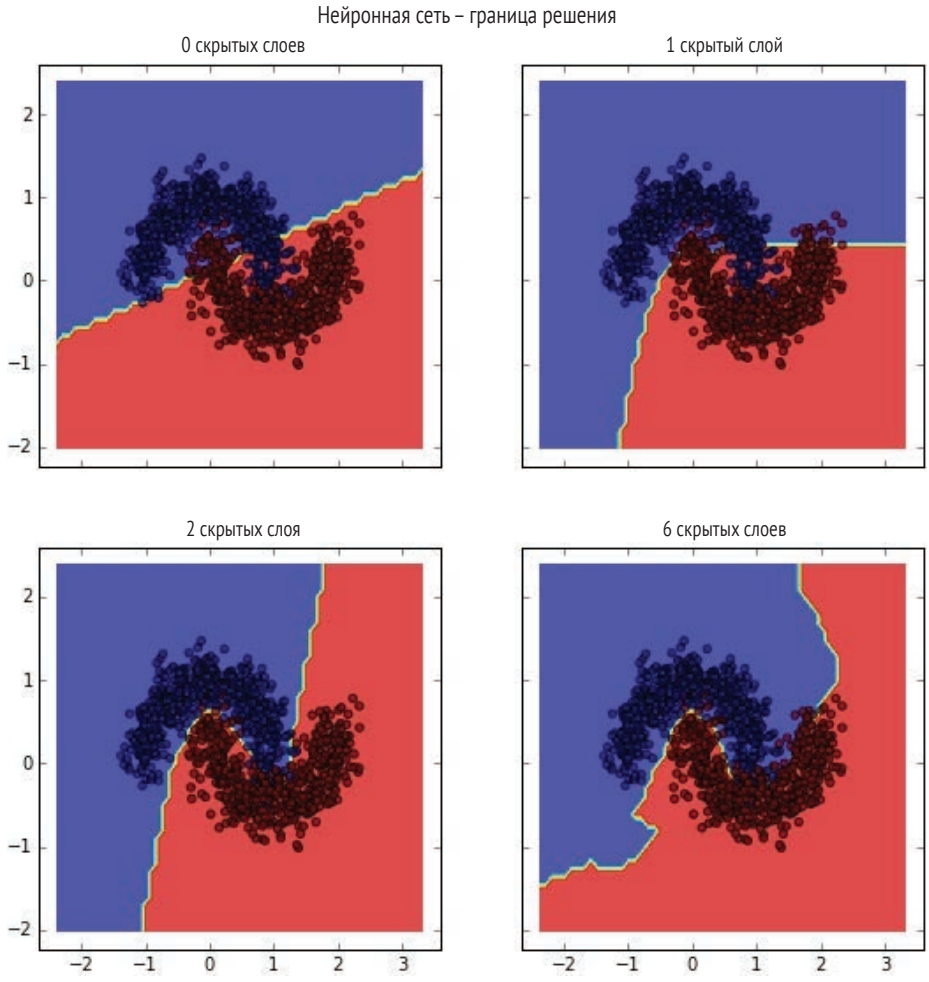
Нейронные сети и границы решения

В предыдущем разделе мы обсудили, что, добавляя скрытые узлы в нейронную сеть, можно ближе аппроксимировать функцию. Однако мы не применили ее

к задаче классификации. Чтобы сделать это, мы сгенерируем данные с нелинейным целевым значением и посмотрим, как при добавлении в архитектуру скрытых узлов изменяется поверхность решения. Пора взглянуть на универсальную теорему аппроксимации в деле! Сначала сгенерируем немного нелинейно разделимых данных с двумя признаками, сформируем нейросетевую архитектуру и посмотрим, как границы решения изменяются вместе с каждой архитектурой:

```
In:
import numpy as np
from itertools import product
from sklearn import preprocessing
from sklearn import datasets
from sklearn.datasets import make_blobs
from sknn.mlp import Classifier, Layer

X, y = datasets.make_moons(n_samples=500, noise=.2, random_state=222)

net1 = Classifier(
    layers=[Layer("Softmax")],
    random_state=222,
    learning_rate=0.01,
    n_iter=100)

net2 = Classifier(
    layers=[Layer("Rectifier", units=4),
            Layer("Softmax")],
    random_state=12,
    learning_rate=0.01,
    n_iter=100)

net3 = Classifier(
    layers=[Layer("Rectifier", units=4),
            Layer("Rectifier", units=4),
            Layer("Softmax")],
    random_state=22,
    learning_rate=0.01,
    n_iter=100)

net4 = Classifier(
   layers=[Layer("Rectifier", units=4),
           Layer("Rectifier", units=4),
           Layer("Rectifier", units=4),
           Layer("Rectifier", units=4),
           Layer("Rectifier", units=4),
           Layer("Rectifier", units=4),
           Layer("Softmax")],
    random_state=62,
    learning_rate=0.01,
    n_iter=100)

net1.fit(X, y)
net2.fit(X, y)
net3.fit(X, y)
net4.fit(X, y)

# построение графика областей решения
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

```
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

f, arxxx = plt.subplots(2, 2,
                        sharey='row', sharex='col', figsize=(8, 8))
f.suptitle(u'Нейронная сеть - граница решения', fontsize=14)

for idx, clf, ti in zip(product([0, 1], [0, 1]),
                        [net1, net2, net3,net4],
                        [u'0 скрытых слоев', u'1 скрытый слой',
                         u'2 скрытых слоя', u'6 скрытых слоев']):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    arxxx[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.5)
    arxxx[idx[0], idx[1]].scatter(X[:, 0], X[:, 1], c=y, alpha=0.5)
    arxxx[idx[0], idx[1]].set_title(ti)

plt.show()
```



На этом рисунке видно, что по мере добавления скрытых слоев в нейронную сеть можно извлекать все более и более сложные границы решения. Интересно отметить, что двуслойная сеть произвела самые точные прогнозы.


Отметим, что результаты от прогона к прогону могут различаться.

Глубокое обучение в крупном масштабе с H2O

В предыдущих разделах мы рассмотрели нейронные сети и глубокую архитектуру, работающую на локальном компьютере, и узнали, что нейронные сети уже высоковекторизованы, но по-прежнему остаются в вычислительном плане дорогостоящими. Если возникнет потребность сделать алгоритм более масштабируемым на настольном компьютере, то не так уж и многого можно добиться, кроме как использовать Theano и вычисления на GPU. Поэтому если мы действительно хотим масштабировать алгоритмы глубокого обучения, то должны найти инструмент, который может выполнять алгоритмы вне ядра, а не на локальном CPU/GPU. Платформа H2O – это на сегодняшний день единственный внеядерный инструмент с открытым исходным кодом, который может быстро выполнять алгоритмы глубокого обучения. Этот инструмент является также кросс-платформенным; помимо Python, существуют API для R, Scala и Java.

Платформа H2O скомпилирована на основе Java и разработана для широкого спектра связанных с наукой о данных задач, таких как обработка данных и машинное обучение. Данная платформа выполняется на распределенных и параллельных CPU в оперативной памяти, а ее данные хранятся в кластере H2O. На данный момент она имеет приложения для **общих линейных моделей** (General Linear Model, GLM), случайных лесов, **машин градиентного бустинга** (Gradient Boosting Machines, GBM), *k* средних, наивного Байеса, анализа главных компонент, регрессии главных компонент и, конечно, для основной темы этой главы, глубокого обучения.

Отлично, теперь мы готовы выполнить первый внеядерный анализ на платформе H2O.

Давайте запустим экземпляр H2O и загрузим файл в ее систему с распределенной памятью:

```
In:
import sys
sys.prefix = "/usr/local"
import h2o

h2o.init(start_h2o=True)
```

Наберите нижеследующую команду, чтобы получить интересную информацию о спецификации вашего кластера. Обратите внимание на допустимую память и число ядер.

```
h2o.cluster().show_status()
h2o.cluster().shutdown(prompt=False)
```

Эта информация будет более или менее выглядеть следующим образом (могут быть незначительные различия между пробными версиями и системами):

```
Out:
Checking whether there is an H2O instance running at http://localhost:54321..... not found.
Attempting to start a local H2O server...
; Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)
  Starting server from c:\python27\lib\site-packages\h2o\backend\bin\h2o.jar
  Ice root: c:\users\labor\appdata\local\temp\tmpqpvewu
  JVM stdout: c:\users\labor\appdata\local\temp\tmpqpvewu\h2o_labor_started_from_python.out
  JVM stderr: c:\users\labor\appdata\local\temp\tmpqpvewu\h2o_labor_started_from_python.err
  Server is running at http://127.0.0.1:54321
Checking whether there is an H2O instance running at http://localhost:54321. connected.
H2O cluster uptime:        5 mins 38 secs
H2O cluster version:       3.10.4.8
H2O cluster version age:   23 days
H2O cluster name:          H2O_from_python_labor_3ljewe
H2O cluster total nodes:   1
H2O cluster free memory:   1.258 Gb
H2O cluster total cores:   4
H2O cluster allowed cores: 4
H2O cluster status:        locked, healthy
H2O connection url:        http://localhost:54321
H2O connection proxy:      None
H2O internal security:     False
Python version:            2.7.13 final
H2O cluster uptime:        5 mins 38 secs
H2O cluster version:       3.10.4.8
H2O cluster version age:   23 days
H2O cluster name:           H2O_from_python_labor_3ljewe
H2O cluster total nodes:   1
H2O cluster free memory:   1.258 Gb
H2O cluster total cores:   4
H2O cluster allowed cores: 4
H2O cluster status:        locked, healthy
H2O connection url:        http://localhost:54321
H2O connection proxy:      None
H2O internal security:     False
Python version:            2.7.13 final
H2O session _sid_8f32 closed.
```

Крупномасштабное глубокое обучение с H2O

При выполнении задачи глубокого обучения на основе H2O для тренировки модели мы воспользуемся известным набором данных MNIST. Он состоит из изображений рукописных цифр в формате 28×28 разной интенсивности пикселов. Тренировочный набор имеет 70 000 тренировочных элементов с 784 признаками вместе с меткой для каждой записи, содержащей целевую метку *digits*.

Теперь, когда мы чувствуем себя увереннее в управлении данными в H2O, давайте выполним пример с глубоким обучением.

В платформе H2O преобразование либо нормализация входных данных не требуется; эта стандартная процедура встроена и выполняется изнутри автоматически. Каждый признак преобразуется в пространство $N(0, 1)$.

Давайте импортируем набор данных MNIST с изображением рукописных цифр из сервера Amazon в кластер H2O:

```
In:
import h2o

h2o.init(start_h2o=True)

train_url = "https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/mnist/train.csv.gz"
test_url  = "https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/mnist/test.csv.gz"

train = h2o.import_file(train_url)
test  = h2o.import_file(test_url)

train.describe()
test.describe()

In:
y = 'C785'
x = train.names[0:784]
train[y] = train[y].asfactor()
test[y]  = test[y].asfactor()

from h2o.estimators.deeplearning import H2ODeepLearningEstimator

model_cv = H2ODeepLearningEstimator(distribution='multinomial',
                                    activation='RectifierWithDropout',
                                    hidden=[32,32,32],
                                    input_dropout_ratio=.2,
                                    sparse=True,
                                    l1=.0005,
                                    epochs=5,
                                    nfolds=3)
```

Итоговый результат предоставит много подробной информации[1]. Ниже показана первая таблица, которую вы увидите. Она предоставляет все специфические подробности об архитектуре нейронной сети. Хорошо видно, что мы использовали нейронную сеть, вход в которую имеет размерность 717, с тремя скрытыми слоями (состоящими из 32 узлов каждый), с функцией активации softmax, применяемой к выходному слою, и функцией ReLU между скрытыми слоями:

```
model_cv.train(x=x, y=y, training_frame=train)
model_cv.show()
Scoring the model.
Status of Neuron Layers (predicting C785, 10-class classification, multinomial distribution,
CrossEntropy loss, 25 418 weights/biases, 404,0 KB, 8 807 training samples, mini-batch size 1):
```

Layer	Units	Type	Dropout	L1	L2	Mean Rate	Rate RMS	Momen-tum	Mean Weight	Weight RMS	Mean Bias	Bias RMS
1	717	Input	20,00%									
2	32	RectifierDropout	50,00%	0,0005	0,0	0,145513	0,364942	0,0	-0,001101	0,052098	0,491006	0,026888
3	32	RectifierDropout	50,00%	0,0005	0,0	0,001108	0,000626	0,0	-0,001854	0,182977	1,000663	0,115277
4	32	RectifierDropout	50,00%	0,0005	0,0	0,000758	0,000472	0,0	-0,057711	0,179252	0,835465	0,059208
5	10	Softmax		0,0005	0,0	0,0019	0,000741	0,0	0,010953	0,85586	-0,129878	0,10551

[1] Вся информация также хранится в журнале регистрации событий, который можно скачать командой `h2o.download_all_logs(dirname='./data/', filename = 'autoh2o_log.zip')`. В этом архиве нас интересует файл `h2o_127.0.0.1_00000-0-info.log`. – *Прим. перев.*

```
Model Metrics Type: Multinomial
 Description: Metrics reported on temporary training frame with 10130 samples
 model id: DeepLearning_model_python_1497852092123_1_cv_1
 frame id: DeepLearning_model_python_1497852092123_1_cv_1_train.temporary.sample.16,67%
 MSE: 0.80062
 RMSE: 0.8947737
 logloss: 2.257932
 mean_per_class_error: 0.77239877
```

Если нужно получить краткую сводку о результативности модели, то это очень практичный способ.

В следующей таблице показаны самые интересные метрики – это ошибка классификации фазы тренировки и ошибка классификации фазы проверки на каждом блоке. В случае если вы хотите оценить правильность своей модели, эти метрики можно легко сравнить:

```
print(model_cv.scoring_history())
```

	timestamp	duration	training_ speed	epochs	iter- ations	samples	training_rmse	training_logloss	training_ classification_error
0	19.06.2017 16:04	0.000 sec	None	0.000000	0	0.0	NaN	NaN	NaN
1	19.06.2017 16:04	1 min 36.263 sec	11856 obs/sec	0.220533	1	13232.0	0.880721	2.158090	0.731688
2	19.06.2017 16:04	1 min 41.646 sec	12915 obs/sec	1.321517	6	79291.0	0.475903	0.678418	0.152221
3	19.06.2017 16:04	1 min 47.022 sec	13113 obs/sec	2.424400	11	145464.0	0.397889	0.507403	0.116979
4	19.06.2017 16:04	1 min 52.319 sec	13202 obs/sec	3.533083	16	211985.0	0.369503	0.449123	0.100691
5	19.06.2017 16:04	1 min 57.559 sec	13272 obs/sec	4.630967	21	277858.0	0.345978	0.408614	0.093978
6	19.06.2017 16:04	1 min 59.856 sec	13290 obs/sec	5.072600	23	304356.0	0.344370	0.403014	**0.091017**

Ошибка классификации фазы тренировки **.091017** и точность в районе .908 на наборе данных MNIST являются довольно хорошими показателями; они почти столь же хороши, что и сверточное представление нейронной сети Яна Лекуна.

Платформа H2O также предоставляет вспомогательный метод для получения метрик проверки. Это можно сделать, передав проверочную таблицу данных в функцию перекрестной проверки:

```
model_cv.train(x=x, y=y, training_frame=train, validation_frame=test)
model_cv.show()
```

Статистика результативности:

training_rmse	training_logloss	training_ classification_error	validation_rmse	validation_logloss	validation_ classification_error
NaN	NaN	NaN	NaN	NaN	NaN
0.783207	1.640004	0.490323	0.782864	1.640282	0.4876
0.384048	0.478873	0.113892	0.378055	0.471375	0.1069
0.348065	0.416549	**0.104367**	0.344056	0.413099	**0.1009**

В этом случае можно легко сравнить ошибку классификации фазы тренировки **training_classification_error** (.1044) с ошибкой классификации фазы проверки **validation_classification_error** (.1009).

Возможно, удастся улучшить эту отметку; давайте воспользуемся моделью гиперпараметрической оптимизации.

Сеточный поиск в H2O

Учитывая, что предыдущая модель показала достаточно хорошую результативность, мы сосредоточим свои усилия на доводке параметров сетевой архитектуры. Функция H2O сеточного поиска `gridsearch` весьма похожа на рандомизированный поиск в Scikit-learn; а именно, вместо того чтобы выполнять исчерпывающий поиск по всему параметрическому пространству, она выполняет итеративный обход случайного списка параметров. Сначала мы сформируем список параметров, который мы передадим в функцию `gridsearch`. Платформа H2O предоставит нам результат для каждой модели и соответствующую отметку, полученную в результате параметрического поиска:

```
In:
import h2o
from h2o.estimators.deeplearning import H2ODeepLearningEstimator
from h2o.grid.grid_search import H2OGridSearch

h2o.init(start_h2o=True)

train_url = "https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/mnist/train.csv.gz"
test_url  = "https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/mnist/test.csv.gz"

train = h2o.import_file(train_url)
test  = h2o.import_file(test_url)

hidden_opt = [[18,18],[32,32],[32,32,32],[100,100,100]]
hyper_parameters = {"hidden":hidden_opt}

# важно: здесь мы задаем параметры поиска
# Будьте осторожны - время тренировки может иметь взрывной рост
# (см. max_models)
search_c = {"strategy":"RandomDiscrete", "max_models":10,
            "max_runtime_secs":100, "seed":222}

model_grid = H2OGridSearch(H2ODeepLearningEstimator,
                           hyper_params=hyper_parameters)

model_grid.train(x=x, y=y, distribution="multinomial",
                 epochs=1000, training_frame=train,
                 validation_frame=test, score_interval=2,
                 stopping_rounds=3, stopping_tolerance=0.05,
                 search_criteria=search_c)

print(model_grid)

# Результаты сеточного поиска для оценщика H2ODeepLearningEstimator:

Out:
    hidden \
0    [100, 100, 100]
1       [32, 32, 32]
2           [32, 32]
3           [18, 18]

   model_ids    logloss
0  Grid_DeepLearning_py_1_model_python_1464790287811_3_model_3  0.148162
1  Grid_DeepLearning_py_1_model_python_1464790287811_3_model_2  0.173675
```

```
2  Grid_DeepLearning_py_1_model_python_1464790287811_3_model_1  0.212246
3  Grid_DeepLearning_py_1_model_python_1464790287811_3_model_0  0.227706
```

Мы видим, что наша лучшая архитектура будет иметь три слоя со 100 узлами каждый. Мы также ясно видим, что функция `gridsearch` существенно увеличивает время тренировки даже на таком мощном вычислительном кластере, на котором выполняется H2O. Поэтому даже на H2O сеточный поиск следует использовать с осторожностью и быть консервативными с параметрами, которые анализируются в модели.

Теперь, прежде чем продолжить изложение, давайте завершим работу экземпляра H2O:

```
h2o.cluster().shutdown(prompt=False)
```

Глубокое обучение и предтренировка без учителя

В данном разделе мы представим самую важную методологию глубокого обучения, а именно способы улучшения обучения при помощи предварительной тренировки (или инициализации) без учителя. Нейронные сети используются с предтренировкой без учителя для отыскания в данных латентных признаков и факторов, с тем чтобы передать их дальше в нейронную сеть. Этот метод обладает мощной способностью тренировать сети для усвоения задач, которые другие методы машинного обучения не способны решать без ручного конструирования признаков. Мы познакомим со специфическими подробностями этого метода и представим новую мощную библиотеку.

Глубокое обучение с theanets

Нейросетевое приложение Scikit-learn особенно интересно для целей тонкой настройки параметров. Но, к сожалению, его возможности для нейросетевых приложений без учителя ограничены. Для следующей темы, где мы затронем более сложные методы глубокого обучения, нам потребуется еще одна библиотека. В настоящей главе мы сосредоточимся на библиотеке theanets. Разработанная Лифом Джонсоном (Lief Johnson) в Техасском университете библиотека theanets привлекательна простотой использования и своей стабильной работой. Она хорошо отшлифована и поддерживается в хорошем состоянии. Формирование нейросетевой архитектуры происходит вполне аналогично модулю sklearn; а именно инстанцируется цель обучения (классификация или регрессия), определяются слои и тренируется нейросеть. Для получения дополнительной информации относительно библиотеки можно посетить веб-страницу http://theanets.readthedocs.org/en/stable/.

Чтобы начать работать с библиотекой `theanets`, нужно ее установить менеджером библиотек `pip`:

```
$ pip install theanets
```

Поскольку библиотека theanets сконструирована поверх библиотеки Theano, также необходимо, чтобы должным образом была установлена библиотека Theano. Давайте реализуем элементарную нейросетевую модель, чтобы увидеть,

как работает библиотека theanets. Сходство с Scikit-learn будет очевидным. Отметим, что в приводимом ниже примере мы используем инерцию и что функция мягкого максимума softmax применяется в theanets по умолчанию, так что ее указывать не придется:

```
In:
import climate     # предоставляет отчетность в отношении итераций
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score,
                            mean_squared_error
from sklearn import preprocessing, datasets
import theanets
import theano

climate.enable_default_logging()
digits = datasets.load_digits()
digits = datasets.load_digits()

X = np.asarray(digits.data, 'float32')
Y = digits.target

Y = np.array(Y, dtype=np.int32)
#X = (X - np.min(X, 0)) / (np.max(X, 0) + 0.0001)  # шкалирование [0-1]

X_train, X_test, y_train, y_test = \
    train_test_split(X, Y, test_size=0.2, random_state=0)

# Построить модель классификатора с 64 входами,
# 1 скрытым слоем из 100 узлов и с 10 выходами.
net = theanets.Classifier([64,100,10])
N = 10

# Натренировать модель с использованием устойчивого метода
# обратного распространения и инерции.
net.train([X_train,y_train],
          algo='sgd', learning_rate=.001,
          momentum=0.9, patience=0,
          validate_every=N, min_improvement=0.8)

# Показать матрицы ошибок на тренировочном/контрольном наборе.
print(confusion_matrix(y_test, net.predict(X_test)))
print(accuracy_score(y_test, net.predict(X_test)))

Out:
[[27  0  0  0  0  0  0  0  0  0]
 [ 0 32  0  0  0  1  0  0  0  2]
 [ 0  1 34  0  0  0  0  1  0  0]
 [ 0  0  0 29  0  0  0  0  0  0]
 [ 0  0  0  0 29  0  0  1  0  0]
 [ 0  0  0  0  0 38  0  0  0  2]
 [ 0  1  0  0  0  0 43  0  0  0]
 [ 0  0  0  0  1  0  0 38  0  0]
 [ 0  2  1  0  0  0  0  0 36  0]
 [ 0  0  0  0  0  1  0  0  0 40]]
0.961111111111
```

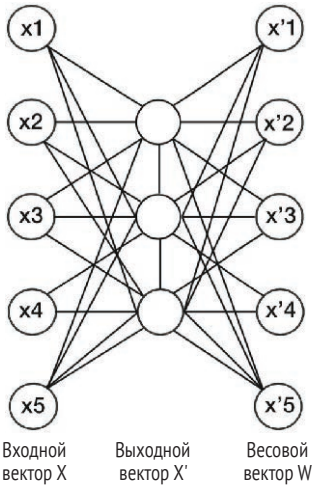
Автокодировщики и обучение без учителя

До настоящего момента в центре обсуждения были нейронные сети с многочисленными слоями и большим разнообразием параметров для оптимизации. Текущее поколение нейронных сетей, которое часто называют глубоким обучением, способно на большее; они могут автоматически обучаться новым признакам, в результате чего почти не требуется конструировать признаки и обладать предметными знаниями. Такие признаки создаются методами машинного обучения без учителя на немаркированных данных, которые дальше подаются в последующий слой нейронной сети. Метод их создания называется **предварительной тренировкой** (без учителя), и он зарекомендовал себя как чрезвычайно успешный подход в распознавании образов, усвоении языка и даже в классических проектах машинного обучения. Самая важная и доминирующая методика последних лет – это **помехоустойчивые автокодировщики** и алгоритмы на основе методов Больцмана. **Машины Больцмана**, которые были структурными элементами для **глубокой сети доверия** (deep belief network, DBN), за последнее время стали выходить из употребления в профессиональной среде глубокого обучения по причине того, что их оказалось трудно тренировать и оптимизировать. Поэтому мы сосредоточимся только на автокодировщиках. Давайте обсудим эту важную тему небольшими и понятными шагами.

Автокодировщик

Мы пытаемся найти функцию (F), которая имеет такой же выход, что и ее вход, с наименее возможной ошибкой $F(x) \approx {}'x$. Такая функция обычно называется **тождественным отображением**, которое мы стремимся оптимизировать так, чтобы x был максимально близок к $'x$. Разница между x и $'x$ называется **ошибкой реконструкции**.

Рассмотрим простую однослойную архитектуру, чтобы получить интуитивно понятное представление о том, что происходит. Мы увидим, что данная архитектура очень гибка и нуждается в тщательной тонкой настройке:



Одноуровневая архитектура автокодировщика

Важно понимать, что когда в скрытом слое узлов меньше, чем входное пространство, мы побуждаем веса сжимать входные данные.

В данном случае мы имеем набор данных с пятью признаками. В середине находится скрытый слой, содержащий три узла (W_{ij}). Эти узлы имеют то же самое свойство, что и весовой вектор, который мы видели в нейронных сетях; а именно они состоят из весов, которые могут быть натренированы методом обратного распространения ошибки. Вместе с выходом из скрытого слоя мы получаем представления признаков как результат тех же самых векторных операций по прямому распространению сигнала, что мы видели у нейронных сетей.

Процесс вычисления вектора $'x$ очень похож на тот, который мы видели у метода прямого распространения сигнала, выполняемого путем вычисления скалярных произведений весовых векторов каждого слоя:

$$h = \text{sigmoid}((W_1, x * x) + b_1(i, 1));$$

$$'x = \text{sigmoid}((W_2, x * h_i) + b_2(i, 1)).$$

Здесь W – это веса. Ошибку реконструкции можно измерить при помощи квадратической ошибки либо в перекрестно-энтропийной форме, которые мы видели в большом количестве других методов. В данном случае $\hat{y}$ обозначает реконструированный выход, а y – истинный вход:

$$\text{Перекрестная энтропия } L(x, y) = -\frac{1}{m}\sum_{i=1}^{m}[y_n \log \hat{y}_n + (1 - y_n)\log(1 - \hat{y}_n)].$$

Отметим важный принцип – при наличии всего одного скрытого слоя захваченные моделью автокодировщика размерности в данных аппроксимируют результаты **анализа главных компонент** (PCA). Однако автокодировщик ведет себя во многом по-другому, если присутствует нелинейность. Автокодировщик обнаружит различные скрытые факторы, которые метод PCA никогда не будет в состоянии обнаружить. Зная больше об архитектуре автокодировщика и то, как можно вычислить отклонение (ошибку) от ее аппроксимации тождества, теперь посмотрим на **параметры разреженности**, при помощи которых мы сжимаем входные данные.

Вы можете задаться вопросом: зачем вообще нужен этот параметр разреженности? Разве нельзя просто выполнить алгоритм, чтобы найти тождественное отображение, и идти дальше?

К сожалению, это не так уж и просто. Существуют случаи, когда тождественное отображение проецирует вход почти идеально, и при этом по-прежнему не удается извлечь латентные размерности входных признаков. В этом случае функция просто запоминает входные данные, вместо того чтобы извлечь значимые признаки. В связи с этим можно сделать две вещи. Во-первых, сознательно добавить шум к сигналу (**помехоустойчивые автокодировщики**) и, во-вторых, ввести параметр разреженности, вызывая деактивацию слабо активированных узлов. Давайте сначала посмотрим на то, как работает разреженность.

Представим порог активации биологического нейрона; нейрон может быть в *активном* состоянии, если его потенциал близок к 1, либо *неактивном*, если его выходное значение близко к 0. Можно наложить на нейроны ограничение оставаться неактивными большую часть времени, увеличив порог активации. Это делается путем уменьшения средней вероятности активации каждого нейрона/

узла. Глядя на следующую ниже формулу, мы видим, как можно минимизировать порог активации:

$$\hat{p}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^{(i)})\right].$$

Здесь $\hat{p}_j$ – это средний порог активации каждого нейрона в скрытом слое, ρ – требуемый порог активации сети, который мы определяем заранее (в большинстве случаев это значение устанавливается в .05), α – весовой вектор скрытых слоев.

Возможность для оптимизации здесь появляется, если штрафовать тренировочный раунд на коэффициент ошибки между порогами $\hat{p}_j$ и ρ.

В этой главе мы не будем слишком переживать по поводу технических подробностей этой цели оптимизации. В большинстве библиотек для этого можно воспользоваться очень простой строкой программного кода (как мы убедимся в следующем примере). Самое главное – понять, что с участием автокодировщиков имеются две основные цели обучения: уменьшение ошибки между входным вектором *x* и выходным вектором *'x* путем оптимизации тождественного отображения и уменьшение разницы между требуемым порогом активации и средней активацией каждого нейрона в сети.

Второй способ, которым можно вынудить автокодировщик обнаруживать латентные признаки, состоит во введении в модель шума; именно отсюда происходит название **помехоустойчивые автокодировщики**. Идея заключается в том, чтобы путем *повреждения* входных данных вынуждать автокодировщик обучаться более устойчивому представлению данных. В нижеследующем примере мы введем в модель автокодировщика гауссов шум, т. е. нормально распределенный.

По-настоящему глубокое обучение с каскадными помехоустойчивыми автокодировщиками – предтренировка для классификации.

Изучив это упражнение, вы станете непохожим на тех многих людей, которые говорят о глубоком обучении, в отличие от тех немногих, кто в действительности им занимается! Теперь мы применим автокодировщик к мини-версии известного набора данных MNIST, который можно легко загрузить из Scikit-learn. Напомним, что этот набор данных состоит из изображений рукописных цифр в формате 28×28 разной пиксельной интенсивности. Тренировочный набор мини-версии имеет 1797 тренировочных элементов с 64 признаками с метками для каждой записи, содержащей целевые метки с *цифрами* от 0 до 9. Таким образом, имеются 64 признака с целевой переменной, состоящей из 10 классов (цифры от 0–9) для предсказания.

Для начала натренируем модель каскадного помехоустойчивого автокодировщика с разреженностью .9 и обследуем ошибку реконструкции. В качестве руководства для настроек мы воспользуемся результатами научно-исследовательских работ по глубокому обучению. Для получения дополнительной информации можно прочесть вот эту статью: http://arxiv.org/pdf/1312.5663.pdf. Однако у нас есть некоторые ограничения вследствие огромной вычислительной нагрузки для таких типов моделей. Поэтому для создаваемого автокодировщика мы воспользуемся пятью слоями с функцией активации ReLU и сожмем данные с 64 до 45 признаков:

```
In:
model = theanets.Autoencoder([64,(45,'relu'),(45,'relu'),
                              (45,'relu'),(45,'relu'),
                              (45,'relu'),64])
dAE_model = model.train([X_train],
                        algo='rmsprop', input_noise=0.1,
                        hidden_l1=.001, sparsity=0.9,
                        num_updates=1000)
X_dAE = model.encode(X_train)
X_dAE = np.asarray(X_dAE, 'float32')
Out:
I 2017-06-14 12:25:27 theanets.layers.base:371 layer Input "in": 64 inputs
I 2017-06-14 12:25:27 theanets.layers.base:207 layer Feedforward "hid1": (in:out)64 -> 45,
relu, 2925 parameters
I 2017-06-14 12:25:27 theanets.layers.base:207 layer Feedforward "hid2": (hid1:out)45 -> 45,
relu, 2070 parameters
I 2017-06-14 12:25:27 theanets.layers.base:207 layer Feedforward "hid3": (hid2:out)45 -> 45,
relu, 2070 parameters
I 2017-06-14 12:25:27 theanets.layers.base:207 layer Feedforward "hid4": (hid3:out)45 -> 45,
relu, 2070 parameters
I 2017-06-14 12:25:27 theanets.layers.base:207 layer Feedforward "hid5": (hid4:out)45 -> 45,
relu, 2070 parameters
I 2017-06-14 12:25:27 theanets.layers.base:207 layer Feedforward "out": (hid5:out)45 -> 64,
linear, 2944 parameters
I 2017-06-14 12:25:27 theanets.graph:94 network has 14149 total parameters
valid: 45 of 45 mini-batches from (1437L, 64L)
train: 45 of 45 mini-batches from (1437L, 64L)
I 2017-06-14 12:25:27 theanets.graph:447 building computation graph
I 2017-06-14 12:25:27 theanets.losses:67 using loss: 1.0 * MeanSquaredError (output out:out)
I 2017-06-14 12:25:27 theanets.regularizers:711 regularizer: 0.1 * GaussianNoise(('in:out',))
I 2017-06-14 12:25:27 theanets.regularizers:160 regularizer: 0.001 * HiddenL1(None)
downhill: compiling evaluation function
```

Теперь, когда имеется результат работы автокодировщика, который мы создали из нового набора со сжатыми признаками, давайте рассмотрим этот новый набор данных поближе:

```
In:
X_dAE.shape

Out:
(1437, 45)
```

Здесь в действительности видно, что мы сжали данные с 64 до 45 признаков. Новый набор данных менее разрежен (т. е. с меньшим числом нулей) и численно более непрерывен. Имея в распоряжении предварительно натренированные данные из автокодировщика, теперь можно применить к ним глубокую нейронную сеть, чтобы выполнить обучение с учителем:

```
# Скрытые слои по умолчанию используют передаточную функцию relu,
# поэтому нам не нужно указывать их.
# Relu - это наилучший вариант для автокодировщиков.
# Классификатор Theanets также по умолчанию использует функцию softmax,
# поэтому нам тоже не нужно их указывать.
```

```
net   = theanets.Classifier(layers=(45,45,45,10))
autoe = net.train([X_dAE, y_train],
                  algo='rmsprop', learning_rate=.0001,
                  batch_size=110, min_improvement=.0001,
                  momentum=.9, nesterov=True, num_updates=1000)

## Получите удовольствие от редкой радости лицезреть
## 100% точность на тренировочном наборе.
```

```
Out:
I 2016-04-19 10:33:07 downhill.base:232 RMSProp 14074 loss=0.000000
err=0.000000 acc=1.000000
I 2016-04-19 10:33:07 downhill.base:232 RMSProp 14075 loss=0.000000
err=0.000000 acc=1.000000
I 2016-04-19 10:33:07 downhill.base:232 RMSProp 14076 loss=0.000000
err=0.000000 acc=1.000000
```

Прежде чем применить эту нейронную сеть для предсказания на тестовом наборе, сперва необходимо применить натренированную модель автокодировщика к тестовому набору:

```
dAE_model = model.train([X_test],
                        algo='rmsprop', input_noise=0.1,
                        hidden_l1=.001, sparsity=0.9,
                        num_updates=100)
X_dAE2=model.encode(X_test)
X_dAE2=np.asarray(X_dAE2, 'float32')
```

Теперь проверим результативность модели на тестовом наборе:

```
final=net.predict(X_dAE2)
from sklearn.metrics import accuracy_score
print accuracy_score(final,y_test)
```

```
Out:
0.972222222222
```

Мы видим, что итоговая точность модели с автокодированными признаками (.9722) превосходит точность модели без них (.9611).

Резюме

В этой главе наряду с масштабируемыми решениями мы рассмотрели самые важные понятия, лежащие в основе глубокого обучения.

Мы убрали с этой темы часть покрова секретности, изучив способы создания соответствующей архитектуры для той или иной задачи и проанализировав механизмы прямого и обратного распространений. Обновление весов нейронной сети представляет собой трудную задачу, причем обычный стохастический градиентный спуск может приводить к застреванию в глобальных минимумах или промаху. Более сложные алгоритмы, такие как инерция, ADAGRAD, RPROP и RMSProp, могут предоставить приемлемые решения. Несмотря на то что нейронные сети тренировать труднее, чем другие методы машинного обучения, они способны преобразовывать представления признаков и могут обучаться любой отдельно взятой функции (согласно универсальной теореме аппроксимации). Мы также

коснулись крупномасштабного глубокого обучения с использованием платформы H2O и даже задействовали очень горячую тему параметрической оптимизации для глубокого обучения.

Предтренировка без учителя при помощи автокодировщиков может увеличивать точность любой отдельно взятой глубокой сети, и мы проанализировали практический пример в рамках библиотеки theanets, чтобы разобраться в этой теме.

В этой главе мы прежде всего работали с библиотеками, созданными поверх платформы Theano. В следующей главе мы раскроем методы глубокого обучения на основе библиотек, созданных поверх новой платформы с открытым исходным кодом TensorFlow.

Глава 5

Глубокое обучение с библиотекой TensorFlow

В этой главе, где в центре нашего внимания будет TensorFlow, мы затронем следующие темы:

- элементарные операции TensorFlow;
- машинное обучение с нуля с TensorFlow – регрессия, классификатор на основе SGD и нейронная сеть;
- глубокое обучение с SkFlow;
- инкрементное глубокое обучение с большими файлами;
- сверточные нейронные сети с Keras.

Платформа TensorFlow была выпущена во время написания данной книги и уже зарекомендовала себя как великолепное дополнение к ландшафту машинного обучения.

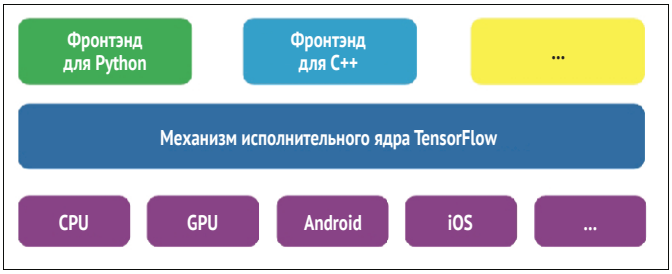
Данная платформа была запущена командой Google Brain, состоящей из большинства исследователей, которые в последнее десятилетие работали над важными разработками в области глубокого обучения (Джеффри Хинтон, Сэми Бенджио и другие). Это, в сущности, развитие платформы более раннего поколения под названием DistBelief для распределенных глубоких нейронных сетей. В отличие от TensorFlow, **DistBelief** не является платформой с открытым исходным кодом. Интересными примерами успешных проектов DistBelief является система обратного поиска изображений, Google'овская программа компьютерного зрения DeepDream и распознавание речи в приложениях Google. DistBelief позволил разработчикам Google использовать тысячи ядер (CPU и GPU) для распределенной тренировки моделей.

Платформа TensorFlow является шагом вперед, по сравнению с DistBelief, в том, что она теперь имеет абсолютно открытый исходный код, а ее язык программирования менее абстрактен. TensorFlow по праву считается более гибкой платформой, которая к тому же имеет более широкий диапазон приложений. Во время написания книги (в конце 2015 г.) платформа TensorFlow находилась на этапе становления, и, как мы увидим далее в этой главе, уже появились интересные легковесные библиотеки, созданные поверх этой платформы.

Аналогично Theano, платформа TensorFlow работает с символьными вычислениями на тензорах, т. е. большинство вычислений в платформе основывается на векторно-матричных умножениях.

В регулярных языках программирования определяются **переменные** со значениями или символами, к которым могут применяться операции. В языках символьного программирования, таких как Theano или TensorFlow, операции струк-

турированы вокруг графов, а не переменных. Это имеет свои вычислительные преимущества, поскольку они могут быть распределены и параллелизированы по всем вычислительным узлам (GPU и CPU):

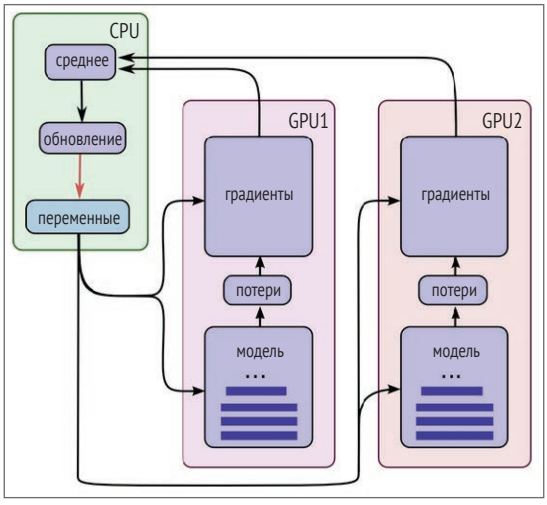


Архитектура TensorFlow, как было представлено в ноябре 2015 г.

Платформа Tensorflow имеет следующие особенности и приложения:

- платформа TensorFlow может быть (горизонтально) параллелизирована на многочисленных модулях GPU;
- наличие платформы разработки для мобильного развертывания;
- наличие инструментальной панели **TensorBoard** для визуализации (на ранней стадии);
- является фронтендом, т. е. внешним представлением, для нескольких языков программирования (Python, Go, Java, Lua, JavaScript, R, C++ и в недалеком будущем Julia);
- обеспечивает интеграцию крупномасштабных решений, таких как платформа Spark и Google-облако (https://cloud.google.com/ml/).

Идея, что тензорные операции с графоподобной структурой (согласно заявлениям компании Google) предлагают новые подходы к параллелизированным вычислениям, может стать вполне понятной, если взглянуть на следующее изображение:



Архитектура распределенной обработки с TensorFlow

Из этого рисунка становится понятно, что каждая модель может быть назначена отдельным модулям GPU, после чего вычисляется среднее значение предсказаний, получаемых из каждой модели. Среди других методов этот подход остается центральным для тренировки очень больших распределенных нейронных сетей на GPU-кластерах.

Инсталляция TensorFlow

В этой главе мы будем использовать версию TensorFlow 0.8, поэтому убедитесь, что установлена именно данная версия. Поскольку платформа TensorFlow находится в стадии активной разработки, то небольшие изменения неизбежны. Платформу TensorFlow можно довольно легко установить при помощи команды `pip install`, независимо от используемой операционной системы:

```
pip install tensorflow
```

Если уже установлены предыдущие версии, то можно обновиться в соответствии с вашей операционной системой:

```
# Ubuntu/Linux 64-разрядная, только CPU:
$ sudo pip install --upgrade https://storage.googleapis.com/
tensorflow/linux/cpu/tensorflow-0.8.1-cp27-none-linux_x86_64.whl

# Ubuntu/Linux 64-разрядная, GPU установлен:
$ sudo pip install --upgrade https://storage.googleapis.com/
tensorflow/linux/gpu/tensorflow-0.8.1-cp27-none-linux_x86_64.whl

# Mac OS X, только CPU:
$ sudo easy_install --upgrade six
$ sudo pip install --upgrade https://storage.googleapis.com/
tensorflow/mac/tensorflow-0.8.1-cp27-none-any.whl
```

Установив TensorFlow, теперь можно протестировать его в терминале:

```
$ python
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))

Out:
>>> Hello, TensorFlow!
```

Операции TensorFlow

Давайте рассмотрим несколько простых примеров, чтобы получить представление о том, как эта платформа работает.

Важное отличие платформы TensorFlow состоит в том, что сначала нам нужно инициализировать переменные и только потом можно применять к ним операции. Для выполнения вычислений TensorFlow работает с бэкендом, т. е. внутренней реализацией, на C++, поэтому, чтобы подключиться к этому бэкенду, сначала необходимо инстанцировать сеанс:

```
In:
x = tf.constant([22,21,32], name='x')
```

```
d = tf.constant([12,23,43], name='x')
y = tf.Variable(x * d, name='y')
print(y)
```

Вместо того чтобы получить на экране выходной вектор *x*d*, вы увидите что-то вроде этого:

```
Out:
<tf.Variable 'y:0' shape=(3,) dtype=int32_ref>
```

Чтобы получить фактические результаты вычисления из бэкенда на C++, необходимо инстанцировать сеанс, как показано ниже:

```
In:
x = tf.constant([22,21,32], name='x')
d = tf.constant([12,23,43], name='d')
y = tf.Variable(x * d, name='y')

model = tf.global_variables_initializer()

with tf.Session() as session:
    session.run(model)
    print(session.run(y))

Out:
[ 264 483 1376]
```

До сих пор мы использовали переменные непосредственно, однако, чтобы во время работы с тензорными операциями получить бóльшую гибкость, удобнее присваивать данные предварительно указанному контейнеру. Таким образом мы получаем возможность выполнять операции на вычислительном графе, не загружая данных в память заранее. Затем в терминах платформы TensorFlow данные передаются в граф через так называемые *заполнители* (placeholder). Именно тут проявляется сходство с языком платформы Theano (см. *приложение*, *введение в GPU и Theano*).

Заполнители в TensorFlow – это просто объектные контейнеры с определенными предварительными настройками и классами. Так, для выполнения операций на объекте сначала для этого объекта создается заполнитель вместе с соответствующим ему классом (в данном случае целым числом):

```
In:
a = tf.placeholder(tf.int32)
b = tf.placeholder(tf.int32)
sess = tf.Session()
sess.run(a+b, feed_dict={a: 111, b: 222})

Out:
333
```

Умножение матриц будет работать следующим образом:

```
In:
matrix1 = tf.constant([[1, 2,32], [3, 4,2], [3,2,11]])
matrix2 = tf.constant([[21,3,12], [3, 56,2], [35,21,61]])
product = tf.matmul(matrix1, matrix2)
```

```
with tf.Session() as sess:
    result = sess.run(product)
print(result)
```

```
Out:
[[1147  787 1968]
 [ 145  275  166]
 [ 454  352  711]]
```

Интересно отметить, что на выходе объект `result` представляет собой объект `ndarray` библиотеки NumPy, к которому можно применять операции, находясь за пределами TensorFlow.

Вычисления на GPU

Если нужно выполнять операции TensorFlow на модулях GPU, то требуется указать только устройство. Предупреждаем: следующее ниже работает только с соответствующим образом установленным CUDA-совместимым модулем GPU компании NVIDIA:

```
with tf.device('/gpu:0'):
    product = tf.matmul(matrix1, matrix2)
with tf.Session() as sess:
    result = sess.run(product)
    print(result)
```

Если вы хотите задействовать многочисленные модули GPU, то нужно присвоить каждый модуль GPU определенной задаче:

```
matrix3 = tf.constant([[13, 21,53], [4, 3,6],[3,1,61]])
matrix4 = tf.constant([[13,23,32], [23, 16,2],[35,51,31]])

with tf.device('/gpu:0'):
    product = tf.matmul(matrix1, matrix2)
with tf.Session() as sess:
    result = sess.run(product)
    print(result)

with tf.device('/gpu:1'):
    product = tf.matmul(matrix3, matrix4)
with tf.Session() as sess:
    result = sess.run(product)
    print(result)
```

Линейная регрессия с SGD

После того как мы рассмотрели основы, теперь можно приступить к написанию с нуля первого алгоритма машинного обучения на платформе TensorFlow. Позже мы воспользуемся более практическими легковесными приложениями в более высоких абстракциях поверх TensorFlow.

Мы выполним очень простую линейную регрессию со стохастическим градиентным спуском, чтобы получить представление о том, как работают процессы тренировки и оценивания модели в TensorFlow. Прежде всего мы создадим несколько рабочих переменных для их размещения в заполнителях, которые будут их содержать. Затем передадим `x` и `y` в функцию `cost` и натренируем модель градиентным спуском:

```
In:
import numpy as np
import tensorflow as tf

X = tf.placeholder("float") # создать символические переменные
Y = tf.placeholder("float")
X_train = np.asarray([1,2.2,3.3,4.1,5.2])
Y_train =  np.asarray([2,3,3.3,4.1,3.9,1.6])

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w) # предсказанные значения

cost = (tf.pow(Y-y_model, 2)) # функция стоимости как квадратическая ошибка

# оптимизация по методу sgd
train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

for trials in range(50):
    for (x, y) in zip(X_train, Y_train):
        sess.run(train_op, feed_dict={X: x, Y: y})

print(sess.run(w))

Out:
0.844732
```

Резюмируя, линейная регрессия на основе алгоритма SGD выполняется следующим образом: сначала инициализируются регрессионные веса (коэффициенты), затем на втором шаге задается функция стоимости, чтобы позже обучить и оптимизировать эту функцию при помощи градиентного спуска. В конце нужно написать цикл `for` и указать число тренировочных раундов, которые требуются для вычисления итоговых прогнозов. Аналогичная базовая структура станет очевидной во время работы с нейронными сетями.

Нейронная сеть с нуля в TensorFlow

Теперь давайте создадим и реализуем нейронную сеть на языке TensorFlow и проанализируем этот процесс.

Мы также воспользуемся набором данных Iris и в данном случае несколькими приложениями Scikit-learn для предобработки:

```
In:
import os
import logging
from datetime import datetime as dt
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn import preprocessing
from sklearn.preprocessing import OneHotEncoder
```

```
from sklearn import model_selection
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
import tensorflow as tf

iris = datasets.load_iris()
X = np.asarray(iris.data, 'float32')
X = preprocessing.scale(X)
min_max_scaler = preprocessing.MinMaxScaler()
X = min_max_scaler.fit_transform(X)

lb = preprocessing.LabelBinarizer()
Y = lb.fit_transform(iris.target)
```

Это важный шаг. Нейронные сети в TensorFlow не могут работать с целевыми метками внутри одиночного вектора. Чтобы нейронная сеть могла работать с выходом по схеме «один против всех», целевые метки должны быть преобразованы в бинаризованные признаки (так называемые фиктивные переменные):

```
X_train, x_test, y_train, y_test = \
    train_test_split(X, Y, test_size=0.3, random_state=22)

def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))
```

Ниже мы видим прямое прохождение по данным:

```
# прохождение с прямым распространением сигнала
def model(X, w_h, w_o):
    h = tf.nn.sigmoid(tf.matmul(X, w_h))
    return tf.matmul(h, w_o)

X = tf.placeholder("float32", [None, 4])
Y = tf.placeholder("float32", [None, 3])
```

Далее мы развертываем слоеную архитектуру с одним скрытым слоем:

```
w_h = init_weights([4, 4])
w_o = init_weights([4, 3])
py_x = model(X, w_h, w_o)

cost = tf.reduce_mean(   # вычислить стоимости
    tf.nn.softmax_cross_entropy_with_logits(logits=py_x, labels=Y))
train_op = tf.train.GradientDescentOptimizer( # сконструировать оптимизатор
    learning_rate=0.01).minimize(cost)

predict_op = tf.argmax(py_x, 1)

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

for i in range(500):
    for start, end in zip(range(0, len(X_train),1 ),
                          range(1, len(X_train),1)):
        sess.run(train_op, feed_dict={X: X_train[start:end],
                                      Y: y_train[start:end]})
    if i % 100 == 0:
        print(i, np.mean(np.argmax(y_test, axis=1) ==
```

```
                    sess.run(predict_op,
                             feed_dict={X: x_test, Y: y_test})))
```

```
Out:
0 0.288888888889
100 0.711111111111
200 0.933333333333
300 0.977777777778
400 0.977777777778
```

Точность этой нейронной сети составляет порядка .977%, но может от прогона к прогону давать немного различающиеся результаты. Данная отметка является более или менее целевым ориентиром для нейронной сети с единственным скрытым слоем и классическим алгоритмом SGD.

Как мы увидели в предыдущих примерах, реализация метода оптимизации и формирование тензоров вполне интуитивно понятны. И понятны больше, чем выполнение того же самого в NumPy (см. *главу 4* «*Нейронные сети и глубокое обучение*»). Единственный недостаток на данный момент состоит в том, что оценивание и предсказание требуют иногда утомительного цикла `for`, тогда как такие библиотеки, как Scikit-learn, могут предоставлять эти методы в одной строке сценария. К счастью, имеются более высокоуровневые библиотеки, которые разработаны поверх TensorFlow, и они намного упрощают процессы тренировки и оценивания. Одной из подобного рода библиотек является SkFlow; как видно из названия библиотеки, она представляет собой обертку, основанную на стиле сценариев, который работает аналогично библиотеке Scikit-learn.

Машинное обучение в TensorFlow посредством SkFlow

Теперь, когда мы познакомились с основными операциями платформы TensorFlow, давайте рассмотрим более высокоуровневые приложения, созданные поверх TensorFlow, которые сделают машинное обучение немного практичнее. И SkFlow будет первым приложением, которое мы рассмотрим. В SkFlow от нас не требуется определять типы и заполнители. Можно загружать данные и управлять ими таким же образом, как это делается в Scikit-learn и NumPy. Давайте установим библиотеку при помощи менеджера библиотек `pip`:

Простейший способ установить данную библиотеку – взять ее прямиком с репозитория Github:

```
$ pip install git+git://github.com/tensorflow/skflow.git
```

Недавно библиотека SkFlow стала составной частью платформы TensorFlow и была перемещена в папку contrib в репозитории Github http://github.com/tensorflow/tensorflow, поэтому ее установка менеджером библиотек `pip` не требуется. Чтобы воспользоваться функционалом библиотеки SkFlow, необходимо ее импортировать из TensorFlow следующим образом:

```
from tensorflow.contrib import learn as skflow
```

Этот вариант относится к версиям TensorFlow, начиная с 1.0. Описываемый в книге программный код относится к версии TensorFlow 0.8.1 и более ранним.

В SkFlow имеются три главных класса обучающихся алгоритмов: линейные классификаторы, линейная регрессия и нейронные сети. Линейный классифи-

катор – это в основном простой (мульти-) классификатор на основе алгоритма SGD, а вот реализация нейронных сетей в SkFlow превосходна. Данная библиотека предлагает относительно простые в использовании обертки для очень глубоких нейронных, рекуррентных нейронных и сверточных нейронных сетей. К сожалению, другие алгоритмы, в частности случайный лес, градиентный бустинг, SVM и наивный Байес, еще не реализованы. Впрочем, на GitHub были дискуссии по поводу реализации в SkFlow алгоритма случайного леса, который является перспективной разработкой и, вероятно, получит название tf_forest.

Давайте применим первый в SkFlow алгоритм многоклассовой классификации. Для этого примера мы будем использовать набор данных Wine, который в своем первоначальном виде взят из репозитория машинного обучения UCI. Это легкий набор данных, который состоит из 13 признаков непрерывных химических метрик, таких как магний, алкоголь, яблочная кислота и т. д., и содержит всего 178 прецедентов и один целевой признак с 3 классами. Целевая переменная состоит из трех разных культурных сортов растения. Вина классифицируются согласно соответствующему им культурному сорту растения (типу используемого для вина винограда), применяя для этого химический анализ тринадцати химических метрик. Вы видите, что мы загружаем данные из URL точно так же, как это делается во время работы в среде Scikit-learn:

```
In:
import urllib.request as urllib2
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_svmlight_file
from sklearn.model_selection import train_test_split
import skflow

url = \
'https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass/wine.scale'
set1 = urllib2.Request(url)
wine = urllib2.urlopen(set1)

X_train, y_train = load_svmlight_file(wine)
X_train = X_train.toarray()

X_train, X_test, y_train, y_test = \
    train_test_split(X_train, y_train, test_size=0.30, random_state=4)

kwargs = {
            "n_classes":4,
            "learning_rate":0.01,
            "optimizer": "SGD",
            "continue_training": True,
            "steps": 1000
}
classifier = skflow.TensorFlowLinearClassifier(**kwargs)
classifier.fit(X_train, y_train)
score = accuracy_score(y_train, classifier.predict(X_train))

d = classifier.predict(X_test)
c = accuracy_score(d,y_test)

print("Точность: %f" % score)
```

```
print('Точность на контрольном/тестовом наборе: %f' % c)
```

```
Out:
Step #1, avg. loss: 1.58672
Step #101, epoch #25, avg. loss: 1.45840
Step #201, epoch #50, avg. loss: 1.09080
Step #301, epoch #75, avg. loss: 0.84564
Step #401, epoch #100, avg. loss: 0.68503
Step #501, epoch #125, avg. loss: 0.57680
Step #601, epoch #150, avg. loss: 0.50120
Step #701, epoch #175, avg. loss: 0.44486
Step #801, epoch #200, avg. loss: 0.40151
Step #901, epoch #225, avg. loss: 0.36760
Точность на тренировочном наборе: 0.967742
Точность на тестовом наборе: 0.981481
```

К настоящему моменту этот метод уже довольно знаком; он, в сущности, представляет собой тот же способ, каким работал бы классификатор в Scikit-learn. Однако тут стоит обратить внимание на две важные вещи. С SkFlow можно использовать объекты NumPy и TensorFlow взаимозаменяемо, в результате чего нам не нужно объединять и преобразовывать объекты в тензорные таблицы и обратно. Благодаря этому работа с TensorFlow посредством такого высокоуровневого приложения, как SkFlow, становится намного гибче. Второй нюанс состоит в том, что к главному объекту данных был применен метод `toarray`. Это вызвано тем, что набор данных довольно разреженный (много нулевых записей), а TensorFlow не в состоянии хорошо обрабатывать разреженные данные.

TensorFlow показывает себя превосходно во время работы с нейронными сетями, и в SkFlow довольно просто натренировать нейронную сеть с многочисленными слоями. Давайте реализуем нейронную сеть на наборе данных Diabetes о больных диабетом. Этот набор данных содержит метрики диабета и (бинарные целевые) диагностические признаки беременных женщин в возрасте от 21 года и старше из племени индейцев Пима. У индейцев племени Пима, проживающих в шт. Аризона, США, имеется самая высокая зарегистрированная степень распространенности диабета среди населения во всем мире, и поэтому эта этническая группа стала добровольным предметом исследования диабета. Указанный набор данных состоит из следующих признаков:

- число беременностей;
- плазменная концентрация глюкозы в два часа в пероральной пробе на толерантность к глюкозе;
- диастолическое кровяное давление (мм рт. ст.);
- толщина кожных складок трицепса (мм);
- 2-часовой инсулин в сыворотке (мкМЕ/мл);
- индекс массы тела (вес в кг/(высота в м)^2);
- функция диабетической наследственности;
- возраст (лет);
- переменная класса (0 либо 1).

В приведенном ниже примере мы сначала загружаем и шкалируем данные:

```
In:
import urllib.request as urllib
import numpy as np
```

```
from sklearn import preprocessing
from sklearn.preprocessing import Normalizer
from sklearn import datasets, metrics, model_selection
from sklearn.model_selection import train_test_split

# набор данных Diabetes (Репозиторий машинного обучения UCI)
url = "http://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-
indians-diabetes.data"
# скачать файл
raw_data = urllib.urlopen(url)
dataset = np.loadtxt(raw_data, delimiter=",")

print(dataset.shape)

X = dataset[:,0:7]
y = dataset[:,8]
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.2, random_state=0)

X = preprocessing.scale(X)
min_max_scaler = preprocessing.MinMaxScaler()
X = min_max_scaler.fit_transform(X)
```

Этот шаг особенно интересен; чтобы нейронные сети лучше сходились, можно использовать более гибкие скорости затухания. Во время тренировки многослойных нейронных сетей обычно полезно со временем уменьшать темп обучения. Вообще говоря, когда темп обучения слишком высокий, то можно промахнуться по оптимуму. С другой стороны, когда он слишком низкий, мы впустую тратим вычислительные ресурсы и застреваем в локальном минимуме. Экспоненциальное затухание – это метод, который со временем ослабляет темп обучения, в результате чего он становится более чувствительным, когда начинает приближаться к минимуму. Существуют три распространенных способа реализовать затухание темпа обучения, а именно пошаговое затухание, затухание 1/t и экспоненциальное затухание:

Экспоненциальное затухание: $\alpha = \alpha_0 e^{-kt}$.

В данном случае α – это темп обучения, k – гиперпараметр и t – итерация.

В этом примере мы будем использовать экспоненциальное затухание, потому что оно очень хорошо зарекомендовало себя на этом наборе данных. Вот как мы реализуем функцию экспоненциального затухания (при помощи встроенной в TensorFlow функции `tf.train.exponential_decay`):

```
In:
import tensorflow
import tensorflow as tf

def exp_decay(global_step):
    return tf.train.exponential_decay(learning_rate=0.01,
                                      global_step=global_step,
                                      decay_steps=steps,
                                      decay_rate=0.01)
```

Теперь можно передать функцию затухания в модель нейронной сети TensorFlow. В этой нейронной сети мы будем использовать двуслойную сеть, в которой первый слой будет состоять из пяти узлов и второй – из четырех узлов. По умол-

чанию в SkFlow реализована функция активации ReLU, и поскольку мы отдаем ей предпочтение по сравнению с другими (tanh, сигмоидальная и т. д.), то будем придерживаться этой функции.

Согласно этому примеру, кроме алгоритма стохастического градиентного спуска (SGD), можно реализовать и другие алгоритмы оптимизации. Давайте реализуем адаптивный алгоритм под названием Adam, основываясь на статье Дидерика Кингма (Diederik Kingma) и Джимми Ба (Jimmy Ba) (http://arxiv.org/abs/1412.6980).

Алгоритм Adam разработан в Амстердамском университете и расшифровывается как adaptive moment estimation (метод адаптивной оценки инерции). В предыдущей главе мы узнали, как работает алгоритм ADAGRAD, – путем сокращения градиентов во времени, по мере того как они движутся (в надежде) к глобальному минимуму. В алгоритме Adam тоже используются адаптивные методы, но в сочетании с идеей тренировки с инерцией, где учитываются предыдущие обновления градиента:

```
In:
from tensorflow.contrib import learn as skflow

steps = 5000
classifier = skflow.DNNClassifier(hidden_units=[5,4],
                                  n_classes=2,
                                  batch_size=300,
                                  steps=steps,
                                  optimizer='Adam', # SGD/RMSProp
                                  # назначаем функцию затухания
                                  learning_rate=exp_decay)

classifier.fit(X_train, y_train)
score1a = metrics.accuracy_score(y_train, classifier.predict(X_train))
score1b = metrics.accuracy_score(y_test, classifier.predict(X_test))

print("Точность: %f" % score1a)
print("Точность на контрольном наборе: %f" % score1b)

Out:
(768, 9)
Step #1, avg. loss: 12.83679
Step #501, epoch #167, avg. loss: 0.69306
Step #1001, epoch #333, avg. loss: 0.56356
Step #1501, epoch #500, avg. loss: 0.54453
Step #2001, epoch #667, avg. loss: 0.54554
Step #2501, epoch #833, avg. loss: 0.53300
Step #3001, epoch #1000, avg. loss: 0.53266
Step #3501, epoch #1167, avg. loss: 0.52815
Step #4001, epoch #1333, avg. loss: 0.52639
Step #4501, epoch #1500, avg. loss: 0.52721
Точность на тренировочном наборе: 0.754072
Точность на тестовом наборе: 0.740260
```

Полученный показатель точности не особо убедителен; можно было бы его улучшить, применив ко входу алгоритм **анализа главных компонент** (PCA). В этой статье за 1999 г. Ставроса Дж. Перантониса (Stavros J Perantonis) и Вассилиса Вирвилиса (Vassilis Virvilis) (http://rexa.info/paper/dc4f2babc5ca4534b435280ae

c32f5816ddb53b0) было показано, что набор данных Diabetes извлекает пользу из снижения размерности методом PCA перед его передачей в нейронную сеть. Для этого набора данных мы воспользуемся конвейерным объектом `Pipeline` библиотеки Scikit-learn:

```
In:
from sklearn.decomposition import PCA
from sklearn import linear_model, decomposition, datasets
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score

pca = PCA(n_components=4,whiten=True)

lr = pca.fit(X)
classifier = skflow.TensorFlowDNNClassifier(hidden_units=[5,4],
                                            n_classes=2,
                                            batch_size=300,
                                            steps=steps,
                                            optimizer='Adam', # SGD/RMSProp
                                            learning_rate=exp_decay)

pipe = Pipeline(steps=[('pca', pca), ('NNET', classifier)])

X_train, X_test, Y_train, Y_test = \
    train_test_split(X, y, test_size=0.2, random_state=0)
pipe.fit(X_train, Y_train)

score2 = metrics.accuracy_score(Y_test, pipe.predict(X_test))

print("Точность на контрольном наборе с pca: %f" % score2)

Out:
Step #1, avg. loss: 1.07512
Step #501, epoch #167, avg. loss: 0.54236
Step #1001, epoch #333, avg. loss: 0.50186
Step #1501, epoch #500, avg. loss: 0.49243
Step #2001, epoch #667, avg. loss: 0.48541
Step #2501, epoch #833, avg. loss: 0.46982
Step #3001, epoch #1000, avg. loss: 0.47928
Step #3501, epoch #1167, avg. loss: 0.47598
Step #4001, epoch #1333, avg. loss: 0.47464
Step #4501, epoch #1500, avg. loss: 0.47712
Точность на тестовом наборе с PCA: 0.805195
```

У нас получилось чуть-чуть улучшить результативность нейронной сети за счет простого шага с предобработкой на основе алгоритма PCA. Мы сократили число признаков с семи до четырех, т. е. четырех размерностей. Алгоритм PCA обычно сглаживает сигнал путем центрирования признаков на нуле, тем самым уменьшая признаковое пространство путем использования векторов, которые содержат только самое высокое *собственное значение*. **Выбеливание** (whitening) гарантирует, что признаки преобразуются в нуль-коррелированные. Оно приводит к более гладкому сигналу и меньшему набору признаков, что позволяет нейронной сети сходиться быстрее. См. *главу 7* «*Обучение без учителя в крупном масштабе*» для получения более подробного объяснения метода PCA.

Глубокое обучение с большими файлами – инкрементное обучение

До сих пор мы имели дело с несколькими операциями TensorFlow и методами машинного обучения в SkFlow на относительно малых наборах данных. Однако настоящая книга посвящена крупномасштабному и масштабируемому машинному обучению. Что же платформа TensorFlow предлагает в этом отношении?

До недавнего времени параллельные вычисления находились на этапе становления и были недостаточно стабильными, чтобы освещать их в этой книге. Вычисления на многочисленных модулях GPU недоступны для читателей без CUDA-совместимых карт NVIDIA. Крупномасштабные облачные сервисы (https://cloud.google.com/products/machine-learning/) или Amazon EC2 сопровождаются значительными финансовыми затратами. Таким образом, остается всего один способ, которым можно масштабировать наш проект, – путем пошагового обучения.

Вообще говоря, любой размер файла, который превышает порядка 25% имеющейся оперативной памяти компьютера, вызовет проблемы, связанные с ее перегрузкой. Поэтому, если вы располагаете компьютером с 2 Гб RAM и хотите применить технологию машинного обучения к файлу объемом 500 МБ, то пора задуматься о способах, которые позволяют избежать излишнего потребления памяти.

Для того чтобы предотвратить перегрузку оперативной памяти, мы советуем использовать метод внеядерного обучения, который разбивает данные на меньшие по объему порции с целью инкрементной тренировки и обновления моделей. Методы частичной подгонки в библиотеке Scikit-learn, которые мы затронули в *главе 2* «*Масштабируемое обучение в Scikit-learn*», тому являются примерами.

Библиотека SkFlow тоже предлагает великолепный метод пошагового обучения, аналогичный методу частичной подгонки в Scikit-learn, для всех своих моделей машинного обучения. В этом разделе мы будем использовать классификатор глубокого обучения в инкрементном режиме, потому что считаем его самым перспективным.

В данном разделе для нашего масштабируемого проекта глубокого внеядерного обучения мы будем использовать две стратегии, а именно пошаговое обучение и случайную подвыборку.

Сначала сгенерируем немного данных, а затем создадим функцию отбора наблюдений, которая будет извлекать случайные подвыборки из синтетического набора данных и инкрементно тренировать модель глубокого обучения на этих подмножествах:

```
In:
import gc
import random
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import tensorflow as tf
import skflow
```

Прежде всего сгенерируем немного демонстрационных данных и запишем их на диск:

```
X, y = make_classification(n_samples=5000000, n_features=10,
                           n_classes=4, n_informative=6,
                           random_state=222, n_clusters_per_class=1)
X_train, X_test, y_train, y_test = \
    train_test_split(X,y, test_size=0.2, random_state=22)

Big_trainm = pd.DataFrame(X_train, y_train)
Big_testm  = pd.DataFrame(X_test, y_test)

Big_trainm.to_csv('lsml-Bigtrainm', sep=',')
Big_testm.to_csv('lsml-Bigtestm', sep=',')
```

Теперь освободим память, удалив из нее все созданные нами объекты. Методом `gc.collect` мы заставляем сборщик «мусора» языка Python очистить память:

```
del(X, y, X_train, y_train, X_test)
gc.collect
```

Ниже мы создаем функцию, которая извлекает случайные подвыборки из диска. Отметим, что используется доля выборки $^1/_2$. Можно было бы использовать доли поменьше, но в этом случае нам также пришлось бы скорректировать две важные вещи. Во-первых, необходимо привести в соответствие размер пакета модели глубокого обучения, так чтобы он никогда не превышал объема выборки. Во-вторых, необходимо скорректировать количество эпох в цикле `for` таким образом, чтобы гарантировать использование для тренировки модели самой большой части тренировочных данных:

```
In:
import random
import pandas as pd

def sample_file():
    global skip_idx
    global train_data
    global X_train
    global y_train
    big_train='lsml-Bigtrainm'
```

Подсчитаем число строк во всем наборе:

```
    num_lines = sum(1 for i in open(big_train))
```

Мы используем одну третью долю тренировочного набора:

```
    size = int(num_lines / 3)
```

Берем индексы пропуска и сохраняем:

```
    skip_idx = random.sample(range(1, num_lines), num_lines - size)
    train_data = pd.read_csv(big_train, skiprows=skip_idx)
    X_train = train_data.drop(train_data.columns[[0]], axis=1)
    y_train = train_data.ix[:,0]
```

В предыдущем разделе мы уже видели, как работает затухание весов. Ниже мы воспользуемся им снова:

```
def exp_decay(global_step):
    return tf.train.exponential_decay(learning_rate=0.01,
                                      global_step=global_step,
                                      decay_steps=steps,
                                      decay_rate=0.01)
```

Далее мы развернем глубокий нейросетевой (DNN) классификатор с тремя скрытыми слоями, соответственно с 5, 4 и 4 узлами. Отметим, что мы задаем размер пакета равным 300, т. е. в каждой эпохе используются 300 тренировочных случаев. Этот размер также помогает предотвратить перегрузку памяти:

```
steps = 5000
clf = skflow.TensorFlowDNNClassifier(hidden_units=[5,4,4],
                                     n_classes=4,
                                     batch_size=300,
                                     steps=steps,
                                     optimizer='Adam',
                                     learning_rate=exp_decay)
```

Ниже мы задаем количество подвыборок, равное трем (epochs=3), т. е. мы инкрементно тренируем модель глубокого обучения на трех последовательных подвыборках:

```
epochs = 3
for i in range(epochs):
    sample_file()
    clf.partial_fit(X_train,y_train)

test_data = pd.read_csv('lsml-Bigtestm', sep=',')
X_test = test_data.drop(test_data.columns[[0]], axis=1)
y_test = test_data.ix[:,0]
score  = accuracy_score(y_test, clf.predict(X_test))

print(score)

Out:
Step #501, avg. loss: 0.55220
Step #1001, avg. loss: 0.31165
Step #1501, avg. loss: 0.27033
Step #2001, avg. loss: 0.25250
Step #2501, avg. loss: 0.24156
Step #3001, avg. loss: 0.23438
Step #3501, avg. loss: 0.23113
Step #4001, avg. loss: 0.23335
Step #4501, epoch #1, avg. loss: 0.23303
Step #1, avg. loss: 2.57968
Step #501, avg. loss: 0.57755
Step #1001, avg. loss: 0.33215
Step #1501, avg. loss: 0.27509
Step #2001, avg. loss: 0.26172
Step #2501, avg. loss: 0.24883
Step #3001, avg. loss: 0.24343
Step #3501, avg. loss: 0.24265
Step #4001, avg. loss: 0.23686
Step #4501, epoch #1, avg. loss: 0.23681
0.929022
```

Нам удалось получить точность на тестовом наборе в размере .929 в течение очень управляемого времени тренировки модели и без перегрузки оперативной памяти, что значительно быстрее, чем если бы мы тренировали эту же модель на всем наборе данных сразу.

Инсталляция библиотеки Keras и платформа TensorFlow

Ранее мы видели практические примеры оберточного функционала библиотеки SkFlow для приложений платформы TensorFlow. Для реализации более продвинутого подхода к нейронным сетям и глубокому обучению, когда имеется больший контроль над параметрами, мы предлагаем библиотеку Keras (http://keras.io/). Данная библиотека была первоначально разработана в рамках платформы Theano, но совсем недавно была также адаптирована под TensorFlow. Таким образом, мы можем использовать библиотеку Keras в качестве более высокого абстрактного уровня поверх платформы TensorFlow. Впрочем, следует иметь в виду, что библиотека Keras в своих методах является немного менее прямолинейной, чем SkFlow. Библиотека Keras может работать как на GPU, так и на CPU, что делает эту библиотеку действительно гибкой при ее переносе в другие среды.

Давайте сначала установим библиотеку Keras и удостоверимся, что она использует бэкенд платформы TensorFlow.

Ее инсталляция выполняется в командной строке при помощи pip:

```
$ pip install Keras
```

Библиотека Keras исходно создавалась поверх платформы Theano, поэтому мы должны сообщить библиотеке Keras вместо этой платформы задействовать платформу TensorFlow. Для этого нам сначала нужно запустить Keras один раз на собственной стандартной платформе, т. е. платформе Theano.

Сначала мы выполним небольшой программный код в Keras, чтобы убедиться, что все элементы библиотеки должным образом установлены. Давайте натренируем элементарную нейронную сеть и по ходу познакомимся с некоторыми ключевыми понятиями.

Ради удобства мы используем данные, сгенерированные в Scikit-learn, которые состоят из четырех признаков и целевой переменной из трех классов. Эти размерности очень важны, потому что они нам нужны для определения архитектуры нейронной сети:

```
In:
import time
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import keras
from keras.utils import np_utils, generic_utils
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD
```

```
nb_classes = 3
X, y = make_classification(n_samples=1000, n_features=4,
                           n_classes=nb_classes, n_informative=3,
                           n_redundant=0, random_state=101)
```

Теперь, когда мы определили переменные, необходимо преобразовать целевую переменную в прямокодированный массив (точно так же, как это делалось в платформе TensorFlow). В противном случае библиотека Keras не сможет вычислить целевые выходы по схеме «один против всех». Для Keras, вместо прямого унитарного кодировщика модуля sklearn, мы будем использовать функцию `np_utils`. Это делается следующим образом:

```
y = np_utils.to_categorical(y, nb_classes)
print(y)
```

Наш массив `y` будет выглядеть так:

```
Out:
[[ 0.  0.  1.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]
 ...,
 [ 0.  0.  1.]
 [ 0.  0.  1.]
 [ 0.  1.  0.]]
```

Теперь разделим данные на тестовые и тренировочные:

```
x_train, x_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.30, random_state=101)
```

С этого места мы начинаем придавать форму задуманной нами архитектуре нейронной сети. Нейронная сеть будет с двумя скрытыми слоями, функцией активации `relu` и тремя узлами в каждом скрытом слое. Первый слой имеет четыре входа, потому что в данном случае у нас четыре признака. После этого мы добавляем скрытые слои с тремя узлами, отсюда `model.add(dense(3)`.

Как и ранее, мы воспользуемся функцией мягкого максимума `softmax`, чтобы передать сеть в выходной слой:

```
model = Sequential()
model.add(Dense(4, input_shape=(4,)))
model.add(Activation('relu'))
model.add(Dense(3))
model.add(Activation('relu'))
model.add(Dense(3))
model.add(Activation('softmax'))
```

Сначала мы определяем функцию SGD, где реализуем самые важные параметры, которые к настоящему моменту нам знакомы, а именно:

- **lr**: темп обучения;
- **decay**: функция затухания для замедления темпа обучения. Не путайте этот параметр с затуханием веса, которое является параметром регуляризации;
- **momentum**: параметр инерции используется, чтобы предотвратить застревание в локальном минимуме;

- **nesterov**: булева переменная, которая определяет, будем ли мы использовать инерцию Нестерова; эта переменная применима, только если мы определили целое число для параметра инерции (обратитесь к *главе 4* «*Нейронные сети и глубокое обучение*» за более подробным объяснением);
- **optimizer**: здесь мы определяем предпочтительный алгоритм оптимизации (из следующего списка: SGD, RMSProp, ADAGRAD, Adadelta и Adam).

Обратимся к следующему фрагменту исходного кода:

```
seed = 22   # использовать для воспроизводимости результатов
np.random.seed(seed)

model = Sequential()
model.add(Dense(4, input_shape=(4,)))
model.add(Activation('relu'))
model.add(Dense(3))
model.add(Activation('relu'))
model.add(Dense(3))
model.add(Activation('softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)
model.fit(x_train, y_train, verbose=1, batch_size=100,
          epochs=50, validation_data=(x_test, y_test))
time.sleep(0.1)
```

В данном случае мы использовали размер пакета `batch_size`, равный `100`, т. е. использовался мини-пакетный градиентный спуск со 100 тренировочными примерами в каждой эпохе. В этой модели были задействованы `50` тренировочных эпох. В итоге мы получим следующий результат:

```
Out:
acc: 0.8129 - val_loss: 0.5391 - val_acc: 0.8000
Train on 700 samples, validate on 300 samples
```

В последней модели, в которой мы использовали алгоритм SGD с инерцией Нестерова, мы не смогли улучшить отметку точности независимо от того, сколько эпох мы использовали для ее тренировки.

Для увеличения точности желательно испытать другие алгоритмы оптимизации. В более раннем примере мы уже успешно применили метод оптимизации Adam, поэтому давайте здесь применим его снова и посмотрим, сможем ли мы увеличить точность. Поскольку алгоритмы с адаптивным темпом обучения, в частности Adam, со временем снижают темп обучения, для достижения оптимального решения требуется больше эпох. Поэтому в приведенном ниже примере мы зададим количество эпох, равное 200:

```
adam = keras.optimizers.Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=adam)
model.fit(x_train, y_train, verbose=1, batch_size=100,
          epochs=200, validation_data=(x_test, y_test))
time.sleep(0.1)

Out:
Epoch 200/200
700/700 [==============================] - 0s - loss: 0.3755 - acc:
0.8657 - val_loss: 0.4725 - val_acc: 0.8200
```

В этот раз, используя алгоритм оптимизации Adam, нам удалось достичь убедительного улучшения с 0.8 до 0.82.

К настоящему моменту мы охватили самые важные элементы нейронных сетей в библиотеке Keras. Теперь приступим к настройке библиотеки Keras, чтобы она могла задействовать платформу TensorFlow. По умолчанию библиотека Keras будет использовать бэкенд Theano. Чтобы заставить Keras работать на TensorFlow, сначала нужно найти местоположение папки библиотеки Keras в папке с библиотеками:

```
In:
import os
print(keras.__file__)
```

Ваш путь может выглядеть по-другому:

```
Out:
/Library/Python/2.7/site-packages/keras/__init__.pyc
```

Определилив местоположение папки библиотеки Keras, теперь необходимо найти файл `~/.keras/keras.json`.

В Windows папка .keras с файлом `keras.json` находится в пользовательском домашнем каталоге (`C:\Users\[ПОЛЬЗОВАТЕЛЬ]\.keras`).

В этом файле есть фрагмент сценария, который выглядит так:

```
{"epsilon": 1e-07, "floatx": "float32", "backend": "theano"}
```

Вам просто нужно поменять `"backend": "theano"` на `"backend": "tensorflow"`, и в результате получится следующая строка:

```
{"epsilon": 1e-07, "floatx": "float32", "backend": "tensorflow"}
```

Если по каким-либо причинам файл JSON в папке Keras отсутствует, т. е. в папке `/Library/Python/2.7/site-packages/keras/`, то можно просто скопировать и вставить показанную ниже строку в текстовый редактор:

```
{"epsilon": 1e-07, "floatx": "float32", "backend": "tensorflow"}
```

Затем сохранить файл с расширением `.json` и разместить его в папке `keras`.

Чтобы проверить, что среда TensorFlow должным образом используется из Keras, можно набрать следующее:

```
In:
from keras import backend as K
input = K.placeholder(shape=(4, 4, 5))
# работает так же:
input = K.placeholder(shape=(None, 2, 5))
# работает так же:
input = K.placeholder(ndim=2)

Out:
Using Theano backend.
```

Некоторые пользователи, возможно, не увидят результата вообще, что тоже допустимо. Ваш бэкенд TensorFlow должен быть готов к использованию.

Сверточные нейронные сети в TensorFlow посредством Keras

Между настоящей и предыдущей главами мы прошли достаточно длинный путь, охватив самые важные темы глубокого обучения. Теперь мы понимаем, как конструировать архитектуры, размещая каскадом многочисленные слои нейронной сети, и различать и применять методы обратного распространения ошибки. Мы также затронули понятие предварительной тренировки без учителя с каскадными помехоустойчивыми автокодировщиками. Следующий и действительно захватывающий шаг в глубоком обучении представлен быстро развивающейся областью **сверточных нейронных сетей** (Convolutional Neural Networks, CNN), т. е. методом построения многослойных локальносвязных сетей. CNN-сети, также именуемые сетями **ConvNet**, так быстро развиваются, что во время написания этой книги нам пришлось переписать и обновить эту главу буквально в течение одного месяца. В этой главе мы осветим большинство фундаментальных и важных понятий, которые лежат в основе CNN-сетей, с тем чтобы можно было выполнить несколько элементарных примеров, не перегружая себя иногда их огромной сложностью. Однако мы не сможем в полной мере отдать должное обширным теоретическим и вычислительным предпосылкам CNN-сетей, вследствие чего этот абзац обеспечит лишь практическую отправную точку[1].

Лучший способ концептуально разобраться в CNN-сетях состоит в том, чтобы вернуться в прошлое, начать с небольшого обзора когнитивной нейробиологии и рассмотреть исследование Хьюбера (Huber) и Визеля (Wiesel) в области зрительной зоны коры головного мозга. Хьюбер и Визель регистрировали нейроактивации зрительной зоны у кошек при измерении нейронной активности путем введения микроэлектродов в зрительную зону коры головного мозга. (Бедные кошки!) Они делали это в то время, когда кошки наблюдали проецируемые на экран примитивные изображения фигур. Самое интересное, что в результате наблюдений они обнаружили, что определенные нейроны откликались только на фигуры определенной ориентации или формы. Этот факт позволил выдвинуть теоретическое предположение, что зрительная зона коры состоит из локальных и специфичных для ориентации нейронов. Другими словами, определенные нейроны откликаются только на изображения характерной ориентации и формы (треугольники, круги или квадраты). Учитывая, что кошки и другие млекопитающие могут воспринимать как единое целое сложные и изменяющиеся фигуры, можно предпо-

---

[1] Свое название сверточная сеть получила по названию операции – свертка (convolution), она часто используется для обработки изображений и может быть описана следующей формулой:

$(f * g)[m, n] = \sum_{k,l} f[m - k, n - l] * g[k, l] (f * g)[m, n] = \sum_{k,l} f[m - k, n - l] * g[k, l]$.

Здесь f – исходная матрица изображения, g – ядро (матрица) свертки.

Неформально эту операцию можно описать следующим образом: окном размера ядра `g` проходим с заданным шагом (обычно 1) все изображение `f`, на каждом шаге поэлементно умножаем содержимое окна на ядро `g`, результат суммируется и записывается в матрицу результата.

При этом в зависимости от метода обработки краев исходной матрицы результат может быть меньше исходного изображения (valid), такого же размера (same) или большего размера (full). – *Прим. перев.*

ложить, что восприятие является агрегатом всех этих локально и иерархически организованных нейронов. К тому времени уже в полной мере были разработаны первые многослойные персептроны, и поэтому прошло совсем немного времени, прежде чем эта идея локальности и характерной чувствительности в нейронах была смоделирована в персептронной архитектуре. С точки зрения вычислительной нейробиологии эта идея развилась в карты локальных рецептивных областей в мозгу с добавлением селективно-связных слоев. Эта идея была воспринята уже существующей областью нейронных сетей и искусственного интеллекта. Первым, кто заявил, что применил это понятие локальной специфичности в вычислительном плане к многослойному персептрону, был Фукусима (Fukushima) с его так называемым неокогнитроном (1982).

Ян Лекун разработал идею неокогнитрона в своей версии под названием LeNet с добавлением алгоритма обратного распространения методом градиентного спуска. Архитектура LeNet по-прежнему остается основой для многих более эволюционно развитых архитектур CNN, которые были внедрены в последнее время. Элементарная CNN-сеть, как и LeNet, учится обнаруживать края у необработанных пикселов в первом слое, затем использует эти края для обнаружения простых форм во втором слое и позже в процессе использует эти формы для обнаружения высокоуровневых признаков, таких как объекты в окружающей среде в более высоких слоях. Слой, который находится в нейронной последовательности глубже, является заключительным классификатором, который использует эти высокоуровневые признаки. Мы видим прохождение по CNN-сети с прямым распространением сигнала следующим образом: мы переходим от матричного входа к пикселам, обнаруживаем края, исходя из пикселей, затем фигуры из краев и обнаруживаем все более и более отличительные и более абстрактные и сложные признаки из фигур.

Каждая свертка или слой в сети восприимчивы к определенному признаку (такому как фигура, угол или цвет).
Более глубокие слои будут сочетать эти признаки в более многосложный агрегат. Таким образом, сеть может обрабатывать цельные изображения, не обременяя себя полным входным пространством изображения на этом шаге.

До сих пор мы работали только с полносвязными нейронными сетями, каждый слой которых подсоединен к каждому соседнему слою. Такие сети доказали свою достаточную эффективность, но они имеют недостаток существенно увеличивать число параметров, которые мы должны натренировать. Следует отметить, что в случае когда мы тренируем небольшое по размеру изображение (28×28), можно обойтись и полносвязной сетью. Однако тренировка полносвязной сети на более крупных изображениях, которые занимают всю площадь изображения, была бы в вычислительном плане чрезвычайно затратной.

Резюмируя, можем констатировать, что CNN-сети обладают следующими преимуществами перед полносвязными нейронными сетями:

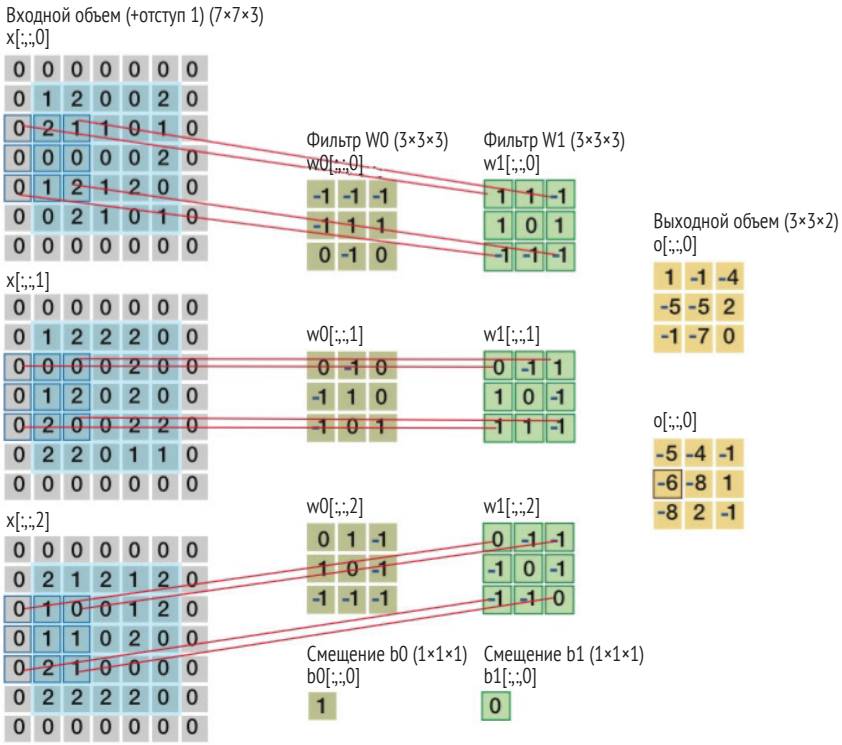
- они уменьшают пространство параметров и тем самым предотвращают переподгонку и вычислительную перегрузку;
- CNN-сети инвариантны для ориентации объектов (представьте распознавание объектов в различных местах с целью их классификации);
- CNN-сети способны обучаться и обобщать сложные многомерные признаки;
- CNN-сети могут быть полезными в распознавании речи, классификации изображений и в последнее время в сложных рекомендательных механизмах.

В CNN-сетях задействованы так называемые рецептивные поля для соединения входа с картой признаков. Лучший способ разобраться в работе CNN-сетей состоит в том, чтобы глубже проанализировать их архитектуру и, конечно же, получить практический опыт. Поэтому давайте рассмотрим типы слоев, которые составляют CNN-сети. Архитектура CNN-сети состоит из трех типов слоев, а именно сверточного слоя, объединяющего слоя и полносвязного слоя, где каждый слой принимает входной 3D-объем (h, w, d) и преобразует его в 3D-выход посредством дифференцируемой функции.

Сверточный слой

Значение свертки можно понять, если представить прожектор определенного размера, скользящий по входным данным (пиксельные значения и размерности RGB-цвета), после чего вычисляется скалярное произведение между отфильтрованными значениями (также именуемыми заплатками – patch) и истинным входом. Благодаря этому выполняются две важные вещи: во-первых, сжимаются данные на входе, и, во-вторых, что еще более важно, сеть обучается фильтрам, которые активируются, только когда на входе они видят некий определенный тип положения признака в пространстве.

Посмотрим на следующее ниже изображение, чтобы понять, как это работает:



Введены два сверточных уровня, обрабатывающих изображение [7×7×3] входной объем: изображение ширины 7, высоты 7 и с тремя цветовыми каналами R, G, B

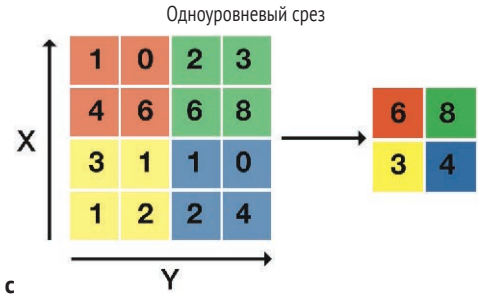
По этому рисунку видно, что имеются два уровня фильтров (W_0 и W_1) и три размерности входов (в форме массивов), и результатом всего этого является скалярное произведение прожектора/окна на входной матрице. Размер этого прожектора называется *шагом*; это означает, что чем больше шаг, тем меньше конечный результат.

Как видно по рисунку, когда мы применяем фильтр 3×3, весь объем фильтра обрабатывается в центре матрицы, но когда мы смещаемся близко к краям либо за их пределы, мы начинаем его терять на краях входа. В этом случае применяется так называемое *дополнение нулями*. Все элементы, которые попадают за пределы входных размерностей, в этом случае обнуляются. Дополнение нулями в последнее время стало более или менее использоваться в настройках по умолчанию в большинстве приложений на основе CNN-сетей.

Объединяющий слой

Следующий тип слоя, который часто помещается в промежутке между фильтрующими слоями, называется объединяющим (pooling) слоем, или так называемым *субдискретизирующим* слоем, т. е. слоем с уменьшением размерности. Его работа заключается в отборе данных с пониженной частотой вдоль пространственных размерностей (ширина, высота), что, в свою очередь, помогает решать проблемы с переподгонкой и сокращением вычислительной нагрузки. Существует несколько способов выполнения субдискретизации, но в последнее время наиболее эффективным оказался метод объединения по максимальному значению элемента max pooling.

Метод выбора максимального элемента – это простой метод сжатия признаков путем взятия максимального значения заплатки соседнего признака. Следующий ниже рисунок разъяснит эту идею; каждая цветная ячейка в матрице представляет собой подвыборку с шагом 2:



Объединяющий слой
с выбором максимального элемента с шагом 2

Объединяющие слои используются в основном по следующим причинам:

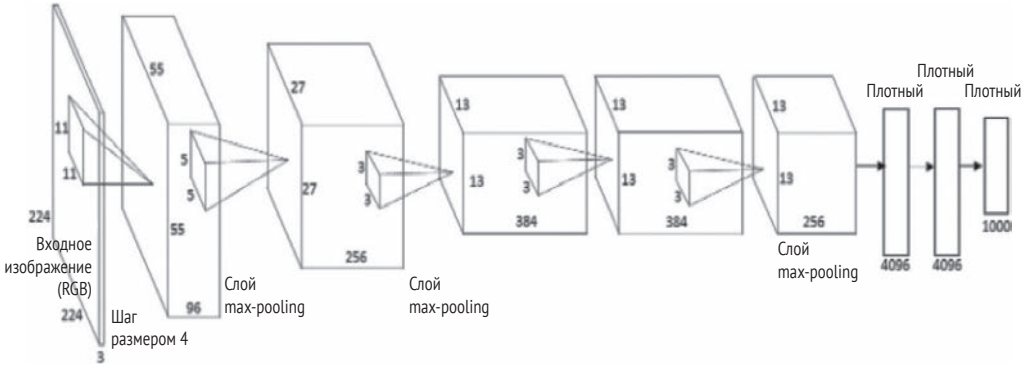
- уменьшение количества параметров и тем самым вычислительной нагрузки;
- регуляризация.

Интересно отметить, что последние результаты исследований предлагают не учитывать объединяющего слоя совсем, что приводит к улучшению точности (правда, за счет большей нагрузки на CPU или GPU).

Полносвязный слой

По поводу этого типа слоя особо нечего комментировать. Конечный выход, где вычисляются классификации (главным образом функцией мягкого максимума softmax), является полносвязным слоем. Однако в промежуточных сверточных слоях (хотя и редко) тоже имеются полносвязные слои.

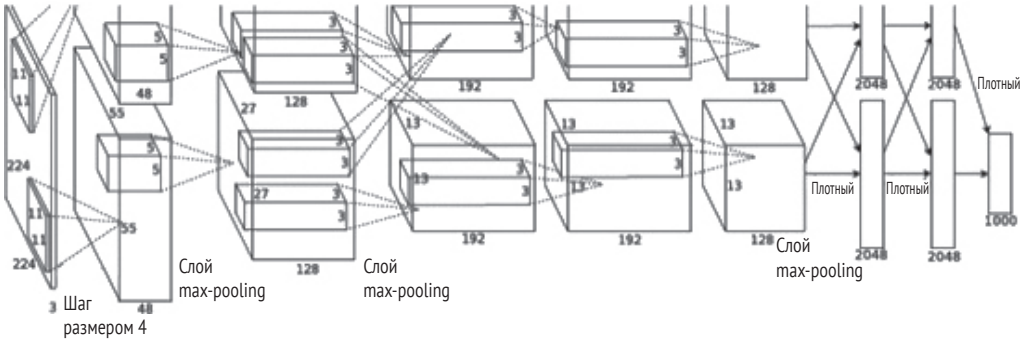
Прежде чем мы самостоятельно применим CNN-сеть, давайте возьмем то, чему вы научились к этому времени, и обследуем архитектуру CNN, чтобы проверить наше понимание. Когда мы смотрим на архитектуру ConvNet на следующем ниже рисунке, мы уже имеем представление о том, что именно ConvNet делает со входом. В качестве примера представлена эффективная сверточная нейронная сеть под названием AlexNet, предназначенная для классификации 1,2 млн изображений на 1000 классов. Она использовалась для конкурса ImageNet в 2012 г. ImageNet – это самое важное соревнование в области классификации и локализации изображений в мире, которое проходит каждый год. В самом названии AlexNet имеются в виду ее создатели Алекс Крыжевски (Alex Krizhevsky) в сотрудничестве с Вайнодом Наиром (Vinod Nair) и Джеффри Хинтоном (Geoffrey Hinton).



Архитектура AlexNet

Разглядывая архитектуру сети, сразу же видна входная размерность 224×224 с трехмерной глубиной. Шаг в размере 4 во входе, где объединяющие слои max pooling уложены каскадно, уменьшает размерность входа. В свою очередь, за ними следует сверточный слой. Два плотных слоя размером 4096 – это полносвязные слои, которые ведут к конечному выходу, о чем говорилось ранее.

В предыдущем абзаце мы уже упомянули, что в платформе TensorFlow вычисления на графах позволяют выполнять параллелизацию по модулям GPU. К сведению: авторы нейронной сети AlexNet сделали то же самое. Взгляните на следующий ниже рисунок. На нем показано, каким образом они параллелизировали архитектуру по всем модулям GPU:



Классификация в ImageNet
на основе глубоких сверточных нейронных сетей, NIPS 2012

Приведенный выше рисунок взят из работы ImageNet Classification with Deep Convolutional Neural Networks (ImageNet – классификация на основе глубокой сверточной нейронной сети) http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf.

Нейронная сеть AlexNet позволяет различным моделям использовать модули GPU путем вертикального разбиения архитектуры, чтобы в дальнейшем ее объединить в конечный результат классификации. Подобного рода CNN-сети больше всего подходят для распределенной обработки, что является одним из самых больших преимуществ локальносвязных сетей над полносвязными. Данная модель была натренирована на наборе из 1,2 млн изображений, и на это потребовалось 5 дней, чтобы завершить вычисления на двух 3-гигабайтных модулях GPU NVIDIA GTX 580. Для этого проекта использовались два блока с несколькими GPU (в общей сложности шесть модулей GPU).

CNN-сети с подходом на основе инкрементной тренировки

Теперь, когда у нас есть порядочное понимание архитектуры CNN-сетей, пора закатать рукава и применить CNN-сеть в библиотеке Keras.

Для этого примера мы воспользуемся известным набором данных с изображениями объектов CIFAR-10, который легкодоступен в домене Keras. Набор данных состоит из 60 000 цветных изображений в формате 32×32 с 10 целевыми классами, включая самолет, автомобиль, птицу, кошку, оленя, собаку, лягушку, лошадь, корабль и грузовик. Это меньший по объему набор данных, чем тот, который использовался для примера с сетью AlexNet. Для получения дополнительной информации в отношении этого набора данных можно обратиться на веб-страницу https://www.cs.toronto.edu/~kriz/cifar.html.

В данной CNN-сети мы будем использовать следующую ниже архитектуру для классификации изображения в соответствии с упомянутыми 10 классами:

```
вход -> свертка 1 (32,3,3) -> свертка 2 (32,3,3) -> объединяющий слой -> прореживание -> выход
(полносвязный слой и функция мягкого максимума)
```

Вычисления на GPU

Если у вас установлена CUDA-совместимая видеокарта, то для этого примера CNN вы можете использовать графический процессор, разместив в своем IDE следующий фрагмент кода в верхней части программы:

```
import os
os.environ['THEANO_FLAGS'] = \
  'device=gpu0, assert_no_cpu_op=raise, on_unused_input=ignore, floatX=float32'
```

Мы, правда, рекомендуем сначала опробовать этот пример на обычном CPU.

Прежде всего выполним импорт необходимых библиотек и подготовим данные.

Мы используем входной размер 32×32, учитывая, что это фактический размер изображения:

```
In:
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.utils import np_utils

batch_size = 32
nb_classes = 10
nb_epoch = 5 # число эпох, пользоваться осторожно, чтобы не испортить cpu/gpu

img_rows, img_cols = 32, 32  # размерности входного изображения
img_channels = 3             # изображения CIFAR10 имеют цветность RGB

# данные перемешиваются и разбиваются на тренировочный и тестовый наборы
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

print('Форма массива X_train:', X_train.shape)
print(X_train.shape[0], 'тренировочных прецедентов')
print(X_test.shape[0], 'тестовых прецедентов')

# напоминаем, что нужно закодировать целевую переменную
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

Теперь давайте развернем архитектуру CNN-сети и построим модель в соответствии с задуманной архитектурой.

Для этого примера мы натренируем модель CNN с использованием классического алгоритма SGD и инерции Нестерова:

```
model = Sequential()

# первый сверточный слой, устанавливаем размер фильтра
model.add(Convolution2D(32, 3, 3,
                        border_mode='same',
                        input_shape=(img_channels,
                                     img_rows, img_cols)))
model.add(Activation('relu'))
```

```
# второй сверточный слой
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))

# задаем объединяющий слой
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# сначала сглаживаем вход в полносвязный слой для передачи в функцию softmax
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

# тренируем модель при помощи алгоритма SGD + инерции, как и ранее
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# применяем к признакам шкалирование
X_train /= 255
X_test /= 255
```

Этот шаг очень важен, потому что здесь мы определяем CNN-сеть для ее инкрементной тренировки. В предыдущих главах (см. *главу 2* «*Масштабируемое обучение в Scikit-learn*») и в предыдущем абзаце мы убедились в вычислительной эффективности онлайнового и инкрементного обучения. Мы можем сымитировать некоторые его свойства и применить его к CNN-сетям, используя очень небольшой размер эпохи с меньшим размером пакета `batch_size` (долей тренировочного набора в каждой эпохе), и натренировать их инкрементно в цикле `for`. Таким образом можно получить одинаковое количество эпох и натренировать CNN-сеть за гораздо более короткое время и с более низкой нагрузкой на оперативную память. Эта очень мощная идея может быть реализована на основе простого цикла `for` следующим образом:

```
for epoch in xrange(nb_epoch):
    model.fit(X_train, Y_train, batch_size=batch_size,
              epochs=1, validation_data=(X_test, Y_test), shuffle=True)

Out:
Форма массива X_train: (50000, 32, 32, 3)
50000 тренировочных прецедентов
10000 тестовых прецедентов
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1480s - loss: 1.4464 -
acc: 0.4803 - val_loss: 1.1774 - val_acc: 0.5785
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1475s - loss: 1.0701 -
acc: 0.6212 - val_loss: 0.9959 - val_acc: 0.6525
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1502s - loss: 0.8841 -
acc: 0.6883 - val_loss: 0.9395 - val_acc: 0.6750
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1555s - loss: 0.7308 -
acc: 0.7447 - val_loss: 0.9138 - val_acc: 0.6920
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1587s - loss: 0.5972 -
acc: 0.7925 - val_loss: 0.9351 - val_acc: 0.6820
```

Выше показан процесс тренировки CNN-сети, в результате которого мы в итоге приходим к точности на проверочном наборе порядка 0.7. Учитывая, что мы натренировали многосложную модель на высокоразмерном наборе данных из 50 000 тренировочных примеров с 10 целевыми классами, этот результат уже достаточно убедительный. Максимально возможная отметка, которая может быть достигнута с CNN-сетью на таком наборе данных, требует, по крайней мере, 200 эпох. Предложенный в этом примере метод ни в коем случае не является окончательным. Это довольно-таки элементарная реализация, чтобы приступить к работе с CNN-сетями. Не стесняйтесь экспериментировать, добавляя или удаляя слои, корректируя размер пакета и т. д. Поиграйте с параметрами, чтобы почувствовать, как они работают.

Если вы хотите больше узнать о последних наработках относительно сверточных слоев, обратитесь к **остаточной сети** (ResNet), которая представляет собой одно из последних усовершенствований в архитектуре CNN-сетей.

Кайминг Хе (Kaiming He) и другие разработчики сети этого типа стали победителями в состязании ImageNet за 2015 г. (ILSVRC). Она располагает интересной архитектурой, в которой используется метод пакетной нормализации, или батч-нормализации, чья задача состоит в нормализации преобразования признаков между слоями. В библиотеке Keras имеется функция пакетной нормализации, с которой можно было бы поэкспериментировать (http://keras.io/layers/normalization/).

Чтобы получить общее представление о последнем поколении сетей ConvNet, следует ознакомиться со следующими настройками параметров для CNN-сетей, которые были признаны более эффективными:

- малый шаг;
- затухание веса (регуляризация вместо прореживания);
- отсутствие прореживания;
- пакетная нормализация между слоями среднего уровня;
- меньший объем предварительной тренировки или полное ее отсутствие (автокодировщики и машины Больцмана постепенно выходят из употребления в классификации образов).

Еще одним интересным моментом является то, что в последнее время сверточные сети стали использоваться в других приложениях, которые выходят за пределы идентификации образов. Они используются для классификации языков и текстов, заполнения пропусков в предложении и даже в рекомендательных системах. Любопытным примером является музыкальный рекомендательный механизм

Spotify, который основан на сверточных нейронных сетях. Чтобы получить о нем дополнительную информацию, можно пройти по указанным ниже ссылкам:

- http://benanne.github.io/2014/08/05/spotify-cnns.html;
- http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2013_5004.pdf.

В настоящее время сверточные сети используются в следующих областях:

- распознавание лиц (Facebook);
- классификация кинолент (YouTube);
- обработка речи и текста;
- генеративное искусство (к примеру, Google DeepDream);
- рекомендательные механизмы (рекомендация музыкальных композиций – Spotify).

Резюме

В этой главе мы проделали по-настоящему длинный путь, охватив среду TensorFlow и соответствующие методы. Мы познакомились с тем, как настраивать элементарные регрессоры, классификаторы и нейронные сети с единственным скрытым слоем. Несмотря на то что программирование операций платформы TensorFlow для стандартных задач машинного обучения выполняется относительно прямолинейно, платформа TensorFlow может быть немного утомительной. Именно здесь на первый план выходит более высокоуровневая библиотека SkFlow с интерфейсом, весьма похожим на Scikit-learn. Для инкрементных решений или даже внеядерных решений SkFlow предоставляет метод частичной подгонки, который можно легко настроить. Другие крупномасштабные решения либо ограничены приложениями на основе GPU, либо находятся на преждевременной стадии. Поэтому в том, что касается масштабируемых решений, на данный момент мы должны остановиться на стратегии инкрементного обучения.

Мы также предоставили введение в сверточные нейронные сети и увидели, как они могут реализовываться в библиотеке Keras.

Глава 6

Классификационные и регрессионные деревья в крупном масштабе

В этой главе мы сосредоточимся на масштабируемых методах для классификационных и регрессионных деревьев. При этом будут затронуты следующие темы:

- советы и рекомендации для приложений на основе быстрых случайных лесов с использованием Scikit-learn;
- аддитивные модели со случайными лесами и подвыборкой;
- алгоритм градиентного бустинга GBM;
- алгоритм XGBoost вместе с потоковыми методами;
- очень быстрый алгоритм GBM и случайный лес в среде H2O.

Цель дерева решений состоит в том, чтобы, основываясь на тренировочных данных, обучиться серии решающих правил, которые позволяют прийти к заключению о целевых метках. Процесс обучения на основе рекурсивного алгоритма начинается в корне дерева и далее расщепляет данные по признаку, который приводит к самой низкой неоднородности. В настоящее время самое широкое распространение получили масштабируемые приложения с построением деревьев решений на основе алгоритма CART. Данный алгоритм был разработан в 1984 г. профессорами статистики Брейманом (Breiman), Фридманом (Friedman), Стоуном (Stone) и Олшеном (Ohlson), а его название является аббревиатурой от Classification and Regression Trees, т. е. **классификационные и регрессионные деревья**. Алгоритм CART отличается от других моделей на основе дерева решений (таких как ID3, C4.5/C5.0, CHAID и MARS) двумя аспектами. Во-первых, CART применим к задачам классификации и регрессии. Во-вторых, он создает бинарные деревья (каждое расщепление в результате является бинарным). Это позволяет деревьям CART рекурсивно оперировать заданными признаками и в жадном режиме выполнять оптимизацию на метрике ошибки в форме неоднородности. Подобного рода бинарные деревья вместе с масштабируемыми решениями находятся в центре внимания настоящей главы.

Давайте приглядимся к тому, каким образом эти деревья сконструированы. Мы видим дерево решений в виде графа с узлами, в котором информация передается сверху вниз, начиная с самой вершины. Каждое решение внутри дерева принимается путем бинарных расщеплений как для классов (булева переменная), так и для

непрерывных переменных (пороговое значение), которые приводят к итоговому предсказанию.

Деревья конструируются и обучаются при помощи следующей процедуры.

Выполняется рекурсивный поиск переменной, которая наилучшим образом расщепляет целевую метку от корня до терминального узла. Данный процесс измеряется коэффициентом неоднородности (impurity) для каждого признака, который мы минимизируем, основываясь на целевом исходе. В настоящей главе используются соответствующие меры неоднородности: индекс Джини и перекрестная энтропия.

Индекс Джини:

$$Gini(S) = 1 - \sum_{i=1}^{k} p_i^2.$$

Индекс Джини (Gini impurity) – это метрика, которая измеряет степень расхождения между вероятностями p_i целевых классов (k) таким образом, что равный разброс значений вероятностей по целевым классам приводит к высокому индексу Джини[1].

Перекрестная энтропия:

$$D = -\sum_{i=1}^{k} \hat{p}_{mk} \log \hat{p}_{mk}.$$

В перекрестной энтропии мы смотрим на логарифмическую вероятность неправильной классификации. Обе метрики, как было доказано, приводят к очень похожим результатам. Однако индекс Джини в вычислительном плане более эффективен, потому что не требует вычисления логарифмов.

Мы продолжаем, пока не будет удовлетворен критерий останова. Этот критерий может примерно означать две вещи: во-первых, добавление новых переменных больше не улучшает целевого результата, и, во-вторых, достигнута максимальная глубина дерева, или порог сложности дерева. Отметим, что очень глубокие и сложные деревья со многими узлами могут легко привести к переподгонке. Для ее предотвращения мы обычно подрезаем дерево путем ограничения его глубины.

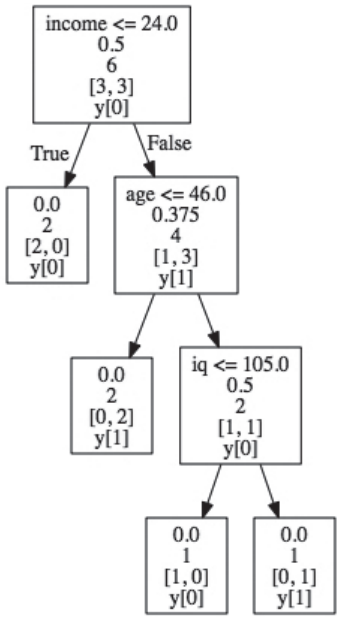
Для получения интуитивно понятного представления о том, как этот процесс работает, создадим дерево решений при помощи Scikit-learn и визуализируем его в программе graphviz. Прежде всего создадим миниатюрный набор данных, который позволит предсказывать курильщика, основываясь на следующих признаках: IQ (число), возраст (число), годовой доход (число), предприниматель (булева переменная) и университетский диплом (булева переменная). Программу graphviz необходимо скачать с http://www.graphviz.org; она потребуется для загрузки и визуализации файла `tree.dot`, который мы собираемся создать при помощи Scikit-learn:

---

[1] Этот показатель был разработан итальянским экономистом, статистиком и демографом Коррадо Джини (1884–1965 гг.). В статистике это количественный показатель, который показывает степень неравенства различных вариантов распределения доходов и называется коэффициентом Джини. Процентное представление этого коэффициента называется индексом Джини, и, в частности, в машинном обучении он применяется для предсказания непрерывных величин, где его смысл – погрешность должна быть настолько равномерной, т. е. однородной, насколько можно. – *Прим. перев.*

```
In:
import numpy as np
from sklearn import tree

iq  = [90,110,100,140,110,100]
age = [42,20,50,40,70,50]
anincome = [40,20,46,28,100,20]
businessowner = [0,1,0,1,0,0]
univdegree = [0,1,0,1,0,0]
smoking = [1,0,0,1,1,0]

ids   = np.column_stack((iq, age, anincome, businessowner, univdegree))
names = ['iq', 'age', 'income', 'univdegree']

dt = tree.DecisionTreeClassifier(random_state=99)

dt.fit(ids,smoking)
dt.predict(ids)
tree.export_graphviz(dt, out_file='data\\tree2.dot',
                     feature_names=names, label=all,
                     max_depth=5, class_names=True)
```

Теперь в рабочем каталоге можно найти файл `tree.dot`. Найдя его там, откройте этот файл в программе graphviz:



- Корневой узел (доход): стартовый узел, который представляет признак с самым высоким приростом информации и самой низкой неоднородностью (Gini = .5).
- Внутренние узлы (возраст и IQ): это каждый узел между корневым узлом и терминальным. Родительские узлы передают решающие правила вниз по направлению к принимающей стороне – дочерним узлам (левому и правому).

- Терминальные узлы (листовые узлы): целевые метки, разделенные иерархической структурой.

Глубина дерева – это число дуг из корневого узла до терминальных узлов. В данном случае глубина дерева равна 3.

Теперь мы видим все бинарные расщепления, которые получились в результате генерирования дерева. Вверху в корневом узле мы видим, что человек с доходом ниже 24k не является курильщиком (доход < 24). В каждом узле мы также видим соответствующий уровень неоднородности, т. е. индекс Gini (.5) для данного расщепления. Левосторонние дочерние узлы отсутствуют, потому что решение является окончательным. В этом месте путь просто заканчивается, потому что он полностью разделяет целевой класс. Однако в правостороннем дочернем узле дохода (возраст) дерево ответвляется. Здесь, если возраст меньше или равен 46, этот человек не курильщик, но с возрастом более 46 и IQ ниже 105 этот человек – курильщик. В такой же степени важно несколько созданных нами признаков – диплом и предпринимательство, но они не являются частью дерева. Это вызвано тем, что переменные в дереве способны расклассифицировать целевые метки без них. Эти пропущенные признаки просто не участвуют в уменьшении уровня неоднородности в дереве.

Одиночные деревья имеют недостатки – они легко поддаются переподгонке и потому плохо обобщаются на ранее не встречавшихся данных. Тренировка текущего поколения этих методов выполняется при помощи ансамблевых методов, где одиночные деревья агрегируются в намного более мощные модели. Такие ансамблевые CART-методы получили в машинном обучении самое широкое распространение вследствие их точности, простоты использования и возможности обрабатывать гетерогенные данные. Эти методы успешно применялись на недавних конкурсах в области науки о данных, в частности Kaggle и KDD-cup. Поскольку ансамблевые методы для классификационных и регрессионных деревьев в настоящее время в мире ИИ и науки о данных являются нормой, масштабируемые решения для ансамблевых CART-методов будут основной темой настоящей главы.

Мы обычно различаем два класса ансамблевых методов, которые используются с CART-моделями, а именно бэггинг и бустинг. Давайте разберемся в этих понятиях, чтобы сформировать достаточное понимание в отношении того, как работает процесс создания ансамбля.

Агрегация бутстрапированных выборок

Бэггинг – это аббревиатура от англ. термина bootstrap aggregation, т. е. **агрегация бутстрап-выборок**. Метод бутстрапирования первоначально появился в контексте, где аналитикам приходилось иметь дело с дефицитом данных. На основе этого статистического подхода подвыборки использовались для оценки параметров генеральной совокупности, когда статистическое распределение невозможно было вычислить априорно. Цель бустрапирования состоит в том, чтобы обеспечить более устойчивую оценку параметров генеральной совокупности, где бóльшая вариабельность вносится в меньший по объему набор данных случайной подвыборкой с возвратом.

Обычно бутстрапирование проходит по следующему базовому сценарию.

1. Взять из отдельно взятого набора данных произвольную выборку размером *x* с возвратом.

2. Вычислить метрику или параметр из каждой выборки с целью оценки параметров генеральной совокупности.
3. Агрегировать результаты.

В последние годы бутстрап-методы стали использоваться и для параметров моделей машинного обучения. Ансамбль показывает свою наибольшую эффективность, когда его классификаторы обеспечивают весьма разнообразные границы решения. Это разнообразие в ансамблях может быть достигнуто за счет разнообразия его базовых моделей и данных, на которых выполняется тренировка этих моделей. Деревья очень хорошо подходят для такого разнообразия среди классификаторов, потому что структура деревьев может быть крайне изменчивой. Однако самый популярный метод ансамблирования состоит в использовании разных тренировочных наборов данных для тренировки отдельных классификаторов. Нередко подобного рода наборы данных получают посредством методов подвыборки-выборки, в частности методами бутстрапирования и бэггинга. Все началось с идеи, что путем привлечения большего количества данных мы можем уменьшить дисперсию в оценочных значениях. Если нет возможности иметь под рукой больший объем данных, то повторная выборка может обеспечить существенное улучшение ситуации, потому что она позволяет выполнять повторную тренировку алгоритма на многих версиях тренировочной выборки. Как раз отсюда появилась идея беггинга; используя бэггинг, мы продвигаем идею бутстрапирования еще немного дальше за счет агрегации (к примеру, усреднения) результатов многих повторных выборок, с тем чтобы в итоге прийти к заключительному предсказанию, где ошибки из-за внутривыборочной переподгонки взаимно сглаживаются.

Когда мы применяем ансамблевый метод, в частности бэггинг, к древовидным моделям, мы создаем многочисленные деревья на каждой отдельной бустрапированной выборке (т. е. подвыборке с использованием отбора прецедентов без возврата) из исходного набора данных и затем агрегируем результаты (обычно путем арифметического, геометрического усреднения или голосования).

В таком варианте стандартный алгоритм бэггинга выглядит следующим образом.

1. Из полного набора данных взять n случайных выборок размером K с возвратом (S_1, S_2, ..., S_n).
2. Натренировать отдельные деревья на выборках (S_1, S_2, ..., S_n).
3. Вычислить предсказания на новых данных, исходя из выборок (S_1, S_2, ..., S_n), и агрегировать их результаты.

CART-модели извлекают огромную выгоду из методов бэггинга вследствие вносимой ими стохастичности и разнообразия.

Случайный лес и экстремально рандомизированный лес

Помимо бэггинга, который основан на тренировочных примерах, случайные подвыборки могут также извлекаться на основе признаков. Такой метод называется методом **случайных подпространств**. Метод случайных подпространств особенно полезен для высокоразмерных данных (данных с большим числом признаков) и является основой метода, который называется «случайным лесом». Во время написания этой главы алгоритм случайного леса являлся самым популярным алгоритмом машинного обучения вследствие своей простоты использования,

устойчивости к грязным данным и параллелизируемости. Он нашел свое применение в самых разнообразных приложениях, в том числе в приложениях позиционирования, играх и методах скрининга в медицинских программах. Например, в игровой приставке Xbox Kinect используется модель обнаружения движения на основе случайного леса. Учитывая, что алгоритм случайного леса основывается на методах бэггинга, он является относительно прямолинейным.

1. Взять из имеющейся выборки m бустрап-выборок размера N.
2. Независимо строить деревья на каждом подмножестве ($S_1, S_2, \ldots, S_n$) с использованием в каждом расщеплении узла разной доли набора признаков G (без возврата).
3. Минимизировать ошибку расщеплений узлов (основываясь на индексе Джини или энтропийной мере).
4. Дать каждому дереву сделать предсказание и агрегировать результаты с использованием голосования для классификации и усреднения для регрессии.

Поскольку бэггинг опирается на многочисленные подвыборки, он является превосходным кандидатом для параллелизации, где каждый модуль CPU выделяется для вычисления отдельных моделей. В силу этого обучение можно ускорить благодаря широкой доступности многочисленных ядер. Правда, в такой стратегии масштабирования имеется предел, поскольку мы должны осознавать, что Python – это однопоточный язык, и нам придется реплицировать множество экземпляров Python, причем каждый будет тиражировать область памяти с задействованными в ней внутривыборочными примерами. Следовательно, мы должны иметь в распоряжении достаточно много оперативной памяти, чтобы в ней уместились тренировочная матрица и ряд процессов. Если объема доступной RAM не будет хватать, то задание числа параллельных древовидных вычислений, работающих на компьютере одновременно, масштабировать алгоритм не поможет. В этом случае память RAM и использование CPU являются серьезными узкими местами.

Модели на основе случайных лесов довольно просты для применения в машинном обучении, потому что для своей качественной работы они не требуют какой-то особой доводки гиперпараметров. Самые важные параметры, представляющие интерес, – это *количество деревьев* и глубина деревьев, имеющие наибольшее влияние на результативность модели. Оперирование этими двумя гиперпараметрами приводит к компромиссу между точностью и результативностью, когда большее число деревьев и бóльшая глубина приводят к более высокой вычислительной нагрузке. Как показывает наш практический опыт, рекомендуется не устанавливать значение *количества деревьев* слишком высоким, потому что в конечном счете модель достигнет плато результативности и не будет больше улучшаться при добавлении большего количества деревьев, а попросту приведет к чрезмерной нагрузке на ядра CPU. Исходя из таких соображений, несмотря на то что модель на основе случайного леса со стандартными параметрами выполняется хорошо «из коробки», мы по-прежнему можем увеличить ее результативность путем настройки числа деревьев. Посмотрим на приведенную ниже таблицу с обзором гиперпараметров для случайных лесов.

Самые важные параметры для бэггинга с алгоритмом случайного леса следующие:

- `n_estimators`: число деревьев в модели;
- `max_features`: число признаков, используемых для конструирования дерева;

- `min_sample_leaf`: расщепление узла удаляется, если терминальный узел содержит меньше выборок, чем минимальное значение;
- `max_depth`: число узлов, которые мы передаем сверху вниз от корня до терминального узла;
- `criterion`: метод, используемый для вычисления оптимального расщепления (Джини или энтропия);
- `min_samples_split`: минимальное число выборок, требуемых для расщепления внутреннего узла.

Библиотека Scikit-learn предоставляет широкий спектр мощных ансамблевых CART-приложений, некоторые из которых в вычислительном плане вполне эффективны. В том, что касается случайных лесов, существует нередко упускаемый из виду алгоритм, который называется extra-trees, более известный как **экстремально рандомизированный лес** (extremely randomized forest). И когда речь заходит об эффективности CPU, алгоритм extra-trees может обеспечить значительное ускорение по сравнению с обычными случайными лесами, иногда даже в десятки раз.

В следующей ниже таблице вы видите скорость вычисления для каждого метода. Алгоритм extra-trees значительно быстрее с более явным отрывом, когда мы увеличиваем объем выборки:

Объем выборки	Экстральные деревья, сек.	Случайный лес, сек.
100 000	25.9	164
50 000	9.95	35.1
10 000	2.11	6.3

Модели тренировались 50 признаками и 100 оценщиками, как для экстремально рандомизированного леса extra-trees, так и для случайного леса. Для этого примера мы использовали четырехъядерный MacBook Pro с 16 ГБ RAM и измеряли затраченное на тренировку время в секундах.

В этом абзаце мы будем использовать метод экстремальных лесов вместо классического метода случайного леса из Scikit-learn. Поэтому вы справедливо можете задать вопрос: чем они отличаются? Различие не поражает своей сложностью. В случайных лесах правила решения в расщеплении узла опираются на наилучшую отметку, получаемую в результате случайного отбора признаков во время каждой итерации. В чрезвычайно рандомизированных лесах случайное расщепление генерируется на каждом признаке в случайном подмножестве (и потому отсутствуют вычисления, расходуемые на поиск лучшего расщепления для каждого признака), и затем отбирается порог с наилучшей отметкой. Такого рода подход приносит с собой некоторые выгодные свойства, потому что этот метод приводит к получению моделей с более низкой дисперсией, несмотря на то что каждое отдельное дерево строится до тех пор, пока не будет иметь самую большую точность в терминальных узлах. Поскольку в расщепление ветвей добавляется больше случайности, древовидный ученик делает ошибки, которые систематически менее коррелированы среди деревьев в ансамбле. Это приводит к намного более некоррелированным оценочным значениям в ансамбле и, в зависимости от задачи обучения (в конце концов, бесплатные обеды отсутствуют), к более низкой

ошибке обобщения, чем стандартный ансамбль из случайных лесов. На практике, однако, обычная модель случайного леса может обеспечить слегка более высокую точность.

С учетом таких интересных свойств обучения с более эффективным вычислением расщепления узла и тем же параллелизмом, который задействован в случайных лесах, мы рассматриваем экстремально рандомизированные деревья в качестве превосходного кандидата в разряд ансамблевых алгоритмов на основе дерева решений, если хотим ускорить внутриядерное обучение.

Для получения подробного описания алгоритма экстремально рандомизированного леса можно почитать следующую ниже статью, с которой все началось: Geurts D. Ernst и L. Wehenkel, *Extremely randomized trees, Machine Learning*, 63 (1), 3–42, 2006 («Экстремально рандомизированные деревья»). Данная статья находится в свободном доступе, и ее можно скачать по прямой ссылке https://www.semanticscholar.org/paper/Extremely-randomized-trees-Geurts-Ernst/336a165c17c9c56160d332b9f4a2b403fccbdbfb/pdf.

В качестве примера того, как масштабировать внутриядерные древовидные ансамбли, мы выполним пример, где применим эффективный метод случайного леса для кредитных данных Credit. Этот набор данных используется для предсказания для клиента соответствующего уровня его кредитной карты. Данные состоят из 18 признаков и 30 000 тренировочных примеров. Поскольку нужно импортировать файл в формате XLS, необходимо установить библиотеку `xlrd`; это можно сделать, набрав в терминальном окне командной строки следующее:

```
$ pip install xlrd
```

```
import os
import xlrd
import urllib
import pandas as pd
import numpy as np
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.datasets import make_classification
```

```
#os.chdir('/ваш-путь')  # задать свой путь
path = 'data'           # в данном случае подкаталог в главе 6
url  = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00350/default%20of%20
credit%20card%20clients.xls'
filename = 'creditdefault.xls'
fullfilename = os.path.join(path, filename)
urllib.urlretrieve(url, fullfilename)
```

```
data = pd.read_excel('data/creditdefault.xls', skiprows=1)
```

```
target = 'default payment next month'
y = np.asarray(data[target])
features = data.columns.drop(['ID', target])
X = np.asarray(data[features])
```

```
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.30, random_state=101)
```

```
clf = ExtraTreesClassifier(n_estimators=500, random_state=101)
```

```
clf.fit(X_train,y_train)
scores = cross_val_score(clf, X_train, y_train,
                         cv=3, scoring='accuracy', n_jobs=-1)

print('Перекрестно-проверочная точность: среднее = %0.3f стд.откл = %0.3f'
      % (np.mean(scores), np.std(scores)))

Out:
Перекрестно-проверочная точность: среднее = 0.812 стд.откл = 0.003
```

Теперь, когда есть некая базовая оценка точности на тренировочном наборе, посмотрим на результативность модели на тестовом наборе. В этом случае мы хотим проследить ложноположительные и ложноотрицательные исходы, а также выполнить проверку на несбалансированность классов на целевой переменной:

```
In:
from sklearn.metrics import confusion_matrix, accuracy_score

y_pred = clf.predict(X_test)
confusionMatrix = confusion_matrix(y_test, y_pred)

print(confusionMatrix)
print('Общая точность на тестовом наборе %f'
      % accuracy_score(y_test, y_pred))

Out:
[[6610  448]
 [1238  704]]
Общая точность на тестовом наборе 0.812667
```

Интересно отметить, что точность тестового набора равна результатам на тренировочном наборе. Поскольку наша базовая модель функционирует в условиях по умолчанию, можно попытаться улучшить ее результативность путем доводки гиперпараметров, т. е. задачи, которая может оказаться в вычислительном плане дорогостоящей. Недавно были разработаны вычислительно более эффективные методы гиперпараметрической оптимизации, которых мы коснемся в следующем разделе.

Быстрая параметрическая оптимизация посредством рандомизированного поиска

Возможно, вы уже знакомы с функционалом сеточного поиска в библиотеке Scikit-learn. В целом это отличный инструмент, но когда дело доходит до больших файлов, то в зависимости от пространства параметров он может значительно увеличивать продолжительность тренировки. Для экстремальных случайных лесов мы можем ускорить время вычисления доводки параметров при помощи альтернативного метода параметрического поиска, который называется **рандомизированным поиском**. Там, где обычный исчерпывающий сеточный поиск приводит к чрезмерной нагрузке на CPU и оперативную память вследствие систематического тестирования всех возможных комбинаций значений гиперпараметров, рандомизированный поиск выбирает сочетания гиперпараметров наугад.

Этот метод может привести к значительному увеличению скорости вычислений, когда сеточный поиск тестирует более 30 сочетаний (для меньших пространств

поиска сеточный поиск по-прежнему конкурентоспособен). Достижимый прирост находится на том же уровне, который мы видели, когда переключились со случайных лесов на экстремально рандомизированные леса (от двух- до десятикратного увеличения в зависимости от спецификации оборудования, гиперпараметрического пространства и размера набора данных).

В параметре `n_iter` можно конкретизировать число значений гиперпараметров, которые оцениваются в случайном порядке:

```
In:
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

param_dist = {"max_depth": [1,3,7,8,12, None],
              "max_features": [8,9,10,11,16,22],
              "min_samples_split": [8,10,11,14,16,19],
              "min_samples_leaf": [1,2,3,4,5,6,7],
              "bootstrap": [True, False]}

# задать настройки поиска; используются только 25 произвольных
# оцениваний параметров, при этом время тренировки удается
# держать под контролем
rsearch = RandomizedSearchCV(clf,
                             param_distributions=param_dist,
                             n_iter=25)

rsearch.fit(X_train, y_train)
bestclf = rsearch.best_estimator_
#print(rsearch.cv_results_)
print(bestclf)

Out:
ExtraTreesClassifier(bootstrap=True, class_weight=None,
           criterion='gini', max_depth=7, max_features=22,
           max_leaf_nodes=None, min_impurity_split=1e-07,
           min_samples_leaf=2, min_samples_split=10,
           min_weight_fraction_leaf=0.0, n_estimators=500,
           n_jobs=1, oob_score=False, random_state=101,
           verbose=0, warm_start=False)
```

Выше мы видим список оптимальных для нашей модели значений параметров. Теперь можно использовать эту модель для выполнения предсказаний на тестовом наборе:

```
In:
y_pred = bestclf.predict(X_test)
confusionMatrix = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

print(confusionMatrix)
print(accuracy)

Out:
[[6719  339]
 [1242  700]]
0.824333333333
```

Нам удалось увеличить результативность модели в пределах допустимого диапазона продолжительностей тренировки и одновременно увеличить ее точность.

Экстремально рандомизированные деревья и большие наборы данных

До сих пор мы рассматривали решения с горизонтальным масштабированием, задействуя многоядерные CPU и рандомизацию благодаря специфическим характеристикам случайного леса и его более эффективной альтернативе – экстремально рандомизированному лесу. Однако если приходится иметь дело с большим набором данных, который не помещается в оперативной памяти либо требует слишком больших затрат CPU, то можно попробовать внеядерное решение. Лучшим решением для внеядерного подхода с ансамблями является решение, предоставляемое средой H2O, которое далее в настоящей главе будет рассмотрено подробно. Однако мы можем применить еще один практический прием, чтобы заставить алгоритм случайного леса или экстремальных деревьев выполняться гладко на крупномасштабном наборе данных. Второе наилучшее решение состоит в тренировке моделей на подвыборках данных с последующей сборкой в ансамбль результатов всех моделей, построенных на разных подвыборках данных (в конце концов, требуется только усреднить или сгруппировать результаты). В *главе 3* «*Быстрообучающиеся реализации машин SVM*» мы уже ввели понятие резервуарного отбора, в результате которого продуцируются выборки на потоках данных. В настоящей главе мы снова воспользуемся выборкой данных, обратившись к более широкому набору алгоритмов генерирования выборок. Сначала давайте установим по-настоящему удобный инструмент командой строки под названием `subsample`, разработанной Полом Батлером (Paul Butler) (https://github.com/paulgb/subsample) для извлечения выборок из большого набора данных в формате с разделением полей символом новой строки (как правило, CSV-подобного файла). Этот инструмент предоставляет быстродействующие и простые методы отбора, в частности резервуарного отбора.

Как явствует из *главы 3* «*Быстрообучающиеся реализации машин SVM*», резервуарный отбор – это алгоритм отбора наблюдений, который помогает извлекать выборки фиксированного размера из потока. Это концептуально простой алгоритм (в главе 3 была представлена его формула), требующий простого прохождения по данным для продуцирования выборки, которая будет сохранена в новом файле на диске. (Наш сценарий в главе 3 вместо этого сохранял ее в оперативной памяти.)

В следующем ниже примере мы воспользуемся инструментом для извлечения подвыборок `subsample` вместе с методом по ансамблированию натренированных на этих подвыборках моделей.

Собрав все вместе, в этом разделе мы должны выполнить следующие действия:

1) создать набор данных и разделить его на тестовые и тренировочные данные;
2) взять подвыборки тренировочных данных и сохранить их как отдельные файлы на жестком диске;
3) загрузить эти подвыборки и натренировать на них модели на основе экстремально рандомизированного леса;
4) агрегировать модели;
5) проверить результаты.

Сначала при помощи `pip` установим инструмент для извлечения подвыборок:

```
$ pip install subsample
```

В командной строке укажите рабочий каталог, содержащий файл, из которого вы хотите брать выборки:

```
$ cd /ваш-путь
```

В этом месте при помощи команды `cd` можно указать рабочий каталог, где необходимо хранить файл, который мы создадим на следующем шаге.

Мы выполним это следующим образом:

```
In:
import numpy as np
from sklearn.datasets import fetch_covtype
from sklearn.model_selection import train_test_split

#dataset = fetch_covtype(random_state=111, shuffle=True)
dataset = fetch_covtype()

X, y = dataset.data, dataset.target
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.3, random_state=0)
del(X,y)

covtrain = np.c_[X_train,y_train]
covtest  = np.c_[X_test,y_test]
np.savetxt('covtrain.csv', covtrain, delimiter=",")
np.savetxt('covtest.csv', covtest, delimiter=",")
```

Теперь, когда мы разбили набор данных на тестовые и тренировочные наборы, давайте извлечем подвыборки из тренировочного набора, чтобы получить порции данных, которые мы сможем закачать в оперативную память. Учитывая, что размер полного тренировочного набора данных составляет 30 000 примеров, мы извлечем три подвыборки меньшего размера, каждая по 10 000 элементов. Если ваш компьютер оборудован менее 2 Гб RAM, то вы, возможно, посчитаете более приемлемым разбить исходный тренировочный набор на меньшие по размеру файлы, правда, в результате этого полученные модельные результаты, скорее всего, будут отличаться от примера, основанного на трех подвыборках. Как правило, чем меньше примеров в подвыборках, тем больше смещение модели. Работая с подвыборками, мы в действительности пользуемся преимуществом работы на более управляемом объеме данных за счет увеличенного смещения оценок:

```
$ subsample --reservoir -n 10000 covtrain.csv > cov1.csv
$ subsample --reservoir -n 10000 covtrain.csv > cov2.csv
$ subsample --reservoir -n 10000 covtrain.csv > cov3.csv
```

Напомним, что команды операционной системы могут быть исполнены внутри блокнота Jupyter, при этом они предваряются восклицательным знаком:

```
!subsample --reservoir -n 10000 covtrain.csv > cov1.csv
...
```

Теперь можно найти эти наборы в папке, которую вы указали в командной строке.

Сейчас следует удостовериться, что в своем IDE или в блокноте Jupyter вы назначили тот же самый путь.

Загрузим выборки поочередно и натренируем на них модель, основанную на случайном лесе.

Прежде чем объединить их впоследствии для итогового предсказания, обращаем внимание, что мы придерживаемся пошагового подхода, чтобы вы могли проследить последовательность выполняемых шагов.

Для того чтобы эти примеры выполнились успешно, удостоверьтесь, что вы установили тот же самый путь в своем IDE или блокноте Jupyter:

```
import os
os.chdir('/ваш-путь')
```

В этом месте мы готовы начать обучаться на данных и можем поочередно загрузить выборки в оперативную память и натренировать на них ансамбль деревьев:

```
In:
import os
import numpy as np
import pandas as pd
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import cross_val_score, train_test_split
```

После вывода сообщения об отметке перекрестной проверки программа перейдет к тренировке модели поочердно на всех порциях данных. Поскольку мы обучаемся раздельно на разных частях данных поочередными порциями, мы инициализируем ансамблевого ученика (в данном случае классификатор `ExtraTreeClassifier`), используя параметр `warm_start=True` и метод `set_params`, который инкрементно добавляет деревья из предыдущих сеансов тренировки, так как метод подгонки `fit` вызывается многократно:

```
# загружается выборка 1
df = pd.read_csv('cov1.csv')
y  = df[df.columns[54]]
X  = df[df.columns[0:54]]

clf1 = ExtraTreesClassifier(n_estimators=100,
                            random_state=101,
                            warm_start=True)
clf1.fit(X,y)

scores = cross_val_score(clf1, X, y, cv=3,
                         scoring='accuracy', n_jobs=-1)

print('Перекрестно-проверочая точность: среднее = %0.3f стд.откл = %0.3f'
      % (np.mean(scores), np.std(scores)))
print(scores)
print('Количество деревьев в модели: %s'
      % len(clf1.estimators_))

# выборка 2
df = pd.read_csv('cov2.csv')
y  = df[df.columns[54]]
X  = df[df.columns[0:54]]

clf1.set_params(n_estimators=150, random_state=101,
                warm_start=True)
```

```
clf1.fit(X,y)
scores = cross_val_score(clf1, X, y, cv=3,
                         scoring='accuracy', n_jobs=-1)

print('После params ->')
print('Перекрестно-проверочая точность: среднее = %0.3f стд.откл = %0.3f'
      % (np.mean(scores), np.std(scores)))
print(scores)
print('Количество деревьев в модели: %s'
      % len(clf1.estimators_))

# выборка 3
df = pd.read_csv('cov3.csv')
y  = df[df.columns[54]]
X  = df[df.columns[0:54]]
clf1.set_params(n_estimators=200, random_state=101,
                warm_start=True)
clf1.fit(X,y)
scores = cross_val_score(clf1, X, y, cv=3,
                         scoring='accuracy', n_jobs=-1)

print('После params ->')
print('Перекрестно-проверочая точность: среднее = %0.3f стд.откл = %0.3f'
      % (np.mean(scores), np.std(scores)))
print(scores)
print('Количество деревьев в модели: %s'
      % len(clf1.estimators_))

# теперь выполним предсказание комбинированной моделью
# на тестовом наборе и проверим отметку точности

df = pd.read_csv('covtest.csv')
X  = df[df.columns[0:54]]
y  = df[df.columns[54]]
pred2  = clf1.predict(X)
scores = cross_val_score(clf1, X, y, cv=3,
                         scoring='accuracy', n_jobs=-1)
print("Итоговая отметка на тестовом наборе %r"
      % np.mean(scores))

Out:
Перекрестно-проверочая точность: среднее = 0.798 стд.откл = 0.008
[ 0.7862069   0.80408041  0.80276193]
Количество деревьев в модели: 100
После params ->
Перекрестно-проверочая точность: среднее = 0.806 стд.откл = 0.007
[ 0.80635492  0.79687969  0.81441441]
Количество деревьев в модели: 150
После params ->
Перекрестно-проверочая точность: среднее = 0.803 стд.откл = 0.003
[ 0.80569715  0.79867987  0.80396277]
Количество деревьев в модели: 200
Итоговая отметка на тестовом наборе 0.92185447181058278
```

Предупреждение: такая реализация выглядит не совсем по Python'овски, и тем не менее она довольно эффективная.

Мы улучшили отметку относительно итогового предсказания, перейдя от показателя точности на тестовом наборе порядка .8 к точности .922. Это вызвано тем, что мы имеем итоговую объединенную модель со всей древовидной информацией, состоящей из предыдущих трех моделей, основанных на случайном лесе. В полученных результатах работы сценария можно также найти число деревьев, которые были добавлены к исходной модели.

В дальнейшем вы можете попробовать такой подход на наборе данных еще большего объема с привлечением большего числа подвыборок, либо применить рандомизированный поиск к одной из подвыборок для улучшения настроек параметров.

Алгоритм CART и бустинг

Мы начали эту главу с бэггинга, т. е. агрегации бустрапированных выборок; теперь мы завершим наш обзор другим ансамблевым методом – бустингом, т. е. разгоном базового алгоритма. Точно так же, как бэггинг, бустинг может использоваться и для регрессии, и для классификации, и в последнее время затмевает алгоритм случайного леса более высокой точностью.

Как процесс оптимизации, бустинг основывается на принципе стохастического градиентного спуска, который мы видели в других методах, а именно оптимизации модели путем минимизации ошибки в соответствии с градиентами. Самые известные на сегодня методы бустинга представлены алгоритмом **AdaBoost** и градиентным бустингом (GBM и недавно XGBoost). Алгоритм AdaBoost сводится к минимизации ошибки тех случаев, где предсказание немного неправильное, и в силу этого случаи, которые труднее классифицировать, привлекают больше внимания. В последнее время алгоритм AdaBoost не в почете, поскольку другие методы бустинга оказывались, как правило, точнее.

В этой главе мы охватим два самых эффективных алгоритма бустинга, доступных к сегодняшнему дню пользователям Python: **машину градиентного бустинга** (gradient boosting machine, GBM), чью реализацию можно найти в библиотеке Scikit-learn, и **экстремальный градиентный бустинг** (extreme gradient boosting, XGBoost). Поскольку алгоритм GBM по своей природе последовательный, его трудно параллелизировать и тем самым тяжелее масштабировать, чем алгоритм случайного леса, но некоторые приемы с этой задачей справляются. Далее будут рассмотрены некоторые рекомендации и приемы ускорения алгоритма вместе с прекрасным решением вне оперативной памяти для платформы H2O.

Машины градиентного бустинга

Как мы видели в предыдущих разделах, случайные леса и экстремальные деревья являются эффективными алгоритмами – оба этих алгоритма работают вполне хорошо с минимальным усилием. Хотя алгоритм машины градиентного бустинга GBM признается как более точный метод, он не так прост в использовании, и для достижения наилучших результатов всегда существует необходимость в доводке большого количества свойственных ему гиперпараметров. Случайный лес, с другой стороны, показывает вполне хорошее качество работы с учетом всего нескольких параметров (главным образом глубины дерева и числа деревьев). Еще стоит обратить внимание на переподгонку. Случайные леса менее чувствительны

к переподгонке, чем алгоритм GBM. Поэтому, используя его, мы также должны подумать о стратегиях регуляризации. Но самое главное, случайные леса намного проще настроить на выполнение параллельных операций, тогда как алгоритм GBM является последовательным и тем самым вычисляется медленнее.

В настоящей главе мы применим алгоритм GBM из библиотеки Scikit-learn, затем обратимся к следующему поколению алгоритмов бустинга деревьев под названием XGBoost и реализуем бустинг в более крупном масштабе в среде H2O.

Алгоритм GBM, который мы будем использовать в Scikit-learn и H2O, основан на двух важных понятиях: **аддитивном расширении** и градиентной оптимизации алгоритмом **наискорейшего спуска**. Общая идея первого состоит в генерировании последовательности относительно простых деревьев (слабых учеников), где каждое следующее дерево добавляется вдоль градиента. Пусть имеются деревья M, которые агрегируют итоговые предсказания в ансамбле. Дерево в каждой итерации f_k теперь является частью намного более широкого пространства всех возможных деревьев в модели (ø) (в библиотеке Scikit-learn этот параметр больше известен как `n_estimators`):

$$y = \sum_{m=1}^{M} f_k(x_i),\ f_k \in \phi.$$

Новые деревья будут добавляться к предыдущим деревьям по принципу аддитивного расширения поэтапно:

$\hat{y}_i^{(0)} = 0;$

$\hat{y}_i^{(1)} = f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i);$ – это первое дерево

$\hat{y}_i^{(2)} = f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i);$ – второе дерево добавляется к предыдущему

и т. д., пока не будет достигнут критерий останова.

Предсказание ансамбля градиентного бустинга состоит из суммы предсказаний всех предыдущих деревьев и недавно добавленного дерева ($\hat{y}_i^{(t-1)}$) + $f_1(x_i)$, что в более формальном плане приводит к следующему:

$$\hat{y}_i^{(t)} = \sum_{m=1}^{M} f_k(x_i) = \hat{y}_i^{(t-1)} + f_i(x_i).$$

Вторая важная и при этом довольно хитроумная часть алгоритма GBM – это градиентная оптимизация **наискорейшим спуском**. Иными словами, в аддитивную модель мы добавляем все более мощные деревья. Это достигается путем применения к новым деревьям градиентной оптимизации. Каким образом выполняются обновления градиента с деревьями, раз нет никаких параметров, аналогичных тем, которые мы видели во время работы с традиционными обучающимися алгоритмами? Прежде всего мы параметризуем деревья; мы делаем это путем рекурсивного обновления значений расщепления узлов вдоль градиента, где узлы представлены вектором. Вследствие этого направление наискорейшего спуска является отрицательным градиентом функции потерь, и расщепления узлов будут обновляться и обучаться, приводя к:

- λ: параметру сжатия (в данном контексте также именуемом темпом обучения), который заставит ансамбль обучаться медленно с добавлением большего количества деревьев;

- γ_{mi}: параметру обновления градиента, также именуемому длиной шага.

Прогнозная отметка для каждого листа тем самым является итоговой отметкой для нового дерева, которая, в свою очередь, является результатом простого суммирования по каждому листу:

$$\hat{y}_i^{(t)} = \sum_{m=1}^{M} f_k(x_i) = \hat{y}_i^{(t-1)} + \lambda\gamma_{mi} f_i(x_i).$$

Таким образом, если резюмировать, алгоритм GBM работает путем инкрементного добавления более точных деревьев, усвоенных вдоль градиента.

Теперь, когда мы разбираемся в базовых понятиях, выполним пример с алгоритмом GBM и посмотрим на самые важные параметры. Для алгоритма GBM эти параметры имеют дополнительную важность, потому что, когда мы устанавливаем число деревьев слишком высоко, мы обречены на экспоненциальный рост нагрузки на вычислительные ресурсы, поэтому с этими параметрами следует обращаться осторожно. Большинство параметров в приложении GBM библиотеки Scikit-learn совпадает с алгоритмом случайного леса, который мы рассмотрели в предыдущем абзаце. Мы должны учитывать три параметра, которые требуют особого внимания.

Максимальная глубина (max_depth)

В отличие от алгоритма случайного леса, который работает лучше, когда деревья создают свои древовидные структуры в их максимальной протяженности (тем самым конструируя и собирая в ансамбль предикторы с высокой дисперсией), алгоритм GBM демонстрирует тенденцию работать лучше с более мелкими деревьями (тем самым привлекая предикторы с более высоким смещением, т. е. слабых учеников). Работа с более мелкими деревьями решений или только с пнями (деревьями решений только с одним-единственным ответвлением) может сократить продолжительность тренировки, получая скорость выполнения взамен на большее смещение (потому что более мелкое дерево едва способно перехватывать более сложные связи в данных).

Темп обучения (learning_rate)

Этот параметр, также именуемый сжатием λ (shrinkage), связан с оптимизацией методом градиентного спуска и с тем, как каждое дерево будет участвовать в ансамбле. Меньшие значения этого параметра могут улучшить оптимизацию в процессе тренировки, хотя для этого потребуется схождение большего количества оценщиков и тем самым больше времени на вычисления. Поскольку он влияет на вес каждого дерева в ансамбле, его меньшие значения подразумевают, что каждое дерево вносит в процесс оптимизации небольшую часть, и потребуется больше деревьев, прежде чем будет достигнуто хорошее решение. Следовательно, во время оптимизации этого параметра в целях результативности необходимо избегать слишком больших значений, которые могут привести к субоптимальным моделям; также необходимо избегать использования слишком низких значений, потому что это будет серьезно затрагивать продолжительность вычисления (для схождения ансамбля к решению потребуется больше деревьев). На нашем опыте хорошей начальной точкой является использование темпа обучения в интервале < 0.1 и $> .001$.

Подвыборка

Вспомним принципы бэггинга и вставки, где мы извлекаем случайные выборки и создаем деревья на их основе. Если в алгоритме GBM применить подвыборку, то мы рандомизируем создание деревьев и избегаем переподгонки, уменьшаем нагрузку на память и даже иногда увеличиваем точность. Эту процедуру можно также применить к алгоритму GBM, делая его более стохастическим и тем самым задействуя преимущества бэггинга. Можно рандомизировать создание деревьев в алгоритме GBM путем установки параметра подвыборки в .5.

Ускоренный GBM с теплым стартом (warm_start)

Этот параметр позволяет сохранять новую древовидную информацию, после того как каждая итерация добавляется к предыдущей без генерирования новых деревьев. Таким образом можно сэкономить память и значительно ускорить время вычисления.

Используя имеющийся в библиотеке Scikit-learn метод GBM, можно предпринять два действия, чтобы увеличить память и эффективность CPU:

- теплый старт для (полу)инкрементного обучения;
- использование параллельной обработки во время перекрестной проверки.

Проанализируем пример классификации на основе алгоритма GBM с использованием набора данных Spam из библиотеки машинного обучения UCI. Сначала загрузим данные, предварительно их обработаем и посмотрим на важность каждого признака:

```
In:
import pandas
import urllib2
from sklearn import ensemble

columnNames1_url = 'https://archive.ics.uci.edu/ml/machine-learningdatabases/spambase/
spambase.names'
columnNames1 = [
    line.strip().split(':')[0]
    for line in urllib2.urlopen(columnNames1_url).readlines()[33:]]

columnNames1
n = 0
for i in columnNames1:
    columnNames1[n] = i.replace('word_freq_','')
    n += 1
print columnNames1

spamdata = pandas.read_csv(
    'https://archive.ics.uci.edu/ml/machine-learning-databases/
spambase/spambase.data',
    header=None,
    names=(columnNames1 + ['spam'])
)

X = spamdata.values[:,:57]
y = spamdata['spam']

spamdata.head()
```

Выше будут выведены список с именами столбцов и первые 5 записей в таблице. Далее рассчитаем отметки точности:

```
import numpy as np
from sklearn import cross_validation
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict, train_test_split
from sklearn.metrics import classification_report, recall_score
from sklearn.metrics import f1_score, confusion_matrix, accuracy_score
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.3, random_state=22)

clf = ensemble.GradientBoostingClassifier(n_estimators=300,
                                          random_state=222,
                                          max_depth=16,
                                          learning_rate=.1,
                                          subsample=.5)
scores  = clf.fit(X_train,y_train)
scores2 = cross_val_score(clf, X_train, y_train,
                          cv=3, scoring='accuracy', n_jobs=-1)
print('Средняя точность %s' % scores2.mean())

y_pred = cross_val_predict(clf, X_test, y_test, cv=10)

print('Точность на контрольных данных %s'
      % accuracy_score(y_test, y_pred))

Out:
Средняя точность 0.943789291121
Точность на контрольных данных 0.931209268646
```

В заключение покажем матрицу ошибок и признаки, упорядоченные по величине важности.

```
def featureImp_order(clf, X, k=5):
    return X[:,clf.feature_importances_.argsort()[::-1][:k]]
#print(featureImp_order(clf, X, 2))
#print(clf.feature_importances_)

print(classification_report(y_test, y_pred))
confusionMatrix = confusion_matrix(y_test, y_pred)
print(confusionMatrix)

# упорядочим признаки по величине важности
print('\nВажности признаков:')
print(sorted(zip(map(lambda x: round(x, 4),
                     clf.feature_importances_), columnNames1),
             reverse=True))

Out:
             precision    recall  f1-score   support

          0       0.93      0.95      0.94       835
          1       0.93      0.90      0.91       546
```

```
avg / total       0.93      0.93      0.93      1381

[[797  38]
 [ 57 489]]

Важности признаков:
[(0.2193, 'char_freq_;'),
(0.1079, 'report'),
(0.0755, 'capital_run_length_average'),
(0.0498, 'capital_run_length_total'),
(0.0433, 'you'),
(0.0354, 'char_freq_!'),
(0.0325, '000'),
(0.0312, 'capital_run_length_longest'),
(0.0254, 'will'),
(0.0243, 'business'),
(0.0236, 'your'),
(0.0226, 'char_freq_$'),
(0.02, 'char_freq_('),
(0.0161, 'mail'),
(0.0159, 'internet'),
...
(0.0, '415')]
```

Мы видим, что при классификации спама символ «;» является самым дискриминирующим.

Важность признака показывает, насколько расщепление каждого признака уменьшает относительную неоднородность по всем расщеплениям в дереве.

Ускорение алгоритма GBM при помощи параметра warm_start

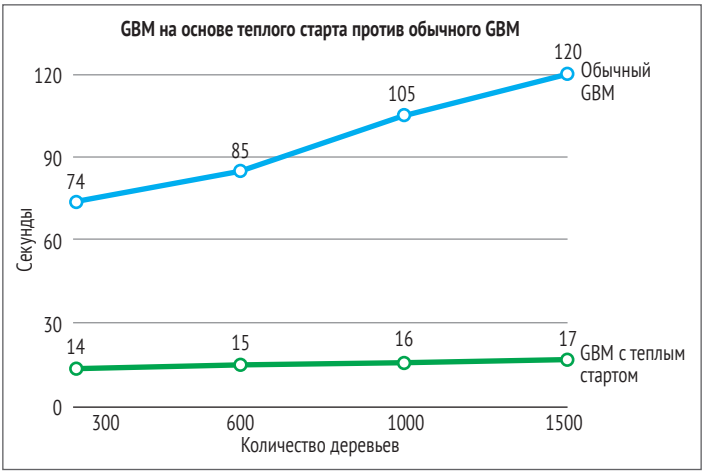
К сожалению, в библиотеке Scikit-learn параллельная обработка алгоритма GBM отсутствует. Параллелизировать можно только перекрестную проверку и сеточный поиск. Тогда что можно сделать для его ускорения? Мы видели, что машина градиентного бустинга GBM работает по принципу аддитивного расширения, где деревья добавляются инкрементно. Мы можем использовать эту идею в Scikit-learn посредством параметра `warm_start`. Это можно легко смоделировать благодаря функционалу GBM библиотеки Scikit-learn, если создавать древовидные модели инкрементно при помощи цикла `for`. Поэтому давайте выполним это с тем же набором данных и обследуем вычислительное преимущество, которое он обеспечивает:

```
gbc = GradientBoostingClassifier(warm_start=True,
                                 learning_rate=.05,
                                 max_depth=20,
                                 random_state=0)

for n_estimators in range(1, 1500, 100):
    gbc.set_params(n_estimators=n_estimators)
    gbc.fit(X_train, y_train)
y_pred = gbc.predict(X_test)
print(classification_report(y_test, y_pred))
print(gbc.set_params)
```

```
Out:
precision    recall  f1-score   support

          0       0.93      0.95      0.94       835
          1       0.92      0.89      0.91       546

avg / total       0.93      0.93      0.93      1381

<bound method GradientBoostingClassifier.set_params of GradientBoostingClassifier(criterion='f
riedman_mse', init=None,
              learning_rate=0.05, loss='deviance', max_depth=20,
              max_features=None, max_leaf_nodes=None,
              min_impurity_split=1e-07, min_samples_leaf=1,
              min_samples_split=2, min_weight_fraction_leaf=0.0,
              n_estimators=1401, presort='auto', random_state=0,
              subsample=1.0, verbose=0, warm_start=True)>
```

Рекомендуем обратить особое внимание на результаты доводки дерева (`n_estimators=1401`). Вы видите, что мы использовали модель размером 1401 дерево. Этот небольшой прием помог намного уменьшить продолжительность тренировки (думаем, наполовину или меньше), если сравнивать его с подобной моделью на основе алгоритма GBM, которую мы бы натренировали на 1401 дереве сразу. Отметим, что этот метод можно также использовать для алгоритма случайного леса и экстремального случайного леса. Тем не менее мы считаем его в особенности полезным для алгоритма GBM.

Взглянем на рисунок, который показывает время тренировки обычного алгоритма GBM и нашего на основе метода `warm_start`. Мы видим значительное ускорение вычислений с отметкой точности относительно на том же уровне:



Тренировка и хранение моделей GBM

Вы когда-нибудь задумывались о тренировке модели на трех компьютерах одновременно? Или о тренировке модели GBM на экземпляре Amazon EC2? Вполне может случиться так, что вам понадобится натренировать модель и сохранить ее для использования в дальнейшем. Когда приходится ждать в течение двух дней

до полного завершения раунда тренировки, мы вовсе не хотим пройти через этот процесс снова. В случае если вы натренировали модель в облаке на экземпляре Amazon EC2, вы можете сохранить эту модель и использовать ее повторно позже на другом компьютере при помощи функционала `joblib` библиотеки Scikit-learn. Поэтому давайте проанализируем этот процесс, раз уж библиотека Scikit-learn предоставила нам удобный инструмент по работе с ним.

Выполним импорт нужных библиотек и укажем каталоги, где расположены наши файлы:

```
import errno
import os

path = '/data/clfs'    # укажите здесь свой путь

clfm = os.makedirs(path)
os.chdir(path)
```

Теперь экспортируем модель в указанное расположение на жестком диске:

```
from sklearn.externals import joblib
joblib.dump(gbc, 'clf_gbc.pkl')
```

Сейчас можно загрузить модель и снова ее использовать для других целей:

```
model_clone = joblib.load('clf_gbc.pkl')
zpred = model_clone.predict(X_test)
print(zpred)
```

Алгоритм XGBoost

Мы только что обсудили, что параллельная обработка при использовании алгоритма машины градиентного бустинга GBM в Scikit-learn отсутствует без вариантов, и именно здесь на первый план выходит алгоритм XGBoost. Расширяя алгоритм GBM, алгоритм экстремального градиентного бустинга XGBoost добавляет больше масштабируемых методов, которые задействуют многопоточность на одиночной машине и параллельную обработку на кластерах из многочисленных серверов (используя сегментирование). Самое важное усовершенствование, вносимое алгоритмом XGBoost, по сравнению с алгоритмом GBM, состоит в возможности первого управлять разреженными данными. Алгоритм XGBoost принимает разреженные данные автоматически, не храня нулевых значений в памяти. Второе преимущество XGBoost заключается в том, каким образом вычисляются значения наилучшего расщепления узлов при ветвлении дерева, при этом используется метод, который называется квантильной схемой (quantile sketch). Этот метод преобразует данные алгоритмом взвешивания, в результате которого потенциальные расщепления сортируются на основе определенного уровня точности. Для получения дополнительной информации читайте статью, которую можете скачать по следующей прямой ссылке: http://arxiv.org/pdf/1603.02754v3.pdf.

Алгоритм XGBoost расшифровывается как «экстремальный градиентный разгон» (extreme gradient boosting). Данный алгоритм градиентного бустинга с открытым исходным кодом получил большую популярность на конкурсах в области науки о данных, в частности Kaggle (https://www.kaggle.com/) и KDD-cup в 2015 г. (Его

исходный код доступен на GitHub по адресу https://github.com/dmlc/XGBoost, как уже упоминалось в *главе 1* «*Первые шаги к масштабируемости*».) По сообщениям авторов работы (Tianqui Chen, Tong He и Carlos Guestrin), посвященной этому методу бустинга, среди 29 соревнований, проводившихся в Kaggle в течение 2015 г., в 17 победивших решениях алгоритм XGBoost использовался автономно либо в качестве составной части какого-нибудь ансамбля из нескольких моделей. В своей статье *XGBoost: A Scalable Tree Boosting System* («*XGBoost: Масштабируемая система бустинга деревьев*» (которую можно скачать по адресу http://learningsys.org/papers/LearningSys_2015_paper_32.pdf)) авторы сообщают, что в недавнем соревновании KDD-cup 2015 г. алгоритм XGBoost использовался каждой командой, которая по итогам соревнований оказалась в десятке лучших. Помимо успешных показателей результативности (как точности, так и вычислительной эффективности), нашей основной задачей в этой книге является масштабируемость, и экстремальный градиентный бустинг XGBoost является по-настоящему масштабируемым решением с различных точек зрения. XGBoost – это новое поколение алгоритмов градиентного бустинга GBM с серьезной доводкой исходного алгоритма бустинга деревьев GBM. Алгоритм XGBoost обеспечивает параллельную обработку; предлагаемая алгоритмом масштабируемость реализуется благодаря доработанным авторами нескольким параметрическим настройкам и добавлениям:

- алгоритм принимает разреженные данные, в которых могут задействоваться разреженные матрицы, экономя оперативную память (отсутствует потребность в плотных матрицах) и продолжительность вычисления (нулевые значения обрабатываются особым образом);
- обучение приближенному дереву (взвешенный метод квантильной схемы), которое показывает аналогичные результаты, но за гораздо меньшее время, чем классический исчерпывающий просмотр возможных точек ветвления;
- параллельные вычисления на одиночной машине (используя многопоточность в фазе поиска лучшего расщепления) и аналогичным образом распределенные вычисления на нескольких машинах;
- внеядерные вычисления на одиночной машине с привлечением решения для хранения данных под названием «постолбцовый блок» (column block), которое располагает данные на диске столбцами, тем самым экономя время – данные с диска поступают в том виде, в котором их ожидает алгоритм оптимизации (который оперирует векторами-столбцами).

С практической точки зрения экстремальный градиентный бустинг XGBoost демонстрирует главным образом те же параметры, что и алгоритм GBM. Алгоритм XGBoost также довольно хорошо обрабатывает пропущенные данные. Другие древовидные ансамбли, основанные на стандартных деревьях решений, требуют сначала импутировать[1] пропущенные данные, используя внешкальное значение (в частности, большое отрицательное число), чтобы выработать надлежащее ветвление дерева в случае пропущенных значений. В отличие от них, алгоритм XGBoost сначала выполняет подгонку всех непропущенных значений и после создания ветвления для переменной затем решает, какая ветвь лучше всего подходит для пропущенных значений с целью уменьшения ошибки прогнозирования.

[1] Импутация (imputation) – процесс замещения пропущенных, некорректных или несостоятельных значений другими значениями. – *Прим. перев.*

Такой подход приводит к более компактным деревьям, а эффективная стратегия импутации – к большей прогнозирующей способности.

Самые важные параметры алгоритма XGBoost следующие:

- `eta` (по умолчанию = 0.3): эквивалент темпа обучения в алгоритме GBM библиотеки Scikit-learn;
- `min_child_weight` (по умолчанию = 1): более высокие значения предотвращают переподгонку и вычислительную сложность деревьев;
- `max_depth` (по умолчанию = 6): число взаимодействий в деревьях;
- `subsample` (по умолчанию = 1): доля выборок из тренировочных данных, которые берутся в каждой итерации;
- `colsample_bytree` (по умолчанию = 1): доля признаков в каждой итерации;
- `lambda` (по умолчанию = 1): регуляризация L2 (булева переменная);
- `seed` (по умолчанию = 0): эквивалент параметра random_state в библиотеке Scikit-learn, обеспечивающий воспроизводимость процессов обучения по всем тестам и на разных машинах.

Теперь, когда мы знаем самые важные параметры алгоритма XGBoost, выполним пример его применения на том же самом наборе данных, который мы использовали для алгоритма GBM, с теми же настройками параметров (насколько это возможно). Применение алгоритма экстремального градиентного бустинга XGBoost немного менее прямолинейно, чем алгоритма GBM в библиотеке Scikit-learn. И поэтому мы предоставим несколько элементарных примеров, которые можно использовать в качестве отправной точки для более сложных моделей. Прежде чем углубиться в приложения на основе экстремального градиентного бустинга XGBoost, сравним этот метод с методом GBM в модуле `sklearn` на наборе данных о спаме; мы уже загрузили эти данные в память:

```
import numpy as np
import xgboost as xgb
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import classification_report
from sklearn import model_selection
from sklearn.model_selection import train_test_split

clf = xgb.XGBClassifier(n_estimators=100, max_depth=8,
                        learning_rate=.1, subsample=.5)

clf1 = GradientBoostingClassifier(n_estimators=100, max_depth=8,
                                  learning_rate=.1, subsample=.5)

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.3, random_state=22)
xgm  = clf.fit(X_train, y_train)
gbmf = clf1.fit(X_train, y_train)

y_pred  = xgm.predict(X_test)
y_pred2 = gbmf.predict(X_test)

print('Результаты алгоритма XGBoost:')
print(classification_report(y_test, y_pred))
print('Результаты алгоритма GBM:')
print(classification_report(y_test, y_pred2))

Out:
1 loop, best of 3: 1.71 s per loop
1 loop, best of 3: 2.91 s per loop
```

Результаты алгоритма XGBoost:

```
             precision    recall  f1-score   support

          0       0.95      0.97      0.96       835
          1       0.95      0.93      0.94       546

avg / total       0.95      0.95      0.95      1381
```

Результаты алгоритма GBM:

```
             precision    recall  f1-score   support

          0       0.95      0.97      0.96       835
          1       0.95      0.92      0.93       546

avg / total       0.95      0.95      0.95      1381
```

Ясно видно, что алгоритм экстремального градиентного бустинга XGBoost выполняется заметно быстрее, чем алгоритм GBM (1.71 против 2.91 сек.), несмотря на то что параллелизация для XGBoost даже не использовалась. Позднее мы даже сможем добиться большего ускорения, когда воспользуемся параллельной обработкой и внеядерными методами для XGBoost во время использования потоковой передачи вне основной памяти. В некоторых случаях модель на основе алгоритма экстремального градиентного бустинга XGBoost приводит к более высокой точности, чем модель на основе алгоритма GBM, и (почти) никогда наоборот.

Регрессия на основе XGBoost

Методы бустинга часто используются для классификации, но также могут быть очень мощными для задач регрессии. Поскольку регрессия нередко упускается из виду, давайте выполним пример регрессии и проанализируем ключевые вопросы. Мы выполним подгонку модели бустинга на наборе данных о жилой недвижимости в шт. Калифорния с использованием сеточного поиска. Набор данных о жилой недвижимости шт. Калифорния был недавно добавлен в Scikit-learn и потому сэкономит нам несколько шагов предобработки:

```
import os
import numpy as np
import pandas as pd
import scipy.sparse
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
import xgboost as xgb
from xgboost.sklearn import XGBClassifier

pd = fetch_california_housing()

# поскольку переменная y сильно скошена,
# мы применяем логарифмическое преобразование
y = np.log(pd.target)
X_train, X_test, y_train, y_test = \
    train_test_split(pd.data, y, test_size=0.15, random_state=111)
names = pd.feature_names

print(names)
```

```
clf = xgb.XGBRegressor(gamma=0, objective="reg:linear", n_jobs=-1)

clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)

print('Отметка до сеточного поиска %r'
      % mean_squared_error(y_test, y_pred))

params = {
    'max_depth' :[4,6,8],
    'n_estimators' :[1000],
    'min_child_weight' :range(1,3),
    'learning_rate' :[.1,.01,.001],
    'colsample_bytree' :[.8,.9,1],
    'gamma' :[0,1]}

# параметром nthread мы задаем параллелизацию алгоритма XGBoost
cvx = xgb.XGBRegressor(objective="reg:linear", n_jobs=-1)
clf = GridSearchCV(estimator=cvx, param_grid=params,
                   n_jobs=-1, scoring='mean_absolute_error',
                   verbose=True)

clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)

print(clf.best_params_)
print('Отметка после сеточного поиска %r'
      % mean_squared_error(y_test, y_pred))

# В зависимости от используемого оборудования
# ваш результат может выглядеть немного иначе

Out:
['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup', 'Latitude',
'Longitude']
Отметка до сеточного поиска 0.072204301145003813
Fitting 3 folds for each of 108 candidates, totalling 324 fits
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:  6.1min (1.9min)
[Parallel(n_jobs=-1)]: Done 192 tasks      | elapsed: 29.4min (11.3min)
[Parallel(n_jobs=-1)]: Done 324 out of 324 | elapsed: 53.9min (22.3min) finished
{'colsample_bytree': 0.9, 'learning_rate': 0.1, 'min_child_weight': 1, 'n_estimators': 1000,
'max_depth': 8, 'gamma': 0}
Отметка после сеточного поиска 0.050217877735748678
```

При помощи сеточного поиска нам удалось порядком улучшить нашу отметку; мы видим оптимальные параметры сеточного поиска. Они аналогичны обычным методам бустинга в модуле sklearn. Однако метод экстремального градиентного бустинга XGBoost по умолчанию параллелизирует алгоритм по всем доступным ядрам. Можно улучшить производительность модели путем увеличения параметра `n_estimators` порядка до 2500 или 3000. Однако мы посчитали, что продолжительность обучения будет излишне долгой для читателей с менее мощными компьютерами.

Алгоритм XGBoost и показатель важности признаков

Алгоритм XGBoost имеет очень практичный встроенный функционал для построения графика важности признаков. Прежде всего имеется удобный инструмент для

отбора признаков относительно текущей модели. Как вы, вероятно, знаете, показатель важности основывается на относительном влиянии каждого признака при создании дерева. Он предоставляет практические методы для отбора признаков и помогает глубже понять природу прогнозной модели. Поэтому давайте посмотрим, каким образом можно вывести на график важность с алгоритмом XGBoost:

```
import numpy as np
import os
from matplotlib import pylab as plt
# %matplotlib inline <- это работает только в блокноте Jupyter

# наш набор наилучших параметров
# {'colsample_bytree': 1, 'learning_rate': 0.1, 'min_child_weight': 1,
#  'n_estimators': 500, #'max_depth': 8, 'gamma': 0}
params = {'objective': "reg:linear",
          'eval_metric': 'rmse',
          'eta': 0.1,
          'max_depth':8,
          'min_samples_leaf':4,
          'subsample':.5,
          'gamma':0
}

dm = xgb.DMatrix(X_train, label=y_train, feature_names=names)
regbgb = xgb.train(params, dm, num_boost_round=100)
np.random.seed(1)
regbgb.get_fscore()

regbgb.feature_names
regbgb.get_fscore()
xgb.plot_importance(regbgb,color='magenta',title='Калифорнийская недвижимость|важность признаков')
```

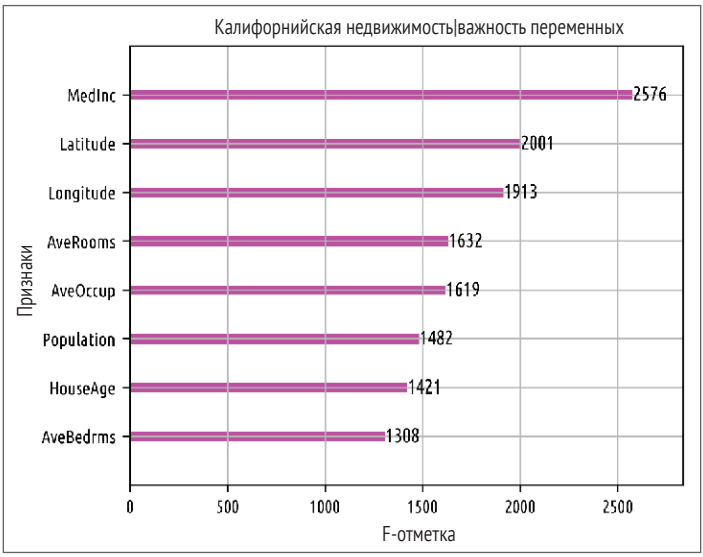


График построен средствами библиотеки xgboost

Метрикой важности признаков следует пользоваться с некоторой осторожностью (то же касается алгоритмов GBM и случайного леса). Показатель важности признаков основан чисто на структуре деревьев, которая опирается на конкретную модель, натренированную параметрами этой модели. Это означает, что если изменить параметры модели, то изменится и метрика важности, и часть ранжирования. Поэтому следует отметить, что любая метрика важности не должна браться в качестве универсального заключения о признаке, обобщаемом на все модели.

Потоковая передача больших наборов данных посредством XGBoost

С точки зрения компромисса между точностью и производительностью экстремальный градиентный бустинг XGBoost является наилучшим настольным решением. На примере со случайным лесом мы увидели, что для предотвращения перегрузки оперативной памяти приходилось прибегать к извлечению подвыборок.

Часто упускается из виду способность алгоритма XGBoost выступать как метод потоковой передачи данных через память. Этот метод выполняет поэтапное преобразование проходящих через оперативную память данных, впоследствии передавая их в соответствующем виде в алгоритм XGBoost для тренировки модели. Данный метод является необходимым предварительным этапом для тренировки моделей на больших наборах данных, которые невозможно разместить в оперативной памяти. Потоковая передача методом XGBoost работает только с файлами LIBSVM, т. е. сначала нужно преобразовать набор данных в формат LIBSVM и затем импортировать его в кэш-память, выделенную для алгоритма XGBoost. Еще стоит отметить, что мы используем разные методы инстанцирования моделей XGBoost. Scikit-learn-подобный интерфейс для алгоритма XGBoost работает только на обычных объектах NumPy. Поэтому давайте посмотрим, как это работает.

Прежде чем перейти к предобработке и тренировке, сначала нужно загрузить набор данных в формате LIBSVM и разбить его на тренировочный и тестовый наборы. Тонкая настройка параметров при помощи сеточного поиска, к сожалению, с этим методом XGBoost не возможна. Если требуется выполнить доводку параметров, то необходимо преобразовать файл LIBSVM в объект Numpy, который выгрузит данные из кэш-памяти в оперативную память. Такая операция, к сожалению, не масштабируема, поэтому, если вы хотите выполнять доводку параметров на больших наборах данных, рекомендуется использовать представленные ранее инструменты резервуарного отбора и выполнить настройку на подвыборках:

```
import urllib
import numpy as np
from sklearn.datasets import dump_svmlight_file, load_svmlight_file
from sklearn.metrics import classification_report
import xgboost as xgb

path = "http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass/"
trainfile = urllib.URLopener()
trainfile.retrieve(path + "poker.bz2", "pokertrain.bz2")
```

```
X, y = load_svmlight_file('pokertrain.bz2')
dump_svmlight_file(X, y, 'pokertrain', zero_based=True,
                   query_id=None, multilabel=False)
testfile = urllib.URLopener()
testfile.retrieve(path + "poker.t.bz2", "pokertest.bz2")
X, y = load_svmlight_file('pokertest.bz2')
dump_svmlight_file(X, y, 'pokertest', zero_based=True,
                   query_id=None, multilabel=False)
del(X, y)

dtrain = xgb.DMatrix('pokertrain#dtrain.cache')
dtest  = xgb.DMatrix('pokertest#dtestin.cache')

param = {'max_depth':8, 'objective':'multi:softmax',
         'nthread':2, 'num_class':10, 'verbose':True}

num_round = 100
watchlist = [(dtest,'eval'), (dtrain,'train')]
bst = xgb.train(param, dtrain, num_round, watchlist)

print(bst)

Out:
[89]   eval-merror:0.228659   train-merror:0.016913
[90]   eval-merror:0.228599   train-merror:0.015954
[91]   eval-merror:0.227671   train-merror:0.015354
[92]   eval-merror:0.227777   train-merror:0.014914
[93]   eval-merror:0.226247   train-merror:0.013355
[94]   eval-merror:0.225397   train-merror:0.012155
[95]   eval-merror:0.224070   train-merror:0.011875
[96]   eval-merror:0.222421   train-merror:0.010676
[97]   eval-merror:0.221881   train-merror:0.010116
[98]   eval-merror:0.221922   train-merror:0.009676
[99]   eval-merror:0.221733   train-merror:0.009316
```

За счет алгоритма XGBoost с обработкой в оперативной памяти можно получить по-настоящему огромное ускорение. Потребовалось бы намного больше времени на тренировку, если бы использовалась версия на основе внутренней памяти. В этом примере мы уже включили тестовый набор в качестве проверочного раунда в список наблюдения *watchlist*. Однако если мы хотим предсказывать значения на ранее не встречавшихся данных, то можно просто использовать ту же процедуру прогнозирования, что и с любой другой моделью в библиотеке Scikit-learn и алгоритме XGBoost:

```
bst.predict(dtest)

Out:
array([0., 0., 1., ..., 0., 0., 1.], dtype=float32)
```

Персистентность модели XGBoost

В предыдущей главе мы коснулись того, как сохранять модель на основе алгоритма GBM на диске, чтобы позже импортировать и использовать ее для прогнозирования. Алгоритм XGBoost обеспечивает ту же самую функциональность. Посмотрим, как можно сохранить и импортировать модель:

```
import pickle
bst.save_model('xgb.model')
```

Теперь можно импортировать сохраненную модель из каталога, который вы указали ранее:

```
imported_model = xgb.Booster(model_file='xgb.model')
```

Отлично, теперь эту модель можно использовать для прогнозирования:

```
imported_model.predict(dtest)
```

```
Out:
array([ 9., 9., 9., ..., 1., 1., 1.], dtype=float32)
```

Внеядерный алгоритм CART в среде H2O

До настоящего момента мы имели дело только с настольными решениями для моделей CART. В *главе 4* «*Нейронные сети и глубокое обучение*» мы представили платформу H2O для глубокого обучения вне оперативной памяти, которая предоставила мощный масштабируемый метод. К счастью, платформа H2O также предоставляет древовидные ансамблевые методы, используя для этого свою мощную параллельную экосистему Hadoop. Поскольку в предыдущих разделах мы уже всесторонне рассматривали алгоритмы градиентного бустинга и случайного леса, то давайте сразу же приступим к делу. Для этого упражнения мы воспользуемся набором данных Spam, который уже применялся ранее.

Случайный лес и сеточный поиск в H2O

Давайте реализуем случайный лес с гиперпараметрической оптимизацией на основе сеточного поиска. В этом разделе мы сначала загрузим набор с данными о спаме из URL-ресурса:

```
In:
import os
import xlrd
import urllib
import tempfile
import numpy as np
import pandas as pd
import h2o
from h2o.estimators.gbm import H2OGradientBoostingEstimator
from h2o.estimators.random_forest import H2ORandomForestEstimator
from h2o.grid.grid_search import H2OGridSearch
```

```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/spambase/spambase.data'
temp_file = tempfile.NamedTemporaryFile().name
urllib.urlretrieve(url, temp_file)
```

```
Out:
('c:\\users\\labor\\appdata\\local\\temp\\tmpcrk5gi',
 <httplib.HTTPMessage instance at 0x000000000CD88AC8>)
```

Загрузив данные, теперь можно инициализировать сеанс H2O:

```
h2o.init(max_mem_size_GB = 2)
```

```
Out:
Checking whether there is an H2O instance running at http://localhost:54321. connected.
H2O cluster uptime:        13 mins 05 secs
H2O cluster version:       3.10.4.8
H2O cluster version age:   28 days, 2 hours and 30 minutes
H2O cluster name:          H2O_from_python_labor_b52cfa
H2O cluster total nodes:   1
H2O cluster free memory:   1.292 Gb
H2O cluster total cores:   4
H2O cluster allowed cores: 4
H2O cluster status:        locked, healthy
H2O connection url:        http://localhost:54321
H2O connection proxy:      None
H2O internal security:     False
Python version:            2.7.13 final
```

Ниже данные подвергаются предобработке, в результате которой они разбиваются на тренировочный, перекрестно-проверочный и тестовый наборы. Это выполняется при помощи функции H2O `split_frame`. Также отметим важный шаг, где происходит преобразование целевого вектора `C58` в факторную переменную:

```
In:
spamdata = h2o.import_file(path=temp_file)
spamdata['C58'] = spamdata['C58'].asfactor()

train, valid, test = spamdata.split_frame([0.6,.2], seed=1234)

spam_X = spamdata.col_names[:-1]
spam_Y = spamdata.col_names[-1]
```

В следующей части мы зададим параметры, которые будем оптимизировать при помощи сеточного поиска. В первую очередь мы устанавливаем число деревьев в модели в единственное значение 300. Параметры, итеративный обход которых выполняется сеточным поиском, приведены ниже:

- `max_depth`: максимальная глубина дерева;
- `balance_classes`: каждая итерация использует сбалансированные классы для целевого исхода;
- `sample_rate`: доля строк, которые отбираются для каждой итерации.

Теперь передадим эти параметры в список Python, который будет использоваться в нашей модели H2O с сеточным поиском:

```
hyper_parameters={'ntrees':[300],
                  'max_depth':[3,6,10,12,50],
                  'balance_classes':['True','False'],
                  'sample_rate':[.5,.6,.8,.9]}

grid_search = H2OGridSearch(H2ORandomForestEstimator,
                            hyper_params=hyper_parameters)
grid_search.train(x=spam_X, y=spam_Y, training_frame=train)

print('Оптимальное решение для гиперпараметрического поиска')
grid_search.show()
```

```
Out:
Оптимальное решение для гиперпараметрического поиска
```

	model_ids	balance_classes max_depth ntrees sample_rate	logloss
0	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_38	true, 50, 300, 0.9	0.11861763801092395
1	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_28	true, 50, 300, 0.8	0.13727604944883123
2	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_8	true, 50, 300, 0.5	0.1453175326621316
3	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_18	true, 50, 300, 0.6	0.14584867383308084
4	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_19	false, 50, 300, 0.6	0.17548394969865572
...	...	...	...
35	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_1	false, 3, 300, 0.5	0.33499715745920294
36	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_0	true, 3, 300, 0.5	0.35441515942802954
37	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_30	true, 3, 300, 0.9	0.3553639619687519
38	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_10	true, 3, 300, 0.6	0.3592075912922027
39	Grid_DRF_py_10_sid_bfff_model_python_1497763697022_5193_model_20	true, 3, 300, 0.8	0.3614194946048834

Из всех возможных сочетаний модель с долей отобранных строк .9, глубиной дерева 50 и сбалансированными классами приводит к самой высокой точности. Теперь натренируем новую модель случайного леса, используя оптимальные параметры, полученные в результате сеточного поиска, и предскажем результат на тестовом наборе:

```
final = H2ORandomForestEstimator(ntrees=300, max_depth=50,
                                 balance_classes=True, sample_rate=.9)
final.train(x=spam_X, y=spam_Y, training_frame=train)

print(final.predict(test))
```

В результате прогнозирования в H2O будет сгенерирован массив, первый столбец которого содержит фактические предсказанные классы, а остальные столбцы – вероятности классов каждой целевой метки:

```
Out:
```

predict	p0	p1
1	0.00393672	0.996063
1	0	1
1	0.0804741	0.919526
1	0.0476355	0.952365
1	0.00146838	0.998532
1	0.0455365	0.954463
0	0.793245	0.206755
1	0.00726585	0.992734
1	0.071177	0.928823
1	0.0330345	0.966966

Стохастический градиентный бустинг и сеточный поиск в H2O

В предыдущих примерах мы видели, что большую часть времени хорошо отстроенная модель на основе алгоритма GBM превосходит по результативности случайный лес. Поэтому теперь выполним алгоритм GBM с сеточным поиском в H2O и посмотрим, сможем ли мы улучшить нашу отметку. Для этого сеанса мы введем

аналогичный метод генерирования случайных подвыборок, который мы использовали для модели на основе случайного леса в H2O (`sample_rate`). Метод **стохастического градиентного бустинга** был представлен на основе статьи 1999 г. Джерома Фридмана (Jerome Friedman) (https://statweb.stanford.edu/~jhf/ftp/stobst.pdf). Добавленная в модель стохастичность во время каждой итерации по дереву использует случайный отбор наблюдений без возврата, что позволяет предотвратить переподгонку и увеличить общий показатель точности. В данном примере мы развиваем эту идею стохастичности дальше, вводя случайный отбор признаков во время каждой итерации.

Подобного рода метод случайного отбора признаков также называется **методом случайных подпространств**, который мы уже видели в разделе «*Случайный лес и чрезвычайно рандомизированный лес*» настоящей главы. Он реализуется параметром `col_sample_rate`. Таким образом, резюмируя все вышесказанное, в модели на основе алгоритма GBM мы выполним оптимизацию методом сеточного поиска со следующими параметрами:

- `max_depth`: максимальная глубина дерева;
- `sample_rate`: часть строк, используемая в каждой итерации;
- `col_sample_rate`: часть признаков, используемая в каждой итерации.

Мы воспользуемся тем же набором с данными о спаме, что и в предыдущем разделе, поэтому сразу же приступаем к его обработке:

```
# Отметим, что из-за случайности результаты сеточного поиска
# могут отличаться от результатов, приводимых в книге.
hyper_parameters = {'ntrees':[300],
                    'max_depth':[12,30,50],
                    'sample_rate':[.5,.7,1],
                    'col_sample_rate':[.9,1],
                    'learn_rate':[.01,.1,.3]}

grid_search = H2OGridSearch(H2OGradientBoostingEstimator,
                            hyper_params=hyper_parameters)
grid_search.train(x=spam_X, y=spam_Y, training_frame=train)

print('Отимальное решение для гиперпараметрического поиска %s'
      % grid_search.show())
```

	model_ids	col_sample_rate learn_rate max_depth ntrees sample_rate	logloss
0	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_34	0.9, 0.3, 50, 300, 0.7	0.00157330308618194
1	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_22	0.9, 0.3, 12, 300, 0.7	0.00157927687447262
2	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_29	1.0, 0.3, 30, 300, 0.7	0.0015832978610301
3	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_35	1.0, 0.3, 50, 300, 0.7	0.00159738892462822
4	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_28	0.9, 0.3, 30, 300, 0.7	0.00159886840370323
...	...	...	...
49	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_13	1.0, 0.01, 50, 300, 0.5	0.104796736092932
50	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_12	0.9, 0.01, 50, 300, 0.5	0.10489031471896
51	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_18	0.9, 0.01, 12, 300, 0.7	0.105071198911258
52	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_1	1.0, 0.01, 12, 300, 0.5	0.115521269067435
53	Grid_GBM_py_10_sid_bfff_model_python_1497763697022_10537_model_0	0.9, 0.01, 12, 300, 0.5	0.117272332422832

Верхняя часть результата сеточного поиска показывает, что мы должны использовать исключительно высокий темп обучения .3, долю отбора столбцов .9 и максимальную глубину дерева 30. Случайный отбор наблюдений на основе строк не увеличил результативности, однако отбор на основе признаков с долей .9 в данном случае был довольно эффективным. Теперь натренируем новую модель на основе алгоритма GBM с использованием оптимальных параметров, полученных в результате оптимизации методом сеточного поиска, и предскажем результат на тестовом наборе:

```
spam_gbm2 = H2OGradientBoostingEstimator(ntrees=300,
                                         learn_rate=0.3,
                                         max_depth=30,
                                         sample_rate=1,
                                         col_sample_rate=0.9,
                                         score_each_iteration=True,
                                         seed=2000000)

spam_gbm2.train(spam_X, spam_Y, training_frame=train,
                validation_frame=valid)
confusion_matrix = spam_gbm2.confusion_matrix(metrics="accuracy")

print(confusion_matrix)
```

Confusion Matrix (Act/Pred) for max accuracy @ threshold = 0.967610499121:

	0	1	Error	Rate
0	1683.0	0.0	0.0	(0.0/1683.0)
1	3.0	1077.0	0.0028	(3.0/1080.0)
Total	1686.0	1077.0	0.0011	(3.0/2763.0)

Полученный результат предоставляет интересные диагностические данные о результативности модели, в частности `accuracy`, `rmse`, `logloss` и `AUC`. Однако конечный результат ее работы слишком объемный, чтобы его можно было здесь показать целиком. Все результаты в полном объеме можно посмотреть в блокноте Jupyter.

Его можно использовать следующим образом:

```
print(spam_gbm2.score_history())
```

Разумеется, итоговые предсказания можно получить следующим образом:

```
print(spam_gbm2.predict(test))
```

Отлично, мы смогли улучшить точность нашей модели почти до 100%. Как видно, в платформе H2O предлагается меньшая гибкость с точки зрения моделирования и подготовки данных, но достигаемые скорость обработки и точность не имеют равных. В заключение, чтобы завершить сеанс, можно сделать следующее:

```
h2o.cluster().shutdown(prompt=False)
```

Резюме

Мы увидели, что методы CART, натренированные ансамблевыми подпрограммами, проявляют свою мощь, когда дело касается точности прогнозирования. Однако они могут быть в вычислительном плане дорогостоящими, и мы раскрыли не-

которые методы их ускорения в приложениях из модуля sklearn. Мы обнаружили, что использование экстремально рандомизированных лесов, настроенных при помощи рандомизированного поиска, смогло ускорить их работу десятикратно при их соответствующем использовании. Тем не менее процедура параллелизации метода GBM в модуле sklearn не реализована, и именно по этой причине на первый план выходит алгоритм экстремального градиентного бустинга XGBoost.

Алгоритм экстремального градиентного бустинга XGBoost имеет встроенную возможность параллелизации, которая значительно его ускоряет. Когда мы используем большие файлы (тренировочные примеры > 100k), на помощь приходит внеядерный метод, который гарантирует, что во время тренировки моделей мы не будем перегружать оперативную память.

Самый большой прирост в скорости и преимущество от использования памяти можно получить благодаря платформе H2O; мы познакомились с ее мощными способностями по доводке параметров вместе с впечатляющей скоростью тренировки моделей.

Глава 7

Обучение без учителя в крупном масштабе

В предыдущих главах в центре задачи было предсказание переменной, которая, возможно, является числом, классом или категорией. В этой главе мы изменим подход и попробуем создать новые признаки и переменные в крупном масштабе, которые для целей нашего предсказания, надо надеяться, окажутся лучше, чем те, которые уже включены в матрицу наблюдений. Сначала мы представим методы машинного обучения без учителя и проиллюстрируем три из них, которые способны масштабироваться до больших данных:

- **анализ главных компонент**, эффективный способ снизить количество признаков;
- **k-средних**, масштабируемый алгоритм кластеризации данных;
- **латентное размещение Дирихле**, очень эффективный алгоритм, который способен извлекать темы из серии текстовых документов.

Методы машинного обучения без учителя

Обучение без учителя – это раздел машинного обучения, алгоритмы которого делают выводы из данных без явно заданной метки (немаркированных данных). Цель таких методов состоит в том, чтобы извлекать скрытые образы, или паттерны, и группировать похожие данные.

В этих алгоритмах вызывающие интерес неизвестные параметры каждого наблюдения (к примеру, принадлежность к группе и тематическая структура) часто моделируются как латентные переменные (или как серия латентных переменных), скрытые в системе наблюдаемых переменных, которые не могут наблюдаться непосредственно, а лишь выводятся из прошлых и настоящих конечных результатов работы системы. Как правило, конечные результаты работы системы содержат шум, который затрудняет эту операцию.

В типичных задачах методы машинного обучения без учителя используются в двух основных ситуациях:

- с маркированными наборами данных для извлечения дополнительных признаков, которые затем обрабатываются вниз по цепочке обработки классификатором/регрессором. Усиленные дополнительными признаками, эти методы могут оказаться более результативными;
- с маркированными или немаркированными наборами данных для извлечения некой информации о структуре данных. Этот класс алгоритмов обыч-

но используется во время фазы моделирования, именуемой **разведочным анализом данных** (exploratory data analysis, EDA).

В первую очередь, прежде чем приступить к иллюстрации, давайте импортируем в наш блокнот Jupyter модули, которые будут необходимы по ходу настоящей главы:

```
In:
import os
import copy
import tempfile
import numpy as np
import pandas as pd
import matplotlib.cm as cm
```

Разложение признаков – PCA

Анализ главных компонент (principal component analysis, PCA) – это алгоритм, который широко применяется для разложения размерностей входного сигнала и сохранения только *главных* из них. С математической точки зрения метод PCA выполняет ортогональное преобразование матрицы наблюдений, продуцируя набор линейных некоррелированных переменных, именуемых главными компонентами. Выходные переменные формируют базисный набор, каждая компонента которого ортонормирована относительно других. Кроме того, есть возможность ранжировать выходные компоненты (с тем чтобы использовать только главные), так как первая компонента содержит самую большую возможную дисперсию входного набора данных, вторая ортогональна первой (по определению) и содержит самую большую дисперсию остаточного сигнала, и третья ортогональна первым двум, и она опирается на остаточную дисперсию, и т. д.

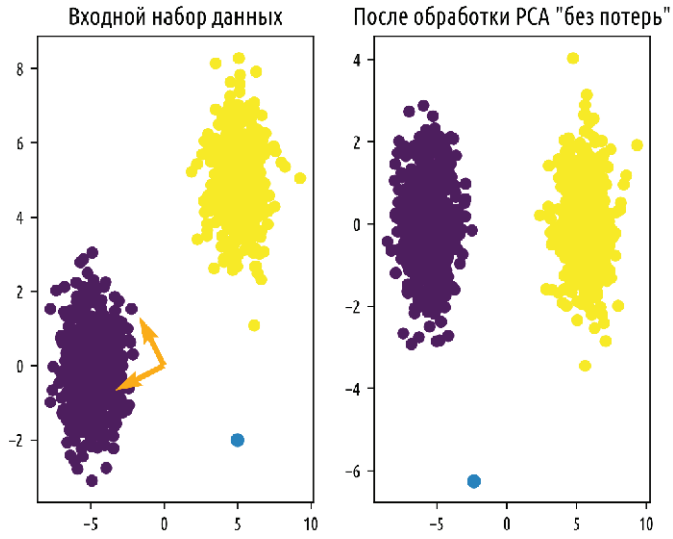
Универсальное преобразование при помощи алгоритма PCA может быть выражено как проекция в пространство. Если из трансформационного базиса берутся только главные компоненты, то выходное пространство будет иметь меньшую размерность, чем входное. Математически это можно выразить следующим образом:

$$\hat{X} = X \cdot T.$$

Здесь X – это обобщенная точка тренировочного набора размерностью N, T – трансформационная матрица, поступающая из алгоритма PCA, и $\hat{X}$ – выходной вектор. Отметим, что в данном матричном уравнении символ «·» обозначает скалярное произведение. С практической точки зрения также стоит отметить, что, прежде чем выполнять эту операцию, все признаки X должны быть центрированы на нуле.

Теперь давайте начнем с практического примера; позже мы подробно объясним математический аппарат алгоритма PCA. В этом примере мы создадим фиктивный набор данных, состоящий из двух скоплений точек – одно центрировано в (–5, 0), а другое – в (5, 5). Мы воспользуемся алгоритмом PCA, чтобы преобразовать набор данных и построить график конечных результатов в сравнении с входными данными. В этом простом примере будут использованы все признаки, т. е. операция сокращения признаков выполняться не будет:

```
In:
from sklearn.datasets.samples_generator import make_blobs
from sklearn.decomposition import PCA

X, y = make_blobs(n_samples=1000, # скопления точек
                  random_state=101, centers=[[-5, 0], [5, 5]])
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
pca_comp = pca.components_.T

test_point = np.matrix([5, -2])
test_point_pca = pca.transform(test_point)

plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='none')
plt.quiver(0, 0, -pca_comp[:,0], -pca_comp[:,1],   # -1
           width=0.02, scale=5, color='orange')
plt.plot(test_point[0, 0], test_point[0, 1], 'o')
plt.title(u'Входной набор данных')

plt.subplot(1, 2, 2)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, edgecolors='none')
plt.plot(-test_point_pca[0, 0], -test_point_pca[0, 1], 'o')  # -1
plt.title(u'После обработки PCA "без потерь"')
plt.show()
```



При выполнении приведенных в книге проекций PCA при помощи библиотеки Scikit-learn некоторые графики представляли собой зеркальное отражение графиков, приводимых в книге. Отметим, что это не следствие ошибки в одной из этих двух реализаций; причина такой разницы состоит в том, что в зависимости от алгоритма вычисления собственных векторов и значений они могут иметь отрицательный либо положительный знак. Не то чтобы это имеет значение, но в случае необходимости можно просто обратить зеркальное отражение, умножив данные на –1, что и было сделано при переводе главы ради сохранения логики изложения.

Как видно, конечный результат более организован, чем пространство исходных признаков, и если бы следующей задачей была классификация, то потребовался бы всего один признак набора данных, экономя почти 50% необходимого пространства и вычисления. На рисунке отчетливо видно ядро PCA: это проекция входного набора данных на трансформационный базис, который на рисунке изображен слева оранжевым цветом. Если вы в этом не уверены, давайте проверим:

```
In:
print('Синяя точка лежит в {0}'.format(test_point[0, :]))
print('После трансформации лежит в {0}'
      .format(-test_point_pca[0, :]))   # -1
print('Поскольку (X-MEAN) * PCA_MATRIX = {0}'
      .format(np.dot(-test_point - -pca.mean_, pca_comp)))  # -1 дважды

Out:
Синяя точка лежит в [[ 5 -2]]
После трансформации лежит в [-2.34969911 -6.2575445 ]
Поскольку (X-MEAN) * PCA_MATRIX = [[-2.34969911 -6.2575445 ]]
```

Теперь проанализируем базовую проблему: каким образом можно сгенерировать T из тренировочного набора? Он должен содержать ортонормированные векторы, и векторы должны быть ранжированы согласно количеству дисперсии (т. е. энергии или информации в матрице наблюдений), которую они могут объяснить. На сегодняшний день было реализовано большое количество решений, но наиболее распространенная реализация основана на **сингулярном разложении**.

Сингулярное разложение (singular value decomposition, SVD) – это метод, который разлагает любую матрицу M на три матрицы (U, Σ, W) с особыми свойствами, умножение которых снова дает M:

$$M = U \cdot \Sigma \cdot W^T.$$

А именно если дана матрица M размера m строк и n столбцов, результирующие элементы эквивалентности следующие:

- U – это матрица размера $m \times n$ (квадратная матрица), она унитарная, и ее столбцы формируют ортонормированный базис. Они также называются левыми, или входными, сингулярными векторами и представляют собой собственные векторы матричного произведения $M \cdot M^T$;
- Σ – это матрица размера $m \times n$, которая имеет на своей диагонали только ненулевые элементы. Эти значения называются сингулярными значениями, все они неотрицательные и являются собственными значениями как $M \cdot M^T$, так и $M^T \cdot M$;
- W – это унитарная матрица размера $n \times n$ (квадратная матрица), ее столбцы формируют ортонормированный базис, и они называются правыми (или выходными) сингулярными векторами. Кроме того, они являются собственными векторами матричного произведения $M^T \cdot M$.

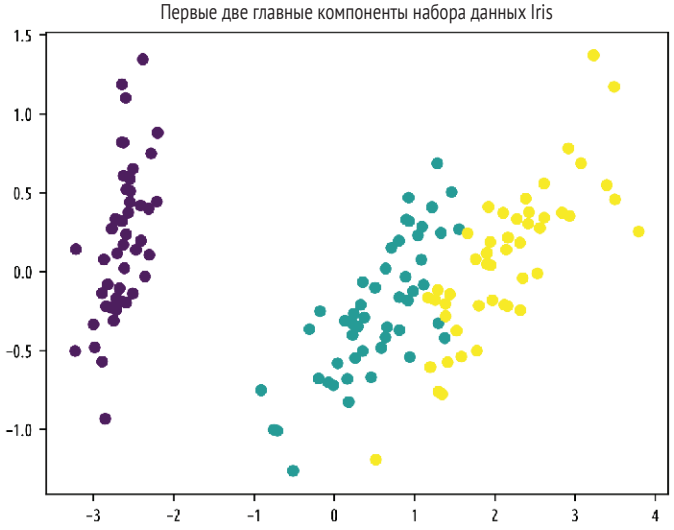
Зачем это необходимо? Решение довольно простое: цель алгоритма PCA состоит в том, чтобы попытаться оценить направления, где дисперсия входного набора данных больше. Для этого сначала из каждого признака нужно удалить среднее значение и затем оперировать с ковариационной матрицей $X^T \cdot X$.

С учетом этого, разложив матрицу X методом SVD, мы имеем столбцы матрицы W, которые являются главными компонентами ковариации (т. е. матрицы T, кото-

рую мы ищем), диагональ матрицы Σ, которая содержит дисперсию, объясненную главными компонентами, и столбцы матрицы U – главные компоненты. Именно поэтому анализ главных компонент всегда выполняется при помощи сингулярного разложения.

Давайте теперь рассмотрим это на реальном примере. Протестируем метод на наборе данных Iris с извлечением первых двух главных компонент (т. е. перейдем от набора данных, состоящего из четырех признаков, к набору данных из двух признаков):

```
In:
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data
y = iris.target

print('Набор данных Iris содержит {0} признаков'
      .format(X.shape[1]))

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

print('После обработки алгоритмом PCA он содержит {0} признаков'
      .format(X_pca.shape[1]))
print('Дисперсия составляет [% от оригинала]: {0}'
      .format(sum(pca.explained_variance_ratio_)))

plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, edgecolors='none')
plt.title(u'Первые 2 главные компоненты набора данных Iris')
plt.show()

Out:
Набор данных Iris содержит 4 признака
После обработки алгоритмом PCA он содержит 2 признака(ов)
Дисперсия составляет [% от оригинала]: 0.977631775025
```



Проведем анализ итоговых результатов процесса:

- объясненная дисперсия составляет почти 98% от исходной дисперсии из входных данных. Число признаков сократилось вдвое, но только 2% информации не находится в конечных результатах на выходе, и, надо надеяться, это просто шум;
- исходя из визуального обследования, по всей видимости, разные классы, которые составляют набор данных Iris, друг от друга разделены. Это означает, что классификатор, который оперирует таким сокращенным набором, будет иметь сопоставимую результативность с точки зрения точности, и при этом тренировка и предсказание будут проходить быстрее.

Как доказательство второго вывода давайте теперь попробуем натренировать и протестировать два классификатора, один из которых использует исходный набор данных, а другой – уменьшенный, и распечатаем их точность:

```
In:
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

def test_classification_accuracy(X_in, y_in):
    X_train, X_test, y_train, y_test = \
        train_test_split(X_in, y_in, random_state=101, train_size=0.50)

    clf = SGDClassifier('log', random_state=101)
    clf.fit(X_train, y_train)
    return accuracy_score(y_test, clf.predict(X_test))

print('Точность SGDClassifier на данных Iris: {0}'
      .format(test_classification_accuracy(X, y)))
print('Точность после обработки алгоритмом PCA (2 компоненты): {0}'
      .format(test_classification_accuracy(X_pca, y)))

Out:
Точность SGDClassifier на данных Iris: 0.586666666667
Точность после обработки алгоритмом PCA (2 компоненты): 0.72
```

Как видно, этот метод не только уменьшает сложность и пространство ученика по цепочке вниз, но и помогает достигнуть обобщения (в точности как гребневая либо лассо-регуляризация).

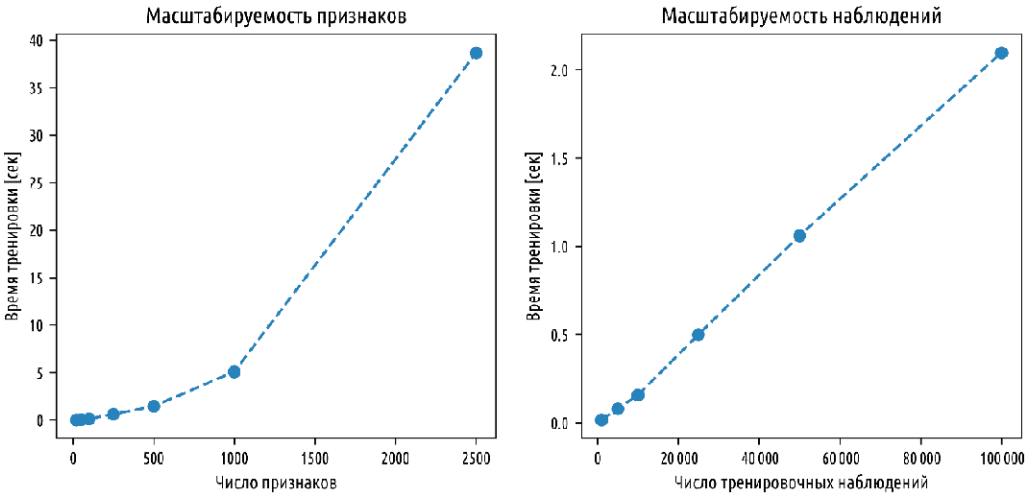
Если вы не уверены в том, сколько компонент должно быть на выходе, обычно в качестве практического ориентира следует выбрать минимальное число, которое в состоянии объяснить, по крайней мере, 90% (или 95%) входной дисперсии. Эмпирически такой выбор обычно гарантирует, что будет отсечен только шум.

Пока все выглядит идеально: мы нашли отличное решение, которое сокращает число признаков и создает некоторые из них с очень высокой прогнозирующей способностью, при этом мы также располагаем эмпирическим правилом для угадывания нужного их числа. Давайте теперь проверим, насколько это решение масштабируемо: мы обследуем масштабируемость решения при увеличении числа наблюдений и признаков. Первое, что следует отметить, – алгоритм SVD, т. е. *ядро* алгоритма PCA, не является стохастическим, и поэтому ему нужна вся матрица, чтобы он смог извлечь ее главные компоненты. Теперь давайте посмотрим на масштабируемый алгоритм PCA в деле на нескольких синтетических наборах

данных с возрастающим числом признаков и наблюдений. Мы выполним полное разложение (без потерь) (аргумент при инстанцировании объекта PCA равен None), поскольку более низкое число признаков не влияет на результативность (это просто вопрос нарезки выходных матриц метода SVD).

В приведенном ниже примере мы сначала создаем матрицы из 10 000 точек и с 20, 50, 100, 250, 1000 и 2500 признаками. Затем создаем матрицы со 100 признаками и 1, 5, 10, 25, 50 и 100 000 наблюдениями. Все они будут обработаны алгоритмом PCA:

```
In:
import time

def check_scalability(test_pca, plotnum=1):
    rcParams['figure.figsize']  = (8, 4)

    # ПРИЗНАКИ
    n_points = 10000
    n_features = [20, 50, 100, 250, 500, 1000, 2500]
    time_results = []

    for n_feature in n_features:
        X, _ = make_blobs(n_points, n_features=n_feature,
                          random_state=101)
        pca = copy.deepcopy(test_pca)
        tik = time.time()
        pca.fit(X)
        time_results.append(time.time()-tik)

    plt.subplot(1, 2, 1)
    plt.plot(n_features, time_results, 'o--')
    plt.title(u'Масштабируемость признаков')
    plt.xlabel(u'Число признаков')
    plt.ylabel(u'Время обучения [сек]')

    # НАБЛЮДЕНИЯ
    n_features = 100
    n_observations = [1000, 5000, 10000, 25000, 50000, 100000]
    time_results = []

    for n_points in n_observations:
        X, _ = make_blobs(n_points, n_features=n_features,
                          random_state=101)
        pca = copy.deepcopy(test_pca)
        tik = time.time()
        pca.fit(X)
        time_results.append(time.time()-tik)

    plt.subplot(1, 2, 2)
    plt.plot(n_observations, time_results, 'o--')
    plt.title(u'Масштабируемость наблюдений')
    plt.xlabel(u'Число тренировочных наблюдений')
    plt.ylabel(u'Тренировочное время [сек]')
    plt.show()
    rcParams['figure.figsize']  = (5, 4)

check_scalability(PCA(None), 1)
```



Четко видно, что алгоритм PCA на основе метода SVD не масштабируется: если число признаков увеличивается линейно, то необходимое на тренировку алгоритма время возрастает экспоненциально. Кроме того, время, требующееся на обработку матрицы с несколькими сотнями наблюдений, становится слишком продолжительным, и (на рисунке не показано) объем потребления оперативной памяти делает проблему невыполнимой для домашнего компьютера (с 16 Гб или меньшим количеством оперативной памяти). Становится очевидным, что алгоритм PCA на основе метода SVD не подходит для больших данных; к счастью, в последние годы было предложено много обходных решений. В последующих разделах будет предложено краткое введение в каждое из них.

Рандомизированный алгоритм PCA

Правильное название этой методики – *алгоритм PCA на основе рандомизированного метода SVD*, но она стала популярна именно под названием **рандомизированного алгоритма PCA**. В основе рандомизации лежит центральная идея избыточности всех главных компонент; фактически, если целью метода является снижение размерности, можно ожидать, что на выходе будет нужно всего несколько векторов (K главных). Если сосредоточиться на проблеме нахождения наилучших K главных векторов, то алгоритм будет масштабироваться лучше. Отметим, что в этом алгоритме K – число главных компонент на выходе – является ключевым параметром: если задать его слишком большим, то результативность не будет лучше, чем у классического алгоритма PCA; а если задать слишком низким, то объясненная итоговыми векторами дисперсия тоже будет слишком низкой.

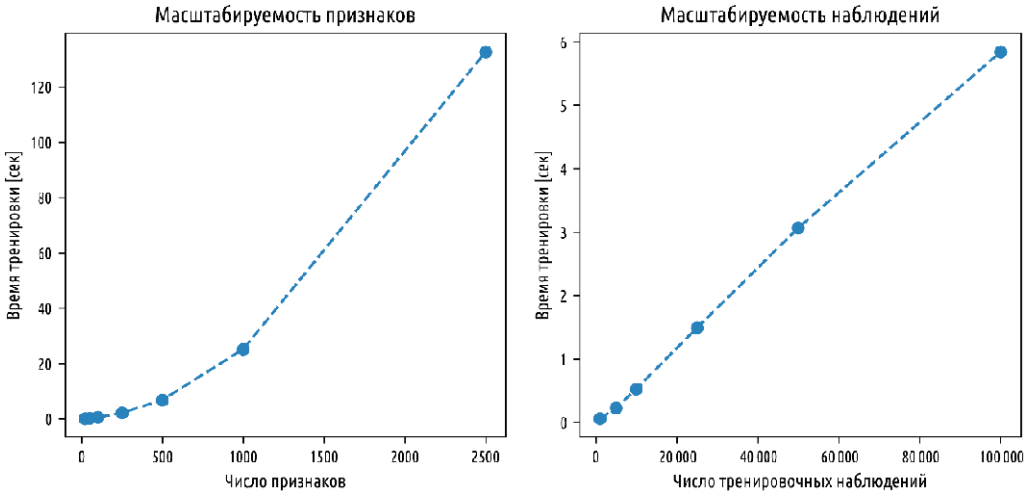
Как и в классическом алгоритме PCA, мы хотим найти приближение матрицы, содержащее наблюдения X, такие что $X \approx Q \cdot Q^T \cdot X$; мы также хотим получить

матрицу Q с ортонормированными столбцами K (они будут называться главными компонентами). При помощи метода SVD мы теперь можем вычислить разложение *малой* матрицы $Q^T \cdot X = U \cdot \Sigma \cdot W^T$. Как мы уже доказали, это не займет продолжительное время. Поскольку $X \approx Q \cdot Q^T \cdot X = Q \cdot U \cdot \Sigma \cdot W^T$, беря $Q \cdot U = S$, мы теперь имеем усеченное приближение X, основанное на низкоранговом SVD, $X \approx S \cdot \Sigma \cdot W^T$.

Математически это выглядит идеально, но по-прежнему не хватает ответа на два вопроса: какую роль здесь играет рандомизация и как получить матрицу Q? Ответы на оба вопроса приводятся далее: извлекается гауссова случайная матрица Ω, и вычисляется Y как $Y = X \cdot \Omega$. Затем Y подвергается QR-разложению, приводя к $Y = Q \cdot R$, где получаем искомую матрицу Q из K ортонормированных столбцов.

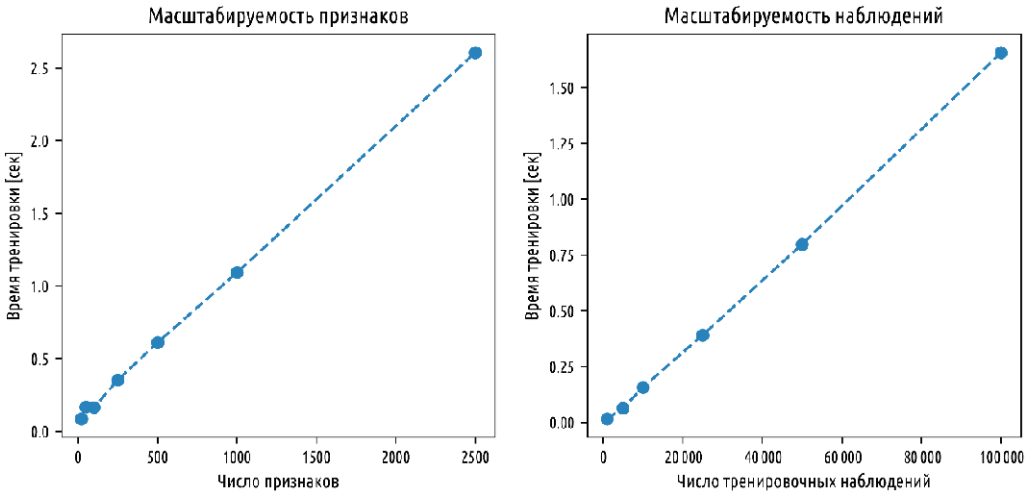
Математический аппарат в основе этого разложения довольно тяжел, но, к счастью, в библиотеке Scikit-learn весь уже реализован, и поэтому не нужно выяснять, как обращаться с гауссовыми случайными переменными и т. д. Давайте сперва посмотрим, насколько плохо рандомизированный алгоритм PCA работает, когда он вычисляет полное разложение (без потерь):

```
In:
check_scalability(PCA(svd_solver='randomized'), 2)
```



Результативность хуже, чем у классического алгоритма PCA; на самом деле это преобразование работает очень хорошо, когда запрашивается уменьшенный набор компонент. Теперь посмотрим на результативность, когда $K = 20$:

```
In:
check_scalability(PCA(n_components=20, svd_solver='randomized'), 3)
```



Как и ожидалось, вычисления очень быстрые; алгоритм способен выполнить самые сложные разложения на факторы меньше чем за секунду.

Проверяя результаты и алгоритм, мы по-прежнему замечаем нечто странное: для того чтобы тренировочный набор данных *X* можно было подвергнуть разложению, он целиком должен находиться в оперативной памяти, даже в случае рандомизированного алгоритма PCA. Существует ли онлайновая версия алгоритма PCA, которая в состоянии инкрементно выполнять подгонку главных векторов без размещения всего набора данных в памяти? Да, такая версия существует. Это инкрементный алгоритм PCA.

Инкрементный алгоритм PCA

Инкрементный, или мини-пакетный, алгоритм PCA является онлайновой версией анализа главных компонент. Ядро алгоритма очень простое: пакет данных первоначально разделяется на мини-пакеты с одинаковым числом наблюдений. (Единственное ограничение – число наблюдений в расчете на мини-пакет должно быть больше числа признаков.) Затем первый мини-пакет центрируется (удаляется среднее значение), и выполняется его SVD-разложение, сохраняя главные компоненты. Потом, когда в процесс вступает следующий мини-пакет, он сначала центрируется и затем накладывается вместе с главными компонентами, извлеченными из предыдущего мини-пакета (они вставляются как дополнительные наблюдения). Теперь выполняется другое SVD-разложение, и главные компоненты перезаписываются новыми. Процесс идет до последнего мини-пакета: для каждого из них сначала выполняется центрирование, затем наложение, и наконец SVD-разложение. Поступая таким образом, вместо *большого* SVD-разложения мы выполняем столько *малых* SVD-разложений, сколько имеется мини-пакетов.

Как можно понять, этот метод не превосходит по результативности рандомизированного алгоритма PCA, но его цель состоит в том, чтобы предложить решение (или единственное решение), когда алгоритм PCA требуется на наборе данных, который не умещается в оперативной памяти. Инкрементный алгоритм PCA предназначен не для того, чтобы выиграть состязание на скорость, а чтобы ограничить потребление памяти; в течение тренировки использование памяти остается по-

стоянным и может быть оптимизировано настройкой размера мини-пакета. Как показывает опыт, объем потребляемой памяти имеет приблизительно такой же порядок величины, что и квадрат размера мини-пакета.

В качестве примера давайте теперь проверим, как инкрементный алгоритм PCA справляется с большим набором данных, который в целях иллюстрации состоит из 10 млн наблюдений и 100 признаков. Ни один из предыдущих алгоритмов не в состоянии с ним справиться, если вы не хотите повредить свой компьютер (либо стать свидетелем огромного файла подкачки между оперативной памятью и дисковым хранилищем). При помощи инкрементного алгоритма PCA выполнение такой задачи становится проще простого, и, учитывая все обстоятельства, процесс не такой уж и медленный (отметим, что мы выполняем полное разложение без потерь со стабильным объемом потребляемой памяти):

```
In:
from sklearn.decomposition import IncrementalPCA

X, _ = make_blobs(100000, n_features=100, random_state=101)
pca = IncrementalPCA(None, batch_size=1000)

tik = time.time()
for i in range(100):
    pca.partial_fit(X)
print('PCA на 10M точках выполнялся с постоянным потреблением памяти {0} сек.'
      .format(time.time() - tik))

Out:
PCA на 10M точках выполнялся с постоянным потреблением памяти 223.279000044 сек.
```

Разреженный алгоритм PCA

Разреженный алгоритм PCA действует по-другому, чем предыдущие алгоритмы; вместо того чтобы управлять сокращением признаков при помощи SVD-разложения, применяемого к ковариационной матрице (после центрирования), он выполняет операцию, похожую на отбор признаков этой матрицы, отыскивая набор разреженных компонент, которые лучше всего реконструируют данные. Как и в лассо-регуляризации, количество разреженности контролируется штрафом (либо ограничением), налагаемым на коэффициенты.

По сравнению с классическим алгоритмом PCA, разреженный алгоритм PCA не гарантирует, что получающиеся компоненты будут ортогональными, но результат больше поддается толкованию, поскольку главные векторы в действительности являются составной частью входного набора данных. Кроме того, он масштабируется с точки зрения числа признаков: если классический алгоритм PCA и его масштабируемые версии зависают, когда число признаков становится больше (скажем, более 1000), разреженный алгоритм PCA по-прежнему остается оптимальным решением с точки зрения скорости благодаря внутреннему методу, решающему Lasso-задачу, который, как правило, основывается на алгоритме LARS[1] или методе координатного спуска. (Напомним, что Lasso[2] пытается мини-

---

[1] Метод наименьших углов (least angle regression, LARS) – алгоритм отбора признаков в задачах линейной регрессии. – *Прим. перев.*

[2] Lasso (Least absolute shrinkage and selection operator) – метод оценивания коэффициентов линейной регрессионной модели, который одновременно выполняет отбор признаков и регуляризацию. – *Прим. перев.*

мизировать L1-норму коэффициентов.) Кроме того, он великолепен, когда число признаков больше числа наблюдений, как, например, в некоторых наборах изображений.

Давайте теперь посмотрим, как он работает на наборе данных из 25 000 наблюдений и 10 000 признаков. Для этого примера мы воспользуемся мини-пакетной версией алгоритма `SparsePCA`, гарантирующей постоянное использование памяти и могущей справиться с крупными наборами данных, которые в конечном счете больше имеющейся в распоряжении оперативной памяти (отметим, что пакетная версия называется `SparsePCA`, но она не поддерживает тренировку в онлайновом режиме):

```
In:
from sklearn.decomposition import MiniBatchSparsePCA

X, _ = make_blobs(25000, n_features=10000, random_state=101)

tik = time.time()
pca = MiniBatchSparsePCA(20, method='cd',
                         random_state=101, n_iter=1000)
pca.fit(X)
print('Разреженный алгоритм PCA на матрице {0} выполнялся {1} сек.'
      .format(X.shape, time.time() - tik))

Out:
```

Разреженный алгоритм PCA на матрице (25000L, 10000L) выполнялся 134.220999956 сек.

Алгоритм `SparsePCA` способен произвести решение примерно за 40 секунд при постоянном потреблении памяти.

Алгоритм PCA в среде H2O

Можно также воспользоваться реализацией алгоритма PCA, предлагаемой средой H2O. (Мы уже встречались с платформой H2O в предыдущей главе и не раз ее упоминали на протяжении всей книги.)

Работая в среде H2O, сначала при помощи метода `init` необходимо включить сервер. Затем выгрузить набор данных в файл (в CSV-файл, если быть точным) и наконец выполнить PCA-анализ. В качестве последнего шага мы закрываем сервер.

Испытаем эту реализацию на самых больших наборах данных, которые мы встречали до сих пор, – с 100K наблюдениями и 100 признаками и с 10K наблюдениями и 2500 признаками:

```
In:
import h2o
from h2o.transforms.decomposition import H2OPCA

h2o.init(max_mem_size_GB=4)

def testH2O_pca(nrows, ncols, k=20):
    temp_file = tempfile.NamedTemporaryFile().name
    X, _ = make_blobs(nrows, n_features=ncols, random_state=101)
    np.savetxt(temp_file, np.c_[X], delimiter=",")
    del X
```

```
    pca = H2OPCA(k=k, transform='NONE', pca_method='Power')
    tik = time.time()
    pca.train(x=range(100), training_frame=h2o.import_file(temp_file))

    print('H2O PCA на матрице {0} выполнялся {1} сек.'
          .format((nrows, ncols), time.time() - tik))
    os.remove(temp_file)

testH2O_pca(100000, 100)
testH2O_pca(10000, 2500)
h2o.cluster().shutdown(prompt=False)

Out:
...
H2O PCA на матрице (100000, 100) выполнялся 40.0310001373 сек.
...
H2O PCA на матрице (10000, 2500) выполнялся 19.6699998379 сек.
```

Как видно, в обоих случаях платформа H2O действительно работает очень быстро, вполне сопоставимо с библиотекой Scikit-learn (если вообще не превосходит его по производительности).

Кластеризация – алгоритм K-средних

Алгоритм K-средних (K-means) – это алгоритм без учителя, который создает K непересекающихся кластеров точек с равной дисперсией, минимизируя искажение (так называемую инерцию).

При наличии всего одного параметра K, представляющего число кластеров, которые будут создаваться, алгоритм K-средних создает K множеств точек S_1, S_2, ..., S_k, каждое из которых представлено его центроидом: C_1, C_2, ..., C_k. Обобщенный центроид C_i – это просто среднее значение из выборок точек, связанных с кластером S_i, с целью минимизации внутрикластерного расстояния. Конечные результаты работы системы следующие.

1. Композиция кластеров S_1, S_2, ..., S_k, т. е. множество точек, составляющее тренировочный набор, которые связаны с номером кластера 1, 2, ..., K.
2. Центроиды каждого кластера C_1, C_2, ..., C_k. Центроиды могут использоваться для будущих ассоциаций.
3. Искажение, вносимое кластеризацией, которое вычисляется следующим образом:

$$D = \sum_{i=1}^{K} \sum_{x \in S_i} \|x - C_i\|^2.$$

Данное уравнение обозначает оптимизацию, которая органически выполняется в алгоритме K-средних: центроиды выбираются с целью минимизации внутрикластерного искажения, т. е. суммы евклидовых норм расстояний между каждой входной точкой и центроидом кластера, с которым точка ассоциирована. Другими словами, алгоритм пытается подобрать наилучшее векторное квантование.

Тренировочная фаза алгоритма K-средних также именуется алгоритмом Ллойда, названным в честь Стюарта Ллойда (Stuart Lloyd), который первым предложил этот алгоритм. Это итеративный алгоритм, состоящий из двух фаз,

многократно итерируемых до сходимости (искажение достигает минимума). Этот алгоритм является вариантом обобщенного алгоритма **максимизации ожидания** (expectation-maximization, EM), так как первый шаг создает функцию для **ожидания** (E) отметки, а шаг **максимизации** (M) вычисляет параметры, которые максимизируют эту отметку. (Отметим, что в этой формулировке мы пытаемся достигнуть противоположного, т. е. минимизации искажения.) Вот его формула:

- шаг ожидания: на этом шаге точки в тренировочном наборе назначаются ближайшему центроиду:

  $$S_i^{(t)} = \{x : \|x - C_i\|^2 = \min_j \|x - C_j\|^2\}.$$

  Этот шаг также называется присвоением, или векторным квантованием;
- шаг максимизации: центроид каждого кластера перемещается в середину кластера путем усреднения составляющих его точек:

  $$C_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \cdot \sum_{x \in S_i^{(t)}} x \text{ для } i = 1, \ldots, K.$$

  Этот шаг также называется шагом обновления.

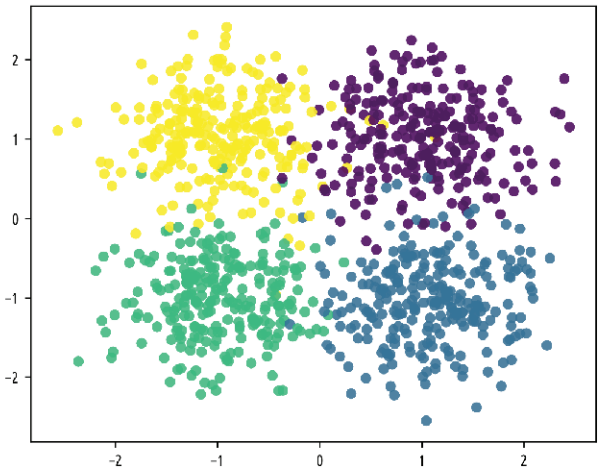
Эти два шага выполняются до сходимости (точки стабильны в своем кластере), либо пока алгоритм не достигнет предварительно установленного числа итераций. Отметим, что в соответствии с композицией искажение не может увеличиться на протяжении тренировочной фазы (в отличие от методов на основе стохастического градиентного спуска); следовательно, в этом алгоритме чем больше итераций, тем лучше результат.

Теперь давайте посмотрим, как он выглядит на фиктивном двумерном наборе данных. Сначала мы создаем набор из 1000 точек, сконцентрированных в четырех расположениях, симметричных относительно источника. Каждый кластер в соответствии с конструкцией имеет одинаковую дисперсию:

```
In:
import numpy as np
import pandas as pd
from sklearn.datasets.samples_generator import make_blobs

centers = [[1, 1], [1, -1], [-1, -1], [-1, 1]]
X, y = make_blobs(n_samples=1000, centers=centers,
                  cluster_std=0.5, random_state=101)
```

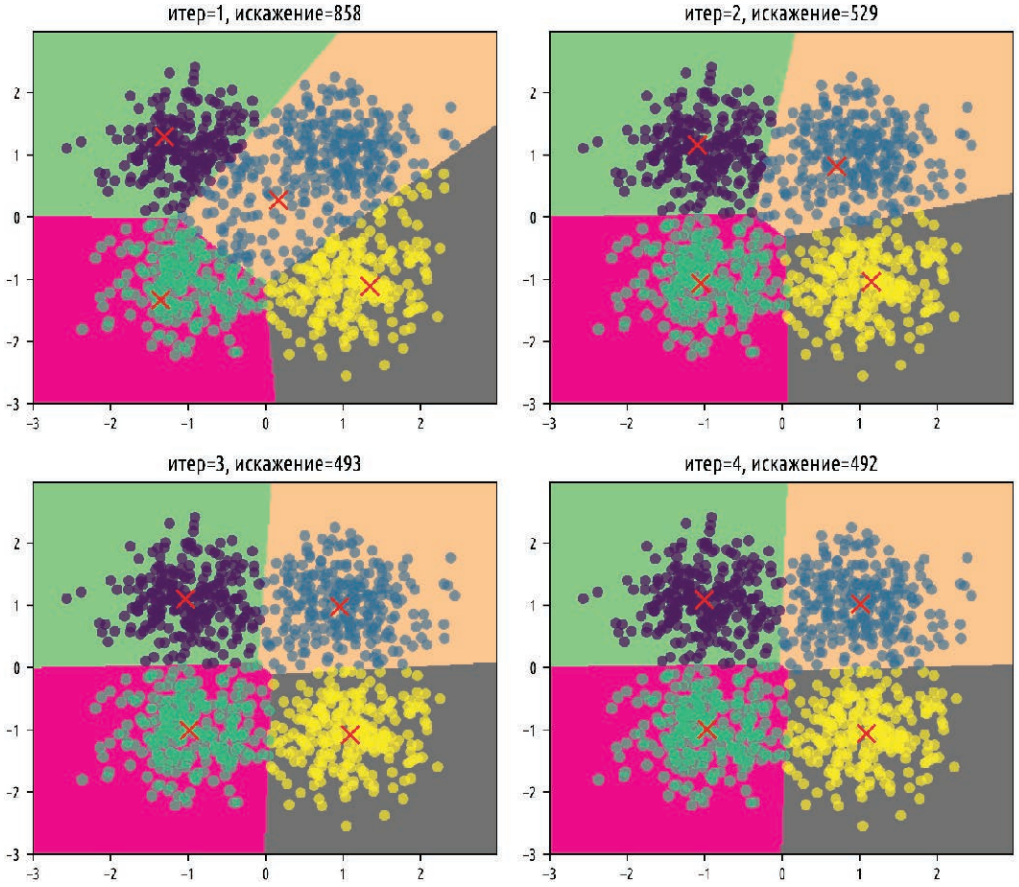
Теперь построим график набора данных. Чтобы упростить, мы окрасим кластеры в разные цвета:

```
In:
plt.scatter(X[:,0], X[:,1], c=y, edgecolors='none', alpha=0.9)
plt.show()
```



Теперь выполним алгоритм K-средних и проинспектируем, что происходит на каждой итерации. Для этого мы будем останавливать итеративный процесс на 1, 2, 3 и 4-й итерациях и строить график с изображением точек со связанным с ними кластером (с цветной маркировкой), а также изображением центроида, искажения (в заголовке) и границы решения (так называемые ячейки Вороного). Первоначальный отбор центроидов выполняется случайным образом, т. е. четыре тренировочные точки – это отобранные центроиды в первой итерации во время тренировки в фазе ожидания:

```
In:
rcParams['figure.figsize']  = (8, 7)
from sklearn.cluster import KMeans

for n_iter in range(1, 5):

    cls = KMeans(n_clusters=4, max_iter=n_iter, n_init=1,
                 init='random', random_state=101)
    cls.fit(X)

    # Вывести на графике ячейки Вороного

    plt.subplot(2, 2, n_iter)
    h=0.02
    xx, yy = np.meshgrid(np.arange(-3, 3, h), np.arange(-3, 3, h))
    Z = cls.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
    plt.imshow(Z, interpolation='nearest', cmap=plt.cm.Accent,
               extent=(xx.min(), xx.max(), yy.min(), yy.max()),
               aspect='auto', origin='lower')

    plt.scatter(X[:,0], X[:,1], c=cls.labels_,
                edgecolors='none', alpha=0.7)
    plt.scatter(cls.cluster_centers_[:,0], cls.cluster_centers_[:,1],
                marker='x', color='r', s=100, linewidths=4)
    plt.title(u'итер=%s, искажение=%s' % (n_iter, int(cls.inertia_)))

plt.show()
rcParams['figure.figsize']  = (5, 4)
```



Как видно из графиков, искажение становится все ниже и ниже по мере того, как число итераций увеличивается. Для этого фиктивного набора данных, по всей видимости, достаточно нескольких итераций (пяти итераций), чтобы достичь сходимости.

Методы инициализации

Нахождение глобального минимума искажения в алгоритме K-средних является задачей недетерминированной полиномиальной сложности (NP-hard); кроме того, точно так же, как и со стохастическим градиентным спуском, этот метод демонстрирует тенденцию сходиться к локальным минимумам, в особенности если число размерностей высокое. Во избежание такого поведения и ограничения максимального количества итераций можно воспользоваться следующими контрмерами:

- многократно выполнить алгоритм, используя разные исходные условия. В библиотеке Scikit-learn класс KMeans имеет параметр n_init, который управляет тем, сколько раз алгоритм K-средних будет выполнен с разными первоначальными числами центроидов. В конце отбирается модель, которая обеспечивает более низкое искажение. Если CPU имеет несколько ядер,

то этот процесс может выполняться параллельно путем установки параметра n_jobs в число требуемых заданий для запуска. Отметим, что потребление памяти линейно зависит от числа параллельных заданий;

- отдать предпочтение инициализации методом K-средних++ (класс KMeans используется по умолчанию) случайному выбору тренировочных точек. В результате инициализации методом K-средних++ отбираются *далеко отстоящие* друг от друга точки; это должно гарантировать, что центроиды смогут сформировать кластеры в универсальных подпространствах пространства. Как было доказано, этот факт также гарантирует, что этот метод *с большей вероятностью* найдет наилучшее решение.

Допущения алгоритма K-средних

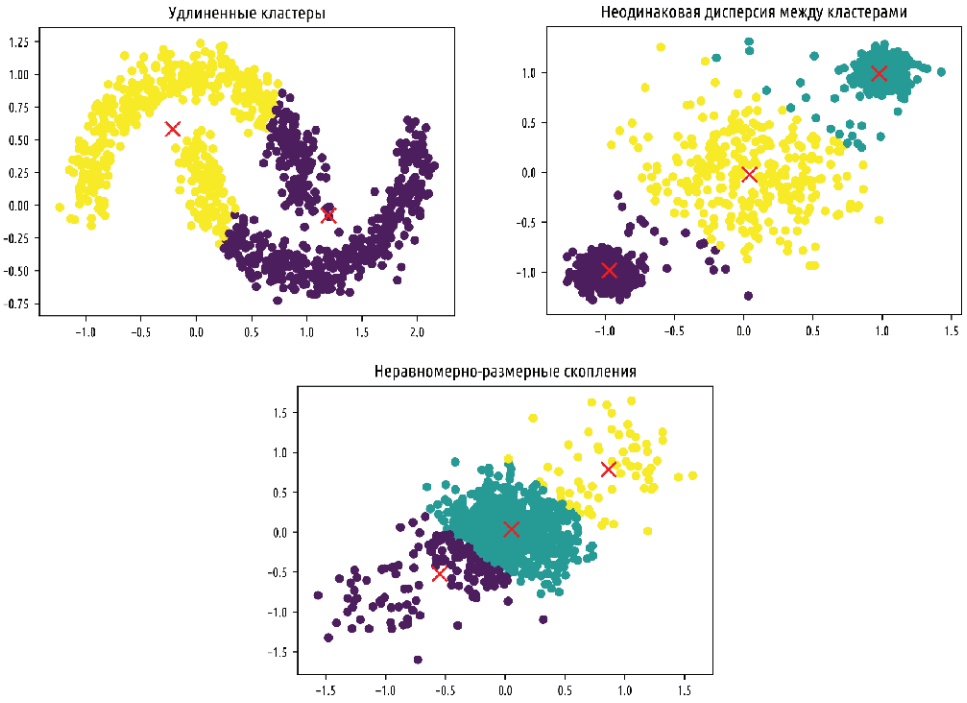
Алгоритм K-средних опирается на допущения, что каждый кластер имеет (гипер) сферическую форму, т. е. он не имеет удлиненной формы (подобно стреле), внутри все кластеры имеют одинаковую дисперсию, а их размер сопоставим (либо они отстоят очень далеко).

Все эти допущения могут быть обеспечены при помощи мощного шага предобработки признаков; классический алгоритм PCA, ядерный алгоритм PCA, нормализация признаков и выборка могут быть неплохим началом.

Давайте теперь посмотрим, что происходит, когда допущения, лежащие в основе алгоритма K-средних, не удовлетворяются:

```
In:
pylab.rcParams['figure.figsize'] = (5.0, 10.0)
from sklearn.datasets import make_moons

# Продолговатые/удлиненные наборы
X, _ = make_moons(n_samples=1000, noise=0.1, random_state=101)
cls = KMeans(n_clusters=2, random_state=101)
y_pred = cls.fit_predict(X)

plt.subplot(3, 1, 1)
plt.scatter(X[:, 0], X[:, 1], c=y_pred, edgecolors='none')
plt.scatter(cls.cluster_centers_[:,0], cls.cluster_centers_[:,1],
            marker='x', color='r', s=100, linewidths=4)
plt.title("Удлиненные кластеры")

# Разная дисперсия среди кластеров
centers = [[-1, -1], [0, 0], [1, 1]]
X, _ = make_blobs(n_samples=1000, cluster_std=[0.1, 0.4, 0.1],
                  centers=centers, random_state=101)
cls = KMeans(n_clusters=3, random_state=101)
y_pred = cls.fit_predict(X)

plt.subplot(3, 1, 2)
plt.scatter(X[:, 0], X[:, 1], c=y_pred, edgecolors='none')
plt.scatter(cls.cluster_centers_[:,0], cls.cluster_centers_[:,1],
            marker='x', color='r', s=100, linewidths=4)
plt.title("Неодинаковая дисперсия между кластерами")

# Неравномерно-размерные скопления
centers = [[-1, -1], [1, 1]]
```

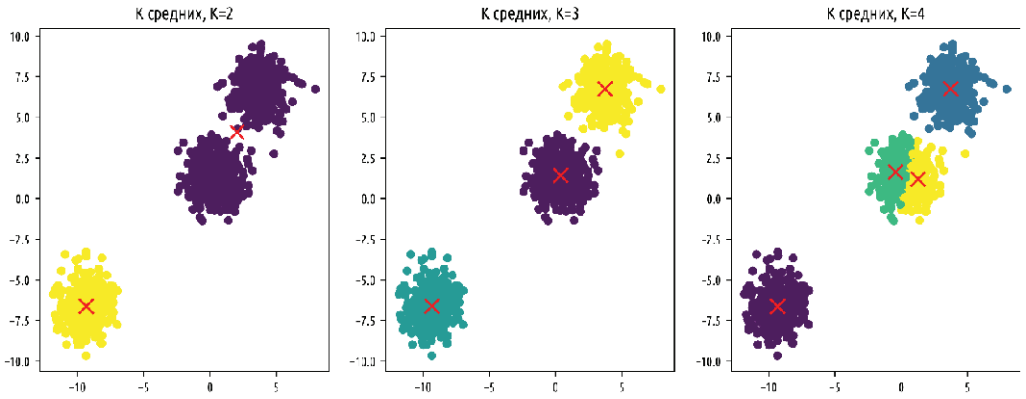
```
centers.extend([[0,0]]*20)
X, _ = make_blobs(n_samples=1000, centers=centers,
                  cluster_std=0.28, random_state=101)
cls = KMeans(n_clusters=3, random_state=101)
y_pred = cls.fit_predict(X)

plt.subplot(3, 1, 3)
plt.scatter(X[:, 0], X[:, 1], c=y_pred, edgecolors='none')
plt.scatter(cls.cluster_centers_[:,0], cls.cluster_centers_[:,1],
            marker='x', color='r', s=100, linewidths=4)
plt.title("Неравномерно-размерные скопления")

plt.show()
```



Во всех предыдущих примерах операция кластеризации не идеальна, производя на выходе неверный и нестабильный результат.

До настоящего момента мы приняли, что знаем наверняка, каково точное значение *K*, т. е. число кластеров, которые мы ожидаем использовать в операции кластеризации. На деле в реальных задачах это не всегда соответствует истине. Мы часто используем метод обучения без учителя для обнаружения глубинной структуры данных, включая число кластеров, составляющих набор данных. Давайте посмотрим, что происходит, когда мы пытаемся выполнить алгоритм K-средних с неправильным числом *K*, на простом фиктивном наборе данных; мы испытаем более низкое и более высокое значение *K*:

```
In:
rcParams['figure.figsize']  = (10, 4)
X, _ = make_blobs(n_samples=1000, centers=3, random_state=101)

for K in [2, 3, 4]:
    cls = KMeans(n_clusters=K, random_state=101)
    y_pred = cls.fit_predict(X)

    plt.subplot(1, 3, K-1)
    plt.title(u'K средних, K=%s' % K)
    plt.scatter(X[:, 0], X[:, 1], c=y_pred, edgecolors='none')
    plt.scatter(cls.cluster_centers_[:,0], cls.cluster_centers_[:,1],
                marker='x', color='r', s=100, linewidths=4)

plt.show()
rcParams['figure.figsize']  = (5, 4)
```



Как видно, результаты являются в массовом порядке неправильными в случае, если правильное число *K* не угадано, даже для такого простого фиктивного набора данных. В следующем разделе мы объясним некоторые приемы того, как наилучшим образом подобрать значение числа *K*.

Подбор оптимальной величины K

Если допущения, лежащие в основе алгоритма K-средних, удовлетворены, то для обнаружения наилучшей величины числа *K* разработано несколько методов. Некоторые из них основываются на перекрестной проверке и метриках конечных результатов; они могут использоваться на всех методах кластеризации, но только когда имеются сведения, полученные в результате непосредственных наблюдений (ground truth) (это так называемые контролируемые метрики). Некоторые другие основываются на внутренне присущих параметрах алгоритма кластеризации и могут использоваться независимо от присутствия или отсутствия непосредственных наблюдений (это так называемые неконтролируемые метрики). К сожалению, ни один из них не гарантирует 100%-ную точность нахождения правильного результата.

Контролируемые метрики требуют непосредственных наблюдений (содержащих истинные ассоциации в наборах), и они обычно объединяются с анализом на основе сеточного поиска для понимания того, какая величина *K* будет оптимальной. Некоторые из этих метрик вытекают из эквивалентных классификационных метрик, но они позволяют иметь разное число неупорядоченных наборов в качестве предсказанных меток. Первая метрика, которую мы собираемся рассмотреть, называется **однородностью**. Как можно ожидать, она дает меру того, сколько из предсказанных кластеров содержат точки одного класса. Данная мера основана на энтропии и является кластерным эквивалентом прецизионности[1] в классификации. Значения этой метрики лежат между 0 (худший) и 1 (лучший), а ее математическая формула следующая:

$$h = 1 - \frac{H(C \mid K)}{H(C)}.$$

Здесь $H(C|K)$ – это условная энтропия распределения классов при наличии предложенного назначения кластеров, и $H(C)$ – энтропия классов. $H(C|K)$ максимальна и равна $H(C)$, когда кластеризация не предоставляет новой информации, и равна нулю, когда каждый кластер содержит член только одного класса.

С ней связана, как в случае прецизионности и полноты для классификации, отметка **полноты** (completeness): она дает меру того, сколько всех членов класса назначено одному и тому же кластеру. И данная метрика тоже ограничена значениями между 0 (худший) и 1 (лучший), а ее математическая формула прочно основана на энтропии:

$$c = 1 - \frac{H(K \mid C)}{H(K)}.$$

Здесь $H(K|C)$ – это условная энтропия предложенного распределения кластеров при наличии класса, и $H(K)$ – энтропия кластеров.

Наконец, эквивалентная отметке F1 для задачи классификации V-мера – это среднее гармоническое однородности и полноты:

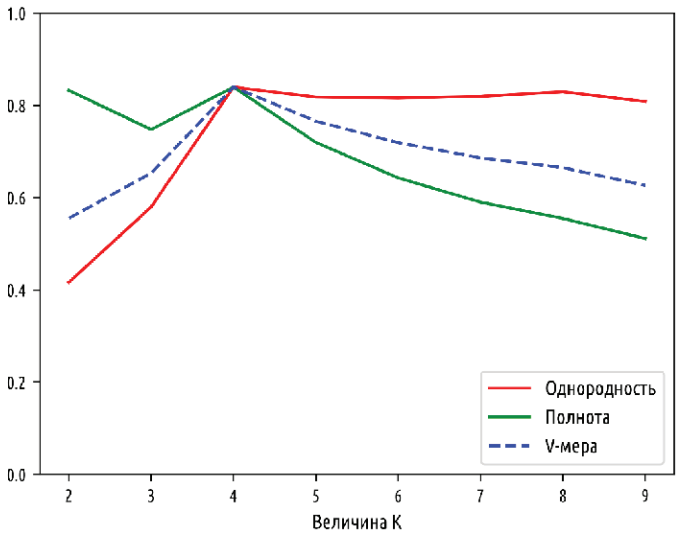
$$v = 2 \cdot \frac{h \cdot c}{h + c}.$$

Давайте вернемся к первому набору данных (четыре симметричных шумных кластера) и попытаемся увидеть, как эти отметки действуют и способны ли они выделить оптимальную величину *K* для использования:

```
In:
from sklearn.metrics import homogeneity_completeness_v_measure

centers = [[1, 1], [1, -1], [-1, -1], [-1, 1]]
X, y = make_blobs(n_samples=1000, centers=centers,
                  cluster_std=0.5, random_state=101)

Ks = range(2, 10)
```

[1] Прецизионность (англ. *precision* – точность измерений) – степень близости друг к другу независимых результатов измерений, полученных в конкретных установленных условиях. – *Прим. перев.*

```
HCVs = []
for K in Ks:
    y_pred = KMeans(n_clusters=K, random_state=101).fit_predict(X)
    HCVs.append(homogeneity_completeness_v_measure(y, y_pred))

plt.plot(Ks, [el[0] for el in HCVs], 'r', label=u'Однородность')
plt.plot(Ks, [el[1] for el in HCVs], 'g', label=u'Полнота')
plt.plot(Ks, [el[2] for el in HCVs], 'b--', label=u'V-мера')
plt.ylim([0, 1])
plt.xlabel(u'Величина K')
plt.legend(loc=4)
plt.show()
```



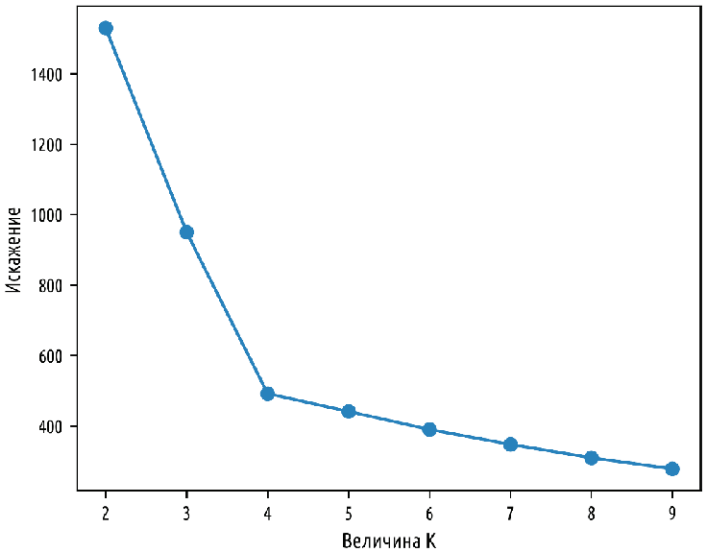
На графике первоначально ($K < 4$) полнота высокая, но однородность низкая; для $K > 4$ все наоборот: однородность высокая, но полнота низкая. В обоих случаях V-мера является низкой. Для $K = 4$, напротив, вся мера достигает их максимума, указывая, что это наилучшая величина K, т. е. число кластеров.

За пределами этих контролируемых метрик существуют другие так называемые неконтролируемые, которые не требуют непосредственных наблюдений, а просто основаны на самом ученике.

Первая метрика, которую мы рассмотрим в этом разделе, называется **методом локтя** и применяется к искажению. Она очень простая и не требует никакой математики: просто необходимо построить график искажения нескольких моделей K-средних с разными значениями K, затем выбрать ту, в которой увеличение K не вносит в решение *намного более низкого* искажения. На Python это реализуется очень просто:

```
In:
Ks = range(2, 10)
Ds = []
for K in Ks:
    cls = KMeans(n_clusters=K, random_state=101)
```

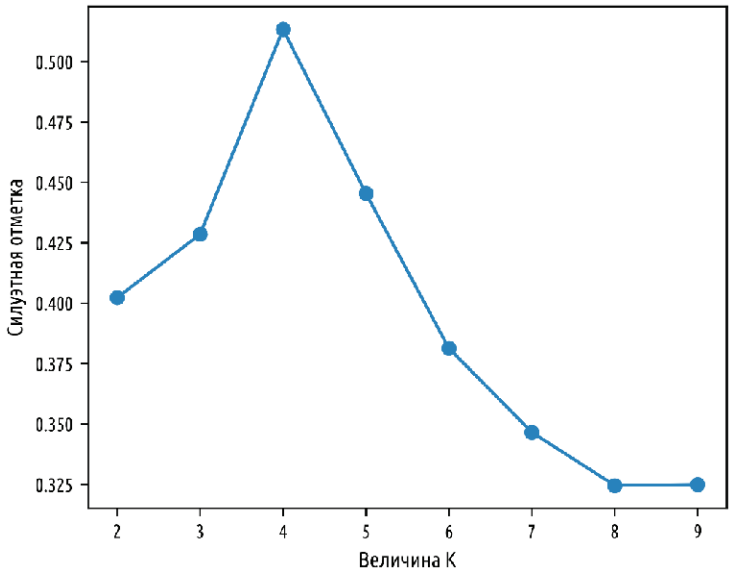
```
    cls.fit(X)
    Ds.append(cls.inertia_)

plt.plot(Ks, Ds, 'o-')
plt.xlabel(u'Величина K')
plt.ylabel(u'Искажение')
plt.show()
```



Как можно ожидать, искажение резко падает вплоть до $K = 4$ и затем уменьшается медленно. Здесь оптимальная величина K равна 4.

Еще одна неконтролируемая метрика, которую мы собираемся рассмотреть, – это **силуэтная отметка**. Данная метрика более сложная, но и более мощная, чем предыдущая эвристика. На очень высоком уровне она измеряет то, насколько близко (похоже) наблюдение назначенному кластеру и насколько неточно (непохоже) оно совпадает с данными соседних кластеров. Силуэтная отметка 1 указывает, что все данные находятся в наилучшем кластере, и –1 указывает на абсолютно неправильный результат кластеризации. Python'овский программный код для получения этой меры очень прост благодаря ее реализации в библиотеке Scikit-learn:

```
In:
from sklearn.metrics import silhouette_score

Ks = range(2, 10)
Ds = []
for K in Ks:
    cls = KMeans(n_clusters=K, random_state=101)
    Ds.append(silhouette_score(X, cls.fit_predict(X)))

plt.plot(Ks, Ds, 'o-')
plt.xlabel(u'Величина K')
plt.ylabel(u'Силуэтная отметка')
plt.show()
```



И в этом случае мы тоже пришли к тому же самому выводу: оптимальное значение для *K* равняется 4, так как силуэтная отметка намного ниже при более низком и более высоком значениях *K*.

Масштабирование алгоритма K-средних – мини-пакет

Теперь давайте протестируем масштабируемость алгоритма K-средних. На вебсайте UCI мы отобрали подходящий для этой задачи набор данных: данные переписи населения США 1990 года. Этот набор данных содержит почти 2.5 млн наблюдений и 68 категориальных (кодированных числами) атрибутов. Пропущенные данные отсутствуют, и файл находится в формате CSV. Каждое наблюдение содержит идентификатор человека (который должен быть удален перед кластеризацией) и другую информацию о половой принадлежности, доходе, семейном положении, работе и т. д.

Дополнительную информацию о наборе данных можно найти на http://archive.ics.uci.edu/ml/datasets/US+Census+Data+%281990%29 либо в работе, опубликованной Миком, Тиссоном и Хекерманом (Meek, Thiesson, Heckerman) в сборнике Journal of Machine Learning Research (2001), под названием *The Learning Curve Method Applied to Clustering («Метод кривой обучения применительно к кластеризации»)*.

Первое, что нужно сделать, – скачать файл, содержащий набор данных, и сохранить его во временном каталоге. Отметим, что он имеет объем 345 Мб, и поэтому на низкоскоростных соединениях его скачивание может занять продолжительное время:

```
In:
import urllib
import os.path

url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/census1990-mld/USCensus1990.
data.txt'
```

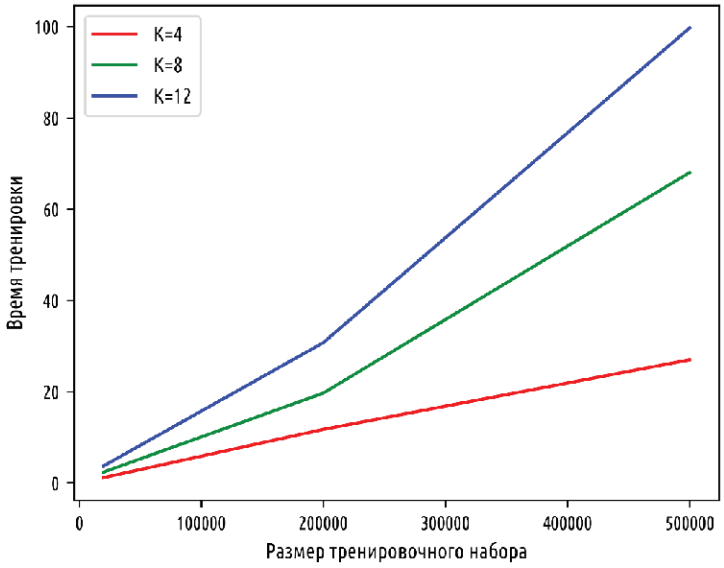
```
census_csv_file = './USCensus1990.data.txt'

if not os.path.exists(census_csv_file):
    testfile = urllib.URLopener()
    testfile.retrieve(url, census_csv_file)
if not os.path.exists(census_csv_file):
    testfile = urllib.URLopener()
    testfile.retrieve(url, census_csv_file)
```

Теперь давайте выполним несколько тестов, замеряя время, требующееся на тренировку ученика K-средних с величиной *K*, равной 4, 8 и 12, и с набором данных, содержащим 20K, 200K и 0.5M наблюдений. Поскольку мы не хотим перегружать оперативную память машины, поэтому мы прочтем первые 500K строк и отбросим столбец, содержащий идентификатор пользователя. В заключение построим график времен тренировки, чтобы выполнить полную оценку результативности[1]:

```
In:
piece_of_dataset = \
    pd.read_csv(census_csv_file, iterator=True)
                .get_chunk(500000).drop('caseid', axis=1).as_matrix()

time_results = {4: [], 8:[], 12:[]}
dataset_sizes = [20000, 200000, 500000]

for dataset_size in dataset_sizes:
    print('Размер набора данных: {0}'.format(dataset_size))
    X = piece_of_dataset[:dataset_size,:]

    for K in [4, 8, 12]:
        print('K: {0}'.format(K))
        cls = KMeans(K, random_state=101)
        timeit = %timeit -o -n1 -r1 cls.fit(X)
        time_results[K].append(timeit.best)

plt.plot(dataset_sizes, time_results[4], 'r', label='K=4')
plt.plot(dataset_sizes, time_results[8], 'g', label='K=8')
plt.plot(dataset_sizes, time_results[12], 'b', label='K=12')

plt.xlabel(u'Размер тренировочного набора')
plt.ylabel(u'Время тренировки')
plt.legend(loc=0)
plt.show()

Out:
Размер набора данных: 20000
K: 4
1 loop, best of 1: 1.16 s per loop (478 ms)
K: 8
1 loop, best of 1: 2.38 s per loop (1.22 s)
K: 12
```

[1] В связи с тем, что прилагаемый к книге программный код тестировался на ноутбуке достаточно скромной производительности Asus Intel Core i3 CPU @ 1.8 GHz 6 Гб, в нескольких местах в переводе среди результатов работы фрагментов программного кода в скобках показаны временные отметки, приводимые в оригинале книги. – *Прим. перев.*

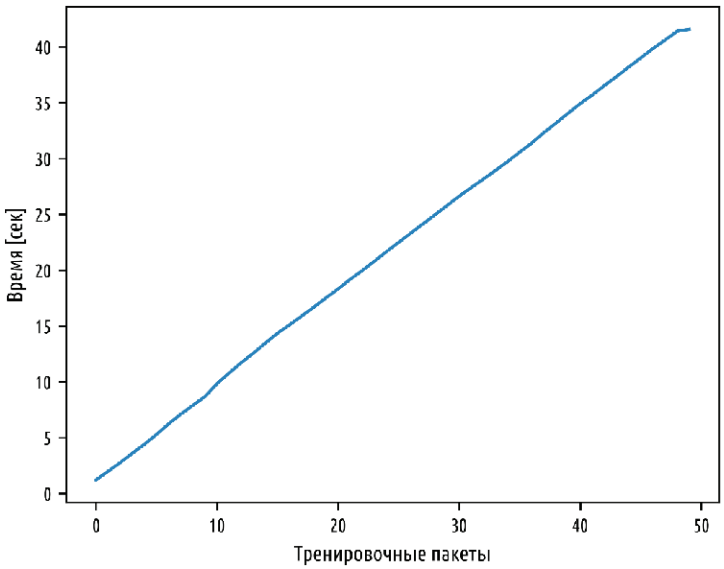
```
1 loop, best of 1: 3.72 s per loop (1.76 s)
Размер набора данных: 200000
K: 4
1 loop, best of 1: 11.7 s per loop (6.35 s)
K: 8
1 loop, best of 1: 19.7 s per loop (10.5 s)
K: 12
1 loop, best of 1: 30.7 s per loop (17.7 s)
Размер набора данных: 500000
K: 4
1 loop, best of 1: 27 s per loop (13.4 s)
K: 8
1 loop, best of 1: 1min 8s per loop (48.6 s)
K: 12
1 loop, best of 1: 1min 39s per loop (1min 5s)
```



Представляется очевидным, что с учетом графика и фактического хронометража время тренировки увеличивается линейно вместе с K и размером тренировочного набора, но для крупных значений K и размеров тренировочного набора такая связь становится нелинейной. Исчерпывающий поиск с полным тренировочным набором для многих значений K не выглядит как масштабируемая задача.

К счастью, онлайновая версия алгоритма K-средних на основе мини-пакетов в Scikit-learn уже реализована и называется `MiniBatchKMeans`. Давайте испытаем ее на самом медленном случае из предыдущего графика, т. е. с $K = 12$. Тренировка классического алгоритма K-средних на 500 000 выборках (примерно 20% от полного набора данных) заняла больше минуты; теперь посмотрим на производительность онлайновой мини-пакетной версии, установив размер пакета в 1000 и импортировав из набора данных порции в объеме 50 000 наблюдений. В качестве конечного результата мы построим график времени тренировки против числа порций, уже пропущенных через тренировочную фазу:

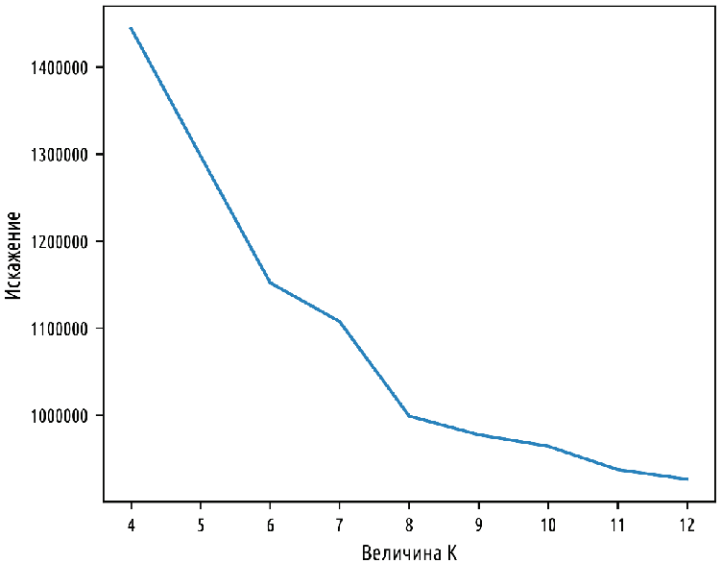
```
In:
import time
from sklearn.cluster import MiniBatchKMeans

cls = MiniBatchKMeans(12, batch_size=1000, random_state=101)
ts = []

tik = time.time()
for chunk in pd.read_csv(census_csv_file, chunksize=50000):
    cls.partial_fit(chunk.drop('caseid', axis=1))
    ts.append(time.time()-tik)

plt.plot(range(len(ts)), ts)
plt.xlabel(u'Тренировочные пакеты')
plt.ylabel(u'Время [сек]')
plt.show()
```



Время тренировки линейно для каждой порции, причем кластеризация выполняется на полном наборе данных из 2.5 млн наблюдений почти за 20 секунд. При такой реализации мы сможем выполнить полный поиск, чтобы выбрать оптимальную величину *K*, воспользовавшись для этого методом локтя на искажении. Давайте выполним исчерпывающий поиск по списку значений *K* от 4 до 12 и построим график искажения:

```
In:
Ks = list(range(4, 13))
ds = []

for K in Ks:
    print('K={0}'.format(K))
    cls = MiniBatchKMeans(K, batch_size=1000, random_state=101)

    for chunk in pd.read_csv(census_csv_file, chunksize=50000):
```

```
        cls.partial_fit(chunk.drop('caseid', axis=1))
    ds.append(cls.inertia_)

plt.plot(Ks, ds)
plt.xlabel(u'Величина K')
plt.ylabel(u'Искажение')
plt.show()
```



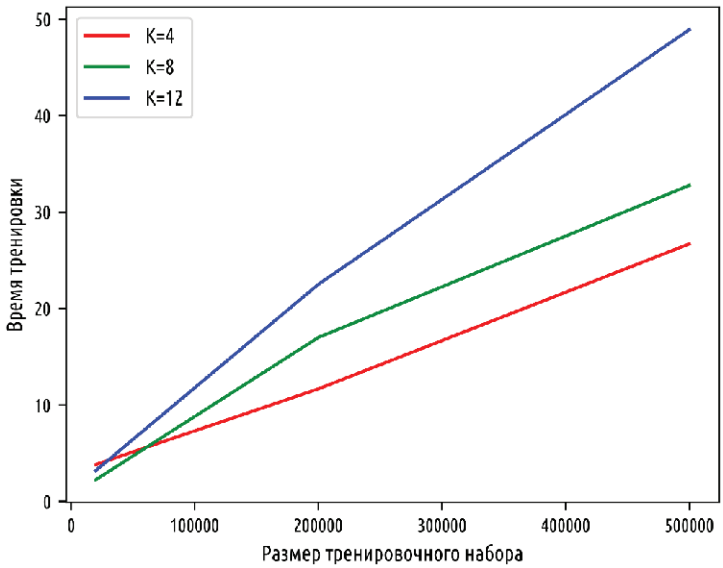
Из графика видно, что локоть, похоже, соответствует $K = 8$. Помимо этого значения, хотелось бы отметить, что благодаря пакетной реализации нам удалось выполнить массивную операцию на большом наборе данных менее чем пару минут; поэтому, если набор данных становится большим, то стоит отказаться от использования простого классического алгоритма K-средних.

Алгоритм K-средних в среде H2O

В этом разделе мы сравним реализацию алгоритма K-средних в среде H2O с его реализацией в библиотеке Scikit-learn. Если быть более точным, мы выполним мини-пакетный эксперимент с использованием имеющегося в среде H2O объекта `H2OKMeansEstimator` для алгоритма K-средних. Его настройка аналогична той, которая была показана в разделе «*Алгоритм PCA в среде H2O*», и данный эксперимент полностью совпадает с тем, который рассматривался в предыдущем разделе:

```
In:
import h2o
from h2o.estimators.kmeans import H2OKMeansEstimator
h2o.init(max_mem_size_GB=4)

def testH2O_kmeans(X, k):
    temp_file = tempfile.NamedTemporaryFile().name
    np.savetxt(temp_file, np.c_[X], delimiter=",")
```

```
    cls = H2OKMeansEstimator(k=k, standardize=True)
    blobdata = h2o.import_file(temp_file)

    tik = time.time()
    cls.train(x=range(blobdata.ncol), training_frame=blobdata)
    fit_time = time.time() - tik
    os.remove(temp_file)

    return fit_time

piece_of_dataset = \
    pd.read_csv(census_csv_file,
                iterator=True).get_chunk(500000).drop('caseid', axis=1).as_matrix()
time_results = {4: [], 8:[], 12:[]}
dataset_sizes = [20000, 200000, 500000]

for dataset_size in dataset_sizes:
    print('Размер набора данных: {0}'.format(dataset_size))
    X = piece_of_dataset[:dataset_size,:]

    for K in [4, 8, 12]:
        print('K: {0}'.format(K))
        fit_time = testH2O_kmeans(X, K)
        time_results[K].append(fit_time)

plt.plot(dataset_sizes, time_results[4], 'r', label='K=4')
plt.plot(dataset_sizes, time_results[8], 'g', label='K=8')
plt.plot(dataset_sizes, time_results[12], 'b', label='K=12')

plt.xlabel(u'Размер тренировочного набора')
plt.ylabel(u'Время тренировки')
plt.legend(loc=0)
plt.show()

h2o.cluster().shutdown(prompt=False)
```



Благодаря архитектуре платформы H2O ее реализация алгоритма K-средних очень быстра, масштабируема и способна выполнять кластеризацию наборов точечных данных объемом 500K менее чем за 30 секунд для всех отобранных значений *K*.

Алгоритм LDA

Алгоритм LDA – это одна из наиболее широко распространенных методик, которая применяется для анализа корпусов текстовых документов. Аббревиатура LDA означает **латентное размещение Дирихле** (latent Dirichlet allocation).

Аббревиатура LDA также используется для еще одного метода – линейного дискриминантного анализа (linear discriminant analysis), являющегося методом машинного обучения с учителем, предназначенным для задач классификации. Следует быть внимательным с тем, как данная аббревиатура используется, поскольку между этими двумя алгоритмами нет никакой связи.

Полное математическое объяснение алгоритма LDA потребовало бы знания вероятностного моделирования, которое выходит за рамки этой практической книги. Здесь же вместо этого мы предоставим наиболее важные интуитивные соображения, которые лежат в основе модели, и советы по ее практическому применению на крупном наборе данных.

Прежде всего латентное размещение Дирихле используется в разделе науки о данных под названием «глубинный анализ текстов» (text mining), где основное внимание уделяется созданию учеников для понимания естественного языка, например на основе текстовых примеров. Если быть точным, алгоритм LDA принадлежит к категории алгоритмов тематического моделирования, поскольку он пытается смоделировать темы, входящие в состав документа. В идеальном случае алгоритм LDA в состоянии понять, посвящен ли документ, к примеру, финансам, политике или религии. Однако, в отличие от классификатора, он также в состоянии оценивать присутствие тем в документе количественно. Например, возьмем роман Роулинг о Гарри Поттере. Классификатор смог бы идентифицировать его категорию (роман в стиле фэнтези); алгоритм LDA со своей стороны в состоянии понять, в каком количестве там присутствуют комедия, драма, мистерия, роман и приключение. Кроме того, алгоритм LDA не требует никаких меток; это метод машинного обучения без учителя, который изнутри строит выходные категории или тему и ее структуру (т. е. заданную набором слов, составляющих тему).

Во время обработки алгоритм LDA создает модель с темами в каждом документе и словами в каждой теме, моделируемыми как распределения Дирихле. Несмотря на то что его вычислительная сложность высокая, время обработки, необходимое для выработки стабильных результатов, не слишком продолжительное благодаря итеративной ядерной функции, похожей на метод Монте-Карло.

Модель LDA нетрудно понять: каждый документ моделируется как распределение тем, и каждая тема моделируется как распределение слов. В качестве распределений вероятностей приняты априорные распределения Дирихле (с разными параметрами, поскольку число слов в расчете на тему обычно разное, в отличие от числа тем на документ). Благодаря методу сэмплирования по Гиббсу каждое распределение не приходится отбирать непосредственно, а вместо этого итеративно

рассчитывается его точная аппроксимация. Аналогичные результаты могут быть получены при помощи вариационного байесовского метода, где аппроксимация генерируется вследствие подхода на основе максимизации ожидания.

Результирующая модель LDA генеративна (как в случае со скрытыми марковскими моделями, наивным Байесом и ограниченными машинами Больцмана), поэтому каждая переменная может моделироваться и наблюдаться.

Теперь давайте посмотрим, как этот алгоритм работает на реальном наборе данных – наборе данных о 20 группах новостей (20 Newsgroups). Он состоит из набора электронных сообщений, передававшихся в 20 группах новостей. Прежде всего мы его загрузим, удалив из отправленных электронных писем заголовки с адресом электронной почты, нижние колонтитулы и кавычки:

```
In:
from sklearn.datasets import fetch_20newsgroups

documents = fetch_20newsgroups(remove=('headers', 'footers', 'quotes'),
                               random_state=101).data
```

Проверим размер набора данных (т. е. количество документов) и распечатаем один из них, чтобы увидеть, из чего документ состоит в действительности:

```
In:
len(documents)

Out:
11314

In:
document_num = 9960
print documents[document_num]

Out:
Help!!!

I have an ADB graphicsd tablet which I want to connect to my
Quadra 950. Unfortunately, the 950 has only one ADB port and
it seems I would have to give up my mouse.

Please, can someone help me? I want to use the tablet as well as
the mouse (and the keyboard of course!!!).

Thanks in advance.
```

Согласно примеру, некто ищет помощь по поводу видеосокета на своем планшете.

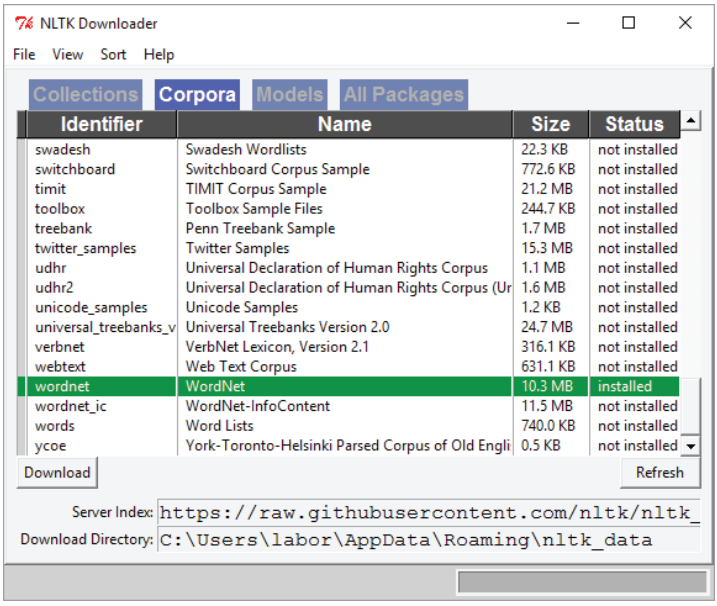
Теперь мы выполним импорт библиотек Python, необходимых для выполнения модели LDA. Библиотека Gensim – одна из лучших и, как вы убедитесь в конце настоящего раздела, тоже отлично масштабируется:

```
In:
import gensim
import nltk
from gensim.utils import simple_preprocess
from gensim.parsing.preprocessing import STOPWORDS
from nltk.stem import WordNetLemmatizer, SnowballStemmer

np.random.seed(101)
```

```
# появится окно загрузчика, где выбрать Corpora -> Wordnet -> Download
nltk.download()
```

В приведенном выше примере использована естественно-языковая библиотека NLTK, поставляемая с большим корпусом текстовых данных, миниатюрных грамматик, натренированных моделей и т. д., которая устанавливается стандартным образом менеджером библиотек pip. Полный их перечень размещен на http://nltk.org/nltk_data/. Чтобы установить эти данные, можно воспользоваться загрузчиком NLTK, как описано ниже.

Помимо отдельных пакетов данных, можно скачать всю коллекцию (используя параметр «all») либо только данные, которые требуются для примеров и упражнений из книги по NLTK, либо только корпус документов без грамматик и натренированных моделей (используя параметр «all-corpora»). В данном случае нужно в окне загрузчика **NLTK Downloader** в разделе **Corpora** выбрать из списка **Wordnet** и нажать **Download**.



В качестве первого этапа мы должны очистить текст. Для этого необходимо выполнить несколько шагов, которые типичны для любой естественно-языковой обработки текста.

1. Лексемизация, т. е. когда текст разбивается на предложения, а предложения разбиваются на слова. В заключение слова переводятся в строчные буквы. В этом месте удаляется пунктуация (и диакритические знаки).
2. Удаление слов, состоящих менее чем из трех символов. (На этом шаге удаляется большинство акронимов, эмограмм и союзов.)
3. Удаление слов из списка английских *стоп-слов*. Слова в этом списке очень распространены и не имеют никакой прогнозирующей способности (такие как the, an, so, then, have и т. д.).
4. Лексемы затем лемматизируются; слова в третьем лице переводятся в первое лицо, глаголы в прошедшем и будущем временах переводятся в настоящее (например, goes, went и gone становятся go).

5. Наконец, стемминг, или процедура выделения основы слова, удаляет словоизменение, сводя слово к его корню (например, shoes становится shoe).

В следующем фрагменте программного кода мы как раз это и сделаем: попытаемся очистить текст настолько, насколько это возможно, и выведем список слов, которые его составляют. В конце мы увидим, как эта работа изменит приведенный выше документ:

```
In:
lm = WordNetLemmatizer()
stemmer = SnowballStemmer('english')

def lem_stem(text):
    return stemmer.stem(lm.lemmatize(text, pos='v'))

def tokenize_lemmatize(text):
    return [lem_stem(token)
            for token in gensim.utils.simple_preprocess(text)
            if token not in gensim.parsing.preprocessing.STOPWORDS
            and len(token) > 3]

print(tokenize_lemmatize(documents[document_num]))

Out:
[u'help', u'graphicsd', u'tablet', u'want', u'connect', u'quadra', u'unfortun', u'port',
u'mous', u'help', u'want', u'tablet', u'mous', u'keyboard', u'cours', u'thank', u'advanc']
```

Сейчас в качестве следующего этапа давайте выполним шаги по очистке всех документов. После этого нам нужно создать словарь, содержащий количества вхождений слов в тренировочный набор. Благодаря библиотеке Gensim эта работа достаточно прямолинейна:

```
In:
processed_docs = [tokenize(doc) for doc in documents]
word_count_dict = gensim.corpora.Dictionary(processed_docs)
```

Теперь, поскольку мы хотим создать универсальное и быстрое решение, давайте удалим все очень редкие и очень общие слова. Например, мы можем отфильтровать все слова, которые встречаются меньше 20 раз (всего) и не более, чем в 20% документов:

```
In:
word_count_dict.filter_extremes(no_below=20, no_above=0.2)
```

На следующем этапе, располагая таким сокращенным набором слов, мы создадим модель мешка слов для каждого документа, т. е. по каждому документу мы создаем словарь, который предоставляет сведения о том, сколько слов используется и сколько раз они встречаются в документе:

```
In:
bag_of_words_corpus = [word_count_dict.doc2bow(pdoc)
                       for pdoc in processed_docs]
```

В качестве примера давайте взглянем на модель мешка слов для предыдущего документа:

```
In:
bow_doc1 = bag_of_words_corpus[document_num]
```

```
for i in range(len(bow_doc1)):
    print('Слово {0} ("{1}") появляется {2} раз[а]'
          .format(bow_doc1[i][0],
                  word_count_dict[bow_doc1[i][0]],
                  bow_doc1[i][1]))
```

```
Out:
Слово 179 ("want") появляется 2 раз[а]
Слово 250 ("keyboard") появляется 1 раз[а]
Слово 832 ("unfortun") появляется 1 раз[а]
Слово 1036 ("port") появляется 1 раз[а]
Слово 1142 ("help") появляется 2 раз[а]
Слово 1542 ("quadra") появляется 1 раз[а]
Слово 2001 ("advanc") появляется 1 раз[а]
Слово 2121 ("cours") появляется 1 раз[а]
Слово 2391 ("thank") появляется 1 раз[а]
Слово 2898 ("mous") появляется 2 раз[а]
Слово 3313 ("connect") появляется 1 раз[а]
```

Сейчас мы подошли к базовой части алгоритма: выполнению модели LDA. Что касается нашего решения, давайте запросим 12 тем (имеются 20 разных новостных рассылок, но некоторые из них похожие):

```
In:
# Алгоритм LDA в моноядерном режиме
lda_model = gensim.models.LdaModel(bag_of_words_corpus,
                                   num_topics=12,
                                   id2word=word_count_dict,
                                   iterations=200,
                                   passes=50)

# Алгоритм LDA в мультиядерном режиме
# (в этой конфигурации по умолчанию n_cores-1)
'''lda_model = gensim.models.LdaMulticore(bag_of_words_corpus,
                                      num_topics=12,
                                      id2word=word_count_dict,
                                      passes=50,
                                      workers=4)'''
```

Если вы получите сообщение с кодом ошибки, попробуйте выполнить обработку версии в монорежиме при помощи класса `gensim.models.LdaModel` вместо класса `gensim.models.LdaMulticore`.

Теперь распечатаем тематическую структуру, т. е. слова, появляющиеся в каждой теме, и их относительный вес:

```
In:
for idx, topic in lda_model.print_topics(-1):
    print('Тема:{0} Словарный состав:{1}\n'.format(idx, topic))
```

```
Out:
Тема:0 Словарный состав:0.031*"window" + 0.018*"file" + 0.016*"program" + 0.014*"imag" +
0.012*"version" + 0.011*"display" + 0.011*"graphic" + 0.009*"color" + 0.009*"applic" +
0.009*"softwar"

Тема:1 Словарный состав:0.010*"time" + 0.009*"bike" + 0.008*"good" + 0.008*"look" + 0.008*"go"
+ 0.007*"drive" + 0.006*"right" + 0.006*"car" + 0.006*"engin" + 0.006*"turn"
```

```
Тема:2 Словарный состав:0.041*"file" + 0.022*"entri" + 0.017*"section" + 0.015*"line"
+ 0.014*"program" + 0.014*"output" + 0.012*"return" + 0.010*"error" + 0.009*"read" +
0.009*"rule"

Тема:3 Словарный состав:0.024*"space" + 0.011*"launch" + 0.011*"year" + 0.009*"nasa" +
0.008*"orbit" + 0.008*"satellit" + 0.007*"earth" + 0.006*"mission" + 0.006*"develop" +
0.006*"cost"

Тема:4 Словарный состав:0.031*"game" + 0.022*"team" + 0.019*"play" + 0.018*"year" +
0.014*"player" + 0.012*"season" + 0.010*"hockey" + 0.010*"leagu" + 0.008*"score" +
0.007*"good"

Тема:5 Словарный состав:0.027*"christian" + 0.021*"jesus" + 0.014*"believ" + 0.012*"bibl"
+ 0.012*"church" + 0.012*"religion" + 0.011*"book" + 0.010*"faith" + 0.009*"word" +
0.009*"christ"

Тема:6 Словарный состав:0.028*"armenian" + 0.018*"say" + 0.017*"peopl" + 0.015*"kill" +
0.012*"go" + 0.011*"come" + 0.011*"turkish" + 0.008*"jew" + 0.007*"leav" + 0.007*"live"

Тема:7 Словарный состав:0.015*"state" + 0.013*"govern" + 0.013*"right" + 0.012*"peopl"
+ 0.009*"presid" + 0.006*"work" + 0.006*"american" + 0.006*"say" + 0.006*"israel" +
0.006*"countri"

Тема:8 Словарный состав:0.026*"drive" + 0.019*"card" + 0.014*"disk" + 0.014*"thank"
+ 0.013*"work" + 0.012*"problem" + 0.011*"price" + 0.011*"scsi" + 0.011*"control" +
0.010*"driver"

Тема:9 Словарный состав:0.017*"inform" + 0.014*"mail" + 0.013*"list" + 0.012*"post" +
0.011*"send" + 0.010*"univers" + 0.009*"address" + 0.009*"anonym" + 0.008*"public" +
0.008*"internet"

Тема:10 Словарный состав:0.016*"peopl" + 0.012*"thing" + 0.009*"time" + 0.009*"good" +
0.008*"say" + 0.008*"question" + 0.007*"mean" + 0.007*"want" + 0.007*"reason" + 0.007*"post"

Тема:11 Словарный состав:0.019*"chip" + 0.019*"encrypt" + 0.010*"key" + 0.010*"clipper"
+ 0.010*"secur" + 0.009*"number" + 0.009*"wire" + 0.009*"data" + 0.008*"devic" +
0.008*"algorithm"
```

К сожалению, алгоритм LDA не генерирует названия каждой темы; мы должны сделать это вручную самостоятельно, основываясь на нашей интерпретации результатов алгоритма. Тщательно исследовав тематическую структуру, мы можем назначить имена обнаруженным темам следующим образом:

Тема	Имя
0	Программное обеспечение
1	Приложения
2	Аргументация
3	Виды транспорта
4	Правительство
5	Религия
6	Поступки людей
7	Ближний Восток
8	Устройства ПК
9	Пространство
10	Игры
11	Накопители

Теперь давайте попытаемся понять, какие темы представлены в предыдущем документе и каковы их веса:

```
In:
for index, score in sorted(lda_model[bag_of_words_corpus[document_num]],
                           key=lambda tup: -1*tup[1]):
    print('Отметка: {}\t Тема: {}'.format(score,
                                          lda_model.print_topic(index, 10)))
```

```
Out:
Отметка: 0.488365059302 Тема: 0.026*"drive" + 0.019*"card" + 0.014*"disk" + 0.014*"thank"
+ 0.013*"work" + 0.012*"problem" + 0.011*"price" + 0.011*"scsi" + 0.011*"control" +
0.010*"driver"
Отметка: 0.45607837152  Тема: 0.031*"window" + 0.018*"file" + 0.016*"program" + 0.014*"imag"
+ 0.012*"version" + 0.011*"display" + 0.011*"graphic" + 0.009*"color" + 0.009*"applic" +
0.009*"softwar"
```

Самая высокая отметка связана с темой «*Устройства ПК*». Основываясь на наших предыдущих сведениях о наборах документов, по всей видимости, процесс извлечения тем прошел вполне хорошо.

Теперь оценим модель в целом. Перплексивность, т. е. степень неопределенности вероятностной модели (или ее логарифм), обеспечивает метрикой, помогающей понять, насколько хорошо модель LDA показала себя на тренировочном наборе данных:

```
In:
print('Логарифмическая перплексивность модели составляет {0}'
      .format(lda_model.log_perplexity(bag_of_words_corpus)))
```

```
Out:
Логарифмическая перплексивность модели составляет -7.2481550955
```

В данном случае перплексивность равна 2-7.298 и связана с (логарифмическим) правдоподобием, что модель LDA в состоянии генерировать документы в тестовом наборе при наличии распределения тем для этих документов. Чем ниже перплексивность, тем лучше модель, потому что это в основном означает, что модель может регенерировать текст достаточно хорошо.

Теперь попробуем использовать модель на ранее не встречавшемся документе. Для простоты документ содержит только следующие предложения: *Golf or tennis? Which is the best sport to play?* («Гольф или теннис? В какой из них лучше всего поиграть?»):

```
In:
unseen_document = 'Golf or tennis? Which is the best sport to play?'

bow_vector = \
    word_count_dict.doc2bow(tokenize_lemmatize(unseen_document))
for index, score in sorted(lda_model[bow_vector],
                           key=lambda tup: -1*tup[1]):
    print('Отметка: {0}\t Тема: {1}'
          .format(score, lda_model.print_topic(index, 5)))
```

```
Out:
Отметка: 0.816665932757 Тема: 0.031*"game" + 0.022*"team" + 0.019*"play" + 0.018*"year" +
0.014*"player"
```

```
Отметка: 0.0166668646637  Тема: 0.026*"drive" + 0.019*"card" + 0.014*"disk" + 0.014*"thank" +
0.013*"work"
...
```

Как и ожидалось, тема с самой высокой отметкой относится к «Играм», за которой следуют другие с отметкой относительно меньше.

Насколько хорошо алгоритм LDA масштабируется вместе с размером корпуса текстовых документов? К счастью, очень хорошо. Алгоритм является итеративным и позволяет выполнять онлайновое обучение, аналогично мини-пакетному. Ключом для онлайнового процесса является метод `update`, предлагаемый классом `LdaModel` (или `LdaMulticore`).

Давайте проведем его тестирование на подмножестве исходного корпуса текстовых документов, состоящем из первых 1000 документов, при этом мы будем обновлять модель LDA пакетами по 50, 100, 200 и 500 документов. Для каждого мини-пакетного обновления модели мы будем записывать время и строить график:

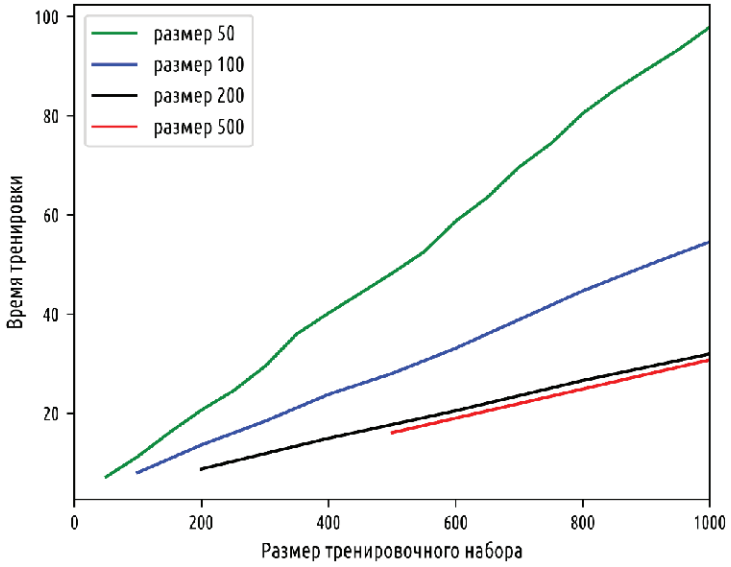
```
In:
small_corpus = bag_of_words_corpus[:1000]
batch_times = {}

for batch_size in [50, 100, 200, 500]:
    print('batch_size = {0}'.format(batch_size))
    tik0 = time.time()
    lda_model = gensim.models.LdaModel(num_topics=12,
                                       id2word=word_count_dict)
    batch_times[batch_size] = []

    for i in range(0, len(small_corpus), batch_size):
        lda_model.update(small_corpus[i:i+batch_size],
                         update_every=25, passes=1+500/batch_size)
        batch_times[batch_size].append(time.time() - tik0)

Out:
batch_size = 50
batch_size = 100
batch_size = 200
batch_size = 500
```

Отметим, что мы установили параметры `update_every` и `passes` в функции обновления модели. Это необходимо, чтобы заставить модель сходиться при каждой итерации и не возвращать несходящуюся модель. Обратите внимание, что значение 500 для размера пакета было выбрано эвристически; если задать это значение ниже, то вы получите большое количество предупреждений из библиотеки Gensim о несходимости модели.

Теперь отобразим результаты на графике:

```
In:
plt.plot(range(50, 1001, 50),
         batch_times[50], 'g', label=u'размер 50')
plt.plot(range(100, 1001, 100),
         batch_times[100], 'b', label=u'размер 100')
plt.plot(range(200, 1001, 200),
         batch_times[200], 'k', label=u'размер 200')
```

```
plt.plot(range(500, 1001, 500),
         batch_times[500], 'r', label=u'размер 500')

plt.xlabel(u'Размер тренировочного набора')
plt.ylabel(u'Время тренировки')
plt.xlim([0, 1000])
plt.legend(loc=0)
plt.show()
```



Чем пакет больше, тем быстрее проходит тренировка. (Напомним, что большие пакеты требуют при обновлении модели меньшего числа прохождений по данным.) С другой стороны, чем больше пакет, тем больше объем памяти, требуемый для хранения и обработки корпуса текстовых документов. Благодаря мини-пакетному методу обновления алгоритм LDA способен масштабироваться для обработки корпусов, состоящих из миллионов документов. По сути дела, предоставленная библиотекой Gensim реализация способна на домашнем компьютере отмасштабировать и обработать всю Википедию за всего пару часов. Если вы достаточно храбры, чтобы попытаться самому, то по ссылке можно получить полные инструкции по выполнению этой задачи, которые были предоставлены автором библиотеки: https://radimrehurek.com/gensim/wiki.html.

Масштабирование алгоритма LDA – оперативная память, CPU и машины

Библиотека Gensim очень гибка и создана для уплотнения больших текстовых корпусов; фактически данная библиотека способна масштабировать без какой-либо модификации или дополнительно скачиваемых материалов, используя для этого:

1) число модулей CPU, позволяя выполнять параллельную обработку на одиночном узле (при помощи классов, как уже было показано в первом примере);

2) число наблюдений, позволяя выполнять онлайновое обучение на основе мини-пакетов. Это можно реализовать при помощи метода update, который имеется в классах `LdaModel` и `LdaMulticore` (как показано в предыдущем примере);
3) выполнение на кластере, распределяя рабочую нагрузку по узлам в кластере благодаря Python'овской библиотеке Pyro4 и объектам `models.lda_dispatcher` (в качестве планировщика) и `models.lda_worker` (в качестве рабочего процесса), оба из которых предоставляются библиотекой Gensim.

Помимо классического алгоритма LDA, библиотека Gensim также предоставляет его иерархическую версию, именуемую **иерархическим процессом Дирихле** (hierarchical Dirichlet processing, HDP). Используя этот алгоритм, темы подчиняются многоуровневой структуре, позволяя пользователю лучше понять многосложные корпуса текстовых документов (т. е. где одни документы универсальны, а другие более конкретны по теме). Этот модуль является довольно новым и на конец 2015 г. не был масштабируем в той же степени, что и классический алгоритм LDA.

Резюме

В этой главе мы представили трех популярных учеников без учителя, которые способны масштабироваться для работы с большими данными. Первый из них, алгоритм PCA, способен сократить количество признаков, создавая такие, которые содержат бóльшую часть дисперсии (т. е. главные). Алгоритм кластеризации K-средних способен собирать в группы подобные точки и связывать их с центроидом. Мощный алгоритм LDA предназначен для тематического моделирования на текстовых данных, т. е. совместного моделирования распределения тем в каждом документе и слов, встречающихся в теме.

В следующей главе мы представим некоторые усовершенствованные и новейшие методы машинного обучения, которые пока не являются частью господствующей тенденции, однако они естественным образом великолепно подходят для малых наборов данных, а также для обработки крупномасштабного машинного обучения.

Глава 8

Распределенные среды – Hadoop и Spark

В этой главе мы представим новый способ обработки данных, осуществляя масштабирование горизонтально. До сих пор в центре нашего внимания прежде всего находилась обработка больших данных на автономной машине; здесь мы представим некоторые методы, которые работают на кластере из машин.

Если быть точным, мы сначала проиллюстрируем побудительные причины и обстоятельства, когда нам необходим кластер для обработки больших данных. Затем на нескольких примерах мы представим платформу Hadoop и все ее компоненты (HDFS, MapReduce и YARN), и, наконец, мы познакомимся с платформой Spark и ее Python'овским интерфейсом – pySpark.

От автономной машины к набору узлов

Объем хранения данных во всем мире увеличивается экспоненциально. В наше время уже стало обычным делом, когда исследователю-аналитику поступает запрос на обработку нескольких терабайт данных в день. Дело осложняется тем, что обычно данные поступают из большого числа различных неоднородных систем, при этом бизнес рассчитывает, что модель будет произведена в течение короткого времени.

Обработка больших данных поэтому не является просто вопросом размера, это на самом деле трехмерное явление. В сущности, согласно 3V-модели (от англ. volume, velocity и variety), оперирующие на больших данных системы могут быть классифицированы при помощи трех (ортогональных) критериев.

1. Первый критерий – это скорость, которую система достигает во время обработки данных. Хотя еще несколько лет назад скорость означала, как быстро система может обрабатывать пакет данных, в наше время она свидетельствует о том, может ли система обеспечивать непосредственные результаты на потоковых данных в реальном времени.
2. Второй критерий – объем, т. е. сколько имеется информации для обработки. Это может быть выражено числом строк, признаков либо чисто простым количеством байт. В потоковой передаче данных объем означает пропускную способность прибывающих в систему данных.
3. Последний критерий – разнообразие, т. е. тип источников данных. Несколько лет назад разнообразие было ограничено структурированными наборами данных; сегодня же данные могут быть структурированными (таблицы,

изображения и т. д.), полуструктурированными (JSON, XML и т. д.) и неструктурированными (веб-страницы, социальные данные и т. д.). Системы обработки больших данных обычно пытаются перерабатывать как можно больше соответствующих источников, перемешивая все виды источников.

Помимо этих критериев, в последние годы появились многие другие V, стремящиеся объяснить другие особенности больших данных. Некоторые из них следующие:

- правдивость (veracity, указывает на ненормальность, смещение и шум, которые содержатся в данных, и в конечном счете на их достоверность);
- волатильность (volatility, указывает на то, как долго данные могут использоваться для извлечения осмысленной информации);
- валидность (validity, корректность данных);
- стоимость (value, указывает на отдачу от вложений в данные).

В последние годы роль всех V существенно увеличилась; теперь многие компании осознали, что хранящиеся у них данные имеют огромную стоимость, которая может быть монетизирована, и они испытывают потребность в извлечении из нее информации. Техническая проблема переместилась в сторону обладания достаточным количеством накопителей и вычислительных мощностей, чтобы быть в состоянии извлекать значимые величины быстро, в крупном масштабе и с использованием различных входных потоков данных.

Нынешние компьютеры, даже новейшие и самые дорогие, имеют ограниченный объем дисковой и оперативной памяти и CPU. И задача обработки ими терабайт (или петабайт) информации в день с генерированием быстрой модели выглядит неподъемной. Кроме того, автономный сервер, который содержит и данные, и обрабатывающее программное обеспечение, нуждается в своей репликации; иначе он может стать единственной точкой отказа в системе.

Мир больших данных поэтому переместился в кластеры: они состоят из переменного числа *не очень-то дорогостоящих* узлов и находятся на высокоскоростном интернет-соединении. Обычно кластеры выделяют для хранения данных (большой жесткий диск, небольшой CPU и низкий объем памяти), другие же предназначаются для их обработки (мощный CPU, объем памяти от среднего до большого и маленький жесткий диск). Кроме того, если кластер настроен грамотно, то он способен гарантировать надежность (нет точек отказа) и высокую доступность.

Стоит отметить, что когда мы храним данные в распределенной среде (в виде кластера), мы также должны рассмотреть ограничение вследствие теоремы Брюера (CAP-теоремы – Consistency, Availability, Partition tolerance); система может гарантировать только два из следующих трех свойств:

- согласованность: все узлы в состоянии поставлять клиенту те же самые данные и в то же самое время;
- доступность: клиент, который запрашивает данные, всегда гарантированно получит отклик на успешные и неуспешные запросы;
- устойчивость к разделению: если сеть сбоит и не все узлы на связи, то система остается в рабочем состоянии.

Говоря конкретно, последствия теоремы CAP следующие:

- если отказаться от согласованности, то вы создадите среду, где данные распределены по всем узлам, и даже если сеть испытывает некоторые проблемы, система по-прежнему в состоянии обеспечить отклик на любой запрос,

хотя при этом не гарантируется, что отклик на тот же самый запрос будет тем же самым (он может оказаться противоречивым). Типичными примерами такой конфигурации являются СУБД DynamoDB, CouchDB и Cassandra;

- если отказаться от доступности, то вы создадите распределенную систему, которая может не дать отклика на запрос. Примерами такого класса являются СУБД с распределенным кэшем, в частности Redis, MongoDb и MemcacheDb;
- наконец, если отказаться от устойчивости к разделению, то вы окажетесь в жесткой схеме реляционных СУБД, которые не позволяют сети быть разделенной. Данная категория включает в себя MySQL, Oracle и SQLServer.

Зачем нужна распределенная платформа?

Самый простой способ создать кластер состоит в том, чтобы несколько узлов использовать в качестве узлов хранения данных, а остальные – в качестве узлов обработки. Такая конфигурация выглядит очень простой, так как, для того чтобы с ней справиться, вовсе не нужна сложная платформа. По сути дела, многие небольшие кластеры построены именно таким образом: пара серверов занята хранением данных (плюс их копий), другая же группа их обрабатывает.

Несмотря на то что это решение может показаться отличным, по многим причинам оно используется нечасто:

- оно работает только для чрезвычайно параллельных алгоритмов. Если алгоритм требует общей зоны памяти, разделяемой обрабатывающими серверами, то этот подход использоваться не может;
- если один или несколько узлов хранения выходят из строя, то не гарантируется, что данные останутся согласованными (представьте ситуацию, где узел и его копия выходят из строя одновременно, или где узел выходит из строя сразу после выполнения операции записи, которая еще не была реплицирована);
- если обрабатывающий узел выходит из строя, то мы не в состоянии отследить процесс, который он выполнял, что затрудняет возобновление обработки на другом узле;
- если сеть регулярно испытывает отказы, то очень трудно предсказать эту ситуацию, после того как она вернется в нормальное состояние.

Давайте вычислим, какова вероятность отказа узла. Является ли она действительно такой редкой, что мы можем ее отбросить? Или же эта вероятность является чем-то более конкретным, и мы будем всегда ее учитывать? Решение простое: возьмем кластер со 100 узлами, где каждый узел имеет вероятность отказа 1% (суммарно определяемую выходом из строя аппаратного и программного обеспечений) в первый год. Какова вероятность, что все эти 100 узлов переживут первый год? Исходя из гипотезы, что каждый сервер независим (т. е. каждый узел может выйти из строя независимо от всех остальных), эту вероятность получают простым произведением:

$$\begin{aligned} P(\text{кластер} = \text{ok}) &= P(\text{узел}_1 = \text{ok}, \text{узел}_2 = \text{ok}, \ldots, \text{узел}_{100} = \text{ok}) \\ &= (1 - P(\mathit{fail}))^{100} \\ &= 37\%. \end{aligned}$$

Полученный результат поначалу очень удивляет, но он объясняет, почему в прошедшее десятилетие сообщество специалистов, работающих с большими данными, сделало большой акцент на этой проблеме и разработало много решений для управления кластерами. Исходя из результатов формулы, представляется, что событие выхода из строя (или даже несколько таких событий) имеет большую вероятность, и этот факт требует, чтобы такое происшествие было предусмотрено заранее и обработано должным образом для обеспечения непрерывности операций на данных. Более того, при использовании дешевых аппаратных средств или более крупного кластера сбой, по меньшей мере, одного узла выглядит почти бесспорным.

Важный урок здесь состоит в том, что, раз вы внедряете обработку больших данных на предприятии, вы обязаны принять достаточно контрмер против отказов узлов; это норма, а не исключение, и к ним следует подходить грамотно, чтобы обеспечить непрерывность операций.

Сегодня в подавляющем большинстве кластерных платформ используется подход *divide et impera* (разделяй и властвуй):

- существуют модули*, специально предназначенные* для узлов данных, и несколько других, специально предназначенных для узлов обработки данных (так называемых рабочих узлов);
- данные тиражируются по всем узлам данных, при этом один узел является ведущим, гарантируя успешность операций записи и чтения;
- шаги обработки разделены между рабочими узлами. Они не имеют разделяемых между собой состояний (если только они не хранятся в узлах данных), и их ведущий узел гарантирует, что все задачи выполняются положительно и в нужном порядке.

Чуть позже в этой главе мы представим платформу Apache Hadoop; хотя сегодня это зрелая система управления кластерами, она по-прежнему опирается на прочный фундамент. Прежде чем перейти к ознакомлению с платформой Apache Hadoop, давайте настроим на наших машинах правильную рабочую среду.

Настройка виртуальной машины

Поднятие кластера – это долгая и трудная работа; старшие инженеры в области больших данных зарабатывают свои (высокие) зарплаты, не просто скачивая и выполняя двоичное приложение, а профессионально и тщательно настраивая менеджер кластера к требующейся рабочей среде. Это сложная и трудоемкая операция, которая может потребовать много времени, и если результаты будут ниже ожиданий, то весь бизнес (включая исследователей-аналитиков и разработчиков программного обеспечения) не сможет быть продуктивным. Инженеры данных должны знать все мельчайшие подробности относительно узлов, данных, выполняемых операций и сети, прежде чем начинать строить кластер. В результате обычно получается сбалансированный, настраиваемый, быстрый и надежный кластер, который может использоваться в течение многих лет всем техническим персоналом компании.

Что лучше: кластер с малым числом очень мощных узлов или кластер с большим числом менее мощных серверов? Ответ на этот вопрос следует давать только после индивидуальной оценки в каждом конкретном случае, и он сильно зависит от данных, обрабатывающих

алгоритмов, числа людей, имеющих нему доступ, скорости, на которой мы хотим получать результаты, общей себестоимости, устойчивой масштабируемости, скорости сети и многих других факторов. Проще говоря, совсем не легко решить, что лучше!

Поскольку настройка среды является трудоемкой задачей, мы, авторы, предпочитаем предоставить своим читателям образ виртуальной машины, содержащий все, в чем вы нуждаетесь, для того чтобы испытать некоторые операции на кластере. В следующих разделах вы научитесь настраивать на своей машине гостевую операционную систему, содержащую один узел кластера со всем программным обеспечением, которое вы найдете на реальном кластере.

Почему всего один узел? Поскольку платформа, которую мы использовали, не легковесная, мы решили остановиться на атомарной части кластера, гарантирующей, что среда, которую вы найдете в узле, точно такая же, что и та, которую вы найдете в реальной ситуации. Для того чтобы запустить виртуальную машину на компьютере, понадобятся два программных продукта: Virtualbox и Vagrant. Обе эти программы бесплатные и с открытым исходным кодом.

Виртуализатор VirtualBox

Программный продукт VirtualBox с открытым исходным кодом используется для виртуализации гостевых операционных систем по схеме «один ко многим» в Windows, macOS и хост-машинах Linux. С точки зрения пользователя виртуализированная машина выглядит как еще один компьютер, работающий в окне со всем своим функционалом.

Программа VirtualBox стала очень популярной из-за своей высокой производительности, простоты и чистого **графического интерфейса пользователя** (GUI). Запуск, останов, импорт и завершение работы виртуальной машины в VirtualBox – это лишь дело нескольких нажатий.

С технической стороны VirtualBox является гипервизором, который поддерживает создание двух и более **виртуальных машин** (VM) и управление ими, в том числе многие версии Windows, Linux и BSD-подобные дистрибутивы. Машина, на которой выполняется VirtualBox, называется *хостом*, в то время как виртуализированные машины называются *гостями*. Отметим, что между хостом и гостями нет никаких ограничений; например, хост Windows может выполнять Windows (той же версии, предыдущей или самой последней), а также любую Linux и BSD-дистрибутив, которые совместимы с VirtualBox.

Виртуализатор Virtualbox часто используется для выполнения системнозависимого программного обеспечения; некоторое программное обеспечение выполняется только под Windows или только в определенной версии Windows, некоторые доступны только в Linux и т. д. Еще одно применение состоит в моделировании нового функционала в клонированной эксплуатационной среде; прежде чем опробовать модификации в живой (эксплуатационной) среде, разработчики программного обеспечения обычно тестируют его на клоне, подобном тому, который выполняется в VirtualBox. Благодаря гостевой изоляции от хоста, если в госте что-то идет не так, как надо (даже форматирование жесткого диска), это не повлияет на хост. Чтобы вернуть его назад, просто клонируйте свою машину, прежде чем сделать что-либо опасное, и вы всегда успеете его восстановить.

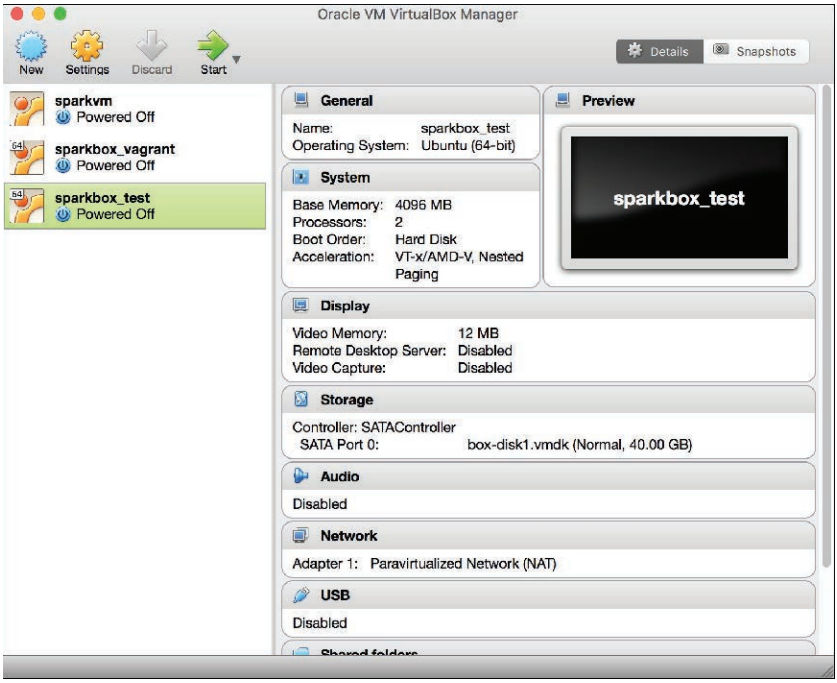
Для тех, кто хочет начать с нуля, программа VirtualBox поддерживает виртуальные жесткие диски (включая жесткие диски, CD-, DVD- и гибкие диски); это во

многом упрощает инсталляцию новой ОС. Например, если вы хотите установить классическую версию Linux Ubuntu 14.04, то сначала скачайте файл `.iso`. Вместо того чтобы записывать его на CD/DVD, можно просто добавить его в VirtualBox как виртуальный диск. Затем благодаря простому пошаговому интерфейсу можно выбрать размер жесткого диска и компоненты гостевой машины (RAM, число модулей CPU, видеопамять и сетевое соединение). Работая с реальной BIOS, можно выбрать порядок начальной загрузки: назначая CD/DVD-диску более высокий приоритет, можно запустить процесс инсталляции Ubuntu, как только вы включаете гостя.

Теперь давайте скачаем программу VirtualBox, при этом следует помнить о необходимости выбрать правильную версию, соответствующую вашей операционной системе.

Для того чтобы установить этот программный продукт на компьютер, следуйте инструкциям на https://www.virtualbox.org/wiki/Downloads.

Во время написания настоящей главы последней была версия 5.1 (5.1.22 на момент публикации перевода). После инсталляции графический интерфейс будет выглядеть, как показано на приведенном ниже снимке экрана:



Мы настоятельно рекомендуем разобраться в том, как поднимать гостевую машину на своем компьютере. Каждая гостевая машина будет появляться в левой стороне окна. (На снимке вы видите, что на нашем компьютере имеются три остановленных гостя.) Если нажать на каждом из них, то на правой стороне появится подробное описание виртуализированных аппаратных средств. На демонстраци-

онном снимке, если включить виртуальную машину `sparkbox_test` (выделена слева), она будет запущена на виртуальном компьютере, аппаратные средства которого состоят из 4 Гб RAM, двух процессоров, жесткого диска на 40 Гб и видеокарты с 12 Мб RAM, подключенных к сети через NAT.

Конфигуратор Vagrant

Программный продукт Vagrant (версии 1.9.5 на момент публикации) конфигурирует виртуальные среды на высоком уровне. Ключевым элементом программы Vagrant является возможность по созданию сценариев, которые часто используются для программного и автоматического создания конкретных виртуальных сред. Для построения и конфигурирования виртуальной машины программа Vagrant использует виртуализатор VirtualBox (а также другие виртуализаторы).

Чтобы проинсталлировать этот программный продукт, следуйте инструкциям, которые приведены на https://www.vagrantup.com/downloads.html.

Использование виртуальной машины

После установки программного обеспечения Vagrant и VirtualBox все готово к запуску узла кластерной среды. Создайте пустой каталог и вставьте в новый файл с именем `Vagrantfile` следующие ниже команды Vagrant:

```
Vagrant.configure("2") do |config|
    config.vm.box = "sparkpy/sparkbox_test_1"
    config.vm.hostname = "sparkbox"
    config.ssh.insert_key = false

    # Менеджер ресурсов Hadoop
    config.vm.network :forwarded_port, guest: 8088, host: 8088,
    auto_correct: true

    # Узел имен Hadoop
    config.vm.network :forwarded_port, guest: 50070, host: 50070,
    auto_correct: true

    # Узел данных Hadoop
    config.vm.network :forwarded_port, guest: 50075, host: 50075,
    auto_correct: true

# Блокноты Ipython (yarn и автономные)
    config.vm.network :forwarded_port, guest: 8888, host: 8888,
    auto_correct: true

# Spark UI (автономный)
    config.vm.network :forwarded_port, guest: 4040, host: 4040,
    auto_correct: true
    config.vm.provider "virtualbox" do |v|
    v.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
    v.customize ["modifyvm", :id, "--natdnsproxy1", "on"]
    v.customize ["modifyvm", :id, "--nictype1", "virtio"]
    v.name = "sparkbox_test"
    v.memory = "4096"
    v.cpus = "2"
    end
end
```

Если смотреть сверху вниз, первые строки скачивают нужную виртуальную машину (которую мы, авторы, создали и закачали в репозиторий). Затем мы настраиваем несколько портов, которые будут направлены в гостевую машину; тем самым вы сможете иметь доступ к нескольким веб-службам виртуализированной машины. И в конце мы настраиваем аппаратные средства узла.

Конфигурация настроена для виртуальной машины с исключительным использованием 4 Гб RAM и двух ядер. Если ваша система не может удовлетворить этим требованиям, то поменяйте значения `v.memory` и `v.cpus` на те, которые подходят для вашей машины. Отметим, что некоторые из приведенных ниже примеров программного кода могут не работать, если настроенная вами конфигурация не отвечает требованиям.

Теперь откройте терминал и перейдите в каталог с файлом `Vagrantfile`, находясь в нем, запустите виртуальную машину при помощи следующей команды:

```
$ vagrant up
```

В первый раз эта команда займет достаточно продолжительное время, т. к. она будет скачивать (скачиваемые данные занимают почти 2 Гб) и строить соответствующую структуру виртуальной машины. В следующий раз она займет минимальное количество времени, поскольку больше ничего скачивать не потребуется

После включения в вашей локальной системе виртуальной машины к ней можно получить доступ следующим образом:

```
$ vagrant ssh
```

Эта команда имитирует SSH-доступ. В результате вы наконец окажетесь внутри виртуализированной машины.

На машинах Windows эта команда может не сработать и выдать ошибку из-за отсутствия исполнимого SSH. В такой ситуации скачайте и установите SSH-клиент для Windows, в частности Putty (http://www.putty.org/), Cygwin openssh (http://www.cygwin.com/) либо Openssh для Windows (http://sshwindows.sourceforge.net/). Системы Unix эта проблема не должна затрагивать.

В Windows запуск Vagrant может также быть осуществлен посредством Git. Для этого надо установить Git (http://git-scm.com/download/win). При этом в системную переменную Path должен быть добавлен путь к папке bin: `C:\Program Files\Git\usr\bin`. Затем открыть окно терминала (Стартовое меню -> cmd), нажав комбинацию клавиш **Shift**+**Enter**, чтобы войти в качестве Администратора, и набрать либо скопировать/вставить:

```
set PATH=%PATH%;C:\Program Files\Git\usr\bin
vagrant ssh
```

В качестве бонуса можно вместо командной строки Windows воспользоваться консолью Console (http://sourceforge.net/projects/console/), в которой есть возможность указывать необходимый путь PATH, чтобы впоследствии всегда открывать новый терминал с нужными настройками (конфигурируется в разделе **Options**).

Для того чтобы выключить виртуальную машину, сначала нужно из нее выйти. Находясь внутри виртуальной машины, просто примените команду `exit`, чтобы выйти из SSH-соединения, и затем закройте виртуальную машину:

```
$ vagrant halt
```

Виртуальная машина потребляет ресурсы. Не забудьте по окончании работы ее выключить при помощи команды `vagrant halt` из каталога, где она расположена.

Приведенная выше команда закрывает виртуальную машину точно так же, как это делается при закрытии сервера. Чтобы ее удалить вместе со всем ее содержимым, нужно применить команду `vagrant destroy`. Следует использовать ее осторожно: после уничтожения машины вы не сможете восстановить в ней файлы.

Ниже приведены инструкции по использованию блокнота Jupyter в виртуальной машине.

- Запустить `vagrant up` и `vagrant ssh` из папки, содержащей файл `Vagrantfile`. Теперь вы должны быть внутри виртуальной машины.
- Запустить сценарий:

```
vagrant@sparkbox:~$ ./start_hadoop.sh
```

- В этом месте запустите следующий ниже сценарий оболочки:

```
vagrant@sparkbox:~$ ./start_jupyter_yarn.sh
```

Откройте браузер на своей локальной машине и направьте его в `http://localhost:8888`.

Такой блокнот будет поддерживаться кластерным узлом. Чтобы выключить блокнот и виртуальную машину, выполните следующие шаги:

1) для закрытия консоли Jupyter нажмите **Ctrl**+**C** (и затем наберите **Y** для **Yes**);
2) закройте платформу Hadoop следующим образом:

```
vagrant@sparkbox:~$ ./stop_hadoop.sh
```

3) выйдите из виртуальной машины при помощи следующей команды:

```
vagrant@sparkbox:~$ exit
```

4) закройте виртуальную машину VirtualBox командой `vagrant halt`.

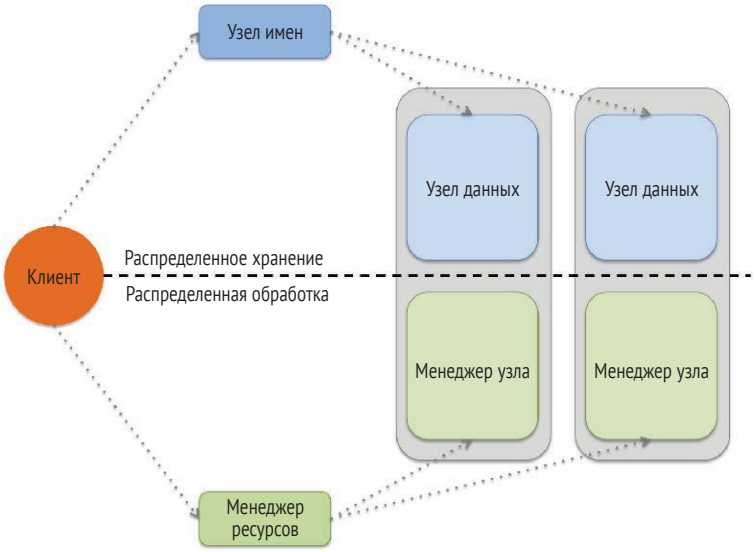
Экосистема Hadoop

Платформа Apache Hadoop – это очень популярная программная платформа для распределенного хранения данных и распределенной обработки на кластере. Ее сильная сторона заключается в ее цене (она бесплатная), гибкости (имеет открытый исходный код и, хотя она написана на Java, может использоваться другими языками программирования), масштабируемости (может обрабатывать кластеры, состоящие из тысяч узлов) и устойчивости (толчком для ее создания послужила публикация компанией Google работы о вычислительной концепции MapReduce, и она в ходу начиная с 2011 г.). Все это делает ее де-факто стандартом поддержания и обработки больших данных. Более того, ее функционал расширен большим числом других проектов от компании Apache Foundation.

Архитектура

Логически Hadoop состоит из двух частей: распределенного хранения данных (HDFS) и распределенной обработки (YARN и MapReduce). Несмотря на то что ее программная реализация очень сложная, общая архитектура довольно понятная. Клиент может получать доступ к хранению данных и их обработке посредством

двух специализированных модулей; затем они отвечают за распределение задания по всем рабочим узлам:



Все модули Hadoop работают как службы (или экземпляры), т. е. физический или виртуальный узел может выполнять многие из них. Как правило, в небольших кластерах все узлы выполняют распределенные вычисления и обрабатывающие службы; в больших же кластерах бывает лучше разделить эти две функциональности и детализировать узлы.

Рассмотрим подробно функциональности, предлагаемые этими двумя уровнями.

Распределенная файловая система HDFS

Распределенная файловая система Hadoop (Hadoop distributed file system, HDFS) – это отказоустойчивая распределенная файловая система, предназначенная для работы на товарном недорогом оборудовании и способная обрабатывать очень большие наборы данных (в порядке сотен петабайт до эксабайт). Несмотря на это, распределенная файловая система HDFS требует наличия высокоскоростного сетевого соединения для передачи данных через узлы, задержка не настолько же низкая, что и в классических файловых системах (она может быть в порядке нескольких секунд); поэтому HDFS разработана, имея в виду пакетную обработку данных и высокую пропускную способность. Каждый узел HDFS содержит часть данных этой файловой системы, и те же самые данные также реплицируются в других экземплярах, что гарантирует отказоустойчивость и доступ с высокой пропускной способностью.

Архитектура HDFS имеет тип «ведущий–ведомый». Если ведущее устройство (узел имен) дает сбой, то имеется вторичный/резервный, готовый взять на себя управление. Все другие экземпляры являются ведомыми (узлы данных); если один из них дает сбой, то это не представляет проблем, так как HDFS был разработан с учетом такой ситуации.

Узлы данных содержат блоки данных: каждый сохраненный в HDFS файл разбит на порции (или блоки), обычно по 64 Мб каждый блок, и затем распределяется и реплицируется в наборах узлов данных.

Узел имен просто хранит метаданные файлов из распределенной файловой системы; в нем хранятся не фактические данные, а соответствующие указания относительно того, как получить доступ к файлам в многочисленных узлах данных, которыми он управляет.

Клиент, делающий запрос на чтение файла, должен сначала связаться с узлом имен, который вернет таблицу, содержащую упорядоченный список блоков и их расположений (в узлах данных). В этом месте клиент должен отдельно связаться с узлами данных, скачав все блоки и восстановив файл (путем конкатенации блоков).

Для записи файла клиент вместо этого должен прежде всего связаться с узлом имен, который сначала решит, как поступить с запросом, обновит свои учетные данные и затем ответит клиенту упорядоченным списком узлов данных, куда можно записать каждый блок файла. Клиент затем установит контакт и закачает блоки в узлы данных, как указано в ответе узла имен.

Запросы к пространству имен (например, вывод содержимого каталога, создание папки и т. д.), напротив, полностью обрабатываются узлом имен путем реализации доступа к своим метаданным.

Узел имен, помимо этого, также отвечает за соответствующую обработку ситуации отказа узла данных (он помечается как мертвый, если тактовые пакеты не доходят), и заново тиражирует его данные по другим узлам.

Несмотря на то что эти операции долго и сложно реализуемы с обеспечением устойчивой работы, они абсолютно прозрачны для пользователя благодаря многим библиотекам и оболочке HDFS. Характер работы в распределенной файловой системе HDFS сильно напоминает то, что вы в настоящее время делаете в своей файловой системе, что является большим преимуществом Hadoop: сокрытие вычислительной сложности и предоставление пользователю возможности с легкостью ее использовать.

Теперь взглянем на оболочку HDFS и чуть позднее на библиотеку Python.

Используйте предыдущие инструкции относительно того, как на своем компьютере включить виртуальную машину и запустить блокнот Jupyter.

Теперь откройте блокнот; эта работа займет больше времени, чем обычно, поскольку каждый блокнот подключается к кластерной платформе Hadoop. Когда блокнот будет готов к использованию, вы увидите, что флажок с сообщением **Kernel starting, please wait…** (Запуск ядра, ожидайте…) в правом верхнем углу исчезнет.

Первая часть исходного кода касается оболочки распределенной файловой системы HDFS; поэтому все следующие команды могут выполняться в командой строке либо в оболочке виртуальной машины. Чтобы их выполнить в блокноте Jupyter, все они предваряются вопросительным знаком «`!`», который представляет собой быстрый способ выполнения команд bash в блокноте.

Общий знаменатель следующих ниже командных строк – это исполнение; мы будем всегда выполнять команду `hdfs`. Это основной интерфейс для получения доступа к системе HDFS и управления ею и основная команда оболочки HDFS.

Мы начинаем с отчета о состоянии HDFS. Чтобы получить подробности о **распределенной файловой системе** (dfs) и ее узлах данных, следует использовать подкоманду `dfsadmin`:

```
In:
!hdfs dfsadmin -report

Out:
Configured Capacity: 42241163264 (39.34 GB)
Present Capacity: 37569168058 (34.99 GB)
DFS Remaining: 37378433024 (34.81 GB)
DFS Used: 190735034 (181.90 MB)
DFS Used%: 0.51%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Live datanodes (1):

Name: 127.0.0.1:50010 (localhost)
Hostname: sparkbox
Decommission Status : Normal
Configured Capacity: 42241163264 (39.34 GB)
DFS Used: 190735034 (181.90 MB)
Non DFS Used: 4668290330 (4.35 GB)
DFS Remaining: 37380775936 (34.81 GB)
DFS Used%: 0.45%
DFS Remaining%: 88.49%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Tue Feb 09 19:41:17 UTC 2016
```

Подкоманда dfs позволяет использовать некоторые известные команды Unix для доступа и взаимодействия с распределенной файловой системой. Например, перечислить содержимое корневого каталога можно следующим образом:

```
In:
!hdfs dfs -ls /

Out:
Found 2 items
drwxr-xr-x - vagrant supergroup          0 2016-01-30 16:33 /spark
drwxr-xr-x - vagrant supergroup          0 2016-01-30 18:12 /user
```

Результат аналогичен использованию имеющейся в Linux команды `ls`, которая перечисляет полномочия, число ссылок, владеющих файлом пользователя, и группу, размер, метку времени последнего изменения и имя каждого файла или каталога.

Аналогично команде `df` можно вызвать аргумент `-df` для отображения объема доступного дискового пространства в HDFS. Опция `-h` сделает вывод более читаемым (вместо байт используются гигабайты и мегабайты):

```
In:
!hdfs dfs -df -h /

Out:
Filesystem              Size     Used      Available   Use%
hdfs://localhost:9000   39.3 G   181.9 M   34.8 G      0%
```

Аналогично команде `du` для отображения размера каждой папки, содержавшейся в корне, можно использовать аргумент `-du`. И опять-таки, опция `-h` произведет более читаемый результат:

```
In:
!hdfs dfs -du -h /

Out:
178.9 M    /spark
1.4 M    /user
```

До сих пор мы извлекали некоторую информацию из HDFS. Теперь давайте выполним несколько операций в распределенной файловой системе, которые ее изменят. Мы можем начать с создания папки, используя для этого опцию `-mkdir`, после которой следует имя. Отметим, что данная операция может не дать результата, если каталог уже существует (точно так же, как в Linux с командой `mkdir`):

```
In:
!hdfs dfs -mkdir /datasets
```

Теперь передадим несколько файлов из жесткого диска узла в распределенную файловую систему. В созданной нами виртуальной машине в каталоге `../datasets` уже существует текстовый файл; давайте скачаем текстовый файл из Интернета и переместим оба файла в каталог HDFS, который мы создали предыдущей командой:

```
In:
!wget -q http://www.gutenberg.org/cache/epub/100/pg100.txt \
      -O ../datasets/shakespeare_all.txt

!hdfs dfs -put ../datasets/shakespeare_all.txt \
/datasets/shakespeare_all.txt

!hdfs dfs -put ../datasets/hadoop_git_readme.txt \
/datasets/hadoop_git_readme.txt
```

Импорт прошел успешно? Да, ошибок не было. Однако, чтобы избавиться от каких-либо сомнений, давайте выведем содержимое каталога/наборов данных распределенной файловой системы HDFS на экран, и мы увидим эти два файла:

```
In:
!hdfs dfs -ls /datasets

Out:
Found 2 items
-rw-r--r-- 1 vagrant supergroup          1365 2016-01-31 12:41 /
datasets/hadoop_git_readme.txt
-rw-r--r-- 1 vagrant supergroup       5589889 2016-01-31 12:41 /
datasets/shakespeare_all.txt
```

Для конкатенации нескольких файлов в стандартном потоке вывода можно применить аргумент `-cat`. В следующем ниже фрагменте кода мы подсчитаем количество символов новой строки в текстовом файле. Отметим, что первая команда передается по конвейеру («|») в следующую команду, которая действует на локальной машине:

```
In:
!hdfs dfs -cat /datasets/hadoop_git_readme.txt | wc -l

Out:
30
```

В сущности, аргументом `-cat` можно связать несколько файлов из локальной машины и из распределенной файловой системы HDFS. Чтобы в этом убедиться, теперь давайте подсчитаем количество символов новых строк, когда файл, который хранится в HDFS, конкатенируется к тому же самому файлу, который хранится на локальной машине. Чтобы избежать недоразумений, можно применить **универсальный идентификатор ресурса** (URI) по схеме: `hdfs:` при обращении к файлам в HDFS и `file:` при обращении к локальным файлам.

```
In:
!hdfs dfs -cat \
    hdfs:///datasets/hadoop_git_readme.txt \
    file:///home/vagrant/datasets/hadoop_git_readme.txt | wc -l

Out:
60
```

Для копирования в HDFS применяется аргумент `-cp`:

```
In:
!hdfs dfs -cp /datasets/hadoop_git_readme.txt \
/datasets/copy_hadoop_git_readme.txt
```

Для удаления файла (или каталогов с правильной опцией) используется аргумент `-rm`. В этом фрагменте кода мы удаляем файл, который мы только что создали предыдущей командой. Отметим, что распределенная файловая система HDFS имеет thrash-механизм; следовательно, удаленный файл в действительности не удаляется из HDFS, а просто перемещается в специальный каталог:

```
In:
!hdfs dfs -rm /datasets/copy_hadoop_git_readme.txt

Out:
16/02/09 21:41:44 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval
= 0 minutes, Emptier interval = 0 minutes.

Deleted /datasets/copy_hadoop_git_readme.txt
```

Для очистки удаленных данных предназначена следующая команда:

```
In:
!hdfs dfs -expunge

Out:
16/02/09 21:41:44 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval
= 0 minutes, Emptier interval = 0 minutes.
```

Чтобы передать файл из распределенной файловой системы HDFS на локальную машину, используется аргумент `-get`:

```
In:
!hdfs dfs -get /datasets/hadoop_git_readme.txt \
/tmp/hadoop_git_readme.txt
```

Чтобы взглянуть на хранящийся в HDFS файл, используется аргумент `-tail`. Отметим, что функция head в HDFS отсутствует, поскольку данная функция может быть выполнена при помощи команды `cat`, и ее результат затем передается по конвейеру в локальную команду head. Что касается функции `tail`, то оболочка HDFS просто отображает последний килобайт данных:

```
In:
!hdfs dfs -tail /datasets/hadoop_git_readme.txt
```

```
Out:
ntry, of encryption software. BEFORE using any encryption software, please check your
country's laws, regulations and policies concerning the import, possession, or use, and re-
export of encryption software, to see if this is permitted. See <http://www.wassenaar.org/>
for more information.
...
```

Команда `hdfs` является основной точкой входа для распределенной файловой системы HDFS, но она – медленная, а вызов системных команд из Python и чтение назад выходных данных весьма утомительны. Для этой цели существует библиотека Python под названием Snakebite, которая служит клиентской оберткой для многих операций распределенной файловой системы Hadoop. К сожалению, данная библиотека не настолько полная, как оболочка HDFS, и привязана к узлу имен. Чтобы ее установить на локальную машину, просто используйте команду `pip install snakebite`.

Для инстанцирования клиентского объекта мы должны предоставить IP (или его псевдоним) и порт узла имен. Наша виртуальная машина работает на порту 9000:

```
In:
from snakebite.client import Client
client = Client("localhost", 9000)
```

Чтобы распечатать информацию о HDFS, клиентский объект имеет метод `serverdefaults`:

```
In:
client.serverdefaults()
```

```
Out:
{'blockSize': 134217728L,
 'bytesPerChecksum': 512,
 'checksumType': 2,
 'encryptDataTransfer': False,
 'fileBufferSize': 4096,
 'replication': 1,
 'trashInterval': 0L,
 'writePacketSize': 65536}
```

Чтобы перечислить файлы и каталоги, находящиеся в корне, можно применить `ls`. Ее результатом является список словарей, один для каждого файла, содержащий, в частности, информацию о полномочиях, метке времени последнего изменения и т. д. В этом примере мы просто интересуемся путями (т. е. именами):

```
In:
for x in client.ls(['/']):
    print(x['path'])

Out:
/datasets
/spark
/user
```

В точности, как и в приведенном выше примере, клиент Snakebite имеет в распоряжении методы `du` (от disk usage для получения информации об использовании диска) и `df` (от disk free для получения информации о свободном пространстве). Отметим, что многие методы (подобно `du`) возвращают генераторы, т. е. чтобы их исполнить, необходимо их потребить (аналогично итератору или списку):

```
In:
client.df()

Out:
{'capacity': 42241163264L,
 'corrupt_blocks': 0L,
 'filesystem': 'hdfs://localhost:9000',
 'missing_blocks': 0L,
 'remaining': 37373218816L,
 'under_replicated': 0L,
 'used': 196237268L}

In:
list(client.du(["/"]))

Out:
[{'length': 5591254L, 'path': '/datasets'},
 {'length': 187548272L, 'path': '/spark'},
 {'length': 1449302L, 'path': '/user'}]
```

Что касается примера оболочки HDFS, попробуем теперь подсчитать символы новой строки в том же файле при помощи библиотеки Snakebite. Отметим, что ее метод `cat` возвращает генератор:

```
In:
for el in client.cat(['/datasets/hadoop_git_readme.txt']):
    print(el.next().count("\n"))

Out:
30
```

Давайте теперь удалим файл из распределенной файловой системы HDFS. И опять-таки, обращаем внимание, что метод `delete` возвращает генератор, и его выполнение никогда не бывает неуспешным, даже если попытаться удалить несуществующий каталог. По сути дела, библиотека Snakebite не поднимает исклю-

чения, а просто отправляет пользователю сообщение в результирующем словаре, что операция не удалась:

```
In:
client.delete(['/datasets/shakespeare_all.txt']).next()

Out:
{'path': '/datasets/shakespeare_all.txt', 'result': True}
```

Теперь скопируем файл из распределенной файловой системы HDFS в локальную файловую систему. Отметим, что результат является генератором, и, чтобы увидеть, была ли операция успешной, необходимо проверить выходной словарь:

```
In:
(client
.copyToLocal(['/datasets/hadoop_git_readme.txt'],
             '/tmp/hadoop_git_readme_2.txt')
.next())

Out:
{'error': '',
 'path': '/tmp/hadoop_git_readme_2.txt',
 'result': True,
 'source_path': '/datasets/hadoop_git_readme.txt'}
```

Наконец, создадим каталог и удалим все файлы, совпадающие со строкой:

```
In:
list(client.mkdir(['/datasets_2']))

Out:
[{'path': '/datasets_2', 'result': True}]

In:
client.delete(['/datasets*'], recurse=True).next()

Out:
[{'path': '/datasets', 'result': True},
 {'path': '/datasets_2', 'result': True}]
```

Почему нет примера программного кода для размещения файла в распределенной файловой системе HDFS и примера копирования файла из HDFS в другой файл? Весь этот функционал в библиотеке Snakebite пока еще не реализован. Для этих операций мы будем использовать оболочку HDFS посредством системных вызовов[1].

Вычислительная парадигма MapReduce

Модель программирования в вычислительной парадигме MapReduce была реализована еще в самых ранних версиях Hadoop. Это очень простая модель, предназначенная для обработки параллельными пакетами больших наборов данных

[1] Операция `put(источник, место_назначения)` для выгрузки локального файла в HDFS реализована в последних версиях библиотеки. См. документацию по библиотеке на https://snakebite.readthedocs.io/en/latest/genindex.html. – *Прим. перев.*

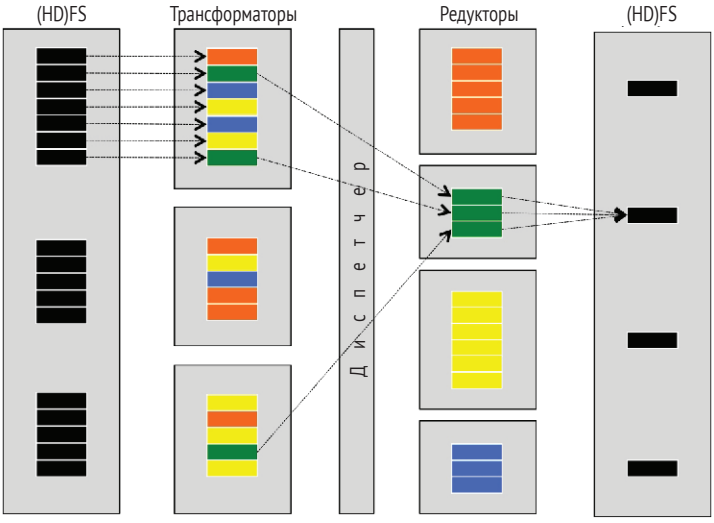
на распределенном кластере. Ядро MapReduce состоит из двух программируемых функций – трансформатора (mapper), который выполняет фильтрацию, и редуктора (reducer), который выполняет агрегацию, – а также диспетчера (shuffler), который перемещает объекты от трансформаторов в соответствующие редукторы.

Компания Google опубликовала работу по MapReduce в 2004 г., спустя несколько месяцев после получения патента на эту технологию.

Говоря конкретно, ниже приведены реализованные в Hadoop шаги вычислительной парадигмы MapReduce.

- Фрагментатор данных (data chunker). Данные считываются из файловой системы и разбиваются на порции. Порция является частью входного набора данных; обычно это блок фиксированного размера (например, блок HDFS, считанный из узла данных) либо другое более подходящее разбиение.
  Например, если мы хотим подсчитать число символов, слов и строк в текстовом файле, хорошим разбиением может быть строка текста.
- Трансформатор (mapper): из каждой порции генерируется серия пар «ключ-значение». Каждый экземпляр трансформатора применяет к разным порциям данных одинаковую функцию преобразования.
  Если продолжить с предыдущим примером, то на этом шаге для каждой строки генерируются три пары «ключ-значение» – одна содержит число символов в строке (ключом может быть строка *chars*, «символы»), другая содержит число слов (в данном случае ключ должен отличаться, скажем, это *words*, «слова»), и еще одна содержит число строк, которое всегда равно единице (в данном случае ключом может быть *lines*, «строки»).
- Диспетчер (или расфасовщик, shuffler): при наличии ключа и числа имеющихся редукторов диспетчер направляет все пары «ключ-значение» с одинаковым ключом в одинаковые редукторы. Как правило, данная операция сводится к хэшированию ключа, остаток от деления на число редукторов, что гарантирует достаточное количество ключей для каждого редуктора. Этот функционал не программируется пользователем и предоставляется платформой MapReduce.
- Редуктор (reducer): каждый редуктор получает все пары «ключ-значение» для конкретного набора ключей и производит ноль или более конечных результатов.
  В приведенном выше примере в редуктор прибывают все значения, связанные с ключом *words*; его работа – просто суммировать все значения. То же самое происходит и для остальных ключей, приводя к трем итоговым значениям: количеству символов, слов и строк. Отметим, что эти результаты могут находиться на разных редукторах.
- Выходной регистратор (output writer): результаты работы редукторов записываются в файловую систему (или в распределенную файловую систему HDFS). В конфигурации платформы Hadoop по умолчанию каждый редуктор пишет свой файл (`part-r-00000` – это выход первого редуктора, `part-r-00001` – выход второго и т. д.). Чтобы получить полный список выходных результатов в одном файле, необходимо их конкатенировать.

Эту работу можно визуально представить и понять следующим образом:



После шага преобразования имеется еще один дополнительный шаг, который может выполняться каждым экземпляром трансформатора, – объединитель (combiner). Он в основном предугадывает, если это возможно, шаг редукции на трансформаторе и часто используется для уменьшения объема информации для диспетчера, тем самым ускоряя процесс. В предыдущем примере, если трансформатор обрабатывает более одной строки входного файла, то во время (дополнительного) шага объединителя он может предварительно выполнить агрегирование результатов, выведя меньшее число пар «ключ-значение». Например, если трансформатор обрабатывает 100 строк текста в каждой порции, то зачем выводить 300 пар «ключ-значение» (100 для числа символов, 100 для слов и 100 для строк), когда всю информацию можно агрегировать в три значения? В этом в действительности состоит цель объединителя.

В реализации MapReduce, предоставляемой платформой Hadoop, операция диспетчеризации распределена, тем самым оптимизируя стоимость обмена данными, и можно выполнять более одного трансформатора и редуктора в расчете на узел, в полной мере задействуя имеющиеся на узлах аппаратные ресурсы. Кроме того, инфраструктура Hadoop обеспечивает избыточность и отказоустойчивость, т. е. одна и та же задача может быть назначена многочисленным рабочим процессам.

Теперь давайте посмотрим, как это работает. Несмотря на то что платформа Hadoop написана на Java, благодаря утилите Hadoop Streaming трансформаторы и редукторы могут исполняться в любой программной среде, включая Python. Для потоковой передачи содержимого утилита Hadoop Streaming использует конвейер и стандартные потоки вводы-вывода; поэтому трансформаторы и редукторы должны реализовывать функцию чтения из stdin и функцию записи пар «ключ-значение» в stdout.

Теперь включим виртуальную машину и откроем новый блокнот Jupyter. И в этом случае тоже сначала познакомимся с предоставляемым платформой

Hadoop способом выполнения задания MapReduce из командной строки, и только потом представим чистую библиотеку Python. Первый пример будет в точности, как мы описали ранее: счетчик числа символов, слов и строк в текстовом файле.

Сначала вставим наборы данных в распределенную файловую систему HDFS; мы будем использовать файл Hadoop Git (короткий текстовый файл readme, распространяемый вместе с Apache Hadoop) и полный текст всех сочинений Уильяма Шекспира, предоставленных проектом Gutenberg (несмотря на то что этот файл имеет объем всего 5 Мб, он содержит почти 125K строк). В первой ячейке мы очистим папку от предыдущего эксперимента, затем скачаем файл с библиографией Уильяма Шекспира в папку dataset и, наконец, поместим оба набора данных в HDFS:

```
In:
!hdfs dfs -mkdir -p /datasets
!wget -q http://www.gutenberg.org/cache/epub/100/pg100.txt \
       -O ../datasets/shakespeare_all.txt
!hdfs dfs -put -f \
       ../datasets/shakespeare_all.txt/datasets/shakespeare_all.txt
!hdfs dfs -put -f \
       ../datasets/hadoop_git_readme.txt/datasets/hadoop_git_readme.txt
!hdfs dfs -ls /datasets
```

Теперь давайте создадим исполняемые файлы Python, содержащие трансформатор и редуктор. Здесь мы воспользуемся очень грязным приемом: мы напишем файлы Python (и сделаем их исполняемыми), используя операцию записи из блокнота.

Как трансформатор, так и редуктор читают из стандартного потока stdin и пишут в стандартный поток stdout (при помощи простых команд печати). Если быть точным, трансформатор читает строки из stdin и распечатывает пары «ключ-значение» с количеством символов (кроме символа новой строки), количеством слов (разбивая строки по пробелу) и количеством строк, всегда равным единице. Редуктор, со своей стороны, суммирует значения для каждого ключа и распечатывает общую сумму:

```
In:
with open('mapper_hadoop.py', 'w') as fh:
    fh.write("""#!/usr/bin/env python

import sys

for line in sys.stdin:
    print "chars", len(line.rstrip('\\n'))
    print "words", len(line.split())
    print "lines", 1
    """)

with open('reducer_hadoop.py', 'w') as fh:
    fh.write("""#!/usr/bin/env python

import sys

counts = {"chars": 0, "words":0, "lines":0}

for line in sys.stdin:
```

```
    kv = line.rstrip().split()
    counts[kv[0]] += int(kv[1])

for k,v in counts.items():
    print k, v
    """)

In:
!chmod a+x *_hadoop.py
```

Чтобы увидеть, как этот сценарий работает, давайте испытаем его локально, не используя Hadoop. По сути дела, поскольку трансформаторы и редукторы читают и пишут в стандартный поток ввода-вывода, мы можем связать все операции в один конвейер. Отметим, что диспетчер можно заменить командой `sort -k1,1`, которая сортирует входные строки по первому полю (т. е. ключу):

```
In:
!cat ../datasets/hadoop_git_readme.txt | ./mapper_hadoop.py |  \
        sort -k1,1 | ./reducer_hadoop.py

Out:
chars 1335
lines 31
words 179
```

Теперь давайте воспользуемся реализацией вычислительной парадигмы MapReduce в среде Hadoop, чтобы получить тот же самый результат. В первую очередь мы должны создать пустой каталог в распределенной файловой системе HDFS, который будет хранить результаты. В данном случае мы создадим каталог `/tmp` и удалим все, что находится внутри одноименного каталога, что и каталог результата работы задания (Hadoop не будет работать, если выходной файл уже существует). Затем мы применим команду, необходимую для выполнения задания MapReduce. Данная команда включает следующее:

- использование возможности утилиты Hadoop Streaming (с указанием на jar-файл данной утилиты);
- трансформаторы и редукторы, которые мы хотим применить (опции `-mapper` и `-reducer`);
- файлы, которые мы хотим направить в каждый трансформатор, так как это локальные файлы (при помощи опции `-files`);
- входной файл (опция `-input`) и выходной каталог (опция `-output`).

```
In:
!hdfs dfs -mkdir -p /tmp
!hdfs dfs -rm -f -r /tmp/mr.out

!hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.6.4.jar \
        -files mapper_hadoop.py,reducer_hadoop.py \
        -mapper mapper_hadoop.py -reducer reducer_hadoop.py \
        -input /datasets/hadoop_git_readme.txt \
        -output /tmp/mr.out

Out:
...
17/06/17 06:13:28 INFO mapreduce.Job: Running job: job_1497616070323_0009
```

```
17/06/17 06:13:38 INFO mapreduce.Job: Job job_1497616070323_0009 running in uber mode : false
17/06/17 06:13:38 INFO mapreduce.Job:  map 0% reduce 0%
17/06/17 06:13:48 INFO mapreduce.Job:  map 50% reduce 0%
17/06/17 06:13:55 INFO mapreduce.Job:  map 100% reduce 0%
17/06/17 06:14:06 INFO mapreduce.Job:  map 100% reduce 100%
17/06/17 06:14:07 INFO mapreduce.Job: Job job_1497616070323_0009 completed successfully
...
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
...
17/06/17 06:14:07 INFO streaming.StreamJob: Output directory: /tmp/mr.out
```

Полученный результат весьма многословный; мы извлекли из него только три важных раздела. Первый показывает продвижение работы задания MapReduce; эта информация очень полезна для отслеживания и оценки времени, необходимого для завершения работы. Второй раздел показывает ошибки, которые, возможно, произошли во время работы задания, и последний раздел сообщает о выходном каталоге и метке времени завершения задания. Весь процесс на небольшом файле (из 30 строк) занял почти полминуты! Причины очень простые: во-первых, вычислительная парадигма MapReduce в среде Hadoop была реализована для устойчивой обработки больших данных и содержит много накладных расходов, и, во-вторых, идеальной для нее средой является кластер из мощных машин, а не виртуальная машина с 4 Гб RAM. С другой стороны, этот программный код без каких-либо изменений можно выполнять на наборах данных гораздо большего объема и в кластере из очень мощной машины.

Не будем судить по первым результатам. Сначала посмотрим на выходной каталог в распределенной файловой системе HDFS:

```
In:
!hdfs dfs -ls /tmp/mr.out

Out:
Found 2 items
-rw-r--r--   1 vagrant supergroup          0 2017-06-17 06:14 /tmp/mr.out/_SUCCESS
-rw-r--r--   1 vagrant supergroup         33 2017-06-17 06:14 /tmp/mr.out/part-00000
```

Там имеются два файла: первый, пустой, под названием `_SUCCESS`. Он указывает, что задание MapReduce закончило этап записи в каталог; второй называется `part-00000` и содержит фактические результаты (поскольку мы работаем на узле всего с одним редуктором). Прочитав этот файл, мы получим конечные результаты:

```
In:
!hdfs dfs -cat /tmp/mr.out/part-00000

Out:
chars 1335
lines 31
words 179
```

Как и ожидалось, они совпадают с конвейерной командной строкой, показанной ранее.

Несмотря на то что утилита Hadoop Streaming концептуально простая, она не самый лучший способ выполнения заданий Hadoop с использованием программного кода Python. Для этого на PyPy существует много других библиотек, и здесь мы представим одну из самых гибких и поддерживаемых общедоступных библиотек с открытым исходным кодом – **MrJob**. Она позволяет беспрепятственно выполнять задания на локальной машине, кластере Hadoop или тех же облачных кластерных средах, таких как Эластичный MapReduce компании Amazon; она объединяет весь исходный код в один автономный файл, даже если необходимы многочисленные шаги MapReduce (к примеру, в итеративных алгоритмах), и интерпретирует ошибки Hadoop в ходе выполнения исходного кода. Кроме того, данная библиотека очень легко устанавливается; чтобы библиотека MrJob имелась на локальной машине, просто примените команду `pip install mrjob`.

Хотя библиотека MrJob является замечательным образцом программного обеспечения, она не очень хорошо работает с блокнотами Jupyter, поскольку требует основной функции в качестве точки входа. Нам придется записать Python'овский программный код MapReduce в отдельный файл и затем выполнить его в командной строке.

Начнем с примера, который мы уже встречали много раз: подсчет символов, слов и строк в файле. Сначала напишем файл Python с использованием функционала библиотеки MrJob; трансформаторы и редукторы *обернуты* в подкласс `MRJob`. Входные данные не считываются из стандартного потока stdin, а передаются как аргумент функции, а конечные результаты не распечатываются, а выдаются в ленивом режиме (или возвращаются).

Благодаря библиотеке MrJob целая программа MapReduce становится всего несколькими строками программного кода:

```
In:
with open("MrJob_job1.py", "w") as fh:
    fh.write("""
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
    """)
```

Теперь давайте выполним этот программный код локально (с локальной версией набора данных). Библиотека MrJob, помимо выполнения шагов трансформатора и редуктора (в данном случае локально), распечатывает результат и очищает временный каталог:

```
In:
!python MrJob_job1.py ../datasets/hadoop_git_readme.txt

Out:
...
Streaming final output from /tmp/MrJob_job1.vagrant.20170617.061809.672548/output...
"chars" 1335
"lines" 31
"words" 179
Removing temp directory /tmp/MrJob_job1.vagrant.20170617.061809.672548...
```

Для выполнения того же самого процесса в Hadoop просто запустите этот же файл Python, но на этот раз вставив в командной строке опцию `-r hadoop`, и MrJob автоматически выполнит его при помощи реализованной в среде Hadoop вычислительной парадигмы MapReduce и с использованием распределенной файловой системы HDFS. В данном случае следует помнить, что необходимо указать путь `hdfs` входного файла:

```
In:
!python MrJob_job1.py -r hadoop hdfs:///datasets/hadoop_git_readme.txt

Out:
...
Job job_1497616070323_0010 running in uber mode : false
   map 0% reduce 0%
   map 50% reduce 0%
   map 100% reduce 0%
   map 100% reduce 100%
Job job_1497616070323_0010 completed successfully
...
Shuffle Errors
    BAD_ID=0
    CONNECTION=0
    IO_ERROR=0
    WRONG_LENGTH=0
    WRONG_MAP=0
    WRONG_REDUCE=0
Streaming final output from hdfs:///user/vagrant/tmp/mrjob/MrJob_job1.
vagrant.20170617.061919.506254/output...
"chars" 1335
"lines" 31
"words" 179
Removing HDFS temp directory hdfs:///user/vagrant/tmp/mrjob/MrJob_job1.vagra
nt.20170617.061919.506254...
Removing temp directory /tmp/MrJob_job1.vagrant.20170617.061919.506254...
```

Используя утилиту Hadoop Streaming из командной строки, вы увидите те же самые выходные данные, что и ранее, плюс результаты. В данном случае временный каталог распределенной файловой системы HDFS, используемый для хранения результатов, после завершения задания удаляется.

Теперь, чтобы убедиться в гибкости библиотеки MrJob, попробуем выполнить процесс, который требует больше одного шага MapReduce. Его выполнение из командной строки было бы очень трудно осуществить; фактически пришлось бы

выполнить первую итерацию MapReduce, проверить ошибки, прочитать результаты и затем выполнить вторую итерацию, снова проверить ошибки и наконец прочитать результаты. Такой процесс требует больших временных затрат и подвержен ошибкам. Благодаря библиотеке MrJob эта работа становится очень простой: в программном коде можно создать каскад операций MapReduce, где каждый выход является входом на следующий этап.

В качестве примера теперь отыщем наиболее распространенное слово в сочинениях Уильяма Шекспира (используя в качестве входных данных файл, состоящий из 125K строк). Эту операцию невозможно выполнить за один шаг MapReduce; требуются, по крайней мере, два таких шага. Мы реализуем очень простой алгоритм на основе двух итераций MapReduce:

- фрагментатор данных: входной файл разбивается по каждой строке, что в библиотеке MrJob тоже является операцией по умолчанию;
- этап 1 – трансформация: по каждому слову выдается кортеж «ключ-значение»; ключ – это слово строчными буквами, значение – это всегда 1;
- этап 1 – редукция: по каждому ключу (слово строчными буквами) суммируются все значения. Выход сообщит, сколько раз слово появляется в тексте;
- этап 2 – трансформация: во время этого шага мы переворачиваем кортежи «ключ-значение» и помещаем их как значения новой пары ключей. Чтобы назначить одному редуктору все кортежи, мы присваиваем каждому выходному кортежу одинаковый ключ *None*;
- этап 2 – редукция: просто отбрасываем единственный имеющийся ключ и извлекаем максимум из значений, в частности извлекаем максимум из всех кортежей (количество, слово).

```
In:
with open("MrJob_job2.py", "w") as fh:
    fh.write("""

from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")

class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   reducer=self.reducer_count_words),
            MRStep(mapper=self.mapper_word_count_one_key,
                   reducer=self.reducer_find_max_word)
        ]

    def mapper_get_words(self, _, line):
        # выдавать по одному слову в строке
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def reducer_count_words(self, word, counts):
        # отправить все пары (слово, число появлений) в тот же редуктор.
        yield (word, sum(counts))
```

```
    def mapper_word_count_one_key(self, word, counts):
        # отправить все кортежи в тот же редуктор
        yield None, (counts, word)

    def reducer_find_max_word(self, _, count_word_pairs):
        # каждый элемент пары word_count_pairs, т. е.
        # слово-частота - это кортеж (частота, слово)
        yield max(count_word_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()
""")
```

Далее можно выбрать, исполнить этот сценарий локально или на кластере Hadoop, получив тот же самый результат: наиболее распространенным словом в сочинениях Уильяма Шекспира является слово *the*, которое использовалось более 27K раз. В этом фрагменте программного кода требуется просто вывести результат, и поэтому мы запускаем задание с опцией `-quiet`:

```
In:
!python MrJob_job2.py --quiet ../datasets/shakespeare_all.txt

Out:
27801    "the"

In:
!python MrJob_job2.py -r hadoop --quiet hdfs:///datasets/shakespeare_all.txt

Out:
27801    "the"
```

Менеджер ресурсов YARN

С выходом Hadoop 2 (текущее ответвление по состоянию на 2016 г.) был введен новый слой поверх распределенной файловой системы HDFS, который обеспечивает выполнение многочисленных приложений. Например, MapReduce является одним из них (который предназначается для пакетной обработки данных). Этот слой называется **Yet Another Resource Negotiator** (YARN, или «еще один посредник между ресурсами и кластером»), и его задача заключается в управлении распределением ресурсов в кластере.

Менеджер YARN придерживается парадигмы «ведущий-ведомый» и состоит из двух служб: менеджера ресурсов и менеджера узла.

Менеджер ресурсов – это ведущий процесс, который отвечает за две задачи: планирование (выделение ресурсов) и управление приложениями (передача заданий и отслеживание их состояния). Каждый менеджер узла, согласно архитектуре, является ведомым и представляет собой предваряющую рабочий процесс (preworker) платформу, которая выполняет задачи и сообщает результаты менеджеру ресурсов.

Введенный вместе с Hadoop 2 слой YARN гарантирует следующее:

- мультиарендность, т. е. предоставление многочисленных механизмов, которые используют Hadoop;
- усовершенствованное использование кластера, так как выделение задач является динамическим и планируемым;

- продвинутая масштабируемость; YARN не предоставляет обрабатывающего алгоритма, он является просто менеджером ресурсов кластера;
- совместимость с MapReduce (более высокий уровень, чем Hadoop 1).

Платформа Spark

Ставшая за последние несколько лет очень популярной платформа Apache Spark является результатом эволюции платформы Hadoop. В противоположность Hadoop и ее Java-ориентированной и пакетно-центрированной конструкции, платформа Spark способна быстро и легко продуцировать итеративные алгоритмы. Кроме того, она имеет очень богатый комплект API для языков параллельного программирования и исходно поддерживает много разных типов обработки данных (машинное обучение, потоковая передача, анализ графов, SQL и т. д.).

Apache Spark – это кластерная платформа, разработанная для быстрой и общецелевой обработки больших данных. Одно из усовершенствований в скорости вызвано тем фактом, что после выполнения каждого задания данные остаются в оперативной памяти и не сохраняются в файловой системе (если вы, разумеется, не хотите обратного), подобно тому как это происходит с Hadoop, MapReduce и HDFS. Эта особенность делает итеративные задания (такие как алгоритм кластеризации K-средних) гораздо быстрее, поскольку задержка и пропускная способность, обеспечиваемые оперативной памятью, более производительны, чем у физического диска. Отсюда кластеры, на которых работает платформа Spark, нуждаются в большом объеме оперативной памяти для каждого узла.

Несмотря на то что платформа Spark была разработана на языке Scala (который работает в JVM, как Java), она имеет API для языков параллельного программирования, включая Java, Scala, Python и R. В этой книге мы сосредоточимся на Python.

Платформа Spark может работать двумя разными способами:

- в автономном режиме, при этом она работает на вашей локальной машине. В этом случае максимальная параллелизация определяется числом ядер локальной машины, и доступный объем оперативной памяти точно такой же, что и у локальной машины;
- в кластерном режиме, при этом она работает на кластере из многочисленных узлов, используя менеджер кластера, в частности YARN. В этом случае максимальная параллелизация определяется числом ядер во всех составляющих кластер узлах, и объем оперативной памяти является суммой объемов оперативной памяти каждого узла.

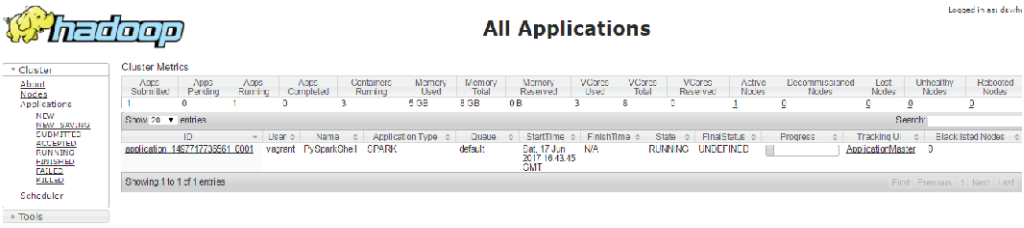
Библиотека pySpark

Чтобы воспользоваться функционалом платформы Spark (либо библиотеки pySpark, содержащей программные интерфейсы Python для платформы Spark), необходимо инстанцировать специальный объект под названием SparkContext. Он сообщает платформе Spark способ получения доступа к кластеру и содержит некоторые специфические для приложения параметры. В содержащемся в виртуальной машине блокноте Jupyter данная переменная уже имеется и называется `sc` (эта опция задана по умолчанию, когда блокнот Jupyter запущен); давайте посмотрим, что он содержит.

Сначала откройте новый блокнот Jupyter. Когда он будет готов к использованию, в первой ячейке наберите следующее:

```
In:
sc._conf.getAll()

Out:
[(u'spark.rdd.compress', u'True'),
 (u'spark.master', u'yarn-client'),
 (u'spark.serializer.objectStreamReset', u'100'),
 (u'spark.yarn.isPython', u'true'),
 (u'spark.submit.deployMode', u'client'),
 (u'spark.executor.cores', u'2'),
 (u'spark.app.name', u'PySparkShell')]
```

Эта переменная содержит многочисленную информацию, и самыми важными ее элементами являются параметр `spark.master`, в этом случае установлен в качестве клиента в YARN, `spark.executor.cores` установлен в 2 по числу модулей CPU виртуальной машины, и `spark.app.name` – имя приложения. Имя приложения пригодится, в частности, когда (YARN-)кластер будет использоваться совместно; перейдя по адресу `http:/127.0.0.1:8088`, можно проверить состояние приложения:



В платформе Spark используется модель данных под названием **отказоустойчивый распределенный набор данных** (resilient distributed dataset, RDD). Это распределенный набор элементов, который может обрабатываться параллельно. Набор элементов RDD может быть создан из существующего набора (список Python, например) или из внешнего набора данных, хранящегося в виде файла на локальной машине, в распределенной файловой системе HDFS или же в других источниках.

Теперь давайте создадим RDD-набор, содержащий целые числа от 0 до 9. Для этого можно воспользоваться методом `parallelize`, предоставляемым объектом `SparkContext`:

```
In:
numbers = range(10)
numbers_rdd = sc.parallelize(numbers)
numbers_rdd

Out:
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:423
```

Как видно, нельзя просто взять и распечатать содержимое RDD-набора, поскольку он разделен на многочисленные разделы (и распределен в кластере). Число разделов по умолчанию в два раза больше числа модулей CPU (т. е. 4 в предоставленной виртуальной машине), но его можно установить вручную при помощи второго аргумента метода `parallelize`.

Чтобы распечатать данные, содержащиеся в RDD-наборе, необходимо вызвать метод `collect`. Отметим, что этот метод, выполненный на кластере, собирает все данные на узле, поэтому узел должен располагать достаточным объемом памяти, чтобы они все уместились:

```
In:
numbers_rdd.collect()

Out:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Для оперативного просмотра по частям используйте метод `take`, указав, сколько элементов вы хотели бы увидеть. Отметим, что, поскольку мы имеем дело с распределенным набором данных, не гарантируется, что элементы в нем находятся в том же порядке, в каком их вставляли:

```
In:
numbers_rdd.take()

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

Чтобы прочесть текстовый файл, можно воспользоваться методом `textFile`, предоставляемым контекстом платформы Spark. Этот метод позволяет читать файлы HDFS и локальные файлы, при этом он разделяет текст по символу новой строки, поэтому первый элемент RDD-набора (применяя метод `first`) является первой строкой текстового файла. Отметим, что при использовании локального пути все узлы, которые составляют кластер, должны обращаться к тому же самому файлу через тот же самый путь:

```
In:
sc.textFile("hdfs:///datasets/hadoop_git_readme.txt").first()

Out:
u'For the latest information about Hadoop, please visit our
website at:'

In:
sc.textFile("file:///home/vagrant/datasets/hadoop_git_readme.txt").
first()

Out:
u'For the latest information about Hadoop, please visit our
website at:' '
```

Чтобы сохранить содержимое RDD-набора на диске, можно воспользоваться методом `saveAsTextFile`, предоставляемым моделью данных RDD. Здесь можно использовать несколько мест назначения. В приведенном ниже примере мы сохраним его в распределенной файловой системе HDFS и затем выведем содержимое на экран:

```
In:
numbers_rdd.saveAsTextFile("hdfs:///tmp/numbers_1_10.txt")

In:
!hdfs dfs -ls /tmp/numbers_1_10.txt
```

```
Out:
Found 5 items
-rw-r--r--   1 vagrant supergroup          0 2017-06-17 16:49 /tmp/numbers_1_10.txt/_SUCCESS
-rw-r--r--   1 vagrant supergroup          4 2017-06-17 16:49 /tmp/numbers_1_10.txt/part-00000
-rw-r--r--   1 vagrant supergroup          4 2017-06-17 16:48 /tmp/numbers_1_10.txt/part-00001
-rw-r--r--   1 vagrant supergroup          4 2017-06-17 16:49 /tmp/numbers_1_10.txt/part-00002
-rw-r--r--   1 vagrant supergroup          8 2017-06-17 16:48 /tmp/numbers_1_10.txt/part-00003
```

Платформа Spark пишет один файл для каждого раздела, в точности как MapReduce, который пишет один файл для каждого редуктора. Такой способ записи ускоряет время сохранения, поскольку каждый раздел сохраняется независимо, но на кластере с 1 узлом этот способ утяжеляет операцию чтения.

Можно ли собрать все разделы в один перед записью файла или, в общем случае, можно ли снизить число разделов в RDD-наборе? Ответ да, это можно сделать посредством метода `coalesce`, предоставляемого моделью данных RDD, передав в качестве аргумента число разделов, которые мы хотели бы иметь. Если передать `1`, то это вынудит RDD-набор находиться в автономном разделе и во время сохранения произведет всего один выходной файл. Отметим, что это происходит даже во время сохранения в локальной файловой системе: для каждого раздела создается файл. Следует учитывать, что выполнение этого метода в кластерной среде, состоящей из многочисленных узлов, не гарантирует, что все узлы будут видеть те же самые выходные файлы:

```
In:
numbers_rdd.coalesce(1)
    .saveAsTextFile("hdfs:///tmp/numbers_1_10_one_file.txt")
```

```
In:
!hdfs dfs -ls /tmp/numbers_1_10_one_file.txt
```

```
Out:
Found 2 items
-rw-r--r--   1 vagrant supergroup          0 2017-06-17 16:49 /tmp/numbers_1_10_one_file.
txt/_SUCCESS
-rw-r--r--   1 vagrant supergroup         20 2017-06-17 16:49 /tmp/numbers_1_10_one_file.txt/
part-00000
```

```
In:
!hdfs dfs -cat /tmp/numbers_1_10_one_file.txt/part-00000
```

```
Out:
0
1
2
3
4
5
6
7
8
9
```

```
In:
numbers_rdd.saveAsTextFile("file:///tmp/numbers_1_10.txt")
```

```
In:
!ls /tmp/numbers_1_10.txt

Out:
part-00000  part-00001      part-00002  part-00003      _SUCCESS
```

Модель данных RDD поддерживает всего два типа операций:

- *преобразования* трансформируют набор данных в другой вид. Входами и выходами преобразований являются RDD-наборы, поэтому многочисленные преобразования можно объединять в цепь, приближаясь к функциональному стилю программирования. Более того, все преобразования ленивые, т. е. они не вычисляют своих результатов сразу же;
- *действия* возвращают значения из RDD-наборов, в частности сумму элементов и количества, или просто собирают все элементы. Действия являются триггером для выполнения цепочки (ленивых) преобразований, так как наличие выходных данных обязательно.

Типичные программы платформы Spark представляют собой цепочку преобразований с действием в конце. По умолчанию все преобразования на RDD-наборе выполняются всегда, когда вы выполняете действие (т. е. промежуточное состояние после каждого трансформатора не сохраняется). Однако такое поведение можно переопределять при помощи метода `persist` (на RDD-наборе) всегда, когда вы хотите кэшировать значение преобразованных элементов. Метод `persist` обеспечивает постоянное хранение данных в памяти и на диске.

В следующем примере мы возведем в квадрат все значения в RDD-наборе и затем их просуммируем; этот алгоритм может быть выполнен посредством трансформатора (возведение элементов в квадрат), за которым следует редуктор (суммирование массива). В соответствии с платформой Spark метод `map` является трансформатором, поскольку он просто выполняет поэлементное преобразование, а метод `reduce` – действием, поскольку он генерирует значение из всех элементов.

Давайте подойдем к решению этой задачи в пошаговом режиме, чтобы увидеть многочисленные способы, которыми мы можем действовать. Прежде всего начнем с преобразования: сначала мы определим функцию, которая возвращает квадрат входного аргумента, затем передадим эту функцию методу `map` в модели данных RDD и наконец соберем элементы в RDD-набор:

```
In:
def sq(x):
    return x**2

numbers_rdd.map(sq).collect()

Out:
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Несмотря на правильность результата, функция `sq` занимает много места. Благодаря лямбда-выражению языка Python можно переписать преобразование более сжато; оно будет выглядеть следующим образом:

```
In:
numbers_rdd.map(lambda x: x**2).collect()

Out:
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Напомним, почему для распечатывания значения в преобразованном RDD-наборе нужно вызывать метод `collect`. Это вызвано тем, что метод `map` не переходит к действию, а просто лениво оценивается. С другой стороны, метод `reduce` является действием, и поэтому добавление шага редукции к предыдущему RDD-набору обязательно сгенерирует значение. Как и в отношении метода `map`, метод `reduce` берет в качестве аргумента функцию, которая должна иметь два аргумента (левое и правое значения) и должна вернуть значение. И даже в этом случае данная функция может быть многословной, если определять ее при помощи оператора `def` или функции `lambda`:

```
In:
numbers_rdd.map(lambda x: x**2).reduce(lambda a,b: a+b)

Out:
285
```

Чтобы сделать ее еще проще, вместо редуктора можно воспользоваться действием суммирования:

```
In:
numbers_rdd.map(lambda x: x**2).sum()

Out:
285
```

Мы пока показали очень простой пример работы с библиотекой pySpark. Давайте разберемся, что происходит в этот момент под капотом: сначала загружается набор данных и распределяется по всему кластеру, затем в распределенной среде выполняется операция преобразования, и потом все разделы сворачиваются, чтобы сгенерировать конечный результат (операция `sum` либо метод `reduce`), который в итоге распечатывается в блокноте Jupyter. Таким образом, задача огромной сложности сверхупрощается библиотекой pySpark.

Теперь продвинемся еще на один шаг и задействуем пары «ключ-значение»; хотя RDD-наборы могут содержать любой вид объектов (до сих пор мы видели целые числа и текстовые строки), кое-какие операции можно выполнить, когда элементами являются кортежи, состоящие из двух элементов: ключа и значения.

Чтобы проиллюстрировать на примере, давайте сначала сгруппируем числа в RDD-наборе на четные и нечетные и затем вычислим сумму этих двух групп по отдельности. Как и в отношении модели MapReduce, было бы неплохо промаркировать каждое число ключом (`even` и `odd`) и затем выполнить редукцию по каждому ключу с использованием операции суммирования.

Можно начать с операции преобразования: сначала создадим функцию, которая тегирует числа, выводя `even`, если число в аргументе четное, и `odd` в противном случае. Затем создадим преобразование «ключ-значение», которое создает пару «ключ-значение» для каждого числа, где ключ – это тег, а значение – само число:

```
In:
def tag(x):
    return 'even' if x%2==0 else 'odd'

numbers_rdd.map(lambda x: (tag(x), x) ).collect()
```

```
Out:
[('even', 0),
 ('odd', 1),
 ('even', 2),
 ('odd', 3),
 ('even', 4),
 ('odd', 5),
 ('even', 6),
 ('odd', 7),
 ('even', 8),
 ('odd', 9)]
```

Чтобы выполнить редукцию (свертку) каждого ключа по отдельности, теперь можно воспользоваться методом `reduceByKey` (который в платформе Spark не является действием). В качестве аргумента мы должны передать функцию, которую следует применить ко всем значениям, поставленным в соответствие каждому ключу; в данном случае мы их все просуммируем. Наконец, необходимо вызвать метод `collect`, чтобы распечатать результаты:

```
In:
numbers_rdd.map(lambda x: (tag(x), x) ).reduceByKey(
    lambda a,b: a+b).collect()

Out:
[('even', 20), ('odd', 25)]
```

Теперь давайте перечислим некоторые самые важные методы, которые имеются в платформе Spark; это не исчерпывающее руководство, но оно содержит самые используемые из них.

Начнем с преобразований; они могут применяться к RDD-набору, и они генерируют RDD-набор:

- `map(функция)`: возвращает RDD-набор, формируемый в результате прохождения каждого элемента через функцию;
- `flatMap(функция)`: возвращает RDD-набор, формируемый в результате уплощения (линеаризации) результата функции для каждого элемента входного RDD-набора. Используется, когда каждое значение на входе может быть преобразовано в 0 или более элементов на выходе.

Например, для подсчета количества вхождений каждого слова в текст мы должны преобразовать каждое слово в пару «ключ-значение» (слово будет ключом, а единица – значением), благодаря этому сгенерировав более одного элемента «ключ-значение» для каждой входной строки текста;

- `filter(функция)`: возвращает набор данных, состоящий из всех значений, где функция возвращает истину;
- `sample(с_заменой, доля, начальное число)`: извлекает бутстрап-выборку из RDD-набора, позволяя создавать выборочный RDD-набор (с заменой или без), чья длина составляет долю входного набора данных; начальное число соответствует начальному числу (seed) для генератора случайных чисел;
- `distinct()`: возвращает RDD-набор, содержащий уникальные элементы входного RDD-набора;
- `coalesce(число_разделов)`: сокращает число разделов в RDD;

- `repartition(число_разделов)`: изменяет число разделов в RDD. Этот метод всегда перетасовывает все данные по сети;
- `groupByKey()`: создает RDD-набор, где каждому ключу соответствует значение, являющееся последовательностью значений, которые имеют этот ключ во входном наборе данных;
- `reduceByKey(функция)`: агрегирует входной RDD-набор по ключу и затем применяет функцию reduce к значениям каждой группы;
- `sortByKey(по_возрастанию)`: сортирует элементы в RDD-наборе по ключу в восходящем или нисходящем порядке;
- `union(другой_RDD)`: объединяет два RDD-набора;
- `intersection(другой_RDD)`: возвращает RDD-набор, состоящий только из значений, встречающихся и во входном RDD-наборе, и в RDD-наборе, переданном в качестве аргумента;
- `join(другой_RDD)`: возвращает набор данных, где входные пары «ключ-значение» присоединены (по ключу) к RDD-набору, переданному в качестве аргумента.

Так же как в SQL, помимо функции join, также имеются следующие методы: `cartesian`, `leftOuterJoin`, `rightOuterJoin` и `fullOuterJoin`.

Теперь давайте получим общее представление о наиболее популярных действиях, которые имеются в библиотеке pySpark. Отметим, что действия инициируют обработку RDD-набора всеми находящимися в цепочке преобразователями:

- `reduce(функция)`: агрегирует элементы в RDD-наборе, генерируя выходное значение;
- `count()`: возвращает количество элементов в RDD-наборе;
- `countByKey()`: возвращает словарь Python, где каждому ключу поставлено в соответствие число элементов в RDD-наборе с этим ключом;
- `collect()`: возвращает все элементы в трансформированном RDD-наборе локально;
- `first()`: возвращает первое значение в RDD-наборе;
- `take(N)`: возвращает первые *N* значений в RDD-наборе;
- `takeSample(с_заменой, N, начальное_число)`: возвращает бутстрап-выборку из *N* элементов в RDD-наборе с заменой или без, возможно, при помощи начального числа для генератора случайных чисел, передаваемого в качестве аргумента;
- `takeOrdered(N, упорядочивание)`: возвращает верхние *N* элементов в RDD-наборе после его сортировки по значению (в восходящем или нисходящем порядке);
- `saveAsTextFile(путь)`: сохраняет RDD-набор как набор текстовых файлов в указанном каталоге.

Кроме того, имеется несколько методов, которые не являются ни преобразователями, ни действиями:

- `cache()`: кэширует элементы RDD-набора; поэтому в будущих расчетах на основе того же самого RDD-набора он может использоваться повторно в качестве отправной точки;
- `persist(хранилище)`: то же самое, что и метод `cache`, но имеется возможность указать устройство, куда сохранять элементы RDD-набора (оперативная память, диск либо оба этих устройства);
- `unpersist()`: отменяет операцию сохранения либо кэширования.

Давайте теперь попробуем реплицировать примеры, которые мы видели в разделе о реализации вычислительной парадигмы MapReduce в среде Hadoop. В платформе Spark алгоритм должен быть следующим:

1. Входной файл считывается и параллелизируется на RDD-наборе. Эта операция может быть выполнена при помощи метода `textFile`, предоставляемого контекстом Spark.
2. Для каждой строки входного файла возвращаются три пары «ключ-значение»: одна содержит число символов, другая – число слов, и последняя – число строк. В платформе Spark эта операция выполняется методом `flatMap`, так как три результата генерируются для каждой входной строки.
3. Для каждого ключа суммируются все значения. Это может быть сделано при помощи метода `reduceByKey`.
4. В заключение собираются результаты. В данном случае можно применить метод `collectAsMap`, который собирает пары «ключ-значение» в RDD-набор и возвращает словарь Python. Отметим, что эта операция является действием, поэтому на RDD-наборе исполняется цепочка преобразований и возвращается результат.

```
In:
def emit_feats(line):
    return [('chars', len(line)),
            ('words', len(line.split())),
            ('lines', 1)]

print(sc.textFile('/datasets/hadoop_git_readme.txt')
      .flatMap(emit_feats)
      .reduceByKey(lambda a,b: a+b)
      .collectAsMap())

Out:
{'chars': 1335, 'lines': 31, 'words': 179}
```

Сразу же можно отметить огромную скорость этого метода по сравнению с реализацией MapReduce. Это вызвано тем, что весь набор данных хранится в оперативной памяти, а не в распределенной файловой системе HDFS. Во-вторых, это чистая реализация на Python, и нам не нужно вызывать внешние команды или библиотеки – библиотека pySpark автономна.

Теперь давайте для примера поработаем на более крупном файле с сочинениями Уильяма Шекспира и извлечем самое популярное слово. В реализации Hadoop MapReduce эта задача требует двух шагов трансформации-редукции и, следовательно, четырех операций записи-чтения в HDFS. В библиотеке pySpark все это можно сделать в RDD-наборе.

1. Входной файл считывается и параллелизируется на RDD-наборе при помощи метода `textFile`.
2. Из каждой строки извлекаются все слова. Для этой операции можно воспользоваться методом `flatMap` и регулярным выражением.
3. Каждое слово в тексте (т. е. каждый элемент RDD-набора) теперь преобразуется в пару «ключ-значение»: ключ – это слово строчными буквами, а значение всегда равняется единице. Это операция преобразования.
4. При помощи вызова метода `reduceByKey` подсчитывается, сколько раз каждое слово (ключ) появляется в тексте (RDD-наборе). На выходе получаем пары

«ключ-значение», где ключ – это слово, а значение – количество вхождений слова в текст.

5. Переворачиваем ключи и значения, создавая новый RDD-набор. Это операция преобразования.
6. Сортируем RDD-набор в порядке убывания и извлекаем (take) первый элемент. Это действие, и оно может быть выполнено одной операцией при помощи метода `takeOrdered`.

```
In:
import re
WORD_RE = re.compile(r"[\w']+")

print(sc.textFile('/datasets/shakespeare_all.txt')
      .flatMap(lambda line: WORD_RE.findall(line))
      .map(lambda word: (word.lower(), 1))
      .reduceByKey(lambda a,b: a+b)
      .map(lambda (k,v): (v,k))
      .takeOrdered(1, key = lambda x: -x[0]))

Out:
[(27801, u'the')]
```

Конечные результаты такие же, что и при использовании платформы Hadoop и вычислительной парадигмы MapReduce, но в данном случае все вычисление занимает намного меньше времени. В действительности можно еще больше улучшить это решение, свернув второй и третий шаги (обработав пару «ключ-значение» для каждого слова методом `flatMap`, где ключ – это слово строчными буквами, а значение – количество появлений слова), а также пятый и шестой шаги (взяв первый элемент и упорядочив элементы в RDD-наборе по их значению, т. е. второму элементу пары):

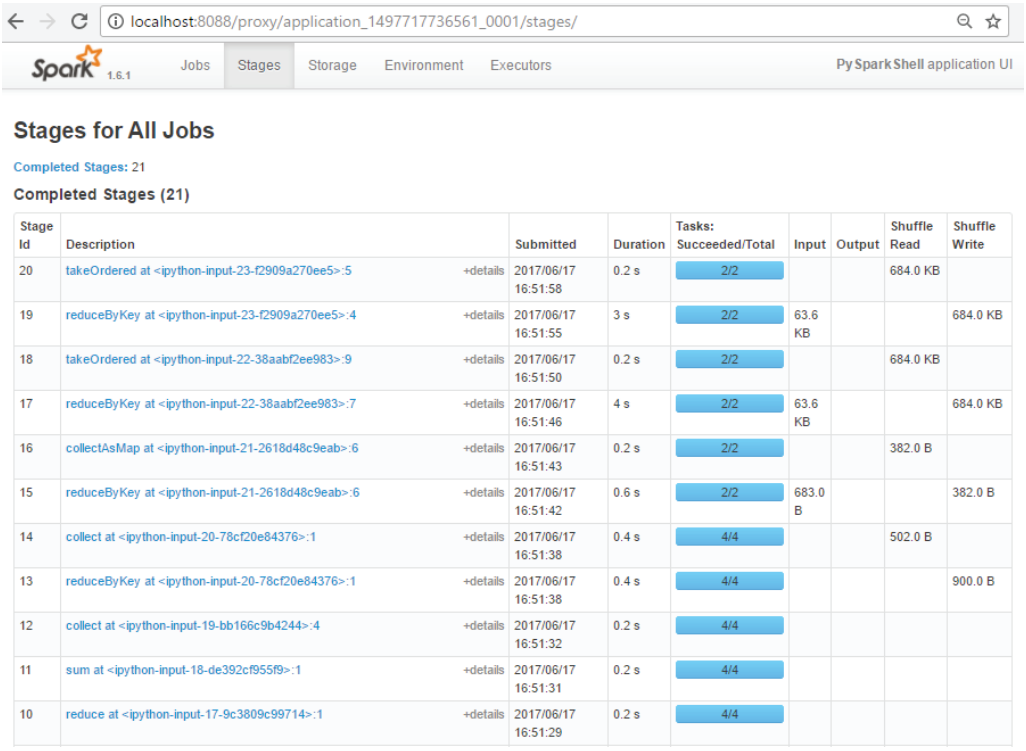
```
In:
print(sc.textFile('/datasets/shakespeare_all.txt')
      .flatMap(lambda line: [(word.lower(), 1)
                             for word in WORD_RE.findall(line)])
      .reduceByKey(lambda a,b: a+b)
      .takeOrdered(1, key = lambda x: -x[1]))

Out:
[(u'the', 27801)]
```

Для проверки состояния обработки можно воспользоваться веб-интерфейсом платформы Spark: это графический интерфейс, который пошагово показывает задания, выполняемые платформой Spark. Для доступа к веб-интерфейсу необходимо сначала выяснить имя приложения pySpark Jupyter, отыскав его имя в окне оболочки bash в том месте, где вы запустили блокнот (обычно имя приложения имеет форму `application_<число>_<число>`), и затем направить свой браузер на страницу `http://localhost:8088/proxy/application_<число>_<число>`[1].

---

[1] Имя приложения можно выяснить на странице приложений `http://localhost:8088/cluster`. – *Прим. перев.*

Будет получен результат, который аналогичен показанному на следующем ниже изображении. Он содержит все задания, выполненные в платформе Spark (в качестве ячеек блокнота Jupyter), при этом можно также визуализировать план исполнения в виде **направленного ациклического графа** (DAG):



Резюме

В этой главе мы представили некоторые примитивы, которые позволяют выполнять распределенные задания на кластере, состоящем из многочисленных узлов. Мы продемонстрировали платформу Hadoop и все ее компоненты, функции и ограничения, а затем проиллюстрировали работу платформы Spark.

В следующей главе мы более детально остановимся на платформе Spark и покажем, каким образом можно реализовывать проекты в области науки о данных в распределенной среде.

Глава 9

Практическое машинное обучение в среде Spark

В предыдущей главе мы познакомились с основным функционалом обработки данных при помощи платформы Spark. В этой главе в центре внимания будет решение реальных задач в области науки о данных при помощи платформы Spark. По ходу настоящей главы вы изучите следующие темы:

- как распространять переменные по всем узлам кластера;
- как создавать объекты DataFrame из структурированных (CSV) и полуструктурированных (JSON) файлов, сохранять их на диске и загружать с диска;
- как использовать SQL-подобный синтаксис, чтобы выполнять выборку, фильтрацию, соединение, группировку и агрегацию данных, тем самым чрезвычайно упрощая предобработку;
- как обращаться с пропущенными данными в наборе данных;
- какие в платформе Spark имеются готовые к использованию алгоритмы для конструирования признаков и как их применять в реальном сценарии;
- какие есть ученики и как измерять их результативность в распределенной среде;
- как выполнять перекрестную проверку для гиперпараметрической оптимизации в кластере.

Настройка виртуальной машины для данной главы

Поскольку машинное обучение нуждается в больших вычислительных мощностях, в настоящей главе, чтобы сэкономить немного ресурсов (в особенности оперативную память), мы будем использовать среду Spark без поддержки менеджера ресурсов YARN. Этот операционный режим называется автономным и создает узел Spark без кластерного функционала; вся обработка будет на драйверной (ведущей) машине и не будет разделенной. Не переживайте: программный код, который мы увидим в этой главе, будет работать и в кластерной среде тоже.

Для того чтобы начать работать в такого рода режиме, необходимо выполнить следующие шаги:

1) включить виртуальную машину командой `vagrant up`;
2) получить доступ к виртуальной машине, когда она будет готова, при помощи команды `vagrant ssh`;

3) запустить автономный режим платформы Spark вместе с блокнотом Jupyter изнутри виртуальной машины при помощи `./start_jupyter.sh`;
4) открыть браузер и направить его по URL-адресу `http://localhost:8888`.

Чтобы его выключить, используйте клавиши **Ctrl+C** для выхода из блокнота Jupyter и команду `vagrant halt` для выключения виртуальной машины.

Отметим, что даже в такой конфигурации можно получить доступ к веб-интерфейсу Spark (когда, как минимум, работает блокнот Jupyter) по следующему URL-адресу: `http://localhost:4040`.

Распространение переменных по всем узлам кластера

Когда мы работаем в распределенной среде, иногда требуется делиться информацией между узлами, чтобы все узлы могли работать, используя согласованные переменные. Платформа Spark регулирует этот случай, обеспечивая два вида переменных: переменные только для записи и переменные только для чтения. Не гарантируя больше, что разделяемая переменная доступна одновременно и для чтения, и для записи, платформа Spark также отказывается от необходимости в обеспечении согласованности, перекладывая всю тяжелую работу по управлению такой ситуацией на плечи разработчика. Как правило, решение достигается быстро, так как платформа Spark на самом деле является гибкой и настраиваемой средой.

Широковещательные переменные только для чтения

В нашей конфигурации широковещательные переменные – это переменные, которые распространяются драйверным узлом, т. е. узлом, выполняющим блокнот Jupyter, по всем узлам в кластере. Такая переменная доступна только для чтения, поскольку она распространяется одним узлом и никогда не считывается назад, если другой узел ее изменяет.

Теперь на простом примере посмотрим, как это работает: мы применим прямое унитарное кодирование к набору данных с информацией о половой принадлежности в виде сроки. Если быть точным, фиктивный набор данных содержит всего один признак, который может иметь три значения: мужчина *M*, женщина *F* и неизвестно *U* (если информация отсутствует). Говоря конкретно, нужно, чтобы все узлы использовали прямое унитарное кодирование, как показано в приведенном ниже словаре:

```
In:
one_hot_encoding = {"M": (1, 0, 0),
                    "F": (0, 1, 0),
                    "U": (0, 0, 1)
                    }
```

Теперь выполним этот пример в пошаговом режиме.

Самое простое решение (которое, впрочем, не работает) – параллелизировать фиктивный набор данных (или прочесть его с диска) и затем применить метод `map` с лямбда-функцией к RDD-набору, чтобы линейно преобразовать информацию о половой принадлежности в соответствующий кодированный кортеж:

```
In:
(sc.parallelize(["M", "F", "U", "F", "M", "U"])
   .map(lambda x: one_hot_encoding[x])
   .collect())

Out:
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 0, 1)]
```

Это решение работает локально, но оно не будет работать в реальной распределенной среде, поскольку не все узлы имеют переменную `one_hot_encoding` в своей рабочей области. Быстрое обходное решение состоит во включении словаря Python в функцию, (распределенно) выполняющую линейное преобразование, как нам удалось сделать ниже:

```
In:
def map_ohe(x):
    ohe = {"M": (1, 0, 0),
           "F": (0, 1, 0),
           "U": (0, 0, 1)
           }
    return ohe[x]

sc.parallelize(["M", "F", "U", "F", "M", "U"]).map(map_ohe).collect()

Out:
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 0, 1)]
```

Такое решение работает и локально, и на сервере, но оно не очень хорошее: мы смешали данные и процесс, сделав преобразующую функцию немногоразовой. Гораздо лучше, если преобразующая функция будет обращаться к широковещательной переменной, которая может использоваться с любым возможным преобразованием, необходимым для прямого кодирования набора данных.

Для этого сначала следует распространить словарь Python (вызвав метод `broadcast`, предоставленный контекстом Spark `sc`) внутри функции, выполняющей линейное преобразование. Теперь к словарю можно получить доступ, используя его свойство `value`. В результате этой операции мы получим обобщенную функцию преобразования, которая может работать с любым прямокодированным словарем:

```
In:
bcast_map = sc.broadcast(one_hot_encoding)

def bcast_map_ohe(x, shared_ohe):
    return shared_ohe[x]

(sc.parallelize(["M", "F", "U", "F", "M", "U"])
   .map(lambda x: bcast_map_ohe(x, bcast_map.value))
   .collect())

Out:
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 0, 1)]
```

Думайте о широковещательной переменной как о файле, записанном в распределенной файловой системе HDFS. Впоследствии, когда обобщенный узел хочет получить к этой переменной доступ, ему нужен только путь в HDFS (который

передается как аргумент метода `map`). Таким образом, если будет использоваться одинаковый путь, то все узлы гарантированно прочитают то же самое. Разумеется, платформа Spark использует не саму распределенную файловую систему HDFS, а ее вариант в оперативной памяти.

Широковещательные переменные хранятся в оперативной памяти во всех узлах, составляющих кластер, поэтому на них никогда не приходится большой объем данных, который мог бы их заполнить и сделать невозможной последующую обработку.

Для удаления широковещательной переменной примените к ней метод `unpersist`. Данная операция высвободит память от этой переменной на всех узлах:

```
In:
bcast_map.unpersist()
```

Аккумуляторные переменные только для записи

Аккумуляторы – это еще один вид переменных, которые могут быть распространены в кластере Spark. Эти переменные доступны только для записи; они могут складываться с другими и обычно используются для реализации сумм или счетчиков. Значение аккумуляторной переменной может быть прочитано только драйверным узлом, т. е. тем, который выполняет блокнот Jupyter; все другие узлы этого делать не могут.

Давайте посмотрим на примере, как работает аккумулятор: требуется обработать текстовый файл и понять, сколько строк в нем являются пустыми. Разумеется, это можно сделать, дважды просканировав набор данных (с использованием двух заданий Spark): в первый раз, прочитав пустые строки, и второй раз, выполнив реальную обработку, однако такое решение будет не очень эффективным.

В первом неэффективном решении с извлечением количества пустых строк, используя два автономных задания Spark, как показано ниже, мы читаем текстовый файл, фильтруем пустые строки и подсчитываем их количество:

```
In:
print('Количество пустых строк: {0}'
      .format(sc.textFile(
            'file:///home/vagrant/datasets/hadoop_git_readme.txt')
               .filter(lambda line: len(line) == 0).count()))

Out:
Количество пустых строк: 6
```

Второе решение, напротив, будет эффективнее (и сложнее). Мы инстанцируем аккумулятор (начальным значением `0`) и добавляем `1` для каждой пустой строки, которую мы находим при обработке каждой строки входного файла (при помощи функции `map`). Одновременно с этим можно выполнять некую обработку каждой строки. В следующем ниже фрагменте кода, например, для каждой строки просто возвращается `1`, благодаря чему подсчитываются все строки в файле.

В конце обработки будут два элемента данных: первый – это число строк, полученное в результате действия `count()` на преобразованном RDD-наборе, и второй – число пустых строк в свойстве `value` аккумулятора. Напомним – оба этих значения доступны после однократного сканирования набора данных:

```
In:
accum = sc.accumulator(0)

def split_line(line):
    if len(line) == 0:
        accum.add(1)
    return 1

tot_lines = (
    sc.textFile('file:///home/vagrant/datasets/hadoop_git_readme.txt')
      .map(split_line)
      .count())

empty_lines = accum.value

print('Файл содержит %d строк(у),' % tot_lines)
print('из которых %d пустые' % empty_lines)

Out:
Файл содержит 31 строк(у),
из которых 6 пустые
```

Платформа Spark нативно поддерживает аккумуляторы числовых типов, при этом операция суммирования в них задана по умолчанию. Приложив еще немного усилий, эту переменную можно превратить в нечто более сложное.

Широковещательные и аккумуляторные переменные – пример

Несмотря на то что широковещательные и аккумуляторные переменные просты и функционально очень ограничены (одна доступна только для чтения, другая – только для записи), они могут активно использоваться для создания очень сложных операций. Например, теперь попробуем применить разные алгоритмы машинного обучения в распределенной среде на наборе данных Iris. Мы сконструируем задание Spark следующим образом:

1. Набор данных считывается и распространяется по всем узлам (поскольку он достаточно небольшой, чтобы уместиться в оперативной памяти).
2. Каждый узел применит к набору данных отличающийся классификатор и вернет имя классификатора и его отметку точности на полном наборе данных. Здесь необходимо отметить, что в этом простом примере предобработка, разделение набора данных на тренировочный и тестовый наборы и гиперпараметрическая оптимизация выполняться не будут. Это делается с целью оставить все как можно проще.
3. Если классификаторы поднимут исключение, то в аккумуляторе должны быть сохранены строковое представление ошибки и имя классификатора.
4. Конечный результат должен содержать список классификаторов, которые выполнили задачу классификации без ошибок, и их отметку точности.

В качестве первого шага загрузим набор данных Iris и распространим его по всем узлам в кластере:

```
In:
from sklearn.datasets import load_iris
bcast_dataset = sc.broadcast(load_iris())
```

Теперь создадим пользовательский аккумулятор. Он будет содержать список кортежей, в которых будут храниться имя классификатора и произошедшее ис-

ключение в виде строки символов. Пользовательский аккумулятор является производным от класса `AccumulatorParam` и содержит как минимум два метода: `zero` (который вызывается во время его инициализации) и `addInPlace` (который вызывается, когда на аккумуляторе вызывается метод `add`).

Самый простой способ реализации пользовательского аккумулятора показан в следующем ниже примере, в конце которого выполняется его инициализация в виде пустого списка. Следует учесть, что аддитивная операция немного сложнее: необходимо объединить два элемента, кортеж и список, но мы не знаем, какой элемент является списком, а какой кортежем, поэтому сначала обеспечивается, чтобы оба элемента были списками, и только потом выполняется их конкатенация простым способом (при помощи оператора `+`):

```
In:
from pyspark import AccumulatorParam

class ErrorAccumulator(AccumulatorParam):
    def zero(self, initialList):
        return initialList

    def addInPlace(self, v1, v2):
        if not isinstance(v1, list):
            v1 = [v1]
        if not isinstance(v2, list):
            v2 = [v2]
        return v1 + v2

errAccum = sc.accumulator([], ErrorAccumulator())
```

Теперь определим преобразующую функцию: каждый узел должен натренировать классификатор, протестировать и оценить его на распространенном наборе данных Iris. В качестве своего аргумента функция получит объект классификатора и должна вернуть список, состоящий из кортежа с именем классификатора и его отметкой точности.

Если при выполнении будет поднято исключение, то в аккумулятор будут добавлены имя классификатора и исключение в виде строки символов и возвращен пустой список:

```
In:
def apply_classifier(clf, dataset):

    clf_name = clf.__class__.__name__
    X = dataset.value.data
    y = dataset.value.target

    try:
        from sklearn.metrics import accuracy_score

        clf.fit(X, y)
        y_pred = clf.predict(X)
        acc = accuracy_score(y, y_pred)

        return [(clf_name, acc)]

    except Exception as e:
        errAccum.add((clf_name, str(e)))
        return []
```

Наконец, мы дошли до сердцевины задания. Теперь мы проинстанцируем несколько объектов из библиотеки Scikit-learn (некоторые из них не являются классификаторами с целью протестировать аккумулятор). Мы преобразуем их в RDD-набор и применим функцию преобразования, которая была создана в предыдущей ячейке. Поскольку возвращенное значение является списком, мы можем воспользоваться методом `flatMap` для сбора результатов только из тех преобразователей, в которых не было перехвачено исключение:

```
In:
from sklearn.linear_model import SGDClassifier
from sklearn.dummy import DummyClassifier
from sklearn.decomposition import PCA
from sklearn.manifold import MDS

classifiers = [DummyClassifier('most_frequent'),
               SGDClassifier(),
               PCA(),
               MDS()]

(sc.parallelize(classifiers)
     .flatMap(lambda x: apply_classifier(x, bcast_dataset))
     .collect())

Out:
[('DummyClassifier', 0.33333333333333331),
 ('SGDClassifier', 0.6666666666666663)]
```

Как и ожидалось, результат содержит только *реальные* классификаторы. Теперь посмотрим, какие классификаторы сгенерировали ошибку. Неудивительно, здесь мы обнаруживаем два классификатора, пропущенных в предыдущем результате:

```
In:
print('Список ошибок: {0}'.format(errAccum.value))

Out:
Список ошибок: [('PCA', "'PCA' object has no attribute 'predict'"), ('MDS', "Proximity must be
'precomputed' or 'euclidean'. Got euclidean instead")]
```

В качестве последнего шага очистим распространенный по всем узлам набор данных:

```
In:
bcast_dataset.unpersist()
```

Напомним, что в указанном примере был использован небольшой набор данных, который можно было распространить по всем узлам. В реальных задачах с большими данными придется загрузить набор данных из распределенной файловой системы HDFS и распространить по всем узлам путь в HDFS.

Предобработка данных в среде Spark

До сих пор мы видели способы загрузки текстовых данных из локальной файловой системы и распределенной файловой системы HDFS. Текстовые файлы могут содержать любые неструктурированные данные (к примеру, текстовый документ) или структурированные данные (как, например, CSV-файл). Что касается полу-

структурированных данных, то так же, как и файлы с объектами JSON, платформа Spark имеет специальные подпрограммы, которые в состоянии преобразовывать файл в объект DataFrame, который аналогичен таблицам данных DataFrame в среде R и в библиотеке pandas языка Python. Объекты DataFrame очень похожи на таблицы реляционных СУБД, в которых задана схема данных.

Файлы JSON и объекты DataFrame платформы Spark

Чтобы импортировать JSON-совместимые файлы, прежде всего необходимо создать контекст SQL, в свою очередь, создав объект `sqlContext` из локального контекста Spark:

```
In:
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

Теперь посмотрим на содержимое небольшого файла JSON (он находится в виртуальной машине Vagrant). Он представляет собой таблицу с шестью строками и тремя столбцами, представленную в формате JSON, в которой некоторые атрибуты пропущены (в частности, атрибут `gender` у пользователя с идентификатором `user_id=0`):

```
In:
!cat /home/vagrant/datasets/users.json
```

```
Out:
{"user_id":0, "balance": 10.0}
{"user_id":1, "gender":"M", "balance": 1.0}
{"user_id":2, "gender":"F", "balance": -0.5}
{"user_id":3, "gender":"F", "balance": 0.0}
{"user_id":4, "balance": 5.0}
{"user_id":5, "gender":"M", "balance": 3.0}
```

Благодаря методу `read.json`, предоставленному объектом `sqlContext`, в переменной уже имеется хорошо отформатированная таблица со всеми нужными именами столбцов. Выходная переменная имеет тип DataFrame платформы Spark. Чтобы показать переменную в хорошо отформатированной таблице, используется ее метод `show`:

```
In:
df = sqlContext.read.json('file:///home/vagrant/datasets/users.json')
df.show()
```

```
Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|   10.0|  null|      0|
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
|    5.0|  null|      4|
|    3.0|     M|      5|
+-------+------+-------+
```

Помимо этого, при помощи метода `printSchema` можно просмотреть схему объекта DataFrame. Очевидно, что при чтении файла JSON каждый тип столбца определяется по типу находящихся в нем данных (в примере столбец `user_id` содержит длинное целое, столбец `gender` – последовательности символов и `balance` – вещественное число двойной точности с плавающей точкой):

```
In:
df.printSchema()

Out:
 root
 |-- balance: double (nullable = true)
 |-- gender: string (nullable = true)
 |-- user_id: long (nullable = true)
```

Данные в объекте DataFrame можно нарезать горизонтально и вертикально в точности, как в таблице реляционной СУБД, отбирая столбцы и фильтруя данные по атрибутам. В этом примере нужно распечатать баланс, пол и идентификатор пользователей, чей пол не пропущен и которые имеют баланс строго больше нуля. Для этого можно воспользоваться методами `filter` и `select`:

```
In:
(df.filter(df['gender'] != 'null')
   .filter(df['balance'] > 0)
   .select(['balance', 'gender', 'user_id'])
   .show())

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|    3.0|     M|      5|
+-------+------+-------+
```

Кроме того, можно переписать все предыдущее задание на SQL-подобном языке. По сути дела, методы `filter` и `select` могут принимать строки, отформатированные по правилам SQL:

```
In:
(df.filter('gender is not null')
   .filter('balance > 0').select("*").show())

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|    3.0|     M|      5|
+-------+------+-------+
```

Можно также применить всего один вызов метода `filter`:

```
In:
df.filter('gender is not null and balance > 0').show()
```

```
Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|    3.0|     M|      5|
+-------+------+-------+
```

Работа с пропущенными данными

Типичную проблему во время предобработки представляет управление пропущенными данными. Объекты DataFrame платформы Spark, аналогично таблицам данных DataFrame библиотеки pandas, предлагают широкий диапазон операций, предназначенных для их обработки. Например, самый простой вариант собрать набор данных только из полных строк состоит в том, чтобы отбросить строки, содержащие пропущенную информацию. Для этого в объектах DataFrame платформы Spark сначала нужно обратиться к атрибуту `na` объекта DataFrame и затем вызвать метод `drop`. Результирующая таблица будет содержать только полные строки:

```
In:
df.na.drop().show()
```

```
Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
|    3.0|     M|      5|
+-------+------+-------+
```

Если такая операция удаляет слишком много строк, то всегда можно определить, какие столбцы служат причиной для удаления строки (применив расширенное подмножество метода `drop`):

```
In:
df.na.drop(subset=["gender"]).show()
```

```
Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
|    3.0|     M|      5|
+-------+------+-------+
```

Кроме того, если вместо удаления этих строк нужно для каждого столбца установить значения по умолчанию, то можно применить метод `fill`, передав словарь, состоящий из имени столбца (в котором ключом является имя столбца) и значения по умолчанию для замены пропущенных данных в этом столбце (т. е. в виде значения ключа в словаре).

В качестве одного из примеров, если нужно обеспечить, чтобы переменные balance и gender там, где их значения пропущены, были установлены соответственно в 0 и U, можно просто сделать следующее:

```
In:
df.na.fill({'gender': "U", 'balance': 0.0}).show()
```

```
Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|   10.0|     U|      0|
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
|    5.0|     U|      4|
|    3.0|     M|      5|
+-------+------+-------+
```

Группирование и создание таблиц в оперативной памяти

Чтобы применить функцию к группе строк (как в случае с командой SQL GROUP BY), можно воспользоваться двумя аналогичными методами. В следующем ниже примере вычисляется средний баланс в расчете на пол:

```
In:
(df.na.fill({'gender': "U", 'balance': 0.0})
   .groupBy("gender").avg('balance').show())
```

```
Out:
+------+------------+
|gender|avg(balance)|
+------+------------+
|     F|       -0.25|
|     M|         2.0|
|     U|         7.5|
+------+------------+
```

До сих пор мы работали только с объектами DataFrame, но, как вы могли уже заметить, расстояние между методами объекта DataFrame и командами SQL минимальное. На самом деле при помощи платформы Spark можно зарегистрировать объект DataFrame как таблицу SQL и в полной мере воспользоваться возможностями SQL. Такая таблица хранится в памяти и распределяется аналогично RDD-набору.

Чтобы зарегистрировать таблицу, нужно предоставить имя, которое будет использоваться в будущих командах SQL. В данном случае мы назначаем имя users:

```
In:df.registerTempTable("users")
```

Посредством вызова метода sql, предоставленного объектом sqlContext платформы Spark, можно сгенерировать любую SQL-совместимую таблицу:

```
In:
sqlContext.sql("""
   SELECT gender, AVG(balance)
```

```
   FROM users
   WHERE gender IS NOT NULL
   GROUP BY gender""").show()
```

```
Out:
+------+-----+
|gender|  _c1|
+------+-----+
|     F|-0.25|
|     M|  2.0|
+------+-----+
```

Неудивительно, что выведенная этой командой таблица (а также сама таблица `users`) имеет тип DataFrame платформы Spark:

```
In:
type(sqlContext.table("users"))
```

```
Out:
pyspark.sql.dataframe.DataFrame
```

Объекты DataFrame, таблицы и RDD-наборы тесно между собой связаны, и методы RDD могут использоваться на объектах DataFrame. Напомним, что каждая строка в объекте DataFrame является элементом RDD-набора. Давайте рассмотрим этот функционал детально и сначала соберем полную таблицу:

```
In:
sqlContext.table("users").collect()
```

```
Out:
[Row(balance=10.0, gender=None, user_id=0),
 Row(balance=1.0, gender=u'M', user_id=1),
 Row(balance=-0.5, gender=u'F', user_id=2),
 Row(balance=0.0, gender=u'F', user_id=3),
 Row(balance=5.0, gender=None, user_id=4),
 Row(balance=3.0, gender=u'M', user_id=5)]
```

```
In:
a_row = sqlContext.sql("SELECT * FROM users").first()
a_row
```

```
Out:
Row(balance=10.0, gender=None, user_id=0)
```

Результатом является список объектов `Row` (они аналогичны именованному кортежу `namedtuple` в языке Python). Давайте здесь остановимся подробнее: объект `Row` содержит многочисленные атрибуты, к которым можно получить доступ через свойство либо по ключу словаря. Иными словами, чтобы извлечь из первой строки баланс, можно выбрать между двумя следующими вариантами:

```
In:
print(a_row['balance'])
print(a_row.balance)
```

```
Out:
10.0
10.0
```

Кроме того, объект `Row` можно собрать как словарь Python при помощи метода `asDict` объекта `Row`. Результат будет содержать имена свойств в качестве ключа и значения свойств в качестве значений словаря:

```
In:
a_row.asDict()

Out:
{'balance': 10.0, 'gender': None, 'user_id': 0}
```

Запись предобработанного объекта DataFrame или RDD-набора на диск

Для записи объекта DataFrame или RDD-набора на диск можно воспользоваться методом `write`. Имеется ряд форматов на выбор; в данном случае мы сохраним его на локальной машине, как файл JSON:

```
In:
(df.na.drop().write
      .save("file:///tmp/complete_users.json", format='json'))
```

Проверяя результаты в локальной файловой системе, сразу видно, что что-то отличается от того, что ожидалось получить: эта операция создает многочисленные файлы (`part-r-...`).

Каждый из них содержит несколько строк, сериализованных как объекты JSON, и их слияние на выходе создаст исчерпывающий результат. Поскольку платформа Spark создана для обработки больших и распределенных файлов, операция записи соответствующим образом адаптирована для таких задач, и каждый узел пишет часть полного RDD-набора:

```
In:
!ls -als /tmp/complete_users.json

Out:
total 28
4 drwxrwxr-x 2 vagrant vagrant 4096 Jun 17 14:40 .
4 drwxrwxrwt 9 root    root    4096 Jun 17 14:40 ..
4 -rw-r--r-- 1 vagrant vagrant   83 Jun 17 14:40 part-r-00000-78a81d94-92fd-4853-acda-
1f199c40dbab
4 -rw-rw-r-- 1 vagrant vagrant   12 Jun 17 14:40 .part-r-00000-78a81d94-92fd-4853-acda-
1f199c40dbab.crc
4 -rw-r--r-- 1 vagrant vagrant   82 Jun 17 14:40 part-r-00001-78a81d94-92fd-4853-acda-
1f199c40dbab
4 -rw-rw-r-- 1 vagrant vagrant   12 Jun 17 14:40 .part-r-00001-78a81d94-92fd-4853-acda-
1f199c40dbab.crc
0 -rw-r--r-- 1 vagrant vagrant    0 Jun 17 14:40 _SUCCESS
4 -rw-rw-r-- 1 vagrant vagrant    8 Jun 17 14:40 ._SUCCESS.crc
```

Для того чтобы его прочитать обратно, создавать автономный файл не требуется – операция чтения прекрасно обрабатывает даже многочисленные части. Файл JSON можно также прочитать в операторе `FROM` SQL-запроса. Теперь попробуем распечатать файл JSON, который мы только что записали на диск, не создавая промежуточного объекта DataFrame:

```
In:
sqlContext.sql(
    "SELECT * FROM json.`file:///tmp/complete_users.json`").show()

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
|    3.0|     M|      5|
+-------+------+-------+
```

Помимо JSON, существует еще один формат, очень популярный при работе со структурированными большими наборами данных: формат Parquet. Формат Parquet – это колоночный формат хранения данных, который имеется в экосистеме Hadoop; он сжимает и кодирует данные и может работать с вложенными структурами: все эти особенности делают его очень эффективным.

Запись и загрузка файла в формате Parquet очень похожи на работу с форматом JSON, при этом операция записи в этом случае тоже производит многочисленные файлы, которые записываются на диск:

```
In:
df.na.drop().write.save('file:///tmp/complete_users.parquet',
                        format='parquet')

In:
!ls -als /tmp/complete_users.parquet/

Out:
total 44
4 drwxrwxr-x  2 vagrant vagrant 4096 Jun 17 14:40 .
4 drwxrwxrwt 10 root    root    4096 Jun 17 14:40 ..
4 -rw-r--r--  1 vagrant vagrant  376 Jun 17 14:40 _common_metadata
4 -rw-rw-r--  1 vagrant vagrant   12 Jun 17 14:40 ._common_metadata.crc
4 -rw-r--r--  1 vagrant vagrant 1082 Jun 17 14:40 _metadata
4 -rw-rw-r--  1 vagrant vagrant   20 Jun 17 14:40 ._metadata.crc
4 -rw-r--r--  1 vagrant vagrant  750 Jun 17 14:40 part-r-00000-e637364c-fa40-466b-8c9e-
90d9d3ed047e.gz.parquet
4 -rw-rw-r--  1 vagrant vagrant   16 Jun 17 14:40 .part-r-00000-e637364c-fa40-466b-8c9e-
90d9d3ed047e.gz.parquet.crc
4 -rw-r--r--  1 vagrant vagrant  746 Jun 17 14:40 part-r-00001-e637364c-fa40-466b-8c9e-
90d9d3ed047e.gz.parquet
4 -rw-rw-r--  1 vagrant vagrant   16 Jun 17 14:40 .part-r-00001-e637364c-fa40-466b-8c9e-
90d9d3ed047e.gz.parquet.crc
0 -rw-r--r--  1 vagrant vagrant    0 Jun 17 14:40 _SUCCESS
4 -rw-rw-r--  1 vagrant vagrant    8 Jun 17 14:40 ._SUCCESS.crc
```

Работа с объектами DataFrame

До сих пор давалось описание того, как загружать объекты DataFrame из файлов JSON и Parquet, но не как их создавать из существующего RDD-набора. Для этого необходимо создать один объект `Row` для каждой записи в RDD-наборе и вызвать

метод `createDataFrame` контекста SQL. В самом конце полученный объект можно зарегистрировать как временную таблицу, что позволит в полной мере воспользоваться возможностями синтаксиса SQL:

```
In:
from pyspark.sql import Row

rdd_gender = \
    sc.parallelize([Row(short_gender="M", long_gender='Male'),
                    Row(short_gender="F", long_gender='Female')])

(sqlContext.createDataFrame(rdd_gender)
           .registerTempTable('gender_maps'))

In:
sqlContext.table("gender_maps").show()

Out:
+-----------+------------+
|long_gender|short_gender|
+-----------+------------+
|       Male|           M|
|     Female|           F|
+-----------+------------+
```

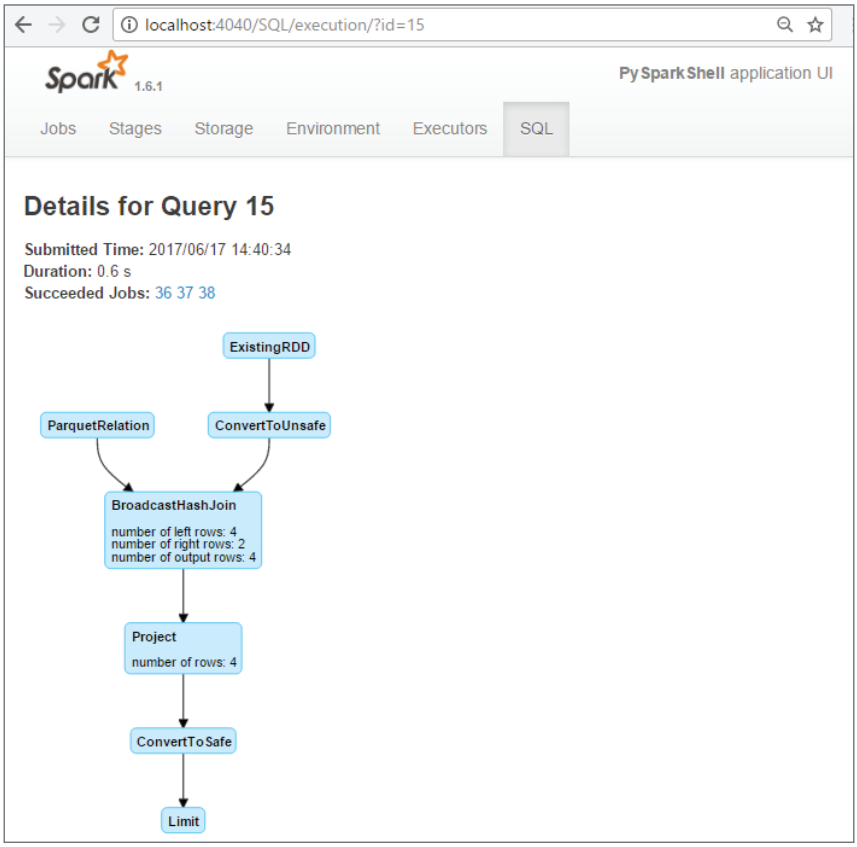
Этот способ, кроме того, является предпочтительным для работы с CSV-файлами. Сначала файл считывается методом `sc.textFile`; затем при помощи метода `split`, конструктора объекта `Row` и метода `createDataFrame` создается итоговый объект DataFrame.

Когда в оперативной памяти много объектов DataFrame или когда они могут быть загружены из диска, можно воспользоваться операциями, которые имеются в классической реляционной СУБД. В приведенном ниже примере созданный из RDD-набора объект DataFrame объединяется операцией `join` с набором данных `users`, содержащимся в сохраненном ранее Parquet-файле. Результат поразителен:

```
In:
sqlContext.sql("""
    SELECT balance, long_gender, user_id
    FROM parquet.`file:///tmp/complete_users.parquet`
    JOIN gender_maps ON gender=short_gender""").show()

Out:
+-------+-----------+-------+
|balance|long_gender|user_id|
+-------+-----------+-------+
|    3.0|       Male|      5|
|    1.0|       Male|      1|
|    0.0|     Female|      3|
|   -0.5|     Female|      2|
+-------+-----------+-------+
```

В веб-интерфейсе под вкладкой **SQL** каждый SQL-запрос отображается как виртуальный **направленный ациклический граф (DAG)**. Он замечательно помогает отслеживать процесс исполнения вашего задания и понять сложность запроса. При выполнении предыдущего SQL-запроса с оператором JOIN четко видно, что

два ответвления входят в одинаковый блок `BroadcastHashJoin`: первый – из RDD-набора, и второй – из Parquet-файла. Затем следующий блок попросту является проекцией на отобранные столбцы:



Поскольку таблицы находятся в оперативной памяти, последней является операция очистки с целью высвобождения памяти, используемой для их хранения. В результате вызова метода `tableNames`, предоставленного объектом `sqlContext`, мы получим список всех таблиц, которые в настоящее время находятся в памяти. Затем, чтобы высвободить место, можно воспользоваться методом `dropTempTable` с именем таблицы в качестве аргумента. После этого вызова дальнейшее обращение к этим таблицам вернет ошибку:

```
In:
sqlContext.tableNames()

Out:
[u'gender_maps', u'users']

In:
for table in sqlContext.tableNames():
    sqlContext.dropTempTable(table))
```

Начиная с версии Spark 1.3 объект DataFrame является предпочтительным способом работы с набором данных при выполнении задач в области науки о данных.

Машинное обучение с платформой Spark

Здесь мы подходим к главной цели вашей работы: созданию модели для предсказания одного или многочисленных атрибутов, отсутствующих в наборе данных. Для этого мы построим несколько моделей машинного обучения, и платформа Spark может оказать нам в этом контексте неоценимую помощь.

MLlib – это библиотека машинного обучения в рамках платформы Spark; несмотря на то что она написана на Scala и Java, ее функционал также доступен в среде Python. Она содержит ученики для выполнения задач классификации, регрессии и генерирования рекомендаций, несколько подпрограмм для снижения размерности и отбора признаков, имеет обширный функционал по обработке текста. Все они в состоянии справиться с огромными наборами данных и используют мощь всех узлов кластера для достижения поставленной цели.

На данный момент (2016 г.) эта библиотека состоит из двух основных библиотек: библиотеки `mllib`, которая оперирует на RDD-наборах, и библиотеки `ml`, которая работает на объектах DataFrame. Поскольку последняя из двух имеет хорошую производительность и является наиболее популярным способом представления данных в науке о данных, разработчики сделали выбор в пользу участия и усовершенствования ветки `ml`, позволив первой библиотеке остаться, как есть, но без дальнейшего развития. На первый взгляд, библиотека MLlib выглядит законченной, но, начав использовать платформу Spark, вы заметите, что в ее стандартном комплекте нет ни статистической, ни числовой библиотек. И здесь на выручку приходят библиотеки SciPy и NumPy, которые, повторимся, очень важны для науки о данных!

В этом разделе мы попытаемся разведать функционал *новой* библиотеки `pyspark.ml`; на данный момент она по-прежнему находится на ранних стадиях по сравнению с современным состоянием библиотеки Scikit-learn, но она определенно имеет большой потенциал для будущего.

Платформа Spark – это высокоуровневое, распределенное и многосложное программное обеспечение, которое должно использоваться только на больших данных и с кластером из многочисленных узлов; на самом деле если набор данных может уместиться в оперативной памяти, то удобнее пользоваться другими библиотеками, такими как Scikit-learn или ей подобными, в центре внимания которых находится только та сторона задачи, которая имеет отношение к науке о данных. Выполнение платформы Spark на единственном узле на небольшом наборе данных может быть в пять раз медленнее эквивалентного алгоритма в Scikit-learn.

Платформа Spark на наборе данных KDD99

Давайте выполним эту разведку с использованием реального набора данных. Набор данных KDD99 использовался для третьего международного соревнования по обнаружению знаний и инструментам глубинного анализа данных. Цель соревнования состояла в том, чтобы создать систему обнаружения вторжений в сеть (intrusion detection system, IDS), способную распознавать, какой сетевой поток является злонамеренным и какой нет. Более того, много разных атак находится

в самом наборе данных; цель состоит в том, чтобы точно их предсказать при помощи содержащихся в наборе данных признаков пакетного потока.

В качестве дополнения отметим, что указанный набор данных был чрезвычайно полезен для разработки отличных решений в области систем обнаружения вторжений в первые несколько лет после его выпуска. Сегодня, как результат, все включенные в набор данных атаки очень легко обнаруживаются, и поэтому для разработки IDS он больше не используется.

В частности, в указанном наборе данных представлены следующие признаки: протокол (tcp, icmp и udp), служба (http, smtp и т. д.), размер пакетов, активные в протоколе флаги, количество попыток стать корнем и т. д.

Дополнительную информацию о соревновании KDD99 и наборах данных можно найти на http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

Описанная выше задача является классической задачей многоклассовой классификации. Несмотря на это, мы остановимся на ней подробнее, чтобы показать, как выполнять эту задачу в платформе Spark. Чтобы все было чисто, мы воспользуемся новым блокнотом Jupyter.

Чтение набора данных

Прежде всего скачаем и распакуем этот набор данных. Мы будем очень консервативными и воспользуемся всего 10% исходного тренировочного набора данных (75 Мб в несжатом виде), так как весь анализ выполняется на небольшой виртуальной машине. Если есть желание попробовать, то в следующем ниже фрагменте кода можно раскомментировать строки и загрузить полный тренировочный набор данных (750 Мб в несжатом виде). Мы скачаем тренировочный набор данных, тестовый набор (47 Мб) и имена признаков при помощи команд bash:

```
In:
!rm -rf kdd*

# !wget -q -O ../datasets/kddtrain.gz \
# http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data.gz

!wget -q -O ../datasets/kddtrain.gz \
    http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data_10_percent.gz

!wget -q -O ../datasets/kddtest.gz \
    http://kdd.ics.uci.edu/databases/kddcup99/corrected.gz

!wget -q -O ../datasets/kddnames \
    http://kdd.ics.uci.edu/databases/kddcup99/kddcup.names

!gunzip ../datasets/kdd*gz
```

Теперь распечатаем первые несколько строк, что получить представление о его формате. Совершенно очевидно, что это классический CSV без заголовка и с точкой в конце каждой строки. Кроме того, мы видим, что большинство полей является числовыми, но несколько из них – текстовые, и целевая переменная содержится в последнем поле:

```
In:
!head -3 ../datasets/kddtrain
```

```
Out:
0,tcp,http,SF,181,5450,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,8,8,0.00,0.00,0.00,0.00,1.00,0.00,0.00,
9,9,1.00,0.00,0.11,0.00,0.00,0.00,0.00,0.00,normal.
0,tcp,http,SF,239,486,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,8,8,0.00,0.00,0.00,0.00,1.00,0.00,0.00,1
9,19,1.00,0.00,0.05,0.00,0.00,0.00,0.00,0.00,normal.
0,tcp,http,SF,235,1337,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,8,8,0.00,0.00,0.00,0.00,1.00,0.00,0.00,
29,29,1.00,0.00,0.03,0.00,0.00,0.00,0.00,0.00,normal.
```

Для создания объекта DataFrame с именованными полями сначала необходимо прочитать заголовок, включенный в файл `kddnames`. Целевое поле будет называться просто `target`.

Прочитав файл и выполнив его разбор, распечатаем количество признаков решаемой задачи (напомнив, что целевая переменная не является признаком) и первые 10 имен признаков:

```
In:
with open('../datasets/kddnames', 'r') as fh:
    header = [line.split(':')[0]
              for line in fh.read().splitlines()][1:]

header.append('target')

print('Число признаков: {0}'.format(len(header)-1))
print('Первые 10: {0}'.format(header[:10]))

Out:
Число признаков: 41
Первые 10: ['duration', 'protocol_type', 'service', 'flag', 'src_bytes', 'dst_bytes', 'land',
'wrong_fragment', 'urgent', 'hot']
```

Теперь создадим два отдельных RDD-набора – один для тренировочных данных и другой для тестовых:

```
In:
train_rdd = sc.textFile('file:///home/vagrant/datasets/kddtrain')
test_rdd = sc.textFile('file:///home/vagrant/datasets/kddtest')
```

Далее необходимо выполнить разбор каждой строки каждого файла для создания объекта DataFrame. Сначала разобьем каждую строку CSV-файла на отдельные поля и затем приведем каждое численное значение к типу с плавающей точкой и каждое текстовое значение к строковому типу. В заключение удалим точку в конце каждой строки.

В качестве последнего шага при помощи метода `createDataFrame`, предоставляемого объектом `sqlContext`, создадим два объекта DataFrame с именованными столбцами для тренировочного и тестового наборов данных:

```
In:
def line_parser(line):

    def piece_parser(piece):
            if "." in piece or piece.isdigit():
                return float(piece)
            else:
                return piece

    return [piece_parser(piece) for piece in line[:-1].split(',')]
```

```
train_df = sqlContext.createDataFrame(
    train_rdd.map(line_parser), header)

test_df = sqlContext.createDataFrame(
    test_rdd.map(line_parser), header)
```

До настоящего момента для RDD-наборов были написаны только преобразователи; теперь введем действие, чтобы увидеть, сколько наблюдений имеется в наборах данных, и одновременно проверим правильность предыдущего программного кода.

```
In:
print('Тренировочные наблюдения: {0}'.format(train_df.count()))
print('Тестовые наблюдения: {0}'.format(test_df.count()))

Out:
Тренировочные наблюдения: 494021
Тестовые наблюдения: 311029
```

Несмотря на то что мы используем всего одну десятую полного набора данных KDD99, все равно получается полмиллиона наблюдений. Умножив их на число признаков (41), ясно понимаем, что будем тренировать классификатор на матрице наблюдений, содержащей более 20 млн значений. Для платформы Spark это не такой уж и большой набор данных (кстати, и полный набор данных KDD99 тоже); разработчики по всему миру уже используют ее на петабайтах и миллиардах записей. Не пугайтесь, если числа выглядят большими: платформа Spark разработана для того, чтобы с ними справляться!

Теперь посмотрим, как он выглядит на схеме объекта DataFrame. Если быть точным, мы хотим установить, какие поля являются числовыми и какие содержат строковые значения (заметим, что результат для краткости сокращен):

```
In:
train_df.printSchema()

Out:
root
|-- duration: double (nullable = true)
|-- protocol_type: string (nullable = true)
|-- service: string (nullable = true)
|-- flag: string (nullable = true)
|-- src_bytes: double (nullable = true)
|-- dst_bytes: double (nullable = true)
|-- land: double (nullable = true)
|-- wrong_fragment: double (nullable = true)
|-- urgent: double (nullable = true)
|-- hot: double (nullable = true)
...
|-- target: string (nullable = true)
```

Конструирование признаков

Исходя из визуального анализа, только четыре поля являются строковыми: `protocol_type`, `service`, `flag` и цель `target` (которая, как ожидалось, является целевой мультиклассовой меткой).

Поскольку мы будем использовать древовидный классификатор, необходимо по каждой переменной закодировать текст каждого уровня в число. В Scikit-learn эта операция может быть выполнена при помощи объекта `sklearn.preprocessing.LabelEncoder`. Его эквивалентом в Spark является `StringIndexer` из модуля `pyspark.ml.feature`.

В среде Spark нужно закодировать четыре переменные; затем объединить в каскадную цепь четыре объекта `StringIndexer`: каждый из них будет оперировать определенным столбцом объекта DataFrame, выводя объект DataFrame с дополнительным столбцом (аналогично операции преобразования `map`). Преобразование выполняется автоматически и упорядочивает элементы по частоте: платформа Spark ранжирует счетчик каждого уровня в выбранном столбце, ставя в соответствие самому популярному уровню 0, следующему – 1 и т. д. Отметим, что, выполняя эту операцию, набор данных будет пройден один раз для подсчета случаев каждого уровня. Если бы соответствие было уже известно, то было бы гораздо эффективнее его распространить по всем узлам и применить операцию `map`, как было показано в начале настоящей главы.

Аналогичным образом можно воспользоваться прямым унитарным кодировщиком для генерирования числовой матрицы наблюдений. В случае прямого кодировщика в объекте DataFrame были бы многочисленные выходные столбцы, один для каждого уровня каждого категориального признака. Для этой цели платформа Spark предлагает класс `pyspark.ml.feature.OneHotEncoder`.

В более общем плане все содержащиеся в библиотеке `pyspark.ml.feature` классы используются для извлечения, преобразования и отбора признаков из объекта DataFrame. Все они *читают* столбцы и *создают* другие столбцы в объекте DataFrame.

Начиная с версии Spark 1.6 имеющиеся в Python операции на признаках содержатся в следующем ниже исчерпывающем списке (их все можно найти в библиотеке `pyspark.ml.feature`). Имена операций должны быть интуитивно понятными, за исключением пары из них, которые будут объяснены во фрагментах кода или позднее в тексте.

- Для текстовых входных данных (идеально):
  - HashingTF и IDF для векторизации признаков с хэшированием;
  - Tokenizer и его реализация RegexTokenizer на основе регулярных выражений для лексемизации текста;
  - Word2vec для векторизации слов;
  - StopWordsRemover для удаления стоп-слов;
  - N-gramm для выделения *n*-грамм.
- Для категориальных признаков:
  - строковый индексатор StringIndexer и обратный ему кодировщик IndexToString;
  - прямой унитарный кодировщик OneHotEncoder;
  - VectorIndexer (нестандартный индексатор с переводом из категорий в числа).
- Для других входных данных:
  - Binarizer для двоичного преобразования;
  - PCA для анализа главных компонент;
  - PolynomialExpansion для полиномиального разложения;

  - нормализатор Normalizer, стандартный шкалировщик StandardScaler и минимаксный шкалировщик MinMaxScaler;
  - Bucketizer (упаковщик значений признака);
  - ElementwiseProduct (перемножает столбцы) для поэлементного умножения.
- Обобщенные:
  - SQLTransformer (реализует преобразования, определенные SQL-оператором, обращаясь к объекту DataFrame как таблице под названием __THIS);
  - RFormula (выбирает столбцы, используя синтаксические конструкции в стиле языка R);
  - VectorAssembler (создает вектор признаков из многочисленных столбцов).

Возвращаясь к примеру, теперь закодируем уровни в каждой категориальной переменной дискретными числами. Как уже объяснялось выше, для этого по каждой переменной применяется объект `StringIndexer`. Более того, воспользуемся конвейером машинного обучения и установим их в качестве его этапов.

Затем, чтобы выполнить подгонку все индексаторов, нужно просто вызвать метод `fit` конвейера. Внутренне он последовательно выполнит подгонку всех поэтапных объектов. Когда он завершит операцию подгонки, будет создан новый объект, к которому можно обращаться как подогнанному конвейеру. Вызов метода `transform` этого нового объекта последовательно вызовет все поэтапные элементы (которые уже подогнаны), при этом каждый будет вызываться после того, как предыдущий завершен. В приведенном ниже фрагменте кода можно увидеть конвейер в действии. Отметим, что конвейер состоит из преобразователей. И поэтому на самом деле ничего не выполняется, так как действия отсутствуют. В выходном объекте DataFrame будут четыре дополнительных столбца под теми же самыми именами, что и исходные категориальные, но с суффиксом `_cat`:

```
In:
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer

cols_categorical = ['protocol_type', 'service', 'flag', 'target']
preproc_stages = []

for col in cols_categorical:
    out_col = col + '_cat'
    preproc_stages.append(
        StringIndexer(
            inputCol=col, outputCol=out_col, handleInvalid='skip'))

pipeline = Pipeline(stages=preproc_stages)
indexer = pipeline.fit(train_df)

train_num_df = indexer.transform(train_df)
test_num_df = indexer.transform(test_df)
```

Давайте исследуем конвейер чуть подробнее. В конвейере можно увидеть этапы: неподогнанный конвейер и подогнанный конвейер. Отметим, что существует большая разница между платформой Spark и библиотекой Scikit-learn: в Scikit-learn методы `fit` и `transform` вызываются на одном и том же объекте, в платформе Spark метод `fit` генерирует новый объект (обычно его имя дополняется суффик-

сом Model, так же, как для Pipeline и PipelineModel), в котором можно вызвать метод transform. Эта разница вытекает из замыканий – подогнанный объект проще распределять по всем процессам и всему кластеру:

```
In:
print(pipeline.getStages())
print()
print(pipeline)
print(indexer)

Out:
[StringIndexer_4103bd5fa259e598be03, StringIndexer_436da471ddb860cb168f, StringIndexer_485b8e5
8313de366d730, StringIndexer_4b6d961bf2d526a76f75]
()
Pipeline_4e94a8cb25b9b64db564
PipelineModel_4ae799b82c588eb50907
```

Давайте посмотрим, как изменяется первое наблюдение, т. е. первая строка в CSV-файле, после прохождения через конвейер. Отметим, что здесь используется действие, поэтому все этапы в конвейере и в конвейерной модели исполняются:

```
In:
print('Первое наблюдение после 4 индексаторов StringIndexer:\n')
print(train_num_df.first())

Out:
Первое наблюдение после 4 индексаторов StringIndexer:

Row(duration=0.0, protocol_type=u'tcp', service=u'http', flag=u'SF', src_bytes=181.0, dst_
bytes=5450.0, land=0.0, wrong_fragment=0.0, urgent=0.0, hot=0.0, num_failed_logins=0.0,
logged_in=1.0, num_compromised=0.0, root_shell=0.0, su_attempted=0.0, num_root=0.0, num_
file_creations=0.0, num_shells=0.0, num_access_files=0.0, num_outbound_cmds=0.0, is_host_
login=0.0, is_guest_login=0.0, count=8.0, srv_count=8.0, serror_rate=0.0, srv_serror_rate=0.0,
rerror_rate=0.0, srv_rerror_rate=0.0, same_srv_rate=1.0, diff_srv_rate=0.0, srv_diff_host_
rate=0.0, dst_host_count=9.0, dst_host_srv_count=9.0, dst_host_same_srv_rate=1.0, dst_
host_diff_srv_rate=0.0, dst_host_same_src_port_rate=0.11, dst_host_srv_diff_host_rate=0.0,
dst_host_serror_rate=0.0, dst_host_srv_serror_rate=0.0, dst_host_rerror_rate=0.0, dst_host_
srv_rerror_rate=0.0, target=u'normal', protocol_type_cat=1.0, service_cat=2.0, flag_cat=0.0,
target_cat=2.0)
```

Получившийся объект DataFrame выглядит вполне завершенным и понятным: все переменные имеют имена и значения. Мы сразу отмечаем, что категориальные признаки по-прежнему присутствуют, например protocol_type (категориальная переменная) и protocol_type_cat (числовая версия переменной, полученная из категориальной).

Несколько столбцов из объекта DataFrame можно извлечь так же просто, как и при использовании оператора SELECT в SQL-запросе. Теперь сконструируем список названий всех числовых признаков: начиная с найденных в заголовке имен, удаляем категориальные и заменяем их производными числовыми версиями. В заключение удалим целевую переменную и производный от нее числовой эквивалент, так как требуются только признаки:

```
In:
features_header = set(header) \
                - set(cols_categorical) \
                | set([c + "_cat" for c in cols_categorical]) \
                - set(["target", "target_cat"])
features_header = list(features_header)

print(features_header)
print('Всего числовых признаков: {0}'.format(len(features_header)))

Out:
['num_access_files', 'src_bytes', 'srv_count', 'num_outbound_cmds', 'rerror_rate', 'urgent',
'protocol_type_cat', 'dst_host_same_srv_rate', 'duration', 'dst_host_diff_srv_rate', 'srv_
serror_rate', 'is_host_login', 'wrong_fragment', 'serror_rate', 'num_compromised', 'is_guest_
login', 'dst_host_rerror_rate', 'dst_host_srv_serror_rate', 'hot', 'dst_host_srv_count',
'logged_in', 'srv_rerror_rate', 'dst_host_srv_diff_host_rate', 'srv_diff_host_rate', 'dst_
host_same_src_port_rate', 'root_shell', 'service_cat', 'su_attempted', 'dst_host_count',
'num_file_creations', 'flag_cat', 'count', 'land', 'same_srv_rate', 'dst_bytes', 'num_shells',
'dst_host_srv_rerror_rate', 'num_root', 'diff_srv_rate', 'num_failed_logins', 'dst_host_
serror_rate']
Всего числовых признаков: 41
```

Ниже на помощь приходит класс `VectorAssembler`, который создает признаковую матрицу. В качестве аргумента в этот класс нужно передать отбираемые столбцы и создаваемый в объекте DataFrame новый столбец. Мы решаем назвать выходной столбец просто `features` и применяем это преобразование к обоим наборам: тренировочному и тестовому, а затем отбираем только два интересующих нас столбца – `features` и `target_cat`:

```
In:
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(inputCols=features_header,
                            outputCol='features')

Xy_train = (assembler.transform(train_num_df)
            .select('features', 'target_cat'))
Xy_test = (assembler.transform(test_num_df)
           .select('features', 'target_cat'))
```

Кроме того, поведение векторного сборщика `VectorAssembler` по умолчанию сводится к созданию плотных или разреженных векторов, соответственно `DenseVector` или `SparseVector`. В данном случае сборщик возвращает разреженный вектор, поскольку вектор `features` содержит много нулей. Чтобы посмотреть, что находится внутри конечного результата, можно распечатать первую строку. Отметим, что это будет действием. Следовательно, задание исполнится прежде, чем будет получен распечатанный результат:

```
In:
Xy_train.first()

Out:
Row(features=SparseVector(41, {1: 181.0, 2: 8.0, 6: 1.0, 7: 1.0, 19: 9.0, 20: 1.0, 24: 0.11,
26: 2.0, 28: 9.0, 31: 8.0, 33: 1.0, 34: 5450.0}), target_cat=2.0)
```

Тренировка ученика

Наконец, мы достигли самой «заводной» части задачи: тренировки классификатора. Классификаторы содержатся в библиотеке `pyspark.ml.classification`, и для данного примера мы используем случайный лес.

Начиная с версии Spark 1.6 расширен список классификаторов, в которых используется интерфейс Python. Ниже приводим этот список.

- Классификация (библиотека `pyspark.ml.classification`):
  - `LogisticRegression` для классификации с использованием логистической регрессии;
  - `DecisionTreeClassifier` для классификации с использованием деревьев решений;
  - `GBTClassifier` для классификации с использованием деревьев решений (реализация градиентного бустинга);
  - `RandomForestClassifier` для классификации с использованием случайного леса;
  - `NaiveBayes` для классификации с использованием наивного Байеса;
  - `MultilayerPerceptronClassifier` для классификации с использованием многослойного персептрона.

Отметим, что не все из них способны обрабатывать мультиклассовые задачи, и они могут иметь отличающиеся параметры; следует всегда проверять документацию, связанную с используемой версией. Помимо классификаторов, в Spark 1.6 также реализованы и другие ученики с интерфейсом Python. Ниже приводим их список.

- Кластеризация (библиотека `pyspark.ml.clustering`):
  - KMeans для кластеризации методом K-средних.
- Регрессия (библиотека `pyspark.ml.regression`):
  - AFTSurvivalRegression для регрессии с использованием модели ускоренного отказа (на основе анализа выживаемости);
  - DecisionTreeRegressor для регрессии с использованием деревьев решений;
  - GBTRegressor для регрессии с использованием регрессионных деревьев (реализация градиентного бустинга);
  - IsotonicRegression для изотонической регрессии;
  - LinearRegression для линейной регрессии;
  - RandomForestRegressor для регрессии с использованием случайного леса.
- Рекомендатель (библиотека `pyspark.ml.recommendation`):
  - ALS (рекомендательная система с использованием коллаборативной фильтрации на основе чередующихся наименьших квадратов).

Давайте вернемся к цели задачи KDD99. Теперь пора инстанцировать классификатор на основе случайного леса и задать его параметры. Такими параметрами являются `featuresCol` (столбец, содержащий признаковую матрицу), `labelCol` (столбец объекта DataFrame, содержащий целевую метку), `seed` (начальное число для генератора случайных чисел, чтобы обеспечить воспроизводимость эксперимента) и `maxBins` (максимальное число дискретных интервалов, используемых для точки расщепления в каждом узле дерева). Значение числа деревьев в лесе по умолчанию равно 20, и каждое дерево имеет максимум пять уровней в глубину. Кроме

того, этот классификатор по умолчанию создает три выходных столбца в объекте DataFrame: `rawPrediction` (для хранения показателя точности предсказания для каждой возможной метки), `probability` (для хранения вероятности каждой метки) и `prediction` (наиболее вероятная метка):

```
In:
from pyspark.ml.classification import RandomForestClassifier

clf = RandomForestClassifier(labelCol='target_cat',
                             featuresCol='features',
                             maxBins=100, seed=101)
fit_clf = clf.fit(Xy_train)
```

Однако даже и в этом случае натренированный классификатор будет представлен другим объектом. Как и ранее, натренированный классификатор имеет то же самое имя, что и классификатор плюс суффикс `Model`:

```
In:
print(clf)
print(fit_clf)

Out:
RandomForestClassifier_40cca1c5d6cb00c6099b
RandomForestClassificationModel (uid=rfc_11eebe9e9fa6) with 20 trees
```

На объекте обученного классификатора `RandomForestClassificationModel` можно вызвать метод `transform`. Теперь предскажем метку на тренировочном и тестовом наборах данных и распечатаем первую строку тестового набора данных; как определено в классификаторе, предсказания будут находиться в столбце `prediction`:

```
In:
Xy_pred_train = fit_clf.transform(Xy_train)
Xy_pred_test = fit_clf.transform(Xy_test)


In:
print('Первое наблюдение после этапа классификации:')
print(Xy_pred_test.first())

Out:
Первое наблюдение после этапа классификации:
Row(features=SparseVector(41, {1: 105.0, 2: 1.0, 6: 2.0, 7: 1.0, 9: 0.01, 19: 254.0, 26: 1.0,
28: 255.0, 31: 1.0, 33: 1.0, 34: 146.0}), target_cat=2.0, rawPrediction=DenseVector([0.0283,
0.0112, 19.3474, 0.0677, 0.0251, 0.1414, 0.0357, 0.1194, 0.1309, 0.041, 0.0257, 0.0079,
0.0046, 0.0004, 0.0029, 0.0016, 0.002, 0.0023, 0.0013, 0.0008, 0.0012, 0.0006, 0.0006]), pro
bability=DenseVector([0.0014, 0.0006, 0.9674, 0.0034, 0.0013, 0.0071, 0.0018, 0.006, 0.0065,
0.002, 0.0013, 0.0004, 0.0002, 0.0, 0.0001, 0.0001, 0.0001, 0.0001, 0.0001, 0.0, 0.0001, 0.0,
0.0]), prediction=2.0)
```

Оценка результативности ученика

Следующий шаг в любой задаче науки о данных состоит в проверке результативности ученика на тренировочном и тестовом наборах. Для этой задачи мы воспользуемся метрикой F1, поскольку она удачно объединяет показатели результативности – прецизионность и полноту.

Метрики оценивания включены в библиотеку `pyspark.ml.evaluation`; среди нескольких вариантов для выбора мы используем вариант с оценкой мультиклассовых классификаторов: `MulticlassClassificationEvaluator`. В качестве параметров мы предоставляем метрику (прецизионность, полноту, точность, метрику F1 и т. д.) и имена столбцов, содержащих истинную и предсказанную метки:

```
In:
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(labelCol='target_cat',
                                              predictionCol='prediction',
                                              metricName='f1')

print('Отметка F1 на тренировочном наборе: {0}'
      .format(evaluator.evaluate(Xy_pred_train)))
print('Отметка F1 на тестовом наборе: {0}'
      .format(evaluator.evaluate(Xy_pred_test)))

Out:
Отметка F1 на тренировочном наборе: 0.991904372002
Отметка F1 на тестовом наборе: 0.966840043466
```

Полученные значения довольно высокие, и между результативностью на тренировочном и тестовом наборах существует большая разница.

Помимо оценщика мультиклассовых классификаторов, в той же самой библиотеке имеется объект с оценщиком регрессора (где метриками могут быть среднеквадратическая ошибка MSE, корень из среднеквадратической ошибки RMSE, R-квадрат либо средняя абсолютная ошибка MAE) и бинарных классификаторов.

Возможности конвейера машинного обучения

До сих пор результаты создавались и выводились поэтапно. Помимо этого, можно также помещать все операции в каскад и задавать их в качестве этапов конвейера. По сути дела, в автономный конвейер можно связать цепочкой все, что мы видели до сих пор (четыре кодировщика меток, векторный построитель и классификатор), выполнить его подгонку на тренировочном наборе данных и, наконец, применить его на тестовом наборе данных для получения прогнозов.

Этот способ работы более эффективен, но при этом теряются разведочные возможности постепенного анализа. Читателям, занимающимся аналитикой данных, рекомендуется использовать сквозные конвейеры, только когда они абсолютно уверены в том, что происходит внутри, и только для построения серийных (эксплуатационных) моделей.

Чтобы показать, что конвейер эквивалентен тому, что мы видели до сих пор, вычислим отметку F1 на тестовом наборе и ее распечатаем. Неудивительно, что значение будет в точности таким же:

```
In:
full_stages = preproc_stages + [assembler, clf]
full_pipeline = Pipeline(stages=full_stages)
full_model = full_pipeline.fit(train_df)
predictions = full_model.transform(test_df)
print('Отметка F1 на тестовом наборе: {0}'
      .format(evaluator.evaluate(predictions)))
```

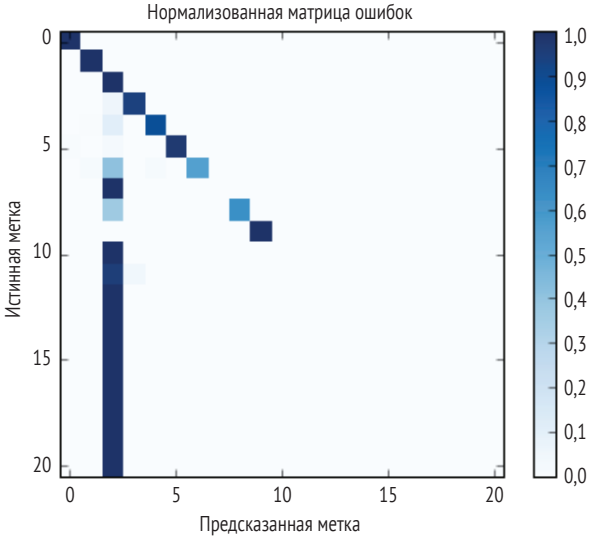
```
Out:
Отметка F1 на тестовом наборе: 0.966840043466
```

На драйверном узле, т. е. на том, который выполняет блокнот Jupyter, можно также использовать библиотеку `matplotlib` для визуализации результатов анализа. Например, чтобы показать нормализованную матрицу ошибок результатов классификации (нормализованную поддержкой каждого класса), можно создать следующую функцию:

```
In:
import numpy as np

def plot_confusion_matrix(cm):
    cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    plt.imshow(
        cm_normalized, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title(u'Нормализованная матрица ошибок')
    plt.colorbar()
    plt.tight_layout()
    plt.ylabel(u'Истинная метка')
    plt.xlabel(u'Предсказанная метка')
```

Платформа Spark способна создавать матрицу ошибок, но соответствующий метод находится в библиотеке `pyspark.mllib`. Чтобы иметь возможность использовать методы из этой библиотеки, необходимо при помощи метода `rdd` преобразовать объект DataFrame в RDD-набор:

```
In:
from pyspark.mllib.evaluation import MulticlassMetrics

metrics = MulticlassMetrics(predictions.select('prediction',
                                               'target_cat').rdd)
conf_matrix = metrics.confusionMatrix().toArray()
plot_confusion_matrix(conf_matrix)
```



Ручная доводка

Несмотря на то что отметка F1 приблизилась к 0.97, нормализованная матрица ошибок показывает, что классы сильно разбалансированы, и классификатор только что обучился правильно классифицировать самые популярные из них. Для улучшения результатов можно сделать перевыборку каждого класса в попытке получше сбалансировать тренировочный набор данных.

Сначала подсчитаем количество случаев в тренировочном наборе данных для каждого класса:

```
In:
train_composition = train_df.groupBy("target").count().rdd.collectAsMap()
train_composition

Out:
{u'back': 2203,
 u'buffer_overflow': 30,
 u'ftp_write': 8,
 u'guess_passwd': 53,
 u'neptune': 107201,
 u'nmap': 231,
 u'normal': 97278,
 u'perl': 3,
 ...
 u'warezmaster': 20}
```

Мы получили явное доказательство сильного дисбаланса. Можно попробовать улучшить результативность путем отбора с повышенной частотой только *редких* классов и отбора с пониженной частотой слишком популярных классов.

В приведенном ниже примере будет создан тренировочный набор данных, где каждый класс представлен, по крайней мере, 1000 раз, но не более 25 000 раз. Для этого сначала рассчитаем уровень пониженной/повышенной частоты отбора наблюдений и распространим этот уровень по всему кластеру, а затем обработаем каждую строку тренировочного набора данных методом `flatMap`, чтобы выполнить его тщательную перевыборку:

```
In:
def set_sample_rate_between_vals(cnt, the_min, the_max):
    if the_min <= cnt <= the_max:
        # без выборки
        return 1

    elif cnt < the_min:
        # Выборка с повышенной частотой (oversampling)
        # вернуть много раз одно и то же наблюдение
        return the_min/float(cnt)

    else:
        # Выборка с пониженной частотой (subsampling):
        # иногда наблюдение не будет возвращено
        return the_max/float(cnt)

sample_rates = {k:set_sample_rate_between_vals(v, 1000, 25000)
                for k,v in train_composition.iteritems()}
```

```
sample_rates
```

```
Out:
{u'back': 1,
 u'buffer_overflow': 33.333333333333336,
 u'ftp_write': 125.0,
 u'guess_passwd': 18.867924528301888,
 u'neptune': 0.23320677978750198,
 u'nmap': 4.329004329004329,
 u'normal': 0.2569954152017928,
 u'perl': 333.3333333333333,
 ...
 u'warezmaster': 50.0}
```

```
In:
bc_sample_rates = sc.broadcast(sample_rates)
```

```
def map_and_sample(el, rates):
    rate = rates.value[el['target']]
    if rate > 1:
        return [el]*int(rate)
    else:
        import random
        return [el] if random.random() < rate else []
```

```
sampled_train_df = (train_df
                    .flatMap(
                      lambda x: map_and_sample(x, bc_sample_rates))
                    .toDF()
                    .cache())
```

Перевыбранный набор данных в переменной объекта DataFrame `sampled_train_df` также кэшируется; мы будем использовать ее много раз во время шага гиперпараметрической оптимизации. Он должен легко уместиться в оперативной памяти, поскольку число строк в этом наборе данных ниже исходного:

```
In:
sampled_train_df.count()
```

```
Out:
96015
```

Чтобы понять, что находится внутри, можно распечатать первую строку. Распечатка значения выполняется заметно быстро. Разумеется, ведь набор кэширован!

```
In:
sampled_train_df.first()
```

```
Out:
Row(duration=0.0, protocol_type=u'tcp', service=u'http', flag=u'SF', src_bytes=217.0, dst_
bytes=2032.0, land=0.0, wrong_fragment=0.0, urgent=0.0, hot=0.0, num_failed_logins=0.0,
logged_in=1.0, num_compromised=0.0, root_shell=0.0, su_attempted=0.0, num_root=0.0, num_file_
creations=0.0, num_shells=0.0, num_access_files=0.0, num_outbound_cmds=0.0, is_host_login=0.0,
is_guest_login=0.0, count=6.0, srv_count=6.0, serror_rate=0.0, srv_serror_rate=0.0, rerror_
rate=0.0, srv_rerror_rate=0.0, same_srv_rate=1.0, diff_srv_rate=0.0, srv_diff_host_rate=0.0,
dst_host_count=49.0, dst_host_srv_count=49.0, dst_host_same_srv_rate=1.0, dst_host_diff_
```

```
srv_rate=0.0, dst_host_same_src_port_rate=0.02, dst_host_srv_diff_host_rate=0.0, dst_host_
serror_rate=0.0, dst_host_srv_serror_rate=0.0, dst_host_rerror_rate=0.0, dst_host_srv_rerror_
rate=0.0, target=u'normal')
```

Теперь воспользуемся созданным конвейером, чтобы выполнить несколько предсказаний и распечатать отметку F1 этого нового решения:

```
In:
full_model = full_pipeline.fit(sampled_train_df)
predictions = full_model.transform(test_df)

print('Отметка F1 на тестовом наборе: {0}'
      .format(evaluator.evaluate(predictions)))

Out:
Отметка F1 на тестовом наборе: 0.967037720279
```

Протестируем его на классификаторе из 50 деревьев. Для этого построим еще один конвейер (под именем `refined_pipeline`) и заменим заключительный этап новым классификатором. Показатели результативности выглядят одинаковыми, даже несмотря на то что тренировочный набор был резко сокращен:

```
In:
clf = RandomForestClassifier(
    numTrees=50, maxBins=100, seed=101,
    labelCol='target_cat', featuresCol='features')

stages = full_pipeline.getStages()[:-1]
stages.append(clf)

refined_pipeline = Pipeline(stages=stages)

refined_model = refined_pipeline.fit(sampled_train_df)
predictions = refined_model.transform(test_df)

print('Отметка F1 на тестовом наборе: {0}'
      .format(evaluator.evaluate(predictions)))

Out:
Отметка F1 на тестовом наборе: 0.96774867423
```

Перекрестная проверка

Мы можем обойтись ручной оптимизацией и найти правильную модель после исчерпывающего испытания разных конфигураций. Однако такой режим работы приведет к огромной потере времени (и возможности многократного использования программного кода) и вызовет переподгонку тестового набора данных. С другой стороны, для гиперпараметрической оптимизации предназначена перекрестная проверка. Давайте посмотрим, как платформа Spark выполняет эту принципиально важную задачу.

Прежде всего, поскольку тренировка будет использоваться многократно, ее результаты можно кэшировать. Поэтому занесем их в кэш после всех преобразований:

```
In:
pipeline_to_clf = Pipeline(
    stages=preproc_stages + [assembler]).fit(sampled_train_df)
```

```
train = pipeline_to_clf.transform(sampled_train_df).cache()
test = pipeline_to_clf.transform(test_df)
```

Классы, используемые для гиперпараметрической оптимизации посредством перекрестной проверки, содержатся в библиотеке `pyspark.ml.tuning`. При этом крайне важны два элемента: карта-сетка параметров (которая может быть сконструирована при помощи `ParamGridBuilder`), и фактическая процедура перекрестной проверки (выполняемая классом `CrossValidator`).

В данном примере нужно установить несколько параметров классификатора, которые не изменятся во время перекрестной проверки. Они устанавливаются в точности как в Scikit-learn, т. е. когда создается объект классификации (в данном случае имена столбцов, начальное значение для генератора случайных чисел и максимальное число дискретных интервалов).

Затем благодаря сеточному построителю мы выбираем, какие параметры должны быть изменены в каждой итерации алгоритма перекрестной проверки. В данном примере нужно проверить результативность классификации, изменяя максимальную глубину каждого дерева в лесу с 3 до 12 (с приращением 3) и число деревьев в лесу до 20 или 50.

В конце, после того как будут заданы карта-сетка, тестируемый классификатор и число блоков, мы запускаем перекрестную проверку (методом `fit`). Оценщик параметров `estimatorParamMaps` играет крайне важную роль: он сообщает, какую модель лучше всего сохранить после перекрестной проверки. Отметим, что выполнение этой работы может занять 15–20 минут (за кадром выполняются тренировка и тестирование *4*2*3 = 24* моделей):

```
In:
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Может занять порядка 10 минут
rf = RandomForestClassifier(cacheNodeIds=True, seed=101,
                            labelCol='target_cat',
                            featuresCol='features',
                            maxBins=100)

grid = (ParamGridBuilder()
        .addGrid(rf.maxDepth, [3, 6, 9, 12])
        .addGrid(rf.numTrees, [20, 50])
        .build())

cv = CrossValidator(estimator=rf, estimatorParamMaps=grid,
                    evaluator=evaluator, numFolds=3)
cvModel = cv.fit(train)
```

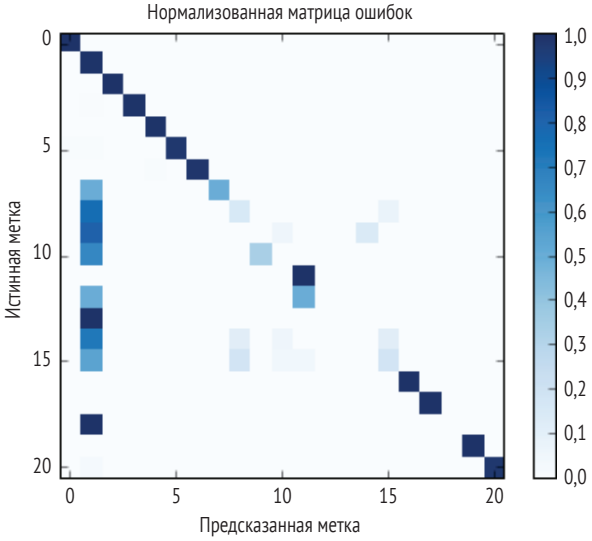
Наконец, при помощи перекрестно проверенной модели можно предсказать метку, используя конвейер либо непосредственно сам классификатор. В этом случае результативность выбранного перекрестной проверкой классификатора немного лучше, чем в предыдущем случае, позволяя преодолеть отметку 0.97:

```
In:
predictions = cvModel.transform(test)

print('Отметка F1 на тестовом наборе: {0}'
      .format(evaluator.evaluate(predictions)))
```

```
Out:
Отметка F1 на тестовом наборе: 0.970034862617
```

Более того, построив график нормализованной матрицы ошибок, также сразу становится понятным, что это решение способно обнаруживать более широкое разнообразие атак и даже менее распространенные из них:

```
In:
metrics = MulticlassMetrics(predictions.select('prediction',
                                               'target_cat').rdd)
conf_matrix = metrics.confusionMatrix().toArray()
plot_confusion_matrix(conf_matrix)
```



Заключительная очистка

Здесь мы подходим к концу задачи классификации. Напомним о необходимости удалить все использованные переменные и созданную из кэша временную таблицу:

```
In:
bc_sample_rates.unpersist()
sampled_train_df.unpersist()
train.unpersist()
```

После того как память платформы Spark очищена, можно выключить блокнот.

Резюме

Это была последняя глава книги. Мы увидели способы решения задач в области науки о данных в крупном масштабе на кластере машин. Платформа Spark способна тренировать и тестировать алгоритмы машинного обучения с использованием

всех узлов в кластере при помощи простого интерфейса, который очень похож на Scikit-learn. Это программное решение подтвердило свою состоятельность во время работы с петабайтами информации, создавая действенную альтернативу подвыборке из набора наблюдений и онлайновому обучению.

Чтобы стать экспертом по платформе Spark и потоковой обработке данных, мы настоятельно рекомендуем прочесть книгу *Mastering Apache Spark, Mike Frampton, Packt Publishing*.

Если вы достаточно храбры, чтобы перейти на язык Scala, который является основным языком программирования для среды Spark, то следующая книга будет самой подходящей для такого перехода: *Scala for Data Science, Pascal Bugnion, Packt Publishing*.

Приложение

Введение в графические процессоры и платформа Theano

До сих пор мы генерировали нейронные сети и выполняли задачи глубокого обучения, используя обычные центральные процессоры (CPU). Однако в последнее время широкое распространение получили вычислительные преимущества графических процессоров (GPU). Данное приложение посвящено основам GPU и их применению на платформе Theano, разработанной для решения задач в области глубокого обучения.

Вычисления на GPU

Когда мы используем библиотеки для машинного обучения с вычислениями на обычных CPU, в частности библиотеку Scikit-learn, объем параллелизации необычайно ограничен, потому что алгоритм по умолчанию использует всего одно ядро, даже когда в наличии имеются многочисленные ядра. В главе о **классификационных и регрессионных деревьях** (CART) показаны некоторые продвинутые примеры ускорения алгоритмов Scikit-learn.

В отличие от CPU, модули GPU с самого начала были разработаны, чтобы работать в параллельном режиме. Представим проецирование изображения на экран через графическую карту. Ни для кого не является секретом, что модуль GPU должен уметь одновременно обрабатывать и проецировать большой объем информации (движение, цвет и пространственность). Модули CPU, с другой стороны, разработаны для последовательной обработки, подходящей для задач, где необходимо больше контроля, в частности ветвление и проверка. В отличие от CPU, модули GPU состоят из большого количества ядер, которые могут обрабатывать тысячи задач одновременно. GPU может превзойти CPU в 100 раз и по более низкой цене. Еще одно преимущество современных модулей GPU состоит в том, что они являются относительно дешевыми, по сравнению с ультрасовременными модулями CPU.

Таким образом, все это выглядит великолепно, но помните, что модуль GPU хорош только при выполнении определенного типа задач. Модуль CPU состоит из нескольких ядер, оптимизированных для последовательной серийной обработки, в то время как GPU состоит из тысяч более мелких и более эффективных ядер, предназначенных для обработки задач параллельно.

Модули CPU и GPU имеют разную архитектуру, которая делает их более подходящими для разных задач. Однако существует много задач, таких как проверка, отладка и переключение, где модули GPU по-прежнему неспособны работать эффективно вследствие своей архитектуры.

Простой способ понять различие между CPU и GPU состоит в том, чтобы сравнить то, как они обрабатывают задачи. Часто приводится аналогия с аналитическим и последовательным левым полушарием мозга (CPU) и всеобъемлющим правым полушарием (GPU). Правда, эту простую аналогию не стоит принимать слишком серьезно.

GPU	CPU
Большое число ядер, но медленнее ядер CPU	Малое число ядер, но намного быстрее ядер GPU
Высокая пропускная способность памяти для управления ядрами	Меньшая пропускная способность памяти
Целевого назначения	Общего назначения
Высокопараллельная обработка	Последовательная обработка

О GPU можно узнать больше, перейдя по следующим ссылкам:

- http://www.nvidia.com/object/what-is-gpu-computing.html;
- http://www.nvidia.com/object/gpu-applications-domain.html.

Для того чтобы задействовать модули GPU для машинного обучения, требуется особая платформа. К сожалению, на данный момент, кроме платформы CUDA[1], других стабильных платформ для вычислений на GPU не существует; это означает, что на вашем компьютере должна быть установлена графическая карта NVIDIA. Вычисления на GPU работать без карты NVIDIA НЕ будут. Да, мы знаем, что это плохие новости для большинства пользователей компьютеров Mac, и нам действительно жаль, что не выходит по-другому, но с этим ограничением необходимо смириться. Существуют другие проекты, в частности OpenCL, которые предоставляют вычисления на GPU для других брендов GPU посредством таких инициатив, как **BLAS** (https://github.com/clMathLibraries/clBLAS), но они находятся в стадии интенсивной разработки и не полностью оптимизированы для приложений глубокого обучения в среде Python. Еще одно ограничение проекта OpenCL состоит в том, что в нем активно участвует только компания AMD, в результате чего он будет выгоден исключительно для модулей GPU производства AMD. В ближайшие годы (и даже десятилетия!) нет никакой надежды на появление аппаратно-независимого приложения на основе GPU для машинного обучения. Тем не менее стоит последить за новостями и разработками проекта OpenCL (https://www.khronos.org/opencl/). Судя по широким откликам средств массовой информации, это ограничение в отношении доступности GPU может быть весьма разочаровывающим. По всей видимости, только компания NVIDIA ведет научно-исследовательские работы с целью разработки платформ GPU, и крайне маловероятно, что мы увидим какие-либо новые серьезные наработки в этой области в ближайшие годы.

---

[1] CUDA (англ. Compute Unified Device Architecture) – программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia. – *Прим. перев.*

Для использования платформы CUDA понадобится следующее.

Необходимо проверить, подходит ли графическая карта на компьютере для платформы CUDA. Как минимум, это должна быть карта NVIDIA. Можно протестировать пригодность модуля GPU для CUDA при помощи следующих команд в окне терминала:

```
$ su
```

Находясь в корне, теперь наберите свой пароль:

```
$ lspci | grep-i Nvidia
```

Если модуль GPU действительно основан на NVIDIA, то можно скачать инструментарий NVIDIA CUDA Toolkit (http://developer.nvidia.com/cuda-downloads).

Во время работы над этой книгой компания NVIDIA вот-вот должна была выпустить платформу CUDA версии 8, в которой предусмотрен другой порядок инсталляции, поэтому мы рекомендуем следовать указаниям на веб-сайте CUDA. Для дальнейших процедур инсталляции проконсультируйтесь с веб-сайтом NVIDIA: http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/#axzz3xBimv9ou.

Платформа Theano – параллельные вычисления на GPU

Библиотека Python `Theano` была первоначально разработана Джеймсом Бергстра (James Bergstra) в Монреальском университете. Она призвана обеспечить более выразительные способы написания математических функций при помощи средств символического представления (F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley and Y. Bengio. *Theano: new features and speed improvements* («*Theano: новые возможности и повышение быстродействия*»). NIPS 2012 Deep Learning Workshop). Интересно, что данная библиотека названа в честь ученицы Пифагора Феано, которая, возможно, была его женой. Самыми сильными сторонами библиотеки являются быстрые откомпилированные на C вычисления, символические выражения и вычисления на GPU, при этом библиотека Theano находится в процессе активного развития, в результате которого регулярно вносятся улучшения в виде нового функционала. Реализации с использованием библиотеки Theano гораздо шире масштабируемого машинного обучения, и поэтому мы сузим круг и будем использовать Theano для глубокого обучения. Посетите веб-сайт Theano для получения дополнительной информации – http://deeplearning.net/software/theano/.

Когда возникает потребность в выполнении более сложных вычислений на многомерных матрицах, базовая библиотека NumPy будет обращаться к дорогостоящим циклам и итерациям, увеличивающим нагрузку на CPU, что мы и видели ранее. Библиотека Theano стремится эти вычисления оптимизировать, компилируя их в высокооптимизированный C-код и, если это возможно, задействуя GPU. Что касается нейронных сетей и глубокого обучения, то библиотека Theano имеет полезную способность автоматически дифференцировать математические функции, что очень удобно для вычисления частных производных при использовании таких алгоритмов, как обратное распространение ошибки.

В настоящее время библиотека Theano используется во всех видах проектов глубокого обучения и стала наиболее применяемой платформой в этой области.

В последнее время были созданы новые библиотеки поверх библиотеки Theano, чтобы упростить использование функционала глубокого обучения. Учитывая крутую кривую обучения работе с библиотекой Theano, мы будем использовать библиотеки, построенные поверх Theano, в частности theanets, pylearn2 и Lasagne.

Установка платформы Theano

Сначала удостоверьтесь, что вы устанавливаете дорабатываемую (development) версию со страницы Theano. Отметим, что если сделать `$ pip install theano`, то в итоге можно столкнуться с проблемами. Установка дорабатываемой версии непосредственно с GitHub является беспроигрышным вариантом:

```
$ git clone git://github.com/Theano/Theano.git
$ pip install Theano
```

Если требуется обновить библиотеку Theano, то можно воспользоваться следующей командой:

```
$ sudo pip install --upgrade theano
```

Если у вас есть вопросы и вы желаете связаться с сообществом Theano, то можно обратиться на https://groups.google.com/forum/#!forum/theano-users.

Вот и все, а теперь приступим к работе!

Чтобы удостовериться в том, что мы задали путь к папке с библиотекой Theano, необходимо сделать следующее:

```
#!/usr/bin/python
import os
import cPickle as pickle
from six.moves import cPickle as pickle

# здесь задать путь к папке theano
path = '/Users/Quandbee1/Desktop/pthw/Theano/'
```

Теперь давайте установим все необходимые библиотеки:

```
import numpy
import numpy as np
from theano import tensor
import theano.tensor as T
import theano.tensor.nnet as nnet
```

Для того чтобы библиотека Theano заработала на GPU (если есть карта NVIDIA и установлена платформа CUDA), сначала необходимо сконфигурировать платформу Theano.

В обычной ситуации библиотеки NumPy и Theano используют формат числа двойной точности с плавающей точкой (`float64`). Однако если нужно использовать GPU для Theano, то применяется 32-разрядный формат числа с плавающей точкой. Это означает, что необходимо изменить настройки между 32-разрядным и 64-разрядным форматами с плавающей точкой в зависимости от потребностей. Если вы желаете посмотреть, какая конфигурация используется вашей системой по умолчанию, то наберите следующее:

```
print(theano.config.floatX)
output: float64
```

Поменять конфигурацию на 32-разрядную для вычислений на GPU можно следующим образом:

```
theano.config.floatX = 'float32'
```

Иногда практичнее поменять настройки через терминал.

Для 32-разрядного числа с плавающей точкой наберите следующее:

```
$ export THEANO_FLAGS=floatX=float32
```

Для 64-разрядного числа с плавающей точкой следует набрать следующее:

```
$ export THEANO_FLAGS=floatX=float64
```

Если нужно закрепить заданную настройку за конкретным сценарием Python, то можно сделать так:

```
$ THEANO_FLAGS=floatX=float32 python you_name_here.py
```

Если нужно посмотреть, какой вычислительный метод используется в вашей системе Theano, то наберите следующее:

```
print(theano.config.device)
```

Если нужно изменить все настройки, разрядность с плавающей точкой и вычислительный метод (GPU или CPU) конкретного фрагмента сценария, то наберите следующее:

```
$ THEANO_FLAGS=device=gpu,floatX=float32 python your_script.py
```

Это может оказаться очень удобным для тестирования и программирования. Вполне возможно, что использовать GPU все время не потребуется; иногда лучше использовать CPU для прототипирования и эскизирования и выполнить свой сценарий на GPU, только когда он будет готов.

Сначала проверим, работает ли GPU в вашей конфигурации. Если карта NVIDIA GPU на компьютере отсутствует, то следующее ниже можно пропустить:

```
from theano import function, config, shared, sandbox
import theano.tensor as T
import numpy
import time

vlen = 10 * 30 * 768 # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], T.exp(x))
print(f.maker.fgraph.toposort())
t0 = time.time()
for i in xrange(iters):
    r = f()
t1 = time.time()
print("Выполнение цикла %d раз заняло %f секунд" % (iters, t1 - t0))
print("Result is %s" % (r,))
if numpy.any([isinstance(x.op, T.Elemwise) for x in f.maker.fgraph.
```

```
toposort()]):
    print('Использован cpu')
else:
    print('Использован gpu')
```

Теперь, когда мы знаем, как сконфигурировать платформу Theano, давайте пробежимся по нескольким простым примерам, чтобы увидеть, как она работает. В основном любая часть кода Theano имеет одинаковую структуру:

1) инициализация, где декларируются переменные в классе;
2) компиляция, где формируются функции;
3) исполнение, где применяются функции к типам данных.

Воспользуемся этими принципами в нескольких элементарных примерах векторных вычислений и математических выражений:

```
# инициализировать простой скаляр
x = T.dscalar()

fx = T.exp(T.tan(x**2)) # инициализировать требующуюся функцию

type(fx) # просто чтобы показать, что fx - это переменная theano
type

# скомпилировать, чтобы создать функцию tanh
f = theano.function(inputs=[x], outputs=[fx])

# исполнить функцию в данном случае на числе
f(10)
```

Как было упомянуто ранее, библиотека Theano может использоваться для математических выражений. Посмотрите на следующий ниже пример, где мы используем мощный функционал Theano под названием *автодифференцирование*, который становится очень полезным для алгоритма обратного распространения ошибки:

```
fp = T.grad(fx, wrt=x)
fs= theano.function([x], fp)

fs(3)

output: 4.59
```

Теперь, когда понятно, как используются переменные и функции, выполним простую логистическую функцию:

```
# теперь можно применить эту функцию к матрицам тоже
x = T.dmatrix('x')
s = 1 / (1 + T.exp(-x))
logistic = theano.function([x], s)
logistic([[2, 3], [.7, -2],[1.5,2.3]])

output:
array([[ 0.88079708, 0.95257413],
[ 0.66818777, 0.11920292],
[ 0.81757448, 0.90887704]])
```

Здесь четко видно, что Theano предоставляет более скоростные методы применения функций к объектам данных, чем это было бы возможным в библиотеке NumPy.

Предметный указатель

О

П

H

I

J

K

L

M

N

O

P

S

T

U

V

W

X

Y

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: books@alians-kniga.ru.

Бастиан Шарден, Лука Массарон, Альберто Боскетти

Крупномасштабное машинное обучение вместе с Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 33,5625. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

С распространением больших данных растет спрос на вычислительную и алгоритмическую эффективность. Главная задача настоящей книги состоит в том, чтобы предоставить способы применения мощных методов машинного обучения с открытым исходным кодом в крупномасштабных проектах без привлечения дорогостоящих корпоративных решений или больших вычислительных кластеров. Описаны масштабируемое обучение в Scikit-learn, нейронные сети и глубокое обучение с использованием Theano, H2O и TensorFlow. Рассмотрены классификационные и регрессионные деревья, а также обучение без учителя. Охвачены эффективные методы машинного обучения в вычислительной среде MapReduce на платформах Hadoop и Spark на языке Python.

С этой книгой вы научитесь:

- применять большинство масштабируемых алгоритмов машинного обучения;
- работать с новейшими крупномасштабными методами машинного обучения;
- увеличивать прогнозную точность при помощи методов глубокого обучения и масштабируемых методов обработки данных;
- работать с вычислительной парадигмой MapReduce в платформе Spark;
- применять эффективные алгоритмы машинного обучения на основе платформ Spark и Hadoop;
- создавать мощные ансамбли в крупном масштабе;
- использовать потоки данных для обучения линейных и нелинейных прогнозных моделей на чрезвычайно больших наборах данных, используя всего одну машину.

Кому адресована эта книга

Эта книга предназначена для любого читателя, кто намеревается работать с большими и сложными наборами данных. Рекомендуется знание основ программирования на Python и понятий машинного обучения. Также были бы полезны практические знания в области статистики и вычислительной математики. .

ISBN 978-5-97060-506-6

ДМК ПРЕСС
издательство
www.дмк.рф

Интернет-магазин: www.dmkpress.com
Книга-почтой: orders@alians-kniga.ru
Оптовая продажа: «Альянс-книга»
(499)782-3889, books@alians-kniga.ru