

Pascal Bugnion, Patrick R. Nicolas,
Alex Kozlov

Scala: Applied Machine Learning

Leverage the power of Scala and master the art of building, improving, and validating scalable machine learning and AI applications using Scala's most advanced and finest features.



Packt

Scala:Applied Machine Learning

Leverage the power of Scala and master the art of building, improving, and validating scalable machine learning and AI applications using Scala's most advanced and finest features

A course in three modules

Packt

BIRMINGHAM - MUMBAI

Scala:Applied Machine Learning

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: October 2016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78712-664-0

www.packtpub.com

Credits

Authors

Pascal Bugnion

Patrick R. Nicolas

Alex Kozlov

Content Development Editor

Sumeet Sawant

Graphics

Arvindkumar Gupta

Reviewers

Umanga Bista

Radek Ostrowski

Yuanhang Wang

Subhajit Datta

Rui Gonçalves

Patricia Hoffman, PhD

Md Zahidul Islam

Rok Kralj

Production Coordinator

Arvindkumar Gupta

Preface

Scala is considered to be a successor to Java in the area of Big Data by many. It is particularly good at analyzing large sets of data without any significant impact on performance and thus Scala is being adopted by many developers and data scientists

This Learning Path aims to put the entire world of machine learning with Scala in front of you. We will begin by introducing you to the libraries for ingesting, storing, manipulating, processing, and visualizing data in Scala. Moving on, we'll introduce machine learning in Scala and take a very deep dive into leveraging Scala to construct and study systems that can learn from data. Finally, we will master Scala machine learning in breadth and impart expertise for you to be able to build complex machine learning projects using Scala.

What this learning path covers

Module 1, Scala for Data Science, is a tutorial guide that provides tutorials on some of the most common Scala libraries for data science, allowing you to quickly get up to speed building data science and data engineering solutions.

Module 2, Scala for Machine Learning, guides you through the process of building AI applications with diagrams, formal mathematical notation, source code snippets, and useful tips. A review of the Akka framework and Apache Spark clusters concludes the tutorial.

Module 3, Mastering Scala Machine Learning, is the final step in this course. It will take your knowledge to next level and help you use the knowledge to build advanced applications such as social media mining, intelligent news portals, and more. After a quick refresher on functional programming concepts using REPL, you will see some practical examples of setting up the development environment and tinkering with data. We will then explore working with Spark and MLlib using k-means and decision trees.

What you need for this learning path

You'll need the following set up for all the three modules:

Module 1

The examples provided in this course require that you have a working Scala installation and SBT, the *Simple Build Tool*, a command line utility for compiling and running Scala code. We will walk you through how to install these in the next sections.

We do not require a specific IDE. The code examples can be written in your favorite text editor or IDE.

Installing the JDK

Scala code is compiled to Java byte code. To run the byte code, you must have the Java Virtual Machine (JVM) installed, which comes as part of a Java Development Kit (JDK). There are several JDK implementations and, for the purpose of this course, it does not matter which one you choose. You may already have a JDK installed on your computer. To check this, enter the following in a terminal:

```
$ java -version
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b17, mixed mode)
```

If you do not have a JDK installed, you will get an error stating that the `java` command does not exist.

If you do have a JDK installed, you should still verify that you are running a sufficiently recent version. The number that matters is the minor version number: the 8 in 1.8.0_66. Versions 1.8.xx of Java are commonly referred to as Java 8. For the first twelve chapters of this course, Java 7 will be sufficient (your version number should be something like 1.7.xx or newer). However, you will need Java 8 for the last two chapters, since the Play framework requires it. We therefore recommend that you install Java 8.

On Mac, the easiest way to install a JDK is using Homebrew:

```
$ brew install java
```

This will install Java 8, specifically the Java Standard Edition Development Kit, from Oracle.

Homebrew is a package manager for Mac OS X. If you are not familiar with Homebrew, I highly recommend using it to install development tools. You can find installation instructions for Homebrew on: <http://brew.sh>.

To install a JDK on Windows, go to <http://www.oracle.com/technetwork/java/javase/downloads/index.html> (or, if this URL does not exist, to the Oracle website, then click on Downloads and download **Java Platform, Standard Edition**). Select Windows x86 for 32-bit Windows, or Windows x64 for 64 bit. This will download an installer, which you can run to install the JDK.

To install a JDK on Ubuntu, install OpenJDK with the package manager for your distribution:

```
$ sudo apt-get install openjdk-8-jdk
```

If you are running a sufficiently old version of Ubuntu (14.04 or earlier), this package will not be available. In this case, either fall back to `openjdk-7-jdk`, which will let you run examples in the first twelve chapters, or install the Java Standard Edition Development Kit from Oracle through a PPA (a non-standard package archive):

```
$ sudo add-apt-repository ppa:webupd8team/java  
$ sudo apt-get update  
$ sudo apt-get install oracle-java8-installer
```

You then need to tell Ubuntu to prefer Java 8 with:

```
$ sudo update-java-alternatives -s java-8-oracle
```

Installing and using SBT

The Simple Build Tool (SBT) is a command line tool for managing dependencies and building and running Scala code. It is the de facto build tool for Scala. To install SBT, follow the instructions on the SBT website (<http://www.scala-sbt.org/0.13/tutorial/Setup.html>).

When you start a new SBT project, SBT downloads a specific version of Scala for you. You, therefore, do not need to install Scala directly on your computer. Managing the entire dependency suite from SBT, including Scala itself, is powerful: you do not have to worry about developers working on the same project having different versions of Scala or of the libraries used.

Since we will use SBT extensively in this course, let's create a simple test project. If you have used SBT previously, do skip this section.

Create a new directory called `sbt-example` and navigate to it. Inside this directory, create a file called `build.sbt`. This file encodes all the dependencies for the project. Write the following in `build.sbt`:

```
// build.sbt  
  
scalaVersion := "2.11.7"
```

This specifies which version of Scala we want to use for the project. Open a terminal in the `sbt-example` directory and type:

```
$ sbt
```

This starts an interactive shell. Let's open a Scala console:

```
> console
```

This gives you access to a Scala console in the context of your project:

```
scala> println("Scala is running!")  
Scala is running!
```

Besides running code in the console, we will also write Scala programs. Open an editor in the `sbt-example` directory and enter a basic "hello, world" program.

Name the file `HelloWorld.scala`:

```
// HelloWorld.scala  
  
object HelloWorld extends App {  
    println("Hello, world!")  
}
```

Return to SBT and type:

```
> run
```

This will compile the source files and run the executable, printing "Hello, world!".

Besides compiling and running your Scala code, SBT also manages Scala dependencies. Let's specify a dependency on Breeze, a library for numerical algorithms. Modify the `build.sbt` file as follows:

```
// build.sbt

scalaVersion := "2.11.7"

libraryDependencies ++= Seq(
  "org.scalanlp" %% "breeze" % "0.11.2",
  "org.scalanlp" %% "breeze-natives" % "0.11.2"
)
```

SBT requires that statements be separated by empty lines, so make sure that you leave an empty line between `scalaVersion` and `libraryDependencies`. In this example, we have specified a dependency on Breeze version "0.11.2". How did we know to use these coordinates for Breeze? Most Scala packages will quote the exact SBT string to get the latest version in their documentation.

If this is not the case, or you are specifying a dependency on a Java library, head to the Maven Central website (<http://mvnrepository.com>) and search for the package of interest, for example "Breeze". The website provides a list of packages, including several named `breeze_2.xx` packages. The number after the underscore indicates the version of Scala the package was compiled for. Click on "`breeze_2.11`" to get a list of the different Breeze versions available. Choose "0.11.2". You will be presented with a list of package managers to choose from (Maven, Ivy, Leiningen, and so on). Choose SBT. This will print a line like:

```
libraryDependencies += "org.scalanlp" % "breeze_2.11" % "0.11.2"
```

These are the coordinates that you will want to copy to the `build.sbt` file. Note that we just specified "`breeze`", rather than "`breeze_2.11`". By preceding the package name with two percentage signs, `%%`, SBT automatically resolves to the correct Scala version. Thus, specifying `%% "breeze"` is identical to `% "breeze_2.11"`.

Now return to your SBT console and run:

```
> reload
```

This will fetch the Breeze jars from Maven Central. You can now import Breeze in either the console or your scripts (within the context of this Scala project). Let's test this in the console:

```
> console
scala> import breeze.linalg._
import breeze.linalg._

scala> import breeze.numerics._
import breeze.numerics._

scala> val vec = linspace(-2.0, 2.0, 100)
vec: breeze.linalg.DenseVector[Double] = DenseVector(-2.0,
-1.9595959595959596, ...

scala> sigmoid(vec)
breeze.linalg.DenseVector[Double] = DenseVector(0.11920292202211755,
0.12351078065 ...)
```

You should now be able to compile, run and specify dependencies for your Scala scripts.

Module 2

A decent command of the Scala programming language is a prerequisite. Reading through a mathematical formulation, conveniently defined in an information box, is optional. However, some basic knowledge of mathematics and statistics might be helpful to understand the inner workings of some algorithms.

The course uses the following libraries:

- Scala 2.10.3 or higher
- Java JDK 1.7.0_45 or 1.8.0_25
- SBT 0.13 or higher
- JFreeChart 1.0.1
- Apache Commons Math library 3.5 (*Chapter 3, Data Preprocessing*, *Chapter 4, Unsupervised Learning*, and *Chapter 6, Regression and Regularization*)
- Indian Institute of Technology Bombay CRF 0.2 (*Chapter 7, Sequential Data Models*)

- LIBSVM 0.1.6 (*Chapter 8, Kernel Models and Support Vector Machines*)
- Akka 2.2.4 or higher (or Typesafe activator 1.2.10 or higher) (*Chapter 12, Scalable Frameworks*)
- Apache Spark 1.3.0 or higher (*Chapter 12, Scalable Frameworks*)

Module 3

This course is based on open source software. First, it's Java. One can download Java from Oracle's Java Download page. You have to accept the license and choose an appropriate image for your platform. Don't use OpenJDK—it has a few problems with Hadoop/Spark.

Second, Scala. If you are using Mac, I recommend installing Homebrew:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Multiple open source packages will also be available to you. To install Scala, run `brew install scala`. Installation on a Linux platform requires downloading an appropriate Debian or RPM package from the <http://www.scala-lang.org/download/> site. We will use the latest version at the time, that is, 2.11.7.

Spark distributions can be downloaded from <http://spark.apache.org/downloads.html>. We use pre-build for Hadoop 2.6 and later image. As it's Java, you need to just unzip the package and start using the scripts from the `bin` subdirectory.

Who this learning path is for

This Learning Path is for engineers and scientists who are familiar with Scala and want to learn how to create, validate, and apply machine learning algorithms. It will also benefit software developers with a background in Scala programming who want to apply machine learning.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a course, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Scala-Applied-Machine-Learning-Code>.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Course Module 1: Scala for Data Science

Chapter 1: Scala and Data Science	3
Data science	3
Programming in data science	6
Why Scala?	7
When not to use Scala	14
Summary	14
References	15
Chapter 2: Manipulating Data with Breeze	17
Code examples	17
Installing Breeze	18
Getting help on Breeze	18
Basic Breeze data types	19
An example – logistic regression	37
Towards re-usable code	45
Alternatives to Breeze	47
Summary	47
References	47
Chapter 3: Plotting with breeze-viz	49
Diving into Breeze	50
Customizing plots	52
Customizing the line type	55
More advanced scatter plots	60
Multi-plot example – scatterplot matrix plots	62

Table of Contents

Managing without documentation	67
Breeze-viz reference	68
Data visualization beyond breeze-viz	69
Summary	69
Chapter 4: Parallel Collections and Futures	71
Parallel collections	71
Futures	85
Summary	95
References	95
Chapter 5: Scala and SQL through JDBC	97
Interacting with JDBC	98
First steps with JDBC	98
JDBC summary	106
Functional wrappers for JDBC	107
Safer JDBC connections with the loan pattern	108
Enriching JDBC statements with the "pimp my library" pattern	110
Wrapping result sets in a stream	113
Looser coupling with type classes	115
Creating a data access layer	121
Summary	122
References	122
Chapter 6: Slick – A Functional Interface for SQL	125
FEC data	125
Invokers	137
Operations on columns	138
Aggregations with "Group by"	140
Accessing database metadata	142
Slick versus JDBC	143
Summary	143
References	143
Chapter 7: Web APIs	145
A whirlwind tour of JSON	146
Querying web APIs	147
JSON in Scala – an exercise in pattern matching	148
Extraction using case classes	154
Concurrency and exception handling with futures	158
Authentication – adding HTTP headers	160
Summary	164
References	165

Table of Contents

Chapter 8: Scala and MongoDB	167
MongoDB	168
Connecting to MongoDB with Casbah	169
Inserting documents	172
Extracting objects from the database	178
Complex queries	182
Casbah query DSL	184
Custom type serialization	185
Beyond Casbah	187
Summary	187
References	188
Chapter 9: Concurrency with Akka	189
GitHub follower graph	189
Actors as people	191
Hello world with Akka	193
Case classes as messages	195
Actor construction	196
Anatomy of an actor	197
Follower network crawler	198
Fetcher actors	200
Routing	204
Message passing between actors	205
Queue control and the pull pattern	211
Accessing the sender of a message	213
Stateful actors	214
Follower network crawler	215
Fault tolerance	218
Custom supervisor strategies	220
Life-cycle hooks	222
What we have not talked about	226
Summary	227
References	227
Chapter 10: Distributed Batch Processing with Spark	229
Installing Spark	229
Acquiring the example data	230
Resilient distributed datasets	231
Building and running standalone programs	246
Spam filtering	250
Lifting the hood	258
Data shuffling and partitions	261

Table of Contents

Summary	263
Reference	263
Chapter 11: Spark SQL and DataFrames	265
DataFrames – a whirlwind introduction	265
Aggregation operations	270
Joining DataFrames together	272
Custom functions on DataFrames	274
DataFrame immutability and persistence	276
SQL statements on DataFrames	277
Complex data types – arrays, maps, and structs	279
Interacting with data sources	282
Standalone programs	284
Summary	285
References	285
Chapter 12: Distributed Machine Learning with MLlib	287
Introducing MLlib – Spam classification	288
Pipeline components	291
Evaluation	302
Regularization in logistic regression	308
Cross-validation and model selection	310
Beyond logistic regression	315
Summary	315
References	315
Chapter 13: Web APIs with Play	317
Client-server applications	318
Introduction to web frameworks	318
Model-View-Controller architecture	319
Single page applications	321
Building an application	323
The Play framework	324
Dynamic routing	329
Actions	330
Interacting with JSON	335
Querying external APIs and consuming JSON	337
Creating APIs with Play: a summary	344
Rest APIs: best practice	344
Summary	345
References	345

Table of Contents

Chapter 14: Visualization with D3 and the Play Framework	347
GitHub user data	348
Do I need a backend?	348
JavaScript dependencies through web-jars	349
Towards a web application: HTML templates	350
Modular JavaScript through RequireJS	353
Bootstrapping the applications	355
Client-side program architecture	357
Drawing plots with NVD3	366
Summary	369
References	370
Appendix: Pattern Matching and Extractors	371
Pattern matching in for comprehensions	374
Pattern matching internals	374
Extracting sequences	376
Summary	377
Reference	378

Course Module 2: Scala for Machine Learning

Chapter 1: Getting Started	381
Mathematical notation for the curious	382
Why machine learning?	382
Why Scala?	383
Model categorization	389
Taxonomy of machine learning algorithms	390
Don't reinvent the wheel!	395
Tools and frameworks	395
Source code	398
Let's kick the tires	402
Summary	423
Chapter 2: Hello World!	425
Modeling	425
Defining a methodology	428
Monadic data transformation	429
A workflow computational model	435
Profiling data	446
Assessing a model	448
Summary	462

Table of Contents

Chapter 3: Data Preprocessing	463
Time series in Scala	463
Moving averages	469
Fourier analysis	477
The discrete Kalman filter	491
Alternative preprocessing techniques	506
Summary	506
Chapter 4: Unsupervised Learning	507
Clustering	508
Dimension reduction	537
Performance considerations	546
Summary	548
Chapter 5: Naïve Bayes Classifiers	549
Probabilistic graphical models	549
Naïve Bayes classifiers	551
The Multivariate Bernoulli classification	571
Naïve Bayes and text mining	573
Pros and cons	585
Summary	586
Chapter 6: Regression and Regularization	587
Linear regression	587
Regularization	605
Numerical optimization	614
Logistic regression	616
Summary	630
Chapter 7: Sequential Data Models	631
Markov decision processes	631
The hidden Markov model	633
Conditional random fields	659
Regularized CRFs and text analytics	663
Comparing CRF and HMM	676
Performance consideration	677
Summary	678
Chapter 8: Kernel Models and Support Vector Machines	679
Kernel functions	680
Support vector machines	687
Support vector classifiers – SVC	693

Table of Contents

Anomaly detection with one-class SVC	715
Support vector regression	717
Performance considerations	722
Summary	722
Chapter 9: Artificial Neural Networks	723
Feed-forward neural networks	723
The multilayer perceptron	727
Evaluation	758
Convolution neural networks	770
Benefits and limitations	773
Summary	775
Chapter 10: Genetic Algorithms	777
Evolution	777
Genetic algorithms and machine learning	780
Genetic algorithm components	780
Implementation	790
GA for trading strategies	807
Advantages and risks of genetic algorithms	820
Summary	821
Chapter 11: Reinforcement Learning	823
Reinforcement learning	823
Learning classifier systems	857
Summary	869
Chapter 12: Scalable Frameworks	871
An overview	872
Scala	873
Scalability with Actors	882
Akka	884
Apache Spark	901
Summary	919
Appendix: Basic Concepts	921
Scala programming	921
Mathematics	934
Finances 101	944
Suggested online courses	951
References	951

Course Module 3: Mastering Scala Machine Learning

Chapter 1: Exploratory Data Analysis	955
Getting started with Scala	956
Distinct values of a categorical field	958
Summarization of a numeric field	961
Basic, stratified, and consistent sampling	962
Working with Scala and Spark Notebooks	965
Basic correlations	971
Summary	974
Chapter 2: Data Pipelines and Modeling	975
Influence diagrams	976
Sequential trials and dealing with risk	979
Exploration and exploitation	985
Unknown unknowns	987
Basic components of a data-driven system	988
Optimization and interactivity	998
Summary	999
Chapter 3: Working with Spark and MLlib	1001
Setting up Spark	1002
Understanding Spark architecture	1003
Applications	1012
ML libraries	1022
Spark performance tuning	1026
Running Hadoop HDFS	1028
Summary	1034
Chapter 4: Supervised and Unsupervised Learning	1035
Records and supervised learning	1036
Unsupervised learning	1051
Problem dimensionality	1058
Summary	1061
Chapter 5: Regression and Classification	1063
What regression stands for?	1063
Continuous space and metrics	1064
Linear regression	1069
Logistic regression	1075
Regularization	1077

Multivariate regression	1078
Heteroscedasticity	1078
Regression trees	1080
Classification metrics	1082
Multiclass problems	1083
Perceptron	1084
Generalization error and overfitting	1087
Summary	1087
Chapter 6: Working with Unstructured Data	1089
Nested data	1090
Other serialization formats	1101
Hive and Impala	1105
Sessionization	1108
Working with traits	1114
Working with pattern matching	1115
Other uses of unstructured data	1118
Probabilistic structures	1119
Projections	1119
Summary	1120
Chapter 7: Working with Graph Algorithms	1121
A quick introduction to graphs	1122
SBT	1123
Graph for Scala	1126
GraphX	1135
Summary	1152
Chapter 8: Integrating Scala with R and Python	1153
Integrating with R	1154
Integrating with Python	1180
Summary	1189
Chapter 9: NLP in Scala	1191
Text analysis pipeline	1193
MLlib algorithms in Spark	1202
Segmentation, annotation, and chunking	1212
POS tagging	1213
Using word2vec to find word relationships	1217
Summary	1221

Table of Contents

Chapter 10: Advanced Model Monitoring	1223
System monitoring	1225
Process monitoring	1227
Model monitoring	1235
Summary	1236
Bibliography	1237

Module 1

Scala for Data Science

Leverage the power of Scala with different tools to build scalable,
robust data science applications

1

Scala and Data Science

The second half of the 20th century was the age of silicon. In fifty years, computing power went from extremely scarce to entirely mundane. The first half of the 21st century is the age of the Internet. The last 20 years have seen the rise of giants such as Google, Twitter, and Facebook—giants that have forever changed the way we view knowledge.

The Internet is a vast nexus of information. Ninety percent of the data generated by humanity has been generated in the last 18 months. The programmers, statisticians, and scientists who can harness this glut of data to derive real understanding will have an ever greater influence on how businesses, governments, and charities make decisions.

This book strives to introduce some of the tools that you will need to synthesize the avalanche of data to produce true insight.

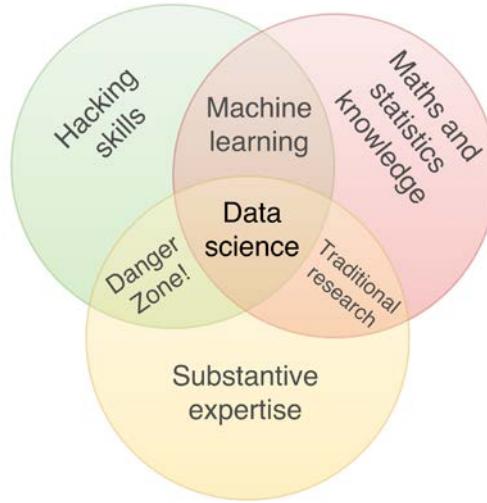
Data science

Data science is the process of extracting useful information from data. As a discipline, it remains somewhat ill-defined, with nearly as many definitions as there are experts. Rather than add yet another definition, I will follow *Drew Conway's* description (<http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram>). He describes data science as the culmination of three orthogonal sets of skills:

- Data scientists must have *hacking skills*. Data is stored and transmitted through computers. Computers, programming languages, and libraries are the hammers and chisels of data scientists; they must wield them with confidence and accuracy to sculpt the data as they please. This is where Scala comes in: it's a powerful tool to have in your programming toolkit.

- Data scientists must have a sound understanding of *statistics and numerical algorithms*. Good data scientists will understand how machine learning algorithms function and how to interpret results. They will not be fooled by misleading metrics, deceptive statistics, or misinterpreted causal links.
- A good data scientist must have a sound understanding of the *problem domain*. The data science process involves building and discovering knowledge about the problem domain in a scientifically rigorous manner. The data scientist must, therefore, ask the right questions, be aware of previous results, and understand how the data science effort fits in the wider business or research context.

Drew Conway summarizes this elegantly with a Venn diagram showing data science at the intersection of hacking skills, maths and statistics knowledge, and substantive expertise:



It is, of course, rare for people to be experts in more than one of these areas. Data scientists often work in cross-functional teams, with different members providing the expertise for different areas. To function effectively, every member of the team must nevertheless have a general working knowledge of all three areas.

To give a more concrete overview of the workflow in a data science project, let's imagine that we are trying to write an application that analyzes the public perception of a political campaign. This is what the data science pipeline might look like:

- **Obtaining data:** This might involve extracting information from text files, polling a sensor network or querying a web API. We could, for instance, query the Twitter API to obtain lists of tweets with the relevant hashtags.
- **Data ingestion:** Data often comes from many different sources and might be unstructured or semi-structured. Data ingestion involves moving data from the data source, processing it to extract structured information, and storing this information in a database. For tweets, for instance, we might extract the username, the names of other users mentioned in the tweet, the hashtags, text of the tweet, and whether the tweet contains certain keywords.
- **Exploring data:** We often have a clear idea of what information we want to extract from the data but very little idea how. For instance, let's imagine that we have ingested thousands of tweets containing hashtags relevant to our political campaign. There is no clear path to go from our database of tweets to the end goal: insight into the overall public perception of our campaign. Data exploration involves mapping out how we are going to get there. This step will often uncover new questions or sources of data, which requires going back to the first step of the pipeline. For our tweet database, we might, for instance, decide that we need to have a human manually label a thousand or more tweets as expressing "positive" or "negative" sentiments toward the political campaign. We could then use these tweets as a training set to construct a model.
- **Feature building:** A machine learning algorithm is only as good as the features that enter it. A significant fraction of a data scientist's time involves transforming and combining existing features to create new features more closely related to the problem that we are trying to solve. For instance, we might construct a new feature corresponding to the number of "positive" sounding words or pairs of words in a tweet.
- **Model construction and training:** Having built the features that enter the model, the data scientist can now train machine learning algorithms on their datasets. This will often involve trying different algorithms and optimizing model **hyperparameters**. We might, for instance, settle on using a random forest algorithm to decide whether a tweet is "positive" or "negative" about the campaign. Constructing the model involves choosing the right number of trees and how to calculate impurity measures. A sound understanding of statistics and the problem domain will help inform these decisions.

- **Model extrapolation and prediction:** The data scientists can now use their new model to try and infer information about previously unseen data points. They might pass a new tweet through their model to ascertain whether it speaks positively or negatively of the political campaign.
- **Distillation of intelligence and insight from the model:** The data scientists combine the outcome of the data analysis process with knowledge of the business domain to inform business decisions. They might discover that specific messages resonate better with the target audience, or with specific segments of the target audience, leading to more accurate targeting. A key part of informing stakeholders involves data visualization and presentation: data scientists create graphs, visualizations, and reports to help make the insights derived clear and compelling.

This is far from a linear pipeline. Often, insights gained at one stage will require the data scientists to backtrack to a previous stage of the pipeline. Indeed, the generation of business insights from raw data is normally an iterative process: the data scientists might do a rapid first pass to verify the premise of the problem and then gradually refine the approach by adding new data sources or new features or trying new machine learning algorithms.

In this book, you will learn how to deal with each step of the pipeline in Scala, leveraging existing libraries to build robust applications.

Programming in data science

This book is not a book about data science. It is a book about how to use Scala, a programming language, for data science. So, where does programming come in when processing data?

Computers are involved at every step of the data science pipeline, but not necessarily in the same manner. The style of programs that we build will be drastically different if we are just writing throwaway scripts to explore data or trying to build a scalable application that pushes data through a well-understood pipeline to continuously deliver business intelligence.

Let's imagine that we work for a company making games for mobile phones in which you can purchase in-game benefits. The majority of users never buy anything, but a small fraction is likely to spend a lot of money. We want to build a model that recognizes big spenders based on their play patterns.

The first step is to explore data, find the right features, and build a model based on a subset of the data. In this exploration phase, we have a clear goal in mind but little idea of how to get there. We want a light, flexible language with strong libraries to get us a working model as soon as possible.

Once we have a working model, we need to deploy it on our gaming platform to analyze the usage patterns of all the current users. This is a very different problem: we have a relatively clear understanding of the goals of the program and of how to get there. The challenge comes in designing software that will scale out to handle all the users and be robust to future changes in usage patterns.

In practice, the type of software that we write typically lies on a spectrum ranging from a single throwaway script to production-level code that must be proof against future expansion and load increases. Before writing any code, the data scientist must understand where their software lies on this spectrum. Let's call this the **permanence spectrum**.

Why Scala?

You want to write a program that handles data. Which language should you choose?

There are a few different options. You might choose a dynamic language such as Python or R or a more traditional object-oriented language such as Java. In this section, we will explore how Scala differs from these languages and when it might make sense to use it.

When choosing a language, the architect's trade-off lies in a balance of provable correctness versus development speed. Which of these aspects you need to emphasize will depend on the application requirements and where on the permanence spectrum your program lies. Is this a short script that will be used by a few people who can easily fix any problems that arise? If so, you can probably permit a certain number of bugs in rarely used code paths: when a developer hits a snag, they can just fix the problem as it arises. By contrast, if you are developing a database engine that you plan on releasing to the wider world, you will, in all likelihood, favor correctness over rapid development. The SQLite database engine, for instance, is famous for its extensive test suite, with 800 times as much testing code as application code (<https://www.sqlite.org/testing.html>).

What matters, when estimating the *correctness* of a program, is not the perceived absence of bugs, it is the degree to which you can prove that certain bugs are absent.

There are several ways of proving the absence of bugs before the code has even run:

- Static type checking occurs at compile time in statically typed languages, but this can also be used in strongly typed dynamic languages that support type annotations or type hints. Type checking helps verify that we are using functions and classes as intended.
- Static analyzers and linters that check for undefined variables or suspicious behavior (such as parts of the code that can never be reached).
- Declaring some attributes as immutable or constant in compiled languages.
- Unit testing to demonstrate the absence of bugs along particular code paths.

There are several more ways of checking for the absence of some bugs at runtime:

- Dynamic type checking in both statically typed and dynamic languages
- Assertions verifying supposed program invariants or expected contracts

In the next sections, we will examine how Scala compares to other languages in data science.

Static typing and type inference

Scala's static typing system is very versatile. A lot of information as to the program's behavior can be encoded in types, allowing the compiler to guarantee a certain level of correctness. This is particularly useful for code paths that are rarely used. A dynamic language cannot catch errors until a particular branch of execution runs, so a bug can persist for a long time until the program runs into it. In a statically typed language, any bug that can be caught by the compiler will be caught at compile time, before the program has even started running.

Statically typed object-oriented languages have often been criticized for being needlessly verbose. Consider the initialization of an instance of the `Example` class in Java:

```
Example myInstance = new Example();
```

We have to repeat the class name twice – once to define the compile-time type of the `myInstance` variable and once to construct the instance itself. This feels like unnecessary work: the compiler knows that the type of `myInstance` is `Example` (or a superclass of `Example`) as we are binding a value of the `Example` type.

Scala, like most functional languages, uses type inference to allow the compiler to infer the type of variables from the instances bound to them. We would write the equivalent line in Scala as follows:

```
val myInstance = new Example()
```

The Scala compiler infers that `myInstance` has the `Example` type at compile time. A lot of the time, it is enough to specify the types of the arguments and of the return value of a function. The compiler can then infer types for all the variables defined in the body of the function. Scala code is usually much more concise and readable than the equivalent Java code, without compromising any of the type safety.

Scala encourages immutability

Scala encourages the use of immutable objects. In Scala, it is very easy to define an attribute as immutable:

```
val amountSpent = 200
```

The default collections are immutable:

```
val clientIds = List("123", "456") // List is immutable
clientIds(1) = "589" // Compile-time error
```

Having immutable objects removes a common source of bugs. Knowing that some objects cannot be changed once instantiated reduces the number of places bugs can creep in. Instead of considering the lifetime of the object, we can narrow in on the constructor.

Scala and functional programs

Scala encourages functional code. A lot of Scala code consists of using higher-order functions to transform collections. You, as a programmer, do not have to deal with the details of iterating over the collection. Let's write an `occurrencesOf` function that returns the indices at which an element occurs in a list:

```
def occurrencesOf[A] (elem:A, collection>List[A]):List[Int] = {
    for {
        (currentElem, index) <- collection.zipWithIndex
        if (currentElem == elem)
    } yield index
}
```

How does this work? We first declare a new list, `collection.zipWithIndex`, whose elements are `(collection(0), 0)`, `(collection(1), 1)`, and so on: pairs of the collection's elements and their indexes.

We then tell Scala that we want to iterate over this collection, binding the `currentElem` variable to the current element and `index` to the index. We apply a filter on the iteration, selecting only those elements for which `currentElem == elem`. We then tell Scala to just return the `index` variable.

We did not need to deal with the details of the iteration process in Scala. The syntax is very declarative: we tell the compiler that we want the index of every element equal to `elem` in `collection` and let the compiler worry about how to iterate over `collection`.

Consider the equivalent in Java:

```
static <T> List<Integer> occurrencesOf(T elem, List<T> collection) {  
    List<Integer> occurrences = new ArrayList<Integer>() ;  
    for (int i=0; i<collection.size(); i++) {  
        if (collection.get(i).equals(elem)) {  
            occurrences.add(i) ;  
        }  
    }  
    return occurrences ;  
}
```

In Java, you start by defining a (mutable) list in which to put occurrences as you find them. You then iterate over the collection by defining a counter, considering each element in turn and adding its index to the list of occurrences, if need be. There are many more moving parts that we need to get right for this method to work. These moving parts exist because we must tell Java how to iterate over the collection, and they represent a common source of bugs.

Furthermore, as a lot of code is taken up by the iteration mechanism, the line that defines the logic of the function is harder to find:

```
static <T> List<Integer> occurrencesOf(T elem, List<T> collection) {  
    List<Integer> occurrences = new ArrayList<Integer>() ;  
    for (int i=0; i<collection.size(); i++) {  
        if (collection.get(i).equals(elem)) {  
            occurrences.add(i) ;  
        }  
    }  
    return occurrences ;  
}
```

Note that this is not meant as an attack on Java. In fact, Java 8 adds a slew of functional constructs, such as lambda expressions, the `Optional` type that mirrors Scala's `Option`, or stream processing. Rather, it is meant to demonstrate the benefit of functional approaches in minimizing the potential for errors and maximizing clarity.

Null pointer uncertainty

We often need to represent the possible absence of a value. For instance, imagine that we are reading a list of usernames from a CSV file. The CSV file contains name and e-mail information. However, some users have declined to enter their e-mail into the system, so this information is absent. In Java, one would typically represent the e-mail as a string or an `Email` class and represent the absence of e-mail information for a particular user by setting that reference to `null`. Similarly, in Python, we might use `None` to demonstrate the absence of a value.

This approach is dangerous because we are not encoding the possible absence of e-mail information. In any nontrivial program, deciding whether an instance attribute can be `null` requires considering every occasion in which this instance is defined. This quickly becomes impractical, so programmers either assume that a variable is not `null` or code too defensively.

Scala (following the lead of other functional languages) introduces the `Option[T]` type to represent an attribute that might be absent. We might then write the following:

```
class User {  
    ...  
    val email: Option[Email]  
    ...  
}
```

We have now encoded the possible absence of e-mail in the type information. It is obvious to any programmer using the `User` class that e-mail information is possibly absent. Even better, the compiler knows that the `email` field can be absent, forcing us to deal with the problem rather than recklessly ignoring it to have the application burn at runtime in a conflagration of null pointer exceptions.

All this goes back to achieving a certain level of provable correctness. Never using `null`, we know that we will never run into null pointer exceptions. Achieving the same level of correctness in languages without `Option[T]` requires writing unit tests on the client code to verify that it behaves correctly when the e-mail attribute is `null`.

Note that it is possible to achieve this in Java using, for instance, Google's Guava library (<https://code.google.com/p/guava-libraries/wiki/UsingAndAvoidingNullExplained>) or the `Optional` class in Java 8. It is more a matter of convention: using `null` in Java to denote the absence of a value has long been the norm.

Easier parallelism

Writing programs that take advantage of parallel architectures is challenging. It is nevertheless necessary to tackle all but the simplest data science problems.

Parallel programming is difficult because we, as programmers, tend to think sequentially. Reasoning about the order in which different events can happen in a concurrent program is very challenging.

Scala provides several abstractions that greatly facilitate the writing of parallel code. These abstractions work by imposing constraints on the way parallelism is achieved. For instance, parallel collections force the user to phrase the computation as a sequence of operations (such as **map**, **reduce**, and **filter**) on collections. Actor systems require the developer to think in terms of actors that encapsulate the application state and communicate by passing messages.

It might seem paradoxical that restricting the programmer's freedom to write parallel code as they please avoids many of the problems associated with concurrency. However, limiting the number of ways in which a program behaves facilitates thinking about its behavior. For instance, if an actor is misbehaving, we know that the problem lies either in the code for this actor or in one of the messages that the actor receives.

As an example of the power afforded by having coherent, restrictive abstractions, let's use parallel collections to solve a simple probability problem. We will calculate the probability of getting at least 60 heads out of 100 coin tosses. We can estimate this using Monte Carlo: we simulate 100 coin tosses by drawing 100 random Boolean values and check whether the number of true values is at least 60. We repeat this until results have converged to the required accuracy, or we get bored of waiting.

Let's run through this in a Scala console:

```
scala> val nTosses = 100
nTosses: Int = 100

scala> def trial = (0 until nTosses).count { i =>
    util.Random.nextBoolean() // count the number of heads
}
trial: Int
```

The `trial` function runs a single set of 100 throws, returning the number of heads:

```
scala> trial  
Int = 51
```

To get our answer, we just need to repeat `trial` as many times as we can and aggregate the results. Repeating the same set of operations is ideally suited to parallel collections:

```
scala> val nTrials = 100000  
nTrials: Int = 100000  
  
scala> (0 until nTrials).par.count { i => trial >= 60 }  
Int = 2745
```

The probability is thus approximately 2.5% to 3%. All we had to do to distribute the calculation over every CPU in our computer is use the `par` method to parallelize the range `(0 until nTrials)`. This demonstrates the benefits of having a coherent abstraction: parallel collections let us trivially parallelize any computation that can be phrased in terms of higher-order functions on collections.

Clearly, not every problem is as easy to parallelize as a simple Monte Carlo problem. However, by offering a rich set of intuitive abstractions, Scala makes writing parallel applications manageable.

Interoperability with Java

Scala runs on the Java virtual machine. The Scala compiler compiles programs to Java byte code. Thus, Scala developers have access to Java libraries natively. Given the phenomenal number of applications written in Java, both open source and as part of the legacy code in organizations, the interoperability of Scala and Java helps explain the rapid uptake of Scala.

Interoperability has not just been unidirectional: some Scala libraries, such as the Play framework, are becoming increasingly popular among Java developers.

When not to use Scala

In the previous sections, we described how Scala's strong type system, preference for immutability, functional capabilities, and parallelism abstractions make it easy to write reliable programs and minimize the risk of unexpected behavior.

What reasons might you have to avoid Scala in your next project? One important reason is familiarity. Scala introduces many concepts such as implicits, type classes, and composition using traits that might not be familiar to programmers coming from the object-oriented world. Scala's type system is very expressive, but getting to know it well enough to use its full power takes time and requires adjusting to a new programming paradigm. Finally, dealing with immutable data structures can feel alien to programmers coming from Java or Python.

Nevertheless, these are all drawbacks that can be overcome with time. Scala does fall short of the other data science languages in library availability. The IPython Notebook, coupled with matplotlib, is an unparalleled resource for data exploration. There are ongoing efforts to provide similar functionality in Scala (Spark Notebooks or Apache Zeppelin, for instance), but there are no projects with the same level of maturity. The type system can also be a minor hindrance when one is exploring data or trying out different models.

Thus, in this author's biased opinion, Scala excels for more *permanent* programs. If you are writing a throwaway script or exploring data, you might be better served with Python. If you are writing something that will need to be reused and requires a certain level of provable correctness, you will find Scala extremely powerful.

Summary

Now that the obligatory introduction is over, it is time to write some Scala code. In the next chapter, you will learn about leveraging Breeze for numerical computations with Scala. For our first foray into data science, we will use logistic regression to predict the gender of a person given their height and weight.

References

By far, the best book on Scala is *Programming in Scala* by Martin Odersky, Lex Spoon, and Bill Venners. Besides being authoritative (Martin Odersky is the driving force behind Scala), this book is also approachable and readable.

Scala Puzzlers by Andrew Phillips and Nermin Šerifović provides a fun way to learn more advanced Scala.

Scala for Machine Learning by Patrick R. Nicholas provides examples of how to write machine learning algorithms with Scala.

2

Manipulating Data with Breeze

Data science is, by and large, concerned with the manipulation of structured data. A large fraction of structured datasets can be viewed as tabular data: each row represents a particular instance, and columns represent different attributes of that instance. The ubiquity of tabular representations explains the success of spreadsheet programs like Microsoft Excel, or of tools like SQL databases.

To be useful to data scientists, a language must support the manipulation of columns or tables of data. Python does this through NumPy and pandas, for instance. Unfortunately, there is no single, coherent ecosystem for numerical computing in Scala that quite measures up to the SciPy ecosystem in Python.

In this chapter, we will introduce Breeze, a library for fast linear algebra and manipulation of data arrays as well as many other features necessary for scientific computing and data science.

Code examples

The easiest way to access the code examples in this book is to clone the GitHub repository:

```
$ git clone 'https://github.com/pbugnion/s4ds'
```

The code samples for each chapter are in a single, standalone folder. You may also browse the code online on GitHub.

Installing Breeze

If you have downloaded the code examples for this book, the easiest way of using Breeze is to go into the `chap02` directory and type `sbt console` at the command line. This will open a Scala console in which you can import Breeze.

If you want to build a standalone project, the most common way of installing Breeze (and, indeed, any Scala module) is through SBT. To fetch the dependencies required for this chapter, copy the following lines to a file called `build.sbt`, taking care to leave an empty line after `scalaVersion`:

```
scalaVersion := "2.11.7"

libraryDependencies ++= Seq(
  "org.scalanlp" %% "breeze" % "0.11.2",
  "org.scalanlp" %% "breeze-natives" % "0.11.2"
)
```

Open a Scala console in the same directory as your `build.sbt` file by typing `sbt console` in a terminal. You can check that Breeze is working correctly by importing Breeze from the Scala prompt:

```
scala> import breeze.linalg._
import breeze.linalg._
```

Getting help on Breeze

This chapter gives a reasonably detailed introduction to Breeze, but it does not aim to give a complete API reference.

To get a full list of Breeze's functionality, consult the Breeze Wiki page on GitHub at <https://github.com/scalanlp/breeze/wiki>. This is very complete for some modules and less complete for others. The source code (<https://github.com/scalanlp/breeze/>) is detailed and gives a lot of information. To understand how a particular function is meant to be used, look at the unit tests for that function.

Basic Breeze data types

Breeze is an extensive library providing fast and easy manipulation of arrays of data, routines for optimization, interpolation, linear algebra, signal processing, and numerical integration.

The basic linear algebra operations underlying Breeze rely on the netlib-java library, which can use system-optimized **BLAS** and **LAPACK** libraries, if present. Thus, linear algebra operations in Breeze are often extremely fast. Breeze is still undergoing rapid development and can, therefore, be somewhat unstable.

Vectors

Breeze makes manipulating one- and two-dimensional data structures easy. To start, open a Scala console through SBT and import Breeze:

```
$ sbt console
scala> import breeze.linalg._
```

Let's dive straight in and define a vector:

```
scala> val v = DenseVector(1.0, 2.0, 3.0)
breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0, 3.0)
```

We have just defined a three-element vector, v. Vectors are just one-dimensional arrays of data exposing methods tailored to numerical uses. They can be indexed like other Scala collections:

```
scala> v(1)
Double = 2.0
```

They support element-wise operations with a scalar:

```
scala> v :* 2.0 // :* is 'element-wise multiplication'
breeze.linalg.DenseVector[Double] = DenseVector(2.0, 4.0, 6.0)
```

They also support element-wise operations with another vector:

```
scala> v :+ DenseVector(4.0, 5.0, 6.0) // :+ is 'element-wise addition'
breeze.linalg.DenseVector[Double] = DenseVector(5.0, 7.0, 9.0)
```

Breeze makes writing vector operations intuitive and considerably more readable than the native Scala equivalent.

Note that Breeze will refuse (at compile time) to coerce operands to the correct type:

```
scala> v :* 2 // element-wise multiplication by integer
<console>:15: error: could not find implicit value for parameter op:
...
```

It will also refuse (at runtime) to add vectors together if they have different lengths:

```
scala> v :+ DenseVector(8.0, 9.0)
java.lang.IllegalArgumentException: requirement failed: Vectors must have
same length: 3 != 2
...
```

Basic manipulation of vectors in Breeze will feel natural to anyone used to working with NumPy, MATLAB, or R.

So far, we have only looked at *element-wise* operators. These are all prefixed with a colon. All the usual suspects are present: `:+`, `:*`, `:-`, `:/`, `:%` (remainder), and `:^` (power) as well as Boolean operators. To see the full list of operators, have a look at the API documentation for `DenseVector` or `DenseMatrix` (<https://github.com/scalanlp/breeze/wiki/Linear-Algebra-Cheat-Sheet>).

Besides element-wise operations, Breeze vectors support the operations you might expect of mathematical vectors, such as the dot product:

```
scala> val v2 = DenseVector(4.0, 5.0, 6.0)
breeze.linalg.DenseVector[Double] = DenseVector(4.0, 5.0, 6.0)

scala> v dot v2
Double = 32.0
```

Pitfalls of element-wise operators

Besides the `:+` and `:-` operators for element-wise addition and subtraction that we have seen so far, we can also use the more traditional `+` and `-` operators:

```
scala> v + v2
breeze.linalg.DenseVector[Double] = DenseVector(5.0,
7.0, 9.0)
```

One must, however, be very careful with operator precedence rules when mixing `:+` or `:+` with `:+` operators. The `:+` and `:+` operators have very low operator precedence, so they will be evaluated last. This can lead to some counter-intuitive behavior:



```
scala> 2.0 :* v + v2 // !! equivalent to 2.0 :* (v + v2)
breeze.linalg.DenseVector[Double] = DenseVector(10.0,
14.0, 18.0)
```

By contrast, if we use `:+` instead of `+`, the mathematical precedence of operators is respected:

```
scala> 2.0 :* v :+ v2 // equivalent to (2.0 :* v) :+ v2
breeze.linalg.DenseVector[Double] = DenseVector(6.0,
9.0, 12.0)
```

In summary, one should avoid mixing the `:+` style operators with the `+` style operators as much as possible.

Dense and sparse vectors and the vector trait

All the vectors we have looked at thus far have been dense vectors. Breeze also supports sparse vectors. When dealing with arrays of numbers that are mostly zero, it may be more computationally efficient to use sparse vectors. The point at which a vector has enough zeros to warrant switching to a sparse representation depends strongly on the type of operations, so you should run your own benchmarks to determine which type to use. Nevertheless, a good heuristic is that, if your vector is about 90% zero, you may benefit from using a sparse representation.

Sparse vectors are available in Breeze as the `SparseVector` and `HashVector` classes. Both these types support many of the same operations as `DenseVector` but use a different internal implementation. The `SparseVector` instances are very memory-efficient, but adding non-zero elements is slow. `HashVector` is more versatile, at the cost of an increase in memory footprint and computational time for iterating over non-zero elements. Unless you need to squeeze the last bits of memory out of your application, I recommend using `HashVector`. We will not discuss these further in this book, but the reader should find them straightforward to use if needed.

`DenseVector`, `SparseVector`, and `HashVector` all implement the `Vector` trait, giving them a common interface.



Breeze remains very experimental and, as of this writing, somewhat unstable. I have found dealing with specific implementations of the `Vector` trait, such as `DenseVector` or `SparseVector`, to be more reliable than dealing with the `Vector` trait directly. In this chapter, we will explicitly type every vector as `DenseVector`.

Matrices

Breeze allows the construction and manipulation of two-dimensional arrays in a similar manner:

```
scala> val m = DenseMatrix((1.0, 2.0, 3.0), (4.0, 5.0, 6.0))
breeze.linalg.DenseMatrix[Double] =
1.0 2.0 3.0
4.0 5.0 6.0

scala> 2.0 :* m
breeze.linalg.DenseMatrix[Double] =
2.0 4.0 6.0
8.0 10.0 12.0
```

Building vectors and matrices

We have seen how to explicitly build vectors and matrices by passing their values to the constructor (or rather, to the companion object's `apply` method): `DenseVector(1.0, 2.0, 3.0)`. Breeze offers several other powerful ways of building vectors and matrices:

```
scala> DenseVector.ones[Double](5)
breeze.linalg.DenseVector[Double] = DenseVector(1.0, 1.0, 1.0, 1.0, 1.0)

scala> DenseVector.zeros[Int](3)
breeze.linalg.DenseVector[Int] = DenseVector(0, 0, 0)
```

The `linspace` method (available in the `breeze.linalg` package object) creates a Double vector of equally spaced values. For instance, to create a vector of 10 values distributed uniformly between 0 and 1, perform the following:

```
scala> linspace(0.0, 1.0, 10)
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.1111111111111111,
..., 1.0)
```

The `tabulate` method lets us construct vectors and matrices from functions:

```
scala> DenseVector.tabulate(4) { i => 5.0 * i }
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 5.0, 10.0, 15.0)

scala> DenseMatrix.tabulate[Int](2, 3) {
  (irow, icol) => irow*2 + icol
}
breeze.linalg.DenseMatrix[Int] =
0 1 2
2 3 4
```

The first argument to `DenseVector.tabulate` is the size of the vector, and the second is a function returning the value of the vector at a particular position. This is useful for creating ranges of data, among other things.

The `rand` function lets us create random vectors and matrices:

```
scala> DenseVector.rand(2)
breeze.linalg.DenseVector[Double] = DenseVector(0.8072865137359484,
0.5566507203838562)

scala> DenseMatrix.rand(2, 3)
breeze.linalg.DenseMatrix[Double] =
0.5755491874682879  0.8142161471517582  0.9043780212739738
0.31530195124023974 0.2095094278911871  0.22069103504148346
```

Finally, we can construct vectors from Scala arrays:

```
scala> DenseVector(Array(2, 3, 4))
breeze.linalg.DenseVector[Int] = DenseVector(2, 3, 4)
```

To construct vectors from other Scala collections, you must use the `splat` operator, `:_ *`:

```
scala> val l = Seq(2, 3, 4)
l: Seq[Int] = List(2, 3, 4)

scala> DenseVector(l :_ *)
breeze.linalg.DenseVector[Int] = DenseVector(2, 3, 4)
```

Advanced indexing and slicing

We have already seen how to select a particular element in a vector `v` by its index with, for instance, `v(2)`. Breeze also offers several powerful methods for selecting parts of a vector.

Let's start by creating a vector to play around with:

```
scala> val v = DenseVector.tabulate(5) { _.toDouble }
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 1.0, 2.0, 3.0, 4.0)
```

Unlike native Scala collections, Breeze vectors support negative indexing:

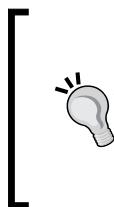
```
scala> v(-1) // last element
Double = 4.0
```

Breeze lets us slice the vector using a range:

```
scala> v(1 to 3)
breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0, 3.0)

scala> v(1 until 3) // equivalent to Python v[1:3]
breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0)

scala> v(v.length-1 to 0 by -1) // reverse view of v
breeze.linalg.DenseVector[Double] = DenseVector(4.0, 3.0, 2.0, 1.0, 0.0)
```



Indexing by a range returns a *view* of the original vector: when running `val v2 = v(1 to 3)`, no data is copied. This means that slicing is extremely efficient. Taking a slice of a huge vector does not increase the memory footprint at all. It also means that one should be careful updating a slice, since it will also update the original vector. We will discuss mutating vectors and matrices in a subsequent section in this chapter.

Breeze also lets us select an arbitrary set of elements from a vector:

```
scala> val vSlice = v(2, 4) // Select elements at index 2 and 4
breeze.linalg.SliceVector[Int,Double] = breeze.linalg.SliceVector@9c04d22
```

This creates a `SliceVector`, which behaves like a `DenseVector` (both implement the `Vector` interface), but does not actually have memory allocated for values: it just knows how to map from its indices to values in its parent vector. One should think of `vSlice` as a specific view of `v`. We can materialize the view (give it its own data rather than acting as a lens through which `v` is viewed) by converting it to `DenseVector`:

```
scala> vSlice.toDenseVector
breeze.linalg.DenseVector[Double] = DenseVector(2.0, 4.0)
```

Note that if an element of a slice is out of bounds, an exception will only be thrown when that element is accessed:

```
scala> val vSlice = v(2, 7) // there is no v(7)
breeze.linalg.SliceVector[Int[Double]] = breeze.linalg.
SliceVector@2a83f9d1

scala> vSlice(0) // valid since v(2) is still valid
Double = 2.0

scala> vSlice(1) // invalid since v(7) is out of bounds
java.lang.IndexOutOfBoundsException: 7 not in [-5,5)
...
```

Finally, one can index vectors using Boolean arrays. Let's start by defining an array:

```
scala> val mask = DenseVector(true, false, false, true, true)
breeze.linalg.DenseVector[Boolean] = DenseVector(true, false, false,
true, true)
```

Then, `v(mask)` results in a view containing the elements of `v` for which `mask` is `true`:

```
scala> v(mask).toDenseVector
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 3.0, 4.0)
```

This can be used as a way of filtering certain elements in a vector. For instance, to select the elements of `v` which are less than 3.0:

```
scala> val filtered = v(v < 3.0) // < is element-wise "less than"
breeze.linalg.SliceVector[Int[Double]] = breeze.linalg.
SliceVector@2b1edef3

scala> filtered.toDenseVector
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 1.0, 2.0)
```

Matrices can be indexed in much the same way as vectors. Matrix indexing functions take two arguments – the first argument selects the row(s) and the second one slices the column(s):

```
scala> val m = DenseMatrix((1.0, 2.0, 3.0), (5.0, 6.0, 7.0))
m: breeze.linalg.DenseMatrix[Double] =
1.0  2.0  3.0
5.0  6.0  7.0

scala> m(1, 2)
Double = 7.0

scala> m(1, -1)
Double = 7.0

scala> m(0 until 2, 0 until 2)
breeze.linalg.DenseMatrix[Double] =
1.0  2.0
5.0  6.0
```

You can also mix different slicing types for rows and columns:

```
scala> m(0 until 2, 0)
breeze.linalg.DenseVector[Double] = DenseVector(1.0, 5.0)
```

Note how, in this case, Breeze returns a vector. In general, slicing returns the following objects:

- A scalar when single indices are passed as the row and column arguments
- A vector when the row argument is a range and the column argument is a single index
- A vector transpose when the column argument is a range and the row argument is a single index
- A matrix otherwise

The symbol `::` can be used to indicate *every element along a particular direction*. For instance, we can select the second column of `m`:

```
scala> m(::, 1)
breeze.linalg.DenseVector[Double] = DenseVector(2.0, 6.0)
```

Mutating vectors and matrices

Breeze vectors and matrices are mutable. Most of the slicing operations described above can also be used to set elements of a vector or matrix:

```
scala> val v = DenseVector(1.0, 2.0, 3.0)
v: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0, 3.0)

scala> v(1) = 22.0 // v is now DenseVector(1.0, 22.0, 3.0)
```

We are not limited to mutating single elements. In fact, all the indexing operations outlined above can be used to set the elements of vectors or matrices. When mutating slices of vectors or matrices, use the element-wise assignment operator, `:=`:

```
scala> v(0 until 2) := DenseVector(50.0, 51.0) // set elements at
position 0 and 1
breeze.linalg.DenseVector[Double] = DenseVector(50.0, 51.0)

scala> v
breeze.linalg.DenseVector[Double] = DenseVector(50.0, 51.0, 3.0)
```

The assignment operator, `:=`, works like other element-wise operators in Breeze. If the right-hand side is a scalar, it will automatically be broadcast to a vector of the given shape:

```
scala> v(0 until 2) := 0.0 // equivalent to v(0 until 2) :=
DenseVector(0.0, 0.0)
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.0)

scala> v
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.0, 3.0)
```

All element-wise operators have an update counterpart. For instance, the `:+=` operator acts like the element-wise addition operator `:+`, but also updates its left-hand operand:

```
scala> val v = DenseVector(1.0, 2.0, 3.0)
v: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0, 3.0)

scala> v :+= 4.0
breeze.linalg.DenseVector[Double] = DenseVector(5.0, 6.0, 7.0)

scala> v
breeze.linalg.DenseVector[Double] = DenseVector(5.0, 6.0, 7.0)
```

Notice how the update operator updates the vector in place and returns it.

We have learnt how to slice vectors and matrices in Breeze to create new views of the original data. These views are not independent of the vector they were created from—updating the view will update the underlying vector and vice-versa. This is best illustrated with an example:

```
scala> val v = DenseVector.tabulate(6) { _.toDouble }
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 1.0, 2.0, 3.0, 4.0,
5.0)

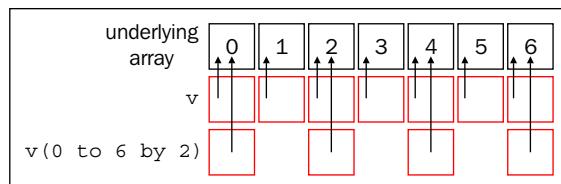
scala> val viewEvens = v(0 until v.length by 2)
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 2.0, 4.0)

scala> viewEvens := 10.0 // mutate viewEvens
breeze.linalg.DenseVector[Double] = DenseVector(10.0, 10.0, 10.0)

scala> viewEvens
breeze.linalg.DenseVector[Double] = DenseVector(10.0, 10.0, 10.0)

scala> v // v has also been mutated!
breeze.linalg.DenseVector[Double] = DenseVector(10.0, 1.0, 10.0, 3.0,
10.0, 5.0)
```

This quickly becomes intuitive if we remember that, when we create a vector or matrix, we are creating a view of an underlying data array rather than creating the data itself:



A vector slice `v(0 to 6 by 2)` of the `v` vector is just a different view of the array underlying `v`.

The view itself contains no data. It just contains pointers to the data in the original array. Internally, the view is just stored as a pointer to the underlying data and a recipe for iterating over that data: in the case of this slice, the recipe is just "start at the first element of the underlying data and go to the seventh element of the underlying data in steps of two".

Breeze offers a `copy` function for when we want to create independent copies of data. In the previous example, we can construct a copy of `viewEvens` as:

```
scala> val copyEvens = v(0 until v.length by 2).copy
breeze.linalg.DenseVector[Double] = DenseVector(10.0, 10.0, 10.0)
```

We can now update `copyEvens` independently of `v`.

Matrix multiplication, transposition, and the orientation of vectors

So far, we have mostly looked at element-wise operations on vectors and matrices. Let's now look at matrix multiplication and related operations.

The matrix multiplication operator is `*`:

```
scala> val m1 = DenseMatrix((2.0, 3.0), (5.0, 6.0), (8.0, 9.0))
breeze.linalg.DenseMatrix[Double] =
2.0 3.0
5.0 6.0
8.0 9.0

scala> val m2 = DenseMatrix((10.0, 11.0), (12.0, 13.0))
breeze.linalg.DenseMatrix[Double]
10.0 11.0
12.0 13.0

scala> m1 * m2
56.0 61.0
122.0 133.0
188.0 205.0
```

Besides matrix-matrix multiplication, we can use the matrix multiplication operator between matrices and vectors. All vectors in Breeze are column vectors. This means that, when multiplying matrices and vectors together, a vector should be viewed as an $(n * 1)$ matrix. Let's walk through an example of matrix-vector multiplication. We want the following operation:

$$\begin{pmatrix} 2 & 3 \\ 5 & 6 \\ 8 & 9 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

```
scala> val v = DenseVector(1.0, 2.0)
breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0)

scala> m1 * v
breeze.linalg.DenseVector[Double] = DenseVector(8.0, 17.0, 26.0)
```

By contrast, if we wanted:

$$(1 \ 2) \begin{pmatrix} 10 & 11 \\ 12 & 13 \end{pmatrix}$$

We must convert `v` to a row vector. We can do this using the transpose operation:

```
scala> val vt = v.t
breeze.linalg.Transpose[breeze.linalg.DenseVector[Double]] =
Transpose(DenseVector(1.0, 2.0))

scala> vt * m2
breeze.linalg.Transpose[breeze.linalg.DenseVector[Double]] =
Transpose(DenseVector(34.0, 37.0))
```

Note that the type of `v.t` is `Transpose[DenseVector[_]]`. A `Transpose[DenseVector[_]]` behaves in much the same way as a `DenseVector` as far as element-wise operations are concerned, but it does not support mutation or slicing.

Data preprocessing and feature engineering

We have now discovered the basic components of Breeze. In the next few sections, we will apply them to real examples to understand how they fit together to form a robust base for data science.

An important part of data science involves preprocessing datasets to construct useful features. Let's walk through an example of this. To follow this example and access the data, you will need to download the code examples for the book (www.github.com/pbugnion/s4ds).

You will find, in directory `chap02/data/` of the code attached to this book, a CSV file with true heights and weights as well as self-reported heights and weights for 181 men and women. The original dataset was collected as part of a study on body image. Refer to the following link for more information: <http://vincentarelbundock.github.io/Rdatasets/doc/car/Davis.html>.

There is a helper function in the package provided with the book to load the data into Breeze arrays:

```
scala> val data = HWData.load
HWData [ 181 rows ]

scala> data.genders
breeze.linalg.Vector[Char] = DenseVector(M, F, F, M, ... )
```

The data object contains five vectors, each 181 element long:

- `data.genders`: A Char vector describing the gender of the participants
- `data.heights`: A Double vector of the true height of the participants
- `data.weights`: A Double vector of the true weight of the participants
- `data.reportedHeights`: A Double vector of the self-reported height of the participants
- `data.reportedWeights`: A Double vector of the self-reported weight of the participants

Let's start by counting the number of men and women in the study. We will define an array that contains just 'M' and do an element-wise comparison with `data.genders`:

```
scala> val maleVector = DenseVector.fill(data.genders.length) ('M')
breeze.linalg.DenseVector[Char] = DenseVector(M, M, M, M, M, M, ... )

scala> val isMale = (data.genders :== maleVector)
breeze.linalg.DenseVector[Boolean] = DenseVector(true, false, false, true
...)
```

The `isMale` vector is the same length as `data.genders`. It is `true` where the participant is male, and `false` otherwise. We can use this Boolean array as a mask for the other arrays in the dataset (remember that `vector(mask)` selects the elements of `vector` where `mask` is `true`). Let's get the height of the men in our dataset:

```
scala> val maleHeights = data.heights(isMale)
breeze.linalg.SliceVector[Int[Double] = breeze.linalg.
SliceVector@61717d42

scala> maleHeights.toDenseVector
breeze.linalg.DenseVector[Double] = DenseVector(182.0, 177.0, 170.0, ...
```

To count the number of men in our dataset, we can use the `indicator` function. This transforms a Boolean array into an array of doubles, mapping `false` to `0.0` and `true` to `1.0`:

```
scala> import breeze.numerics._
import breeze.numerics._
```

```
scala> sum(I(isMale))
Double: 82.0
```

Let's calculate the mean height of men and women in the experiment. We can calculate the mean of a vector using `mean(v)`, which we can access by importing `breeze.stats._`:

```
scala> import breeze.stats._
import breeze.stats._
```

```
scala> mean(data.heights)
Double = 170.75690607734808
```

To calculate the mean height of the men, we can use our `isMale` array to slice `data.heights`; `data.heights(isMale)` is a view of the `data.heights` array with all the height values for the men:

```
scala> mean(data.heights(isMale)) // mean male height
Double = 178.0121951219512
```

```
scala> mean(data.heights(!isMale)) // mean female height
Double = 164.74747474747474
```

As a somewhat more involved example, let's look at the discrepancy between real and reported weight for both men and women in this experiment. We can get an array of the percentage difference between the reported weight and the true weight:

```
scala> val discrepancy =
  (data.weights - data.reportedWeights) / data.weights
breeze.linalg.Vector[Double] = DenseVector(0.0, 0.1206896551724138,
-0.018867924528301886, -0.029411764705882353, ... )
```

Notice how Breeze's overloading of mathematical operators allows us to manipulate data arrays easily and elegantly.

We can now calculate the mean and standard deviation of this array for men:

```
scala> mean(discrepancy(isMale))
res6: Double = -0.008451852933123775

scala> stddev(discrepancy(isMale))
res8: Double = 0.031901519634244195
```

We can also calculate the fraction of men who overestimated their height:

```
scala> val overReportMask =
  (data.reportedHeights :> data.heights).toDenseVector
breeze.linalg.DenseVector[Boolean] = DenseVector(false, false, false,
false...

scala> sum(I(overReportMask :& isMale))
Double: 10.0
```

There are thus ten men who believe they are taller than they actually are. The element-wise AND operator `:&` returns a vector that is true for all indices for which both its arguments are true. The vector `overReportMask :& isMale` is thus true for all participants that are male and over-reported their height.

Breeze – function optimization

Having studied feature engineering, let's now look at the other end of the data science pipeline. Typically, a machine learning algorithm defines a loss function that is a function of a set of parameters. The value of the loss function represents how well the model fits the data. The parameters are then optimized to minimize (or maximize) the loss function.

In *Chapter 12, Distributed Machine Learning with MLlib*, we will look at **MLlib**, a machine learning library that contains many well-known algorithms. Often, we don't need to worry about optimizing loss functions directly since we can rely on the machine learning algorithms provided by MLlib. It is nevertheless useful to have a basic knowledge of optimization.

Breeze has an `optimize` module that contains functions for finding a local minimum:

```
scala> import breeze.optimize._  
import breeze.optimize._
```

Let's create a toy function that we want to optimize:

$$f(x) = \sum_i x_i^2$$

We can represent this function in Scala as follows:

```
scala> def f(xs:DenseVector[Double]) = sum(xs :^ 2.0)  
f: (xs: breeze.linalg.DenseVector[Double])Double
```

Most local optimizers also require the gradient of the function being optimized. The gradient is a vector of the same dimension as the arguments to the function. In our case, the gradient is:

$$\nabla f = 2\bar{x}$$

We can represent the gradient in Breeze with a function that takes a vector argument and returns a vector of the same length:

```
scala> def gradf(xs:DenseVector[Double]) = 2.0 :* xs  
gradf: (xs:breeze.linalg.DenseVector[Double])breeze.linalg.  
DenseVector[Double]
```

For instance, at the point $(1, 1, 1)$, we have:

```
scala> val xs = DenseVector.ones[Double](3)  
breeze.linalg.DenseVector[Double] = DenseVector(1.0, 1.0, 1.0)  
  
scala> f(xs)  
Double = 3.0  
  
scala> gradf(xs)  
breeze.linalg.DenseVector[Double] = DenseVector(2.0, 2.0, 2.0)
```

Let's set up the optimization problem. Breeze's optimization methods require that we pass in an implementation of the `DiffFunction` trait with a single method, `calculate`. This method must return a tuple of the function and its gradient:

```
scala> val optTrait = new DiffFunction[DenseVector[Double]] {
    def calculate(xs:DenseVector[Double]) = (f(xs), gradf(xs))
}
breeze.optimize.DiffFunction[breeze.linalg.DenseVector[Double]] =
<function1>
```

We are now ready to run the optimization. The `optimize` module provides a `minimize` function that does just what we want. We pass it `optTrait` and a starting point for the optimization:

```
scala> val minimum = minimize(optTrait, DenseVector(1.0, 1.0, 1.0))
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.0, 0.0)
```

The true minimum is at $(0.0, 0.0, 0.0)$. The optimizer therefore correctly finds the minimum.

The `minimize` function uses the **L-BFGS** method to run the optimization by default. It takes several additional arguments to control the optimization. We will explore these in the next sections.

Numerical derivatives

In the previous example, we specified the gradient of `f` explicitly. While this is generally good practice, calculating the gradient of a function can often be tedious. Breeze provides a gradient approximation function using finite differences. Reusing the same objective function `def f(xs:DenseVector[Double]) = sum(xs :^ 2.0)` as in the previous section:

```
scala> val approxOptTrait = new ApproximateGradientFunction(f)
breeze.optimize.ApproximateGradientFunction[Int,breeze.linalg.
DenseVector[Double]] = <function1>
```

The trait `approxOptTrait` has a `gradientAt` method that returns an approximation to the gradient at a point:

```
scala> approxOptTrait.gradientAt(DenseVector.ones(3))
breeze.linalg.DenseVector[Double] = DenseVector(2.00001000001393,
2.00001000001393, 2.00001000001393)
```

Note that this can be quite inaccurate. The `ApproximateGradientFunction` constructor takes an `epsilon` optional argument that controls the size of the step taken when calculating the finite differences. Changing the value of `epsilon` can improve the accuracy of the finite difference algorithm.

The `ApproximateGradientFunction` instance implements the `DiffFunction` trait. It can therefore be passed to `minimize` directly:

```
scala> minimize(approxOptTrait, DenseVector.ones[Double] (3))
breeze.linalg.DenseVector[Double] = DenseVector(-5.000001063126813E-6,
-5.000001063126813E-6, -5.000001063126813E-6)
```

This, again, gives a result close to zero, but somewhat further away than when we specified the gradient explicitly. In general, it will be significantly more efficient and more accurate to calculate the gradient of a function analytically than to rely on Breeze's numerical gradient. It is probably best to only use the numerical gradient during data exploration or to check analytical gradients.

Regularization

The `minimize` function takes many optional arguments relevant to machine learning algorithms. In particular, we can instruct the optimizer to use a regularization parameter when performing the optimization. Regularization introduces a penalty in the loss function to prevent the parameters from growing arbitrarily. This is useful to avoid overfitting. We will discuss regularization in greater detail in *Chapter 12, Distributed Machine Learning with MLlib*.

For instance, to use L2Regularization with a hyperparameter of 0.5:

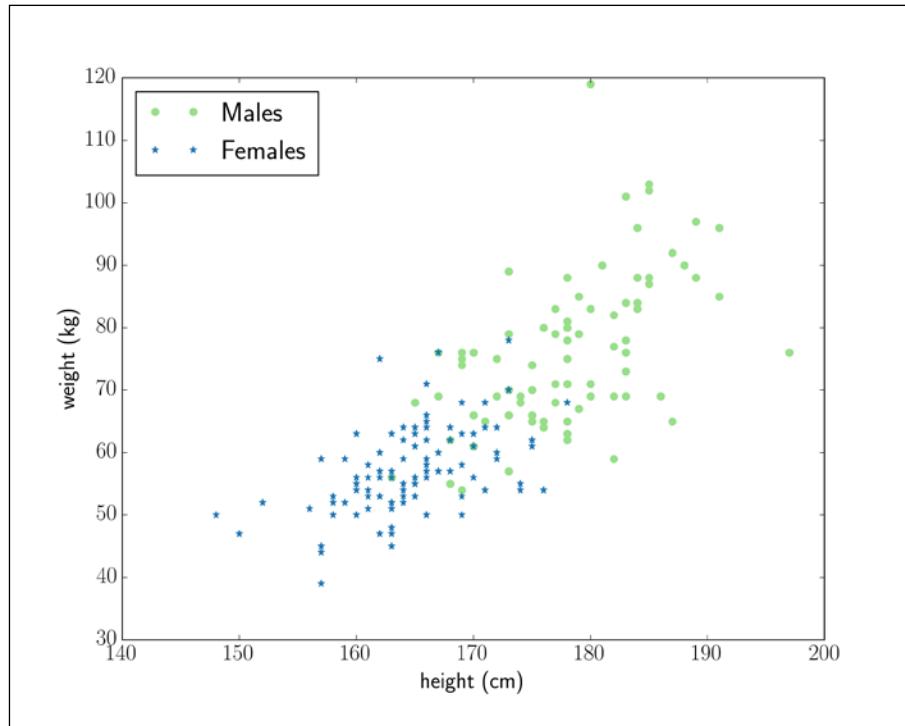
```
scala> minimize(optTrait,
  DenseVector(1.0, 1.0, 1.0), L2Regularization(0.5))
breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.0, 0.0)
```

The regularization makes no difference in this case, since the parameters are zero at the minimum.

To see a list of optional arguments that can be passed to `minimize`, consult the Breeze documentation online.

An example – logistic regression

Let's now imagine we want to build a classifier that takes a person's **height** and **weight** and assigns a probability to their being **Male** or **Female**. We will reuse the height and weight data introduced earlier in this chapter. Let's start by plotting the dataset:



Height versus weight data for 181 men and women

There are many different algorithms for classification. A first glance at the data shows that we can, approximately, separate men from women by drawing a straight line across the plot. A linear method is therefore a reasonable initial attempt at classification. In this section, we will use logistic regression to build a classifier.

A detailed explanation of logistic regression is beyond the scope of this book. The reader unfamiliar with logistic regression is referred to *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman. We will just give a brief summary here.

Logistic regression estimates the probability of a given *height* and *weight* belonging to a *male* with the following sigmoid function:

$$P(\text{male} | \text{height}, \text{weight}) = \frac{1}{1 + \exp(-f(\text{height}, \text{weight}; \text{params}))}$$

Here, f is a linear function:

$$f(\text{height}, \text{weight}; \text{params}) = \text{params}(0) + \text{height} \cdot \text{params}(1) + \text{weight} \cdot \text{params}(2)$$

Here, params is an array of parameters that we need to determine using the training set. If we consider the height and weight as a $\text{features} = (\text{height}, \text{weight})$ matrix, we can re-write the sigmoid kernel f as a matrix multiplication of the features matrix with the params vector:

$$f(\text{features}; \text{params}) = \text{params}(0) + \text{features} \cdot \text{params}(1:)$$

To simplify this expression further, it is common to add a dummy feature whose value is always 1 to the features matrix. We can then multiply $\text{params}(0)$ by this feature, allowing us to write the entire sigmoid kernel f as a single matrix-vector multiplication:

$$f(\text{features}; \text{params}) = \text{params} \cdot \text{features}$$

The feature matrix, features , is now a $(181 * 3)$ matrix, where each row is $(1, \text{height}, \text{weight})$ for a particular participant.

To find the optimal values of the parameters, we can maximize the likelihood function, $L(\text{params} | \text{features})$. The likelihood takes a given set of parameter values as input and returns the probability that these particular parameters gave rise to the training set. For a set of parameters and associated probability function $P(\text{male} | \text{features})$, the likelihood is:

$$L(\text{params} | \text{features}) = \prod_i P(\text{male} | \text{features}_i) \times \\ \begin{cases} & \text{target}_i \text{ is male} \\ & \prod_i 1 - P(\text{male} | \text{features}_i) \\ & \text{target}_i \text{ not male} \end{cases}$$

If we magically know, ahead of time, the gender of everyone in the population, we can assign $P(\text{male})=1$ for the men and $P(\text{male})=0$ for the women. The likelihood function would then be 1. Conversely, any uncertainty leads to a reduction in the likelihood function. If we choose a set of parameters that consistently lead to classification errors (low $P(\text{male})$ for men or high $P(\text{male})$ for women), the likelihood function drops to 0.

The maximum likelihood corresponds to those values of the parameters most likely to describe the observed data. Thus, to find the parameters that best describe our training set, we just need to find parameters that maximize $L(\text{params} | \text{features})$. However, maximizing the likelihood function itself is very rarely done, since it involves multiplying many small values together, which quickly leads to floating point underflow. It is best to maximize the log of the likelihood, which has the same maximum as the likelihood. Finally, since most optimization algorithms are geared to minimize a function rather than maximize it, we will minimize $-\log(L(\text{params} | \text{features}))$.

For logistic regression, this is equivalent to minimizing:

$$\text{Cost}(\text{params}) = \sum_i \text{target}_i \times (\text{params} \cdot \text{features}_i) - \log(\exp(\text{params} \cdot \text{features}_i) + 1)$$

Here, the sum runs over all participants in the training data, features_i is a vector $(1, \text{height}_i, \text{weight}_i)$ of the i -th observation in the training set, and target_i is 1 if the person is male, and 0 if the participant is female.

To minimize the Cost function, we must also know its gradient with respect to the parameters. This is:

$$\nabla_{\text{params}} \text{Cost} = \sum_i \text{features}_i \cdot [P(\text{male} | \text{features}_i) - \text{target}_i]$$

We will start by rescaling the height and weight by their mean and standard deviation. While this is not strictly necessary for logistic regression, it is generally good practice. It facilitates the optimization and would become necessary if we wanted to use regularization methods or build superlinear features (features that allow the boundary separating men from women to be curved rather than a straight line).

For this example, we will move away from the Scala shell and write a standalone Scala script. Here's the full code listing. Don't worry if this looks daunting. We will break it up into manageable chunks in a minute:

```
import breeze.linalg._  
import breeze.numerics._  
import breeze.optimize._  
import breeze.stats._  
  
object LogisticRegressionHWData extends App {  
  
    val data = HWData.load  
  
    // Rescale the features to have mean of 0.0 and s.d. of 1.0  
    def rescaled(v:DenseVector[Double]) =  
        (v - mean(v)) / stddev(v)  
  
    val rescaledHeights = rescaled(data.heights)  
    val rescaledWeights = rescaled(data.weights)  
  
    // Build the feature matrix as a matrix with  
    // 181 rows and 3 columns.  
    val rescaledHeightsAsMatrix = rescaledHeights.toDenseMatrix.t  
    val rescaledWeightsAsMatrix = rescaledWeights.toDenseMatrix.t  
  
    val featureMatrix = DenseMatrix.horzcat(  
        DenseMatrix.ones[Double](rescaledHeightsAsMatrix.rows, 1),  
        rescaledHeightsAsMatrix,  
        rescaledWeightsAsMatrix  
    )  
  
    println(s"Feature matrix size: ${featureMatrix.rows} x " +  
        s"${featureMatrix.cols}")  
  
    // Build the target variable to be 1.0 where a participant  
    // is male, and 0.0 where the participant is female.  
    val target = data.genders.values.map {
```

```

    gender => if(gender == 'M') 1.0 else 0.0
}

// Build the loss function ready for optimization.
// We will worry about refactoring this to be more
// efficient later.
def costFunction(parameters:DenseVector[Double]):Double = {
    val xBeta = featureMatrix * parameters
    val expXBeta = exp(xBeta)
    - sum((target :* xBeta) - log1p(expXBeta))
}

def costFunctionGradient(parameters:DenseVector[Double]):DenseVector[Double] = {
    val xBeta = featureMatrix * parameters
    val probs = sigmoid(xBeta)
    featureMatrix.t * (probs - target)
}

val f = new DiffFunction[DenseVector[Double]] {
    def calculate(parameters:DenseVector[Double]) =
        (costFunction(parameters), costFunctionGradient(parameters))
}

val optimalParameters = minimize(f, DenseVector(0.0, 0.0, 0.0))

println(optimalParameters)
// => DenseVector(-0.0751454743, 2.476293647, 2.23054540)
}

```

That was a mouthful! Let's take this one step at a time. After the obvious imports, we start with:

```
object LogisticRegressionHWData extends App {
```

By extending the built-in `App` trait, we tell Scala to treat the entire object as a `main` function. This just cuts out `def main(args:Array[String])` boilerplate. We then load the data and rescale the height and weight to have a mean of zero and a standard deviation of one:

```

def rescaled(v:DenseVector[Double]) =
    (v - mean(v)) / stddev(v)

val rescaledHeights = rescaled(data.heights)
val rescaledWeights = rescaled(data.weights)

```

The `rescaledHeights` and `rescaledWeights` vectors will be the features of our model. We can now build the training set matrix for this model. This is a $(181 * 3)$ matrix, for which the i -th row is $(1, \text{height}(i), \text{weight}(i))$, corresponding to the values of the height and weight for the i th participant. We start by transforming both `rescaledHeights` and `rescaledWeights` from vectors to $(181 * 1)$ matrices

```
val rescaledHeightsAsMatrix = rescaledHeights.toDenseMatrix.t
val rescaledWeightsAsMatrix = rescaledWeights.toDenseMatrix.t
```

We must also create a $(181 * 1)$ matrix containing just 1 to act as the dummy feature. We can do this using:

```
DenseMatrix.ones[Double](rescaledHeightsAsMatrix.rows, 1)
```

We now need to combine our three $(181 * 1)$ matrices together into a single feature matrix of shape $(181 * 3)$. We can use the `horzcat` method to concatenate the three matrices together:

```
val featureMatrix = DenseMatrix.horzcat(
  DenseMatrix.ones[Double](rescaledHeightsAsMatrix.rows, 1),
  rescaledHeightsAsMatrix,
  rescaledWeightsAsMatrix
)
```

The final step in the data preprocessing stage is to create the target variable. We need to convert the `data.genders` vector to a vector of ones and zeros. We assign a value of one for men and zero for women. Thus, our classifier will predict the probability that any given person is male. We will use the `.values.map` method, a method equivalent to the `.map` method on Scala collections:

```
val target = data.genders.values.map {
  gender => if(gender == 'M') 1.0 else 0.0
}
```

Note that we could also have used the indicator function which we discovered earlier:

```
val maleVector = DenseVector.fill(data.genders.size)('M')
val target = I(data.genders :== maleVector)
```

This results in the allocation of a temporary array, `maleVector`, and might therefore increase the program's memory footprint if there were many participants in the experiment.

We now have a matrix representing the training set and a vector denoting the target variable. We can write the loss function that we want to minimize. As mentioned previously, we will minimize $-\log(L(\text{parameters} | \text{training}))$. The loss function takes as input a set of values for the linear coefficients and returns a number indicating how well those values of the linear coefficients fit the training data:

```
def costFunction(parameters:DenseVector[Double]):Double = {
    val xBeta = featureMatrix * parameters
    val expXBeta = exp(xBeta)
    - sum((target :* xBeta) - log1p(expXBeta))
}
```

Note that we use `log1p(x)` to calculate $\log(1+x)$. This is robust to underflow for small values of x.

Let's explore the cost function:

```
costFunction(DenseVector(0.0, 0.0, 0.0)) // 125.45963968135031
costFunction(DenseVector(0.0, 0.1, 0.1)) // 113.33336518036882
costFunction(DenseVector(0.0, -0.1, -0.1)) // 139.17134594294433
```

We can see that the cost function is somewhat lower for slightly positive values of the height and weight parameters. This indicates that the likelihood function is larger for slightly positive values of the height and weight. This, in turn, implies (as we expect from the plot) that people who are taller and heavier than average are more likely to be male.

We also need a function that calculates the gradient of the loss function, since that will help with the optimization:

```
def costFunctionGradient(parameters:DenseVector[Double]):DenseVector[Double] = {
    val xBeta = featureMatrix * parameters
    val probs = sigmoid(xBeta)
    featureMatrix.t * (probs - target)
}
```

Having defined the loss function and gradient, we are now in a position to set up the optimization:

```
val f = new DiffFunction[DenseVector[Double]] {
    def calculate(parameters:DenseVector[Double]) =
        (costFunction(parameters), costFunctionGradient(parameters))
}
```

All that is left now is to run the optimization. The cost function for logistic regression is convex (it has a single minimum), so the starting point for optimization is irrelevant in principle. In practice, it is common to start with a coefficient vector that is zero everywhere (equating to assigning a 0.5 probability of being male to every participant):

```
val optimalParameters = minimize(f, DenseVector(0.0, 0.0, 0.0))
```

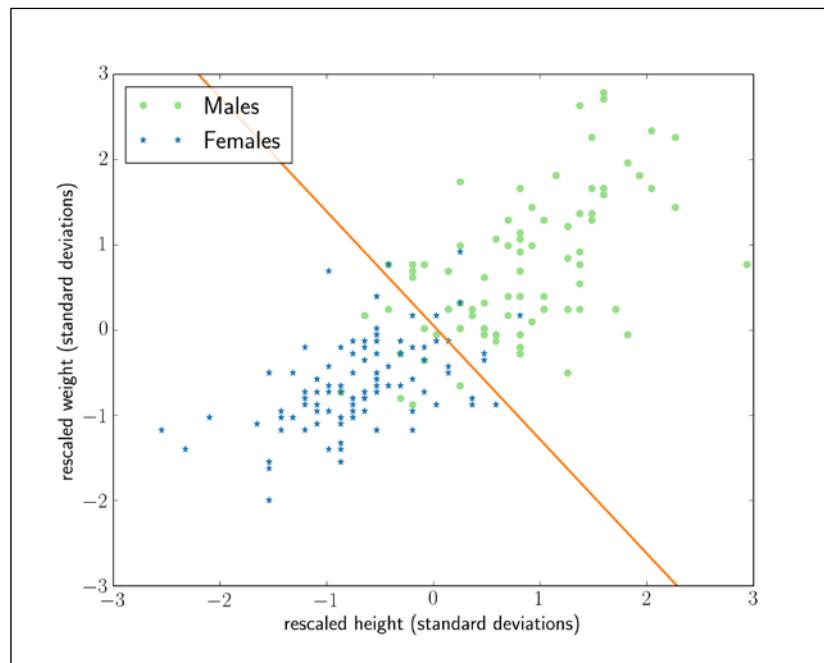
This returns the vector of optimal parameters:

```
DenseVector(-0.0751454743, 2.476293647, 2.23054540)
```

How can we interpret the values of the optimal parameters? The coefficients for the height and weight are both positive, indicating that people who are taller and heavier are more likely to be male.

We can also get the decision boundary (the line separating (height, weight) pairs more likely to belong to a woman from (height, weight) pairs more likely to belong to a man) directly from the coefficients. The decision boundary is:

$$-0.075 + 2.48 \text{ rescaledHeight} + 2.23 \text{ rescaledWeight} = 0$$



Height and weight data (shifted by the mean and rescaled by the standard deviation). The orange line is the logistic regression decision boundary. Logistic regression predicts that individuals above the boundary are male.

Towards re-usable code

In the previous section, we performed all of the computation in a single script. While this is fine for data exploration, it means that we cannot reuse the logistic regression code that we have built. In this section, we will start the construction of a machine learning library that you can reuse across different projects.

We will factor the logistic regression algorithm out into its own class. We construct a `LogisticRegression` class:

```
import breeze.linalg._
import breeze.numerics._
import breeze.optimize._

class LogisticRegression(
    val training:DenseMatrix[Double],
    val target:DenseVector[Double])
{
```

The class takes, as input, a matrix representing the training set and a vector denoting the target variable. Notice how we assign these to `vals`, meaning that they are set on class creation and will remain the same until the class is destroyed. Of course, the `DenseMatrix` and `DenseVector` objects are mutable, so the values that `training` and `target` point to might change. Since programming best practice dictates that mutable state makes reasoning about program behavior difficult, we will avoid taking advantage of this mutability.

Let's add a method that calculates the cost function and its gradient:

```
def costFunctionAndGradient(coefficients:DenseVector[Double]):(Double, DenseVector[Double]) = {
    val xBeta = training * coefficients
    val expXBeta = exp(xBeta)
    val cost = - sum((target :* xBeta) - log1p(expXBeta))
    val probs = sigmoid(xBeta)
    val grad = training.t * (probs - target)
    (cost, grad)
}
```

We are now all set up to run the optimization to calculate the coefficients that best reproduce the training set. In traditional object-oriented languages, we might define a `getOptimalCoefficients` method that returns a `DenseVector` of the coefficients. Scala, however, is more elegant. Since we have defined the `training` and `target` attributes as `vals`, there is only one possible set of values of the optimal coefficients. We could, therefore, define a `val optimalCoefficients = ???` class attribute that holds the optimal coefficients. The problem with this is that it forces all the computation to happen when the instance is constructed. This will be unexpected for the user and might be wasteful: if the user is only interested in accessing the cost function, for instance, the time spent minimizing it will be wasted. The solution is to use a `lazy val`. This value will only be evaluated when the client code requests it:

```
lazy val optimalCoefficients = ???
```

To help with the calculation of the coefficients, we will define a private helper method:

```
private def calculateOptimalCoefficients  
: DenseVector[Double] = {  
    val f = new DiffFunction[DenseVector[Double]] {  
        def calculate(parameters:DenseVector[Double]) =  
            costFunctionAndGradient(parameters)  
    }  
  
    minimize(f, DenseVector.zeros[Double](training.cols))  
}
```



```
lazy val optimalCoefficients = calculateOptimalCoefficients
```

We have refactored the logistic regression into its own class, that we can reuse across different projects.

If we were planning on reusing the height-weight data, we could, similarly, refactor it into a class of its own that facilitates data loading, feature scaling, and any other functionality that we find ourselves reusing often.

Alternatives to Breeze

Breeze is the most feature-rich and approachable Scala framework for linear algebra and numeric computation. However, do not take my word for it: experiment with other libraries for tabular data. In particular, I recommend trying *Saddle*, which provides a `Frame` object similar to data frames in pandas or R. In the Java world, the *Apache Commons Maths library* provides a very rich toolkit for numerical computation. In *Chapter 10, Distributed Batch Processing with Spark*, *Chapter 11, Spark SQL and DataFrames*, and *Chapter 12, Distributed Machine Learning with MLlib*, we will explore *Spark* and *MLlib*, which allow the user to run distributed machine learning algorithms.

Summary

This concludes our brief overview of Breeze. We have learned how to manipulate basic Breeze data types, how to use them for linear algebra, and how to perform convex optimization. We then used our knowledge to clean a real dataset and performed logistic regression on it.

In the next chapter, we will discuss *breeze-viz*, a plotting library for Scala.

References

The Elements of Statistical Learning, by *Hastie, Tibshirani, and Friedman*, gives a lucid, practical description of the mathematical underpinnings of machine learning. Anyone aspiring to do more than mindlessly apply machine learning algorithms as black boxes ought to have a well-thumbed copy of this book.

Scala for Machine Learning, by *Patrick R. Nicholas*, describes practical implementations of many useful machine learning algorithms in Scala.

The Breeze documentation (<https://github.com/scalanlp/breeze/wiki/Quickstart>), API docs (<http://www.scalanlp.org/api/breeze/#package>), and source code (<https://github.com/scalanlp/breeze>) provide the most up-to-date sources of documentation on Breeze.

3

Plotting with breeze-viz

Data visualization is an integral part of data science. Visualization needs fall into two broad categories: during the development and validation of new models and, at the end of the pipeline, to distill meaning from the data and the models to provide insight to external stakeholders.

The two types of visualizations are quite different. At the data exploration and model development stage, the most important feature of a visualization library is its ease of use. It should take as few steps as possible to go from having data as arrays of numbers (or CSVs or in a database) to having data displayed on a screen. The lifetime of graphs is also quite short: once the data scientist has learned all he can from the graph or visualization, it is normally discarded. By contrast, when developing visualization widgets for external stakeholders, one is willing to tolerate increased development time for greater flexibility. The visualizations can have significant lifetime, especially if the underlying data changes over time.

The tool of choice in Scala for the first type of visualization is breeze-viz. When developing visualizations for external stakeholders, web-based visualizations (such as D3) and Tableau tend to be favored.

In this chapter, we will explore breeze-viz. In *Chapter 14, Visualization with D3 and the Play Framework*, we will learn how to build Scala backends for JavaScript visualizations.

Breeze-viz is (no points for guessing) Breeze's visualization library. It wraps **JFreeChart**, a very popular Java charting library. Breeze-viz is still very experimental. In particular, it is much less feature-rich than matplotlib in Python, or R or MATLAB. Nevertheless, breeze-viz allows access to the underlying JFreeChart objects so one can always fall back to editing these objects directly. The syntax for breeze-viz is inspired by MATLAB and matplotlib.

Diving into Breeze

Let's get started. We will work in the Scala console, but a program similar to this example is available in `BreezeDemo.scala` in the examples corresponding to this chapter. Create a `build.sbt` file with the following lines:

```
scalaVersion := "2.11.7"

libraryDependencies ++= Seq(
  "org.scalanlp" %% "breeze" % "0.11.2",
  "org.scalanlp" %% "breeze-viz" % "0.11.2",
  "org.scalanlp" %% "breeze-natives" % "0.11.2"
)
```

Start an sbt console:

```
$ sbt console

scala> import breeze.linalg._
import breeze.linalg._

scala> import breeze.plot._
import breeze.plot._

scala> import breeze.numerics._
import breeze.numerics._
```

Let's start by plotting a sigmoid curve, $f(x) = 1/(1+e^{-x})$. We will first generate the data using Breeze. Recall that the `linspace` method creates a vector of doubles, uniformly distributed between two values:

```
scala> val x = linspace(-4.0, 4.0, 200)
x: DenseVector[Double] = DenseVector(-4.0, -3.959798...,

scala> val fx = sigmoid(x)
fx: DenseVector[Double] = DenseVector(0.0179862099620915,...
```

We now have the data ready for plotting. The first step is to create a figure:

```
scala> val fig = Figure()
fig: breeze.plot.Figure = breeze.plot.Figure@37e36de9
```

This creates an empty Java Swing window (which may appear on your taskbar or equivalent). A figure can contain one or more plots. Let's add a plot to our figure:

```
scala> val plt = fig.subplot(0)
plt: breeze.plot.Plot = breeze.plot.Plot@171c2840
```

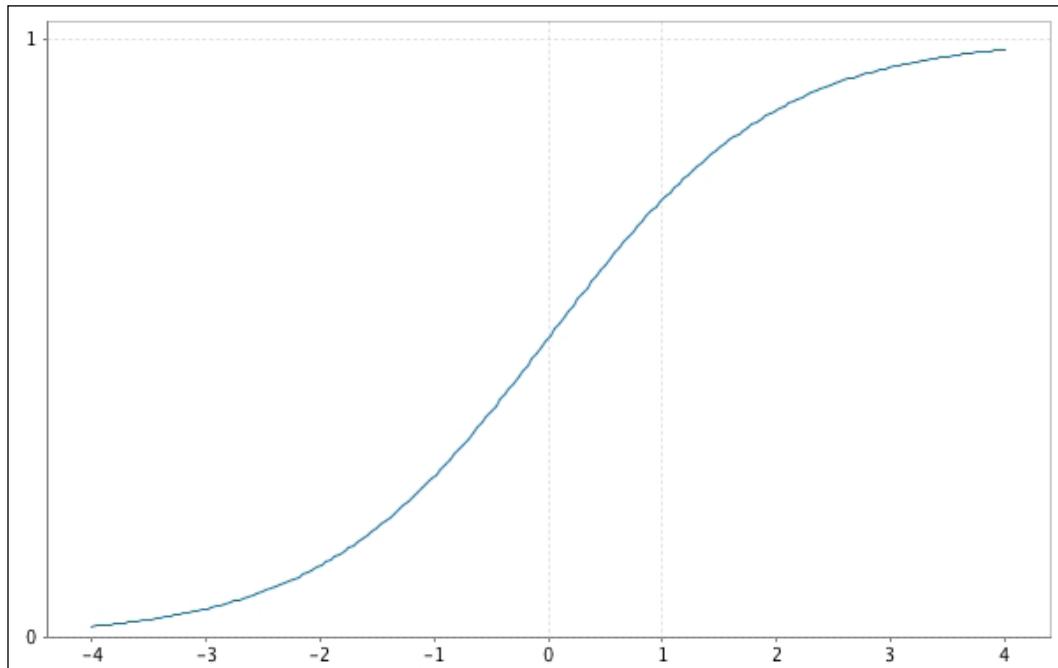
For now, let's ignore the `0` passed as argument to `.subplot`. We can add data points to our plot:

```
scala> plt += plot(x, fx)
breeze.plot.Plot = breeze.plot.Plot@63d6a0f8
```

The `plot` function takes two arguments, corresponding to the x and y values of the data series to be plotted. To view the changes, you need to refresh the figure:

```
scala> fig.refresh()
```

Look at the Swing window now. You should see a beautiful sigmoid, similar to the one below. Right-clicking on the window lets you interact with the plot and save the image as a PNG:



You can also save the image programmatically as follows:

```
scala> fig.saveas("sigmoid.png")
```

Breeze-viz currently only supports exporting to PNG.

Customizing plots

We now have a curve on our chart. Let's add a few more:

```
scala> val f2x = sigmoid(2.0*x)
f2x: breeze.linalg.DenseVector[Double] = DenseVector(3.353501304664E-4...

scala> val f10x = sigmoid(10.0*x)
f10x: breeze.linalg.DenseVector[Double] = DenseVector(4.2483542552
9E-18...

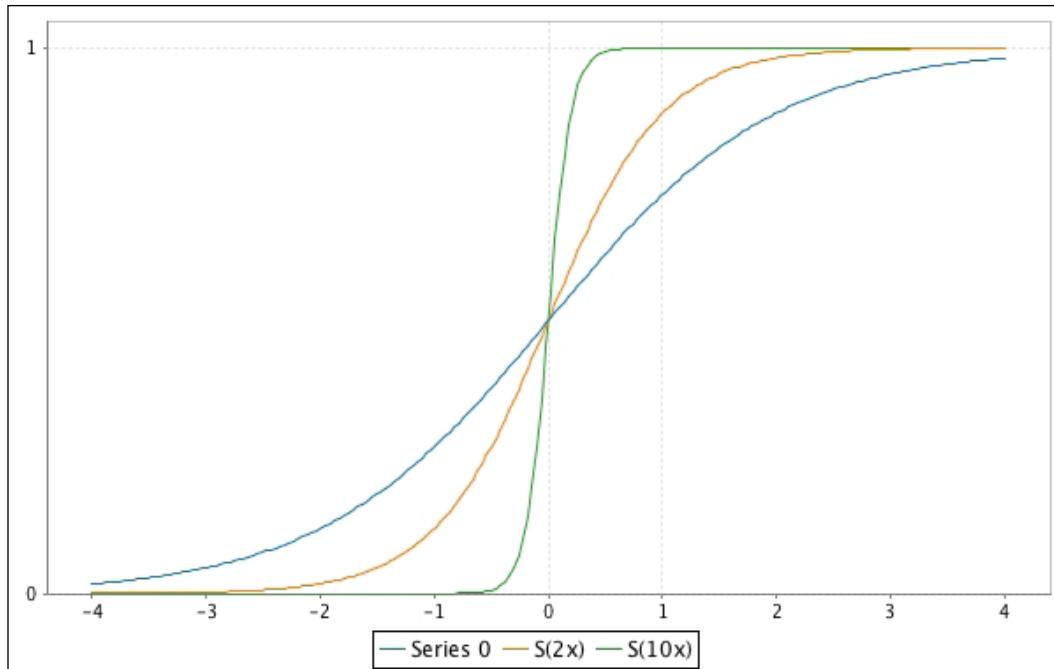
scala> plt += plot(x, f2x, name="S(2x)")
breeze.plot.Plot = breeze.plot.Plot@63d6a0f8

scala> plt += plot(x, f10x, name="S(10x)")
breeze.plot.Plot = breeze.plot.Plot@63d6a0f8

scala> fig.refresh()
```

Looking at the figure now, you should see all three curves in different colors. Notice that we named the data series as we added them to the plot, using the `name=""` keyword argument. To view the names, we must set the `legend` attribute:

```
scala> plt.legend = true
```



Our plot still leaves a lot to be desired. Let's start by restricting the range of the x axis to remove the bands of white space on either side of the plot:

```
scala> plt.xlim = (-4.0, 4.0)
plt.xlim: (Double, Double) = (-4.0, 4.0)
```

Now, notice how, while the x ticks are sensibly spaced, there are only two y ticks: at 0 and 1. It would be useful to have ticks every 0.1 increment. Breeze does not provide a way to set this directly. Instead, it exposes the underlying JFreeChart Axis object belonging to the current plot:

```
scala> plt.yaxis
org.jfree.chart.axis.NumberAxis = org.jfree.chart.axis.NumberAxis@0
```

The Axis object supports a `.setTickUnit` method that lets us set the tick spacing:

```
scala> import org.jfree.chart.axis.NumberTickUnit
import org.jfree.chart.axis.NumberTickUnit

scala> plt.yaxis.setTickUnit(new NumberTickUnit(0.1))
```

JFreeChart allows extensive customization of the `Axis` object. For a full list of methods available, consult the JFreeChart documentation (<http://www.jfree.org/jfreechart/api/javadoc/org/jfree/chart/axis/Axis.html>).

Let's also add a vertical line at $x=0$ and a horizontal line at $f(x)=1$. We will need to access the underlying JFreeChart plot to add these lines. This is available (somewhat confusingly) as the `.plot` attribute in our Breeze Plot object:

```
scala> plt.plot
org.jfree.chart.plot.XYPlot = org.jfree.chart.plot.XYPlot@17e4db6c
```

We can use the `.addDomainMarker` and `.addRangeMarker` methods to add vertical and horizontal lines to JFreeChart `XYPlot` objects:

```
scala> import org.jfree.chart.plot.ValueMarker
import org.jfree.chart.plot.ValueMarker

scala> plt.plot.addDomainMarker(new ValueMarker(0.0))

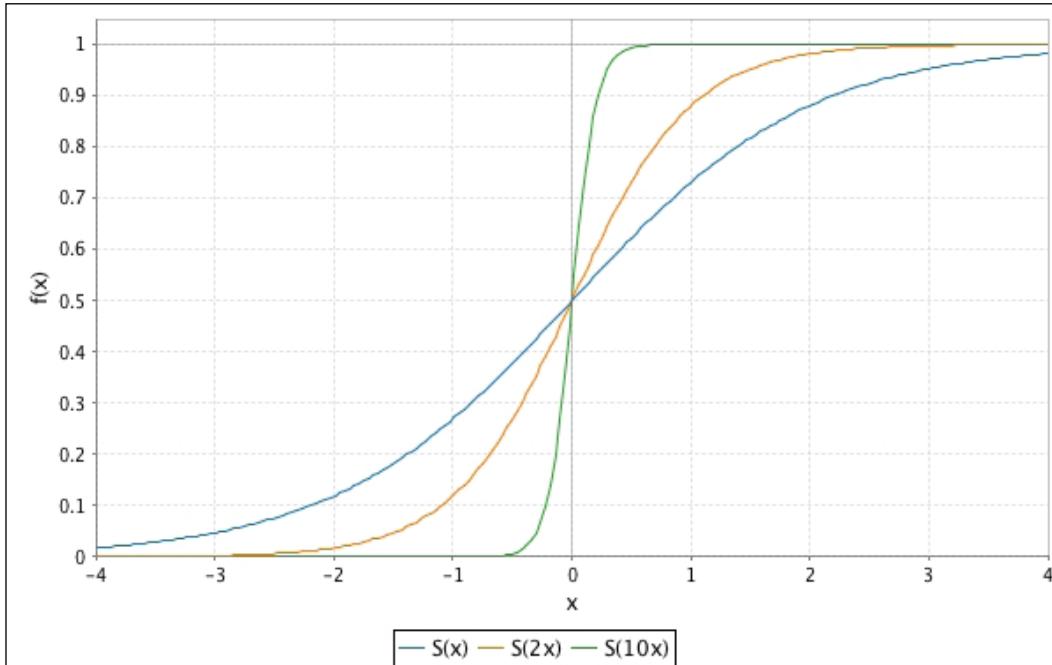
scala> plt.plot.addRangeMarker(new ValueMarker(1.0))
```

Let's also add labels to the axes:

```
scala> plt.xlabel = "x"
plt.xlabel: String = x

scala> plt.ylabel = "f(x)"
plt.ylabel: String = f(x)
```

If you have run all these commands, you should have a graph that looks like this:



We now know how to customize the basic building blocks of a graph. The next step is to learn how to change how curves are drawn.

Customizing the line type

So far, we have just plotted lines using the default settings. Breeze lets us customize how lines are drawn, at least to some extent.

For this example, we will use the height-weight data discussed in *Chapter 2, Manipulating Data with Breeze*. We will use the Scala shell here for demonstrative purposes, but you will find a program in `BreezeDemo.scala` that follows the example shell session.

The code examples for this chapter come with a module for loading the data, `HWData.scala`, that loads the data from the CSVs:

```
scala> val data = HWData.load
data: HWData = HWData [ 181 rows ]

scala> data.heights
```

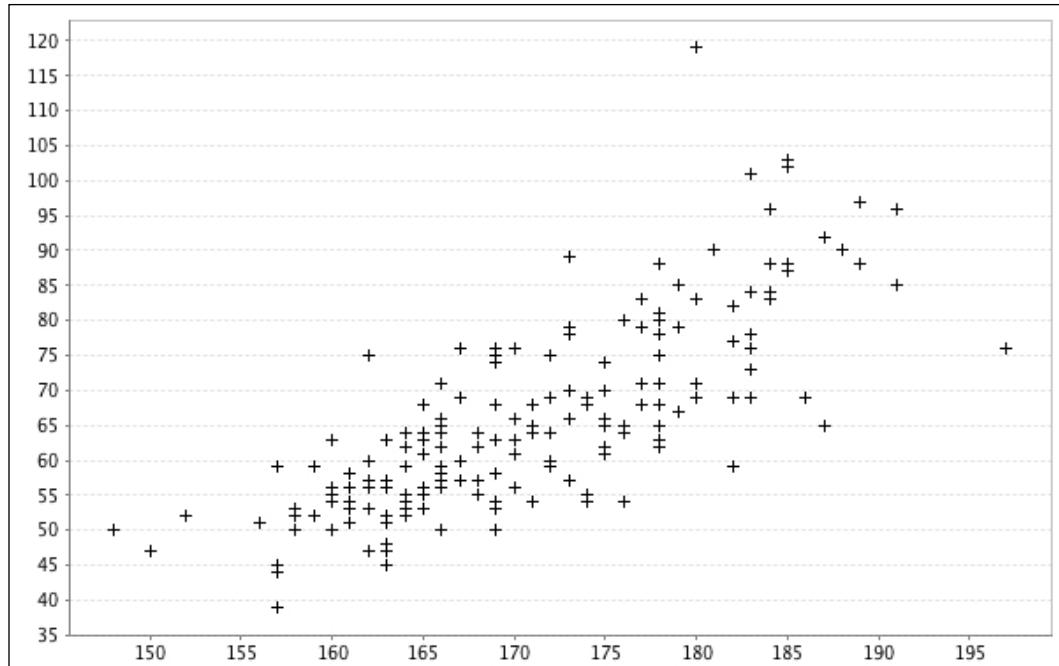
Plotting with breeze-viz

```
breeze.linalg.DenseVector[Double] = DenseVector(182.0, ...  
  
scala> data.weights  
breeze.linalg.DenseVector[Double] = DenseVector(77.0, 58.0...)
```

Let's create a scatter plot of the heights against the weights:

```
scala> val fig = Figure("height vs. weight")  
fig: breeze.plot.Figure = breeze.plot.Figure@743f2558  
  
scala> val plt = fig.subplot(0)  
plt: breeze.plot.Plot = breeze.plot.Plot@501ea274  
  
scala> plt += plot(data.heights, data.weights, '+',  
colorcode="black")  
breeze.plot.Plot = breeze.plot.Plot@501ea274
```

This produces a scatter-plot of the height-weight data:



Note that we passed a third argument to the `plot` method, `'+'`. This controls the plotting style. As of this writing, there are three available styles: `'-'` (the default), `'+'`, and `'..'`. Experiment with these to see what they do. Finally, we pass a `colorcode="black"` argument to control the color of the line. This is either a color name or an RGB triple, written as a string. Thus, to plot red points, we could have passed `colorcode=" [255, 0, 0] "`.

Looking at the height-weight plot, there is clearly a trend between height and weight. Let's try and fit a straight line through the data points. We will fit the following function:

$$\text{weight}(\text{Kg}) = a + b \times \text{height}(\text{cm})$$



Scientific literature suggests that it would be better to fit something more like $\text{mass} \propto \text{height}^2$. You should find it straightforward to fit a quadratic line to the data, should you wish to.

We will use Breeze's least squares function to find the values of a and b . The `leastSquares` method expects an input matrix of features and a target vector, just like the `LogisticRegression` class that we defined in the previous chapter. Recall that in *Chapter 2, Manipulating Data with Breeze*, when we prepared the training set for logistic regression classification, we introduced a dummy feature that was one for every participant to provide the degree of freedom for the y intercept. We will use the same approach here. Our feature matrix, therefore, contains two columns – one that is 1 everywhere and one for the height:

```
scala> val features = DenseMatrix.horzcat(
  DenseMatrix.ones[Double](data.npoints, 1),
  data.heights.toDenseMatrix.t
)
features: breeze.linalg.DenseMatrix[Double] =
1.0  182.0
1.0  161.0
1.0  161.0
1.0  177.0
1.0  157.0
...

```

```
scala> import breeze.stats.regression._
```

```
import breeze.stats.regression._

scala> val leastSquaresResult = leastSquares(features, data.weights)
leastSquaresResult: breeze.stats.regression.LeastSquaresRegressionResult
= <function1>
```

The `leastSquares` method returns an instance of `LeastSquaresRegressionResult`, which contains a `coefficients` attribute containing the coefficients that best fit the data:

```
scala> leastSquaresResult.coefficients
breeze.linalg.DenseVector[Double] = DenseVector(-131.042322, 1.1521875)
```

The best-fit line is therefore:

$$weight(Kg) = -131.04 + 1.1522 \times height(cm)$$

Let's extract the coefficients. An elegant way of doing this is to use Scala's pattern matching capabilities:

```
scala> val Array(a, b) = leastSquaresResult.coefficients.toArray
a: Double = -131.04232269750622
b: Double = 1.1521875435418725
```

By writing `val Array(a, b) = ...`, we are telling Scala that the right-hand side of the expression is a two-element array and to bind the first element of that array to the value `a` and the second to the value `b`. See *Appendix, Pattern Matching and Extractors*, for a discussion of pattern matching.

We can now add the best-fit line to our graph. We start by generating evenly-spaced dummy height values:

```
scala> val dummyHeights = linspace(min(data.heights),
  max(data.heights), 200)
dummyHeights: breeze.linalg.DenseVector[Double] = DenseVector(148.0, ...

scala> val fittedWeights = a :+ (b :* dummyHeights)
fittedWeights: breeze.linalg.DenseVector[Double] = DenseVector(39.4814...

scala> plt += plot(dummyHeights, fittedWeights, colorcode="red")
breeze.plot.Plot = breeze.plot.Plot@501ea274
```

Let's also add the equation for the best-fit line to the graph as an annotation. We will first generate the label:

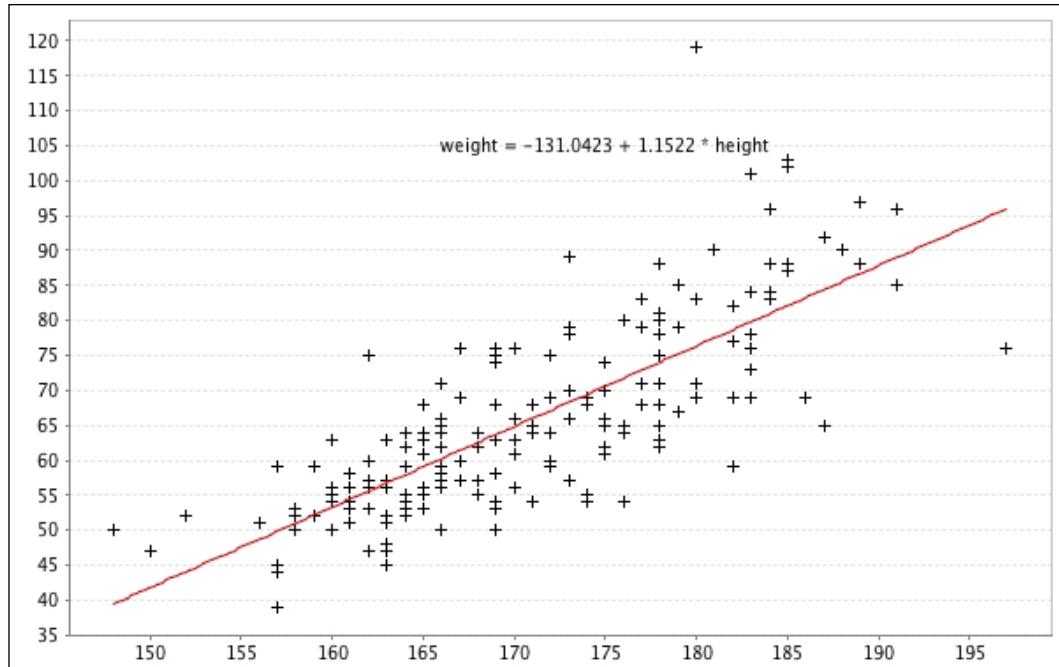
```
scala> val label = f"weight = $a%.4f + $b%.4f * height"
label: String = weight = -131.0423 + 1.1522 * height
```

To add an annotation, we must access the underlying JFreeChart plot:

```
scala> import org.jfree.chart.annotations.XYTextAnnotation
import org.jfree.chart.annotations.XYTextAnnotation
```

```
scala> plt.plot.addAnnotation(new XYTextAnnotation(label, 175.0, 105.0))
```

The `XYTextAnnotation` constructor takes three parameters: the annotation string and a pair of (x, y) coordinates defining the centre of the annotation on the graph. The coordinates of the annotation are expressed in the coordinate system of the data. Thus, calling `new XYTextAnnotation(label, 175.0, 105.0)` generates an annotation whose centroid is at the point corresponding to a height of 175 cm and weight of 105 kg:



More advanced scatter plots

Breeze-viz offers a `scatter` function that adds a significant degree of customization to scatter plots. In particular, we can use the size and color of the marker points to add additional dimensions of information to the plot.

The `scatter` function takes, as its first two arguments, collections of x and y points. The third argument is a function mapping an integer i to a `Double` indicating the size of the i th point. The size of the point is measured in units of the x axis. If you have the sizes as a Scala collection or a Breeze vector, you can use that collection's `apply` method as the function. Let's see how this works in practice.

As with the previous examples, we will use the REPL, but you can find a sample program in `BreezeDemo.scala`:

```
scala> val fig = new Figure("Advanced scatter example")
fig: breeze.plot.Figure = breeze.plot.Figure@220821bc

scala> val plt = fig.subplot(0)
plt: breeze.plot.Plot = breeze.plot.Plot@668f8ae0

scala> val xs = linspace(0.0, 1.0, 100)
xs: breeze.linalg.DenseVector[Double] = DenseVector(0.0,
0.010101010101010102, 0.0202 ...

scala> val sizes = 0.025 * DenseVector.rand(100) // random sizes
sizes: breeze.linalg.DenseVector[Double] =
DenseVector(0.014879265631723166, 0.00219551...

scala> plt += scatter(xs, xs :^ 2.0, sizes.apply)
breeze.plot.Plot = breeze.plot.Plot@668f8ae0
```

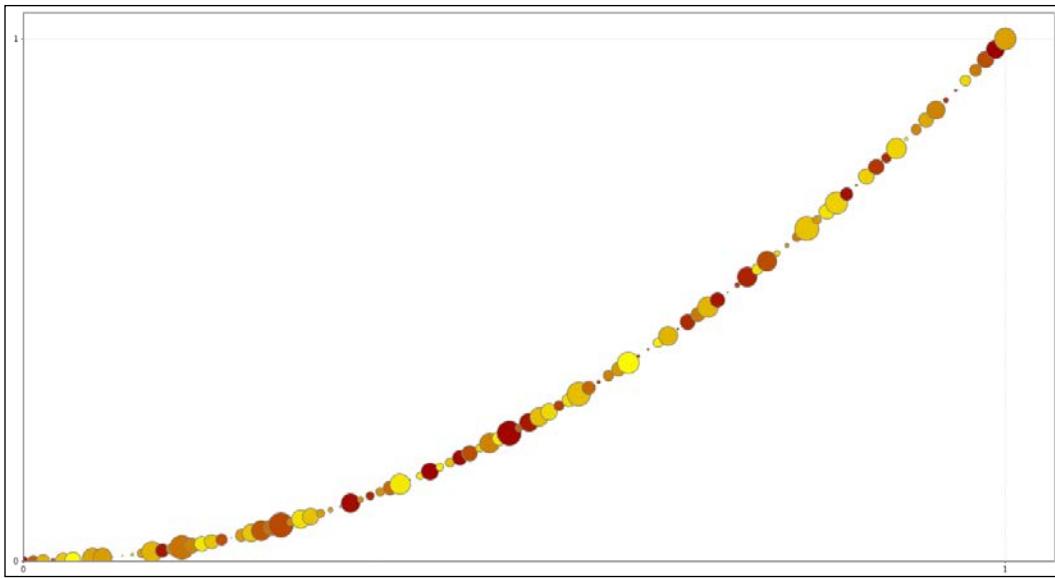
Selecting custom colors works in a similar manner: we pass in a `colors` argument that maps an integer index to a `java.awt.Paint` object. Using these directly can be cumbersome, so Breeze provides some default palettes. For instance, the `GradientPaintScale` maps doubles in a given domain to a uniform color gradient. Let's map doubles in the range 0.0 to 1.0 to the colors between red and green:

```
scala> val palette = new GradientPaintScale(  
    0.0, 1.0, PaintScale.RedToGreen)  
palette: breeze.plot.GradientPaintScale[Double] = <function1>  
  
scala> palette(0.5) // half-way between red and green  
java.awt.Paint = java.awt.Color[r=127,g=127,b=0]  
  
scala> palette(1.0) // green  
java.awt.Paint = java.awt.Color[r=0,g=254,b=0]
```

Besides the `GradientPaintScale`, `breeze-viz` provides a `CategoricalPaintScale` class for categorical palettes. For an overview of the different palettes, consult the source file `PaintScale.scala` at `scala: https://github.com/scalanlp/breeze/blob/master/viz/src/main/scala/breeze/plot/PaintScale.scala`.

Let's use our newfound knowledge to draw a multicolor scatter plot. We will assume the same initialization as the previous example. We will assign a random color to each point:

```
scala> val palette = new GradientPaintScale(0.0, 1.0,  
    PaintScale.MaroonToGold)  
palette: breeze.plot.GradientPaintScale[Double] = <function1>  
  
scala> val colors = DenseVector.rand(100).mapValues(palette)  
colors: breeze.linalg.DenseVector[java.awt.Paint] = DenseVector(java.awt.  
Color[r=162,g=5,b=0], ...  
  
scala> plt += scatter(xs, xs ^ 2.0, sizes.apply, colors.apply)  
breeze.plot.Plot = breeze.plot.Plot@8ff7e27
```



Multi-plot example – scatterplot matrix plots

In this section, we will learn how to have several plots in the same figure.

The key new method that allows multiple plots in the same figure is `fig.subplot(nrows, ncols, plotIndex)`. This method, an overloaded version of the `fig.subplot` method we have been using up to now, both sets the number of rows and columns in the figure and returns a specific subplot. It takes three arguments:

- `nrows`: The number of rows of subplots in the figure
- `ncols`: The number of columns of subplots in the figure
- `plotIndex`: The index of the plot to return

Users familiar with MATLAB or matplotlib will note that the `.subplot` method is identical to the eponymous methods in these frameworks. This might seem a little complex, so let's look at an example (you will find the code for this in `BreezeDemo.scala`):

```
import breeze.plot._

def subplotExample {
    val data = HWData.load
```

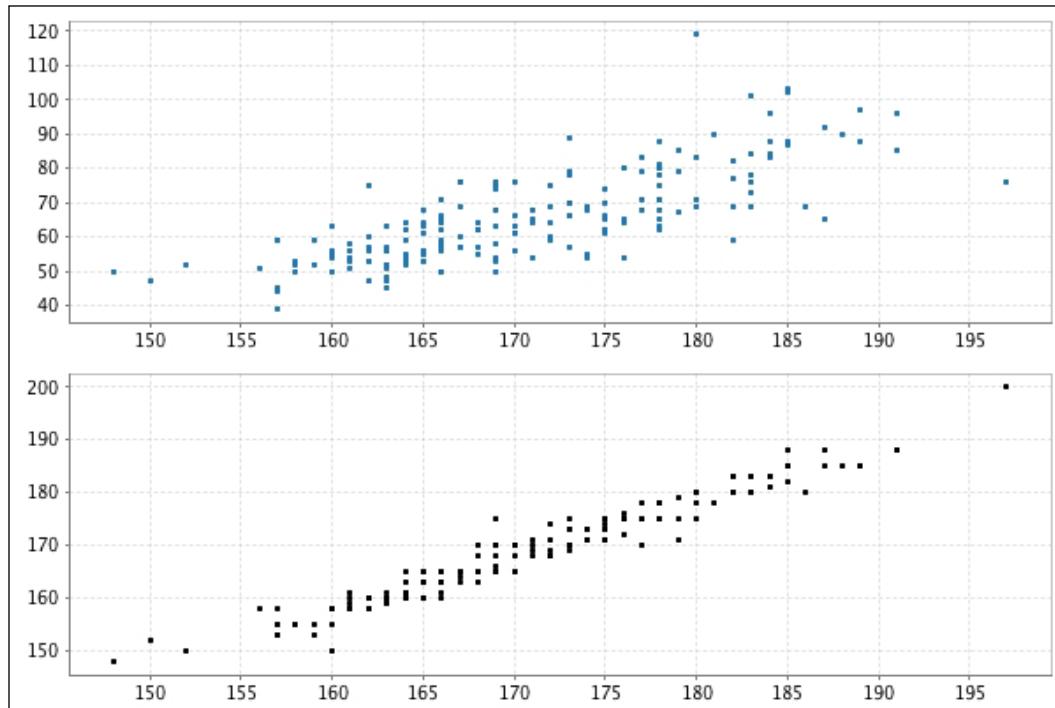
```
val fig = new Figure("Subplot example")

// upper subplot: plot index '0' refers to the first plot
var plt = fig.subplot(2, 1, 0)
plt += plot(data.heights, data.weights, '.')

// lower subplot: plot index '1' refers to the second plot
plt = fig.subplot(2, 1, 1)
plt += plot(data.heights, data.reportedHeights, '.',
            colorcode="black")

fig.refresh
}
```

Running this example produces the following plot:



Now that we have a basic grasp of how to add several subplots to the same figure, let's do something a little more interesting. We will write a class to draw scatterplot matrices. These are useful for exploring correlations between different features.

If you are not familiar with scatterplot matrices, have a look at the figure at the end of this section for an idea of what we are constructing. The idea is to build a square matrix of scatter plots for each pair of features. Element (i, j) in the matrix is a scatter plot of feature i against feature j . Since a scatter plot of a variable against itself is of limited use, one normally draws histograms of each feature along the diagonal. Finally, since a scatter plot of feature i against feature j contains the same information as a scatter plot of feature j against feature i , one normally only plots the upper triangle or the lower triangle of the matrix.

Let's start by writing functions for the individual plots. These will take a `Plot` object referencing the correct subplot and vectors of the data to plot:

```
import breeze.plot._  
import breeze.linalg._  
  
class ScatterplotMatrix(val fig:Figure) {  
  
    /** Draw the histograms on the diagonal */  
    private def plotHistogram(plt:Plot)  
    (data:DenseVector[Double], label:String) {  
        plt += hist(data)  
        plt.xlabel = label  
    }  
  
    /** Draw the off-diagonal scatter plots */  
    private def plotScatter(plt:Plot)  
    (xdata:DenseVector[Double],  
     ydata:DenseVector[Double],  
     xlabel:String,  
     ylabel:String) {  
        plt += plot(xdata, ydata, '.')  
        plt.xlabel = xlabel  
        plt.ylabel = ylabel  
    }  
  
    ...  
}
```

Notice the use of `hist(data)` to draw a histogram. The argument to `hist` must be a vector of data points. The `hist` method will bin these and represent them as a histogram.

Now that we have the machinery for drawing individual plots, we just need to wire everything together. The tricky part is to know how to select the correct subplot for a given row and column position in the matrix. We can select a single plot by calling `fig.subplot(nrows, ncolumns, plotIndex)`, but translating from a $(row, column)$ index pair to a single `plotIndex` is not obvious. The plots are numbered in increasing order, first from left to right, then from top to bottom:

```
0 1 2 3  
4 5 6 7  
...
```

Let's write a short function to select a plot at a $(row, column)$ index pair:

```
private def selectPlot(ncols:Int) (irow:Int, icol:Int):Plot = {  
    fig.subplot(ncols, ncols, (irow)*ncols + icol)  
}
```

We are now in a position to draw the matrix plot itself:

```
/** Draw a scatterplot matrix.  
 *  
 * This function draws a scatterplot matrix of the correlation  
 * between each pair of columns in `featureMatrix`.  
 *  
 * @param featureMatrix A matrix of features, with each column  
 *   representing a feature.  
 * @param labels Names of the features.  
 */  
def plotFeatures(  
    featureMatrix:DenseMatrix[Double],  
    labels>List[String]  
) {  
    val ncols = featureMatrix.cols  
    require(ncols == labels.size,  
        "Number of columns in feature matrix "+  
        "must match length of labels")  
}  
fig.clear  
fig.subplot(ncols, ncols, 0)  
  
(0 until ncols) foreach { irow =>  
    val p = selectPlot(ncols) (irow, irow)  
    plotHistogram(p)(featureMatrix(::, irow), labels(irow))  
  
(0 until irow) foreach { icol =>
```

```
    val p = selectPlot(ncols) (irow, icol)
    plotScatter(p) (
      featureMatrix(::, irow),
      featureMatrix(::, icol),
      labels(irow),
      labels(icolumn)
    )
  }
}
}
```

Let's write an example for our class. We will use the height-weight data again:

```
import breeze.linalg._
import breeze.numerics._
import breeze.plot._

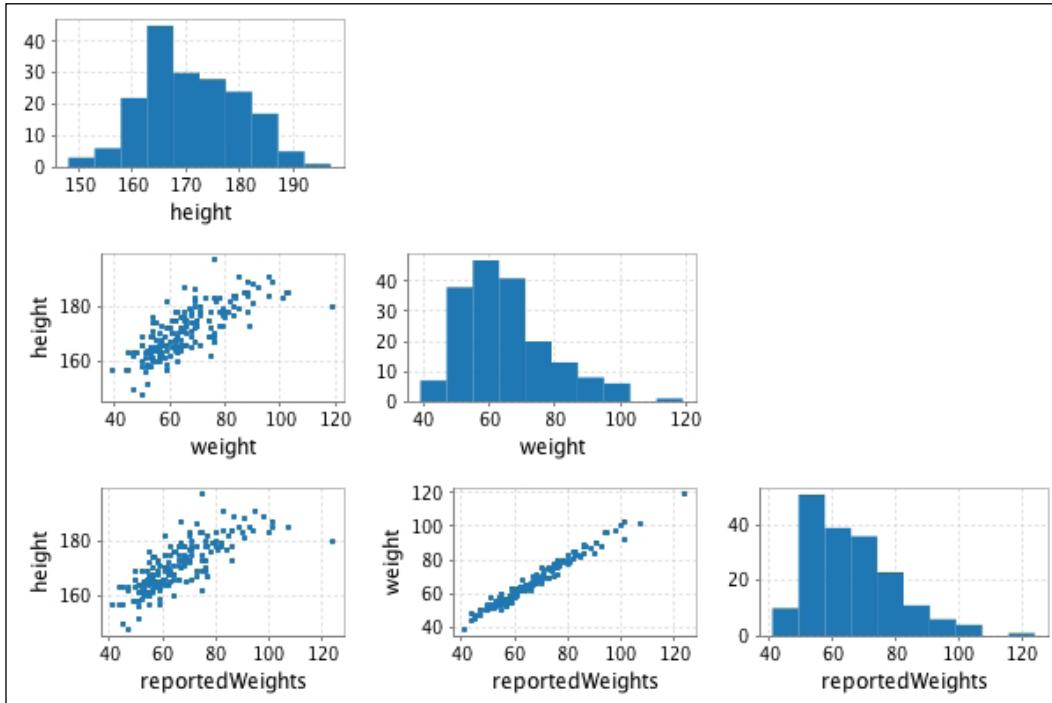
object ScatterplotMatrixDemo extends App {

  val data = HWData.load
  val m = new ScatterplotMatrix(Figure("Scatterplot matrix demo"))

  // Make a matrix with three columns: the height, weight and
  // reported weight data.
  val featureMatrix = DenseMatrix.horzcat(
    data.heights.toDenseMatrix.t,
    data.weights.toDenseMatrix.t,
    data.reportedWeights.toDenseMatrix.t
  )
  m.plotFeatures(featureMatrix,
    List("height", "weight", "reportedWeights"))

}
```

Running this through SBT produces the following plot:



Managing without documentation

Breeze-viz is unfortunately rather poorly documented. This can make the learning curve somewhat steep. Fortunately, it is still quite a small project: at the time of writing, there are just ten source files (<https://github.com/scalanlp/breeze/tree/master/viz/src/main/scala/breeze/plot>). A good way to understand exactly what breeze-viz does is to read the source code. For instance, to see what methods are available on a `Plot` object, read the source file `Plot.scala`. If you need functionality beyond that provided by Breeze, consult the documentation for JFreeChart to discover if you can implement what you need by accessing the underlying JFreeChart objects.

Breeze-viz reference

Writing a reference in a programming book is a dangerous exercise: you quickly become out of date. Nevertheless, given the paucity of documentation for breeze-viz, this section becomes more relevant – it is easier to compete against something that does not exist. Take this section with a pinch of salt, and if a command in this section does not work, head over to the source code:

Command	Description
<code>plt += plot(xs, ys)</code>	This plots a series of (xs, ys) values. The xs and ys values must be collection-like objects (Breeze vectors, Scala arrays, or lists, for instance).
<code>plt += scatter(xs, ys, size)</code> <code>plt += scatter(xs, ys, size, color)</code>	This plots a series of (xs, ys) values as a scatter plot. The size argument is an <code>(Int) => Double</code> function mapping the index of a point to its size (in the same units as the x axis). The color argument is an <code>(Int) => java.awt.Paint</code> function mapping from integers to colors. Read the <i>more advanced scatter plots</i> section for further details.
<code>plt += hist(xs)</code> <code>plt += hist(xs, bins=10)</code>	This bins xs and plots a histogram. The bins argument controls the number of bins.
<code>plt += image(mat)</code>	This plots an image or matrix. The mat argument should be <code>Matrix[Double]</code> . Read the package <code>scala</code> source file in <code>breeze.plot</code> for details (https://github.com/scalanlp/breeze/blob/master/viz/src/main/scala/breeze/plot/package.scala).

It is also useful to summarize the options available on a `plot` object:

Attribute	Description
<code>plt.xlabel = "x-label"</code>	This sets the axis label
<code>plt.ylabel = "y-label"</code>	
<code>plt.xlim = (0.0, 1.0)</code>	This sets the axis maximum and minimum value
<code>plt.ylim = (0.0, 1.0)</code>	
<code>plt.logScaleX = true</code>	This switches the axis to a log scale
<code>plt.logScaleY = true</code>	
<code>plt.title = "title"</code>	This sets the plot title

Data visualization beyond breeze-viz

Other tools for data visualization in Scala are emerging: Spark notebooks (<https://github.com/andypetrella/spark-notebook#description>) based on the IPython notebook and Apache Zeppelin (<https://zeppelin.incubator.apache.org>). Both of these rely on Apache Spark, which we will explore later in this book.

Summary

In this chapter, we learned how to draw simple charts with breeze-viz. In the last chapter of this book, we will learn how to build interactive visualizations using JavaScript libraries.

Next, we will learn about basic Scala concurrency constructs—specifically, parallel collections.

4

Parallel Collections and Futures

Data science often involves processing medium or large amounts of data. Since the previously exponential growth in the speed of individual CPUs has slowed down and the amount of data continues to increase, leveraging computers effectively must entail parallel computation.

In this chapter, we will look at ways of parallelizing computation and data processing over a single computer. Virtually all new computers have more than one processing unit, and distributing a calculation over these cores can be an effective way of hastening medium-sized calculations.

Parallelizing calculations over a single chip is suitable for calculations involving gigabytes or a few terabytes of data. For larger data flows, we must resort to distributing the computation over several computers in parallel. We will discuss Apache Spark, a framework for parallel data processing in *Chapter 10, Distributed Batch Processing with Spark*.

In this book, we will look at three common ways of leveraging parallel architectures in a single machine: parallel collections, futures, and actors. We will consider the first two in this chapter, and leave the study of actors to *Chapter 9, Concurrency with Akka*.

Parallel collections

Parallel collections offer an extremely easy way to parallelize independent tasks. The reader, being familiar with Scala, will know that many tasks can be phrased as operations on collections, such as *map*, *reduce*, *filter*, or *groupBy*. Parallel collections are an implementation of Scala collections that parallelize these operations to run over several threads.

Let's start with an example. We want to calculate the frequency of occurrence of each letter in a sentence:

```
scala> val sentence = "The quick brown fox jumped over the lazy dog"  
sentence: String = The quick brown fox jumped ...
```

Let's start by converting our sentence from a string to a vector of characters:

```
scala> val characters = sentence.toVector  
Vector[Char] = Vector(T, h, e, , q, u, i, c, k, ...)
```

We can now convert characters to a *parallel* vector, a ParVector. To do this, we use the `par` method:

```
scala> val charactersPar = characters.par  
ParVector[Char] = ParVector(T, h, e, , q, u, i, c, k, , ...)
```

ParVector collections support the same operations as regular vectors, but their methods are executed in parallel over several threads.

Let's start by filtering out the spaces in `charactersPar`:

```
scala> val lettersPar = charactersPar.filter { _ != ' ' }  
ParVector[Char] = ParVector(T, h, e, q, u, i, c, k, ...)
```

Notice how Scala hides the execution details. The `filter` operation was performed using multiple threads, and you barely even noticed! The interface and behavior of a parallel vector is identical to its serial counterpart, save for a few details that we will explore in the next section.

Let's now use the `toLowerCase` function to make the letters lowercase:

```
scala> val lowerLettersPar = lettersPar.map { _.toLowerCase }  
ParVector[Char] = ParVector(t, h, e, q, u, i, c, k, ...)
```

As before, the `map` method was applied in parallel. To find the frequency of occurrence of each letter, we use the `groupBy` method to group characters into vectors containing all the occurrences of that character:

```
scala> val intermediateMap = lowerLettersPar.groupBy(identity)  
ParMap[Char, ParVector[Char]] = ParMap(e -> ParVector(e, e, e, e), ...)
```

Note how the `groupBy` method has created a `ParMap` instance, the parallel equivalent of an immutable map. To get the number of occurrences of each letter, we do a `mapValues` call on `intermediateMap`, replacing each vector by its length:

```
scala> val occurrenceNumber = intermediateMap.mapValues { _.length }
ParMap [Char,Int] = ParMap(e -> 4, x -> 1, n -> 1, j -> 1, ...)
```

Congratulations! We've written a multi-threaded algorithm for finding the frequency of occurrence of each letter in a few lines of code. You should find it straightforward to adapt this to find the frequency of occurrence of each word in a document, a common preprocessing problem for analyzing text data.

Parallel collections make it very easy to parallelize some operation pipelines: all we had to do was call `.par` on the `characters` vector. All subsequent operations were parallelized. This makes switching from a serial to a parallel implementation very easy.

Limitations of parallel collections

Part of the power and the appeal of parallel collections is that they present the same interface as their serial counterparts: they have a `map` method, a `foreach` method, a `filter` method, and so on. By and large, these methods work in the same way on parallel collections as they do in serial. There are, however, some notable caveats. The most important one has to do with side effects. If an operation on a parallel collection has a side effect, this may result in a race condition: a situation in which the final result depends on the order in which the threads perform their operations.

Side effects in collections arise most commonly when we update a variable defined outside of the collection. To give a trivial example of unexpected behavior, let's define a `count` variable and increment it a thousand times using a parallel range:

```
scala> var count = 0
count: Int = 0

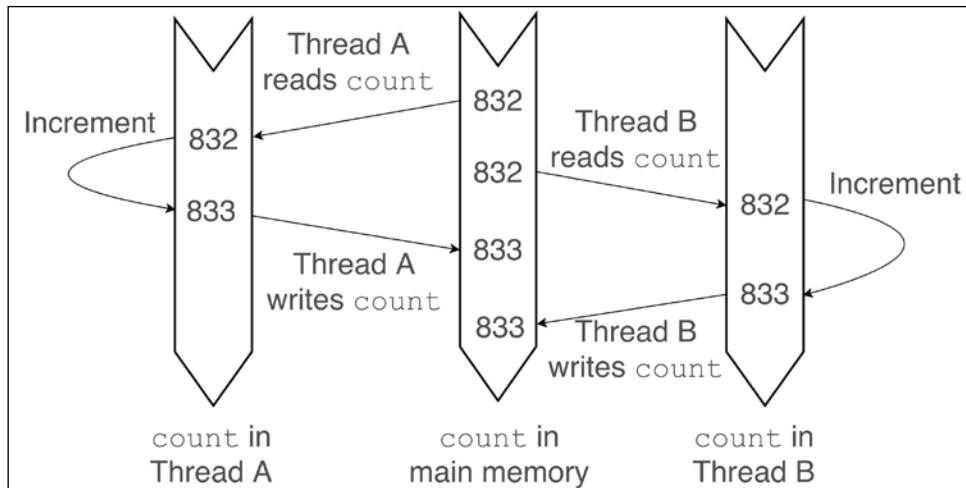
scala> (0 until 1000).par.foreach { i => count += 1 }

scala> count
count: Int = 874 // not 1000!
```

What happened here? The function passed to `foreach` has a side effect: it increments `count`, a variable outside of the scope of the function. This is a problem because the `+=` operator is a sequence of two operations:

- Retrieve the value of `count` and add one to it
- Assign the result back to `count`

To understand why this causes unexpected behavior, let's imagine that the `foreach` loop has been parallelized over two threads. **Thread A** might read the `count` variable when it is `832` and add one to it to give `833`. Before it has time to reassign `833` to `count`, **Thread B** reads `count`, still at `832`, and adds one to give `833`. **Thread A** then assigns `833` to `count`. **Thread B** then assigns `833` to `count`. We've run through two updates but only incremented the count by one. The problem arises because `+=` can be separated into two instructions: it is not *atomic*. This leaves room for threads to interleave their operations:



The anatomy of a race condition: both thread A and thread B are trying to update `count` concurrently, resulting in one of the updates being overwritten. The final value of `count` is `833` instead of `834`.

To give a somewhat more realistic example of problems caused by non-atomicity, let's look at a different method for counting the frequency of occurrence of each letter in our sentence. We define a mutable `Char -> Int` hash map outside of the loop. Each time we encounter a letter, we increment the corresponding integer in the map:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val occurrenceNumber = mutable.Map.empty[Char, Int]
```

```
occurenceNumber: mutable.Map[Char,Int] = Map()

scala> lowerLettersPar.foreach { c =>
    occurenceNumber(c) = occurenceNumber.getOrElse(c, 0) + 1
}

scala> occurenceNumber('e') // Should be 4
Int = 2
```

The discrepancy occurs because of the non-atomicity of the operations in the `foreach` loop.

In general, it is good practice to avoid side effects in higher-order functions on collections. They make the code harder to understand and preclude switching from serial to parallel collections. It is also good practice to avoid exposing mutable state: immutable objects can be shared freely between threads and cannot be affected by side effects.

Another limitation of parallel collections occurs in reduction (or folding) operations. The function used to combine items together must be *associative*. For instance:

```
scala> (0 until 1000).par.reduce {_ - _} // should be -499500
Int = 63620
```

The *minus* operator, `-`, is not associative. The order in which consecutive operations are applied matters: $(a - b) - c$ is not the same as $a - (b - c)$. The function used to reduce a parallel collection must be associative because the order in which the reduction occurs is not tied to the order of the collection.

Error handling

In single-threaded programs, exception handling is relatively straightforward: if an exception occurs, the function can either handle it or escalate it. This is not nearly as obvious when parallelism is introduced: a single thread might fail, but the others might return successfully.

Parallel collection methods will throw an exception if they fail on any element, just like their serial counterparts:

```
scala> Vector(2, 0, 5).par.map { 10 / _ }
java.lang.ArithmetricException: / by zero
...
```

There are cases when this isn't the behavior that we want. For instance, we might be using a parallel collection to retrieve a large number of web pages in parallel. We might not mind if a few of the pages cannot be fetched.

Scala's Try type was designed for sandboxing code that might throw exceptions. It is similar to Option in that it is a one-element container:

```
scala> import scala.util._  
import scala.util._  
  
scala> Try { 2 + 2 }  
Try[Int] = Success(4)
```

Unlike the Option type, which indicates whether an expression has a useful value, the Try type indicates whether an expression can be executed without throwing an exception. It takes on the following two values:

- Try { 2 + 2 } == Success(4) if the expression in the Try statement is evaluated successfully
- Try { 2 / 0 } == Failure(java.lang.ArithmaticException: / by zero) if the expression in the Try block results in an exception

This will make more sense with an example. To see the Try type in action, we will try to fetch web pages in a fault tolerant manner. We will use the built-in Source.fromURL method which fetches a web page and opens an iterator of the page's content. If it fails to fetch the web page, it throws an error:

```
scala> import scala.io.Source  
import scala.io.Source  
  
scala> val html = Source.fromURL("http://www.google.com")  
scala.io.BufferedSource = non-empty iterator  
  
scala> val html = Source.fromURL("garbage")  
java.net.MalformedURLException: no protocol: garbage  
...
```

Instead of letting the expression propagate out and crash the rest of our code, we can wrap the call to Source.fromURL in Try:

```
scala> Try { Source.fromURL("http://www.google.com") }
```

```
Try[BufferedSource] = Success(non-empty iterator)

scala> Try { Source.fromURL("garbage") }
Try[BufferedSource] = Failure(java.net.MalformedURLException: no
protocol: garbage)
```

To see the power of our Try statement, let's now retrieve a list of URLs in parallel in a fault tolerant manner:

```
scala> val URLs = Vector("http://www.google.com",
  "http://www.bbc.co.uk",
  "not-a-url"
)
URLs: Vector[String] = Vector(http://www.google.com, http://www.bbc.
co.uk, not-a-url)

scala> val pages = URLs.par.map { url =>
  url -> Try { Source.fromURL(url) }
}
pages: ParVector[(String, Try[BufferedSource])] = ParVector((http://
www.google.com,Success(non-empty iterator)), (http://www.bbc.
co.uk,Success(non-empty iterator)), (not-a-url,Failure(java.net.
MalformedURLException: no protocol: not-a-url)))
```

We can then use a collect statement to act on the pages we could fetch successfully. For instance, to get the number of characters on each page:

```
scala> pages.collect { case(url, Success(it)) => url -> it.size }
ParVector[(String, Int)] = ParVector((http://www.google.com,18976),
(http://www.bbc.co.uk,132893))
```

By making good use of Scala's built-in Try classes and parallel collections, we have built a fault tolerant, multithreaded URL retriever in a few lines of code. (Compare this to the myriad of Java/C++ books that prefix code examples with 'error handling is left out for clarity'.)

The Try type versus try/catch statements

Programmers with imperative or object-oriented backgrounds will be more familiar with try/catch blocks for handling exceptions. We could have accomplished similar functionality here by wrapping the code for fetching URLs in a try block, returning null if the call raises an exception.

However, besides being more verbose, returning null is less satisfactory: we lose all information about the exception and null is less expressive than Failure(exception). Furthermore, returning a Try[T] type forces the caller to consider the possibility that the function might fail, by encoding this possibility in the type of the return value. In contrast, just returning T and coding failure with a null value allows the caller to ignore failure, raising the possibility of a confusing NullPointerException being thrown at a completely different point in the program.

In short, Try[T] is just another higher-order type, like Option[T] or List[T]. Treating the possibility of failure in the same way as the rest of the code adds coherence to the program and encourages programmers to tackle the possibility of exceptions explicitly.



Setting the parallelism level

So far, we have considered parallel collections as black boxes: add `par` to a normal collection and all the operations are performed in parallel. Often, we will want more control over how the tasks are executed.

Internally, parallel collections work by distributing an operation over multiple threads. Since the threads share memory, parallel collections do not need to copy any data. Changing the number of threads available to the parallel collection will change the number of CPUs that are used to perform the tasks.

Parallel collections have a `tasksupport` attribute that controls task execution:

```
scala> val parRange = (0 to 100).par
parRange: ParRange = ParRange(0, 1, 2, 3, 4, 5, ...
 
scala> parRange.tasksupport
TaskSupport = scala.collection.parallel.ExecutionContextTaskSupport@311a0
b3e

scala> parRange.tasksupport.parallelismLevel
Int = 8 // Number of threads to be used
```

The task support object of a collection is an *execution context*, an abstraction capable of executing Scala expressions in a separate thread. By default, the execution context in Scala 2.11 is a *work-stealing thread pool*. When a parallel collection submits tasks, the context allocates these tasks to its threads. If a thread finds that it has finished its queued tasks, it will try and steal outstanding tasks from the other threads. The default execution context maintains a thread pool with number of threads equal to the number of CPUs.

The number of threads over which the parallel collection distributes the work can be changed by changing the task support. For instance, to parallelize the operations performed by a range over four threads:

```
scala> import scala.collection.parallel._  
import scala.collection.parallel._  
  
scala> parRange.tasksupport = new ForkJoinTaskSupport(  
    new scala.concurrent.forkjoin.ForkJoinPool(4)  
)  
parRange.tasksupport: scala.collection.parallel.TaskSupport = scala.  
collection.parallel.ForkJoinTaskSupport@6e1134e1  
  
scala> parRange.tasksupport.parallelismLevel  
Int: 4
```

An example – cross-validation with parallel collections

Let's apply what you have learned so far to solve data science problems. There are many parts of a machine learning pipeline that can be parallelized trivially. One such part is cross-validation.

We will give a brief description of cross-validation here, but you can refer to *The Elements of Statistical Learning*, by *Hastie, Tibshirani, and Friedman* for a more in-depth discussion.

Typically, a supervised machine learning problem involves training an algorithm over a training set. For instance, when we built a model to calculate the probability of a person being male based on their height and weight, the training set was the (height, weight) data for each participant, together with the male/female label for each row. Once the algorithm is trained on the training set, we can use it to classify new data. This process only really makes sense if the training set is representative of the new data that we are likely to encounter.

The training set has a finite number of entries. It will thus, inevitably, have idiosyncrasies that are not representative of the population at large, merely due to its finite nature. These idiosyncrasies will result in prediction errors when predicting whether a new person is male or female, over and above the prediction error of the algorithm on the training set itself. Cross-validation is a tool for estimating the error caused by the idiosyncrasies of the training set that do not reflect the population at large.

Cross-validation works by dividing the training set in two parts: a smaller, new training set and a cross-validation set. The algorithm is trained on the reduced training set. We then see how well the algorithm models the cross-validation set. Since we know the right answer for the cross-validation set, we can measure how well our algorithm is performing when shown new information. We repeat this procedure many times with different cross-validation sets.

There are several different types of cross-validation, which differ in how we choose the cross-validation set. In this chapter, we will look at repeated random subsampling: we select k rows at random from the training data to form the cross-validation set. We do this many times, calculating the cross-validation error for each subsample. Since each iteration is independent of the previous ones, we can parallelize this process trivially. It is therefore a good candidate for parallel collections. We will look at an alternative form of cross-validation, *k-fold cross-validation*, in *Chapter 12, Distributed Machine Learning with MLlib*.

We will build a class that performs cross-validation in parallel. I encourage you to write the code as you go, but you will find the source code corresponding to these examples on GitHub (<https://github.com/pbugnion/s4ds>). We will use parallel collections to handle the parallelism and Breeze data types in the inner loop. The `build.sbt` file is identical to the one we used in *Chapter 2, Manipulating Data with Breeze*:

```
scalaVersion := "2.11.7"

libraryDependencies ++= Seq(
  "org.scalanlp" %% "breeze" % "0.11.2",
  "org.scalanlp" %% "breeze-natives" % "0.11.2"
)
```

We will build a `RandomSubsample` class. The class exposes a type alias, `CVFunction`, for a function that takes two lists of indices—the first corresponding to the reduced training set and the second to the validation set—and returns a `Double` corresponding to the cross-validation error:

```
type CVFunction = (Seq[Int], Seq[Int]) => Double
```

The `RandomSubsample` class will expose a single method, `mapSamples`, which takes a `CVFunction`, repeatedly passes it different partitions of indices, and returns a vector of the errors. This is what the class looks like:

```
// RandomSubsample.scala

import breeze.linalg._
import breeze.numerics._

/** Random subsample cross-validation
 *
 * @param nElems Total number of elements in the training set.
 * @param nCrossValidation Number of elements to leave out of
 * training set.
 */
class RandomSubsample(val nElems:Int, val nCrossValidation:Int) {

    type CVFunction = (Seq[Int], Seq[Int]) => Double

    require(nElems > nCrossValidation,
           "nCrossValidation, the number of elements " +
           "withheld, must be < nElems")

    private val indexList = DenseVector.range(0, nElems)

    /** Perform multiple random sub-sample CV runs on f
     *
     * @param nShuffles Number of random sub-sample runs.
     * @param f user-defined function mapping from a list of
     * indices in the training set and a list of indices in the
     * test-set to a double indicating the out-of sample score
     * for this split.
     * @returns DenseVector of the CV error for each random split.
     */
    def mapSamples(nShuffles:Int)(f:CVFunction)
    :DenseVector[Double] = {
        val cvResults = (0 to nShuffles).par.map { i =>

            // Randomly split indices between test and training
            val shuffledIndices = breeze.linalg.shuffle(indexList)
            val Seq(testIndices, trainingIndices) =
                split(shuffledIndices, Seq(nCrossValidation))

            // Apply f for this split
        }
    }
}
```

```
f(trainingIndices.toScalaVector,
  testIndices.toScalaVector)
}
DenseVector(cvResults.toArray)
}
}
```

Let's look at what happens in more detail, starting with the arguments passed to the constructor:

```
class RandomSubsample(val nElems:Int, val nCrossValidation:Int)
```

We pass the total number of elements in the training set and the number of elements to leave out for cross-validation in the class constructor. Thus, passing 100 to `nElems` and 20 to `nCrossValidation` implies that our training set will have 80 random elements of the total data and that the test set will have 20 elements.

We then construct a list of all integers between 0 and `nElems`:

```
private val indexList = DenseVector.range(0, nElems)
```

For each iteration of the cross-validation, we will shuffle this list and take the first `nCrossValidation` elements to be the indices of rows in our test set and the remaining to be the indices of rows in our training set.

Our class exposes a single method, `mapSamples`, that takes two curried arguments: `nShuffles`, the number of times to perform random subsampling, and `f`, a `CVFunction`:

```
def mapSamples(nShuffles:Int)(f:CVFunction):DenseVector[Double]
```

With all this set up, the code for doing cross-validation is deceptively simple. We generate a parallel range from 0 to `nShuffles` and, for each item in the range, generate a new train-test split and calculate the cross-validation error:

```
val cvResults = (0 to nShuffles).par.map { i =>
  val shuffledIndices = breeze.linalg.shuffle(indexList)
  val Seq(testIndices, trainingIndices) =
    split(shuffledIndices, Seq(nCrossValidation))
  f(trainingIndices.toScalaVector, testIndices.toScalaVector)
}
```

The only tricky part of this function is splitting the shuffled index list into a list of indices for the training set and a list of indices for the test set. We use Breeze's `split` method. This takes a vector as its first argument and a list of split-points as its second, and returns a list of fragments of the original vector. We then use pattern matching to extract the individual parts.

Finally, `mapSamples` converts `cvResults` to a Breeze vector:

```
DenseVector(cvResults.toArray)
```

Let's see this in action. We can test our class by running cross-validation on the logistic regression example developed in *Chapter 2, Manipulating Data with Breeze*. In that chapter, we developed a `LogisticRegression` class that takes a training set (in the form of a `DenseMatrix`) and target (in the form of a `DenseVector`) at construction time. The class then calculates the parameters that best represent the training set. We will first add two methods to the `LogisticRegression` class to use the trained model to classify previously unseen examples:

- The `predictProbabilitiesMany` method uses the trained model to calculate the probability of having the target variable set to one. In the context of our example, this is the probability of being male, given a height and weight.
- The `classifyMany` method assigns classification labels (one or zero) to members of a test set. We will assign a one if `predictProbabilitiesMany` returns a value greater than 0.5.

With these two functions, our `LogisticRegression` class becomes:

```
// Logistic Regression.scala

class LogisticRegression(
    val training:DenseMatrix[Double],
    val target:DenseVector[Double]
) {
    ...
    /** Probability of classification for each row
     * in test set.
     */
    def predictProbabilitiesMany(test:DenseMatrix[Double])
    :DenseVector[Double] = {
        val xBeta = test * optimalCoefficients
        sigmoid(xBeta)
    }

    /** Predict the value of the target variable
     * for each row in test set.
     */
    def classifyMany(test:DenseMatrix[Double])
    :DenseVector[Double] = {
        val probabilities = predictProbabilitiesMany(test)
        I((probabilities :> 0.5).toDenseVector)
    }
    ...
}
```

We can now put together an example program for our `RandomSubsample` class. We will use the same height-weight data as in *Chapter 2, Manipulating Data with Breeze*. The data preprocessing will be similar. The code examples for this chapter provide a helper module, `HWData`, to load the height-weight data into Breeze vectors. The data itself is in the `data/` directory of the code examples for this chapter (available on GitHub at <https://github.com/pbugnion/s4ds/tree/master/chap04>).

For each new subsample, we create a new `LogisticRegression` instance, train it on the subset of the training set to get the best coefficients for this train-test split, and use `classifyMany` to generate predictions on the cross-validation set in this split. We then calculate the classification error and report the average classification error over every train-test split:

```
// RandomSubsampleDemo.scala

import breeze.linalg._
import breeze.linalg.functions.manhattanDistance
import breeze.numerics._
import breeze.stats._

object RandomSubsampleDemo extends App {

  /* Load and pre-process data */
  val data = HWData.load

  val rescaledHeights:DenseVector[Double] =
    (data.heights - mean(data.heights)) / stddev(data.heights)

  val rescaledWeights:DenseVector[Double] =
    (data.weights - mean(data.weights)) / stddev(data.weights)

  val featureMatrix:DenseMatrix[Double] =
    DenseMatrix.horzcat(
      DenseMatrix.ones[Double](data.npoints, 1),
      rescaledHeights.toDenseMatrix.t,
      rescaledWeights.toDenseMatrix.t
    )

  val target:DenseVector[Double] = data.genders.values.map {
    gender => if(gender == 'M') 1.0 else 0.0
  }

  /* Cross-validation */
  val testSize = 20
```

```
val cvCalculator = new RandomSubsample(data.npoints, testSize)

// Start parallel CV loop
val cvErrors = cvCalculator.mapSamples(1000) {
  (trainingIndices, testIndices) =>

  val regressor = new LogisticRegression(
    data.featureMatrix(trainingIndices, ::).toDenseMatrix,
    data.target(trainingIndices).toDenseVector
  )
  // Predictions on test-set
  val genderPredictions = regressor.classifyMany(
    data.featureMatrix(testIndices, ::).toDenseMatrix
  )
  // Calculate number of mis-classified examples
  val dist = manhattanDistance(
    genderPredictions, data.target(testIndices)
  )
  // Calculate mis-classification rate
  dist / testSize.toDouble
}

println(s"Mean classification error: ${mean(cvErrors)}")
}
```

Running this program on the height-weight data gives a classification error of 10%.

We now have a fully working, parallelized cross-validation class. Scala's parallel range made it simple to repeatedly compute the same function in different threads.

Futures

Parallel collections offer a simple, yet powerful, framework for parallel operations. However, they are limited in one respect: the total amount of work must be known in advance, and each thread must perform the same function (possibly on different inputs).

Imagine that we want to write a program that fetches a web page (or queries a web API) every few seconds and extracts data for further processing from this web page. A typical example might involve querying a web API to maintain an up-to-date value of a particular stock price. Fetching data from an external web page takes a few hundred milliseconds, typically. If we perform this operation on the main thread, it will needlessly waste CPU cycles waiting for the web server to reply.

The solution is to wrap the code for fetching the web page in a *future*. A future is a one-element container containing the future result of a computation. When you create a future, the computation in it gets off-loaded to a different thread in order to avoid blocking the main thread. When the computation finishes, the result is written to the future and thus made accessible to the main thread.

As an example, we will write a program that queries the "Markit on demand" API to fetch the price of a given stock. For instance, the URL for the current price of a Google share is `http://dev.markitondemand.com/MODApis/Api/v2/Quote?symbol=GOOG`. Go ahead and paste this in the address box of your web browser. You will see an XML string appear with, among other things, the current stock price. Let's fetch this programmatically without resorting to a future first:

```
scala> import scala.io._  
import scala.io._  
  
scala> val url = "http://dev.markitondemand.com/MODApis/Api/v2/  
Quote?symbol=GOOG"  
url: String = http://dev.markitondemand.com/MODApis/Api/v2/  
Quote?symbol=GOOG  
  
scala> val response = Source.fromURL(url).mkString  
response: String = <StockQuote><Status>SUCCESS</Status>  
...
```

Notice how it takes a little bit of time to query the API. Let's now do the same, but using a future (don't worry about the imports for now, we will discuss what they mean in more detail further on):

```
scala> import scala.concurrent._  
import scala.concurrent._  
  
scala> import scala.util._  
import scala.util._  
  
scala> import scala.concurrent.ExecutionContext.Implicits.global  
import scala.concurrent.ExecutionContext.Implicits.global  
  
scala> val response = Future { Source.fromURL(url).mkString }  
response: Future[String] = Promise$DefaultPromise@3301801b
```

If you run this, you will notice that control returns to the shell instantly before the API has had a chance to respond. To make this evident, let's simulate a slow connection by adding a call to `Thread.sleep`:

```
scala> val response = Future {
    Thread.sleep(10000) // sleep for 10s
    Source.fromURL(url).mkString
}
response: Future[String] = Promise$DefaultPromise@231f98ef
```

When you run this, you do not have to wait for ten seconds for the next prompt to appear: you regain control of the shell straightaway. The bit of code in the future is executed asynchronously: its execution is independent of the main program flow.

How do we retrieve the result of the computation? We note that `response` has type `Future[String]`. We can check whether the computation wrapped in the future has finished by querying the future's `isCompleted` attribute:

```
scala> response.isCompleted
Boolean = true
```

The future exposes a `value` attribute that contains the computation result:

```
scala> response.value
Option[Try[String]] = Some(Success(<StockQuote><Status>SUCCESS</Status>
...

```

The `value` attribute of a future has type `Option[Try[T]]`. We have already seen how to use the `Try` type to handle exceptions gracefully in the context of parallel collections. It is used in the same way here. A future's `value` attribute is `None` until the future is complete, then it is set to `Some(Success(value))` if the future ran successfully, or `Some(Failure(error))` if an exception was thrown.

Repeatedly calling `f.value` until the future completes works well in the shell, but it does not generalize to more complex programs. Instead, we want to tell the computer to do something once the future is complete: we want to bind a *callback* function to the future. We can do this by setting the future's `onComplete` attribute. Let's tell the future to print the API response when it completes:

```
scala> response.onComplete {
    case Success(s) => println(s)
```

```
case Failure(e) => println(s"Error fetching page: $e")
}

scala>
// Wait for response to complete, then prints:
<StockQuote><Status>SUCCESS</Status><Name>Alphabet Inc</
Name><Symbol>GOOGL</Symbol><LastPrice>695.22</LastPrice><Chan...
```

The function passed to `onComplete` runs when the future is finished. It takes a single argument of type `Try[T]` containing the result of the future.

Failure is normal: how to build resilient applications

By wrapping the output of the code that it runs in a `Try` type, futures force the client code to consider the possibility that the code might fail. The client can isolate the effect of failure to avoid crashing the whole application. They might, for instance, log the exception. In the case of a web API query, they might add the offending URL to be queried again at a later date. In the case of a database failure, they might roll back the transaction.

By treating failure as a first-class citizen rather than through exceptional control flow bolted on at the end, we can build applications that are much more resilient.



Future composition – using a future's result

In the previous section, you learned about the `onComplete` method to bind a callback to a future. This is useful to cause a side effect to happen when the future is complete. It does not, however, let us transform the future's return value easily.

To carry on with our stocks example, let's imagine that we want to convert the query response from a string to an XML object. Let's start by including the `scala-xml` library as a dependency in `build.sbt`:

```
libraryDependencies += "org.scala-lang" % "scala-xml" % "2.11.0-M4"
```

Let's restart the console and reimport the dependencies on `scala.concurrent._`, `scala.concurrent.ExecutionContext.Implicits.global`, and `scala.io._`. We also want to import the `XML` library:

```
scala> import scala.xml.XML
import scala.xml.XML
```

We will use the same URL as in the previous section:

```
http://dev.markitondemand.com/MODApis/Api/v2/Quote?symbol=GOOG
```

It is sometimes useful to think of a future as a collection that either contains one element if a calculation has been successful, or zero elements if it has failed. For instance, if the web API has been queried successfully, our future contains a string representation of the response. Like other container types in Scala, futures support a map method that applies a function to the element contained in the future, returning a new future, and does nothing if the calculation in the future failed. But what does this mean in the context of a computation that might not be finished yet? The map method gets applied as soon as the future is complete, like the onComplete method.

We can use the future's map method to apply a transformation to the result of the future asynchronously. Let's poll the "Markit on demand" API again. This time, instead of printing the result, we will parse it as XML.

```
scala> val strResponse = Future {
    Thread.sleep(20000) // Sleep for 20s
    val res = Source.fromURL(url).mkString
    println("finished fetching url")
    res
}
strResponse: Future[String] = Promise$DefaultPromise@1ddda9bc8

scala> val xmlResponse = strResponse.map { s =>
    println("applying string to xml transformation")
    XML.loadString(s)
}
xmlResponse: Future[xml.Elem] = Promise$DefaultPromise@25d1262a

// wait while the remainder of the 20s elapses
finished fetching url
applying string to xml transformation

scala> xmlResponse.value
Option[Try[xml.Elem]] = Some(Success(<StockQuote><Status>SUCCESS</
Status>...)
```

By registering subsequent maps on futures, we are providing a road map to the executor running the future for what to do.

If any of the steps fail, the failed `Try` instance containing the exception gets propagated instead:

```
scala> val strResponse = Future {
    Source.fromURL("empty").mkString
}

scala> val xmlResponse = strResponse.map {
    s => XML.loadString(s)
}

scala> xmlResponse.value
Option[Try[xml.Elem]] = Some(Failure(MalformedURLException: no protocol: empty))
```

This behavior makes sense if you think of a failed future as an empty container. When applying a map to an empty list, it returns the same empty list. Similarly, when applying a map to an empty (failed) future, the empty future is returned.

Blocking until completion

The code for fetching stock prices works fine in the shell. However, if you paste it in a standalone program, you will notice that nothing gets printed and the program finishes straightforwardly. Let's look at a trivial example of this:

```
// BlockDemo.scala
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._

object BlockDemo extends App {
    val f = Future { Thread.sleep(10000) }
    f.onComplete { _ => println("future completed") }
    // "future completed" is not printed
}
```

The program stops running as soon as the main thread has completed its tasks, which, in this example, just involves creating the futures. In particular, the line "future completed" is never printed. If we want the main thread to wait for a future to execute, we must explicitly tell it to block execution until the future has finished running. This is done using the `Await.ready` or `Await.result` methods. Both these methods block the execution of the main thread until the future completes. We could make the above program work as intended by adding this line:

```
Await.ready(f, 1 minute)
```

The `Await` methods take the future as their first argument and a `Duration` object as the second. If the future takes longer to complete than the specified duration, a `TimeoutException` is thrown. Pass `Duration.Inf` to set an infinite timeout.

The difference between `Await.ready` and `Await.result` is that the latter returns the value inside the future. In particular, if the future resulted in an exception, that exception will get thrown. In contrast, `Await.ready` returns the future itself.

In general, one should try to avoid blocking as much as possible: the whole point of futures is to run code in background threads in order to keep the main thread of execution responsive. However, a common, legitimate use case for blocking is at the end of a program. If we are running a large-scale integration process, we might dispatch several futures to query web APIs, read from text files, or insert data into a database. Embedding the code in futures is more scalable than performing these operations sequentially. However, as the majority of the intensive work is running in background threads, we are left with many outstanding futures when the main thread completes. It makes sense, at this stage, to block until all the futures have completed.

Controlling parallel execution with execution contexts

Now that we know how to define futures, let's look at controlling how they run. In particular, you might want to control the number of threads to use when running a large number of futures.

When a future is defined, it is passed an *execution context*, either directly or implicitly. An execution context is an object that exposes an `execute` method that takes a block of code and runs it, possibly asynchronously. By changing the execution context, we can change the "backend" that runs the futures. We have already seen how to use execution contexts to control the execution of parallel collections.

So far, we have just been using the default execution context by importing `scala.concurrent.ExecutionContext.Implicits.global`. This is a fork / join thread pool with as many threads as there are underlying CPUs.

Let's now define a new execution context that uses sixteen threads:

```
scala> import java.util.concurrent.Executors
import java.util.concurrent.Executors

scala> val ec = ExecutionContext.fromExecutorService(
  Executors.newFixedThreadPool(16)
)
ec: ExecutionContextExecutorService = ExecutionContextImpl$$anon$1@1351
ce60
```

Having defined the execution context, we can pass it explicitly to futures as they are defined:

```
scala> val f = Future { Thread.sleep(1000) } (ec)
f: Future[Unit] = Promise$DefaultPromise@458b456
```

Alternatively, we can define the execution context implicitly:

```
scala> implicit val context = ec
context: ExecutionContextExecutorService = ExecutionContextImpl$$anon$1@1
351ce60
```

It is then passed as an implicit parameter to all new futures as they are constructed:

```
scala> val f = Future { Thread.sleep(1000) }
f: Future[Unit] = Promise$DefaultPromise@3c4b7755
```

You can shut the execution context down to destroy the thread pool:

```
scala> ec.shutdown()
```

When an execution context receives a shutdown command, it will finish executing its current tasks but will refuse any new tasks.

Futures example – stock price fetcher

Let's bring some of the concepts that we covered in this section together to build a command-line application that prompts the user for the name of a stock and fetches the value of that stock. The catch is that, to keep the UI responsive, we will fetch the stock using a future:

```
// StockPriceDemo.scala

import scala.concurrent._
```

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.io._
import scala.xml.XML
import scala.util._

object StockPriceDemo extends App {

    /* Construct URL for a stock symbol */
    def urlFor(stockSymbol:String) =
        ("http://dev.markitondemand.com/MODApis/Api/v2/Quote?" +
         s"symbol=${stockSymbol}")

    /* Build a future that fetches the stock price */
    def fetchStockPrice(stockSymbol:String):Future[BigDecimal] = {
        val url = urlFor(stockSymbol)
        val strResponse = Future { Source.fromURL(url).mkString }
        val xmlResponse = strResponse.map { s => XML.loadString(s) }
        val price = xmlResponse.map {
            r => BigDecimal((r \ "LastPrice").text)
        }
        price
    }

    /* Command line interface */
    println("Enter symbol at prompt.")
    while (true) {
        val symbol = readLine("> ") // Wait for user input
        // When user puts in symbol, fetch data in background
        // thread and print to screen when complete
        fetchStockPrice(symbol).onComplete { res =>
            println()
            res match {
                case Success(price) => println(s"$symbol: USD $price")
                case Failure(e) => println(s"Error fetching $symbol: $e")
            }
            print("> ") // Simulate the appearance of a new prompt
        }
    }
}
```

Try running the program and entering the code for some stocks:

```
[info] Running StockPriceDemo
Enter symbol at prompt:
> GOOG
> MSFT
>
GOOG: USD 695.22
>
MSFT: USD 47.48
> AAPL
>
AAPL: USD 111.01
```

Let's summarize how the code works. When you enter a stock, the main thread constructs a future that fetches the stock information from the API, converts it to XML, and extracts the price. We use `(r \ "LastPrice").text` to extract the text inside the `LastPrice` tag from the XML node `r`. We then convert the value to a big decimal. When the transformations are complete, the result is printed to screen by binding a callback through `onComplete`. Exception handling is handled naturally through our use of `.map` methods to handle transformations.

By wrapping the code for fetching a stock price in a future, we free up the main thread to just respond to the user. This means that the user interface does not get blocked if we have, for instance, a slow internet connection.

This example is somewhat artificial, but you could easily wrap much more complicated logic: stock prices could be written to a database and we could add additional commands to plot the stock price over time, for instance.

We have only scratched the surface of what futures can offer in this section. We will revisit futures in more detail when we look at polling web APIs in *Chapter 7, Web APIs* and *Chapter 9, Concurrency with Akka*.

Futures are a key part of the data scientist's toolkit for building scalable systems. Moving expensive computation (either in terms of CPU time or wall time) to background threads improves scalability greatly. For this reason, futures are an important part of many Scala libraries such as **Akka** and the **Play** framework.

Summary

By providing high-level concurrency abstractions, Scala makes writing parallel code intuitive and straightforward. Parallel collections and futures form an invaluable part of a data scientist's toolbox, allowing them to parallelize their code with minimal effort. However, while these high-level abstractions obviate the need to deal directly with threads, an understanding of the internals of Scala's concurrency model is necessary to avoid race conditions.

In the next chapter, we will put concurrency on hold and study how to interact with SQL databases. However, this is only temporary: futures will play an important role in many of the remaining chapters in this book.

References

Aleksandar Prokopec, Learning Concurrent Programming in Scala. This is a detailed introduction to the basics of concurrent programming in Scala. In particular, it explores parallel collections and futures in much greater detail than this chapter.

Daniel Westheide's blog gives an excellent introduction to many Scala concepts, in particular:

- **Futures:** <http://danielwestheide.com/blog/2013/01/09/the-neophytes-guide-to-scala-part-8-welcome-to-the-future.html>
- **The Try type:** <http://danielwestheide.com/blog/2012/12/26/the-neophytes-guide-to-scala-part-6-error-handling-with-try.html>

For a discussion of cross-validation, see *The Elements of Statistical Learning* by *Hastie, Tibshirani, and Friedman*.

5

Scala and SQL through JDBC

One of data science's raison d'être is the difficulty of manipulating large datasets. Much of the data of interest to a company or research group cannot fit conveniently in a single computer's RAM. Storing the data in a way that is easy to query is therefore a complex problem.

Relational databases have been successful at solving the data storage problem. Originally proposed in 1970 (<http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf>), the overwhelming majority of databases in active use today are still relational. In that time, the price of RAM per megabyte has decreased by a factor of a hundred million. Similarly, hard drive capacity has increased from tens or hundreds of megabytes to terabytes. It is remarkable that, despite this exponential growth in data storage capacity, the relational model has remained dominant.

Virtually all relational databases are described and queried with variants of **SQL** (**Structured Query Language**). With the advent of distributed computing, the position of SQL databases as the de facto data storage standard is being challenged by other types of databases, commonly grouped under the umbrella term NoSQL. Many NoSQL databases are more partition-tolerant than SQL databases: they can be split into several parts residing on different computers. While this author expects that NoSQL databases will become increasingly popular, SQL databases are likely to remain prevalent as a data persistence mechanism; hence, a significant portion of this book is devoted to interacting with SQL from Scala.

While SQL is standardized, most implementations do not follow the full standard. Additionally, most implementations provide extensions to the standard. This means that, while many of the concepts in this book will apply to all SQL backends, the exact syntax will need to be adjusted. We will consider only the MySQL implementation here.

In this chapter, you will learn how to interact with SQL databases from Scala using JDBC, a bare bones Java API. In the next chapter, we will consider Slick, an **Object Relational Mapper (ORM)** that gives a more Scala-esque feel to interacting with SQL.

This chapter is roughly composed of two sections: we will first discuss the basic functionality for connecting and interacting with SQL databases, and then discuss useful functional patterns that can be used to create an elegant, loosely coupled, and coherent data access layer.

This chapter assumes that you have a basic working knowledge of SQL. If you do not, you would be better off first reading one of the reference books mentioned at the end of the chapter.

Interacting with JDBC

JDBC is an API for connecting to SQL databases in Java. It remains the simplest way of connecting to SQL databases from Scala. Furthermore, the majority of higher-level abstractions for interacting with databases still use JDBC as a backend.

JDBC is not a library in itself. Rather, it exposes a set of interfaces to interact with databases. Relational database vendors then provide specific implementations of these interfaces.

Let's start by creating a `build.sbt` file. We will declare a dependency on the MySQL JDBC connector:

```
scalaVersion := "2.11.7"

libraryDependencies += "mysql" % "mysql-connector-java" % "5.1.36"
```

First steps with JDBC

Let's start by connecting to JDBC from the command line. To follow with the examples, you will need access to a running MySQL server. If you added the MySQL connector to the list of dependencies, open a Scala console by typing the following command:

```
$ sbt console
```

Let's import JDBC:

```
scala> import java.sql._  
import java.sql._
```

We then need to tell JDBC to use a specific connector. This is normally done using reflection, loading the driver at runtime:

```
scala> Class.forName("com.mysql.jdbc.Driver")  
Class[_] = class com.mysql.jdbc.Driver
```

This loads the appropriate driver into the namespace at runtime. If this seems somewhat magical to you, it's probably not worth worrying about exactly how this works. This is the only example of reflection that we will consider in this book, and it is not particularly idiomatic Scala.

Connecting to a database server

Having specified the SQL connector, we can now connect to a database. Let's assume that we have a database called `test` on host `127.0.0.1`, listening on port `3306`. We create a connection as follows:

```
scala> val connection = DriverManager.getConnection(  
        "jdbc:mysql://127.0.0.1:3306/test",  
        "root", // username when connecting  
        "" // password  
)  
java.sql.Connection = com.mysql.jdbc.JDBC4Connection@12e78a69
```

The first argument to `getConnection` is a URL-like string with `jdbc:mysql://host[:port]/database`. The second and third arguments are the username and password. Pass in an empty string if you can connect without a password.

Creating tables

Now that we have a database connection, let's interact with the server. For these examples, you will find it useful to have a MySQL shell open (or a MySQL GUI such as **MySQLWorkbench**) as well as the Scala console. You can open a MySQL shell by typing the following command in a terminal:

```
$ mysql
```

As an example, we will create a small table to keep track of famous physicists. In a mysql shell, we would run the following command:

```
mysql> USE test;
mysql> CREATE TABLE physicists (
    id INT(11) AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(32) NOT NULL
);
```

To achieve the same with Scala, we send a JDBC statement to the connection:

```
scala> val statementString = """
CREATE TABLE physicists (
    id INT(11) AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(32) NOT NULL
)
"""

scala> val statement = connection.prepareStatement(statementString)
PreparedStatement = JDBC4PreparedStatement@c983201: CREATE TABLE ... 

scala> statement.executeUpdate()
results: Int = 0
```

Let's ignore the return value of `executeUpdate` for now.

Inserting data

Now that we have created a table, let's insert some data into it. We can do this with a SQL `INSERT` statement:

```
scala> val statement = connection.prepareStatement("""
    INSERT INTO physicists (name) VALUES ('Isaac Newton')
""")

scala> statement.executeUpdate()
Int = 1
```

In this case, `executeUpdate` returns 1. When inserting rows, it returns the number of rows that were inserted. Similarly, if we had used a `SQL UPDATE` statement, this would return the number of rows that were updated. For statements that do not manipulate rows directly (such as the `CREATE TABLE` statement in the previous section), `executeUpdate` just returns 0.

Let's just jump into a `mysql` shell to verify the insertion performed correctly:

```
mysql> select * from physicists ;  
+----+-----+  
| id | name |  
+----+-----+  
| 1 | Isaac Newton |  
+----+-----+  
1 row in set (0.00 sec)
```

Let's quickly summarize what we have seen so far: to execute SQL statements that do not return results, use the following:

```
val statement = connection.prepareStatement("SQL statement string")  
statement.executeUpdate()
```

In the context of data science, we frequently need to insert or update many rows at a time. For instance, we might have a list of physicists:

```
scala> val physicistNames = List("Marie Curie", "Albert Einstein", "Paul  
Dirac")
```

We want to insert all of these into the database. While we could create a statement for each physicist and send it to the database, this is quite inefficient. A better solution is to create a *batch* of statements and send them to the database together. We start by creating a statement template:

```
scala> val statement = connection.prepareStatement("""  
INSERT INTO physicists (name) VALUES (?)  
""")  
PreparedStatement = JDBC4PreparedStatement@621a8225: INSERT INTO  
physicists (name) VALUES (** NOT SPECIFIED **)
```

This is identical to the previous `prepareStatement` calls, except that we replaced the physicist's name with a ? placeholder. We can set the placeholder value with the `statement.setString` method:

```
scala> statement.setString(1, "Richard Feynman")
```

This replaces the first placeholder in the statement with the string Richard Feynman:

```
scala> statement
com.mysql.jdbc.JDBC4PreparedStatement@5fdd16c3:
INSERT INTO physicists (name) VALUES ('Richard Feynman')
```

Note that JDBC, somewhat counter-intuitively, counts the placeholder positions from 1 rather than 0.

We have now created the first statement in the batch of updates. Run the following command:

```
scala> statement.addBatch()
```

By running the preceding command, we initiate a batch insert: the statement is added to a temporary buffer that will be executed when we run the `executeBatch` method. Let's add all the physicists in our list:

```
scala> physicistNames.foreach { name =>
    statement.setString(1, name)
    statement.addBatch()
}
```

We can now execute all the statements in the batch:

```
scala> statement.executeBatch
Array[Int] = Array(1, 1, 1, 1)
```

The return value of `executeBatch` is an array of the number of rows altered or inserted by each item in the batch.

Note that we used `statement.setString` to fill in the template with a particular name. The `PreparedStatement` object has `setXXX` methods for all basic types. To get a complete list, read the `PreparedStatement` API documentation (<http://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html>).

Reading data

Now that we know how to insert data into a database, let's look at the converse: reading data. We use SQL `SELECT` statements to query the database. Let's do this in the MySQL shell first:

```
mysql> SELECT * FROM physicists;
+----+-----+
| id | name |
+----+-----+
```

```
+----+-----+
| 1 | Isaac Newton |
| 2 | Richard Feynman |
| 3 | Marie Curie |
| 4 | Albert Einstein |
| 5 | Paul Dirac |
+----+-----+
5 rows in set (0.01 sec)
```

To extract this information in Scala, we define a `PreparedStatement`:

```
scala> val statement = connection.prepareStatement("""
    SELECT name FROM physicists
""")
PreparedStatement = JDBC4PreparedStatement@3c577c9d:
SELECT name FROM physicists
```

We execute this statement by running the following command:

```
scala> val results = statement.executeQuery()
results: java.sql.ResultSet = com.mysql.jdbc.JDBC4ResultSet@74a2e158
```

This returns a JDBC `ResultSet` instance. The `ResultSet` is an abstraction representing a set of rows from the database. Note that we used `statement.executeQuery` rather than `statement.executeUpdate`. In general, one should execute statements that return data (in the form of `ResultSet`) with `executeQuery`. Statements that modify the database without returning data (`insert`, `create`, `alter`, or `update` statements, among others) are executed with `executeUpdate`.

The `ResultSet` object behaves somewhat like an iterator. It exposes a `next` method that advances itself to the next record, returning `true` if there are records left in `ResultSet`:

```
scala> results.next // Advance to the first record
Boolean = true
```

When the `ResultSet` instance points to a record, we can extract fields in this record by passing in the field name:

```
scala> results.getString("name")
String = Isaac Newton
```

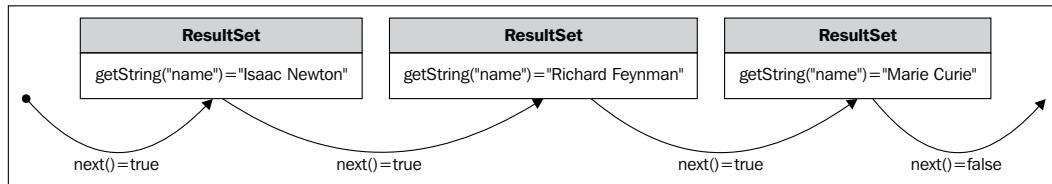
We can also extract fields using positional arguments. The fields are indexed from one:

```
scala> results.getString(1) // first positional argument
String = Isaac Newton
```

When we are done with a particular record, we call the `next` method to advance the `ResultSet` to the next record:

```
scala> results.next // advances the ResultSet by one record
Boolean = true

scala> results.getString("name")
String = Richard Feynman
```



A `ResultSet` object supports the `getXXX(fieldName)` methods to access the fields of a record and a `next` method to advance to the next record in the result set.

One can iterate over a result set using a while loop:

```
scala> while(results.next) { println(results.getString("name")) }
Marie Curie
Albert Einstein
Paul Dirac
```

A word of warning applies to reading fields that are nullable. While one might expect JDBC to return null when faced with a null SQL field, the return type depends on the `getXXX` command used. For instance, `getInt` and `getLong` will return 0 for any field that is null. Similarly, `getDouble` and `getFloat` return 0.0. This can lead to some subtle bugs in code. In general, one should be careful with getters that return Java value types (`int`, `long`) rather than objects. To find out if a value is null in the database, query it first with `getInt` (or `getLong` or `getDouble`, as appropriate), then use the `wasNull` method that returns a Boolean if the last read value was null:



```
scala> rs.getInt("field")
0
scala> rs.wasNull // was the last item read null?
true
```

This (surprising) behavior makes reading from `ResultSet` instances error-prone. One of the goals of the second part of this chapter is to give you the tools to build an abstraction layer on top of the `ResultSet` interface to avoid having to call methods such as `getInt` directly.

Reading values directly from `ResultSet` objects feels quite unnatural in Scala. We will look, further on in this chapter, at constructing a layer through which you can access the result set using type classes.

We now know how to read and write to a database. Having finished with the database for now, we close the result sets, prepared statements, and connections:

```
scala> results.close

scala> statement.close

scala> connection.close
```

While closing statements and connections is not important in the Scala shell (they will get closed when you exit), it is important when you run programs; otherwise, the objects will persist, leading to "out of memory exceptions". In the next sections, we will look at establishing connections and statements with the **loan pattern**, a design pattern that closes a resource automatically when we finish using it.

JDBC summary

We now have an overview of JDBC. The rest of this chapter will concentrate on writing abstractions that sit above JDBC, making database accesses feel more natural. Before we do this, let's summarize what we have seen so far.

We have used three JDBC classes:

- The `Connection` class represents a connection to a specific SQL database. Instantiate a connection as follows:

```
import java.sql._  
Class.forName("com.mysql.jdbc.Driver")  
val connection = DriverManager.getConnection(  
    "jdbc:mysql://127.0.0.1:3306/test",  
    "root", // username when connecting  
    "" // password  
)
```

Our main use of `Connection` instances has been to generate `PreparedStatement` objects:

```
connection.prepareStatement("SELECT * FROM physicists")
```

- A `PreparedStatement` instance represents a SQL statement about to be sent to the database. It also represents the template for a SQL statement with placeholders for values yet to be filled in. The class exposes the following methods:

<code>statement.executeUpdate</code>	This sends the statement to the database. Use this for SQL statements that modify the database and do not return any data, such as <code>INSERT</code> , <code>UPDATE</code> , <code>DELETE</code> , and <code>CREATE</code> statements.
<code>val results = statement.executeQuery</code>	This sends the statement to the database. Use this for SQL statements that return data (predominantly, the <code>SELECT</code> statements). This returns a <code>ResultSet</code> instance.
<code>statement.addBatch</code> <code>statement.executeBatch</code>	The <code>addBatch</code> method adds the current statement to a batch of statements, and <code>executeBatch</code> sends the batch of statements to the database.

<pre>statement.setString(1, "Scala") statement.setInt(1, 42) statement.setBoolean(1, true)</pre>	<p>Fill in the placeholder values in the <code>PreparedStatement</code>. The first argument is the position in the statement (counting from 1). The second argument is the value.</p> <p>One common use case for these is in a batch update or insert: we might have a Scala list of objects that we want to insert into the database. We fill in the placeholders for each object in the list using the <code>.setXXX</code> methods, then add this statement to the batch using <code>.addBatch</code>. We can then send the entire batch to the database using <code>.executeBatch</code>.</p>
<pre>statement.setNull(1, java.sql.Types.BOOLEAN)</pre>	<p>This sets a particular item in the statement to <code>NULL</code>. The second argument specifies the <code>NULL</code> type. If we are setting a cell in a Boolean column, for instance, this should be <code>Types.BOOLEAN</code>. A full list of types is given in the API documentation for the <code>java.sql.Types</code> package (http://docs.oracle.com/javase/7/docs/api/java/sql/Types.html).</p>

- A `ResultSet` instance represents a set of rows returned by a `SELECT` or `SHOW` statement. `ResultSet` exposes methods to access fields in the current row:

<code>rs.getString(i)</code> <code>rs.getInt(i)</code>	These methods get the value of the <code>i</code> th field in the current row; <code>i</code> is measured from 1.
<code>rs.getString("name")</code> <code>rs.getInt("age")</code>	These methods get the value of a specific field, which is indexed by the column name.
<code>rs.wasNull</code>	This returns whether the last column read was <code>NULL</code> . This is particularly important when reading Java value types, such as <code>getInt</code> , <code>getBoolean</code> , or <code>getDouble</code> , as these return a default value when reading a <code>NULL</code> value.

The `ResultSet` instance exposes the `.next` method to move to the next row; `.next` returns `true` until the `ResultSet` has advanced to just beyond the last row.

Functional wrappers for JDBC

We now have a basic overview of the tools afforded by JDBC. All the objects that we have interacted with so far feel somewhat clunky and out of place in Scala. They do not encourage a functional style of programming.

Of course, elegance is not necessarily a goal in itself (or, at least, you will probably struggle to convince your CEO that he should delay the launch of a product because the code lacks elegance). However, it is usually a symptom: either the code is not extensible or too tightly coupled, or it is easy to introduce bugs. The latter is particularly the case for JDBC. Forgot to check `wasNull?` That will come back to bite you. Forgot to close your connections? You'll get an "out of memory exception" (hopefully not in production).

In the next sections, we will look at patterns that we can use to wrap JDBC types in order to mitigate many of these risks. The patterns that we introduce here are used very commonly in Scala libraries and applications. Thus, besides writing robust classes to interact with JDBC, learning about these patterns will, I hope, give you greater understanding of Scala programming.

Safer JDBC connections with the loan pattern

We have already seen how to connect to a JDBC database and send statements to the database for execution. This technique, however, is somewhat error prone: you have to remember to close statements; otherwise, you will quickly run out of memory. In more traditional imperative style, we write the following try-finally block around every connection:

```
// WARNING: poor Scala code
val connection = DriverManager.getConnection(url, user, password)
try {
    // do something with connection
}
finally {
    connection.close()
}
```

Scala, with first-class functions, provides us with an alternative: the *loan pattern*. We write a function that is responsible for opening the connection, loaning it to the client code to do something interesting with it, and then closing it when the client code is done. Thus, the client code is not responsible for closing the connection any more.

Let's create a new `SqlUtils` object with a `usingConnection` method that leverages the loan pattern:

```
// SqlUtils.scala

import java.sql._

object SqlUtils {

    /** Create an auto-closing connection using
     * the loan pattern */
    def usingConnection[T](
        db:String,
        host:String="127.0.0.1",
        user:String="root",
        password:String="",
        port:Int=3306
    )(f:Connection => T) :T = {

        // Create the connection
        val Url = s"jdbc:mysql://$host:$port/$db"
        Class.forName("com.mysql.jdbc.Driver")
        val connection = DriverManager.getConnection(
            Url, user, password)

        // give the connection to the client, through the callable
        // `f` passed in as argument
        try {
            f(connection)
        }
        finally {
            // When client is done, close the connection
            connection.close()
        }
    }
}
```

Let's see this function in action:

```
scala> SqlUtils.usingConnection("test") {
  connection => println(connection)
}

com.mysql.jdbc.JDBC4Connection@46fd3d66
```

Thus, the client doesn't have to remember to close the connection, and the resultant code (for the client) feels much more like Scala.

How does our `usingConnection` function work? The function definition is `def usingConnection(...)(f : Connection => T) : T`. It takes, as its second set of arguments, a function that acts on a `Connection` object. The body of `usingConnection` creates the connection, then passes it to `f`, and finally closes the connection. This syntax is somewhat similar to code blocks in Ruby or the `with` statement in Python.



Be careful when mixing the loan pattern with lazy operations. This applies particularly to returning iterators, streams, and futures from `f`. As soon as the thread of execution leaves `f`, the connection will be closed. Any data structure that is not materialized at this point will not be able to carry on accessing the connection.

The loan pattern is, of course, not exclusive to database connections. It is useful whenever you have the following pattern, in pseudocode:

```
open resource (eg. database connection, file ...)
use resource somehow // loan resource to client for this part.
close resource
```

Enriching JDBC statements with the "pimp my library" pattern

In the previous section, we saw how to create self-closing connections with the loan pattern. This allows us to open connections to the database without having to remember to close them. However, we still have to remember to close any `ResultSet` and `PreparedStatement` that we open:

```
// WARNING: Poor Scala code
SqlUtils.usingConnection("test") { connection =>
    val statement = connection.prepareStatement(
        "SELECT * FROM physicists")
    val results = statement.executeQuery
    // do something useful with the results
    results.close
    statement.close
}
```

Having to open and close the statement is somewhat ugly and error prone. This is another natural use case for the loan pattern. Ideally, we would like to write the following:

```
usingConnection("test") { connection =>
    connection.withQuery("SELECT * FROM physicists") {
        resultSet => // process results
    }
}
```

How can we define a `.withQuery` method on the `Connection` class? We do not control the `Connection` class definition as it is part of the JDBC API. We would like to be able to somehow reopen the `Connection` class definition to add the `withQuery` method.

Scala does not let us reopen classes to add new methods (a practice known as monkey-patching). We can still, however, enrich existing libraries with implicit conversions using the **pimp my library** pattern (<http://www.artima.com/weblogs/viewpost.jsp?thread=179766>). We first define a `RichConnection` class that contains the `withQuery` method. This `RichConnection` class is created from an existing `Connection` instance.

```
// RichConnection.scala

import java.sql.{Connection, ResultSet}

class RichConnection(val underlying: Connection) {

    /** Execute a SQL query and process the ResultSet */
    def withQuery[T](query: String)(f: ResultSet => T): T = {
        val statement = underlying.prepareStatement(query)
        val results = statement.executeQuery
        try {
            f(results) // loan the ResultSet to the client
        }
        finally {
            // Ensure all the resources get freed.
            results.close
            statement.close
        }
    }
}
```

We could use this class by just wrapping every `Connection` instance in a `RichConnection` instance:

```
// Warning: poor Scala code
SqlUtils(usingConnection("test")) { connection =>
    val richConnection = new RichConnection(connection)
    richConnection.withQuery("SELECT * FROM physicists") {
        resultSet => // process resultSet
    }
}
```

This adds unnecessary boilerplate: we have to remember to convert every connection instance to `RichConnection` to use `withQuery`. Fortunately, Scala provides an easier way with implicit conversions: we tell Scala how to convert from `Connection` to `RichConnection` and vice versa, and tell it to perform this conversion automatically (implicitly), if necessary:

```
// Implicits.scala
import java.sql.Connection

// Implicit conversion methods are often put in
// an object called Implicits.
object Implicits {
    implicit def pimpConnection(conn: Connection) =
        new RichConnection(conn)
    implicit def depimpConnection(conn: RichConnection) =
        conn.underlying
}
```

Now, whenever `pimpConnection` and `depimpConnection` are in the current scope, Scala will automatically use them to convert from `Connection` instances to `RichConnection` and back as needed.

We can now write the following (I have added type information for emphasis):

```
// Bring the conversion functions into the current scope
import Implicits._

SqlUtils(usingConnection("test")) { (connection: Connection) =>
    connection.withQuery("SELECT * FROM physicists") {
        // Wow! It's like we have just added
        // .withQuery to the JDBC Connection class!
        resultSet => // process results
    }
}
```

This might look like magic, so let's step back and look at what happens when we call `withQuery` on a `Connection` instance. The Scala compiler will first look to see if the class definition of `Connection` defines a `withQuery` method. When it finds that it does not, it will look for implicit methods that convert a `Connection` instance to a class that defines `withQuery`. It will find that the `pimpConnection` method allows conversion from `Connection` to `RichConnection`, which defines `withQuery`. The Scala compiler automatically uses `pimpConnection` to transform the `Connection` instance to `RichConnection`.

Note that we used the names `pimpConnection` and `depimpConnection` for the conversion functions, but they could have been anything. We never call these methods explicitly.

Let's summarize how to use the *pimp my library* pattern to add methods to an existing class:

1. Write a class that wraps the class you want to enrich: `class RichConnection(val underlying: Connection)`. Add all the methods that you wish the original class had.
2. Write a method to convert from your original class to your enriched class as part of an object called (conventionally) `Implicits`. Make sure that you tell Scala to use this conversion automatically with the `implicit` keyword: `implicit def pimpConnection(conn: Connection): RichConnection`. You can also tell Scala to automatically convert back from the enriched class to the original class by adding the reverse conversion method.
3. Allow implicit conversions by importing the implicit conversion methods:
`import Implicits._`

Wrapping result sets in a stream

The JDBC `ResultSet` object plays very badly with Scala collections. The only real way of doing anything useful with it is to loop through it directly with a `while` loop. For instance, to get a list of the names of physicists in our database, we could write the following code:

```
// WARNING: poor Scala code
import Implicits._ // import implicit conversions

SqlUtils.usingConnection("test") { connection =>
  connection.withQuery("SELECT * FROM physicists") { resultSet =>
    var names = List.empty[String]
    while(resultSet.next) {
```

```
    val name = resultSet.getString("name")
    names = name :: names
}
names
}
//=> List[String] = List(Paul Dirac, Albert Einstein, Marie Curie,
Richard Feynman, Isaac Newton)
```

The `ResultSet` interface feels unnatural because it behaves very differently from Scala collections. In particular, it does not support the higher-order functions that we take for granted in Scala: no `map`, `filter`, `fold`, or `for` comprehensions. Thankfully, writing a *stream* that wraps `ResultSet` is quite straightforward. A Scala stream is a lazily evaluated list: it evaluates the next element in the collection when it is needed and forgets previous elements when they are no longer used.

We can define a `stream` method that wraps `ResultSet` as follows:

```
// SqlUtils.scala
object SqlUtils {
  ...
  def stream(results:ResultSet):Stream[ResultSet] =
    if (results.next) { results #:: stream(results) }
    else { Stream.empty[ResultSet] }
}
```

This might look quite confusing, so let's take it slowly. We define a `stream` method that wraps `ResultSet`, returning a `Stream[ResultSet]`. When the client calls `stream` on an empty result set, this just returns an empty stream. When the client calls `stream` on a non-empty `ResultSet`, the `ResultSet` instance is advanced by one row, and the client gets back `results #:: stream(results)`. The `#::` operator on a stream is similar to the `cons` operator, `::`, on a list: it prepends `results` to an existing `Stream`. The critical difference is that, unlike a list, `stream(results)` does not get evaluated until necessary. This, therefore, avoids duplicating the entire `ResultSet` in memory.

Let's use our brand new `stream` function to get the name of all the physicists in our database:

```
import Implicits._

SqlUtils(usingConnection("test")) { connection =>
  connection.withQuery("SELECT * FROM physicists") { results =>
    val resultsStream = SqlUtils.stream(results)
```

```

    resultsStream.map { _.getString("name") }.toVector
}
}
//=> Vector(Richard Feynman, Albert Einstein, Marie Curie, Paul Dirac)

```

Streaming the results, rather than using the result set directly, lets us interact with the data much more naturally as we are now dealing with just a Scala collection.

When you use `stream` in a `withQuery` block (or, generally, in a block that automatically closes the result set), you must always materialize the stream within the function, hence the call to `toVector`. Otherwise, the stream will wait until its elements are needed to materialize them, and by then, the `ResultSet` instance will be closed.

Looser coupling with type classes

So far, we have been reading and writing simple types to the database. Let's imagine that we want to add a `gender` column to our database. We will store the gender as an enumeration in our `physicists` database. Our table is now as follows:

```

mysql> CREATE TABLE physicists (
    id INT(11) AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(32) NOT NULL,
    gender ENUM("Female", "Male") NOT NULL
);

```

How can we represent genders in Scala? A good way of doing this is with an enumeration:

```

// Gender.scala

object Gender extends Enumeration {
    val Male = Value
    val Female = Value
}

```

However, we now have a problem when deserializing objects from the database: JDBC has no built-in mechanism to convert from a SQL `ENUM` type to a Scala `Gender` type. We could achieve this by just converting manually every time we need to read gender information:

```

resultsStream.map {
    rs => Gender.withName(rs.getString("gender"))
}.toVector

```

However, we would need to write this everywhere that we want to read the gender field. This goes against the DRY (don't repeat yourself) principle, leading to code that is difficult to maintain. If we decide to change the way gender is stored in the database, we would need to find every instance in the code where we read the gender field and change it.

A somewhat better solution would be to add a `getGender` method to the `ResultSet` class using the pimp my library idiom that we used extensively in this chapter. This solution is still not optimal. We are adding unnecessary specificity to `ResultSet`: it is now coupled to the structure of our databases.

We could create a subclass of `ResultSet` using inheritance, such as `PhysicistResultSet`, that can read the fields in a specific table. However, this approach is not composable: if we had another table that kept track of pets, with name, species, and gender fields, we would have to either reimplement the code for reading gender in a new `PetResultSet` or factor out a `GenderedResultSet` superclass. As the number of tables grows, the inheritance hierarchy would become unmanageable. A better approach would let us compose the functionality that we need. In particular, we want to decouple the process of extracting Scala objects from a result set from the code for iterating over a result set.

Type classes

Scala provides an elegant solution using *type classes*. Type classes are a very powerful arrow in the Scala architect's quiver. However, they can present a bit of a learning curve, especially as there is no direct equivalent in object-oriented programming.

Instead of presenting an abstract explanation, I will dive into an example: I will describe how we can leverage type classes to convert fields in a `ResultSet` to Scala types. The aim is to define a `read[T](field)` method on `ResultSet` that knows exactly how to deserialize to objects of type `T`. This method will replace and extend the `getXXX` methods in `ResultSet`:

```
// results is a ResultSet instance
val name = results.read[String]("name")
val gender = results.read[Gender.Value]("gender")
```

We start by defining an abstract `SqlReader[T]` trait that exposes a `read` method to read a specific field from a `ResultSet` and return an instance of type `T`:

```
// SqlReader.scala

import java.sql._

trait SqlReader[T] {
    def read(results:ResultSet, field:String) :T
}
```

We now need to provide a concrete implementation of `SqlReader[T]` for every `T` type that we want to read. Let's provide concrete implementations for the `Gender` and `String` fields. We will place the implementation in a `SqlReader` companion object:

```
// SqlReader.scala

object SqlReader {
    implicit object StringReader extends SqlReader[String] {
        def read(results:ResultSet, field:String) :String =
            results.getString(field)
    }

    implicit object GenderReader extends SqlReader[Gender.Value] {
        def read(results:ResultSet, field:String) :Gender.Value =
            Gender.withName(StringReader.read(results, field))
    }
}
```

We could now use our `ReadableXXX` objects to read from a result set:

```
import SqlReader._

val name = StringReader.read(results, "name")
val gender = GenderReader.read(results, "gender")
```

This is already somewhat better than using the following:

```
Gender.withName(results.getString("gender"))
```

This is because the code to map from a `ResultSet` field to `Gender.Value` is centralized in a single place: `ReadableGender`. However, it would be great if we could tell Scala to use `ReadableGender` whenever it needs to read `Gender.Value`, and use `ReadableString` whenever it needs to read a `String` value. This is exactly what type classes do.

Coding against type classes

We defined a `Readable[T]` interface that abstracts how to read an object of type `T` from a field in a `ResultSet`. How do we tell Scala that it needs to use this `Readable` object to convert from the `ResultSet` fields to the appropriate Scala type?

The key is the `implicit` keyword that we used to prefix the `GenderReader` and `StringReader` object definitions. It lets us write:

```
implicitly[SqlReader[Gender.Value]].read(results, "gender")
implicitly[SqlReader[String]].read(results, "name")
```

By writing `implicitly[SqlReader[T]]`, we are telling the Scala compiler to find a class (or an object) that extends `SqlReader[T]` that is marked for implicit use. Try this out by pasting the following in the command line, for instance:

```
scala> :paste

import Implicits._ // Connection to RichConnection conversion
SqlUtils.usingConnection("test") {
  _.withQuery("select * from physicists") {
    rs => {
      rs.next() // advance to first record
      implicitly[SqlReader[Gender.Value]].read(rs, "gender")
    }
  }
}
```

Of course, using `implicitly[SqlReader[T]]` everywhere is not particularly elegant. Let's use the pimp my library idiom to add a `read[T]` method to `ResultSet`. We first define a `RichResultSet` class that we can use to "pimp" the `ResultSet` class:

```
// RichResultSet.scala

import java.sql.ResultSet

class RichResultSet(val underlying:ResultSet) {
  def read[T : SqlReader](field:String):T = {
    implicitly[SqlReader[T]].read(underlying, field)
  }
}
```

The only unfamiliar part of this should be the `read[T : SqlReader]` generic definition. We are stating here that `read` will accept any `T` type, provided an instance of `SqlReader[T]` exists. This is called a *context bound*.

We must also add implicit methods to the `Implicits` object to convert from `ResultSet` to `RichResultSet`. You should be familiar with this now, so I will not bore you with the details. You can now call `results.read[T](fieldName)` for any `T` for which you have a `SqlReader[T]` implicit object defined:

```
import Implicits._

SqlUtils(usingConnection("test")) { connection =>
  connection.withQuery("SELECT * FROM physicists") {
    results =>
      val resultStream = SqlUtils.stream(results)
      resultStream.map { row =>
        val name = row.read[String]("name")
        val gender = row.read[Gender.Value]("gender")
        (name, gender)
      }.toVector
  }
}
//=> Vector[(String, Gender.Value)] = Vector((Albert Einstein, Male),
  (Marie Curie, Female))
```

Let's summarize the steps needed for type classes to work. We will do this in the context of deserializing from SQL, but you will be able to adapt these steps to solve other problems:

- Define an abstract generic trait that provides the interface for the type class, for example, `SqlReader[T]`. Any functionality that is independent of `T` can be added to this base trait.
- Create the companion object for the base trait and add implicit objects extending the trait for each `T`, for example,

```
implicit object StringReader extends SqlReader[T].
```
- Type classes are always used in generic methods. A method that relies on the existence of a type class for an argument must contain a context bound in the generic definition, for example, `def read[T : SqlReader](field:String) : T`. To access the type class in this method, use the `implicitly` keyword: `implicitly[SqlReader[T]]`.

When to use type classes

Type classes are useful when you need a particular behavior for many different types, but exactly how this behavior is implemented varies between these types. For instance, we need to be able to read several different types from `ResultSet`, but exactly how each type is read differs between types: for strings, we must read from `ResultSet` using `getString`, whereas for integers, we must use `getInt` followed by `wasNull`.

A good rule of thumb is when you start thinking "Oh, I could just write a generic method to do this. Ah, but wait, I will have to write the `Int` implementation as a specific edge case as it behaves differently. Oh, and the `Gender` implementation. I wonder if there's a better way?", then type classes might be useful.

Benefits of type classes

Data scientists frequently have to deal with new input streams, changing requirements, and new data types. Having an object-relational mapping layer that is easy to extend or alter is therefore critical to responding to changes efficiently. Minimizing coupling between code entities and separation of concerns are the only ways to ensure that the code can be changed in response to new data.

With type classes, we maintain orthogonality between accessing records in the database (through the `ResultSet` class) and how individual fields are transformed to Scala objects: both can vary independently. The only coupling between these two concerns is through the `SqlReader[T]` interface.

This means that both concerns can evolve independently: to read a new data type, we just need to implement a `SqlReader[T]` object. Conversely, we can add functionality to `ResultSet` without needing to reimplement how fields are converted. For instance, we could add a `getColumn` method that returns a `Vector[T]` of all the values of a field in a `ResultSet` instance:

```
def getColumn[T : SqlReader](field:String):Vector[T] = {  
    val resultStream = SqlUtils.stream(results)  
    resultStream.map { _.read[T](field) }.toVector  
}
```

Note how we could do this without increasing the coupling to the way in which individual fields are read.

Creating a data access layer

Let's bring together everything that we have seen and build a *data-mapper* class for fetching `Physicist` objects from the database. These classes (also called *data access objects*) are useful to decouple the internal representation of an object from its representation in the database.

We start by defining the `Physicist` class:

```
// Physicist.scala
case class Physicist(
    val name:String,
    val gender:Gender.Value
)
```

The data access object will expose a single method, `readAll`, that returns a `Vector[Physicist]` of all the physicists in our database:

```
// PhysicistDao.scala

import java.sql.{ ResultSet, Connection }
import Implicits._ // implicit conversions

object PhysicistDao {

    /* Helper method for reading a single row */
    private def readFromResultSet(results:ResultSet):Physicist = {
        Physicist(
            results.read[String] ("name"),
            results.read[Gender.Value] ("gender")
        )
    }

    /* Read the entire 'physicists' table. */
    def readAll(connection:Connection):Vector[Physicist] = {
        connection.withQuery("SELECT * FROM physicists") {
            results =>
            val resultStream = SqlUtils.stream(results)
            resultStream.map(readFromResultSet).toVector
        }
    }
}
```

The data access layer can be used by client code as in the following example:

```
object PhysicistDaoDemo extends App {  
  
    val physicists = SqlUtils.usingConnection("test") {  
        connection => PhysicistDao.readAll(connection)  
    }  
  
    // physicists is a Vector[Physicist] instance.  
    physicists.foreach { println }  
    //=> Physicist(Albert Einstein, Male)  
    //=> Physicist(Marie Curie, Female)  
}
```

Summary

In this chapter, we learned how to interact with SQL databases using JDBC. We wrote a library to wrap native JDBC objects, aiming to give them a more functional interface.

In the next chapter, you will learn about Slick, a Scala library that provides functional wrappers to interact with relational databases.

References

The API documentation for JDBC is very complete: <http://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html>

The API documentation for the `ResultSet` interface (<http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html>), for the `PreparedStatement` class (<http://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html>) and the `Connection` class (<http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html>) is particularly relevant.

The data mapper pattern is described extensively in Martin Fowler's *Patterns of Enterprise Application Architecture*. A brief description is also available on his website (<http://martinfowler.com/eaaCatalog/dataMapper.html>).

For an introduction to SQL, I suggest *Learning SQL* by Alan Beaulieu (O'Reilly).

For another discussion of type classes, read <http://danielwestheide.com/blog/2013/02/06/the-neophytes-guide-to-scala-part-12-type-classes.html>.

This post describes how some common object-oriented design patterns can be reimplemented more elegantly in Scala using type classes:

<https://staticallytyped.wordpress.com/2013/03/24/gang-of-four-patterns-with-type-classes-and-implicits-in-scala-part-2/>

This post by Martin Odersky details the *Pimp my Library* pattern:

<http://www.artima.com/weblogs/viewpost.jsp?thread=179766>

6

Slick – A Functional Interface for SQL

In *Chapter 5, Scala and SQL through JDBC*, we investigated how to access SQL databases with JDBC. As interacting with JDBC feels somewhat unnatural, we extended JDBC using custom wrappers. The wrappers were developed to provide a functional interface to hide the imperative nature of JDBC.

With the difficulty of interacting directly with JDBC from Scala and the ubiquity of SQL databases, you would expect there to be existing Scala libraries that wrap JDBC. *Slick* is such a library.

Slick styles itself as a *functional-relational mapping* library, a play on the more traditional *object-relational mapping* name used to denote libraries that build objects from relational databases. It presents a functional interface to SQL databases, allowing the client to interact with them in a manner similar to native Scala collections.

FEC data

In this chapter, we will use a somewhat more involved example dataset. The **Federal Electoral Commission of the United States (FEC)** records all donations to presidential candidates greater than \$200. These records are publicly available. We will look at the donations for the campaign leading up to the 2012 general elections that resulted in Barack Obama's re-election. The data includes donations to the two presidential candidates, Obama and Romney, and also to the other contenders in the Republican primaries (there were no Democrat primaries).

In this chapter, we will take the transaction data provided by the FEC, store it in a table, and learn how to query and analyze it.

The first step is to acquire the data. If you have downloaded the code samples from the Packt website, you should already have two CSVs in the `data` directory of the code samples for this chapter. If not, you can download the files using the following links:

- data.scala4datascience.com/fec/ohio.csv.gz (or `ohio.csv.zip`)
- data.scala4datascience.com/fec/us.csv.gz (or `us.csv.zip`)

Decompress the two files and place them in a directory called `data/` in the same location as the source code examples for this chapter. The data files correspond to the following:

- The `ohio.csv` file is a CSV of all the donations made by donors in Ohio.
- The `us.csv` file is a CSV of all the donations made by donors across the country. This is quite a large file, with six million rows.

The two CSV files contain identical columns. Use the Ohio dataset for more responsive behavior, or the nationwide data file if you want to wrestle with a larger dataset. The dataset is adapted from a list of contributions downloaded from <http://www.fec.gov/disclosurep/PDownload.do>.

Let's start by creating a Scala case class to represent a transaction. In the context of this chapter, a transaction is a single donation from an individual to a candidate:

```
// Transaction.scala
import java.sql.Date

case class Transaction(
    id:Option[Int], // unique identifier
    candidate:String, // candidate receiving the donation
    contributor:String, // name of the contributor
    contributorState:String, // contributor state
    contributorOccupation:Option[String], // contributor job
    amount:Long, // amount in cents
    date:Date // date of the donation
)
```

The code repository for this chapter includes helper functions in an `FECData` singleton object to load the data from CSVs:

```
scala> val ohioData = FECData.loadOhio
s4ds.FECData = s4ds.FECData@718454de
```

Calling `FECDATA.loadOhio` or `FECDATA.loadAll` will create an `FECDATA` object with a single attribute, `transactions`, which is an iterator over all the donations coming from Ohio or the entire United States:

```
scala> val ohioTransactions = ohioData.transactions
Iterator[Transaction] = non-empty iterator

scala> ohioTransactions.take(5).foreach(println)
Transaction(None,Paul, Ron,BROWN, TODD W MR.,OH,Some(ENGINE
ER),5000,2011-01-03)
Transaction(None,Paul, Ron,DIEHL, MARGO SONJA,OH,Some(RETIR
ED),2500,2011-01-03)
Transaction(None,Paul, Ron,KIRCHMEYER, BENJAMIN,OH,Some(COMPUTER
PROGRAMMER),20120,2011-01-03)
Transaction(None,Obama, Barack,KEYES, STEPHEN,OH,Some(HR EXECUTIVE /
ATTORNEY),10000,2011-01-03)
Transaction(None,Obama, Barack,MURPHY, MIKE W,OH,Some(MANAG
ER),5000,2011-01-03)
```

Now that we have some data to play with, let's try and put it in the database so that we can run some useful queries on it.

Importing Slick

To add Slick to the list of dependencies, you will need to add `"com.typesafe.slick" %% "slick" % "2.1.0"` to the list of dependencies in your `build.sbt` file. You will also need to make sure that Slick has access to a JDBC driver. In this chapter, we will connect to a MySQL database, and must, therefore, add the MySQL connector `"mysql" % "mysql-connector-java" % "5.1.37"` to the list of dependencies.

Slick is imported by importing a specific database driver. As we are using MySQL, we must import the following:

```
scala> import slick.driver.MySQLDriver.simple._
import slick.driver.MySQLDriver.simple._
```

To connect to a different flavor of SQL database, import the relevant driver. The easiest way of seeing what drivers are available is to consult the API documentation for the `slick.driver` package, which is available at <http://slick.typesafe.com/doc/2.1.0/api/#scala/slick.driver.package>. All the common SQL flavors are supported (including **H2**, **PostgreSQL**, **MS SQL Server**, and **SQLite**).

Defining the schema

Let's create a table to represent our transactions. We will use the following schema:

```
CREATE TABLE transactions (
    id INT(11) AUTO_INCREMENT PRIMARY KEY,
    candidate VARCHAR(254) NOT NULL,
    contributor VARCHAR(254) NOT NULL,
    contributor_state VARCHAR(2) NOT NULL,
    contributor_occupation VARCHAR(254),
    amount BIGINT(20) NOT NULL,
    date DATE
);
```

Note that the donation amount is in *cents*. This allows us to use an integer field (rather than a fixed point decimal, or worse, a float).

You should never use a floating point format to represent money or, in fact, any discrete quantity because floats cannot represent most fractions exactly:

 `scala> 0.1 + 0.2`
`Double = 0.30000000000000004`

This seemingly nonsensical result occurs because there is no way to store 0.3 exactly in doubles.

This post gives an extensive discussion of the limitations of the floating point format:

http://docs.oracle.com/cd/E19957-01/806-3568/ngc_goldberg.html

To use Slick with tables in our database, we first need to tell Slick about the database schema. We do this by creating a class that extends the `Table` abstract class. The way in which a schema is defined is quite straightforward, so let's dive straight into the code. We will store our schema in a `Tables` singleton. We define a `Transactions` class that provides the mapping to go from collections of `Transaction` instances to SQL tables structured like the `transactions` table:

```
// Tables.scala

import java.sql.Date
import slick.driver.MySQLDriver.simple._

/** Singleton object for table definitions */
```

```
object Tables {

    // Transactions table definition
    class Transactions(tag:Tag)
        extends Table[Transaction](tag, "transactions") {
        def id = column[Int]("id", O.PrimaryKey, O.AutoInc)
        def candidate = column[String]("candidate")
        def contributor = column[String]("contributor")
        def contributorState = column[String](
            "contributor_state", O.DBType("VARCHAR(2)"))
        def contributorOccupation = column[Option[String]](
            "contributor_occupation")
        def amount = column[Long]("amount")
        def date = column[Date]("date")

        def * = (id.?, candidate, contributor,
            contributorState, contributorOccupation, amount, date) <-> (
            Transaction.tupled, Transaction.unapply)
    }

    val transactions = TableQuery[Transactions]
}

}
```

Let's go through this line by line. We first define a `Transactions` class, which must take a Slick `Tag` object as its first argument. The `Tag` object is used by Slick internally to construct SQL statements. The `Transactions` class extends a `Table` object, passing it the tag and name of the table in the database. We could, optionally, have added a database name by extending `Table[Transaction](tag, Some("fec"), "transactions")` rather than just `Table[Transaction](tag, "transactions")`. The `Table` type is parametrized by `Transaction`. This means that running `SELECT` statements on the database returns `Transaction` objects. Similarly, we will insert data into the database by passing a transaction or list of transactions to the relevant Slick methods.

Let's look at the `Transactions` class definition in more detail. The body of the class starts by listing the database columns. For instance, the `id` column is defined as follows:

```
def id = column[Int]("id", O.PrimaryKey, O.AutoInc)
```

We tell Slick that it should read the column called `id` and transform it to a Scala integer. Additionally, we tell Slick that this column is the primary key and that it is auto-incrementing. The Slick documentation contains a list of available options for `column`.

The `candidate` and `contributor` columns are straightforward: we tell Slick to read these as `String` from the database. The `contributor_state` column is a little more interesting. Besides specifying that it should be read from the database as a `String`, we also tell Slick that it should be stored in the database with type `VARCHAR(2)`.

The `contributor_occupation` column in our table can contain `NULL` values. When defining the schema, we pass the `Option[String]` type to the `column` method:

```
def contributorOccupation =  
    column[Option[String]] ("contributor_occupation")
```

When reading from the database, a `NULL` field will get converted to `None` for columns specified as `Option[T]`. Conversely, if the field has a value, it will be returned as `Some(value)`.

The last line of the class body is the most interesting part: it specifies how to transform the raw data read from the database into a `Transaction` object and how to convert a `Transaction` object to raw fields ready for insertion:

```
def * = (id.?, candidate, contributor,  
        contributorState, contributorOccupation, amount, date) <-> (  
    Transaction.tupled, Transaction.unapply)
```

The first part is just a tuple of fields to be read from the database: `(id.?, candidate, contributor, contributorState, contributorOccupation, amount, date)`, with a small amount of metadata. The second part is a pair of functions that describe how to transform this tuple into a `Transaction` object and back. In this case, as `Transaction` is a case class, we can take advantage of the `Transaction.tupled` and `Transaction.unapply` methods automatically provided for case classes.

Notice how we followed the `id` entry with `.?`. In our `Transaction` class, the donation `id` has the `Option[Int]` type, but the column in the database has the `INT` type with the additional `O.AutoInc` option. The `.?` suffix tells Slick to use the default value provided by the database (in this case, the database's auto-increment) if `id` is `None`.

Finally, we define the value:

```
val transactions = TableQuery[Transactions]
```

This is the handle that we use to actually interact with the database. For instance, as we will see later, to get a list of donations to Barack Obama, we run the following query (don't worry about the details of the query for now):

```
Tables.transactions.filter {_.candidate === "Obama, Barack"}.list
```

Let's summarize the parts of our `Transactions` mapper class:

- The `Transactions` class must extend the `Table` abstract class parametrized by the type that we want to return: `Table[Transaction]`.
- We define the columns to read from the database explicitly using `column`, for example, `def contributorState = column[String]("contributor_state", O.DBType("VARCHAR(2)"))`. The `[String]` type parameter defines the Scala type that this column gets read as. The first argument is the SQL column name. Consult the Slick documentation for a full list of additional arguments (<http://slick.typesafe.com/doc/2.1.0/schemas.html>).
- We describe how to convert from a tuple of the column values to a Scala object and vice versa using `def * = (id.?, candidate, ...) <=> (Transaction.tupled, Transaction.unapply)`.

Connecting to the database

So far, you have learned how to define `Table` classes that encode the transformation from rows in a SQL table to Scala case classes. To move beyond table definitions and start interacting with a database server, we must connect to a database. As in the previous chapter, we will assume that there is a MySQL server running on localhost on port 3306.

We will use the console to demonstrate the functionality in this chapter, but you can find an equivalent sample program in `SlickDemo.scala`. Let's open a Scala console and connect to the database running on port 3306:

```
scala> import slick.driver.MySQLDriver.simple._
import slick.driver.MySQLDriver.simple._

scala> val db = Database.forName(
  "jdbc:mysql://127.0.0.1:3306/test",
  driver="com.mysql.jdbc.Driver"
)
db: slick.driver.MySQLDriver.backend.DatabaseDef = slick.jdbc.JdbcBackend
$DatabaseDef@3632d1dd
```

If you have read the previous chapter, you will recognize the first argument as a JDBC-style URL. The URL starts by defining a protocol, in this case, `jdbc:mysql`, followed by the IP address and port of the database server, followed by the database name (`test`, here).

The second argument to `forURL` is the class name of the JDBC driver. This driver is imported at runtime using reflection. Note that the driver specified here must match the Slick driver imported statically.

Having defined the database, we can now use it to create a connection:

```
scala> db.withSession { implicit session =>
  // do something useful with the database
  println(session)
}

scala.slick.jdbc.JdbcBackend$BaseSession@af5a276
```

Slick functions that require access to the database take a `Session` argument implicitly: if a `Session` instance marked as `implicit` is available in scope, they will use it. Thus, preceding `session` with the `implicit` keyword saves us having to pass `session` explicitly every time we run an operation on the database.

If you have read the previous chapter, you will recognize that Slick deals with the need to close connections with the *loan pattern*: a database connection is created in the form of a `Session` object and passed temporarily to the client. When the client code returns, the session is closed, ensuring that all opened connections are closed. The client code is therefore spared the responsibility of closing the connection.

The loan pattern is very useful in production code, but it can be somewhat cumbersome in the shell. Slick lets us create a session explicitly as follows:

```
scala> implicit val session = db.createSession
session: slick.driver.MySQLDriver.backend.Session = scala.slick.jdbc.Jdbc
Backend$BaseSession@2b775b49

scala> session.close
```

Creating tables

Let's use our new connection to create the transaction table in the database. We can access methods to create and drop tables using the `ddl` attribute on our `TableQuery[Transactions]` instance:

```
scala> db.withSession { implicit session =>
  Tables.transactions.ddl.create
}
```

If you jump into a mysql shell, you will see that a transactions table has been created:

```
mysql> describe transactions ;
+-----+-----+-----+-----+
| Field          | Type       | Null | Key |
+-----+-----+-----+-----+
| id             | int(11)    | NO   | PRI  |
| candidate      | varchar(254)| NO   |       |
| contributor    | varchar(254)| NO   |       |
| contributor_state | varchar(2) | NO   |       |
| contributor_occupation | varchar(254)| YES  |       |
| amount          | bigint(20) | NO   |       |
| date            | date       | NO   |       |
+-----+-----+-----+-----+
7 rows in set (0.01 sec)
```

The `ddl` attribute also includes a `drop` method to drop the table. Incidentally, `ddl` stands for "data-definition language" and is commonly used to refer to the parts of SQL relevant to schema and constraint definitions.

Inserting data

Slick `TableQuery` instances let us interact with SQL tables with an interface similar to Scala collections.

Let's create a transaction first. We will pretend that a donation occurred on the 22nd of June, 2010. Unfortunately, the code to create dates in Scala and pass these to JDBC is particularly clunky. We first create a `java.util.Date` instance, which we must then convert to a `java.sql.Date` to use in our newly created transaction:

```
scala> import java.text.SimpleDateFormat
import java.text.SimpleDateFormat

scala> val date = new SimpleDateFormat("dd-MM-yyyy").parse("22-06-2010")
date: java.util.Date = Tue Jun 22 00:00:00 BST 2010

scala> val sqlDate = new java.sql.Date(date.getTime())
sqlDate: java.sql.Date = 2010-06-22

scala> val transaction = Transaction(
```

```
None, "Obama, Barack", "Doe, John", "TX", None, 200, sqlDate
)
transaction: Transaction = Transaction(None,Obama, Barack,Doe,
John,TX,None,200,2010-06-22)
```

Much of the interface provided by the `TableQuery` instance mirrors that of a mutable list. To insert a single row in the transaction table, we can use the `+=` operator:

```
scala> db.withSession {
    implicit session => Tables.transactions += transaction
}
Int = 1
```

Under the hood, this will create a JDBC prepared statement and run this statement's `executeUpdate` method.

If you are committing many rows at a time, you should use Slick's bulk insert operator: `++=`. This takes a `List[Transaction]` as input and inserts all the transactions in a single batch by taking advantage of JDBC's `addBatch` and `executeBatch` functionality.

Let's insert all the FEC transactions so that we have some data to play with when running queries in the next section. We can load an iterator of transactions for Ohio by calling the following:

```
scala> val transactions = FECData.loadOhio.transactions
transactions: Iterator[Transaction] = non-empty iterator
```

We can also load the transactions for the whole of United States:

```
scala> val transactions = FECData.loadAll.transactions
transactions: Iterator[Transaction] = non-empty iterator
```

To avoid materializing all the transactions in a single fell swoop—thus potentially exceeding our computer's available memory—we will take batches of transactions from the iterator and insert them:

```
scala> val batchSize = 100000
batchSize: Int = 100000

scala> val transactionBatches = transactions.grouped(batchSize)
transactionBatches: transactions.GroupedIterator[Transaction] = non-empty
iterator
```

An iterator's grouped method splits the iterator into batches. It is useful to split a long collection or iterator into manageable batches that can be processed one after the other. This is important when integrating or processing large datasets.

All that we have to do now is iterate over our batches, inserting them into the database as we go:

```
scala> db.withSession { implicit session =>
  transactionBatches.foreach {
    batch => Tables.transactions ++= batch.toList
  }
}
```

While this works, it is sometimes useful to see progress reports when doing long-running integration processes. As we have split the integration into batches, we know (to the nearest batch) how far into the integration we are. Let's print the progress information at the beginning of every batch:

```
scala> db.withSession { implicit session =>
  transactionBatches.zipWithIndex.foreach {
    case (batch, batchNumber) =>
      println(s"Processing row ${batchNumber*batchSize}")
      Tables.transactions ++= batch.toList
  }
}
Processing row 0
Processing row 100000
...
```

We use the `.zipWithIndex` method to transform our iterator over batches into an iterator of (*batch, current index*) pairs. In a full-scale application, the progress information would probably be written to a log file rather than to the screen.

Slick's well-designed interface makes inserting data very intuitive, integrating well with native Scala types.

Querying data

In the previous section, we used Slick to insert donation data into our database. Let's explore this data now.

When defining the `Transactions` class, we defined a `TableQuery` object, `transactions`, that acts as the handle for accessing the transaction table. It exposes an interface similar to Scala iterators. For instance, to see the first five elements in our database, we can call `take(5)`:

```
scala> db.withSession { implicit session =>
  Tables.transactions.take(5).list
}
List[Tables.Transactions#TableElementType] =
List(Transaction(Some(1),Obama, Barack, Doe, ...)
```

Internally, Slick implements the `.take` method using a SQL `LIMIT`. We can, in fact, get the SQL statement using the `.selectStatement` method on the query:

```
scala> db.withSession { implicit session =>
  println(Tables.transactions.take(5).selectStatement)
}
select x2.`id`, x2.`candidate`, x2.`contributor`, x2.`contributor_
state`, x2.`contributor_occupation`, x2.`amount`, x2.`date` from
(select x3.`date` as `date`, x3.`contributor` as `contributor`,
x3.`amount` as `amount`, x3.`id` as `id`, x3.`candidate` as `candidate`,
x3.`contributor_state` as `contributor_state`, x3.`contributor_
occupation` as `contributor_occupation` from `transactions` x3 limit 5)
x2
```

Our Slick query is made up of the following two parts:

- `.take(n)`: This part is called the *invoker*. Invokers build up the SQL statement but do not actually fire it to the database. You can chain many invokers together to build complex SQL statements.
- `.list`: This part sends the statement prepared by the invoker to the database and converts the result to Scala object. This takes a session argument, possibly implicitly.

Invokers

Invokers are the components of a Slick query that build up the SQL select statement. Slick exposes a variety of invokers that allow the construction of complex queries. Let's look at some of these invokers here:

- The `map` invoker is useful to select individual columns or apply operations to columns:

```
scala> db.withSession { implicit session =>
  Tables.transactions.map {
    _.candidate
  }.take(5).list
}
List[String] = List(Obama, Barack, Paul, Ron, Paul, Ron, Paul,
Ron, Obama, Barack)
```

- The `filter` invoker is the equivalent of the `WHERE` statements in SQL. Note that Slick fields must be compared using `==`:

```
scala> db.withSession { implicit session =>
  Tables.transactions.filter {
    _.candidate === "Obama, Barack"
  }.take(5).list
}
List[Tables.Transactions#TableElementType] =
List(Transaction(Some(1),Obama, Barack,Doe,
John,TX,None,200,2010-06-22), ...)
```

Similarly, to filter out donations to Barack Obama, use the `=!=` operator:

```
scala> db.withSession { implicit session =>
  Tables.transactions.filter {
    _.candidate =!= "Obama, Barack"
  }.take(5).list
}
List[Tables.Transactions#TableElementType] =
List(Transaction(Some(2),Paul, Ron,BROWN, TODD W MR.,OH,...)
```

- The `sortBy` invoker is the equivalent of the `ORDER BY` statement in SQL:

```
scala> db.withSession { implicit session =>
  Tables.transactions.sortBy {
    _.date.desc
}
```

```
    }.take(5).list
}
List[Tables.Transactions#TableElementType] = List(Transaction(
  Some(65536), Obama, Barack, COPELAND, THOMAS, OH, Some(COLLEGE
  TEACHING), 10000, 2012-01-02)
```

- The `leftJoin`, `rightJoin`, `innerJoin`, and `outerJoin` invokers are used for joining tables. As we do not cover interactions between multiple tables in this tutorial, we cannot demonstrate joins. See the Slick documentation (<http://slick.typesafe.com/doc/2.1.0/queries.html#joining-and-zipping>) for examples of these.
- Aggregation invokers such as `length`, `min`, `max`, `sum`, and `avg` can be used for computing summary statistics. These must be executed using `.run`, rather than `.list`, as they return single numbers. For instance, to get the total donations to Barack Obama:

```
scala> db.withSession { implicit session =>
  Tables.transactions.filter {
    _.candidate === "Obama, Barack"
  }.map { _.amount }.sum.run
}
Option[Int] = Some(849636799) // (in cents)
```

Operations on columns

In the previous section, you learned about the different invokers and how they mapped to SQL statements. We brushed over the methods supported by columns themselves, however: we can compare for equality using `==`, but what other operations are supported by Slick columns?

Most of the SQL functions are supported. For instance, to get the total donations to candidates whose name starts with "O", we could run the following:

```
scala> db.withSession { implicit session =>
  Tables.transactions.filter {
    _.candidate.startsWith("O")
  }.take(5).list
}
List[Tables.Transactions#TableElementType] = List(Transaction(
  Some(1594098) ...
```

Similarly, to count donations that happened between January 1, 2011 and February 1, 2011, we can use the `.between` method on the date column:

```
scala> val dateParser = new SimpleDateFormat("dd-MM-yyyy")
dateParser: java.text.SimpleDateFormat = SimpleDateFormat

scala> val startDate = new java.sql.Date(dateParser.parse("01-01-2011") .
getTime())
startDate: java.sql.Date = 2011-01-01

scala> val endDate = new java.sql.Date(dateParser.parse("01-02-2011") .
getTime())
endDate: java.sql.Date = 2011-02-01

scala> db.withSession { implicit session =>
  Tables.transactions.filter {
    _.date.between(startDate, endDate)
  }.length.run
}
Int = 9772
```

The equivalent of the SQL `IN (...)` operator that selects values in a specific set is `inSet`. For instance, to select all transactions to Barack Obama and Mitt Romney, we can use the following:

```
scala> val candidateList = List("Obama, Barack", "Romney, Mitt")
candidateList: List[String] = List(Obama, Barack, Romney, Mitt)

scala> val donationCents = db.withSession { implicit session =>
  Tables.transactions.filter {
    _.candidate.inSet(candidateList)
  }.map { _.amount }.sum.run
}
donationCents: Option[Long] = Some(2874484657)

scala> val donationDollars = donationCents.map { _ / 100 }
donationDollars: Option[Long] = Some(28744846)
```

So, between them, Mitt Romney and Barack Obama received over 28 million dollars in registered donations.

We can also negate a Boolean column with the `!` operator. For instance, to calculate the total amount of donations received by all candidates apart from Barack Obama and Mitt Romney:

```
scala> db.withSession { implicit session =>
  Tables.transactions.filter {
    !_candidate.inSet(candidateList)
  }.map { _.amount }.sum.run
}.map { _ / 100 }
Option[Long] = Some(1930747)
```

Column operations are added by implicit conversion on the base `Column` instances. For a full list of methods available on String columns, consult the API documentation for the `StringColumnExtensionMethods` class (<http://slick.typesafe.com/doc/2.1.0/api/#scala.slick.lifted.StringColumnExtensionMethods>). For the methods available on Boolean columns, consult the API documentation for the `BooleanColumnExtensionMethods` class (<http://slick.typesafe.com/doc/2.1.0/api/#scala.slick.lifted.BooleanColumnExtensionMethods>). For the methods available on numeric columns, consult the API documentation for `NumericColumnExtensionMethods` (<http://slick.typesafe.com/doc/2.1.0/api/#scala.slick.lifted.NumericColumnExtensionMethods>).

Aggregations with "Group by"

Slick also provides a `groupBy` method that behaves like the `groupBy` method of native Scala collections. Let's get a list of candidates with all the donations for each candidate:

```
scala> val grouped = Tables.transactions.groupBy { _.candidate }
grouped: scala.slick.lifted.Query[(scala.slick.lifted.Column[...]
```



```
scala> val aggregated = grouped.map {
  case (candidate, group) =>
    (candidate -> group.map { _.amount }.sum)
}
aggregated: scala.slick.lifted.Query[(scala.slick.lifted.Column[...]
```



```
scala> val groupedDonations = db.withSession {
  implicit session => aggregated.list
}
```

```
groupedDonations: List[(String, Option[Long])] = List((Bachmann,  
Michele,Some(7439272)),...)
```

Let's break this down. The first statement, `transactions.groupBy { _.candidate }`, specifies the key by which to group. You can think of this as building an intermediate list of `(String, List[Transaction])` tuples mapping the group key to a list of all the table rows that satisfy this key. This behavior is identical to calling `groupBy` on a Scala collection.

The call to `groupBy` must be followed by a `map` that aggregates the groups. The function passed to `map` must take the tuple `(String, List[Transaction])` pair created by the `groupBy` call as its sole argument. The `map` call is responsible for aggregating the `List[Transaction]` object. We choose to first pick out the `amount` field of each transaction, and then to run a sum over these. Finally, we call `.list` on the whole pipeline to actually run the query. This just returns a Scala list. Let's convert the total donations from cents to dollars:

```
scala> val groupedDonationDollars = groupedDonations.map {  
    case (candidate, donationCentsOption) =>  
        candidate -> (donationCentsOption.getOrElse(0L) / 100)  
}  
  
groupedDonationDollars: List[(String, Long)] = List((Bachmann,  
Michele,74392),...)  
  
scala> groupedDonationDollars.sortBy {  
    _._2  
}.reverse.foreach { println }  
(Romney, Mitt,20248496)  
(Obama, Barack,8496347)  
(Paul, Ron,565060)  
(Santorum, Rick,334926)  
(Perry, Rick,301780)  
(Gingrich, Newt,277079)  
(Cain, Herman,210768)  
(Johnson, Gary Earl,83610)  
(Bachmann, Michele,74392)  
(Pawlenty, Timothy,42500)  
(Huntsman, Jon,23571)  
(Roemer, Charles E. 'Buddy' III,8579)  
(Stein, Jill,5270)  
(McCotter, Thaddeus G,3210)
```

Accessing database metadata

Commonly, especially during development, you might start the script by dropping the table if it exists, then recreating it. We can find if a table is defined by accessing the database metadata through the `MTable` object. To get a list of tables with name matching a certain pattern, we can run `MTable.getTables(pattern)`:

```
scala> import slick.jdbc.meta.MTable
import slick.jdbc.meta.MTable

scala> db.withSession { implicit session =>
  MTable.getTables("transactions").list
}

List[scala.slick.jdbc.meta.MTable] = List(MTable(MQName(fec.transactions)
, TABLE, ,None, None, None) ...)
```

Thus, to drop the `transactions` table if it exists, we can run the following:

```
scala> db.withSession { implicit session =>
  if(MTable.getTables("transactions").list.nonEmpty) {
    Tables.transactions.ddl.drop
  }
}
```

The `MTable` instance contains a lot of metadata about the table. Go ahead and recreate the `transactions` table if you dropped it in the previous example. Then, to find information about the table's primary keys:

```
scala> db.withSession { implicit session =>
  val tableMeta = MTable.getTables("transactions").first
  tableMeta.getPrimaryKeys.list
}

List[MPrimaryKey] = List(MPrimaryKey(MQName(test.transactions), id, 1, Some(
PRIMARY)))
```

For a full list of methods available on `MTable` instances, consult the Slick documentation (<http://slick.typesafe.com/doc/2.1.0/api/index.html#scala.slick.jdbc.meta.MTable>).

Slick versus JDBC

This chapter and the previous one introduced two different ways of interacting with SQL. In the previous chapter, we described how to use JDBC and build extensions on top of JDBC to make it more usable. In this chapter, we introduced Slick, a library that provides a functional interface on top of JDBC.

Which method should you choose? If you are starting a new project, you should consider using Slick. Even if you spend a considerable amount of time writing wrappers that sit on top of JDBC, it is unlikely that you will achieve the fluidity that Slick offers.

If you are working on an existing project that makes extensive use of JDBC, I hope that the previous chapter demonstrates that, with a little time and effort, you can write JDBC wrappers that reduce the impedance between the imperative style of JDBC and Scala's functional approach.

Summary

In the previous two chapters, we looked extensively at how to query relational databases from Scala. In this chapter, you learned how to use Slick, a "functional-relational" mapper that allows interacting with SQL databases as one would with Scala collections.

In the next chapter, you will learn how to ingest data by querying web APIs.

References

To learn more about Slick, you can refer to the Slick documentation (<http://slick.typesafe.com/doc/2.1.0/>) and its API documentation (<http://slick.typesafe.com/doc/2.1.0/api/#package>).

7

Web APIs

Data scientists and data engineers get data from a variety of different sources. Often, data might come as CSV files or database dumps. Sometimes, we have to obtain the data through a web API.

An individual or organization sets up a web API to distribute data to programs over the Internet (or an internal network). Unlike websites, where the data is intended to be consumed by a web browser and shown to the user, the data provided by a web API is agnostic to the type of program querying it. Web servers serving HTML and web servers backing an API are queried in essentially the same way: through HTTP requests.

We have already seen an example of a web API in *Chapter 4, Parallel Collections and Futures*, where we queried the "Markit on demand" API for current stock prices. In this chapter, we will explore how to interact with web APIs in more detail; specifically, how to convert the data returned by the API to Scala objects and how to add additional information to the request through HTTP headers (for authentication, for instance).

The "Markit on demand" API returned the data formatted as an XML object, but increasingly, new web APIs return data formatted as JSON. We will therefore focus on JSON in this chapter, but the concepts will port easily to XML.

JSON is a language for formatting structured data. Many readers will have come across JSON in the past, but if not, there is a brief introduction to the syntax and concepts later on in this chapter. You will find it quite straightforward.

In this chapter, we will poll the GitHub API. GitHub has, over the last few years, become the de facto tool for collaborating on open source software. It provides a powerful, feature-rich API that gives programmatic access to nearly all the data available through the website.

Let's get a taste of what we can do. Type `api.github.com/users/odersky` in your web browser address bar. This will return the data offered by the API on a particular user (Martin Odersky, in this case):

```
{  
  "login": "odersky",  
  "id": 795990,  
  ...  
  "public_repos": 8,  
  "public_gists": 3,  
  "followers": 707,  
  "following": 0,  
  "created_at": "2011-05-18T14:51:21Z",  
  "updated_at": "2015-09-15T15:14:33Z"  
}
```

The data is returned as a JSON object. This chapter is devoted to learning how to access and parse this data programmatically. In *Chapter 13, Web APIs with Play*, you will learn how to build your own web API.



The GitHub API is extensive and very well-documented. We will explore some of the features of the API in this chapter. To see the full extent of the API, visit the documentation (<https://developer.github.com/v3/>).

A whirlwind tour of JSON

JSON is a format for transferring structured data. It is flexible, easy for computers to generate and parse, and relatively readable for humans. It has become very common as a means of persisting program data structures and transferring data between programs.

JSON has four basic types: **Numbers**, **Strings**, **Booleans**, and **null**, and two compound types: **Arrays** and **Objects**. Objects are unordered collections of key-value pairs, where the key is always a string and the value can be any simple or compound type. We have already seen a JSON object: the data returned by the API call `api.github.com/users/odersky`.

Arrays are ordered lists of simple or compound types. For instance, type `api.github.com/users/odersky/repos` in your browser to get an array of objects, each representing a GitHub repository:

```
[  
 {  
   "id": 17335228,  
   "name": "dotty",  
   "full_name": "odersky/dotty",  
   ...  
 },  
 {  
   "id": 15053153,  
   "name": "frontend",  
   "full_name": "odersky/frontend",  
   ...  
 },  
 ...  
 ]
```

We can construct complex structures by nesting objects within other objects or arrays. Nevertheless, most web APIs return JSON structures with no more than one or two levels of nesting. If you are not familiar with JSON, I encourage you to explore the GitHub API through your web browser.

Querying web APIs

The easiest way of querying a web API from Scala is to use `Source.fromURL`. We have already used this in *Chapter 4, Parallel Collections and Futures*, when we queried the "Markit on demand" API. `Source.fromURL` presents an interface similar to `Source.fromFile`:

```
scala> import scala.io._  
import scala.io._  
  
scala> val response = Source.fromURL(  
  "https://api.github.com/users/odersky"  
)  
response: String = {"login":"odersky","id":795990, ...
```

`Source.fromURL` returns an iterator over the characters of the response. We materialize the iterator into a string using its `.mkString` method. We now have the response as a Scala string. The next step is to parse the string with a JSON parser.

JSON in Scala – an exercise in pattern matching

There are several libraries for manipulating JSON in Scala. We prefer json4s, but if you are a die-hard fan of another JSON library, you should be able to readily adapt the examples in this chapter. Let's create a `build.sbt` file with a dependency on json4s:

```
// build.sbt  
scalaVersion := "2.11.7"  
  
libraryDependencies += "org.json4s" %% "json4s-native" % "3.2.11"
```

We can then import json4s into an SBT console session with:

```
scala> import org.json4s._  
import org.json4s._  
  
scala> import org.json4s.native.JsonMethods._  
import org.json4s.native.JsonMethods._
```

Let's use json4s to parse the response to our GitHub API query:

```
scala> val jsonResponse = parse(response)  
jsonResponse: org.json4s.JValue = JObject(List((login,JString(odersky)),(  
id,JInt(795990))),...)
```

The `parse` method takes a string (that contains well-formatted JSON) and converts it to a `JValue`, a supertype for all json4s objects. The runtime type of the response to this particular query is `JObject`, which is a json4s type representing a JSON object.

`JObject` is a wrapper around a `List[JField]`, and `JField` represents an individual key-value pair in the object. We can use *extractors* to access this list:

```
scala> val JObject(fields) = jsonResponse  
fields: List[JField] = List((login,JString(odersky)),...)
```

What's happened here? By writing `val JObject(fields) = ...`, we are telling Scala:

- The right-hand side has runtime type of `JObject`
- Go into the `JObject` instance and bind the list of fields to the constant `fields`

Readers familiar with Python might recognize the similarity with tuple unpacking, though Scala extractors are much more powerful and versatile. Extractors are used extensively to extract Scala types from `json4s` types.

Pattern matching using case classes

How exactly does the Scala compiler know what to do with an extractor such as:



`val JObject(fields) = ...`

`JObject` is a case class with the following constructor:

```
case class JObject(obj: List[JField])
```

Case classes all come with an extractor that reverses the constructor exactly. Thus, writing `val JObject(fields)` will bind `fields` to the `obj` attribute of the `JObject`. For further details on how extractors work, read *Appendix, Pattern Matching and Extractors*.

We have now extracted `fields`, a (plain old Scala) list of fields from the `JObject`. A `JField` is a key-value pair, with the key being a string and value being a subtype of `JValue`. Again, we can use extractors to extract the values in the field:

```
scala> val firstField = fields.head
firstField: JField = (login,JString(odersky))
```

```
scala> val JField(key, JString(value)) = firstField
key: String = login
value: String = odersky
```

We matched the right-hand side against the pattern `JField(_, JString(_))`, binding the first element to `key` and the second to `value`. What happens if the right-hand side does not match the pattern?

```
scala> val JField(key, JInt(value)) = firstField
scala.MatchError: (login,JString(odersky)) (of class scala.Tuple2)
...
```

The code throws a `MatchError` at runtime. These examples demonstrate the power of nested pattern matching: in a single line, we managed to verify the type of `firstField`, that its value has type `JString`, and we have bound the key and value to the `key` and `value` variables, respectively. As another example, if we *know* that the first field is the `login` field, we can both verify this and extract the value:

```
scala> val JField("login", JString(loginName)) = firstField
loginName: String = odersky
```

Notice how this style of programming is *declarative* rather than imperative: we declare that we want a `JField("login", JString(_))` variable on the right-hand side. We then let the language figure out how to check the variable types. Pattern matching is a recurring theme in functional languages.

We can also use pattern matching in a for loop when looping over fields. When used in a for loop, a pattern match defines a *partial function*: only elements that match the pattern pass through the loop. This lets us filter the collection for elements that match a pattern and also apply a transformation to these elements. For instance, we can extract every string field in our `fields` list:

```
scala> for {
    JField(key, JString(value)) <- fields
} yield (key -> value)
List[(String, String)] = List((login,odersky), (avatar_url,https://
avatars.githubusercontent.com/...)
```

We can use this to search for specific fields. For instance, to extract the "followers" field:

```
scala> val followersList = for {
    JField("followers", JInt(followers)) <- fields
} yield followers
followersList: List[Int] = List(707)

scala> val followers = followersList.headOption
followers: Option[Int] = Some(707)
```

We first extracted all fields that matched the pattern `JField("follower", JInt(_))`, returning the integer inside the `JInt`. As the source collection, `fields`, is a list, this returns a list of integers. We then extract the first value from this list using `headOption`, which returns the head of the list if the list has at least one element, or `None` if the list is empty.

We are not limited to extracting a single field at a time. For instance, to extract the "id" and "login" fields together:

```
scala> {
  for {
    JField("login", JString(loginName)) <- fields
    JField("id", JInt(id)) <- fields
  } yield (id -> loginName)
}.headOption
Option[(BigInt, String)] = Some((795990,odersky))
```

Scala's pattern matching and extractors provide you with an extremely powerful way of traversing the json4s tree, extracting the fields that we need.

JSON4S types

We have already discovered parts of json4s's type hierarchy: strings are wrapped in `JString` objects, integers (or big integers) are wrapped in `JInt`, and so on. In this section, we will take a step back and formalize the type structure and what Scala types they extract to. These are the json4s runtime types:

- `val JString(s)` // => extracts to a `String`
- `val JDouble(d)` // => extracts to a `Double`
- `val JDecimal(d)` // => extracts to a `BigDecimal`
- `val JInt(i)` // => extracts to a `BigInt`
- `val JBool(b)` // => extracts to a `Boolean`
- `val JObject(l)` // => extracts to a `List[JField]`
- `val JArray(l)` // => extracts to a `List[JValue]`
- `JNull` // => represents a JSON null

All these types are subclasses of `JValue`. The compile-time result of `parse` is `JValue`, which you normally need to cast to a concrete type using an extractor.

The last type in the hierarchy is `JField`, which represents a key-value pair. `JField` is just a type alias for the `(String, JValue)` tuple. It is thus not a subtype of `JValue`. We can extract the key and value using the following extractor:

```
val JField(key, JInt(value)) = ...
```

Extracting fields using XPath

In the previous sections, you learned how to traverse JSON objects using extractors. In this section, we will look at a different way of traversing JSON objects and extracting specific fields: the *XPath DSL* (domain-specific language). XPath is a query language for traversing tree-like structures. It was originally designed for addressing specific nodes in an XML document, but it works just as well with JSON. We have already seen an example of XPath syntax when we extracted the stock price from the XML document returned by the "Markit on demand" API in *Chapter 4, Parallel Collections and Futures*. We extracted the node with tag "LastPrice" using `r \ "LastPrice"`. The \ operator was defined by the `scala.xml` package.

The `json4s` package exposes a similar DSL to extract fields from `JSONObject` instances. For instance, we can extract the "login" field from the JSON object `jsonResponse`:

```
scala> jsonResponse \ "login"
org.json4s.JValue = JString(odersky)
```

This returns a `JValue` that we can transform into a Scala string using an extractor:

```
scala> val JString(loginName) = jsonResponse \ "login"
loginName: String = odersky
```

Notice the similarity between the XPath DSL and traversing a filesystem: we can think of `JSONObject` instances as directories. Field names correspond to file names and the field value to the content of the file. This is more evident for nested structures. The `users` endpoint of the GitHub API does not have nested documents, so let's try another endpoint. We will query the API for the repository corresponding to this book: "<https://api.github.com/repos/pbugnion/s4ds>". The response has the following structure:

```
{
  "id": 42269470,
  "name": "s4ds",
  ...
  "owner": { "login": "pbugnion", "id": 1392879 ... }
  ...
}
```

Let's fetch this document and use the XPath syntax to extract the repository owner's login name:

```
scala> val jsonResponse = parse(Source.fromURL(
  "https://api.github.com/repos/pbugnion/s4ds"
).mkString)

jsonResponse: JValue = JObject(List((id,JInt(42269470)),
(name,JString(s4ds))...))

scala> val JString(ownerLogin) = jsonResponse \ "owner" \ "login"
ownerLogin: String = pbugnion
```

Again, this is much like traversing a filesystem: `jsonResponse \ "owner"` returns a `JObject` corresponding to the "owner" object. This `JObject` can, in turn, be queried for the "login" field, returning the value `JString(pbugnion)` associated with this key.

What if the API response is an array? The filesystem analogy breaks down somewhat. Let's query the API endpoint listing Martin Odersky's repositories: <https://api.github.com/users/odersky/repos>. The response is an array of JSON objects, each of which represents a repository:

```
[  
  {  
    "id": 17335228,  
    "name": "dotty",  
    "size": 14699,  
    ...  
  },  
  {  
    "id": 15053153,  
    "name": "frontend",  
    "size": 392  
    ...  
  },  
  {  
    "id": 2890092,  
    "name": "scala",  
    "size": 76133,  
    ...  
  },  
  ...  
]
```

Let's fetch this and parse it as JSON:

```
scala> val jsonResponse = parse(Source.fromURL(
  "https://api.github.com/users/odersky/repos"
).mkString)
jsonResponse: JValue = JArray(List(JObject(List((id,JInt(17335228)),
(name,Jstring(dotty)), ...
```

This returns a `JArray`. The XPath DSL works in the same way on a `JArray` as on a `JObject`, but now, instead of returning a single `JValue`, it returns an array of fields matching the path in every object in the array. Let's get the size of all Martin Odersky's repositories:

```
scala> jsonResponse \ "size"
JValue = JArray(List(JInt(14699), JInt(392), ...
```

We now have a `JArray` of the values corresponding to the `"size"` field in every repository. We can iterate over this array with a `for` comprehension and use extractors to convert elements to Scala objects:

```
scala> for {
  JInt(size) <- (jsonResponse \ "size")
} yield size
List[BigInt] = List(14699, 392, 76133, 32010, 98166, 1358, 144, 273)
```

Thus, combining extractors with the XPath DSL gives us powerful, complementary tools to extract information from JSON objects.

There is much more to the XPath syntax than we have space to cover here, including the ability to extract fields nested at any level of depth below the current root or fields that match a predicate or a certain type. We find that well-designed APIs obviate the need for many of these more powerful functions, but do consult the documentation (json4s.org) to get an overview of what you can do.

In the next section, we will look at extracting JSON directly into case classes.

Extraction using case classes

In the previous sections, we extracted specific fields from the JSON response using Scala extractors. We can do one better and extract full case classes.

When moving beyond the REPL, programming best practice dictates that we move from json4s types to Scala objects as soon as possible rather than passing json4s types around the program. Converting from json4s types to Scala types (or case classes representing domain objects) is good practice because:

- It decouples the program from the structure of the data that we receive from the API, something we have little control over.
- It improves type safety: a JObject is, as far as the compiler is concerned, always a JObject, whatever fields it contains. By contrast, the compiler will never mistake a User for a Repository.

Json4s lets us extract case classes directly from JObject instances, making writing the layer converting JObject instances to custom types easy.

Let's define a case class representing a GitHub user:

```
scala> case class User(id:Long, login:String)  
defined class User
```

To extract a case class from a JObject, we must first define an implicit Formats value that defines how simple types should be serialized and deserialized. We will use the default DefaultFormats provided with json4s:

```
scala> implicit val formats = DefaultFormats  
formats: DefaultFormats.type = DefaultFormats@750e685a
```

We can now extract instances of User. Let's do this for Martin Odersky:

```
scala> val url = "https://api.github.com/users/odersky"  
url: String = https://api.github.com/users/odersky  
  
scala> val jsonResponse = parse(Source.fromURL(url).mkString)  
jsonResponse: JValue = JObject(List((login,JString(odersky)), ...  
  
scala> jsonResponse.extract[User]  
User = User(795990,odersky)
```

This works as long as the object is well-formatted. The extract method looks for fields in the JObject that match the attributes of User. In this case, extract will note that the JObject contains the "login": "odersky" field and that JString("odersky") can be converted to a Scala string, so it binds "odersky" to the login attribute in User.

What if the attribute names differ from the field names in the JSON object? We must first transform the object to have the correct fields. For instance, let's rename the `login` attribute to `userName` in our `User` class:

```
scala> case class User(id:Long, userName:String)  
defined class User
```

If we try to use `extract [User]` on `jsonResponse`, we will get a mapping error because the deserializer is missing a `login` field in the response. We can fix this using the `transformField` method on `jsonResponse` to rename the `login` field:

```
scala> jsonResponse.transformField {  
    case("login", n) => "userName" -> n  
}.extract[User]  
User = User(795990,odersky)
```

What about optional fields? Let's assume that the JSON object returned by the GitHub API does not always contain the `login` field. We could symbolize this in our object model by giving the `login` parameter the type `Option[String]` rather than `String`:

```
scala> case class User(id:Long, login:Option[String])  
defined class User
```

This works just as you would expect. When the response contains a non-null `login` field, calling `extract [User]` will deserialize it to `Some(value)`, and when it's missing or `NULL`, it will produce `None`:

```
scala> jsonResponse.extract[User]  
User = User(795990,Some(odersky))  
  
scala> jsonResponse.removeField {  
    case(k, _) => k == "login" // remove the "login" field  
}.extract[User]  
User = User(795990,None)
```

Let's wrap this up in a small program. The program will take a single command-line argument, the user's login name, extract a `User` instance, and print it to screen:

```
// GitHubUser.scala  
  
import scala.io._  
import org.json4s._
```

```
import org.json4s.native.JsonMethods._

object GitHubUser {

    implicit val formats = DefaultFormats

    case class User(id:Long, userName:String)

    /** Query the GitHub API corresponding to `url`
     * and convert the response to a User.
     */
    def fetchUserFromUrl(url:String):User = {
        val response = Source.fromURL(url).mkString
        val jsonResponse = parse(response)
        extractUser(jsonResponse)
    }

    /** Helper method for transforming the response to a User */
    def extractUser(obj:JValue):User = {
        val transformedObject = obj.transformField {
            case ("login", name) => ("userName", name)
        }
        transformedObject.extract[User]
    }

    def main(args:Array[String]) {
        // Extract username from argument list
        val name = args.headOption.getOrElse {
            throw new IllegalArgumentException(
                "Missing command line argument for user.")
        }

        val user = fetchUserFromUrl(
            s"https://api.github.com/users/$name")

        println(s"** Extracted for $name:")
        println()
        println(user)

    }
}
```

We can run this from an SBT console as follows:

```
$ sbt  
> runMain GitHubUser pbugnion  
** Extracted for pbugnion:  
User(1392879,pbugnion)
```

Concurrency and exception handling with futures

While the program that we wrote in the previous section works, it is very brittle. It will crash if we enter a non-existent user name or the GitHub API changes or returns a badly-formatted response. We need to make it fault-tolerant.

What if we also wanted to fetch multiple users? The program, as written, is entirely single-threaded. The `fetchUserFromUrl` method fires a call to the API and blocks until the API sends data back. A better solution would be to fetch multiple users in parallel.

As you learned in *Chapter 4, Parallel Collections and Futures*, there are two straightforward ways to implement both fault tolerance and parallel execution: we can either put all the user names in a parallel collection and wrap the code for fetching and extracting the user in a `Try` block or we can wrap each query in a future.

When querying web APIs, it is sometimes the case that a request can take abnormally long. To prevent this from blocking the other threads, it is preferable to rely on futures rather than parallel collections for concurrency, as we saw in the *Parallel collection or Future?* section at the end of *Chapter 4, Parallel Collections and Futures*.

Let's rewrite the code from the previous section to handle fetching multiple users concurrently in a fault-tolerant manner. We will change the `fetchUserFromUrl` method to query the API asynchronously. This is not terribly different from *Chapter 4, Parallel Collections and Futures*, in which we queried the "Markit on demand" API:

```
// GitHubUserConcurrent.scala  
  
import scala.io._  
import scala.concurrent._  
import scala.concurrent.duration._  
import ExecutionContext.Implicits.global  
import scala.util._  
  
import org.json4s._
```

```
import org.json4s.native.JsonMethods._

object GitHubUserConcurrent {

    implicit val formats = DefaultFormats

    case class User(id:Long, userName:String)

    // Fetch and extract the `User` corresponding to `url`
    def fetchUserFromUrl(url:String) :Future[User] = {
        val response = Future { Source.fromURL(url).mkString }
        val parsedResponse = response.map { r => parse(r) }
        parsedResponse.map { extractUser }
    }

    // Helper method for extracting a user from a JObject
    def extractUser(jsonResponse:JValue):User = {
        val o = jsonResponse.transformField {
            case ("login", name) => ("userName", name)
        }
        o.extract[User]
    }

    def main(args:Array[String]) {
        val names = args.toList

        // Loop over each username and send a request to the API
        // for that user
        val name2User = for {
            name <- names
            url = s"https://api.github.com/users/$name"
            user = fetchUserFromUrl(url)
        } yield name -> user

        // callback function
        name2User.foreach { case(name, user) =>
            user.onComplete {
                case Success(u) => println(s" ** Extracted for $name: $u")
                case Failure(e) => println(s" ** Error fetching $name:
                    $e")
            }
        }

        // Block until all the calls have finished.
    }
}
```

```
Await.ready(Future.sequence(name2User.map { _._2 }), 1 minute)
    }
}
```

Let's run the code through sbt:

```
$ sbt
> runMain GitHubUserConcurrent odersky derekwyatt not-a-user-675
** Error fetching user not-a-user-675: java.io.FileNotFoundException:
https://api.github.com/users/not-a-user-675
** Extracted for odersky: User(795990,odersky)
** Extracted for derekwyatt: User(62324,derekwyatt)
```

The code itself should be straightforward. All the concepts used here have been explored in this chapter or in *Chapter 4, Parallel Collections and Futures*, apart from the last line:

```
Await.ready(Future.sequence(name2User.map { _._2 }), 1 minute)
```

This statement tells the program to wait until all futures in our list have been completed. `Await.ready(..., 1 minute)` takes a future as its first argument and blocks execution until this future returns. The second argument is a time-out on this future. The only catch is that we need to pass a single future to `Await` rather than a list of futures. We can use `Future.sequence` to merge a collection of futures into a single future. This future will be completed when all the futures in the sequence have completed.

Authentication – adding HTTP headers

So far, we have been using the GitHub API without authentication. This limits us to sixty requests per hour. Now that we can query the API in parallel, we could exceed this limit in seconds.

Fortunately, GitHub is much more generous if you authenticate when you query the API. The limit increases to 5,000 requests per hour. You must have a GitHub user account to authenticate, so go ahead and create one now if you need to. After creating an account, navigate to <https://github.com/settings/tokens> and click on the **Generate new token** button. Accept the default settings and enter a token description and a long hexadecimal number should appear on the screen. Copy the token for now.

HTTP – a whirlwind overview

Before using our newly generated token, let's take a few minutes to review how HTTP works.

HTTP is a protocol for transferring information between different computers. It is the protocol that we have been using throughout the chapter, though Scala hid the details from us in the call to `Source.fromURL`. It is also the protocol that you use when you point your web browser to a website, for instance.

In HTTP, a computer will typically make a *request* to a remote server, and the server will send back a *response*. Requests contain a *verb*, which defines the type of request, and a URL identifying a *resource*. For instance, when we typed `api.github.com/users/pbugnion` in our browsers, this was translated into a GET (the verb) request for the `users/pbugnion` resource. All the calls that we have made so far have been GET requests. You might use a different type of request, for instance, a POST request, to modify (rather than just view) some content on GitHub.

Besides the verb and resource, there are two more parts to an HTTP request:

- The *headers* include metadata about the request, such as the expected format and character set of the response or the authentication credentials. Headers are just a list of key-value pairs. We will pass the OAuth token that we have just generated to the API using the `Authorization` header. This Wikipedia article lists commonly used header fields: en.wikipedia.org/wiki/List_of_HTTP_header_fields.
- The request body is not used in GET requests but becomes important for requests that modify the resource they query. For instance, if I wanted to create a new repository on GitHub programmatically, I would send a POST request to `/pbugnion/repos`. The POST body would then be a JSON object describing the new repository. We will not use the request body in this chapter.

Adding headers to HTTP requests in Scala

We will pass the OAuth token as a header with our HTTP request. Unfortunately, the `Source.fromURL` method is not particularly suited to adding headers when creating a GET request. We will, instead, use a library, `scalaj-http`.

Let's add `scalaj-http` to the dependencies in our `build.sbt`:

```
libraryDependencies += "org.scalaj" %% "scalaj-http" % "1.1.6"
```

We can now import scalaj-http:

```
scala> import scalaj.http._  
import scalaj.http._
```

We start by creating an `HttpRequest` object:

```
scala> val request = Http("https://api.github.com/users/pbugnion")  
request:scalaj.http.HttpRequest = HttpRequest(api.github.com/users/  
pbugnion,GET,...)
```

We can now add the authorization header to the request (add your own token string here):

```
scala> val authorizedRequest = request.header("Authorization", "token  
e836389ce ...")  
authorizedRequest:scalaj.http.HttpRequest = HttpRequest(api.github.com/  
users/pbugnion,GET,...)
```



The `.header` method returns a new `HttpRequest` instance. It does not modify the request in place. Thus, just calling `request.header(...)` does not actually add the header to `request` itself, which can be a source of confusion.

Let's fire the request. We do this through the request's `asString` method, which queries the API, fetches the response, and parses it as a Scala String:

```
scala> val response = authorizedRequest.asString  
response:scalaj.http.HttpResponse[String] = HttpResponse({"login":"pbugni  
on",...})
```

The response is made up of three components:

- The status code, which should be 200 for a successful request:

```
scala> response.code  
Int = 200
```

- The response body, which is the part that we are interested in:

```
scala> response.body  
String = {"login":"pbugnion","id":1392879,...}
```

- The response headers (metadata about the response):

```
scala> response.headers  
Map[String,String] = Map(Access-Control-Allow-Credentials -> true,  
...)
```

To verify that the authorization was successful, query the `X-RateLimit-Limit` header:

```
scala> response.headers("X-RateLimit-Limit")
String = 5000
```

This value is the maximum number of requests per hour that you can make to the GitHub API from a single IP address.

Now that we have some understanding of how to add authentication to GET requests, let's modify our script for fetching users to use the OAuth token for authentication. We first need to import `scalaj-http`:

```
import scalaj.http._
```

Injecting the value of the token into the code can be somewhat tricky. You might be tempted to hardcode it, but this prohibits you from sharing the code. A better solution is to use an *environment variable*. Environment variables are a set of variables present in your terminal session that are accessible to all processes running in that session. To get a list of the current environment variables, type the following on Linux or Mac OS:

```
$ env
HOME=/Users/pascal
SHELL=/bin/zsh
...
...
```

On Windows, the equivalent command is `SET`. Let's add the GitHub token to the environment. Use the following command on Mac OS or Linux:

```
$ export GHTOKEN="e83638..." # enter your token here
```

On Windows, use the following command:

```
$ SET GHTOKEN="e83638..."
```

If you were to reuse this environment variable across many projects, entering `export GHTOKEN=...` in the shell for every session gets old quite quickly. A more permanent solution is to add `export GHTOKEN="e83638..."` to your shell configuration file (your `.bashrc` file if you are using Bash). This is safe provided your `.bashrc` is readable by the user only. Any new shell session will have access to the `GHTOKEN` environment variable.

We can access environment variables from a Scala program using `sys.env`, which returns a `Map[String, String]` of the variables. Let's add a `lazy val` token to our class, containing the token value:

```
lazy val token:Option[String] = sys.env.get("GHTOKEN") orElse {
    println("No token found: continuing without authentication")
    None
}
```

Now that we have the token, the only part of the code that must change, to add authentication, is the `fetchUserFromUrl` method:

```
def fetchUserFromUrl(url:String):Future[User] = {
    val baseRequest = Http(url)
    val request = token match {
        case Some(t) => baseRequest.header(
            "Authorization", s"token $t")
        case None => baseRequest
    }
    val response = Future {
        request.asString.body
    }
    val parsedResponse = response.map { r => parse(r) }
    parsedResponse.map(extractUser)
}
```

Additionally, we can, to gain clearer error messages, check that the response's status code is 200. As this is straightforward, it is left as an exercise.

Summary

In this chapter, you learned how to query the GitHub API, converting the response to Scala objects. Of course, merely printing results to screen is not terribly interesting. In the next chapter, we will look at the next step of the data ingestion process: storing data in a database. We will query the GitHub API and store the results in a MongoDB database.

In *Chapter 13, Web APIs with Play*, we will look at building our own simple web API.

References

The GitHub API, with its extensive documentation, is a good place to explore how a rich API is constructed. It has a **Getting Started** section that is worth reading:

<https://developer.github.com/guides/getting-started/>

Of course, this is not specific to Scala: it uses cURL to query the API.

Read the documentation (<http://json4s.org>) and source code (<https://github.com/json4s/json4s>) for json4s for a complete reference. There are many parts of this package that we have not explored, in particular, how to build JSON from Scala.

8

Scala and MongoDB

In *Chapter 5, Scala and SQL through JDBC*, and *Chapter 6, Slick – A Functional Interface for SQL*, you learned how to insert, transform, and read data in SQL databases. These databases remain (and are likely to remain) very popular in data science, but NoSQL databases are emerging as strong contenders.

The needs for data storage are growing rapidly. Companies are producing and storing more data points in the hope of acquiring better business intelligence. They are also building increasingly large teams of data scientists, who all need to access the data store. Maintaining constant access time as the data load increases requires taking advantage of parallel architectures: we need to distribute the database across several computers so that, as the load on the server increases, we can just add more machines to improve throughput.

In MySQL databases, the data is naturally split across different tables. Complex queries necessitate joining across several tables. This makes partitioning the database across different computers difficult. NoSQL databases emerged to fill this gap.

In this chapter, you will learn to interact with MongoDB, an open source database that offers high performance and can be distributed easily. MongoDB is one of the more popular NoSQL databases with a strong community. It offers a reasonable balance of speed and flexibility, making it a natural alternative to SQL for storing large datasets with uncertain query requirements, as might happen in data science. Many of the concepts and recipes in this chapter will apply to other NoSQL databases.

MongoDB

MongoDB is a *document-oriented* database. It contains collections of documents. Each document is a JSON-like object:

```
{  
    _id: ObjectId("558e846730044ede70743be9") ,  
    name: "Gandalf" ,  
    age: 2000 ,  
    pseudonyms: [ "Mithrandir" , "Olorin" , "Greyhame" ] ,  
    possessions: [  
        { name: "Glamdring" , type: "sword" } ,  
        { name: "Narya" , type: "ring" }  
    ]  
}
```

Just as in JSON, a document is a set of key-value pairs, where the values can be strings, numbers, Booleans, dates, arrays, or subdocuments. Documents are grouped in collections, and collections are grouped in databases.

You might be thinking that this is not very different from SQL: a document is similar to a row and a collection corresponds to a table. There are two important differences:

- The values in documents can be simple values, arrays, subdocuments, or arrays of subdocuments. This lets us encode one-to-many and many-to-many relationships in a single collection. For instance, consider the wizard collection. In SQL, if we wanted to store pseudonyms for each wizard, we would have to use a separate `wizard2pseudonym` table with a row for each wizard-pseudonym pair. In MongoDB, we can just use an array. In practice, this means that we can normally use a single document to represent an entity (a customer, transaction, or wizard, for instance). In SQL, we would normally have to join across several tables to retrieve all the information on a specific entity.
- MongoDB is *schemaless*. Documents in a collection can have varying sets of fields with different types for the same field across different documents. In practice, MongoDB collections have a loose schema enforced either client side or by convention: most documents will have a subset of the same fields, and fields will, in general, contain the same data type. Having a flexible schema makes adjusting the data structure easy as there is no need for time-consuming `ALTER TABLE` statements. The downside is that there is no easy way of enforcing our flexible schema on the database side.

Note the `_id` field: this is a unique key. MongoDB will generate one automatically if we insert a document without an `_id` field.

This chapter gives recipes for interacting with a MongoDB database from Scala, including maintaining type safety and best practices. We will not cover advanced MongoDB functionality (such as aggregation or distributing the database). We will assume that you have MongoDB installed on your computer (<http://docs.mongodb.org/manual/installation/>). It will also help to have a very basic knowledge of MongoDB (we discuss some references at the end of this chapter, but any basic tutorial available online will be sufficient for the needs of this chapter).

Connecting to MongoDB with Casbah

The official MongoDB driver for Scala is called **Casbah**. Rather than a fully-fledged driver, Casbah wraps the Java Mongo driver, providing a more functional interface. There are other MongoDB drivers for Scala, which we will discuss briefly at the end of this chapter. For now, we will stick to Casbah.

Let's start by adding Casbah to our `build.sbt` file:

```
scalaVersion := "2.11.7"

libraryDependencies += "org.mongodb" %% "casbah" % "3.0.0"
```

Casbah also expects `slf4j` bindings (a Scala logging framework) to be available, so let's also add `slf4j-nop`:

```
libraryDependencies += "org.slf4j" % "slf4j-nop" % "1.7.12"
```

We can now start an SBT console and import Casbah in the Scala shell:

```
$ sbt console
scala> import com.mongodb.casbah.Imports._
import com.mongodb.casbah.Imports._

scala> val client = MongoClient()
client: com.mongodb.casbah.MongoClient = com.mongodb.casbah.
MongoClient@4ac17318
```

This connects to a MongoDB server on the default host (`localhost`) and default port (27017). To connect to a different server, pass the host and port as arguments to `MongoClient`:

```
scala> val client = MongoClient("192.168.1.1", 27017)
client: com.mongodb.casbah.MongoClient = com.mongodb.casbah.
MongoClient@584c6b02
```

Note that creating a client is a lazy operation: it does not attempt to connect to the server until it needs to. This means that if you enter the wrong URL or password, you will not know about it until you try and access documents on the server.

Once we have a connection to the server, accessing a database is as simple as using the client's `apply` method. For instance, to access the `github` database:

```
scala> val db = client("github")
db: com.mongodb.casbah.MongoDB = DB{name='github'}
```

We can then access the "users" collection:

```
scala> val coll = db("users")
coll: com.mongodb.casbah.MongoCollection = users
```

Connecting with authentication

MongoDB supports several different authentication mechanisms. In this section, we will assume that your server is using the **SCRAM-SHA-1** mechanism, but you should find adapting the code to a different type of authentication straightforward.

The easiest way of authenticating is to pass `username` and `password` in the URI when connecting:

```
scala> val username = "USER"
username: String = USER

scala> val password = "PASSWORD"
password: String = PASSWORD

scala> val uri = MongoClientURI(
  "mongodb://$username:$password@localhost/?authMechanism=SCRAM-SHA-1"
)
uri: MongoClientURI = mongodb://USER:PASSWORD@
localhost/?authMechanism=SCRAM-SHA-1

scala> val mongoClient = MongoClient(uri)
client: com.mongodb.casbah.MongoClient = com.mongodb.casbah.
MongoClient@4ac17318
```

In general, you will not want to put your password in plain text in the code. You can either prompt for a password on the command line or pass it through environment variables, as we did with the GitHub OAuth token in *Chapter 7, Web APIs*. The following code snippet demonstrates how to pass credentials through the environment:

```
// Credentials.scala

import com.mongodb.casbah.Imports._

object Credentials extends App {

    val username = sys.env.getOrElse("MONGouser",
        throw new IllegalStateException(
            "Need a MONGouser variable in the environment")
    )
    val password = sys.env.getOrElse("Mongopassword",
        throw new IllegalStateException(
            "Need a MONGOPASSWORD variable in the environment")
    )

    val host = "127.0.0.1"
    val port = 27017

    val uri = s"mongodb:
        // $username:$password@$host:$port/?authMechanism=SCRAM-SHA-1"

    val client = MongoClient(MongoClientURI(uri))
}
```

You can run it through SBT as follows:

```
$ MONGouser="pascal" MONGOPASSWORD="scalarulez" sbt
> runMain Credentials
```

Inserting documents

Let's insert some documents into our newly created database. We want to store information about GitHub users, using the following document structure:

```
{  
    id: <mongodb object id>,  
    login: "pbugnion",  
    github_id: 1392879,  
    repos: [  
        {  
            name: "scikit-monaco",  
            id: 14821551,  
            language: "Python"  
        },  
        {  
            name: "contactpp",  
            id: 20448325,  
            language: "Python"  
        }  
    ]  
}
```

Casbah provides a `DBObject` class to represent MongoDB documents (and subdocuments) in Scala. Let's start by creating a `DBObject` instance for each repository subdocument:

```
scala> val repo1 = DBObject("name" -> "scikit-monaco", "id" -> 14821551,  
"language" -> "Python")  
repo1: DBObject = { "name" : "scikit-monaco" , "id" : 14821551,  
"language" : "Python"}
```

As you can see, a `DBObject` is just a list of key-value pairs, where the keys are strings. The values have compile-time type `AnyRef`, but Casbah will fail (at runtime) if you try to add a value that cannot be serialized.

We can also create `DBObject` instances from lists of key-value pairs directly. This is particularly useful when converting from a Scala map to a `DBObject`:

```
scala> val fields:Map[String, Any] = Map(  
    "name" -> "contactpp",  
    "id" -> 20448325,  
    "language" -> "Python"  
)
```

```
Map[String, Any] = Map(name -> contactpp, id -> 20448325, language ->
Python)

scala> val repo2 = DBObject(fields.toList)
repo2: dDBObject = { "name" : "contactpp" , "id" : 20448325, "language" :
"Python" }
```

The DBObject class provides many of the same methods as a map. For instance, we can address individual fields:

```
scala> repol("name")
AnyRef = scikit-monaco
```

We can construct a new object by adding a field to an existing object:

```
scala> repol + ("fork" -> true)
mutable.Map[String,Any] = { "name" : "scikit-monaco" , "id" : 14821551,
"language" : "python", "fork" : true}
```

Note the return type: `mutable.Map[String,Any]`. Rather than implementing methods such as `+` directly, Casbah adds them to `DBObject` by providing an implicit conversion to and from `mutable.Map`.

New `DBObject` instances can also be created by concatenating two existing instances:

```
scala> repol ++ DBObject(
  "locs" -> 6342,
  "description" -> "Python library for Monte Carlo integration"
)
DBObject = { "name" : "scikit-monaco" , "id" : 14821551, "language" :
"Python", "locs" : 6342 , "description" : "Python library for Monte Carlo
integration" }
```

`DBObject` instances can then be inserted into a collection using the `+=` operator. Let's insert our first document into the `user` collection:

```
scala> val userDocument = DBObject(
  "login" -> "pbugnion",
  "github_id" -> 1392879,
  "repos" -> List(repo1, repo2)
)
```

```
userDocument:DBObject = { "login" : "pbugnion" , ... }

scala> val coll = MongoClient()("github")("users")
coll: com.mongodb.casbah.MongoCollection = users

scala> coll += userDocument
com.mongodb.casbah.TypeImports.WriteResult = WriteResult{, n=0,
updateOfExisting=false, upsertedId=null}
```

A database containing a single document is a bit boring, so let's add a few more documents queried directly from the GitHub API. You learned how to query the GitHub API in the previous chapter, so we won't dwell on how to do this here.

In the code examples for this chapter, we have provided a class called `GitHubUserIterator` that queries the GitHub API (specifically the `/users` endpoint) for user documents, converts them to a case class, and offers them as an iterator. You will find the class in the code examples for this chapter (available on GitHub at <https://github.com/pbugnion/s4ds/tree/master/chap08>) in the `GitHubUserIterator.scala` file. The easiest way to have access to the class is to open an SBT console in the directory of the code examples for this chapter. The API then fetches users in increasing order of their login ID:

```
scala> val it = new GitHubUserIterator
it: GitHubUserIterator = non-empty iterator

scala> it.next // Fetch the first user
User = User(mojombo,1,List(Repo(...
```

`GitHubUserIterator` returns instances of the `User` case class, defined as follows:

```
// User.scala
case class User(login:String, id:Long, repos>List[Repo])

// Repo.scala
case class Repo(name:String, id:Long, language:String)
```

Let's write a short program to fetch 500 users and insert them into the MongoDB database. We will need to authenticate with the GitHub API to retrieve these users. The constructor for `GitHubUserIterator` takes the GitHub OAuth token as an optional argument. We will inject the token through the environment, as we did in the previous chapter.

We first give the entire code listing before breaking it down—if you are typing this out, you will need to copy `GitHubUserIterator.scala` from the code examples for this chapter to the directory in which you are running this to access the `GitHubUserIterator` class. The class relies on `scalaj-http` and `json4s`, so either copy the `build.sbt` file from the code examples or specify those packages as dependencies in your `build.sbt` file.

```
// InsertUsers.scala

import com.mongodb.casbah.Imports._

object InsertUsers {

    /** Function for reading GitHub token from environment. */
    lazy val token:Option[String] = sys.env.get("GHTOKEN")orElse {
        println("No token found: continuing without authentication")
        None
    }

    /** Transform a Repo instance to a DBObject */
    def repoToDBObject(repo:Repo):DBObject = DBObject(
        "github_id" -> repo.id,
        "name" -> repo.name,
        "language" -> repo.language
    )

    /** Transform a User instance to a DBObject */
    def userToDBObject(user:User):DBObject = DBObject(
        "github_id" -> user.id,
        "login" -> user.login,
        "repos" -> user.repos.map(repoToDBObject)
    )

    /** Insert a list of users into a collection. */
    def insertUsers(coll:MongoCollection)(users:Iterable[User]) {
        users.foreach { user => coll += userToDBObject(user) }
    }

    /** Fetch users from GitHub and passes them to `inserter` */
    def ingestUsers(nusers:Int)(inserter:Iterable[User] => Unit) {
        val it = new GitHubUserIterator(token)
        val users = it.take(nusers).toList
        inserter(users)
    }
}
```

```
}

def main(args:Array[String]) {
    val coll = MongoClient()("github")("users")
    val nusers = 500
    coll.dropCollection()
    val inserter = insertUsers(coll)_
    ingestUsers(inserter)(nusers)
}

}
```

Before diving into the details of how this program works, let's run it through SBT. You will want to query the API with authentication to avoid hitting the rate limit. Recall that we need to set the `GHTOKEN` environment variable:

```
$ GHTOKEN="e83638..." sbt
$ runMain InsertUsers
```

The program will take about five minutes to run (depending on your Internet connection). To verify that the program works, we can query the number of documents in the `users` collection of the `github` database:

```
$ mongo github --quiet --eval "db.users.count()"
500
```

Let's break the code down. We first load the OAuth token to authenticate with the GitHub API. The token is stored as an environment variable, `GHTOKEN`. The `token` variable is a `lazy val`, so the token is loaded only when we formulate the first request to the API. We have already used this pattern in *Chapter 7, Web APIs*.

We then define two methods to transform from classes in the domain model to `DBObject` instances:

```
def repoToDBObject(repo:Repo):DBObject = ...
def userToDBObject(user:User):DBObject = ...
```

Armed with these two methods, we can add `users` to our MongoDB collection easily:

```
def insertUsers(coll:MongoCollection)(users:Iterable[User]):Unit = {
    users.foreach { user => coll += userToDBObject(user) }
}
```

We used currying to split the arguments of `insertUsers`. This lets us use `insertUsers` as a function factory:

```
val inserter = insertUsers(coll)_
```

This creates a new method, `inserter`, with signature `Iterable[User] => Unit` that inserts users into `coll`. To see how this might come in useful, let's write a function to wrap the whole data ingestion process. This is how a first attempt at this function could look:

```
def ingestUsers(nusers:Int)(inserter:Iterable[User] => Unit) {  
    val it = new GitHubUserIterator(token)  
    val users = it.take(nusers).toList  
    inserter(users)  
}
```

Notice how `ingestUsers` takes a method that specifies how the list of users is inserted into the database as its second argument. This function encapsulates the entire code specific to insertion into a MongoDB collection. If we decide, at some later date, that we hate MongoDB and must insert the documents into a SQL database or write them to a flat file, all we need to do is pass a different `inserter` function to `ingestUsers`. The rest of the code remains the same. This demonstrates the increased flexibility afforded by using higher-order functions: we can easily build a framework and let the client code plug in the components that it needs.

The `ingestUsers` method, as defined previously, has one problem: if the `nusers` value is large, it will consume a lot of memory in constructing the entire list of users. A better solution would be to break it down into batches: we fetch a batch of users from the API, insert them into the database, and move on to the next batch. This allows us to control memory usage by changing the batch size. It is also more fault tolerant: if the program crashes, we can just restart from the last successfully inserted batch.

The `.grouped` method, available on all iterables, is useful for batching. It returns an iterator over fragments of the original iterable:

```
scala> val it = (0 to 10)  
it: Range.Inclusive = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
  
scala> it.grouped(3).foreach { println } // In batches of 3  
Vector(0, 1, 2)  
Vector(3, 4, 5)  
Vector(6, 7, 8)  
Vector(9, 10)
```

Let's rewrite our `ingestUsers` method to use batches. We will also add a progress report after each batch in order to give the user some feedback:

```
/** Fetch users from GitHub and pass them to `inserter` */  
def ingestUsers(nusers:Int)(inserter:Iterable[User] => Unit) {
```

```
val batchSize = 100
val it = new GitHubUserIterator(token)
print("Inserted #users: ")
it.take(nusers).grouped(batchSize).zipWithIndex.foreach {
    case (users, batchNumber) =>
        print(s"${batchNumber*batchSize} ")
        inserter(users)
}
println()
```

Let's look at the highlighted line more closely. We start from the user iterator, `it`. We then take the first `nusers`. This returns an `Iterator[User]` that, instead of happily churning through every user in the GitHub database, will terminate after `nusers`. We then group this iterator into batches of 100 users. The `.grouped` method returns `Iterator[Iterator[User]]`. We then zip each batch with its index so that we know which batch we are currently processing (we use this in the `print` statement). The `.zipWithIndex` method returns `Iterator[(Iterator[User], Int)]`. We unpack this tuple in the loop using a case statement that binds `users` to `Iterator[User]` and `batchNumber` to the index. Let's run this through SBT:

```
$ GHTOKEN="2502761..." sbt
> runMain InsertUsers
[info] Running InsertUsers
Inserted #users: 0 100 200 300 400
[success] Total time: 215 s, completed 01-Nov-2015 18:44:30
```

Extracting objects from the database

We now have a database populated with a few users. Let's query this database from the REPL:

```
scala> import com.mongodb.casbah.Imports._
import com.mongodb.casbah.Imports._

scala> val collection = MongoClient()("github")("users")
MongoCollection = users

scala> val maybeUser = collection.findOne
Option[collection.T] = Some({ "_id" : { "$oid" :
"562e922546f953739c43df02" } , "github_id" : 1 , "login" : "mojombo" ,
"repos" : ... }
```

The `findOne` method returns a single `DBObject` object wrapped in an option, unless the collection is empty, in which case it returns `None`. We must therefore use the `get` method to extract the object:

```
scala> val user = maybeUser.get
collection.T = { "_id" : { "$oid" : "562e922546f953739c43df02" } ,
"github_id" : 1 , "login" : "mojombo" , "repos" : ... }
```

As you learned earlier in this chapter, `DBObject` is a map-like object with keys of type `String` and values of type `AnyRef`:

```
scala> user("login")
AnyRef = mojombo
```

In general, we want to restore compile-time type information as early as possible when importing objects from the database: we do not want to pass `AnyRefs` around when we can be more specific. We can use the `getAs` method to extract a field and cast it to a specific type:

```
scala> user.getAs[String]("login")
Option[String] = Some(mojombo)
```

If the field is missing in the document or if the value cannot be cast, `getAs` will return `None`:

```
scala> user.getAs[Int]("login")
Option[Int] = None
```

The astute reader may note that the interface provided by `getAs[T]` is similar to the `read[T]` method that we defined on a JDBC result set in *Chapter 5, Scala and SQL through JDBC*.

If `getAs` fails (for instance, because the field is missing), we can use the `orElse` partial function to recover:

```
scala> val loginName = user.getAs[String]("login") orElse {
  println("No login field found. Falling back to 'name'")
  user.getAs[String]("name")
}
loginName: Option[String] = Some(mojombo)
```

The `getAsOrElse` method allows us to substitute a default value if the cast fails:

```
scala> user.getAsOrElse[Int]("id", 5)
Int = 1392879
```

Note that we can also use `getAsOrElse` to throw an exception:

```
scala> user.getAsOrElse[String] ("name",
  throw new IllegalArgumentException(
    "Missing value for name")
)
java.lang.IllegalArgumentException: Missing value for name
...
```

Arrays embedded in documents can be cast to `List[T]` objects, where `T` is the type of elements in the array:

```
scala> user.getAsOrElse[List[DBObject]] ("repos",
  List.empty[DBObject])
List[DBObject] = List({ "github_id" : 26899533 , "name" :
"30daysoflaptops.github.io" ...}
```

Retrieving a single document at a time is not very useful. To retrieve all the documents in a collection, use the `.find` method:

```
scala> val userIterator = collection.find()
userIterator: collection.CursorType = non-empty iterator
```

This returns an iterator of `DBObject`s. To actually fetch the documents from the database, you need to materialize the iterator by transforming it into a collection, using, for instance, `.toList`:

```
scala> val userList = userIterator.toList
List[DBObject] = List({ "_id" : { "$oid": ...
```

Let's bring all of this together. We will write a toy program that prints the average number of repositories per user in our collection. The code works by fetching every document in the collection, extracting the number of repositories from each document, and then averaging over these:

```
// RepoNumber.scala

import com.mongodb.casbah.Imports._

object RepoNumber {

  /** Extract the number of repos from a DBObject
   * representing a user.
   */
  def extractNumber(obj:DBObject):Option[Int] = {
```

```

val repos = obj.getAs[List[DBObject]]("repos") orElse {
    println("Could not find or parse 'repos' field")
    None
}
repos.map { _.size }
}

val collection = MongoClient()("github")("users")

def main(args:Array[String]) {
    val userIterator = collection.find()

    // Convert from documents to Option[Int]
    val repoNumbers = userIterator.map { extractNumber }

    // Convert from Option[Int] to Int
    val wellFormattedNumbers = repoNumbers.collect {
        case Some(v) => v
    }.toList

    // Calculate summary statistics
    val sum = wellFormattedNumbers.reduce { _ + _ }
    val count = wellFormattedNumbers.size

    if (count == 0) {
        println("No repos found")
    }
    else {
        val mean = sum.toDouble / count.toDouble
        println(s"Total number of users with repos: $count")
        println(s"Total number of repos: $sum")
        println(s"Mean number of repos: $mean")
    }
}
}

```

Let's run this through SBT:

```

> runMain RepoNumber
Total number of users with repos: 500
Total number of repos: 9649
Mean number of repos: 19.298

```

The code starts with the `extractNumber` function, which extracts the number of repositories from each `DBObject`. The return value is `None` if the document does not contain the `repos` field.

The main body of the code starts by creating an iterator over `DBObject`s in the collection. This iterator is then mapped through the `extractNumber` function, which transforms it into an iterator of `Option[Int]`. We then run `.collect` on this iterator to collect all the values that are not `None`, converting from `Option[Int]` to `Int` in the process. Only then do we materialize the iterator to a list using `.toList`. The resulting list, `wellFormattedNumbers`, has the `List[Int]` type. We then just take the mean of this list and print it to screen.

Note that, besides the `extractNumber` function, none of this program deals with Casbah-specific types: the iterator returned by `.find()` is just a Scala iterator. This makes Casbah straightforward to use: the only data type that you need to familiarize yourself with is `DBObject` (compare this with JDBC's `ResultSet`, which we had to explicitly wrap in a stream, for instance).

Complex queries

We now know how to convert `DBObject` instances to custom Scala classes. In this section, you will learn how to construct queries that only return a subset of the documents in the collection.

In the previous section, you learned to retrieve all the documents in a collection as follows:

```
scala> val objs = collection.find().toList
List[DBObject] = List({ "_id" : { "$oid" : "56365cec46f9534fae8ffd7f" }
,...
```

The `collection.find()` method returns an iterator over all the documents in the collection. By calling `.toList` on this iterator, we materialize it to a list.

We can customize which documents are returned by passing a query document to the `.find` method. For instance, we can retrieve documents for a specific login name:

```
scala> val query = DBObject("login" -> "mojombo")
query: DBObject = { "login" : "mojombo" }

scala> val objs = collection.find(query).toList
List[DBObject] = List({ "_id" : { "$oid" : "562e922546f953739c43df02" } ,
"login" : "mojombo",...)
```

MongoDB queries are expressed as `DBObject` instances. Keys in the `DBObject` correspond to fields in the collection's documents, and the values are expressions controlling the allowed values of this field. Thus, `DBObject("login" -> "mojombo")` will select all the documents for which the `login` field is `mojombo`. Using a `DBObject` instance to represent a query might seem a little obscure, but it will quickly make sense if you read the MongoDB documentation (<https://docs.mongodb.org/manual/core/crud-introduction/>): queries are themselves just JSON objects in MongoDB. Thus, the fact that the query in Casbah is represented as a `DBObject` is consistent with other MongoDB client implementations. It also allows someone familiar with MongoDB to start writing Casbah queries in no time.

MongoDB supports more complex queries. For instance, to query everyone with `"github_id"` between 20 and 30, we can write the following query:

```
scala> val query = DBObject("github_id" ->
  DBObject("$gte" -> 20, "$lt" -> 30))
query: DBObject = { "github_id" : { "$gte" : 20 , "$lt" : 30}}

scala> collection.find(query).toList
List[com.mongodb.casbah.Imports.DBObject] = List({ "_id" : { "$oid" :
"562e922546f953739c43df0f" } , "github_id" : 23 , "login" : "takeo" , ...}
```

We limit the range of values that `github_id` can take with `DBObject("$gte" -> 20, "$lt" -> 30)`. The `"$gte"` string indicates that `github_id` must be greater or equal to 20. Similarly, `"$lt"` denotes the *less than* operator. To get a full list of operators that you can use when querying, consult the MongoDB reference documentation (<http://docs.mongodb.org/manual/reference/operator/query/>).

So far, we have only looked at queries on top-level fields. Casbah also lets us query fields in subdocuments and arrays using the *dot* notation. In the context of array values, this will return all the documents for which at least one value in the array matches the query. For instance, to retrieve all users who have a repository whose main language is Scala:

```
scala> val query = DBObject("repos.language" -> "Scala")
query: DBObject = { "repos.language" : "Scala"}

scala> collection.find(query).toList
List[DBObject] = List({ "_id" : { "$oid" : "5635da4446f953234ca634df" } ,
"login" : "kevinclark" ... }
```

Casbah query DSL

Using `DBObject` instances to express queries can be very verbose and somewhat difficult to read. Casbah provides a DSL to express queries much more succinctly. For instance, to get all the documents with the `github_id` field between 20 and 30, we would write the following:

```
scala> collection.find("github_id" $gte 20 $lt 30).toList
List[com.mongodb.casbah.Imports.DBObject] = List({ "_id" : { "$oid" :
"562e922546f953739c43df0f" } , "github_id" : 23 , "login" : "takeo" ,
"repos" : ... }
```

The operators provided by the DSL will automatically construct `DBObject` instances. Using the DSL operators as much as possible generally leads to much more readable and maintainable code.

Going into the full details of the query DSL is beyond the scope of this chapter. You should find it quite easy to use. For a full list of the operators supported by the DSL, refer to the Casbah documentation at http://mongodb.github.io/casbah/3.0/reference/query_dsl/. We summarize the most important operators here:

Operators	Description
"login" \$eq "mojombo"	This selects documents whose <code>login</code> field is exactly <code>mojombo</code>
"login" \$ne "mojombo"	This selects documents whose <code>login</code> field is not <code>mojombo</code>
"github_id" \$gt 1 \$lt 20	This selects documents with <code>github_id</code> greater than 1 and less than 20
"github_id" \$gte 1 \$lte 20	This selects documents with <code>github_id</code> greater than or equal to 1 and less than or equal to 20
"login" \$in ("mojombo", "defunkt")	The <code>login</code> field is either <code>mojombo</code> or <code>defunkt</code>
"login" \$nin ("mojombo", "defunkt")	The <code>login</code> field is not <code>mojombo</code> or <code>defunkt</code>
"login" \$regex "^moj.*"	The <code>login</code> field matches the particular regular expression
"login" \$exists true	The <code>login</code> field exists
\$or("login" \$eq "mojombo", "github_id" \$gte 22)	Either the <code>login</code> field is <code>mojombo</code> or the <code>github_id</code> field is greater or equal to 22
\$and("login" \$eq "mojombo", "github_id" \$gte 22)	The <code>login</code> field is <code>mojombo</code> and the <code>github_id</code> field is greater or equal to 22

We can also use the *dot* notation to query arrays and subdocuments. For instance, the following query will count all the users who have a repository in Scala:

```
scala> collection.find("repos.language" $eq "Scala").size
Int = 30
```

Custom type serialization

So far, we have only tried to serialize and deserialize simple types. What if we wanted to decode the language field in the repository array to an enumeration rather than a string? We might, for instance, define the following enumeration:

```
scala> object Language extends Enumeration {
    val Scala, Java, JavaScript = Value
}
defined object Language
```

Casbah lets us define custom serializers tied to a specific Scala type: we can inform Casbah that whenever it encounters an instance of the `Language.Value` type in a `DBObject`, the instance should be passed through a custom transformer that will convert it to, for instance, a string, before writing it to the database.

To define a custom serializer, we need to define a class that extends the `Transformer` trait. This trait exposes a single method, `transform(o: AnyRef) : AnyRef`. Let's define a `LanguageTransformer` trait that transforms from `Language.Value` to `String`:

```
scala> import org.bson.{BSON, Transformer}
import org.bson.{BSON, Transformer}

scala> trait LanguageTransformer extends Transformer {
    def transform(o: AnyRef) : AnyRef = o match {
        case l: Language.Value => l.toString
        case _ => o
    }
}
defined trait LanguageTransformer
```

We now need to register the trait to be used whenever an instance of type `Language.Value` needs to be decoded. We can do this using the `addEncodingHook` method:

```
scala> BSON.addEncodingHook(
    classOf[Language.Value], new LanguageTransformer {})
```

We can now construct `DBObject` instances containing values of the `Language` enumeration:

```
scala> val repoObj = DBObject(  
    "github_id" -> 1234L,  
    "language" -> Language.Scala  
)  
repoObj: DBObject = { "github_id" : 1234 , "language" : "Scala"}
```

What about the reverse? How do we tell Casbah to read the `"language"` field as `Language.Value`? This is not possible with custom deserializers: `"Scala"` is now stored as a string in the database. Thus, when it comes to deserialization, `"Scala"` is no different from, say, `"mojombo"`. We thus lose type information when `"Scala"` is serialized.

Thus, while custom encoding hooks are useful for serialization, they are much less useful when deserializing. A cleaner, more consistent alternative to customize both serialization and deserialization is to use *type classes*. We have already covered how to use these extensively in *Chapter 5, Scala and SQL through JDBC*, in the context of serializing to and from SQL. The procedure here would be very similar:

1. Define a `MongoReader[T]` type class with a `read(v: Any): T` method.
2. Define concrete implementations of `MongoReader` in the `MongoReader` companion object for all types of interest, such as `String`, `Language.Value`.
3. Enrich `DBObject` with a `read[T: MongoReader]` method using the *pimp my library* pattern.

For instance, the implementation of `MongoReader` for `Language.Value` would be as follows:

```
implicit object LanguageReader extends MongoReader[Language.Value] {  
    def read(v: Any): Language.Value = v match {  
        case s: String => Language.withName(s)  
    }  
}
```

We could then do the same with a `MongoWriter` type class. Using type classes is an idiomatic and extensible approach to custom serialization and deserialization.

We provide a complete example of type classes in the code examples associated with this chapter (in the `typeclass` directory).

Beyond Casbah

We have only considered Casbah in this chapter. There are, however, other drivers for MongoDB.

ReactiveMongo is a driver that focusses on asynchronous read and writes to and from the database. All queries return a future, forcing asynchronous behavior. This fits in well with data streams or web applications.

Salat sits at a higher level than Casbah and aims to provide easy serialization and deserialization of case classes.

A full list of drivers is available at <https://docs.mongodb.org/ecosystem/drivers-scala/>.

Summary

In this chapter, you learned how to interact with a MongoDB database. By weaving the constructs learned in the previous chapter – pulling information from a web API – with those learned in this chapter, we can now build a concurrent, reactive program for data ingestion.

In the next chapter, you will learn to build distributed, concurrent structures with greater flexibility using Akka actors.

References

MongoDB: The Definitive Guide, by Kristina Chodorow, is a good introduction to MongoDB. It does not cover interacting with MongoDB in Scala at all, but Casbah is intuitive enough for anyone familiar with MongoDB.

Similarly, the MongoDB documentation (<https://docs.mongodb.org/manual/>) provides an in-depth discussion of MongoDB.

Casbah itself is well-documented (<http://mongodb.github.io/casbah/3.0/>). There is a *Getting Started* guide that is somewhat similar to this chapter and a complete reference guide that will fill in the gaps left by this chapter.

This gist, <https://gist.github.com/switzer/4218526>, implements type classes to serialize and deserialize objects in the domain model to `DBObject`s. The premise is a little different from the suggested usage of type classes in this chapter: we are converting from Scala types to `AnyRef` to be used as values in `DBObject`. However, the two approaches are complementary: one could imagine a set of type classes to convert from `User` or `Repo` to `DBObject` and another to convert from `Language`. `Value` to `AnyRef`.

9

Concurrency with Akka

Much of this book focusses on taking advantage of multicore and distributed architectures. In *Chapter 4, Parallel Collections and Futures*, you learned how to use parallel collections to distribute batch processing problems over several threads and how to perform asynchronous computations using futures. In *Chapter 7, Web APIs*, we applied this knowledge to query the GitHub API with several concurrent threads.

Concurrency abstractions such as futures and parallel collections simplify the enormous complexity of concurrent programming by limiting what you can do. Parallel collections, for instance, force you to phrase your parallelization problem as a sequence of pure functions on collections.

Actors offer a different way of thinking about concurrency. Actors are very good at encapsulating *state*. Managing state shared between different threads of execution is probably the most challenging part of developing concurrent applications, and, as we will discover in this chapter, actors make it manageable.

GitHub follower graph

In the previous two chapters, we explored the GitHub API, learning how to query the API and parse the results using *json-4s*.

Let's imagine that we want to extract the GitHub follower graph: we want a program that will start from a particular user, extract this user followers, and then extract their followers until we tell it to stop. The catch is that we don't know ahead of time what URLs we need to fetch: when we download the login names of a particular user's followers, we need to verify whether we have fetched these users previously. If not, we add them to a queue of users whose followers we need to fetch. Algorithm aficionados might recognize this as *breadth-first search*.

Let's outline how we might write this in a single-threaded way. The central components are a set of visited users and queue of future users to visit:

```
val seedUser = "odersky" // the origin of the network

// Users whose URLs need to be fetched
val queue = mutable.Queue(seedUser)

// set of users that we have already fetched
// (to avoid re-fetching them)
val fetchedUsers = mutable.Set.empty[String]

while (queue.nonEmpty) {
    val user = queue.dequeue
    if (!fetchedUsers(user)) {
        val followers = fetchFollowersForUser(user)
        followers foreach { follower =>
            // add the follower to queue of people whose
            // followers we want to find.
            queue += follower
        }
        fetchedUsers += user
    }
}
```

Here, the `fetchFollowersForUser` method has signature `String => Iterable[String]` and is responsible for taking a login name, transforming it into a URL in the GitHub API, querying the API, and extracting a list of followers from the response. We will not implement it here, but you can find a complete example in the `chap09/single_threaded` directory of the code examples for this book (<https://github.com/pbugnion/s4ds>). You should have all the tools to implement this yourself if you have read *Chapter 7, Web APIs*.

While this works, it will be painfully slow. The bottleneck is clearly the `fetchFollowersForUser` method, in particular, the part that queries the GitHub API. This program does not lend itself to the concurrency constructs that we have seen earlier in the book because we need to protect the state of the program, embodied by the user queue and set of fetched users, from race conditions. Note that it is not just a matter of making the queue and set thread-safe. We must also keep the two synchronized.

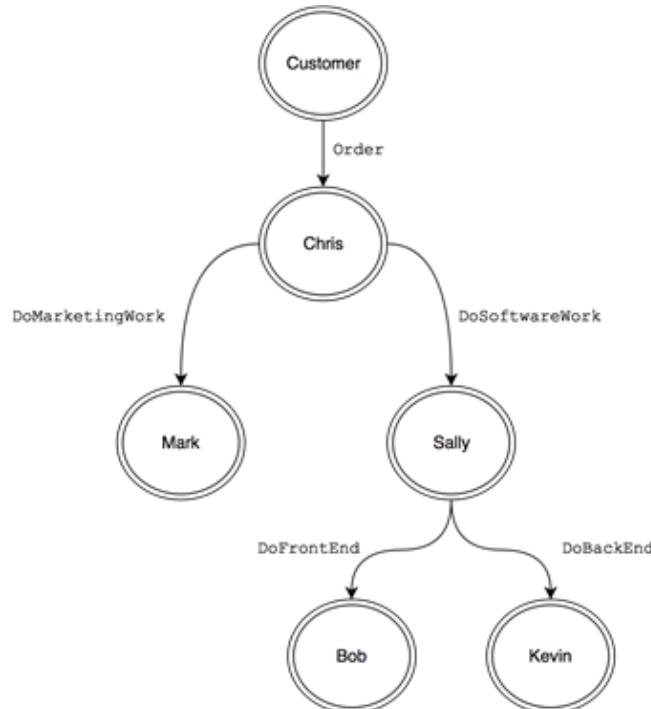
Actors offer an elegant abstraction to encapsulate state. They are lightweight objects that each perform a single task (possibly repeatedly) and communicate with each other by passing messages. The internal state of an actor can only be changed from within the actor itself. Importantly, actors only process messages one at a time, effectively preventing race conditions.

By hiding program state inside actors, we can reason about the program more effectively: if a bug is introduced that makes this state inconsistent, the culprit will be localized entirely in that actor.

Actors as people

In the previous section, you learned that an actor encapsulates state, interacting with the outside world through messages. Actors make concurrent programming more intuitive because they behave a little bit like an ideal workforce.

Let's think of an actor system representing a start-up with five people. There's Chris, the CEO, and Mark, who's in charge of marketing. Then there's Sally, who heads the engineering team. Sally has two minions, Bob and Kevin. As every good organization needs an organizational chart, refer to the following diagram:



Let's say that Chris receives an order. He will look at the order, decide whether it is something that he can process himself, and if not, he will forward it to Mark or Sally. Let's assume that the order asks for a small program so Bob forwards the order to Sally. Sally is very busy working on a backlog of orders so she cannot process the order message straightforwardly, and it will just sit in her mailbox for a short while. When she finally gets round to processing the order, she might decide to split the order into several parts, some of which she will give to Kevin and some to Bob.

As Bob and Kevin complete items, they will send messages back to Sally to inform her. When every part of the order is fulfilled, Sally will aggregate the parts together and message either the customer directly or Chris with the results.

The task of keeping track of which jobs must be fulfilled to complete the order rests with Sally. When she receives messages from Bob and Kevin, she must update her list of tasks in progress and check whether every task related to this order is complete. This sort of coordination would be more challenging with traditional *synchronize* blocks: every access to the list of tasks in progress and to the list of completed tasks would need to be synchronized. By embedding this logic in Sally, who can only process a single message at a time, we can be sure that there will not be race conditions.

Our start-up works well because each person is responsible for doing a single thing: Chris either delegates to Mark or Sally, Sally breaks up orders into several parts and assigns them to Bob and Kevin, and Bob and Kevin fulfill each part. You might think "hold on, all the logic is embedded in Bob and Kevin, the employees at the bottom of the ladder who do all the actual work". Actors, unlike employees, are cheap, so if the logic embedded in an actor gets too complicated, it is easy to introduce additional layers of delegation until tasks get simple enough.

The employees in our start-up refuse to multitask. When they get a piece of work, they process it completely and then move on to the next task. This means that they cannot get muddled by the complexities of multitasking. Actors, by processing a single message at a time, greatly reduce the scope for introducing concurrency errors such as race conditions.

More importantly, by offering an abstraction that programmers can intuitively understand – that of human workers – Akka makes reasoning about concurrency easier.

Hello world with Akka

Let's install Akka. We add it as a dependency to our `build.sbt` file:

```
scalaVersion := "2.11.7"

libraryDependencies += "com.typesafe.akka" %% "akka-actor" %
  "2.4.0"
```

We can now import Akka as follows:

```
import akka.actor._
```

For our first foray into the world of actors, we will build an actor that echoes every message it receives. The code examples for this section are in a directory called `chap09/hello_akka` in the sample code provided with this book (<https://github.com/pbugnion/s4ds>):

```
// EchoActor.scala
import akka.actor._

class EchoActor extends Actor with ActorLogging {
  def receive = {
    case msg:String =>
      Thread.sleep(500)
      log.info(s"Received '$msg'")
  }
}
```

Let's pick this example apart, starting with the constructor. Our actor class must extend `Actor`. We also add `ActorLogging`, a utility trait that adds the `log` attribute.

The Echo actor exposes a single method, `receive`. This is the actor's only way of communicating with the external world. To be useful, all actors must expose a `receive` method. The `receive` method is a partial function, typically implemented with multiple `case` statements. When an actor starts processing a message, it will match it against every `case` statement until it finds one that matches. It will then execute the corresponding block.

Our echo actor accepts a single type of message, a plain string. When this message gets processed, the actor waits for half a second and then echoes the message to the log file.

Let's instantiate a couple of Echo actors and send them messages:

```
// HelloAkka.scala

import akka.actor._

object HelloAkka extends App {

    // We need an actor system before we can
    // instantiate actors
    val system = ActorSystem("HelloActors")

    // instantiate our two actors
    val echo1 = system.actorOf(Props[EchoActor], name="echo1")
    val echo2 = system.actorOf(Props[EchoActor], name="echo2")

    // Send them messages. We do this using the "!" operator
    echo1 ! "hello echo1"
    echo2 ! "hello echo2"
    echo1 ! "bye bye"

    // Give the actors time to process their messages,
    // then shut the system down to terminate the program
    Thread.sleep(500)
    system.shutdown
}
```

Running this gives us the following output:

```
[INFO] [07/19/2015 17:15:23.954] [HelloActor akka.actor.default-
dispatcher-2] [akka://HelloActor/user/echo1] Received 'hello echo1'
[INFO] [07/19/2015 17:15:23.954] [HelloActor akka.actor.default-
dispatcher-3] [akka://HelloActor/user/echo2] Received 'hello echo2'
[INFO] [07/19/2015 17:15:24.955] [HelloActor akka.actor.default-
dispatcher-2] [akka://HelloActor/user/echo1] Received 'bye bye'
```

Note that the echo1 and echo2 actors are clearly acting concurrently: hello echo1 and hello echo2 are logged at the same time. The second message, passed to echo1, gets processed after the actor has finished processing hello echo1.

There are a few different things to note:

- To start instantiating actors, we must first create an actor system. There is typically a single actor system per application.
- The way in which we instantiate actors looks a little strange. Instead of calling the constructor, we create an actor properties object, `Props[T]`. We then ask the actor system to create an actor with these properties. In fact, we never instantiate actors with `new`: they are either created by calling the `actorOf` method in the actor system or a similar method from within another actor (more on this later).

We never call an actor's methods from outside that actor. The only way to interact with the actor is to send messages to it. We do this using the `tell` operator, `!`. There is thus no way to mess with an actor's internals from outside that actor (or at least, Akka makes it difficult to mess with an actor's internals).

Case classes as messages

In our "hello world" example, we constructed an actor that is expected to receive a string as message. Any object can be passed as a message, provided it is immutable. It is very common to use case classes to represent messages. This is better than using strings because of the additional type safety: the compiler will catch a typo in a case class but not in a string.

Let's rewrite our `EchoActor` to accept instances of case classes as messages. We will make it accept two different messages: `EchoMessage(message)` and `EchoHello`, which just echoes a default message. The examples for this section and the next are in the `chap09/hello_akka_case_classes` directory in the sample code provided with this book (<https://github.com/pbugnion/s4ds>).

A common Akka pattern is to define the messages that an actor can receive in the actor's companion object:

```
// EchoActor.scala

object EchoActor {
    case object EchoHello
    case class EchoMessage(msg:String)
}
```

Let's change the actor definition to accept these messages:

```
class EchoActor extends Actor with ActorLogging {  
    import EchoActor._ // import the message definitions  
    def receive = {  
        case EchoHello => log.info("hello")  
        case EchoMessage(s) => log.info(s)  
    }  
}
```

We can now send `EchoHello` and `EchoMessage` to our actors:

```
echo1 ! EchoActor.EchoHello  
echo2 ! EchoActor.EchoMessage("We're learning Akka.")
```

Actor construction

Actor construction is a common source of difficulty for people new to Akka. Unlike (most) ordinary objects, you never instantiate actors explicitly. You would never write, for instance, `val echo = new EchoActor`. In fact, if you try this, Akka raises an exception.

Creating actors in Akka is a two-step process: you first create a `Props` object, which encapsulates the properties needed to construct an actor. The way to construct a `Props` object differs depending on whether the actor takes constructor arguments. If the constructor takes no arguments, we simply pass the actor class as a type parameter to `Props`:

```
val echoProps = Props[EchoActor]
```

If we have an actor whose constructor does take arguments, we must pass these as additional arguments when defining the `Props` object. Let's consider the following actor, for instance:

```
class TestActor(a:String, b:Int) extends Actor { ... }
```

We pass the constructor arguments to the `Props` object as follows:

```
val testProps = Props(classOf[TestActor], "hello", 2)
```

The `Props` instance just embodies the configuration for creating an actor. It does not actually create anything. To create an actor, we pass the `Props` instance to the `system.actorOf` method, defined on the `ActorSystem` instance:

```
val system = ActorSystem("HelloActors")  
val echo1 = system.actorOf(echoProps, name="hello-1")
```

The `name` parameter is optional but is useful for logging and error messages. The value returned by `.actorOf` is not the actor itself: it is a *reference* to the actor (it helps to think of it as an address that the actor lives at) and has the `ActorRef` type. `ActorRef` is immutable, but it can be serialized and duplicated without affecting the underlying actor.

There is another way to create actors besides calling `actorOf` on the actor system: each actor exposes a `context.actorOf` method that takes a `Props` instance as its argument. The context is only accessible from within the actor:

```
class TestParentActor extends Actor {
    val echoChild = context.actorOf(echoProps, name="hello-child")
    ...
}
```

The difference between an actor created from the actor system and an actor created from another actor's context lies in the actor hierarchy: each actor has a parent. Any actor created within another actor's context will have that actor as its parent. An actor created by the actor system has a predefined actor, called the *user guardian*, as its parent. We will understand the importance of the actor hierarchy when we study the actor lifecycle at the end of this chapter.

A very common idiom is to define a `props` method in an actor's companion object that acts as a factory method for `Props` instances for that actor. Let's amend the `EchoActor` companion object:

```
object EchoActor {
    def props: Props = Props[EchoActor]

    // message case class definitions here
}
```

We can then instantiate the actor as follows:

```
val echoActor = system.actorOf(EchoActor.props)
```

Anatomy of an actor

Before diving into a full-blown application, let's look at the different components of the actor framework and how they fit together:

- **Mailbox:** A mailbox is basically a queue. Each actor has its own mailbox. When you send a message to an actor, the message lands in its mailbox and does nothing until the actor takes it off the queue and passes it through its `receive` method.

- **Messages:** Messages make synchronization between actors possible. A message can have any type with the sole requirement that it should be immutable. In general, it is better to use case classes or case objects to gain the compiler's help in checking message types.
- **Actor reference:** When we create an actor using `val echo1 = system.actorOf(Props[EchoActor])`, `echo1` has type `ActorRef`. An `ActorRef` is a proxy for an actor and is what the rest of the world interacts with: when you send a message, you send it to the `ActorRef`, not to the actor directly. In fact, you can never obtain a handle to an actor directly in Akka. An actor can obtain an `ActorRef` for itself using the `.self` method.
- **Actor context:** Each actor has a `context` attribute through which you can access methods to create or access other actors and find information about the outside world. We have already seen how to create new actors with `context.actorOf(props)`. We can also obtain a reference to an actor's parent through `context.parent`. An actor can also stop another actor with `context.stop(actorRef)`, where `actorRef` is a reference to the actor that we want to stop.
- **Dispatcher:** The dispatcher is the machine that actually executes the code in an actor. The default dispatcher uses a fork/join thread pool. Akka lets us use different dispatchers for different actors. Tweaking the dispatcher can be useful to optimize the performance and give priority to certain actors. The dispatcher that an actor runs on is accessible through `context.dispatcher`. Dispatchers implement the `ExecutionContext` interface so they can be used to run futures.

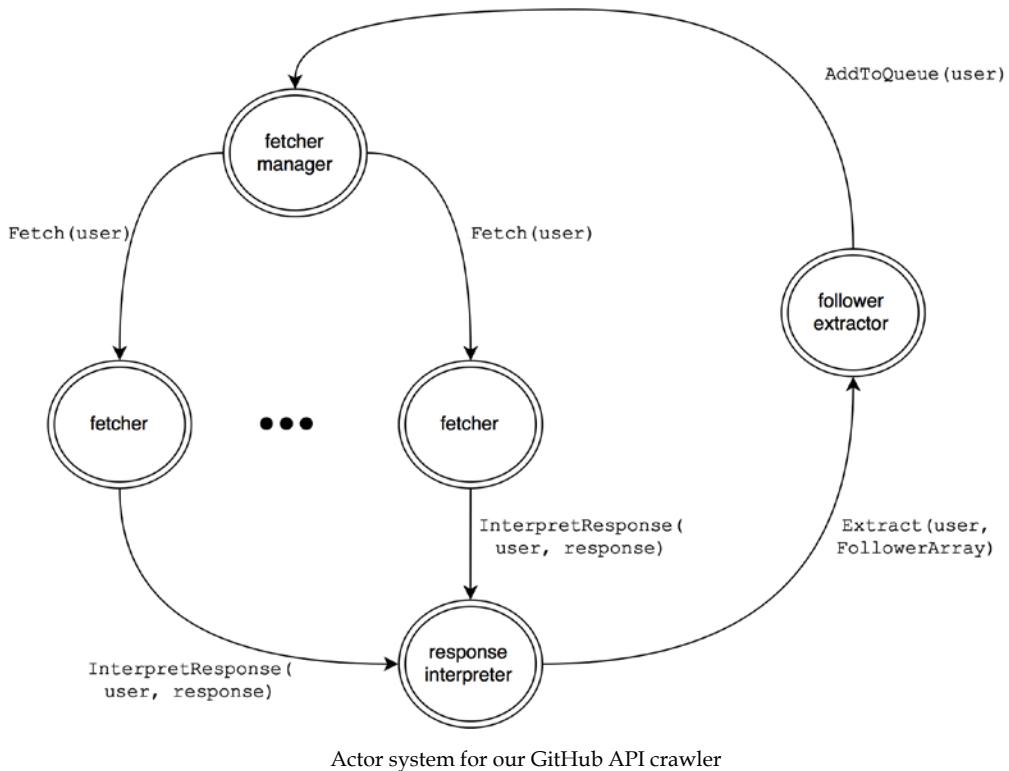
Follower network crawler

The end game for this chapter is to build a crawler to explore GitHub's follower graph. We have already outlined how we can do this in a single-threaded manner earlier in this chapter. Let's design an actor system to do this concurrently.

The moving parts in the code are the data structures managing which users have been fetched or are being fetched. These need to be encapsulated in an actor to avoid race conditions arising from multiple actors trying to change them concurrently. We will therefore create a *fetcher manager* actor whose job is to keep track of which users have been fetched and which users we are going to fetch next.

The part of the code that is likely to be a bottleneck is querying the GitHub API. We therefore want to be able to scale the number of workers doing this concurrently. We will create a pool of *fetchers*, actors responsible for querying the API for the followers of a particular user. Finally, we will create an actor whose responsibility is to interpret the API's response. This actor will forward its interpretation of the response to another actor who will extract the followers and give them to the fetcher manager.

This is what the architecture of the program will look like:



Each actor in our program performs a single task: fetchers just query the GitHub API and the queue manager just distributes work to the fetchers. Akka best practice dictates giving actors as narrow an area of responsibility as possible. This enables better granularity when scaling out (for instance, by adding more fetcher actors, we just parallelize the bottleneck) and better resilience: if an actor fails, it will only affect his area of responsibility. We will explore actor failure later on in this chapter.

We will build the app in several steps, exploring the Akka toolkit as we write the program. Let's start with the `build.sbt` file. Besides Akka, we will mark `scalaj-http` and `json4s` as dependencies:

```
// build.sbt
scalaVersion := "2.11.7"

libraryDependencies ++= Seq(
  "org.json4s" %% "json4s-native" % "3.2.10",
  "org.scalaj" %% "scalaj-http" % "1.1.4",
  "com.typesafe.akka" %% "akka-actor" % "2.3.12"
)
```

Fetcher actors

The workhorse of our application is the fetcher, the actor responsible for fetching the follower details from GitHub. In the first instance, our actor will accept a single message, `Fetch(user)`. It will fetch the followers corresponding to `user` and log the response to screen. We will use the recipes developed in *Chapter 7, Web APIs*, to query the GitHub API with an OAuth token. We will inject the token through the actor constructor.

Let's start with the companion object. This will contain the definition of the `Fetch(user)` message and two factory methods to create the `Props` instances. You can find the code examples for this section in the `chap09/fetchers_alone` directory in the sample code provided with this book (<https://github.com/pbugnion/s4ds>):

```
// Fetcher.scala
import akka.actor._
import scalaj.http._
import scala.concurrent.Future

object Fetcher {
  // message definitions
  case class Fetch(login:String)

  // Props factory definitions
  def props(token:Option[String]):Props =
    Props(classOf[Fetcher], token)
  def props():Props = Props(classOf[Fetcher], None)
}
```

Let's now define the fetcher itself. We will wrap the call to the GitHub API in a future. This avoids a single slow request blocking the actor. When our actor receives a `Fetch` request, it wraps this request into a future, sends it off, and can then process the next message. Let's go ahead and implement our actor:

```
// Fetcher.scala
class Fetcher(val token:Option[String])
extends Actor with ActorLogging {
    import Fetcher._ // import message definition

    // We will need an execution context for the future.
    // Recall that the dispatcher doubles up as execution
    // context.
    import context.dispatcher

    def receive = {
        case Fetch(login) => fetchUrl(login)
    }

    private def fetchUrl(login:String) {
        val unauthorizedRequest = Http(
            s"https://api.github.com/users/$login/followers")
        val authorizedRequest = token.map { t =>
            unauthorizedRequest.header("Authorization", s"token $t")
        }

        // Prepare the request: try to use the authorized request
        // if a token was given, and fall back on an unauthorized
        // request
        val request = authorizedRequest.getOrElse(unauthorizedRequest)

        // Fetch from github
        val response = Future { request.asString }
        response.onComplete { r =>
            log.info(s"Response from $login: $r")
        }
    }
}
```

Let's instantiate an actor system and four fetchers to check whether our actor is working as expected. We will read the GitHub token from the environment, as described in *Chapter 7, Web APIs*, then create four actors and ask each one to fetch the followers of a particular GitHub user. We wait five seconds for the requests to get completed, and then shut the system down:

```
// FetcherDemo.scala
import akka.actor._

object FetcherDemo extends App {
    import Fetcher._ // Import the messages

    val system = ActorSystem("fetchers")

    // Read the github token if present.
    val token = sys.env.get("GHTOKEN")

    val fetchers = (0 until 4).map { i =>
        system.actorOf(Fetcher.props(token))
    }

    fetchers(0) ! Fetch("odersky")
    fetchers(1) ! Fetch("derekwyatt")
    fetchers(2) ! Fetch("rkuhn")
    fetchers(3) ! Fetch("tototoshi")

    Thread.sleep(5000) // Wait for API calls to finish
    system.shutdown // Shut system down
}
```

Let's run the code through SBT:

```
$ GHTOKEN="2502761..." sbt run
[INFO] [11/08/2015 16:28:06.500] [fetchers-akka.actor.default-
dispatcher-2] [akka://fetchers/user/$d] Response from tototoshi: Success
(HttpResponse([{"login":"akr4","id":10892,"avatar_url":"https://avatars.
githubusercontent.com/u/10892?v=3","gravatar_id":""...]
```

Notice how we explicitly need to shut the actor system down using `system.shutdown`. The program hangs until the system is shut down. However, shutting down the system will stop all the actors, so we need to make sure that they have finished working. We do this by inserting a call to `Thread.sleep`.

Using `Thread.sleep` to wait until the API calls have finished to shut down the actor system is a little crude. A better approach could be to let the actors signal back to the system that they have completed their task. We will see examples of this pattern later when we implement the *fetcher manager* actor.

Akka includes a feature-rich *scheduler* to schedule events. We can use the scheduler to replace the call to `Thread.sleep` by scheduling a system shutdown five seconds in the future. This is preferable as the scheduler does not block the calling thread, unlike `Thread.sleep`. To use the scheduler, we need to import a global execution context and the duration module:

```
// FetcherDemoWithScheduler.scala

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
```

We can then schedule a system shutdown by replacing our call to `Thread.sleep` with the following:

```
system.scheduler.scheduleOnce(5.seconds) { system.shutdown }
```

Besides `scheduleOnce`, the scheduler also exposes a `schedule` method that lets you schedule events to happen regularly (every two seconds, for instance). This is useful for heartbeat checks or monitoring systems. For more information, read the API documentation on the scheduler available at <http://doc.akka.io/docs/akka/snapshot-scala/scheduler.html>.

Note that we are actually cheating a little bit here by not fetching every follower. The response to the follower's query is actually paginated, so we would need to fetch several pages to fetch all the followers. Adding logic to the actor to do this is not terribly complicated. We will ignore this for now and assume that users are capped at 100 followers each.

Routing

In the previous example, we created four fetchers and dispatched messages to them, one after the other. We have a pool of identical actors among which we distribute tasks. Manually routing the messages to the right actor to maximize the utilization of our pool is painful and error-prone. Fortunately, Akka provides us with several routing strategies that we can use to distribute work among our pool of actors. Let's rewrite the previous example with automatic routing. You can find the code examples for this section in the `chap09/fetchers_routing` directory in the sample code provided with this book (<https://github.com/pbugnion/s4ds>). We will reuse the same definition of `Fetchers` and its companion object as we did in the previous section.

Let's start by importing the routing package:

```
// FetcherDemo.scala
import akka.routing._
```

A *router* is an actor that forwards the messages that it receives to its children. The easiest way to define a pool of actors is to tell Akka to create a router and pass it a `Props` object for its children. The router will then manage the creation of the workers directly. In our example (we will only comment on the parts that differ from the previous example in the text, but you can find the full code in the `fetchers_routing` directory with the examples for this chapter), we replace the custom `Fetcher` creation code with the following:

```
// FetcherDemo.scala

// Create a router with 4 workers of props Fetcher.props()
val router = system.actorOf(
    RoundRobinPool(4).props(Fetcher.props(token))
)
```

We can then send the fetch messages directly to the router. The router will route the messages to the children in a round-robin manner:

```
List("odersky", "derekwyatt", "rkuhn", "tototoshi").foreach {
    login => router ! Fetch(login)
}
```

We used a round-robin router in this example. Akka offers many different types of routers, including routers with dynamic pool size, to cater to different types of load balancing. Head over to the Akka documentation for a list of all the available routers, at <http://doc.akka.io/docs/akka/snapshot/scala/routing.html>.

Message passing between actors

Merely logging the API response is not very useful. To traverse the follower graph, we must perform the following:

- Check the return code of the response to make sure that the GitHub API was happy with our request
- Parse the response as JSON
- Extract the login names of the followers and, if we have not fetched them already, push them into the queue

You learned how to do all these things in *Chapter 7, Web APIs*, but not in the context of actors.

We could just add the additional processing steps to the `receive` method of our `Fetcher` actor: we could add further transformations to the API response by future composition. However, having actors do several different things, and possibly failing in several different ways, is an anti-pattern: when we learn about managing the actor life cycle, we will see that it becomes much more difficult to reason about our actor systems if the actors contain several bits of logic.

We will therefore use a pipeline of three different actors:

- The fetchers, which we have already encountered, are responsible just for fetching a URL from GitHub. They will fail if the URL is badly formatted or they cannot access the GitHub API.
- The response interpreter is responsible for taking the response from the GitHub API and parsing it to JSON. If it fails at any step, it will just log the error (in a real application, we might take different corrective actions depending on the type of failure). If it manages to extract JSON successfully, it will pass the JSON array to the follower extractor.
- The follower extractor will extract the followers from the JSON array and pass them on to the queue of users whose followers we need to fetch.

We have already built the fetchers, though we will need to modify them to forward the API response to the response interpreter rather than just logging it.

You can find the code examples for this section in the `chap09/all_workers` directory in the sample code provided with this book (<https://github.com/pbugnion/s4ds>). The first step is to modify the fetchers so that, instead of logging the response, they forward the response to the response interpreter. To be able to forward the response to the response interpreter, the fetchers will need a reference to this actor. We will just pass the reference to the response interpreter through the fetcher constructor, which is now:

```
// Fetcher.scala
class Fetcher(
    val token:Option[String],
    val responseInterpreter:ActorRef)
extends Actor with ActorLogging {
    ...
}
```

We must also modify the `Props` factory method in the companion object:

```
// Fetcher.scala
def props(
    token:Option[String], responseInterpreter:ActorRef
):Props = Props(classOf[Fetcher], token, responseInterpreter)
```

We must also modify the `receive` method to forward the HTTP response to the interpreter rather than just logging it:

```
// Fetcher.scala
class Fetcher(...) extends Actor with ActorLogging {
    ...
    def receive = {
        case Fetch(login) => fetchFollowers(login)
    }

    private def fetchFollowers(login:String) {
        val unauthorizedRequest = Http(
            s"https://api.github.com/users/$login/followers")
        val authorizedRequest = token.map { t =>
            unauthorizedRequest.header("Authorization", s"token $t")
        }

        val request = authorizedRequest.getOrElse(unauthorizedRequest)
        val response = Future { request.asString }

        // Wrap the response in an InterpretResponse message and
        // forward it to the interpreter.
    }
}
```

```
    response.onComplete { r =>
      responseInterpreter !
        ResponseInterpreter.InterpretResponse(login, r)
    }
  }
}
```

The *response interpreter* takes the response, decides if it is valid, parses it to JSON, and forwards it to a follower extractor. The response interpreter will need a reference to the follower extractor, which we will pass in the constructor.

Let's start by defining the `ResponseInterpreter` companion. It will just contain the definition of the messages that the response interpreter can receive and a factory to create a `Props` object to help with instantiation:

```
// ResponseInterpreter.scala
import akka.actor._
import scala.util._

import scalaj.http._
import org.json4s._
import org.json4s.native.JsonMethods._

object ResponseInterpreter {

  // Messages
  case class InterpretResponse(
    login:String, response:Try[HttpResponse[String]]
  )

  // Props factory
  def props(followerExtractor:ActorRef) =
    Props(classOf[ResponseInterpreter], followerExtractor)
}
```

The body of `ResponseInterpreter` should feel familiar: when the actor receives a message giving it a response to interpret, it parses it to JSON using the techniques that you learned in *Chapter 7, Web APIs*. If we parse the response successfully, we forward the parsed JSON to the follower extractor. If we fail to parse the response (possibly because it was badly formatted), we just log the error. We could recover from this in other ways, for instance, by re-adding this login to the queue manager to be fetched again:

```
// ResponseInterpreter.scala
class ResponseInterpreter(followerExtractor:ActorRef)
```

```
extends Actor with ActorLogging {
    // Import the message definitions
    import ResponseInterpreter._

    def receive = {
        case InterpretResponse(login, r) => interpret(login, r)
    }

    // If the query was successful, extract the JSON response
    // and pass it onto the follower extractor.
    // If the query failed, or is badly formatted, throw an error
    // We should also be checking error codes here.
    private def interpret(
        login:String, response:Try[HttpResponse[String]])
    ) = response match {
        case Success(r) => responseToJson(r.body) match {
            case Success(jsonResponse) =>
                followerExtractor ! FollowerExtractor.Extract(
                    login, jsonResponse)
            case Failure(e) =>
                log.error(
                    s"Error parsing response to JSON for $login: $e")
        }
        case Failure(e) => log.error(
            s"Error fetching URL for $login: $e")
    }

    // Try and parse the response body as JSON.
    // If successful, coerce the `JValue` to a `JArray`.
    private def responseToJson(responseBody:String):Try[JArray] = {
        val jvalue = Try { parse(responseBody) }
        jvalue.flatMap {
            case a:JArray => Success(a)
            case _ => Failure(new IllegalStateException(
                "Incorrectly formatted JSON: not an array"))
        }
    }
}
```

We now have two-thirds of our worker actors. The last link is the follower extractor. This actor's job is simple: it takes the `JArray` passed to it by the response interpreter and converts it to a list of followers. For now, we will just log this list, but when we build our fetcher manager, the follower extractor will send messages asking the manager to add the followers to its queue of logins to fetch.

As before, the companion just defines the messages that this actor can receive and a Props factory method:

```
// FollowerExtractor.scala
import akka.actor._

import org.json4s._
import org.json4s.native.JsonMethods._

object FollowerExtractor {

    // Messages
    case class Extract(login:String, jsonResponse:JArray)

    // Props factory method
    def props = Props[FollowerExtractor]
}
```

The `FollowerExtractor` class receives `Extract` messages containing a `JArray` of information representing a follower. It extracts the `login` field and logs it:

```
class FollowerExtractor extends Actor with ActorLogging {
    import FollowerExtractor._

    def receive = {
        case Extract(login, followerArray) => {
            val followers = extractFollowers(followerArray)
            log.info(s"$login -> ${followers.mkString(", ")}")
        }
    }

    def extractFollowers(followerArray:JArray) = for {
        JObject(follower) <- followerArray
        JField("login", JString(login)) <- follower
    } yield login
}
```

Let's write a new `main` method to exercise all our actors:

```
// FetchNetwork.scala

import akka.actor._
import akka.routing._
import scala.concurrent.ExecutionContext.Implicits.global
```

```
import scala.concurrent.duration._

object FetchNetwork extends App {

    import Fetcher._ // Import messages and factory method

    // Get token if exists
    val token = sys.env.get("GHTOKEN")

    val system = ActorSystem("fetchers")

    // Instantiate actors
    val followerExtractor = system.actorOf(FollowerExtractor.props)
    val responseInterpreter =
        system.actorOf(ResponseInterpreter.props(followerExtractor))

    val router = system.actorOf(RoundRobinPool(4).props(
        Fetcher.props(token, responseInterpreter)))
    )

    List("odersky", "derekwyatt", "rkuhn", "tototoshi") foreach {
        login => router ! Fetch(login)
    }

    // schedule a shutdown
    system.scheduler.scheduleOnce(5.seconds) { system.shutdown }

}
```

Let's run this through SBT:

```
$ GHTOKEN="2502761d..." sbt run
[INFO] [11/05/2015 20:09:37.048] [fetchers-akka.actor.default-
dispatcher-3] [akka://fetchers/user/$a] derekwyatt -> adulteratedjedi,
joonas, Psycojoker, trapd00r, tyru, ...
[INFO] [11/05/2015 20:09:37.050] [fetchers-akka.actor.default-
dispatcher-3] [akka://fetchers/user/$a] tototoshi -> akr4, yuroyoro,
seratch, yyuu, ...
[INFO] [11/05/2015 20:09:37.051] [fetchers-akka.actor.default-
dispatcher-3] [akka://fetchers/user/$a] odersky -> misto, gkossakowski,
mushtaq, ...
[INFO] [11/05/2015 20:09:37.052] [fetchers-akka.actor.default-
dispatcher-3] [akka://fetchers/user/$a] rkuhn -> arnbak, uzoice, jond3k,
TimothyKlim, relrod, ...
```

Queue control and the pull pattern

We have now defined the three worker actors in our crawler application. The next step is to define the manager. The *fetcher manager* is responsible for keeping a queue of logins to fetch as well as a set of login names that we have already seen in order to avoid fetching the same logins more than once.

A first attempt might involve building an actor that keeps a set of users that we have already seen and just dispatches it to a round-robin router for fetchers when it is given a new user to fetch. The problem with this approach is that the number of messages in the fetchers' mailboxes would accumulate quickly: for each API query, we are likely to get tens of followers, each of which is likely to make it back to a fetcher's inbox. This gives us very little control over the amount of work piling up.

The first problem that this is likely to cause involves the GitHub API rate limit: even with authentication, we are limited to 5,000 requests per hour. It would be useful to stop queries as soon as we hit this threshold. We cannot be responsive if each fetcher has a backlog of hundreds of users that they need to fetch.

A better alternative is to use a *pull* system: the fetchers request work from a central queue when they find themselves idle. Pull systems are common in Akka when we have a producer that produces work faster than consumers can process it (refer to <http://www.michaelpollmeier.com/akka-work-pulling-pattern/>).

Conversations between the manager and fetchers will proceed as follows:

- If the manager goes from a state of having no work to having work, it sends a `WorkAvailable` message to all the fetchers.
- Whenever a fetcher receives a `WorkAvailable` message or when it completes an item of work, it sends a `GiveMeWork` message to the queue manager.
- When the queue manager receives a `GiveMeWork` message, it ignores the request if no work is available or it is throttled. If it has work, it sends a `Fetch(user)` message to the actor.

Let's start by modifying our fetcher. You can find the code examples for this section in the `chap09/ghub_crawler` directory in the sample code provided with this book (<https://github.com/pbugnion/s4ds>). We will pass a reference to the fetcher manager through the constructor. We need to change the companion object to add the `WorkAvailable` message and the props factory to include the reference to the manager:

```
// Fetcher.scala
object Fetcher {
  case class Fetch(url:String)
```

```
case object WorkAvailable

def props(
    token:Option[String],
    fetcherManager:ActorRef,
    responseInterpreter:ActorRef):Props =
  Props(classOf[Fetcher],
    token, fetcherManager, responseInterpreter)
}
```

We also need to change the `receive` method so that it queries the `FetcherManager` asking for more work once it's done processing a request or when it receives a `WorkAvailable` message.

This is the final version of the fetchers:

```
class Fetcher(
  val token:Option[String],
  val fetcherManager:ActorRef,
  val responseInterpreter:ActorRef)
extends Actor with ActorLogging {
  import Fetcher._
  import context.dispatcher

  def receive = {
    case Fetch(login) => fetchFollowers(login)
    case WorkAvailable =>
      fetcherManager ! FetcherManager.GiveMeWork
  }

  private def fetchFollowers(login:String) {
    val unauthorizedRequest = Http(
      s"https://api.github.com/users/$login/followers")
    val authorizedRequest = token.map { t =>
      unauthorizedRequest.header("Authorization", s"token $t")
    }
    val request = authorizedRequest.getOrElse(unauthorizedRequest)
    val response = Future { request.asString }

    response.onComplete { r =>
      responseInterpreter !
        ResponseInterpreter.InterpretResponse(login, r)
      fetcherManager ! FetcherManager.GiveMeWork
    }
  }
}
```

Now that we have a working definition of the fetchers, let's build the `FetcherManager`. This is the most complex actor that we have built so far, and, before we dive into building it, we need to learn a bit more about the components of the Akka toolkit.

Accessing the sender of a message

When our fetcher manager receives a `GiveMeWork` request, we will need to send work back to the correct fetcher. We can access the actor who sent a message using the `sender` method, which is a method of `Actor` that returns the `ActorRef` corresponding to the actor who sent the message currently being processed. The case statement corresponding to `GiveMeWork` in the fetcher manager is therefore:

```
def receive = {
    case GiveMeWork =>
        login = // get next login to fetch
        sender ! Fetcher.Fetch(login)
        ...
}
```

As `sender` is a *method*, its return value will change for every new incoming message. It should therefore only be used synchronously with the `receive` method. In particular, using it in a future is dangerous:

```
def receive = {
    case DoSomeWork =>
        val work = Future { Thread.sleep(20000) ; 5 }
        work.onComplete { result =>
            sender ! Complete(result) // NO!
        }
}
```

The problem is that when the future is completed 20 seconds after the message is processed, the actor will, in all likelihood, be processing a different message so the return value of `sender` will have changed. We will thus send the `Complete` message to a completely different actor.

If you need to reply to a message outside of the `receive` method, such as when a future completes, you should bind the value of the current sender to a variable:

```
def receive = {
    case DoSomeWork =>
        // bind the current value of sender to a val
        val requestor = sender
        val work = Future { Thread.sleep(20000) ; 5 }
        work.onComplete { result => requestor ! Complete(result) }
}
```

Stateful actors

The behavior of the fetcher manager depends on whether it has work to give out to the fetchers:

- If it has work to give, it needs to respond to `GiveMeWork` messages with a `Fetcher.Fetch` message
- If it does not have work, it must ignore the `GiveMeWork` messages and, if work gets added, it must send a `WorkAvailable` message to the fetchers

Encoding the notion of state is straightforward in Akka. We specify different `receive` methods and switch from one to the other depending on the state. We will define the following `receive` methods for our fetcher manager, corresponding to each of the states:

```
// receive method when the queue is empty
def receiveWhileEmpty: Receive = {
    ...
}

// receive method when the queue is not empty
def receiveWhileNotEmpty: Receive = {
    ...
}
```

Note that we must define the return type of the `receive` methods as `Receive`. To switch the actor from one method to the other, we can use `context.become(methodName)`. Thus, for instance, when the last login name is popped off the queue, we can transition to using the `receiveWhileEmpty` method with `context.become(receiveWhileEmpty)`. We set the initial state by assigning `receiveWhileEmpty` to the `receive` method:

```
def receive = receiveWhileEmpty
```

Follower network crawler

We are now ready to code up the remaining pieces of our network crawler. The largest missing piece is the fetcher manager. Let's start with the companion object. As with the worker actors, this just contains the definitions of the messages that the actor can receive and a factory to create the `Props` instance:

```
// FetcherManager.scala
import scala.collection.mutable
import akka.actor._

object FetcherManager {
  case class AddToQueue(login:String)
  case object GiveMeWork

  def props(token:Option[String], nFetchers:Int) =
    Props(classOf[FetcherManager], token, nFetchers)
}
```

The manager can receive two messages: `AddToQueue`, which tells it to add a username to the queue of users whose followers need to be fetched, and `GiveMeWork`, emitted by the fetchers when they are unemployed.

The manager will be responsible for launching the fetchers, response interpreter, and follower extractor, as well as maintaining an internal queue of usernames and a set of usernames that we have seen:

```
// FetcherManager.scala

class FetcherManager(val token:Option[String], val nFetchers:Int)
extends Actor with ActorLogging {

  import FetcherManager._

  // queue of usernames whose followers we need to fetch
  val fetchQueue = mutable.Queue.empty[String]

  // set of users we have already fetched.
  val fetchedUsers = mutable.Set.empty[String]

  // Instantiate worker actors
  val followerExtractor = context.actorOf(
    FollowerExtractor.props(self))
  val responseInterpreter = context.actorOf(
    ResponseInterpreter.props(followerExtractor))
```

```
val fetchers = (0 until nFetchers).map { i =>
    context.actorOf(
        Fetcher.props(token, self, responseInterpreter))
}

// receive method when the actor has work:
// If we receive additional work, we just push it onto the
// queue.
// If we receive a request for work from a Fetcher,
// we pop an item off the queue. If that leaves the
// queue empty, we transition to the 'receiveWhileEmpty'
// method.
def receiveWhileNotEmpty:Receive = {
    case AddToQueue(login) => queueIfNotFetched(login)
    case GiveMeWork =>
        val login = fetchQueue.dequeue
        // send a Fetch message back to the sender.
        // we can use the `sender` method to reply to a message
        sender ! Fetcher.Fetch(login)
        if (fetchQueue.isEmpty) {
            context.become(receiveWhileEmpty)
        }
}

// receive method when the actor has no work:
// if we receive work, we add it onto the queue, transition
// to a state where we have work, and notify the fetchers
// that work is available.
def receiveWhileEmpty:Receive = {
    case AddToQueue(login) =>
        queueIfNotFetched(login)
        context.become(receiveWhileNotEmpty)
        fetchers.foreach { _ ! Fetcher.WorkAvailable }
    case GiveMeWork => // do nothing
}

// Start with an empty queue.
def receive = receiveWhileEmpty

def queueIfNotFetched(login:String) {
    if (! fetchedUsers(login)) {
        log.info(s"Pushing $login onto queue")
        // or do something useful...
        fetchQueue += login
    }
}
```

```
        fetchedUsers += login  
    }  
}  
}
```

We now have a fetcher manager. The rest of the code can remain the same, apart from the follower extractor. Instead of logging followers names, it must send AddToQueue messages to the manager. We will pass a reference to the manager at construction time:

```
// FollowerExtractor.scala
import akka.actor._
import org.json4s._
import org.json4s.native.JsonMethods._

object FollowerExtractor {

    // messages
    case class Extract(login:String, jsonResponse:JArray)

    // props factory method
    def props(manager:ActorRef) =
        Props(classOf[FollowerExtractor], manager)
}

class FollowerExtractor(manager:ActorRef)
extends Actor with ActorLogging {
    import FollowerExtractor._

    def receive = {
        case Extract(login, followerArray) =>
            val followers = extractFollowers(followerArray)
            followers foreach { f =>
                manager ! FetcherManager.AddToQueue(f)
            }
    }
}

def extractFollowers(followerArray:JArray) = for {
    JObject(follower) <- followerArray
    JField("login", JString(login)) <- follower
} yield login
}
```

The `main` method running all this is remarkably simple as all the code to instantiate actors has been moved to the `FetcherManager`. We just need to instantiate the manager and give it the first node in the network, and it will do the rest:

```
// FetchNetwork.scala
import akka.actor._

object FetchNetwork extends App {

    // Get token if exists
    val token = sys.env.get("GHTOKEN")

    val system = ActorSystem("GithubFetcher")
    val manager = system.actorOf(FetcherManager.props(token, 2))
    manager ! FetcherManager.AddToQueue("odersky")

}
```

Notice how we do not attempt to shut down the actor system anymore. We will just let it run, crawling the network, until we stop it or hit the authentication limit. Let's run this through SBT:

```
$ GHTOKEN="2502761d..." sbt "runMain FetchNetwork"
[INFO] [11/06/2015 06:31:04.614] [GithubFetcher-akka.actor.default-
dispatcher-2] [akka://GithubFetcher/user/$a] Pushing odersky onto queue
[INFO] [11/06/2015 06:31:05.563] [GithubFetcher-akka.actor.default-
dispatcher-4] [akka://GithubFetcher/user/$a] Pushing misto onto
queue[INFO] [11/06/2015 06:31:05.563] [GithubFetcher-akka.actor.default-
dispatcher-4] [akka://GithubFetcher/user/$a] Pushing gkossakowski onto
queue
^C
```

Our program does not actually do anything useful with the followers that it retrieves besides logging them. We could replace the `log.info` call to, for instance, store the nodes in a database or draw the graph to screen.

Fault tolerance

Real programs fail, and they fail in unpredictable ways. Akka, and the Scala community in general, favors planning explicitly for failure rather than trying to write infallible applications. A *fault tolerant* system is a system that can continue to operate when one or more of its components fails. The failure of an individual subsystem does not necessarily mean the failure of the application. How does this apply to Akka?

The actor model provides a natural unit to encapsulate failure: the actor. When an actor throws an exception while processing a message, the default behavior is for the actor to restart, but the exception does not leak out and affect the rest of the system. For instance, let's introduce an arbitrary failure in the response interpreter. We will modify the receive method to throw an exception when it is asked to interpret the response for `misto`, one of Martin Odersky's followers:

```
// ResponseInterpreter.scala
def receive = {
    case InterpretResponse("misto", r) =>
        throw new IllegalStateException("custom error")
    case InterpretResponse(login, r) => interpret(login, r)
}
```

If you rerun the code through SBT, you will notice that an error gets logged. The program does not crash, however. It just continues as normal:

```
[ERROR] [11/07/2015 12:05:58.938] [GithubFetcher-akka.actor.default-
dispatcher-2] [akka://GithubFetcher/user/$a/$b] custom error
java.lang.IllegalStateException: custom error
at ResponseInterpreter$ ...
[INFO] [11/07/2015 12:05:59.117] [GithubFetcher-akka.actor.default-
dispatcher-2] [akka://GithubFetcher/user/$a] Pushing samfoo onto queue
```

None of the followers of `misto` will get added to the queue: he never made it past the `ResponseInterpreter` stage. Let's step through what happens when the exception gets thrown:

- The interpreter is sent the `InterpretResponse ("misto", ...)` message. This causes it to throw an exception and it dies. None of the other actors are affected by the exception.
- A fresh instance of the response interpreter is created with the same `Props` instance as the recently deceased actor.
- When the response interpreter has finished initializing, it gets bound to the same `ActorRef` as the deceased actor. This means that, as far as the rest of the system is concerned, nothing has changed.
- The mailbox is tied to `ActorRef` rather than the actor, so the new response interpreter will have the same mailbox as its predecessor, without the offending message.

Thus, if, for whatever reason, our crawler crashes when fetching or parsing the response for a user, the application will be minimally affected – we will just not fetch this user's followers.

Any internal state that an actor carries is lost when it restarts. Thus, if, for instance, the fetcher manager died, we would lose the current value of the queue and visited users. The risks associated with losing the internal state can be mitigated by the following:

- Adopting a different strategy for failure: we can, for instance, carry on processing messages without restarting the actor in the event of failure. Of course, this is of little use if the actor died because its internal state is inconsistent. In the next section, we will discuss how to change the failure recovery strategy.
- Backing up the internal state by writing it to disk periodically and loading from the backup on restart.
- Protecting actors that carry critical state by ensuring that all "risky" operations are delegated to other actors. In our crawler example, all the interactions with external services, such as querying the GitHub API and parsing the response, happen with actors that carry no internal state. As we saw in the previous example, if one of these actors dies, the application is minimally affected. By contrast, the precious fetcher manager is only allowed to interact with sanitized inputs. This is called the *error kernel* pattern: code likely to cause errors is delegated to kamikaze actors.

Custom supervisor strategies

The default strategy of restarting an actor on failure is not always what we want. In particular, for actors that carry a lot of data, we might want to resume processing after an exception rather than restarting the actor. Akka lets us customize this behavior by setting a *supervisor strategy* in the actor's supervisor.

Recall that all actors have parents, including the top-level actors, who are children of a special actor called the *user guardian*. By default, an actor's supervisor is his parent, and it is the supervisor who decides what happens to the actor on failure.

Thus, to change how an actor reacts to failure, you must set its parent's supervisor strategy. You do this by setting the `supervisorStrategy` attribute. The default strategy is equivalent to the following:

```
val supervisorStrategy = OneForOneStrategy() {  
    case _:ActorInitializationException => Stop  
    case _:ActorKilledException => Stop  
    case _:DeathPactException => Stop  
    case _:Exception => Restart  
}
```

There are two components to a supervisor strategy:

- `OneForOneStrategy` determines that the strategy applies only to the actor that failed. By contrast, we can use `AllForOneStrategy`, which applies the same strategy to all the supervisees. If a single child fails, all the children will be restarted (or stopped or resumed).
- A partial function mapping `Throwables` to a `Directive`, which is an instruction on what to do in response to a failure. The default strategy, for instance, maps `ActorInitializationException` (which happens if the constructor fails) to the `Stop` directive and (almost all) other exceptions to `Restart`.

There are four directives:

- `Restart`: This destroys the faulty actor and restarts it, binding the newborn actor to the old `ActorRef`. This clears the internal state of the actor, which may be a good thing (the actor might have failed because of some internal inconsistency).
- `Resume`: The actor just moves on to processing the next message in its inbox.
- `Stop`: The actor stops and is not restarted. This is useful in throwaway actors that you use to complete a single operation: if this operation fails, the actor is not needed any more.
- `Escalate`: The supervisor itself rethrows the exception, hoping that its supervisor will know what to do with it.

A supervisor does not have access to which of its children failed. Thus, if an actor has children that might require different recovery strategies, it is best to create a set of intermediate supervisor actors to supervise the different groups of children.

As an example of setting the supervisor strategy, let's tweak the `FetcherManager` supervisor strategy to adopt an all-for-one strategy and stop its children when one of them fails. We start with the relevant imports:

```
import akka.actor.SupervisorStrategy._
```

Then, we just need to set the `supervisorStrategy` attribute in the `FetcherManager` definition:

```
class FetcherManager(...) extends Actor with ActorLogging {  
  
    ...  
  
    override val supervisorStrategy = AllForOneStrategy() {  
        case _:ActorInitializationException => Stop  
        case _:ActorKilledException => Stop  
        case _:Exception => Stop  
    }  
  
    ...  
}
```

If you run this through SBT, you will notice that when the code comes across the custom exception thrown by the response interpreter, the system halts. This is because all the actors apart from the fetcher manager are now defunct.

Life-cycle hooks

Akka lets us specify code that runs in response to specific events in an actor's life, through *life-cycle hooks*. Akka defines the following hooks:

- `preStart()`: This runs after the actor's constructor has finished but before it starts processing messages. This is useful to run initialization code that depends on the actor being fully constructed.
- `postStop()`: This runs when the actor dies after it has stopped processing messages. This is useful to run cleanup code before terminating the actor.
- `preRestart(reason: Throwable, message: Option[Any])`: This is called just after an actor receives an order to restart. The `preRestart` method has access to the exception that was thrown and to the offending message, allowing for corrective action. The default behavior of `preRestart` is to stop each child and then call `postStop`.
- `postRestart(reason: Throwable)`: This is called after an actor has restarted. The default behavior is to call `preStart()`.

Let's use system hooks to persist the state of `FetcherManager` between runs of the programs. You can find the code examples for this section in the `chap09/ghub_crawler_fault_tolerant` directory in the sample code provided with this book (<https://github.com/pbugnion/s4ds>). This will make the fetcher manager fault-tolerant. We will use `postStop` to write the current queue and set of visited users to text files and `preStart` to read these text files from the disk. Let's start by importing the libraries necessary to read and write files:

```
// FetcherManager.scala

import scala.io.Source
import scala.util._
import java.io._
```

We will store the names of the two text files in which we persist the state in the `FetcherManager` companion object (a better approach would be to store them in a configuration file):

```
// FetcherManager.scala
object FetcherManager {

    ...
    val fetchedUsersFileName = "fetched-users.txt"
    val fetchQueueFileName = "fetch-queue.txt"
}
```

In the `preStart` method, we load both the set of fetched users and the backlog of users to fetch from the text files, and in the `postStop` method, we overwrite these files with the new values of these data structures:

```
class FetcherManager(
    val token:Option[String], val nFetchers:Int
) extends Actor with ActorLogging {

    ...
    /** pre-start method: load saved state from text files */
    override def preStart {
        log.info("Running pre-start on fetcher manager")

        loadFetchedUsers
        log.info(
            s"Read ${fetchedUsers.size} visited users from source"
        )
    }

    loadFetchQueue
```

```
log.info(
    s"Read ${fetchQueue.size} users in queue from source"
)

// If the saved state contains a non-empty queue,
// alert the fetchers so they can start working.
if (fetchQueue.nonEmpty) {
    context.become(receiveWhileNotEmpty)
    fetchers.foreach { _ ! Fetcher.WorkAvailable }
}

/** Dump the current state of the manager */
override def postStop {
    log.info("Running post-stop on fetcher manager")
    saveFetchedUsers
    saveFetchQueue
}

/* Helper methods to load from and write to files */
def loadFetchedUsers {
    val fetchedUsersSource = Try {
        Source.fromFile(fetchedUsersFileName)
    }
    fetchedUsersSource.foreach { s =>
        try s.getLines.foreach { l => fetchedUsers += l }
        finally s.close
    }
}

def loadFetchQueue {
    val fetchQueueSource = Try {
        Source.fromFile(fetchQueueFileName)
    }
    fetchQueueSource.foreach { s =>
        try s.getLines.foreach { l => fetchQueue += l }
        finally s.close
    }
}

def saveFetchedUsers {
    val fetchedUsersFile = new File(fetchedUsersFileName)
    val writer = new BufferedWriter(
```

```
    new FileWriter(fetchedUsersFile))
fetchedUsers.foreach { user => writer.write(user + "\n") }
writer.close()
}

def saveFetchQueue {
    val queueUsersFile = new File(fetchQueueFileName)
    val writer = new BufferedWriter(
        new FileWriter(queueUsersFile))
    fetchQueue.foreach { user => writer.write(user + "\n") }
    writer.close()
}

...
}
```

Now that we save the state of the crawler when it shuts down, we can put a better termination condition for the program than simply interrupting the program once we get bored. In production, we might halt the crawler when we have enough names in a database, for instance. In this example, we will simply let the crawler run for 30 seconds and then shut it down.

Let's modify the main method:

```
// FetchNetwork.scala
import akka.actor._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._

object FetchNetwork extends App {

    // Get token if exists
    val token = sys.env.get("GHTOKEN")

    val system = ActorSystem("GithubFetcher")
    val manager = system.actorOf(FetcherManager.props(token, 2))

    manager ! FetcherManager.AddToQueue("odersky")

    system.scheduler.scheduleOnce(30.seconds) { system.shutdown }

}
```

After 30 seconds, we just call `system.shutdown`, which stops all the actors recursively. This will stop the fetcher manager, calling the `postStop` life cycle hook. After one run of the program, I have 2,164 names in the `fetched-users.txt` file. Running it again increases this number to 3,728 users.

We could improve fault tolerance further by making the fetcher manager dump the data structures at regular intervals while the code runs. As writing to the disk (or to a database) carries a certain element of risk (What if the database server goes down or the disk is full?) it would be better to delegate writing the data structures to a custom actor rather than endangering the manager.

Our crawler has one minor problem: when the fetcher manager stops, it stops the fetcher actors, response interpreter, and follower extractor. However, none of the users currently going through these actors are stored. This also results in a small number of undelivered messages at the end of the code: if the response interpreter stops before a fetcher, the fetcher will try to deliver to a non-existent actor. This only accounts for a small number of users. To recover these login names, we can create a reaper actor whose job is to coordinate the killing of all the worker actors in the correct order and harvest their internal state. This pattern is documented in a blog post by *Derek Wyatt* (<http://letitcrash.com/post/30165507578/shutdown-patterns-in akka-2>).

What we have not talked about

Akka is a very rich ecosystem, far too rich to do it justice in a single chapter. There are some important parts of the toolkit that you will need, but we have not covered them here. We will give brief descriptions, but you can refer to the Akka documentation for more details:

- The ask operator, `?`, offers an alternative to the tell operator, `!`, that we have used to send messages to actors. Unlike "tell", which just fires a message to an actor, the ask operator expects a response. This is useful when we need to ask actors questions rather than just telling them what to do. The ask pattern is documented at http://doc.akka.io/docs/akka/snapshot/scala/actors.html#Ask__Send-And-Receive-Future.
- Deathwatch allows actors to watch another actor and receive a message when it dies. This is useful for actors that might depend on another actor but not be its direct supervisor. This is documented at http://doc.akka.io/docs/akka/snapshot/scala/actors.html#Lifecycle_Monitoring_aka_DeathWatch.

- In our crawler, we passed references to actors explicitly through the constructor. We can also look up actors using the actor hierarchy with a syntax reminiscent of files in a filesystem at http://doc.akka.io/docs/akka/snapshot/scala/actors.html#Identifying_Actors_via_Actor_Selection.
- We briefly explored how to implement stateful actors with different receive methods and using `context.become` to switch between them. Akka offers a more powerful alternative, based on finite state machines, to encode a more complex set of states and transitions: <http://doc.akka.io/docs/akka/snapshot/scala/fsm.html>.
- We have not discussed distributing actor systems across several nodes in this chapter. The message passing architecture works well with distributed setups: <http://doc.akka.io/docs/akka/2.4.0/common/cluster.html>.

Summary

In this chapter, you learned how to weave actors together to tackle a difficult concurrent problem. More importantly, we saw how Akka's actor framework encourages us to think about concurrent problems in terms of many separate chunks of encapsulated mutable data, synchronized through message passing. Akka makes concurrent programming easier to reason about and more fun.

References

Derek Wyatt's book, *Akka Concurrency*, is a fantastic introduction to Akka. It should definitely be the first stop for anyone wanting to do serious Akka programming.

The **LET IT CRASH** blog (<http://letitcrash.com>) is the official Akka blog, and contains many examples of idioms and patterns to solve common issues.

10

Distributed Batch Processing with Spark

In *Chapter 4, Parallel Collections and Futures*, we discovered how to use parallel collections for "embarrassingly" parallel problems: problems that can be broken down into a series of tasks that require no (or very little) communication between the tasks.

Apache Spark provides behavior similar to Scala parallel collections (and much more), but, instead of distributing tasks across different CPUs on the same computer, it allows the tasks to be distributed across a computer cluster. This provides arbitrary horizontal scalability, since we can simply add more computers to the cluster.

In this chapter, we will learn the basics of Apache Spark and use it to explore a set of emails, extracting features with the view of building a spam filter. We will explore several ways of actually building a spam filter in *Chapter 12, Distributed Machine Learning with MLlib*.

Installing Spark

In previous chapters, we included dependencies by specifying them in a `build.sbt` file, and relying on SBT to fetch them from the Maven Central repositories. For Apache Spark, downloading the source code or pre-built binaries explicitly is more common, since Spark ships with many command line scripts that greatly facilitate launching jobs and interacting with a cluster.

Head over to <http://spark.apache.org/downloads.html> and download Spark 1.5.2, choosing the "pre-built for Hadoop 2.6 or later" package. You can also build Spark from source if you need customizations, but we will stick to the pre-built version since it requires no configuration.

Clicking **Download** will download a tarball, which you can unpack with the following command:

```
$ tar xzf spark-1.5.2-bin-hadoop2.6.tgz
```

This will create a `spark-1.5.2-bin-hadoop2.6` directory. To verify that Spark works correctly, navigate to `spark-1.5.2-bin-hadoop2.6/bin` and launch the Spark shell using `./spark-shell`. This is just a Scala shell with the Spark libraries loaded.

You may want to add the `bin/` directory to your system path. This will let you call the scripts in that directory from anywhere on your system, without having to reference the full path. On Linux or Mac OS, you can add variables to the system path by entering the following line in your shell configuration file (`.bash_profile` on Mac OS, and `.bashrc` or `.bash_profile` on Linux):

```
export PATH=/path/to/spark/bin:$PATH
```

The changes will take effect in new shell sessions. On Windows (if you use PowerShell), you need to enter this line in the `profile.ps1` file in the `WindowsPowerShell` folder in `Documents`:

```
$env:Path += ";C:\Program Files\GnuWin32\bin"
```

If this worked correctly, you should be able to open a Spark shell in any directory on your system by just typing `spark-shell` in a terminal.

Acquiring the example data

In this chapter, we will explore the Ling-Spam email dataset (The original dataset is described at <http://csmining.org/index.php/ling-spam-datasets.html>). Download the dataset from <http://data.scala4datascience.com/ling-spam.tar.gz> (or `ling-spam.zip`, depending on your preferred mode of compression), and unpack the contents to the directory containing the code examples for this chapter. The archive contains two directories, `spam/` and `ham/`, containing the spam and legitimate emails, respectively.

Resilient distributed datasets

Spark expresses all computations as a sequence of transformations and actions on distributed collections, called **Resilient Distributed Datasets (RDD)**. Let's explore how RDDs work with the Spark shell. Navigate to the examples directory and open a Spark shell as follows:

```
$ spark-shell
scala>
```

Let's start by loading an email in an RDD:

```
scala> val email = sc.textFile("ham/9-463msg1.txt")
email: rdd.RDD[String] = MapPartitionsRDD[1] at textFile
```

`email` is an RDD, with each element corresponding to a line in the input file. Notice how we created the RDD by calling the `textFile` method on an object called `sc`:

```
scala> sc
spark.SparkContext = org.apache.spark.SparkContext@459bf87c
```

`sc` is a `SparkContext` instance, an object representing the entry point to the Spark cluster (for now, just our local machine). When we start a Spark shell, a context is created and bound to the variable `sc` automatically.

Let's split the email into words using `flatMap`:

```
scala> val words = email.flatMap { line => line.split("\\s") }
words: rdd.RDD[String] = MapPartitionsRDD[2] at flatMap
```

This will feel natural if you are familiar with collections in Scala: the `email` RDD behaves just like a list of strings. Here, we split using the regular expression `\s`, denoting white space characters. Instead of using `flatMap` explicitly, we can also manipulate RDDs using Scala's syntactic sugar:

```
scala> val words = for {
    line <- email
    word <- line.split("\\s")
} yield word
words: rdd.RDD[String] = MapPartitionsRDD[3] at flatMap
```

Let's inspect the results. We can use `.take(n)` to extract the first n elements of an RDD:

```
scala> words.take(5)
Array[String] = Array(subject:, tsd98, workshop, -, -)
```

We can also use `.count` to get the number of elements in an RDD:

```
scala> words.count
Long = 939
```

RDDs support many of the operations supported by collections. Let's use `filter` to remove punctuation from our email. We will remove all words that contain any non-alphanumeric character. We can do this by filtering out elements that match this *regular expression* anywhere in the word: `[^a-zA-Z0-9]`.

```
scala> val nonAlphaNumericPattern = "[^a-zA-Z0-9]".r
nonAlphaNumericPattern: Regex = [^a-zA-Z0-9]

scala> val filteredWords = words.filter {
    word => nonAlphaNumericPattern.findFirstIn(word) == None
}
filteredWords: rdd.RDD[String] = MapPartitionsRDD[4] at filter

scala> filteredWords.take(5)
Array[String] = Array(tsd98, workshop, 2nd, call, paper)

scala> filteredWords.count
Long = 627
```

In this example, we created an RDD from a text file. We can also create RDDs from Scala iterables using the `sc.parallelize` method available on a Spark context:

```
scala> val words = "the quick brown fox jumped over the dog".split(" ")
words: Array[String] = Array(the, quick, brown, fox, ...)

scala> val wordsRDD = sc.parallelize(words)
wordsRDD: RDD[String] = ParallelCollectionRDD[1] at parallelize at
<console>:23
```

This is useful for debugging and for trialling behavior in the shell. The counterpart to parallelize is the `.collect` method, which converts an RDD to a Scala array:

```
scala> val wordLengths = wordsRDD.map { _.length }
wordLengths: RDD[Int] = MapPartitionsRDD[2] at map at <console>:25

scala> wordLengths.collect
Array[Int] = Array(3, 5, 5, 3, 6, 4, 3, 3)
```

The `.collect` method requires the entire RDD to fit in memory on the master node. It is thus either used for debugging with a reduced dataset, or at the end of a pipeline that trims down a dataset.

As you can see, RDDs offer an API much like Scala iterables. The critical difference is that RDDs are *distributed* and *resilient*. Let's explore what this means in practice.

RDDs are immutable

You cannot change an RDD once it is created. All operations on RDDs either create new RDDs or other Scala objects.

RDDs are lazy

When you execute operations like `map` and `filter` on a Scala collection in the interactive shell, the REPL prints the values of the new collection to screen. The same isn't true of Spark RDDs. This is because operations on RDDs are lazy: they are only evaluated when needed.

Thus, when we write:

```
val email = sc.textFile(...)
val words = email.flatMap { line => line.split("\\s") }
```

We are creating an RDD, `words` that knows how to build itself from its parent RDD, `email`, which, in turn, knows that it needs to read a text file and split it into lines. However, none of the commands actually happen until we force the evaluation of the RDDs by calling an *action* to return a Scala object. This is most evident if we try to read from a non-existent text file:

```
scala> val inp = sc.textFile("nonexistent")
inp: rdd.RDD[String] = MapPartitionsRDD[5] at textFile
```

We can create the RDD without a hitch. We can even define further transformations on the RDD. The program crashes only when these transformations are finally evaluated:

```
scala> inp.count // number of lines
org.apache.hadoop.mapred.InvalidInputException: Input path does not
exist: file:/Users/pascal/...
```

The action `.count` is expected to return the number of elements in our RDD as an integer. Spark has no choice but to evaluate `inp`, which results in an exception.

Thus, it is probably more appropriate to think of an RDD as a pipeline of operations, rather than a more traditional collection.

RDDs know their lineage

RDDs can only be constructed from stable storage (for instance, by loading data from a file that is present on every node in the Spark cluster), or through a set of transformations based on other RDDs. Since RDDs are lazy, they need to know how to build themselves when needed. They do this by knowing who their parent RDD is, and what operation they need to apply to the parent. This is a well-defined process since the parent RDD is immutable.

The `toDebugString` method provides a diagram of how an RDD is constructed:

```
scala> filteredWords.toDebugString
(2) MapPartitionsRDD[6] at filter at <console>:27 []
|  MapPartitionsRDD[3] at flatMap at <console>:23 []
|  MapPartitionsRDD[1] at textFile at <console>:21 []
|  ham/9-463msg1.txt HadoopRDD[0] at textFile at <console>:21 []
```

RDDs are resilient

If you run an application on a single computer, you generally don't need to worry about hardware failure in your application: if the computer fails, your application is doomed anyway.

Distributed architectures should, by contrast, be fault-tolerant: the failure of a single machine should not crash the entire application. Spark RDDs are built with fault tolerance in mind. Let's imagine that one of the worker nodes fails, causing the destruction of some of the data associated with an RDD. Since the Spark RDD knows how to build itself from its parent, there is no permanent data loss: the elements that were lost can just be re-computed when needed on another computer.

RDDs are distributed

When you construct an RDD, for instance from a text file, Spark will split the RDD into a number of partitions. Each partition will be entirely localized on a single machine (though there is, in general, more than one partition per machine).

Many transformations on RDDs can be executed on each partition independently. For instance, when performing a `.map` operation, a given element in the output RDD depends on a single element in the parent: data does not need to be moved between partitions. The same is true of `.flatMap` and `.filter` operations. This means that the partition in the RDD produced by one of these operations depends on a single partition in the parent RDD.

On the other hand, a `.distinct` transformation, which removes all duplicate elements from an RDD, requires the data in a given partition to be compared to the data in every other partition. This requires *shuffling* the data across the nodes. Shuffling, especially for large datasets, is an expensive operation and should be avoided if possible.

Transformations and actions on RDDs

The set of operations supported by an RDD can be split into two categories:

- **Transformations** create a new RDD from the current one. Transformations are lazy: they are not evaluated immediately.
- **Actions** force the evaluation of an RDD, and normally return a Scala object, rather than an RDD, or have some form of side-effect. Actions are evaluated immediately, triggering the execution of all the transformations that make up this RDD.

In the tables below, we give some examples of useful transformations and actions. For a full, up-to-date list, consult the Spark documentation (<http://spark.apache.org/docs/latest/programming-guide.html#rdd-operations>).

For the examples in these tables, we assume that you have created an RDD with:

```
scala> val rdd = sc.parallelize(List("quick", "brown", "quick", "dog"))
```

The following table lists common transformations on an RDD. Recall that transformations always generate a new RDD, and that they are lazy operations:

Transformation	Notes	Example (assuming rdd is { "quick", "brown", "quick", "dog" })
rdd.map(func)		rdd.map { _.size } // => { 5, 5, 5, 3 }
rdd.filter(pred)		rdd.filter { _.length < 4 } // => { "dog" }
rdd.flatMap(func)		rdd.flatMap { _.toCharArray } // => { 'q', 'u', 'i', 'c', 'k', 'b', 'r', 'o' ... }
rdd.distinct()	Remove duplicate elements in RDD.	rdd.distinct // => { "dog", "brown", "quick" }
rdd.pipe(command, [envVars])	Pipe through an external program. RDD elements are written, line-by-line, to the process's stdin. The output is read from stdout.	rdd.pipe("tr a-z A-Z") // => { "QUICK", "BROWN", "QUICK", "DOG" }

The following table describes common actions on RDDs. Recall that actions always generate a Scala type or cause a side-effect, rather than creating a new RDD. Actions force the evaluation of the RDD, triggering the execution of the transformations underpinning the RDD.

Action	Notes	Example (assuming rdd is { "quick", "brown", "quick", "dog" })
rdd.first	First element in the RDD.	rdd.first // => quick
rdd.collect	Transform the RDD to an array (the array must be able to fit in memory on the master node).	rdd.collect // => Array[String] ("quick", "brown", "quick", "dog")
rdd.count	Number of elements in the RDD.	rdd.count // => 4

Action	Nodes	Example (assuming rdd is { "quick", "brown", "quick", "dog" })
rdd.countByValue	Map of element to the number of times this element occurs. The map must fit on the master node.	rdd.countByValue // => Map(quick -> 2, brown -> 1, dog -> 1)
rdd.take(n)	Return an array of the first n elements in the RDD.	rdd.take(2) // => Array(quick, brown)
rdd. takeOrdered(n:Int) (implicit ordering: Ordering[T])	Top n elements in the RDD according to the element's default ordering, or the ordering passed as second argument. See the Scala docs for Ordering for how to define custom comparison functions (http://www.scala-lang.org/api/current/index.html#scala.math.Ordering).	rdd.takeOrdered(2) // => Array(brown, dog) rdd.takeOrdered(2) (Ordering.by { _.size }) // => Array[String] = Array(dog, quick)
rdd.reduce(func)	Reduce the RDD according to the specified function. Uses the first element in the RDD as the base. func should be commutative and associative.	rdd.map { _.size }.reduce { _ + _ } // => 18

Action	Nodes	Example (assuming rdd is { "quick", "brown", "quick", "dog" })
rdd. aggregate(zeroValue) (seqOp, combOp)	Reduction for cases where the reduction function returns a value of type different to the RDD's type. In this case, we need to provide a function for reducing within a single partition (seqOp) and a function for combining the value of two partitions (combOp).	rdd.aggregate(0) (_ + _ .size, _ + _) // => 18

Persisting RDDs

We have learned that RDDs only retain the sequence of operations needed to construct the elements, rather than the values themselves. This, of course, drastically reduces memory usage since we do not need to keep intermediate versions of our RDDs in memory. For instance, let's assume we want to trawl through transaction logs to identify all the transactions that occurred on a particular account:

```
val allTransactions = sc.textFile("transaction.log")
val interestingTransactions = allTransactions.filter {
    _.contains("Account: 123456")
}
```

The set of all transactions will be large, while the set of transactions on the account of interest will be much smaller. Spark's policy of remembering *how* to construct a dataset, rather than the dataset itself, means that we never have all the lines of our input file in memory at any one time.

There are two situations in which we may want to avoid re-computing the elements of an RDD every time we use it:

- For interactive use: we might have detected fraudulent behavior on account "123456", and we want to investigate how this might have arisen. We will probably want to perform many different exploratory calculations on this RDD, without having to re-read the entire log file every time. It therefore makes sense to persist interestingTransactions.

- When an algorithm re-uses an intermediate result, or a dataset. A canonical example is logistic regression. In logistic regression, we normally use an iterative algorithm to find the 'optimal' coefficients that minimize the loss function. At every step in our iterative algorithm, we must calculate the loss function and its gradient from the training set. We should avoid re-computing the training set (or re-loading it from an input file) if at all possible.

Spark provides a `.persist` method on RDDs to achieve this. By calling `.persist` on an RDD, we tell Spark to keep the dataset in memory next time it is computed.

```
scala> words.persist  
rdd.RDD[String] = MapPartitionsRDD[3] at filter
```

Spark supports different levels of persistence, which you can tune by passing arguments to `.persist`:

```
scala> import org.apache.spark.storage.StorageLevel  
import org.apache.spark.storage.StorageLevel  
  
scala> interestingTransactions.persist(  
  StorageLevel.MEMORY_AND_DISK)  
rdd.RDD[String] = MapPartitionsRDD[3] at filter
```

Spark provides several persistence levels, including:

- `MEMORY_ONLY`: the default storage level. The RDD is stored in RAM. If the RDD is too big to fit in memory, parts of it will not persist, and will need to be re-computed on the fly.
- `MEMORY_AND_DISK`: As much of the RDD is stored in memory as possible. If the RDD is too big, it will spill over to disk. This is only worthwhile if the RDD is expensive to compute. Otherwise, re-computing it may be faster than reading from the disk.

If you persist several RDDs and run out of memory, Spark will clear the least recently used out of memory (either discarding them or saving them to disk, depending on the chosen persistence level). RDDs also expose an `unpersist` method to explicitly tell Spark than an RDD is not needed any more.

Persisting RDDs can have a drastic impact on performance. What and how to persist therefore becomes very important when tuning a Spark application. Finding the best persistence level generally requires some tinkering, benchmarking and experimentation. The Spark documentation provides guidelines on when to use which persistence level (<http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence>), as well as general tips on tuning memory usage (<http://spark.apache.org/docs/latest/tuning.html>).

Importantly, the `persist` method does not force the evaluation of the RDD. It just notifies the Spark engine that, next time the values in this RDD are computed, they should be saved rather than discarded.

Key-value RDDs

So far, we have only considered RDDs of Scala value types. RDDs of more complex data types support additional operations. Spark adds many operations for *key-value* RDDs: RDDs whose type parameter is a tuple (K, V) , for any type K and V .

Let's go back to our sample email:

```
scala> val email = sc.textFile("ham/9-463msg1.txt")
email: rdd.RDD[String] = MapPartitionsRDD[1] at textFile

scala> val words = email.flatMap { line => line.split("\\s") }
words: rdd.RDD[String] = MapPartitionsRDD[2] at flatMap
```

Let's persist the `words` RDD in memory to avoid having to re-read the `email` file from disk repeatedly:

```
scala> words.persist
```

To access key-value operations, we just need to apply a transformation to our RDD that creates key-value pairs. Let's use the words as keys. For now, we will just use 1 for every value:

```
scala> val wordsKeyValue = words.map { _ -> 1 }
wordsKeyValue: rdd.RDD[(String, Int)] = MapPartitionsRDD[32] at map

scala> wordsKeyValue.first
(String, Int) = (Subject:,1)
```

Key-value RDDs support several operations besides the core RDD operations. These are added through an implicit conversion, using the "pimp my library" pattern that we explored in *Chapter 5, Scala and SQL through JDBC*. These additional transformations fall into two broad categories: *by-key* transformations and *joins* between RDDs.

By-key transformations are operations that aggregate the values corresponding to the same key. For instance, we can count the number of times each word appears in our email using `reduceByKey`. This method takes all the values that belong to the same key and combines them using a user-supplied function:

```
scala> val wordCounts = wordsKeyValue.reduceByKey { _ + _ }
wordCounts: rdd.RDD[(String, Int)] = ShuffledRDD[35] at reduceByKey

scala> wordCounts.take(5).foreach { println }
(university,6)
(under,1)
(call,3)
(paper,2)
(chasm,2)
```

Note that `reduceByKey` requires (in general) shuffling the RDD, since not every occurrence of a given key will be in the same partition:

```
scala> wordCounts.toDebugString
(2) ShuffledRDD[36] at reduceByKey at <console>:30 []
+- (2) MapPartitionsRDD[32] at map at <console>:28 []
  |  MapPartitionsRDD[7] at flatMap at <console>:23 []
  |    CachedPartitions: 2; MemorySize: 50.3 KB;
ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
  |  MapPartitionsRDD[3] at textFile at <console>:21 []
  |    CachedPartitions: 2; MemorySize: 5.1 KB;
ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
  |  ham/9-463msg1.txt HadoopRDD[2] at textFile at <console>:21 []
```

Note that key-value RDDs are not like Scala Maps: the same key can occur multiple times, and they do not support $O(1)$ lookup. A key-value RDD can be transformed to a Scala map using the `.collectAsMap` action:

```
scala> wordCounts.collectAsMap
scala.collection.Map[String,Int] = Map(follow -> 2, famous -> 1...)
```

This requires pulling the entire RDD onto the main Spark node. You therefore need to have enough memory on the main node to house the map. This is often the last stage in a pipeline that filters a large RDD to just the information that we need.

There are many by-key operations, which we describe in the table below. For the examples in the table, we assume that `rdd` is created as follows:

```
scala> val words = sc.parallelize(List("quick", "brown", "quick", "dog"))
words: RDD[String] = ParallelCollectionRDD[25] at parallelize at
<console>:21

scala> val rdd = words.map { word => (word -> word.size) }
rdd: RDD[(String, Int)] = MapPartitionsRDD[26] at map at <console>:23

scala> rdd.collect
Array[(String, Int)] = Array((quick,5), (brown,5), (quick,5), (dog,3))
```

Transformation	Notes	Example (assumes rdd is { quick -> 5, brown -> 5, quick -> 5, dog -> 3 })
<code>rdd.mapValues</code>	Apply an operation to the values.	<code>rdd.mapValues { _ * 2 } // => { quick -> 10, brown -> 10, quick -> 10, dog -> 6 }</code>
<code>rdd.groupByKey</code>	Return a key-value RDD in which values corresponding to the same key are grouped into iterables.	<code>rdd.groupByKey // => { quick -> Iterable(5, 5), brown -> Iterable(5), dog -> Iterable(3) }</code>
<code>rdd.reduceByKey(func)</code>	Return a key-value RDD in which values corresponding to the same key are combined using a user-supplied function.	<code>rdd.reduceByKey { _ + _ } // => { quick -> 10, brown -> 5, dog -> 3 }</code>
<code>rdd.keys</code>	Return an RDD of the keys.	<code>rdd.keys // => { quick, brown, quick, dog }</code>
<code>rdd.values</code>	Return an RDD of the values.	<code>rdd.values // => { 5, 5, 5, 3 }</code>

The second category of operations on key-value RDDs involves joining different RDDs together by key. This is somewhat similar to SQL joins, where the keys are the column being joined on. Let's load a spam email and apply the same transformations we applied to our ham email:

```
scala> val spamEmail = sc.textFile("spam/spmsgb17.txt")
spamEmail: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[52] at
textFile at <console>:24

scala> val spamWords = spamEmail.flatMap { _.split("\\s") }
spamWords: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[53] at
flatMap at <console>:26

scala> val spamWordCounts = spamWords.map {
    _ -> 1 }.reduceByKey { _ + _ }
spamWordCounts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[55]
at reduceByKey at <console>:30

scala> spamWordCounts.take(5).foreach { println }
(banner,3)
(package,14)
(call,1)
(country,2)
(officer,1)
```

Both `spamWordCounts` and `wordCounts` are key-value RDDs for which the keys correspond to unique words in the message, and the values are the number of times that word occurs. There will be some overlap in keys between `spamWordCounts` and `wordCounts`, since the emails will share many of the same words. Let's do an *inner join* between those two RDDs to get the words that occur in both emails:

```
scala> val commonWordCounts = wordCounts.join(spamWordCounts)
res93: rdd.RDD[(String, (Int, Int))] = MapPartitionsRDD[58] at join at
<console>:41

scala> commonWordCounts.take(5).foreach { println }
(call,(3,1))
(include,(6,2))
(minute,(2,1))
(form,(1,7))
((,(36,5))
```

The values in the RDD resulting from an inner join will be pairs. The first element in the pair is the value for that key in the first RDD, and the second element is the value for that key in the second RDD. Thus, the word *call* occurs three times in the legitimate email and once in the spam email.

Spark supports all four join types. For instance, let's perform a left join:

```
scala> val leftWordCounts = wordCounts.leftOuterJoin(spamWordCounts)
leftWordCounts: rdd.RDD[(String, (Int, Option[Int]))] =
MapPartitionsRDD[64] at leftOuterJoin at <console>:40

scala> leftWordCounts.take(5).foreach { println }
(call, (3,Some(1)))
(paper, (2,None))
(chasm, (2,None))
(antonio, (1,None))
(event, (3,None))
```

Notice that the second element in our pair has type `Option[Int]`, to accommodate keys absent in `spamWordCounts`. The word *paper*, for instance, occurs twice in the legitimate email and never in the spam email. In this case, it is more useful to have zeros to indicate absence, rather than `None`. Replacing `None` with a default value is simple with `getOrElse`:

```
scala> val defaultWordCounts = leftWordCounts.mapValues {
  case(leftValue, rightValue) => (leftValue, rightValue.getOrElse(0))
}
org.apache.spark.rdd.RDD[(String, (Int, Option[Int]))] =
MapPartitionsRDD[64] at leftOuterJoin at <console>:40

scala> defaultWordCounts.take(5).foreach { println }
(call, (3,1))
(paper, (2,0))
(chasm, (2,0))
(antonio, (1,0))
(event, (3,0))
```

The table below lists the most common joins on key-value RDDs:

Transformation	Result (assuming rdd1 is { quick -> 1, brown -> 2, quick -> 3, dog -> 4 } and rdd2 is { quick -> 78, brown -> 79, fox -> 80 })
rdd1.join(rdd2)	{ quick -> (1, 78), quick -> (3, 78), brown -> (2, 79) }
rdd1.leftOuterJoin(rdd2)	{ dog -> (4, None), quick -> (1, Some(78)), quick -> (3, Some(78)), brown -> (2, Some(79)) }
rdd1.rightOuterJoin(rdd2)	{ quick -> (Some(1), 78), quick -> (Some(3), 78), brown -> (Some(2), 79), fox -> (None, 80) }
rdd1.fullOuterJoin(rdd2)	{ dog -> (Some(4), None), quick -> (Some(1), Some(78)), quick -> (Some(3), Some(78)), brown -> (Some(2), Some(79)), fox -> (None, Some(80)) }

For a complete list of transformations, consult the API documentation for `PairRDDFunctions`, <http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions>.

Double RDDs

In the previous section, we saw that Spark adds functionality to key-value RDDs through an implicit conversion. Similarly, Spark adds statistics functionality to RDDs of doubles. Let's extract the word frequencies for the ham message, and convert the values from integers to doubles:

```
scala> val counts = wordCounts.values.map { _.toDouble }
counts: rdd.RDD[Double] = MapPartitionsRDD[9] at map
```

We can then get summary statistics using the `.stats` action:

```
scala> counts.stats
org.apache.spark.util.StatCounter = (count: 397, mean: 2.365239, stdev: 5.740843, max: 72.000000, min: 1.000000)
```

Thus, the most common word appears 72 times. We can also use the `.histogram` action to get an idea of the distribution of values:

```
scala> counts.histogram(5)
(Array(1.0, 15.2, 29.4, 43.6, 57.8, 72.0), Array(391, 1, 3, 1, 1))
```

The `.histogram` method returns a pair of arrays. The first array indicates the bounds of the histogram bins, and the second is the count of elements in that bin. Thus, there are 391 words that appear less than 15.2 times. The distribution of words is very skewed, such that a histogram with regular-sized bin is not really appropriate. We can, instead, pass in custom bins by passing an array of bin edges to the `histogram` method. For instance, we might distribute the bins logarithmically:

```
scala> counts.histogram(Array(1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0,  
128.0))  
res13: Array[Long] = Array(264, 94, 22, 11, 1, 4, 1)
```

Building and running standalone programs

So far, we have interacted exclusively with Spark through the Spark shell. In the section that follows, we will build a standalone application and launch a Spark program either locally or on an EC2 cluster.

Running Spark applications locally

The first step is to write the `build.sbt` file, as you would if you were running a standard Scala script. The Spark binary that we downloaded needs to be run against Scala 2.10 (You need to compile Spark from source to run against Scala 2.11. This is not difficult to do, just follow the instructions on <http://spark.apache.org/docs/latest/building-spark.html#building-for-scala-211>).

```
// build.sbt file  
  
name := "spam_mi"  
  
scalaVersion := "2.10.5"  
  
libraryDependencies ++= Seq(  
  "org.apache.spark" %% "spark-core" % "1.4.1"  
)
```

We then run `sbt package` to compile and build a jar of our program. The jar will be built in `target/scala-2.10/`, and called `spam_mi_2.10-0.1-SNAPSHOT.jar`. You can try this with the example code provided for this chapter.

We can then run the jar locally using the `spark-submit` shell script, available in the `bin/` folder in the Spark installation directory:

```
$ spark-submit target/scala-2.10/spam_mi_2.10-0.1-SNAPSHOT.jar  
... runs the program
```

The resources allocated to Spark can be controlled by passing arguments to `spark-submit`. Use `spark-submit --help` to see the full list of arguments.

If the Spark programs has dependencies (for instance, on other Maven packages), it is easiest to bundle them into the application jar using the *SBT assembly* plugin. Let's imagine that our application depends on `breeze-viz`. The `build.sbt` file now looks like:

```
// build.sbt  
  
name := "spam_mi"  
  
scalaVersion := "2.10.5"  
  
libraryDependencies ++= Seq(  
  "org.apache.spark" %% "spark-core" % "1.5.2" % "provided",  
  "org.scalanlp" %% "breeze" % "0.11.2",  
  "org.scalanlp" %% "breeze-viz" % "0.11.2",  
  "org.scalanlp" %% "breeze-natives" % "0.11.2"  
)
```

SBT assembly is an SBT plugin that builds *fat jars*: jars that contain not only the program itself, but all the dependencies for the program.

Note that we marked Spark as "provided" in the list of dependencies, which means that Spark itself will not be included in the jar (it is provided by the Spark environment anyway). To include the SBT assembly plugin, create a file called `assembly.sbt` in the project/ directory, with the following line:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.0")
```

You will need to re-start SBT for the changes to take effect. You can then create the assembly jar using the `assembly` command in SBT. This will create a jar called `spam_mi-assembly-0.1-SNAPSHOT.jar` in the `target/scala-2.10` directory. You can run this jar using `spark-submit`.

Reducing logging output and Spark configuration

Spark is, by default, very verbose. The default log-level is set to `INFO`. To avoid missing important messages, it is useful to change the log settings to `WARN`. To change the default log level system-wide, go into the `conf` directory in the directory in which you installed Spark. You should find a file called `log4j.properties.template`. Rename this file to `log4j.properties` and look for the following line:

```
log4j.rootCategory=INFO, console
```

Change this line to:

```
log4j.rootCategory=WARN, console
```

There are several other configuration files in that directory that you can use to alter Spark's default behavior. For a full list of configuration options, head over to <http://spark.apache.org/docs/latest/configuration.html>.

Running Spark applications on EC2

Running Spark locally is useful for testing, but the whole point of using a distributed framework is to run programs harnessing the power of several different computers. We can set Spark up on any set of computers that can communicate with each other using HTTP. In general, we also need to set up a distributed file system like HDFS, so that we can share input files across the cluster. For the purpose of this example, we will set Spark up on an Amazon EC2 cluster.

Spark comes with a shell script, `ec2/spark-ec2`, for setting up an EC2 cluster and installing Spark. It will also install HDFS. You will need an account with Amazon Web Services (AWS) to follow these examples (<https://aws.amazon.com>). You will need the AWS access key and secret key, which you can access through the **Account / Security Credentials / Access Credentials** menu in the AWS web console. You need to make these available to the `spark-ec2` script through environment variables. Inject them into your current session as follows:

```
$ export AWS_ACCESS_KEY_ID=ABCDEF...
$ export AWS_SECRET_ACCESS_KEY=2dEf...
```

You can also write these lines into the configuration script for your shell (your `.bashrc` file, or equivalent), to avoid having to re-enter them every time you run the `setup-ec2` script. We discussed environment variables in *Chapter 6, Slick – A Functional Interface for SQL*.

You will also need to create a key pair by clicking on **Key Pairs** in the EC2 web console, creating a new key pair and downloading the certificate file. I will assume you named the key pair `test_ec2` and the certificate file `test_ec2.pem`. Make sure that the key pair is created in the *N. Virginia* region (by choosing the correct region in the upper right corner of the EC2 Management console), to avoid having to specify the region explicitly in the rest of this chapter. You will need to set access permissions on the certificate file to user-readable only:

```
$ chmod 400 test_ec2.pem
```

We are now ready to launch the cluster. Navigate to the `ec2` directory and run:

```
$ ./spark-ec2 -k test_ec2 -i ~/path/to/certificate/test_ec2.pem -s 2  
launch test_cluster
```

This will create a cluster called `test_cluster` with a master and two slaves. The number of slaves is set through the `-s` command line argument. The cluster will take a while to start up, but you can verify that the instances are launching correctly by looking at the **Instances** window in the EC2 Management Console.

The setup script supports many options for customizing the type of instances, the number of hard drives and so on. You can explore these options by passing the `--help` command line option to `spark-ec2`.

The life cycle of the cluster can be controlled by passing different commands to the `spark-ec2` script, such as:

```
# shut down 'test_cluster'  
$ ./spark-ec2 stop test_cluster  
  
# start 'test_cluster'  
$ ./spark-ec2 -i test_ec2.pem start test_cluster  
  
# destroy 'test_cluster'  
$ ./spark-ec2 destroy test_cluster
```

For more detail on using Spark on EC2, consult the official documentation at <http://spark.apache.org/docs/latest/ec2-scripts.html#running-applications>.

Spam filtering

Let's put all we've learned to good use and do some data exploration for our spam filter. We will use the Ling-Spam email dataset: <http://csmining.org/index.php/ling-spam-datasets.html>. The dataset contains 2412 ham emails and 481 spam emails, all of which were received by a mailing list on linguistics. We will extract the words that are most informative of whether an email is spam or ham.

The first steps in any natural language processing workflow are to remove stop words and lemmatization. Removing stop words involves filtering very common words such as *the*, *this* and so on. Lemmatization involves replacing different forms of the same word with a canonical form: both *colors* and *color* would be mapped to *color*, and *organize*, *organizing* and *organizes* would be mapped to *organize*. Removing stop words and lemmatization is very challenging, and beyond the scope of this book (if you do need to remove stop words and lemmatize a dataset, your go-to tool should be the Stanford NLP toolkit: <http://nlp.stanford.edu/software/corenlp.shtml>). Fortunately, the Ling-Spam e-mail dataset has been cleaned and lemmatized already (which is why the text in the emails looks strange).

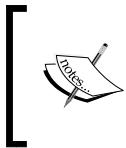
When we do build the spam filter, we will use the presence of a particular word in an email as the feature for our model. We will use a *bag-of-words* approach: we consider which words appear in an email, but not the word order.

Intuitively, some words will be more important than others when deciding whether an email is spam. For instance, an email that contains *language* is likely to be ham, since the mailing list was for linguistics discussions, and *language* is a word unlikely to be used by spammers. Conversely, words which are common to both message types, for instance *hello*, are unlikely to be much use.

One way of quantifying the importance of a word in determining whether a message is spam is through the **Mutual Information (MI)**. The mutual information is the gain in information about whether a message is ham or spam if we know that it contains a particular word. For instance, the presence of *language* in a particular email is very informative as to whether that email is spam or ham. Similarly, the presence of the word *dollar* is informative since it appears often in spam messages and only infrequently in ham messages. By contrast, the presence of the word *morning* is uninformative, since it is approximately equally common in both spam and ham messages. The formula for the mutual information between the presence of a particular word in an email, and whether that email is spam or ham is:

$$MI(word) = \sum_{\substack{wordPresent \in \{true, false\} \\ class \in \{spam, ham\}}} P(wordPresent, class) \cdot \log_2 \frac{P(wordPresent, class)}{P(wordPresent)P(class)}$$

where $P(\text{wordPresent}, \text{class})$ is the joint probability of an email containing a particular word and being of that class (either ham or spam), $P(\text{wordPresent})$ is the probability that a particular word is present in an email, and $P(\text{class})$ is the probability that any email is of that class. The MI is commonly used in decision trees.



The derivation of the expression for the mutual information is beyond the scope of this book. The interested reader is directed to David MacKay's excellent *Information Theory, Inference, and Learning Algorithms*, especially the chapter *Dependent Random Variables*.

A key component of our MI calculation is evaluating the probability that a word occurs in spam or ham messages. The best approximation to this probability, given our data set, is the fraction of messages a word appears in. Thus, for instance, if *language* appears in 40% of messages, we will assume that the probability $P(\text{languagePresent})$ of language being present in any message is 0.4. Similarly, if 40% of the messages are ham, and *language* appears in 50% of those, we will assume that the probability of language being present in an email, and that email being ham is $P(\text{languagePresent}, \text{ham}) = 0.5 \times 0.4 = 0.2$.

Let's write a `wordFractionInFiles` function to calculate the fraction of messages in which each word appears, for all the words in a given corpus. Our function will take, as argument, a path with a shell wildcard identifying a set of files, such as `ham/*`, and it will return a key-value RDD, where the keys are words and the values are the probability that that word occurs in any of those files. We will put the function in an object called `MutualInformation`.

We first give the entire code listing for this function. Don't worry if this doesn't all make sense straight-away: we explain the tricky parts in more detail just after the code. You may find it useful to type some of these commands in the shell, replacing `fileGlob` with, for instance "ham/*":

```
// MutualInformation.scala
import org.apache.spark.{ SparkConf, SparkContext }
import org.apache.spark.SparkContext._
import org.apache.spark.rdd.RDD

object MutualInformation extends App {

    def wordFractionInFiles(sc:SparkContext) (fileGlob:String)
        : (RDD[(String, Double)], Long) = {

        // A set of punctuation words that need to be filtered out.
        val wordsToOmit = Set[String] (
            "", ".", ",", ":" , "-", "\\"", "'", "
        )
    }
}
```

```
"(, "@", "/", "Subject:")
)

val messages = sc.wholeTextFiles(fileGlob)
// wholeTextFiles generates a key-value RDD of
// file name -> file content

val nMessages = messages.count()

// Split the content of each message into a Set of unique
// words in that message, and generate a new RDD mapping:
// message -> word
val message2Word = messages.flatMapValues {
    mailBody => mailBody.split("\\s").toSet
}

val message2FilteredWords = message2Word.filter {
    case(email, word) => ! wordsToOmit(word)
}

val word2Message = message2FilteredWords.map { _.swap }

// word -> number of messages it appears in.
val word2NumberMessages = word2Message.mapValues {
    _ => 1
}.reduceByKey { _ + _ }

// word -> fraction of messages it appears in
val pPresent = word2NumberMessages.mapValues {
    _ / nMessages.toDouble
}

(pPresent, nMessages)
}
}
```

Let's play with this function in the Spark shell. To be able to access this function from the shell, we need to create a jar with the `MutualInformation` object. Write a `build.sbt` file similar to the one presented in the previous section and package the code into a jar using `sbt package`. Then, open a Spark shell with:

```
$ spark-shell --jars=target/scala-2.10/spam_mi_2.10-0.1-SNAPSHOT.jar
```

This will open a Spark shell with our newly created jar on the classpath. Let's run our wordFractionInFiles method on the ham emails:

```
scala> import MutualInformation._  
import MutualInformation._  
  
scala> val (fractions, nMessages) = wordFractionInFiles(sc) ("ham/*")  
fractions: org.apache.spark.rdd.RDD[(String, Double)] =  
MapPartitionsRDD[13] at mapValues  
nMessages: Long = 2412
```

Let's get a snapshot of the fractions RDD:

```
scala> fractions.take(5)  
Array[(String, Double)] = Array((rule-base,0.002902155887230514), (re  
union,4.1459369817578774E-4), (embarrassingly,4.1459369817578774E-4),  
(mller,8.291873963515755E-4), (sapore,4.1459369817578774E-4))
```

It would be nice to see the words that come up most often in ham messages. We can use the .takeOrdered action to take the top values of an RDD, with a custom ordering. .takeOrdered expects, as its second argument, an instance of the type class Ordering[T], where T is the type parameter of our RDD: (string, Double) in this case. Ordering[T] is a trait with a single compare(a:T, b:T) method describing how to compare a and b. The easiest way of creating an Ordering[T] is through the companion object's by method, which defines a key by which to compare the elements of our RDD.

We want to order the elements in our key-value RDD by the value and, since we want the most common words, rather than the least, we need to reverse that ordering:

```
scala> fractions.takeOrdered(5)(Ordering.by { - _._2 })  
res0: Array[(String, Double)] = Array((language,0.6737147595356551),  
(university,0.6048922056384743), (linguistic,0.5149253731343284),  
(information,0.45480928689883915), ('s,0.4369817578772803))
```

Unsurprisingly, language is present in 67% of ham emails, university in 60% of ham emails and so on. A similar investigation on spam messages reveals that the exclamation mark character ! is present in 83% of spam emails, our is present in 61% and free in 57%.

We are now in a position to start writing the body of our application to calculate the mutual information between each word and whether a message is spam or ham. We will put the body of the code in the MutualInformation object, which already contains the wordFractionInFiles method.

The first step is to create a Spark context:

```
// MutualInformation.scala
import org.apache.spark.{ SparkConf, SparkContext }
import org.apache.spark.SparkContext._
import org.apache.spark.rdd.RDD

object MutualInformation extends App {

    def wordFractionInFiles(sc:SparkContext) (fileGlob:String)
        :(RDD[(String, Double)], Long) = {
        ...
    }

    val conf = new SparkConf().setAppName("lingSpam")
    val sc = new SparkContext(conf)
}
```

Note that we did not need to do this when we were using the Spark shell because the shell comes with a pre-built context bound to the variable `sc`.

We can now calculate the conditional probabilities of a message containing a particular word given that it is *spam*, $P(\text{wordPresent} | \text{spam})$. This is just the fraction of messages containing that word in the *spam* corpus. This, in turn, lets us infer the joint probability of a message containing a certain word and being *spam* $P(\text{wordPresent}, \text{spam}) = P(\text{wordPresent} | \text{spam}) \times P(\text{spam})$. We will do this for all four combinations of classes: whether any given word is present or absent in a message, and whether that message is spam or ham:

```
/* Conditional probabilities RDD:
   word -> P(present | spam)
*/
val (pPresentGivenSpam, nSpam) = wordFractionInFiles(sc) ("spam/*")
val pAbsentGivenSpam = pPresentGivenSpam.mapValues { 1.0 - _ }
val (pPresentGivenHam, nHam) = wordFractionInFiles(sc) ("ham/*")
val pAbsentGivenHam = pPresentGivenHam.mapValues { 1.0 - _ }

// pSpam is the fraction of spam messages
val nMessages = nSpam + nHam
val pSpam = nSpam / nMessages.toDouble

// pHam is the fraction of ham messages
```

```
val pHam = 1.0 - pSpam

/* pPresentAndSpam is a key-value RDD of joint probabilities
   word -> P(word present, spam)
*/
val pPresentAndSpam = pPresentGivenSpam.mapValues {
  _ * pSpam
}
val pPresentAndHam = pPresentGivenHam.mapValues { _ * pHam }
val pAbsentAndSpam = pAbsentGivenSpam.mapValues { _ * pSpam }
val pAbsentAndHam = pAbsentGivenHam.mapValues { _ * pHam }
```

We will re-use these RDDs in several places in the calculation, so let's tell Spark to keep them in memory to avoid having to re-calculate them:

```
pPresentAndSpam.persist
pPresentAndHam.persist
pAbsentAndSpam.persist
pAbsentAndHam.persist
```

We now need to calculate the probabilities of words being present, $P(\text{wordPresent})$. This is just the sum of `pPresentAndSpam` and `pPresentAndHam`, for each word. The tricky part is that not all words are present in both the ham and spam messages. We must therefore do a full outer join of those RDDs. This will give an RDD mapping each word to a pair of `Option[Double]` values. For words absent in either the ham or spam messages, we must use a default value. A sensible default is $P(\text{wordPresent} | \text{spam}) = (0.5 / nSpam) \times P(\text{spam})$ for spam messages (a more rigorous approach would be to use *additive smoothing*). This implies that the word would appear once if the corpus was twice as large.

```
val pJoined = pPresentAndSpam.fullOuterJoin(pPresentAndHam)
val pJoinedDefault = pJoined.mapValues {
  case (presentAndSpam, presentAndHam) =>
    (presentAndSpam.getOrElse(0.5/nSpam * pSpam),
     presentAndHam.getOrElse(0.5/nHam * pHam))
}
```

Note that we could also have chosen 0 as the default value. This complicates the information gain calculation somewhat, since we cannot just take the log of a zero value, and it seems unlikely that a particular word has exactly zero probability of occurring in an email.

We can now construct an RDD mapping words to $P(\text{wordPresent})$, the probability that a word exists in either a spam or a ham message:

```
val pPresent = pJoinedDefault.mapValues {
    case (presentAndHam, presentAndSpam) =>
        presentAndHam + presentAndSpam
}
pPresent.persist

val pAbsent = pPresent.mapValues { 1.0 - _ }
pAbsent.persist
```

We now have all the RDDs that we need to calculate the mutual information between the presence of a word in a message and whether it is ham or spam. We need to bring them all together using the equation for the mutual information outlined earlier.

We will start by defining a helper method that, given an RDD of joint probabilities $P(X, Y)$ and marginal probabilities $P(X)$ and $P(Y)$, calculates $P(X, Y) \times \log\left(\frac{P(X, Y)}{P(X)P(Y)}\right)$. Here, $P(X)$ could, for instance, be the probability of a word being present in a message $P(\text{wordPresent})$ and $P(Y)$ would be the probability that that message is *spam*, $P(\text{spam})$:

```
def miTerm(
    pXYS:RDD[(String, Double)],
    pXs:RDD[(String, Double)],
    pY: Double,
    default: Double // for words absent in PXY
) :RDD[(String, Double)] =
    pXs.leftOuterJoin(pXYS).mapValues {
        case (pX, Some(pXY)) => pXY * math.log(pXY/(pX*pY))
        case (pX, None) => default * math.log(default/(pX*pY))
    }
```

We can use our function to calculate the four terms in the mutual information sum:

```
val miTerms = List(
    miTerm(pPresentAndSpam, pPresent, pSpam, 0.5/nSpam * pSpam),
    miTerm(pPresentAndHam, pPresent, pHam, 0.5/nHam * pHam),
    miTerm(pAbsentAndSpam, pAbsent, pSpam, 0.5/nSpam * pSpam),
    miTerm(pAbsentAndHam, pAbsent, pHam, 0.5/nHam * pHam)
)
```

Finally, we just need to sum those four terms together:

```
val mutualInformation = miTerms.reduce {
  (term1, term2) => term1.join(term2).mapValues {
    case (l, r) => l + r
  }
}
```

The RDD `mutualInformation` is a key-value RDD mapping each word to a measure of how informative the presence of that word is in discerning whether a message is spam or ham. Let's print out the twenty words that are most informative of whether a message is ham or spam:

```
mutualInformation.takeOrdered(20)(Ordering.by { - _.value })
.foreach { println }
```

Let's run this using `spark-submit`:

```
$ sbt package
$ spark-submit target/scala-2.10/spam_mi_2.10-0.1-SNAPSHOT.jar
(!,0.1479941771292119)
(language,0.14574624861510874)
(remove,0.11380645864246142)
(free,0.1073496947123657)
(university,0.10695975885487692)
(money,0.07531772498093084)
(click,0.06887598051593441)
(our,0.058950906866052394)
(today,0.05485248095680509)
(sell,0.05385519653184113)
(english,0.053509319455430575)
(business,0.05299311289740539)
(market,0.05248394151802276)
(product,0.05096229706182162)
(million,0.050233193237964546)
(linguistics,0.04990172586630499)
(internet,0.04974101556655623)
(company,0.04941817269989519)
(%,0.04890193809823071)
(save,0.04861393414892205)
```

Thus, we find that the presence of words like `language` or `free` or `!` carry the most information, because they are almost exclusively present in either just spam messages or just ham messages. A very simple classification algorithm could just take the top 10 (by mutual information) spam words, and the top 10 ham words and see whether a message contains more spam words or ham words. We will explore machine learning algorithms for classification in more depth in *Chapter 12, Distributed Machine Learning with MLlib*.

Lifting the hood

In the last section of this chapter, we will discuss, very briefly, how Spark works internally. For a more detailed discussion, see the *References* section at the end of the chapter.

When you open a Spark context, either explicitly or by launching the Spark shell, Spark starts a web UI with details of how the current task and past tasks have executed. Let's see this in action for the example mutual information program we wrote in the last section. To prevent the context from shutting down when the program completes, you can insert a call to `readLine` as the last line of the `main` method (after the call to `takeOrdered`). This expects input from the user, and will therefore pause program execution until you press *enter*.

To access the UI, point your browser to `127.0.0.1:4040`. If you have other instances of the Spark shell running, the port may be `4041`, or `4042` and so on.

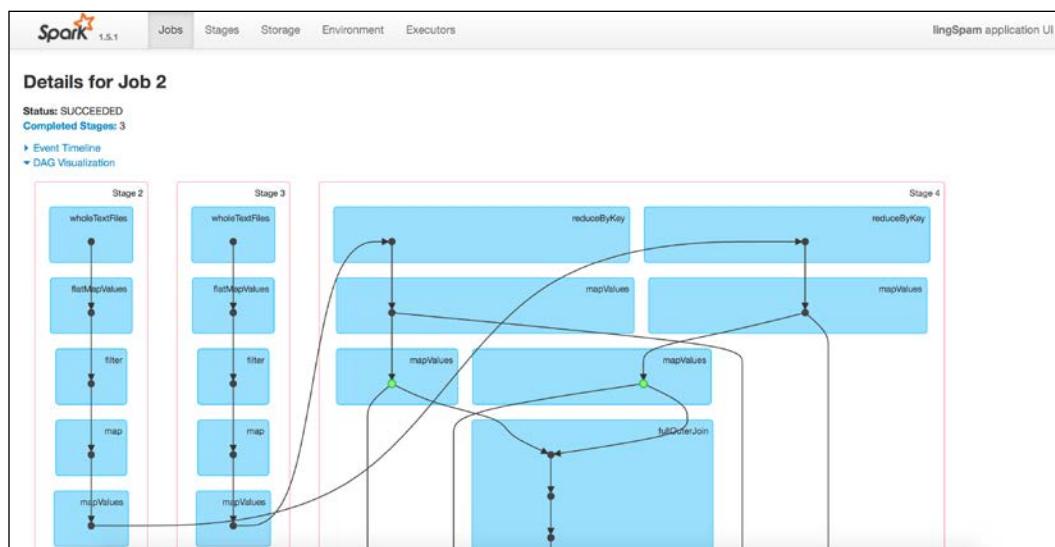


The first page of the UI tells us that our application contains three *jobs*. A job occurs as the result of an action. There are, indeed, three actions in our application: the first two are called within the `wordFractionInFiles` function:

```
val nMessages = messages.count()
```

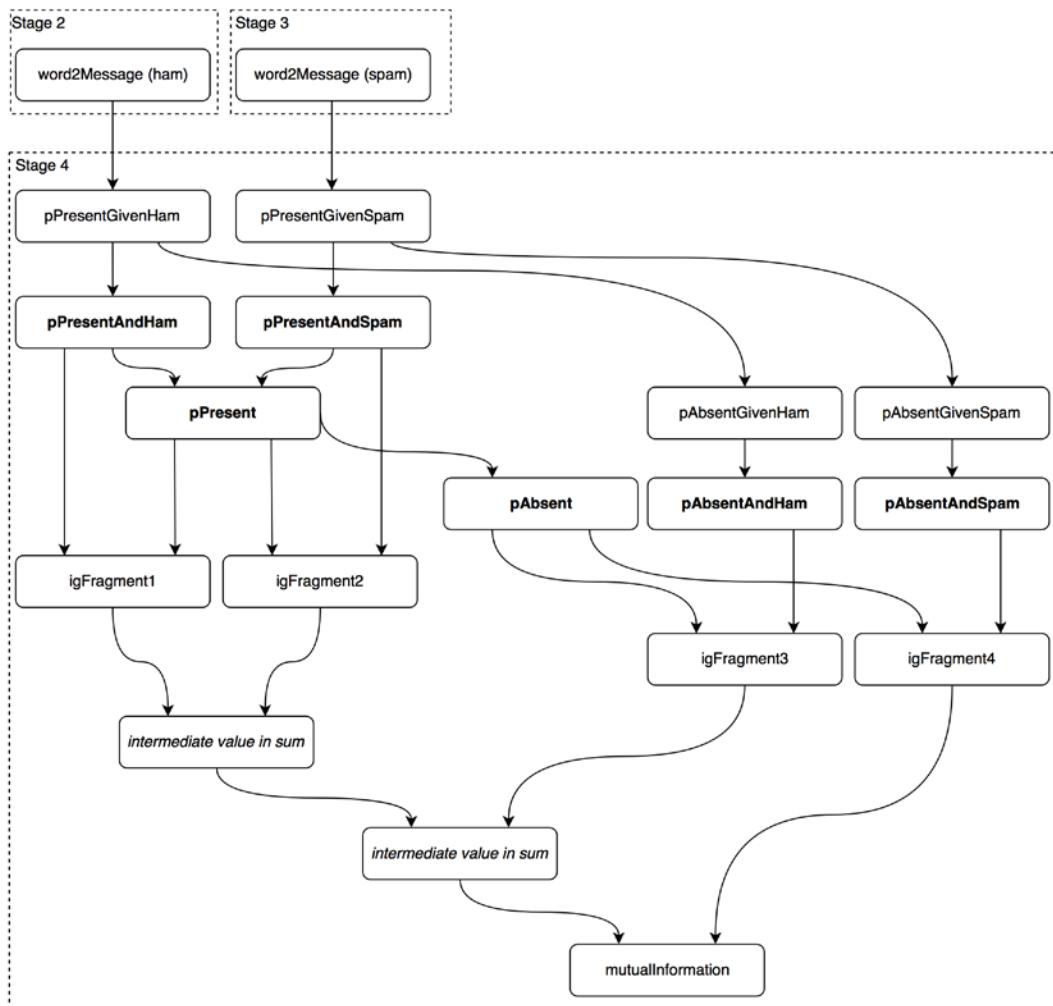
The last job results from the call to `takeOrdered`, which forces the execution of the entire pipeline of RDD transformations that calculate the mutual information.

The web UI lets us delve deeper into each job. Click on the `takeOrdered` job in the job table. You will get taken to a page that describes the job in more detail:



Of particular interest is the **DAG visualization** entry. This is a graph of the execution plan to fulfill the action, and provides a glimpse of the inner workings of Spark.

When you define a job by calling an action on an RDD, Spark looks at the RDD's lineage and constructs a graph mapping the dependencies: each RDD in the lineage is represented by a node, with directed edges going from this RDD's parent to itself. This type of graph is called a **directed acyclic graph (DAG)**, and is a data structure useful for dependency resolution. Let's explore the DAG for the `takeOrdered` job in our program using the web UI. The graph is quite complex, and it is therefore easy to get lost, so here is a simplified reproduction that only lists the RDDs bound to variable names in the program.



As you can see, at the bottom of the graph, we have the `mutualInformation` RDD. This is the RDD that we need to construct for our action. This RDD depends on the intermediate elements in the sum, `igFragment1`, `igFragment2`, and so on. We can work our way back through the list of dependencies until we reach the other end of the graph: RDDs that do not depend on other RDDs, only on external sources.

Once the graph is built, the Spark engines formulates a plan to execute the job. The plan starts with the RDDs that only have external dependencies (such as RDDs built by loading files from disk or fetching from a database) or RDDs that already have cached data. Each arrow along the graph is translated to a set of *tasks*, with each task applying a transformation to a partition of the data.

Tasks are grouped into *stages*. A stage consists of a set of tasks that can all be performed without needing an intermediate shuffle.

Data shuffling and partitions

To understand data shuffling in Spark, we first need to understand how data is partitioned in RDDs. When we create an RDD by, for instance, loading a file from HDFS, or reading a file in local storage, Spark has no control over what bits of data are distributed in which partitions. This becomes a problem for key-value RDDs: these often require knowing where occurrences of a particular key are, for instance to perform a join. If the key can occur anywhere in the RDD, we have to look through every partition to find the key.

To prevent this, Spark allows the definition of a *partitioner* on key-value RDDs. A partitioner is an attribute of the RDD that determines which partition a particular key lands in. When an RDD has a partitioner set, the location of a key is entirely determined by the partitioner, and not by the RDD's history, or the number of keys. Two different RDDs with the same partitioner will map the same key to the same partition.

Partitions impact performance through their effect on transformations. There are two types of transformations on key-value RDDs:

- Narrow transformations, like `mapValues`. In narrow transformations, the data to compute a partition in the child RDD resides on a single partition in the parent. The data processing for a narrow transformation can therefore be performed entirely locally, without needing to communicate data between nodes.
- Wide transformations, like `reduceByKey`. In wide transformations, the data to compute any single partition can reside on all the partitions in the parent. The RDD resulting from a wide transformation will, in general, have a partitioner set. For instance, the output of a `reduceByKey` transformation are hash-partitioned by default: the partition that a particular key ends up in is determined by `hash(key) % numPartitions`.

Thus, in our mutual information example, the RDDs `pPresentAndSpam` and `pPresentAndHam` will have the same partition structure since they both have the default hash partitioner. All descendent RDDs retain the same keys, all the way down to `mutualInformation`. The word `language`, for instance, will be in the same partition for each RDD.

Why does all this matter? If an RDD has a partitioner set, this partitioner is retained through all subsequent narrow transformations originating from this RDD. Let's go back to our mutual information example. The RDDs `pPresentGivenHam` and `pPresentGivenSpam` both originate from `reduceByKey` operations, and they both have string keys. They will therefore both have the same hash-partitioner (unless we explicitly set a different partitioner). This partitioner is retained as we construct `pPresentAndSpam` and `pPresentAndHam`. When we construct `pPresent`, we perform a full outer join of `pPresentAndSpam` and `pPresentAndHam`. Since both these RDDs have the same partitioner, the child RDD `pPresent` has narrow dependencies: we can just join the first partition of `pPresentAndSpam` with the first partition of `pPresentAndHam`, the second partition of `pPresentAndSpam` with the second partition of `pPresentAndHam` and so on, since any string key will be hashed to the same partition in both RDDs. By contrast, without partitioner, we would have to join the data in each partition of `pPresentAndSpam` with every partition of `pPresentAndSpam`. This would require sending data across the network to all the nodes holding `pPresentAndSpam`, a time-consuming exercise.

This process of having to send the data to construct a child RDD across the network, as a result of wide dependencies, is called *shuffling*. Much of the art of optimizing a Spark program involves reducing shuffling and, when shuffling is necessary, reducing the amount of shuffling.

Summary

In this chapter, we explored the basics of Spark and learned how to construct and manipulate RDDs. In the next chapter, we will learn about Spark SQL and DataFrames, a set of implicit conversions that allow us to manipulate RDDs in a manner similar to pandas DataFrames, and how to interact with different data sources using Spark.

Reference

- *Learning Spark*, by Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia, O'Reilly, provides a much more complete introduction to Spark than this chapter can provide. I thoroughly recommend it.
- If you are interested in learning more about information theory, I recommend David MacKay's book *Information Theory, Inference, and Learning Algorithms*.
- *Information Retrieval*, by Manning, Raghavan, and Schütze, describes how to analyze textual data (including lemmatization and stemming). An online version is available at <http://nlp.stanford.edu/IR-book/>.
- On the Ling-Spam dataset, and how to analyze it: http://www.aueb.gr/users/ion/docs/ir_memory_based_antispam_filtering.pdf.
- This blog post delves into the Spark Web UI in more detail. <https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html>.
- This blog post, by Sandy Ryza, is the first in a two-part series discussing Spark internals, and how to leverage them to improve performance: <http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/>.

11

Spark SQL and DataFrames

In the previous chapter, we learned how to build a simple distributed application using Spark. The data that we used took the form of a set of e-mails stored as text files.

We learned that Spark was built around the concept of **resilient distributed datasets (RDDs)**. We explored several types of RDDs: simple RDDs of strings, key-value RDDs, and RDDs of doubles. In the case of key-value RDDs and RDDs of doubles, Spark added functionality beyond that of the simple RDDs through implicit conversions. There is one important type of RDD that we have not explored yet: **DataFrames** (previously called **SchemaRDD**). DataFrames allow the manipulation of objects significantly more complex than those we have explored to date.

A DataFrame is a distributed tabular data structure, and is therefore very useful for representing and manipulating structured data. In this chapter, we will first investigate DataFrames through the Spark shell, and then use the Ling-spam e-mail dataset, presented in the previous chapter, to see how DataFrames can be integrated in a machine learning pipeline.

DataFrames – a whirlwind introduction

Let's start by opening a Spark shell:

```
$ spark-shell
```

Let's imagine that we are interested in running analytics on a set of patients to estimate their overall health level. We have measured, for each patient, their height, weight, age, and whether they smoke.

We might represent the readings for each patient as a case class (you might wish to write some of this in a text editor and paste it into the Scala shell using :paste):

```
scala> case class PatientReadings(  
    val patientId: Int,  
    val heightCm: Int,  
    val weightKg: Int,  
    val age:Int,  
    val isSmoker:Boolean  
)  
defined class PatientReadings
```

We would, typically, have many thousands of patients, possibly stored in a database or a CSV file. We will worry about how to interact with external sources later in this chapter. For now, let's just hard-code a few readings directly in the shell:

```
scala> val readings = List(  
    PatientReadings(1, 175, 72, 43, false),  
    PatientReadings(2, 182, 78, 28, true),  
    PatientReadings(3, 164, 61, 41, false),  
    PatientReadings(4, 161, 62, 43, true)  
)  
List[PatientReadings] = List(...
```

We can convert readings to an RDD by using sc.parallelize:

```
scala> val readingsRDD = sc.parallelize(readings)  
readingsRDD: RDD[PatientReadings] = ParallelCollectionRDD[0] at  
parallelize at <console>:25
```

Note that the type parameter of our RDD is PatientReadings. Let's convert the RDD to a DataFrame using the .toDF method:

```
scala> val readingsDF = readingsRDD.toDF  
readingsDF: sql.DataFrame = [patientId: int, heightCm: int, weightKg:  
int, age: int, isSmoker: boolean]
```

We have created a DataFrame where each row corresponds to the readings for a specific patient, and the columns correspond to the different features:

```
scala> readingsDF.show
+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|
+-----+-----+-----+-----+
|      1 |    175 |     72 |  43 |  false |
|      2 |    182 |     78 |  28 |   true |
|      3 |    164 |     61 |  41 |  false |
|      4 |    161 |     62 |  43 |   true |
+-----+-----+-----+-----+
```

The easiest way to create a DataFrame is to use the `toDF` method on an RDD. We can convert any `RDD[T]`, where `T` is a case class or a tuple, to a DataFrame. Spark will map each attribute of the case class to a column of the appropriate type in the DataFrame. It uses reflection to discover the names and types of the attributes. There are several other ways of constructing DataFrames, both from RDDs and from external sources, which we will explore later in this chapter.

DataFrames support many operations for manipulating the rows and columns. For instance, let's add a column for the **Body Mass Index (BMI)**. The BMI is a common way of aggregating *height* and *weight* to decide if someone is overweight or underweight. The formula for the BMI is:

$$BMI = \text{weight}(kg) / \text{height}(m)^2$$

Let's start by creating a column of the height in meters:

```
scala> val heightM = readingsDF("heightCm") / 100.0
heightM: sql.Column = (heightCm / 100.0)
```

`heightM` has data type `Column`, representing a column of data in a DataFrame. Columns support many arithmetic and comparison operators that apply element-wise across the column (similarly to Breeze vectors encountered in *Chapter 2, Manipulating Data with Breeze*). Operations on columns are lazy: the `heightM` column is not actually computed when defined. Let's now define a BMI column:

```
scala> val bmi = readingsDF("weightKg") / (heightM*heightM)
bmi: sql.Column = (weightKg / ((heightCm / 100.0) * (heightCm / 100.0)))
```

It would be useful to add the `bmi` column to our readings DataFrame. Since DataFrames, like RDDs, are immutable, we must define a new DataFrame that is identical to `readingsDF`, but with an additional column for the BMI. We can do this using the `withColumn` method, which takes, as its arguments, the name of the new column and a `Column` instance:

```
scala> val readingsWithBmiDF = readingsDF.withColumn("BMI", bmi)
readingsWithBmiDF: sql.DataFrame = [heightCm: int, weightKg: int, age: int, isSmoker: boolean, BMI: double]
```

All the operations we have seen so far are *transformations*: they define a pipeline of operations that create new DataFrames. These transformations are executed when we call an **action**, such as `show`:

```
scala> readingsWithBmiDF.show
+-----+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|          BMI |
+-----+-----+-----+-----+-----+
|      1 |    175 |     72 |  43 |   false | 23.510204081632654 |
|      2 |    182 |     78 |  28 |    true | 23.54788069073783 |
|      3 |    164 |     61 |  41 |   false | 22.679952409280194 |
|      4 |    161 |     62 |  43 |    true | 23.9188302920412 |
+-----+-----+-----+-----+-----+
```

Besides creating additional columns, DataFrames also support filtering rows that satisfy a certain predicate. For instance, we can select all smokers:

```
scala> readingsWithBmiDF.filter {
  readingsWithBmiDF("isSmoker")
}.show
+-----+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|          BMI |
+-----+-----+-----+-----+-----+
|      2 |    182 |     78 |  28 |    true | 23.54788069073783 |
|      4 |    161 |     62 |  43 |    true | 23.9188302920412 |
+-----+-----+-----+-----+-----+
```

Or, to select everyone who weighs more than 70 kgs:

```
scala> readingsWithBmiDF.filter {
  readingsWithBmiDF("weightKg") > 70
}.show
+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|          BMI |
+-----+-----+-----+-----+
|      1 |     175 |      72 |  43 |  false| 23.510204081632654 |
|      2 |     182 |      78 |  28 |   true| 23.54788069073783 |
+-----+-----+-----+-----+
```

It can become cumbersome to keep repeating the DataFrame name in an expression. Spark defines the operator `$` to refer to a column in the current DataFrame. Thus, the filter expression above could have been written more succinctly using:

```
scala> readingsWithBmiDF.filter { $"weightKg" > 70 }.show
+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|          BMI |
+-----+-----+-----+-----+
|      1 |     175 |      72 |  43 |  false| 23.510204081632654 |
|      2 |     182 |      78 |  28 |   true| 23.54788069073783 |
+-----+-----+-----+-----+
```

The `.filter` method is overloaded. It accepts either a column of Boolean values, as above, or a string identifying a Boolean column in the current DataFrame. Thus, to filter our `readingsWithBmiDF` DataFrame to sub-select smokers, we could also have used the following:

```
scala> readingsWithBmiDF.filter("isSmoker").show
+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|          BMI |
+-----+-----+-----+-----+
|      2 |     182 |      78 |  28 |   true| 23.54788069073783 |
|      4 |     161 |      62 |  43 |   true| 23.9188302920412 |
+-----+-----+-----+-----+
```

When comparing for equality, you must compare columns with the special *triple-equals* operator:

```
scala> readingsWithBmiDF.filter { $"age" === 28 }.show
+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|          BMI |
+-----+-----+-----+-----+
|      2 |     182 |      78 |  28 |    true | 23.54788069073783 |
+-----+-----+-----+-----+
```

Similarly, you must use `!==` to select rows that are not equal to a value:

```
scala> readingsWithBmiDF.filter { $"age" !== 28 }.show
+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|          BMI |
+-----+-----+-----+-----+
|      1 |     175 |      72 |  43 |  false | 23.510204081632654 |
|      3 |     164 |      61 |  41 |  false | 22.679952409280194 |
|      4 |     161 |      62 |  43 |   true |  23.9188302920412 |
+-----+-----+-----+-----+
```

Aggregation operations

We have seen how to apply an operation to every row in a DataFrame to create a new column, and we have seen how to use filters to build new DataFrames with a sub-set of rows from the original DataFrame. The last set of operations on DataFrames is grouping operations, equivalent to the `GROUP BY` statement in SQL. Let's calculate the average BMI for smokers and non-smokers. We must first tell Spark to group the DataFrame by a column (the `isSmoker` column, in this case), and then apply an aggregation operation (averaging, in this case) to reduce each group:

```
scala> val smokingDF = readingsWithBmiDF.groupBy(
  "isSmoker").agg(avg("BMI"))
smokingDF: org.apache.spark.sql.DataFrame = [isSmoker: boolean, AVG(BMI): double]
```

This has created a new DataFrame with two columns: the grouping column and the column over which we aggregated. Let's show this DataFrame:

```
scala> smokingDF.show
+-----+-----+
| isSmoker | AVG(BMI) |
+-----+-----+
| true | 23.733355491389517 |
| false | 23.095078245456424 |
+-----+
```

Besides averaging, there are several operators for performing the aggregation across each group. We outline some of the more important ones in the table below, but, for a full list, consult the *Aggregate functions* section of [http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions\\$](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$):

Operator	Notes
avg(column)	Group averages of the values in the specified column.
count(column)	Number of elements in each group in the specified column.
countDistinct(column, ...)	Number of distinct elements in each group. This can also accept multiple columns to return the count of unique elements across several columns.
first(column), last(column)	First/last element in each group
max(column), min(column)	Largest/smallest element in each group
sum(column)	Sum of the values in each group

Each aggregation operator takes either the name of a column, as a string, or an expression of type `Column`. The latter allows aggregation of compound expressions. If we wanted the average height, in meters, of the smokers and non-smokers in our sample, we could use:

```
scala> readingsDF.groupBy("isSmoker").agg {
  avg($"heightCm"/100.0)
}.show
+-----+
| isSmoker | AVG((heightCm / 100.0)) |
+-----+
```

```
|   true|      1.715|
| false| 1.694999999999998|
+-----+
```

We can also use compound expressions to define the column on which to group. For instance, to count the number of patients in each age group, increasing by decade, we can use:

```
scala> readingsDF.groupBy(floor($"age"/10)).agg(count("*")).show
+-----+-----+
|FLOOR((age / 10))|count(1)|
+-----+-----+
|        4.0|      3|
|        2.0|      1|
+-----+-----+
```

We have used the short-hand "*" to indicate a count over every column.

Joining DataFrames together

So far, we have only considered operations on a single DataFrame. Spark also offers SQL-like joins to combine DataFrames. Let's assume that we have another DataFrame mapping the patient id to a (systolic) blood pressure measurement. We will assume we have the data as a list of pairs mapping patient IDs to blood pressures:

```
scala> val bloodPressures = List((1 -> 110), (3 -> 100), (4 -> 125))
bloodPressures: List[(Int, Int)] = List((1,110), (3,100), (4,125))

scala> val bloodPressureRDD = sc.parallelize(bloodPressures)
res16: rdd.RDD[(Int, Int)] = ParallelCollectionRDD[74] at parallelize at
<console>:24
```

We can construct a DataFrame from this RDD of tuples. However, unlike when constructing DataFrames from RDDs of case classes, Spark cannot infer column names. We must therefore pass these explicitly to `.toDF`:

```
scala> val bloodPressureDF = bloodPressureRDD.toDF(
  "patientId", "bloodPressure")
bloodPressureDF: DataFrame = [patientId: int, bloodPressure: int]

scala> bloodPressureDF.show
```

```
+-----+-----+
|patientId|bloodPressure|
+-----+-----+
|      1|      110|
|      3|      100|
|      4|      125|
+-----+-----+
```

Let's join `bloodPressureDF` with `readingsDF`, using the patient ID as the join key:

```
scala> readingsDF.join(bloodPressureDF,
  readingsDF("patientId") === bloodPressureDF("patientId"))
) .show
+-----+-----+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|patientId|bloodPressure|
+-----+-----+-----+-----+-----+-----+
|      1|     175|      72|  43|  false|      1|      110|
|      3|     164|      61|  41|  false|      3|      100|
|      4|     161|      62|  43|   true|      4|      125|
+-----+-----+-----+-----+-----+-----+
```

This performs an *inner join*: only patient IDs present in both DataFrames are included in the result. The type of join can be passed as an extra argument to `join`. For instance, we can perform a *left join*:

```
scala> readingsDF.join(bloodPressureDF,
  readingsDF("patientId") === bloodPressureDF("patientId"),
  "leftouter")
) .show
+-----+-----+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|patientId|bloodPressure|
+-----+-----+-----+-----+-----+-----+
|      1|     175|      72|  43|  false|      1|      110|
|      2|     182|      78|  28|   true|    null|    null|
|      3|     164|      61|  41|  false|      3|      100|
|      4|     161|      62|  43|   true|      4|      125|
+-----+-----+-----+-----+-----+-----+
```

Possible join types are `inner`, `outer`, `leftouter`, `rightouter`, or `leftsemi`. These should all be familiar, apart from `leftsemi`, which corresponds to a *left semi join*. This is the same as an inner join, but only the columns on the left-hand side are retained after the join. It is thus a way to filter a DataFrame for rows which are present in another DataFrame.

Custom functions on DataFrames

So far, we have only used built-in functions to operate on DataFrame columns. While these are often sufficient, we sometimes need greater flexibility. Spark lets us apply custom transformations to every row through **user-defined functions (UDFs)**. Let's assume that we want to use the equation that we derived in *Chapter 2, Manipulating Data with Breeze*, for the probability of a person being male, given their height and weight. We calculated that the decision boundary was given by:

$$f = -0.75 + 2.48 \times rescaledHeight + 2.23 \times rescaledWeight$$

Any person with $f > 0$ is more likely to be male than female, given their height and weight and the training set used for *Chapter 2, Manipulating Data with Breeze* (which was based on students, so is unlikely to be representative of the population as a whole). To convert from a height in centimeters to the normalized height, *rescaledHeight*, we can use this formula:

$$rescaledHeight = \frac{height - \langle height \rangle}{\sigma_{height}} = \frac{height - 171}{8.95}$$

Similarly, to convert a weight (in kilograms) to the normalized weight, *rescaledWeight*, we can use:

$$rescaledWeight = \frac{weight - \langle weight \rangle}{\sigma_{weight}} = \frac{weight - 65.7}{13.4}$$

The average and standard deviation of the *height* and *weight* are calculated from the training set. Let's write a Scala function that returns whether a person is more likely to be male, given their height and weight:

```
scala> def likelyMale(height:Int, weight:Int):Boolean = {
    val rescaledHeight = (height - 171.0)/8.95
```

```
val rescaledWeight = (weight - 65.7)/13.4
-0.75 + 2.48*rescaledHeight + 2.23*rescaledWeight > 0
}
```

To use this function on Spark DataFrames, we need to register it as a **user-defined function (UDF)**. This transforms our function, which accepts integer arguments, into one that accepts column arguments:

```
scala> val likelyMaleUdf = sqlContext.udf.register(
  "likelyMaleUdf", likelyMale _)
likelyMaleUdf: org.apache.spark.sql.UserDefinedFunction = UserDefinedFunction(<function2>,BooleanType,List())
```

To register a UDF, we must have access to a `sqlContext` instance. The SQL context provides the entry point for DataFrame operations. The Spark shell creates a SQL context at startup, bound to the variable `sqlContext`, and destroys it when the shell session is closed.

The first argument passed to the `register` function is the name of the UDF (we will use the UDF name later when we write SQL statements on the DataFrame, but you can ignore it for now). We can then use the UDF just like the built-in transformations included in Spark:

```
scala> val likelyMaleColumn = likelyMaleUdf(
  readingsDF("heightCm"), readingsDF("weightKg"))
likelyMaleColumn: org.apache.spark.sql.Column = UDF(heightCm,weightKg)

scala> readingsDF.withColumn("likelyMale", likelyMaleColumn).show
+-----+-----+-----+---+-----+
|patientId|heightCm|weightKg|age|isSmoker|likelyMale|
+-----+-----+-----+---+-----+
|       1|    175|     72| 43|   false|    true|
|       2|    182|     78| 28|    true|    true|
|       3|    164|     61| 41|   false|   false|
|       4|    161|     62| 43|    true|   false|
+-----+-----+-----+---+-----+
```

As you can see, Spark applies the function underlying the UDF to every row in the DataFrame. We are not limited to using UDFs to create new columns. We can also use them in `filter` expressions. For instance, to select rows likely to correspond to women:

```
scala> readingsDF.filter(
  ! likelyMaleUdf($"heightCm", $"weightKg")
).show
+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|
+-----+-----+-----+-----+
|      3 |    164 |      61 |  41 |  false |
|      4 |    161 |      62 |  43 |   true |
+-----+-----+-----+-----+
```

Using UDFs lets us define arbitrary Scala functions to transform rows, giving tremendous additional power for data manipulation.

DataFrame immutability and persistence

DataFrames, like RDDs, are immutable. When you define a transformation on a DataFrame, this always creates a new DataFrame. The original DataFrame cannot be modified in place (this is notably different to pandas DataFrames, for instance).

Operations on DataFrames can be grouped into two: *transformations*, which result in the creation of a new DataFrame, and *actions*, which usually return a Scala type or have a side-effect. Methods like `filter` or `withColumn` are transformations, while methods like `show` or `head` are actions.

Transformations are lazy, much like transformations on RDDs. When you generate a new DataFrame by transforming an existing DataFrame, this results in the elaboration of an execution plan for creating the new DataFrame, but the data itself is not transformed immediately. You can access the execution plan with the `queryExecution` method.

When you call an action on a DataFrame, Spark processes the action as if it were a regular RDD: it implicitly builds a direct acyclic graph to resolve dependencies, processing the transformations needed to build the DataFrame on which the action was called.

Much like RDDs, we can persist DataFrames in memory or on disk:

```
scala> readingsDF.persist  
readingsDF.type = [patientId: int, heightCm: int,...]
```

This works in the same way as persisting RDDs: next time the RDD is calculated, it will be kept in memory (provided there is enough space), rather than discarded. The level of persistence can also be set:

```
scala> import org.apache.spark.storage.StorageLevel  
import org.apache.spark.storage.StorageLevel  
  
scala> readingsDF.persist(StorageLevel.MEMORY_AND_DISK)  
readingsDF.type = [patientId: int, heightCm: int, ...]
```

SQL statements on DataFrames

By now, you will have noticed that many operations on DataFrames are inspired by SQL operations. Additionally, Spark allows us to register DataFrames as tables and query them with SQL statements directly. We can therefore build a temporary database as part of the program flow.

Let's register `readingsDF` as a temporary table:

```
scala> readingsDF.registerTempTable("readings")
```

This registers a temporary table that can be used in SQL queries. Registering a temporary table relies on the presence of a SQL context. The temporary tables are destroyed when the SQL context is destroyed (when we close the shell, for instance).

Let's explore what we can do with our temporary tables and the SQL context. We can first get a list of all the tables currently registered with the context:

```
scala> sqlContext.tables  
DataFrame = [tableName: string, isTemporary: boolean]
```

This returns a DataFrame. In general, all operations on a SQL context that return data return DataFrames:

```
scala> sqlContext.tables.show  
+-----+-----+  
| tableName | isTemporary |  
+-----+-----+  
| readings |      true |  
+-----+-----+
```

We can query this table by passing SQL statements to the SQL context:

```
scala> sqlContext.sql("SELECT * FROM readings").show
+-----+-----+-----+-----+
|patientId|heightCm|weightKg|age|isSmoker|
+-----+-----+-----+-----+
|      1|     175|      72|  43|  false|
|      2|     182|      78|  28|   true|
|      3|     164|      61|  41|  false|
|      4|     161|      62|  43|   true|
+-----+-----+-----+-----+
```

Any UDFs registered with the `sqlContext` are available through the name given to them when they were registered. We can therefore use them in SQL queries:

```
scala> sqlContext.sql("""
  SELECT
    patientId,
    likelyMaleUdf(heightCm, weightKg) AS likelyMale
  FROM readings
""").show
+-----+-----+
|patientId|likelyMale|
+-----+-----+
|      1|    true|
|      2|    true|
|      3|   false|
|      4|   false|
+-----+-----+
```

You might wonder why one would want to register DataFrames as temporary tables and run SQL queries on those tables, when the same functionality is available directly on DataFrames. The main reason is for interacting with external tools. Spark can run a SQL engine that exposes a JDBC interface, meaning that programs that know how to interact with a SQL database will be able to make use of the temporary tables.

We don't have the space to cover how to set up a distributed SQL engine in this book, but you can find details in the Spark documentation (<http://spark.apache.org/docs/latest/sql-programming-guide.html#distributed-sql-engine>).

Complex data types – arrays, maps, and structs

So far, all the elements in our DataFrames were simple types. DataFrames support three additional collection types: arrays, maps, and structs.

Structs

The first compound type that we will look at is the **struct**. A struct is similar to a case class: it stores a set of key-value pairs, with a fixed set of keys. If we convert an RDD of a case class containing nested case classes to a DataFrame, Spark will convert the nested objects to a struct.

Let's imagine that we want to serialize Lords of the Ring characters. We might use the following object model:

```
case class Weapon(name:String, weaponType:String)
case class LotrCharacter(name:String, val weapon:Weapon)
```

We want to create a DataFrame of `LotrCharacter` instances. Let's create some dummy data:

```
scala> val characters = List(
    LotrCharacter("Gandalf", Weapon("Glamdring", "sword")),
    LotrCharacter("Frodo", Weapon("Sting", "dagger")),
    LotrCharacter("Aragorn", Weapon("Anduril", "sword"))
)
characters: List[LotrCharacter] = List(LotrCharacter...
```



```
scala> val charactersDF = sc.parallelize(characters).toDF
charactersDF: DataFrame = [name: string, weapon: struct<name:string,weaponType:string>]

scala> charactersDF.printSchema
root
| -- name: string (nullable = true)
| -- weapon: struct (nullable = true)
|   | -- name: string (nullable = true)
```

```
|     | -- weaponType: string (nullable = true)

scala> charactersDF.show
+-----+-----+
|   name|      weapon|
+-----+-----+
|Gandalf|[Glamdring,sword]|
| Frodo |[Sting,dagger] |
|Aragorn |[Anduril,sword] |
+-----+
```

The weapon attribute in the case class was converted to a struct column in the DataFrame. To extract sub-fields from a struct, we can pass the field name to the column's .apply method:

```
scala> val weaponTypeColumn = charactersDF("weapon")("weaponType")
weaponTypeColumn: org.apache.spark.sql.Column = weapon[weaponType]
```

We can use this derived column just as we would any other column. For instance, let's filter our DataFrame to only contain characters who wield a sword:

```
scala> charactersDF.filter { weaponTypeColumn === "sword" }.show
+-----+
|   name|      weapon|
+-----+
|Gandalf|[Glamdring,sword]|
|Aragorn |[Anduril,sword] |
+-----+
```

Arrays

Let's return to the earlier example, and assume that, besides height, weight, and age measurements, we also have phone numbers for our patients. Each patient might have zero, one, or more phone numbers. We will define a new case class and new dummy data:

```
scala> case class PatientNumbers(
    patientId:Int, phoneNumbers>List[String])
```

```
defined class PatientNumbers

scala> val numbers = List(
    PatientNumbers(1, List("07929123456")),
    PatientNumbers(2, List("07929432167", "07929234578")),
    PatientNumbers(3, List.empty),
    PatientNumbers(4, List("07927357862"))
)

scala> val numbersDF = sc.parallelize(numbers).toDF
numbersDF: org.apache.spark.sql.DataFrame = [patientId: int,
phoneNumbers: array<string>]
```

The List [String] array in our case class gets translated to an array<string> data type:

```
scala> numbersDF.printSchema
root
|-- patientId: integer (nullable = false)
|-- phoneNumbers: array (nullable = true)
|   |-- element: string (containsNull = true)
```

As with structs, we can construct a column for a specific index the array. For instance, we can select the first element in each array:

```
scala> val bestNumberColumn = numbersDF("phoneNumbers")(0)
bestNumberColumn: org.apache.spark.sql.Column = phoneNumbers[0]

scala> numbersDF.withColumn("bestNumber", bestNumberColumn).show
+-----+-----+-----+
|patientId|    phoneNumbers| bestNumber|
+-----+-----+-----+
|      1|List(07929123456)|07929123456|
|      2|List(07929432167,...|07929432167|
|      3|          List()|        null|
|      4|List(07927357862)|07927357862|
+-----+-----+-----+
```

Maps

The last compound data type is the map. Maps are similar to structs inasmuch as they store key-value pairs, but the set of keys is not fixed when the DataFrame is created. They can thus store arbitrary key-value pairs.

Scala maps will be converted to DataFrame maps when the DataFrame is constructed. They can then be queried in a manner similar to structs.

Interacting with data sources

A major challenge in data science or engineering is dealing with the wealth of input and output formats for persisting data. We might receive or send data as CSV files, JSON files, or through a SQL database, to name a few.

Spark provides a unified API for serializing and de-serializing DataFrames to and from different data sources.

JSON files

Spark supports loading data from JSON files, provided that each line in the JSON file corresponds to a single JSON object. Each object will be mapped to a DataFrame row. JSON arrays are mapped to arrays, and embedded objects are mapped to structs.

This section would be a little dry without some data, so let's generate some from the GitHub API. Unfortunately, the GitHub API does not return JSON formatted as a single object per line. The code repository for this chapter contains a script, `FetchData.scala` which will download and format JSON entries for Martin Odersky's repositories, saving the objects to a file named `odersky_repos.json` (go ahead and change the GitHub user in `FetchData.scala` if you want). You can also download a pre-constructed data file from data.scala4datascience.com/odersky_repos.json.

Let's dive into the Spark shell and load this data into a DataFrame. Reading from a JSON file is as simple as passing the file name to the `sqlContext.read.json` method:

```
scala> val df = sqlContext.read.json("odersky_repos.json")
df: DataFrame = [archive_url: string, assignees_url: ...]
```

Reading from a JSON file loads data as a DataFrame. Spark automatically infers the schema from the JSON documents. There are many columns in our DataFrame. Let's sub-select a few to get a more manageable DataFrame:

```
scala> val reposDF = df.select("name", "language", "fork", "owner")
reposDF: DataFrame = [name: string, language: string, ...]

scala> reposDF.show
+-----+-----+-----+
|      name| language|   fork|        owner|
+-----+-----+-----+
|      dotty|    Scala|  true|[https://avatars....|
|    frontend|JavaScript|  true|[https://avatars....|
|       scala|    Scala|  true|[https://avatars....|
|  scala-dist|    Scala|  true|[https://avatars....|
| scala.github.com|JavaScript|  true|[https://avatars....|
|      scalax|    Scala|false|[https://avatars....|
|        sips|      CSS|false|[https://avatars....|
+-----+-----+-----+
```

Let's save the DataFrame back to JSON:

```
scala> reposDF.write.json("repos_short.json")
```

If you look at the files present in the directory in which you are running the Spark shell, you will notice a `repos_short.json` directory. Inside it, you will see files named `part-000000`, `part-000001`, and so on. When serializing JSON, each partition of the DataFrame is serialized independently. If you are running this on several machines, you will find parts of the serialized output on each computer.

You may, optionally, pass a mode argument to control how Spark deals with the case of an existing `repos_short.json` file:

```
scala> import org.apache.spark.sql.SaveMode
import org.apache.spark.sql.SaveMode

scala> reposDF.write.mode(
  SaveMode.Overwrite).json("repos_short.json")
```

Available save modes are `ErrorIfExists`, `Append` (only available for Parquet files), `Overwrite`, and `Ignore` (do not save if the file exists already).

Parquet files

Apache Parquet is a popular file format well-suited for storing tabular data. It is often used for serialization in the Hadoop ecosystem, since it allows for efficient extraction of specific columns and rows without having to read the entire file.

Serialization and deserialization of Parquet files is identical to JSON, with the substitution of `json` with `parquet`:

```
scala> reposDF.write.parquet("repos_short.parquet")

scala> val newDF = sqlContext.read.parquet("repos_short.parquet")
newDF: DataFrame = [name: string, language: string, fo...]

scala> newDF.show
+-----+-----+-----+
|      name|language| fork|          owner|
+-----+-----+-----+
|      dotty|   Scala| true|[https://avatars....|
|    frontend|JavaScript| true|[https://avatars....|
|      scala|   Scala| true|[https://avatars....|
|  scala-dist|   Scala| true|[https://avatars....|
|scala.github.com|JavaScript| true|[https://avatars....|
|      scalax|   Scala|false|[https://avatars....|
|        sips|     CSS|false|[https://avatars....|
+-----+-----+-----+
```

In general, Parquet will be more space-efficient than JSON for storing large collections of objects. Parquet is also much more efficient at retrieving specific columns or rows, if the partition can be inferred from the row. Parquet is thus advantageous over JSON unless you need the output to be human-readable, or de-serializable by an external program.

Standalone programs

So far, we have been using Spark SQL and DataFrames through the Spark shell. To use it in standalone programs, you will need to create it explicitly, from a Spark context:

```
val conf = new SparkConf().setAppName("applicationName")
val sc = new SparkContext(conf)
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Additionally, importing the `implicits` object nested in `sqlContext` allows the conversions of RDDs to DataFrames:

```
import sqlContext.implicits._
```

We will use DataFrames extensively in the next chapter to manipulate data to get it ready for use with MLlib.

Summary

In this chapter, we explored Spark SQL and DataFrames. DataFrames add a rich layer of abstraction on top of Spark's core engine, greatly facilitating the manipulation of tabular data. Additionally, the source API allows the serialization and de-serialization of DataFrames from a rich variety of data files.

In the next chapter, we will build on our knowledge of Spark and DataFrames to build a spam filter using MLlib.

References

DataFrames are a relatively recent addition to Spark. There is thus still a dearth of literature and documentation. The first port of call should be the Scala docs, available at: <http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame>.

The Scaladocs for operations available on the DataFrame `Column` type can be found at: <http://spark.apache.org/docs/latest/api/scala/#org.apache.spark.sql.Column>.

There is also extensive documentation on the Parquet file format: <https://parquet.apache.org>.

12

Distributed Machine Learning with MLlib

Machine learning describes the construction of algorithms that make predictions from data. It is a core component of most data science pipelines, and is often seen to be the component adding the most value: the accuracy of the machine learning algorithm determines the success of the data science endeavor. It is also, arguably, the section of the data science pipeline that requires the most knowledge from fields beyond software engineering: a machine learning expert will be familiar, not just with algorithms, but also with statistics and with the business domain.

Choosing and tuning a machine learning algorithm to solve a particular problem involves significant exploratory analysis to try and determine which features are relevant, how features are correlated, whether there are outliers in the dataset, and so on. Designing suitable machine learning pipelines is difficult. Add on an additional layer of complexity resulting from the size of datasets and the need for scalability, and you have a real challenge.

MLlib helps mitigate this difficulty. MLlib is a component of Spark that provides machine learning algorithms on top of the core Spark libraries. It offers a set of learning algorithms that parallelize well over distributed datasets.

MLlib has evolved into two separate layers. MLlib itself contains the core algorithms, and **ml**, also called the *pipeline API*, defines an API for gluing algorithms together and provides a higher level of abstraction. The two libraries differ in the data types on which they operate: the original MLlib predates the introduction of DataFrames, and acts mainly on RDDs of feature vectors. The pipeline API operates on DataFrames.

In this chapter, we will study the newer pipeline API, diving into MLlib only when the functionality is missing from the pipeline API.

This chapter does not try to teach the machine learning fundamentals behind the algorithms that we present. We assume that the reader has a good enough grasp of machine learning tools and techniques to understand, at least superficially, what the algorithms presented here do, and we defer to better authors for in-depth explanations of the mechanics of statistical learning (we present several references at the end of the chapter).

MLlib is a rich library that is evolving rapidly. This chapter does not aim to give a complete overview of the library. We will work through the construction of a machine learning pipeline to train a spam filter, learning about the parts of MLlib that we need along the way. Having read this chapter, you will have an understanding of how the different parts of the library fit together, and can use the online documentation, or a more specialized book (see references at the end of this chapter) to learn about the parts of MLlib not covered here.

Introducing MLlib – Spam classification

Let's introduce MLlib with a concrete example. We will look at spam classification using the Ling-Spam dataset that we used in the *Chapter 10, Distributed Batch Processing with Spark*. We will create a spam filter that uses logistic regression to estimate the probability that a given message is spam.

We will run through examples using the Spark shell, but you will find an analogous program in `LogisticRegressionDemo.scala` among the examples for this chapter. If you have not installed Spark, refer to *Chapter 10, Distributed Batch Processing with Spark*, for installation instructions.

Let's start by loading the e-mails in the Ling-Spam dataset. If you have not done this for *Chapter 10, Distributed Batch Processing with Spark*, download the data from data4datascience.com/ling-spam.tar.gz or data4datascience.com/ling-spam.zip, depending on whether you want a `.tar.gz` file or a `.zip` file, and unpack the archive. This will create a `spam` directory and a `ham` directory containing spam and ham messages, respectively.

Let's use the `wholeTextFiles` method to load spam and ham e-mails:

```
scala> val spamText = sc.wholeTextFiles("spam/*")
spamText: RDD[(String, String)] = spam/...
 
scala> val hamText = sc.wholeTextFiles("ham/*")
hamText: RDD[(String, String)] = ham/...
```

The `wholeTextFiles` method creates a key-value RDD where the keys are the file names and the values are the contents of the files:

```
scala> spamText.first
(String, String) =
(file:spam/spmsgal.txt,"Subject: great part-time summer job! ...")

scala> spamText.count
Long = 481
```

The algorithms in the pipeline API work on DataFrames. We must therefore convert our key-value RDDs to DataFrames. We define a new case class, `LabelledDocument`, which contains a message text and a category label identifying whether a message is spam or ham:

```
scala> case class LabelledDocument(
  fileName:String,
  text:String,
  category:String
)
defined class LabelledDocument

scala> val spamDocuments = spamText.map {
  case (fileName, text) =>
    LabelledDocument(fileName, text, "spam")
}
spamDocuments: RDD[LabelledDocument] = MapPartitionsRDD[2] at map

scala> val hamDocuments = hamText.map {
  case (fileName, text) =>
    LabelledDocument(fileName, text, "ham")
}
hamDocuments: RDD[LabelledDocument] = MapPartitionsRDD[3] at map
```

To create models, we will need all the documents in a single DataFrame. Let's therefore take the union of our two LabelledDocument RDDs, and transform that to a DataFrame. The `union` method concatenates RDDs together:

```
scala> val allDocuments = spamDocuments.union(hamDocuments)
allDocuments: RDD[LabelledDocument] = UnionRDD [4] at union

scala> val documentsDF = allDocuments.toDF
documentsDF: DataFrame = [fileName: string, text: string, category: string]
```

Let's do some basic checks to verify that we have loaded all the documents. We start by persisting the DataFrame in memory to avoid having to re-create it from the raw text files.

```
scala> documentsDF.persist
documentsDF.type = [fileName: string, text: string, category: string]

scala> documentsDF.show
+-----+-----+-----+
|   fileName|          text|category|
+-----+-----+-----+
|file:/Users/pasca...|Subject: great pa...|    spam|
|file:/Users/pasca...|Subject: auto ins...|    spam|
|file:/Users/pasca...|Subject: want bes...|    spam|
|file:/Users/pasca...|Subject: email 57...|    spam|
|file:/Users/pasca...|Subject: n't miss...|    spam|
|file:/Users/pasca...|Subject: amaze wo...|    spam|
|file:/Users/pasca...|Subject: help loa...|    spam|
|file:/Users/pasca...|Subject: beat irs...|    spam|
|file:/Users/pasca...|Subject: email 57...|    spam|
|file:/Users/pasca...|Subject: best , b...|    spam|
|...
+-----+-----+-----+
```

```
scala> documentsDF.groupBy("category").agg(count("*")).show
+-----+-----+
|category|COUNT(1) |
+-----+-----+
```

```
|   spam|     481|
|   ham|    2412|
+-----+
```

Let's now split the DataFrame into a training set and a test set. We will use the test set to validate the model that we build. For now, we will just use a single split, training the model on 70% of the data and testing it on the remaining 30%. In the next section, we will look at cross-validation, which provides more rigorous way to check the accuracy of our models.

We can achieve this 70-30 split using the DataFrame's `.randomSplit` method:

```
scala> val Array(trainDF, testDF) = documentsDF.randomSplit(
  Array(0.7, 0.3))
trainDF: DataFrame = [fileName: string, text: string, category: string]
testDF: DataFrame = [fileName: string, text: string, category: string]
```

The `.randomSplit` method takes an array of weights and returns an array of DataFrames, of approximately the size specified by the weights. For instance, we passed weights 0.7 and 0.3, indicating that any given row has a 70% chance of ending up in `trainDF`, and a 30% chance of ending up in `testDF`. Note that this means the split DataFrames are not of fixed size: `trainDF` is approximately, but not exactly, 70% the size of `documentsDF`:

```
scala> trainDF.count / documentsDF.count.toDouble
Double = 0.7013480815762184
```

If you need a fixed size sample, use the DataFrame's `.sample` method to obtain `trainDF` and filter `documentDF` for rows not in `trainDF`.

We are now in a position to start using MLlib. Our attempt at classification will involve performing logistic regression on *term-frequency vectors*: we will count how often each word appears in each message, and use the frequency of occurrence as a feature. Before jumping into the code, let's take a step back and discuss the structure of machine learning pipelines.

Pipeline components

Pipelines consist of a set of components joined together such that the DataFrame produced by one component is used as input for the next component. The components available are split into two classes: *transformers* and *estimators*.

Transformers

Transformers transform one DataFrame into another, normally by appending one or more columns.

The first step in our spam classification algorithm is to split each message into an array of words. This is called **tokenization**. We can use the Tokenizer transformer, provided by MLlib:

```
scala> import org.apache.spark.ml.feature._  
import org.apache.spark.ml.feature._  
  
scala> val tokenizer = new Tokenizer()  
tokenizer: org.apache.spark.ml.feature.Tokenizer = tok_75559f60e8cf
```

The behavior of transformers can be customized through getters and setters. The easiest way of obtaining a list of the parameters available is to call the `.explainParams` method:

```
scala> println(tokenizer.explainParams)  
inputCol: input column name (undefined)  
outputCol: output column name (default: tok_75559f60e8cf__output)
```

We see that the behavior of a Tokenizer instance can be customized using two parameters: `inputCol` and `outputCol`, describing the header of the column containing the input (the string to be tokenized) and the output (the array of words), respectively. We can set these parameters using the `setInputCol` and `setOutputCol` methods.

We set `inputCol` to "text", since that is what the column is called in our training and test DataFrames. We will set `outputCol` to "words":

```
scala> tokenizer.setInputCol("text").setOutputCol("words")  
org.apache.spark.ml.feature.Tokenizer = tok_75559f60e8cf
```

In due course, we will integrate `tokenizer` into a pipeline, but, for now, let's just use it to transform the training DataFrame, to verify that it works correctly.

```
scala> val tokenizedDF = tokenizer.transform(trainDF)  
tokenizedDF: DataFrame = [fileName: string, text: string, category:  
string, words: array<string>]  
  
scala> tokenizedDF.show
```

fileName	text	category	words
file:/Users... Subject: auto...	spam	[subject:, auto, ...]	
file:/Users... Subject: want...	spam	[subject:, want, ...]	
file:/Users... Subject: n't ...	spam	[subject:, n't, m...]	
file:/Users... Subject: amaz...	spam	[subject:, amaze,...]	
file:/Users... Subject: help...	spam	[subject:, help, ...]	
file:/Users... Subject: beat...	spam	[subject:, beat, ...]	
...			

The `tokenizer` transformer produces a new DataFrame with an additional column, `words`, containing an array of the words in the `text` column.

Clearly, we can use our `tokenizer` to transform any DataFrame with the correct schema. We could, for instance, use it on the test set. Much of machine learning involves calling the same (or a very similar) pipeline on different data sets. By providing the pipeline abstraction, MLlib facilitates reasoning about complex machine learning algorithms consisting of many cleaning, transformation, and modeling components.

The next step in our pipeline is to calculate the frequency of occurrence of each word in each message. We will eventually use these frequencies as features in our algorithm. We will use the `HashingTF` transformer to transform from arrays of words to word frequency vectors for each message.

The `HashingTF` transformer constructs a sparse vector of word frequencies from input iterables. Each element in the word array gets transformed to a hash code. This hash code is truncated to a value between 0 and a large number n , the total number of elements in the output vector. The term frequency vector is just the number of occurrences of the truncated hash.

Let's run through an example manually to understand how this works. We will calculate the term frequency vector for `Array("the", "dog", "jumped", "over", "the")`. Let's set n , the number of elements in the sparse output vector, to 16 for this example. The first step is to calculate the hash code for each element in our array. We can use the built-in `##` method, which calculates a hash code for any object:

```
scala> val words = Array("the", "dog", "jumped", "over", "the")
words: Array[String] = Array(the, dog, jumped, over, the)

scala> val hashCodes = words.map { _.## }
hashCodes: Array[Int] = Array(114801, 99644, -1148867251, 3423444,
114801)
```

To transform the hash codes into valid vector indices, we take the modulo of each hash by the size of the vector (16, in this case):

```
scala> val indices = hashCodes.map { code => Math.abs(code % 16) }
indices: Array[Int] = Array(1, 12, 3, 4, 1)
```

We can then create a mapping from indices to the number of times that index appears:

```
scala> val indexFrequency = indices.groupBy(identity).mapValues {
  _.size.toDouble
}
indexFrequency: Map[Int,Double] = Map(4 -> 1.0, 1 -> 2.0, 3 -> 1.0, 12 ->
1.0)
```

Finally, we can convert this map to a sparse vector, where the value at each element in the vector is the frequency with which this particular index occurs:

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

scala> val termFrequencies = Vectors.sparse(16, indexFrequency.toSeq)
termFrequencies: linalg.Vector = (16, [1,3,4,12], [2.0,1.0,1.0,1.0])
```

Note that the `.toString` output for a sparse vector consists of three elements: the total size of the vector, followed by two lists: the first is a series of indices, and the second is a series of values at those indices.

Using a sparse vector provides a compact and efficient way of representing the frequency of occurrence of words in the message, and is exactly how `HashingTF` works under the hood. The disadvantage is that the mapping from words to indices is not necessarily unique: truncating hash codes by the length of the vector will map different strings to the same index. This is known as a *collision*. The solution is to make n large enough that the frequency of collisions is minimized.

 HashingTF is similar to building a hash table (for example, a Scala map) whose keys are words and whose values are the number of times that word occurs in the message, with one important difference: it does not attempt to deal with hash collisions. Thus, if two words map to the same hash, they will have the wrong frequency. There are two advantages to using this algorithm over just constructing a hash table:

- We do not have to maintain a list of distinct words in memory.
- Each e-mail can be transformed to a vector independently of all others: we do not have to reduce over different partitions to get the set of keys in the map. This greatly eases applying this algorithm to each e-mail in a distributed manner, since we can apply the `HashingTF` transformation on each partition independently.

The main disadvantage is that we must use machine learning algorithms that can take advantage of the sparse representation efficiently. This is the case with logistic regression, which we will use here.

As you might expect, the `HashingTF` transformer takes, as parameters, the input and output columns. It also takes a parameter defining the number of distinct hash buckets in the vector. Increasing the number of buckets decreases the number of collisions. In practice, a value between $2^{18} = 262144$ and $2^{20} = 1048576$ is recommended.

```
scala> val hashingTF = (new HashingTF()
    .setInputCol("words")
    .setOutputCol("features")
    .setNumFeatures(1048576))

hashingTF: org.apache.spark.ml.feature.HashingTF = hashingTF_3b78eca9595c

scala> val hashedDF = hashingTF.transform(tokenizedDF)
```

```
hashedDF: DataFrame = [fileName: string, text: string, category: string,
words: array<string>, features: vector]
```

```
scala> hashedDF.select("features").show
+-----+
|      features|
+-----+
|(1048576,[0,33,36...])
|(1048576,[0,36,40...])
|(1048576,[0,33,34...])
|(1048576,[0,33,36...])
|(1048576,[0,33,34...])
|(1048576,[0,33,34...])
+-----+
```

Each element in the features column is a sparse vector:

```
scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row

scala> val firstRow = hashedDF.select("features").first
firstRow: org.apache.spark.sql.Row = ...

scala> val Row(v:Vector) = firstRow
v: Vector = (1048576,[0,33,36,37,...],[1.0,3.0,4.0,1.0,...])
```

We can thus interpret our vector as: the word that hashes to element 33 occurs three times, the word that hashes to element 36 occurs four times etc.

Estimators

We now have the features ready for logistic regression. The last step prior to running logistic regression is to create the target variable. We will transform the category column in our DataFrame to a binary 0/1 target column. Spark provides a `StringIndexer` class that replaces a set of strings in a column with doubles. A `StringIndexer` is not a transformer: it must first be 'fitted' to a set of categories to calculate the mapping from string to numeric value. This introduces the second class of components in the pipeline API: *estimators*.

Unlike a transformer, which works "out of the box", an estimator must be fitted to a DataFrame. For our string indexer, the fitting process involves obtaining the list of unique strings ("spam" and "ham") and mapping each of these to a double. The fitting process outputs a transformer which can be used on subsequent DataFrames.

```
scala> val indexer = (new StringIndexer()
    .setInputCol("category")
    .setOutputCol("label"))
indexer: org.apache.spark.ml.feature.StringIndexer = strIdx_16db03fd0546

scala> val indexTransform = indexer.fit(trainDF)
indexTransform: StringIndexerModel = strIdx_16db03fd0546
```

The transformer produced by the fitting process has a `labels` attribute describing the mapping it applies:

```
scala> indexTransform.labels
Array[String] = Array(ham, spam)
```

Each label will get mapped to its index in the array: thus, our transformer maps ham to 0 and spam to 1:

```
scala> val labelledDF = indexTransform.transform(hashedDF)
labelledDF: org.apache.spark.sql.DataFrame = [fileName: string, text: string, category: string, words: array<string>, features: vector, label: double]
```

```
scala> labelledDF.select("category", "label").distinct.show
+-----+----+
|category|label|
+-----+----+
|    ham|  0.0|
|   spam|  1.0|
+-----+----+
```

We now have the feature vectors and classification labels in the correct format for logistic regression. The component for performing logistic regression is an estimator: it is fitted to a training DataFrame to create a trained model. The model can then be used to transform test DataFrames.

```
scala> import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.classification.LogisticRegression

scala> val classifier = new LogisticRegression().setMaxIter(50)
classifier: LogisticRegression = logreg_a5e921e7c1a1
```

The LogisticRegression estimator expects the feature column to be named "features" and the label column (the target) to be named "label", by default. There is no need to set these explicitly, since they match the column names set by hashingTF and indexer. There are several parameters that can be set to control how logistic regression works:

```
scala> println(classifier.explainParams)

elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For
alpha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1
penalty. (default: 0.0)

fitIntercept: whether to fit an intercept term (default: true)

labelCol: label column name (default: label)

maxIter: maximum number of iterations (>= 0) (default: 100, current: 50)

regParam: regularization parameter (>= 0) (default: 0.0)

threshold: threshold in binary classification prediction, in range [0, 1]
(default: 0.5)

tol: the convergence tolerance for iterative algorithms (default: 1.0E-6)

...
```

For now, we just set the maxIter parameter. We will look at the effect of other parameters, such as regularization, later on. Let's now fit the classifier to labelledDF:

```
scala> val trainedClassifier = classifier.fit(labelledDF)
trainedClassifier: LogisticRegressionModel = logreg_353d18f6a5f0
```

This produces a transformer that we can use on a DataFrame with a features column. The transformer appends a prediction column and a probability column. We can, for instance use `trainedClassifier` to transform `labelledDF`, the training set itself:

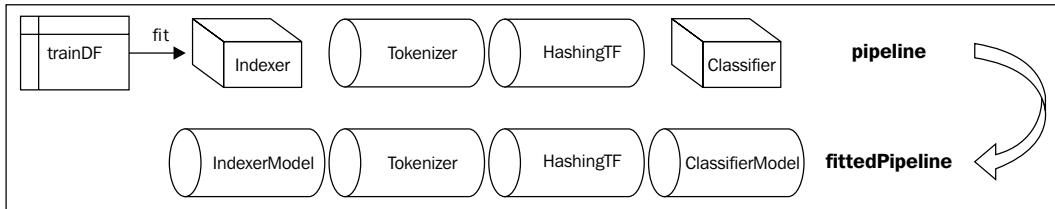
```
scala> val labelledDFWithPredictions = trainedClassifier.transform(  
    labelledDF)  
labelledDFWithPredictions: DataFrame = [fileName: string, ...  
  
scala> labelledDFWithPredictions.select($"label", $"prediction").show  
+-----+-----+  
|label|prediction|  
+-----+-----+  
| 1.0 |      1.0 |  
| 1.0 |      1.0 |  
| 1.0 |      1.0 |  
| 1.0 |      1.0 |  
| 1.0 |      1.0 |  
| 1.0 |      1.0 |  
| 1.0 |      1.0 |  
| 1.0 |      1.0 |  
| 1.0 |      1.0 |  
+-----+-----+
```

A quick way of checking the performance of our model is to just count the number of misclassified messages:

```
scala> labelledDFWithPredictions.filter {  
    $"label" != $"prediction"  
}.count  
Long = 1
```

In this case, logistic regression managed to correctly classify every message but one in the training set. This is perhaps unsurprising, given the large number of features and the relatively clear demarcation between the words used in spam and legitimate e-mails.

Of course, the real test of a model is not how well it performs on the training set, but how well it performs on a test set. To test this, we could just push the test DataFrame through the same stages that we used to train the model, replacing estimators with the fitted transformer that they produced. MLlib provides the *pipeline* abstraction to facilitate this: we wrap an ordered list of transformers and estimators in a pipeline. This pipeline is then fitted to a DataFrame corresponding to the training set. The fitting produces a `PipelineModel` instance, equivalent to the pipeline but with estimators replaced by transformers, as shown in this diagram:



Let's construct the pipeline for our logistic regression spam filter:

```
scala> import org.apache.spark.ml.Pipeline  
import org.apache.spark.ml.Pipeline  
  
scala> val pipeline = new Pipeline().setStages(  
    Array(indexer, tokenizer, hashingTF, classifier)  
)  
pipeline: Pipeline = pipeline_7488113e284d
```

Once the pipeline is defined, we fit it to the DataFrame holding the training set:

```
scala> val fittedPipeline = pipeline.fit(trainDF)  
fittedPipeline: org.apache.spark.ml.PipelineModel = pipeline_089525c6f100
```

When fitting a pipeline to a DataFrame, estimators and transformers are treated differently:

- Transformers are applied to the DataFrame and copied, as is, into the pipeline model.
- Estimators are fitted to the DataFrame, producing a transformer. The transformer is then applied to the DataFrame, and appended to the pipeline model.

We can now apply the pipeline model to the test set:

```
scala> val testDFWithPredictions = fittedPipeline.transform(testDF)
testDFWithPredictions: DataFrame = [fileName: string, ...]
```

This has added a prediction column to the DataFrame with the predictions of our logistic regression model. To measure the performance of our algorithm, we calculate the classification error on the test set:

```
scala> testDFWithPredictions.filter {
  &nbsp;&nbsp;&nbsp; $"label" != $"prediction"
}.count
Long = 20
```

Thus, our naive logistic regression algorithm, with no model selection, or regularization, mis-classifies 2.3% of e-mails. You may, of course, get slightly different results, since the train-test split was random.

Let's save the training and test DataFrames, with predictions, as parquet files:

```
scala> import org.apache.spark.sql.SaveMode
import org.apache.spark.sql.SaveMode

scala> (labelledDFWithPredictions
  &nbsp;&nbsp;&nbsp;.select("fileName", "label", "prediction", "probability")
  &nbsp;&nbsp;&nbsp;.write.mode(SaveMode.Overwrite)
  &nbsp;&nbsp;&nbsp;.parquet("transformedTrain.parquet"))

scala> (testDFWithPredictions
  &nbsp;&nbsp;&nbsp;.select("fileName", "label", "prediction", "probability")
  &nbsp;&nbsp;&nbsp;.write.mode(SaveMode.Overwrite)
  &nbsp;&nbsp;&nbsp;.parquet("transformedTest.parquet"))
```



In spam classification, a false positive is considerably worse than a false negative: it is much worse to classify a legitimate message as spam, than it is to let a spam message through. To account for this, we could increase the threshold for classification: only messages that score, for instance, 0.7 or above would get classified as spam. This raises the obvious question of choosing the right threshold. One way to do this would be to investigate the false positive rate incurred in the test set for different thresholds, and choosing the lowest threshold to give us an acceptable false positive rate. A good way of visualizing this is to use ROC curves, which we will investigate in the next section.

Evaluation

Unfortunately, the functionality for evaluating model quality in the pipeline API remains limited, as of version 1.5.2. Logistic regression does output a summary containing several evaluation metrics (available through the `summary` attribute on the trained model), but these are calculated on the training set. In general, we want to evaluate the performance of the model both on the training set and on a separate test set. We will therefore dive down to the underlying MLlib layer to access evaluation metrics.

MLlib provides a module, `org.apache.spark.mllib.evaluation`, with a set of classes for assessing the quality of a model. We will use the `BinaryClassificationMetrics` class here, since spam classification is a binary classification problem. Other evaluation classes provide metrics for multi-class models, regression models and ranking models.

As in the previous section, we will illustrate the concepts in the shell, but you will find analogous code in the `ROC.scala` script in the code examples for this chapter. We will use `breeze-viz` to plot curves, so, when starting the shell, we must ensure that the relevant libraries are on the classpath. We will use SBT assembly, as described in *Chapter 10, Distributed Batch Processing with Spark* (specifically, the *Building and running standalone programs* section), to create a JAR with the required dependencies. We will then pass this JAR to the Spark shell, allowing us to import `breeze-viz`. Let's write a `build.sbt` file that declares a dependency on `breeze-viz`:

```
// build.sbt
name := "spam_filter"

scalaVersion := "2.10.5"

libraryDependencies += Seq(
```

```
"org.apache.spark" %% "spark-core" % "1.5.2" % "provided",
"org.apache.spark" %% "spark-mllib" % "1.5.2" % "provided",
"org.scalanlp" %% "breeze" % "0.11.2",
"org.scalanlp" %% "breeze-viz" % "0.11.2",
"org.scalanlp" %% "breeze-natives" % "0.11.2"
)
```

Package the dependencies into a jar with:

```
$ sbt assembly
```

This will create a jar called `spam_filter-assembly-0.1-SNAPSHOT.jar` in the `target/scala-2.10/` directory. To include this jar in the Spark shell, re-start the shell with the `--jars` command line argument:

```
$ spark-shell --jars=target/scala-2.10/spam_filter-assembly-0.1-SNAPSHOT.jar
```

To verify that the packaging worked correctly, try to import `breeze.plot`:

```
scala> import breeze.plot._  
import breeze.plot._
```

Let's load the test set, with predictions, which we created in the previous section and saved as a parquet file:

```
scala> val testDFWithPredictions = sqlContext.read.parquet(  
    "transformedTest.parquet")  
testDFWithPredictions: org.apache.spark.sql.DataFrame = [fileName:  
string, label: double, prediction: double, probability: vector]
```

The `BinaryClassificationMetrics` object expects an `RDD[(Double, Double)]` object of pairs of scores (the probability assigned by the classifier that a particular e-mail is spam) and labels (whether an e-mail is actually spam). We can extract this RDD from our DataFrame:

```
scala> import org.apache.spark.mllib.linalg.Vector  
import org.apache.spark.mllib.linalg.Vector  
  
scala> import org.apache.spark.sql.Row  
import org.apache.spark.sql.Row  
  
scala> val scoresLabels = testDFWithPredictions.select(
```

```
"probability", "label") .map {  
    case Row(probability:Vector, label:Double) =>  
        (probability(1), label)  
}  
org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[3] at map  
at <console>:23  
  
scala> scoresLabels.take(5).foreach(println)  
(0.999999967713409,1.0)  
(0.9999983827108793,1.0)  
(0.9982059900606365,1.0)  
(0.9999790713978142,1.0)  
(0.9999999999999272,1.0)
```

We can now construct the `BinaryClassificationMetrics` instance:

```
scala> import org.apache.spark.mllib.evaluation.  
BinaryClassificationMetrics  
import mllib.evaluation.BinaryClassificationMetrics  
  
scala> val bm = new BinaryClassificationMetrics(scoresLabels)  
bm: BinaryClassificationMetrics = mllib.evaluation.BinaryClassificationMe  
trics@254ed9ba
```

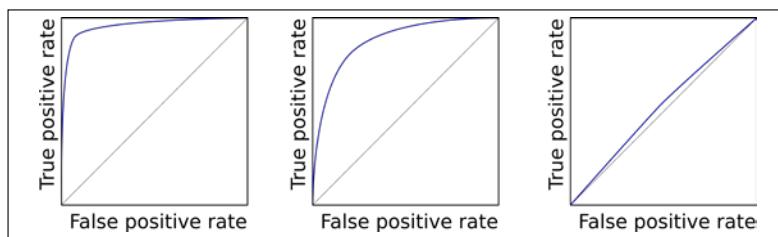
The `BinaryClassificationMetrics` objects contain many useful metrics for evaluating the performance of a classification model. We will look at the **receiver operating characteristic (ROC)** curve.

ROC Curves

Imagine gradually decreasing, from 1.0, the probability threshold at which we assume a particular e-mail is spam. Clearly, when the threshold is set to 1.0, no e-mails will get classified as spam. This means that there will be no **false positives** (ham messages which we incorrectly classify as spam), but it also means that there will be no **true positives** (spam messages that we correctly identify as spam): all spam e-mails will be incorrectly identified as ham.

As we gradually lower the probability threshold at which we assume a particular e-mail is spam, our spam filter will, hopefully, start identifying a large fraction of e-mails as spam. The vast majority of these will, if our algorithm is well-designed, be real spam. Thus, our rate of true positives increases. As we gradually lower the threshold, we start classifying messages about which we are less sure of as spam. This will increase the number of messages correctly identified as spam, but it will also increase the number of false positives.

The ROC curve plots, for each threshold value, the fraction of true positives against the fraction of false positives. In the best case, the curve is always 1: this happens when all spam messages are given a score of 1.0, and all ham messages are given a score of 0.0. By contrast, the worst case happens when the curve is a diagonal $P(\text{true positive}) = P(\text{false positive})$, which occurs when our algorithm does no better than random. In general, ROC curves fall somewhere in between, forming a convex shell above the diagonal. The deeper this shell, the better our algorithm.



(left) ROC curve for a model performing much better than random: the curve reaches very high true positive rates for a low false positive rate.

(middle) ROC curve for a model performing significantly better than random.

(right) ROC curve for a model performing only marginally better than random: the true positive rate is only marginally larger than the rate of false positives, for any given threshold, meaning that nearly half the examples are misclassified.

We can calculate an array of points on the ROC curve using the `.roc` method on our `BinaryClassificationMetrics` instance. This returns an `RDD[(Double, Double)]` of (*false positive*, *true positive*) fractions for each threshold value. We can collect this as an array:

```
scala> val rocArray = bm.roc.collect
rocArray: Array[(Double, Double)] = Array((0.0,0.0),
(0.0,0.16793893129770993), ...
```

Of course, an array of numbers is not very enlightening, so let's plot the ROC curve with breeze-viz. We start by transforming our array of pairs into two arrays, one of false positives and one of true positives:

```
scala> val falsePositives = rocArray.map { _.1 }
falsePositives: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0, ...
scala> val truePositives = rocArray.map { _.2 }
truePositives: Array[Double] = Array(0.0, 0.16793893129770993,
0.19083969465...
```

Let's plot these two arrays:

```
scala> import breeze.plot._
import breeze.plot.

scala> val f = Figure()
f: breeze.plot.Figure = breeze.plot.Figure@3aa746cd

scala> val p = f.subplot(0)
p: breeze.plot.Plot = breeze.plot.Plot@5ed1438a

scala> p += plot(falsePositives, truePositives)
p += plot(falsePositives, truePositives)

scala> p.xlabel = "false positives"
p.xlabel: String = false positives

scala> p.ylabel = "true positives"
p.ylabel: String = true positives

scala> p.title = "ROC"
```

```
p.title: String = ROC
```

```
scala> f.refresh
```

The ROC curve hits 1.0 for a small value of x : that is, we retrieve all true positives at the cost of relatively few false positives. To visualize the curve more accurately, it is instructive to limit the range on the x -axis from 0 to 0.1.

```
scala> p.xlim = (0.0, 0.1)
p.xlim: (Double, Double) = (0.0,0.1)
```

We also need to tell breeze-viz to use appropriate tick spacing, which requires going down to the JFreeChart layer underlying breeze-viz:

```
scala> import org.jfree.chart.axis.NumberTickUnit
import org.jfree.chart.axis.NumberTickUnit

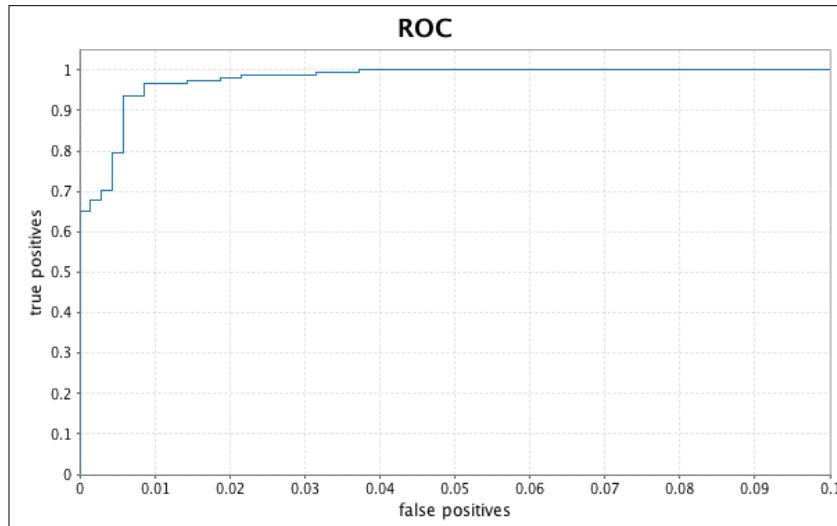
scala> p.xaxis.setTickUnit(new NumberTickUnit(0.01))

scala> p.yaxis.setTickUnit(new NumberTickUnit(0.1))
```

We can now save the graph:

```
scala> f.saveas("roc.png")
```

This produces the following graph, stored in `roc.png`:



ROC curve for spam classification with logistic regression.
Note that we have limited the false positive axis at 0.1

By looking at the graph, we see that we can filter out 85% of spam without a single **false positive**. Of course, we would need a larger test set to really validate this assumption.

A graph is useful to really understand the behavior of a model. Sometimes, however, we just want to have a single measure of the quality of a model. The area under the ROC curve can be a good such metric:

```
scala> bm.areaUnderROC
res21: Double = 0.9983061235861147
```

This can be interpreted as follows: given any two messages randomly drawn from the test set, one of which is ham, and one of which is spam, there is a 99.8% probability that the model assigned a greater likelihood of spam to the spam message than to the ham message.

Other useful measures of model quality are the precision and recall for particular thresholds, or the F1 score. All of these are provided by the `BinaryClassificationMetrics` instance. The API documentation lists the methods available: <https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.evaluation.BinaryClassificationMetrics>.

Regularization in logistic regression

One of the dangers of machine learning is over-fitting: the algorithm captures not only the signal in the training set, but also the statistical noise that results from the finite size of the training set.

A way to mitigate over-fitting in logistic regression is to use regularization: we impose a penalty for large values of the parameters when optimizing. We can do this by adding a penalty to the cost function that is proportional to the magnitude of the parameters. Formally, we re-write the logistic regression cost function (described in *Chapter 2, Manipulating Data with Breeze*) as:

$$\text{Cost}(\text{params}) = \text{Cost}_{LR}(\text{params}) + \lambda \|\text{params}\|_n$$

where Cost_{LR} is the normal logistic regression cost function:

$$\text{Cost}_{LR}(\text{params}) = \sum_i \text{target}_i \times (\text{params} \cdot \text{training}_i) - \log[\exp(\text{params} \cdot \text{training}_i) + 1]$$

Here, $params$ is the vector of parameters, $training_i$ is the vector of features for the i th training example, and $target_i$ is 1 if the i th training example is spam, and 0 otherwise. This is identical to the logistic regression cost-function introduced in Chapter 2, *Manipulating data with Breeze*, apart from the addition of the regularization term $\lambda \|params\|_n$, the L_n norm of the parameter vector. The most common value of n is 2, in which case $\|params\|_2$ is just the magnitude of the parameter vector:

$$\|params\|_2 = \sqrt{\sum_i params_i^2}$$

The additional regularization term drives the algorithm to reduce the magnitude of the parameter vector. When using regularization, features must all have comparable magnitude. This is commonly achieved by normalizing the features. The logistic regression estimator provided by MLlib normalizes all features by default. This can be turned off with the `setStandardization` parameter.

Spark has two hyperparameters that can be tweaked to control regularization:

- The type of regularization, set with the `elasticNetParam` parameter. A value of 0 indicates L_2 regularization.
- The degree of regularization (λ in the cost function), set with the `regParam` parameter. A high value of the regularization parameter indicates a strong regularization. In general, the greater the danger of over-fitting, the larger the regularization parameter ought to be.

Let's create a new logistic regression instance that uses regularization:

```
scala> val lrWithRegularization = (new LogisticRegression()
    .setMaxIter(50))
lrWithRegularization: LogisticRegression = logreg_16b65b325526

scala> lrWithRegularization.setElasticNetParam(0)
lrWithRegularization.type = logreg_1e3584a59b3a
```

To choose the appropriate value of λ , we fit the pipeline to the training set and calculate the classification error on the test set for several values of λ . Further on in the chapter, we will learn about cross-validation in MLlib, which provides a much more rigorous way of choosing hyper-parameters.

```
scala> val lambdas = Array(0.0, 1.0E-12, 1.0E-10, 1.0E-8)
lambdas: Array[Double] = Array(0.0, 1.0E-12, 1.0E-10, 1.0E-8)

scala> lambdas foreach { lambda =>
    lrWithRegularization.setRegParam(lambda)
    val pipeline = new Pipeline().setStages(
        Array(indexer, tokenizer, hashingTF, lrWithRegularization))
    val model = pipeline.fit(trainDF)
    val transformedTest = model.transform(testDF)
    val classificationError = transformedTest.filter {
        $"prediction" !== $"label"
    }.count
    println(s"$lambda => $classificationError")
}
0 => 20
1.0E-12 => 20
1.0E-10 => 20
1.0E-8 => 23
```

For our example, we see that any attempt to add L₂ regularization leads to a decrease in classification accuracy.

Cross-validation and model selection

In the previous example, we validated our approach by withholding 30% of the data when training, and testing on this subset. This approach is not particularly rigorous: the exact result changes depending on the random train-test split. Furthermore, if we wanted to test several different hyperparameters (or different models) to choose the best one, we would, unwittingly, choose the model that best reflects the specific rows in our test set, rather than the population as a whole.

This can be overcome with *cross-validation*. We have already encountered cross-validation in *Chapter 4, Parallel Collections and Futures*. In that chapter, we used random subsample cross-validation, where we created the train-test split randomly.

In this chapter, we will use **k-fold cross-validation**: we split the training set into k parts (where, typically, k is 10 or 3) and use $k-1$ parts as the training set and the last as the test set. The train/test cycle is repeated k times, keeping a different part as test set each time.

Cross-validation is commonly used to choose the best set of hyperparameters for a model. To illustrate choosing suitable hyperparameters, we will go back to our regularized logistic regression example. Instead of intuiting the hyper-parameters ourselves, we will choose the hyper-parameters that give us the best cross-validation score.

We will explore setting both the regularization type (through `elasticNetParam`) and the degree of regularization (through `regParam`). A crude, but effective way to find good values of the parameters is to perform a grid search: we calculate the cross-validation score for every pair of values of the regularization parameters of interest.

We can build a grid of parameters using MLlib's `ParamGridBuilder`.

```
scala> import org.apache.spark.ml.tuning.{ParamGridBuilder,
CrossValidator}

import org.apache.spark.ml.tuning.{ParamGridBuilder, CrossValidator}

scala> val paramGridBuilder = new ParamGridBuilder()
paramGridBuilder: ParamGridBuilder = ParamGridBuilder@1dd694d0

To add hyper-parameters over which to optimize to the grid, we use the addGrid
method:

scala> val lambdas = Array(0.0, 1.0E-12, 1.0E-10, 1.0E-8)
Array[Double] = Array(0.0, 1.0E-12, 1.0E-10, 1.0E-8)

scala> val elasticNetParams = Array(0.0, 1.0)
elasticNetParams: Array[Double] = Array(0.0, 1.0)

scala> paramGridBuilder.addGrid(
  lrWithRegularization.regParam, lambdas).addGrid(
  lrWithRegularization.elasticNetParam, elasticNetParams)
paramGridBuilder.type = ParamGridBuilder@1dd694d0
```

Once all the dimensions are added, we can just call the `build` method on the builder to build the grid:

```
scala> val paramGrid = paramGridBuilder.build
paramGrid: Array[org.apache.spark.ml.param.ParamMap] =
Array({
  logreg_f7dfb27bed7d-elasticNetParam: 0.0,
  logreg_f7dfb27bed7d-regParam: 0.0
}, {
  logreg_f7dfb27bed7d-elasticNetParam: 1.0,
  logreg_f7dfb27bed7d-regParam: 0.0
} ...)

scala> paramGrid.length
Int = 8
```

As we can see, the grid is just a one-dimensional array of sets of parameters to pass to the logistic regression model prior to fitting.

The next step in setting up the cross-validation pipeline is to define a metric for comparing model performance. Earlier in the chapter, we saw how to use `BinaryClassificationMetrics` to estimate the quality of a model. Unfortunately, the `BinaryClassificationMetrics` class is part of the core MLlib API, rather than the new pipeline API, and is thus not (easily) compatible. The pipeline API offers a `BinaryClassificationEvaluator` class instead. This class works directly on `DataFrames`, and thus fits perfectly into the pipeline API flow:

```
scala> import org.apache.spark.ml.evaluation.
BinaryClassificationEvaluator
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

scala> val evaluator = new BinaryClassificationEvaluator()
evaluator: BinaryClassificationEvaluator = binEval_64b08538f1a2

scala> println(evaluator.explainParams)
labelCol: label column name (default: label)
metricName: metric name in evaluation (areaUnderROC|areaUnderPR)
(default: areaUnderROC)
rawPredictionCol: raw prediction (a.k.a. confidence) column name
(default: rawPrediction)
```

From the parameter list, we see that the `BinaryClassificationEvaluator` class supports two metrics: the area under the ROC curve, and the area under the precision-recall curve. It expects, as input, a DataFrame containing a `label` column (the model truth) and a `rawPrediction` column (the column containing the probability that an e-mail is spam or ham).

We now have all the parameters we need to run cross-validation. We first build the pipeline, and then pass the pipeline, the evaluator and the array of parameters over which to run the cross-validation to an instance of `CrossValidator`:

```
scala> val pipeline = new Pipeline().setStages(  
    Array(indexer, tokenizer, hashingTF, lrWithRegularization))  
pipeline: Pipeline = pipeline_3ed29f72a4cc  
  
scala> val crossval = (new CrossValidator()  
    .setEstimator(pipeline)  
    .setEvaluator(evaluator)  
    .setEstimatorParamMaps(paramGrid)  
    .setNumFolds(3))  
crossval: CrossValidator = cv_5ebfa1143a9d
```

We will now fit `crossval` to `trainDF`:

```
scala> val cvModel = crossval.fit(trainDF)  
cvModel: CrossValidatorModel = cv_5ebfa1143a9d
```

This step can take a fairly long time (over an hour on a single machine). This creates a transformer, `cvModel`, corresponding to the logistic regression object with the parameters that best represent `trainDF`. We can use it to predict the classification error on the test DataFrame:

```
scala> cvModel.transform(testDF).filter {  
    $"prediction" != $"label"  
}.count  
Long = 20
```

Cross-validation has therefore resulted in a model that performs identically to the original, naive logistic regression model with no hyper-parameters. `cvModel` also contains a list of the evaluation score for each set of parameter in the parameter grid:

```
scala> cvModel.avgMetrics  
Array[Double] = Array(0.996427805316161, ...)
```

The easiest way to relate this to the hyper-parameters is to zip it with `cvModel.getEstimatorParamMaps`. This gives us a list of (*hyperparameter values, cross-validation score*) pairs:

```
scala> val params2score = cvModel.getEstimatorParamMaps.zip(
  cvModel.avgMetrics)
Array[(ml.param.ParamMap[Double])] = Array(({{
  logreg_8f107aabb304-elasticNetParam: 0.0,
  logreg_8f107aabb304-regParam: 0.0
}, 0.996427805316161), ...
```



```
scala> params2score.foreach {
  case (params, score) =>
    val lambda = params(lrWithRegularization.regParam)
    val elasticNetParam = params(
      lrWithRegularization.elasticNetParam)
    val l2OrL1 = if(elasticNetParam == 0.0) "L2" else "L1"
    println(s"$l2OrL1, $lambda => $score")
}
```

L2, 0.0 => 0.996427805316161
L1, 0.0 => 0.996427805316161
L2, 1.0E-12 => 0.9964278053175655
L1, 1.0E-12 => 0.9961429402772803
L2, 1.0E-10 => 0.9964382546369551
L1, 1.0E-10 => 0.9962223090037103
L2, 1.0E-8 => 0.9964159754613495
L1, 1.0E-8 => 0.9891008277659763

The best set of hyper-parameters correspond to L₂ regularization with a regularization parameter of 1E-10, though this only corresponds to a tiny improvement in AUC.

This completes our spam filter example. We have successfully trained a spam filter for this particular Ling-Spam dataset. To obtain better results, one could experiment with better feature extraction: we could remove stop words or use TF-IDF vectors, rather than just term frequency vectors as features, and we could add additional features like the length of messages, or even *n*-grams. We could also experiment with non-linear algorithms, such as random forest. All of these steps would be straightforward to add to the pipeline.

Beyond logistic regression

We have concentrated on logistic regression in this chapter, but MLlib offers many alternative algorithms that will capture non-linearity in the data more effectively. The consistency of the pipeline API makes it easy to try out different algorithms and see how they perform. The pipeline API offers decision trees, random forest and gradient boosted trees for classification, as well as a simple feed-forward neural network, which is still experimental. It offers lasso and ridge regression and decision trees for regression, as well as PCA for dimensionality reduction.

The lower level MLlib API also offers principal component analysis for dimensionality reduction, several clustering methods including k -means and latent Dirichlet allocation and recommender systems using alternating least squares.

Summary

MLlib tackles the challenge of devising scalable machine learning algorithms head-on. In this chapter, we used it to train a simple scalable spam filter. MLlib is a vast, rapidly evolving library. The best way to learn more about what it can offer is to try and port code that you might have written using another library (such as scikit-learn).

In the next chapter, we will look at how to build web APIs and interactive visualizations to share our results with the rest of the world.

References

The best reference is the online documentation, including:

- The pipeline API: <http://spark.apache.org/docs/latest/ml-features.html>
- A full list of transformers: <http://spark.apache.org/docs/latest/mllib-guide.html#sparkml-high-level-apis-for-ml-pipelines>

Advanced Analytics with Spark, by Sandy Ryza, Uri Laserson, Sean Owen and Josh Wills provides a detailed and up-to-date introduction to machine learning with Spark.

There are several books that introduce machine learning in more detail than we can here. We have mentioned *The Elements of Statistical Learning*, by Friedman, Tibshirani and Hastie several times in this book. It is one of the most complete introductions to the mathematical underpinnings of machine learning currently available.

Andrew Ng's Machine Learning course on <https://www.coursera.org/> provides a good introduction to machine learning. It uses Octave/MATLAB as the programming language, but should be straightforward to adapt to Breeze and Scala.

13

Web APIs with Play

In the first 12 chapters of this book, we introduced basic tools and libraries for anyone wanting to build data science applications: we learned how to interact with SQL and MongoDB databases, how to build fast batch processing applications using Spark, how to apply state-of-the-art machine learning algorithms using MLlib, and how to build modular concurrent applications in Akka.

In the last chapters of this book, we will branch out to look at a web framework: *Play*. You might wonder why a web framework would feature in a data science book; surely such topics are best left to software engineers or web developers. Data scientists, however, rarely exist in a vacuum. They often need to communicate results or insights to stakeholders. As compelling as an ROC curve may be to someone well versed in statistics, it may not carry as much weight with less technical people. Indeed, it can be much easier to sell insights when they are accompanied by an engaging visualization.

Many modern interactive data visualization applications are web applications running in a web browser. Often, these involve **D3.js**, a JavaScript library for building data-driven web pages. In this chapter and the next, we will look at integrating D3 with Scala.

Writing a web application is a complex endeavor. We will split this task over this chapter and the next. In this chapter, we will learn how to write a REST API that we can use as backend for our application, or query in its own right. In the next chapter, we will look at integrating front-end code with Play to query the API exposed by the backend and display it using D3. We assume at least a basic familiarity with HTTP in this chapter: you should have read *Chapter 7, Web APIs*, at least.

Many data scientists or aspiring data scientists are unlikely to be familiar with the inner workings of web technologies. Learning how to build complex websites or web APIs can be daunting. This chapter therefore starts with a general discussion of dynamic websites and the architecture of web applications. If you are already familiar with server-side programming and with web frameworks, you can easily skip over the first few sections.

Client-server applications

A website works through the interaction between two computers: the client and the server. If you enter the URL `www.github.com/pbugnion/s4ds/graphs` in a web browser, your browser queries one of the GitHub servers. The server will look through its database for information concerning the repository that you are interested in. It will serve this information as HTML, CSS, and JavaScript to your computer. Your browser is then responsible for interpreting this response in the correct way.

If you look at the URL in question, you will notice that there are several graphs on that page. Unplug your internet connection and you can still interact with the graphs. All the information necessary for interacting with the graphs was transferred, as JavaScript, when you loaded that webpage. When you play with the graphs, the CPU cycles necessary to make those changes happen are spent on *your* computer, not a GitHub server. The code is executed *client-side*. Conversely, when you request information about a new repository, that request is handled by a GitHub server. It is said to be handled *server-side*.

A web framework like Play can be used on the server. For client-side code, we can only use a language that the client browser will understand: HTML for the layout, CSS for the styling and JavaScript, or languages that can compile to JavaScript, for the logic.

Introduction to web frameworks

This section is a brief introduction to how modern web applications are designed. Go ahead and skip it if you already feel comfortable writing backend code.

Loosely, a web framework is a set of tools and code libraries for building web applications. To understand what a web framework provides, let's take a step back and think about what you would need to do if you did not have one.

You want to write a program that listens on port 80 and sends HTML (or JSON or XML) back to clients that request it. This is simple if you are serving the same file back to every client: just load the HTML from file when you start the server, and send it to clients who request it.

So far, so good. But what if you now want to customize the HTML based on the client request? You might choose to respond differently based on part of the URL that the client put in his browser, or based on specific elements in the HTTP request. For instance, the product page on `amazon.com` is different to the payment page. You need to write code to parse the URL and the request, and then route the request to the relevant handler.

You might now want to customize the HTML returned dynamically, based on specific elements of the request. The page for every product on `amazon.com` follows the same outline, but specific elements are different. It would be wasteful to store the entire HTML content for every product. A better way is to store the details for each product in a database and inject them into an HTML template when a client requests information on that product. You can do this with a *template processor*. Of course, writing a good template processor is difficult.

You might deploy your web framework and realize that it cannot handle the traffic directed to it. You decide that handlers responding to client requests should run asynchronously. You now have to deal with concurrency.

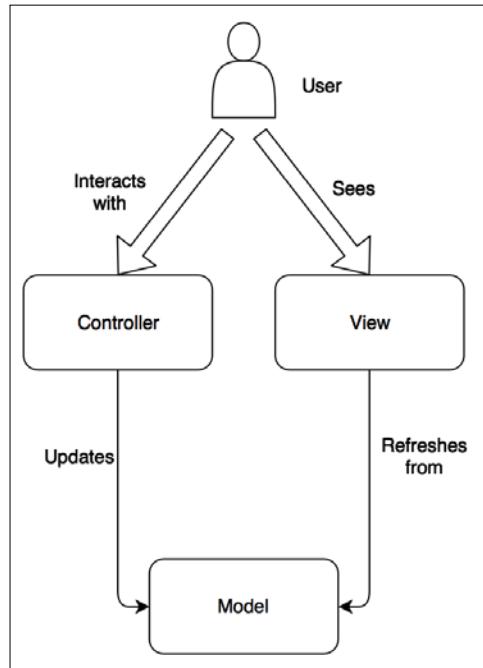
A web framework essentially provides the wires to bind everything together. Besides bundling an HTTP server, most frameworks will have a router that automatically routes a request, based on the URL, to the correct handler. In most cases, the handler will run asynchronously, giving you much better scalability. Many frameworks have a template processor that lets you write HTML (or sometimes JSON or XML) templates intuitively. Some web frameworks also provide functionality for accessing a database, for parsing JSON or XML, for formulating HTTP requests and for localization and internationalization.

Model-View-Controller architecture

Many web frameworks impose program architectures: it is difficult to provide wires to bind disparate components together without making some assumptions about what those components are. The **Model-View-Controller (MVC)** architecture is particularly popular on the Web, and it is the architecture the Play framework assumes. Let's look at each component in turn:

- The model is the data underlying the application. For example, I expect the application underlying GitHub has models for users, repositories, organizations, pull requests and so on. In the Play framework, a model is often an instance of a case class. The core responsibility of the model is to remember the current state of the application.
- Views are representations of a model or a set of models on the screen.

- The controller handles client interactions, possibly changing the model. For instance, if you *star* a project on GitHub, the controller will update the relevant models. Controllers normally carry very little application state: remembering things is the job of the models.



MVC architecture: the state of the application is provided by the model. The view provides a visual representation of the model to the user, and the controller handles logic: what to do when the user presses a button or submits a form.

The MVC framework works well because it decouples the user interface from the underlying data and structures the flow of actions: a controller can update the model state or the view, a model can send signals to the view to tell it to update, and the view merely displays that information. The model carries no information related to the user interface. This separation of concerns results in an easier mental model of information flow, better encapsulation and greater testability.

Single page applications

The client-server duality adds a degree of complication to the elegant MVC architecture. Where should the model reside? What about the controller?

Traditionally, the model and the controller ran almost entirely on the server, which just pushed the relevant HTML view to the client.

The growth in client-side JavaScript frameworks, such AngularJS, has resulted in a gradual shift to putting more code in the client. Both the controller and a temporary version of the model typically run client-side. The server just functions as a web API: if, for instance, the user updates the model, the controller will send an HTTP request to the server informing it of the change.

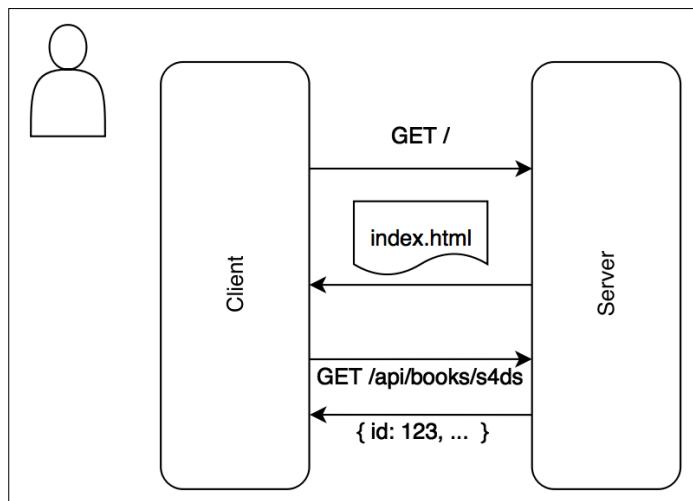
It then makes sense to think of the program running server-side and the one running client-side as two separate applications: the server persists data in databases, for instance, and provides a programmatic interface to this data, usually as a web service returning JSON or XML data. The client-side program maintains its own model and controller, and polls the server whenever it needs a new model, or whenever it needs to inform the server that the persistent view of the model should be changed.

Taken to the extreme, this results in **Single-Page Applications**. In a single-page application, the first time the client requests a page from the server, he receives the HTML and the JavaScript necessary to build the framework for the entire application. If the client needs further data from the server, he will poll the server's API. This data is returned as JSON or XML.

This might seem a little complicated in the abstract, so let's think how the Amazon website might be structured as a single-page application. We will just concern ourselves with the products page here, since that's complicated enough. Let's imagine that you are on the home page, and you hit a link for a particular product. The application running on your computer knows how to display products, for instance through an HTML template. The JavaScript also has a prototype for the model, such as:

```
{  
    product_id: undefined,  
    product_name: undefined,  
    product_price: undefined,  
    ...  
}
```

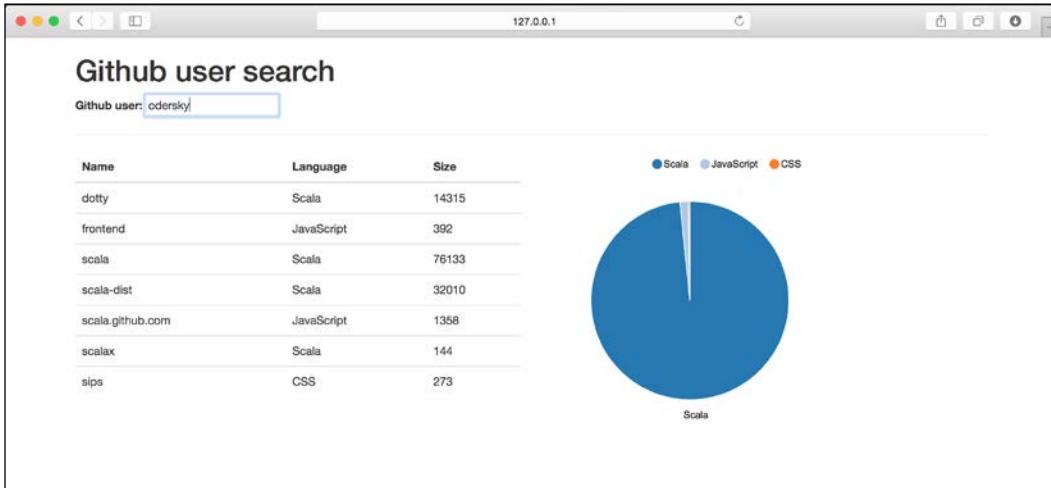
What it's currently missing is knowledge of what data to put in those fields for the product you have just selected: there is no way that information could have been sent to your computer when the website loaded, since there was no way to know what product you might click on (and sending information about every product would be prohibitively costly). So the Amazon client sends a request to the server for information on that product. The Amazon server replies with a JSON object (or maybe XML). The client then updates its model with that information. When the update is complete, an event is fired to update the view:



Client-server communications in a single-page application: when the client first accesses the website, it receives HTML, CSS and JavaScript files that contain the entire logic for the application. From then on, the client only uses the server as an API when it requests additional data. The application running in the user's web browser and the one running on the server are nearly independent. The only coupling is through the structure of the API exposed by the server.

Building an application

In this chapter and the next, we will build a single-page application that relies on an API written in Play. We will build a webpage that looks like this:



The user enters the name of someone on GitHub and can view a list of their repositories and a chart summarizing what language they use. You can find the application deployed at app.scala4datascience.com. Go ahead and give it a whirl.

To get a glimpse of the innards, type `app.scala4datascience.com/api/repos/odersky`. This returns a JSON object like:

```
[{"name": "dotty", "language": "Scala", "is_fork": true, "size": 14653},  
 {"name": "frontend", "language": "JavaScript", "is_fork": true, "size": 392},  
 {"name": "legacy-svn-scala", "language": "Scala", "is_fork": true, "size": 296706},  
 ...
```

We will build the API in this chapter, and write the front-end code in the next chapter.

The Play framework

The Play framework is a web framework built on top of Akka. It has a proven track record in industry, and is thus a reliable choice for building scalable web applications.

Play is an *opinionated* web framework: it expects you to follow the MVC architecture, and it has a strong opinion about the tools you should be using. It comes bundled with its own JSON and XML parsers, with its own tools for accessing external APIs, and with recommendations for how to access databases.

Web applications are much more complex than the command line scripts we have been developing in this book, because there are many more components: the backend code, routing information, HTML templates, JavaScript files, images, and so on. The Play framework makes strong assumptions about the directory structure for your project. Building that structure from scratch is both mind-numbingly boring and easy to get wrong. Fortunately, we can use **Typesafe activators** to bootstrap the project (you can also download the code from the Git repository in <https://github.com/pbugnion/s4ds> but I encourage you to start the project from a basic activator structure and code along instead, using the finished version as an example).

Typesafe activator is a custom version of SBT that includes templates to get Scala programmers up and running quickly. To install activator, you can either download a JAR from <https://www.typesafe.com/activator/download>, or, on Mac OS, via homebrew:

```
$ brew install typesafe-activator
```

You can then launch the activator console from the terminal. If you downloaded activator:

```
$ ./path/to/activator/activator new
```

Or, if you installed via Homebrew:

```
$ activator new
```

This starts a new project in the current directory. It starts by asking what template you want to start with. Choose `play-scala`. It then asks for a name for your application. I chose `ghub-display`, but go ahead and be creative!

Let's explore the newly created project structure (I have only retained the most important files):

```
|__ app
|   |__ controllers
|   |   |__ Application.scala
```

```
|   └── views
|       ├── main.scala.html
|       └── index.scala.html
└── build.sbt
└── conf
    ├── application.conf
    └── routes
└── project
    ├── build.properties
    └── plugins.sbt
└── public
    ├── images
    |   └── favicon.png
    ├── javascripts
    |   └── hello.js
    └── stylesheets
        └── main.css
└── test
    ├── ApplicationSpec.scala
    └── IntegrationSpec.scala
```

Let's run the app:

```
$ ./activator
[ghub-display] $ run
```

Head over to your browser and navigate to the URL `127.0.0.1:9000/`. The page may take a few seconds to load. Once it is loaded, you should see a default page that says **Your application is ready.**

Before we modify anything, let's walk through how this happens. When you ask your browser to take you to `127.0.0.1:9000/`, your browser sends an HTTP request to the server listening at that address (in this case, the Netty server bundled with Play). The request is a GET request for the route `/`. The Play framework looks in `conf/routes` to see if it has a route satisfying `/`:

```
$ cat conf/routes
# Home page
GET      /                               controllers.Application.index
...
...
```

We see that the `conf/routes` file does contain the route `/` for GET requests. The second part of that line, `controllers.Application.index`, is the name of a Scala function to handle that route (more on that in a moment). Let's experiment. Change the route end-point to `/hello`. Refresh your browser without changing the URL. This will trigger recompilation of the application. You should now see an error page:



The error page tells you that the app does not have an action for the route `/` any more. If you navigate to `127.0.0.1:9000/hello`, you should see the landing page again.

Besides learning a little of how routing works, we have also learned two things about developing Play applications:

- In development mode, code gets recompiled when you refresh your browser and there have been code changes
- Compilation and runtime errors get propagated to the web page

Let's change the route back to `/`. There is a lot more to say on routing, but it can wait till we start building our application.

The `conf/routes` file tells the Play framework to use the method `controllers.Application.index` to handle requests to `/`. Let's look at the `Application.scala` file in `app/controllers`, where the `index` method is defined:

```
// app/controllers/Application.scala
package controllers

import play.api._
import play.api.mvc._

class Application extends Controller {

    def index = Action {
        Ok(views.html.index("Your new application is ready."))
    }

}
```

We see that `controllers.Application.index` refers to the method `index` in the class `Application`. This method has return type `Action`. An `Action` is just a function that maps HTTP requests to responses. Before explaining this in more detail, let's change the action to:

```
def index = Action {  
    Ok("hello, world")  
}
```

Refresh your browser and you should see the landing page replaced with "hello world". By having our action return `Ok("hello, world")`, we are asking Play to return an HTTP response with status code 200 (indicating that the request was successful) and the body "hello world".

Let's go back to the original content of `index`:

```
Action {  
    Ok(views.html.index("Your new application is ready."))  
}
```

We can see that this calls the method `views.html.index`. This might appear strange, because there is no `views` package anywhere. However, if you look at the `app/views` directory, you will notice two files: `index.scala.html` and `main.scala.html`. These are templates, which, at compile time, get transformed into Scala functions. Let's have a look at `main.scala.html`:

```
// app/views/main.scala.html  
@(title: String)(content: Html)  
  
<!DOCTYPE html>  
  
<html lang="en">  
  <head>  
    <title>@title</title>  
    <!-- not so important stuff -->  
  </head>  
  <body>  
    @content  
  </body>  
</html>
```

At compile time, this template is compiled to a function `main(title:String)` (`content:Html`) in the package `views.html`. Notice that the function package and name comes from the template file name, and the function arguments come from the first line of the template. The template contains embedded `@title` and `@content` values, which get filled in by the arguments to the function. Let's experiment with this in a Scala console:

```
$ activator console
scala> import views.html._
import views.html._

scala> val title = "hello"
title: String = hello

scala> val content = new play.twirl.api.Html("<b>World</b>")
content: play.twirl.api.Html = <b>World</b>

scala> main(title)(content)
res8: play.twirl.api.HtmlFormat.Appendable =
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>hello</title>
    <!-- not so important stuff -->
  </head>
  <body>
    <b>World</b>
  </body>
</html>
```

We can call `views.html.main`, just like we would call a normal Scala function. The arguments we pass in get embedded in the correct place, as defined by the template in `views/main.scala.html`.

This concludes our introductory tour of Play. Let's briefly go over what we have learnt: when a request reaches the Play server, the server reads the URL and the HTTP verb and checks that these exist in its `conf/routes` file. It will then pass the request to the `Action` defined by the controller for that route. This `Action` returns an HTTP response that gets fed back to the browser. In constructing the response, the `Action` may make use of a template, which, as far as it is concerned is just a function `(arguments list) => String` or `(arguments list) => HTML`.

Dynamic routing

Routing, as we saw, is the mapping of HTTP requests to Scala handlers. Routes are stored in `conf/routes`. A route is defined by an HTTP verb, followed by the end-point, followed by a Scala function:

```
// verb    // end-point          // Scala handler  
GET      /                   controllers.Application.index
```

We learnt to add new routes by just adding lines to the `routes` file. We are not limited to static routes, however. The Play framework lets us include wild cards in routes. The value of the wild card can be passed as an argument to the controller. To see how this works, let's create a controller that takes the name of a person as argument. In the `Application` object in `app.controllers`, add:

```
// app/controllers/Application.scala  
  
class Application extends Controller {  
  
  ...  
  
  def hello(name:String) = Action {  
    Ok(s"hello, $name")  
  }  
}
```

We can now define a route handled by this controller:

```
// conf/routes  
GET  /hello/:name           controllers.Application.hello(name)
```

If you now point your browser to `127.0.0.1:9000/hello/Jim`, you will see **hello, Jim** appear on the screen.

Any string between : and the following / is treated as a wild card: it will match any combination of characters. The value of the wild card can be passed to the controller. Note that the wild card can appear anywhere in the URL, and there can be more than one wild card. The following are all valid route definitions, for instance:

```
GET /hello/person-:name controllers.Application.hello(name)
// ... matches /hello/person-Jim

GET /hello/:name/picture controllers.Application.pictureFor(name)
// ... matches /hello/Jim/picture

GET /hello/:first:last controllers.Application.hello(first, last)
// ... matches /hello/john/doe
```

There are many other options for selecting routes and passing arguments to the controller. Consult the documentation for the Play framework for a full discussion on the routing possibilities: <https://www.playframework.com/documentation/2.4.x/ScalaRouting>.

URL design

It is generally considered best practice to leave the URL as simple as possible. The URL should reflect the hierarchical structure of the information of the website, rather than the underlying implementation. GitHub is a very good example of this: its URLs make intuitive sense. For instance, the URL for the repository for this book is:

<https://github.com/pbugnion/s4ds>

To access the issues page for that repository, add /issues to the route. To access the first issue, add /1 to that route. These are called **semantic URLs** (https://en.wikipedia.org/wiki/Semantic_URL).



Actions

We have talked about routes, and how to pass parameters to controllers. Let's now talk about what we can do with the controller.

The method defined in the route must return a `play.api.mvc.Action` instance. The Action type is a thin wrapper around the type `Request[A] => Result`, where `Request[A]` identifies an HTTP request and `Result` is an HTTP response.

Composing the response

An HTTP response, as we saw in *Chapter 7, Web APIs*, is composed of:

- the status code (such as 200 for a successful response, or 404 for a missing page)
- the response headers, a key-value list indicating metadata related to the response
- The response body. This can be HTML for web pages, or JSON, XML or plain text (or many other formats). This is generally the bit that we are really interested in.

The Play framework defines a `play.api.mvc.Result` object that symbolizes a response. The object contains a `header` attribute with the status code and the headers, and a `body` attribute containing the body.

The simplest way to generate a `Result` is to use one of the factory methods in `play.api.mvc.Results`. We have already seen the `Ok` method, which generates a response with status code 200:

```
def hello(name:String) = Action {  
    Ok("hello, $name")  
}
```

Let's take a step back and open a Scala console so we can understand how this works:

```
$ activator console  
scala> import play.api.mvc._  
import play.api.mvc._  
  
scala> val res = Results.Ok("hello, world")  
res: play.api.mvc.Result = Result(200, Map(Content-Type -> text/plain;  
charset=utf-8))  
  
scala> res.header.status  
Int = 200  
  
scala> res.header.headers  
Map[String,String] = Map(Content-Type -> text/plain; charset=utf-8)  
  
scala> res.body  
play.api.libs.iteratee.Enumerator[Array[Byte]] = play.api.libs.iteratee.  
Enumerator$$anon$18@5fb83873
```

We can see how the `Results.Ok(...)` creates a `Result` object with status 200 and (in this case), a single header denoting the content type. The body is a bit more complicated: it is an enumerator that can be pushed onto the output stream when needed. The enumerator contains the argument passed to `Ok: "hello, world"`, in this case.

There are many factory methods in `Results` for returning different status codes. Some of the more relevant ones are:

- `Action { Results.NotFound }`
- `Action { Results.BadRequest("bad request") }`
- `Action { Results.InternalServerError("error") }`
- `Action { Results.Forbidden }`
- `Action { Results.Redirect("/home") }`

For a full list of `Result` factories, consult the API documentation for `Results` (<https://www.playframework.com/documentation/2.4.x/api/scala/index.html#play.api.mvc.Results>).

We have, so far, been limiting ourselves to passing strings as the content of the `ok` result: `Ok("hello, world")`. We are not, however, limited to passing strings. We can pass a JSON object:

```
scala> import play.api.libs.json._  
import play.api.libs.json._  
  
scala> val jsonObj = Json.obj("hello" -> "world")  
jsonObj: play.api.libs.json.JsObject = {"hello":"world"}  
  
scala> Results.Ok(jsonObj)  
play.api.mvc.Result = Result(200, Map(Content-Type -> application/json;  
charset=utf-8))
```

We will cover interacting with JSON in more detail when we start building the API. We can also pass HTML as the content. This is most commonly the case when returning a view:

```
scala> val htmlObj = views.html.index("hello")  
htmlObj: play.twirl.api.HtmlFormat.Appendable =  
  
<!DOCTYPE html>
```

```
<html lang="en">
  <head>
    ...
  </head>
<body>
  <h1>Hello, <name>!</h1>
  <p>This is a simple <a href="#">HTTP request</a> example.</p>
  <pre>scala> Results.Ok(htmlObj)
play.api.mvc.Result = Result(200, Map(Content-Type -> text/html;
charset=utf-8))</pre>
```

Note how the Content-Type header is set based on the type of content passed to `Ok`. The `Ok` factory uses the `Writeable` type class to convert its argument to the body of the response. Thus, any content type for which a `Writeable` type class exists can be used as argument to `Ok`. If you are unfamiliar with type classes, you might want to read the *Looser coupling with type classes* section in *Chapter 5, Scala and SQL through JDBC*.

Understanding and parsing the request

We now know how to formulate (basic) responses. The other half of the equation is the HTTP request. Recall that an `Action` is just a function mapping `Request => Result`. We can access the request using:

```
def hello(name:String) = Action { request =>
  ...
}
```

One of the reasons for needing a reference to the request is to access parameters in the query string. Let's modify the `Hello, <name>` example that we wrote earlier to, optionally, include a title in the query string. Thus, a URL could be formatted as `/hello/Jim?title=Dr`. The `request` instance exposes the `getQueryString` method for accessing specific keys in the query string. This method returns `Some[String]` if the key is present in the query, or `None` otherwise. We can re-write our `hello` controller as:

```
def hello(name:String) = Action { request =>
  val title = request.getQueryString("title")
  val titleString = title.map { _ + " " }.getOrElse("")
  Ok(s"Hello, $titleString$name")
}
```

Try this out by accessing the URL `127.0.0.1:9000/hello/Odersky?title=Dr` in your browser. The browser should display `Hello, Dr Odersky`.

We have, so far, been concentrating on GET requests. These do not have a body. Other types of HTTP request, most commonly POST requests, do contain a body. Play lets the user pass *body parsers* when defining the action. The request body will be passed through the body parser, which will convert it from a byte stream to a Scala type. As a very simple example, let's define a new route that accepts POST requests:

```
POST      /hello          controllers.Application.helloPost
```

We will apply the predefined `parse.text` body parser to the incoming request body. This converts the body of the request to a string. The `helloPost` controller looks like:

```
def helloPost = Action(parse.text) { request =>
    Ok("Hello. You told me: " + request.body)
}
```

You cannot test POST requests easily in the browser. You can use cURL instead. cURL is a command line utility for dispatching HTTP requests. It is installed by default on Mac OS and should be available via the package manager on Linux distributions. The following will send a POST request with "I think that Scala is great" in the body:

```
$ curl --data "I think that Scala is great" --header
"Content-type:text/plain" 127.0.0.1:9000/hello
```

This prints the following line to the terminal:

```
Hello. You told me: I think that Scala is great
```



There are several types of built-in body parsers:

- `parse.file(new File("filename.txt"))` will save the body to a file.
- `parse.json` will parse the body as JSON (we will learn more about interacting with JSON in the next section).
- `parse.xml` will parse the body as XML.
- `parse.urlFormEncoded` will parse the body as returned by submitting an HTML form. The `request.body` attribute is a Scala map from `String` to `Seq[String]`, mapping each form element to its value(s).

For a full list of body parsers, the best source is the Scala API documentation for `play.api.mvc.BodyParsers.parse` available at: [https://www.playframework.com/documentation/2.5.x/api/scala/index.html#play.api.mvc.BodyParsers\\$parse\\$](https://www.playframework.com/documentation/2.5.x/api/scala/index.html#play.api.mvc.BodyParsers$parse$).

Interacting with JSON

JSON, as we discovered in previous chapters, is becoming the de-facto language for communicating structured data over HTTP. If you develop a web application or a web API, it is likely that you will have to consume or emit JSON, or both.

In *Chapter 7, Web APIs*, we learned how to parse JSON through `json4s`. The Play framework includes its own JSON parser and emitter. Fortunately, it behaves in much the same way as `json4s`.

Let's imagine that we are building an API that summarizes information about GitHub repositories. Our API will emit a JSON array listing a user's repositories when queried about a specific user (much like the GitHub API, but with just a subset of fields).

Let's start by defining a model for the repository. In Play applications, models are normally stored in the folder `app/models`, in the `models` package:

```
// app/models/Repo.scala

package models

case class Repo (
  val name:String,
  val language:String,
  val isFork: Boolean,
  val size: Long
)
```

Let's add a route to our application that serves arrays of repos for a particular user. In `conf/routes`, add the following line:

```
// conf/routes
GET  /api/repos/:username      controllers.Api.repos(username)
```

Let's now implement the framework for the controller. We will create a new controller for our API, imaginatively called `Api`. For now, we will just have the controller return dummy data. This is what the code looks like (we will explain the details shortly):

```
// app/controllers/Api.scala
package controllers
import play.api._
import play.api.mvc._
```

```
import play.api.libs.json._

import models.Repo

class Api extends Controller {

    // Some dummy data.
    val data = List[Repo](
        Repo("dotty", "Scala", true, 14315),
        Repo("frontend", "JavaScript", true, 392)
    )

    // Typeclass for converting Repo -> JSON
    implicit val writesRepos = new Writes[Repo] {
        def writes(repo:Repo) = Json.obj(
            "name" -> repo.name,
            "language" -> repo.language,
            "is_fork" -> repo.isFork,
            "size" -> repo.size
        )
    }

    // The controller
    def repos(username:String) = Action {

        val repoArray = Json.toJson(data)
        // toJson(data) relies on existence of
        // `Writes[List[Repo]]` type class in scope

        Ok(repoArray)
    }
}
```

If you point your web browser to `127.0.0.1:9000/api/repos/odersky`, you should now see the following JSON object:

```
[{"name": "dotty", "language": "Scala", "is_fork": true, "size": 14315}, {"name": "frontend", "language": "JavaScript", "is_fork": true, "size": 392}]
```

The only tricky part of this code is the conversion from `Repo` to JSON. We call `Json.toJson` on `data`, an instance of type `List[Repo]`. The `toJson` method relies on the existence of a type class `Writes[T]` for the type `T` passed to it.

The Play framework makes extensive use of type classes to define how to convert models to specific formats. Recall that we learnt how to write type classes in the context of SQL and MongoDB. The Play framework's expectations are very similar: for the `Json.toJson` method to work on an instance of type `Repo`, there must be a `Writes[Repo]` implementation available that specifies how to transform `Repo` objects to JSON.

In the Play framework, the `Writes[T]` type class defines a single method:

```
trait Writes[T] {  
    def writes(obj:T) :Json  
}
```

`Writes` methods for built-in simple types and for collections are already built into the Play framework, so we do not need to worry about defining `Writes[Boolean]`, for instance.

The `Writes[Repo]` instance is commonly defined either directly in the controller, if it is just used for that controller, or in the `Repo` companion object, where it can be used across several controllers. For simplicity, we just embed it in the controller.

Note how type-classes allow for separation of concerns. The model just defines the `Repo` type, without attaching any behavior. The `Writes[Repo]` type class just knows how to convert from a `Repo` instance to JSON, but knows nothing of the context in which it is used. Finally, the controller just knows how to create a JSON HTTP response.

Congratulations, you have just defined a web API that returns JSON! In the next section, we will learn how to fetch data from the GitHub web API to avoid constantly returning the same array.

Querying external APIs and consuming JSON

So far, we have learnt how to provide the user with a dummy JSON array of repositories in response to a request to `/api/repos/:username`. In this section, we will replace the dummy data with the user's actual repositories, downloaded from GitHub.

In *Chapter 7, Web APIs*, we learned how to query the GitHub API using Scala's `Source.fromURL` method and `scalaj-http`. It should come as no surprise that the Play framework implements its own library for interacting with external web services.

Let's edit the `Api` controller to fetch information about a user's repositories from GitHub, rather than using dummy data. When called with a username as argument, the controller will:

1. Send a GET request to the GitHub API for that user's repositories.
2. Interpret the response, converting the body from a JSON object to a `List[Repo]`.
3. Convert from the `List[Repo]` to a JSON array, forming the response.

We start by giving the full code listing before explaining the thornier parts in detail:

```
// app/controllers/Api.scala

package controllers

import play.api._
import play.api.mvc._
import play.api.libs.ws // query external APIs
import play.api.Play.current
import play.api.libs.json._ // parsing JSON
import play.api.libs.functional.syntax._
import play.api.libs.concurrent.Execution.Implicits.defaultContext

import models.Repo

class Api extends Controller {

    // type class for Repo -> Json conversion
    implicit val writesRepo = new Writes[Repo] {
        def writes(repo:Repo) = Json.obj(
            "name" -> repo.name,
            "language" -> repo.language,
            "is_fork" -> repo.isFork,
            "size" -> repo.size
        )
    }

    // type class for Github Json -> Repo conversion
    implicit val readsRepoFromGithub:Reads[Repo] = (
        JsPath \ "name").read[String] and
        (JsPath \ "language").read[String] and
        (JsPath \ "fork").read[Boolean] and
        (JsPath \ "size").read[Int]
    )
}
```

```
(JsPath \ "size").read[Long]
)(Repo.apply _)

// controller
def repos(username:String) = Action.async {

    // GitHub URL
    val url = s"https://api.github.com/users/$username/repos"
    val response = WS.url(url).get() // compose get request

    // "response" is a Future
    response.map { r =>
        // executed when the request completes
        if (r.status == 200) {

            // extract a list of repos from the response body
            val reposOpt = Json.parse(r.body).validate[List[Repo]]
            reposOpt match {
                // if the extraction was successful:
                case JsSuccess(repos, _) => Ok(Json.toJson(repos))

                // If there was an error during the extraction
                case _ => InternalServerError
            }
        }
        else {
            // GitHub returned something other than 200
            NotFound
        }
    }
}
```

}

If you have written all this, point your browser to, for instance, `127.0.0.1:9000/api/repos/odersky` to see the list of repositories owned by Martin Odersky:

```
[{"name": "dotty", "language": "Scala", "is_fork": true, "size": 14653}, {"name": "frontend", "language": "JavaScript", "is_fork": true, "size": 392}, ...]
```

This code sample is a lot to take in, so let's break it down.

Calling external web services

The first step in querying external APIs is to import the `ws` object, which defines factory methods for creating HTTP requests. These factory methods rely on a reference to an implicit Play application in the namespace. The easiest way to ensure this is the case is to import `play.api.Play.current`, a reference to the current application.

Let's ignore the `readsRepoFromGithub` type class for now and jump straight to the controller body. The URL that we want to hit with a GET request is `"https://api.github.com/users/$username/repos"`, with the appropriate value for `$username`. We create a GET request with `ws.url(url).get()`. We can also add headers to an existing request. For instance, to specify the content type, we could have written:

```
ws.url(url).withHeaders("Content-Type" ->  
    "application/json").get()
```

We can use headers to pass a GitHub OAuth token using:

```
val token = "2502761d..."  
ws.url(url).withHeaders("Authorization" -> s"token $token").get()
```

To formulate a POST request, rather than a GET request, replace the final `.get()` with `.post(data)`. Here, `data` can be JSON, XML or a string.

Adding `.get` or `.post` fires the request, returning a `Future[WSResponse]`. You should, by now, be familiar with futures. By writing `response.map { r => ... }`, we specify a transformation to be executed on the future result, when it returns. The transformation verifies the response's status, returning `NotFound` if the status code of the response is anything but 200.

Parsing JSON

If the status code is 200, the callback parses the response body to JSON and converts the parsed JSON to a `List[Repo]` instance. We already know how to convert from a `Repo` object to JSON using the `Writes[Repo]` type class. The converse, going from JSON to a `Repo` object, is a little more challenging, because we have to account for incorrectly formatted JSON. To this effect, the Play framework provides the `.validate[T]` method on JSON objects. This method tries to convert the JSON to an instance of type `T`, returning `JsSuccess` if the JSON is well-formatted, or `JsError` otherwise (similar to Scala's `Try` object). The `.validate` method relies on the existence of a type class `Reads[Repo]`. Let's experiment with a Scala console:

```
$ activator console  
  
scala> import play.api.libs.json._
```

```
import play.api.libs.json._

scala> val s = """
  { "name": "dotty", "size": 150, "language": "Scala", "fork": true }
"""

s: String = "
  { "name": "dotty", "size": 150, "language": "Scala", "fork": true }
"
"

scala> val parsedJson = Json.parse(s)
parsedJson: play.api.libs.json.JsValue = {"name":"dotty","size":150,"language":"Scala","fork":true}
```

Using `Json.parse` converts a string to an instance of `JsValue`, the super-type for JSON instances. We can access specific fields in `parsedJson` using XPath-like syntax (if you are not familiar with XPath-like syntax, you might want to read *Chapter 6, Slick – A Functional Interface for SQL*):

```
scala> parsedJson \ "name"
play.api.libs.json.JsLookupResult = JsDefined("dotty")
```

XPath-like lookups return an instance with type `JsLookupResult`. This takes two values: either `JsDefined`, if the path is valid, or `JsUndefined` if it is not:

```
scala> parsedJson \ "age"
play.api.libs.json.JsLookupResult = JsUndefined('age' is undefined on
object: {"name": "dotty", "size": 150, "language": "Scala", "fork": true})
```

To go from a `JsLookupResult` instance to a `String` in a type-safe way, we can use the `.validate[String]` method:

```
scala> (parsedJson \ "name").validate[String]
play.api.libs.json.JsResult[String] = JsSuccess(dotty,)
```

The `.validate[T]` method returns either `JsSuccess` if the `JsDefined` instance could be successfully cast to `T`, or `JsError` otherwise. To illustrate the latter, let's try validating this as an `Int`:

```
scala> (parsedJson \ "name").validate[Int]
play.api.libs.json.JsResult[Int] = JsError(List((List(ValidationError(List(error.expected.jsnumber),WrappedArray())))))
```

Calling `.validate` on an instance of type `JsUndefined` also returns in a `JsError`:

```
scala> (parsedJson \ "age").validate[Int]
play.api.libs.json.JsResult[Int] = JsError(List((),List(ValidationError
(List('age' is undefined on object: {"name":"dotty","size":150,
"language":"Scala","fork":true}),WrappedArray()))))
```

To convert from an instance of `JsResult[T]` to an instance of type `T`, we can use pattern matching:

```
scala> val name = (parsedJson \ "name").validate[String] match {
  case JsSuccess(n, _) => n
  case JsError(e) => throw new IllegalStateException(
    s"Error extracting name: $e")
}
name: String = dotty
```

We can now use `.validate` to cast JSON to simple types in a type-safe manner. But, in the code example, we used `.validate[Repo]`. This works provided a `Reads[Repo]` type class is implicitly available in the namespace.

The most common way of defining `Reads[T]` type classes is through a DSL provided in `import play.api.libs.functional.syntax._`. The DSL works by chaining operations returning either `JsSuccess` or `JsError` together. Discussing exactly how this DSL works is outside the scope of this chapter (see, for instance, the Play framework documentation page on JSON combinators: <https://www.playframework.com/documentation/2.4.x/ScalaJsonCombinators>). We will stick to discussing the syntax.

```
scala> import play.api.libs.functional.syntax._
import play.api.libs.functional.syntax._

scala> import models.Repo
import models.Repo

scala> implicit val readsRepoFromGithub:Reads[Repo] = (
  (JsPath \ "name").read[String] and
  (JsPath \ "language").read[String] and
  (JsPath \ "fork").read[Boolean] and
```

```
(JsPath \ "size").read[Long]
) (Repo.apply _)
readsRepoFromGithub: play.api.libs.json.Reads[models.Repo] = play.api.
libs.json.Reads$$anon$8@a198ddb
```

The Reads type class is defined in two stages. The first chains together `read[T]` methods with `and`, combining successes and errors. The second uses the `apply` method of the companion object of a case class (or `Tuple` instance) to construct the object, provided the first stage completed successfully. Now that we have defined the type class, we can call `validate[Repo]` on a `JsValue` object:

```
scala> val repoOpt = parsedJson.validate[Repo]
play.api.libs.json.JsResult[models.Repo] = JsSuccess(Repo(dotty,Scala,true,150),)
```

We can then use pattern matching to extract the `Repo` object from the `JsSuccess` instance:

```
scala> val JsSuccess(repo, _) = repoOpt
repo: models.Repo = Repo(dotty,Scala,true,150)
```

We have, so far, only talked about validating single repos. The Play framework defines type classes for collection types, so, provided `Reads[Repo]` is defined, `Reads[List[Repo]]` will also be defined.

Now that we understand how to extract Scala objects from JSON, let's get back to the code. If we manage to successfully convert the repositories to a `List[Repo]`, we emit it again as JSON. Of course, converting from GitHub's JSON representation of a repository to a Scala object, and from that Scala object directly to our JSON representation of the object, might seem convoluted. However, if this were a real application, we would have additional logic. We could, for instance, store repos in a cache, and try and fetch from that cache instead of querying the GitHub API. Converting from JSON to Scala objects as early as possible decouples the code that we write from the way GitHub returns repositories.

Asynchronous actions

The last bit of the code sample that is new is the call to `Action.async`, rather than just `Action`. Recall that an `Action` instance is a thin wrapper around a `Request => Result` method. Our code, however, returns a `Future[Result]`, rather than a `Result`. When that is the case, use the `Action.async` to construct the action, rather than `Action` directly. Using `Action.async` tells the Play framework that the code creating the `Action` is asynchronous.

Creating APIs with Play: a summary

In the last section, we deployed an API that responds to GET requests. Since this is a lot to take in, let's summarize how to go about API creation:

1. Define appropriate routes in `/conf/routes`, using wildcards in the URL as needed.
2. Create Scala case classes in `/app/models` to represent the models used by the API.
3. Create `Write[T]` methods to write models to JSON or XML so that they can be returned by the API.
4. Bind the routes to controllers. If the controllers need to do more than a trivial amount of work, wrap the work in a future to avoid blocking the server.

There are many more useful components of the Play framework that you are likely to need, such as, for instance, how to use Slick to access SQL databases. We do not, unfortunately, have time to cover these in this introduction. The Play framework has extensive, well-written documentation that will fill the gaping holes in this tutorial.

Rest APIs: best practice

As the Internet matures, REST (representational state transfer) APIs are emerging as the most reliable design pattern for web APIs. An API is described as *RESTful* if it follows these guiding principles:

- The API is designed as a set of resources. For instance, the GitHub API provides information about users, repositories, followers, etc. Each user, or repository, is a specific resource. Each resource can be addressed through a different HTTP end-point.
- The URLs should be simple and should identify the resource clearly. For instance, `api.github.com/users/odersky` is simple and tells us clearly that we should expect information about the user Martin Odersky.
- There is no *world resource* that contains all the information about the system. Instead, top-level resources contain links to more specialized resources. For instance, the user resource in the GitHub API contains links to that user's repositories and that user's followers, rather than having all that information embedded in the user resource directly.

- The API should be discoverable. The response to a request for a specific resource should contain URLs for related resources. When you query the user resource on GitHub, the response contains the URL for accessing that user's followers, repositories etc. The client should use the URLs provided by the API, rather than attempting to construct them client-side. This makes the client less brittle to changes in the API.
- There should be as little state maintained on the server as possible. For instance, when querying the GitHub API, we must pass the authentication token with every request, rather than expecting our authentication status to be *remembered* on the server. Having each interaction be independent of the history provides much better scalability: if any interaction can be handled by any server, load balancing is much easier.

Summary

In this chapter, we introduced the Play framework as a tool for building web APIs. We built an API that returns a JSON array of a user's GitHub repositories. In the next chapter, we will build on this API and construct a single-page application to represent this data graphically.

References

- This Wikipedia page gives information on semantic URLs:
https://en.wikipedia.org/wiki/Semantic_URL and
<http://apiux.com/2013/04/03/url-design-restful-web-services/>.
- For a much more in depth discussion of the Play framework, I suggest *Play Framework Essentials* by Julien Richard-Foy.
- *REST in Practice: Hypermedia and Systems Architecture*, by Jim Webber, Savas Parastatidis and Ian Robinson describes how to architect REST APIs.

14

Visualization with D3 and the Play Framework

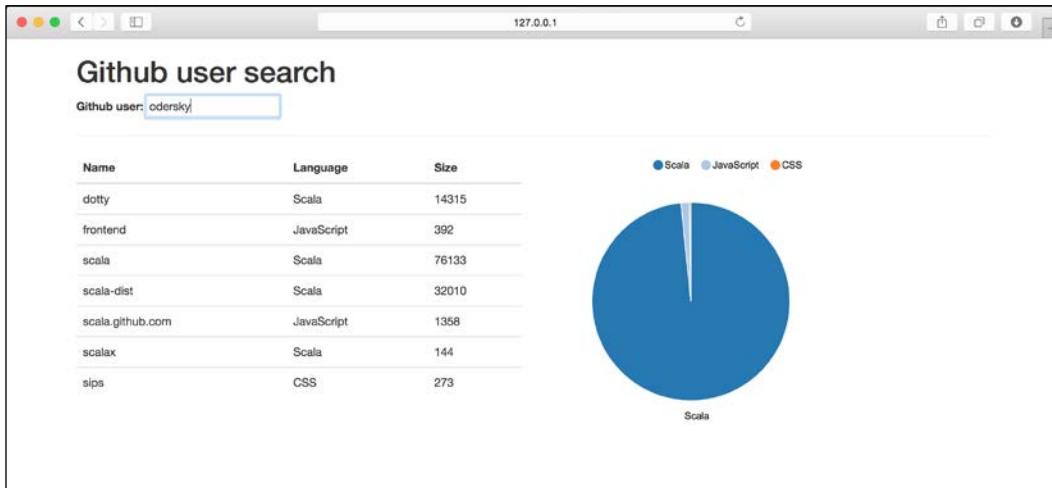
In the previous chapter, we learned about the Play framework, a web framework for Scala. We built an API that returns a JSON array describing a user's GitHub repositories.

In this chapter, we will construct a fully-fledged web application that displays a table and a chart describing a user's repositories. We will learn to integrate **D3.js**, a JavaScript library for building data-driven web pages, with the Play framework. This will set you on the path to building compelling interactive visualizations that showcase results obtained with machine learning.

This chapter assumes that you are familiar with HTML, CSS, and JavaScript. We present references at the end of the chapter. You should also have read the previous chapter.

GitHub user data

We will build a single-page application that uses, as its backend, the API developed in the previous chapter. The application contains a form where the user enters the login name for a GitHub account. The application queries the API to get a list of repositories for that user and displays them on the screen as both a table and a pie chart summarizing programming language use for that user:



To see a live version of the application, head over to
<http://app.scala4datascience.com>.

Do I need a backend?

In the previous chapter, we learned about the client-server model that underpins how the internet works: when you enter a website URL in your browser, the server serves HTML, CSS, and JavaScript to your browser, which then renders it in the appropriate manner.

What does this all mean for you? Arguably the second question that you should be asking yourself when building a web application is whether you need to do any server-side processing (right after "is this really going to be worth the effort?"). Could you just create an HTML web page with some JavaScript?

You can get away without a backend if the data needed to build the whole application is small enough: typically a few megabytes. If your application is larger, you will need a backend to transfer just the data the client currently needs. Surprisingly, you can often build visualizations without a backend: while data science is accustomed to dealing with terabytes of data, the goal of the data science process is often condensing these huge data sets to a few meaningful numbers.

Having a backend also lets you include logic invisible to the client. If you need to validate a password, you clearly cannot send the code to do that to the client computer: it needs to happen out of sight, on the server.

If your application is small enough and you do not need to do any server-side processing, stop reading this chapter, brush up on your JavaScript if you have to, and forget about Scala for now. Not having to worry about building a backend will make your life easier.

Clearly, however, we do not have that freedom for the application that we want to build: the user could enter the name of anyone on GitHub. Finding information about that user requires a backend with access to tremendous storage and querying capacity (which we simulate by just forwarding the request to the GitHub API and re-interpreting the response).

JavaScript dependencies through web-jars

One of the challenges of developing web applications is that we are writing two quasi-separate programs: the server-side program and the client-side program. These generally require different technologies. In particular, for any but the most trivial application, we must keep track of JavaScript libraries, and integrate processing the JavaScript code (for instance, for minification) in the build process.

The Play framework manages JavaScript dependencies through *web-jars*. These are just JavaScript libraries packaged as jars. They are deployed on Maven Central, which means that we can just add them as dependencies to our `build.sbt` file. For this application, we will need the following JavaScript libraries:

- Require.js, a library for writing modular JavaScript
- JQuery
- Bootstrap
- Underscore.js, a library that adds many functional constructs and client-side templating.
- D3, the graph plotting library
- NVD3, a graph library built on top of D3

If you are planning on coding up the examples provided in this chapter, the easiest will be for you to start from the code for the previous chapter (You can download the code for *Chapter 13, Web APIs with Play*, from GitHub: <https://github.com/pbugnion/s4ds/tree/master/chap13>). We will assume this as a starting point here onwards.

Let's include the dependencies on the web-jars in the `build.sbt` file:

```
libraryDependencies ++= Seq(  
    "org.webjars" % "requirejs" % "2.1.22",  
    "org.webjars" % "jquery" % "2.1.4",  
    "org.webjars" % "underscorejs" % "1.8.3",  
    "org.webjars" % "nvd3" % "1.8.1",  
    "org.webjars" % "d3js" % "3.5.6",  
    "org.webjars" % "bootstrap" % "3.3.6"  
)
```

Fetch the modules by running `activator update`. Once you have done this, you will notice the JavaScript libraries in `target/web/public/main/lib`.

Towards a web application: HTML templates

In the previous chapter, we briefly saw how to construct HTML templates by interleaving Scala snippets in an HTML file. We saw that templates are compiled to Scala functions, and we learned how to call these functions from the controllers.

In single-page applications, the majority of the logic governing what is actually displayed in the browser resides in the client-side JavaScript, not in the server. The pages served by the server contain the bare-bones HTML framework.

Let's create the HTML layout for our application. We will save this in `views/index.scala.html`. The template will just contain the layout for the application, but will not contain any information about any user's repositories. To fetch that information, the application will have to query the API developed in the previous chapter. The template does not take any parameters, since all the dynamic HTML generation will happen client-side.

We use the Bootstrap grid layout to control the HTML layout. If you are not familiar with Bootstrap layouts, consult the documentation at <http://getbootstrap.com/css/#grid-example-basic>.

```
// app/views/index.scala.html
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>Github User display</title>
    <link rel="stylesheet" media="screen"
          href="@routes.Assets.versioned("stylesheets/main.css")">
    <link rel="shortcut icon" type="image/png"
          href="@routes.Assets.versioned("images/favicon.png")">
    <link rel="stylesheet" media="screen"
          href=@routes.Assets.versioned("lib/nvd3/nv.d3.css") >
    <link rel="stylesheet" media="screen"
          href=@routes.Assets.versioned(
            "lib/bootstrap/css/bootstrap.css")>
  </head>

  <body>
    <div class="container">

      <!-- Title row -->
      <div class="row">
        <h1>Github user search</h1>
      </div>

      <!-- User search row -->
      <div class="row">
        <label>Github user: </label>
        <input type="text" id="user-selection">
      </div>
    </div>
  </body>

```

```
<span id="searching-span"></span>
<hr />
</div>

<!-- Results row -->
<div id="response" class="row"></div>
</div>
</body>
</html>
```

In the HTML head, we link the CSS stylesheets that we need for the application. Instead of specifying the path explicitly, we use the `@routes.Assets.versioned(...)` function. This resolves to a URI corresponding to the location where the assets are stored post-compilation. The argument passed to the function should be the path from `target/web/public/main` to the asset you need.

We want to serve the compiled version of this view when the user accesses the route `/` on our server. We therefore need to add this route to `conf/routes`:

```
# conf/routes
GET   /      controllers.Application.index
```

The route is served by the `index` function in the `Application` controller. All this controller needs to do is serve the `index` view:

```
// app/controllers/Application.scala
package controllers

import play.api._
import play.api.mvc._

class Application extends Controller {

    def index = Action {
        Ok(views.html.index())
    }
}
```

Start the Play framework by running `activator run` in the root directory of the application and point your web browser to `127.0.0.1:9000/`. You should see the framework for our web application. Of course, the application does not do anything yet, since we have not written any of the JavaScript logic yet.



Modular JavaScript through RequireJS

The simplest way of injecting JavaScript libraries into the namespace is to add them to the HTML framework via `<script>...</script>` tags in the HTML header. For instance, to add JQuery, we would add the following line to the head of the document:

```
<script src=@routes.Assets.versioned("lib/jquery/jquery.js")
       type="text/javascript"></script>
```

While this works, it does not scale well to large applications, since every library gets imported into the global namespace. Modern client-side JavaScript frameworks such as AngularJS provide an alternative way of defining and loading modules that preserve encapsulation.

We will use RequireJS. In a nutshell, RequireJS lets us encapsulate JavaScript modules through functions. For instance, if we wanted to write a module example that contains a function for hiding a div, we would define the module as follows:

```
// example.js
define(["jquery", "underscore"], function($, _) {
    // hide a div
    function hide(div_name) {
        $(div_name).hide();
    }

    // what the module exports.
    return { "hide": hide }
}) ;
```

We encapsulate our module as a callback in a function called `define`. The `define` function takes two arguments: a list of dependencies, and a function definition. The `define` function binds the dependencies to the arguments list of the callback: in this case, functions in JQuery will be bound to `$` and functions in Underscore will be bound to `_`. This creates a module which exposes whatever the callback function returns. In this case, we export the `hide` function, binding it to the name "`hide`". Our example module thus exposes the `hide` function.

To load this module, we pass it as a dependency to the module in which we want to use it:

```
define(["example"], function(example) {
    function hide_all() {
```

```
example.hide("#top") ;
example.hide("#bottom") ;
}

return { "hide_all": hide_all } ;
});
```

Notice how the functions in `example` are encapsulated, rather than existing in the global namespace. We call them through `example.<function-name>`. Furthermore, any functions or variables defined internally to the `example` module remain private.

Sometimes, we want JavaScript code to exist outside of modules. This is often the case for the script that bootstraps the application. For these, replace `define` with `require`:

```
require(["jquery", "example"], function($, example) {
  $(document).ready(function() {
    example.hide("#header") ;
  }) ;
}) ;
```

Now that we have an overview of RequireJS, how do we use it in the Play framework? The first step is to add the dependency on the RequireJS web jar, which we have done. The Play framework also adds a RequireJS SBT plugin (<https://github.com/sbt/sbt-rjs>), which should be installed by default if you used the `play-scala` activator. If this is missing, it can be added with the following line in `plugins.sbt`:

```
// project/plugins.sbt

addSbtPlugin("com.typesafe.sbt" % "sbt-rjs" % "1.0.7")
```

We also need to add the plugin to the list of stages. This allows the plugin to manipulate the JavaScript assets when packaging the application as a jar. Add the following line to `build.sbt`:

```
pipelineStages := Seq(rjs)
```

You will need to restart the activator for the changes to take effect.

We are now ready to use RequireJS in our application. We can use it by adding the following line in the head section of our view:

```
// index.scala.html
```

```
<html>
<head>
```

```
...
<script
  type="text/javascript"
  src=@routes.Assets.versioned("lib/requirejs/require.js").url
  data-main=@routes.Assets.versioned("javascripts/main.js").url>
</script>

</head>
...
</html>
```

When the view is compiled, this is resolved to tags like:

```
<script type="text/javascript"
  data-main="/assets/javascripts/main.js"
  src="/assets/lib/requirejs/require.min.js">
</script>
```

The argument passed to `data-main` is the entry point for our application. When RequireJS loads, it will execute `main.js`. That script must therefore bootstrap our application. In particular, it should contain a configuration object for RequireJS, to make it aware of where all the libraries are.

Bootstrapping the applications

When we linked `require.js` to our application, we told it to use `main.js` as our entry point. To test that this works, let's start by entering a dummy `main.js`. JavaScript files in Play applications go in `/public/javascripts`:

```
// public/javascripts/main.js

require([], function() {
  console.log("hello, JavaScript");
});
```

To verify that this worked, head to `127.0.0.1:9000` and open the browser console. You should see "hello, JavaScript" in the console.

Let's now write a more useful `main.js`. We will start by configuring RequireJS, giving it the location of modules we will use in our application. Unfortunately, NVD3, the graph library that we use, does not play very well with RequireJS so we have to use an ugly hack to make it work. This complicates our `main.js` file somewhat:

```
// public/javascripts/main.js

(function (requirejs) {
    'use strict';

    // -- RequireJS config --
    requirejs.config({
        // path to the web jars. These definitions allow us
        // to use "jquery", rather than "../lib/jquery/jquery",
        // when defining module dependencies.
        paths: {
            "jquery": "../lib/jquery/jquery",
            "underscore": "../lib/underscorejs/underscore",
            "d3": "../lib/d3js/d3",
            "nvd3": "../lib/nvd3/nv.d3",
            "bootstrap": "../lib/bootstrap/js/bootstrap"
        },
        shim: {
            // hack to get nvd3 to work with requirejs.
            // see this so question:
            // http://stackoverflow.com/questions/13157704/how-to-integrate-
            d3-with-require-js#comment32647365_13171592
            nvd3: {
                deps: ["d3.global"],
                exports: "nv"
            },
            bootstrap : { deps :['jquery'] }
        }
    });

    // hack to get nvd3 to work with requirejs.
    // see this so question on Stack Overflow:
}) (requirejs) ;
```

```
// http://stackoverflow.com/questions/13157704/how-to-integrate-d3-
with-require-js#comment32647365_13171592
define("d3.global", ["d3"], function(d3global) {
    d3 = d3global;
});

require([], function() {
    // Our application
    console.log("hello, JavaScript");
}) ;
```

Now that we have the configuration in place, we can dig into the JavaScript part of the application.

Client-side program architecture

The basic idea is simple: the user searches for the name of someone on GitHub in the input box. When he enters a name, we fire a request to the API designed earlier in this chapter. When the response from the API returns, the program binds that response to a model and emits an event notifying that the model has been changed. The views listen for this event and refresh from the model in response.

Designing the model

Let's start by defining the client-side model. The model holds information regarding the repos of the user currently displayed. It gets filled in after the first search.

```
// public/javascripts/model.js

define([], function(){
    return {
        ghubUser: "", // last name that was searched for
        exists: true, // does that person exist on github?
        repos: [] // list of repos
    } ;
});
```

To see a populated value of the model, head to the complete application example on `app.scala4datascience.com`, open a JavaScript console in your browser, search for a user (for example, `odersky`) in the application and type the following in the console:

```
> require(["model"], function(model) { console.log(model) ; })
{ghubUser: "odersky", exists: true, repos: Array}

> require(["model"], function(model) {
  console.log(model.repos[0]);
})
{name: "dotty", language: "Scala", is_fork: true, size: 14653}
```

These import the `"model"` module, bind it to the variable `model`, and then print information to the console.

The event bus

We need a mechanism for informing the views when the model is updated, since the views need to refresh from the new model. This is commonly handled through *events* in web applications. JQuery lets us bind callbacks to specific events. The callback is executed when that event occurs.

For instance, to bind a callback to the event `"custom-event"`, enter the following in a JavaScript console:

```
> $(window).on("custom-event", function() {
  console.log("custom event received") ;
});
```

We can fire the event using:

```
> $(window).trigger("custom-event");
custom event received
```

Events in JQuery require an *event bus*, a DOM element on which the event is registered. In this case, we used the window DOM element as our event bus, but any JQuery element would have served. Centralizing event definitions to a single module is helpful. We will, therefore, create an `events` module containing two functions: `trigger`, which triggers an event (specified by a string) and `on`, which binds a callback to a specific event:

```
// public/javascripts/events.js

define(["jquery"], function($) {

    var bus = $(window); // widget to use as an event bus

    function trigger(eventType) {
        $(bus).trigger(eventType);
    }

    function on(eventType, f) {
        $(bus).on(eventType, f);
    }

    return {
        "trigger": trigger,
        "on": on
    };
});
```

We can now emit and receive events using the `events` module. You can test this out in a JavaScript console on the live version of the application (at `app.scala4datascience.com`). Let's start by registering a listener:

```
> require(["events"], function(events) {
    // register event listener
    events.on("hello_event", function() {
        console.log("Received event");
    });
});
```

If we now trigger the event "hello_event", the listener prints "Received event":

```
> require(["events"], function(events) {
    // trigger the event
    events.trigger("hello_event");
});
```

Using events allows us to decouple the controller from the views. The controller does not need to know anything about the views, and vice-versa. The controller just needs to emit a "model_updated" event when the model is updated, and the views need to refresh from the model when they receive that event.

AJAX calls through JQuery

We can now write the controller for our application. When the user enters a name in the text input, we query the API, update the model and trigger a `model_updated` event.

We use JQuery's `$.getJSON` function to query our API. This function takes a URL as its first argument, and a callback as its second argument. The API call is asynchronous: `$.getJSON` returns immediately after execution. All request processing must, therefore, be done in the callback. The callback is called if the request is successful, but we can define additional handlers that are always called, or called on failure. Let's try this out in the browser console (either your own, if you are running the API developed in the previous chapter, or on `app.scala4datascience.com`). Recall that the API is listening to the end-point `/api/repos/:user`:

```
> $.getJSON("/api/repos/odersky", function(data) {  
  console.log("API response:") ;  
  console.log(data);  
  console.log(data[0]);  
}) ;  
{readyState: 1, getResponseHeader: function, ...}  
  
API response:  
[Object, Object, Object, Object, Object, ...]  
{name: "dotty", language: "Scala", is_fork: true, size: 14653}
```

`getJSON` returns immediately. A few tenths of a second later, the API responds, at which point the response gets fed through the callback.

The callback only gets executed on success. It takes, as its argument, the JSON object returned by the API. To bind a callback that is executed when the API request fails, call the `.fail` method on the return value of `getJSON`:

```
> $.getJSON("/api/repos/junk123456", function(data) {  
  console.log("called on success");  
}).fail(function() {  
  console.log("called on failure") ;
```

```
) ;  
{readyState: 1, getResponseHeader: function, ...}
```

called on failure

We can also use the `.always` method on the return value of `getJSON` to specify a callback that is executed, whether the API query was successful or not.

Now that we know how to use `$.getJSON` to query our API, we can write the controller. The controller listens for changes to the `#user-selection` input field. When a change occurs, it fires an AJAX request to the API for information on that user. It binds a callback which updates the model when the API replies with a list of repositories. We will define a controller module that exports a single function, `initialize`, that creates the event listeners:

```
// public/javascripts/controller.js  
define(["jquery", "events", "model"], function($, events, model) {  
  
    function initialize() {  
        $("#user-selection").change(function() {  
  
            var user = $("#user-selection").val() ;  
            console.log("Fetching information for " + user) ;  
  
            // Change cursor to a 'wait' symbol  
            // while we wait for the API to respond  
            $("*").css({"cursor": "wait"}) ;  
  
            $.getJSON("/api/repos/" + user, function(data) {  
                // Executed on success  
                model.exists = true ;  
                model.repos = data ;  
            }).fail(function() {  
                // Executed on failure  
                model.exists = false ;  
                model.repos = [] ;  
            }).always(function() {  
                // Always executed  
                model.ghubUser = user ;  
  
                // Restore cursor  
                $("*").css({"cursor": "initial"}) ;  
  
                // Tell the rest of the application
```

```
// that the model has been updated.  
events.trigger("model_updated") ;  
});  
});  
};  
  
return { "initialize": initialize };  
  
});
```

Our controller module just exposes the `initialize` method. Once the initialization is performed, the controller interacts with the rest of the application through event listeners. We will call the controller's `initialize` method in `main.js`. Currently, the last lines of that file are just an empty `require` block. Let's import our controller and initialize it:

```
// public/javascripts/main.js  
  
require(["controller"], function(controller) {  
    controller.initialize();  
});
```

To test that this works, we can bind a dummy listener to the `"model_updated"` event. For instance, we could log the current model to the browser JavaScript console with the following snippet (which you can write directly in the JavaScript console):

```
> require(["events", "model"],  
function(events, model) {  
    events.on("model_updated", function () {  
        console.log("model_updated event received");  
        console.log(model);  
    });  
});
```

If you then search for a user, the model will be printed to the console. We now have the controller in place. The last step is writing the views.

Response views

If the request fails, we just display **Not found** in the response div. This part is the easiest to code up, so let's do that first. We define an `initialize` method that generates the view. The view then listens for the "model_updated" event, which is fired by the controller after it updates the model. Once the initialization is complete, the only way to interact with the response view is through "model_updated" events:

```
// public/javascripts/responseView.js

define(["jquery", "model", "events"],
function($, model, events) {

    var failedResponseHtml =
        "<div class='col-md-12'>Not found</div>" ;

    function initialize() {
        events.on("model_updated", function() {
            if (model.exists) {
                // success - we will fill this in later.
                console.log("model exists")
            }
            else {
                // failure - the user entered
                // is not a valid GitHub login
                $("#response").html(failedResponseHtml) ;
            }
        }) ;
    }

    return { "initialize": initialize } ;

}) ;
```

To bootstrap the view, we must call the `initialize` function from `main.js`. Just add a dependency on `responseView` in the require block, and call `responseView.initialize()`. With these modifications, the final require block in `main.js` is:

```
// public/javascripts/main.js

require(["controller", "responseView"],
function(controller, responseView) {
    controller.initialize();
    responseView.initialize() ;
}) ;
```

You can check that this all works by entering junk in the user input to deliberately cause the API request to fail.

When the user enters a valid GitHub login name and the API returns a list of repos, we must display those on the screen. We display a table and a pie chart that aggregates the repository sizes by language. We will define the pie chart and the table in two separate modules, called `repoGraph.js` and `repoTable.js`. Let's assume those exist for now and that they expose a `build` method that accepts a model and the name of a `div` in which to appear.

Let's update the code for `responseView` to accommodate the user entering a valid GitHub user name:

```
// public/javascripts/responseView.js

define(["jquery", "model", "events", "repoTable", "repoGraph"] ,
function($, model, events, repoTable, repoGraph) {

    // HTMHTML to inject when the model represents a valid user
    var successfulResponseHtml =
        "<div class='col-md-6' id='response-table'></div>" +
        "<div class='col-md-6' id='response-graph'></div>" ;

    // HTML to inject when the model is for a non-existent user
    var failedResponseHtml =
        "<div class='col-md-12'>Not found</div>" ;

    function initialize() {
        events.on("model_updated", function() {
            if (model.exists) {
                $("#response").html(successfulResponseHtml) ;
                repoTable.build(model, "#response-table") ;
                repoGraph.build(model, "#response-graph") ;
            }
            else {
                $("#response").html(failedResponseHtml) ;
            }
        }) ;
    }

    return { "initialize": initialize } ;

}) ;
```

Let's walk through what happens in the event of a successful API call. We inject the following bit of HTML in the #response div:

```
var successfulResponseHtml =
  "<div class='col-md-6' id='response-table'></div>" +
  "<div class='col-md-6' id='response-graph'></div>" ;
```

This adds two HTML divs, one for the table of repositories, and the other for the graph. We use Bootstrap classes to split the response div vertically.

Let's now turn our attention to the table view, which needs to expose a single `build` method, as described in the previous section. We will just display the repositories in an HTML table. We will use *Underscore templates* to build the table dynamically. Underscore templates work much like string interpolation in Scala: we define a template with placeholders. Let's try this in a browser console:

```
> require(["underscore"], function(_) {
  var myTemplate = _.template(
    "Hello, <%= title %> <%= name %>!"
  ) ;
});
```

This creates a `myTemplate` function which accepts an object with attributes `title` and `name`:

```
> require(["underscore"], function(_) {
  var myTemplate = _.template( ... );
  var person = { title: "Dr.", name: "Odersky" } ;
  console.log(myTemplate(person)) ;
});
```

Underscore templates thus provide a convenient mechanism for formatting an object as a string. We will create a template for each row in our table, and pass the model for each repository to the template:

```
// public/javascripts/repoTable.js

define(["underscore", "jquery"], function(_, $) {

  // Underscore template for each row
  var rowTemplate = _.template("<tr>" +
    "<td><%= name %></td>" +
    "<td><%= language %></td>" +
    "<td><%= size %></td>" +
    "</tr>") ;

  // template for the table
```

```
var repoTable = _.template(
  "<table id='repo-table' class='table'>" +
  "  <thead>" +
  "    <tr>" +
  "      <th>Name</th><th>Language</th><th>Size</th>" +
  "    </tr>" +
  "  </thead>" +
  "  <tbody>" +
  "    <%= tbody %>" +
  "  </tbody>" +
  "</table>" ;

// Builds a table for a model
function build(model, divName) {
  var tbody = "" ;
  _.each(model.repos, function(repo) {
    tbody += rowTemplate(repo) ;
 }) ;
  var table = repoTable({tbody: tbody}) ;
  $(divName).html(table) ;
}

return { "build": build } ;
}) ;
```

Drawing plots with NVD3

D3 is a library that offers low-level components for building interactive visualizations in JavaScript. By offering the low-level components, it gives a huge degree of flexibility to the developer. The learning curve can, however, be quite steep. In this example, we will use NVD3, a library which provides pre-made graphs for D3. This can greatly speed up initial development. We will place the code in the file `repoGraph.js` and expose a single method, `build`, which takes, as arguments, a model and a div and draws a pie chart in that div. The pie chart will aggregate language use across all the user's repositories.

The code for generating a pie chart is nearly identical to the example given in the NVD3 documentation, available at <http://nvd3.org/examples/pie.html>. The data passed to the graph must be available as an array of objects. Each object must contain a `label` field and a `size` field. The `label` field identifies the language, and the `size` field is the total size of all the repositories for that user written in that language. The following would be a valid data array:

```
[  
  { label: "Scala", size: 1234 },  
  { label: "Python", size: 4567 }  
]
```

To get the data in this format, we must aggregate sizes across the repositories written in a particular language in our model. We write the `generateDataFromModel` function to transform the `repos` array in the model to an array suitable for NVD3. The crux of the aggregation is performed by a call to Underscore's `groupBy` method, to group repositories by language. This method works exactly like Scala's `groupBy` method. With this in mind, the `generateDataFromModel` function is:

```
// public/javascripts/repoGraph.js  
  
define(["underscore", "d3", "nvd3"],  
function(_, d3, nv) {  
  
  // Aggregate the repo size by language.  
  // Returns an array of objects like:  
  // [ { label: "Scala", size: 1245},  
  //   { label: "Python", size: 432 } ]  
  function generateDataFromModel(model) {  
  
    // Build an initial object mapping each  
    // language to the repositories written in it  
    var language2Repos = _.groupBy(model.repos,  
      function(repo) { return repo.language ; }) ;  
  
    // Map each { "language": [ list of repos ], ...}  
    // pairs to a single document { "language": totalSize }  
    // where totalSize is the sum of the individual repos.  
    var plotObjects = _.map(language2Repos,  
      function(repos, language) {  
        var sizes = _.map(repos, function(repo) {  
          return repo.size;  
        });  
        // Sum over the sizes using 'reduce'  
        var totalSize = _.reduce(sizes,
```

```
        function(memo, size) { return memo + size; },
    0) ;
    return { label: language, size: totalsize } ;
}) ;

return plotObjects;
}
```

We can now build the pie chart, using NVD3's addGraph method:

```
// Build the chart.
function build(model, divName) {
    var transformedModel = generateDataFromModel(model) ;
    nv.addGraph(function() {

        var height = 350;
        var width = 350;

        var chart = nv.models.pieChart()
            .x(function (d) { return d.label ; })
            .y(function (d) { return d.size ;})
            .width(width)
            .height(height) ;

        d3.select(divName).append("svg")
            .datum(transformedModel)
            .transition()
            .duration(350)
            .attr('width', width)
            .attr('height', height)
            .call(chart) ;

        return chart ;
    });
}

return { "build" : build } ;
});
```

This was the last component of our application. Point your browser to 127.0.0.1:9000 and you should see the application running.

Congratulations! We have built a fully-functioning single-page web application.

Summary

In this chapter, we learned how to write a fully-featured web application with the Play framework. Congratulations on making it this far. Building web applications are likely to push many data scientists beyond their comfort zone, but knowing enough about the web to build basic applications will allow you to share your results in a compelling, engaging manner, as well as facilitate communications with software engineers and web developers.

This concludes our whistle stop tour of Scala libraries. Over the course of this book, we have learned how to tackle linear algebra and optimization problems efficiently using Breeze, how to insert and query data in SQL databases in a functional manner, and both how to interact with web APIs and how to create them. We have reviewed some of tools available to the data scientist for writing concurrent or parallel applications, from parallel collections and futures to Spark via Akka. We have seen how pervasive these constructs are in Scala libraries, from futures in the Play framework to Akka as the backbone of Spark. If you have read this far, pat yourself on the back.

This books gives you the briefest of introduction to the libraries it covers, hopefully just enough to give you a taste of what each tool is good for, what you could accomplish with it, and how it fits in the wider Scala ecosystem. If you decide to use any of these in your data science pipeline, you will need to read the documentation in more detail, or a more complete reference book. The references listed at the end of each chapter should provide a good starting point.

Both Scala and data science are evolving rapidly. Do not stay wedded to a particular toolkit or concept. Remain on top of current developments and, above all, remain pragmatic: find the right tool for the right job. Scala and the libraries discussed here will often be that tool, but not always: sometimes, a shell command or a short Python script will be more effective. Remember also that programming skills are but one aspect of the data scientist's body of knowledge. Even if you want to specialize in the engineering side of data science, learn about the problem domain and the mathematical underpinnings of machine learning.

Most importantly, if you have taken the time to read this book, it is likely that you view programming and data science as more than a day job. Coding in Scala can be satisfying and rewarding, so have fun and be awesome!

References

There are thousands of HTML and CSS tutorials dotted around the web. A simple Google search will give you a much better idea of the resources available than any list of references I can provide.

Mike Bostock's website has a wealth of beautiful D3 visualizations: <http://bostocks.org/mike/>. To understand a bit more about D3, I recommend *Scott Murray's Interactive Data Visualization for the Web*.

You may also wish to consult the references given in the previous chapter for reference books on the Play framework and designing REST APIs.

Pattern Matching and Extractors

Pattern matching is a powerful tool for control flow in Scala. It is often underused and under-estimated by people coming to Scala from imperative languages.

Let's start with a few examples of pattern matching before diving into the theory. We start by defining a tuple:

```
scala> val names = ("Pascal", "Bugnion")
names: (String, String) = (Pascal,Bugnion)
```

We can use pattern matching to extract the elements of this tuple and bind them to variables:

```
scala> val (firstName, lastName) = names
firstName: String = Pascal
lastName: String = Bugnion
```

We just extracted the two elements of the `names` tuple, binding them to the variables `firstName` and `lastName`. Notice how the left-hand side defines a pattern that the right-hand side must match: we are declaring that the variable `names` must be a two-element tuple. To make the pattern more specific, we could also have specified the expected types of the elements in the tuple:

```
scala> val (firstName:String, lastName:String) = names
firstName: String = Pascal
lastName: String = Bugnion
```

What happens if the pattern on the left-hand side does not match the right-hand side?

```
scala> val (firstName, middleName, lastName) = names
<console>:13: error: constructor cannot be instantiated to expected type;
          found   : (T1, T2, T3)
          required: (String, String)
              val (firstName, middleName, lastName) = names
```

This results in a compile error. Other types of pattern matching failures result in runtime errors.

Pattern matching is very expressive. To achieve the same behavior without pattern matching, you would have to do the following explicitly:

- Verify that the variable names is a two-element tuple
- Extract the first element and bind it to firstName
- Extract the second element and bind it to lastName

If we expect certain elements in the tuple to have specific values, we can verify this as part of the pattern match. For instance, we can verify that the first element of the names tuple matches "Pascal":

```
scala> val ("Pascal", lastName) = names
lastName: String = Bugnion
```

Besides tuples, we can also match on Scala collections:

```
scala> val point = Array(1, 2, 3)
point: Array[Int] = Array(1, 2, 3)

scala> val Array(x, y, z) = point
x: Int = 1
y: Int = 2
z: Int = 3
```

Notice the similarity between this pattern matching and array construction:

```
scala> val point = Array(x, y, z)
point: Array[Int] = Array(1, 2, 3)
```

Syntactically, Scala expresses pattern matching as the reverse process to instance construction. We can think of pattern matching as the deconstruction of an object, binding the object's constituent parts to variables.

When matching against collections, one is sometimes only interested in matching the first element, or the first few elements, and discarding the rest of the collection, whatever its length. The operator `_*` will match against any number of elements:

```
scala> val Array(x, _) = point
x: Int = 1
```

By default, the part of the pattern matched by the `_*` operator is not bound to a variable. We can capture it as follows:

```
scala> val Array(x, xs @ _) = point
x: Int = 1
xs: Seq[Int] = Vector(2, 3)
```

Besides tuples and collections, we can also match against case classes. Let's start by defining a case representing a name:

```
scala> case class Name(first: String, last: String)
defined class Name

scala> val name = Name("Martin", "Odersky")
name: Name = Name(Martin,Odersky)
```

We can match against instances of `Name` in much the same way we matched against tuples:

```
scala> val Name(firstName, lastName) = name
firstName: String = Martin
lastName: String = Odersky
```

All these patterns can also be used in `match` statements:

```
scala> def greet(name:Name) = name match {
  case Name("Martin", "Odersky") => "An honor to meet you"
  case Name(first, "Bugnion") => "Wow! A family member!"
  case Name(first, last) => s"Hello, $first"
}
greet: (name: Name)String
```

Pattern matching in for comprehensions

Pattern matching is useful in *for* comprehensions for extracting items from a collection that match a specific pattern. Let's build a collection of `Name` instances:

```
scala> val names = List(Name("Martin", "Odersky"),
  Name("Derek", "Wyatt"))
names: List[Name] = List(Name(Martin,Odersky), Name(Derek,Wyatt))
```

We can use pattern matching to extract the internals of the class in a *for*-comprehension:

```
scala> for { Name(first, last) <- names } yield first
List[String] = List(Martin, Derek)
```

So far, nothing terribly ground-breaking. But what if we wanted to extract the surname of everyone whose first name is "Martin"?

```
scala> for { Name("Martin", last) <- names } yield last
List[String] = List(Odersky)
```

Writing `Name("Martin", last) <- names` extracts the elements of `names` that match the pattern. You might think that this is a contrived example, and it is, but the examples in *Chapter 7, Web APIs* demonstrate the usefulness and versatility of this language pattern, for instance, for extracting specific fields from JSON objects.

Pattern matching internals

If you define a case class, as we saw with `Name`, you get pattern matching against the constructor *for free*. You should be using case classes to represent your data as much as possible, thus reducing the need to implement your own pattern matching. It is nevertheless useful to understand how pattern matching works.

When you create a case class, Scala automatically builds a companion object:

```
scala> case class Name(first: String, last: String)
defined class Name

scala> Name.<tab>
apply    asInstanceOf   curried    isInstanceOf   toString   tupled
unapply
```

The method used (internally) for pattern matching is `unapply`. This method takes, as argument, an object and returns `Option[T]`, where `T` is a tuple of the values of the case class.

```
scala> val name = Name("Martin", "Odersky")
name: Name = Name(Martin,Odersky)

scala> Name.unapply(name)
Option[(String, String)] = Some((Martin,Odersky))
```

The `unapply` method is an *extractor*. It plays the opposite role of the constructor: it takes an object and extracts the list of parameters needed to construct that object. When you write `val Name(firstName, lastName)`, or when you use `Name` as a case in a match statement, Scala calls `Name.unapply` on what you are matching against. A value of `Some[(String, String)]` implies a pattern match, while a value of `None` implies that the pattern fails.

To write custom extractors, you just need an object with an `unapply` method. While `unapply` normally resides in the companion object of a class that you are deconstructing, this need not be the case. In fact, it does not need to correspond to an existing class at all. For instance, let's define a `NonZeroDouble` extractor that matches any non-zero double:

```
scala> object NonZeroDouble {
  def unapply(d:Double):Option[Double] = {
    if (d == 0.0) { None } else { Some(d) }
  }
}
defined object NonZeroDouble

scala> val NonZeroDouble(denominator) = 5.5
denominator: Double = 5.5

scala> val NonZeroDouble(denominator) = 0.0
scala.MatchError: 0.0 (of class java.lang.Double)
... 43 elided
```

We defined an extractor for `NonZeroDouble`, despite the absence of a corresponding `NonZeroDouble` class.

This `NonZeroDouble` extractor would be useful in a match object. For instance, let's define a `safeDivision` function that returns a default value when the denominator is zero:

```
scala> def safeDivision(numerator:Double,  
denominator:Double, fallBack:Double) =  
denominator match {  
  case NonZeroDouble(d) => numerator / d  
  case _ => fallBack  
}  
safeDivision: (numerator: Double, denominator: Double, fallBack: Double)  
Double  
  
scala> safeDivision(5.0, 2.0, 100.0)  
Double = 2.5  
  
scala> safeDivision(5.0, 0.0, 100.0)  
Double = 100.0
```

This is a trivial example because the `NonZeroDouble.unapply` method is so simple, but you can hopefully see the usefulness and expressiveness, if we were to define a more complex test. Defining custom extractors lets you define powerful control flow constructs to leverage `match` statements. More importantly, they enable the client using the extractors to think about control flow declaratively: the client can declare that they need a `NonZeroDouble`, rather than instructing the compiler to check whether the value is zero.

Extracting sequences

The previous section explains extraction from case classes, and how to write custom extractors, but it does not explain how extraction works on sequences:

```
scala> val Array(a, b) = Array(1, 2)  
a: Int = 1  
b: Int = 2
```

Rather than relying on an `unapply` method, sequences rely on an `unapplySeq` method defined in the companion object. This is expected to return an `Option[Seq[A]]`:

```
scala> Array.unapplySeq(Array(1, 2))  
Option[IndexedSeq[Int]] = Some(Vector(1, 2))
```

Let's write an example. We will write an extractor for Breeze vectors (which do not currently support pattern matching). To avoid clashing with the DenseVector companion object, we will write our unapplySeq in a separate object, called DV. All our unapplySeq method needs to do is convert its argument to a Scala Vector instance. To avoid muddying the concepts with generics, we will write this implementation for [Double] vectors only:

```
scala> import breeze.linalg._  
import breeze.linalg._  
  
scala> object DV {  
    // Just need to convert to a Scala vector.  
    def unapplySeq(v:DenseVector[Double]) = Some(v.toScalaVector)  
}  
defined object DV
```

Let's try our new extractor implementation:

```
scala> val vec = DenseVector(1.0, 2.0, 3.0)  
vec: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0, 3.0)  
  
scala> val DV(x, y, z) = vec  
x: Double = 1.0  
y: Double = 2.0  
z: Double = 3.0
```

Summary

Pattern matching is a powerful tool for control flow. It encourages the programmer to think declaratively: declare that you expect a variable to match a certain pattern, rather than explicitly tell the computer how to check that it matches this pattern. This can save many lines of code and enhance clarity.

Reference

For an overview of pattern matching in Scala, there is no better reference than *Programming in Scala*, by Martin Odersky, Bill Venners, and Lex Spoon. An online version of the first edition is available at: <https://www.artima.com/pins1ed/case-classes-and-pattern-matching.html>.

Daniel Westheide's blog covers slightly more advanced Scala constructs, and is a very useful read: <http://danielwestheide.com/blog/2012/11/21/the-neophytes-guide-to-scala-part-1-extractors.html>.

Module 2

Scala for Machine Learning

Leverage Scala and Machine Learning to construct and study systems that can learn from data

1

Getting Started

It is critical for any computer scientist to understand the different classes of machine learning algorithms and be able to select the ones that are relevant to the domain of their expertise and dataset. However, the application of these algorithms represents a small fraction of the overall effort needed to extract an accurate and performing model from input data. A common data mining workflow consists of the following sequential steps:

1. Defining the problem to solve.
2. Loading the data.
3. Preprocessing, analyzing, and filtering the input data.
4. Discovering patterns, affinities, clusters, and classes, if needed.
5. Selecting the model features and appropriate machine learning algorithm(s).
6. Refining and validating the model.
7. Improving the computational performance of the implementation.

In this book, each stage of the process is critical to build the *right* model.



It is impossible to describe the key machine learning algorithms and their implementations in detail in a single book. The sheer quantity of information and Scala code would overwhelm even the most dedicated readers. Each chapter focuses on the mathematics and code that are absolutely essential to the understanding of the topic. Developers are encouraged to browse through the following:

- The Scala coding convention and standard used in the book in the *Appendix A, Basic Concepts*
- API Scala docs
- A fully documented source code that is available online

This first chapter introduces you to the taxonomy of machine learning algorithms, the tools and frameworks used in the book, and a simple application of logistic regression to get your feet wet.

Mathematical notation for the curious

Each chapter contains a small section dedicated to the formulation of the algorithms for those interested in the mathematical concepts behind the science and art of machine learning. These sections are optional and defined within a tip box. For example, the mathematical expression of the mean and the variance of a variable X mentioned in a tip box will be as follows:



Convention and notation

This book uses zero-based indexing of datasets in the mathematical formulas.

M1: A set of N observations is denoted as $\{x_i\} = x_0, x_1, \dots, x_{N-1}$, and the arithmetic mean value for the random value with x_i as values is defined as:

$$E[X] = \mu = \frac{1}{n} \sum_{i=0}^{n-1} x_i$$

Why machine learning?

The explosion in the number of digital devices generates an ever-increasing amount of data. The best analogy I can find to describe the need, desire, and urgency to extract knowledge from large datasets is the process of extracting a precious metal from a mine, and in some cases, extracting blood from a stone.

Knowledge is quite often defined as a model that can be constantly updated or tweaked as new data comes into play. Models are obviously domain-specific ranging from credit risk assessment, face recognition, maximization of quality of service, classification of pathological symptoms of disease, optimization of computer networks, and security intrusion detection, to customers' online behavior and purchase history.

Machine learning problems are categorized as classification, prediction, optimization, and regression.

Classification

The purpose of classification is to extract knowledge from historical data. For instance, a classifier can be built to identify a disease from a set of symptoms. The scientist collects information regarding the body temperature (continuous variable), congestion (discrete variables *HIGH*, *MEDIUM*, and *LOW*), and the actual diagnostic (flu). This dataset is used to create a model such as *IF temperature > 102 AND congestion = HIGH THEN patient has the flu (probability 0.72)*, which doctors can use in their diagnostic.

Prediction

Once the model is trained using historical observations and validated against historical observations, it can be used to predict some outcome. A doctor collects symptoms from a patient, such as body temperature and nasal congestion, and anticipates the state of his/her health.

Optimization

Some global optimization problems are intractable using traditional linear and non-linear optimization methods. Machine learning techniques improve the chances that the optimization method converges toward a solution (intelligent search). You can imagine that fighting the spread of a new virus requires optimizing a process that may evolve over time as more symptoms and cases are uncovered.

Regression

Regression is a classification technique that is particularly suitable for a continuous model. Linear (least squares), polynomial, and logistic regressions are among the most commonly used techniques to fit a parametric model, or function, $y=f(x)$, $x=\{x_i\}$, to a dataset. Regression is sometimes regarded as a specialized case of classification for which the output variables are continuous instead of categorical.

Why Scala?

Like most functional languages, Scala provides developers and scientists with a toolbox to implement iterative computations that can be easily woven into a coherent dataflow. To some extent, Scala can be regarded as an extension of the popular MapReduce model for distributed computation of large amounts of data. Among the capabilities of the language, the following features are deemed essential in machine learning and statistical analysis.

Abstraction

Functors and **monads** are important concepts in functional programming. Monads are derived from the category and group theory that allow developers to create a high-level abstraction as illustrated in **Scalaz**, Twitter's **Algebird**, or Google's **Breeze Scala** libraries. More information about these libraries can be found at the following links:

- <https://github.com/scalaz>
- <https://github.com/twitter/algebird>
- <https://github.com/dlwh/breeze>

In mathematics, a category **M** is a structure that is defined by:

- Objects of some type: $\{x \in X, y \in Y, z \in Z, \dots\}$
- Morphisms or maps applied to these objects: $x \in X, y \in Y, f: x \rightarrow y$
- Composition of morphisms: $f: x \rightarrow y, g: y \rightarrow z \Rightarrow g \circ f: x \rightarrow z$

Covariant, contravariant functors, and **bifunctors** are well-understood concepts in algebraic topology that are related to manifold and vector bundles. They are commonly used in differential geometry and generation of non-linear models from data.

Higher-kind projection

Scientists define observations as sets or vectors of features. Classification problems rely on the estimation of the similarity between vectors of observations. One technique consists of comparing two vectors by computing the normalized inner product. A **co-vector** is defined as a linear map α of a vector to the inner product (field).



Inner product
M1: The definition of a $\langle \cdot, \cdot \rangle$ inner product and a α co-vector is as follows:

$$\langle \vec{v}, \vec{w} \rangle = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}| \cdot |\vec{w}|} \quad \alpha: \vec{v} \rightarrow \langle \vec{v}, \vec{w} \rangle$$

Let's define a vector as a constructor from any `_ => Vector[_]` field (or `Function1[_ , Vector]`). A co-vector is then defined as the mapping function of a vector to its `Vector[_] => _` field (or `Function1[Vector, _]`).

Let's define a two-dimensional (two types or fields) higher kind structure, `Hom`, that can be defined as either a vector or co-vector by fixing one of the two types:

```
type Hom[T] = {
    type Right[X] = Function1[X, T] // Co-vector
    type Left[X] = Function1[T, X]   // Vector
}
```

Tensors and manifolds

Vectors and co-vectors are classes of tensor (contravariant and covariant). Tensors (fields) are used in manifold learning of nonlinear models and in the generation of kernel functions. Manifolds are briefly introduced in the *Manifolds* section under *Dimension reduction* in *Chapter 4, Unsupervised Learning*. The topic of tensor fields and manifold learning is beyond the scope of this book.



The projections of the higher kind, `Hom`, to the `Right` or `Left` single parameter types are known as functors, which are as follows:

- A covariant functor for the `right` projection
- A contravariant functor for the `left` projection.

Covariant functors for vectors

A **covariant functor** of a variable is a map $F: C \Rightarrow C$ such that:

- If $f: x \rightarrow y$ is a morphism on C , then $F(x) \rightarrow F(y)$ is also a morphism on C
- If $id: x \rightarrow x$ is the identity morphism on C , then $F(id)$ is also an identity morphism on C
- If $g: y \rightarrow z$ is also a morphism on C , then $F(g \circ f) = F(g) \circ F(f)$

The definition of the `F[U => V] := F[U] => F[V]` covariant functor in Scala is as follows:

```
trait Functor[M[_]] {
    def map[U, V](m: M[U])(f: U => V): M[V]
}
```

For example, let's consider an observation defined as a n dimension vector of a T type, $\text{Obs}[T]$. The constructor for the observation can be represented as $\text{Function1}[T, \text{Obs}]$. Its ObsFunctor functor is implemented as follows:

```
trait ObsFunctor[T] extends Functor[(Hom[T])#Left] { self =>
    override def map[U, V](vu: Function1[T, U])(f: U => V):
        Function1[T, V] = f.compose(vu)
}
```

The functor is qualified as a **covariant functor** because the morphism is applied to the return type of the element of Obs as $\text{Function1}[T, \text{Obs}]$. The Hom projection of the two parameters types to a vector is implemented as $(\text{Hom}[T])\#\text{Left}$.

Contravariant functors for co-vectors

A contravariant functor of one variable is a map $F: C \Rightarrow C$ such that:

- If $f: x \rightarrow y$ is a morphism on C , then $F(y) \rightarrow F(x)$ is also a morphism on C
- If $id: x \rightarrow x$ is the identity morphism on C , then $F(id)$ is also an identity morphism on C
- If $g: y \rightarrow z$ is also a morphism on C , then $F(g \circ f) = F(f) \circ F(g)$

The definition of the $F[U \Rightarrow V] := F[V] \Rightarrow F[U]$ contravariant functor in Scala is as follows:

```
trait CoFunctor[M[_]] {
    def map[U, V](m: M[U])(f: V => U): M[V]
}
```

Note that the input and output types in the f morphism are reversed from the definition of a covariant functor. The constructor for the co-vector can be represented as $\text{Function1}[\text{Obs}, T]$. Its CoObsFunctor functor is implemented as follows:

```
trait CoObsFunctor[T] extends CoFunctor[(Hom[T])#Right] {
    self =>
    override def map[U, V](vu: Function1[U, T])(f: V => U):
        Function1[V, T] = f.andThen(vu)
}
```

Monads

Monads are structures in algebraic topology that are related to the category theory. Monads extend the concept of a functor to allow a composition known as the **monadic composition** of morphisms on a single type. They enable the chaining or weaving of computation into a sequence of steps or pipeline. The collections bundled with the Scala standard library (`List`, `Map`, and so on) are constructed as monads [1:1].

Monads provide the ability for those collections to perform the following functions:

- Create the collection
- Transform the elements of the collection
- Flatten nested collections

An example is as follows:

```
trait Monad[M[_]] {
    def unit[T](a: T): M[T]
    def map[U,V](m: M[U])(f: U => V): M[V]
    def flatMap[U,V](m: M[U])(f: U => M[V]): M[V]
}
```

Monads are therefore critical in machine learning as they enable you to compose multiple data transformation functions into a sequence or workflow. This property is applicable to any type of complex scientific computation [1:2].



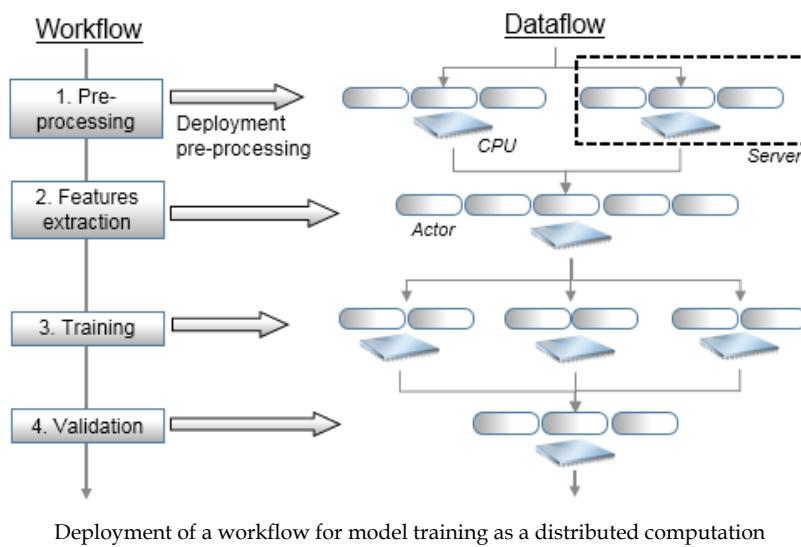
The monadic composition of kernel functions

Monads are used in the composition of kernel functions in the *Kernel monadic composition section under Kernel functions* section in *Chapter 8, Kernel Models and Support Vector Machines*.

Scalability

As seen previously, functors and monads enable parallelization and chaining of data processing functions by leveraging the Scala higher-order methods. In terms of implementation, **actors** are one of the core elements that make Scala scalable. Actors provide Scala developers with a high level of abstraction to build scalable, distributed, and concurrent applications. Actors hide the nitty-gritty implementation details of concurrency and the management of the underlying threads pool. Actors communicate through asynchronous immutable messages. A distributed computing Scala framework such as **Akka** or **Apache Spark** extends the capabilities of the Scala standard library to support computation on very large datasets. Akka and Apache Spark are described in detail in the last chapter of this book [1:3].

In a nutshell, a workflow is implemented as a sequence of activities or computational tasks. These tasks consist of high-order Scala methods such as `flatMap`, `map`, `fold`, `reduce`, `collect`, `join`, or `filter` that are applied to a large collection of observations. Scala provides developers with the tools to partition datasets and execute the tasks through a cluster of actors. Scala also supports message dispatching and routing between local and remote actors. A developer can decide to deploy a workflow either locally or across multiple CPU cores and servers with very few code alterations.



In the preceding diagram, a controller, that is, the master node, manages the sequence of tasks **1** to **4** similar to a scheduler. These tasks are actually executed over multiple worker nodes, which are implemented by actors. The master node or actor exchanges messages with the workers to manage the state of the execution of the workflow as well as its reliability, as illustrated in the *Scalability with Actors* section in *Chapter 12, Scalable Frameworks*. High availability of these tasks is implemented through a hierarchy of supervising actors.

Configurability

Scala supports **dependency injection** using a combination of abstract variables, self-referenced composition, and stackable traits. One of the most commonly used dependency injection patterns, the **cake pattern**, is described in the *Composing mixins to build a workflow* section in *Chapter 2, Hello World!*

Maintainability

Scala embeds **Domain Specific Languages (DSL)** natively. DSLs are syntactic layers built on top of Scala native libraries. DSLs allow software developers to abstract computation in terms that are easily understood by scientists. The most notorious application of DSLs is the definition of the emulation of the syntax used in the MATLAB program, which data scientists are familiar with.

Computation on demand

Lazy methods and values allow developers to execute functions and allocate computing resources on demand. The Spark framework relies on lazy variables and methods to chain **Resilient Distributed Datasets (RDD)**.

Model categorization

A model can be predictive, descriptive, or adaptive.

Predictive models discover patterns in historical data and extract fundamental trends and relationships between factors (or features). They are used to predict and classify future events or observations. Predictive analytics is used in a variety of fields, including marketing, insurance, and pharmaceuticals. Predictive models are created through supervised learning using a preselected training set.

Descriptive models attempt to find unusual patterns or affinities in data by grouping observations into clusters with similar properties. These models define the first and important step in knowledge discovery. They are generated through unsupervised learning.

A third category of models, known as **adaptive modeling**, is created through **reinforcement learning**. Reinforcement learning consists of one or several decision-making agents that recommend and possibly execute actions in the attempt of solving a problem, optimizing an objective function, or resolving constraints.

Taxonomy of machine learning algorithms

The purpose of machine learning is to teach computers to execute tasks without human intervention. An increasing number of applications such as genomics, social networking, advertising, or risk analysis generate a very large amount of data that can be analyzed or mined to extract knowledge or insight into a process, customer, or organization. Ultimately, machine learning algorithms consist of identifying and validating models to optimize a performance criterion using historical, present, and future data [1:4].

Data mining is the process of extracting or identifying patterns in a dataset.

Unsupervised learning

The goal of **unsupervised learning** is to discover patterns of regularities and irregularities in a set of observations. The process is known as density estimation in statistics is broken down into two categories: discovery of data clusters and discovery of latent factors. The methodology consists of processing input data to understand patterns similar to the natural learning process in infants or animals. Unsupervised learning does not require labeled data (or expected values), and therefore, it is easy to implement and execute because no expertise is needed to validate an output. However, it is possible to label the output of a clustering algorithm and use it for future classification.

Clustering

The purpose of **data clustering** is to partition a collection of data into a number of clusters or data segments. Practically, a clustering algorithm is used to organize observations into clusters by minimizing the distance between observations within a cluster and maximizing the distance between observations across clusters. A clustering algorithm consists of the following steps:

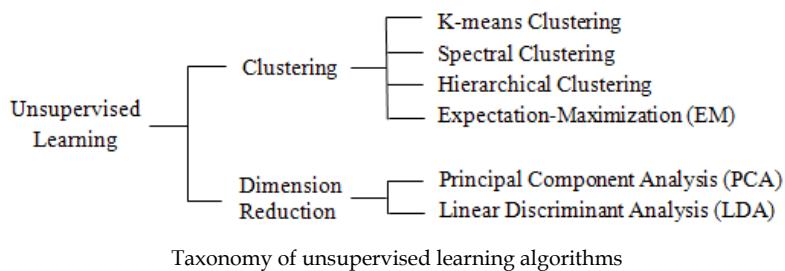
1. Creating a model by making an assumption on the input data.
2. Selecting the objective function or goal of the clustering.
3. Evaluating one or more algorithms to optimize the objective function.

Data clustering is also known as **data segmentation** or **data partitioning**.

Dimension reduction

Dimension reduction techniques aim at finding the smallest but most relevant set of features needed to build a reliable model. There are many reasons for reducing the number of features or parameters in a model, from avoiding overfitting to reducing computation costs.

There are many ways to classify the different techniques used to extract knowledge from data using unsupervised learning. The following taxonomy breaks down these techniques according to their purpose, although the list is far from being exhaustive, as shown in the following diagram:



Supervised learning

The best analogy for supervised learning is **function approximation** or **curve fitting**. In its simplest form, supervised learning attempts to find a relation or function $f: x \rightarrow y$ using a training set $\{x, y\}$. Supervised learning is far more accurate than any other learning strategy as long as the input (labeled data) is available and reliable. The downside is that a domain expert may be required to label (or tag) data as a training set.

Supervised machine learning algorithms can be broken into two categories:

- Generative models
- Discriminative models

Generative models

In order to simplify the description of a statistics formula, we adopt the following simplification: the probability of an X event is the same as the probability of the discrete X random variable to have a value x : $p(X) = p(X=x)$.

The notation for the joint probability is $p(X, Y) = p(X=x, Y=y)$.

The notation for the conditional probability is $p(X | Y) = p(X=x | Y=y)$.

Generative models attempt to fit a joint probability distribution, $p(X, Y)$, of two X and Y events (or random variables), representing two sets of observed and hidden x and y variables. Discriminative models compute the conditional probability, $p(Y | X)$, of an event or random variable Y of hidden variables y , given an event or random variable X of observed variables x . Generative models are commonly introduced through the Bayes' rule. The conditional probability of a Y event, given an X event, is computed as the product of the conditional probability of the X event, given the Y event, and the probability of the X event normalized by the probability of the Y event [1:5].

Bayes' rule

Joint probability for independent random variables, $X=x$ and $Y=y$, is given by:

$$p(X, Y) = p(X \cap Y) = p(X) \cdot p(Y)$$



Conditional probability of a random variable, $Y = y$, given $X = x$, is given by:

$$p(Y|X) = p(Y, X)/p(X)$$

Bayes' formula is given by:

$$p(Y|X) = p(X|Y) \cdot p(Y)/p(X)$$

The Bayes' rule is the foundation of the Naïve Bayes classifier, as described in the *Introducing the multinomial Naïve Bayes* section in *Chapter 5, Naïve Bayes Classifiers*.

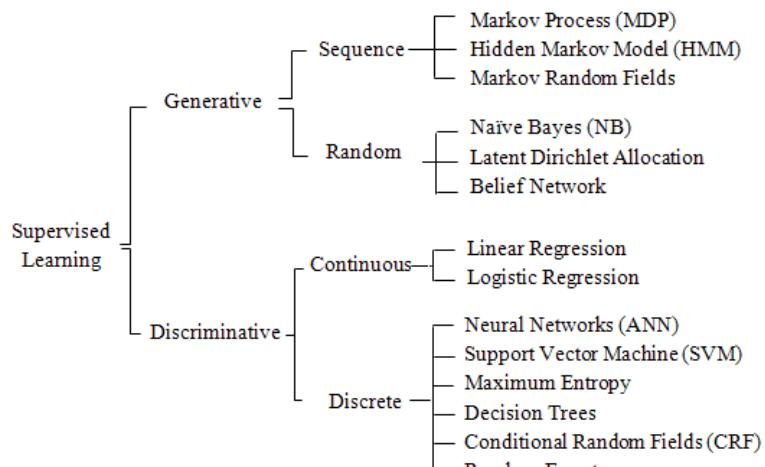
Discriminative models

Contrary to generative models, discriminative models compute the conditional probability $p(Y | X)$ directly, using the same algorithm for training and classification.

Generative and discriminative models have their respective advantages and disadvantages. Novice data scientists learn to match the appropriate algorithm to each problem through experimentation. Here is a brief guideline describing which type of models make sense according to the objective or criteria of the project:

Objective	Generative models	Discriminative models
Accuracy	Highly dependent on the training set.	This depends on the training set and algorithm configuration (that is, kernel functions)
Modeling requirements	There is a need to model both observed and hidden variables, which requires a significant amount of training.	The quality of the training set does not have to be as rigorous as for generative models.
Computation cost	This is usually low. For example, any graphical method derived from the Bayes' rule has low overhead.	Most algorithms rely on optimization of a convex function with significant performance overhead.
Constraints	These models assume some degree of independence among the model features.	Most discriminative algorithms accommodate dependencies between features.

We can further refine the taxonomy of supervised learning algorithms by segregating arbitrarily between sequential and random variables for generative models and breaking down discriminative methods as applied to continuous processes (regression) and discrete processes (classification):



Taxonomy of supervised learning algorithms

Semi-supervised learning

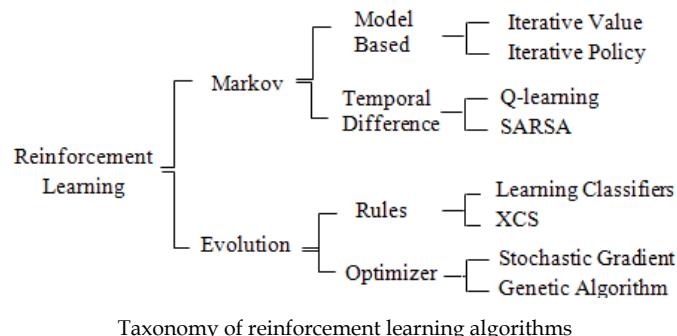
Semi-supervised learning is used to build models from a dataset with incomplete labels. Manifold learning and information geometry algorithms are commonly applied to large datasets that are partially labeled. The description of semi-supervised learning techniques is beyond the scope of this book.

Reinforcement learning

Reinforcement learning is not as well understood as supervised and unsupervised learning outside the realms of robotics or game strategy. However, since the 90s, genetic-algorithms-based classifiers have become increasingly popular to solve problems that require collaboration with a domain expert. For some types of applications, reinforcement learning algorithms output a set of recommended actions for the adaptive system to execute. In its simplest form, these algorithms estimate the best course of action. Most complex systems based on reinforcement learning establish and update policies that can be vetoed by an expert, if necessary. The foremost challenge developers of reinforcement learning systems face is that the recommended action or policy may depend on partially observable states.

Genetic algorithms are not usually considered part of the reinforcement learning toolbox. However, advanced models, such as learning classifier systems, use genetic algorithms to classify and reward the most performing rules and policies.

As with the two previous learning strategies, reinforcement learning models can be categorized as Markovian or evolutionary:



Taxonomy of reinforcement learning algorithms

This is a brief overview of machine learning algorithms with a suggested, approximate taxonomy. There are almost as many ways to introduce machine learning as there are data and computer scientists. We encourage you to browse through the list of references at the end of the book to find the documentation appropriate to your level of interest and understanding.

Don't reinvent the wheel!

There are numerous robust, accurate, and efficient Java libraries for mathematics, linear algebra, or optimization that have been widely used for many years:

- JBlas/Linpack (<https://github.com/mikiobraun/jblas>)
- Parallel Colt (<https://github.com/rwl/ParallelColt>)
- Apache Commons Math (<http://commons.apache.org/proper/commons-math>)

There is absolutely no need to rewrite, debug, and test these components in Scala. Developers should consider creating a wrapper or interface to his/her favorite and reliable Java library. The book leverages the Apache Commons Math library for some specific linear algebra algorithms.

Tools and frameworks

Before getting your hands dirty, you need to download and deploy a minimum set of tools and libraries; there is no need to reinvent the wheel after all. A few key components have to be installed in order to compile and run the source code described throughout the book. We focus on open source and commonly available libraries, although you are invited to experiment with equivalent tools of your choice. The learning curve for the frameworks described here is minimal.

Java

The code described in this book has been tested with JDK 1.7.0_45 and JDK 1.8.0_25 on Windows x64 and Mac OS X x64. You need to install the Java Development Kit if you have not already done so. Finally, the `JAVA_HOME`, `PATH`, and `CLASSPATH` environment variables have to be updated accordingly.

Scala

The code has been tested with Scala 2.10.4 and 2.11.4. We recommend that you use Scala Version 2.10.4 or higher with SBT 0.13 or higher. Let's assume that Scala runtime (REPL) and libraries have been properly installed and the `SCALA_HOME` and `PATH` environment variables have been updated.

The description and installation instructions of the **Scala plugin for Eclipse** (version 4.0 or higher) are available at <http://scala-ide.org/docs/user/gettingstarted.html>. You can also download the **Scala plugin for IntelliJ IDEA** (version 13 or higher) from the JetBrains website at <http://confluence.jetbrains.com/display/SCA/>.

The ubiquitous **Simple Build Tool (SBT)** will be our primary building engine. The syntax of the build file, `sbt/build.sbt`, conforms to the Version 0.13 and is used to compile and assemble the source code presented throughout the book. Sbt can be downloaded as part of Typesafe activator or directly from <http://www.scala-sbt.org/download.html>.

Apache Commons Math

Apache Commons Math is a Java library used for numerical processing, algebra, statistics, and optimization [1:6].

Description

This is a lightweight library that provides developers with a foundation of small, ready-to-use Java classes that can be easily weaved into a machine learning problem. The examples used throughout the book require Version 3.5 or higher.

The math library supports the following:

- Functions, differentiation, and integral and ordinary differential equations
- Statistics distributions
- Linear and nonlinear optimization
- Dense and sparse vectors and matrices
- Curve fitting, correlation, and regression

For more information, visit <http://commons.apache.org/proper/commons-math>.

Licensing

We need Apache Public License 2.0; the terms are available at <http://www.apache.org/licenses/LICENSE-2.0>.

Installation

The installation and deployment of the Apache Commons Math library are quite simple. The steps are as follows:

1. Go to the download page at http://commons.apache.org/proper/commons-math/download_math.cgi.
2. Download the latest .jar files to the binary section, `commons-math3-3.5-bin.zip` (for instance, for Version 3.5).
3. Unzip and install the .jar file.

4. Add commons-math3-3.5.jar to the classpath as follows:
 - **For Mac OS X:** export CLASSPATH=\$CLASSPATH:/Commons_Math_path/commons-math3-3.5.jar
 - **For Windows:** Go to system **Properties** | **Advanced system settings** | **Advanced** | **Environment Variables**, then edit the CLASSPATH variable
5. Add the commons-math3-3.5.jar file to your IDE environment if needed (that is, for Eclipse, go to **Project** | **Properties** | **Java Build Path** | **Libraries** | **Add External JARs** and for IntelliJ IDEA, go to **File** | **Project Structure** | **Project Settings** | **Libraries**).

You can also download commons-math3-3.5-src.zip from the **Source** section.

JFreeChart

JFreeChart is an open source chart and plotting Java library, widely used in the Java programmer community. It was originally created by David Gilbert [1:7].

Description

The library supports a variety of configurable plots and charts (scatter, dial, pie, area, bar, box and whisker, stacked, and 3D). We use JFreeChart to display the output of data processing and algorithms throughout the book, but you are encouraged to explore this great library on your own, as time permits.

Licensing

It is distributed under the terms of the **GNU Lesser General Public License (LGPL)**, which permits its use in proprietary applications.

Installation

To install and deploy JFreeChart, perform the following steps:

1. Visit <http://www.jfree.org/jfreechart/>.
2. Download the latest version from Source Forge at <http://sourceforge.net/projects/jfreechart/files>.
3. Unzip and deploy the .jar file.

4. Add `jfreechart-1.0.17.jar` (for Version 1.0.17) to the classpath as follows:
 - **For Mac OS X:** `export CLASSPATH=$CLASSPATH:/JFreeChart_path/jfreechart-1.0.17.jar`
 - **For Windows:** Go to system **Properties** | **Advanced system settings** | **Advanced** | **Environment Variables**, then edit the `CLASSPATH` variable
5. Add the `jfreechart-1.0.17.jar` file to your IDE environment, if needed

Other libraries and frameworks

Libraries and tools that are specific to a single chapter are introduced along with the topic. Scalable frameworks are presented in the last chapter along with the instructions to download them. Libraries related to the conditional random fields and support vector machines are described in their respective chapters.



Why not use the Scala algebra and numerical libraries?

Libraries such as Breeze, ScalaNLP, and Algebird are interesting Scala frameworks for linear algebra, numerical analysis, and machine learning. They provide even the most seasoned Scala programmer with a high-quality layer of abstraction. However, this book is designed as a tutorial that allows developers to write algorithms from the ground up using existing or legacy Java libraries [1:8].

Source code

The Scala programming language is used to implement and evaluate the machine learning techniques covered in *Scala for Machine Learning*. However, the source code snippets are reduced to the strict minimum essential to the understanding of machine learning algorithms discussed throughout the book. The formal implementation of these algorithms is available on the website of Packt Publishing (<http://www.packtpub.com>).



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Context versus view bounds

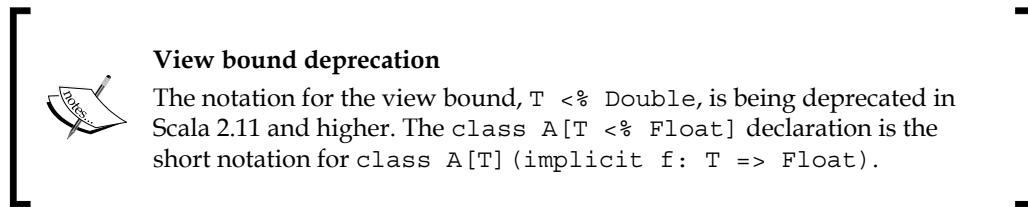
Most Scala classes discussed in the book are parameterized with the type associated with the discrete/categorical value (`Int`) or continuous value (`Double`). Context bounds would require that any type used by the client code has `Int` or `Double` as upper bounds:

```
class A[T <: Int] (param: Param)
class B[T <: Double] (param: Param)
```

Such a design introduces constraints on the client to inherit from simple types and to deal with covariance and contravariance for container types [1:9].

For this book, **view bounds** are used instead of context bounds because they only require an implicit conversion to the parameterized type to be defined:

```
class A[T <: AnyVal] (param: Param) (implicit f: T => Int)
class C[T <: AnyVal] (param: Param) (implicit f: T => Float)
```



Presentation

For the sake of readability of the implementation of algorithms, all nonessential code such as error checking, comments, exceptions, or imports are omitted. The following code elements are omitted in the code snippet presented in the book:

- Code documentation:


```
// ....
/* ... */
```
- Validation of class parameters and method arguments:


```
require( Math.abs(x) < EPS, " ...")
```
- Class qualifiers and scope declaration:


```
final protected class SVM { ... }
private[this] val lsError = ...
```

- Method qualifiers:

```
final protected def dot: = ...
```

- Exceptions:

```
try {
    correlate ...
} catch {
    case e: MathException => ....
}
Try {    .. } match {
    case Success(res) =>
    case Failure(e => ..
}
```

- Logging and debugging code:

```
private val logger = Logger.getLogger("...")
logger.info( ... )
```

- Nonessential annotation:

```
@inline def main = ....
@throw(classOf[IllegalStateException])
```

- Nonessential methods

The complete list of Scala code elements omitted in the code snippets in this book can be found in the *Code snippets format* section in the *Appendix A, Basic Concepts*.

Primitives and implicits

The algorithms presented in this book share the same primitive types, generic operators, and implicit conversions.

Primitive types

For the sake of readability of the code, the following primitive types will be used:

```
type DblPair = (Double, Double)
type DblArray = Array[Double]
type DblMatrix = Array[DblArray]
type DblVector = Vector[Double]
type XSeries[T] = Vector[T]           // One dimensional vector
type XVSeries[T] = Vector[Array[T]]   // multi-dimensional vector
```

The times series introduced in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*, is implemented as `XSeries[T]` or `XVSeries[T]` of a parameterized `T` type.



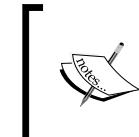
Make a note of these six types; they are used throughout the book.



Type conversions

Implicit conversion is an important feature of the Scala programming language. It allows developers to specify a type conversion for an entire library in a single place. Here are a few of the implicit type conversions that are used throughout the book:

```
object Types {
    object ScalaML {
        implicit def double2Array(x: Double): DblArray =
            Array[Double](x)
        implicit def dblPair2Vector(x: DblPair): Vector[DblPair] =
            Vector[DblPair](x._1, x._2)
        ...
    }
}
```



Library-specific conversion

The conversion between the primitive type listed here and types introduced in a particular library (such as, the Apache Commons Math library) are described in the relevant chapters.



Immutability

It is usually a good idea to reduce the number of states of an object. A method invocation transitions an object from one state to another. The larger the number of methods or states, the more cumbersome the testing process becomes.

There is no point in creating a model that is not defined (trained). Therefore, making the training of a model as part of the constructor of the class it implements makes a lot of sense. Therefore, the only public methods of a machine learning algorithm are as follows:

- Classification or prediction
- Validation
- Retrieval of model parameters (weights, latent variables, hidden states, and so on), if needed

Performance of Scala iterators

The evaluation of the performance of Scala high-order iterative methods is beyond the scope of this book. However, it is important to be aware of the trade-off of each method.

The `for` construct is to be avoided as a counting iterator. It is designed to implement the for-comprehensive monad (`map` and `flatMap`). The source code presented in this book uses the high-order `foreach` method instead.

Let's kick the tires

This final section introduces the key elements of the training and classification workflow. A test case using a simple logistic regression is used to illustrate each step of the computational workflow.

An overview of computational workflows

In its simplest form, a computational workflow to perform runtime processing of a dataset is composed of the following stages:

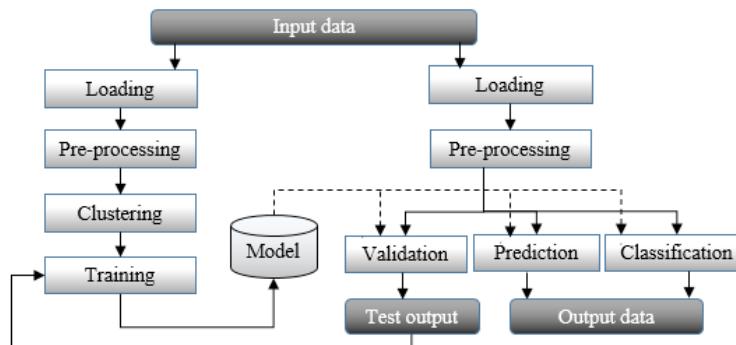
1. Loading the dataset from files, databases, or any streaming devices.
2. Splitting the dataset for parallel data processing.
3. Preprocessing data using filtering techniques, analysis of variance, and applying penalty and normalization functions whenever necessary.
4. Applying the model – either a set of clusters or classes – to classify new data.
5. Assessing the quality of the model.

A similar sequence of tasks is used to extract a model from a training dataset:

1. Loading the dataset from files, databases, or any streaming devices.
2. Splitting the dataset for parallel data processing.
3. Applying filtering techniques, analysis of variance, and penalty and normalization functions to the raw dataset whenever necessary.
4. Selecting the training, testing, and validation set from the cleansed input data.
5. Extracting key features and establishing affinity between a similar group of observations using clustering techniques or supervised learning algorithms.
6. Reducing the number of features to a manageable set of attributes to avoid overfitting the training set.

7. Validating the model and tuning the model by iterating steps 5, 6, and 7 until the error meets a predefined convergence criteria.
8. Storing the model in a file or database so that it can be applied to future observations.

Data clustering and data classification can be performed independent of each other or as part of a workflow that uses clustering techniques at the preprocessing stage of the training phase of a supervised learning algorithm. Data clustering does not require a model to be extracted from a training set, while classification can be performed only if a model has been built from the training set. The following image gives an overview of training, classification, and validation:



A generic data flow for training and running a model

The preceding diagram is an overview of a typical data mining processing pipeline. The first phase consists of extracting the model through clustering or training of a supervised learning algorithm. The model is then validated against test data for which the source is the same as the training set but with different observations. Once the model is created and validated, it can be used to classify real-time data or predict future behavior. Real-world workflows are more complex and require dynamic configuration to allow experimentation of different models. Several alternative classifiers can be used to perform a regression and different filtering algorithms are applied against input data, depending on the latent noise in the raw data.

Writing a simple workflow

This book relies on financial data to experiment with different learning strategies. The objective of the exercise is to build a model that can discriminate between volatile and nonvolatile trading sessions for stock or commodities. For the first example, we select a simplified version of the binomial logistic regression as our classifier as we treat stock-price-volume action as a continuous or pseudo-continuous process.

An introduction to the logistic regression



Logistic regression is explained in depth in the *Logistic regression* section in *Chapter 6, Regression and Regularization*. The model treated in this example is the simple binomial logistic regression classifier for two-dimension observations.

The steps for classification of trading sessions according to their volatility and volume is as follows:

1. Scoping the problem
2. Loading data
3. Preprocessing raw data
4. Discovering patterns, whenever possible
5. Implementing the classifier
6. Evaluating the model

Step 1 – scoping the problem

The objective is to create a model for stock price using its daily trading volume and volatility. Throughout the book, we will rely on financial data to evaluate and discuss the merits of different data processing and machine learning methods. In this example, the data is extracted from **Yahoo Finances** using the CSV format with the following fields:

- Date
- Price at open
- Highest price in the session
- Lowest price in the session
- Price at session close
- Volume
- Adjust price at session close

The `YahooFinancials` enumerator extracts the historical daily trading information from the Yahoo finance site:

```
type Fields = Array[String]
object YahooFinancials extends Enumeration {
    type YahooFinancials = Value
```

```

val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, ADJ_CLOSE = Value

deftoDouble(v: Value): Fields => Double = //1
(s: Fields) => s(v.id).toDouble
deftoDblArray(vs: Array[Value]): Fields => DblArray = //2
(s: Fields) => vs.map(v => s(v.id).toDouble)

...
}

```

The `toDouble` method converts an array of string into a single value (line 1) and `toDblArray` converts an array of string into an array of values (line 2). The `YahooFinancials` enumerator is described in the *Data sources* section in *Appendix A, Basic Concepts* in detail.

Let's create a simple program that loads the content of the file, executes some simple preprocessing functions, and creates a simple model. We selected the CSCO stock price between January 1, 2012 and December 1, 2013 as our data input.

Let's consider the two variables, *price* and *volume*, as shown in the following screenshot. The top graph displays the variation of the price of Cisco stock over time and the bottom bar chart represents the daily trading volume on Cisco stock over time:



Price-Volume action for Cisco stock 2012-2013

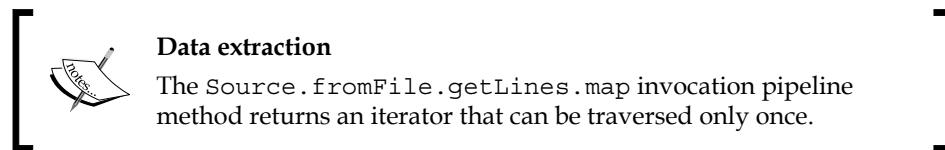
Step 2 – loading data

The second step is loading the dataset from a local or remote data storage.

Typically, large datasets are loaded from a database or distributed filesystems such as **Hadoop Distributed File System (HDFS)**. The `load` method takes an absolute pathname, extract, and transforms the input data from a file into a time series of a `Vector[DblPair]` type:

```
def load(fileName: String): Try[Vector[DblPair]] = Try {  
    val src = Source.fromFile(fileName) //3  
    val data = extract(src.getLines.map(_.split(",")).drop(1)) //4  
    src.close //5  
    data  
}
```

The data file is extracted through an invocation of the `Source.fromFile` static method (line 3), and then the fields are extracted through a map before the header (first row in the file) is removed using `drop` (line 4). The file has to be closed to avoid leaking of the file handle (line 5).



The purpose of the `extract` method is to generate a time series of two variables (*relative stock volatility* and *relative stock daily trading volume*):

```
def extract(cols: Iterator[Array[String]]): Xvseries[Double] = {  
    val features = Array[YahooFinancials](LOW, HIGH, VOLUME) //6  
    val conversion = YahooFinancials.toDblArray(features) //7  
    cols.map(c => conversion(c)).toVector  
        .map(x => Array[Double](1.0 - x(0)/x(1), x(2))) //8  
}
```

The only purpose of the `extract` method is to convert the raw textual data into a two-dimensional time series. The first step consists of selecting the three features to extract `LOW` (the lowest stock price in the session), `HIGH` (the highest price in the session), and `VOLUME` (trading volume for the session) (line 6). This feature set is used to convert each line of fields into a corresponding set of three values (line 7). Finally, the feature set is reduced to the following two variables (line 8):

- Relative volatility of the stock price in a session: $1.0 - \text{LOW}/\text{HIGH}$
- Trading volume for the stock in the session: `VOLUME`

Code readability

A long pipeline of Scala high-order methods make the code and underlying code quite difficult to read. It is recommended that you break down long chains of method calls, such as the following:

```
val cols = Source.fromFile.getLines.map(_.split(",")).  
toArray.drop(1)
```

We can break down method calls into several steps as follows:

```
val lines = Source.fromFile.getLines  
val fields = lines.map(_.split(",")).toArray  
val cols = fields.drop(1)
```

We strongly encourage you to consult the excellent guide *Effective Scala*, written by Marius Eriksen from Twitter. This is definitively a must read for any Scala developer [1:10].

Step 3 – preprocessing the data

The next step is to normalize the data in the range $[0.0, 1.0]$ to be trained by the binomial logistic regression. It is time to introduce an immutable and flexible normalization class.

Immutable normalization

The logistic regression relies on the sigmoid curve or logistic function is described in the *Logistic function* section in *Chapter 6, Regression and Regularization*. The logistic functions are used to segregate training data into classes. The output value of the logistic function ranges from 0 for $x = -\text{INFINITY}$ to 1 for $x = +\text{INFINITY}$. Therefore, it makes sense to normalize the input data or observation over $[0, 1]$.

Normalize or not normalize?

The purpose of normalizing data is to impose a single range of values for all the features, so the model does not favor any particular feature.

Normalization techniques include linear normalization and Z-score.

Normalization is an expensive operation that is not always needed.

The normalization is a linear transformation of the raw data that can be generalized to any range $[l, h]$.

Linear normalization

M2: [0, 1] Normalization of features $\{x_i\}$ with minimum x_{min} and maximum x_{max} values:



$$y_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

M3: [l, h] Normalization of features $\{x_i\}$:

$$y_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} (h - l) + l$$

The normalization of input data in supervised learning has a specific requirement: the classification and prediction of new observations have to use the normalization parameters (*min* and *max*) extracted from the training set, so all the observations share the same scaling factor.

Let's define the `MinMax` normalization class. The class is immutable: the minimum, `min`, and maximum, `max`, values are computed within the constructor. The class takes a time series of a parameterized `T` type and values as arguments (line 8). The steps of the normalization process are defined as follows:

1. Initialize the minimum values for a given time series during instantiation (line 9).
2. Compute the normalization parameters (line 10) and normalize the input data (line 11).
3. Normalize any new data points reusing the normalization parameters (line 14):

```
class MinMax[T <: AnyVal] (val values: XSeries[T]) (f : T =>
Double) { //8
    val zero = (Double.MaxValue, -Double.MaxValue)
    val minMax = values./:(zero)((mM, x) => { //9
        val min = mM._1
        val max = mM._2
        (if(x < min) x else min, if(x > max) x else max)
    })
    case class ScaleFactors(low:Double ,high:Double, ratio: Double)
    var scaleFactors: Option[ScaleFactors] = None //10

    def min = minMax._1
    def max = minMax._2
    def normalize(low: Double, high: Double): DblVector //11
    def normalize(value: Double): Double
}
```

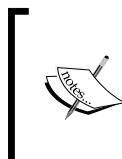
The class constructor computes the tuple of minimum and maximum values, `minMax`, using a fold (line 9). The `scaleFactors` scaling parameters are computed during the normalization of the time series (line 11), which are described as follows. The `normalize` method initializes the scaling factor parameters (line 12) before normalizing the input data (line 13):

```
def normalize(low: Double, high: Double): DblVector =  
    setScaleFactors(low, high).map( scale => { //12  
        values.map(x =>(x - min)*scale.ratio + scale.low) //13  
    }).getOrElse(/* ... */)  
  
def setScaleFactors(l: Double, h: Double): Option[ScaleFactors] ={  
    // .. error handling code  
    Some(ScaleFactors(l, h, (h - l)/(max - min))  
}
```

Subsequent observations use the same scaling factors extracted from the input time series in `normalize` (line 14):

```
def normalize(value: Double):Double = setScaleFactors.map(scale =>  
    if(value < min) scale.low  
    else if (value > max) scale.high  
    else (value - min)* scale.high + scale.low  
).getOrElse( /* ... */)
```

The `MinMax` class normalizes single variable observations.



The statistics class

The class that extracts the basic statistics from a `Stats` dataset, which is introduced in the *Profiling data* section in *Chapter 2, Hello World!*, inherits the `MinMax` class.

The test case with the binomial logistic regression uses a multiple variable normalization, implemented by the `MinMaxVector` class, which takes observations of the `XVSeries[Double]` type as inputs:

```
class MinMaxVector(series: XVSeries[Double]) {  
    val minMaxVector: Vector[MinMax[Double]] = //15  
        series.transpose.map(new MinMax[Double](_)  
    def normalize(low: Double, high: Double): XVSeries[Double]  
}
```

The constructor of the `MinMaxVector` class transposes the vector of array of observations in order to compute the minimum and maximum value for each dimension (line 15).

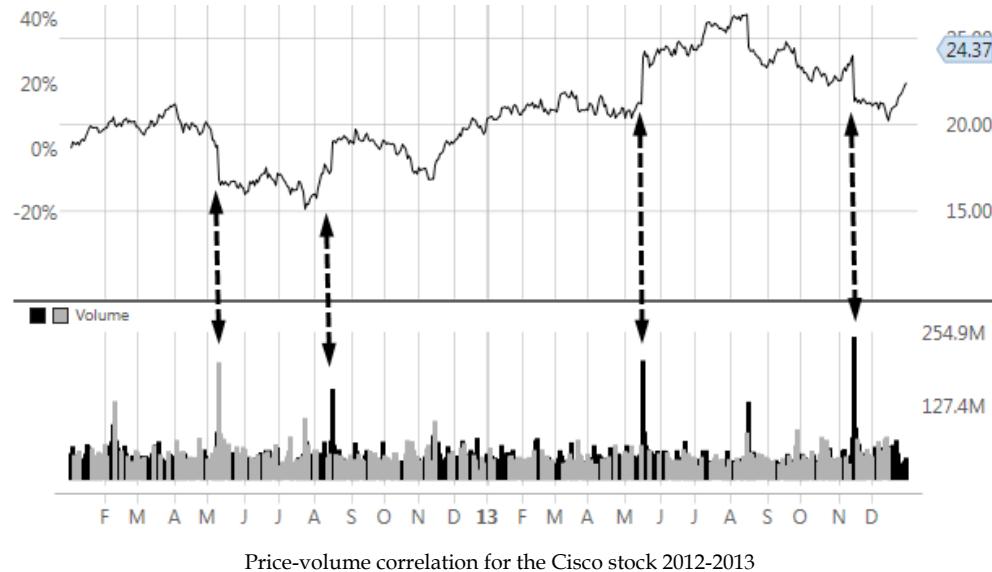
Step 4 – discovering patterns

The price action chart has a very interesting characteristic.

Analyzing data

At a closer look, a sudden change in price and increase in volume occurs about every three months or so. Experienced investors will undoubtedly recognize that these price-volume patterns are related to the release of quarterly earnings of Cisco. Such a regular but unpredictable pattern can be a source of concern or opportunity if risk can be properly managed. The strong reaction of the stock price to the release of corporate earnings may scare some long-term investors while enticing day traders.

The following graph visualizes the potential correlation between sudden price change (volatility) and heavy trading volume:



The next section is not required for the understanding of the test case. It illustrates the capabilities of JFreeChart as a simple visualization and plotting library.

Plotting data

Although charting is not the primary goal of this book, we thought that you will benefit from a brief introduction to JFreeChart.



Plotting classes

This section illustrates a simple Scala interface to JFreeChart Java classes. Reading this is not required for the understanding of machine learning. The visualization of the results of a computation is beyond the scope of this book.

Some of the classes used in visualization are described in the *Appendix A, Basic Concepts*.

The dataset (volatility and volume) is converted into internal JFreeChart data structures. The `ScatterPlot` class implements a simple configurable scatter plot with the following arguments:

- `config`: This includes information, labels, fonts, and so on, of the plot
- `theme`: This is the predefined theme for the plot (black, white background, and so on)

The code will be as follows:

```
class ScatterPlot(config: PlotInfo, theme: PlotTheme) { //16
    def display(xy: Vector[DblPair], width: Int, height) //17
    def display(xt: XVSeries[Double], width: Int, height)
    // ...
}
```

The `PlotTheme` class defines a specific theme or preconfiguration of the chart (line 16). The class offers a set of `display` methods to accommodate a wide range of data structures and configuration (line 17).



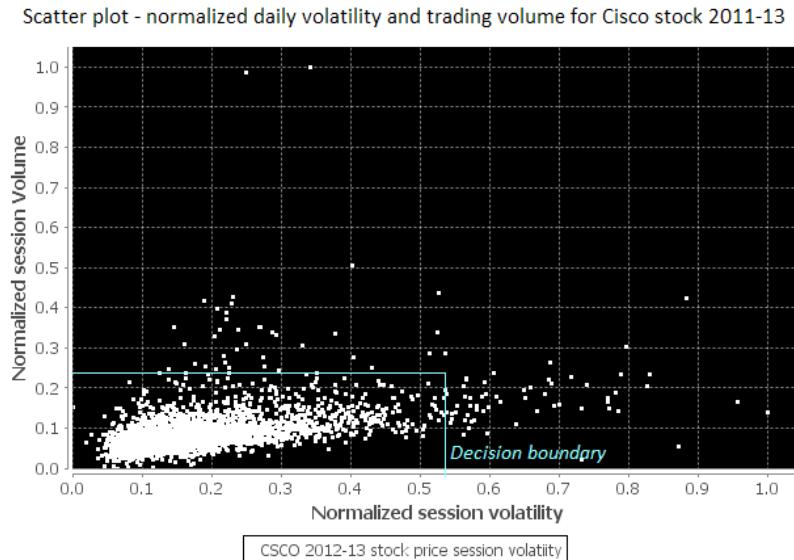
Visualization

The JFreeChart library is introduced as a robust charting tool. The code related to plots and charts is omitted from the book in order to keep the code snippets concise and dedicated to machine learning. On a few occasions, output data is formatted as a CSV file to be imported into a spreadsheet.

Getting Started

The `ScatterPlot.display` method is used to display the normalized input data used in the binomial logistic regression as follows:

```
val plot = new ScatterPlot(("CSCO 2012-2013",
    "Session High - Low", "Session Volume"), new BlackPlotTheme)
plot.display(volatility_vol, 250, 340)
```



A scatter plot of volatility and volume for the Cisco stock 2012-2013

The scatter plot shows a level of correlation between session volume and session volatility and confirms the initial finding in the stock price and volume chart. We can leverage this information to classify trading sessions by their volatility and volume. The next step is to create a two class model by loading a training set, observations, and expected values, into our logistic regression algorithm. The classes are delimited by a **decision boundary** (also known as a hyperplane) drawn on the scatter plot.

Visualizing labels – the normalized variation of the stock price between the opening and closing of the trading session is selected as the label for this classifier.

Step 5 – implementing the classifier

The objective of this training is to build a model that can discriminate between volatile and nonvolatile trading sessions. For the sake of the exercise, session volatility is defined as the relative difference between the session highest price and lower price. The total trading volume within a session constitutes the second parameter of the model. The relative price movement within a trading session (that is, *closing price/open price - 1*) is our expected values or labels.

Logistic regression is commonly used in statistics inference.

M4: Logistic regression model



$$f(\mathbf{x}|\mathbf{w}) = w_0 + \sum_{i=1}^{N-1} x_i w_i \quad l(\mathbf{x}|\mathbf{w}) = \frac{1}{1 + e^{-f(\mathbf{x}|\mathbf{w})}}$$

The first weight w_0 is known as the intercept. The binomial logistic regression is described in the *Logistic regression* section in *Chapter 6, Regression and Regularization*, in detail.

The following implementation of the binomial logistic regression classifier exposes a single `classify` method to comply with our desire to reduce the complexity and life cycle of objects. The model weights parameters are computed during training when the `LogBinRegression` class/model is instantiated. As mentioned earlier, the sections of the code nonessential to the understanding of the algorithm are omitted.

The `LogBinRegression` constructor has five arguments (line 18):

- `obsSet`: These are vector observations that represent volume and volatility
- `expected`: This is a vector of expected values
- `maxIters`: This is the maximum number of iterations allowed for the optimizer to extract the regression weights during training
- `eta`: This is the learning or training rate
- `eps`: This is the maximum value of the error (*predicted – expected*) for which the model is valid

The code is as follows:

```
class LogBinRegression(
    obsSet: Vector[DblArray],
    expected: Vector[Int],
    maxIters: Int,
    eta: Double,
    eps: Double) { //18

    val model: LogBinRegressionModel = train //19
    def classify(obs: DblArray): Try[(Int, Double)] //20
    def train: LogBinRegressionModel
    def intercept(weights: DblArray): Double
    ...
}
```

The `LogBinRegressionModel` model is generated through training during the instantiation of the `LogBinRegression` logistic regression class (line 19):

```
case class LogBinRegressionModel(val weights: DblArray)
```

The model is fully defined by its weights, as described in the mathematical formula **M3**. The `weights(0)` intercept represents the mean value of the prediction for observations for which variables are zero. The intercept does not have any specific meaning for most of the cases and it is not always computable.

Intercept or not intercept?

The intercept corresponds to the value of weights when the observations have null values. It is a common practice to estimate, whenever possible, the intercept for binomial linear or logistic regression independently from the slope of the model in the minimization of the error function. The multinomial regression models treat the intercept or weight w_0 as part of the regression model, as described in the *Ordinary least squares regression* section of *Chapter 6, Regression and Regularization*.

The code will be as follows:

```
def intercept(weights: DblArray): Double = {
    val zeroObs = obsSet.filter(!_.exists(_ > 0.01))
    if(zeroObs.size > 0)
        zeroObs.aggregate(0.0)((s, z) => s + dot(z, weights),
            _ + _) / zeroObs.size
    else 0.0
}
```

The `classify` methods takes new observations as inputs and compute the index of the classes (0 or 1) the observations belong to and the actual likelihood (line 20).

Selecting an optimizer

The goal of the training of a model using expected values is to compute the optimal weights that minimizes the **error or cost function**. We select the **batch gradient descent** algorithm to minimize the cumulative error between the predicted and expected values for all the observations. Although there are quite a few alternative optimizers, the gradient descent is quite robust and simple enough for this first chapter. The algorithm consists of updating the weights w_i of the regression model by minimizing the cost.

Cost function

M5: Cost (or *compound error* = *predicted* – *expected*):

$$\text{cost} = \frac{1}{2N} \sum_{j=0}^{N-1} (l(x_j|w_i) - y_j)^2$$

 M6: The batch gradient descent method to update model weights w_i is as follows:

$$w'_i = w_i + \frac{\partial \text{cost}}{\partial w_i} = w_i + \frac{1}{N} \sum_{j=0}^{N-1} (l(x_j|w_i) - y_j) \cdot x_j$$

For those interested in learning about of optimization techniques, the *Summary of optimization techniques* section in the *Appendix A, Basic Concepts* presents an overview of the most commonly used optimizers. The batch descent gradient method is also used for the training of the multilayer perceptron (refer to *The training epoch* section under *The multilayer perceptron* section in *Chapter 9, Artificial Neural Networks*).

The execution of the batch gradient descent algorithm follows these steps:

1. Initialize the weights of the regression model.
2. Shuffle the order of observations and expected values.
3. Aggregate the cost or error for the entire observation set.
4. Update the model weights using the cost as the objective function.
5. Repeat from step 2 until either the maximum number of iterations is reached or the incremental update of the cost is close to zero.

The purpose of **shuffling** the order of the observations between iterations is to avoid the minimization of the cost reaching a local minimum.

Batch and stochastic gradient descent

 The stochastic gradient descent is a variant of the gradient descent that updates the model weights after computing the error on each observation. Although the stochastic gradient descent requires a higher computation effort to process each observation, it converges toward the optimal value of weights fairly quickly after a small number of iterations. However, the stochastic gradient descent is sensitive to the initial value of the weights and the selection of the learning rate, which is usually defined by an adaptive formula.

Training the model

The `train` method consists of iterating through the computation of the weight using a simple descent gradient method. The method computes weights and returns an instance of the `LogBinRegressionModel` model:

```
def train: LogBinRegressionModel = {  
    val nWeights = obsSet.head.length + 1 //21  
    val init = Array.fill(nWeights) (Random.nextDouble) //22  
    val weights = gradientDescent(obsSet.zip(expected), 0.0, 0, init)  
    new LogBinRegressionModel(weights) //23  
}
```

The `train` method extracts the number of weights, `nWeights`, for the regression model as the *number of variables in each observation + 1* (line 21). The method initializes weights with random values over $[0, 1]$ (line 22). The weights are computed through the tail recursive `gradientDescent` method, and the method returns a new model for the binomial logistic regression (line 23).

Unwrapping values from Try

It is usually not recommended to invoke the `get` method to a `Try` value, unless it is enclosed in a `Try` statement. The best course of action is to do the following:

1. Catch the failure with `match{ case Success(m) => .. case Failure(e) => }`
2. Extract the `getOrElse(/* ... */)` result safely
3. Propagate the results as a `Try` type `map(_ .m)`

Let's take a look at the computation for weights through the minimization of the cost function in the `gradientDescent` method:

```
type LabelObs = Vector[(DblArray, Int)]  
  
@tailrec  
def gradientDescent(  
    obsAndLbl: LabelObs,  
    cost: Double,  
    nIters: Int,  
    weights: DblArray): DblArray = { //24
```

```

if(nIters >= maxIters)
    throw new IllegalStateException(..)//25
val shuffled = shuffle(obsAndLbl) //26
val errorGrad = shuffled.map{ case(x, y) => { //27
    val error = sigmoid(dot(x, weights)) - y
    (error, x.map(_ * error)) //28
} }.unzip

val scale = 0.5/obsAndLbl.size
val newCost = errorGrad._1 //29
.aggregate(0.0)((s,c) =>s + c*c, _ + _) *scale
val relativeError = cost/newCost - 1.0

if( Math.abs(relativeError) < eps) weights //30
else {
    val derivatives = Vector[Double](1.0) ++
        errorGrad._2.transpose.map(_.sum) //31
    val newWeights = weights.zip(derivatives)
        .map{ case (w, df) => w - eta*df} //32
    newWeights.copyToArray(weights)
    gradientDescent(shuffled, newCost, nIters+1, newWeights)//33
}
}
}

```

The `gradientDescent` method recurses on the vector of pairs (observations and expected values), `obsAndLbl`, `cost`, and the model weights (line 24). It throws an exception if the maximum number of iterations allowed for the optimization is reached (line 25). It shuffles the order of the observations (line 26) before computing the `errorGrad` derivatives of the cost over each weights (line 27). The computation of the derivative of the cost (or $error = predicted\ value - expected\ value$) in formula **M5** returns a pair of cumulative cost and derivative values using the formula (line 28).

Next, the method computes the overall compound cost using the formula **M4** (line 29), converts it to a relative incremental `relativeError` cost that is compared to the `eps` convergence criteria (line 30). The method extracts derivatives of cost over weights by transposing the matrix of errors, and then prepends the bias `1.0` value to match the array of weights (line 31).

Bias value

The purpose of the bias value is to prepend 1.0 to the vector of observation so it can be directly processed (for example, zip and dot) with the weights. For instance, a regression model for two-dimensional observations (x, y) has three weights (w_0, w_1, w_2). The bias value +1 is prepended to the observations to compute the predicted value 1.0: $w_0 + x.w_1 + y.w_2$.

This technique is used in the computation of the activation function of the multilayer perceptron, as described in the *The multilayer perceptron* section in *Chapter 9, Artificial Neural Networks*.



The formula **M6** updates the weights for the next iteration (line 32) before invoking the method with new weights, cost, and iteration count (line 33).

Let's take a look at the shuffling of the order of observations using a random sequence generator. The following implementation is an alternative to the Scala standard library method `scala.util.Random.shuffle` for shuffling elements of collections. The purpose is to change the order of observations and labels between iterations in order to prevent the optimizer to reach a local minimum. The `shuffle` method permutes the order in the `labelObs` vector of observations by partitioning it into segments of random size and reversing the order of the other segment:

```
val SPAN = 5
def shuffle(labelObs: LabelObs): LabelObs = {
    shuffle(new ArrayBuffer[Int](0, 0).map(labelObs(_))) //34
}
```

Once the order of the observations is updated, the vector of pair (observations, labels) is easily built through a map (line 34). The actual shuffling of the index is performed in the following `shuffle` recursive function:

```
val maxChunkSize = Random.nextInt(SPAN)+2 //35

@tailrec
def shuffle(indices: ArrayBuffer[Int], count: Int, start: Int):
    Array[Int] = {
    val end = start + Random.nextInt(maxChunkSize) //36
    val isOdd = ((count & 0x01) != 0x01)
    if(end >= sz)
        indices.toArray ++ slice(isOdd, start, sz) //37
    else
        shuffle(indices ++ slice(isOdd, start, end), count+1, end)
}
```

The maximum size of partition of the `maxChunkSize` vector observations is randomly computed (line 35). The method extracts the next slice (`start, end`) (line 36). The slice is either added to the existing indices vector and returned once all the observations have been shuffled (line 37) or passed to the next invocation.

The `slice` method returns an array of indices over the range (`start, end`) either in the right order if the number of segments processed is odd, or in reverse order if the number of segment processed is even:

```
def slice(isOdd: Boolean, start: Int, end: Int): Array[Int] = {
    val r = Range(start, end).toArray
    (if(isOdd) r else r.reverse)
}
```

Iterative versus tail recursive computation

The tail recursion in Scala is a very efficient alternative to the iterative algorithm. Tail recursion avoids the need to create a new stack frame for each invocation of the method. It is applied to the implementation of many machine learning algorithms presented throughout the book.

In order to train the model, we need to label the input data. The labeling process consists of associating the relative price movement during a session (price at *close*/*price at open - 1*) with one of the following two configurations:

- Volatile trading sessions with high trading volume
- Trading sessions with low volatility and low trading volume

The two classes of training observations are segregated by a decision boundary drawn on the scatter plot in the previous section. The labeling process is usually quite cumbersome and should be automated as much as possible.

Automated labeling

Although quite convenient, automated creation of training labels is not without risk as it may mislabel singular observations. This technique is used in this test for convenience, but it is not recommended unless a domain expert reviews the labels manually.

Classifying observations

Once the model is successfully created through training, it is available to classify new observation. The runtime classification of observations using the binomial logistic regression is implemented by the `classify` method:

```
def classify(obs: DblArray): Try[(Int, Double)] =  
    val linear = dot(obs, model.weights) //37  
    val prediction = sigmoid(linear)  
    (if(linear > 0.0) 1 else 0, prediction) //38  
}
```

The method applies the logistic function to the linear inner product, `linear`, of the new `obs` and `weights` observations of the model (line 37). The method returns the tuple (the predicted class of the observation {0, 1}, prediction value) where the class is defined by comparing the prediction to the boundary value 0.0 (line 38).

The computation of the `dot` product of weights and observations uses the bias value as follows:

```
def dot(obs: DblArray, weights: DblArray): Double =  
    weights.zip(Array[Double](1.0) ++ obs)  
    .aggregate(0.0){case (s, (w,x)) => s + w*x, _ + _ }
```

The alternative implementation of the `dot` product of weights and observations consists of extracting the first `w.head` weight:

```
def dot(x: DblArray, w: DblArray): Double =  
    x.zip(w.drop(1)).map {case (_x,_w) => _x*_w}.sum + w.head
```

The `dot` method is used in the `classify` method.

Step 6 – evaluating the model

The first step is to define the configuration parameters for the test: the maximum number of `NITERS` iterations, the `EPS` convergence criteria, the `ETA` learning rate, the decision boundary used to label the `BOUNDARY` training observations, and the path to the training and test sets:

```
val NITERS = 800; val EPS = 0.02; val ETA = 0.0001  
val path_training = "resources/data/chap1/CSCO.csv"  
val path_test = "resources/data/chap1/CSCO2.csv"
```

The various activities of creating and testing the model, loading, normalizing data, training the model, loading, and classifying test data is organized as a workflow using the monadic composition of the Try class:

```

for {
    volatilityVol <- load(path_training)      //39
    minMaxVec <- Try(new MinMaxVector(volatilityVol))    //40
    normVolatilityVol <- Try(minMaxVec.normalize(0.0,1.0))//41
    classifier <- logRegr(normVolatilityVol)      //42
    testValues <- load(path_test)      //43
    normTestValue0 <- minMaxVec.normalize(testValues(0))  //44
    class0 <- classifier.classify(normTestValue0)      //45
    normTestValue1 <- minMaxVec.normalize(testValues(1))
    class1 <- classifier.classify(normTestValue1)
} yield {
    val modelStr = model.toString
    ...
}

```

First, the daily trading volatility and volume for the `volatilityVol` stock price is loaded from file (line 39). The workflow initializes the multi-dimensional `MinMaxVec` normalizer (line 40) and uses it to normalize the training set (line 41). The `logRegr` method instantiates the binomial `classifier` logistic regression (line 42). The `testValues` test data is loaded from file (line 43), normalized using `MinMaxVec` already applied to the training data (line 44), and classified (line 45).

The `load` method extracts data (observations) of a `XVSeries[Double]` type from the file. The heavy lifting is done by the `extract` method (line 46), and then the file handle is closed (line 47) before returning the vector of raw observations:

```

def load(fileName: String): Try[XVSeries[Double], XSeries[Double]] = {
    val src = Source.fromFile(fileName)
    val data = extract(src.getLines.map(_.split(",")).drop(1)) //46
    src.close; data //47
}

```

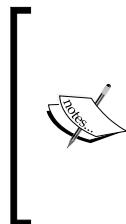
The private `logRegr` method has the following two purposes:

- Labeling automatically the `obs` observations to generate the expected values (line 48)
- Initializing (instantiation and training of the model) the binomial logistic regression (line 49)

The code is as follows:

```
def logRegr(obs: XVSeries[Double]): Try[LogBinRegression] = Try {  
    val expected = normalize(labels._2).get //48  
    new LogBinRegression(obs, expected, NITERS, ETA, EPS) //49  
}
```

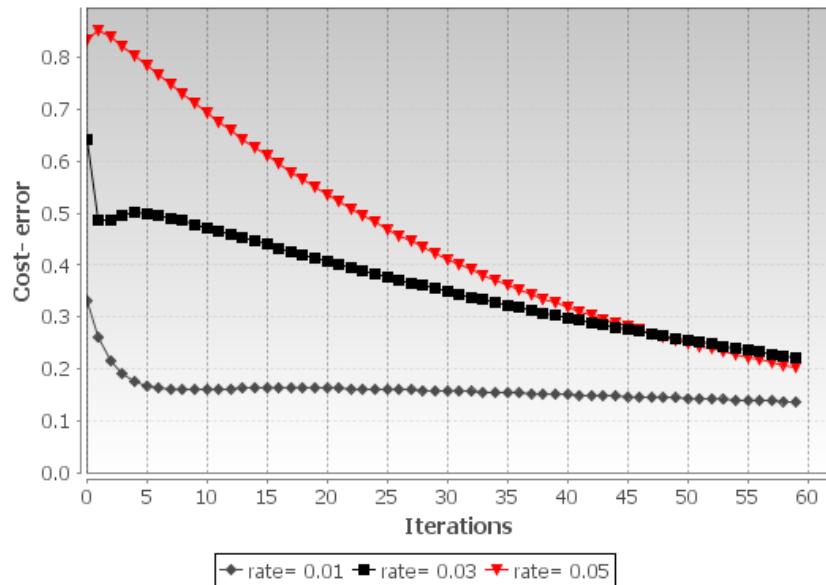
The method labels observations by evaluating if they belong to any one of the two classes delimited by the BOUNDARY condition, as illustrated in the scatter plot in a previous section.



Validation

The simple classification in this test case is provided for illustrating the runtime application of the model. It does not constitute a validation of the model by any stretch of imagination. The next chapter digs into validation methodologies (refer to the *Assessing a model* section in *Chapter 2, Hello World!*)

The training run is performed with three different values of the learning rate. The following chart illustrates the convergence of the batch gradient descent in the minimization of the cost, given different values of learning rates:



Impact of the learning rate on the batch gradient descent on the convergence of the cost (error)

As expected, the execution of the optimizer with a higher learning rate produces a steepest descent in the cost function.

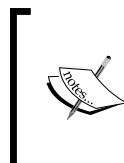
The execution of the test produces the following model:

iters = 495

weights: 0.859-3.6177923,-64.927832

input (0.0088, 4.10E7) normalized (0.063,0.061) class 1 prediction 0.515

input (0.0694, 3.68E8) normalized (0.517,0.641) class 0 prediction 0.001



Learning more about regressive models

The binomial logistic regression is merely used to illustrate the concept of training and prediction. It is described in the *Logistic regression* section in *Chapter 6, Regression and Regularization* in detail.

Summary

I hope you enjoyed this introduction to machine learning. You learned how to leverage your skills in Scala programming to create a simple logistic regression program for predicting stock price/volume action. Here are the highlights of this introductory chapter:

- From monadic composition and high order collection methods for parallelization to configurability and reusability patterns, Scala is the perfect fit to implement data mining and machine learning algorithms for large-scale projects.
- There are many logical steps to create and deploy a machine learning model.
- The implementation of the binomial logistic regression classifier presented as part of the test case is simple enough to encourage you to learn how to write and apply more advanced machine learning algorithms.

To the delight of Scala programming aficionados, the next chapter will dig deeper into building a flexible workflow by leveraging monadic data transformation and stackable traits.

2

Hello World!

In the first chapter, you were acquainted with some rudimentary concepts regarding data processing, clustering, and classification. This chapter is dedicated to the creation and maintenance of a flexible end-to-end workflow to train and classify data. The first section of the chapter introduces a data-centric (functional) approach to create number-crunching applications.

You will learn how to:

- Apply the concept of monadic design to create dynamic workflows
- Leverage some of Scala's advanced patterns, such as the cake pattern, to build portable computational workflows
- Take into account the bias-variance trade-off in selecting a model
- Overcome overfitting in modeling
- Break down data into training, test, and validation sets
- Implement model validation in Scala using precision, recall, and F score

Modeling

Data is the lifeline of any scientist, and the selection of data providers is critical in developing or evaluating any statistical inference or machine learning algorithm.

A model by any other name

We briefly introduced the concept of a **model** in the *Model categorization* section in *Chapter 1, Getting Started*.

What constitutes a model? Wikipedia provides a reasonably good definition of a model as understood by scientists [2:1]:

A scientific model seeks to represent empirical objects, phenomena, and physical processes in a logical and objective way.

...

Models that are rendered in software allow scientists to leverage computational power to simulate, visualize, manipulate and gain intuition about the entity, phenomenon or process being represented.

In statistics and the probabilistic theory, a model describes data that one might observe from a system to express any form of uncertainty and noise. A model allows us to infer rules, make predictions, and learn from data.

A model is composed of **features**, also known as **attributes** or **variables**, and a set of relation between those features. For instance, the model represented by the function $f(x, y) = x \cdot \sin(2y)$ has two features, x and y , and a relation, f . Those two features are assumed to be independent. If the model is subject to a constraint such as $f(x, y) < 20$, then the **conditional independence** is no longer valid.

An astute Scala programmer would associate a model to a monoid for which the set is a group of observations and the operator is the function implementing the model.

Models come in a variety of shapes and forms:

- **Parametric:** This consists of functions and equations (for example, $y = \sin(2t + w)$)
- **Differential:** This consists of ordinary and partial differential equations (for example, $dy = 2x.dx$)
- **Probabilistic:** This consists of probability distributions (for example, $p(x | c) = \exp(k \cdot \log x - x)/x!$)
- **Graphical:** This consists of graphs that abstract out the conditional independence between variables (for example, $p(x, y | c) = p(x | c) \cdot p(y | c)$)
- **Directed graphs:** This consists of temporal and spatial relationships (for example, a scheduler)
- **Numerical method:** This consists of computational methods such as finite difference, finite elements, or Newton-Raphson
- **Chemistry:** This consists of formula and components (for example, H_2O , $Fe + C_{12} = FeC_{13}$, and so on)

- **Taxonomy:** This consists of a semantic definition and relationship of concepts (for example, *APG/Eudicots/Rosids/Huaceae/Malvales*)
- **Grammar and lexicon:** This consists of a syntactic representation of documents (for example, the Scala programming language)
- **Inference logic:** This consists of rules (for example, *IF (stock vol > 1.5 * average) AND rsi > 80 THEN ...*)

Model versus design

The confusion between a model and design is quite common in computer science, the reason being that these terms have different meanings for different people depending on the subject. The following metaphors should help with your understanding of these two concepts:

- **Modeling:** This describes something you know. A model makes an assumption, which becomes an assertion if proven correct (for example, the US population, p , increases by 1.2 percent a year, $dp/dt = 1.012$).
- **Designing:** This manipulates the representation of things you don't know. Designing can be regarded as the exploration phase of modeling (for example, what are the features that contribute to the growth of the US population? Birth rate? Immigration? Economic conditions? Social policies?).

Selecting features

The selection of a model's features is the process of discovering and documenting the minimum set of variables required to build the model. Scientists assume that data contains many redundant or irrelevant features. Redundant features do not provide information already given by the selected features, and irrelevant features provide no useful information.

A **features selection** consists of two consecutive steps:

1. Searching for new feature subsets.
2. Evaluating these feature subsets using a scoring mechanism.

The process of evaluating each possible subset of features to find the one that maximizes the objective function or minimizes the error rate is computationally intractable for large datasets. A model with n features requires $2^n - 1$ evaluations.

Extracting features

An **observation** is a set of indirect measurements of hidden, also known as latent, variables, which may be noisy or contain a high degree of correlation and redundancies. Using raw observations in a classification task would very likely produce inaccurate results. Using all features in each observation also incurs a high computation cost.

The purpose of **features extraction** is to reduce the number of variables or dimensions of the model by eliminating redundant or irrelevant features. The features are extracted by transforming the original set of observations into a smaller set at the risk of losing some vital information embedded in the original set.

Defining a methodology

A data scientist has many options in selecting and implementing a classification or clustering algorithm.

Firstly, a mathematical or statistical model is to be selected to extract knowledge from the raw input data or the output of a data upstream transformation. The selection of the model is constrained by the following parameters:

- Business requirements such as accuracy of results or computation time
- Availability of training data, algorithms, and libraries
- Access to a domain or subject matter expert, if needed

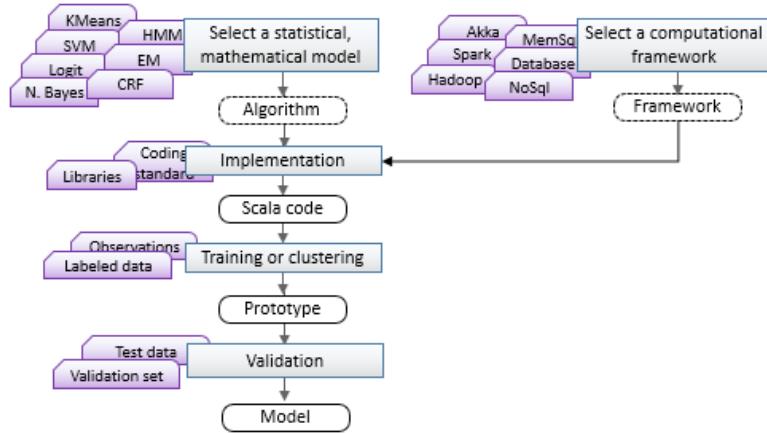
Secondly, the engineer has to select a computational and deployment framework suitable for the amount of data to be processed. The computational context is to be defined by the following parameters:

- Available resources such as machines, CPU, memory, or I/O bandwidth
- An implementation strategy such as iterative versus recursive computation or caching
- Requirements for the responsiveness of the overall process such as duration of computation or display of intermediate results

Thirdly, a domain expert has to tag or label the observations in order to generate an accurate classifier.

Finally, the model has to be validated against a reliable test dataset.

The following diagram illustrates the selection process to create a workflow:



Statistical and computation modeling for machine learning applications

Domain expertise, data science, and software engineering

A domain or subject matter expert is a person with authoritative or credited expertise in a particular area or topic. A chemist is an expert in the domain of chemistry and possibly related fields.

A data scientist solves problems related to data in a variety of fields, such as biological sciences, health care, marketing, or finances. Data and text mining, signal processing, statistical analysis, and modeling using machine learning algorithms are some of the activities performed by a data scientist.

A software developer performs all the tasks related to the creation of software applications, including analysis, design, coding, testing, and deployment.



The parameters of a data transformation may need to be reconfigured according to the output of the upstream data transformation. Scala's higher-order functions are particularly suitable for implementing configurable data transformations.

Monadic data transformation

The first step is to define a trait and method that describe the transformation of data by the computation units of a workflow. The data transformation is the foundation of any workflow for processing and classifying a dataset, training and validating a model, and displaying results.

There are two symbolic models used for defining a data processing or data transformation:

- **Explicit model:** The developer creates a model explicitly from a set of configuration parameters. Most of deterministic algorithms and unsupervised learning techniques use an explicit model.
- **Implicit model:** The developer provides a training set that is a set of labeled observations (observations with an expected outcome). A classifier extracts a model through the training set. Supervised learning techniques rely on models implicitly generated from labeled data.

Error handling

The simplest form of data transformation is **morphism** between the two `U` and `V` types. The data transformation enforces a *contract* for validating an input and returning either a value or an error. From now on, we use the following convention:

- **Input value:** The validation is implemented through a partial function of the `PartialFunction` type that is returned by the data transformation. A `MatchErr` error is thrown in case the input value does not meet the required condition (contract).
- **Output value:** The type of a return value is `Try[V]` for which an exception is returned in case of an error.

Reusability of partial functions

Reusability is another benefit of partial functions, which is illustrated in the following code snippet:



```
class F {  
    def f: PartialFunction[Int, Try[Double]] = case n:  
        Int ...  
    }  
    val pfn = (new F).f  
    pfn(4)  
    pfn(10)
```

Partial functions enable developers to implement methods that address the most common (primary) use cases for which input values have been tested. All other nontrivial use cases (or input values) generate a `MatchErr` exception. At a later stage in the development cycle, the developer can implement the code to handle the less common use cases.

Runtime validation of a partial function

It is a good practice to validate if a partial function is defined for a specific value of the argument:

```

for {
    pfn.isDefinedAt(input)
    value <- pfn(input)
} yield { ... }
```

This preemptive approach allows the developer to select an alternative method or a full function. It is an efficient alternative to catch a `MathErr` exception. The validation of a partial function is omitted throughout the book for the sake of clarity.

Therefore, the signature of a data transformation is defined as follows:

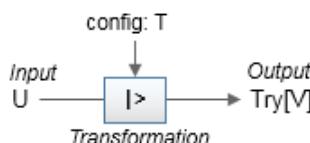
```
def |> : PartialFunction[U, Try[V]]
```

F# language references

The `|>` notation used as the signature of the transform is borrowed from the F# language [2:2].

Explicit models

The objective is to define a symbolic representation of the transformation of different types of data without exposing the internal state of the algorithm implementing the data transformation. The transformation on a dataset is performed using a model or configuration that is fully defined by the user, which is illustrated in the following diagram:



Visualization of explicit models

The transformation of an explicit configuration or model, `config`, is defined as an `ETransform` abstract class parameterized by the `T` type of the model:

```
abstract class ETransform[T] (val config: T) { //explicit model
    type U    // type of input
    type V    // type of output
    def |> : PartialFunction[U, Try[V]]    // data transformation
}
```

The input U type and output V type have to be defined in the subclasses of `ETransform`. The `|>` transform operator returns a partial function that can be reused for different input values.

The creation of a class that implements a specific transformation using an explicit configuration is quite simple: all you need is the definition of an input/output U/V type and an implementation of the `|>` transformation method.

Let's consider the extraction of data from a financial source, `DataSource`, that takes a list of functions that convert some text fields, `Fields`, into a `Double` value as the input and produce a list of observations of the `XSeries[Double]` type. The extraction parameters are defined in the `DataSourceConfig` class:

```
class DataSource(
    config: DataSourceConfig,      //1
    srcFilter: Option[Fields => Boolean] = None)
    extends ETransform[DataSourceConfig](config) { //2
    type U = List[Fields => Double]      //3
    type V = List[XSeries[Double]]        //4
    override def |> : PartialFunction[U, Try[V]] = { //5
        case u: U if (!u.isEmpty) => ...
    }
}
```

The `DataSourceConfig` configuration is explicitly provided as an argument of the constructor for `DataSource` (line 1). The constructor implements the basic type and data transformation associated with an explicit model (line 2). The class defines the U type of input values (line 3), V type of output values (line 4), and `|>` transformation method that returns a partial function (line 5).



The `DataSource` class

The *Data extraction* section of the *Appendix A, Basic Concepts* describes the `DataSource` class functionality. The `DataSource` class is used throughout the book.

Data transformations using an explicit model or configuration constitute a category with monadic operations. The monad associated with the `ETransform` class subclasses the definition of the higher kind, `_Monad`:

```
private val eTransformMonad = new _Monad[ETransform] {
    override def unit[T](t:T) = eTransform(t)      //6
    override def map[T,U](m: ETransform[T])         //7
        (f: T => U): ETransform[U] = eTransform( f(m.config) )
    override def flatMap[T,U](m: ETransform[T])     //8
        (f: T => ETransform[U]): ETransform[U] = f(m.config)
}
```

The singleton `eTransformMonad` implements the following basic monadic operators introduced in the *Monads* section under *Abstraction in Chapter 1, Getting Started*:

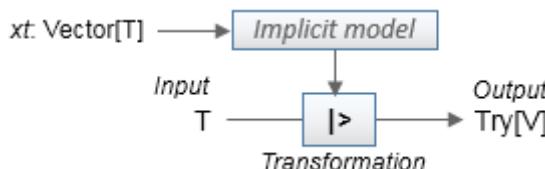
- The `unit` method is used to instantiate `ETransform` (line 6)
- The `map` is used to transform an `ETransform` object by morphing its elements (line 7)
- The `flatMap` is used to transform an `ETransform` object by instantiating its elements (line 8)

For practical purposes, an implicit class is created to convert an `ETransform` object to its associated monad, allowing transparent access to the `unit`, `map`, and `flatMap` methods:

```
implicit class eTransform2Monad[T] (fct: ETransform[T]) {
    def unit(t: T) = eTransformMonad.unit(t)
    final def map[U](f: T => U): ETransform[U] =
        eTransformMonad.map(fct)(f)
    final def flatMap[U](f: T => ETransform[U]): ETransform[U] =
        eTransformMonad.flatMap(fct)(f)
}
```

Implicit models

Supervised learning models are extracted from a training set. Transformations, such as classification or regression use the implicit models to process the input data, as illustrated in the following diagram:



Visualization of implicit models

The transformation for a model implicitly extracted from the training data is defined as an abstract `ITransform` class parameterized by the `T` type of observations, `xt`:

```
abstract class ITransform[T] (val xt: Vector[T]) { //Model input
    type V    // type of output
    def |> : PartialFunction[T, Try[V]] // data transformation
}
```

The type of the data collection is `Vector`, which is an immutable and effective container. An `ITransform` type is created by defining the `T` type of the observation, the `v` output of the data transformation, and the `|>` method that implements the transformation, usually a classification or regression. Let's consider the support vector machine algorithm, `SVM`, to illustrate the implementation of a data transformation using an implicit model:

```
class SVM[T <: AnyVal] ( //9
    config: SVMConfig,
    xt: Vector[Array[T]],
    expected: Vector[Double]) (implicit f: T => Double)
  extends ITransform[Array[T]] (xt) { //10

  type V = Double //11
  override def |> : PartialFunction[Array[T], Try[V]] = { //12
    case x: Array[T] if(x.length == data.size) => ...
  }
}
```

The support vector machine is a discriminative supervised learning algorithm described in *Chapter 8, Kernel Models and Support Vector Machines*. A support vector machine, `SVM`, is instantiated with a configuration and training set: the `xt` observations and `expected` data (line 9). Contrary to the explicit model, the `config` configuration does not define the model used in the data transformation; the model is implicitly generated from the training set of the `xt` input data and `expected` values. An `SVM` instance is created as an `ITransform` (line 10) by specifying the `V` output type (line 11) and overriding the `|>` transformation method (line 12).

The `|>` classification method produces a partial function that takes an `x` observation as an input and returns the prediction value of a `Double` type.

Similar to the explicit transformation, we define the monadic operation for the `ITransform` by overriding the `unit` (line 13), `map` (line 14), and `flatMap` (line 15) methods:

```
private val iTransformMonad = new _Monad[ITransform] {
  override def unit[T](t: T) = iTransform(Vector[T](t)) //13

  override def map[T,U](m: ITransform[T])(f: T => U):
    ITransform[U] = iTransform(m.xt.map(f)) //14

  override def flatMap[T,U](m: ITransform[T])
    (f: T=>ITransform[U]): ITransform[U] =
    iTransform(m.xt.flatMap(t => f(t).xt)) //15
}
```

Finally, let's create an implicit class to automatically convert an `ITransform` object into its associated monad so that it can access the `unit`, `map`, and `flatMap` monad methods transparently:

```
implicit class iTransform2Monad[T] (fct: ITransform[T]) {
    def unit(t: T) = iTransformMonad.unit(t)

    final def map[U](f: T => U): ITransform[U] =
        iTransformMonad.map(fct)(f)
    final def flatMap[U](f: T => ITransform[U]): ITransform[U] =
        iTransformMonad.flatMap(fct)(f)
    def filter(p: T => Boolean): ITransform[T] = //16
        iTransform(fct.xt.filter(p))
}
```

The `filter` method is strictly not an operator of the monad (line 16). However, it is commonly included to constrain (or guard) a sequence of transformation (for example, for comprehension closure). As stated in the *Presentation* section under *Source code in Chapter 1, Getting Started*, code related to exceptions, error checking, and validation of arguments is omitted.

Immutable transformations

The model for a data transformation (or a processing unit or classifier) class should be immutable. Any modification will alter the integrity of the model or parameters used to process data. In order to ensure that the same model is used in processing the input data for the entire lifetime of a transformation, we do the following:

- A model for an `ETransform` is defined as an argument of its constructor.
- The constructor of an `ITransform` generates the model from a given training set. The model has to be rebuilt from the training set (not altered), if it provides an incorrect outcome or prediction.

Models are created by the constructor of classifiers or data transformation classes to ensure their immutability. The design of an immutable transformation is described in the *Design template for immutable classifiers* section under *Scala programming* of the *Appendix A, Basic Concepts*.

A workflow computational model

Monads are very useful for manipulating and chaining data transformations using implicit configurations or explicit models. However, they are restricted to a single morphism $T \Rightarrow U$ type. More complex and flexible workflows require weaving transformations of different types using a generic factory pattern.

Traditional factory patterns rely on a combination of composition and inheritance and do not provide developers with the same level of flexibility as stackable traits.

In this section, we introduce you to the concept of modeling using mixins and a variant of the cake pattern to provide a workflow with three degrees of configurability.

Supporting mathematical abstractions

Stackable traits enable developers to follow a strict mathematical formalism while implementing a model in Scala. Scientists use a universally accepted template to solve a mathematical problem:

1. Declare the variables relevant to the problem.
2. Define a model (equations, algorithms, formulas, and so on) as the solution to the problem.
3. Instantiate the variables and execute the model to solve the problem.

Let's consider the example of the concept of kernel functions (described in the *Kernel functions* section in *Chapter 8, Kernel Models and Support Vector Machines*), a model that consists of a composition of two mathematical functions and its potential implementation in Scala.

Step 1 – variable declaration

The implementation consists of wrapping (scope) the two functions into traits and defining these functions as abstract values.

The mathematical formalism is as follows:

$$f: \mathbb{R}^n \rightarrow \mathbb{R}^n \quad g: \mathbb{R}^n \rightarrow \mathbb{R}$$

The Scala implementation is as follows:

```
type V = Vector[Double]
trait F { val f: V => V}
trait G { val g: V => Double }
```

Step 2 – model definition

The model is defined as the composition of the two functions. The `G` and `F` stack of traits describe the type of compatible functions that can be composed using the self-referenced `self: G with F` constraint.

The formalism will be $h = f \circ g$.

The Scala implementation is as follows:

```
class H {self: G with F => def apply(v:V): Double =g(f(v))}
```

Step 3 – instantiation

The model is executed once the f and g variables are instantiated.

The formalism will be as follows:

$$f: x \rightarrow e^x \quad g: x \rightarrow \sum_0^{n-1} x_i$$

The Scala implementation is as follows:

```
val h = new H with G with F {
    val f: V => V = (v: V) => v.map(Math.exp(_))
    val g: V => Double = (v: V) => v.sum
}
```

Lazy value triggers

In the preceding example, the value of $h(v) = g(f(v))$ can be automatically computed as soon as g and f are initialized, by declaring h a lazy value.

Clearly, Scala preserves the formalism of mathematical models, making it easier for scientists and developers to migrate their existing projects written in scientific-oriented languages, such as R.

Emulation of R

Most data scientists use the R language to create models and apply learning strategies. They may consider Scala as an alternative to R in some cases, as Scala preserves the mathematical formalism used in models implemented in R.

Let's extend the concept preservation of mathematical formalism to the dynamic creation of workflows using traits. The design pattern described in the next section is sometimes referred to as the **Cake pattern**.

Composing mixins to build a workflow

This section presents the key constructs behind the Cake pattern. A workflow composed of configurable data transformations requires a dynamic modularization (substitution) of the different stages of the workflow.

Traits and mixins



Mixins are traits that are stacked against a class. The composition of mixins and the Cake pattern described in this section are important for defining the sequences of data transformations. However, the topic is not directly related to machine learning and so you can skip this section.

The Cake pattern is an advanced class composition pattern that uses mixin traits to meet the demands of a configurable computation workflow. It is also known as stackable modification traits [2:4].

This is not an in-depth analysis of the **stackable trait injection** and **self-reference** in Scala. There are few interesting articles on dependencies injection that are worth a look [2:5].

Java relies on packages tightly coupled with the directory structure and prefixed to modularize the code base. Scala provides developers with a flexible and reusable approach to create and organize modules: traits. Traits can be nested, mixed with classes, stacked, and inherited.

Understanding the problem

Dependency injection is a fancy name for a reverse look-up and binding to dependencies. Let's consider a simple application that requires data preprocessing, classification, and validation. A simple implementation using traits looks like this:

```
val app = new Classification with Validation with PreProcessing {  
    val filter = ...  
}
```

If, at a later stage, you need to use an unsupervised clustering algorithm instead of a classifier, then the application has to be rewired:

```
val app = new Clustering with Validation with PreProcessing {  
    val filter = ...  
}
```

This approach results in code duplication and lack of flexibility. Moreover, the `filter` class member needs to be redefined for each new class in the composition of the application. The problem arises when there is a dependency between traits used in the composition. Let's consider the case for which the `filter` depends on the `validation` methodology.

Mixins linearization [2:6]

The linearization or invocation of methods between mixins follows a right-to-left and base-to-subtype pattern:

- Trait `B` extends `A`
- Trait `C` extends `A`
- Class `M` extends `N` with `C` with `B`

The Scala compiler implements the linearization as $A \Rightarrow B \Rightarrow C \Rightarrow N$.

Although you can define `filter` as an abstract value, it still has to be redefined each time a new validation type is introduced. The solution is to use the `self` type in the definition of the newly composed `PreProcessingWithValidation` trait:

```
trait PreProcessingWithValidation extends PreProcessing {
    self: Validation => val filter = ...
}
```

The application is built by stacking the `PreProcessingWithValidation` mixin against the `Classification` class:

```
val app = new Classification with PreProcessingWithValidation {
    val validation: Validation
}
```

Overriding def with val

It is advantageous to override the declaration of a method with a declaration of a value with the same signature. Contrary to a value that is assigned once for all during instantiation, a method may return a different value for each invocation. A `def` is a `proc` that can be redefined as a `def`, `val`, or `lazy val`. Therefore, you should not override a value declaration with a method with the same signature:

```
trait Validator { val g = (n: Int) => ... }
trait MyValidator extends Validator { def g(n: Int) =
... } //WRONG
```

Let's adapt and generalize this pattern to construct a boilerplate template in order to create dynamic computational workflows.

Defining modules

The first step is to generate different modules to encapsulate different types of data transformation.

Use case for describing the cake pattern

 It is difficult to build an example of a real-world workflow using classes and algorithms introduced later in the book. The following simple example is realistic enough to illustrate the different components of the Cake pattern.

Let's define a sequence of the three parameterized modules that each define a specific data transformation using the explicit configuration of the `Etransform` type:

- Sampling: This is used to extract a sample from raw data
- Normalization: This is used to normalize the sampled data over [0, 1]
- Aggregation: This is used to aggregate or reduce the data

The code will be as follows:

```
trait Sampling[T,A,B] {  
    val sampler: ETransform[T] { type U = A; type V = B }  
}  
trait Normalization[T,A,B] {  
    val normalizer: ETransform[T] { type U = A; type V = B }  
}  
trait Aggregation[T,A,B] {  
    val aggregator: ETransform[T] { type U = A; type V = B }  
}
```

The modules contain a single abstract value. One characteristic of the Cake pattern is to enforce strict modularity by initializing the abstract values with the type encapsulated in the module. One of the objectives in building the framework is allowing developers to create data transformation (inherited from `ETransform`) independently from any workflow.

Scala traits and Java packages

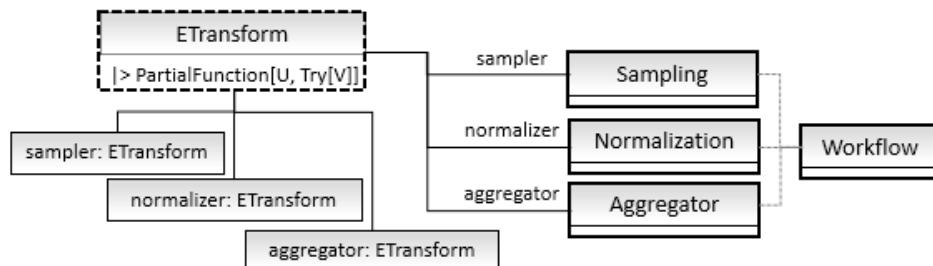
 There is a major difference between Scala and Java in terms of modularity. Java packages constrain developers into following a strict syntax that requires, for instance, the source file to have the same name as the class it contains. Scala modules based on stackable traits are far more flexible.

Instantiating the workflow

The next step is to *write* the different modules into a workflow. This is achieved by using the `self` reference to the stack of the three traits defined in the previous section:

```
class Workflow[T,U,V,W,Z] {
    self: Sampling[T,U,V] with
        Normalization[T,V,W] with
            Aggregation[T,W,Z] =>
    def |> (u: U): Try[Z] = for {
        v <- sampler |> u
        w <- normalizer |> v
        z <- aggregator |> w
    } yield z
}
```

A picture is worth a thousand words; the following UML class diagram illustrates the workflow factory (or Cake) design pattern:



The UML class diagram of the workflow factory

Finally, the workflow is instantiated by dynamically initializing the `sampler`, `normalizer`, and `aggregator` abstract values of the transformation as long as the signature (input and output types) matches the parameterized types defined in each module (line 1):

```
type Dbl_F = Function1[Double, Double]
val samples = 100; val normRatio = 10; val splits = 4

val workflow = new Workflow[Int, Dbl_F, DblVector, DblVector, Int]
  with Sampling[Int, Dbl_F, DblVector]
    with Normalization[Int, DblVector, DblVector]
      with Aggregation[Int, DblVector, Int] {
  val sampler = new ETransform[Int](samples) { /* .. */ } //1
  val normalizer = new ETransform[Int](normRatio) { /* .. */ }
  val aggregator = new ETransform[Int](splits) {/* .. */ }
}
```

Let's implement the data transformation function for each of the three modules/traits by assigning a transformation to the abstract values.

The first transformation, `sampler`, samples a `f` function with frequency as $1/samples$ over the interval $[0, 1]$. The second transformation, `normalizer`, normalizes the data over the range $[0, 1]$ using the `Stats` class introduced in the next chapter. The last transformation, `aggregator`, extracts the index of the large sample (value 1.0):

```
val sampler = new ETransform[Int](samples) { //2
  type U = Dbl_F //3
  type V = DblVector //4
  override def |> : PartialFunction[U, Try[V]] = {
    case f: U =>
      Try(Vector.tabulate(samples)(n => f(1.0*n/samples))) //5
  }
}
```

The `sampler` transformation uses a single model or configuration parameter, `sample`, (line 2). The `U` type of an input is defined as `Double => Double` (line 3) and the `V` type of an output is defined as a vector of floating point values, `DblVector` (line 4). In this particular case, the transformation consists of applying the input `f` function to a vector of increasing normalized values (line 5).

The `normalizer` and `aggregator` transforms follow the same design pattern as `sampler`:

```
val normalizer = new ETransform[Int](normRatio) {
  type U = DblVector; type V = DblVector
  override def |> : PartialFunction[U, Try[V]] = { case x: U
```

```

        if(x.size >0) => Try((Stats[Double](x)).normalize)
    }
}
val aggregator = new ETransform[Int](splits) {
    type U = DblVector; type V = Int
    override def |> : PartialFunction[U, Try[V]] = case x: U
        if(x.size > 0) => Try(Range(0,x.size).find(x(_)==1.0).get)
    }
}

```

The instantiation of the transformation function follows the template described in the *Explicit models* section in this chapter.

The workflow is now ready to process any function as an input:

```

val g = (x: Double) => Math.log(x+1.0) + Random.nextDouble
Try( workflow |> g ) //6

```

The workflow is executed by providing the input `g` function to the first `sampler` mixin (line 6).

Scala's strong type checking catches any inconsistent data types at compilation time. It reduces the development cycle because runtime errors are more difficult to track down.

Mixins composition for `ITransform`

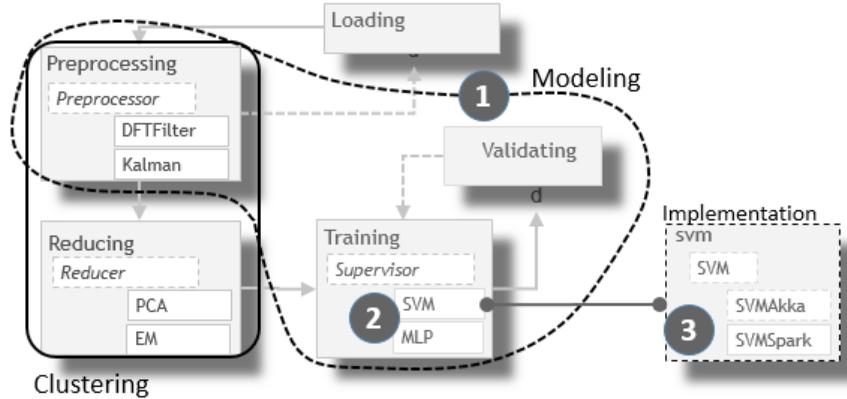
We arbitrary selected a data transformation using an explicit `ETransform` configuration to illustrate the concept of mixins composition. The same pattern applies to the implicit `ITransform` data transformation.

Modularization

The last step is the modularization of the workflow. For complex scientific computations, you need to be able to do the following:

1. Select the appropriate *workflow* as a sequence of modules or tasks according to the objective of the execution (regression, classification, clustering, and so on).
2. Select the appropriate *algorithm* to fulfill a task according to the data (noisy data, an incomplete training set, and so on).

3. Select the appropriate *implementation* of the algorithm according to the environment (distributed with a high-latency network, single host, and so on).



An Illustration of the dynamic creation of a workflow from modules/traits

Let's consider a simple preprocessing task defined in the `PreprocessingModule` module. The module (or task) is declared as a trait to hide its internal workings from other modules. The preprocessing task is executed by a preprocessor of a `Preprocessor` type. We arbitrary list two algorithms: the exponential moving average of the `ExpMovingAverage` type and the discrete Fourier transform low pass filter of the `DFTFilter` type as a potential preprocessor:

```
trait PreprocessingModule[T] {
    trait Preprocessor[T] { //7
        def execute(x: Vector[T]): Try[DblVector]
    }
    val preprocessor: Preprocessor[T] //8

    class ExpMovingAverage[T <: AnyVal]( //9
        p: Int)
        (implicit num: Numeric[T], f: T => Double)
    extends Preprocessor[T] {

        val expMovingAvg = filtering.ExpMovingAverage[T](p) //10
        val pfn = expMovingAvg |> //11
```

```

override def execute(x: Vector[T]): Try[DblVector] =
  pfn(x).map(_.toVector)
}

class DFTFilter[T <: AnyVal] (
  fc: Double)
  (g: (Double,Double) =>Double)
  (implicit f : T => Double)
extends Preprocessor[T] { //12

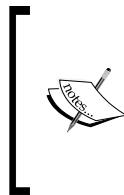
  val filter = filtering.DFTFir[T](g, fc, 1e-5)
  val pfn = filter |>
    override def execute(x: Vector[T]): Try[DblVector] =
      pfn(x).map(_.toVector)
}
}

```

The generic preprocessor trait, `Preprocessor`, declares a single `execute` method whose purpose is to filter an `x` input vector of an element of a `T` type for noise (line 7). The instance of the preprocessor is declared as an abstract class to be instantiated as one of the filtering algorithms (line 8).

The first filtering algorithm of an `ExpMovingAverage` type implements the `Preprocessor` trait and overrides the `execute` method (line 9). The class declares the algorithm but delegates its implementation to a class with an identical `org.scalaml.filtering.ExpMovingAverage` signature (line 10). The partial function returned from the `|>` method is instantiated as a `pfn` value, so it can be applied multiple times (line 11). The same design pattern is used for the discrete Fourier transform filter (line 12).

The filtering algorithm (`ExpMovingAverage` or `DFTFir`) is selected according to the profile or characteristic of the input data. Its implementation in the `org.scalaml.filtering` package depends on the environment (a single host, cluster, Apache spark, and so on).



Filtering algorithms

The filtering algorithms used to illustrate the concept of modularization in the context of the Cake pattern are described in detail in *Chapter 3, Data Preprocessing*.

Profiling data

The selection of a preprocessing, clustering, or classification algorithm depends highly on the quality and profile of the input data (observations and expected values whenever available). The *Step 3 - preprocessing the data* section under *Let's kick the tires in Chapter 1, Getting Started*, introduced the `MinMax` class for normalizing a dataset using the minimum and maximum values.

Immutable statistics

The mean and standard deviation are the most commonly used statistics.

Mean and variance

Arithmetic mean is defined as:

$$E[X] = \mu = \frac{1}{n} \sum_{i=0}^{n-1} x_i$$

Variance is defined as:


$$\text{Var}(X) = \frac{\sum(E(X) - x_j)^2}{n - 1}$$

Variance adjusted for a sampling bias is defined as:

$$\widehat{\text{Var}}(X) = \frac{1}{n - 1} \sum_{i=1}^n (x_i - E[X])^2$$

Let's extend the `MinMax` class with some basic statistics capabilities using `Stats`:

```
class Stats[T < : AnyVal] (
    values: Vector[T])(implicit f : T => Double)
extends MinMax[T](values) {

    val zero = (0.0, 0.0)
    val sums = values.//: (zero)((s,x) =>(s._1 +x, s._2 + x*x)) //1

    lazy val mean = sums._1/values.size //2
    lazy val variance =
        (sums._2 - mean*mean*values.size)/(values.size-1)
    lazy val stdDev = Math.sqrt(variance)
    ...
}
```

The Stats class implements **immutable statistics**. Its constructor computes the sum of values and sum of square values, sums (line 1). The statistics such as mean and variance are computed once when needed by declaring these values as lazy (line 2). The Stats class inherits the normalization functions of MinMax.

Z-Score and Gauss

The Gaussian distribution of the input data is implemented by the gauss method of the Stats class.

The Gaussian distribution

M1: Gaussian for a mean μ and a standard deviation σ transformation is defined as:

$$y = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The code is as follows:

```
def gauss(mu: Double, sigma: Double, x: Double): Double = {
    val y = (x - mu)/sigma
    INV_SQRT_2PI*Math.exp(-0.5*y*y)/sigma
}
val normal = gauss(1.0, 0.0, _: Double)
```

The computation of the normal distribution is computed as a partially applied function. The Z-score is computed as a normalization of the raw data taking into account the standard deviation.

Z-score normalization

M2: Z-score for a mean μ and a standard deviation σ is defined as:

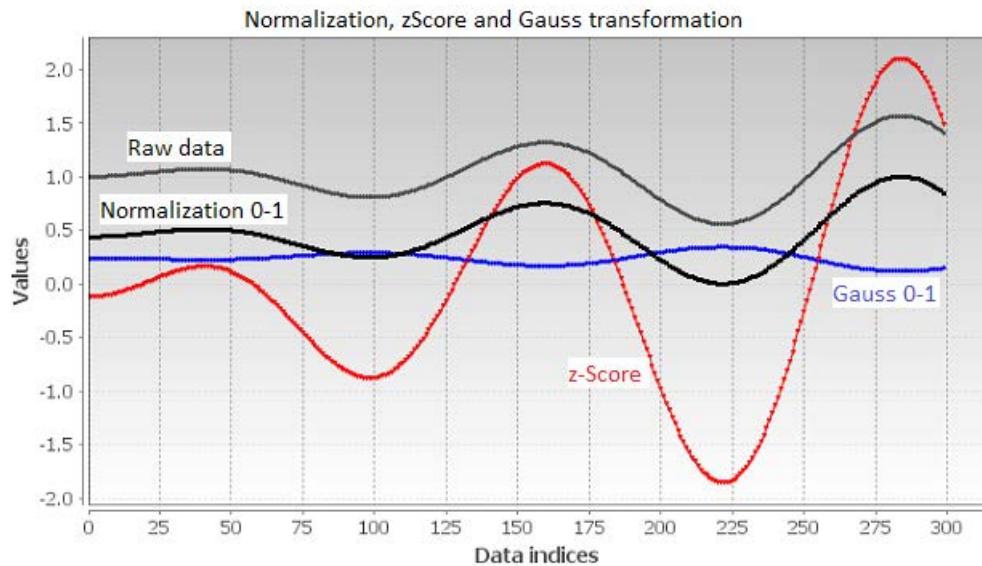
$$z_i = \frac{x_i - \mu}{\sigma}$$

The computation of the Z-score is implemented by the zScore method of Stats:

```
def zScore: DblVector = values.map(x => (x - mean) / stdDev )
```

Hello World!

The following graph illustrates the relative behavior of the zScore normalization and normal transformation:



A comparative analysis of linear, Gaussian, and Z-score normalization

Assessing a model

Evaluating a model is an essential part of the workflow. There is no point in creating the most sophisticated model if you do not have the tools to assess its quality. The validation process consists of defining some quantitative reliability criteria, setting a strategy such as a **K-fold cross-validation** scheme, and selecting the appropriate labeled data.

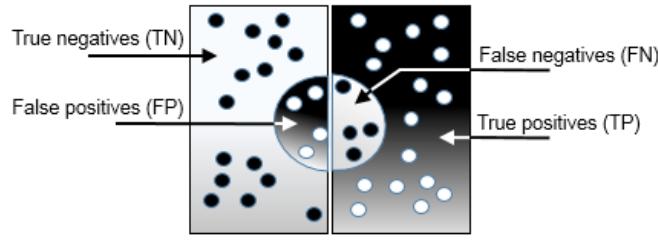
Validation

The purpose of this section is to create a reusable Scala class to validate models. For starters, the validation process relies on a set of metrics to quantify the fitness of a model generated through training.

Key quality metrics

Let's consider a simple classification model with two classes defined as positive (with respect to negative) represented with Black (with respect to White) color in the following diagram. Data scientists use the following terminologies:

- **True positives (TP):** These are observations that are correctly labeled as those that belong to the positive class (white dots on a dark background)
- **True negatives (TN):** These are observations that are correctly labeled as those that belong to the negative class (black dots on a light background)
- **False positives (FP):** These are observations incorrectly labeled as those that belong to the positive class (white dots on a dark background)
- **False negatives (FN):** These are observations incorrectly labeled as those that belong to the negative class (dark dots on a light background)



Categorization of validation results

This simplistic representation can be extended to classification problems that involve more than two classes. For instance, false positives are defined as observations incorrectly labeled that belong to any class other than the correct one. These four factors are used for evaluating accuracy, precision, recall, and F and G measures, as follows:

- **Accuracy:** This is the percentage of observations correctly classified and is represented as ac .
- **Precision:** This is the percentage of observations correctly classified as positive in the group that the classifier has declared positive. It is represented as p .
- **Recall:** This is the percentage of observations labeled as positive that are correctly classified and is represented as r .
- **F_1 -measure or F_1 -score:** This measure strikes a balance between precision and recall. It is computed as the harmonic mean of the precision and recall with values ranging between 0 (worst score) and 1 (best score). It is represented as F_1 .

- **F_n score:** This is the generic F scoring method with an arbitrary n degree. It is represented as F_n .
- **G measure:** This is similar to the F-measure but is computed as the geometric mean of precision p and recall r . It is represented as G .

Validation metrics

M3: Accuracy ac , precision p , recall r , F_1 , F_n , and G scores are defined as follows:

$$ac = \frac{TP + TN}{TP + TN + FP + FN} \quad p = \frac{TP}{TP + FP} \quad r = \frac{TP}{TP + FN}$$

$$F_1 = \frac{2pr}{p+r} \quad F_n = \frac{(1+n^2)pr}{n^2p+r} \quad G = \sqrt{pr}$$

The computation of the precision, recall, and F_1 score depends on the number of classes used in the classifier. We will consider the following implementations:

- F-score validation for binomial (two classes) classification (that is, a positive and negative outcome)
- F-score validation for multinomial (more than two classes) classification

F-score for binomial classification

The binomial F validation computes the precision, recall, and F scores for the positive class.

Let's implement the F-score or F-measure as a specialized validation of the following:

```
trait Validation { def score: Double }
```

The `BinFValidation` class encapsulates the computation of the F_n score as well as precision and recall by counting the occurrences of TP , TN , FP , and FN values. It implements the **M3** formula. In the tradition of Scala programming, the class is immutable; it computes the counters for TP , TN , FP , and FN when the class is instantiated. The class takes the following three parameters:

- The expected values with the `0` value for a negative outcome and `1` for a positive outcome
- The set of observations, `xt`, is used for validating the model
- The predictive `predict` function classifies observations (line 1)

The code will be as follows:

```

class BinFValidation[T <: AnyVal] (
    expected: Vector[Int],
    xt: XVSeries[T])
    (predict: Array[T] => Int)(implicit f: T => Double)
extends Validation { //1

    val counters = {
        val predicted = xt.map( predict(_) )
        expected.zip(predicted)
            .aggregate(new Counter[Label])((cnt, ap) =>
                cnt + classify(ap._1, ap._2), _ ++ _) //2
    }

    override def score: Double = f1 //3
    lazy val f1 = 2.0*precision*recall/(precision + recall)
    lazy val precision = compute(FP) //4
    lazy val recall = compute(FN)

    def compute(n: Label): Double = {
        val denom = counters(TP) + counters(n)
        counters(TP).toDouble/denom
    }
    def classify(predicted: Int, expected: Int): Label = //5
        if(expected == predicted) if(expected == POSITIVE) TP else TN
        else if(expected == POSITIVE) FN else FP
    }
}

```

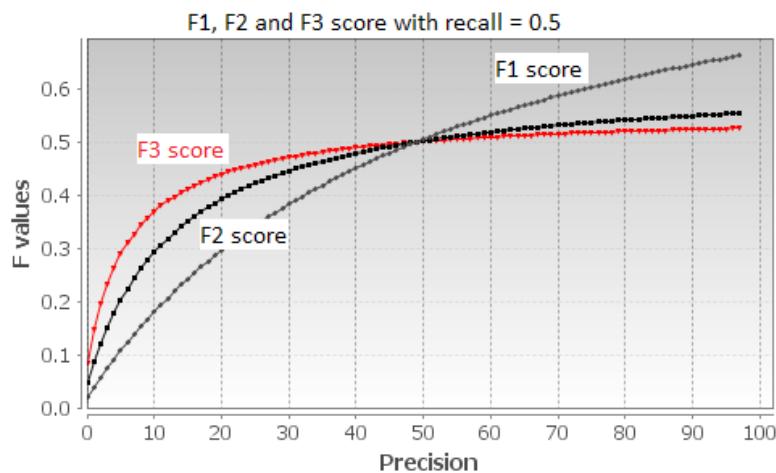
The constructor counts the number of occurrences for each of the four outcomes (TP , TN , FP , and FN) (line 2). The `precision`, `recall`, and `f1` values are defined as lazy values so they are computed only once, when they are accessed directly or the `score` method is invoked (line 4). The F_1 measure is the most commonly used scoring value for validating classifiers. Therefore, it is the default score (line 3). The `classify` private method extracts the qualifier from the `expected` and `predicted` values (line 5).

The `BinFValidation` class is independent of the type of classifier, training, labeling process, and type of observations.

Contrary to Java, which defines an enumerator as a class of types, Scala requires enumerators to be singletons. Enumerators extend the `scala.Enumeration` abstract class:

```
object Label extends Enumeration {
    type Label = Value
    val TP, TN, FP, FN = Value
}
```

The F-score formula with higher cardinality (F_n) with $n > 1$ favors precision over recall, which is shown in the following graph:



A comparative analysis of the impact of precision on F1, F2, and F3 score for a given recall

Multiclass scoring

Our implementation of the binomial validation computes the precision, recall, and F_1 score for the positive class only. The generic multinomial validation class presented in the next section computes these quality metrics for both positive and negative classes.

F-score for multinomial classification

The validation metric is defined by the M3 formula. The idea is quite simple: the precision and recall values are computed for all the classes and then they are averaged to produce a single precision and recall value for the entire model. The precision and recall for the entire model leverage the counts of TP , FP , FN , and TN introduced in the previous section.

There are two commonly used set of formulas to compute the precision and recall for a model:

- **Macro:** This method computes the precision and recall for each class, sums and then averages them up.
- **Micro:** This method sums the numerator and denominator of the precision and recall formulas for all the classes before computing the precision and recall.

We will use the macro formulas from now on.

Macro formulas for multinomial precision and recall

M4: The macro version of the precision p and recall r for a model of the c classes is computed as follows:

$$P = \frac{1}{c} \sum_{i=0}^{c-1} \frac{TP_i}{TP_i + FP_i} \quad r = \frac{1}{c} \sum_{i=0}^{c-1} \frac{TP_i}{TP_i + FN_i}$$

The computation of the precision and recall factor for a classifier with more than two classes requires the extraction and manipulation of the **confusion matrix**. We use the following convention: *expected values are defined as columns and predicted values are defined as rows*.

		Actual					
		1	2	3	4	5	6
Predicted	1	167	3	19	8	0	2
	2	11	107	3	27	4	12
	3	4	21	145	3	7	14
	4	9	17	4	179	20	0
	5	15	0	18	2	139	8
	6	1	6	0	24	8	164
		False negatives					

A confusion matrix for six class classification

The `MultiFValidation` multinomial validation class takes the following four parameters:

- The expected class index with the `0` value for a negative outcome and `1` for a positive outcome
- The set of observations, `xv`, is used for validating the model
- The number of classes in the model
- The `predict` predictive function classifies observations (line 7)

The code will be as follows:

```
class MultiFValidation[T <: AnyVal] {  
    expected: Vector[Int],  
    xv: XVSeries[T],  
    classes: Int  
    (predict: Array[T] => Int) (implicit f : T => Double)  
    extends Validation { //7  
  
        val confusionMatrix: Matrix[Int] = //8  
        labeled./:(Matrix[Int](classes)){case (m, (x,n)) =>  
            m + (n, predict(x), 1)} //9  
  
        val macroStats: DblPair = { //10  
            val pr= Range(0, classes)./:((0.0,0.0)((s, n) => {  
                val tp = confusionMatrix(n, n) //11  
                val fn = confusionMatrix.col(n).sum - tp //12  
                val fp = confusionMatrix.row(n).sum - tp //13  
                (s._1 + tp.toDouble/(tp + fp), s._2 +tp.toDouble/(tp + fn))  
            }))  
            (pr._1/classes, pr._2/classes)  
        }  
        lazy val precision: Double = macroStats._1  
        lazy val recall: Double = macroStats._1  
        def score: Double = 2.0*precision*recall/(precision+recall)  
    }  
}
```

The core element of the multiclass validation is the confusion matrix, `confusionMatrix` (line 8). Its elements at indices $(i, j) = (\text{index of expected class for an observation}, \text{index of the predicted class for the same observation})$ are computed using the expected and predictive outcome for each class (line 9).

As stated in the introduction of the section, we use the macro definition of the precision and recall (line 10). The count of a true positive, t_p , for each class corresponds to the diagonal element of the confusion matrix (line 11). The count of the f_n false negatives for a class is computed as the sum of the counts for all the predicted classes (column values), given an expected class except the true positive class (line 12). The count of the f_p false positives for a class is computed as the sum of the counts for all the expected classes (row values), given a predicted class except the true positive class (line 13).

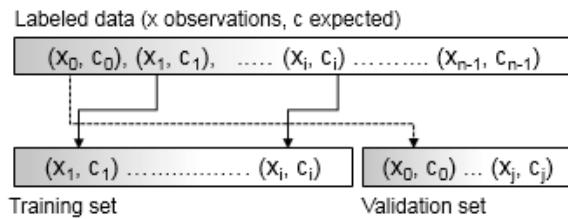
The formula for the computation of the F_1 score is the same as the formula used in the binomial validation.

Cross-validation

It is quite common that the labeled dataset (observations plus the expected outcome) available to the scientists is not very large. The solution is to break the original labeled dataset into K groups of data.

One-fold cross validation

The one-fold cross validation is the simplest scheme used for extracting a training set and validation set from a labeled dataset, as described in the following diagram:



An illustration of the generation of a one-fold validation set

The one-fold cross validation methodology consists of the following three steps:

1. Select the ratio of the size of the training set over the size of the validation set.
2. Randomly select the labeled observations for the validation phase.
3. Create the training set as the remaining labeled observations.

The one-fold cross validation is implemented by the `OneFoldXValidation` class. It takes the following three arguments: an `xt` vector of observations, the `expected` vector of expected classes, and `ratio` of the size of the training set over the size of the validation set (line 14):

```
type ValidationType[T] = Vector[(Array[T], Int)]
class OneFoldXValidation[T <: AnyVal] (
    xt: XVSeries[T],
    expected: Vector[Int],
    ratio: Double)(implicit f : T => Double) { //14
    val dataSet: (ValidationType[T], ValidationType[T]) //15
    def trainingSet: ValidationType[T] = dataSet._1
    def validationSet: ValidationType[T] = dataSet._1
}
```

The constructor of the `OneFoldXValidation` class generates the segregated training and validation sets from the set of observations and expected classes (line 15):

```
val dataSet: (Vector[LabeledData[T]], Vector[LabeledData[T]]) = {
    val labeledData = xt.drop(1).zip(expected) //16
    val trainingSize = (ratio*expected.size).floor.toInt //17

    val valSz = labeledData.size - trainingSize
    val adjSz = if(valSz < 2) 1
                else if(valSz >= labeledData.size) labeledData.size - 1
                else valSz //18
    val iter = labeledData.grouped(adjSz) //18
    val ordLabeledData = labeledData
        .map( _, Random.nextDouble) //19
        .sortWith( _._2 < _._2).unzip._1 //20

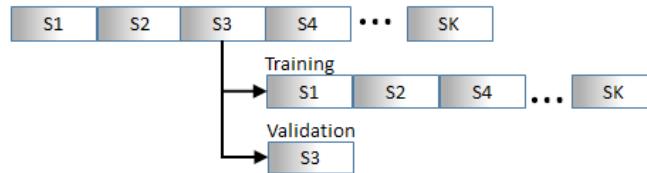
    (ordLabeledData.takeRight(adjValSz),
     ordLabeledData.dropRight(adjValSz)) //21
}
```

The initialization of the `OneFoldXValidation` class creates a `labeledData` vector of labeled observations by zipping the observations and the expected outcome (line 16). The training `ratio` value is used to compute the respective size of the training set (line 17) and validation set (line 18).

In order to create training and validations sets randomly, we zip the labeled dataset with a random generator (line 19), and then reorder the labeled dataset by sorting the random values (line 20). Finally, the method returns the pair of training set and validation set (line 21).

K-fold cross validation

The data scientist creates K training-validation datasets by selecting one of the groups as a validation set and then combining all the remaining groups into a training set, as illustrated in the following diagram. The process is known as the **K-fold cross validation** [2:7].



An illustration of the generation of a K-fold cross validation set

The third segment is used as validation data and all other dataset segments except S3 are combined into a single training set. This process is applied to each segment of the original labeled dataset.

Bias-variance decomposition

The challenge is to create a model that fits both the training set and subsequent observations to be classified during the validation phase.

If the model tightly fits the observations selected for training, there is a high probability that new observations may not be correctly classified. This is usually the case when the model is complex. This model is characterized as having a low bias with a high variance. Such a scenario can be attributed to the fact that the scientist is overly confident that the observations she/he selected for training are representative of the real world.

The probability of a new observation being classified as belonging to a positive class increases as the selected model fits loosely the training set. In this case, the model is characterized as having a high bias with a low variance.

The mathematical definition for the **bias**, **variance**, and **mean square error** (MSE) of the distribution are defined by the following formulas:

M5: Variance and bias for a true model, θ , is defined as:


$$\text{var } \hat{\theta} = E[(\hat{\theta} - E[\hat{\theta}])^2] \quad \text{bias}(\hat{\theta}) = \hat{\theta} - \theta \quad (\hat{\theta}: \theta \text{ estimate})$$

M6: Mean square error is defined as:

$$MSE = \text{var}(\hat{\theta}) + \text{bias}(\hat{\theta})^2$$

Let's illustrate the concept of bias, variance, and mean square error with an example. At this stage, you have not been introduced to most of the machines learning techniques. Therefore, we create a simulator to illustrate the relation between the bias and variance of a classifier. The components of the simulation are as follows:

- A training set, `training`
- A simulated target model of the target: `Double => Double` type extracted from the training set
- A set of possible models to evaluate

A model that exactly matches the training data overfits the target model. Models that approximate the target model will most likely underfit. The models in this example are defined by single variable functions.

These models are evaluated against a validation dataset. The `BiasVariance` class takes the target model, `target`, and the size of the `nValues` validation test as parameters (line 22). It merely implements the formula to compute the bias and variance for each model:

```
type Dbl_F = Double => Double
class BiasVariance[T](target: Dbl_F, nValues: Int)
  (implicit f: T => Double) //22
  def fit(models: List[Dbl_F]): List[DblPair] = { //23
    models.map(accumulate(_, models.size)) //24
  }
}
```

The `fit` method computes the variance and bias for each of the `models` model compared to the target model (line 23). It computes the mean, variance, and bias in the `accumulate` method (line 24):

```
def accumulate(f: Dbl_F, y:Double, numModels: Int): DblPair =
  Range(0, nValues) ./:(0.0, 0.0){ case((s,t) x) => {
    val diff = (f(x) - y)/numModels
    (s + diff*diff, t + Math.abs(f(x)-target(x))) //25
  } }
```

The training data is generated by the single variable function with the r_1 and r_2 noise components:

$$y = \frac{x}{5} \left(1 + \frac{1}{n} \sin \left(\frac{x}{10} + r1 \right) \right) + r2$$

The `accumulate` method returns a tuple (variance, bias) for each model, f (line 25). The model candidates are defined by the following family of single variable for values $n = 1, 2, \text{ and } 4$:

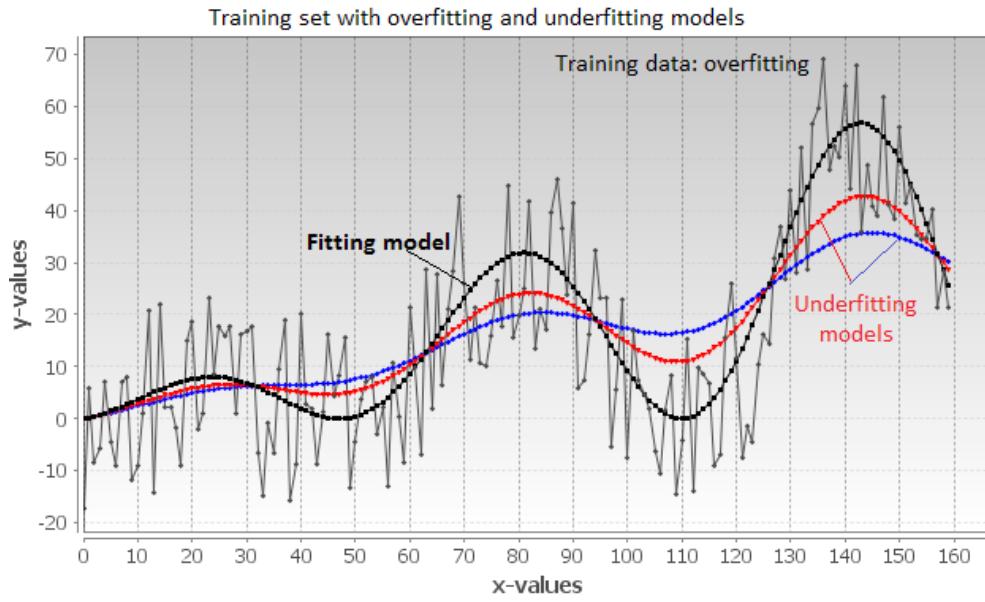
$$y = \frac{x}{5} \left(1 + \frac{1}{n} \sin \left(\frac{x}{10} \right) \right)$$

The target model (line 26) and `models` (line 27) belong to the same family of single variable functions:

```
val template = (x: Double, n : Int) =>
  0.2*x*(1.0 + Math.sin(x*0.1)/n)
val training = (x: Double) => {
  val r1 = 0.45*(Random.nextDouble-0.5)
  val r2 = 38.0*(Random.nextDouble - 0.5) + Math.sin(x*0.3)
  0.2*x*(1.0 + Math.sin(x*0.1 + r1)) + r2
}
Val target = (x: Double) => template(x, 1) //26
val models = List[(Dbl_F, String)] ( //27
  ((x: Double) => template(x, 4), "Underfit1"),
  ((x: Double) => template(x, 2), "Underfit2"),
  ((x : Double) => training(x), "Overfit"),
  (target, "target"),
)
val evaluator = new BiasVariance[Double](target, 200)
evaluator.fit(models.map(_._1)) match { /* ... */ }
```

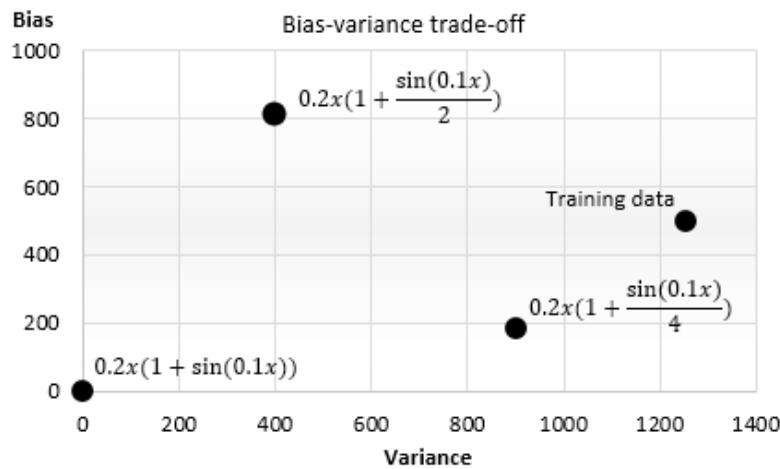
Hello World!

The **JFreeChart** library is used to display the training dataset and the models:



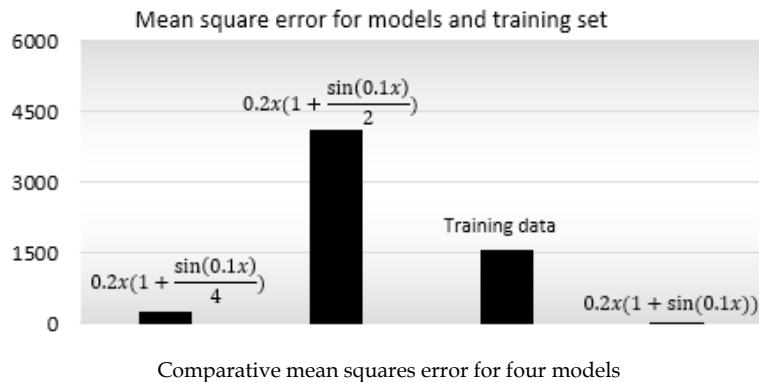
Fitting models to dataset

The model that replicates the training data overfits. The models that smooth the model with lower amplitude for the sine component of the template function underfit. The **variance-bias trade-off** for the different models and training data is illustrated in the following scatter chart:



Scatter plot for the bias-variance trade-off for four models, one duplicating the training set

The variance of each of the smoothing or approximating models is lower than the variance of the training set. As expected the target model, $0.2x(1+\sin(x/10))$, has no bias and no variance. The training set has a very high variance because it overfits any target model. The last chart compares the mean square error between each of the models, training set, and the target model:



Evaluating bias and variance

The section uses a fictitious target model and training set to illustrate the concept of the bias and variance of models. The bias and variance of machine learning models are actually estimated using validation data.

Overfitting

You can apply the methodology presented in the example to any classification and regression model. The list of models with low variance includes constant functions and models independent of the training set. High degree polynomials, complex functions, and deep neural networks have high variance. Linear regression applied to linear data has a low bias, while linear regression applied to nonlinear data has a higher bias [2:8].

Overfitting affects all aspects of the modeling process negatively, for example:

- It renders debugging difficult
- It makes the model too dependent of minor fluctuations (long tail) and noisy data
- It may discover irrelevant relationships between observed and latent features
- It leads to poor predictive performance

However, there are well-proven solutions to reduce overfitting [2:9]:

- Increasing the size of the training set whenever possible
- Reducing noise in labeled observations using smoothing and filtering techniques
- Decreasing the number of features using techniques such as principal components analysis, as discussed in the *Principal components analysis* section in *Chapter 4, Unsupervised Learning*
- Modeling observable and latent noisy data using Kalman or auto regressive models, as discussed in *Chapter 3, Data Preprocessing*
- Reducing inductive bias in a training set by applying cross-validation
- Penalizing extreme values for some of the model's features using regularization techniques, as discussed in the *Regularization* section in *Chapter 6, Regression and Regularization*

Summary

In this chapter, we established the framework for the different data processing units that will be introduced in this book. There is a very good reason why the topics of model validation and overfitting are explored early in this book. There is no point in building models and selecting algorithms if we do not have a methodology to evaluate their relative merits.

In this chapter, you were introduced to the following:

- The concept of monadic transformation for implicit and explicit models
- The versatility and cleanness of the Cake pattern and mixins composition in Scala as an effective scaffolding tool for data processing
- A robust methodology to validate machine learning models
- The challenge in fitting models to both training and real-world data

The next chapter will address the problem of overfitting by identifying outliers and reducing noise in data.

3

Data Preprocessing

Real-world data is usually noisy and inconsistent with missing observations. No classification, regression, or clustering model can extract relevant information from raw data.

Data preprocessing consists of cleaning, filtering, transforming, and normalizing raw observations using statistics in order to correlate features or groups of features, identify trends and models, and filter out noise. The purpose of cleansing raw data is as follows:

- To extract some basic knowledge from raw datasets
- To evaluate the quality of data and generate clean datasets for unsupervised or supervised learning

You should not underestimate the power of traditional statistical analysis methods to infer and classify information from textual or unstructured data.

In this chapter, you will learn how to:

- Apply commonly used moving average techniques to detect long-term trends in a time series
- Identify market and sector cycles using discrete Fourier series
- Leverage the discrete Kalman filter to extract the state of a linear dynamic system from incomplete and noisy observations

Time series in Scala

The overwhelming majority of examples used to illustrate the different machine algorithms in this book deal with time series or sequential, time-ordered set of observations.

Types and operations

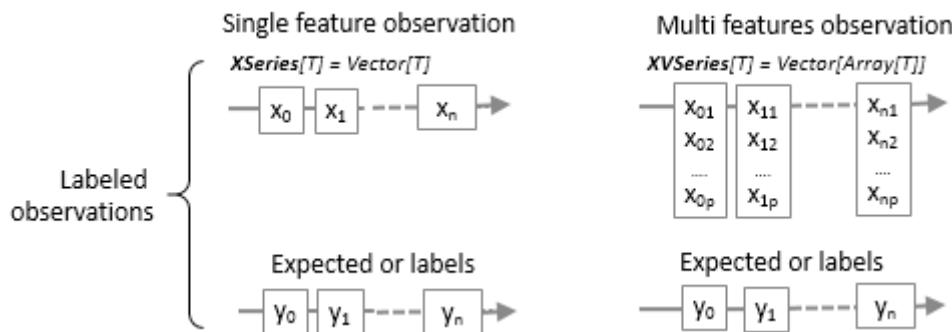
The *Primitives types* section under *Source code in Chapter 1, Getting Started*, introduced the types for a time series of a single `xSeries [T]` variable and multiple `xvSeries [T]` variables.

A time series of observations is a vector (a `Vector` type) of observation elements of the following types:

- A `T` type in the case of a single variable/feature observation
- An `Array[T]` type for observations with more than one variable/feature

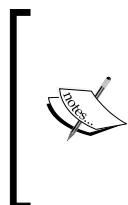
A time series of labels or expected values is a single variable vector for which elements may have a primitive `Int` type for classification and `Double` for regression.

A time series of labeled observations is a pair of a vector of observations and a vector of labels:



Visualization of the single features and multi-feature observations

The two generic `xSeries` and `xvSeries` types for the time series will be used as the two primary classes for the input data, from now on.



Structure of labeled observations

Throughout the book, labeled observations are defined either as a pair of vector of observations and a vector of labels/expected values or as a vector of a pair of {observation, label/expected value}.

The `Stats` class introduced in the *Profiling data* section in *Chapter 2, Hello World!*, implements some basic statistics and normalization for single variable observations. Let's create an `XTSeries` singleton to compute the statistics and normalize multidimensional observations:

```
object XTSeries {
    def zipWithShift[T] (xv: XSeries[T], n: Int): Vector[(T, T)] =
        xv.drop(n).zip(xv.view.dropRight(n)) //1

    def zipWithShift1[T] (xv: XSeries[T]): Vector[(T, T)] =
        xv.zip(xv.view.drop(n))

    def statistics[T <: AnyVal] (xt: XVSeries[T])
        (implicit f: T =>: Double): Vector[Stats[T]] =
        xt.transpose.map(Stats[T](_)) //2

    def normalize[T <: AnyVal] ( //3
        xt: XSeries[T], low: Double, high: Double)
        (implicit ordering: Ordering[T],
         f: T => Double): Try[DblVector] =
        Try(Stats[T](xt).normalize(low, high))
    ...
}
```

The first method of the `XTSeries` singleton generates a vector of a pair of elements by zipping the last $size - n$ elements of a time series with its first $size - n$ elements (line 1). The `statistics` (line 2) and `normalize` (line 3) methods operate on both the single and multivariable observations. These three methods are subsets of functionalities implemented in `XTSeries`.

Create a time series of the `XVSeries[T]` type by zipping the two `x` and `y` vectors and converting the pair into an array:

```
def zipToSeries[T: ClassTag] (
    x: Vector[T], y: Vector[T]): XVSeries[T]
```

Split a single or multidimensional time series, `xv`, into a two-time series at index, n :

```
def splitAt[T] (xv: XSeries[T], n: Int): (XSeries[T], XSeries[T])
```

Apply a `zScore` transform to a single dimension time series:

```
def zScore[T <: AnyVal] (xt: XSeries[T])
    (implicit f: T => Double): Try[DblVector]
```

Apply a `zScore` transform to a multidimension time series:

```
def zScores[T <: AnyVal](xt: XVSeries[T])
  (implicit f: T => Double): Try[XVSeries[Double]]
```

Transform a single dimension time series `x` into a new time series whose elements are $x(n) - x(n-1)$:

```
def delta(x: DblVector): DblVector
```

Transform a single dimension time series `x` into a new time series which elements if $(x(n) - x(n-1) > 0.0) 1 \text{ else } 0$:

```
def binaryDelta(x: DblVector): Vector[Int]
```

Compute the sum of the squared error between the two `x` and `z` arrays:

```
def sse[T <: AnyVal](x: Array[T], z: Array[T])
  (implicit f: T => Double): Double
```

Compute the mean squared error between the two `x` and `z` arrays:

```
def mse[T <: AnyVal](x: Array[T], z: Array[T])
  (implicit f: T => Double): Double
```

Compute the mean squared error between the two `x` and `z` vectors:

```
def mse(x: DblVector, z: DblVector): Double
```

Compute the statistics for each feature of a multidimensional time series:

```
def statistics[T <: AnyVal](xt: XVSeries[T])
  (implicit f: T => Double): Vector[Stats[T]]
```

Apply a `f` function to a zipped pair of multidimensional vectors of the `XVSeries` type:

```
def zipToVector[T](x: XVSeries[T], y: XVSeries[T])
  (f: (Array[T], Array[T]) => Double): XSeries[Double] =
  x.zip(y.view).map{ case (x, y) => f(x, y)}
```

The magnet pattern

Some operations on the time series that are implemented as the `XTSERIES` methods may have a large variety of input and output types. Scala and Java support method overloading that has the following limitations:

- It does not prevent the type collision caused by the erasure type in the JVM
- It does not allow lifting to a single, generic function
- It does not completely reduce code redundancy

The transpose operator

Let's consider the transpose operator for any kind of multidimensional time series. The transpose operator can be objectified as the `Transpose` trait:

```
sealed trait Transpose {
    type Result //4
    def apply(): Result //5
}
```

The trait has an abstract `Result` type (line 4) and an abstract `apply()` constructor (line 5) that allows us to create a generic `transpose` method with any kind of combination of input and output types. The conversion type for the input and output types of the `transpose` method is defined as `implicit`:

```
implicit def xvSeries2Matrix[T: ClassTag] (from: XVSeries[T]) =
    new Transpose { type Result = Array[Array[T]] //6
        def apply(): Result = from.toArray.transpose
    }
implicit def list2Matrix[T: ClassTag] (from: List[Array[T]]) =
    new Transpose { type Result = Array[Array[T]] //7
        def apply(): Result = from.toArray.transpose
    }
...
```

The first `xvSeries2Matrix` implicit transposes a time series of the `XVSeries[T]` type into a matrix with elements of the `T` type (line 6). The `list2Matrix` implicit transposes a time series of the `List[Array[T]]` type into a matrix with elements of the `T` type (line 7).

The generic `transpose` method is written as follows:

```
def transpose(tpose: Transpose): tpose.Result = tpose()
```

The differential operator

The second candidate for the magnet pattern is the computation of the differential in a time series. The purpose is to generate the time series $\{x_{t+1} - x_t\}$ from a time series $\{x_t\}$:

```
sealed trait Difference[T] {  
    type Result  
    def apply(f: (Double, Double) => T): Result  
}
```

The `Difference` trait allows us to compute the differential of a time series with arbitrary element types. For instance, the differential of a one-dimensional vector of the `Double` type is defined by the following implicit conversion:

```
implicit def vector2Double[T](x: DblVector) = new Difference[T] {  
    type Result = Vector[T]  
    def apply(f: (Double, Double) => T): Result = //8  
        zipWithShift(x, 1).collect{case(next,prev) =>f(prev,next)}  
}
```

The `apply()` constructor takes one argument: the user-defined `f` function that computes the difference between two consecutive elements of the time series (line 8). The generic difference method is as follows:

```
def difference[T] (  
    diff: Difference[T],  
    f: (Double, Double) => T): diff.Result = diff(f)
```

Here are some of the predefined differential operators of a time series for which the output of the operator has the `Double` (line 9), `Int` (line 10), and `Boolean` (line 11) types:

```
val diffDouble = (x: Double,y: Double) => y -x //9  
val diffInt = (x: Double,y: Double) => if(y > x) 1 else 0 //10  
val diffBoolean = (x: Double,y: Double) => (y > x) //11
```

The differential operator is used to implement the `labeledData` method to generate labeled data from observations with two features and a target (labels) dataset:

```
def differentialData[T] (  
    x: DblVector,  
    y: DblVector,  
    target: DblVector,  
    f: (Double,Double) =>T): Try[(XVSeries[Double],Vector[T])] =  
    Try((zipToSeries(x,y), difference(target, f)))
```

The structure of the labeled data is the pair of observations and the differential of target values.

Lazy views

A view in Scala is a proxy collection that represents a collection but implements the data transformation or higher-order method lazily. The elements of a view are defined as lazy values, which are instantiated on demand.

One important advantage of views over a **strict** (or fully allocated) collection is the reduced memory consumption.

Let's take a look at the `aggregator` data transformation introduced in the *Instantiating the workflow* section under *A workflow computational model* in Chapter 2, *Hello World!*. There is no need to allocate the entire set of `x.size` of elements: the higher-order `find` method may exit after only a few elements have been read (line 12):

```
val aggregator = new ETransform[Int] (splits) {
    override def |> : PartialFunction[U, Try[V]] = {
        case x: U if (!x.isEmpty) =>
            Try( Range(0, x.size).view.find(x(_) == 1.0).get) //12
    }
}
```

Views, iterators, and streams

Views, iterators, and streams share the same objective of constructing elements on demand. There are, however, some major differences:

- Iterators do not persist elements of the collection (read once)
- Streams allow operations to be performed on the collection with an undefined size

Moving averages

Moving averages provide data analysts and scientists with a basic predictive model. Despite its simplicity, the moving average method is widely used in a variety of fields, such as marketing survey, consumer behavior, or sport statistics. Traders use the moving average method to identify different levels of support and resistance for the price of a given security.

An average reducing function

Let's consider the time series $x_t = x(t)$ and function $f(x_{t-p}, \dots, x_t)$ that reduces the last p observations into a value or average. The estimation of the observation at t is defined by the following formula:



$$\tilde{x}_t = f(x_{t-p+1}, \dots, x_t) \quad \forall t \geq p$$

Here, f is an average reducing function from the previous p data points.

The simple moving average

The simple moving average is the simplest form of the moving averages algorithms [3:1]. The simple moving average of period p estimates the value at time t by computing the average value of the previous p observations using the following formula.

The simple moving average

M1: The simple moving average of a time series $\{x_t\}$ with a period p is computed as the average of the last p observations:



$$\tilde{x}_t = \frac{1}{p} \sum_{j=t-p+1}^t x_j \quad \forall t \geq p$$

$$0 \quad \forall t < p$$

M2: The computation is implemented iteratively using the following formula:

$$\tilde{x}_t = \tilde{x}_{t-1} + \frac{1}{p} (x_t - x_{t-p}) \quad \forall t \geq p$$

Here, \tilde{x}_t is the estimate or simple moving average value at time t .

Let's build a class hierarchy of moving average algorithms, with the parameterized `MovingAverage` trait as its root:

```
trait MovingAverage[T]
```

We use the generic `XSeries[T]` type and the data transform with the `ETransform` explicit configuration, introduced in the *Explicit models* section under *Monadic data transformation* in *Chapter 2, Hello World!*, to implement the simple moving average, `SimpleMovingAverage`:

```
class SimpleMovingAverage[T <: AnyVal](period: Int)
    (implicit num: Numeric[T], f: T => Double) //1
    extends Etransform[Int](period) with MovingAverage[T] {

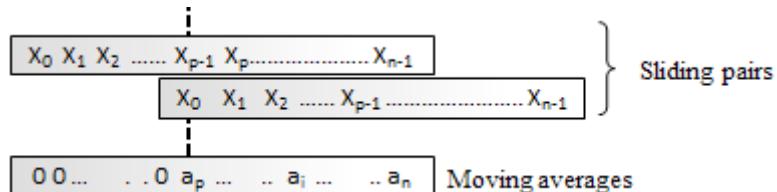
    type U = XSeries[T] //2
    type V = DblVector //3

    val zeros = Vector.fill(0.0)(period-1)
    override def |> : PartialFunction[U, Try[V]] = {
        case xt: U if( xt.size >= period ) => {
            val splits = xt.splitAt(period)
            val slider = xt.take(xt.size - period).zip(splits._2) //4

            val zero = splits._1.sum/period //5
            Try( zeros ++ slider.scanLeft(zero) {
                case (s, (x,y)) => s + (x - y)/period }) //7
        }
    }
}
```

The class is parameterized for the `T` type of elements of the input time series; *we cannot make any assumption regarding the type of input data*. The type of the elements of the output time series is `Double`. The implicit instantiation of the `Numeric[T]` class is required by the `sum` and / arithmetic operators (line 1). The simple moving average implements `ETransform` by defining the abstract `U` types for the input (line 2) and `V` for the output (line 3) as a time series, `DblVector`.

The implementation has a few interesting elements. First, the set of observations is duplicated and the index in the resulting clone instance is shifted by `p` observations before being zipped with the original to the array of a pair of `slider` values (line 4):



The sliding algorithm to compute moving averages

The average value is initialized with the mean value of the first period data points (line 5). The first period values of the trends are initialized to zero (line 6). The method concatenates the initial null values and the computed average values to implement the **M2** formula (line 7).

The weighted moving average

The weighted moving average method is an extension of the simple moving average by computing the weighted average of the last p observations [3:2]. The weights a_j are assigned to each of the last p data points x_j and are normalized by the sum of the weights.

The weighted moving average

M3: The weighted moving average of a series $\{x_t\}$ with a period p and a normalized weights distribution $\{a_j\}$ is given by the following formula:



$$\tilde{x}_t = \sum_{j=t-p+1}^t a_{j-t+p-1} x_j \quad \forall t \geq p \quad \text{subject to } \sum_{i=0}^{p-1} a_i = 1$$

0 $\forall t < p$

Here, \tilde{x}_t is the estimate or simple moving average value at time t .

The implementation of the `WeightedMovingAverage` class requires the computation of the last p (`weights.size`) data points. There is no simple iterative formula to compute the weighted moving average at time $t + 1$ using the moving average at time t :

```
class WeightedMovingAverage[@specialized(Double) T <: AnyVal] (
    weights: DblArray)
    (implicit num: Numeric[T], f: T => Double)
extends SimpleMovingAverage[T](weights.length) { //8

    override def |> : PartialFunction[U, Try[V]] = {
        case xt: U if(xt.size >= weights.length) => {
            val smoothed = (config to xt.size).map(i =>
                xt.slice(i - config, i).zip(weights) //9
                    .map { case(x, w) => x * w }.sum //10
            )
            Try(zeros ++ smoothed) //11
        }
    }
}
```

The computation of the weighted moving average is a bit more involved than the simple moving average. Therefore, we specify the generation of the byte code that is dedicated to the `Double` type using the specialized annotation. The weighted moving average inherits the `SimpleMovingAverage` class, and therefore, implements the `ETransform` explicit transformation for a configuration of weights, with input observations of the `XSeries[T]` type and output observations of the `DblVector` type. The implementation of `M3` formula generates a smoothed time series by slicing (line 9) the input time series and then computing the inner product of weights and the slice of the time series (line 10).

As with the simple moving average, the output is the concatenation of the initial `weights.size` null values, `zeros`, and the smoothed data (line 11).

The exponential moving average

The exponential moving average is widely used in financial analysis and marketing surveys because it favors the latest values. The older the value, the less impact it has on the moving average value at time t [3:3].

The exponential moving average

M4: The exponential moving average of a series $\{x_t\}$ and smoothing factor α is computed by the following iterative formula:

$$\begin{aligned}\tilde{x}_t &= (1 - \alpha)\tilde{x}_{t-1} + \alpha x_t \quad \forall t > 0 \quad 0 < \alpha < 1 \\ \tilde{x}_0 &= x_0\end{aligned}$$

Here, \tilde{x} is the value of the exponential average at t .

The implementation of the `ExpMovingAverage` class is rather simple. The constructor has a single α argument (the decay rate) (line 12):

```
class ExpMovingAverage[@specialized(Double) T <: AnyVal] (
  alpha: Double)      //12
  (implicit f: T => Double)
extends ETransform[Double](alpha) with MovingAverage[T] { //13

  type U = XSeries[T]      //14
  type V = DblVector       //15

  override def |> : PartialFunction[U, Try[V]] = {
    case xt: U if( xt.size > 0) => {
      val alpha_1 = 1-alpha
      var y: Double = data(0)
      Try( xt.view.map(x => {
        val z = x*alpha + y*alpha_1; y = z; z})) //16
    }
  }
}
```

```
    }
}
}
```

The exponential moving average implements the `ETransform` (line 13) by defining the abstract `U` types for the input (line 14) as a time series named `xseries[T]` and `V` for the output (line 15) as a time series named `DblVector`. The `|>` method applies the **M4** formula to all the observations of the time series within a `map` (line 16).

The version of the constructor that uses the `p` period to compute $\alpha = 1/(p+1)$ as an argument is implemented using the Scala `apply` method:

```
def apply[T <: AnyVal] (p: Int)
  (implicit f: T => Double): ExpMovingAverage[T] =
  new ExpMovingAverage[T] (2/(p + 1))
```

Let's compare the results generated from these three moving averages methods with the original price. We use a data source, `DataSource`, to load and extract values from the historical daily closing stock price of Bank of America (BAC), which is available at the Yahoo Financials pages. The `DataSink` class is responsible for formatting and saving the results into a CSV file for further analysis. The `DataSource` and `DataSink` classes are described in detail in the *Data extraction* section in the *Appendix A, Basic Concepts*:

```
import YahooFinancials._
val hp = p >>1
val w = Array.tabulate(p) (n =>
  if(n == hp) 1.0 else 1.0/(Math.abs(n - hp)+1)) //17
val sum = w.sum
val weights = w.map { _ / sum } //18

val dataSrc = DataSource(s"$RESOURCE_PATH$symbol.csv", false)//19
val pfnsMvAve = SimpleMovingAverage[Double] (p) |> //20
val pfwnMvAve = WeightedMovingAverage[Double] (weights) |>
val pfneMvAve = ExpMovingAverage[Double] (p) |>

for {
  price <- dataSrc.get(adjClose) //21
  if(pfnsMvSve.isDefinedAt(price) )
  sMvOut <- pfnsMvAve(price) //22
  if(pfwnMvSve.isDefinedAt(price)
  eMvOut <- pfwnMvAve(price)
  if(pfneMvSve.isDefinedAt(price)
  wMvOut <- pfneMvAve(price)
} yield {
  val dataSink = DataSink[Double] (s"$OUTPUT_PATH$p.csv")
  val results = List[DblSeries] (price, sMvOut, eMvOut, wMvOut)
  dataSink |> results //23
}
```

 **isDefinedAt**

Each of the partial function is validated by a call to `isDefinedAt`. From now on, the validation of a partial function will be omitted throughout the book for the sake of clarity.

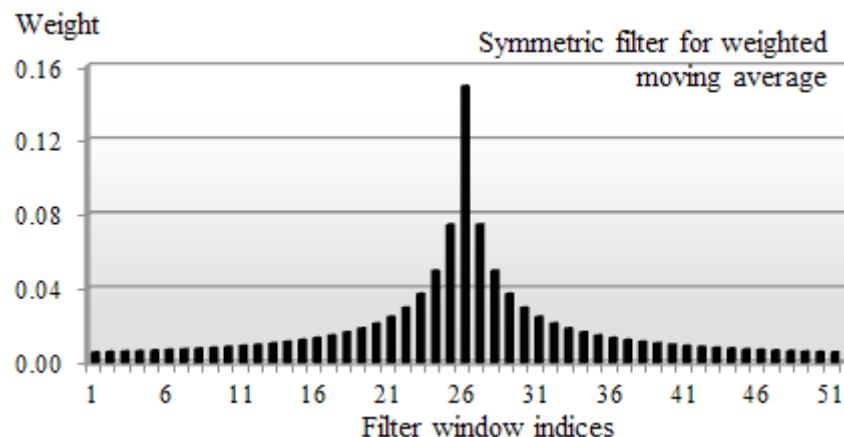
The coefficients for the weighted moving average are generated (line 17) and normalized (line 18). The trading data regarding the ticker symbol, BAC, is extracted from the Yahoo Finances CSV file (line 19), `YahooFinancials`, using the `adjClose` extractor (line 20). The next step is to initialize the `pfnSMvAve`, `pfnWMvAve`, and `pfnEMvAve` partial functions related to each of the moving average (line 21). The invocation of the partial functions with `price` as an argument generates the three smoothed time series (line 22).

Finally, a `DataSink` instance formats and dumps the results into a file (line 23).

 **Implicit postfixOps**

The instantiation of the `filter` $|>$ partial function requires that the post fix operation, `postfixOps`, be made visible by importing `scala.language.postfixOps`.

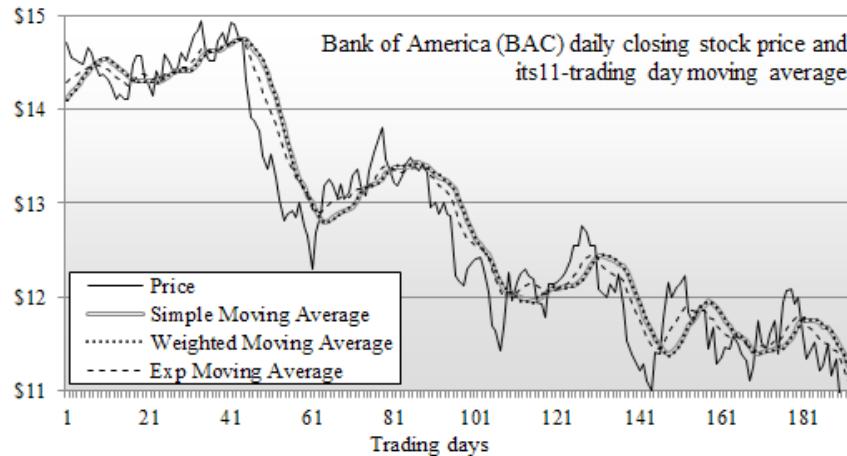
The weighted moving average method relies on a symmetric distribution of normalized weights computed by a function passed as an argument of the generic `tabulate` method. Note that the original price time series is displayed if one of the specific moving averages cannot be computed. The following graph is an example of a symmetric filter for weighted moving averages:



An example of a symmetric filter for weighted moving averages

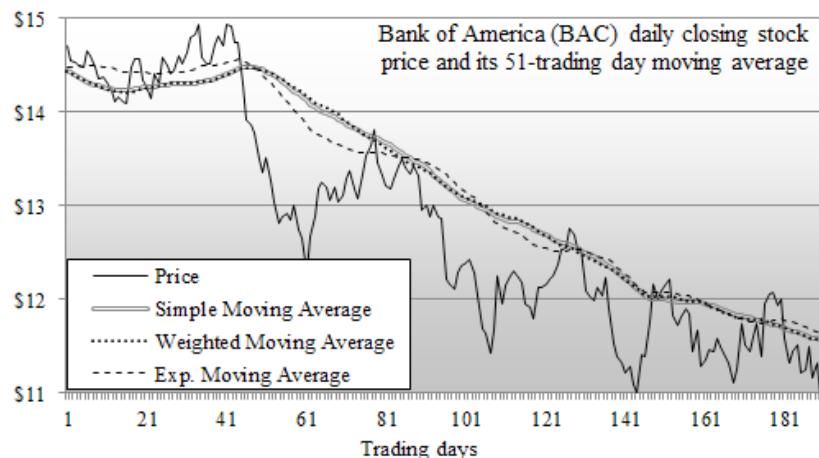
Data Preprocessing

The three moving average techniques are applied to the price of the stock of Bank of America stock (BAC) over 200 trading days. Both the simple and weighted moving averages use a period of 11 trading days. The exponential moving average method uses a scaling factor of $2/(11+1) = 0.1667$:



11-day moving averages of the historical stock price of Bank of America

The three techniques filter the noise out of the original historical price time series. The exponential moving average reacts to a sudden price fluctuation despite the fact that the smoothing factor is low. If you increase the period to 51 trading days (which is equivalent to two calendar months), the simple and weighted moving averages produce a time series smoother than the exponential moving average with $\alpha = 2/(p+1) = 0.038$:



51-day moving averages of the historical stock price of Bank of America

You are invited to experiment further with different smooth factors and weight distributions. You will be able to confirm the following basic rule: as the period of the moving average increases, noise with decreasing frequencies is eliminated. In other words, the window of allowed frequencies is shrinking. The moving average acts as a **low-pass filter** that preserves only lower frequencies.

Fine-tuning the period of a smoothing factor is time consuming. Spectral analysis, or more specifically, Fourier analysis transforms a time series into a sequence of frequencies, which provide the statistician with a more powerful frequency analysis tool.

The moving average on a multidimensional time series

The moving average techniques are presented for a single feature or variable time series, for the sake of simplicity. Moving averages on multidimensional time series are computed by executing a single variable moving average for each feature using the `transform` method of `XTSeries`, which is introduced in the first section. For example, the simple moving average applied to a multidimensional time series, `xt`. The smoothed values are computed as follows:

```
val pfnMv = SimpleMovingAverage[Double] (period) |>
  val smoothed = transform(xt, pfnMv)
```



Fourier analysis

The purpose of **spectral density estimation** is to measure the amplitude of a signal or a time series according to its frequency [3:4]. The objective is to estimate the spectral density by detecting periodicities in the dataset. A scientist can better understand a signal or time series by analyzing its harmonics.

The spectral theory

Spectral analysis for a time series should not be confused with the spectral theory, a subset of linear algebra that studies eigenfunctions on **Hilbert** and **Banach** spaces. In fact, harmonic analysis and Fourier analysis are regarded as subsets of the spectral theory.



Let's explore the concept behind the discrete Fourier series as well as its benefits as applied to financial markets. The **Fourier analysis** approximates any generic function as the sum of trigonometric functions, sine and cosine.

Complex Fourier transform

This section focuses on the discrete Fourier series for real values. The generic Fourier transform applies to complex values [3:5].

The decomposition in a basic trigonometric function process is known as the **Fourier transform** [3:6].

Discrete Fourier transform

A time series $\{x_i\}$ can be represented as a discrete real-time domain function f , $x = f(t)$. In the 18th century, Jean Baptiste Joseph Fourier demonstrated that any continuous periodic function f can be represented as a linear combination of sine and cosine functions. The **discrete Fourier transform (DFT)** is a linear transformation that converts a time series into a list of coefficients of a finite combination of complex or real trigonometric functions, ordered by their frequencies.

The frequency ω of each trigonometric function defines one of the harmonics of the signal. The space that represents the signal amplitude versus frequency of the signal is known as the **frequency domain**. The generic DFT transforms a time series into a sequence of frequencies defined as complex numbers $a + j\varphi$ ($j^2 = -1$), where a is the amplitude of the frequency and φ is the phase.

This section is dedicated to the real DFT that converts a time series into an ordered sequence of frequencies with real values.

Real discrete Fourier transform

M5: A periodic function f can be represented as an infinite combination of sine and cosine functions:

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(nx) + \sum_{k=1}^{\infty} b_k \sin(nx)$$

M6: The Fourier cosine transform of a function f is defined as:

$$\mathcal{F}^c(f, k) = \int_{-\infty}^{\infty} \cos(2\pi kx) f(x) dx$$

M7: The discrete real cosine series of a function $f(-x) = f(x)$ is defined as:

$$f(x) = f(-x) = \frac{a_0}{2} + \sum_{k=1}^{2N-3} a_k \cos(kx) \quad \text{where } a_k = \frac{2}{\pi} \int_0^{\pi} f(t) \cos(kt) . dt$$

M8: The Fourier sine transform of a function is defined as:



$$\mathcal{F}^s(f, k) = \int_{-\infty}^{\infty} \sin(2\pi kx) f(x) dx$$

M9: The discrete real sine series of a function $f(-x) = f(x)$ is defined as:

$$f(x) = -f(-x) = \sum_{k=1}^{2N-3} b_k \sin(kx) \quad \text{where } b_k = \frac{2}{\pi} \int_0^{\pi} f(t) \sin(kt) . dt$$

The computation of the Fourier trigonometric series is time consuming with an asymptotic time complexity of $O(n^2)$. Scientists and mathematicians have been working to make the computation as effective as possible. The most common numerical algorithm used to compute the Fourier series is the **Fast Fourier Transform (FFT)** created by J.W. Cooley and J. Tukey [3:7].

The algorithm called Radix-2 version recursively breaks down the Fourier transform for a time series of N data points into any combination of N_1 and N_2 sized segments such as $N = N_1 N_2$. Ultimately, the discrete Fourier transform is applied to the deeper-nested segments.



The Cooley-Tukey algorithm

I encourage you to implement the Radix-2 Cooley-Tukey algorithm in Scala using a tail recursion.

The Radix-2 implementation requires that the number of data points is $N=2^n$ for even functions (sine) and $N=2^n+1$ for cosine. There are two approaches to meet this constraint:

- Reduce the actual number of points to the next lower radix, $2^n < N$
- Extend the original time series by padding it with 0 to the next higher radix, $N < 2^n+1$

Padding the time series is the preferred option because it does not affect the original set of observations.

Let's define a `DTransform` trait for any variant of the discrete Fourier transform. The first step is to wrap the default configuration parameters used in the Apache Commons Math library into a `Config` singleton:

```
trait DTransform {
    object Config {
        final val FORWARD = TransformType.FORWARD
        final val INVERSE = TransformType.INVERSE
        final val SINE = DstNormalization.STANDARD_DST_I
        final val COSINE = DctNormalization.STANDARD_DCT_I
    }
    ...
}
```

The main purpose of the `DTransform` trait is to pad the `vec` time series with zero values:

```
def pad(vec: DblVector,
       even: Boolean = true)(implicit f: T => Double): DblArray = {
    val newSize = padSize(vec.size, even) //1
    val arr: DblVector = vec.map(_.toDouble)
    if( newSize > 0) arr ++ Array.fill(newSize)(0.0) else arr //2
}

def padSize(xtSz: Int, even: Boolean= true): Int = {
    val sz = if( even ) xtSz else xtSz-1 //3
    if( (sz & (sz-1)) == 0) 0
    else {
        var bitPos = 0
        do { bitPos += 1 } while( (sz >> bitPos) > 0) //4
        (if(even) (1<<bitPos) else (1<<bitPos)+1) - xtSz
    }
}
```

The `pad` method computes the optimal size of the frequency vector as 2^N by invoking the `padSize` method (line 1). It then concatenates the padding with the original time series or vector of observations (line 2). The `padSize` method adjusts the size of the data depending on whether the time series has initially an even or odd number of observations (line 3). It relies on bit operations to find the next radix, N (line 4).

The while loop



Scala developers prefer Scala higher-order methods for collections to implement the iterative computation. However, nothing prevents you from using the traditional `while` or `do { ... } while` loop if either readability or performance is an issue.

The fast implementation of the padding method, `pad`, consists of detecting the number of N observations as a power of 2 (the next highest radix). The method evaluates if N and $(N-1)$ are zero after it shifts the number of bits in the value, N . The code illustrates the effective use of implicit conversion to make the code readable in the `pad` method:

```
val arr: DblVector = vec.map(_.toDouble)
```

The next step is to write the `DFT` class for the real sine and cosine discrete transforms by subclassing `DTransform`. The class relies on the padding mechanism implemented in `DTransform` whenever necessary:

```
class DFT[@specialized(Double) T <: AnyVal] (
    eps: Double)(implicit f: T => Double)
    extends ETransform[Double](eps) with DTransform { //5
    type U = XSeries[T] //6
    type V = DblVector

    override def |> : PartialFunction[U, Try[V]] = { //7
        case xv: U if(xv.size >= 2) => fwrd(xv).map(_.toArray)
    }
}
```

We treat the discrete Fourier transform as a transformation on the time series using an explicit `ETransform` configuration (line 5). The `U` data type of the input and the `V` type of the output have to be defined (line 6). The `|>` transformation function delegates the computation to the `fwrd` method (line 7):

```
def fwrd(xv: U): Try[(RealTransformer, DblArray)] = {
    val rdt = if(Math.abs(xv.head) < config) //8
        new FastSineTransformer(SINE) //9
    else new FastCosineTransformer(COSINE) //10

    val padded = pad(xv.map(_.toDouble), xv.head == 0.0).toArray
    Try( (rdt, rdt.transform(padded, FORWARD)) )
}
```

The `fwd` method selects the discrete Fourier sine series if the first value of the time series is 0.0, otherwise it selects the discrete cosine series. This implementation automates the selection of the appropriate series by evaluating `xt.head` (line 8). The transformation invokes the `FastSineTransformer` (line 9) and `FastCosineTransformer` (line 10) classes of the Apache Commons Math library [3:8] introduced in the first chapter.

This example uses the standard formulation of the cosine and sine transformations, defined by the `COSINE` argument. The orthogonal normalization that normalizes the frequency by a factor of $1/\sqrt{2(N-1)}$, where N is the size of the time series, generates a cleaner frequency spectrum for a higher computation cost.

The `@specialized` annotation

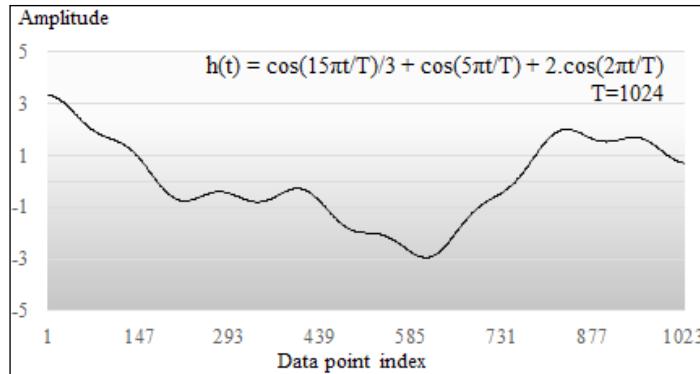
The `@specialized(Double)` annotation is used to instruct the Scala compiler to generate a specialized and more efficient version of the class for the `Double` type. The drawback of the specialization is the duplication of byte code as the specialized version coexists with the parameterized classes [3:9].

In order to illustrate the different concepts behind DFTs, let's consider the case of a time series generated by a `h` sequence of sinusoidal functions:

```
val F = Array[Double] (2.0, 5.0, 15.0)
val A = Array[Double] (2.0, 1.0, 0.33)

def harmonic(x: Double, n: Int): Double =
    A(n)*Math.cos(Math.PI*F(n)*x)
val h = (x: Double) =>
    Range(0, A.size).aggregate(0.0)((s, i) =>
        s + harmonic(x, i), _ + _)
```

As the signal is synthetically created, we can select the size of the time series to avoid padding. The first value in the time series is not null, so the number of observations is 2^n+1 . The data generated by the `h` function is plotted as follows:



An example of sinusoidal time series

Let's extract the frequencies' spectrum for the time series generated by the `h` function. The data points are created by tabulating the `h` function. The frequencies spectrum is computed with a simple invocation of the explicit `| >` data transformation of the DFT class:

```

val OUTPUT1 = "output/chap3/simulated.csv"
val OUTPUT2 = "output/chap3/smoothed.csv"
val FREQ_SIZE = 1025; val INV_FREQ = 1.0/FREQ_SIZE

val pfnDFT = DFT[Double] |> //11
for {
    values <- Try(Vector.tabulate(FREQ_SIZE)
                  (n => h(n*INV_FREQ))) //12
    output1 <- DataSink[Double](OUTPUT1).write(values)
    spectrum <- pfnDFT(values)
    output2 <- DataSink[Double](OUTPUT2).write(spectrum) //13
} yield {
    val results = format(spectrum.take(DISPLAY_SIZE),
    "x/1025", SHORT)
    show(s"$DISPLAY_SIZE frequencies: $results")
}

```

The execution of the data simulator follows these steps:

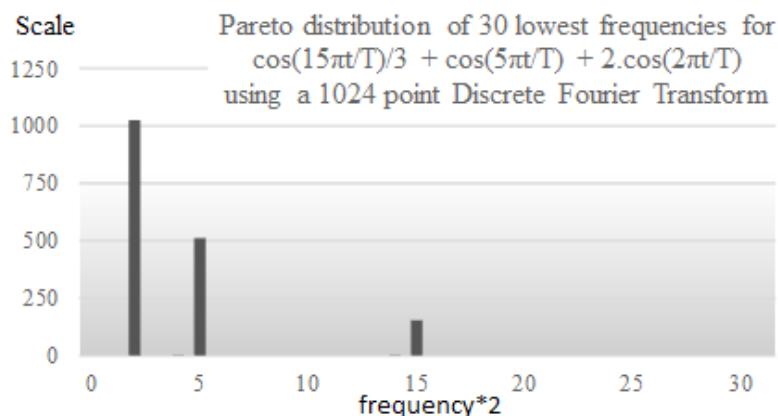
1. Generate a raw data with the 3-harmonic `h` function (line 12).
2. Instantiate the partial function generated by the transformation (line 11).
3. Store the resulting frequencies in a data sink (filesystem) (line 13).

Data sinks and spreadsheets



In this particular case, the results of the discrete Fourier transform are dumped into a CSV file so that it can be loaded into a spreadsheet. Some spreadsheets support a set of filtering techniques that can be used to validate the result of the example. A simpler alternative would be to use JFreeChart.

The spectrum of frequencies of the time series, plotted for the first 32 points, clearly shows three frequencies at $k = 2, 5$, and 15 . This result is expected because the original signal is composed of three sinusoidal functions. The amplitude of these frequencies are $1024/1$, $1024/2$, and $1024/6$, respectively. The following plot represents the first 32 harmonics for the time series:



The frequency spectrum for a three-frequency sinusoidal

The next step is to use the frequencies spectrum to create a low-pass filter using DFT. There are many algorithms available to implement a low or pass band filter in the time domain, from autoregressive models to the Butterworth algorithm. However, the discrete Fourier transform is still a very popular technique to smooth signals and identify trends.

Big Data



A DFT for a large time series can be very computation intensive. One option is to treat the time series as a continuous signal and sample it using the **Nyquist** frequency. The Nyquist frequency is half of the sampling rate of a continuous signal.

DFT-based filtering

The purpose of this section is to introduce, describe, and implement a noise filtering mechanism that leverages the discrete Fourier transform. The idea is quite simple: the forward and inverse Fourier series are used sequentially to convert the raw data from the time domain to the frequency domain and back. The only input you need to supply is a function g that modifies the sequence of frequencies. This operation is known as the convolution of the filter g and the frequencies' spectrum.

A convolution is similar to an inner product of two time series in the frequencies domain. Mathematically, the convolution is defined as follows:

Convolution

M10: The convolution of two functions f and g is defined as:

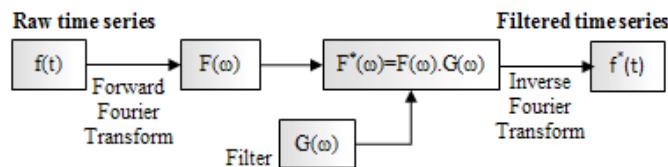
$$\langle f, g \rangle = \int_{-\infty}^{\infty} f(t) \cdot g(x-t) dt$$

M11: The convolution F of a time series $x = \{x_i\}$ with a frequency spectrum ω^x and a filter f in frequency domain ω^f is defined as:

$$F(x * f) = F(x) \cdot F(f) = \sum_{j=0}^{N-1} \omega_j^x \cdot \omega_{k-j}^f$$

Let's apply the convolution to our filtering problem. The filtering algorithm using the discrete Fourier transform consists of five steps:

1. Pad the time series to enable the discrete sine or cosine transform.
2. Generate the ordered sequence of frequencies using the forward transform F .
3. Select the filter function G in the frequency domain and a cutoff frequency.
4. Convolute the sequence of frequency with the filter function G .
5. Generate the filtered signal in the time domain by applying the inverse DFT transform to the convoluted frequencies.



A diagram of the discrete Fourier filter

The most commonly used low-pass filter functions are known as the `sinc` and `sinc2` functions, which are defined as a rectangular function and triangular function, respectively. These functions are partially applied functions that are derived from a generic `convol` method. The simplest `sinc` function returns 1 for frequencies below a cutoff frequency, `fC`, and 0 if the frequency is higher:

```
val convol = (n: Int, f: Double, fC: Double) =>
  if( Math.pow(f, n) < fC) 1.0 else 0.0
val sinc = convol(1, _: Double, _:Double)
val sinc2 = convol(2, _: Double, _:Double)
val sinc4 = convol(4, _: Double, _:Double)
```

Partially applied functions versus partial functions

Partial functions and partially applied functions are not actually related.

A partial function f' is a function that is applied to a subset X' of the input space X . It does not execute all possible input values:



$$f: X \rightarrow Y \quad X' \subset X \quad f': X' \rightarrow Y$$

A partially applied function f'' is a function value for which the user supplies the value for one or more arguments. The projection reduces the dimension of the input space (X, Z) :

$$f: (X, Z) \rightarrow Y \quad f'': X \rightarrow Y$$

The `DFTFilter` class inherits from the `DFT` class in order to reuse the `fwrd` forward transform function. The `g` frequency domain function is an attribute of the filter. The `g` function takes the `fC` frequency cutoff value as the second argument (line 14). The two `sinc` and `sinc2` filters defined in the previous section are examples of filtering functions:

```
class DFTFilter[@specialized(Double) T <: AnyVal] (
  fC: Double,
  eps: Double)
  (g: (Double, Double) =>Double) (implicit f: T => Double)
extends DFT[T](eps) { //14

  override def |> : PartialFunction[U, Try[V]] = {
    case xt: U if( xt.size >= 2 ) => {
      fwrd(xt).map{ case(trf, freq) => { //15
        val cutOff = fC*freq.size
        val filtered = freq.zipWithIndex
          .map{ case(x, n) => x*g(n, cutOff) } //16
        trf.transform(filtered, INVERSE).toVector } //17
      }
    }
  }
}
```

The filtering process follows three steps:

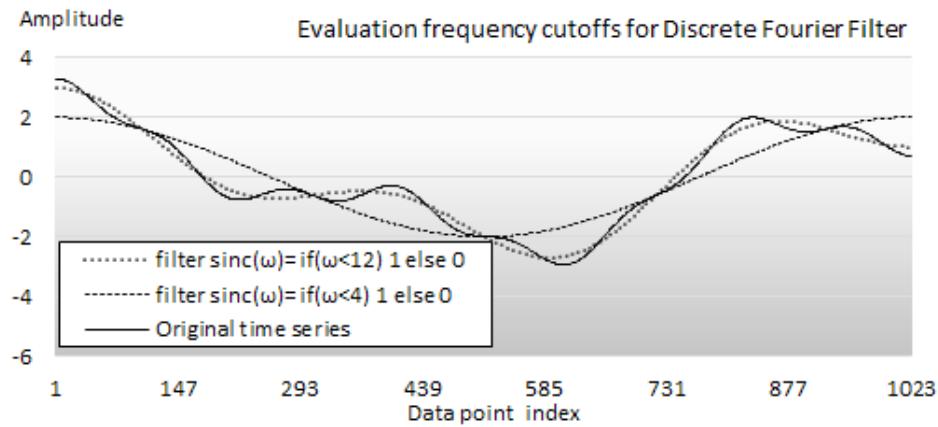
1. Computation of the `fwrd` discrete Fourier forward transformation (sine or cosine) (line 15).
2. Apply the filter function (formula M11) through a Scala `map` method (line 16).
3. Apply the inverse transform to the frequencies (line 17).

Let's evaluate the impact of the cutoff values on the filtered data. The implementation of the test program consists of loading the data from the file (line 19) and then invoking the `DFTFilter` of the `pfnDFTfilter` partial function (line 19):

```
import YahooFinancials._

val inputFile = s"$RESOURCE_PATH$symbol.csv"
val src = DataSource(input, false, true, 1)
val CUTOFF = 0.005
val pfnDFTfilter = DFTFilter[Double](CUTOFF)(sinc) |>
for {
    price <- src.get(adjClose) //18
    filtered <- pfnDFTfilter(price) //19
}
yield { /* ... */ }
```

Filtering out the noise is accomplished by selecting the cutoff value between any of the three harmonics with the respective frequencies of 2, 5, and 15. The original and the two filtered time series are plotted on the following graph:



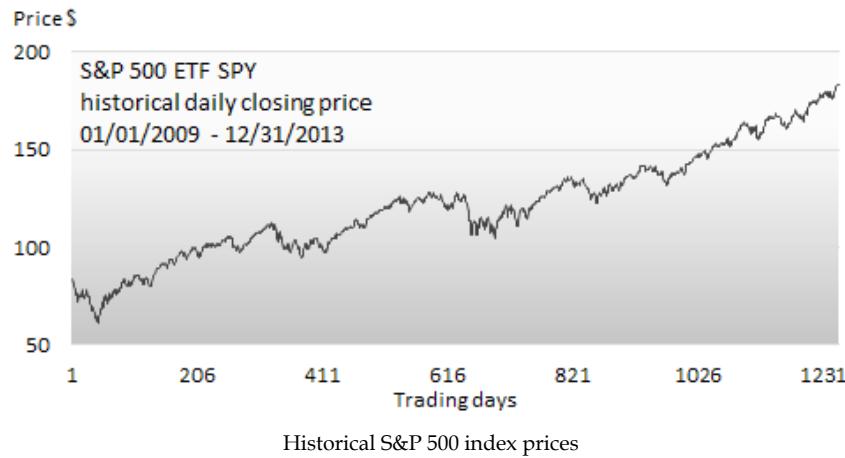
Plotting of the discrete Fourier filter-based smoothing

As you would expect, the low-pass filter with a cutoff value of 12 eliminates the noise with the highest frequencies. The filter with the cutoff value 4 cancels out the second harmonic (low-frequency noise), leaving out only the main trend cycle.

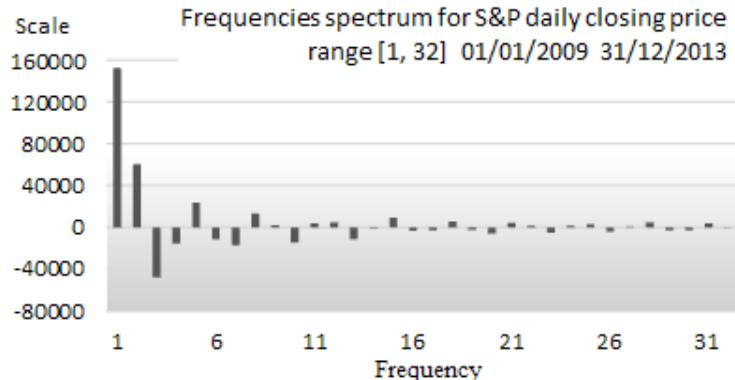
Detection of market cycles

Using the discrete Fourier transform to generate the frequencies spectrum of a periodical time series is easy. However, what about real-world signals such as the time series that represent the historical price of a stock?

The purpose of the next exercise is to detect, if any, the long term cycle(s) of the overall stock market by applying the discrete Fourier transform to the quote of the S&P 500 index between January 1, 2009 and December 31, 2013, as illustrated in the following graph:



The first step is to apply the DFT to extract a frequencies spectrum for the S&P 500 historical prices, as shown in the following graph, with the first 32 harmonics:



Frequencies spectrum for the historical S&P index

The frequency domain chart highlights some interesting characteristics regarding the S&P 500 historical prices:

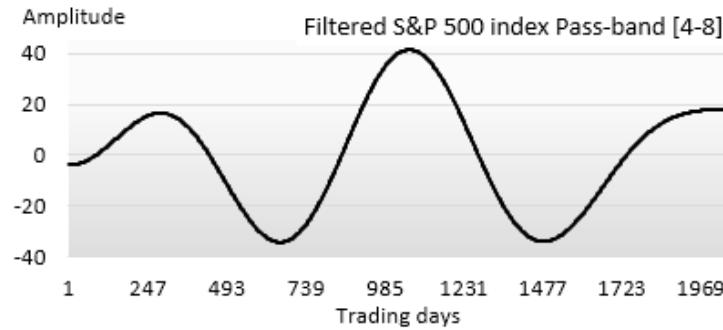
- Both positive and negative amplitudes are present, as you would expect in a time series with complex values. The cosine series contributes to the positive amplitudes while the sine series affects both positive and negative amplitudes, ($\cos(x+\pi) = \sin(x)$).
- The decay of the amplitude along the frequencies is steep enough to warrant further analysis beyond the first harmonic, which represents the main trend of the historical stock price. The next step is to apply a band-pass filter technique to the S&P 500 historical data in order to identify short-term trends with lower periodicity.

A low-pass filter is limited to reduce or cancel out the noise in the raw data. In this case, a band-pass filter using a range or window of frequencies is appropriate to isolate the frequency or the group of frequencies that characterize a specific cycle. The `sinc` function, which was introduced in the previous section to implement a low-pass filter, is modified to enforce the band-pass filter within a window, $[w_1, w_2]$, as follows:

```
def sinc(f: Double, w: (Double, Double)): Double =
  if(f > w._1 && f < w._2) 1.0 else 0.0
```

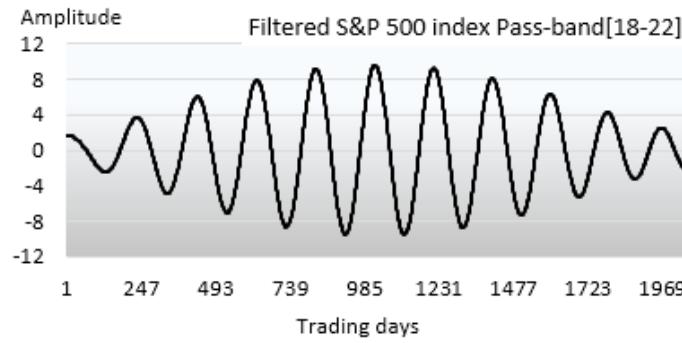
Data Preprocessing

Let's define a DFT-based band-pass filter with a window of width 4, $w=(i, i+4)$, with i ranging between 2 and 20. Applying the window [4, 8] isolates the impact of the second harmonic on the price. As we eliminate the main upward trend with frequencies less than 4, all the filtered data varies within a short range relative to the main trend. The following graph shows the output of this filter:



The output of a band-pass DFT filter range 4-8 on the historical S&P index

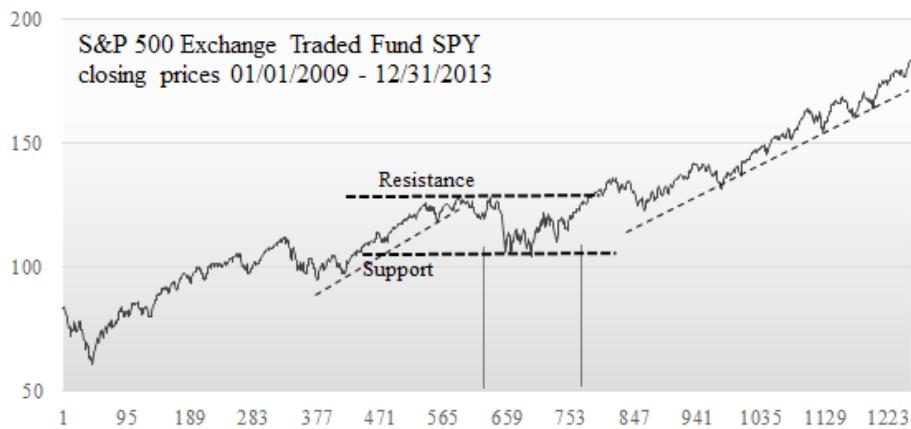
In this case, we filter the S&P 500 index around the third group of harmonics with frequencies ranging from 18 to 22; the signal is converted into a familiar sinusoidal function, as shown here:



The output of a band-pass DFT filter range 18-22 on the historical S&P index

There is a possible rational explanation for the shape of the S&P 500 data filtered by a band-pass filter with a frequency of 20, as illustrated in the previous graph. The S&P 500 historical data plot shows that the frequency of the fluctuation in the middle of the uptrend (trading sessions 620 to 770) increases significantly.

This phenomenon can be explained by the fact that the S&P 500 index reaches a resistance level around the trading session 545 when the existing uptrend breaks. A tug of war starts between the bulls, betting the market nudges higher, and the bears, who are expecting a correction. The back and forth between the traders ends when the S&P 500 index breaks through its resistance and resumes a strong uptrend characterized by a high amplitude low frequency, as shown in the following graph:



An illustration of support and resistance levels for the historical S&P 500 index prices

One of the limitations of using the discrete Fourier-based filters to clean up data is that it requires the data scientist to extract the frequencies spectrum and modify the filter on a regular basis, as he or she is never sure that the most recent batch of data does not introduce noise with a different frequency. The Kalman filter addresses this limitation.

The discrete Kalman filter

The Kalman filter is a mathematical model that provides an accurate and recursive computation approach to estimate the previous states and predict the future states of a process for which some variables may be unknown. R.E. Kalman introduced it in the early 60s to model dynamics systems and predict a trajectory in aerospace [3:10]. Today, the Kalman filter is used to discover a relationship between two observed variables that may or may not be associated with other hidden variables. In this respect, the Kalman filter shares some similarities with the Hidden Markov models, as described in the *The hidden Markov model* section in *Chapter 7, Sequential Data Models* [3:11].

The Kalman filter is used as:

- A predictor of the next data point from the current observation
- A filter that weeds out noise by processing the last two observations
- A smoothing model that identifies trends from a history of observations

Smoothing versus filtering

 Smoothing is an operation that removes high-frequency fluctuations from a time series or signal. Filtering consists of selecting a range of frequencies to process the data. In this regard, smoothing is somewhat similar to low-pass filtering. The only difference is that a low-pass filter is usually implemented through linear methods.

Conceptually, the Kalman filter estimates the state of a system from noisy observations. The Kalman filter has two characteristics:

- **Recursive:** A new state is predicted and corrected using the input of a previous state
- **Optimal:** This is an optimal estimator because it minimizes the mean square error of the estimated parameters (against actual values)

The Kalman filter is one of the stochastic models that are used in adaptive control [3:12].

Kalman and nonlinear systems

 The Kalman filter estimates the internal state of a linear dynamic system. However, it can be extended to a nonlinear state space model using linear or quadratic approximation functions. These filters are known as, you guessed it, **Extended Kalman Filters (EKF)**, the theory of which is beyond the scope of this book.

The following section is dedicated to discrete Kalman filters for linear systems, as applied to financial engineering. A continuous signal can be converted to a time series using the Nyquist frequency.

The state space estimation

The Kalman filter model consists of two core elements of a dynamic system: a process that generates data and a measurement that collects data. These elements are referred to as the state space model. Mathematically speaking, the state space model consists of two equations:

- **The transition equation:** This describes the dynamics of the system, including the unobserved variables
- **The measurement equation:** This describes the relationship between the observed and unobserved variables

The transition equation

Let's consider a system with a linear state x_t of n variables and a control input vector u_t . The prediction of the state at time t is computed by a linear stochastic equation (**M12**):

$$x_t = A_t \cdot x_{t-1} + B_t \cdot u_t + w_t$$

- A_t is the square matrix of dimension n that represents the transition from a state x_{t-1} at $t-1$ to a state x_t at t . The matrix is intrinsic to the dynamic system under consideration.
- B_t is a n by n matrix that describes the control input model (an external action on the system or model). It is applied to the control vector, u_t .
- w_t represents the noise generated by the system, or from a probabilistic point of view, it represents the uncertainty on the model. It is known as the process white noise.

The control input vector represents the external input (or control) to the state of the system. Most systems, including our financial example later in this chapter, have no external input to the state of the model.



A white and Gaussian noise

A white noise is a Gaussian noise, following a normal distribution with zero mean.

The measurement equation

The measurement of m values z_t of the state of the system is defined by the following equation (M13):

$$z_t = H_t \cdot x_t + v_t$$

- H_t is a m by n matrix that models the dependency of the measurement to the state of the system.
- v_t is the white noise introduced by the measuring devices. Similarly to the process noise, v follows a Gaussian distribution with zero mean and a variance R , known as the **measurement noise covariance**.

The time dependency model



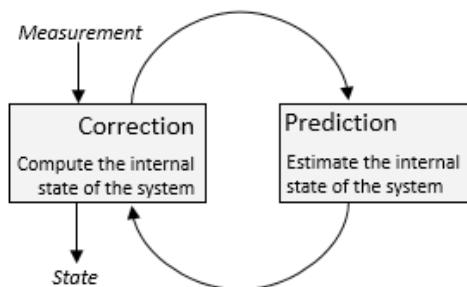
We cannot assume that the parameters of the generalized discrete Kalman filter, such as the state transition A_t , control input B_t and observation matrices (or measurement dependency) H_t are independent of time. However, these parameters are constant in most practical applications.

The recursive algorithm

The set of equations for the discrete Kalman filter is implemented as a recursive computation with two distinct steps:

- The algorithm uses the transition equations to estimate the next observation
- The estimation is created with the actual measurement for this observation

The recursion is visualized in the following diagram:



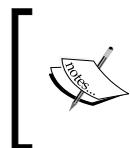
An overview diagram of the recursive Kalman algorithm

Let's illustrate the prediction and correction phases in the context of filtering financial data, in a manner similar to the moving average and Fourier transform. The objective is to extract the trend and the transitory component of the yield of the 10-year Treasury bond. The Kalman filter is particularly suitable for the analysis of interest rates for two reasons:

- Yields are the results of multiple factors, some of which are not directly observable.
- Yields are influenced by the policy of the Federal Reserve that can be easily modeled by the control matrix.

The 10-year Treasury bond has a higher trading volume than bonds with longer maturity, making trends in interest rates a bit more reliable [3:13].

Applying the Kalman filter to clean raw data requires you to define a model that encompasses both observed and non-observed states. In the case of the trend analysis, we can safely create our model with a two-variable state: the current yield x_t and the previous yield x_{t-1} .



The state of dynamic systems

The term "state" refers to the state of the dynamic system under consideration and not the state of the execution of the algorithm.

This implementation of the Kalman filter uses the Apache Commons Math library. Therefore, we need to specify the implicit conversion from our primitives, introduced in the *Primitives and implicits* section in *Chapter 1, Getting Started*, to the `RealMatrix`, `RealVector`, `Array2DRowRealMatrix`, and `ArrayRealVector` Apache Commons Math types:

```
implicit def double2RealMatrix(x: DblMatrix): RealMatrix =
  new Array2DRowRealMatrix(x)
implicit def double2RealRow(x: DblVector): RealMatrix =
  new Array2DRowRealMatrix(x)
implicit def double2RealVector(x: DblVector): RealVector =
  new ArrayRealVector(x)
```

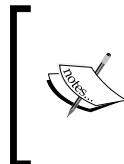
The client code has to import the implicit conversion functions within its scope.

The Kalman model assumes that the process and measurement noise follows a Gaussian distribution, also known as a white noise. For the sake of maintainability, the generation of the white noise is encapsulated in the `QRNoise` class with the following arguments (line 1):

- `qr`: This is the tuple of scale factors for the process noise matrix Q and the measurement noise R
- `profile`: This is the noise profile with the normal distribution as default

The two `noiseQ` and `noiseR` methods generate an array of two independent white noise elements (line 2):

```
val normal = Stats.normal(_)
class QRNoise(qr: DblPair, profile: Double=>Double = normal){ //1
    def q = profile(qr._1)
    def r = profile(qr._2)
    lazy val noiseQ = Array[Double](q, q)    //2
    lazy val noiseR = Array[Double](r, r)
}
```



Experimenting with a noise profile

Although the discrete Kalman filter assumes that the noise profile follows a normal distribution, the `QRNoise` class allows the user to experiment with different noise profiles.

The easiest approach to manage the matrices and vectors used in the recursion is to define them as arguments of a `kalmanConfig` configuration class. The arguments of the configuration follow the naming convention defined in the mathematical formulas: A is the state transition matrix, B is the control matrix, H is the matrix of observations that define the dependencies between the measurement and system state, and P is the covariance error matrix:

```
case class KalmanConfig(A: DblMatrix, B: DblMatrix,
                        H: DblMatrix, P: DblMatrix)
```

Let's implement the Kalman filter as a `DKalman` transformation of the `ETransform` type on a time series with a predefined `KalmanConfig` configuration:

```
class DKalman(config: KalmanConfig)(implicit qrNoise: QRNoise)
  extends ETransform[KalmanConfig](config) {
  type U = Vector[DblPair]    //3
  type V = Vector[DblPair]    //4
  type KRState = (KalmanFilter, RealVector) //5
  override def |> : PartialFunction[U, Try[V]] =
  ...
}
```

As with any explicit data transformation, we need to specify the `U` and `V` types (lines 3 and 4), which are identical. The Kalman filter does not alter the structure of the data, it alters only the values. We define an internal state for the `KRState` Kalman computation by creating a tuple of two `KalmanFilter` and `RealVector` (line 5) Apache Commons Math types.

The key elements of the filter are now in place and it's time to implement the prediction-correction cycle portion of the Kalman algorithm.

Prediction

The prediction phase consists of estimating the x state (yield of the Treasury bond) using the transition equation. We assume that the Federal Reserve has no material effect on the interest rates, making the B control input matrix null. The transition equation can be easily resolved using simple operations on matrices:

$$\begin{bmatrix} \hat{x}_t \\ \hat{x}_{t-1} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} x_t \\ x_{t-1} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} u_t \\ u_{t-1} \end{bmatrix} + \begin{bmatrix} w_t \\ w_{t-1} \end{bmatrix}$$

Visualization of the transition equation of the Kalman filter

The purpose of this exercise is to evaluate the impact of the different parameters of the transition matrix A in terms of smoothing.

The control input matrix B

In this example, the control matrix B is null because there is no known, deterministic external action on the yield of the 10-year Treasury bond. However, the yield can be affected by unknown parameters that we represent as hidden variables. For example, the matrix B can be used to model the decision of the Federal Reserve regarding asset purchases and federal fund rates.

The mathematics behind the Kalman filter presented as a reference to the implementation in Scala, use the same notation for matrices and vectors. It is absolutely not a prerequisite to understand the Kalman filter and its implementation in the next section. If you have a natural inclination toward linear algebra, the following describe the two equations for the prediction step.

The prediction step

M14: The prediction of the state at time t is computed by extrapolating the state estimate:

$$\hat{x}_t' = A_t \cdot \hat{x}_{t-1} + B_t \cdot u_t$$



- A is the square matrix of dimension n that represents the transition from state x at $t-1$ to state x at time t
- \hat{x}'_t is the predicted state of the system based on the current state and the model A
- B is the vector of n dimension that describes the input to the state

M15: The mean square error matrix, P , which is to be minimized, is updated using the following formula:

$$P_t' = A_t \cdot P_{t-1} \cdot A_t^T + Q_t$$

- A^T is the transpose of the state transition matrix
- Q is the process white noise described as a Gaussian distribution with a zero mean and a variance Q , known as the noise covariance

The state transition matrix is implemented using the matrix and vector classes included in the Apache Commons Math library. The types of matrices and vectors are automatically converted into the `RealMatrix` and `RealVector` classes.

The implementation of the equation **M14** is as follows:

```
x = A.operate(x).add(qrNoise.create(0.03, 0.1))
```

The new state is predicted (or estimated), and then used as an input to the correction step.

Correction

The second step of the recursive Kalman algorithm is the correction of the estimated yield of the 10-year Treasury bond with the actual yield. In this example, the white noise of the measurement is negligible. The measurement equation is simple because the state is represented by the current and previous yield and their measurement, z :

$$\begin{bmatrix} z_t \\ z_{t-1} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \cdot \begin{bmatrix} \hat{x}_t \\ \hat{x}_{t-1} \end{bmatrix} + \begin{bmatrix} v_t \\ v_{t-1} \end{bmatrix}$$

Visualization of the measurement equation of the Kalman filter

The sequence of mathematical equations of the correction phase consists of updating the estimation of the state x using the actual values z and computing the Kalman gain, K .

The correction step

M16: The state of the system x is estimated from the actual measurement z using the following formula:

$$\hat{x}_t = \hat{x}_t' + K_t(z_t - H_t \cdot \hat{x}_t')$$

- r_t is the residual between the predicted measurement and the actual measured values
- K_t is the Kalman gain for the correction factor

M17: The Kalman gain is computed as:

$$K_t = P_t' \cdot H_t^T (H_t \cdot P_t' \cdot H_t^T + R_t)^{-1}$$

Here, H^T is the matrix transpose of H and P_t' is the estimate of the error covariance.

Kalman smoothing

It is time to put our knowledge of the transition and measurement equations to test. The Apache Commons Math library defines the two `DefaultProcessModel` and `DefaultMeasurementModel` classes to encapsulate the components of the matrices and vectors. The historical values for the yield of the 10-year Treasury bond are loaded through the `DataSource` method and mapped to the smoothed series that is the output of the filter:

```
override def |> : PartialFunction[U, Try[V]] = {
  case xt: U if( !xt.isEmpty) => Try(
    xt.map { case(current, prev) => {
      val models = initialize(current, prev) //6
      val nState = newState(models) //7
      (nState(0), nState(1)) //8
    }})
  }
}
```

The data transformation for the Kalman filter initializes the process and measurement model for each data point in the private `initialize` (line 6) method, updates the state using the transition and correction equations iteratively in the `newState` method (line 7), and returns the filtered series of pair values (line 8).

Exception handling

The code to catch and process exceptions thrown by the Apache Commons Math library is omitted as the standard practice in the book. As far as the execution of the Kalman filter is concerned, the following exceptions have to be handled:

- `NonSquareMatrixException`
- `DimensionMismatchException`
- `MatrixDimensionMismatchException`

The `initialize` method encapsulates the initialization of the `pModel` process model (line 9) and the `mModel` measurement (observations dependencies) model (line 10), as defined in the Apache Commons Math library:

```
def initialize(current: Double, prev: Double): KRState = {
    val pModel = new DefaultProcessModel(config.A, config.B,
        Q, input, config.P) //9
    val mModel = new DefaultMeasurementModel(config.H, R) //10
    val in = Array[Double](current, prev)
    (new KalmanFilter(pModel, mModel), new ArrayRealVector(in))
}
```

The exceptions thrown by the Apache Commons Math API are caught and processed through the `Try` monad. The iterative prediction and correction of the smoothed yields of 10-year Treasury bond is implemented by the `newState` method. The method iterates through the following steps:

1. Estimate the new values of the state by invoking the Apache Commons Math `KalmanFilter.predict` method that implements the **M14** formula (line 11).
2. Apply the **M12** formula to the new state x at time t (line 12).
3. Compute the measured value z at time t using the **M13** formula (line 13).
4. Invoke the Apache Commons Math `KalmanFilter.correct` method to implement the **M16** formula (line 14).
5. Return the estimated value of the state x by invoking the Apache Commons Math `KalmanFilter.getStateEstimation` method (line 15).

The code will be as follows:

```
def newState(state: KRState): DblArray = {
    state._1.predict //11
    val x = config.A.operate(state._2).add(qrNoise.noisyQ) //12
    val z = config.H.operate(x).add(qrNoise.noisyR) //13
    state._1.correct(z) //14
    state._1.getStateEstimation //15
}
```

The exit condition

In the code snippet for the `newState` method, the iteration for specific data points exits when the maximum number of iterations is reached. A more elaborate implementation consists of either evaluating the matrix P at each iteration or estimation converged within a predefined range.

Fixed lag smoothing

So far, we have studied the Kalman filtering algorithm. We need to adapt it to the smoothing of a time series. The **fixed lag smoothing** technique consists of backward correcting previous data points, taking into account the latest actual value.

A N-lag smoother defines the input as a vector $X = \{x_{t-N-1}, x_{t-N-2}, \dots, x_t\}$ for which the value x_{t-N-j} is corrected taking into account the current value of x_t .

The strategy is quite similar to the hidden Markov model forward and backward passes (refer to the *Evaluation – CF-1* section under *The hidden Markov model in Chapter 7, Sequential Data Models*).

Complex strategies for lag smoothing

There are numerous formulas or methodologies to implement an accurate fixed lag smoothing strategy and correct the predicted observations. Such strategies are beyond the scope of this book.

Experimentation

The objective is to smoothen the yield of the 10-year Treasury bond using a **two-step lag smoothing** algorithm.

The two-step lag smoothing



M18: The two-step lag smoothing algorithm for state S_t using a single smoothing factor α is defined as:

$$S_t = [x_{t+1}, x_t]^T \quad \text{with} \quad \begin{vmatrix} x_{t+1} \\ x_t \end{vmatrix} = \begin{vmatrix} \alpha & 1 - \alpha \\ 1 & 0 \end{vmatrix} \cdot \begin{vmatrix} x_t \\ x_{t-1} \end{vmatrix}$$

The state equation updates the values of the state $[x_t, x_{t-1}]$ using the previous state $[x_{t-1}, x_{t-2}]$, where x_t represents the yield of the 10-year Treasury bond at time t . This is accomplished by shifting the values of the original time series $\{x_0, \dots, x_{n-1}\}$ by 1 using the drop method, $X_1 = \{x_1, \dots, x_{n-1}\}$, creating a copy of the original time series without the last element $X_2 = \{x_0, \dots, x_{n-2}\}$, and zipping X_1 and X_2 . This process is implemented by the `zipWithShift` method, which is introduced in the first section of the chapter.

The resulting sequence of a state vector $S_k = [x_k, x_{k-1}]^T$ is processed by the Kalman algorithm, as shown in the following code:

```
Import YahooFinancials._
val RESOURCE_DIR = "resources/data/chap3/"
implicit val qrNoise = new QRNoise((0.7, 0.3)) //16

val H: DblMatrix = ((0.9, 0.0), (0.0, 0.1)) //17
val P0: DblMatrix = ((0.4, 0.3), (0.5, 0.4)) //18
val ALPHA1 = 0.5; val ALPHA2 = 0.8
val src = DataSource(s"$${RESOURCE_DIR}$$${symbol}.csv", false)

(src.get(adjClose)).map(zt => { //19
    twoStepLagSmoothen(zt, ALPHA1) //20
    twoStepLagSmoothen(zt, ALPHA2)
})
```

An implicit noise instance

The noise for the process and measurement is defined as an implicit argument to the `DKalman` Kalman filter for the following two reasons:



- The profile of the noise is specific to the process or system under evaluation and its measurement; it is independent of the `A`, `B`, and `H` Kalman configuration parameters. Therefore, it cannot be a member of the `KalmanConfig` class.
- The same noise characteristics should be shared with other alternative filtering techniques, if needed.

The white noise for the process and measurement is initialized implicitly with the `qrNoise` value (line 16). The code initializes the matrices `H` of the measurement dependencies on the state (line 17) and `P0` that contains the initial covariance errors (line 18). The input data is extracted from a CSV file that contains the daily Yahoo financial data (line 19). Finally, the method executes the `twoStepLagSmoothen` two-step lag smoothing algorithm with two different `ALPHA1` and `ALPHA2` alpha parameter values (line 20).

Let's take a look at the `twoStepLagSmoothen` method:

```
def twoStepLagSmoothen(zSeries: DblVector, alpha: Double): Int = {
    val A: DblMatrix = ((alpha, 1.0-alpha), (1.0, 0.0)) //21
    val xt = zipWithShift(1) //22
    val pfnKalman = DKalman(A, H, P0) |> //23
    pfnKalman(xt).map(filtered => //24
        display(zSeries, filtered.map(_.-_1), alpha)
    )
}
```

The `twoStepLagSmoothen` method takes two arguments:

- A `zSeries` single variable time series
- A `alpha` state transition parameter

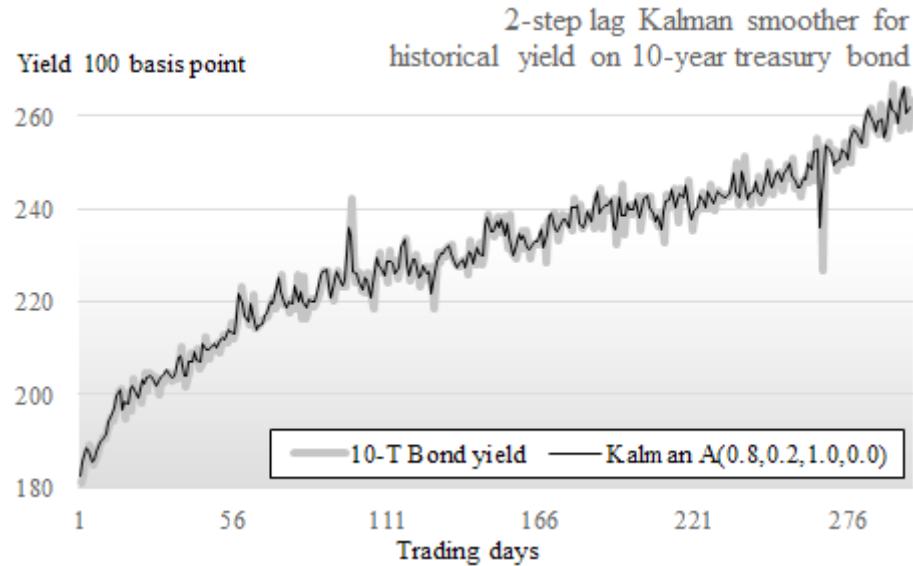
It initializes the state transition matrix `A` using the `alpha` exponential moving average decay parameter (line 21). It creates the two-step lag time series, `xt`, using the `zipWithShift` method (line 22). It extracts the `pfnKalman` partial function (line 23), processes, and finally, displays the two-step lag time series (line 24).

Modeling state transition and noise

The state transition and the noise related to the process have to be selected carefully. The resolution of the state equations relies on the **Cholesky** (QR) decomposition, which requires a nonnegative definite matrix. The implementation in the Apache Commons Math library throws a `NonPositiveDefiniteMatrixException` exception if the principle is violated.

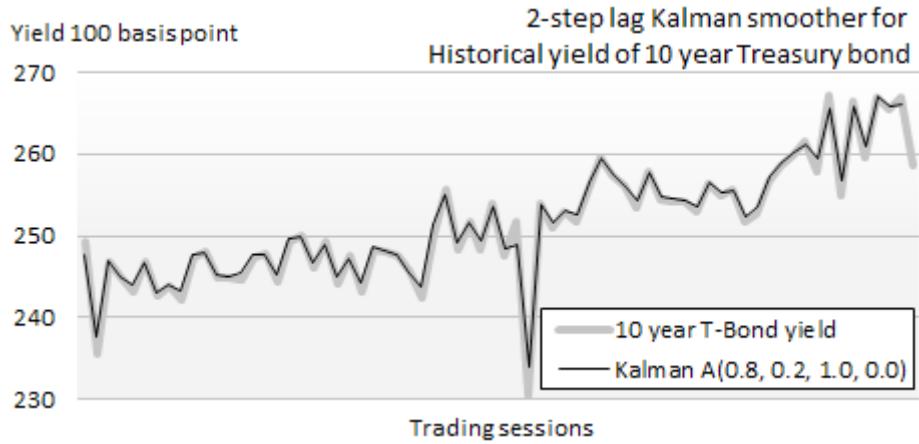
Data Preprocessing

The smoothed yield is plotted along the raw data as follows:



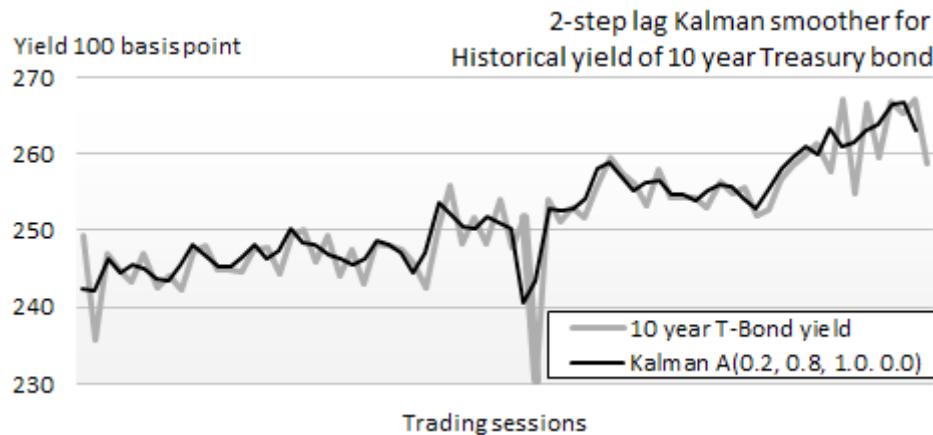
The output of the Kalman filter for the 10-year Treasury-Bond historical prices

The Kalman filter is able to smooth the historical yield of the 10-year Treasury bond while preserving the spikes and lower frequency noise. Let's analyze the data for a shorter period during which the noise is the strongest, between the 190th and the 275th trading days:



The output of the Kalman filter for the 10-year Treasury bond prices 0.8-0.02

The high frequency noise has been significantly reduced without cancelling the actual spikes. The distribution (0.8, 0.2) takes into consideration the previous state and favors the predicted value. Contrarily, a run with a state transition matrix A [0.2, 0.8, 0.0, 1.0] that favors the latest measurement will preserve the noise, as seen in the following graph:



The output of the Kalman filter for the 10-year Treasury bond price 0.2-0.8

Benefits and drawbacks

The Kalman filter is a very useful and powerful tool used to help you understand the distribution of the noise between the process and observation. Contrary to the low or band-pass filters based on the discrete Fourier transform, the Kalman filter does not require the computation of the frequencies spectrum or assume the range of frequencies of the noise.

However, the linear discrete Kalman filter has its limitations, which are as follows:

- The noise generated by both the process and the measurement has to be Gaussian. Processes with non-Gaussian noise can be modeled with techniques such as a Gaussian sum filter or adaptive Gaussian mixture [3:14].
- It requires that the underlying process is linear. However, researchers have been able to formulate extensions to the Kalman filter, known as the **extended Kalman filter (EKF)**, to filter signals from nonlinear dynamic systems, at the cost of significant computational complexity.

The continuous-time Kalman filter



The Kalman filter is not restricted to dynamic systems with discrete states x . The case of continuous state-time is handled by modifying the state transition equation, so the estimated state is computed as the derivative dx/dt .

Alternative preprocessing techniques

For the sake of space and your time, this chapter introduced and applied three filtering and smoothing classes of algorithms. Moving averages, Fourier series, and the Kalman filter are far from being the only techniques used in cleaning raw data. The alternative techniques can be classified into the following categories:

- Autoregressive models that encompass **Autoregressive Moving Average (ARMA)**, **Autoregressive Integrated Moving Average (ARIMA)**, **generalized autoregressive conditional heteroskedasticity (GARCH)**, and Box-Jenkins that relies on some form of autocorrelation function.
- **Curve-fitting** algorithms that include the polynomial and geometric fit with the ordinary least squares method, nonlinear least squares using the **Levenberg-Marquardt** optimizer, and probability distribution fitting.
- Nonlinear dynamic systems with a Gaussian noise such as a **particle filter**.
- Hidden Markov models, as described in the *The hidden Markov model* section in *Chapter 7, Sequential Data Models*.

Summary

This completes the overview of the most commonly used data filtering and smoothing techniques. There are other types of data preprocessing algorithms such as normalization, analysis, and reduction of variance; the identification of missing values is also essential to avoid the **garbage-in garbage-out** conundrum that plagues so many projects that use machine learning for regression or classification.

Scala can be effectively used to make the code understandable and avoid cluttering methods with unnecessary arguments.

The three techniques presented in this chapter, from the simplest moving averages and Fourier transform to the more elaborate Kalman filter, go a long way in setting up data for the concepts introduced in the next chapter: unsupervised learning and more specifically, clustering.

4

Unsupervised Learning

Labeling a set of observations for classification or regression can be a daunting task, especially in the case of a large feature set. In some cases, labeled observations are either unavailable or not possible to create. In an attempt to extract some hidden associations or structures from observations, the data scientist relies on unsupervised learning techniques to detect patterns or similarity in data.

The goal of unsupervised learning is to discover patterns of regularities and irregularities in a set of observations. These techniques are also applied in reducing the solution or feature space.

There are numerous unsupervised algorithms; some are more appropriate to handle dependent features while others generate affinity groups in the case of hidden features [4:1]. In this chapter, you will learn three of the most common unsupervised learning algorithms:

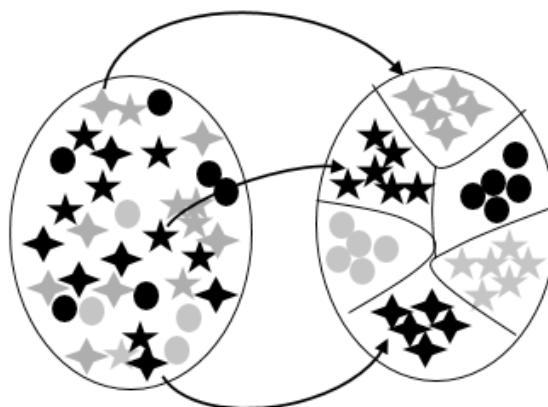
- **K-means:** This used for clustering observed features
- **Expectation-maximization (EM):** This is used for clustering observed and latent features
- **Principal Components Analysis (PCA):** This is used to reduce the dimension of the model

Any of these algorithms can be applied to technical analysis or fundamental analysis. Fundamental analysis of financial ratios and technical analysis of price movements is discussed in the *Technical analysis* section under *Finances 101* in the *Appendix A, Basic Concepts*. The K-means algorithm is fully implemented in Scala while expectation-maximization and Principal Components Analysis leverage the Apache Commons Math library.

The chapter concludes with a brief overview of dimension reduction techniques for non-linear models.

Clustering

Problems involving a large number of features for large datasets become quickly intractable, and it is quite difficult to evaluate the independence between features. Any computation that requires some level of optimization and, at a minimum, the computation of first order derivatives requires a significant amount of computing power to manipulate high-dimension matrices. As with many engineering fields, a divide-and-conquer approach to classifying very large datasets is quite effective. The objective is to reduce very large sets of observations into a small group of observations that share some common attributes.



Visualization of data clustering

This approach is known as vector quantization. Vector quantization is a method that divides a set of observations into groups of similar size. The main benefit of vector quantization is that the analysis using a representative of each group is far simpler than an analysis of the entire dataset [4:2].

Clustering, also known as **cluster analysis**, is a form of vector quantization that relies on a concept of distance or similarity to generate groups known as clusters.

 **Learning vector quantization (LVQ)**
Vector quantization should not be confused with **learning vector quantization**; learning vector quantization is a special case of artificial neural networks that relies on a winner-take-all learning strategy to compress signals, images, or videos.

This chapter introduces two of the most commonly applied clustering algorithms:

- **K-means:** This is used for quantitative types and minimizes the total error (known as the reconstruction error) given the number of clusters and the distance formula.
- **Expectation-maximization (EM):** This is a two-step probabilistic approach that maximizes the likelihood estimates of a set of parameters. EM is particularly suitable for handling missing data.

K-means clustering

K-means is a popular clustering algorithm that can be implemented either iteratively or recursively. The representative of each cluster is computed as the center of the cluster, known as the **centroid**. The similarity between observations within a single cluster relies on the concept of distance (or similarity) between observations.

Measuring similarity

There are many ways to measure the similarity between observations. The most appropriate measure has to be intuitive and avoid computational complexity. This section reviews three similarity measures:

- The Manhattan distance
- The Euclidean distance
- The normalized inner or dot product

The Manhattan distance is defined by the absolute distance between two variables or vectors, $\{x_i\}$ and $\{y_i\}$, of the same size (M1):

$$d(x, y) = \sum |x_i - y_i|$$

The implementation is generic enough to compute the distance between two vectors of elements of different types as long as an implicit conversion between each of these types to the `Double` values is already defined, as follows:

```
def manhattan[T <: AnyVal, U <: AnyVal] (
  x: Array[T],
  y: Array[U])(implicit f: T => Double): Double =
  (x, y).zipped.map{case (u, v) => Math.abs(u-v)}.sum
```

The ubiquitous Euclidean distance between two vectors, $\{x_i\}$ and $\{y_i\}$, of the same size is defined by the following formula (M2):

$$d(x, y) = \sum (x_i - y_i)^2$$

The code will be as follows:

```
def euclidean[T <: AnyVal, U <: AnyVal] (
    x: Array[T],
    y: Array[U])(implicit f: T => Double): Double =
  Math.sqrt((x,y).zipped.map{case (u,v) => u-v}.map(sqr(_)).sum)
```

The normalized inner product or cosine distance between two vectors, $\{x_i\}$ and $\{y_i\}$, is defined by the following formula (M3):

$$d(x, y) = \frac{\sum x_i y_i}{(\sum x_i^2 \sum y_i^2)^{1/2}}$$

In this implementation, the computation of the dot product and the norms for each dataset is done simultaneously using the tuple within the `fold` method:

```
def cosine[T <: AnyVal, U <: AnyVal] (
    x: Array[T],
    y: Array[U])(implicit f: T => Double): Double = {
  val norms = (x,y).zipped
    .map{ case (u,v) => Array[Double](u*v, u*u, v*v) }
    ./:(Array.fill(3)(0.0))((s, t) => s ++ t)
  norms(0)/Math.sqrt(norms(1)*norms(2))
}
```

Performance of zip and zipped

The scalar product of two vectors is one of the most common operations. It is tempting to implement the dot product using the generic `zip` method:



```
def dot(x:Array[Double], y:Array[Double]):  
  Array[Double] =  
    x.zip(y).map{case(x, y) => f(x,y) }
```

A functional alternative is to use the `Tuple2.zipped` method:

```
def dot(x:Array[Double], y:Array[Double]):  
  Array[Double] =  
    (x, y).zipped map (_ * _)
```

If readability is not a primary issue, you can always implement the `dot` method with a `while` loop, which prevents you from using the ubiquitous `while` loop.

Defining the algorithm

The main advantage of the K-means algorithm (and the reason for its popularity) is its simplicity [4:3].

K-means objective



M4: Let's consider K clusters, $\{C_k\}$, with means or centroids, $\{m_k\}$. The K-means algorithm is indeed an optimization problem whose objective is to minimize the reconstruction or total error defined as the total sum of the distance:

$$\min_{c_k} \sum_1^K \sum_{x_i \in C_k} d(x_i, m_k)$$

The four steps of the K-means algorithm are as follows:

1. Cluster configuration (initializing the centroids or means m_k of the K clusters).
2. Cluster assignment (assigning observations to the nearest cluster given the centroids m_k).

3. Error minimization (computing the total reconstruction error):
 1. Compute centroids m_k that minimize the total reconstruction error for the current assignment.
 2. Reassign the observations given the new centroids m_k .
 3. Repeat the computation of the total reconstruction error until no observations are reassigned.
4. Classification of a new observation by assigning the observation to the closest cluster.

We need to define the components of the K-means in Scala before implementing the algorithm.

Step 1 – cluster configuration

Let's create the two main components of the K-means algorithms: clusters of observations and the implementation of the K-means algorithm.

Defining clusters

The first step is to define a cluster. A cluster is defined by the following parameters:

- The centroid (`center`) (line 1)
- The indices of the observations that belong to this cluster (`members`) (line 2)

The code will be as follows:

```
class Cluster[T <: AnyVal] (val center: DblArray)
    (implicit f: T => Double) { //1
  type DistanceFunc[T] = (DblArray, Array[T]) => Double
  val members = new ListBuffer[Int] //2
  def moveCenter(xt: XSeries[T]): Cluster[T]
  ...
}
```

The cluster is responsible for managing its members (data points) at any point of the iterative computation of the K-means algorithm. It is assumed that a cluster will never contain the same data points twice. The two key methods in the `Cluster` class are as follows:

- `moveCenter`: This recomputes the centroid of a cluster
- `stdDev`: This computes the standard deviation of the distance between all the observation members and the centroid

The constructor of the `Cluster` class is implemented by the `apply` method in the companion object. For convenience, refer to the *Class constructor template* section in the *Appendix A, Basic Concepts*:

```
object Cluster {
    def apply[T <: AnyVal](center: DblArray)
        (implicit f: T => Double): Cluster[T] =
        new Cluster[T](center)
}
```

Let's take a look at the `moveCenter` method. It creates a new cluster with the existing members and a new centroid. The computation of the values of the centroid requires the transposition of the matrix of observations by features into a matrix of feature by observations (line 3). The new centroid is computed by normalizing the sum of each feature across all the observations by the number of data points (line 4):

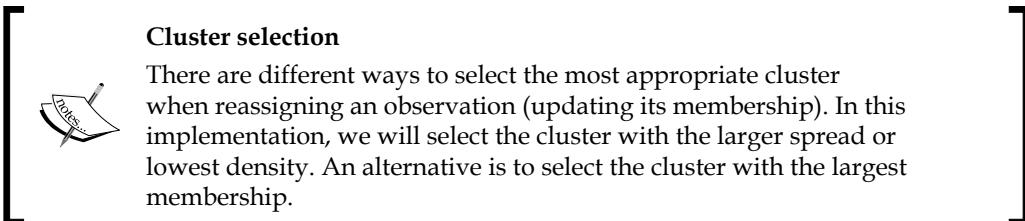
```
def moveCenter(xt: XVSeries[T])
    (implicit m: Manifest[T], num: Numeric[T])
    : Cluster[T] = {
    val sum = transpose(members.map( xt(_)).toList)
        .map(_.sum) //3
    Cluster[T](sum.map( _ / members.size).toArray) //4
}
```

The `stdDev` method computes the standard deviation of all the observations contained in the cluster relative to its center. The distance value between each member and the centroid is extracted through a map invocation (line 5). It is then loaded into a statistics instance to compute the standard deviation (line 6). The function to compute the distance between the center and an observation is an argument of the method. The default distance is euclidean:

```
def stdDev(xt: XVSeries[T], distance: DistanceFunc): Double = {
    val ts = members.map(xt(_)).map(distance(center,_)) //5
    Stats[Double](ts).stdDev //6
}
```

Cluster selection

There are different ways to select the most appropriate cluster when reassigning an observation (updating its membership). In this implementation, we will select the cluster with the larger spread or lowest density. An alternative is to select the cluster with the largest membership.



Initializing clusters

The initialization of the cluster centroids is important to ensure fast convergence of the K-means algorithm. Solutions range from the simple random generation of centroids to the application of genetic algorithms to evaluate the fitness of centroid candidates. We selected an efficient and fast initialization algorithm developed by M. Agha and W. Ashour [4:4].

The steps of the initialization are as follows:

1. Compute the standard deviation of the set of observations.
2. Compute the index of the feature $\{x_{k,0}, x_{k,1} \dots x_{k,n}\}$ with the maximum standard deviation.
3. Rank the observations by their increasing value of standard deviation for the dimension k .
4. Divide the ranked observations set equally into K sets $\{S_m\}$.
5. Find the median value's $size(S_m)/2$.
6. Use the resulting observations as initial centroids.

Let's deconstruct the implementation of the Agha-Ashour algorithm in the `initialize` method:

```
def initialize(xt: U): V = {
    val stats = statistics(xt)    //7
    val maxSDevVar = Range(0, stats.size)  //8
        .maxBy(stats(_).stdDev)

    val rankedObs = xt.zipWithIndex
        .map{case (x, n) => (x(maxSDevVar), n)}
        .sortWith(_._1 < _._1)  //9

    val halfSegSize = ((rankedObs.size>>1)/_config.K)
        .floor.toInt //10
    val centroids = rankedObs
        .filter(isContained(_, halfSegSize, rankedObs.size))
        .map{ case(x, n) => xt(n)} //11
    centroids.aggregate(List[Cluster[T]]())((xs, c) =>
        Cluster[T](c) :: xs, _ ::: _) //12
}
```

The `statistics` method on time series of the `xvseries` type is defined in the *Time series in Scala* section in *Chapter 3, Data Preprocessing* (line 7). The dimension (or feature) with the `maxSDevVar` maximum variance or standard deviation is computed using the `maxBy` method on a `stats` instance (line 8). Then, the observations are ranked by the increasing value of the `rankedObs` standard deviation (line 9).

The ordered sequence of observations is then broken into the `xt.size/_config.K` segments (line 10) and the indices of the centroids are selected as the midpoint (or median) observations of those segments using the `isContained` filtering condition (line 11):

```
def isContained(t: (T, Int), hSz: Int, dim: Int): Boolean =
  (t._2 % hSz == 0) && (t._2 % (hSz << 1) != 0)
```

Finally, the list of clusters is generated using an aggregate call on the set of centroids (line 12).

Step 2 – cluster assignment

The second step in the K-means algorithm is the assignment of the observations to the clusters for which the centroids have been initialized in step 1. This feat is accomplished by the private `assignToClusters` method:

```
def assignToClusters(xt: U, clusters: V,
  members: Array[Int]): Int = {
  xt.zipWithIndex.filter{ case(x, n) => { //13
    val nearestCluster = getNearestCluster(clusters, x) //14
    val reassigned = nearestCluster != members(n)

    clusters(nearestCluster) += n //15
    members(n) = nearestCluster //16
    reassigned
  }}.size
}
```

The core of the assignment of observations to each cluster is the filter on the time series (line 13). The filter computes the index of the closest cluster and checks whether the observation is to be reassigned (line 14). The observation at the `n` index is added to the nearest cluster, `clusters(nearestCluster)` (line 15). The current membership of the observations is then updated (line 16).

The cluster closest to an observation data is computed by the private `getNearestCluster` method as follows:

```
def getNearestCluster(clusters: V, x: Array[T]): Int =  
  clusters.zipWithIndex./:(Double.MaxValue, 0)) {  
    case (p, (c, n) ) => {  
      val measure = distance(c.center, x) //17  
      if( measure < p._1) (measure, n) else p  
    } }._2
```

A fold is used to extract the cluster that is closest to the `x` observation from the list of clusters using the distance metric defined in the K-means constructor (line 17).

As with other data processing units, the extraction of K-means clusters is encapsulated in a data transformation so that clustering can be integrated into a workflow using the composition of mixins described in the *Composing mixins to build a workflow* section in *Chapter 2, Hello World!*

K-means algorithm exit condition

In some rare instances, the algorithm may reassign the same few observations between clusters, preventing its convergence toward a solution in a reasonable time. Therefore, it is recommended that you add a maximum number of iterations as an exit condition. If K-means does not converge with the maximum number of iterations, then the cluster centroids need to be reinitialized and the iterative (or recursive) execution needs to be restarted.



The `| >` transformation requires that the computation of the standard deviation of the distance of the observations related to the centroid, `c`, is computed in the `stdDev` method:

```
type KMeansModel [T] = List[Cluster[T]]  
def stdDev [T] (  
  clusters: KMeansModel[T],  
  xt: XVSeries[T],  
  distance: DistanceFunc[T]): DblVector =  
  clusters.map( _.stdDev(xt, distance)).toVector
```

Centroid versus mean

The terms centroid and mean refer to the same entity: the center of a cluster. This chapter uses these two terms interchangeably.



Step 3 – reconstruction/error minimization

The clusters are initialized with predefined set of observations as their members. The algorithm updates the membership of each cluster by minimizing the total reconstruction error. There are two effective strategies to execute the K-means algorithm:

- Tail recursive execution
- Iterative execution

Creating K-means components

Let's declare the K-means algorithm class, `KMeans`, with its public methods. `KMeans` implements an `ITransform` data transformation using an implicit model extracted from a training set and is described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 18). The configuration of the `KMeansConfig` type consists of the tuple `(K, maxIter)` with `K` being the number of clusters and `maxIter` being the maximum number of iterations allowed for the convergence of the algorithm:

```
case class KMeansConfig(val K: Int, maxIter: Int)
```

The `KMeans` class takes the following three arguments:

- `config`: This is the configuration used for the execution of the algorithm
- `distance`: This is the function used to compute the distance between any observation and a cluster centroid
- `xt`: This is the training set

The implicit conversion of the `T` type to a `Double` is implemented as a view bound. The instantiation of the `KMeans` class initializes a `V` type of output from K-means as `Cluster[T]` (line 20). The `num` instance of the `Numeric` class has to be passed implicitly as a class parameter because it is required by the `sortWith` invocation in `initialize`, the `maxBy` method, and the `Cluster.moveCenter` method (line 19). The `Manifest` is required to preserve the erasure type for `Array[T]` in the JVM:

```
class KMeans[T <: AnyVal](config: KMeansConfig, //18
    distance: DistanceFunc[T],
    xt: XVSeries[T])
    (implicit m: Manifest[T], num: Numeric[T], f: T=>Double) //19
extends ITransform[Array[T]](xt) with Monitor[T] {

    type V = Cluster[T] //20
    val model: Option[KMeansModel[T]] = train
    def train: Option[KMeansModel[T]]
    override def |> : PartialFunction[U, Try[V]]
    ...
}
```

The KMeansModel model is defined as the list of clusters extracted through training.

Tail recursive implementation

The transformation or clustering function is implemented by the train training method that creates a partial function with `xvSeries[T]` as the input and `KMeansModel[T]` as the output:

```
def train: Option[KMeansModel[T]] = Try {
    // STEP 1
    val clusters = initialize(xt) //21
    if( clusters.isEmpty) /* ... */
    else {
        // STEP 2
        val members = Array.fill(xt.size)(0)
        assignToClusters(xt, clusters, members) //22
        var iters = 0

        // Declaration of the tail recursion def update
        if( iters >= _config.maxIters )
            throw new IllegalStateException( /* .. */)
        // STEP 3
        update(clusters, xt, members) //23
    }
} match {
    case Success(clusters) => Some(clusters)
    case Failure(e) => /* ... */
}
```

The K-means training algorithm is implemented through the following three steps:

1. Initialize the cluster's centroid using the `initialize` method (line 21).
2. Assign observations to each cluster using the `assignToClusters` method (line 22).
3. Recompute the total error reconstruction using the `update` recursive method (line 23).

The computation of the total error reconstruction is implemented as a tail recursive method, `update`, as follows:

```
@tailrec
def update(clusters: KMeansModel[T], xt: U,
           members: Array[Int]): KMeansModel[T] = { //24

    val newClusters = clusters.map( c => {
```

```

        if( c.size > 0) c.moveCenter(xt) //25
        else clusters.filter( _.size >0)
            .maxBy(_.stdDev(xt, distance)) //26
    })
iters += 1
if(iters >= config.maxIters ||           //27
    assignToClusters(xt, newClusters, members) ==0)
    newClusters
else
    update(newClusters, xt, membership) //28
}

```

The recursion takes the following three arguments (line 24):

- The current list of `clusters` that is updated during the recursion
- The `xt` input time series
- The indices of membership to the clusters, `members`

A new list of clusters, `newClusters`, is computed by either recalculating each centroid if the cluster is not empty (line 25) or evaluating the standard deviation of the distance of each observation relative to each centroid (line 26). The execution exits when either the maximum number of the `maxIters` recursive calls is reached or when no more observations are reassigned to a different cluster (line 27). Otherwise, the method invokes itself with an updated list of clusters (line 28).

Iterative implementation

The implementation of an iterative execution is presented for an informational purpose. It follows the same sequence of calls as with the recursive implementation. The new clusters are computed (line 29) and the execution exits when either the maximum number of allowed iterations is reached (line 30) or when no more observations are reassigned to a different cluster (line 31):

```

val members = Array.fill(xt.size)(0)
assignToClusters(xt, clusters, members)
var newClusters: KMeansModel[T] = List.empty[Cluster[T]]
Range(0, maxIters).find( _ => { //29
    newClusters = clusters.map( c => { //30
        if( c.size > 0) c.moveCenter(xt)
        else clusters.filter( _.size > 0)
            .maxBy(_.stdDev(xt, distance))
    })
    assignToClusters(xt, newClusters, members) > 0 //31
}).map(_ => newClusters)

```

The density of the clusters is computed in the `KMeans` class as follows:

```
def density: Option[DblVector] =
  model.map( _.map( c =>
    c.getMembers.map(xt(_)).map( distance(c.center, _) ).sum)
```

Step 4 – classification

The objective of the classification is to assign an observation to a cluster with the closest centroid:

```
override def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if( x.length == dimension(xt)
    && model != None) =>
    Try (model.map( _.minBy(c => distance(c.center,x))).get )
}
```

The most appropriate cluster is computed by selecting the `c` cluster whose center is the closest to the `x` observation using the `minBy` higher order method.

The curse of dimensionality

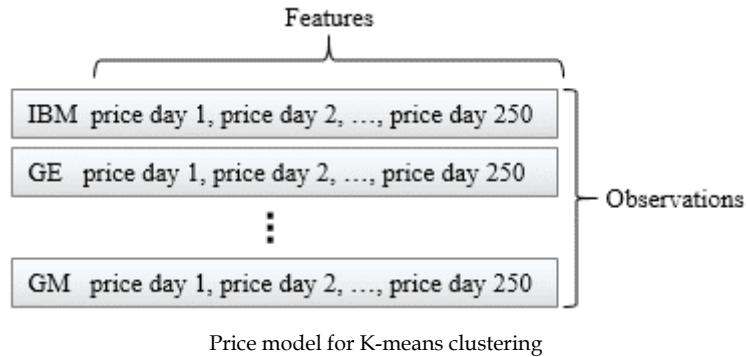
A model with a significant number of features (high dimensions) requires a larger number of observations in order to extract relevant and reliable clusters. K-means clustering with very small datasets (< 50) produces models with high bias and a limited number of clusters, which are affected by the order of observations [4:5]. I have been using the following simple empirical rule of thumb for a training set of size n , expected K clusters, and N features: $n < K.N$.



Dimensionality and the size of the training set

The issue of sizing the training set given the dimensionality of a model is not specific to unsupervised learning algorithms. All supervised learning techniques face the same challenge to set up a viable training plan.

Whichever empirical rule you follow, such a restriction is particularly an issue for analyzing stocks using historical quotes. Let's consider our examples of using technical analysis to categorize stocks according to their price behavior over a period of 1 year (or approximately 250 trading days). The dimension of the problem is 250 (250 daily closing prices). The number of stocks (observations) would have exceeded several hundred!



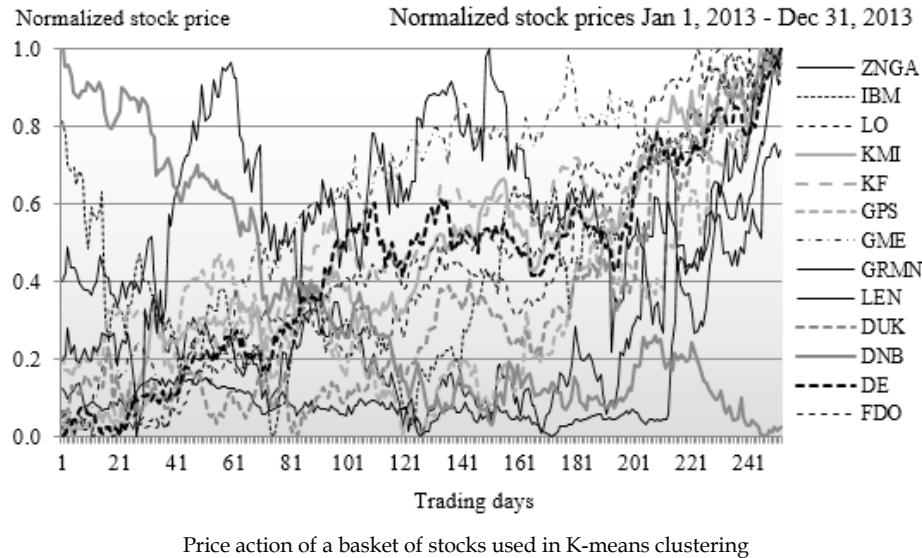
There are options to get around this limitation and shrink the numbers of observations, which are as follows:

- Sampling the trading data without losing a significant amount of information from the raw data, assuming that the distribution of observations follows a known probability density function.
- Smoothing the data to remove the noise as seen in *Chapter 3, Data Preprocessing*, assuming that the noise is Gaussian. In our test, a smoothing technique will remove the price outliers for each stock and therefore reduce the number of features (trading session). This approach differs from the first (sampling) technique because it does not require an assumption that the dataset follows a known density function. On the other hand, the reduction of features will be less significant.

These approaches are workaround solutions at best, used for the sake of this tutorial. You need to consider the quality of your data before applying these techniques to the actual commercial applications. The principal component analysis introduced in the last paragraph of this chapter is one of the most reliable dimension reduction techniques.

Setting up the evaluation

The objective is to extract clusters from a set of stock price actions during a period of time between January 1 and Dec 31, 2013 as features. For this test, 127 stocks are randomly selected from the S&P 500 list. The following chart visualizes the behavior of the normalized price of a subset of these 127 stocks:



The key is to select the appropriate features prior to clustering and the time window to operate on. It would make sense to consider the entire historical price over the 252 trading days as a feature. However, the number of observations (stocks) is too limited to use the entire price range. The observations are the stock closing prices for each trading session between the 80th and 130th trading days. The adjusted daily closing prices are normalized using their respective minimum and maximum values.

First, let's create a simple method to compute the density of the clusters:

```
val MAX_ITERS = 150
def density(K: Int, obs: XSeries[Double]): DblVector =
  KMeans[Double](KMeansConfig(K, MAX_ITERS)).density.get //32
```

The density method invokes `KMeans.density` described in step 3. Let's load the data from CSV files using the `DataSource` class, as described in the *Data extraction* section in the *Appendix A, Basic Concepts*:

```
import YahooFinancials._
val START_INDEX = 80; val NUM_SAMPLES = 50 //33
val PATH = "resources/data/chap4/"
```

```

type INPUT = Array[String] => Double
val extractor = adjClose :: List[INPUT]() //34
val symbolFiles = DataSource.listSymbolFiles(PATH) //35

for {
    prices <- getPrices //36
    values <- Try(getPricesRange(prices)) //37
    stdDev <- Try(ks.map( density(_, values.toVector))) //38
    pfnKmeans <- Try {
        KMeans[Double](KMeansConfig(5,MAX_ITERS),values.toVector) |>
    } //39
    predict <- pfnKmeans(values.head) //40
} yield {
    val results = s"""Daily prices ${prices.size} stocks"""
    | \nClusters density ${stdDev.mkString(", ")}"""
    .stripMargin
    show(results)
}

```

As mentioned earlier, the cluster analysis applies to the closing price in the range between the 80th and 130th trading days (line 33). The extractor function retrieves the adjusted closing price for a stock from YahooFinancials (line 34). The list of stock tickers (or symbols) are extracted as a list of CSV filenames located in path (line 35). For instance, the ticker symbol for General Electric Corp. is GE and the trading data is located in GE.csv.

The execution extracts 50 daily prices using DataSource and then filters out the incorrectly formatted data using filter (line 36):

```

type XVSeriesSet = Array[XVSeries[Double]]
def getPrices: Try[XVSeriesSet] = Try {
    symbolFiles.map( DataSource(_, path) |> extractor )
    .filter( _.isSuccess ).map( _.get )
}

```

The historical stock prices for the trading session between the 80th and 130th days are generated by the getPricesRange closure (line 37):

```

def getPricesRange(prices: XVSeriesSet) =
    prices.view.map(_.head.toArray)
    .map( _.drop(START_INDEX).take(NUM_SAMPLES) )

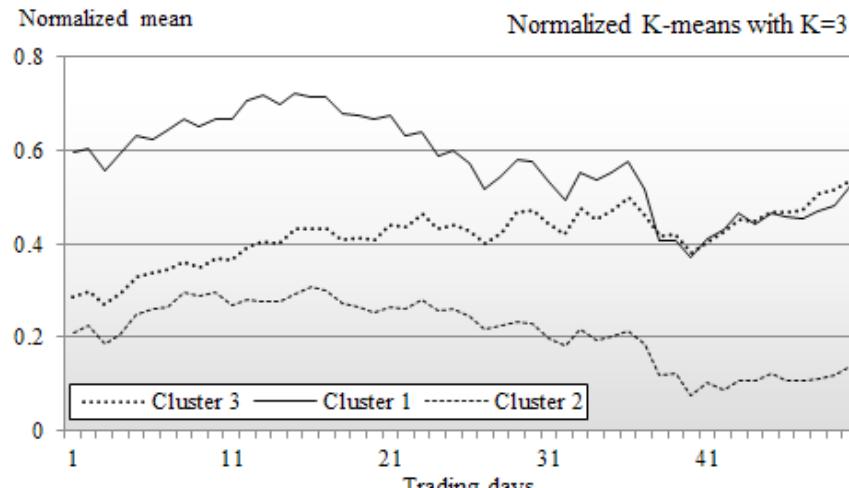
```

It computes the density of the clusters by invoking the density method for each ks value of the number of clusters (line 38).

The `pfnKmeans` partial classification function is created for a 5-cluster, `KMeans` (line 39), and then used to classify one of the observations (line 40).

Evaluating the results

The first test run is executed with $K=3$ clusters. The mean (or centroid) vector for each cluster is plotted as follows:



A chart of means of clusters using K-means K=3

The means vectors of the three clusters are quite distinctive. The top and bottom means **1** and **2** in the chart have the respective standard deviation of 0.34 and 0.27 and share a very similar pattern. The difference between the elements of the **1** and **2** cluster mean vectors is almost constant: 0.37. The cluster with a mean vector **3** represents the group of stocks that behave like the stocks in cluster **2** at the beginning of the time period and behave like the stocks in cluster **1** toward the end of the time period.

This behavior can be easily explained by the fact that the time window or trading period, the 80th to 130th trading day, correspond to the shift in the monetary policy of the federal reserve in regard to the quantitative easing program. Here is the partial list of stocks for each of the clusters whose centroid values are displayed on the chart:

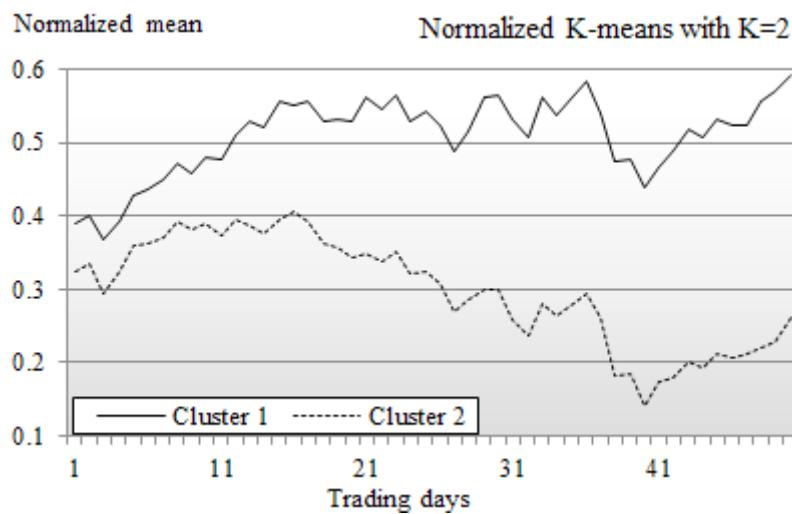
Cluster 1	AET, AHS, BBBY, BRCM, C, CB, CL, CLX, COH, CVX, CYH, DE, ...
Cluster 2	AA, AAPL, ADBE, ADSK, AFAM, AMZN, AU, BHI, BTU, CAT, CCL, ...
Cluster 3	ADM, ADP, AXP, BA, BBT, BEN, BK, BSX, CA, CBS, CCE, CELG, CHK, ...

Let's evaluate the impact of the number of clusters K on the characteristics of each cluster.

Tuning the number of clusters

We repeat the previous test on the 127 stocks and the same time window with the number of clusters varying from 2 to 15.

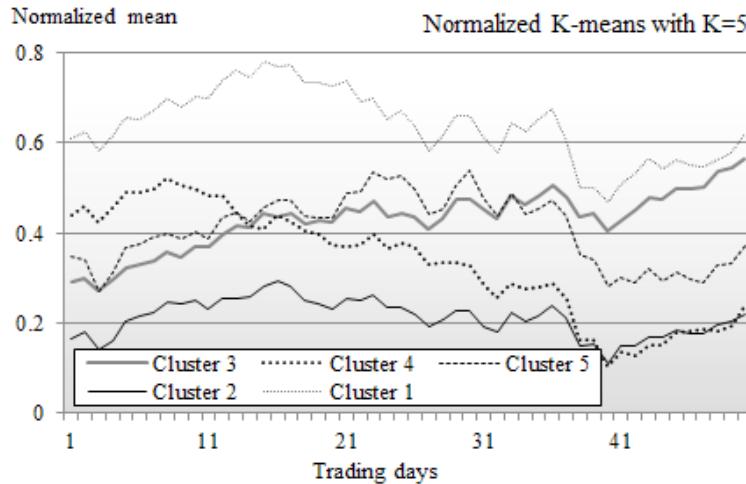
The mean (or centroid) vector for each cluster for $K = 2$ is plotted as follows:



A chart of means of clusters using K-means $K=2$

The chart of the results of the K-means algorithms with 2 clusters shows that the mean vector for the cluster labeled **2** is similar to the mean vector labeled **3** on the chart with $K = 5$ clusters. However, the cluster with the mean vector **1** reflects somewhat the aggregation or summation of the mean vectors for the clusters **1** and **3** in the chart $K = 5$. The aggregation effect explains why the standard deviation for the cluster **1** (0.55) is twice as much as the standard deviation for the cluster **2** (0.28).

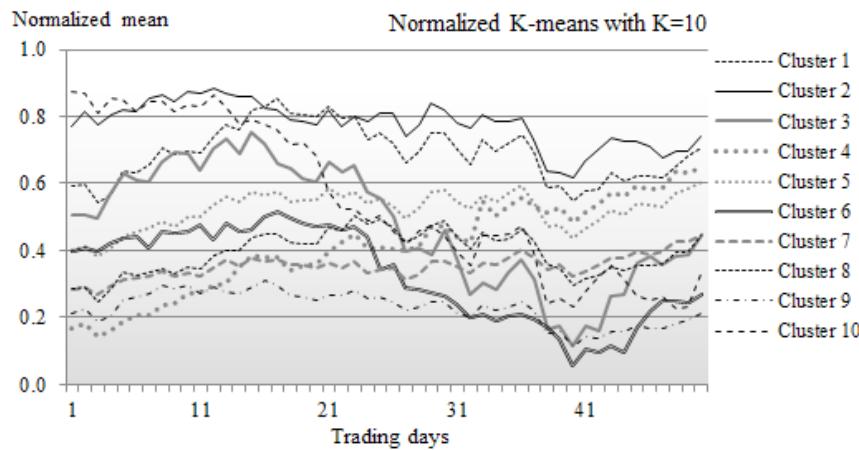
The mean (or centroid) vector for each cluster for $K = 5$ is plotted as follows:



A chart of means of clusters using K-means K=5

In this chart, we can assess that the clusters **1** (with the highest mean), **2** (with the lowest mean), and **3** are very similar to the clusters with the same labels in the chart for $K = 3$. The cluster with the mean vector **4** contains stocks whose behaviors are quite similar to those in cluster **3**, but in the opposite direction. In other words, the stocks in cluster **3** and **4** reacted in opposite ways following the announcement of the change in the monetary policy.

In the tests with high values of K , the distinction between the different clusters becomes murky, as shown in the following chart for $K = 10$:



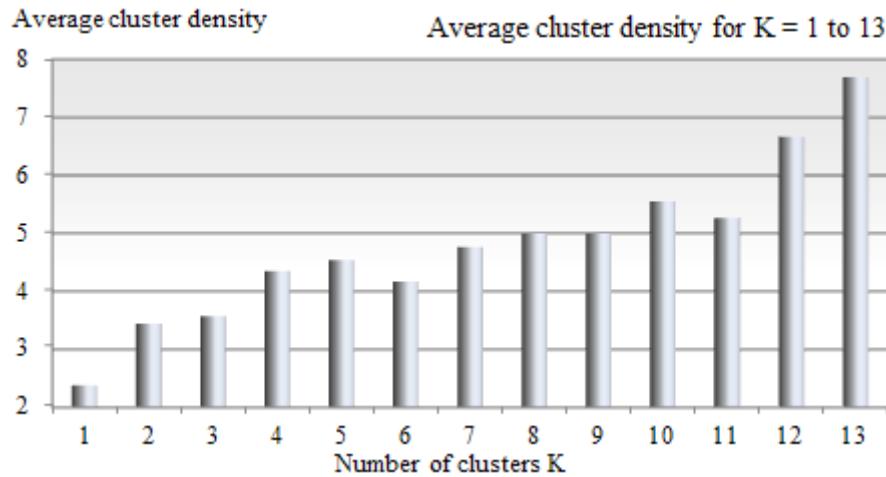
A chart of means of clusters using K-means K=10

The means for clusters **1**, **2**, and **3** seen in the first chart for the case $K = 2$ are still visible. It is fair to assume that these are very likely the most reliable clusters. These clusters happened to have a low standard deviation or high density.

Let's define the density of a cluster, C_j , with a centroid, c_j , as the inverse of the Euclidean distance between all the members of each cluster and its mean (or centroid) (M6):

$$d(C_j) = 1 / \sum_{x \in C_j} (x - c_j)^2$$

The density of the cluster is plotted against the number of clusters with $K = 1$ to $K = 13$:



A bar chart of the average cluster density for $K = 1$ to 13

As expected, the average density of each cluster increases as K increases. From this experiment, we can draw the simple conclusion that the density of each cluster does not significantly increase in the test runs for $K = 5$ and beyond. You may observe that the density does not always increase as the number of clusters increases ($K = 6$ and $K = 11$). The anomaly can be explained by the following three factors:

- The original data is noisy
- The model is somewhat dependent on the initialization of the centroids
- The exit condition is too loose

Validation

There are several methodologies to validate the output of a K-means algorithm from purity to mutual information [4:6]. One effective way to validate the output of a clustering algorithm is to label each cluster and run those clusters through a new batch of labeled observations. For example, if during one of these tests, you find that one of the clusters, CC , contains most of the commodity-related stocks, then you can select another commodity-related stock, SC , which is not part of the first batch, and run the entire clustering algorithm again. If SC is a subset of CC , then K-means has performed as expected. If this is the case, you should run a new set of stocks, some of which are commodity-related, and measure the number of true positives, true negatives, false positives, and false negatives. The values for the precision, recall, and F_1 score introduced in the *Assessing a model* section in *Chapter 2, Hello World!*, confirms whether the tuning parameters and labels you selected for your cluster are indeed correct.

F1 validation for K-means



The quality of the clusters, as measured by the F_1 score, depends on the rule, policy, or formula used to label observations (that is, label a cluster with the industry with the highest relative percentage of stocks in the cluster). This process is quite subjective. The only sure way to validate a methodology is to evaluate several labeling schemes and select the one that generate the highest F_1 score.

An alternative to measure the homogeneity of the distribution of observations across the clusters is to compute the statistical entropy. A low entropy value indicates that the clusters have a low level of impurity. Entropy can be used to find the optimal number of clusters K .

We reviewed some of the tuning parameters that affect the quality of the results of the K-means clustering, which are as follows:

- Initial selection of a centroid
- Number of K clusters

In some cases, the similarity criterion (that is, the Euclidean distance or cosine distance) can have an impact on the *cleanness* or density of the clusters.

The final and important consideration is the computational complexity of the K-means algorithm. The previous sections of the chapter described some of the performance issues with K-means and possible remedies.

Despite its many benefits, the K-means algorithm does not handle missing data or unobserved features very well. Features that depend on each other indirectly may in fact depend on a common hidden (also known as latent) feature. The expectation-maximization algorithm described in the next section addresses some of these limitations.

The expectation-maximization algorithm

The expectation-maximization algorithm was originally introduced to estimate the maximum likelihood in the case of incomplete data [4:7]. It is an iterative method to compute the model features that maximize the likely estimate for observed values, taking into account unobserved values.

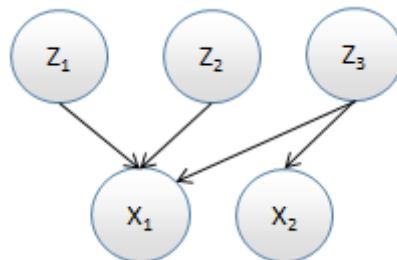
The iterative algorithm consists of computing the following:

- The expectation, E , of the maximum likelihood for the observed data by inferring the latent values (E-step)
- The model features that maximize the expectation E (M-step)

The expectation-maximization algorithm is applied to solve clustering problems by assuming that each latent variable follows a normal or Gaussian distribution. This is similar to the K-means algorithm for which the distance of each data point to the center of each cluster follows a Gaussian distribution [4:8]. Therefore, a set of latent variables is a mixture of Gaussian distributions.

Gaussian mixture models

Latent variables, Z_i , can be visualized as the behavior (or symptoms) of a model (observed) X for which Z are the root causes of the behavior:



Visualization of observed and latent features

The latent values, Z , follow a Gaussian distribution. For the statisticians among us, the mathematics of a mixture model is described here:

Maximization of the log likelihood

M7: If $x = \{x_i\}$ is a set of observed features associated with latent features $z = \{z_i\}$, the probability for the feature x_i of the observation x , given a model parameter θ , is defined as:



$$p(x_i|\theta) = \sum_z p(x_i, z|\theta)$$

M8: The objective is to maximize the likelihood, $L(\theta)$, as shown here:

$$\mathcal{L}(\theta) = \sum_{i=0}^{N-1} \log \left\{ \sum_z p(x_i, z|\theta) \right\} \quad \tilde{\theta} = \text{argmax } \mathcal{L}(\theta)$$

Overview of EM

As far as the implementation is concerned, the expectation-maximization algorithm can be broken down into three stages:

1. The computation of the log likelihood for the model features given some latent variables (initial step).
2. The computation of the expectation of the log likelihood at iteration t (E step).
3. The maximization of the expectation at iteration t (M step).

The E step

M9: The expectation, Q , of the complete data log likelihood for the model parameters, θ_n , at iteration, n , is computed using the posterior distribution of latent variable, z , $p(z|x, \theta)$, and the joint probability of the observation and the latent variable:



$$Q(\theta, \theta^n) = \sum_z p(z|x_i, \theta^n) \cdot \log p(x_i, z|\theta)$$

The M-step

M10: The expectation function Q is maximized for the model features θ to compute the model parameters θ_{n+1} for the next iteration:

$$\theta^{n+1} = \arg \max_{\theta} Q(\theta, \theta^n) \quad |\theta^{n+1} - \theta^n| < \varepsilon$$

A formal, detailed, but short mathematical formulation of the EM algorithm can be found in S. Borman's tutorial [4:9].

Implementation

Let's implement the three steps (initial step, E step, and M step) in Scala. The internal calculations of the EM algorithm are a bit complex and overwhelming. You may not benefit much from the details of a specific implementation such as computation of the eigenvalues of the covariance matrix of the expectation of the log likelihood. This implementation hides some complexities using the Apache Commons Math library package [4:10].



The inner workings of EM

You may want to download the source code for the implementation of the EM algorithm in the Apache Commons Math library, if you need to understand the condition for which an exception is thrown.

The expectation-maximization algorithm of the `MultivariateEM` type is implemented as a data transformation of an `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*. The two arguments of the constructors are the number of `K` clusters (or gauss distribution) and the `xt` training set (line 1). The constructor initializes the `V` type of the output as `EMCluster` (line 2):

```
class MultivariateEM[T <: AnyVal] (K: Int,
  xt: XVSeries[T]) (implicit f: T => Double)
  extends ITransform[Array[T]] (xt) with Monitor[T] { //1
  type V = EMCluster //2
  val model: Option[EMModel] = train //3
  override def |> : PartialFunction[U, Try[V]]
}
```

The multivariate expectation-maximization class has a model that consists of a list of EM clusters of the `EMCluster` type. The `Monitor` trait is used to collect the profiling information during training (refer to the *Monitor* section under *Utility classes* in the *Appendix A, Basic Concepts*).

The information about an EM cluster, `EMCluster`, is defined by `key`, the centroid or means value, and density of the cluster that is the standard deviation of the distance of all the data points to the mean (line 4):

```
case class EMCluster(key: Double, val means: DblArray,
  val density: DblArray) //4
type EMModel = List[EMCluster]
```

The implementation of the EM algorithm in the `train` method uses the Apache Commons Math `MultivariateNormalMixture` for the Gaussian mixture model and `MultivariateNormalMixtureExpectationMaximization` for the EM algorithm:

```
def train: Option[EMModel] = Try {
  val data: DblMatrix = xt //5
  val multivariateEM = new EM(data)
  multivariateEM.fit(estimate(data, K)) //6

  val newMixture = multivariateEM.getFittedModel //7
  val components = newMixture.getComponents.toList //8
  components.map(p => EMCluster(p.getKey, p.getValue.getMeans,
    p.getValue.getStandardDeviations)) //9
} match {/* ... */}
```

Let's take a look at the main `train` method of the `MultivariateEM` wrapper class. The first step is to convert the time series into a primitive matrix of `Double` with observations/historical quotes as rows and the stock symbols as columns.

The `xt` time series of the `xvSeries[T]` type is converted to a `DblMatrix` through an induced implicit conversion (line 5).

The initial mixture of Gaussian distributions can be provided by the user or can be extracted from the `estimate` datasets (line 6). The `getFittedModel` triggers the M-step (line 7).

Conversion from Java and Scala collections



Java primitives need to be converted to Scala types using the `import scala.collection.JavaConversions` package. For example, `java.util.List` is converted to `scala.collection.immutable.List` by invoking the `asScalaIterator` method of the `WrapAsScala` class, one of the base traits of `JavaConversions`.

The Apache Commons Math `getComponents` method returns a `java.util.List` that is converted to `scala.collection.immutable.List` by invoking the `toList` method (line 8). Finally, the `data transform` returns a list of cluster information of the `EMCluster` type (line 9).

Third-party library exceptions

Scala does not enforce the declaration of exceptions as part of the signature of a method. Therefore, there is no guarantee that all types of exceptions will be caught locally. This problem occurs when exceptions are thrown from a third-party library in two scenarios:

- The documentation of the API does not list all the types of exceptions
- The library is updated and a new type of exception is added to a method

One easy workaround is to leverage the Scala exception-handling mechanism:

```
Try {
  ...
} match {
  case Success(results) => ...
  case Failure(exception) => ...
}
```

Classification

The classification of a new observation or data points is implemented by the `| >` method:

```
override def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T]
    if(isModel && x.length == dimension(xt)) =>
      Try(model.map(_.minBy(c => euclidean(c.means, x))).get)
}
```

The `| >` method is similar to the `KMeans.| >` classifier.

Testing

Let's apply the `MultivariateEM` class to the clustering of the same 127 stocks used in evaluating the K-means algorithm.

As discussed in the *The curse of dimensionality* section, the number of stocks (127) to analyze restricts the number of observations to be used by the EM algorithm. A simple option is to filter out some of the noise of the stock's prices and apply a simple sampling method. The maximum sampling rate is restricted by the frequencies in the spectrum of noises of different types in the historical price of every stock.

Filtering and sampling

The preprocessing of the data using a combination of a simple moving average and fixed interval sampling prior to clustering is very rudimentary in this example. For instance, we cannot assume that the historical price of all the stocks share the same noise characteristics. The noise pattern in the quotation of momentum and heavily traded stocks is certainly different from blue-chip securities with a strong ownership, and these stocks are held by large mutual funds.

The sampling rate should take into account the spectrum of frequency of the noise. It should be set as at least twice the frequency of the noise with the lowest frequency.

The object of the test is to evaluate the impact of the sampling rate, `samplingRate`, and the number of `K` clusters used in the EM algorithm:

```
val K = 4; val period = 8
val smAve = SimpleMovingAverage[Double](period) //10
val pfnSmAve = smAve |> //11
```

```

val obs = symbolFiles.map(sym => (
  for {
    xs <- DataSource(sym, path, true, 1) |> extractor //12
    values <- pfnSmAve(xs.head) //13
    y <- Try {
      values.view.zipWithIndex.drop(period+1).toVector
        .filter(_._2 % samplingRate == 0)
        .map(_._1).toArray //14
    }
  } yield y).get)

em(K, obs) //15

```

The first step is to create a simple moving average with a predefined period (line 10), as described in the *The simple moving average* section in *Chapter 3, Data Preprocessing*. The test code instantiates the `pfnSmAve` partial function that implements the moving average computation (line 11). The symbols of the stocks under consideration are extracted from the name of the files in the path directory. The historical data is contained in the CSV file whose name is `path/STOCK_NAME.csv` (line 12).

The execution of the moving average (line 13) generates a set of smoothed values that is sampled given a sampling rate, `samplingRate` (line 14). Finally, the expectation-maximization algorithm is instantiated to cluster the sampled data in the `em` method (line 15):

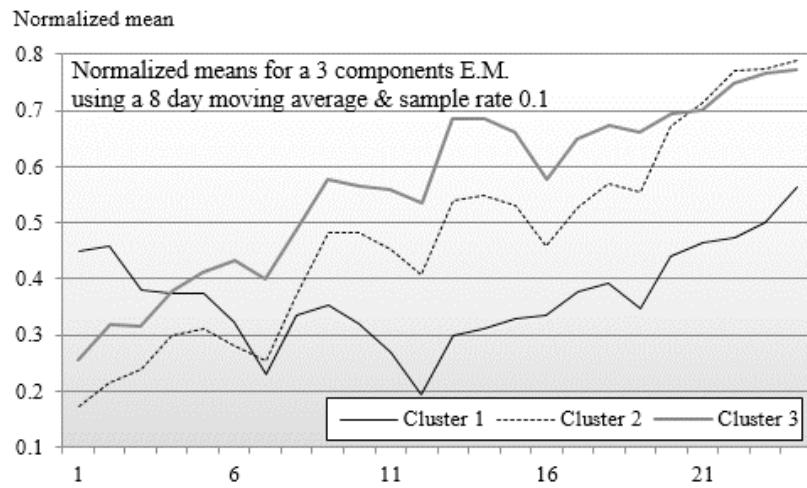
```

def em(K: Int, obs: DblMatrix): Int = {
  val em = MultivariateEM[Double](K, obs.toVector) //16
  show(s"${em.toString}") //17
}

```

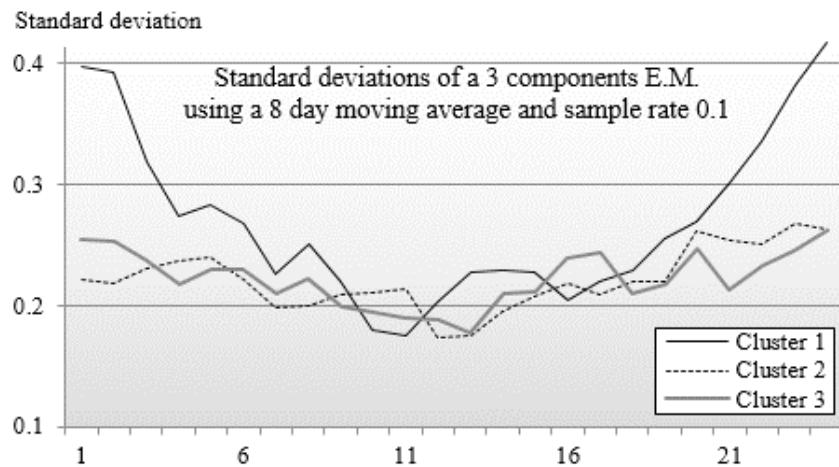
The `em` method instantiates the EM algorithm for a specific number `K` of clusters (line 16). The content of the model is displayed by invoking `MultivariateEM.toString`. The results are aggregated, then displayed in a textual format on the standard output (line 17).

The first test is to execute the EM algorithm with $K = 3$ clusters and a sampling period of 10 on data smoothed by a simple moving average with period of 8. The sampling of historical prices of the 127 stocks between January 1, 2013 and December 31, 2013 with a frequency of 0.1 hertz produces 24 data points. The following chart displays the mean of each of the three clusters:



A chart of the normalized means per cluster using EM K=3

The mean vectors of clusters 2 and 3 have similar patterns, which may suggest that a set of three clusters is accurate enough to provide a first insight into the similarity within groups of stocks. The following is a chart of the normalized standard deviation per cluster using EM with $K = 3$:



A chart of the normalized standard deviation per cluster using EM K=3

The distribution of the standard deviation along with the mean vector of each cluster can be explained by the fact that the price of stocks from a couple of industries went down in synergy, while others went up as a semi-homogenous group following the announcement from the Federal Reserve that the monthly quantity of bonds purchased as part of the quantitative easing program would be reduced in the near future.

Relation to K-means



You may wonder what is the relation between EM and K-means, as both the techniques address the same problem. The K-means algorithm assigns each observation uniquely to one and only one cluster. The EM algorithm assigns an observation based on posterior probability. K-means is a special case of the EM for Gaussian mixtures [4:11].

The online EM algorithm

Online learning is a powerful strategy for training a clustering model when dealing with very large datasets. This strategy has regained interest from scientists lately. The description of the online EM algorithm is beyond the scope of this tutorial. However, you may need to know that there are several algorithms for online EM available if you ever have to deal with large datasets, such as batch EM, stepwise EM, incremental EM, and Monte Carlo EM [4:12].

Dimension reduction

Without prior knowledge of the problem domain, data scientists include all possible features in their first attempt to create a classification, prediction, or regression model. After all, making assumptions is a poor and dangerous approach to reduce the search space. It is not uncommon for a model to use hundreds of features, adding complexity and significant computation costs to build and validate the model.

Noise filtering techniques reduce the sensitivity of the model to features that are associated with sporadic behavior. However, these noise-related features are not known prior to the training phase, and therefore, cannot be completely discarded. As a consequence, training of the model becomes a very cumbersome and time-consuming task.

Overfitting is another hurdle that can arise from a large feature set. A training set of limited size does not allow you to create an accurate model with a large number of features.

Dimension reduction techniques alleviate these problems by detecting features that have little influence on the overall model behavior.

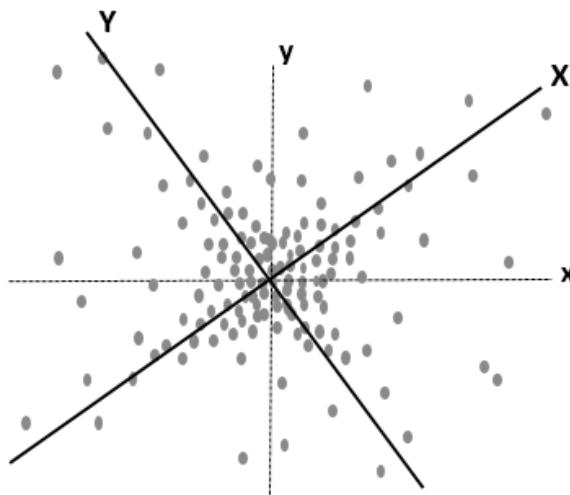
There are three approaches to reduce the number of features in a model:

- Statistical analysis solutions such as ANOVA for smaller feature sets
- Regularization and shrinking techniques, which are introduced in the *Regularization* section in *Chapter 6, Regression and Regularization*
- Algorithms that maximize the variance of the dataset by transforming the covariance matrix

The next section introduces one of the most commonly used algorithms of the third category: principal component analysis.

Principal components analysis

The purpose of principal components analysis is to transform the original set of features into a new set of ordered features by decreasing the order of variance. The original observations are transformed into a set of variables with a lower degree of correlation. Let's consider a model with two features, $\{x, y\}$, and a set of observations, $\{x_i, y_i\}$, plotted on the following chart:



Visualization of principal components analysis for a two-dimensional model

The x and y features are converted into two X and Y variables (that is rotation) to appropriately match the distribution of observations. The variable with the highest variance is known as the first principal component. The variable with the n^{th} highest variance is known as the n^{th} principal component.

Algorithm

I highly recommend that you read the tutorial from Lindsay Smith [4:13] that describes the PCA algorithm in a very concrete and simple way using a two-dimensional model.

PCA and covariance matrix

M11: The covariance of two X and Y features with the observations set $\{x_i, y_i\}$ and their respective mean values is defined as:

$$\text{cov}(X, Y) = \frac{1}{n-1} \sum (x_i - \bar{x})(y_i - \bar{y})$$

Here, \bar{x} and \bar{y} are the respective mean values for the observations, x and y .

M12: The covariance is computed from the Z-score of each observation:

$$x_i = (x_i - \bar{x})/\sigma$$

M13: For a model with n features, x_i , the covariance matrix is defined as:



$$\Sigma = \begin{bmatrix} \text{cov}(x_0, x_0) & \dots & \text{cov}(x_0, x_{n-1}) \\ \vdots & \text{cov}(x_i, x_j) & \vdots \\ \text{cov}(x_{n-1}, x_0) & \dots & \text{cov}(x_{n-1}, x_{n-1}) \end{bmatrix}$$

M14: The transformation of x to X consists of computing the eigenvalues of the covariance matrix:

$$\Sigma' = W^T \Sigma W = \|\text{cov}(X_i, X_j)\| \text{ and } X = W^T x$$

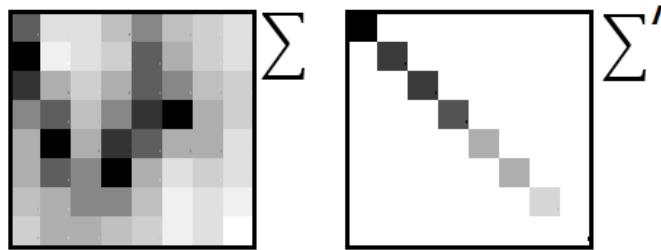
M15: The eigenvalues are ranked by their decreasing order of variance. Finally, the m top eigenvalues for which the cumulative of variance exceeds a predefined threshold (percentage of the trace of the matrix) are the principal components:

$$Z = \{X_i \mid 1:m \mid \sum_{k=1}^m \text{cov}(x_k, x_k) > \epsilon \cdot \text{Tr}(\text{cov})\}$$

The algorithm is implemented in five steps:

1. Compute the Z-score for the observations by standardizing the mean and standard deviation.
2. Compute the covariance matrix Σ for the original set of observations.
3. Compute the new covariance matrix Σ' for the observations with the transformed features by extracting the eigenvalues and eigenvectors.
4. Convert the matrix to rank eigenvalues by decreasing the order of variance. The ordered eigenvalues are the principal components.
5. Select the principal components for which the total sum of variance exceeds a threshold by a percentage of the trace of the new covariance matrix.

The extraction of principal components by diagonalization of the covariance matrix Σ is visualized in the following diagram. The shades of grey used to represent the covariance value varies from white (lowest value) to black (highest value):



Visualization of the extraction of eigenvalues in PCA

The eigenvalues (variance of X) are ranked by the decreasing order of their values. The PCA algorithm succeeds when the cumulative value of the last eigenvalues (the right-bottom section of the diagonal matrix) becomes insignificant.

Implementation

The principal components analysis can be easily implemented using the Apache Commons Math library methods that compute the eigenvalues and eigenvectors. The `PCA` class is defined as a data transformation of the `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*

The `PCA` class has a single argument: the `xt` training set (line 1). The output type has a `Double` for the projection of an observation along with the eigenvectors (line 2). The constructor defines the z-score `norm` function (line 3):

```
class PCA[@specialized(Double) T <: AnyVal] {  
    xt: XVSeries[T]) (implicit f: T => Double)
```

```

extends ITransform[Array[T]](xt) with Monitor[T] { //1
  type V = Double      //2

  val norm = (xv: XVSeries[T]) => zScores(xv) //3
  val model: Option[PCAModel] = train //4
  override def |> : PartialFunction[U, Try[V]] =
}

```

The model for the PCA algorithm is defined by the `PCAModel` case class (line 4).

[

Triggering an implicit conversion



Implicit conversions can be invoked through an assignment to fully declared variables. For example, the conversion from `XVSeries[T]` to `XVseries[Double]` is invoked by the declaring the type of the target variable: `val z: XVSeries[Double] = xv` (line 4).

]

The model for the PCA algorithm, `PCAModel`, consists of the covariance matrix, covariance defined in the formula **M11**, and array of eigenvalues computed in the formula **M16**:

```

case class PCAModel(val covariance: DblMatrix,
  val eigenvalues: DblArray)

```

The `|>` transformative method implements the computation of the principal components (that is, the eigenvector and eigenvalues):

```

def train: Option[PCAModel] = zScores(xt).map(x => { //5
  val obs: DblMatrix = x.toArray
  val cov = new Covariance(obs).getCovarianceMatrix //6

  val transform = new EigenDecomposition(cov) //7
  val eigenVectors = transform.getV //8
  val eigenValues =
    new ArrayRealVector(transform.getRealEigenvalues)

  val covariance = obs.multiply(eigenVectors).getData //9
  PCAModel(covariance, eigenValues.toArray) //10
}) match {/* ... */}

```

The normalization function `zScores` performs the Z-score transformation (formula **M12**) (line 5). Next, the method computes the covariance matrix from the normalized data (line 6). The eigenvectors, `eigenVectors`, are computed (line 7) and then retrieved using the `getv` method in the Apache Commons Math `EigenDecomposition` class (line 8). The method computes the diagonal, transformed covariance matrix from the eigenvector (line 9). Finally, the data transformation returns an instance of the PCA model (line 10).

The `|>` predictive method consists of projecting an observation onto the principal components:

```
override def |> : PartialFunction[Array[T], Try[V]] = {  
    case x: Array[T]  
    if(isModel && x.length == dimension(xt)) =>  
        Try( inner(x, model.get.eigenvalues) )  
}
```

The `inner` method of the `XTSerie` object computes the dot product of the values `x` and the `eigenvalues` model.

Test case

Let's apply the PCA algorithm to extract a subset of the features that represents some of the financial metrics ratios of 34 S&P 500 companies. The metrics under consideration are as follows:

- Trailing Price-to-Earnings ratio (PE)
- Price-to-Sale ratio (PS)
- Price-to-Book ratio (PB)
- Return on Equity (ROE)
- Operation Margin (OM)

The financial metrics are described in the *Terminology* section under *Finances 101* in the *Appendix A, Basic Concepts*.

The input data is specified with the following format as a tuple (a ticker symbol and an array of five financial ratios, PE, PS, PB, ROE, and OM):

```
val data = Array[(String, DblVector)] (  
    // Ticker          PE      PS      PB      ROE      OM  
    ("QCOM", Array[Double](20.8, 5.32, 3.65, 17.65, 29.2)),  
    ("IBM",  Array[Double](13, 1.22, 12.2, 88.1, 19.9)),  
    ...  
)
```

The client code that executes the PCA algorithm is defined simply as follows:

```
val dim = data.head._2.size
val input = data.map( _._2.take(dim))
val pca = new PCA[Double](input) //11
show(s"PCA model: ${pca.toString}") //12
```

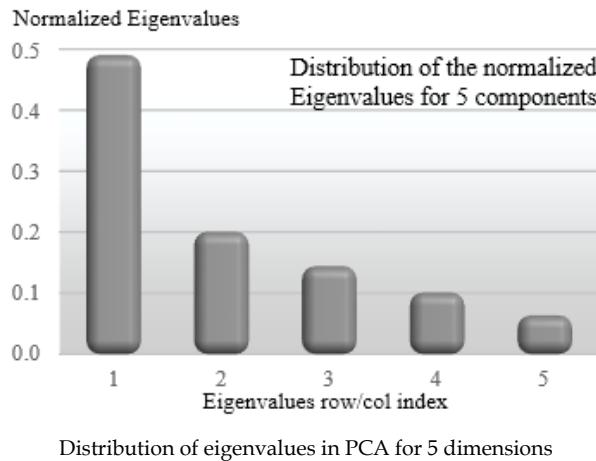
The PCA is instantiated with the `input` data (line 11) and then displayed in a textual format (line 12).

Evaluation

The first test on the 34 financial ratios uses a model that has five dimensions. As expected, the algorithm produces a list of five ordered eigenvalues:

2.5321, 1.0350, 0.7438, 0.5218, 0.3284

Let's plot the relative value of the eigenvalues (that is, relative importance of each feature) on the following bar chart:



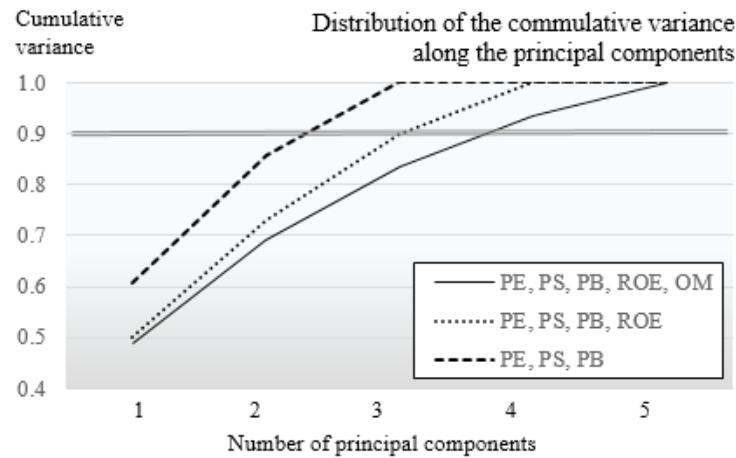
The chart shows that three out of five features account for 85 percent of the total variance (trace of the transformed covariance matrix). I invite you to experiment with different combinations of these features. The selection of a subset of the existing features is as simple as applying Scala's `take` or `drop` methods:

```
data.map( _._2.take(dim))
```

Let's plot the cumulative eigenvalues for the three different model configurations:

- **Five features:** PE, PS, PB, ROE, and OM
- **Four features:** PE, PS, PB, and ROE
- **Three features:** PE, PS, and PB

The graph will be as follows:



Distribution of eigenvalues in PCA for 3, 4, and 5 features

The chart displays the cumulative value of eigenvalues that are the variance of the transformed features, X_i . If we apply a threshold of 90 percent to the cumulative variance, then the number of principal components for each test model is as follows:

- $\{\text{PE, PS, PB}\}$: 2
- $\{\text{PE, PS, PB, ROE}\}$: 3
- $\{\text{PE, PS, PB, ROE, OM}\}$: 3

In conclusion, the PCA algorithm reduced the dimension of the model by 33 percent for the three-feature model, 25 percent for the four-feature model, and 40 percent for the five-feature model for a threshold of 90 percent.

Cross-validation of PCA

 Like any other unsupervised learning technique, the resulting principal components have to be validated through a one or K-fold cross-validation methodology using a regression estimator such as **Partial Least Square Regression** (PLSR) or the **Predicted Residual Error Sum of Squares** (PRESS). For those who are not afraid of statistics, I recommend that you read *Fast Cross-validation in Robust PCA* by S. Engelen and M. Hubert [4:14]. You need to be aware of the fact that the implementation of these regression estimators is not simple. The validation of the PCA is beyond the scope of this book.

Principal components analysis is a special case of the more general factor analysis. The latter class of algorithm does not require the transformation of the covariance matrix to be orthogonal.

Non-linear models

The principal components analysis technique requires the model to be linear. Although the study of such algorithms is beyond the scope of this book, it's worth mentioning two approaches that extend PCA for non-linear models:

- Kernel PCA
- Manifold learning

Kernel PCA

PCA extracts a set of orthogonal linear projections of an array of correlated values, $X = \{x_i\}$. The kernel PCA algorithm consists of extracting a similar set of orthogonal projection of the inner product matrix, $X^T X$. Nonlinearity is supported by applying a kernel function to the inner product. Kernel functions are described in the *Kernel functions* section in *Chapter 8, Kernel Models and Support Vector Machines*. The kernel PCA is an attempt to extract a low dimension feature set (or manifold) from the original observation space. The linear PCA is the projection on the tangent space of the manifold.

Manifolds

The concept of a manifold is borrowed from differential geometry. **Manifolds** generalize the notions of curves in a two-dimension space or surfaces in a three dimension space to a higher dimension. Nonlinear models are associated with Riemann manifolds whose metric is the inner product, $X^T X$, on a tangent space. The manifold represents a low dimension feature space embedded into the original observation space. The idea is to project the principal components from the linear tangent space to a manifold using a exponential map. This feat is accomplished using a variety of techniques from Local Linear Embedding and density preserving maps to Laplacian Eigenmaps [4:15].

The vector of observations cannot be directly used on a manifold because metrics such as norms or inner products depend on the location of the manifold the vector is applied to. Computation on manifolds relies on tensors such as contravariant and covariant vectors. Tensors algebra is supported by covariant and contravariant functors, which is introduced in the *Abstraction* section in *Chapter 1, Getting Started*.

Techniques that use differentiable manifolds are known as spectral dimensionality reduction.

Alternative dimension reduction techniques

Here are some more alternative techniques, listed as references: factor analysis, principal factor analysis, maximum likelihood factor analysis, independent component analysis, Random projection, nonlinear ICA, Kohonen's self-organizing maps, neural networks, and multidimensional scaling, just to name a few [4:16].

Manifold learning algorithms such as classifiers and dimension reduction techniques are associated with semi-supervised learning.

Performance considerations

The three unsupervised learning techniques share the same limitation – a high computational complexity.

K-means

The K-means has the computational complexity of $O(iKnm)$, where i is the number of iterations (or recursions), K is the number of clusters, n is the number of observations, and m is the number of features. Here are some remedies to the poor performance of the K-means algorithm:

- Reducing the average number of iterations by seeding the centroid using a technique such as initialization by ranking the variance of the initial cluster, as described in the beginning of this chapter
- Using a parallel implementation of K-means and leveraging a large-scale framework such as Hadoop or Spark
- Reducing the number of outliers and features by filtering out the noise with a smoothing algorithm such as a discrete Fourier transform or a Kalman filter
- Decreasing the dimensions of the model by following a two-step process:
 1. Execute a first pass with a smaller number of clusters K and/or a loose exit condition regarding the reassignment of data points. The data points close to each centroid are aggregated into a single observation.
 2. Execute a second pass on the aggregated observations.

EM

The computational complexity of the expectation-maximization algorithm for each iteration (E + M steps) is $O(m^2n)$, where m is the number of hidden or latent variables and n is the number of observations.

A partial list of suggested performance improvement includes:

- Filtering of raw data to remove noise and outliers
- Using a sparse matrix on a large feature set to reduce the complexity of the covariance matrix, if possible
- Applying the Gaussian mixture model wherever possible—the assumption of Gaussian distribution simplifies the computation of the log likelihood
- Using a parallel data processing framework such as Apache Hadoop or Spark, as discussed in the *Apache Spark* section in *Chapter 12, Scalable Frameworks*
- Using a kernel function to reduce the estimate of covariance in the E-step

PCA

The computational complexity of the extraction of the principal components is $O(m^2n + n^3)$, where m is the number of features and n is the number of observations. The first term represents the computational complexity for computing the covariance matrix. The second term reflects the computational complexity of the eigenvalue decomposition.

The list of potential performance improvements or alternative solutions for PCA includes the following:

- Assuming that the variance is Gaussian
- Using a sparse matrix to compute eigenvalues for problems with large feature sets and missing data
- Investigating alternatives to PCA to reduce the dimension of a model such as the **discrete Fourier transform (DFT)** or **singular value decomposition (SVD)** [4:17]
- Using PCA in conjunction with EM (at the research stage)
- Deploying a dataset on a parallel data processing framework such as Apache Hadoop or Spark, as described in the *Apache Spark* section in *Chapter 12, Scalable Frameworks*

Summary

This completes the overview of three of the most commonly used unsupervised learning techniques:

- K-means for clustering fully observed features of a model with reasonable dimensions
- Expectation-maximization for clustering a combination of observed and latent features
- Principal components analysis to transform and extract the most critical features in terms of variance for linear models

Manifold learning for non-linear models is a technically challenging field with great potential in terms of dynamic object recognition [4:18].

The key point to remember is that unsupervised learning techniques are used:

- By themselves to extract structures and associations from unlabelled observations
- As a preprocessing stage to supervised learning in reducing the number of features prior to the training phase

The distinction between unsupervised and supervised learning is not as strict as you may think. For instance, the K-means algorithm can be enhanced to support classification.

In the next chapter, we will address the second use case and cover supervised learning techniques starting with generative models.

5

Naïve Bayes Classifiers

This chapter introduces the most common and simple generative classifiers – Naïve Bayes. As mentioned earlier, generative classifiers are supervised learning algorithms that attempt to fit a *joint probability distribution* $p(X, Y)$ of two X and Y events, representing two sets of observed and hidden (or latent) variables, x and y .

In this chapter, you will learn, and hopefully appreciate, the simplicity of the Naïve Bayes technique through a concrete example. Then, you will learn how to build a Naïve Bayes classifier to predict the stock price movement, given some prior technical indicators in the analysis of financial markets.

Finally, you will learn how to apply Naïve Bayes to text mining by predicting stock prices using financial news feed and press releases.

Probabilistic graphical models

Let's start with a refresher course in basic statistics.

Given two events or observations X and Y , the joint probability of X and Y is defined as $p(X, Y) = p(X \cap Y)$. If the observations X and Y are not related, an assumption known as conditional independence, then $p(X, Y) = p(X).p(Y)$. The conditional probability of an event Y , given X , is defined as $p(Y | X) = p(X, Y)/p(X)$.

These two definitions are quite simple. However, **probabilistic reasoning** can be difficult to read in the case of large numbers of variables and sequences of conditional probabilities. As a picture is worth a thousand words, researchers introduced **graphical models** to describe a probabilistic relation between random variables using graphs [5:1].

There are two categories of graphs, and therefore, graphical models, which are as follows:

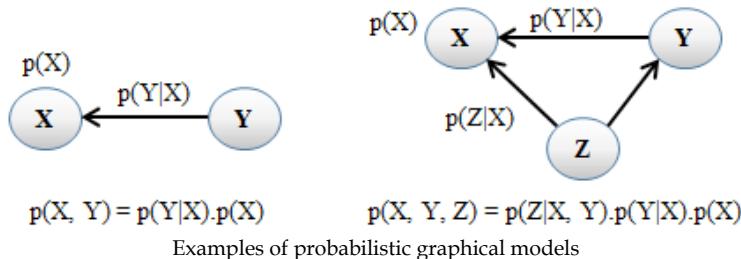
- Directed graphs such as Bayesian networks
- Undirected graphs such as conditional random fields (refer to the *Conditional random fields* section in *Chapter 7, Sequential Data Models*)

Directed graphical models are directed acyclic graphs that have been introduced to:

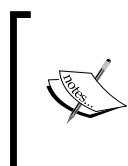
- Provide a simple way to visualize a probabilistic model
- Describe the conditional dependence between variables
- Represent a statistical inference in terms of connectivity between graphical objects

A Bayesian network is a directed graphical model that defines a joint probability over a set of variables [5:2].

The two joint probabilities $p(X, Y)$ and $p(X, Y, Z)$ can be graphically modeled using Bayesian networks, as follows:



The conditional probability $p(Y|X)$ is represented by an arrow directed from the output (or symptoms) Y to the input (or cause) X . Elaborate models can be described as a large directed graph between variables.



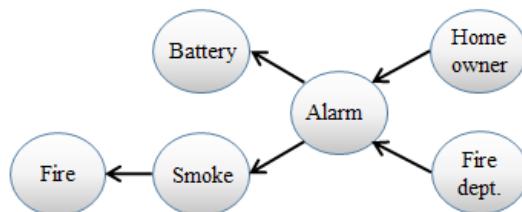
A metaphor for graphical models

From a software engineering perspective, graphical models visualize probabilistic equations in the same way the UML class diagram visualizes the object-oriented source code.

Here is an example of a real-world Bayesian network; the functioning of a smoke detector:

1. A fire may generate smoke.
2. Smoke may trigger an alarm.
3. A depleted battery may trigger an alarm.
4. The alarm may alert the homeowner.
5. The alarm may alert the fire department.

The flow diagram is as follows:



A Bayesian network for smoke detectors

This representation may be a bit counterintuitive, as the vertices are directed from the symptoms (or output) to the cause (or input). Directed graphical models are used in many different models, besides Bayesian networks [5:3].



Plate models

There are several alternate representations of probabilistic models, besides the directed acyclic graph, such as the plate model commonly used for the **Latent Dirichlet Allocation (LDA)** [5:4].

The Naïve Bayes models are probabilistic models based on the Bayes's theorem under the assumption of features independence, as mentioned in the *Generative models* section under *Supervised learning* in *Chapter 1, Getting Started*.

Naïve Bayes classifiers

The conditional independence between X features is an essential requirement for the Naïve Bayes classifier. It also restricts its applicability. The Naïve Bayes classification is better understood through simple and concrete examples [5:5].

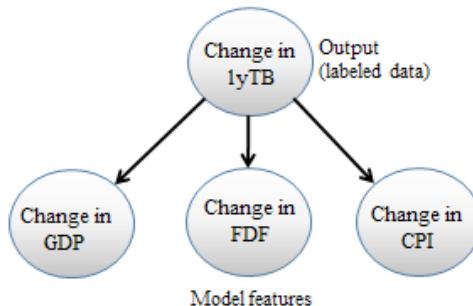
Introducing the multinomial Naïve Bayes

Let's consider the problem of how to predict a change in interest rates.

The first step is to list the factors that potentially may trigger or cause an increase or decrease in the interest rates. For the sake of illustrating Naïve Bayes, we will select the **consumer price index (CPI)** and change the **Federal fund rate (FDF)** and the **gross domestic product (GDP)**, as the first set of features. The terminology is described in the *Terminology* section under *Finances 101* in the *Appendix A, Basic Concepts*.

The use case is used to predict the direction of the change in the yield of the **1-year Treasury bill (1yTB)**, taking into account the change in the current CPI, FDF, and GDP. The objective is, therefore, to create a predictive model using a combination of these three features.

It is assumed that there is no available financial investment expert who can supply rules or policies to predict interest rates. Therefore, the model depends highly on the historical data. Intuitively, if one feature always increases when the yield of the 1-year Treasury bill increases, then we can conclude that there is a strong correlation of causal relationship between the features and the output variation in interest rates.



The Naive Bayes model for predicting the change in the yield of the 1-year T-bill

The correlation (or cause-effect relationship) is derived from the historical data. The methodology consists of counting the number of times each feature either increases (**UP**) or decreases (**DOWN**) and recording the corresponding expected outcome, as illustrated in the following table:

ID	GDP	FDF	CPI	1y-TB
1	UP	DOWN	UP	UP
2	UP	UP	UP	UP
3	DOWN	UP	DOWN	DOWN

4	UP	DOWN	DOWN	DOWN
...				
256	DOWN	DOWN	UP	DOWN

First, let's tabulate the number of occurrences of each change (**UP** and **DOWN**) for the three features and the output value (the direction of the change in the yield of the 1-year Treasury bill):

Number	GDP	FDF	CPI	1yTB
UP	169	184	175	159
DOWN	97	72	81	97
Total	256	256	256	256
UP/Total	0.66	0.72	0.68	0.625

Next, let's compute the number of positive directions for each of the features when the yield of the 1-year Treasury bill increases (159 occurrences):

Number	GDP	Fed Funds	CPI
UP	110	136	127
DOWN	49	23	32
Total	159	159	159
UP/Total	0.69	0.85	0.80

From the preceding table, we conclude that the yield of the 1-year Treasury bill increases when the GDP increases (69 percent of the time), the rate of the Federal funds increases (85 percent of the time), and the CPI increases (80 percent of the time).

Let's formalize the Naïve Bayes model before turning these findings into a probabilistic model.

Formalism

Let's start by clarifying the terminologies used in the Bayesian model:

- **Class prior probability or class prior:** This is the probability of a class
- **Likelihood:** This is the probability to observe a value or event, given a class, also known as the probability of the predictor, given a class
- **Evidence:** This is the probability of observations that occur, also known as the prior probability of the predictor
- **Posterior probability:** This is the probability of an observation x being in a given class

No model can be simpler! The log likelihood, $\log p(x_i | C_j)$, is commonly used instead of the likelihood in order to reduce the impact of the features x_i that have a low likelihood.

The objective of the Naïve Bayes classification of a new observation is to compute the class that has the highest log likelihood. The mathematical notation for the Naïve Bayes model is also straightforward.

The Naïve Bayes classification

M1: The posterior probability $p(C_j | x)$ is defined as:

$$p(C_j | x) = \frac{p(x | C_j) \cdot p(C_j)}{p(x)}$$

Here, $x = \{x_i\} (0, n-1)$ is a set of n features. $\{C_j\}$ is a set of classes with their class prior $p(C_j)$. $x = \{x_i\} (0, n-1)$ with a set of n features. $p(x | C_j)$ is the likelihood for each feature

 M2: The computation of the posterior probability $p(C_j | x)$ is simplified by the assumption of conditional independence of features:

$$p(C_j | x) = \prod_{i=0}^{n-1} p(x_i | C_j) \cdot p(C_j)$$

Here, x_i are independent and the probabilities are normalized for evidence $p(x) = 1$.

M3: The **maximum likelihood estimate (MLE)** is defined as:

$$\mathcal{L}(C_j|x) = \sum_{i=0}^{n-1} \{\log(p(x_i|C_i)) + \log(p(C_j))\}$$

M4: The Naïve Bayes classification of an observation x of a class C_m is defined as:

$$C_m = \arg \max_j \mathcal{L}(C_j|x)$$

This particular use case has a major drawback—the GDP statistics are provided quarterly, while the CPI data is made available once a month and a change in FDF rate is rather infrequent.

The frequentist perspective

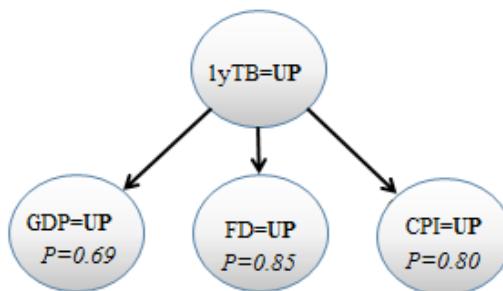
The ability to compute the posterior probability depends on the formulation of the likelihood using historical data. A simple solution is to count the occurrences of observations for each class and compute the frequency.

Let's consider the first example that predicts the direction of the change in the yield of the 1-year Treasury bill, given changes in the GDP, FDF, and CPI.

The results are expressed with simple probabilistic formulas and a directed graphical model:

$$\begin{aligned} P(\text{GDP=UP} | \text{1yTB=UP}) &= 110/159 \\ P(\text{1yTB=UP}) &= \text{num occurrences (1yTB=UP)} / \text{total num of} \\ &\quad \text{occurrences} = 159/256 \\ p(\text{1yTB=UP} | \text{GDP=UP, FDF=UP, CPI=UP}) &= p(\text{GDP=UP} | \text{1yTB=UP}) \times \\ &\quad p(\text{FDF=UP} | \text{1yTB=UP}) \times \end{aligned}$$

$$p(CPI=UP | 1yTB=UP) \times \\ p(1yTB=UP) = 0.69 \times 0.85 \times 0.80 \times \\ 0.625$$



The Bayesian network for the prediction of the change of the yield of the 1-year Treasury bill

Overfitting

The Naïve Bayes model is not immune to overfitting if the number of observations is not large enough relative to the number of features. One approach to address this problem is to perform a feature selection using the mutual information exclusion [5:6].

This problem is not a good candidate for a Bayesian classification for the following two reasons:

- The training set is not large enough to compute accurate prior probabilities and generate a stable model. Decades of quarterly GDP data is needed to train and validate the model.
- The features have different rates of change, which predominately favor the features with the highest frequency; in this case, the CPI.

Let's select another use case for which a large historical dataset is available and can be automatically labeled.

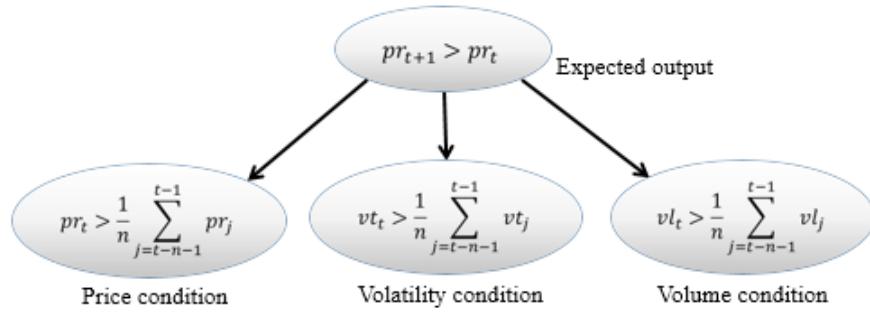
The predictive model

The predictive model is the second use case that consists of predicting the direction of the change of the closing price of a stock, $pr_{t+1} = \{UP, DOWN\}$, at trading day $t + 1$, given the history of its direction of the price, volume, and volatility for the previous t days, pr_i for $i = 0$ to $i = t$. The volume and volatility features have already been used in the *Writing a simple workflow* section in *Chapter 1, Getting Started*.

Therefore, the three features under consideration are as follows:

- The closing price pr_t of the last trading session, t , is above or below the average closing price over the n previous trading days, $[t-n, t]$.
- The volume of the last trading day vl_t is above or below the average volume of the n previous trading days
- The volatility on the last trading day vt_t is above or below the average volatility of the previous n trading days

The directed graphic model can be expressed using one output variable (the price at session $t + 1$ is greater than the price at session t) and three features: the price condition (1), volume condition (2), and volatility condition (3).



The Bayesian model for predicting the future direction of the stock price

This model works under the assumption that there is at least one observation or ideally few observations for each feature and expected value.

The zero-frequency problem

It is possible that the training set does not contain any data actually observed for a feature for a specific label or class. In this case, the mean is $0/N = 0$, and therefore, the likelihood is null, making classification unfeasible. The case for which there are only few observations for a feature in a given class is also an issue, as it skews the likelihood.

There are a couple of correcting or smoothing formulas for unobserved features or features with a low number of occurrences that address this issue, such as the **Laplace** and **Lidstone** smoothing formulas.

The smoothing factor for counters

M5: The Laplace smoothing formula of the mean k/N out of N observations of features of dimension n is defined as:



$$\mu' = \frac{k + 1}{N + n}$$

M6: The Lidstone smoothing formula with a factor a is defined as:

$$\mu' = \frac{k + a}{N + a.n}$$

The two formulas are commonly used in natural language processing applications, for which the occurrence of a specific word or tag is a feature [5:7].

Implementation

I think it is time to write some Scala code and toy around with Naïve Bayes. Let's start with an overview of the software components.

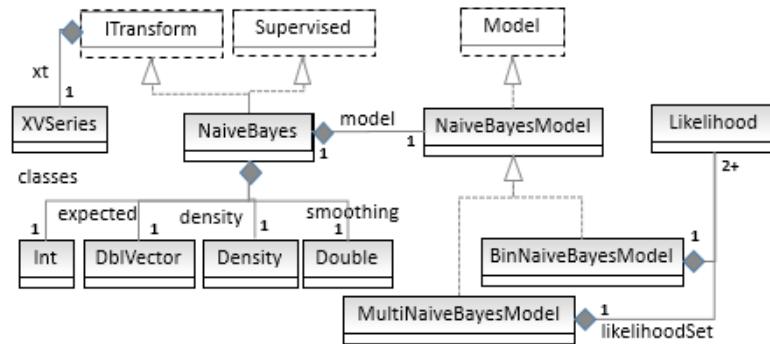
Design

Our implementation of the Naïve Bayes classifier uses the following components:

- A generic model, `NaiveBayesModel`, of the `Model` type that is initialized through training during the instantiation of the class.
- A model for the `BinNaiveBayesModel` binomial classification, which subclasses `NaiveBayesModel`. The model consists of a pair of positive and negative `Likelihood` class instances.
- A model for the `MultiNaiveBayesModel` multinomial classification.
- The `NaiveBayes` classifier class has four parameters: a smoothing function, such as `Laplace` and a set of observations of the `xvSeries` type, a set of labels of the `DblVector` type, a log density function of the `LogDensity` type, and the number of classes.

The principle of software architecture applied to the implementation of classifiers is described in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*.

The key software components of the Naïve Bayes classifier are described in the following UML class diagram:



The UML class diagram for the Naïve Bayes classifier

The UML diagram omits the helper traits or classes such as Monitor or Apache Commons Math components.

Training

The objective of the training phase is to build a model consisting of the likelihood for each feature and the class prior. The likelihood for a feature is identified as follows:

- The number of occurrences k of this feature for $N > k$ observations in the case of binary features or counters
- The mean value for all the observations for this feature in the case of numeric or continuous features

It is assumed, for the sake of this test case, that the features, that is, technical analysis indicators, such as price, volume, and volatility are conditionally independent. This assumption is not actually correct.

Conditional dependency

Recent models, known as **Hidden Naïve Bayes (HNB)**, relax the restrictions on the independence between features. The HNB algorithm uses conditional mutual information to describe the interdependency between some of the features [5:8].

Let's write the code to train the binomial and multinomial Naïve Bayes.

Class likelihood

The first step is to define the class likelihood for each feature using historical data.

The `Likelihood` class has the following attributes (line 1):

- The label for the `label` observation
- An array of tuple Laplace or Lidstone smoothed mean and standard deviation, `muSigma`
- The prior probability of a prior class

As with any code snippet presented in this book, the validation of class parameters and method arguments are omitted in order to keep the code readable:

```
class Likelihood[T <: AnyVal] {  
    val label: Int,  
    val muSigma: Vector[DblPair],  
    val prior: Double) (implicit f: T => Double) { //1  
  
    def score(obs: Array[T], logDensity: LogDensity): Double = //2  
        (obs, muSigma).zipped  
        .map{ case(x, (mu, sig)) => (x, mu, sig)}  
        .//:(0.0)((prob, entry) => {  
            val x = entry._1  
            val mean = entry._2  
            val stdDev = entry._3  
            val logLikelihood = logDensity(mean, stdDev, x) //3  
            val adjLogLikelihood = if(logLikelihood < MINLOGARG)  
                MINLOGVALUE else logLikelihood  
            prob + Math.log(adjLogLikelihood) //4  
        }) + Math.log(prior)  
    }  
}
```

The parameterized `Likelihood` class has the following two purposes:

- Define the statistics regarding a class C_k : its label, its mean and standard deviation, and the prior probability $p(C_k)$.
- Compute the score of a new observation for its runtime classification (line 2). The computation of the log of the likelihood uses a `logDensity` method of the `LogDensity` type (line 3). As seen in the next section, the log density can be either a Gaussian or a Bernoulli distribution. The `score` method uses Scala's `zipped` method to merge the observation values with the labeled values and implements the M3 formula (line 4).

The Gaussian mixture is particularly suited for modeling datasets, for which the features have large sets of discrete values or are continuous variables. The conditional probabilities for the feature x is described by the normal probability density function [5:9].

The log likelihood using the Gaussian density

M7: For a Lidstone or Laplace smoothed mean μ' and a standard deviation σ , the log likelihood of a posterior probability for a Gaussian distribution is defined as:

$$\mathcal{L}(C_j|x) = \sum_{i=0}^{n-1} \left\{ -\frac{1}{2} \log(2\pi) - \log(\sigma) - \frac{(x - \mu')^2}{2\sigma^2} + \log(p(C_j)) \right\}$$

The log of the Gauss, `logGauss`, and the log of the Normal distribution, `logNormal`, are defined in the `stats` class, which was introduced in the *Profiling data* section in *Chapter 2, Hello World!*:

```
def logGauss(mean: Double, stdDev: Double, x: Double): Double = {
    val y = (x - mean)/stdDev
    -LOG_2PI - Math.log(stdDev) - 0.5*y*y
}
val logNormal = logGauss(0.0, 1.0, _: Double)
```

The `logNormal` computation is implemented as a partial applied function.

The functions of the `LogDensity` type compute the probability density for each feature (line 5):

```
type LogDensity = (Double*) => Double
```

Binomial model

The next step is to define the `BinNaiveBayesModel` model for a two-class classification scheme. The two-class model consists of two `Likelihood` instances: positives for the label UP (value 1) and negatives for the label DOWN (value 0).

In order to make the model generic, we created a `NaiveBayesModel` trait that can be extended as needed to support both the binomial and multinomial Naïve Bayes models, as follows:

```
trait NaiveBayesModel[T] {
    def classify(x: Array[T], logDensity: LogDensity): Int //5
}
```

The `classify` method uses the trained model to classify a multivariate observation x of the `Array[T]` type given a `logDensity` probability density function (line 5). The method returns the class the observation belongs to.

Let's start with the definition of the `BinNaiveBayesModel` class that implements the binomial Naïve Bayes:

```
class BinNaiveBayesModel[T <: AnyVal] (
    pos: Likelihood[T],
    neg: Likelihood[T]) (implicit f: T => Double)
extends NaiveBayesModel[T] { //6

    override def classify(x: Array[T], logDensity: logDensity): Int = //7
        if(pos.score(x,density) > neg.score(x,density)) 1 else 0
    ...
}
```

The constructor for the `BinNaiveBayesModel` class takes two arguments:

- `pos`: Class likelihood for observations with a positive outcome
- `neg`: Class likelihood for observations with a negative outcome (line 6)

The `classify` method is called by the `|>` operator in the Naïve Bayes classifier. It returns 1 if the observation x matches the `Likelihood` class that contains the positive cases, and 0 otherwise (line 7).

Model validation

The parameters of the Naïve Bayes model (likelihood) are computed through training and the `model` value is instantiated regardless of whether the model is actually validated in this example. A commercial application would require the model to be validated using a methodology such as the K-fold validation and F1 measure, as described in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*.

The multinomial model

The multinomial Naïve Bayes model defined by the `MultiNaiveBayesModel` class is very similar to the `BinNaiveBayesModel`:

```
class MultiNaiveBayesModel[T <: AnyVal] //8
    likelihoodSet: Seq[Likelihood[T]] (implicit f: T => Double)
extends NaiveBayesModel[T] {
```

```

override def classify(x: Array[T], logDensity: LogDensity): Int = {
    val <<< = (p1: Likelihood[T], p2: Likelihood[T]) =>
        p1.score(x, density) > p1.score(x, density) //9
    likelihoodSet.sortWith(<<<).head.label //10
}
...
}

```

The multinomial Naïve Bayes model differs from its binomial counterpart as follows:

- Its constructor requires a sequence of class likelihood `likelihoodSet` (line 8).
- The `classify` runtime classification method sorts the class likelihoods by their score (posterior probability) using the `<<<` function (line 9). The method returns the ID of the class with the highest log likelihood (line 10).

Classifier components

The Naïve Bayes algorithm is implemented as a data transformation using a model implicitly extracted from a training set of the `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*

The attributes of the multinomial Naïve Bayes are as follows:

- The smoothing formula (Laplace, Lidstone, and so on), `smoothing`
- The set of multivariable observations defined as `xt`
- The expected values (or labels) associated with the set of observations, `expected`
- The log of the probability density function, `logDensity`
- The number of classes—two for the binomial Naïve Bayes with the `BinNaiveBayesModel` type or more for the multinomial Naïve Bayes with the `MultiNaiveBayesModel` type (line 11)

The code will be as follows:

```

class NaiveBayes[T <: AnyVal] (
    smoothing: Double,
    xt: XVSeries[T],
    expected: Vector[Int],
    logDensity: LogDensity,
    classes: Int)(implicit f: T => Double)
extends ITransform[Array[T]](xt)

```

```
    with Supervised[T, Array[T]] with Monitor[Double] { //11
  type V = Int //12
  val model: Option[NaiveBayesModel[T]] //13
  def train(expected: Int): Likelihood[T]
  ...
}
```

The `Monitor` trait defines miscellaneous logging and display functions.

Data transformation of the `ITransform` type requires the output type `V` to be specified (line 12). The output of the Naïve Bayes is the index of the class an observation belongs to. The `model` type of the model can be either `BinNaiveBayesModel` for two classes or `MultiNaiveBayesModel` for a multinomial model (line 13):

```
val model: Option[NaiveBayesModel[T]] = Try {
  if(classes == 2)
    BinNaiveBayesModel[T](train(1), train(0))
  else
    MultiNaiveBayesModel[T](List.tabulate(classes)(train(_)))
} match {
  case Success(_model) => Some(_model)
  case Failure(e) => /* ... */
}
```

Training and class instantiation

There are several benefits of allowing the instantiation of the Naïve Bayes mode only once when it is trained. It prevents the client code from invoking the algorithm on an untrained or partially trained model, and it reduces the number of states of the model (untrained, partially trained, trained, validated, and so on). It is an elegant way to hide the details of the training of the model from the user.

The `train` method is applied to each class. It takes the index or label of the class and generates its log likelihood data (line 14):

```
def train(index: Int): Likelihood[T] = { //14
  val xv: XVSeries[Double] = xt
  val values = xv.zip(expected) //15
    .filter(_._2 == index).map(_._1) //16
  if(values.isEmpty)
    throw new IllegalStateException(/* ... */)
```

```

    val dim = dimension(xt)
    val meanStd = statistics(values).map(stat =>
      (stat.lidstoneMean(smoothing, dim), stat.stdDev)) //17
    Likelihood(index, meanStd, values.size.toDouble/xv.size) //18
  }

```

The training set is generated by zipping the `xt` vector of observations with expected classes, `expected` (line 15). The method filters out the observation for which the label does not correspond to this class (line 16). The `meanStd` pair (mean and standard deviation) is computed using the Lidstone smoothing factor (line 17). Finally, the training method returns the class likelihood corresponding to the index `label` (line 18).

The `NaiveBayes` class also defines the `|>` runtime classification method and the F_1 validation methods. Both methods are described in the next section.



Handling missing data

Naïve Bayes has a no-nonsense approach to handling missing data. You just ignore the attribute in the observations for which the value is missing. In this case, the prior for this particular attribute for these observations is not computed. This workaround is obviously made possible because of the conditional independence between features.

The `apply` constructor for `NaiveBayes` returns the `NaiveBayes` type:

```

object NaiveBayes {
  def apply[T <: AnyVal] (
    smoothing: Double,
    xt: XVSeries[T],
    expected: Vector[Int],
    logDensity: LogDensity,
    classes: Int) (implicit f: T => Double): NaiveBayes[T] =
    new NaiveBayes[T](smoothing, xt, y, logDensity, classes)

  ...
}

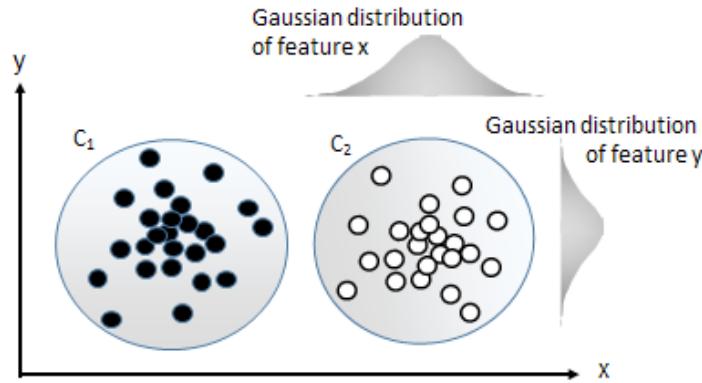
```

Classification

The likelihood and class prior that have been computed through training is used for validating the model and classifying new observations.

The score represents the log of likelihood estimate (or the posterior probability), which is computed as the summation of the log of the Gaussian distribution using the mean and standard deviation extracted from the training phase and the log of the class likelihood.

The Naïve Bayes classification using the Gaussian distribution is illustrated using the two C_1 and C_2 classes and a model with two features (x and y):



An illustration of the Gaussian Naive Bayes using 2-dimensional model

The `|>` method returns the partial function that implements the runtime classification of a new x observation using one of the two Naïve Bayes models. The `model` and the `logDensity` functions are used to assign the x observation to the appropriate class (line 19):

```
override def |> : PartialFunction[Array[T], Try[V]] = {  
    case x: Array[T] if(x.length >0 && model != None) =>  
        Try(model.map(_.classify(x, logDensity)).get) //19  
}
```

F₁ validation

Finally, the Naïve Bayes classifier is implemented by the `NaiveBayes` class. It implements the training and runtime classification using the Naïve Bayes formula. In order to force the developer to define a validation for any new supervised learning technique, the class inherits from the `Supervised` trait that declares the `validate` validation method:

```
trait Supervised[T, V] {  
    self: ITransform[V] => //20  
    def validate(xt: XVSeries[T],  
                expected: Vector[V]): Try[Double] //21  
}
```

The validation of a model applies only to a data transformation of the `ITransform` type (line 20).

The `validate` method takes the following arguments (line 21):

- An `xt` time series of multidimensional observations
- An `expected` vector of expected class values

By default, the `validate` method returns the F_1 score for the model, as described in the *Assessing a model* section in *Chapter 2, Hello World!*

Let's implement the key functionality of the `Supervised` trait for the Naïve Bayes classifier:

```
override def validate(
    xt: XVSeries[T],
    expected: Vector[V]): Try[Double] = Try {           //22
  val predict = model.get.classify(_:Array[Int], logDensity) //23
  MultiFValidation(expected, xt, classes)(predict).score //24
}
```

The predictive `predict` partially applied function is created by assigning a predicted class to a new `x` observation (line 23), and then the prediction, the index of classes, is loaded into the `MultiFValidation` class to compute the F_1 score (line 24).

Feature extraction

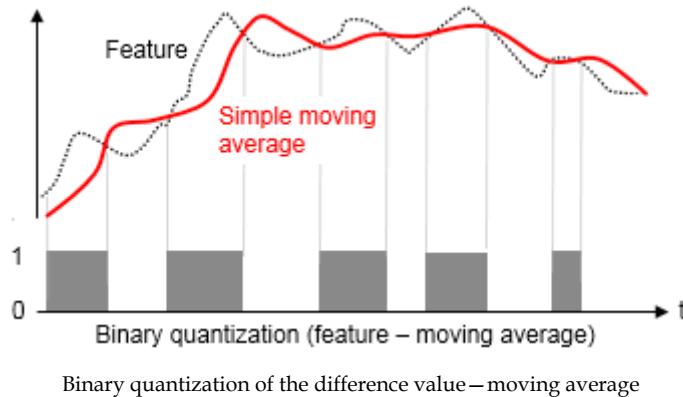
The most critical element in the training of a supervised learning algorithm is the creation of labeled data. Fortunately, in this case, the labels (or expected classes) can be automatically generated. The objective is to predict the direction of the price of a stock for the next trading day, taking into account the moving average price, volume, and volatility over the last n days.

The extraction of features follows these six steps:

1. Extract historical trading data of each feature (that is, price, volume, and volatility).
2. Compute the simple moving average of each feature.
3. Compute the difference between the value and moving average of each feature.
4. Normalize the difference by assigning 1 for positive values and 0 for negative values.

5. Generate a time series of the difference between the closing price of the stock and the closing price of the previous trading session.
6. Normalize the difference by assigning 1 for positive values and 0 for negative values.

The following diagram illustrates the feature extractions for steps 1 to 4:



The first step is to extract the average price, volume, and volatility (that is, $1 - \text{low}/\text{high}$) for each stock during the period of Jan 1, 2000 and Dec 31, 2014 with daily and weekly closing prices. Let's use the simple moving average to compute these averages for the $[t-n, t]$ window.

The extractor variable defines the list of features to extract from the financial data source, as described in the *Data extraction* and *Data sources* section in the *Appendix A, Basic Concepts*:

```
val extractor = toDouble(CLOSE)    // stock closing price
                     :: ratio(HIGH, LOW) //volatility(HIGH-LOW)/HIGH
                     :: toDouble(VOLUME) // daily stock trading volume
                     :: List[Array[String] =>Double] ()
```

The naming convention for the trading data and metrics is described in the *Trading data* section under *Technical analysis* in the *Appendix A, Basic Concepts*.

The training and validation of the binomial Naïve Bayes is implemented using a monadic composition:

```
val trainRatio = 0.8 //25
val period = 4
val symbol ="IBM"
val path = "resources/chap5"
```

```

val pfnMv = SimpleMovingAverage[Double](period, false) |> //26
val pfnSrc = DataSource(symbol, path, true, 1) |> //27

for {
    obs <- pfnSrc(extractor) //28
    (x, delta) <- computeDeltas(obs) //29
    expected <- Try{difference(x.head.toVector, diffInt)}//30
    features <- Try { transpose(delta) } //31
    labeled <- //32
    OneFoldXValidation[Int](features, expected, trainRatio)
    nb <- NaiveBayes[Int](1.0, labeled.trainingSet) //33
    f1Score <- nb.validate(labeled.validationSet) //34
}
yield {
    val labels = Array[String](
        "price/ave price", "volatility/ave. volatility",
        "volume/ave. volume"
    )
    show(s"\nModel: ${nb.toString(labels)}")
}

```

The first step is to distribute the observations between the training set and validation set. The `trainRatio` value (line 25) defines the ratio of the original observation set to be included in the training set. The simple moving average values are generated by the `pfnMv` partial function (line 26). The extracting `pfnSrc` partial function (line 27) is used to generate the three trading time series, price, volatility, and volume (line 28).

The next step consists of applying the `pfnMv` simple moving average to the `obs` multidimensional time series (line 29) using the `computeRatios` method:

```

type LabeledPairs = (XVSeries[Double], Vector[Array[Int]])

def computeDeltas(obs: XVSeries[Double]): Try[LabeledPairs] =
Try{
    val sm = obs.map(_.toVector).map( pfnMv(_).get.toArray) //35
    val x = obs.map(_.drop(period-1) )
    (x, x.zip(sm).map{ case(x,y) => x.zip(y).map(delta(_)) })//36
}

```

The `computeDeltas` method computes the time series of the `sm` observations smoothed with a simple moving average (line 35). The method generates a time series of 0 and 1 for each of the three features in the `xs` observation set and `sm` smoothed dataset (line 36).

Next, the call to the `difference` differential computation generates the labels (0 and 1) representing the change in the direction of the price of a security between two consecutive trading sessions: 0 if the price is decreased and 1 if the price is increased (line 30) (refer to the *The differential operator* section under *Time series in Scala* in *Chapter 3, Data Preprocessing*).

The features for the training of the Naïve Bayes model are extracted from these ratios by transposing the ratios-time series matrix in the `transpose` method of the `XTSeries` singleton (line 31).

Next, the training set and validation set are extracted from the `features` set using the `OneFoldXValidation` class, which was introduced in the *One-fold cross validation* section under *Cross-validation* in *Chapter 2, Hello World!* (line 32).

Selecting the training data

In our example, the training set is simplistically the first `trainRatio` multiplied by the size of dataset observations. Practical applications use a K-fold cross-validation technique to validate models, as described in the *K-fold cross validation* section under *Assessing a model* in *Chapter 2, Hello World!*. A simpler alternative is to create the training set by picking observations randomly and using the remaining data for validation.

The last two stages in the workflow consists of training the Naïve Bayes model by instantiating the `NaiveBayes` class (line 33) and computing the F_1 score for different values of the smoothing coefficient of the simple moving average applied to the stock price, volatility, and volume (line 34).

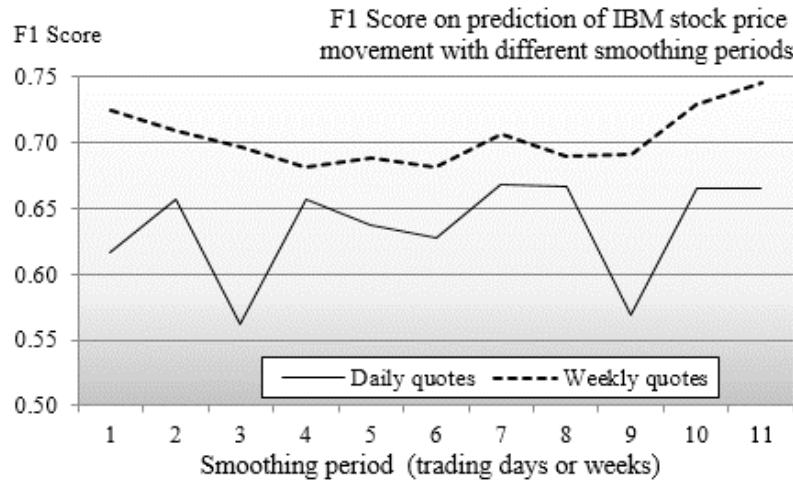
The implicit conversion

The `NaiveBayes` class operates on elements of the `Int` and `Double` types, and therefore, assumes that there is conversion between `Int` and `Double` (view bounded). The Scala compiler may generate a warning because the conversion from `Int` to `Double` has not been defined. Although Scala relies on its own conversion functions, I would recommend that you explicitly define and control your conversion function:

```
implicit def intToDouble(n: Int): Double = n.toDouble
```

Testing

The next chart plots the value of the F_1 measure of the predictor of the direction of the IBM stock using price, volume, and volatility over the previous n trading days, with n varying from 1 to 12 trading days:



A graph of the F1-measure for the validation of the Naïve Bayes model

The preceding chart illustrates the impact of the value of the averaging period (number of trading days) on the quality of the multinomial Naïve Bayes prediction, using the value of the stock price, volatility, and volume relative to their average over the averaging period.

From this experiment, we conclude the following:

- The prediction of the stock movement using the average price, volume, and volatility is not very good. The F_1 score for the models using weekly (with respect to daily) closing prices varies between 0.68 and 0.74 (with respect to 0.56 and 0.66).
- The prediction using weekly closing prices is more accurate than the prediction using the daily closing prices. In this particular example, the distribution of the weekly closing prices is more reflective of an intermediate term trend than the distribution of daily prices.
- The prediction is somewhat independent of the period used to average the features.

The Multivariate Bernoulli classification

The previous example uses the Gaussian distribution for features that are essentially binary ($UP = 1$ and $DOWN = 0$) to represent the change in value. The mean value is computed as the ratio of the number of observations for which $x_i = UP$ over the total number of observations.

As stated in the first section, the Gaussian distribution is more appropriate for either continuous features or binary features for very large labeled datasets. The example is the perfect candidate for the **Bernoulli** model.

Model

The Bernoulli model differs from the Naïve Bayes classifier in such a way that it penalizes the feature x that does not have any observation; the Naïve Bayes classifier ignores it [5:10].

The Bernoulli mixture model

M8: For a feature function f_k with $f_k = 1$, if the feature is observed, and a value of 0 otherwise, and the probability p of the observed feature x_k belongs to the class C_j , then the posterior probability is computed as follows:

$$p(f_i|C_j) = \prod_{k=0}^{n-1} (f_k \cdot p(x_k|C_j) + (1 - f_k)(1 - p(x_k|C_j))$$

Implementation

The implementation of the Bernoulli model consists of modifying the `score` function in the `Likelihood` class using the Bernoulli density method, `bernoulli`, defined in the `Stats` object:

```
object Stats {  
    def bernoulli(mean: Double, p: Int): Double =  
        mean*p + (1-mean)*(1-p)  
    def bernoulli(x: Double*): Double = bernoulli(x(0), x(1).toInt)  
    ...  
}
```

The first version of the Bernoulli algorithm is the direct implementation of the M8 mathematical formula. The second version uses the signature of the `Density` (`Double*`) \Rightarrow `Double` type.

The mean value is the same as in the Gaussian density function. The binary feature is implemented as an `Int` type with the value `UP = 1` (with respect to `DOWN = 0`) for the upward (with respect to downward) direction of the financial technical indicator.

Naïve Bayes and text mining

The multinomial Naïve Bayes classifier is particularly suited for **text mining**. The Naïve Bayes formula is quite effective to classify the following entities:

- E-mail spams
- Business news stories
- Movie reviews
- Technical papers as per field of expertise

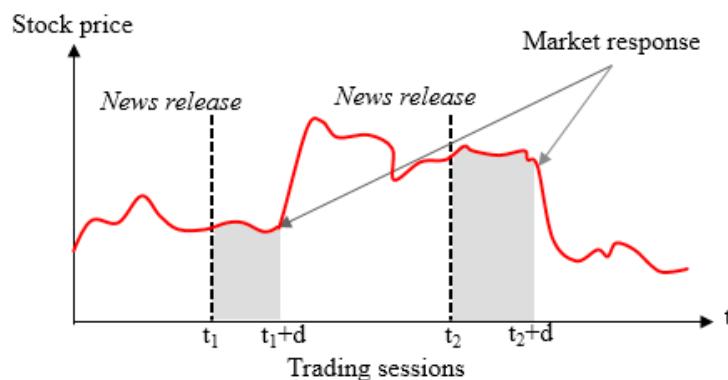
This third use case consists of predicting the direction of a stock given the financial news. There are two type of news that affect the stock of a particular company:

- **Macro trends:** Economic or social news such as conflicts, economic trends, or labor market statistics
- **Micro updates:** Financial or market news related to a specific company such as earnings, change in ownership, or press releases

Macroeconomic news related to a specific company have the potential to affect the sentiments of investors toward the company and may lead to a sudden shift in the price of its stock. Another important feature is the average time it takes for investors to react to the news and affect the price of the stock.

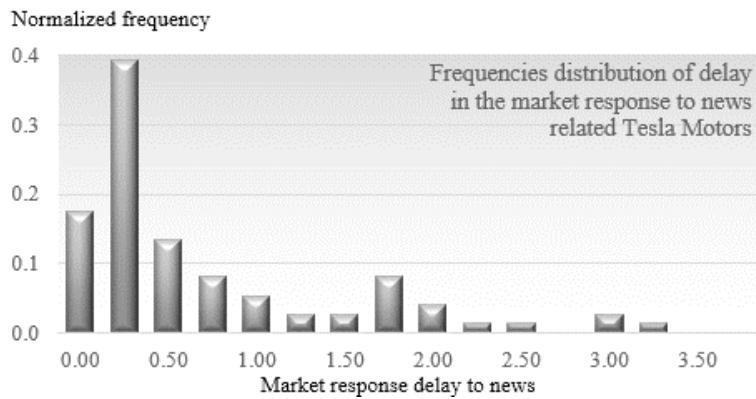
- Long-term investors may react within days or even weeks
- Short-term traders adjust their positions within hours, sometimes within the same trading session

The average time the market takes to react to a significant financial news on a company is illustrated in the following chart:



An illustration of the reaction of investors on the price of a stock following a news release

The delay in the market response is a relevant feature only if the variance of the response time is significant. The distribution of the frequencies of the delay in the market response to any newsworthy articles regarding TSLA is fairly constant. It shows that the stock prices react within the same day in 82 percent of the cases, as seen in the following bar chart:

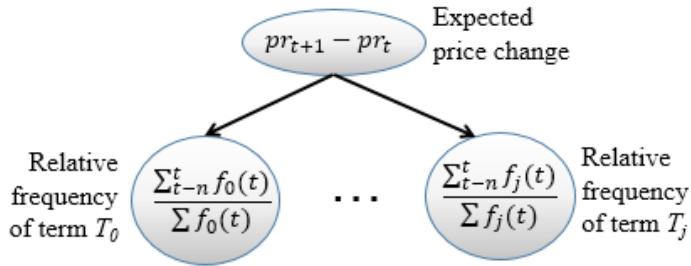


The distribution of the frequencies of the reaction of investors on the price of a stock following a news release

The frequency peak for a market response delay of 1.75 days can be explained by the fact that some news are released over the weekend and investors have to wait until the following Monday to drive the stock price higher or lower. Another challenge is to assign any shift of a stock price to a specific news release, taking into account that some news can be redundant, confusing, or simultaneous.

Therefore, the model features for predicting the stock price pr_{t+1} are the relative frequency f_i of an occurrence of a term T_i within a time window $[t-n, t]$, where t and n are trading days.

The following graphical model formally describes the causal relation or conditional dependency of the relative change of the stock price between two consecutive trading sessions t and $t + 1$, given the relative frequency of appearance of some key terms in the media:



The Bayesian model for the prediction of the stock movement given financial news

For this exercise, the observation sets are the corpus of news feeds and articles released by the most prominent financial news organizations, such as Bloomberg or CNBC. The first step is to devise a methodology to extract and select the most relevant terms associated with a specific stock.

Basics of information retrieval

A full discussion of information retrieval and text mining is beyond the scope of this book [5:11]. For the sake of simplicity, the model will rely on a very simple scheme for extracting relevant terms and computing their relative frequency. The following 10-step sequence of actions describe one of the numerous methodologies used to extract the most relevant terms from a corpus:

1. Create or extract the timestamp for each news article.
2. Extract the title, paragraph, and sentences of each article using a Markovian classifier.
3. Extract the terms from each sentence using regular expressions.
4. Correct terms for typos using a dictionary and metric such as the **Levenshtein** distance.
5. Remove the nonstop words.
6. Perform **stemming** and **lemmatization**.
7. Extract bags of words and generate a list of **n-grams** (as a sequence of n terms).
8. Apply a **tagging model** build using a maximum entropy or conditional random field to extract nouns and adjectives (for example, *NN*, *NNP*, and so on).

9. Match the terms against a dictionary that supports senses, hyponyms, and synonyms, such as **WordNet**.
10. Disambiguate word sense using Wikipedia's repository **DBpedia** [5:12].

 **Text extraction from the Web**

The methodology discussed in this section does not include the process of searching and extracting news and articles from the Web that requires additional steps such as search, crawling, and scraping [5:13].

Implementation

Let's apply the text mining methodology template to predict the direction of a stock, given the financial news. The algorithm relies on a sequence of seven simple steps:

1. Searching and loading the news articles related to a given company and its stock as a \square_t document of the Document type.
2. Extracting the date: T timestamp of the article using a regular expression.
3. Ordering the \square_t documents as per the timestamp.
4. Extracting the $\{T_{i,D}\}$ terms from the content of each \square_t document.
5. Aggregating the $\{T_{t,D}\}$ terms for all the \square_t documents that share the same publication date t .
6. Computing the rtf relative frequency of each $\{T_{i,D}\}$ term for the date t , as the ratio of number of its occurrences in all the articles released at t to the total number of its occurrences of the term in the entire corpus.
7. Normalizing the relative frequency for the average number of articles per date, $nrtf$.

Text analysis metrics

M9: The relative frequency of occurrences for term (or keyword) t_i with n_i^a occurrences in the article a is defined as follows:

$$rtf(t_i) = \frac{\sum_{a \in Dt} n_i^a}{\sum_{a \in Corpus} n_i^a}$$

 M10: The relative frequency of occurrences of a term t_i normalized by the daily average number of articles for which N_a is the total number of articles and N_d is the number of days in the survey is defined as follows:

$$nrft(t_i) = \frac{rtf(t_i) \cdot N_d}{N_a}$$

The news articles are *minimalist* documents with a timestamp, title, and content, as implemented by the Document class:

```
case class Document[T <: AnyVal] ( //1
  date: T, title: String, content: String)
  (implicit f: T => Double)
```

The date timestamp has a type bounded to the `Long` type, so `T` can be converted to the current time in milliseconds of the JVM (line 1).

Analyzing documents

This section is dedicated to the implementation of the simple text analyzer. Its purpose is to convert a set of documents of the `Document` type; in our case, news articles, into a distribution of relative frequencies of keywords.

The `TextAnalyzer` class implements a data transformation of the `ETransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*. It transforms a sequence of documents into a sequence of relative frequency distribution.

The `TextAnalyzer` class has the following two arguments (line 4):

- A simple text parser, `parser`, that extracts an array of keywords from the title and content of each news articles (line 2).
- A `lexicon` type that lists keywords used in monitoring news related to a company and their synonyms. The synonyms or terms that are semantically similar to each keywords are defined in an immutable map.

The code will be as follows:

```
type TermsRF = Map[String, Double]
type TextParser = String => Array[String] //2
type Lexicon = immutable.Map[String, String] //3
type Corpus[T] = Seq[Document[T]]

class TextAnalyzer[T <: AnyVal] ( //4
    parser: TextParser,
    lexicon: Lexicon)(implicit f: T => Double)
  extends ETransform[Lexicon](lexicon) {

  type U = Corpus[T] //5
  type V = Seq[TermsRF] //6

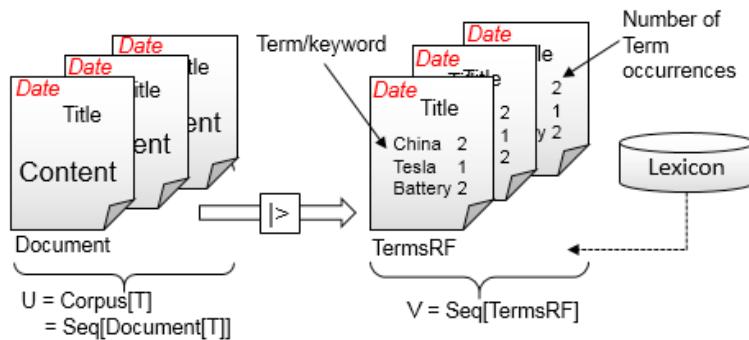
  override def |> : PartialFunction[U, Try[V]] = {
    case docs: U => Try(score(docs))
  }

  def score(corpus: Corpus[T]): Seq[TermsRF] //7
  def quantize(termsRFSeq: Seq[TermsRF]): //8
    Try[(Array[String], XVSeries[Double])]
  def count(term: String): Counter[String] //9
}
```

The `U` type of an input into the data transformation `|>` is the corpus or sequence of news articles (line 5). The `V` type of the output from the data transformation is the sequence of relative frequency distribution of the `TermsRF` type (line 6).

The `score` private method does the heavy lifting for the class (line 7). The `quantize` method creates a homogenous set of observed features (line 8) and the `count` method counts the number of occurrences of terms or keywords across the documents or news articles that share the same publication date (line 9).

The following diagram describes the different components of the text mining process:



An illustration of the components of the text mining procedure

Extracting the frequency of relative terms

Let's dive into the score method:

```
def score(corpus: Corpus[T]): Seq[TermsRF] = {
    val termsCount = corpus.map(doc => //10
        (doc.date, count(doc.content))) //Seq[(T, Counter[String])]

    val termsCountMap = termsCount.groupBy(_._1).map{
        case (t, seq) => (t, seq.aggregate(new Counter[String])
            ((s, cnt) => s ++ cnt._2, _ ++ _)) //11
    }
    val termsCountPerDate = termsCountMap.toSeq
        .sortWith(_._1 < _._1).unzip._2 //12
    val allTermsCounts = termsCountPerDate
        .aggregate(new Counter[String])((s, cnt) =>
            s ++ cnt, _ ++ _)) //13

    termsCountPerDate.map(_ /allTermsCounts).map(_.toMap) //14
}
```

The first step in the execution of the `score` method is the computation of the number of occurrences of keywords of the `lexicon` type on each of the document/news article (line 10). The computation of the number of occurrences is implemented by the `count` method:

```
def count(term: String): Counter[String] =
    parser(term) ./:(new Counter[String])((cnt, w) => //16
        if(lexicon.contains(w)) cnt + lexicon(w) else cnt)
```

The method relies on the term Counter counting class that subclasses `mutable.Map[String, Int]`, as described in the *Counter* section under *Scala programming* in the *Appendix A, Basic Concepts*. It uses a fold to update the count for each of the terms associated with a keyword (line 16). The `count` term for the entire corpus is computed by aggregating the terms count for all the documents (line 11).

The next step consists of aggregating the count of the keywords across the document for each timestamp. A `termsCountMap` map with the date as the key and keywords counter, as values are generated by invoking the `groupBy` higher-order method (line 11). Next, the `score` method extracts a sorted sequence of keywords counts, `termsCountPerDate` (line 12). The total counts for each keyword over the `allTermsCounts` entire corpus (line 13) is used to compute the relative or normalized keywords frequencies (formulas **M9** and **M10**) (line 14).

Generating the features

There is no guarantee that all the news articles associated with a specific publication date are used in the model. The `quantize` method assigns a relative frequency of **0.0** for keywords that are missing from the news articles, as illustrated in the following table:

	Keyword 1	Keyword 2	Keyword 3	...	Keyword N
Date 1	0.42	0.00	0.07		0.23
Date 2	0.00	0.11	0.18		0.04
...					
Date J	0.13	0.29	0.00		0.00

A table on relative frequencies of keywords as per the publishing date

The `quantize` method transforms a sequence of term-relative frequencies into a pair keywords and observations:

```
def quantize(termsRFSeq: Seq[TermsRF]): Try[(Array[String], XVSeries[Double])] = Try {
    val keywords = lexicon.values.toArray.distinct //15
    val relFrequencies =
        termsRFSeq.map( tf => //16
            keywords.map(key =>
                if(tf.contains(key)) tf.get(key).get else 0.0))
    (keywords, relFrequencies.toVector) //17
}
```

The `quantize` method extracts an array of keywords from the lexicon (line 15). The `relFrequencies` vector of features is generated by assigning the relative 0.0 keyword frequency for keywords that are not detected across the news articles published at a specific date (line 16). Finally, the key-value pair (keywords and relative keyword frequency) (line 17).

Sparse relative frequencies vector

Text analysis and natural language processing deals with very large feature sets, with potentially hundreds of thousands of features or keywords. Such computations would be almost intractable if it was not for the fact that the vast majority of keywords are not present in each document. It is a common practice to use sparse vectors and sparse matrices to reduce the memory consumption during training.

Testing

For testing purpose, let's select the news articles that mention Tesla Motors and its ticker symbol TSLA over a period of 2 months.

Retrieving the textual information

Let's start implementing and defining the two components of `TextAnalyzer`: the `parsing` function and the `lexicon` variable:

```
val pathLexicon = "resources/text/lexicon.txt"
val LEXICON = loadLexicon //18

def parse(content: String): Array[String] = {
    val regExpr = "[ '|'.|.|?|!|:|\"|]"
    content.trim.toLowerCase.replace(regExpr, " ") //19
    .split(" ") //20
    .filter(_.length > 2) //21
}
```

The lexicon is loaded from a file (line 18). The `parse` method uses a simple `regExpr` regular expression to replace any punctuation into a space character (line 19), which is used as a word delimiter (line 20). All the words shorter than three characters are discounted (line 21).

Let's describe the workflow to load, parse, and analyze news articles related to the company, Tesla Inc. and its stock, ticker symbol TSLA.

The first step is to load and clean all the articles (corpus) defined in the pathCorpus directory (line 22). This task is performed by the `DocumentsSource` class, as described in the *Data extraction* section under *Scala programming* in the *Appendix A, Basic Concepts*:

```
val pathCorpus = "resources/text/chap5/"      //22
val dateFormat = new SimpleDateFormat("MM.dd.yyyy")
val pfnDocs = DocumentsSource(dateFormat, pathCorpus) |> //23

val textAnalyzer = TextAnalyzer[Long](parse, LEXICON)
val pfnText = textAnalyzer |> //24

for {
    corpus <- pfnDocs(None)    //25
    termsFreq <- pfnText(corpus) //26
    featuresSet <- textAnalyzer.quantize(termsFreq) //27
    expected <- Try(difference(TSLA_QUOTES, diffInt)) //28
    nb <- NaiveBayes[Double](1.0,
                            featuresSet._2.zip(expected)) //29
} yield {
    show(s"Naive Bayes model${nb.toString(quantized._1)}")
    ...
}
```

A document source is fully defined by the path of the data input files and the format used in the timestamp (line 23). The text analyzer and its explicit `pfnText` data transformation is instantiated (line 24). The text processing pipeline is defined by the following steps:

1. The transformation of an input source file into a corpus (a sequence of news articles) using the `pfnDoc` partial function (line 25).
2. The transformation of a corpus into a sequence of a `termsFreq` relative keyword frequency vector using the `pfnText` partial function (line 26).
3. The transformation of a sequence of relative keywords frequency vector into a `featuresSet` using `quantize` (line 27) (refer to the *The differential operator* section under *Time series in Scala* in *Chapter 3, Data Preprocessing*).
4. The creation of the binomial `NaiveBayes` model using the pair (`featuresSet._2` and `expected`) as training data (line 29).

The expected class values (0,1) are extracted from the daily stock price for Tesla Motors, `TSLA_QUOTES`:

```
val TSLA_QUOTES = Array[Double] (250.56, 254.84, ... )
```

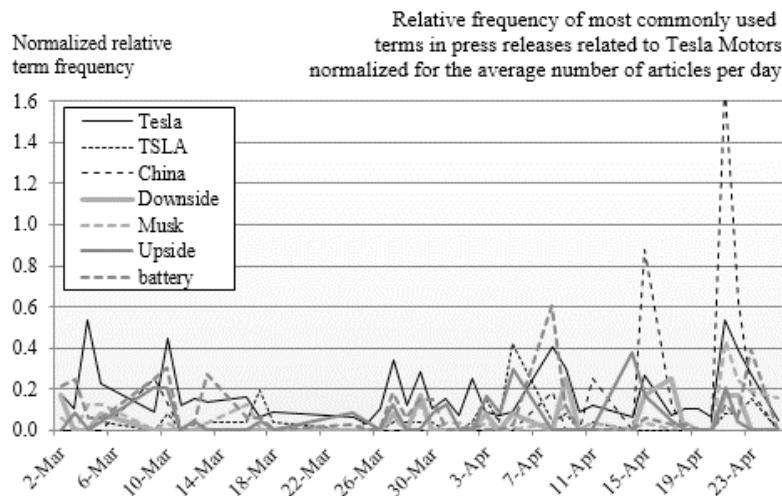
The semantic analysis

This example uses a very primitive semantic map (lexicon) for the sake of illustrating the benefits and inner workings of the multinomial Naïve Bayes algorithm. Commercial applications involving sentiment analysis or topic analysis require a deeper understanding of semantic associations and extraction of topics using advanced generative models, such as the Latent Dirichlet allocation.



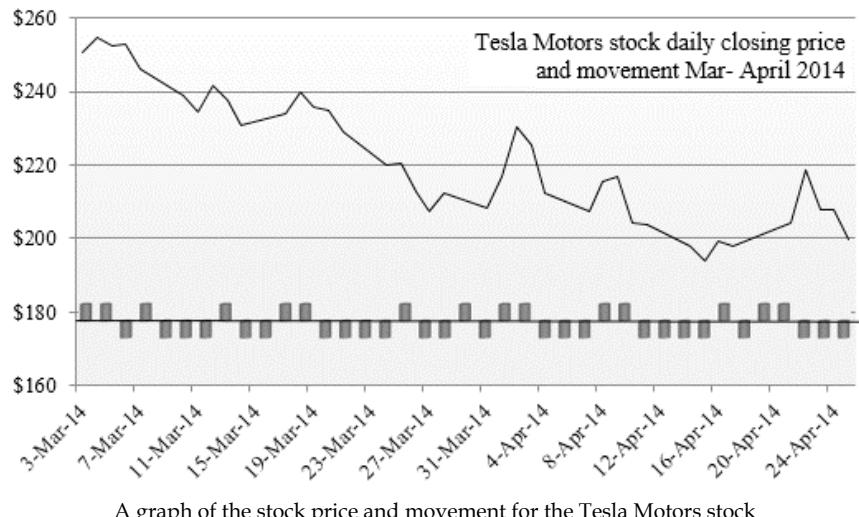
Evaluating the text mining classifier

The following chart describes the frequency of occurrences of some of the keywords related to either Tesla Motors or its stock ticker TSLA:



A graph of the relative frequency of a partial list of stock-related terms

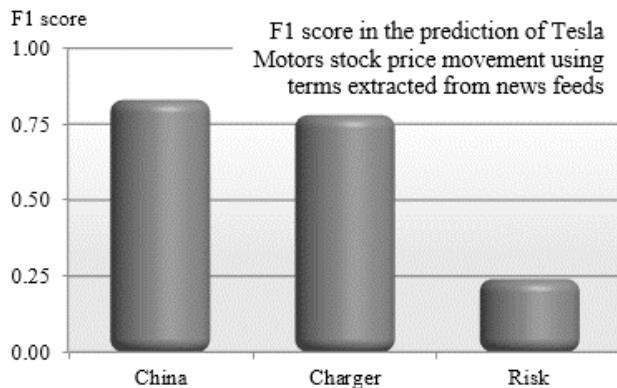
The following chart plots the expected change in the direction of the stock price for the trading day following the press release(s) or news article(s):



A graph of the stock price and movement for the Tesla Motors stock

The preceding chart displays the historical price of the stock TSLA with the direction (UP and DOWN). The classification of 15 percent of the labeled data selected for the validation of the classifier has an F_1 score of 0.71. You need to keep in mind that no preprocessing or clustering was performed to isolate the most relevant features/keywords. We initially selected the keywords according to the frequency of their occurrences in the financial news.

It is fair to assume that some of the keywords have a more significant impact on the direction of the stock price than others. One simple but interesting exercise is to record the value of the F_1 score for a validation for which only the observations that have a high number of occurrences of a specific keyword are used, as shown in the following graph:



A bar chart representing predominant keywords in predicting the TSLA stock movement

The preceding bar chart shows that the terms **China**, representing all the mentions of the activities of Tesla Motors in China, and **Charger**, which covers all the references to the charging stations, have a significant positive impact on the direction of the stock with a probability averaging to 75 percent. The terms under the **Risk** category have a negative impact on the direction of the stock with a probability of 68 percent, or a positive impact of the direction of the stock with a probability of 32 percent. Within the remaining eight categories, 72 percent of them were unusable as a predictor of the direction of the stock price.

This approach can be used for selecting features as an alternative to mutual information for using classifiers that are more elaborate. However, it should not be regarded as the primary methodology for selecting features, but instead as a by-product of the Naïve Bayes formula applied to models with a very small number of relevant features. Techniques such as the principal components analysis, as described in the *Principal components analysis* section under *Dimension reduction* in Chapter 4, *Unsupervised Learning*, are available to reduce the dimension of the problem and make Naïve Bayes a viable classifier.

Pros and cons

The examples selected in this chapter do not do justice to the versatility and accuracy of the Naïve Bayes family of classifiers.

The Naïve Bayes algorithm is a simple and robust generative classifier that relies on prior conditional probabilities to extract a model from a training dataset. The Naïve Bayes model has its benefits, as mentioned here:

- It is easy to implement and parallelize

- It has a very low computational complexity: $O((n+c)*m)$, where m is the number of features, C is the number of classes, and n is the number of observations
- It handles missing data
- It supports incremental updates, insertions, and deletions

However, Naïve Bayes is not a silver bullet. It has the following disadvantages:

- It requires a large training set to achieve reasonable accuracy
- The assumption of the independence of features is not practical in the real world
- It requires dealing with the zero-frequency problem for counters

Summary

There is a reason why the Naïve Bayes model is one of the first supervised learning techniques taught in a machine learning course: it is simple and robust. As a matter of fact, this is the first technique that should come to mind when you are considering creating a model from a labeled dataset, as long as the features are conditionally independent.

This chapter also introduced you to the basics of text mining as an application of Naïve Bayes.

Despite all its benefits, the Naïve Bayes classifier assumes that the features are conditionally independent, a limitation that cannot be always overcome. In the case of the classification of documents or news releases, Naïve Bayes incorrectly assumes that terms are semantically independent: the two entities' age and date of birth are highly correlated. The discriminative classifiers described in the next few chapters address some of Naïve Bayes' limitations [5:14].

This chapter does not treat temporal dependencies, sequence of events, or conditional dependencies between observed and hidden features. These types of dependencies necessitate a different approach to modeling that is the subject of *Chapter 7, Sequential Data Models*.

6

Regression and Regularization

In the first chapter, we briefly introduced the binary logistic regression (the binomial logistic regression for a single variable) as our first test case. The purpose was to illustrate the concept of discriminative classification. There are many more regression models, starting with the ubiquitous ordinary least square linear regression and the logistic regression [6:1].

The purpose of regression is to minimize a loss function, with the **residual sum of squares (RSS)** being one that is commonly used. The problem of overfitting described in the *Overfitting* section under *Assessing a model* in *Chapter 2, Hello World!*, can be addressed by adding a **penalty term** to the loss function. The penalty term is an element of the larger concept of **regularization**.

The first section of this chapter will describe and implement the linear **least-squares regression**. The second section will introduce the concept of regularization with an implementation of the **ridge regression**. Finally, the logistic regression will be revisited in detail from the perspective of a classification model.

Linear regression

Linear regression is by far the most widely used, or at least the most commonly known, regression method. The terminology is usually associated with the concept of fitting a model to data and minimizing the errors between the expected and predicted values by computing the sum of square errors, residual sum of square errors, or least-square errors.

The least squares problems fall into the following two categories:

- Ordinary least squares
- Nonlinear least squares

One-variate linear regression

Let's start with the simplest form of linear regression, which is the single variable regression, in order to introduce the terms and concepts behind linear regression. In its simplest interpretation, the one-variate linear regression consists of fitting a line to a set of data points $\{x, y\}$.

M1: A single variable linear regression for a model f with weights w_j for features x_j and labels (or expected values) y_j is given by the following formula:



$$\tilde{w} = \arg \min_{w,r} \left\{ \sum_{j=0}^{N-1} (y_j - f(x_j|w))^2 \right\} \quad f(x|w) = w_0 + w_1 x$$

Here, w_1 is the slope, w_0 is the intercept, f is the linear function that minimizes the RSS, and (x_j, y_j) is a set of n observations.

The RSS is also known as the **sum of squared errors (SSE)**. The **mean squared error (MSE)** for n observations is defined as the ratio RSS/n .

Terminology

The terminology used in the scientific literature regarding regression is a bit confusing at times. Regression weights are also known as regression coefficients or regression parameters. The weights are referred to as w in formulas and the source code throughout the chapter, although β is also used in reference books.

Implementation

Let's create a `SingleLinearRegression` parameterized class to implement the **M1** formula. The linear regression is a data transformation that uses a model implicitly derived or built from data. Therefore, the simple linear regression implements the `ITransform` trait, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*

The `SingleLinearRegression` class takes the following two arguments:

- An `xt` vector of single variable observations
- A vector of expected values or labels (line 1)

The code will be as follows:

```
class SingleLinearRegression[T <: AnyVal] (
  xt: XSeries[T],
  expected: Vector[T])(implicit f: T => Double)
  extends ITransform[T](xt) with Monitor[Double] { //1
    type V = Double //2

    val model: Option[DblPair] = train //3
    def train: Option[DblPair]
    override def |> : PartialFunction[T, Try[V]]
    def slope: Option[Double] = model.map(_._1)
    def intercept: Option[Double] = model.map(_._2)
}
```

The `Monitor` trait is used to collect the profiling information during training (refer to the `Monitor` section under *Utility classes* in the *Appendix A, Basic Concepts*).

The class has to define the type of the output of the `|>` prediction method, which is a `Double` (line 2).

Model instantiation

The model parameters are computed through training and the model is instantiated regardless of whether the model is actually validated. A commercial application requires the model to be validated using a methodology such as the K-fold validation, as described in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*.

The training generates the model defined as the regression weights (slope and intercept) (line 3). The model is set as `None` if an exception is thrown during training:

```
def train: Option[DblPair] = {
  val regr = new SimpleRegression(true) //4
  regr.addData(zipToSeries(xt, expected).toArray) //5
  Some((regr.getSlope, regr.getIntercept)) //6
}
```

The regression weights or coefficients, that is the `model` tuple, are computed using the `SimpleRegression` class from the `stats.regression` package of the Apache Commons Math library with the `true` argument to trigger the computation of the intercept (line 4). The input time series and the labels (or expected values) are zipped to generate an array of two values (input and expected) (line 5). The `model` is initialized with the slope and intercept computed during the training (line 6).

The `zipToSeries` method of the `XTSerie`s object is described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.

private versus private[this]

 A `private` value or variable can be accessed only by all the instances of a class. A value declared `private [this]` can be manipulated only by the `this` instance. For example, the value `model` can be accessed only by the `this` instance of `SingleLinearRegression`.

Test case

For our first test case, we compute the single variate linear regression of the price of the Copper ETF (ticker symbol: CU) over a period of 6 months (January 1, 2013 to June 30, 2013):

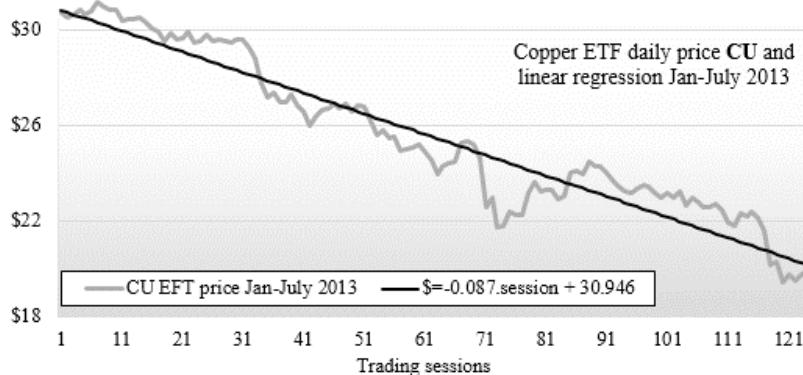
```
val path = "resources/data/chap6/CU.csv"
for {
    price <- DataSource(path, false, true, 1) get adjClose //7
    days <- Try(Vector.tabulate(price.size) (_toDouble)) //8
    linRegr <- SingleLinearRegression[Double] (days, price) //9
} yield {
    if( linRegr.isModel ) {
        val slope = linRegr.slope.get
        val intercept = linRegr.intercept.get
        val error = mse(days, price, slope, intercept)//10
    }
    ...
}
```

The daily closing price of the ETF CU is extracted from a CSV file (line 7) as the expected values using a `DataSource` instance, as described in the *Data extraction and Data sources* section in the *Appendix A, Basic Concepts*. The `x` values `days` are automatically generated as a linear function (line 8). The expected values (`price`) and sessions (`days`) are the inputs to the instantiation of the simple linear regression (line 9).

Once the model is created successfully, the test code computes the `mse` mean squared error of the predicted and expected values (line 10):

```
def mse(
    predicted: DblVector,
    expected: DblVector,
    slope: Double,
    intercept: Double): Double = {
    val predicted = xt.map( slope*_ + intercept)
    XTSeries.mse(predicted, expected) //11
}
```

The mean least squared error is computed using the `mse` method of `XTSeries` (line 11). The original stock price and linear regression equation are plotted on the following chart:



The total least square error is 0.926.

Although the single variable linear regression is convenient, it is limited to a scalar time series. Let's consider the case of multiple variables.

Ordinary least squares regression

The **ordinary least squares regression** computes the parameters w of a linear function $y = f(x_0, x_1, \dots, x_d)$ by minimizing the residual sum of squares. The optimization problem is solved by performing vector and matrix operations (transposition, inversion, and substitution).

M2: The minimization of the loss function is given by the following formula:



$$\tilde{w} = \arg \min_{w,r} \left\{ \sum_{j=0}^{N-1} (y_j - f(x_j|w))^2 \right\} \quad f(x|w) = w_0 + \sum_1^{D-1} w_d x_d$$

Here, w is the weights or parameters of the regression, $(x_i, y_i)_{i:0, n-1}$ is the n observations of a vector x and an expected output value y , and f is the linear multivariate function, $y = f(x_0, x_1, \dots, x_d)$.

There are several methodologies to minimize the residual sum of squares for a linear regression:

- Resolution of the set of n equations with d variables (weights) using the **QR decomposition** of the n by d matrix, representing the time series of n observations of a vector of d dimensions with $n \geq d$ [6:2]
- **Singular value decomposition** on the observations-features matrix, in the case where the dimension d exceeds the number of observations n [6:3]
- **Gradient descent** [6:4]
- **Stochastic gradient descent** [6:5]

An overview of these matrix decompositions and optimization techniques can be found in the *Linear algebra* and *Summary of optimization techniques* sections in the *Appendix A, Basic Concepts*.

The QR decomposition generates the smallest relative error MSE for the most common least squares problem. The technique is used in our implementation of the least squares regression.

Design

The implementation of the least squares regression leverages the Apache Commons Math library implementation of the ordinary least squares regression [6:6].

This chapter describes several types of regression algorithms. It makes sense to define a generic `Regression` trait that defines the key element component of a regression algorithm.

- A model of the `RegressionModel` type (line 1)
- Two methods to access the components of the regression model: `weights` and `rss` (line 2 and 3)

- A `train` polymorphic method that implements the training of this specific regression algorithm (line 4)
- A `training` protected method that wraps `train` into a `Try` monad

The code will be as follows:

```
trait Regression {
  val model: Option[RegressionModel] = training //1

  def weights: Option[DblArray] = model.map(_.weights)//2
  def rss: Option[Double] = model.map(_.rss) //3
  def isModel: Boolean = model != None

  protected def train: RegressionModel //4
  def training: Option[RegressionModel] = Try(train) match {
    case Success(_model) => Some(_model)
    case Failure(e) => e match {
      case err: MatchError => { ... }           case _ => { ... }
    }
  }
}
```

The model is simply defined by its `weights` and its residual sum of squares (line 5):

```
case class RegressionModel( //5
  val weights: DblArray, val rss: Double) extends Model

object RegressionModel {
  def dot[T <: AnyVal](x: Array[T],
    w: DblArray)(implicit f: T => Double): Double =
    x.zip(weights.drop(1))
      .map{ case(x, w) => x*w}.sum + weights.head //6
}
```

The `RegressionModel` companion object implements the computation of the dot inner product of the `weights` regression and an observation, `x` (line 6). The `dot` method is used throughout the chapter.

Let's create a `MultiLinearRegression` class as a data transformation whose model is implicitly derived from the input data (training set), as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*:

```
class MultiLinearRegression[T <: AnyVal] (
  xt: XSeries[T],
  expected: DblVector)(implicit f: T => Double)
  extends ITransform[Array[T]](xt) with Regression
```

```

        with Monitor[Double] { //7
    type V = Double //8

    override def train: Option[RegressionModel] //9
    override def |> : PartialFunction[Array[T], Try[V]] //10
}

```

The `MultiLinearRegression` class takes two arguments: the multidimensional time series of the `xt` observations and the vector of expected values (line 7). The class implements the `ITransform` trait and needs to define the type of the output value for the prediction or regression, `V` as a `Double` (line 8). The constructor for `MultiLinearRegression` creates the `model` through training (line 9). The `ITransform` trait's `|>` method implements the runtime prediction for the multilinear regression (line 10).

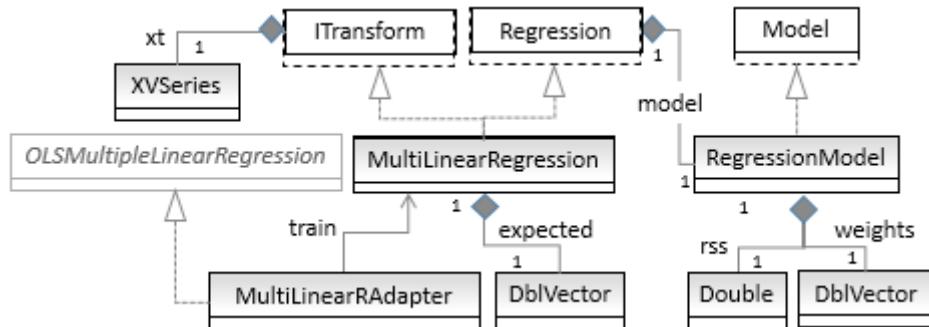
The `Monitor` trait is used to collect the profiling information during training (refer to the `Monitor` section under *Utility classes* in the *Appendix A, Basic Concepts*).

[

The regression model
 The RSS is included in the model because it provides the client code with the important information regarding the accuracy of the underlying technique used to minimize the loss function.

]

The relationship between the different components of the multilinear regression is described in the following UML class diagram:



The UML class diagram for the multilinear (OLS) regression

The UML diagram omits the helper traits and classes such as `Monitor` or the Apache Commons Math components.

Implementation

The training is performed during the instantiation of the `MultiLinearRegression` class (refer to the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*):

```
def train: RegressionModel = {
    val olsMlr = new MultiLinearRAdapter //11
    olsMlr.createModel(expected, data) //12
    RegressionModel(olsMlr.weights, olsMlr.rss) //13
}
```

The functionality of the ordinary least squares regression in the Apache Commons Math library is accessed through an `olsMlr` reference to the `MultiLinearRAdapter` adapter class (line 11).

The `train` method creates the model by invoking the `OLSMultipleLinearRegression` Apache Commons Math class (line 12) and returns the regression model (line 13). The various methods of the class are accessed through the `MultiLinearRAdapter` adapter class:

```
class MultiLinearRAdapter extends OLSMultipleLinearRegression {
    def createModel(y: DblVector, x: Vector[DblArray]): Unit =
        super.newSampleData(y.toArray, x.toArray)

    def weights: DblArray = estimateRegressionParameters
    def rss: Double = calculateResidualSumOfSquares
}
```

The `createModel`, `weights`, and `rss` methods route the request to the corresponding methods in `OLSMultipleLinearRegression`.

The `Try{ }` Scala exception handling monad is used as the return type for the `train` method in order to catch the different types of exceptions thrown by the Apache Commons Math library such as `MathIllegalArgumentException`, `MathRuntimeException`, or `OutOfRangeException`.

Exception handling

Wrapping up invocation of methods in a third party with a `Try { }` Scala exception handler matters for a couple of reasons:

- It makes debugging easier by segregating your code from the third party
- It allows your code to recover from the exception by reexecuting the same function with an alternative third-party library method, whenever possible

The predictive algorithm for the ordinary least squares regression is implemented by the `|>` data transformation. The method predicts the output value, given a model and an input value, x :

```
def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if isModel &&
    x.length == model.get.size-1
    => Try( dot(x, model.get) ) //14
}
```

The predictive value is computed using the `dot` method defined in the `RegressionModel` singleton, which was introduced earlier in this section (line 14).

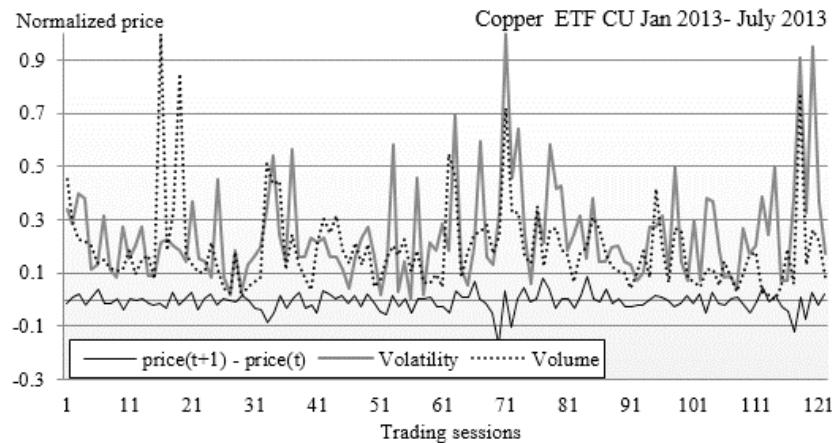
Test case 1 – trending

Trending consists of extracting the long-term movement in a time series. Trend lines are detected using a multivariate least squares regression. The objective of this first test is to evaluate the filtering capability of the ordinary least squares regression.

The regression is performed on the relative price variation of the Copper ETF (ticker symbol: CU). The selected features are `volatility` and `volume`, and the label or target variable is the price change between two consecutive y trading sessions.

The naming convention for the trading data and metrics is described in the *Trading data* section under *Technical analysis* in the *Appendix A, Basic Concepts*.

The volume, volatility, and price variation for CU between January 1, 2013 and June 30, 2013 are plotted on the following chart:



The chart for price variation, volatility, and trading volume for Copper ETF

Let's write the client code to compute the multivariate linear regression, $price\ change = w_0 + volatility.w_1 + volume.w_2$:

```

import YahooFinancials._
val path = "resources/data/chap6/CU.csv" //15
val src = DataSource(path, true, true, 1) //16

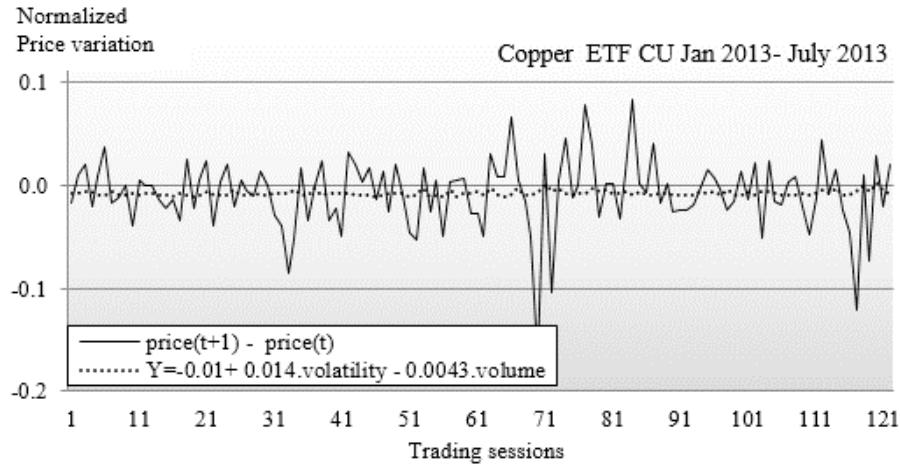
for {
    price <- src.get(adjClose) //17
    volatility <- src.get(volatility) //18
    volume <- src.get(volume) //19
    (features, expected) <- differentialData(volatility,
                                              volume, price, diffDouble) //20
    regression <- MultiLinearRegression[Double] (
        features, expected) //21
} yield {
    if( regression.isModel ) {
        val trend = features.map(dot(_,regression.weights.get))
        display(expected, trend) //22
    }
}
}

```

Let's take a look at the steps required for the execution of the test: it consists of collecting data, extracting the features and expected values, and training the multilinear regression model:

1. Locate the CSV formatted data source file (line 15).
2. Create a data source extractor, `DataSource`, for the trading session closing price, the volatility session, and the volume session for the ETF CU (line 16).
3. Extract the price of the ETF (line 17), its volatility within a trading session (line 18), and the trading volume during the session (line 19) using the `DataSource` transform.
4. Generate the labeled data as a pair of features (relative volatility and relative volume for the ETF) and expected outcome (0, 1) for training the model for which 1 represents the increase in the price and 0 represents the decrease in the price (line 20). The `differentialData` generic method of the `XTSerie`s singleton is described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.
5. The multilinear regression is instantiated using the `features` set and the `expected` change in the daily ETF price (line 21).
6. Display the expected and trending values using JFreeChart (line 22).

The time series of expected values and the data predicted by the regression are plotted on the following chart:



The price variation and the least squares regression for the Copper ETF according to volatility and volume

The least squares regression model is defined by the linear function for the estimation of price variation as follows:

$$\text{price}(t+1)-\text{price}(t) = -0.01 + 0.014 \text{ volatility} - 0.0042 \cdot \text{volume}$$

The estimated price change (the dotted line in the preceding chart) represents the long-term trend from which the noise is filtered out. In other words, the least squares regression operates as a simple low-pass filter as an alternative to some of the filtering techniques such as the discrete Fourier transform or the Kalman filter, as described in *Chapter 3, Data Preprocessing* [6:7].

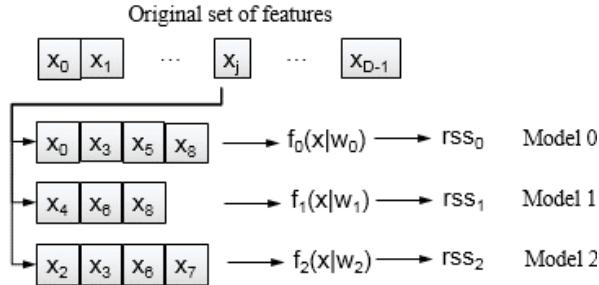
Although trend detection is an interesting application of the least squares regression, the method has limited filtering capabilities for time series [6:8]:

- It is sensitive to outliers
- The first and last few observations need to be discarded
- As a deterministic method, it does not support noise analysis (distribution, frequencies, and so on)

Test case 2 – feature selection

The second test case is related to feature selection. The objective is to discover which subset of initial features generates the most accurate regression model, that is, the model with the smallest residual sum of squares on the training set.

Let's consider an initial set of D features $\{x_i\}$. The objective is to estimate the subset of features $\{x_{id}\}$ that are most relevant to the set of observations using a least squares regression. Each subset of features is associated with a $f_j(x|w_j)$ model:



A diagram for the features set selection

The ordinary least square regression is used to select the model parameters w in the case the feature set is small. Performing the regression of each subset of a large original feature set is not practical.

M3: The features selection can be expressed mathematically as follows:

$$\hat{f} = \arg \min_{f_j} \left\{ \sum_{i=0}^{n-1} (y_i - f_j(x|w))^2 \right\} \quad f_j(x|w) = w_{j0} + \sum_{d=1}^{D_j-1} w_{jd} x_d$$

Here, w_{jk} is the weights of the regression for the function/model $f_j(x, y)$, $i:0, n-1$ is the n observations of a vector x and expected output value y , and f is the linear multivariate function, $y = f(x_0, x_1, \dots, x_d)$.

Let's consider the following four financial time series over the period from January 1, 2009 to December 31, 2013:

- The exchange rate of Chinese Yuan to US Dollar
- The S&P 500 index
- The spot price of gold
- The 10-year Treasury bond price

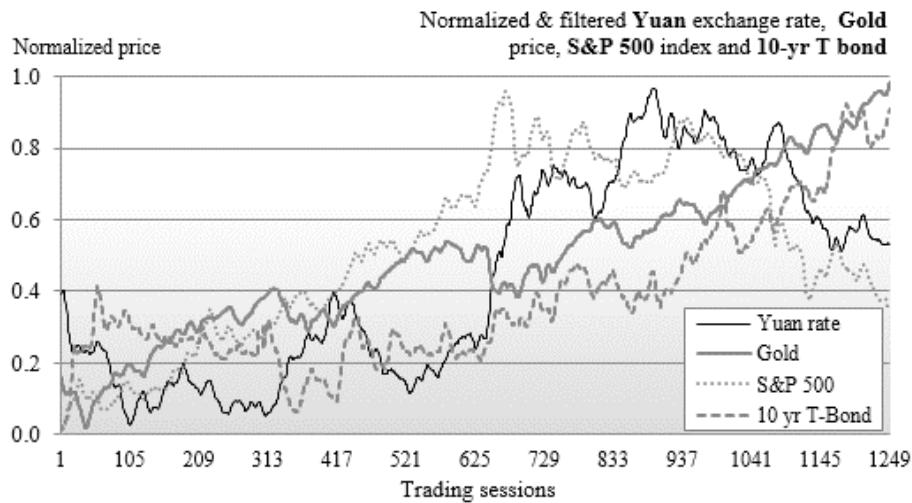
The problem is to estimate which combination of the S&P 500 index, gold price, and 10-year Treasury bond price variables is the most correlated to the exchange rate of the Yuan. For practical reasons, we use the Exchange Trade Funds CYN as the proxy for the Yuan/US dollar exchange rate (similarly, SPY, GLD, and TLT for the S&P 500 index, spot price of gold, and 10-year Treasury bond price, respectively).

Automation of features extraction



The code in this section implements an ad hoc extraction of features with an arbitrary fixed set of models. The process can be easily automated with an optimizer (the gradient descent, genetic algorithm, and so on) using $1/RSS$ as the objective function.

The number of models to evaluate is relatively small, so an ad hoc approach to compute the RSS for each combination is acceptable. Let's take a look at the following graph:



The graph of the Chinese Yuan exchange rate, gold price, 10-year Treasury bond price, and S&P 500 index

The `getRSS` method implements the computation of the RSS value given a set of `xt` observations, `y` expected (smoothed) values, and `featureLabels` labels for features and then returns a textual result:

```
def getRSS(
    xt: Vector[DblArray],
    expected: DblVector,
    featureLabels: Array[String]): String = {
  val regression =
    new MultiLinearRAdapter[Double](xt, expected) //23
  val modelStr = regression.weights.get.view
```

```

    .zipWithIndex.map{ case( w, n) => {
      val weights_str = format(w, emptyString, SHORT)
      if(n == 0 ) s"${featureLabels(n)} = $weights_str"
      else s"$weights_str.${featureLabels(n)}"
    }}.mkString(" + ")
    s"model: $modelStr\nRSS =${regression.get.rss}" //24
  }
}

```

The `getRss` method merely trains the model by instantiating the multilinear regression class (line 23). Once the regression model is trained during the instantiation of the `MultiLinearRegression` class, the coefficients of the regression weights and the RSS values are stringized (line 24). The `getRss` method is invoked for any combination of the `ETF`, `GLD`, `SPY`, and `TLT` variables against the `CNY` label.

Let's take a look at the following test code:

```

val SMOOTHING_PERIOD: Int = 16 //25
val path = "resources/data/chap6/"
val symbols = Array[String] ("CNY", "GLD", "SPY", "TLT") //26
val movAvg = SimpleMovingAverage[Double] (SMOOTHING_PERIOD) //27

for {
  pfnMovAve <- Try(movAvg |>) //28
  smoothed <- filter(pfnMovAve) //29
  models <- createModels(smoothed) //30
  rsses <- Try(getModelsRss(models, smoothed)) //31
  (mses, tss) <- totalSquaresError(models, smoothed.head) //32
} yield {
  s"""$rsses.mkString("\n")\n$mses.mkString("\n")"""
  | \nResidual error= $tss".stripMargin
}

```

The dataset is large (1,260 trading sessions) and noisy enough to warrant filtering using a simple moving average with a period of 16 trading sessions (line 25). The purpose of the test is to evaluate the possible correlation between the four ETFs: `CNY`, `GLD`, `SPY`, and `TLT` (line 26). The execution test instantiates the simple moving average (line 27), as described in the *The simple moving average* section in *Chapter 3, Data Preprocessing*.

The workflow executes the following steps:

1. Instantiate a simple moving average `pfnMovAve` partial function (line 28).
2. Generate smoothed historical prices for the CNY, GLD, SPY, and TLT ETFs using the `filter` function (line 29):

```
Type PFNMOVAVE = PartialFunction[DblVector, Try[DblVector]]
```

```
def filter(pfnMovAve: PFNMOVEAVE): Try[Array[DblVector]] = Try {  
    symbols.map(s => DataSource(s"$path$s.csv", true, true, 1))  
        .map(_.get(adjClose))  
        .map(pfnMovAve(_)).map(_.get)
```

3. Generate the list of features for each model using the `createModels` method (line 30):

```
type Models = List[(Array[String], DblMatrix)]
```

```
def createModels(smoothed: Array[DblVector]): Try[Models] =  
Try {  
    val features = smoothed.drop(1).map(_.toArray) //33  
    List[(Array[String], DblMatrix)]() //34  
        (Array[String]("CNY", "SPY", "GLD", "TLT"), features.transpose),  
        (Array[String]("CNY", "GLD", "TLT"), features.drop(1).transpose),  
        (Array[String]("CNY", "SPY", "GLD"), features.take(2).transpose),  
        (Array[String]("CNY", "SPY", "TLT"), features.zipWithIndex  
            .filter(_.2 != 1).map(_.1).transpose),  
        (Array[String]("CNY", "GLD"), features.slice(1, 2).transpose)  
    )  
}
```

The smoothed values for CNY are used as the expected values. Therefore, they are removed from the features list (line 33). The five models are evaluated by adding or removing elements from the features list (line 34).

4. Next, the workflow computes the residual sum of squares for all the models using `getModelsRss` (line 31). The method invokes `getRss`, which was introduced earlier in this section, for each model (line 35):

```
def getModelsRss(  
    models: Models,  
    y: Array[DblVector]): List[String] =  
models.map{ case (labels, m) =>  
    s"${getRss(m.toVector, y.head, labels)}" } //35
```

5. Finally, the last step of the workflow consists of computing the `mse`s mean squared errors for each model and the total squared errors (line 33):

```
def totalSquaresError(
    models: Models,
    expected: DblVector): Try[(List[String], Double)] = Try {
  val errors = models.map{case (labels,m) =>
    rssSum(m, expected)._1}//36
  val mses = models.zip(errors)
    .map{case(f,e) => s"MSE: ${f._1.mkString(" ")} = $e"}
  (mses, Math.sqrt(errors.sum)/models.size) //37
}
```

The `totalSquaresError` method computes the error for each model by summing the RSS value, `rssSum`, for each model (line 36). The method returns a pair of an array of the mean squared error for each model and the total squared error (line 37).

The RSS does not always provide an accurate visualization of the fitness of the regression model. The fitness of the regression model is commonly assessed using the **r^2 statistics**. The r^2 value is a number that indicates how well data fits into a statistical model.

[



M4: The RSS and r^2 statistics are defined by the following formulae:

$$r^2 = 1 - \frac{RSS}{TSS} \quad TSS = \sum_{i=0}^{n-1} (y_i - \bar{f}(x|w))^2 \quad \bar{f} = \sum_f f_j$$

]

The implementation of the computation of the r^2 statistics is simple. For each model f_j , the `rssSum` method computes the tuple (rss and least squares errors), as defined in the **M4** formula:

```
def rssSum(xt: DblMatrix, expected: DblVector): DblPair = {
  val regression =
    MultiLinearRegression[Double](xt, expected) //38
  val pfnRegr = regression |> //39
  val results = sse(expected.toArray, xt.map(pfnRegr(_).get))
  (regression.rss, results) //40
}
```

Regression and Regularization

The `rssSum` method instantiates the `MultiLinearRegression` class (line 38), retrieves the RSS value, and then validates the `pfnRegr` regressive model (line 39) against the expected values (line 40). The output of the test is presented in the following screenshot:

```
CNY = f(SPY, GLD, TLT)
0.16089780923264457 + -0.21189413823325406.x1 + 0.26299169969099795.x2 + 0.3556562652009136.x3
RSS: 3.681353535940423

CNY = f(SPY, TLT)
0.2039015515038045 + -0.03796334296279046.x1 + 0.26219728078589966.x2
RSS: 3.8589613138639227

CNY = f(GLD, TLT)
0.19290917330324198 + 0.015507174710195552.x1 + 0.2204800144601237.x2
RSS: 3.8849688539396317

CNY = f(SPY, GLD)
0.22242202699107552 + 0.17842973203100937.x1 + -0.12099602178260839.x2
RSS: 4.6681933948464645

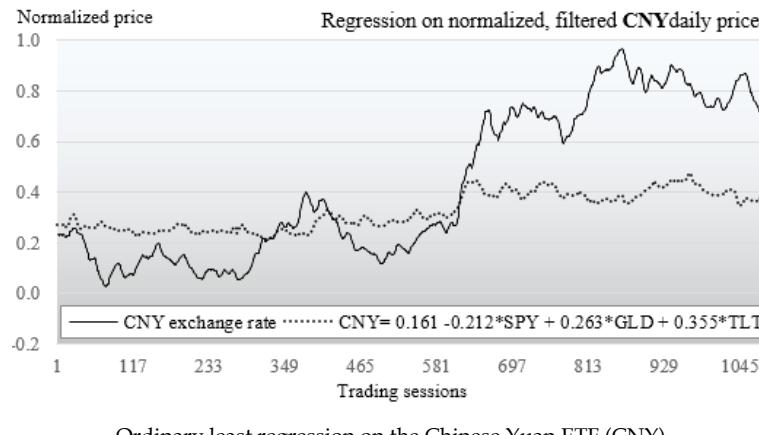
CNY = f(SPY)
0.20238901847764892 + 0.1251591898720694.x1
RSS: 4.7291908591838405

CNY = f(TLT)
0.19724352413716711 + 0.22501420632545652.x1
RSS: 3.8876824376753705

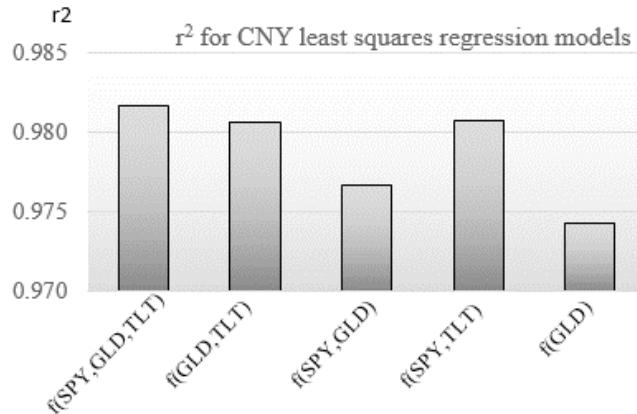
CNY = f(GLD)
0.198195293931846 + 0.16413676262473123.x1
RSS: 5.149661975952835
```

The output results clearly show that the three variable regression, $CNY = f(SPY, GLD, TLT)$, is the most accurate or fittest model for the CNY time series, followed by $CNY = f(SPY, TLT)$. Therefore, the feature selection process generates the features set, $\{SPY, GLD, TLT\}$.

Let's plot the model against the raw data:



The regression model smoothed the original CNY time series. It weeded out all but the most significant price variation. The graph plotting the r^2 value for each of the model confirms that the three features model $CNY=f(SPY, GLD, TLT)$ is the most accurate:



The general linear regression

The concept of a linear regression is not restricted to polynomial fitting models such as $y = w_0 + w_1x + w_2x^2 + \dots + w_nx^n$. Regression models can be also defined as a linear combination of basis functions such as \square_j : $y = w_0 + w_1\square_1(x) + w_2\square_2(x) + \dots + w_n\square_n(x)$ [6:9].

Regularization

The ordinary least squares method for finding the regression parameters is a specific case of the maximum likelihood. Therefore, regression models are subject to the same challenge in terms of overfitting as any other discriminative models. You are already aware of the fact that regularization is used to reduce model complexity and avoid overfitting, as stated in the *Overfitting* section in *Chapter 2, Hello World!*

L_n roughness penalty

Regularization consists of adding a $J(w)$ penalty function to the loss function (or RSS in the case of a regressive classifier) in order to prevent the model parameters (also known as weights) from reaching high values. A model that fits a training set very well tends to have many features variables with relatively large weights. This process is known as **shrinkage**. Practically, shrinkage involves adding a function with model parameters as an argument to the loss function (**M5**):

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}_d} \left\{ \sum_{i=0}^{n-1} (y_i - f(\mathbf{x}_i | \mathbf{w}))^2 + \lambda J(\mathbf{w}) \right\}$$

The penalty function is completely independent of the training set $\{x, y\}$. The penalty term is usually expressed as a power to the function of the norm of the model parameters (or weights) w_d . For a model of D dimension, the generic **L_p-norm** is defined as follows (**M6**):

$$J_{pq}(\mathbf{w}) = \|\mathbf{w}\|_p^q = \left[\sum_{d=1}^{D-1} |w_d|^p \right]^{q/p}$$

Notation

Regularization applies to parameters or weights associated with observations. In order to be consistent with our notation, w_0 being the intercept value, the regularization applies to the parameters $w_1 \dots w_d$.

The two most commonly used penalty functions for regularization are L₁ and L₂.

Regularization in machine learning

The regularization technique is not specific to the linear or logistic regression. Any algorithm that minimizes the residual sum of squares, such as a support vector machine or feed-forward neural network, can be regularized by adding a roughness penalty function to the RSS.

The L₁ regularization applied to the linear regression is known as the **lasso regularization**. The **ridge regression** is a linear regression that uses the L₂ regularization penalty.

You may wonder which regularization makes sense for a given training set. In a nutshell, L_2 and L_1 regularizations differ in terms of computation efficiency, estimation, and features selection [6:10] [6:11]:

- **Model estimation:** L_1 generates a sparser estimation of the regression parameters than L_2 . For a large nonsparse dataset, L_2 has a smaller estimation error than L_1 .
- **Feature selection:** L_1 is more effective in reducing the regression weights for features with a higher value than L_2 . Therefore, L_1 is a reliable features selection tool.
- **Overfitting:** Both L_1 and L_2 reduce the impact of overfitting. However, L_1 has a significant advantage in overcoming overfitting (or excessive complexity of a model); for the same reason, L_1 is more appropriate for selecting features.
- **Computation:** L_2 is conducive to a more efficient computation model. The summation of the loss function and L_2 penalty, w^2 , is a continuous and differentiable function for which the first and second derivatives can be computed (**convex minimization**). The L_1 term is the summation of $|w_i|$ and therefore not differentiable.


Terminology

The ridge regression is sometimes called the **penalized least squares regression**. The L_2 regularization is also known as the **weight decay**.

Let's implement the ridge regression, and then evaluate the impact of the L_2 -norm penalty factor.

Ridge regression

The ridge regression is a multivariate linear regression with an L_2 -norm penalty term (M7):

$$\tilde{w}_{\text{ridge}} = \arg \min_w \left\{ \sum_{j=0}^{N-1} (y - w_0 - w^T x)^2 + \lambda |w|_2^2 \right\} \quad |w|_2^2 = \sum_1^{D-1} w_d^2$$

The computation of the ridge regression parameters requires the resolution of a system of linear equations that are similar to the linear regression.

M8: The matrix representation of ridge regression closed form for an input dataset X , a regularization factor λ , and expected values vector y is defined as follows (I is the identity matrix):



$$(X^T X - \lambda \cdot I) \hat{w}_{Ridge} = X^T y$$

M9: The matrices equation is resolved using the QR decomposition as follows:

$$(X^T X - \lambda \cdot I) = Q \begin{bmatrix} R \\ 0 \end{bmatrix} \quad w_{Ridge} = Q^T y \begin{bmatrix} R \\ 0 \end{bmatrix}^{-1}$$

Design

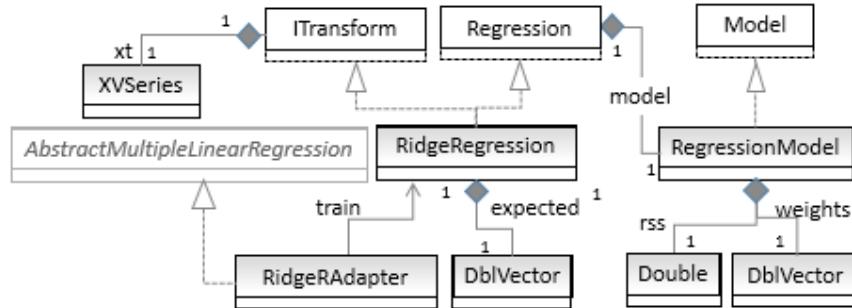
The implementation of the ridge regression adds the L_2 regularization term to the multiple linear regression computation of the Apache Commons Math Library. The methods of `RidgeRegression` have the same signature as their ordinary least squares counterparts except for the lambda L_2 penalty term (line 1):

```
class RidgeRegression[T <: AnyVal] ( //1
    xt: XSeries[T],
    expected: DblVector,
    lambda: Double) (implicit f: T => Double)
  extends ITransform[Array[T]](xt) with Regression
    with Monitor[Double] { //2

  type V = Double //3
  override def train: Option[RegressionModel] //4
  override def |> : PartialFunction[Array[T], Try[V]]
}
```

The `RidgeRegression` class is implemented as an `ITransform` data transformation whose model is implicitly derived from the input data (training set), as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 2). The `V` type of the output of the `|>` predictive function is a `Double` (line 3). The model is created through training during the instantiation of the class (line 4).

The relationship between the different components of the ridge regression is described in the following UML class diagram:



The UML class diagram for the ridge regression

The UML diagram omits the helper traits or classes such as Monitor or the Apache Commons Math components.

Implementation

Let's take a look at the training method, `train`:

```

def train: RegressionModel = {
  val mlr = new RidgeRAdapter(lambda, xt.head.size) //5
  mlr.createModel(data, expected) //6
  RegressionModel(mlr.getWeights, mlr.getRss) //7
}
  
```

It is rather simple; it initialized and executed the regression algorithm implemented in the `RidgeRAdapter` class (line 5), which acts as an adapter to the internal Apache Commons Math library `AbstractMultipleLinearRegression` class in the `org.apache.commons.math3.stat.regression` package (line 6). The method returns a fully initialized regression model that is similar to the ordinary least squared regression (line 7).

Let's take a look at the `RidgeRAdapter` adapter class:

```

class RidgeRAdapter(
  lambda: Double,
  dim: Int) extends AbstractMultipleLinearRegression {
  var qr: QRDecomposition = _ //8
  
```

```
def createModel(x: DblMatrix, y: DblVector): Unit = { //9
    this.newXSampleData(x) //10
    super.newYSampleData(y.toArray)
}
def getWeights: DblArray = calculateBeta.toArray //11
def getRss: Double = rss
}
```

The constructor for the RidgeRAdapter class takes two parameters: the lambda L_2 penalty parameter and the number of features, dim , in an observation. The QR decomposition in the AbstractMultipleLinearRegression base class does not process the penalty term (line 8). Therefore, the creation of the model has to be redefined in the `createModel` method (line 9), which requires to override the `newXSampleData` method (line 10):

```
override protected def newXSampleData(x: DblMatrix): Unit = {
    super.newXSampleData(x) //12
    val r: RealMatrix = getX
    Range(0, dim).foreach(i =>
        r.setEntry(i, i, r.getEntry(i,i) + lambda) ) //13
    qr = new QRDecomposition(r) //14
}
```

The `newXSampleData` method overrides the default observations-features r matrix (line 12) by adding the `lambda` coefficient to its diagonal elements (line 13), and then updating the QR decomposition components (line 14).

The weights for the ridge regression models is computed by implementing the **M6** formula (line 11) in the `calculateBeta` overridden method (line 15):

```
override protected def calculateBeta: RealVector =
    qr.getSolver().solve(getY()) //15
```

The predictive algorithm for the ordinary least squares regression is implemented by the `|>` data transformation. The method predicts the output value, given a model and an input x value (line 16):

```
def |> : PartialFunction[Array[T], Try[V]] = {
    case x: Array[T] if(isModel &&
        x.length == model.get.size-1) =>
        Try( dot(x, model.get) ) //16
}
```

Test case

The objective of the test case is to identify the impact of the L_2 penalization on the RSS value and then compare the predicted values with the original values.

Let's consider the first test case related to the regression on the daily price variation of the Copper ETF (symbol: CU) using the stock daily volatility and volume as features. The implementation of the extraction of observations is identical to that for the least squares regression, as described in the previous section:

```

val LAMBDA: Double = 0.5
val src = DataSource(path, true, true, 1) //17

for {
    price <- src.get(adjClose) //18
    volatility <- src.get(volatility) //19
    volume <- src.get(volume) //20
    (features, expected) <- differentialData(volatility,
                                                volume, price, diffDouble) //21
    regression <- RidgeRegression[Double](features,
                                         expected, LAMBDA) //22
} yield {
    if( regression.isModel ) {
        val trend = features
            .map( dot(_, regression.weights.get) ) //23
    }
}

val y1 = predict(0.2, expected, volatility, volume) //24
val y2 = predict(5.0, expected, volatility, volume)
val output = (2 until 10 by 2).map( n =>
    predict(n*0.1, expected, volatility, volume) )
}
}

```

Let's take a look at the steps required for the execution of the test. The steps consist of collecting data, extracting the features and expected values, and training the ridge regression model:

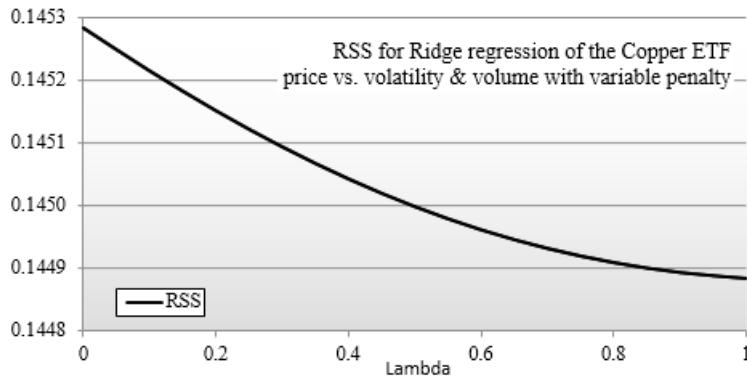
1. Create a data source extractor for the price trading session closing, the volatility session, and the volume session for the ETF CU using the DataSource transformation (line 17).
2. Extract the closing price of the ETF (line 18), its volatility within a trading session (line 19), and the volume trading during the same session (line 20).

3. Generate the labeled data as a pair of features (the relative volatility and relative volume for the ETF) and the expected outcome $\{0, 1\}$ for training the model, where 1 represents the increase in the price and 0 represents the decrease in the price (line 21). The `differentialData` generic method of the `XTSeries` singleton is described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.
4. Instantiate the ridge regression using the features set and the expected change in the daily stock price (line 22).
5. Compute the trend values using the `dot` function of the `RegressionModel` singleton (line 23).
6. Execute a using the ridge regression is implemented by the `predict` method (line 24).

The code is as follows:

```
def predict(  
    lambda: Double,  
    deltaPrice: DblVector,  
    volatility: DblVector,  
    volume: DblVector): DblVector = {  
  
    val observations = zipToSeries(volatility, volume)//25  
    val regression = new RidgeRegression[Double](observations,  
        deltaPrice, lambda)  
    val fnRegr = regression |> //26  
    observations.map( fnRegr(_).get) //27  
}
```

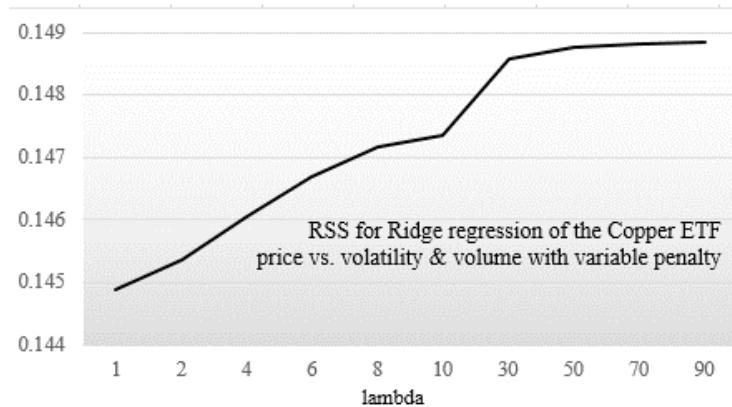
The observations are extracted from the `volatility` and `volume` time series (line 25). The predictive method for the `fnRegr` ridge regression (line 26) is applied to each observation (line 27). The RSS value, `rss`, is plotted for different values of λ , as shown in the following chart:



The graph of RSS versus lambda for the Copper ETF

The residual sum of squares decreases as λ increases. The curve seems to be reaching for a minimum around $\lambda = 1$. The case of $\lambda = 0$ corresponds to the least squares regression.

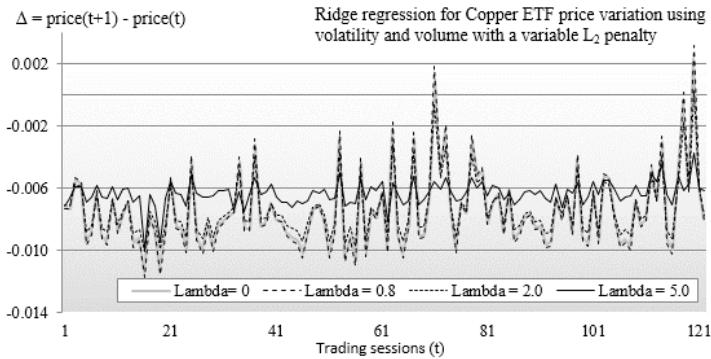
Next, let's plot the RSS value for λ varying between 1 and 100:



The graph of RSS versus a large value Lambda for the Copper ETF

This time around, the value of RSS increases with λ before reaching a maximum for $\lambda > 60$. This behavior is consistent with other findings [6:12]. As λ increases, the overfitting gets more expensive, and therefore, the RSS value increases.

Let's plot the predicted price variation of the Copper ETF using the ridge regression with different values of lambda (λ):



The graph of ridge regression on the Copper ETF price variation with a variable, lambda

The original price variation of the Copper ETF, $\Delta = \text{price}(t + 1) - \text{price}(t)$, is plotted as $\lambda = 0$. Let's analyze the behavior of the predictive model for different values of λ :

- The predicted values for $\lambda = 0.8$ is very similar to the original data.
- The predicted values for $\lambda = 2$ follow the pattern of the original data with a reduction of large variations (peaks and troughs).
- The predicted values for $\lambda = 5$ corresponds to a smoothed dataset. The pattern of the original data is preserved but the magnitude of the price variation is significantly reduced.

The logistic regression, which was briefly introduced in the *Let's kick the tires* section in *Chapter 1, Getting Started*, is the next logical regression model to be discussed. The logistic regression relies on optimization methods. Let's go through a short refresher course in optimization before diving into the logistic regression.

Numerical optimization

This section briefly introduces the different optimization algorithms that can be applied to minimize the loss function, with or without a penalty term. These algorithms are described in more detail in the *Summary of optimization techniques* section in the *Appendix A, Basic Concepts*.

First, let's define the **least squares problem**. The minimization of the loss function consists of nullifying the first order derivatives, which in turn generates a system of D equations (also known as the gradient equations), D being the number of regression weights (parameters). The weights are iteratively computed by solving the system of equations using a numerical optimization algorithm.

M10: The definition of the least squares-based loss function for residual r_i , weights w , a model f , input data x_i , and expected values y_i is as follows:

$$\mathcal{L}(w) = \sum_{i=0}^{n-1} r_i(w)^2 \quad r_i(w) = y_i - f(x_i|w)$$

M10: The generation of gradient equations with a Jacobian J matrix (refer to the *Mathematics* section in the *Appendix A, Basic Concepts*) after minimization of the loss function L is defined as follows:



$$\sum_{i=0}^{n-1} r_i(w) J_{id}(w) = 0 \quad J_{id}(w) = -\frac{\partial r_i(w)}{\partial w_d}$$

M11: The iterative approximation using the Taylor series on the model f for k iterations on the computation of weights w is defined as follows:

$$f(x_i|w) - f(x_i|w^{(k)}) \sim \sum_{jd=0}^{D-1} \frac{\partial f(x_i|w^{(k)})}{\partial w_d} (w - w^{(k)})$$

The logistic regression is a nonlinear function. Therefore, it requires the nonlinear minimization of the sum of least squares. The optimization algorithms for the nonlinear least squares problems can be divided into two categories:

- **Newton** (or 2nd order techniques): These algorithms calculate the second order derivatives (the Hessian matrix) to compute the regression weights that nullify the gradient. The two most common algorithms in this category are the Gauss-Newton and Levenberg-Marquardt methods (refer to the *Nonlinear least squares minimization* section in the *Appendix A, Basic Concepts*). Both algorithms are included in the Apache Commons Math library.
- **Quasi-Newton** (or 1st order techniques): First order algorithms do not compute but estimate the second order derivatives of the least squares residuals from the Jacobian matrix. These methods can minimize any real-valued functions, not just the least squares summation. This category of algorithms includes the Davidon-Fletcher-Powell and the Broyden-Fletcher-Goldfarb-Shannon methods (refer to the *Quasi-Newton algorithms* section in the *Appendix A, Basic Concepts*).

Logistic regression

Despite its name, the *logistic regression* is a classifier. As a matter of fact, the logistic regression is one of the most commonly used discriminative learning techniques because of its simplicity and its ability to leverage a large variety of optimization algorithms. The technique is used to quantify the relationship between an observed target (or expected) variable y and a set of variables x that it depends on. Once the model is created (trained), it is available to classify real-time data.

A logistic regression can be either binomial (two classes) or multinomial (three or more classes). In a binomial classification, the observed outcome is defined as {true, false}, {0, 1}, or {-1, +1}.

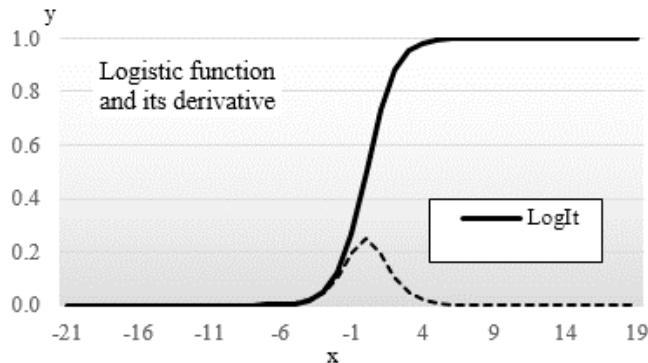
Logistic function

The conditional probability in a linear regression model is a linear function of its weights [6:13]. The logistic regression model addresses the nonlinear regression problem by defining the logarithm of the conditional probability as a linear function of its parameters.

First, let's introduce the logistic function and its derivative, which are defined as follows (M12):

$$f(x) = \frac{1}{(1 - e^{-x})} \quad \frac{df}{dx} = f(x)(1 - f(x))$$

The logistic function and its derivative are illustrated in the following graph:



The graph of the logistic function and its derivative

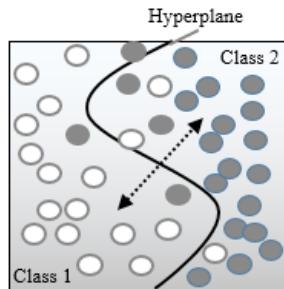
The remainder of this section is dedicated to the application of the multivariate logistic regression to the binomial classification.

Binomial classification

The logistic regression is popular for several reasons; some are as follows:

- It is available with most statistical software packages and open source libraries
- Its S-shape describes the combined effect of several explanatory variables
- Its range of values [0, 1] is intuitive from a probabilistic perspective

Let's consider the classification problem using two classes. As discussed in the *Validation* section in *Chapter 2, Hello World!*, even the best classifier produces false positives and false negatives. The training procedure for a binomial classification is illustrated in the following diagram:



An illustration of the binomial classification for a two-dimension dataset

The purpose of the training is to compute the **hyperplane** that separates the observations into two categories or classes. Mathematically speaking, a hyperplane in an n-dimensional space (number of features) is a subspace of $n - 1$ dimensions, as described in the *Manifolds* section in *Chapter 4, Unsupervised Learning*. The separating hyperplane of a three-dimension space is a curved surface. The separating hyperplane of a two-dimension problem (plane) is a line. In our preceding example, the hyperplane segregates/separates a training set into two very distinct classes (or groups), **Class 1** and **Class 2**, in an attempt to reduce the overlap (false positive and false negative).

The equation of the hyperplane is defined as the logistic function of the dot product of the regression parameters (or weights) and features.

The logistic function accentuates the difference between the two groups of training observations, separated by the hyperplane. It *pushes the observations away* from the separating hyperplane toward either classes.

In the case of two classes, $c1$ and $c2$ with their respective probabilities, $p(C=c1 | X=x_i | w) = p(x_i | w)$ and $p(C=c2 | X=x_i | w) = 1 - p(x_i | w)$, where w is the model parameters set or weights in the case of the logistic regression.

The logistic regression

M13: The log likelihood for N observations x_i given regression weights w is defined as:

$$\mathcal{L}(w) = \sum_{i=0}^{N-1} \log p(x_i | w)$$

M14: Conditional probabilities $p(x | w)$ with regression weights w , using the logistic function for N observations with d features $\{x_{ij}\}_{j=0:d-1}$ is defined as:



$$x_i = \{1, x_{i0}, \dots, x_{id-1}\} \quad p(x_i | w) = \frac{1}{1 + e^{-w^T x_i}} \quad w^T x_i = \sum_{j=0}^d w_j x_{ij}$$

M15: The sum of square errors, sse , for the binomial logistic regression with weights w , input values x_i , and expected binary outcome y is as follows:

$$sse(w) = \frac{1}{2} \sum_{i=0}^{N-1} \left\{ y_i - \log(1 + e^{-w^T x_i}) \right\}^2 \quad y \in \{0,1\}$$

M16: The computation of the weights w of the logistic regression by maximizing the log likelihood, given the input data x_i and expected outcome (labels) y_i is defined as:

$$\frac{\partial \mathcal{L}(\tilde{w})}{\partial w_j} = \sum_{i=0}^{N-1} x_{ij} \left(y_i - \frac{1}{1 + e^{-\tilde{w}^T x_i}} \right) = 0$$

Let's implement the logistic regression without regularization using the Apache Commons Math library. The library contains several least squares optimizers that allow you to specify the optimizer minimizing algorithm for the loss function in the logistic regression class, `LogisticRegression`.

The constructor for the `LogisticRegression` class follows a very familiar pattern: it defines an `ITransform` data transformation, whose model is implicitly derived from the input data (training set), as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 2). The output of the `|>` predictor is a class ID, and therefore, the `V` type of the output is an `Int` (line 3):

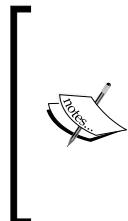
```
class LogisticRegression[T <: AnyVal] (
    xt: XVSeries[T],
    expected: Vector[Int],
    optimizer: LogisticRegressionOptimizer) //1
    (implicit f: T => Double)
extends ITransform[Array[T]](xt) with Regression
with Monitor[Double] { //2

    type V = Int //3
    override def train: RegressionModel //4
    def |> : PartialFunction[Array[T], Try[V]]
}
```

The parameters of the logistic regression class are the multivariate `xt` time series (features), the target or expected classes, `expected`, and the `optimizer` used to minimize the loss function or residual sum of squares (line 1). In the case of the binomial logistic regression, `expected` are assigned the value of `1` for one class and `0` for the other.

The `Monitor` trait is used to collect the profiling information during training (refer to the `Monitor` section under *Utility classes* in the *Appendix A, Basic Concepts*).

The purpose of the training is to determine the regression weights that minimize the loss function, as defined in the **M14** formula as well as the residual sum of squares (line 4).



Target values

There is no specific rule to assign the two values to the observed data for the binomial logistic regression: $\{-1, +1\}$, $\{0, 1\}$, or $\{\text{false}, \text{true}\}$. The values pair $\{0, 1\}$ is convenient because it allows the developer to reuse the code for multinomial logistic regression using normalized class values.

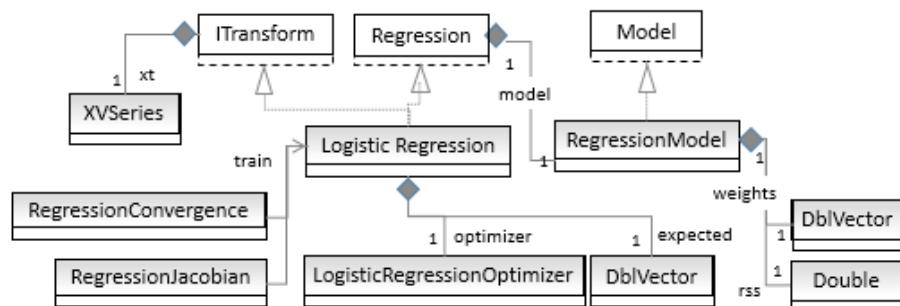
For convenience, the definition and configuration of the optimizer are encapsulated in the `LogisticRegressionOptimizer` class.

Design

The implementation of the logistic regression uses the following components:

- A `RegressionModel` model of the `Model` type that is initialized through training during the instantiation of the classifier. We reuse the `RegressionModel` type, which was introduced in the *Linear regression* section.
- The logistic regression class, `LogisticRegression`, that implements an `ITransform` for the prediction of future observations
- An adapter class named `RegressionJacobian` for the computation of the Jacobian
- An adapter class named `RegressionConvergence` to manage the convergence criteria and exit condition of the minimization of the sum of square errors

The key software components of the logistic regression are described in the following UML class diagram:



The UML class diagram for the logistic regression

The UML diagram omits the helper traits or classes such as `Monitor` or the Apache Commons Math components.

The training workflow

Our implementation of the training of the logistic regression model leverages either the Gauss-Newton or the Levenberg-Marquardt nonlinear least squares optimizers, (refer to the *Nonlinear least squares minimization* section in the *Appendix A, Basic Concepts*) packaged with the Apache Commons Math library.

The training of the logistic regression is performed by the `train` method.

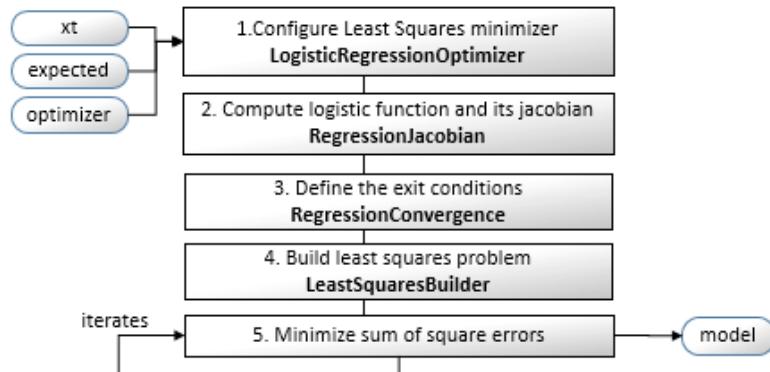
Handling exceptions from the Apache Commons Math library

The training of the logistic regression using the Apache Commons Math library requires handling the `ConvergenceException`, `DimensionMismatchException`, `TooManyEvaluationsException`, `TooManyIterationsException`, and `MathRuntimeException` exceptions. Debugging is greatly facilitated by understanding the context of these exceptions in the Apache library source code.

The implementation of the training method, `train`, relies on the following five steps:

1. Select and configure the least squares optimizer.
2. Define the logistic function and its Jacobian.
3. Specify the convergence and exit criteria.
4. Compute the residuals using the least squares problem builder.
5. Run the optimizer.

The workflow and the Apache Commons Math classes used in the training of the logistic regression are visualized by the following flow diagram:



The workflow for training the logistic regression using the Apache Commons Math library

The first four steps are required by the Apache Commons Math library to initialize the configuration of the logistic regression prior to the minimization of the loss function. Let's start with the configuration of the least squares optimizer:

```
def train: RegressionModel = {
  val weights0 = Array.fill(data.head.length +1)(INITIAL_WEIGHT)
  val lrJacobian = new RegressionJacobian(data, weights0) //5
```

```
val exitCheck = new RegressionConvergence(optimizer) //6

def createBuilder: LeastSquaresProblem //7
val optimum = optimizer.optimize(createBuilder) //8
RegressionModel(optimum.getPoint.toArray, optimum.getRMS)
}
```

The `train` method implements the last four steps of the computation of the regression model:

- Computation of logistic values and the Jacobian matrix (line 5).
- Initialization of the convergence criteria (line 6).
- Definition of the least square problem (line 7).
- Minimization of the sum of square errors (line 8). It is performed by the optimizer as part of the constructor of `LogisticRegression`.

Step 1 – configuring the optimizer

In this step, you have to specify the algorithm to minimize the residual of the sum of the squares. The `LogisticRegressionOptimizer` class is responsible for configuring the optimizer. The class has the following two purposes:

- Encapsulating the configuration parameters for the optimizer
- Invoking the `LeastSquaresOptimizer` interface defined in the Apache Commons Math library

The code will be as follows:

```
class LogisticRegressionOptimizer(
    maxIters: Int,
    maxEvals: Int,
    eps: Double,
    lsOptimizer: LeastSquaresOptimizer) { //9
  def optimize(lsProblem: LeastSquaresProblem): Optimum =
    lsOptimizer.optimize(lsProblem)
}
```

The configuration of the logistic regression optimizer is defined as the maximum number of iterations, `maxIters`, the maximum number of evaluations, `maxEval`, for the logistic function and its derivatives, the `eps` convergence criteria of the residual sum of squares, and the instance of the least squares problem (line 9).

Step 2 – computing the Jacobian matrix

The next step consists of computing the value of the logistic function and its first order partial derivatives with respect to the weights by overriding the `value` method of the `fitting.leastsquares.MultivariateJacobianFunction` Apache Commons Math interface:

```

class RegressionJacobian[T <: AnyVal] ( //10
    xv: XvSeries[T],
    weights0: DblArray)(implicit f: T => Double)
extends MultivariateJacobianFunction {

  type GradientJacobian = Pair[RealVector, RealMatrix]
  override def value(w: RealVector): GradientJacobian = { //11
    val gradient = xv.map( g => { //12
      val f = logistic(dot(g, w))//13
      (f, f*(1.0-f)) //14
    })
    xv.zipWithIndex //15
    ./: (Array.ofDim[Double](xv.size, weights0.size)) {
      case (j, (x,i)) => {
        val df = gradient(i)._2
        Range(0, x.size).foreach(n => j(i)(n+1) = x(n)*df)
        j(i)(0) = 1.0; j //16
      }
    }
    (new ArrayRealVector(gradient.map(_._1).toArray),
     new Array2DRowRealMatrix(jacobian)) //17
  }
}

```

The constructor for the `RegressionJacobian` class requires the following two arguments (line 10):

- The `xv` time series of observations
- The `weights0` initial regression weights

The `value` method uses the `RealVector`, `RealMatrix`, `ArrayRealVector`, and `Array2DRowRealMatrix` primitive types defined in the `org.apache.commons.math3.linear` Apache Commons Math package (line 11). It takes the `w` regression weight as an argument, computes the gradient (line 12) of the logistic function (line 13) for each data point, and returns the value and its derivative (line 14).

The Jacobian matrix is populated with the values of the derivative of the logistic function (line 15). The first element of each column of the Jacobian matrix is set to 1.0 to take into account the intercept (line 16). Finally, the value function returns the pair of gradient values and the Jacobian matrix using types that comply with the signature of the value method in the Apache Commons Math library (line 17).

Step 3 – managing the convergence of the optimizer

The third step defines the exit condition for the optimizer. It is accomplished by overriding the converged method of the parameterized ConvergenceChecker interface in the org.apache.commons.math3.optim Java package:

```
val exitCheck = new ConvergenceChecker[PointVectorValuePair] {
    override def converged(
        iters: Int,
        prev: PointVectorValuePair,
        current: PointVectorValuePair): Boolean =
        sse(prev.getValue, current.getValue) < optimizer.eps
        && iters >= optimizer.maxIterations //18
}
```

This implementation computes the convergence or exit condition as follows:

- The sse sum of square errors between weights of two consecutive iterations is smaller than the eps convergence criteria
- The iters value exceeds the maximum number of iterations, maxIterations, allowed (line 18)

Step 4 – defining the least squares problem

The Apache Commons Math least squares optimizer package requires all the input to the nonlinear least squares minimizer to be defined as an instance of the LeastSquareProblem generated by the factory LeastSquareBuilder class:

```
def createBuilder: LeastSquaresProblem =
  (new LeastSquaresBuilder).model(lrJacobian)      //19
  .weight(MatrixUtils.createRealDiagonalMatrix(
    Array.fill(xt.size)(1.0))) //20
  .target(expected.toArray) //21
  .checkerPair(exitCheck) //22
  .maxEvaluations(optimizer.maxEvals) //23
  .start(weights0) //24
  .maxIterations(optimizer.maxIterations) //25
  .build
```

The diagonal elements of the weights matrix are initialized to 1.0 (line 20). Besides the initialization of the model with the `1rJacobian` Jacobian matrix (line 19), the sequence of method invocations sets the maximum number of evaluations (line 23), maximum number of iterations (line 25), and the exit condition (line 22).

The regression weights are initialized with the `weights0` weights as arguments of the constructor for `LogisticRegression` (line 24). Finally, the expected or target values are initialized (line 21).

Step 5 – minimizing the sum of square errors

The training is executed with a simple call to the `lsp` least squares minimizer:

```
val optimum = optimizer.optimize(lsp)
(optimum.getPoint.toArray, optimum.getRMS)
```

The regression coefficients (or weights) and the **residuals mean square (RMS)** are returned by invoking the `getPoint` method on the `Optimum` class of the Apache Commons Math library.

Test

Let's test our implementation of the binomial multivariate logistic regression using the example of the price variation versus volatility and volume of the Copper ETF, which is used in the previous two sections. The only difference is that we need to define the target values as 0 if the ETF price decreases between two consecutive trading sessions, and 1 otherwise:

```
import YahooFinancials._
val maxIters = 250
val maxEvals = 4500
val eps = 1e-7

val src = DataSource(path, true, true, 1) //26
val optimizer = new LevenbergMarquardtOptimizer //27

for {
    price <- src.get(adjClose) //28
    volatility <- src.get(volatility) //29
    volume <- src.get(volume) //30
    (features, expected) <- differentialData(volatility,
    volume, price, diffInt) //31
    lsOpt <- LogisticRegressionOptimizer(maxIters, maxEvals,
    eps, optimizer) //32
```

```
regr <- LogisticRegression[Double](features, expected, lsOpt)
pfnRegr <- Try(regr |>) //33
}
yield {
  show(s"${LogisticRegressionEval.toString(regr)}")
  val predicted = features.map(pfnRegr(_))
  val delta = predicted.view.zip(expected.view)
    .map{case(p, e) => if(p.get == e) 1 else 0}.sum
  show(s"Accuracy: ${delta.toDouble/expected.size}")
}
```

Let's take a look at the steps required for the execution of the test that consists of collecting data, initializing the parameters for the minimization of the sum of square errors, training a logistic regression model, and running the prediction:

1. Create a `src` data source to extract the market and trading data (line 26).
2. Select the `LevenbergMarquardtOptimizer` Levenberg-Marquardt algorithm as `optimizer` (line 27).
3. Load the daily closing price (line 28), volatility within a trading session (line 29), and the volume daily trading (line 30) for the ETF CU.
4. Generate the labeled data as a pair of features (the relative volatility and relative volume for the ETF) and the `expected` outcome {0, 1} for training the model for which 1 represents the increase in the price and 0 represents the decrease in the price (line 31). The `differentialData` generic method of the `XTSeries` singleton is described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.
5. Instantiate the `lsOpt` optimizer to minimize the sum of square errors during training (line 32).
6. Train the `regr` model and return the `pfnRegr` predictor partial function (line 33).

There are many alternative optimizers available to minimize the sum of square errors optimizers (refer to the *Nonlinear least squares minimization* section in the *Appendix A, Basic Concepts*).

Levenberg-Marquardt parameters

The driver code uses the `LevenbergMarquardtOptimizer` with the default tuning parameters' configuration to keep the implementation simple. However, the algorithm has a few important parameters, such as the relative tolerance for cost and matrix inversion, that are worth tuning for commercial applications (refer to the *Levenberg-Marquardt* section under *Nonlinear least squares minimization* in the *Appendix A, Basic Concepts*).

The execution of the test produces the following results:

- **The residual mean square** is 0.497
- **Weights** are -0.124 for intercept, 0.453 for ETF volatility, and -0.121 for ETF volume

The last step is the classification of the real-time data.

Classification

As mentioned earlier and despite its name, the binomial logistic regression is actually a binary classifier. The classification method is implemented as an implicit data transformation |>:

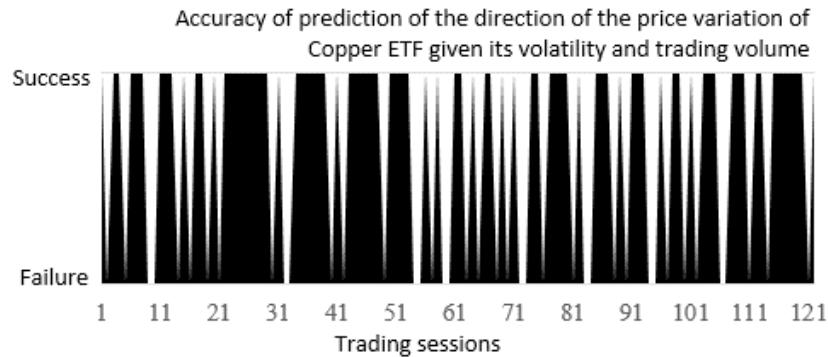
```
val HYPERPLANE = - Math.log(1.0/INITIAL_WEIGHT -1)
def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if(isModel &&
    model.size-1 == x.length && isModel) =>
    Try(if(dot(x, model) > HYPERPLANE) 1 else 0 ) //34
}
```

The dot (or inner) product of the observation x with the weights $model$ is evaluated against the hyperplane. The predicted class is 1 if the produce exceeds `HYPERPLANE`, and 0 otherwise (line 34).

Class identification

 The class that the new data x belongs to is determined by the $dot(x, weights) > 0.5$ test, where dot is the product of the features and the regression weights ($w_0 + w_1 \cdot volatility + w_2 \cdot volume$). You may find different classification schemes in the scientific literature.

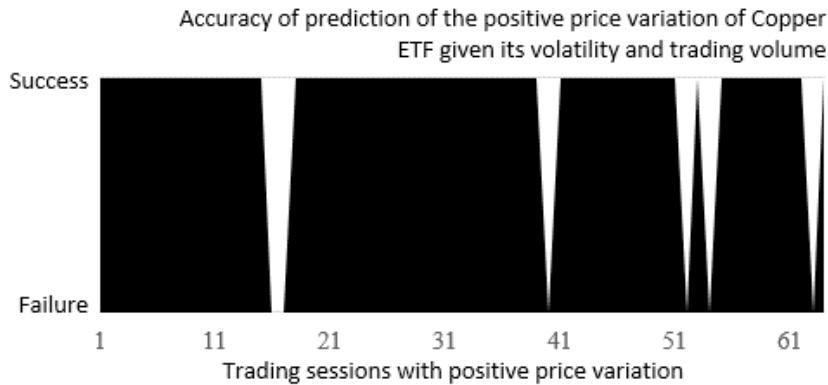
The direction of the price variation of the Copper ETF, $CU\ price(t+1) - price(t)$, is compared to the direction predicted by the logistic regression. The result is plotted with the **success** value if the positive or negative direction is correctly predicted; otherwise, it is plotted with the **failure** value:



The prediction of the direction of the variation of price of the Copper ETF using the logistic regression

The logistic regression was able to classify 78 out of 121 trading sessions (65 percent accuracy).

Now, let's use the logistic regression to predict the positive price variation for the Copper ETF, given its volatility and trading volume. This trading or investment strategy is known as being *long on the market*. This particular use case ignores the trading sessions for which the price was either flat or declined:



The prediction of the direction of the variation of price of the Copper ETF using the logistic regression

The logistic regression was able to correctly predict the positive price variation for 58 out of 64 trading sessions (90.6 percent accuracy). What is the difference between the first and second test cases?

In the first case, the $w_0 + w_1.volatility + w_2.volume$ separating hyperplane equation is used to segregate the features generating either the positive or negative price variation. Therefore, the overall accuracy of the classification is negatively impacted by the overlap of the features from the two classes.

In the second case, the classifier has to consider only the *observations located on the positive side* of the hyperplane equation, without taking into account the false negatives.



Impact of rounding errors

Under some circumstances, the generation of the rounding errors during the computation of the Jacobian matrix has an impact on the accuracy of the $w_0 + w_1.volatility + w_2.volume$ separating hyperplane equation. It reduces the accuracy of the prediction of both the positive and negative price variation.

The accuracy of the binary classifier can be further improved by considering the positive variation of the price using a margin error EPS as $price(t+1) - price(t) > EPS$.



The validation methodology

The validation set is generated by randomly selecting observations from the original labeled dataset. A formal validation requires you to use a K-fold validation methodology to compute the recall, precision, and F1 measure for the logistic regression model.

Summary

This concludes the description and implementation of the linear and logistic regression and the concept of regularization to reduce overfitting. Your first analytical projects using machine learning will (or did) likely involve a regression model of some type. Regression models, along with the Naïve Bayes classification, are the most understood techniques for those without a deep knowledge of statistics or machine learning.

After the completion of this chapter, you will hopefully have a grasp on the following topics:

- The concept of linear and nonlinear least squares-based optimization
- The implementation of ordinary least square regression as well as logistic regression
- The impact of regularization with an implementation of the ridge regression

The logistic regression is also the foundation of the conditional random fields, as described in the *Conditional random fields* section in *Chapter 7, Sequential Data Models*, and multilayer perceptrons, which was introduced in the *The multilayer perceptron* section in *Chapter 9, Artificial Neural Networks*.

Contrary to the Naïve Bayes models (refer to *Chapter 5, Naïve Bayes Classifiers*), the least squares or logistic regression does not impose the condition that the features have to be independent. However, the regression models do not take into account the sequential nature of a time series such as asset pricing. The next chapter, which is dedicated to models for sequential data, introduces two classifiers that take into account the time dependency in a time series: the hidden Markov model and conditional random fields.

7

Sequential Data Models

The universe of Markov models is vast and encompasses computational concepts such as the Markov decision process, discrete Markov, Markov chain Monte Carlo for Bayesian networks, and hidden Markov models.

Markov processes, and more specifically, the **hidden Markov model (HMM)**, are commonly used in speech recognition, language translation, text classification, document tagging, and data compression and decoding.

The first section of this chapter introduces and describes the hidden Markov model with the full implementation of the three canonical forms of the hidden Markov model using Scala. This section covers the different dynamic programming techniques used in the evaluation, decoding, and training of the hidden Markov model. The design of the classifier follows the same pattern as the logistic and linear regression, as described in *Chapter 6, Regression and Regularization*.

The second and last section of this chapter is dedicated to a discriminative (labels conditional to observations) alternative to the hidden Markov model: conditional random fields. The open source CRF Java library authored by Sunita Sarawagi from the Indian Institute of Technology, Bombay, is used to create a predictive model using conditional random fields [7:1].

Markov decision processes

This first section also describes the basic concepts you need to know in order to understand, develop, and apply the hidden Markov model. The foundation of the Markovian universe is the concept known as the **Markov property**.

The Markov property

The Markov property is a characteristic of a stochastic process where the conditional probability distribution of a future state depends on the current state and not on its past states. In this case, the transition between the states occurs at a discrete time, and the Markov property is known as the **discrete Markov chain**.

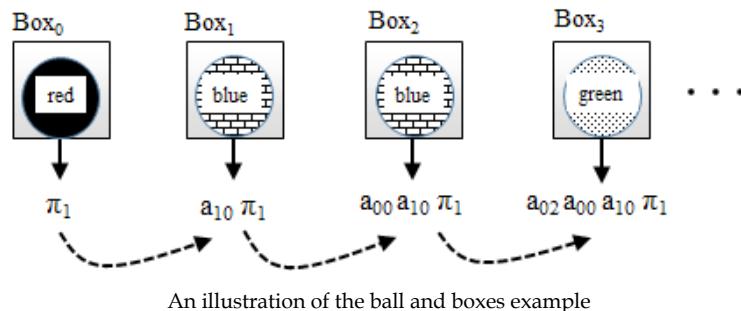
The first order discrete Markov chain

The following example is taken from *Introduction to Machine Learning*, E. Alpaydin [7:2].

Let's consider the following use case. N balls of different colors are hidden in N boxes (one each). The balls can have only three colors (Blue, Red, and Green). The experimenter draws the balls one by one. The state of the discovery process is defined by the color of the latest ball drawn from one of the boxes: $S_0 = \text{Blue}$, $S_1 = \text{Red}$, and $S_2 = \text{Green}$.

Let $\{\pi_0, \pi_1, \pi_2\}$ be the initial probabilities for having an initial set of color in each of the boxes.

Let q_t denote the color of the ball drawn at the time t . The probability of drawing a ball of color S_k at the time k after drawing a ball of the color S_j at the time j is defined as $p(q_t = S_k | q_{t-1} = S_j) = a_{jk}$. The probability of drawing a red ball in the first attempt is $p(q_{t0} = S_1) = \pi_1$. The probability of drawing a blue ball in the second attempt is $p(q_{t1} = S_1 | q_{t0} = S_1) = \pi_1 a_{10}$. The process is repeated to create a sequence of the state $\{S_t\} = \{\text{Red}, \text{Blue}, \text{Blue}, \text{Green}, \dots\}$ with the following probability: $p(q_0 = S_1) \cdot p(q_1 = S_0 | q_0 = S_1) \cdot p(q_2 = S_1 | q_1 = S_0) \cdot p(q_3 = S_2 | q_2 = S_1) \dots = \pi_1 \cdot a_{10} \cdot a_{00} \cdot a_{02} \dots$. The sequence of states/colors can be represented as follows:



Let's estimate the probabilities p using historical data (learning phase):

- The estimation of the probability of drawing a red ball (S_1) in the first attempt is π_{1r} , which is computed as the number of sequences starting with S_1 (red) / total number of balls.
- The estimation of the probability of retrieving a blue ball in the second attempt is a_{10r} , the number of sequences for which a blue ball is drawn after a red ball / total number of sequences, and so on.

Nth-order Markov

 The Markov property is popular mainly because of its simplicity. As you will discover while studying the hidden Markov model, having a state solely dependent on the previous state allows us to apply efficient dynamic programming techniques. However, some problems require dependencies between more than two states. These models are known as Markov random fields.

Although the discrete Markov process can be applied to trial and error types of applications, its applicability is limited to solving problems for which the observations do not depend on hidden states. Hidden Markov models are a commonly applied technique to meet such a challenge.

The hidden Markov model

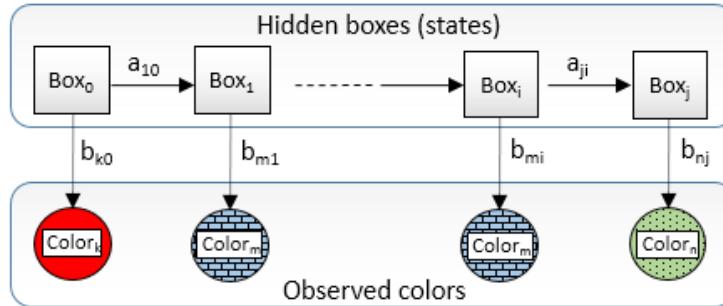
The hidden Markov model has numerous applications related to speech recognition, face identification (biometrics), and pattern recognition in pictures and videos [7:3].

A hidden Markov model consists of a Markov process (also known as a Markov chain) for observations with a discrete time. The main difference with the Markov processes is that the states are not observable. A new observation is emitted with a probability known as the emission probability each time the state of the system or model changes.

There are two sources of randomness, which are as follows:

- Transition between states
- Emission of an observation when a state is given

Let's reuse the boxes and balls example. If the boxes are hidden states (nonobservable), then the user draws the balls whose color is not visible. The emission probability is the probability $b_{ik} = p(o_t = \text{color}_k | q_t = S_i)$ to retrieve a ball of the color k from a hidden box I , as shown in the following diagram:



The hidden Markov model for the balls and boxes example

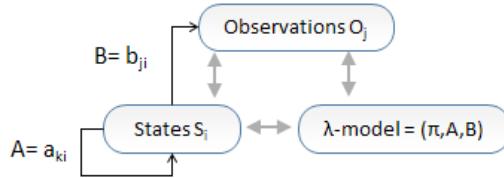
In this example, we do not assume that all the boxes contain balls of different colors. We cannot make any assumptions on the order as defined by the transition a_{ij} . The HMM does not assume that the number of colors (observations) is identical to the number of boxes (states).

 **Time invariance**
Contrary to the Kalman filter, for example, the hidden Markov model requires that the transition elements, a_{ij} , are independent of time. This property is known as stationary or homogeneous restriction.

Keep in mind that the observations, in this case the color of the balls, are the only tangible data available to the experimenter. From this example, we can conclude that a formal HMM has three components:

- A set of observations
- A sequence of hidden states
- A model that maximizes the joint probability of the observations and hidden states, known as the Lambda model

A Lambda model, λ , is composed of initial probabilities π , the probabilities of state transitions as defined by the matrix A , and the probabilities of states emitting one or more observations, as shown in the following diagram:



The visualization of the HMM key components

The preceding diagram illustrates that, given a sequence of observations, the HMM tackles the following three problems known as canonical forms:

- **CF1 (evaluation):** This evaluates the probability of a given sequence of observations O_p given a model $\lambda = (\pi, A, B)$
- **CF2 (training):** This identifies (or learns) a model $\lambda = (\pi, A, B)$, given a set of observations O
- **CF3 (decoding):** This estimates the state sequence Q with the highest probability to generate a given set of observations O and a model λ

The solution to these three problems uses dynamic programming techniques. However, we need to clarify the notations prior to diving into the mathematical foundation of the hidden Markov model.

Notations

One of the challenges of describing the hidden Markov model is the mathematical notation that sometimes differs from author to author. From now on, we will use the following notation:

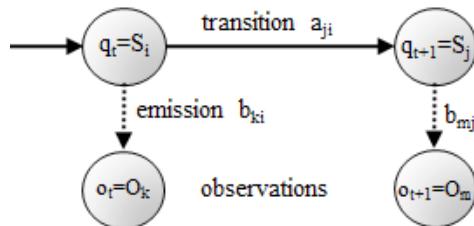
	Description	Formulation
N	The number of hidden states	
S	A finite set of N hidden states	$S = \{S_0, S_1, \dots, S_{N-1}\}$
M	The number of observation symbols	
q_t	The state at time or step t	
Q	A time sequence of states	$Q = \{q_0, q_1, \dots, q_{n-1}\} = Q_{0:n-1}$
T	The number of observations	
o_t	The observation at time t	
O	A finite sequence of T observations	$O = \{o_0, o_1, \dots, o_{T-1}\} = O_{0:T-1}$
A	The state transition probability matrix	$a_{ji} = p(q_{t+1}=S_i \mid q_t=S_j)$
B	The emission probability matrix	$b_{jk} = p(o_t=O_k \mid q_t=S_j)$
π	The initial state probability vector	$\pi_i = p(q_0=S_j)$
λ	The hidden Markov model	$\lambda = (\pi, A, B)$



Variance in the notation

Some authors use the symbol z to represent the hidden states instead of q and x to represent the observations O .

For convenience, let's simplify the notation of the sequence of observations and states using the condensed form: $p(O_{0:T}, q_t | \lambda) = p(O_0, O_1, \dots, O_T, q_t | \lambda)$. It is quite common to visualize a hidden Markov model with a lattice of states and observations, which is similar to our description of the boxes and balls examples, as shown here:



The formal HMM-directed graph

The state S_i is observed as O_k at time t , before being transitioned to the state S_j observed as O_m at the time $t+1$. The first step in the creation of our HMM is the definition of the class that implements the lambda model $\lambda = (\pi, A, B)$ [7:4].

The lambda model

The three canonical forms of the hidden Markov model rely heavily on manipulation and operations on matrices and vectors. For convenience, let's define an `HMMConfig` class that contains the dimensions used in the HMM:

```

class HMMConfig(val numObs: Int, val numStates: Int,
               val numSymbols: Int, val maxIters: Int, val eps: Double)
  extends Config
  
```

The input parameters for the class are as follows:

- `numObs`: This is the number of observations
- `numStates`: This is the number of hidden states
- `numSymbols`: This is the number of observation symbols or features
- `maxIters`: This is the maximum number of iterations required for the HMM training
- `eps`: This is the convergence criteria for the HMM training

 **Consistency with a mathematical notation**

The implementation uses `numObs` (with respect to `numStates` and `numSymbols`) to represent programmatically the number of observations T (with respect to the N hidden states and M features). As a general rule, the implementation reuses the mathematical symbols as much as possible.

The `HMMConfig` companion object defines the operations on ranges of index of matrix rows and columns. The `foreach` (line 1), `foldLeft` (`/:`) (line 2), and `maxBy` (line 3) methods are regularly used in each of the three canonical forms:

```
object HMMConfig {
    def foreach(i: Int, f: Int => Unit): Unit =
        Range(0, i).foreach(f) //1
    def /:(i: Int, f: (Double, Int) => Double, zero: Double) =
        Range(0, i)./:(zero)(f) //2
    def maxBy(i: Int, f: Int => Double): Int =
        Range(0,i).maxBy(f) //3
    ...
}
```

 **The λ notation**

The λ model in the HMM should not be confused with the regularization factor discussed in the L_n roughness penalty section in *Chapter 6, Regression and Regularization*.

As mentioned earlier, the lambda model is defined as a tuple of the transition probability matrix A , emission probability matrix B , and the initial probability π . It is easily implemented as an `HMMModel` class using the `DMatrix` class, as defined in the *Utility classes* section in the *Appendix A, Basic Concepts*. The simplest constructor for the `HMMModel` class is invoked in the case where the state-transition probability matrix, the emission probability matrix, and the initial states are known, as shown in the following code:

```
class HMMModel( val A: DMatrix, val B: DMatrix, var pi: DblArray,
    val numObs: Int) { //4
    val numStates = A.nRows
    val numSymbols = B.nCols

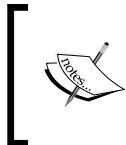
    def setAlpha(obsSeqNum: Vector[Int]): DMatrix
    def getAlphaVal(a: Double, i: Int, obsId: Int): Double
    def getBetaVal(b: Double, i: Int, obsId: Int): Double
    def update(gamma: Gamma, diGamma: DiGamma,
        obsSeq: Vector[Int])
    def normalize: Unit
}
```

The constructor of the `HMMModel` class has the following four arguments (line 4):

- `A`: This is the state transition probabilities matrix
- `B`: This is the omission probabilities matrix
- `pi`: This is the initial probability for the states
- `numObs`: This is the number of observations

The number of states and symbols are extracted from the dimension of the `A` and `B` matrices.

The `HMMModel` class has several methods that will be described in detail whenever they are required for the execution of the model. The probabilities for the `pi` initial states are unknown, and therefore, they are initialized with a random generator of values [0, 1].



Normalization

Input states and observation data may have to be normalized and converted to probabilities before we initialize the `A` and `B` matrices.



The other two components of the HMM are the sequence of observations and the sequence of hidden states.

Design

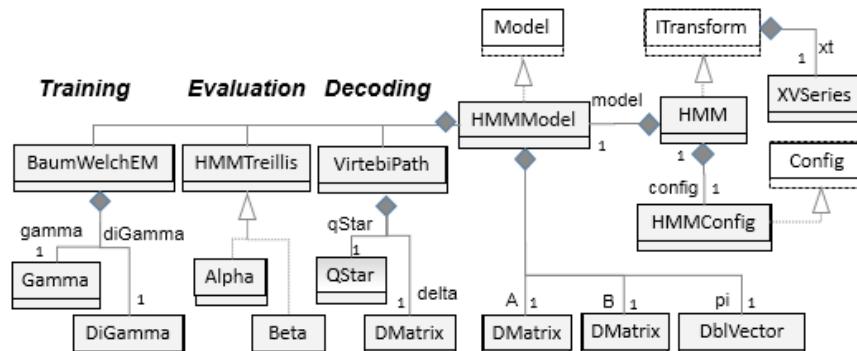
The canonical forms of the HMM are implemented through dynamic programming techniques. These techniques rely on variables that define the state of the execution of the HMM for any of the canonical forms:

- Alpha (the forward pass): The probability of observing the first $t < T$ observations for a specific state at S_i for the observation t is $\alpha_t(i) = p(O_{0:t}, q_t = S_i | \lambda)$
- Beta (the backward pass): The probability of observing the remainder of the sequence qt for a specific state is $\beta_t(i) = p(O_{t+1:T} | q_t = S_i, \lambda)$
- Gamma: The probability of being in a specific state given a sequence of observations and a model is $\gamma_t(i) = p(q_t = S_i | O_{0:T}, \lambda)$
- Delta: This is the sequence that has the highest probability path for the first i observations defined for a specific test $\delta_t(i)$
- Qstar: This is the optimum sequence q^* of states $Q_{0:T}$

- **DiGamma:** The probability of being in a specific state at t and another defined state at $t + 1$ given the sequence of observations and the model is $\gamma_t(i,j) = p(q_t=S_i, q_{t+1}=S_j | O_{0:T-1}, \lambda)$

Each of the parameters is described mathematically and programmatically in the section related to each specific canonical form. The `Gamma` and `DiGamma` classes are used and described in the evaluation canonical form. The `DiGamma` singleton is described as part of the Viterbi algorithm to extract the sequence of states with the highest probability given a λ model and a set of observations.

The list of dynamic programming-related algorithms used in any of the three canonical forms is visualized through the class hierarchy of our implementation of the HMM:



Scala classes' hierarchy for HMM (the UML class diagram)

The UML diagram omits the utility traits and classes such as `Monitor` or the Apache Commons Math components.

The λ model, the HMM state, and the sequence of observations are all the elements needed to implement the three canonical cases. Each class is described as needed in the description of the three canonical forms of HMM. It is time to dive into the implementation details of each of the canonical forms, starting with the evaluation.

The execution of any of the three canonical forms relies on dynamic programming techniques (refer to the *Overview of dynamic programming* section in the *Appendix A, Basic Concepts*) [7:5]. The simplest of the dynamic programming techniques is a single traversal of the observations/state chain.

Evaluation – CF-1

The objective is to compute the probability (or likelihood) of the observed sequence O_t given a λ model. A dynamic programming technique is used to break down the probability of the sequence of observations into two probabilities (**M1**):

$$p(O_{0:T-1}|\lambda) \propto p(O_{0:t}|\lambda) \cdot p(O_{t+1:T-1}|\lambda)$$

The likelihood is computed by marginalizing over all the hidden states $\{S_i\}$ [7:6] (**M2**):

$$p(O_{0:T-1}|\lambda) = \sum_{i=0}^{N-1} p(O_{0:T-1}, q_t = S_i | \lambda)$$

If we use the notation introduced in the previous chapter for alpha and beta variables, the probability for the observed sequence O_t given a λ model can be expressed as follows (**M3**):

$$p(O_{0:T-1}|\lambda) = \sum_i \alpha_t(i) \cdot \beta_t(i)$$

The product of the α and β probabilities can potentially underflow. Therefore, it is recommended that you use the log of the probabilities instead of the probabilities.

Alpha – the forward pass

The computation of the probability of observing a specific sequence given a sequence of hidden states and a λ model relies on a two-pass algorithm. The alpha algorithm consists of the following steps:

1. Compute the initial alpha value [**M4**]. The value is then normalized by the sum of alpha values across all the hidden states [**M5**].
2. Compute the alpha value iteratively for the time 0 to time t , and then normalize it by the sum of alpha values for all states [**M6**].
3. The final step is to compute of the log of the probability of observing the sequence [**M7**].



Performance consideration

A direct computation of the probability of observing a specific sequence requires $2TN_2$ multiplications. The iterative alpha and beta classes reduce the number of multiplications to N_2T .

For those with some inclination toward mathematics, the computation of the alpha matrix is defined in the following information box.



Alpha (the forward pass)

M4: Initialization is defined as:

$$\alpha_0(i) = \pi_i \cdot b_i(O_0)$$

M5: Normalization of initial values $N - 1$ is defined as:

$$\hat{\alpha}_0(i) = \alpha_0(i) / \sum_{j=0}^{N-1} \alpha_0(j)$$

M6: Normalized summation is defined as:



$$\alpha_t(i) = \sum_{j=0}^{N-1} \alpha_{t-1}(j) a_j b_i(O_t) \quad c_t = 1 / \sum_{i=0}^{N-1} \alpha_t(i) \quad \hat{\alpha}_t(i) = \alpha_t(i) \cdot c_t$$

M7: The probability of observing a sequence given a lambda model and states is defined as:

$$\log p(O|\lambda) = - \sum_{j=0}^{T-1} \log \left(\frac{1}{\sum_{i=0}^{N-1} \hat{\alpha}_t(i)} \right)$$

Let's take a look at the implementation of the alpha class in Scala, using the referenced number of the mathematical expressions of the alpha class. The alpha and beta values have to be normalized [M3], and therefore, we define an `HMMTreillis` base class for the alpha and beta algorithms that implements the normalization:

```
class HMMTreillis(numObs: Int, numStates: Int) { //5
    var treillis: DMatrix = _ //6
```

```

val ct = Array.fill(numObs) (0.0)

def normalize(t: Int): Unit = { //7
    ct.update(t, /:(numStates, (s, n) => s + treillis(t, n)))
    treillis /= (t, ct(t))
}
def getTreillis: DMatrix = treillis
}

```

The `HMMTreillis` class has two configuration parameters: the number of observations, `numObs`, and the number of states, `numStates` (line 5). The `treillis` variable represents the scaling matrix used in the alpha (or forward) and beta (or backward) passes (line 6).

The normalization method, `normalize`, implements the **M6** formula by recomputing the `ct` scaling factor (line 7).


Computation efficiency

Scala's `reduce`, `fold`, and `foreach` methods are far more efficient than the `for` loop. You need to keep in mind that the main purpose of the `for` loop in Scala is the monadic composition of the `map` and `flatMap` operations.

The computation of the `alpha` variable in the `Alpha` class follows the same computation flow as defined in the **M4**, **M5**, and **M6** mathematical expressions:

```

class Alpha(lambda: HMMModel, obsSeq: Vector[Int]) //8
extends HMMTreillis(lambda.numObs, lambda.numStates) {

    val alpha: Double = Try {
        treillis = lambda.setAlpha(obsSeq) //9
        normalize(0) //10
        sumUp //11
    }.getOrElse(Double.NaN)

    override def isInitialized: Boolean = alpha != Double.NaN

    val last = lambda.numObs-1
    def sumUp: Double = {
        foreach(1, lambda.numObs, t => {
            updateAlpha(t) //12
            normalize(t) //13
        })
    }
}

```

```

    /:(lambda.numStates, (s,k) => s + treillis(last, k))
}

def updateAlpha(t: Int): Unit =
  foreach(lambda.numStates, i => { //14
    val newAlpha = lambda.getAlphaVal(treillis(t-1, i)
      treillis += (t, i, newAlpha, i, obsSeq(t)))
  })

def logProb: Double = /:(lambda.numObs, (s,t) => //15
  s + Math.log(ct(t)), Math.log(alpha))
}

```

The Alpha class has two arguments: the `lambda` model and the `obsSeq` sequence of observations (line 8). The definition of the scaling factor `alpha` initializes the `treillis` scaling matrix using the `HMMModel.setAlpha` method (line 9), normalizes the initial value of the matrix by invoking the `HMMTreillis.normalize` method for the first observation (line 10), and sums the matrix element to return the scaling factor by invoking `sumUp` (line 11).

The `setAlpha` method implements the mathematical expression **M4** as follows:

```

def setAlpha(obsSeq: Array[Int]): DMatrix =
  Range(0, numStates) ./: (DMatrix(numObs, numStates)) ((m,j) =>
    m += (0, j, pi(j)*B(j, obsSeq.head)))
}

```

The fold generates an instance of the `DMatrix` class, as described in the *Utility classes* section in the *Appendix A, Basic Concepts*.

The `sumUp` method implements the mathematical expression **M6** as follows:

- Update the `treillis` matrix of the scaling factor in the `updateAlpha` method (line 12)
- Normalize all the scaling factors for all the remaining observations (line 13)

The `updateAlpha` method updates the `treillis` scaling matrix by computing all the `alpha` factors for all states (line 14). The `logProb` method implements the mathematical expression **M7**. It computes the logarithm of the probability of observing a specific sequence, given the sequence of states and a predefined λ model (line 15).

The log probability

 The `logProb` method computes the logarithm of the probability instead of the probability itself. The summation of the logarithm of probabilities is less likely to cause an underflow than the product of probabilities.

Beta – the backward pass

The computation of beta values is similar to the `Alpha` class except that the iteration executes backward on the sequence of states.

The implementation of `Beta` is similar to the `alpha` class:

1. Compute (**M5**) and normalize (**M6**) the value of beta at $t = 0$ across states.
2. Compute and normalize iteratively the beta value at time $T - 1$ to t , which is updated from its value at $t + 1$ (**M7**).

Beta (the backward pass)

M8: Initialization of beta $\beta_{T-1}(t) = 1$.

M9: Normalization of initial beta values is defined as:



$$\hat{\beta}_{T-1}(i) = \beta_{T-1}(i) / \sum_{j=0}^{N-1} \beta_{T-1}(j)$$

M10: Normalized summation of beta is defined as:

$$\beta_t(i) = \sum_{j=0}^{N-1} \beta_{t+1}(j) \cdot a_{ij} \cdot b_j(O_{t+1}) \quad c_t = 1 / \sum_{j=0}^{N-1} \beta_t(j) \quad \hat{\beta}_t(i) = \beta_t(i) \cdot c_t$$

The definition of the `Beta` class is very similar to the `Alpha` class:

```
class Beta(lambda: HMMModel, obsSeq: Vector[Int])
  extends HMMTreillis(lambda.numObs, lambda.numStates) {

  val initialized: Boolean  //16

  override def isInitialized: Boolean = initialized
  def sumUp: Unit =    //17
    (lambda.numObs-2 to 0 by -1).foreach(t => { //18
```

```

        updateBeta(t)    //19
        normalize(t)
    })

def updateBeta(t: Int): Unit =
    foreach(lambda.numStates, i => {
        val newBeta = lambda.getBetaVal(treillis(t+1, i)
            treillis += (t, i, newBeta, i, obsSeq(t+1))) //20
    })
}

```

Contrary to the Alpha class, the Beta class does not generate an output value. The Beta class has an `initialized` Boolean attribute to indicate whether the constructor has executed successfully (line 16). The constructor updates and normalizes the beta matrix by traversing the sequence of observations backward from before the last observation to the first:

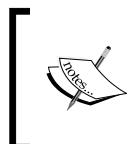
```

val initialized: Boolean = Try {
    treillis = DMatrix(lambda.numObs, lambda.numStates)
    treillis += (lambda.numObs-1, 1.0) //21
    normalize(lambda.numObs-1) //22
    sumUp //23
}.toBoolean("Beta initialization failed")

```

The initialization of the `treillis` beta scaling matrix of the `DMatrix` type assigns the value 1.0 to the last observation (line 21) and normalizes the beta values for the last observation, as defined in **M8** (line 22). It implements the mathematical expressions **M9** and **M10** by invoking the `sumUp` method (line 23).

The `sumUp` method is similar to `Alpha.sumUp` (line 17). It traverses the sequence of observations backward (line 18) and updates the beta scaling matrix, as defined in the mathematical expression **M9** (line 19). The implementation of the mathematical expression **M10** in the `updateBeta` method is similar to the alpha pass: it updates the `treillis` scaling matrix with the `newBeta` values computed in the `lambda` model (line 20).



Constructors and initialization

The alpha and beta values are computed within the constructors of their respective classes. The client code has to validate these instances by invoking `isInitialized`.

What is the value of a model if it cannot be created? The next canonical form CF2 leverages dynamic programming and recursive functions to extract the λ model.

Training – CF-2

The objective of this canonical form is to extract the λ model given a set of observations and a sequence of states. It is similar to the training of a classifier. The simple dependency of a current state on the previous state enables an implementation using an iterative procedure, known as the **Baum-Welch estimator** or **expectation-maximization (EM)**.

The Baum-Welch estimator (EM)

At its core, the algorithm consists of three steps and an iterative method, which is similar to the evaluation canonical form:

1. Compute the probability π (the gamma value at $t = 0$) (**M11**).
2. Compute and normalize the state's transition probabilities matrix A (**M12**).
3. Compute and normalize the matrix of emission probabilities B (**M13**).
4. Repeat steps 2 and 3 until the change of likelihood is insignificant.

The algorithm uses the digamma and summation gamma classes.

The Baum-Welch algorithm

M11: The joint probability of the state q_i at t and q_j at $t+1$ (digamma) is defined as:

$$\gamma_t(i,j) = p(q_t = S_i, q_{t+1} = S_j | O, \lambda)$$

$$\gamma_t(i,j) = \frac{\alpha_t(i)\alpha_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{j=0}^{N-1} \alpha_t(i)\beta_t(j)}$$

M12: The initial probabilities vector $N-1$ and sum of joint probabilities for all the states (gamma) are defined as:



$$\hat{\pi}_i = \gamma_0(i) \quad \gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i,j)$$

M13: The update of the transition probabilities matrix is defined as:

$$\hat{a}_{ij} = \frac{\sum_{t=0}^{T-1} [\gamma_t(i,j)]}{\sum_{t=0}^{T-1} \gamma_t(i)}$$

M14: The update of the emission probabilities matrix is defined as:

$$\hat{b}_{ij} = \frac{\sum_{t=0}^{T-1} \gamma_t(i,j)}{\sum_{t=0}^{T-1} \gamma_t(i)}$$

The Baum-Welch algorithm is implemented in the `BaumWelchEM` class and requires the following two inputs (line 24):

- The λ model, `lambda`, computed from the `config` configuration
- The `obsSeq` sequence (vector) of observations

The code will be as follows:

```
class BaumWelchEM(config: HMMConfig, obsSeq: Vector[Int]) { //24
    val lambda = HMMModel(config)
    val diGamma = new DiGamma(lambda.numObs, lambda.numStates)//25
    val gamma = new Gamma(lambda.numObs, lambda.numStates) //26
    val maxLikelihood: Option[Double] //27
}
```

The `DiGamma` class defines the joint probabilities for any consecutive states (line 25):

```
class DiGamma(numObs: Int, numStates: Int) {
    val diGamma = Array.fill(numObs-1)(DMatrix(numStates))
    def update(alpha: DMatrix, beta: DMatrix, A: DMatrix,
              B: DMatrix, obsSeq: Array[Int]): Try[Int]
}
```

The `diGamma` variable is an array of matrices that represents the joint probabilities of two consecutive states. It is initialized through an invocation of the `update` method, which implements the mathematical expression **M11**.

The `Gamma` class computes the sum of the joint probabilities across all the states (line 26):

```
class Gamma(numObs: Int, numStates: Int) {
    val gamma = DMatrix(numObs, numStates)
    def update(alpha: DMatrix, beta: DMatrix): Unit
}
```

The `update` method of the `Gamma` class implements the mathematical expression **M12**.

Source code for Gamma and DiGamma

The `Gamma` and `DiGamma` classes implement the mathematical expressions for the Baum-Welch algorithm. The `update` method uses simple linear algebra and is not described; refer to the documented source code for details.

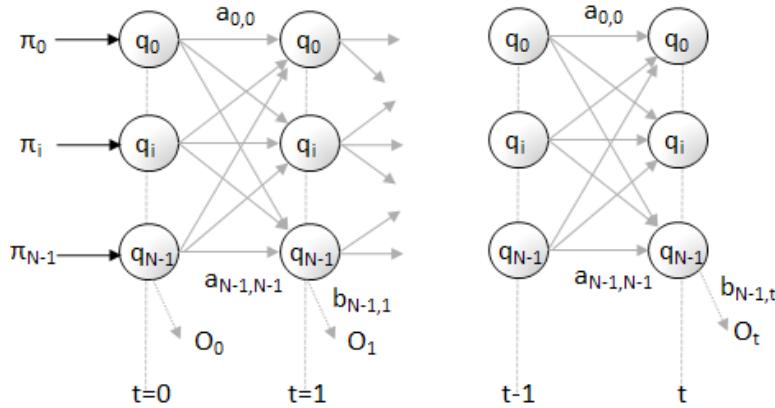
The maximum likelihood, `maxLikelihood`, for the sequence of states given an existing lambda model and a sequence of observations (line 27) is computed using the `getLikelihood` tail recursive method, as follows:

```
val maxLikelihood: Option[Double] = Try {  
  
    @tailrec  
    def getLikelihood(likelihood: Double, index: Int): Double = {  
        lambda.update(gamma, diGamma, obsSeq) //28  
        val _likelihood = frwrdbckwrdLattice //29  
        val diff = likelihood - _likelihood  
  
        if( diff < config.eps ) _likelihood //30  
        else if (index >= config.maxIters) //31  
            throw new IllegalStateException(" ... ")  
        else getLikelihood(_likelihood, index+1)  
    }  
  
    val max = getLikelihood(frwrdbckwrdLattice, 0)  
    lambda.normalize //32  
    max  
}._toOption("BaumWelchEM not initialized", logger)
```

The `maxLikelihood` value implements the mathematical expressions **M13** and **M14**. The `getLikelihood` recursive method updates the lambda model matrices A and B and initial state probabilities p_i (line 28). The likelihood for the sequence of states is recomputed using the forward-backward lattice algorithm implemented in the `frwrdbckwrdLattice` method (line 29).

 **Update of lambda model**
The update method of the `HMMModel` object uses simple linear algebra and is not described; refer to the documented source code for details.

The core of the Baum-Welch expectation maximization is the iterative forward and backward update of the lattice of states and observations between time t and $t + 1$. The lattice-based iterative computation is illustrated in the following diagram:



The visualization of the HMM graph lattice for the Baum-Welch algorithm

The code will be as follows:

```
def frwrdBckwrldLattice: Double = {
    val _alpha = Alpha(lambda, obsSeq) //33
    val beta = Beta(lambda, obsSeq).getTreillis //34
    val alphas = _alpha.getTreillis
    gamma.update(alphas, beta) //35
    diGamma.update(alphas, beta, lambda.A, lambda.B, obsSeq)
    _alpha.alpha
}
```

The forward-backward algorithm uses the `Alpha` class for the computation/update of the `lambda` model in the forward pass (line 33) and the `Beta` class for the update of `lambda` in the backward pass (line 34). The joint probabilities-related `gamma` and `diGamma` matrices are updated at each recursion (line 35), reflecting the iteration of the mathematical expressions **M11** to **M14**.

The recursive computation of `maxLikelihood` exists if the algorithm converges (line 30). It throws an exception if the maximum number of recursions is exceeded (line 31).

Decoding – CF-3

This last canonical form consists of extracting the most likely sequence of states $\{q_t\}$ given a set of observations O_t and a λ model. Solving this problem requires, once again, a recursive algorithm.

The Viterbi algorithm

The extraction of the best state sequence (the sequence of a state that has the highest probability) is very time consuming. An alternative consists of applying a dynamic programming technique to find the best sequence $\{q_t\}$ through iteration. This algorithm is known as the **Viterbi algorithm**. Given a sequence of states $\{q_t\}$ and sequence of observations $\{o_t\}$, the probability $\delta_t(i)$ for any sequence to have the highest probability path for the first T observations is defined for the state S_i [7:7].

The Viterbi algorithm

M12: The definition of the delta function is as follows:

$$\delta_t(i) = \max_{q_j \in \{0, T-1\}} p(q_{0:T-1} = S_i, O_{0:T-1} | \lambda)$$

M13: Initialization of delta is defined as:



$$\delta_0(i) = \pi_i b_i(O_0) \quad \psi_0(i) = 0 \quad \forall i$$

M14: Recursive computation of delta is defined as:

$$\delta_t(j) = \max_i (\delta_{t-1}(i) \cdot a_{ij} \cdot b_j(O_t)) \quad \psi_t(j) = \arg \max_i (\delta_{t-1}(i) \cdot a_{ij})$$

M15: The computation of the optimum state sequence $\{q\}$ is defined as:

$$q^*_{t+1} = \psi_{t+1}(q^*_{t+1}) \quad q^*_T = \arg \max_i \delta_T(i)$$

The `ViterbiPath` class implements the Viterbi algorithm whose purpose is to compute the optimum sequence (or path) of states given a set of observations and a λ model. The optimum sequence or path of states is computed by maximizing the delta function.

The constructors for the `ViterbiPath` class have the same arguments as the forward, backward, and Baum-Welch algorithm: the `lambda` model and the set of observations `obsSeq`:

```
class ViterbiPath(lambda: HMMModel, obsSeq: Vector[Int]) {
    val nObs = lambda.numObs
    val nStates = lambda.numStates
    val psi = Array.fill(nObs)(Array.fill(nStates)(0)) //35
    val qStar = new QStar(nObs, nStates) //36

    val delta = { //37
```

```

Range(0, nStates) ./: (DMatrix(nObs, nStates)) ((m, n) => {
  psi(0)(n) = 0
  m += (0, n, lambda.pi(n) * lambda.B(n, obsSeq.head))
})
val path = HMMPrediction(viterbi(1), qStar()) //38
}

```

As seen in the preceding information box containing the mathematical expressions for the Viterbi algorithm, the following matrices have to be defined:

- `psi`: This is the matrix of indices of `nObs` observations by indices of `nStates` states (line 35).
- `qStar`: This is the optimum sequence of states at each recursion of the Viterbi algorithm (line 36).
- `delta`: This is the sequence that has the highest probability path for the first n observations. It also sets the `psi` values for the first observation to 0 (line 37).

All members of the `ViterbiPath` class are private except `path` that defines the optimum sequence or path of states given the `obsSeq` observations (line 38).

The matrix that defines the maximum probability `delta` of any sequence of states given the `lambda` model and the `obsSeq` observation is initialized using the mathematical expression **M13** (line 37). The predictive model returns the path or optimum sequence of states as an instance of `HMMPrediction`:

```
case class HMMPrediction(likelihood: Double, states: Array[Int])
```

The first argument of `likelihood` is computed by the `viterbi` recursive method. The indices of the states in the `states` optimum sequence is computed by the `QStar` class (line 38).

Let's take a look under the hood of the Viterbi recursive method:

```

@tailrec
def viterbi(t: Int): Double = {
  Range(0, numStates).foreach( updateMaxDelta(t, _) ) //39

  if( t == obsSeq.size-1) { //40
    val idxMaxDelta = Range(0, numStates)
      .map(i => (i, delta(t, i))).maxBy(_.value) //41
    qStar.update(t+1, idxMaxDelta._1) //42
    idxMaxDelta._2
  }
  else viterbi(t+1) //43
}

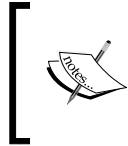
```

The recursion started on the second observation as the `qStar`, `psi`, and `delta` parameters have already been initialized in the constructor. The recursive implementation invokes the `updateMaxDelta` method to update the `psi` indexing matrix and the highest probability for any state, as follows:

```
def updateMaxDelta(t: Int, j: Int): Unit = {  
    val idxDelta = Range(0, nStates)  
        .map(i => (i, delta(t-1, i)*lambda.A(i, j)))  
        .maxBy(_.value) //44  
    psi(t)(j) = idxDelta._1  
    delta += (t, j, idxDelta._2) //45  
}
```

The `updateMaxDelta` method implements the mathematical expression **M14** that extracts the index of the state that maximizes `psi` (line 44). The `delta` probability matrix and the `psi` indexing matrix are updated accordingly (line 45).

The `viterbi` method is called recursively for the remaining observations except the last one (line 43). At the last observation of the `obsSeq.size-1` index, the algorithm executes the mathematical expression **M15**, which is implemented in the `QStar` class (line 42).



The QStar class

The `QStar` class and its `update` method use linear algebra and are not described here; refer to the documented source code and Scaladocs files for details.

This implementation of the decoding form of the hidden Markov model completes the description of the hidden Markov model and its implementation in Scala. Now, let's put this knowledge into practice.

Putting it all together

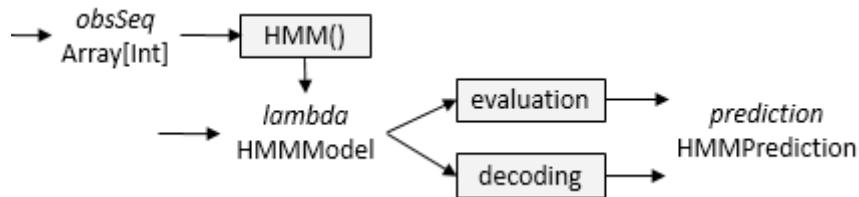
The main `HMM` class implements the three canonical forms. A view bound to an array of integers is used to parameterize the `HMM` class. We assume that a time series of continuous or pseudo-continuous values is quantized into discrete symbol values.

The `@specialized` annotation ensures that the byte code is generated for the `Array[Int]` primitive without executing the conversion implicitly declared by the bound view.

There are two modes that execute any of the three canonical forms of the hidden Markov model:

- The `ViterbiPath` class: The constructor initializes/trains a model similar to any other learning algorithm, as described in the *Design template for immutable classifiers* section of the *Appendix A, Basic Concepts*. The constructor generates the model by executing the Baum-Welch algorithm. Once the model is successfully created, it can be used for decoding or evaluation.
- The `ViterbiPath` object: The companion provides the `decode` and `evaluate` methods for the decoding and evaluation of the sequence of observations using HMM.

The two modes of operations are described in the following diagram:



The computational flow for the hidden Markov model

Let's complete our implementation of the HMM with the definition of its class. The `HMM` class is defined as a data transformation using a model implicitly generated from an `xt` training set, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 46):

```

class HMM[@specialized(Double) T <: AnyVal] (
  config: HMMConfig,
  xt: XVSeries[T],
  form: HMMForm)
  (implicit quantize: Array[T] => Int, f: T => Double)
  extends ITransform[Array[T]](xt) with Monitor[Double] { //46

  type V = HMPrediction //47
  val obsSeq: Vector[Int] = xt.map(quantize(_)) //48

  val model: Option[HMMModel] = train //49
  override def |> : PartialFunction[U, Try[V]] //50
}
  
```

The `HMM` constructor takes the following four arguments (line 46):

- `config`: This is the configuration of the HMM that is the dimension of `lambda` model and execution parameters
- `xt`: This is the multidimensional time series of observations whose features have the `T` type
- `form`: This is the canonical form to be used once the model is generated (evaluation or decoding)
- `quantize`: This is the quantization function that converts an observation of the `Array[T]` type to an `Int` type
- `f`: This is the implicit conversion from the `T` type to `Double`

The constructor has to override the `V` type (`HMMPrediction`) of the output data (line 47) declared in the `ITransform` abstract class. The structure of the `HMMPrediction` class has been defined in the previous section.

The `Monitor` trait is used to collect the profiling information during training (refer to the `Monitor` section under *Utility classes* in the *Appendix A, Basic Concepts*).

The time series of `xt` observations is converted into a vector of `obsSeq` observed states by applying the `quantize` quantization function to each observation (line 48).

As with any supervised learning technique, the model is created through training (line 49). Finally, the `|>` polymorphic predictor invokes either the `decode` method or the `evaluate` method (line 50).

The `train` method consists of the execution of the Baum-Welch algorithm and returns the `lambda` model:

```
def train: Option[HMMModel] = Try {
    BaumWelchEM(config, obsSeq).lambda }.toOption
```

Finally, the `|>` predictor is a simple wrapper to the evaluation form (`evaluate`) and the decoding form (`decode`):

```
override def |>: PartialFunction[U, Try[V]] = {
    case x: Array[T] if(isModel && x.length > 1) =>
        form match {
            case _: EVALUATION =>
                evaluation(model.get, Vector[Int](quantize(x)))
            case _: DECODING =>
                decoding(model.get, Vector[Int](quantize(x)))
        }
}
```

The protected `evaluation` method of the `HMM` companion object is a wrapper around the `Alpha` computation:

```
def evaluation(model: HMMModel,
  obsSeq: Vector[Int]): Try[HMPrediction] = Try {
  HMPrediction(-Alpha(model, obsSeq).logProb, obsSeq.toArray)
}
```

The `evaluate` method of the `HMM` object exposes the evaluation canonical form:

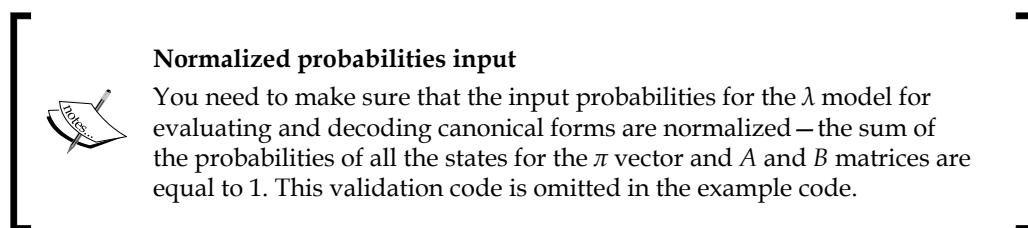
```
def evaluate[T <: AnyVal](model: HMMModel,
  xt: XVSeries[T])(implicit quantize: Array[T] => Int,
  f: T => Double): Option[HMPrediction] =
  evaluation(model, xt.map(quantize(_))).toOption
```

The decoding method wraps the Viterbi algorithm to extract the optimum sequence of states:

```
def decoding(model: HMMModel, obsSeq: Vector[Int]): Try[HMPrediction] = Try {
  ViterbiPath(model, obsSeq).path
}
```

The `decode` method of the `HMM` object exposes the decoding canonical form:

```
def decode[T <: AnyVal](model: HMMModel,
  xt: XVSeries[T])(implicit quantize: Array[T] => Int,
  f: T => Double): Option[HMPrediction] =
  decoding(model, xt.map(quantize(_))).toOption
```

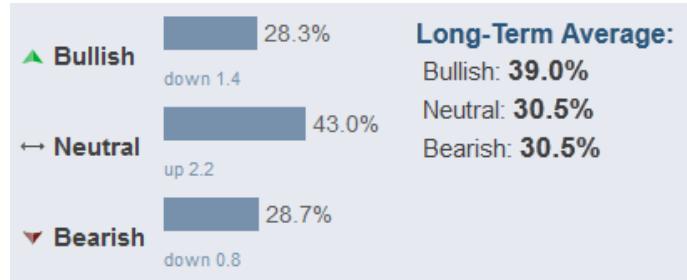


Test case 1 – training

Our first test case is to train an HMM to predict the sentiment of investors as measured by the weekly sentiment survey of the members of the **American Association of Individual Investors (AAII)** [7:8]. The goal is to compute the transition probabilities matrix A , the emission probabilities matrix B , and the steady state probability distribution π , given the observations and hidden states (training canonical forms).

We assume that the change in investor sentiments is independent of time, as required by the hidden Markov model.

The AAII sentiment survey grades the bullishness on the market in terms of percentage:



The weekly AAII market sentiment (reproduced by courtesy from AAII)

The sentiment of investors is known as a contrarian indicator of the future direction of the stock market. Refer to the *Terminology* section in the *Appendix A, Basic Concepts*.

Let's select the ratio of the percentage of investors that are bullish over the percentage of investors that are bearish. The ratio is then normalized.

The following table lists this:

Time	Bullish	Bearish	Neutral	Ratio	Normalized Ratio
t0	0.38	0.15	0.47	2.53	1.0
t1	0.41	0.25	0.34	1.68	0.53
t2	0.25	0.35	0.40	0.71	0.0
...

The sequence of nonnormalized observations (the ratio of bullish sentiments over bearish sentiments) is defined in a CSV file as follows:

```

val OBS_PATH = "resources/data/chap7/obsprob.csv"
val NUM_SYMBOLS = 6
val NUM_STATES = 5
val EPS = 1e-4
val MAX_ITERS = 150
val observations = Vector[Double] (
  0.01, 0.72, 0.78, 0.56, 0.61, 0.56, 0.45, ...
)

val quantize = (x: DblArray) =>
  (x.head * (NUM_STATES+1)).floor.toInt //51
val xt = observations.map(Array[Double](_))

```

```

val config = HMMConfig(zt.size, NUM_STATES, NUM_SYMBOLS,
    MAX_ITERS, EPS)
val hmm = HMM[Array[Int]](config, xt) //52
show(s"Training):\n${hmm.model.toString()}")

```

The constructor for the `HMM` class requires a `T => Array[Int]` implicit conversion, which is implemented by the `quantize` function (line 51). The `hmm.model` model is created by instantiating an `HMM` class with a predefined configuration and an `obsSeq` sequence of observed states (line 52).

The training of the HMM generates the following state transition probabilities matrix:

A	1	2	3	4	5
1	0.090	0.026	0.056	0.046	0.150
2	0.094	0.123	0.074	0.058	0.0
3	0.093	0.169	0.087	0.061	0.056
4	0.033	0.342	0.017	0.031	0.147
5	0.386	0.47	0.314	0.541	0.271

The emission matrix is as follows:

B	1	2	3	4	5	6
1	0.203	0.313	0.511	0.722	0.264	0.307
2	0.149	0.729	0.258	0.389	0.324	0.471
3	0.305	0.617	0.427	0.596	0.189	0.186
4	0.207	0.312	0.351	0.653	0.358	0.442
5	0.674	0.520	0.248	0.294	0.259	0.03

Test case 2 – evaluation

The objective of the evaluation is to compute the probability of the `xt` observed data given a λ model (A_0 , B_0 , and π_0):

```

val A0 = Array[Array[Double]](
    Array[Double](0.21, 0.13, 0.25, 0.06, 0.11, 0.24),
    Array[Double](0.31, 0.17, 0.18, 0.04, 0.19, 0.11),
    ...
)
val B0 = Array[Array[Double]](
    Array[Double](0.61, 0.39),
    Array[Double](0.54, 0.46),
    ...
)

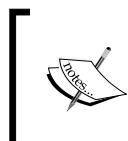
```

```
)  
val PI0 = Array[Double] (  
    0.26, 0.04, 0.11, 0.26, 0.19, 0.14)  
  
val xt = Vector[Double] (  
    0.0, 1.0, 21.0, 1.0, 30.0, 0.0, 1.0, 0.0, ...  
).map(Array[Double](_))  
val max = data.max  
val min = data.min  
implicit val quantize = (x: DblArray) =>  
    ((x.head/(max - min) + min)*(B0.head.length-1)).toInt //55  
val lambda = HMMModel(  
    DMatrix(A0), DMatrix(B0), PI0, xt.length) //53  
evaluation(lambda, xt).map(_.toString).map(show(_)) //54
```

The model is created directly by converting the A_0 state-transition probabilities and B_0 emission probabilities as matrices of the `DMatrix` type (line 53). The evaluation method generates an `HMPrediction` object, which is stringized, and then displays it in the standard output (line 54).

The quantization method consists of normalizing the input data over the number (or range) of symbols associated with the `lambda` model. The number of symbols is the size of the rows of the emission probabilities matrix B . In this case, the range of the input data is [0.0, 3.0]. The range is normalized using the linear transform $f(x) = x / (max - min) + min$, then adjusted for the number of symbols (or values for states) (line 55).

The `quantize` quantization function has to be explicitly defined before invoking the evaluation method.



Test case for decoding

Refer to the source code and the API documents for the test case related to the decoding form.

HMM as a filtering technique

The evaluation form of the hidden Markov model is very suitable for filtering data for discrete states. Contrary to time series filters such as the Kalman filter introduced in the *The discrete Kalman filter* section in *Chapter 3, Data Preprocessing*, the HMM requires data to be stationary in order to create a reliable model. However, the hidden Markov model overcomes some of the limitations of analytical time series analysis. Filters and smoothing techniques assume that the noise (frequency mean, variance, and covariance) is known and usually follows a Gaussian distribution.

The hidden Markov model does not have such a restriction. Filtering techniques, such as moving averaging techniques, discrete Fourier transforms, and Kalman filters apply to both discrete and continuous states while the HMM does not. Moreover, the extended Kalman filter can estimate nonlinear states.

Conditional random fields

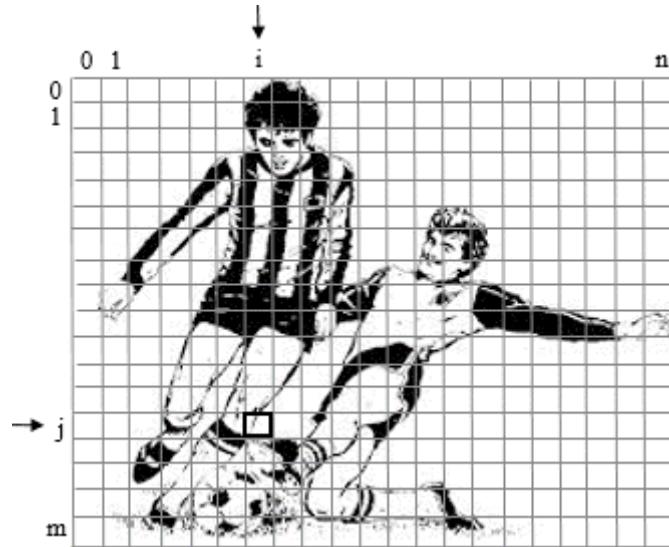
The **conditional random field (CRF)** is a discriminative machine learning algorithm introduced by John Lafferty, Andrew McCallum, and Fernando Pereira [7:9] at the turn of the century as an alternative to the HMM. The algorithm was originally developed to assign labels to a set of observation sequences.

Let's consider a concrete example to understand the conditional relation between the observations and the label data.

Introduction to CRF

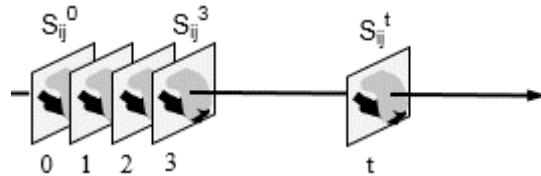
Let's consider the problem of detecting a foul during a soccer game using a combination of video and audio. The objective is to assist the referee and analyze the behavior of the players to determine whether an action on the field is dangerous (red card), inappropriate (yellow card), in doubt to be replayed, or legitimate.

The following image is an example of the segmentation of a video frame for image processing:



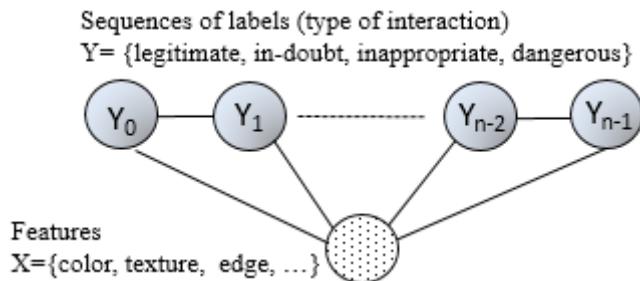
An example of an image processing problem requiring machine learning

The analysis of the video consists of segmenting each video frame and extracting image features such as colors or edges [7:10]. A simple segmentation scheme consists of breaking down each video frame into tiles or groups of pixels indexed by their coordinates on the screen. A time sequence is then created for each tile S_{ij} as represented in the following image:



The learning strategy for pixels in a sequence of video frames

The image segment S_{ij} is one of the labels that is associated with multiple observations. The same features extraction process applies to the audio associated with the video. The relation between the video/image segment and the hidden state of the altercation between the soccer players is illustrated in the following model graph:

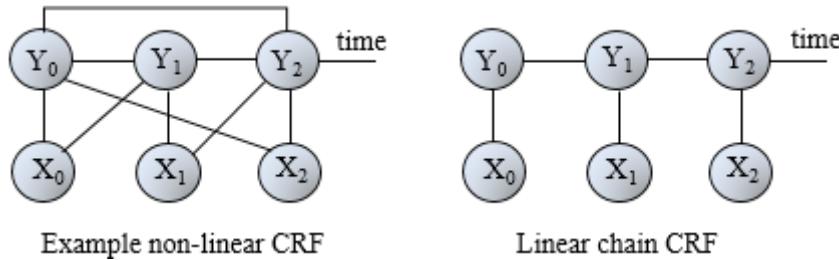


The undirected graph representation of CRF for the soccer infraction detection

CRFs are discriminative models that can be regarded as a structured output extension of the logistic regression. CRFs address the problem of labeling a sequence of data, such as assigning a tag to each word in a sentence. The objective is to estimate the correlation among the output (observed) values Y that are conditional on the input values (features) X .

The correlation between the output and input values is described as a graph (also known as a **graph-structured CRF**). A good example of graph-structured CRFs are cliques. Cliques are sets of connected nodes in a graph for which each vertex has an edge connecting it to every other vertex in the clique.

Such models are complex and their implementation is challenging. Most real-world problems related to time series or ordered sequences of data can be solved as a correlation between a linear sequence of observations and a linear sequence of input data, which is similar to the HMM. Such a model is known as the **linear chain structured CRF** or **linear chain CRF** for short:



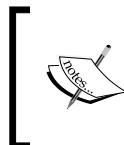
An illustration of a nonlinear and linear chain CRF

One main advantage of the linear chain CRF is that the maximum likelihood $p(Y|X, w)$ can be estimated using dynamic programming techniques such as the Viterbi algorithm used in the HMM. From now on, the section focuses exclusively on the linear chain CRF in order to stay consistent with the HMM, as described in the previous section.

Linear chain CRF

Let's consider a random variable $X=\{x_i\}_{0:n-1}$ representing n observations and a random variable Y representing a corresponding sequence of labels $Y=\{y_j\}_{0:m-1}$. The hidden Markov model estimates the joint probability $p(X, Y)$, as any generative model requires the enumeration of all the sequences of observations.

If each element of Y, y_j obeys the first order of the Markov property, then (Y, X) is a CRF. The likelihood is defined as a conditional probability $p(Y|X, w)$, where w is the model parameters vector.

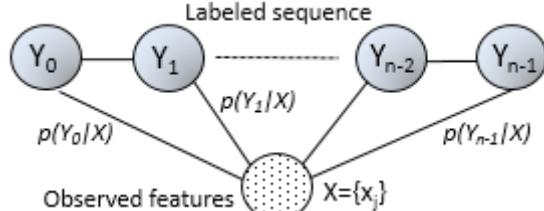


Observation dependencies

The purpose of CRF models is to estimate the maximum likelihood of $p(Y|X, w)$. Therefore, independence between X observations is not required.

A graphical model is a probabilistic model for which a graph denotes the conditional independence between random variables (vertices). The conditional and joint probabilities of random variables are represented as edges. The graph for generic conditional random fields can indeed be complex. The most common and simplistic graph is the linear chain CRF.

A first order linear chain conditional random field can be visualized as an undirected graphical model, which illustrates the conditional probability of a label Y_j given a set of observations X :



A linear, conditional, random field undirected graph

The Markov property simplifies the conditional probabilities of Y , given X , by considering only the neighbor labels $p(Y_1 | X, Y_j, j \neq 1) = p(Y_1 | X, Y_0, Y_2)$ and $p(Y_i | X, Y_j, j \neq i) = p(Y_i | X, Y_{i-1}, Y_{i+1})$.

The conditional random fields introduce a new set of entities and a new terminology:

- **Potential functions (f):** These strictly positive and real value functions represent a set of constraints on the configurations of random variables. They do not have any obvious probabilistic interpretation.
- **Identity potential functions:** These are potential functions $I(x, t)$ that take 1 if the condition on the feature x at time t is true, and 0 otherwise.
- **Transition feature functions:** Simply known as feature functions, t_i , are potential functions that take a sequence of features $\{X_j\}$, the previous label Y_{i-1} , the current label Y_i , and an index i . The transition feature function outputs a real value function. In a text analysis, a transition feature function would be defined by a sentence as a sequence of observed features, the previous word, the current word, and a position of a word in a sentence. Each transition feature function is assigned a weight that is similar to the weights or parameters in the logistic regression. Transition feature functions play a similar role to the state transition factors a_{ij} in the HMM but without a direct probabilistic interpretation.
- **State feature functions (s):** These are potential functions that take the sequence of features $\{X_j\}$, the current label Y_i , and the index i . They play a similar role to the emission factors in the HMM.

A CRF defines the log probability of a particular label sequence Y , given a sequence of observations X as the normalized product of the transition feature and state feature functions. In other words, the likelihood of a particular sequence Y , given the observed features X , is a logistic regression.

The mathematical notation to compute the conditional probabilities in the case of a first order linear chain CRF is described in the following information box:

The CRF conditional distribution

M1: The log probability of a label's sequence y , given an observation x is defined as:

$$\log f_i(y_{i-1}, y_i, x, i) = w_c + \sum_{i=0}^{K-1} w_i t_i(y_{i-1}, y_i, x, i) + \sum_{j=0}^{K-1} \mu_j s_j(y_i, x, i)$$

M2: Transition feature functions are defined as:

$$t_i(y_{i-1}, y_i, x, i) = I(y_{i-1} = l_1) \cdot I(y_i = l_2) \cdot I(x = 0)$$

 M3: Using the notation:

$$F_i(y, x) = \sum_{j=0}^{K-1} f_j(y_{j-1}, y_j, x, i) \quad \log p(y|x, \lambda) \propto \sum_{j=0}^{K-1} w_j F_j(x, y)$$

M4: The conditional distribution of labels y , given x , using the Markov property is defined as:

$$p(y|x, w) = \frac{1}{Z(x)} e^{\sum_{j=0}^{K-1} w_j F_j(x, y)} \quad z(x) = \sum_{i=0}^{N-1} \sum_{j=0}^{K-1} w_j F_j(x, y)$$

The weights w_j are sometimes referred as λ in scientific papers, which may confuse the reader; w is used to avoid any confusion with the λ regularization factor.

Now, let's get acquainted with the conditional random fields algorithm and its implementation by Sunita Sarawagi.

Regularized CRFs and text analytics

Most of the examples used to demonstrate the capabilities of conditional random fields are related to text mining, intrusion detection, or bioinformatics. Although these applications have a great commercial merit, they are not suitable for introductory test cases because they usually require a lengthy description of the model and the training process.

The feature functions model

For our example, we will select a simple problem: how to collect and aggregate an analyst's recommendation on any given stock from different sources with different formats.

Analysts at brokerage firms and investment funds routinely publish the list of recommendations or ratings for any stock. These analysts use different rating schemes from buy/hold/sell, A/B/C rating, and stars rating, to market perform/neutral/market underperform rating. For this example, the rating is normalized as follows:

- 0 for a strong sell (F or 1 star rating)
- 1 for sell (D, 2 stars, or market underperform)
- 2 for neutral (C, hold, 3 stars, market perform, and so on)
- 3 for buy (B, 4 stars, market overperform, and so on)
- 4 for strong buy (A, 5 stars, highly recommended, and so on)

Here are examples of recommendations by stock analysts:

- *Macquarie upgraded AUY from Neutral to Outperform rating*
- *Raymond James initiates Ainsworth Lumber as Outperform*
- *BMO Capital Markets upgrades Bear Creek Mining to Outperform*
- *Goldman Sachs adds IBM to its conviction list*

The objective is to extract the name of the financial institution that publishes the recommendation or rating, the stock rated, the previous rating, if available, and the new rating. The output can be inserted into a database for further trend analysis, prediction, or simply the creation of reports.

The scope of the application

Ratings from analysts are updated every day through different protocols (feed, e-mails, blogs, web pages, and so on). The data has to be extracted from HTML, JSON, plain text, or XML format before being processed. In this exercise, we assume that the input has already been converted into plain text (ASCII) using a regular expression or another classifier.

The first step is to define the labels Y representing the categories or semantics of the rating. A segment or sequence is defined as a recommendation sentence. After reviewing the different recommendations, we are able to specify the following seven labels:

- Source of the recommendation (Goldman Sachs and so on)
- Action (upgrades, initiates, and so on)
- Stock (either the company name or the stock ticker symbol)
- From (an optional keyword)
- Rating (an optional previous rating)
- To
- Rating (the new rating for the stock)

The training set is generated from the raw data by *tagging* the different components of the recommendation. The first (or initial) rating for a stock does not have the labels 4 and 5 from the preceding list defined.

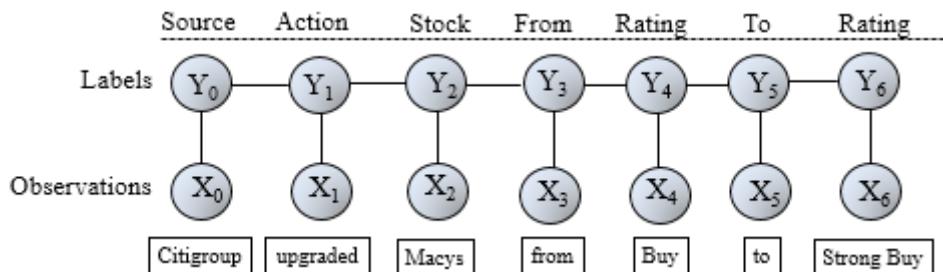
Consider the following example:

```
Citigroup // Y(0) = 1
upgraded // Y(1)
Macys // Y(2)
from // Y(3)
Buy // Y(4)
to // Y(5)
Strong Buy //Y(6) = 7
```

Tagging

Tagging as a word may have a different meaning depending on the context. In **natural language processing (NLP)**, tagging refers to the process of assigning an attribute (an adjective, pronoun, verb, proper name, and so on) to a word in a sentence [7:11].

A training sequence can be visualized in the following undirected graph:



An example of a recommendation as a CRF training sequence

You may wonder why we need to tag the *From* and *To* labels in the creation of the training set. The reason is that these keywords may not always be stated and/or their positions in the recommendation differ from one source to another.

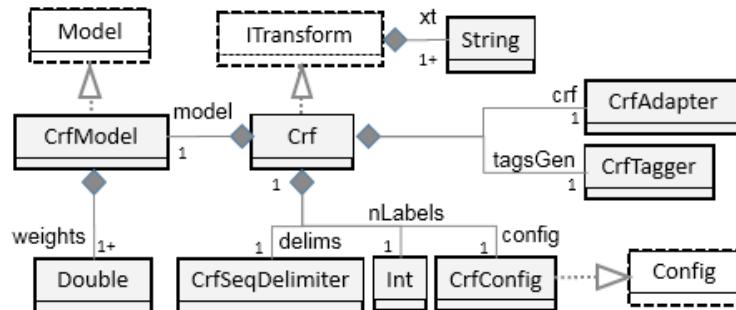
Design

The implementation of the conditional random fields follows the design template for classifiers, which is described in the *Design template for immutable classifiers* section under *Source code considerations* in the *Appendix A, Basic Concepts*.

Its key components are as follows:

- A `CrfModel` model of the `Model` type is initialized through training during the instantiation of the classifier. A model is an array of `weights`.
- The predictive or classification routine is implemented as an implicit data transformation of the `ITransform` type.
- The `Crf` conditional random field classifier has four parameters: the number of labels (or number of features), `nLabels`, configuration of the `CrfConfig` type, the sequence of delimiters of the `CrfSeqDelimiter` type, and a vector of name of files `xt` that contains the tagged observations.
- The `CrfAdapter` class interfaces with the IITB CRF library.
- The `CrfTagger` class extracts features from the tagged files.

The key software components of the conditional random fields are described in the following UML class diagram:



The UML class diagram for the conditional random fields

The UML diagram omits the utility traits and classes such as `Monitor` or the Apache Commons Math components.

Implementation

The test case uses the IIT-B's CRF Java implementation from the Indian Institute of Technology Bombay by Sunita Sarawagi. The JAR files can be downloaded from SourceForge (<http://sourceforge.net/projects/crf/>).

The library is available as JAR files and source code. Some of the functionalities, such as the selection of a training algorithm, is not available through the API. The components (JAR files) of the library are as follows:

- A CRF for the implementation of the CRF algorithm
- LBFGS for limited-memory Broyden-Fletcher-Goldfarb-Shanno nonlinear optimization of convex functions (used in training)
- The CERN Colt library for the manipulation of a matrix
- The GNU generic hash container for indexing

Configuring the CRF classifier

Let's take a look at the `Crf` class that implements the conditional random fields classifier. The `Crf` class is defined as a data transformation of the `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 2):

```
class Crf(nLabels: Int, config: CrfConfig,
  delims: CrfSeqDelimiter, xt: Vector[String])//1
  extends ITransform[String](xt) with Monitor[Double]//2

  type V = Double //3
  val tagsGen = new CrfTagger(nLabels) //4
  val crf = CrfAdapter(nLabels, tagsGen, config.params) //5
  val model: Option[CrfModel] = train //6
  weights: Option[DblArray] = model.map(_.weights)

  override def |> : PartialFunction[String, Try[V]] //7
}
```

The constructor for `Crf` has the following four arguments (line 1):

- `nLabels`: These are the number of labels used for the classification
- `config`: This is the configuration parameter used to train `Crf`
- `delims`: These are the delimiters used in raw and tagged files
- `xt`: This is a vector of name of files that contains raw and tagged data

 **Filenames for raw and tagged data**

For the sake of simplicity, the files for the raw observations and the tagged observations have the same name with different extensions: `filename.raw` and `filename.tagged`.

The `Monitor` trait is used to collect the profiling information during training (refer to the `Monitor` section under *Utility classes* in the *Appendix A, Basic Concepts*).

The `v` type of the output of the `|>` predictor is defined as `Double` (line 3).

The execution of the CRF algorithm is controlled by a wide variety of configuration parameters encapsulated in the `CrfConfig` configuration class:

```
class CrfConfig(w0: Double, maxIters: Int, lambda: Double,  
    eps: Double) extends Config { //8  
    val params = s"""initValue $w0 maxIters $maxIters  
        | lambda $lambda scale true eps $eps""".stripMargin  
}
```

For the sake of simplicity, we use the default `CrfConfig` configuration parameters to control the execution of the learning algorithm, with the exception of the following four variables (line 8):

- Initialization of the `w0` weight that uses either a predefined or random value between 0 and 1 (default 0)
- The maximum number of iterations, `maxIters`, that is used in the computation of the weights during the learning phase (default 50)
- The `lambda` scaling factor for the L_2 penalty function that is used to reduce observations with a high value (default 1.0)

- The `eps` convergence criteria that is used to compute the optimum values for the `wj` weights (default 1e-4)

 **The L_2 regularization**

This implementation of the conditional random fields support the L_2 regularization, as described in the *Regularization* section in *Chapter 6, Regression and Regularization*. The regularization is turned off by setting $\lambda = 0$.

The `CrfSeqDelimiter` case class specifies the following regular expressions:

- `obsDelim` to parse each observation in the raw files
- `labelsDelim` to parse each labeled record in the tagged files
- `seqDelim` to extract records from raw and tagged files

The code will be as follows:

```
case class CrfSeqDelimiter(obsDelim: String,
                            labelsDelim: String, seqDelim: String)
```

The `DEFAULT_SEQ_DELIMITER` instance is the default sequence delimiter used in this implementation:

```
val DEFAULT_SEQ_DELIMITER =
  new CrfSeqDelimiter(",\t/-():.;'?"`&_, "//", "\n")
```

The `CrfTagger` tag or label generator iterates through the tagged file and applies the relevant regular expressions of `CrfSeqDelimiter` to extract the symbols used in training (line 4).

The `CrfAdapter` object defines the different interfaces to the IITB CRF library (line 5). The factory for CRF instances is implemented by the `apply` constructor as follows:

```
object CrfAdapter {
  import iitb.CRF.CRF
  def apply(nLabels: Int, tagger: CrfTagger,
            config: String): CRF = new CRF(nLabels, tagger, config)
  ...
}
```

Adapter classes to the IITB CRF library

The training of the conditional random field for sequences requires to define a few key interfaces:

- `DataSequence` to specify the mechanism to access observations and labels for training and testing data
- `DataIter` to iterate through the sequence of data created using the `DataSequence` interface
- `FeatureGenerator` to aggregate all the feature types

 These interfaces have default implementations bundled in the CRF Java library [7:12]. Each of these interfaces have to be implemented as adapter classes:

```
class CrfTagger(nLabels: Int) extends FeatureGenerator
class CrfDataSeq(nLabels: Int, tags: Vector[String],
delim: String) extends DataSequence
class CrfSeqIter(nLabels: Int, input: String, delim:
CrfSeqDelimiter) extends DataIter
```

Refer to the documented source code and Scaladocs files for the description and implementation of these adapter classes.

Training the CRF model

The objective of the training is to compute the weights w_j that maximize the conditional log-likelihood function without the L_2 penalty function. Maximizing the log-likelihood function is equivalent to minimizing the loss with the L_2 penalty. The function is convex, and therefore, any variant gradient descent (greedy) algorithm can be applied iteratively.

 M5: The conditional log-likelihood for a linear chain CRF training set $D = \{x_i, y_i\}$ is given as follows:

$$\mathcal{L}(w, D) = - \sum_{i=0}^{n-1} \log p(y_i | x_i, w)$$

M6: Maximization of the loss function and L2 penalty is given as follows:

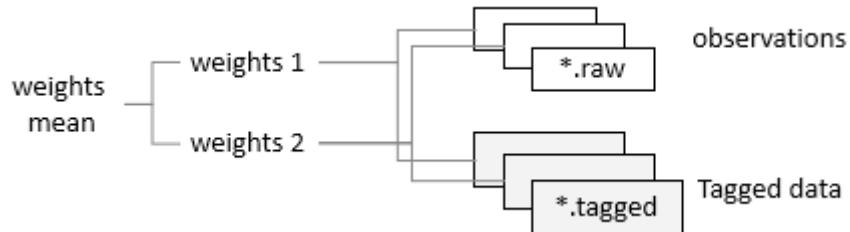
$$w^* = \arg \max_{\lambda} [\mathcal{L}(w, D) + \lambda \|w\|^2] \quad \lambda = \frac{1}{2\sigma^2}$$

The training file consists of a pair of files:

- **Raw datasets:** Recommendations (such as *Raymond James upgrades Gentiva Health Services from Underperform to Market perform*)
- **Tagged datasets:** Tagged recommendations (such as *Raymond James [1] upgrades [2] Gentiva Health Services [3], from [4] Underperform [5] to [6] Market perform [7]*)

 **Tags type**
In this implementation, the tags have the `Int` type. However, alternative types, such as enumerators or even continuous values (that is, `Double`) can be used.

The training or computation of weights can be quite expensive. It is highly recommended that you distribute the observations and tagged observations dataset across multiple files, so they can be processed concurrently:



The distribution of the computation of the weights of the CRF

The `train` method creates the model by computing the weights of the CRF. It is invoked by the constructor of `Crf`:

```

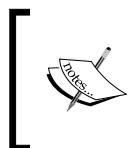
def train: Option[CrfModel] = Try {
  val weights = if(xt.size == 1) //9
    computeWeights(xt.head)
  else {
    val weightsSeries = xt.map( computeWeights(_) )
    statistics(weightsSeries).map(_.mean).toArray //10
  }
  new CrfModel(weights) //11
}._toOption("Crf training failed", logger)
  
```

We cannot assume that there is only one tagged dataset (that is, a single pair of `*.raw` and `*.tagged` files) (line 9). The `computeWeights` method used for computation of weights for the CRF is applied to the first dataset if there is only one pair of raw and tagged file. In the case of multiple datasets, the `train` method computes the mean of all the weights extracted from each tagged dataset (line 10). The mean of the weights are computed using the `statistics` method of the `XTSeries` object, which was introduced in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*. The `train` method returns `CrfModel` if successful, and `None` otherwise (line 11).

For efficiency purpose, the map should be parallelized using the `ParVector` class as follows:

```
val parXt = xt.par
val pool = new ForkJoinPool(nTasks)
v.tasksupport = new ForkJoinTaskSupport(pool)
parXt.map(computeWeights(_))
```

The parallel collections are described in detail in the *Parallel collections* section under *Scala* in *Chapter 12, Scalable Frameworks*.



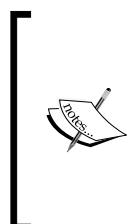
CRF weights computation

It is assumed that input tagged files share the same list of tags or symbols, so each dataset produces the same array of weights.



The `computeWeights` method extracts the weights from each pair of observations and tagged observation files. It invokes the `train` method of the `CrfTagger` tag generator (line 12) to prepare, normalize, and set up the training set, and then invokes the training procedure on the IITB CRF class (line 13):

```
def computeWeights(tagsFile: String): DblArray = {
    val seqIter = CrfSeqIter(nLabels, tagsFile, delims)
    tagsGen.train(seqIter) //12
    crf.train(seqIter) //13
}
```



The scope of the IITB CRF Java library evaluation

The CRF library has been evaluated with three simple text analytics test cases. Although the library is certainly robust enough to illustrate the internal workings of the CRF, I cannot vouch for its scalability or applicability in other fields of interests, such as bioinformatics or process control.



Applying the CRF model

The predictive method implements the `| >` data transformation operator. It takes a new observation (the analyst's recommendation on a stock) and returns the maximum likelihood, as shown here:

```
override def |> : PartialFunction[String, Try[V]] = {
  case obs: String if( !obs.isEmpty && isModel) => {
    val dataSeq = new CrfDataSeq(nLabels,obs,delims.obsDelim)
    Try (crf.apply(dataSeq)) //14
  }
}
```

The `| >` method merely creates a `dataSeq` data sequence and invokes the constructor of the IITB `CRF` class (line 14). The condition on the `obs` input argument to the partial function is rather rudimentary. A more elaborate condition of the observation should be implemented using a regular expression. The code to validate the arguments/parameters of the class and methods are omitted along with the exception handler for the sake of readability.

An advanced CRF configuration

 The CRF model of the **IITB** library is highly configurable. It allows developers to specify a state-label undirected graph with any combination of flat and nested dependencies between states. The source code includes several training algorithms such as the exponential gradient.

Tests

The client code to execute the test consists of defining the number of labels, `NLABELS` (that is, the number of tags for recommendation), the `LAMBDA` L2 penalty factor, the maximum of iterations, `MAX_ITERS`, allowed in the minimization of the loss function, and the `EPS` convergence criteria:

```
val LAMBDA = 0.5
val NLABELS = 9
val MAX_ITERS = 100
val W0 = 0.7
val EPS = 1e-3
val PATH = "resources/data/chap7/rating"
val OBS_DELIM = ",\t/ -():.;'?#`^&_"
```

```

val config = CrfConfig(W0 , MAX_ITERS, LAMBDA, EPS) //15
val delims = CrfSeqDelimiter(DELIM,"//","\n") //16
val crf = Crf(NLABELS, config, delims, PATH) //17
crf.weights.map( display(_) )

```

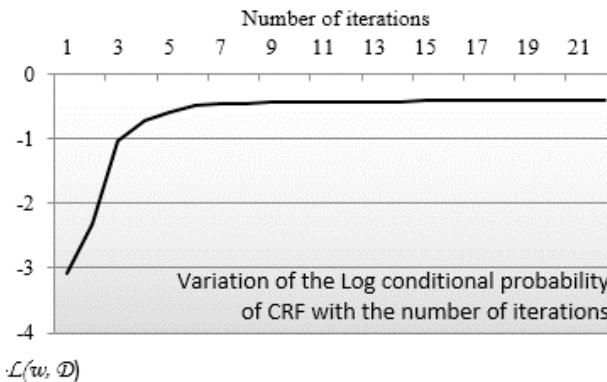
The three simple steps are as follows:

1. Instantiate the `config` configuration for the CRF (line 15)
2. Define the three `delims` delimiters to extract the tagged data (line 16)
3. Instantiate and train the CRF classifier, `crf` (line 17)

For these tests, the initial value of the weights (with respect to the maximum number of iterations for the maximization of the log likelihood and the convergence criteria) are set to 0.7 (with respect to 100 and 1e-3). The delimiters for labels sequence, observed features sequence, and the training set are customized for the format of `rating.raw` and `rating.tagged` input data files.

The training convergence profile

The first training run discovered 136 features from 34 analysts' stock recommendations. The algorithm converged after 21 iterations. The value of the log of the likelihood for each of those iterations is plotted to illustrate the convergence toward a solution of optimum w :



The visualization of the log conditional probability of a CRF during training

The training phase converges quickly toward a solution. It can be explained by the fact that there is little variation in the six-field format of the analyst's recommendations. A loose or free-style format would require a larger number of iterations during training to converge.

Impact of the size of the training set

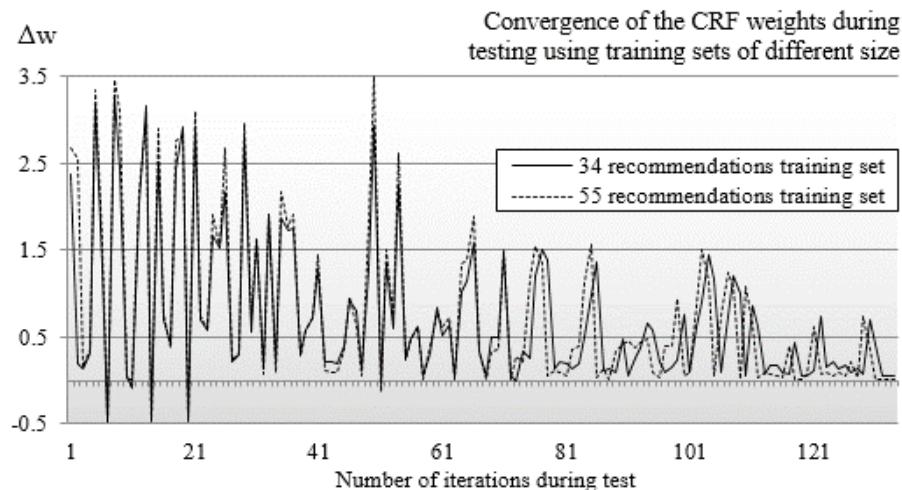
The second test evaluates the impact of the size of the training set on the convergence of the training algorithm. It consists of computing the difference Δw of the model parameters (weights) between two consecutive iterations $\{w_i\}_{t+1}$ and $\{w_i\}_t$:

$$\Delta w = \sum_{i=0}^{D-1} (w_i^{t+1} - w_i^t)$$

The test is run on 163 randomly chosen recommendations using the same model but with two different training sets:

- 34 analysts' stock recommendations
- 55 stock recommendations

The larger training set is a superset of the 34 recommendations' set. The following graph illustrates the comparison of features generated with 34 and 55 CRF training sequences:

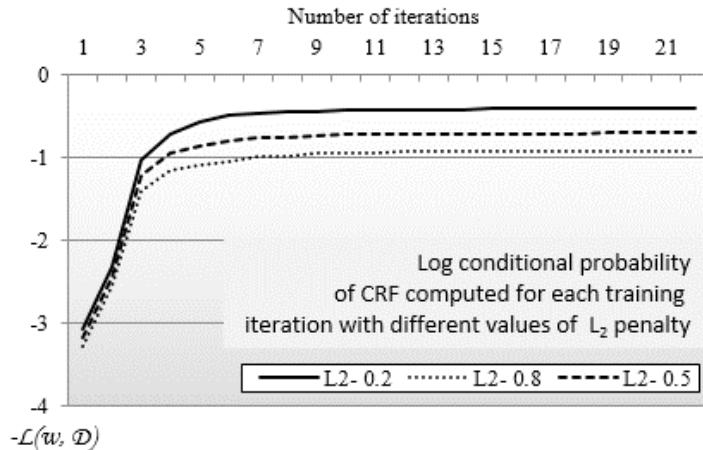


The convergence of the CRF weight using training sets of different sizes

The disparity between the test runs using two different sizes of training sets is very small. This can be easily explained by the fact that there is a small variation in the format between the analyst's recommendations.

Impact of the L_2 regularization factor

The third test evaluates the impact of the L_2 regularization penalty on the convergence toward the optimum weights/features. The test is similar to the first test with a different value of λ . The following chart plots $\log [p(Y|X, w)]$ for different values of $\lambda = 1/\sigma^2$ (0.2, 0.5, and 0.8):



The impact of the L_2 penalty on the convergence of the CRF training algorithm

The log of the conditional probability decreases or the conditional probability increases with the number of iterations. The lower the L_2 regularization factor, the higher the conditional probability.

The variation of the analysts' recommendations within the training set is small, which limits the risk of overfitting. A free-style recommendation format would have been more sensitive to overfitting.

Comparing CRF and HMM

The cost/benefit analysis of discriminative models relative to generative models applies to the comparison of the conditional random field with the hidden Markov model.

Contrary to the hidden Markov model, the conditional random field does not require the observations to be independent (conditional probability). The conditional random field can be regarded as a generalization of the HMM by extending the transition probabilities to arbitrary feature functions that can depend on the input sequence. The HMM assumes the transition probabilities matrix to be constant.

The HMM learns the transition probabilities a_{ij} on its own by processing more training data. The HMM can be regarded as a special case of CRF where the probabilities used in the state transition are constant.

Performance consideration

The time complexity for decoding and evaluating canonical forms of the hidden Markov model for N states and T observations is $O(N_2 T)$. The training of the HMM using the Baum-Welch algorithm is $O(N_2 TM)$, where M is the number of iterations.

There are several options to improve the performance of the HMM:

- Avoid unnecessary multiplication by 0 in the emission probabilities matrix by either using sparse matrices or tracking the null entries.
- Train the HMM on the most *relevant* subset of the training data. This technique can be particularly effective in the case of tagging of words or a bag of words in natural language processing.

The training of the linear chain conditional random fields is implemented using the same dynamic programming techniques as the HMM implementation (Viterbi, forward-backward passes, and so on). Its time complexity for training T data sequences, N labels (or expected outcomes), and M weights/features λ is $O(MTN_2)$.

The time complexity of the training of a CRF can be reduced by distributing the computation of the log likelihood and gradient over multiple nodes using a framework such as Akka or Apache Spark, as described in *Chapter 12, Scalable Frameworks* [7:13].

Summary

In this chapter, we had a closer look at modeling sequences of observations with hidden (or latent) states with the two commonly used algorithms:

- The generative hidden Markov model to maximize $p(X, Y)$
- The discriminative conditional random field to maximize $\log p(Y | X)$

The HMM is a special form of Bayes network. It requires the observations to be independent. Although restrictive, the conditional independence prerequisites make the HMM fairly easy to understand and validate, which is not the case for a CRF.

You learned how to implement three dynamic programming techniques: Viterbi, Baum-Welch, and alpha/beta algorithms in Scala. These algorithms are used to solve diverse type of optimization problems. They should be an essential component of your algorithmic tool box.

The conditional random field relies on the logistic regression to estimate the optimal weights of the model. Such a technique is also used in the multiple layer perceptron, which was introduced in *Chapter 9, Artificial Neural Network*. The next chapter introduces two important alternatives to the logistic regression for discriminating between observations: the Kernel function for nonlinear models and the maximization of the margin between classes of observations.

8

Kernel Models and Support Vector Machines

This chapter introduces kernel functions, binary support vectors classifiers, one-class support vector machines for anomaly detection, and support vector regression.

In the *Binomial classification* section in *Chapter 6, Regression and Regularization*, you learned the concept of hyperplanes to segregate observations from the training set and estimate the linear decision boundary. The logistic regression has at least one limitation: it requires that the datasets be linearly separated using a defined function (sigmoid). This limitation is especially an issue for high-dimension problems (large number of features that are highly nonlinearly dependent). **Support vector machines (SVMs)** overcome this limitation by estimating the optimal separating hyperplane using kernel functions.

In this chapter, we will cover the following topics:

- The impact of some of the SVM configuration parameters and the kernel method on the accuracy of the classification
- How to apply the binary support vector classifier to estimate the risk for a public company to curtail or eliminate its dividend
- How to detect outliers with a one-class support vector classifier
- How the support vector regression is compared to the linear regression

Support vector machines are formulated as a convex optimization problem. The mathematical foundation of the related algorithms is described in this chapter, for reference.

Kernel functions

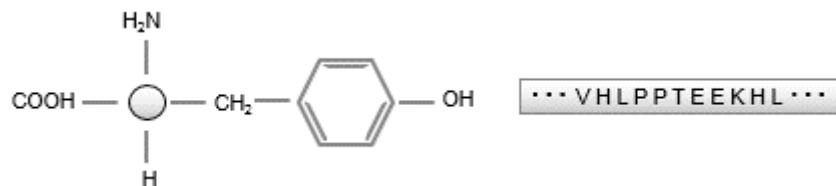
Every machine learning model introduced in this book so far assumes that observations are represented by a feature vector of a fixed size. However, some real-world applications such as text mining or genomics do not lend themselves to this restriction. The critical element of the process of classification is to define a similarity or distance between two observations. Kernel functions allow developers to compute the similarity between observations without the need to encode them in feature vectors [8:1].

An overview

The concept of kernel methods may be a bit odd at first to a novice. Let's consider the example of the classification of proteins. Proteins have different lengths and compositions, but they do not prevent scientists from classifying them [8:2].

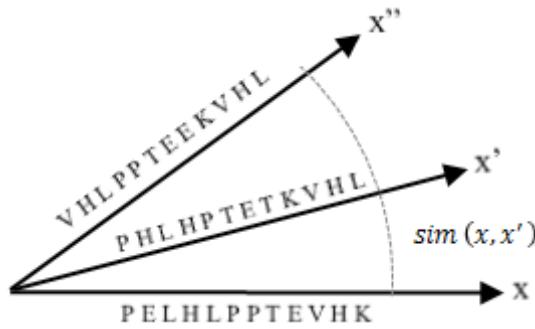
 **Proteins**
Proteins are polymers of amino acids joined together by peptide bonds. They are composed of a carbon atom bonded to a hydrogen atom, another amino acid, or a carboxyl group.

A protein is represented using a traditional molecular notation to which biochemists are familiar. Geneticists describe proteins in terms of a sequence of characters known as the **protein sequence annotation**. The sequence annotation encodes the structure and composition of the protein. The following image illustrates the molecular (left) and encoded (right) representation of a protein:



The sequence annotation of a protein

The classification and the clustering of a set of proteins require the definition of a similarity factor or distance used to evaluate and compare the proteins. For example, the similarity between three proteins can be defined as a normalized dot product of their sequence annotation:



The similarity between the sequence annotations of three proteins

You do not have to represent the entire sequence annotation of the proteins as a feature vector in order to establish that they belong to the same class. You only need to compare each element of each sequence, one by one, and compute the similarity. For the same reason, the estimation of the similarity does not require the two proteins to have the same length.

In this example, we do not have to assign a numerical value to each element of the annotation. Let's consider an element of the protein annotation as its character c and position p (for example, K, 4). The dot product of the two protein annotations x and x' of the respective lengths n and n' are defined as the number of identical elements (character and position) between the two annotations divided by the maximum length between the two annotations (**M1**):

$$\text{sim}(x_{cp}, x'_{c'p'}) = \frac{1}{\max(n, n')} \sum_{i=1}^{\min(n, n')} (c = c') \cap (p = p')$$

The computation of the similarity for the three proteins produces the result as $\text{sim}(x, x') = 6/12 = 0.50$, $\text{sim}(x, x'') = 3/13 = 0.23$, and $\text{sim}(x', x'') = 4/13 = 0.31$.

Another similar aspect is that the similarity of two identical annotations is 1.0 and the similarity of two completely different annotations is 0.0.

The visualization of similarity

It is usually more convenient to use a radial representation to visualize the similarity between features, as in the example of proteins' annotations. The distance $d(x, x') = 1/\text{sim}(x, x')$ is visualized as the angle or cosine between two features. The cosine metric is commonly used in text mining.

In this example, the similarity is known as a kernel function in the space of the sequence annotation of proteins.

Common discriminative kernels

Although the measure of similarity is very useful to understand the concept of a kernel function, kernels have a broader definition. A kernel $K(x, x')$ is a symmetric, nonnegative real function that takes two real arguments (values of two features). There are many different types of kernel functions, among which the most common are as follows:

- **The linear kernel (dot product):** This is useful in the case of very high-dimensional data where problems can be expressed as a linear combination of the original features.
- **The polynomial kernel:** This extends the linear kernel for a combination of features that are not completely linear.
- **The radial basis function (RBF):** This is the most commonly applied kernel. It is used where the labeled or target data is noisy and requires some level of regularization.
- **The sigmoid kernel:** This is used in conjunction with neural networks.
- **The Laplacian kernel:** This is a variant of RBF with a higher regularization impact on training data.
- **The log kernel:** This is used in image processing.

The RBF terminology

In this presentation and the library used in its implementation, the radial basis function is a synonym to the Gaussian kernel function. However, RBF also refers to the family of exponential kernel functions that encompasses Gaussian, Laplacian, and exponential functions.

The simple linear model for regression consists of the dot product of the regression parameters (weights) and the input data (refer to the *Ordinary least squares regression* section in *Chapter 6, Regression and Regularization*).

The model is, in fact, the linear combination of weights and linear combination of inputs. The concept can be extended by defining a general regression model as the linear combination of nonlinear functions, known as basis functions (**M2**):

$$f(x|w) = w_0 + \sum_{d=1}^D w_d \phi_d(x) \quad \phi_d: \mathbb{R} \rightarrow \mathbb{R}$$

The most commonly used basis functions are the power and Gaussian functions. The kernel function is described as the dot product of the two vectors of the basis function $\varphi(x) \cdot \varphi(x')$ of two features vectors x and x' . A partial list of kernel methods is as follows:

M3: The generic kernel function is defined as:

$$K(x, x') = \phi(x) \cdot \phi(x') = \sum_{d=1}^D \phi_d(x) \phi_d(x')$$

M4: The linear kernel is defined as:

$$K(x, x') = x^T x' = \sum_{d=1}^D x_i \cdot x'_i$$

M5: The polynomial kernel with the slope γ , degree n , and constant c is defined as:



$$K(x, x') = (\gamma x^T x' + c)^n \quad \gamma > 0, c \geq 0$$

M6: The sigmoid kernel with the slope γ and constant c is defined as:

$$K(x, x') = \tanh(\gamma x^T x' + c) \quad \gamma > 0, c \geq 0$$

M7: The radial basis function kernel with the slope γ is defined as:

$$K(x, x') = e^{-\gamma \|x - x'\|^2} \quad \gamma > 0$$

M8: The Laplacian kernel with the slope γ is defined as:

$$K(x, x') = e^{-\gamma \|x - x'\|} \quad \gamma > 0$$

M9: The log kernel with the degree n is defined as:

$$K(x, x') = -\log(1 + \|x - x'\|^n)$$

The list of discriminative kernel functions described earlier is just a subset of the kernel methods' universe. The other types of kernels include the following:

- **Probabilistic kernels:** These are kernels derived from generative models. Probabilistic models such as Gaussian processes can be used as a kernel function [8:3].
- **Smoothing kernels:** This is the nonparametric formulation, averaging density with the nearest neighbor observations [8:4].

- **Reproducible kernel Hilbert spaces:** This is the dot product of finite or infinite basis functions [8:5].

The kernel functions play a very important role in support vector machines for nonlinear problems.

Kernel monadic composition

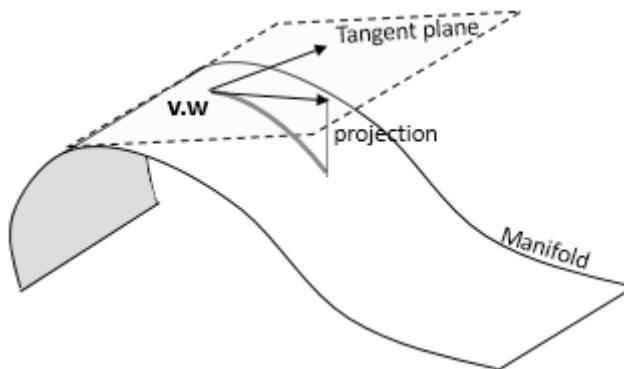
The concept of a kernel function is actually derived from differential geometry and more specifically from manifold, which was introduced in the *Non-linear models* section under *Dimension reduction* in *Chapter 4, Unsupervised Learning*.

A manifold is a low dimension features space embedded in the observation space of higher dimension. The dot (or inner) product of two observations, known as the **Riemann metric**, is computed on a Euclidean tangent space.

The heat kernel function

 The kernel function on a manifold is actually computed by solving the heat equation that uses the Laplace-Beltrami operator. The heat kernel is the solution of the heat differential equation. It associates the dot product with an exponential map.

The kernel function is the composition of the dot product on the tangent space projected on the manifold using an exponential map, as shown in the following diagram:



The visualization of a manifold, Riemann metric, and projection of an inner product

A kernel function is the composition $g \circ f$ of two functions:

- A function h that implements the Riemann metric or similarity between two vectors v and w
- A function g that implements the projection of the similarity $h(v, w)$ to the manifold (exponential map)

The `KF` class implements the kernel function as a composition of the functions g and h :

```
type F1 = Double => Double
type F2 = (Double, Double) => Double

case class KF[G](val g: G, h: F2) {
    def metric(v: DblVector, w: DblVector)
        (implicit gf: G => F1): Double = //1
        g(v.zip(w).map{ case(_v, _w) => h(_v, _w) }.sum) //2
}
```

The `KF` class is parameterized with a `G` type that can be converted to `Function1[Double, Double]`. Therefore, the computation of `metric` (dot product) requires an implicit conversion from `G` to `Function1` (line 1). The `metric` is computed by zipping the two vectors, mapping the `h` similarity function, and summing up the resulting vector (line 2).

Let's define the monadic composition for the `KF` class:

```
val kfMonad = new _Monad[KF] {
    override def map[G, H](kf: KF[G])(f: G => H): KF[H] =
        KF[H](f(kf.g), kf.h) //3
    override def flatMap[G, H](kf: KF[G])(f: G => KF[H]): KF[H] =
        KF[H](f(kf.g).g, kf.h)
}
```

The creation of the `kfMonad` instance overrides the `map` and `flatMap` methods defined in the generic `_Monad` trait, as described in the *Monads* section in *Chapter 1, Getting Started*. The implementation of the `unit` method is not essential to the monadic composition and it is, therefore, omitted.

The function argument of the `map` and `flatMap` methods applies only to the exponential map function g (line 3). The composition of two kernel functions $kf1 = g1 \circ h$ and $kf2 = g2 \circ h$ produces a kernel function $kf3 = g2 \circ (g1 \circ h) = (g2 \circ g1) \circ h = g3 \circ h$.



Interpretation of kernel functions' monadic composition

The visualization of the monadic composition of kernel functions on the manifold is quite intuitive. The composition of two kernel functions consists of composing their respective projections or exponential map functions g . The function g is directly related to the curvature of the manifold around the data point for which the metric is computed. The monadic composition of the kernel functions attempts to adjust the exponential map to fit the curvature of the manifold.

The next step is to define an implicit class to convert a kernel function of the `KF` type to its monadic representation so that it can access the `map` and `flatMap` methods (line 4):

```
implicit class kf2Monad[G](kf: KF[G]) { //4
    def map[H](f: G => H): KF[H] = kfMonad.map(kf)(f)
    def flatMap[H](f: G => KF[H]): KF[H] = kfMonad.flatMap(kf)(f)
}
```

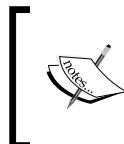
Let's implement the RBF radial basis function and the polynomial kernel function, `Polynomial`, by defining their respective g and h functions. The parameterized type for the kernel function is simply `Function1[Double, Double]`:

```
class RBF(s2: Double) extends KF[F1] {
    (x: Double) => Math.exp(-0.5*x*x/s2),
    (x: Double, y: Double) => x - y
}
class Polynomial(d: Int) extends KF[F1] {
    (x: Double) => Math.pow(1.0+x, d),
    (x: Double, y: Double) => x*y
}
```

Here is an example of the composition of two kernel functions: a `kf1` kernel RBF with a standard deviation of 0.6 (line 5) and a `kf2` polynomial kernel with a degree 3 (line 6):

```
val v = Vector[Double](0.5, 0.2, 0.3)
val w = Vector[Double](0.1, 0.7, 0.2)
val composed = for {
    kf1 <- new RBF(0.6) //5
    kf2 <- new Polynomial(3) //6
} yield kf2
composed.metric(v, w) //7
```

Finally, the `metric` is computed on the composed kernel functions (line 7).



Kernel functions in SVM

Our implementation of the support vector machine uses the kernel function included in the LIBSVM library.



Support vector machines

A support vector machine is a linear discriminative classifier that attempts to maximize the margin between classes during training. This approach is similar to the definition of a hyperplane through the training of the logistic regression (refer to the *Binomial classification* section in *Chapter 6, Regression and Regularization*). The main difference is that the support vector machine computes the optimum separating hyperplane between groups or classes of observations. The hyperplane is indeed the equation that represents the model generated through training.

The quality of the SVM depends on the distance, known as margin, between the different classes of observations. The accuracy of the classifier increases as the margin increases.

The linear SVM

First, let's apply the support vector machine to extract a linear model (classifier or regression) for a labeled set of observations. There are two scenarios for defining a linear model. The labeled observations are as follows:

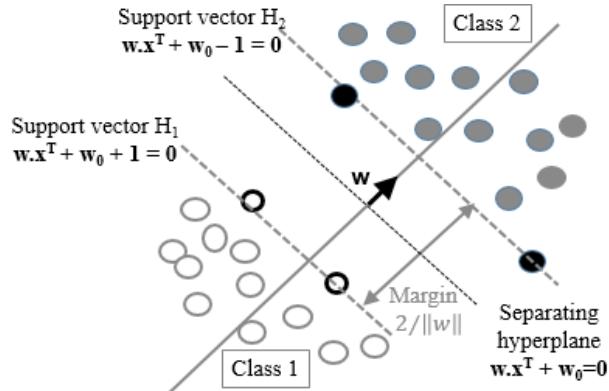
- They are naturally segregated in the features space (the **separable** case)
- They are intermingled with overlap (the **nonseparable** case)

It is easy to understand the concept of an optimal separating hyperplane in cases where the observations are naturally segregated.

The separable case – the hard margin

The concept of separating a training set of observations with a hyperplane is explained in a better way with a two-dimensional (x, y) set of observations with two classes C_1 and C_2 . The label y has the value -1 or +1.

The equation for the separating hyperplane is defined by the linear equation $y=w \cdot x + w_0$, which sits in the midpoint between the boundary data points for the class C_1 ($H_1: w \cdot x^T + w_0 + 1=0$) and class C_2 ($H_2: w \cdot x^T + w_0 - 1=0$). The planes H_1 and H_2 are the support vectors:



The visualization of the hard margin in the support vector machine

In the separable case, the support vectors fully segregate the observations into two distinct classes. The margin between the two support vectors is the same for all the observations and is known as the **hard margin**.

The separable case

M1: The support vectors equation w is represented as:

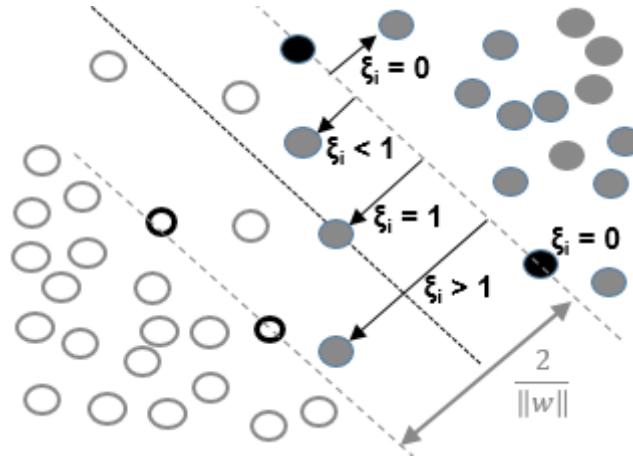
$$y_i(w^T x + w_0) \geq 1 \quad \forall i$$

M2: The hard margin optimization problem is given by:

$$\min_{w, w_0} \left\{ \frac{w^T w}{2} \right\} \text{ subject to } y_i(w^T x + w_0) \geq 1 \quad \forall i$$

The nonseparable case – the soft margin

In the nonseparable case, the support vectors cannot completely segregate observations through training. They merely become linear functions that penalize the few observations or outliers that are located outside (or beyond) their respective support vector H_1 or H_2 . The penalty variable ξ , also known as the slack variable, increases if the outlier is further away from the support vector:



The visualization of the hard margin in the support vector machine

The observations that belong to the appropriate (or own) class do not have to be penalized. The condition is similar to the hard margin, which means that the slack ξ is null. This technique penalizes the observations that belong to the class but are located beyond their support vectors; the slack ξ increases as the observations get closer to the support vector of the other class and beyond. The margin is then known as a soft margin because the separating hyperplane is enforced through a slack variable.

The nonseparable case

M3: The optimization of the soft margin for a linear SVM with C formulation is defined as:



$$\min_{w, \xi} \left\{ \frac{w^T w}{2} + C \sum_{i=0}^{n-1} \xi_i \right\}$$

$$\xi_i \geq 0, \quad y_i (w^T x + w_0) \geq 1 - \xi_i \quad \forall i$$

Here, C is the penalty (or inversed regularization) factor.

You may wonder how the minimization of the margin error is related to the loss function and the penalization factor, introduced for the ridge regression (refer to the *Numerical optimization* section in *Chapter 6, Regression and Regularization*). The second factor in the formula corresponds to the ubiquitous loss function. You will certainly be able to recognize the first term as the L2 regularization penalty with $\lambda = 1/2C$.

The problem can be reformulated as the minimization of a function known as the **primal problem** [8:6].

M4: The primal problem formulation of the support vector classifier using the L_2 regularization is as follows:

$$\min_{w, w_0} \left\{ \frac{w^T w}{2} + C \sum_{i=0}^{n-1} \mathcal{L}_i \right\} \quad \mathcal{L}_i = |1 - y_i(w^T x + w_0)|$$

The C penalty factor is the inverse of the L_2 regularization factor. The loss function L is known as the **hinge loss**. The formulation of the margin using the C penalty (or cost) parameter is known as the **C-SVM** formulation. C-SVM is sometimes called the **C-Epsilon SVM** formulation for the nonseparable case.

The **v-SVM** (or Nu-SVM) is an alternative formulation to C-SVM. The formulation is more descriptive than C-SVM; v represents the upper bound of the training observations that are poorly classified and the lower bound of the observations on the support vectors [8:7].

M5: The **v-SVM** formulation of a linear SVM using the L_2 regularization is defined as:

$$\begin{aligned} & \min_{w, \rho, \xi} \left\{ \frac{w^T w}{2} - \rho + \frac{1}{vn} \sum_{i=0}^{n-1} \xi_i \right\} \\ & \xi_i \geq 0, \quad y_i(w^T x + w_0) \geq \rho - \xi_i \quad \forall i \end{aligned}$$

Here, ρ is a margin factor used as an optimization variable.

The C-SVM formulation is used throughout the chapters for the binary, one class support vector classifier as well as the support vector regression.

Sequential Minimal Optimization

The optimization problem consists of the minimization of a quadratic objective function (w^2) subject to N linear constraints, N being the number of observations. The time complexity of the algorithm is $O(N^3)$. A more efficient algorithm known as **Sequential Minimal Optimization (SMO)** has been introduced to reduce the time complexity to $O(N^2)$.

The nonlinear SVM

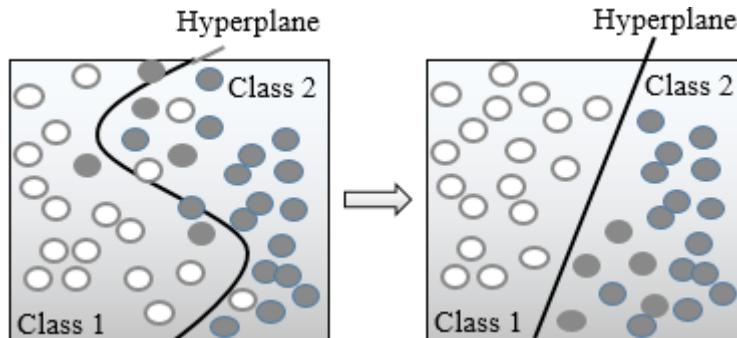
So far, we assumed that the separating hyperplane and therefore the support vectors are linear functions. Unfortunately, such assumptions are not always correct in the real world.

Max-margin classification

Support vector machines are known as large or **maximum margin classifiers**. The objective is to maximize the margin between the support vectors with hard constraints for separable (similarly, soft constraints with slack variables for nonseparable) cases.

The model parameters $\{w_i\}$ are rescaled during optimization to guarantee that the margin is at least 1. Such algorithms are known as maximum (or large) margin classifiers.

The problem of fitting a nonlinear model into the labeled observations using support vectors is not an easy task. A better alternative consists of mapping the problem to a new and higher dimensional space using a nonlinear transformation. The nonlinear separating hyperplane becomes a linear plane in the new space, as illustrated in the following diagram:



An illustration of the kernel trick in the SVM

The nonlinear SVM is implemented using a basis function $\square(x)$. The formulation of the nonlinear C-SVM is very similar to the linear case. The only difference is the constraint along with the support vector, using the basis function φ (**M6**):

$$y_i(w^T \phi(x) + w_0) \geq 1 - \xi_i \quad \xi_i \geq 0 \quad \forall i$$

The minimization of $w^T \cdot \phi(x)$ in the preceding equation requires the computation of the inner product $\phi(x)^T \cdot \phi(x)$. The inner product of the basis functions is implemented using one of the kernel functions introduced in the first section. The optimization of the preceding convex problem computes the optimal hyperplane w^* as the kernelized linear combination of the training samples $y_i \phi(x_i)$ and **Lagrange multipliers**. This formulation of the optimization problem is known as the **SVM dual problem**. The description of the dual problem is mentioned as a reference and is well beyond the scope of this book [8:8].

M7: The optimal hyperplane for the SVM dual problem is defined as:

$$w^* = \sum_{i=0}^{n-1} \alpha_i y_i \phi(x_i)$$

M8: The hard margin formulation for the SVM dual problem is defined as:

$$y_i (w^T \cdot \phi(x) + w_0) = y_i \left(\sum_{i=0}^{n-1} \alpha_i y_i K(x_i, x) + w_0 \right) \geq 1$$

$$K(x_i, x) = \phi(x_i) \cdot \phi(x) \quad \forall i$$

The kernel trick

The transformation $(x, x') \Rightarrow K(x, x')$ maps a nonlinear problem into a linear problem in a higher dimensional space. It is known as the **kernel trick**.

Let's consider, for example, the polynomial kernel defined in the first section with a degree $d = 2$ and coefficient of $C0 = 1$ in a two-dimension space. The polynomial kernel function of two vectors, $x = [x_1, x_2]$ and $z = [x'_1, x'_2]$, is decomposed into a linear function in a 6 dimension space:

$$\begin{aligned} K(x, x') &= (1 + x^T x')^2 \\ &= 1 + 2x_1 x'_1 + 2x_2 x'_2 + 2x_1 x'_1 x_2 x'_2 + (x_1 x'_1)^2 + (x_2 x'_2)^2 \\ &= \phi_1(x) \cdot \phi_1(x') + \phi_2(x) \cdot \phi_2(x') + \phi_3(x) \cdot \phi_3(x') + \dots \\ \phi_2(x) &= 1, \phi_2(x) = \sqrt{2}x_1, \phi_3(x) = \sqrt{2}x_2, \phi_4(x) = x_1^2 \dots \end{aligned}$$

Support vector classifiers – SVC

Support vector machines can be applied to classification, anomalies detection, and regression problems. Let's first dive into the support vector classifiers.

The binary SVC

The first classifier to be evaluated is the binary (2-class) support vector classifier. The implementation uses the LIBSVM library created by Chih-Chung Chang and Chih-Jen Lin from the National Taiwan University [8:9].

LIBSVM

The library was originally written in C before being ported to Java. It can be downloaded from <http://www.csie.ntu.edu.tw/~cjlin/libsvm> as a .zip or tar.gz file. The library includes the following classifier modes:

- Support vector classifiers (C-SVC, ν-SVC, and one-class SVC)
- Support vector regression (ν-SVR and ε-SVR)
- RBF, linear, sigmoid, polynomial, and precomputed kernels

LIBSVM has the distinct advantage of using **Sequential Minimal Optimization (SMO)**, which reduces the time complexity of a training of n observations to $O(n^2)$. The LIBSVM documentation covers both the theory and implementation of hard and soft margins and is available at <http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf>.

Why LIBSVM?

There are alternatives to the LIBSVM library for learning and experimenting with SVM. David Soergel from the University of Berkeley refactored and optimized the Java version [8:10]. Thorsten Joachims' **SVMLight** [8:11] Spark/MLLib 1.0 includes two Scala implementations of the SVM using resilient distributed datasets (refer to the *Apache Spark* section in *Chapter 12, Scalable Frameworks*). However, LIBSVM is the most commonly used SVM library.

The implementation of the different support vector classifiers and the support vector regression in LIBSVM is broken down into the following five Java classes:

- `svm_model`: This defines the parameters of the model created during training
- `svm_node`: This models the element of the sparse matrix Q , which is used in the maximization of the margins
- `svm_parameters`: This contains the different models for support vector classifiers and regressions, the five kernels supported in LIBSVM with their parameters, and the `weights` vectors used in cross-validation
- `svm_problem`: This configures the input to any of the SVM algorithm (the number of observations, input vector data x as a matrix, and the vector of labels y)
- `svm`: This implements algorithms used in training, classification, and regression

The library also includes template programs for training, prediction, and normalization of datasets.

The LIBSVM Java code

The Java version of LIBSVM is a direct port of the original C code. It does not support generic types and is not easily configurable (the code uses switch statements instead of polymorphism). For all its limitations, LIBSVM is a fairly well-tested and robust Java library for SVMs.

Let's create a Scala wrapper to the LIBSVM library to improve its flexibility and ease of use.

Design

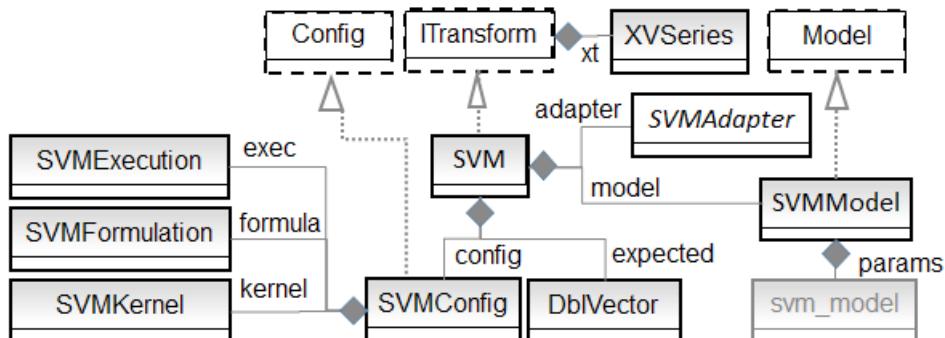
The implementation of the support vector machine algorithm uses the design template for classifiers (refer to the *Design template for classifier* section in the *Appendix A, Basic Concepts*).

The key components of the implementation of a SVM are as follows:

- A model, `SVMModel`, of the `Model` type is initialized through training during the instantiation of the classifier. The model class is an adapter to the `svm_model` structure defined in LIBSVM.
- An `SVMAdapter` object interfaces with the internal LIBSVM data structures and methods.

- The SVM support vector machine class is implemented as an implicit data transformation of the `ITransform` type. It has three parameters: the configuration wrapper of the `SVMConfig` type, the features/time series of the `XVSeries` type, and the target or labeled values, `DblVector`.
- The configuration (the `SVMConfig` type) consists of three distinct elements: `SVMExecution` that defines the execution parameters such as the maximum number of iterations or convergence criteria, `SVMKernel` that specifies the kernel function used during training, and `SVMFormulation` that defines the formula (C , ϵ , or ν) used to compute a nonseparable case for the support vector classifier and regression.

The key software components of the support vector machine are described in the following UML class diagram:



The UML class diagram for the support vector machine

The UML diagram omits the helper traits and classes such as `Monitor` or the Apache Commons Math components.

Configuration parameters

LIBSVM exposes a large number of parameters for the configuration and execution of any of the SVM algorithms. Any SVM algorithm is configured with three categories of parameters, which are as follows:

- Formulation (or type) of the SVM algorithms (the multiclass classifier, one-class classifier, regression, and so on) using the `SVMFormulation` class
- The kernel function used in the algorithm (the RBF kernel, Sigmoid kernel, and so on) using the `SVMKernel` class
- Training and executing parameters (the convergence criteria, number of folds for cross-validation, and so on) using the `SVMExecution` class

The SVM formulation

The instantiation of the configuration consists of initializing the `param LIBSVM` parameter by the SVM type, kernel, and the execution context selected by the user.

Each of the SVM parameters' case class extends the generic `SVMConfigItem` trait:

```
trait SVMConfigItem { def update(param: svm_parameter): Unit }
```

The classes inherited from `SVMConfigItem` are responsible for updating the list of the SVM parameters, `svm_parameter`, defined in LIBSVM. The `update` method encapsulates the configuration of LIBSVM.

The formulation of the SVM algorithm by a class hierarchy with `SVMFormulation` as the base trait is as follows:

```
sealed trait SVMFormulation extends SVMConfigItem {
    def update(param: svm_parameter): Unit
}
```

The list of the formulation for the SVM (`C`, `nu`, and `eps` for regression) is completely defined and known. Therefore, the hierarchy should not be altered and the `SVMFormulation` trait has to be declared sealed. Here is an example of the SVM `CSVCFormulation` formulation class, which defines the C-SVM model:

```
class CSVCFormulation (c: Double) extends SVMFormulation {
    override def update(param: svm_parameter): Unit = {
        param.svm_type = svm_parameter.C_SVC
        param.C = c
    }
}
```

The other SVM `NusVCFormulation`, `OneSVCFormulation`, and `SVRFomulation` formulation classes implement the ν-SVM, 1-SVM, and ε-SVM, respectively for regression models.

The SVM kernel function

Next, you need to specify the kernel functions by defining and implementing the `SVMKernel` trait:

```
sealed trait SVMKernel extends SVMConfigItem {
    override def update(param: svm_parameter): Unit
}
```

Once again, there are a limited number of kernel functions supported in LIBSVM. Therefore, the hierarchy of kernel functions is sealed. The following code snippet configures the radius basis function kernel, `RbfKernel`, as an example of the definition of the kernel definition class:

```
class RbfKernel(gamma: Double) extends SVMKernel {
    override def update(param: svm_parameter): Unit = {
        param.kernel_type = svm_parameter.RBF
        param.gamma = gamma
    }
}
```

The fact that the LIBSVM Java byte code library is not very extensible does not prevent you from defining a new kernel function in the LIBSVM source code. For example, the Laplacian kernel can be added by performing the following steps:

1. Create a new kernel type in `svm_parameter`, such as `svm_parameter.LAPLACE = 5`.
2. Add the kernel function name to `kernel_type_table` in the `svm` class.
3. Add `kernel_type != svm_parameter.LAPLACE` to the `svm_check_parameter` method.
4. Add the implementation of the kernel function to two values in `svm: kernel_function` (Java code):

```
case svm_parameter.LAPLACE:
    double sum = 0.0;
    for(int k = 0; k < x[i].length; k++) {
        final double diff = x[i][k].value - x[j][k].value;
        sum += diff*diff;
    }
    return Math.exp(-gamma*Math.sqrt(sum));
```

5. Add the implementation of the Laplace kernel function in the `svm.k_function` method by modifying the existing implementation of RBF (`distanceSqr`).
6. Rebuild the `libsvm.jar` file

The SVM execution

The `SVMEexecution` class defines the configuration parameters for the execution of the training of the model, namely the `eps` convergence factor for the optimizer (line 2), the size of the cache, `cacheSize` (line 1), and the number of folds, `nFolds`, used during cross-validation:

```
class SVMEexecution(cacheSize: Int, eps: Double, nFolds: Int)
    extends SVMConfigItem {
```

```
    override def update(param: svm_parameter): Unit = {
      param.cache_size = cacheSize //1
      param.eps = eps //2
    }
}
```

The cross-validation is performed only if the `nFolds` value is greater than 1.

We are finally ready to create the `SVMConfig` configuration class, which hides and manages all of the different configuration parameters:

```
class SVMConfig(formula: SVMFormulation, kernel: SVMKernel,
                exec: SVMExecution) {
  val param = new svm_parameter
  formula.update(param) //3
  kernel.update(param) //4
  exec.update(param) //5
}
```

The `SVMConfig` class delegates the selection of the formula to the `SVMFormulation` class (line 3), selection of the kernel function to the `SVMKernel` class (line 4), and the execution of parameters to the `SVMExecution` class (line 5). The sequence of update calls initializes the LIBSVM list of configuration parameters.

Interface to LIBSVM

We need to create an adapter object to encapsulate the invocation to LIBSVM. The `SVMAdapter` object hides the LIBSVM internal data structures: `svm_model` and `svm_node`:

```
object SVMAdapter {
  type SVMNodes = Array[Array[svm_node]]
  class SVMProblem(numObs: Int, expected: DblArray) //6

  def createSVMNode(dim: Int, x: DblArray): Array[svm_node] //7
  def predictSVM(model: SVMModel, x: DblArray): Double //8
  def crossValidateSVM(problem: SVMProblem, //9
                       param: svm_parameter, nFolds: Int, expected: DblArray)
  def trainSVM(problem: SVMProblem, //10
               param: svm_parameter): svm_model
}
```

The `SVMAdapter` object is a single entry point to LIBSVM for training, validating a SVM model, and executing predictions:

- `SVMPProblem` wraps the definition of the training objective or problem in LIBSVM, using the labels or expected values (line 6)
- `createSVMNode` creates a new computation node for each observation x (line 7)
- `predictSVM` predicts the outcome of a new observation x given a model, `svm_model`, generated through training (line 8)
- `crossValidateSVM` validates the model, `svm_model`, with the `nFold` training-validation sets (line 9)
- `trainSVM` executes the `problem` training configuration (line 10)

svm_node

The LIBSVM `svm_node` Java class is defined as a pair of indices of the feature in the observation array and its value:



```
public class svm_node implements java.io.Serializable {
    public int index;
    public double value;
}
```

The `SVMAdapter` methods are described in the next section.

Training

The model for the SVM is defined by the following two components:

- `svm_model`: This is the SVM model parameters defined in LIBSVM
- `accuracy`: This is the accuracy of the model computed during cross-validation

The code will be as follows:

```
case class SVMModel(val svmmode1: svm_node,
                    val accuracy: Double) extends Model {
    lazy val residuals: DblArray = svmmode1.sv_coef(0)
}
```

The `residuals`, that is, $r = y - f(x)$ are computed in the LIBSVM library.



Accuracy in the SVM model

You may wonder why the value of the accuracy is a component of the model. The accuracy component of the model provides the client code with a quality metric associated with the model. Integrating the accuracy into the model, allows the user to make informed decisions in accepting or rejecting the model. The accuracy is stored in the model file for subsequent analysis.

Next, let's create the first support vector classifier for the two-class problems. The SVM class implements the `ITransform` monadic data transformation that implicitly generates a model from a training set, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 11).

The constructor for the SVM follows the template described in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*:

```
class SVM[T <% Double] (config: SVMConfig, xt: XVSeries[T],
  expected: DblVector) extends ITransform[Array[T]](xt) { //11

  type V = Double    //12
  val normEPS = config.eps*1e-7   //13
  val model: Option[SVMModel] = train //14

  def accuracy: Option[Double] = model.map(_.accuracy) //15
  def mse: Option[Double] //16
  def margin: Option[Double] //17
}
```

The implementation of the `ITransform` abstract class requires the definition of the output value of the predictor as a `Double` (line 12). The `normEPS` is used for rounding errors in the computation of the margin (line 13). The model of the `SVMModel` type is generated through training by the `svm` constructor (line 14). The last four methods are used to compute the parameters of the accuracy model (line 15), the mean square of errors, `mse`, (line 16), and the `margin` (line 17).

Let's take a look at the training method, `train`:

```
def train: Option[SVMModel] = Try {
  val problem = new SVMProblem(xt.size, expected.toArray) //18
  val dim = dimension(xt)

  xt.zipWithIndex.foreach{ case (_x, n) => //19
    problem.update(n, createSVMNode(dim, _x))
  }
```

```
new SVMModel(trainSVM(problem, config.param),  
            accuracy(problem)) //20  
}.toOption("SVM training failed", logger)
```

The `train` method creates `SVMProblem` that provides LIBSVM with the training components (line 18). The purpose of the `SVMProblem` class is to manage the definition of training parameters implemented in LIBSVM, as follows:

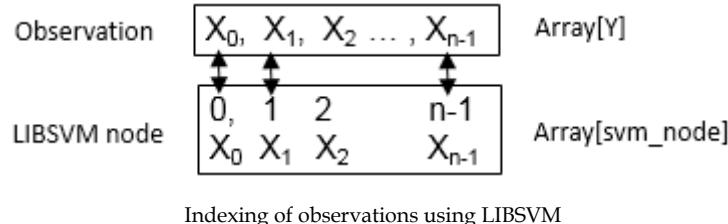
```
class SVMProblem(numObs: Int, expected: DblArray) {  
    val problem = new svm_problem //21  
    problem.l = numObs  
    problem.y = expected  
    problem.x = new SVMNodes(numObs)  
  
    def update(n: Int, node: Array[svm_node]): Unit =  
        problem.x(n) = node //22  
}
```

The arguments of the `SVMProblem` constructor, the number of observations, and the labels or expected values are used to initialize the corresponding `svm_problem` data structure in LIBSVM (line 21). The `update` method maps each observation, which is defined as an array of `svm_node` to the problem (line 22).

The `createSVMNode` method creates an array of `svm_node` from an observation. A `svm_node` in LIBSVM is the pair of the `j` index of a feature in an observation (line 23) and its value, `y` (line 24):

```
def createSVMNode(dim: Int, x: DblArray): Array[svm_node] = {  
    val newNode = new Array[svm_node](dim)  
    x.zipWithIndex.foreach{ case (y, j) => {  
        val node = new svm_node  
        node.index= j //23  
        node.value = y //24  
        newNode(j) = node  
    }}  
    newNode
```

The mapping between an observation and a LIBSVM node is illustrated in the following diagram:



The `trainsvm` method pushes the training request with a well-defined problem and configuration parameters to LIBSVM by invoking the `svm_train` method (line 26):

```
def trainSVM(problem: SVMProblem,
            param: svm_parameter): svm_model =
    svm.svm_train(problem.problem, param) //26
```

The accuracy is the ratio of the true positive plus the true negative over the size of the test sample (refer to the *Key quality metrics* section in *Chapter 2, Hello World!*). It is computed through cross-validation only if the number of folds initialized in the `SVMExecution` configuration class is greater than 1. Practically, the accuracy is computed by invoking the cross-validation method, `svm_cross_validation`, in the LIBSVM package, and then computing the ratio of the number of predicted values that match the labels over the total number of observations:

```
def accuracy(problem: SVMProblem): Double = {
    if( config.isCrossValidation ) {
        val target = new Array[Double](expected.size)
        crossValidateSVM(problem, config.param, //27
                         config.nFolds, target)

        target.zip(expected)
            .filter{case(x, y) =>Math.abs(x- y) < config.eps} //28
            .size.toDouble/expected.size
    }
    else 0.0
}
```

The call to the `crossValidateSVM` method of `SVMAdapter` forwards the configuration and execution of the cross validation with `config.nFolds` (line 27):

```
def crossValidateSVM(problem: SVMProblem, param: svm_parameter,
                     nFolds: Int, expected: DblArray) {
    svm.svm_cross_validation(problem.problem, param,
                            nFolds, expected)
}
```

The Scala `filter` weeds out the observations that were poorly predicted (line 28). This minimalist implementation is good enough to start exploring the support vector classifier.

Classification

The implementation of the `|>` classification method for the `SVM` class follows the same pattern as the other classifiers. It invokes the `predictSVM` method in `SVMAdapter` that forwards the request to LIBSVM (line 29):

```
override def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if(x.size == dimension(xt) && isModel) =>
    Try( predictSVM(model.get.svmmodel, x) ) //29
}
```

C-penalty and margin

The first evaluation consists of understanding the impact of the penalty factor C on the margin in the generation of the classes. Let's implement the computation of the margin. The margin is defined as $2/\|w\|$ and implemented as a method of the `SVM` class, as follows:-

```
def margin: Option[Double] =
  if(isModel) {
    val wNorm = model.get.residuals./:(0.0)((s,r) => s + r*r)
    if(wNorm < normEPS) None else Some(2.0/Math.sqrt(wNorm))
  }
  else None
```

The first instruction computes the sum of the squares, `wNorm`, of the residuals $r = y - f(x|w)$. The margin is ultimately computed if the sum of squares is significant enough to avoid rounding errors.

The margin is evaluated using an artificially generated time series and labeled data. First, we define the method to evaluate the margin for a specific value of the penalty (inversed regularization coefficient) factor C :

```
val GAMMA = 0.8
val CACHE_SIZE = 1<<8
val NFOLDS = 1
val EPS = 1e-5

def evalMargin(features: Vector[DblArray],
  expected: DblVector, c: Double): Int = {
  val execEnv = SVMExecution(CACHE_SIZE, EPS, NFOLDS)
```

```
val config = SVMConfig(new CSVCFormulation(c),
    new RbfKernel(GAMMA), execEnv)
val svc = SVM[Double](config, features, expected)
svc.margin.map(_.toString)      //30
}
```

The `evalMargin` method uses the `CACHE_SIZE`, `EPS`, and `NFOLDS` execution parameters. The execution displays the value of the margin for different values of C (line 30). The method is invoked iteratively to evaluate the impact of the penalty factor on the margin extracted from the training of the model. The test uses a synthetic time series to highlight the relation between C and the margin. The synthetic time series created by the `generate` method consists of two training sets of an equal size, N :

- Data points generated as $y = x(1 + r/5)$ for the label 1, r being a randomly generated number over the range $[0,1]$ (line 31)
- A randomly generated data point $y = r$ for the label -1 (line 32)

Consider the following code:

```
def generate: (Vector[DblArray], DblArray) = {
    val z = Vector.tabulate(N)(i => {
        val ri = i*(1.0 + 0.2*Random.nextDouble)
        Array[Double](i, ri) //31
    }) ++
    Vector.tabulate(N)(i => Array[Double](i, i*Random.nextDouble))
    (z, Array.fill(N)(1) ++ Array.fill(N)(-1)) //32
}
```

The `evalMargin` method is executed for different values of C ranging from 0 to 5:

```
generate.map(y =>
    (0.1 until 5.0 by 0.1)
        .flatMap(evalMargin(y._1, y._2, _)).mkString("\n")
)
```

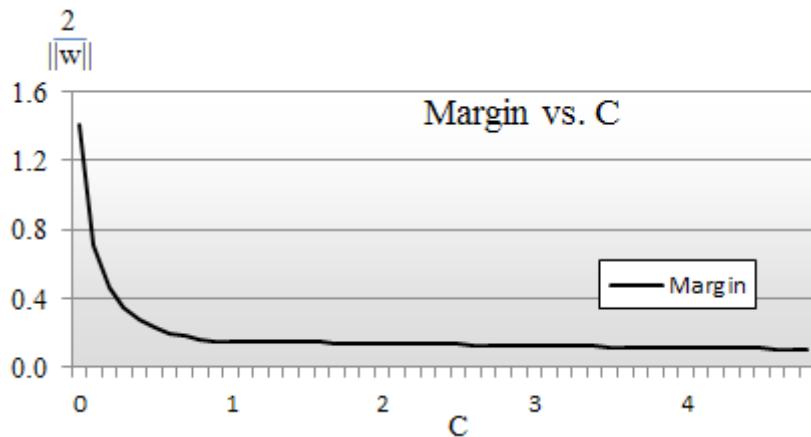
val versus final val

There is a difference between a val and a final val. A nonfinal value can be overridden in a subclass. Overriding a final value produces a compiler error, as follows:



```
class A { val x = 5;  final val y = 8 }
class B extends A {
    override val x = 9 // OK
    override val y = 10 // Error
}
```

The following chart illustrates the relation between the penalty or cost factor C and the margin:



The margin value versus the C -penalty factor for a support vector classifier

As expected, the value of the margin decreases as the penalty term C increases. The C penalty factor is related to the L_2 regularization factor λ as $C \sim 1/\lambda$. A model with a large value of C has a high variance and a low bias, while a small value of C will produce lower variance and a higher bias.

Optimizing C penalty

The optimal value for C is usually evaluated through cross-validation, by varying C in incremental powers of 2: $2^n, 2^{n+1}, \dots$ [8:12].

Kernel evaluation

The next test consists of comparing the impact of the kernel function on the accuracy of the prediction. Once again, a synthetic time series is generated to highlight the contribution of each kernel. The test code uses the runtime prediction or classification method, `|>`, to evaluate the different kernel functions. Let's create a method to evaluate and compare these kernel functions. All we need is the following (line 33):

- An `xt` training set of the `Vector[DblArray]` type
- A test set, `test`, of the `Vector[DblArray]` type
- A set of `labels` for the training set that takes the value 0 or 1
- A `kF` kernel function

Consider the following code:

```
val C = 1.0
def evalKernel(xt: Vector[DblArray], test: Vector[DblArray],
               labels: DblVector, kF: SVMKernel): Double = { //33

    val config = SVMConfig(new CSVCFormulation(C), kF) //34
    val svc = SVM[Double](config, xt, labels)
    val pfnSvc = svc |> //35
    test.zip(labels).count{case(x, y) =>pfnSvc(x).get == y}
        .toDouble/test.size //36
}
```

The `config` configuration of the SVM uses the `C` penalty factor 1, the `C`-formulation, and the default execution environment (line 34). The predictive `pfnSvc` partial function (line 35) is used to compute the predictive values for the test set. Finally, the `evalKernel` method counts the number of successes for which the predictive values match the labeled or expected values. The accuracy is computed as the ratio of the successful prediction over the size of the test sample (line 36).

In order to compare the different kernels, let's generate three datasets of the size $2N$ for a binomial classification using the pseudo-random `genData` data generation method:

```
def genData(variance: Double, mean: Double): Vector[DblArray] = {
    val rGen = new Random(System.currentTimeMillis)
    Vector.tabulate(N) (_ => {
        rGen.setSeed(rGen.nextLong)
        Array[Double](rGen.nextDouble, rGen.nextDouble)
            .map(variance*_ - mean) //37
    })
}
```

The random value is computed through a transformation $f(x) = variance * x + mean$ (line 37). The training and test sets consist of the aggregate of two classes of data points:

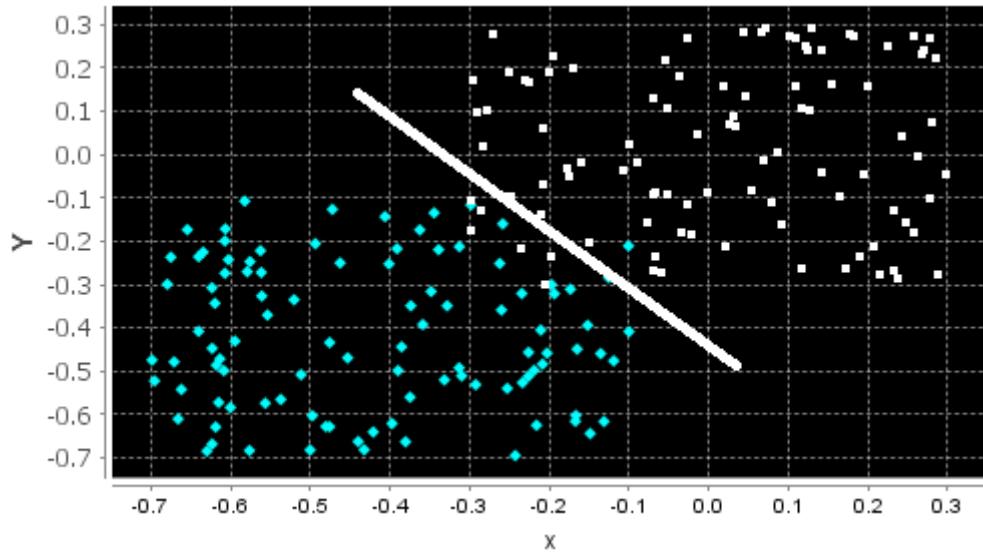
- Random data points with the variance a and mean b associated with the label 0.0
- Random data points with the variance a and mean $1-b$ associated with the label 1.0

Consider the following code for the training set:

```
val trainSet = genData(a, b) ++ genData(a, 1-b)
val testSet = genData(a, b) ++ genData(a, 1-b)
```

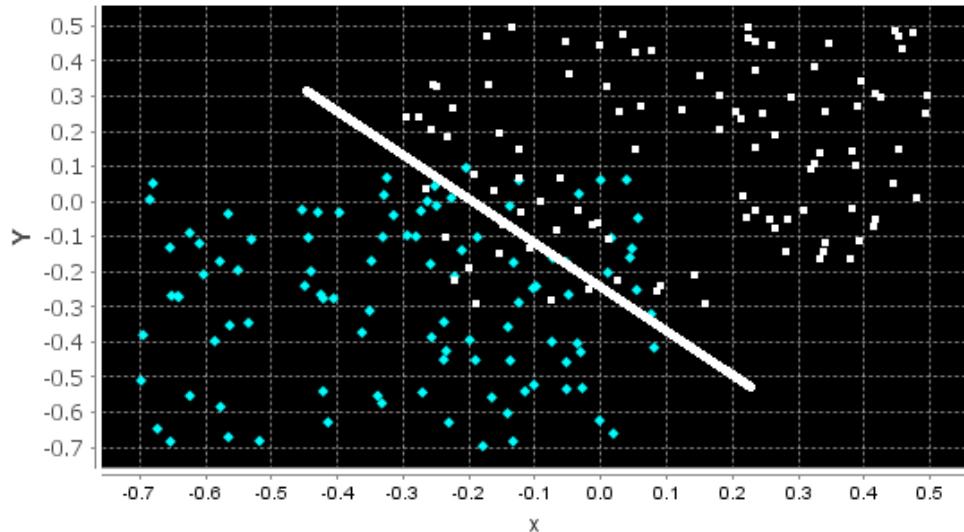
The a and b parameters are selected from two groups of training data points with various degrees of separation to illustrate the separating hyperplane.

The following chart describes the high margin; the first training set generated with the parameters $a = 0.6$ and $b = 0.3$ illustrates the highly separable classes with a clean and distinct hyperplane:



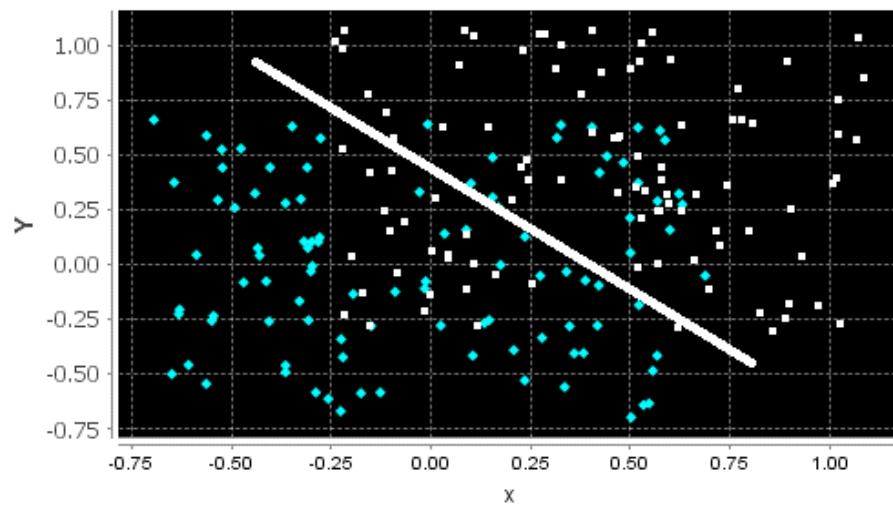
The scatter plot for training and testing sets with $a = 0.6$ and $b = 0.3$

The following chart describes the medium margin; the parameters $a = 0.8$ and $b = 0.3$ generate two groups of observations with some overlap:



The scatter plot for training and testing sets with $a = 0.8$ and $b = 0.3$

The following chart describes the low margin; the two groups of observations in this last training set are generated with $a = 1.4$ and $b = 0.3$ and show a significant overlap:



The scatter plot for training and testing sets with $a = 1.4$ and $b = 0.3$

The test set is generated in a similar fashion as the training set, as they are extracted from the same data source:

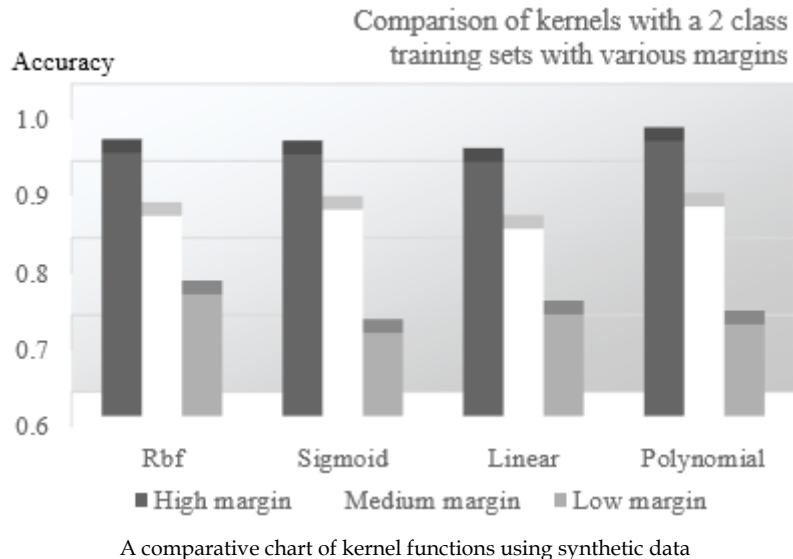
```

val GAMMA = 0.8; val COEF0 = 0.5; val DEGREE = 2 //38
val N = 100

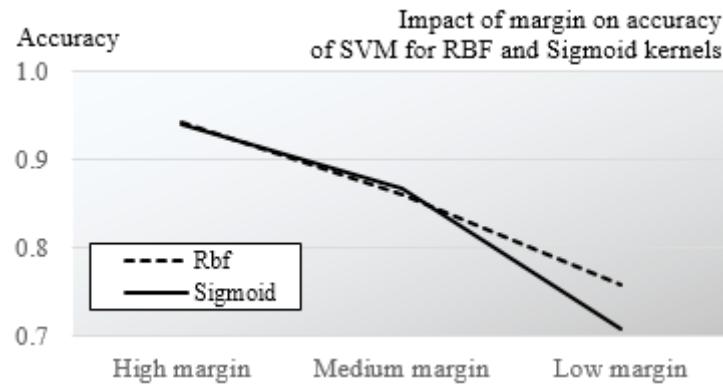
def compareKernel(a: Double, b: Double) {
    val labels = Vector.fill(N)(0.0) ++ Vector.fill(N)(1.0)
    evalKernel(trainSet, testSet, labels, new RbfKernel(GAMMA))
    evalKernel(trainSet, testSet, labels,
               new SigmoidKernel(GAMMA))
    evalKernel(trainSet, testSet, labels, LinearKernel)
    evalKernel(trainSet, testSet, labels,
               new PolynomialKernel(GAMMA, COEF0, DEGREE))
}

```

The parameters for each of the four kernel functions are arbitrary selected from textbooks (line 38). The evalKernel method defined earlier is applied to the three training sets: the high margin ($a = 1.4$), medium margin ($a = 0.8$), and low margin ($a = 0.6$) with each of the four kernels (RBF, sigmoid, linear, and polynomial). The accuracy is assessed by counting the number of observations correctly classified for all of the classes for each invocation of the predictor, |>:



Although the different kernel functions do not differ in terms of the impact on the accuracy of the classifier, you can observe that the RBF and polynomial kernels produce results that are slightly more accurate. As expected, the accuracy decreases as the margin decreases. A decreasing margin indicates that the cases are not easily separable, affecting the accuracy of the classifier:



The impact of the margin value on the accuracy of RBF and Sigmoid kernel functions

A test case design



The test to compare the different kernel methods is highly dependent on the distribution or mixture of data in the training and test sets. The synthetic generation of data in this test case is used for illustrating the margin between classes of observations. Real-world datasets may produce different results.

In summary, there are four steps required to create a SVC-based model:

1. Select a features set.
2. Select the C-penalty (inverse regularization).
3. Select the kernel function.
4. Tune the kernel parameters.

As mentioned earlier, this test case relies on synthetic data to illustrate the concept of the margin and compare kernel methods. Let's use the support vector classifier for a real-world financial application.

Applications in risk analysis

The purpose of the test case is to evaluate the risk for a company to curtail or eliminate its quarterly or yearly dividend. The features selected are financial metrics relevant to a company's ability to generate a cash flow and pay out its dividends over the long term.

We need to select any subset of the following financial technical analysis metrics (refer to *Appendix A, Basic Concepts*):

- Relative change in stock prices over the last 12 months
- Long-term debt-equity ratio
- Dividend coverage ratio
- Annual dividend yield
- Operating profit margin
- Short interest (ratio of shares shorted over the float)
- Cash per share-share price ratio
- Earnings per share trend

The earnings trend has the following values:

- -2 if earnings per share decline by more than 15 percent over the last 12 months.
- -1 if earnings per share decline between 5 percent and 15 percent.
- 0 if earnings per share is maintained within 5 percent.
- +1 if earnings per share increase between 5 percent and 15 percent.
- +2 if earnings per share increase by more than 15 percent. The values are normalized with values 0 and 1.

The labels or expected output (dividend changes) is categorized as follows:

- -1 if the dividend is cut by more than 5 percent
- 0 if the dividend is maintained within 5 percent
- +1 if the dividend is increased by more than 5 percent

Let's combine two of these three labels $\{-1, 0, 1\}$ to generate two classes for the binary SVC:

- Class C1 = stable or decreasing dividends and class C2 = increasing dividends – training set A
- Class C1 = decreasing dividends and class C2 = stable or increasing dividends – training set B

The different tests are performed with a fixed set of C and GAMMA configuration parameters and a 2-fold validation configuration:

```

val path = "resources/data/chap8/dividends2.csv"
val C = 1.0
val GAMMA = 0.5
val EPS = 1e-2
val NFOLDS = 2

val extractor = relPriceChange :: debtToEquity :: dividendCoverage :: cashPerShareToPrice :: epsTrend :: shortInterest :: dividendTrend :: List[Array[String] =>Double] () //39

val pfnSrc = DataSource(path, true, false,1) |> //40
val config = SVMConfig(new CSVCFormulation(C),
    new RbfKernel(GAMMA), SVMExecution(EPS, NFOLDS))

for {
    input <- pfnSrc(extractor) //41
    obs <- getObservations(input) //42
    svc <- SVM[Double](config, obs, input.last.toVector)
} yield {
    show(s"${svc.toString}\naccuracy ${svc.accuracy.get}")
}

```

The first step is to define the `extractor` (which is the list of fields to be retrieved from the `dividends2.csv` file) (line 39). The `pfnSrc` partial function generated by the `DataSource` transformation class (line 40) converts the input file into a set of typed fields (line 41). An observation is an array of fields. The `obs` sequence of observations is generated from the input fields by transposing the matrix `observations x features` (line 42):

```

def getObservations(input: Vector[DblArray]): Try[Vector[DblArray]] = Try {
    transpose( input.dropRight(1).map(_.toArray) ).toVector
}

```

The test computes the model parameters and the accuracy from the cross-validation during the instantiation of the SVM.

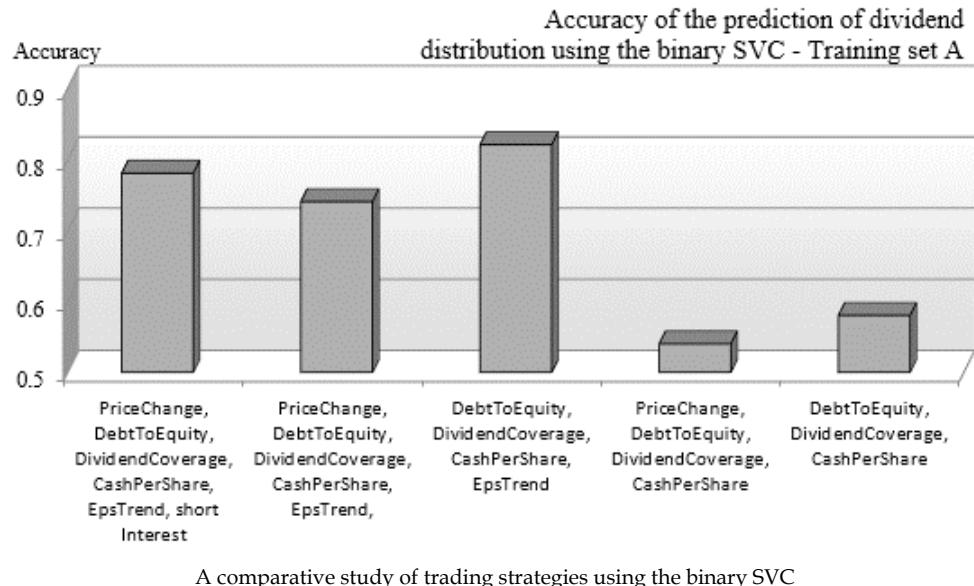
LIBSVM scaling

 LIBSVM supports feature normalization known as scaling, prior to training. The main advantage of scaling is to avoid attributes in greater numeric ranges, dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation. In our examples, we use the normalization method of the normalize time series. Therefore, the scaling flag in LIBSVM is disabled.

The test is repeated with a different set of features and consists of comparing the accuracy of the support vector classifier for different features sets. The features sets are selected from the content of the .csv file by assembling the extractor with different configurations, as follows:

```
val extractor = ... :: dividendTrend :: ...
```

Let's take a look at the following graph:

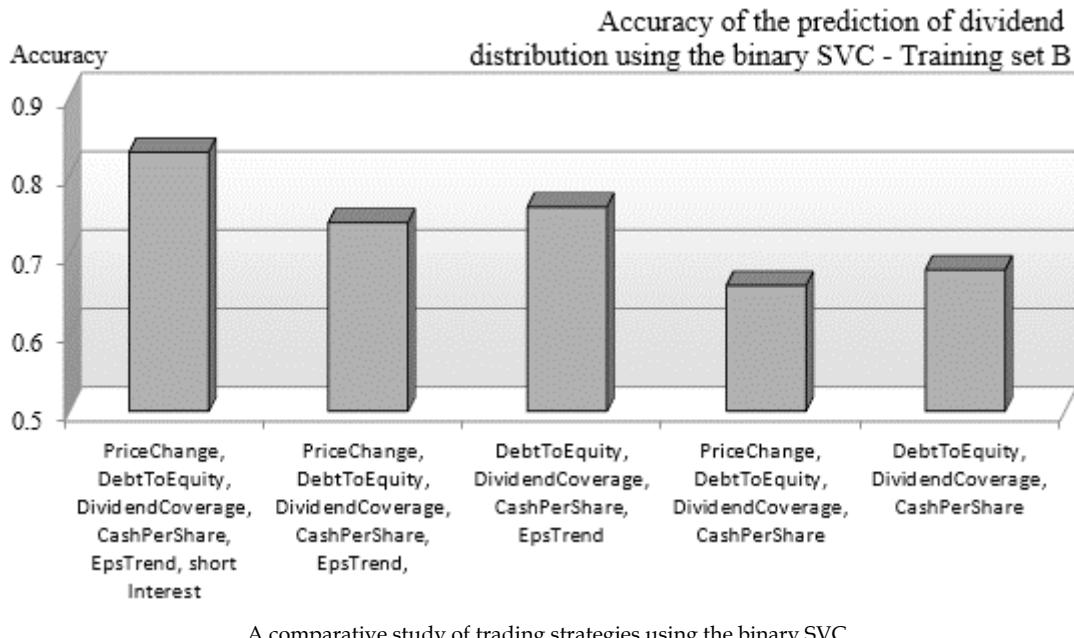


The test demonstrates that the selection of the proper features set is the most critical step in applying the support vector machine, and any other model for that matter, to classification problems. In this particular case, the accuracy is also affected by the small size of the training set. The increase in the number of features also reduces the contribution of each specific feature to the loss function.

The N-fold cross-validation

The cross-validation in this test example uses only two folds because the number of observations is small, and you want to make sure that any class contains at least a few observations.

The same process is repeated for the test B whose purpose is to classify companies with decreasing dividends and companies with stable or increasing dividends, as shown in the following graph:



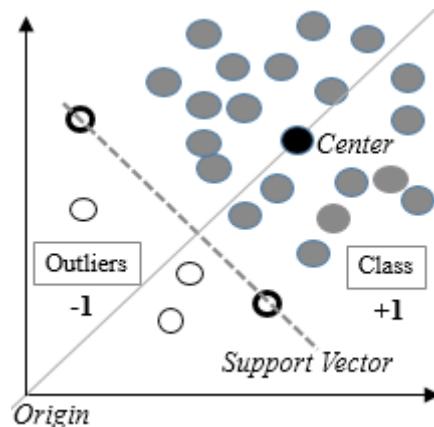
The difference in terms of accuracy of prediction between the first three features set and the last two features set in the preceding graph is more pronounced in test A than test B. In both the tests, the eps feature (earning per share) trend improves the accuracy of the classification. It is a particularly good predictor for companies with increasing dividends.

The problem of predicting the distribution (or not) dividends can be restated as evaluating the risk of a company to dramatically reduce its dividends.

What is the risk if a company eliminates its dividend altogether? Such a scenario is rare, and these cases are actually outliers. A one-class support vector classifier can be used to detect outliers or anomalies [8:13].

Anomaly detection with one-class SVC

The design of the one-class SVC is an extension of the binary SVC. The main difference is that a single class contains most of the baseline (or normal) observations. A reference point, known as the SVC origin, replaces the second class. The outliers (or abnormal) observations reside beyond (or outside) the support vector of the single class:



The visualization of the one-class SVC

The outlier observations have a labeled value of -1, while the remaining training sets are labeled +1. In order to create a relevant test, we add four more companies that have drastically cut their dividends (ticker symbols WLT, RGS, MDC, NOK, and GM). The dataset includes the stock prices and financial metrics recorded prior to the cut in dividends.

The implementation of this test case is very similar to the binary SVC driver code, except for the following:

- The classifier uses the Nu-SVM formulation, `OneSVFormulation`
- The labeled data is generated by assigning -1 to companies that have eliminated their dividends and +1 for all other companies

The test is executed against the resources/data/chap8/dividends2.csv dataset. First, we need to define the formulation for the one-class SVM:

```
class OneSVCFormulation(nu: Double) extends SVMFormulation {
    override def update(param: svm_parameter): Unit = {
        param.svm_type = svm_parameter.ONE_CLASS
        param.nu = nu
    }
}
```

The test code is similar to the execution code for the binomial SVC. The only difference is the definition of the output labels; -1 for companies eliminating dividends and +1 for all other companies:

```
val NU = 0.2
val GAMMA = 0.5
val EPS = 1e-3
val NFOLDS = 2

val extractor = relPriceChange :: debtToEquity :::
    dividendCoverage :: cashPerShareToPrice :: epsTrend :::
    dividendTrend :: List[Array[String] =>Double] ()

val filter = (x: Double) => if(x == 0) -1.0 else 1.0 //43
val pfnsr = DataSource(path, true, false, 1) |>
    val config = SVMConfig(new OneSVCFormulation(NU), //44
        new RbfKernel(GAMMA), SVMExecution(EPS, NFOLDS))

for {
    input <- pfnsr(extractor)
    obs <- getObservations(input)
    svc <- SVM[Double](config, obs,
        input.last.map(filter(_)).toVector)
} yield {
    show(s"${svc.toString}\naccuracy ${svc.accuracy.get}"')
}
```

The labels or expected data is generated by applying a binary filter to the last dividendTrend field (line 43). The formulation in the configuration has the OneSVCFormulation type (line 44).

The model is generated with the accuracy of 0.821. This level of accuracy should not be a surprise; the outliers (companies that eliminated their dividends) are added to the original dividend .csv file. These outliers differ significantly from the baseline observations (companies who have reduced, maintained, or increased their dividends) in the original input file.

In cases where the labeled observations are available, the one-class support vector machine is an excellent alternative to clustering techniques.

The definition of an anomaly

The results generated by a one-class support vector classifier depend heavily on the subjective definition of an outlier. The test case assumes that the companies that eliminate their dividends have unique characteristics that set them apart and are different even from companies who have cut, maintained, or increased their dividends. There is no guarantee that this assumption is indeed always valid.

Support vector regression

Most of the applications using support vector machines are related to classification. However, the same technique can be applied to regression problems. Luckily, as with classification, LIBSVM supports two formulations for support vector regression:

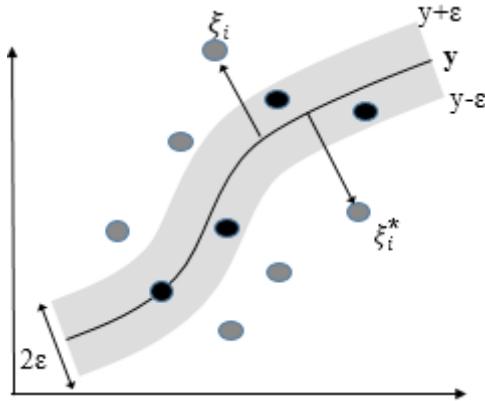
- ϵ -VR (sometimes called C-SVR)
- v-SVR

For the sake of consistency with the two previous cases, the following test uses the ϵ (or C) formulation of the support vector regression.

An overview

The SVR introduces the concept of **error insensitive zone** and insensitive error, ϵ . The insensitive zone defines a range of values around the predictive values, $y(x)$. The penalization component C does not affect the data point $\{x_i, y_i\}$ that belongs to the insensitive zone [8:14].

The following diagram illustrates the concept of an error insensitive zone using a single variable feature x and an output y . In the case of a single variable feature, the error insensitive zone is a band of width 2ϵ (ϵ is known as the insensitive error). The insensitive error plays a similar role to the margin in the SVC.



The visualization of the support vector regression and insensitive error

For the mathematically inclined, the maximization of the margin for nonlinear models introduces a pair of slack variables. As you may remember, the C-support vector classifiers use a single slack variable. The preceding diagram illustrates the minimization formula.

M9: The ϵ -SVR formulation is defined as:

$$\min_{w, \xi, \xi^*} \left\{ \frac{w^T w}{2} + C \sum_{i=0}^{n-1} (\xi_i + \xi_i^*) \right\}$$

$$-\epsilon - \xi_i^* \leq w^T \phi(x_i) + w_0 - y_i \leq \epsilon + \xi_i \quad \forall i$$



Here, ϵ is the insensitive error function.

M10: The ϵ -SVR regression equation is given by:

$$\hat{y}(x) = \sum_{i=0}^{n-1} \alpha_i K(x_i, x) + \hat{w}_0$$

Let's reuse the SVM class to evaluate the capability of the SVR, compared to the linear regression (refer to the *Ordinary least squares regression* section in *Chapter 6, Regression and Regularization*).

SVR versus linear regression

This test consists of reusing the example on single-variate linear regression (refer to the *One-variate linear regression* section in *Chapter 6, Regression and Regularization*). The purpose is to compare the output of the linear regression with the output of the SVR for predicting the value of a stock price or an index. We select the S&P 500 exchange traded fund, SPY, which is a proxy for the S&P 500 index.

The model consists of the following:

- One labeled output: SPY-adjusted daily closing price
- One single variable feature set: the index of the trading session (or index of the values SPY)

The implementation follows a familiar pattern:

1. Define the configuration parameters for the SVR (the `C` cost/penalty function, `GAMMA` coefficient for the RBF kernel, `EPS` for the convergence criteria, and `EPSILON` for the regression insensitive error).
2. Extract the labeled data (the `SPY price`) from the data source (`DataSource`), which is the Yahoo financials CSV-formatted data file.
3. Create the linear regression, `SingleLinearRegression`, with the index of the trading session as the single variable feature and the SPY-adjusted closing price as the labeled output.
4. Create the observations as a time series of indices, `xt`.
5. Instantiate the SVR with the index of trading session as features and the SPY-adjusted closing price as the labeled output.
6. Run the prediction methods for both SVR and the linear regression and compare the results of the linear regression and SVR, `collect`.

The code will be as follows:

```
val path = "resources/data/chap8/SPY.csv"
val C = 12
val GAMMA = 0.3
val EPSILON = 2.5

val config = SVMConfig(new SVRFormulation(C, EPSILON),
                      new RbfKernel(GAMMA)) //45
for {
    price <- DataSource(path, false, true, 1) get close
    (xt, y) <- getLabeledData(price.size) //46
```

```

linRg <- SingleLinearRegression[Double](price, y) //47
    svr <- SVM[Double](config, xt, price)
} yield {
    collect(svr, linRg, price)
}

```

The formulation in the configuration has the `SVRFormulation` type (line 45). The `DataSource` class extracts the price of the SPY ETF. The `getLabeledData` method generates the `xt` input features and the `y` labels (or expected values) (line 46):

```

type LabeledData = (Vector[DblArray], DblVector)
def getLabeledData(numObs: Int): Try[LabeledData] = Try {
    val y = Vector.tabulate(numObs)(_.toDouble)
    val xt = Vector.tabulate(numObs)(Array[Double](_))
    (xt, y)
}

```

The single variate linear regression, `SingleLinearRegression`, is instantiated using the `price` input and `y` labels as inputs (line 47).

Finally, the `collect` method executes the two `pfSvr` and `pfLinr` regression partial functions:

```

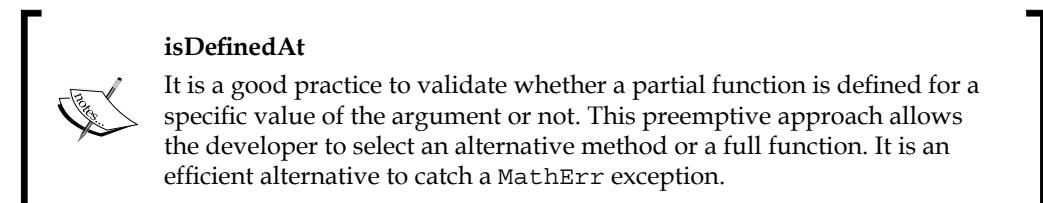
def collect(svr: SVM[Double],
           linr: SingleLinearRegression[Double], price: DblVector) {

    val pfSvr = svr |>
    val pfLinr = linr |>
    for {
        if( pfSvr.isDefinedAt(n.toDouble))
        x <- pfSvr(n.toDouble)
        if( pfLinr.isDefinedAt(n))
        y <- pfLinr(n)
    } yield { ... }
}

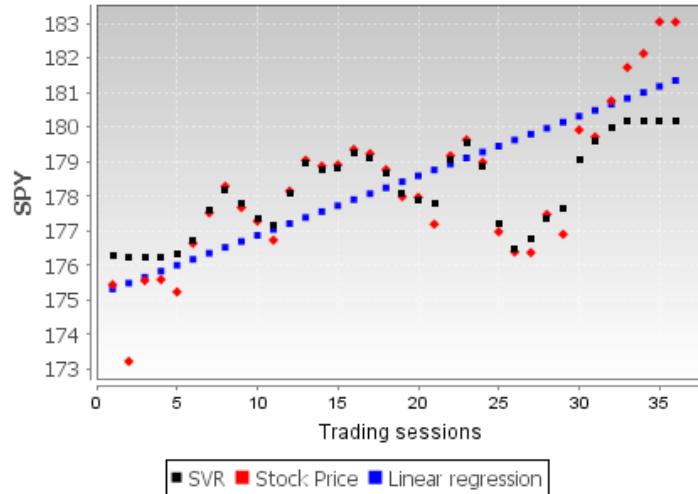
```

isDefinedAt

It is a good practice to validate whether a partial function is defined for a specific value of the argument or not. This preemptive approach allows the developer to select an alternative method or a full function. It is an efficient alternative to catch a `MathErr` exception.



The results are displayed in the following graph, which are generated using the JFreeChart library. The code to plot the data is omitted because it is not essential to the understanding of the application.



A comparative plot of linear regression and SVR

The support vector regression provides a more accurate prediction than the linear regression model. You can also observe that the L_2 regularization term of the SVR penalizes the data points (the SPY price) with a high deviation from the mean of the price. A lower value of C will increase the L_2 -norm penalty factor as $\lambda = 1/C$.



SVR and L_2 regularization

You are invited to run the use case with a different value of C to quantify the impact of the L_2 regularization on the predictive values of the SVR.

There is no need to compare SVR with the logistic regression, as the logistic regression is a classifier. However, the SVM is related to the logistic regression; the hinge loss in the SVM is similar to the loss in the logistic regression [8:15].

Performance considerations

You may have already observed that the training of a model for the support vector regression on a large dataset is time consuming. The performance of the support vector machine depends on the type of optimizer (for example, a sequential minimal optimization) selected to maximize the margin during training:

- A linear model (a SVM without kernel) has an asymptotic time complexity $O(N)$ for training N labeled observations.
- Nonlinear models rely on kernel methods formulated as a quadratic programming problem with an asymptotic time complexity of $O(N^3)$
- An algorithm that uses sequential minimal optimization techniques, such as index caching or elimination of null values (as in LIBSVM), has an asymptotic time complexity of $O(N^2)$ with the worst case scenario (quadratic optimization) of $O(N^3)$
- Sparse problems for very large training sets ($N > 10,000$) also have an asymptotic time of $O(N^2)$

The time and space complexity of the kernelized support vector machine has been receiving a great deal of attention [8:16] [8:17].

Summary

This concludes our investigation of kernel and support vector machines. Support vector machines have become a robust alternative to logistic regression and neural networks for extracting discriminative models from large training sets.

Apart from the unavoidable references to the mathematical foundation of maximum margin classifiers, such as SVMs, you should have developed a basic understanding of the power and complexity of the tuning and configuration parameters of the different variants of SVMs.

As with other discriminative models, the selection of the optimization method for SVMs has a critical impact not only on the quality of the model, but also on the performance (time complexity) of the training and cross-validation process.

The next chapter will describe the third most commonly used discriminative supervised model – artificial neural networks.

9

Artificial Neural Networks

The popularity of neural networks surged in the 90s. They were seen as the silver bullet to a vast number of problems. At its core, a neural network is a nonlinear statistical model that leverages the logistic regression to create a nonlinear distributed model. The concept of artificial neural networks is rooted in biology, with the desire to simulate key functions of the brain and replicate its structure in terms of neurons, activation, and synapses.

In this chapter, you will move beyond the hype and learn the following topics:

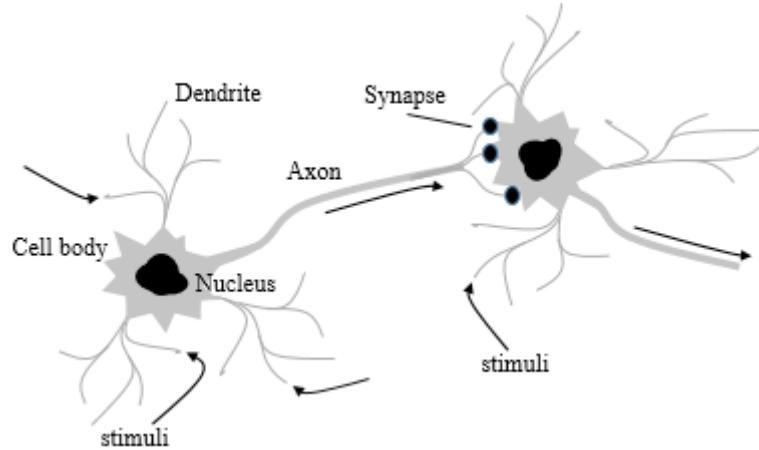
- The concepts and elements of the **multilayer perceptron (MLP)**
- How to train a neural network using error backpropagation
- The evaluation and tuning of MLP configuration parameters
- A full Scala implementation of the MLP classifier
- How to apply MLP to extract correlation models for currency exchange rates
- A brief introduction to **convolutional neural network (CNN)**

Feed-forward neural networks

The idea behind artificial neural networks was to build mathematical and computational models of the natural neural network in the brain. After all, the brain is a very powerful information processing engine that surpasses computers in domains, such as learning, inductive reasoning, prediction and vision, and speech recognition.

The biological background

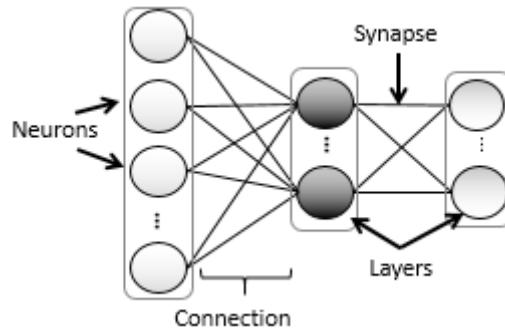
In biology, a neural network is composed of groups of neurons interconnected through synapses [9:1], as shown in the following diagram:



The visualization of biological neurons and synapses

Neuroscientists have been especially interested in understanding how billions of neurons in the brain can interact to provide human beings with parallel processing capabilities. The 60s saw a new field of study emerging, known as **connectionism**. Connectionism marries cognitive psychology, artificial intelligence, and neuroscience. The goal was to create a model for mental phenomena. Although there are many forms of connectionism, the neural network models have become the most popular and the most taught of all connectionism models [9:2].

Biological neurons communicate with electrical charges known as **stimuli**. This network of neurons can be represented as a simple schematic, as follows:



The representation of neuron layers, connections, and synapses

This representation categorizes groups of neurons as layers. The terminology used to describe the natural neural networks has a corresponding nomenclature for the artificial neural network.

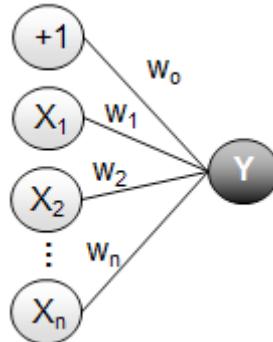
The biological neural network	The artificial neuron network
Axon	Connection
Dendrite	Connection
Synapse	Weight
Potential	Weighted sum
Threshold	Bias weight
Signal, Stimulus	Activation
Group of neurons	Layer of neurons

In the biological world, stimuli do not propagate in any specific direction between neurons. An artificial neural network can have the same degree of freedom. The most commonly used artificial neural networks by data scientists have a predefined direction: from the input layer to output layers. These neural networks are known as a **feed-forward neural network (FFNN)**.

Mathematical background

In the previous chapter, you learned that support vector machines have the ability to formulate the training of a model as a nonlinear optimization for which the objective function is convex. A convex objective function is fairly straightforward to implement. The drawback is that the kernelization of the SVM may result in a large number of basis functions (or model dimensions). Refer to the *The kernel trick* section under *Support vector machines* in *Chapter 8, Kernel Models and Support Vector Machines*. One solution is to reduce the number of basis functions through parameterization, so these functions can adapt to different training sets. Such an approach can be modeled as a FFNN, known as the multilayer perceptron [9:3].

The linear regression can be visualized as a simple connectivity model using neurons and synapses, as follows:



A two-layer neural network

The feature $x_0 = +1$ is known as the **bias input** (or the bias element), which corresponds to the intercept in the classic linear regression.

As with support vector machines, linear regression is appropriate for observations that can be linearly separable. The real world is usually driven by a nonlinear phenomenon. Therefore, the logistic regression is naturally used to compute the output of the perceptron. For a set of input variable $x = \{x_i\}_{0,n}$ and the weights $w = \{w_i\}_{1,n}$, the output y is computed as follows (**M1**):

$$y = \sigma(w_0 + w^T x) = \frac{1}{1 + e^{-(w_0 + w^T x)}}$$

A FFNN can be regarded as a stack of layers of logistic regression with the output layer as a linear regression.

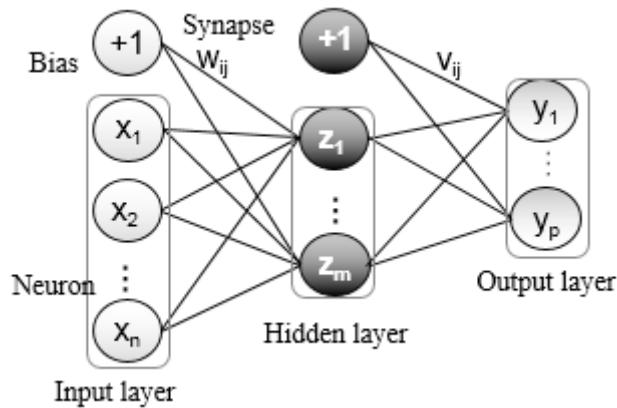
The value of the variables in each hidden layer is computed as the sigmoid of the dot product of the connection weights and the output of the previous layer. Although interesting, the theory behind artificial neural networks is beyond the scope of this book [9:4].

The multilayer perceptron

The perceptron is a basic processing element that performs a binary classification by mapping a scalar or vector to a binary (or XOR) value $\{\text{true}, \text{false}\}$ or $\{-1, +1\}$. The original perceptron algorithm was defined as a single layer of neurons for which each value x_i of the feature vector is processed in parallel and generates a single output y . The perceptron was later extended to encompass the concept of an activation function.

The single layer perceptrons are limited to process a single linear combination of weights and input values. Scientists found out that adding intermediate layers between the input and output layers enable them to solve more complex classification problems. These intermediate layers are known as **hidden layers** because they interface only with other perceptrons. Hidden nodes can be accessed only through the input layer.

From now on, we will use a three-layered perceptron to investigate and illustrate the properties of neural networks, as shown here:



A three-layered perceptron

The three-layered perceptron requires two sets of weights: w_{ij} to process the output of the input layer to the hidden layer and v_{ij} between the hidden layer and the output layer. The intercept value w_0 , in both linear and logistic regression, is represented with +1 in the visualization of the neural network ($w_0 \cdot 1 + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots$).

A FFNN without a hidden layer



A FFNN without a hidden layer is similar to a linear statistical model. The only transformation or connection between the input and output layer is actually a linear regression. A linear regression is a more efficient alternative to the FFNN without a hidden layer.

The description of the MLP components and their implementations rely on the following stages:

1. An overview of the software design.
2. A description of the MLP model components.
3. The implementation of the four-step training cycle.
4. The definition and implementation of the training strategy and the resulting classifier.

Terminology



Artificial neural networks encompass a large variety of learning algorithms, the multilayer perceptron being one of them. Perceptrons are indeed components of a neural network organized as the input, output, and hidden layers. This chapter is dedicated to the multilayer perceptron with hidden layers. The terms "neural network" and "multilayer perceptron" are used interchangeably.

The activation function

The perceptron is represented as a linear combination of weights w_i and input values x_i processed by the output unit activation function h , as shown here (**M2**):

$$\hat{y} = h \left(w_0 + \sum_{i=1}^n w_i x_i \right) = h(w_0 + w^T x)$$

The output activation function h has to be continuous and differentiable for a range of value of the weights. It takes different forms depending on the problems to be solved, as mentioned here:

- An identity for the output layer (linear formula) of the regression mode
- The sigmoid σ for hidden layers and output layers of the binomial classifier

- Softmax for the multinomial classification
- The hyperbolic tangent, $tanh$, for the classification using zero mean

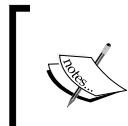
The softmax formula is described in *Step 1 – input forward propagation* under *Training epoch*.

The network topology

The output layers and hidden layers have a computational capability (dot product of weights, inputs, and activation functions). The input layer does not transform data. An n -layer neural network is a network with n computational layers. Its architecture consists of the following components:

- one input layer
- $n-1$ hidden layer
- one output layer

A **fully connected neural network** has all its input nodes connected to hidden layer neurons. Networks are characterized as **partially connected neural networks** if one or more of their input variables are not processed. This chapter deals with a fully connected neural network.



Partially connected networks

Partially connected networks are not as complex as they seem. They can be generated from fully connected networks by setting some of the weights to zero.

The structure of the output layer is highly dependent on the type of problems (regression or classification) you need to solve, also known as the operating mode of the multilayer perceptron. The type of problem at hand defines the number of output nodes [9:5]. Consider the following examples:

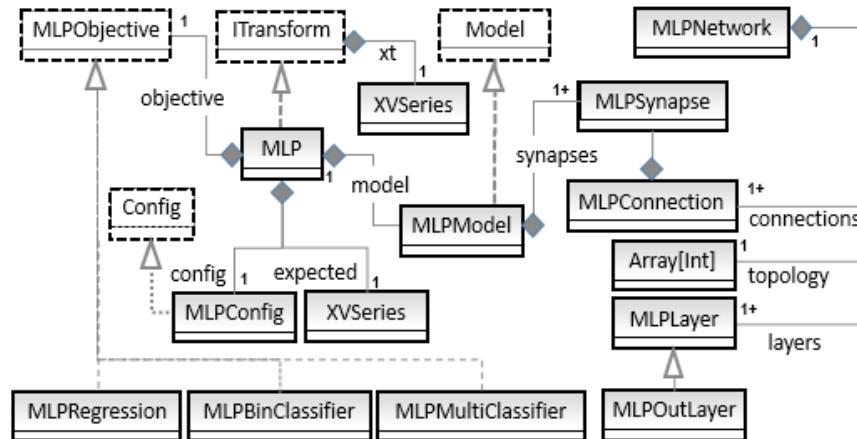
- A one-variate regression has one output node whose value is a real number $[0, 1]$
- A multivariate regression with n variables has n real output nodes
- A binary classification has one binary output node $\{0, 1\}$ or $\{-1, +1\}$
- A multinomial or K-class classification has K binary output nodes

Design

The implementation of the MLP classifier follows the same pattern as previous classifiers (refer to the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*):

- An `MLPNetwork` connectionist network is composed of a layer of neurons of the `MLPLayer` type, connected by synapses of the `MLPSynapse` type contained by a connector of the `MLPConnection` type.
- All of the configuration parameters are encapsulated into a single `MLPConfig` configuration class.
- A model, `MLPModel`, consists of a sequence of connection synapses.
- The `MLP` multilayer perceptron class is implemented as a data transformation, `ITransform`, for which the model is automatically extracted from a training set with labels.
- The `MLP` multilayer perceptron class takes four parameters: a configuration, a features set or time series of the `XVSeries` type, a labeled dataset of the `XVSeries` type, and an activation function of the `Function1[Double, Double]` type.

The software components of the multilayer perceptron are described in the following UML class diagram:



A UML class diagram for the multilayer perceptron

The class diagram is a convenient navigation map used to understand the role and relation of the Scala classes used to build an MLP. Let's start with the implementation of the MLP network and its components. The UML diagram omits the helper traits or classes such as `Monitor` or the Apache Commons Math components.

Configuration

The `MLPConfig` configuration of the multilayer perceptron consists of the definition of the network configuration with its hidden layers, the learning and training parameters, and the activation function:

```
case class MLPConfig(
    val alpha: Double, //1
    val eta: Double,
    val numEpochs: Int,
    val eps: Double,
    val activation: Double => Double) extends Config { //1
}
```

For the sake of readability, the name of the configuration parameters matches the symbols defined in the mathematical formulation (line 1):

- `alpha`: This is the momentum factor α that smoothes the computation of the gradient of the weights for online training. The momentum factor is used in the mathematical expression **M10** in *Step 2 – error backpropagation* under *Training epoch*.
- `eta`: This is the learning rate η used in the gradient descent. The gradient descent updates the weights or parameters of a model by the quantity, $\eta \cdot (predicted - expected).input$, as described in the mathematical formulation **M9** in *Step 2 – error backpropagation* section under *The training epoch*. The gradient descent was introduced in *Let's kick the tires* in *Chapter 1, Getting Started*.
- `numEpochs`: This is the maximum number of epochs (or cycles or episodes) allowed for training the neural network. An epoch is the execution of the error backpropagation across the entire observation set.
- `eps`: This is the convergence criteria used as an exit condition for the training of the neural network when $error < eps$.
- `activation`: This is the activation function used for nonlinear regression applied to hidden layers. The default function is the sigmoid (or the hyperbolic tangent) introduced for the logistic regression (refer to the *Logistic function* section in *Chapter 6, Regression and Regularization*).

Network components

The training and classification of an MLP model relies on the network architecture. The `MLPNetwork` class is responsible for creating and managing the different components and the topology of the network, that is layers, synapses, and connections.

The network topology

The instantiation of the `MLPNetwork` class requires a minimum set of two parameters with an instance of the model as an optional third argument (line 2):

- An MLP execution configuration, `config`, introduced in the previous section
- A topology defined as an array of the number of nodes for each layer: input, hidden, and output layers.
- A model with the `Option[MLPModel]` type if it has already been generated through training, or `None` otherwise
- An implicit reference to the operating mode of the MLP

The code is as follows:

```
class MLPNetwork(config: MLPConfig,  
  topology: Array[Int],  
  model: Option[MLPModel] = None)  
  (implicit mode: MLPMode) { //2  
  
  val layers = topology.zipWithIndex.map { case(t, n) =>  
    if (topology.size != n+1)  
      MLPLayer(n, t+1, config.activation)  
    else MLPOutLayer(n, t)  
  } //3  
  val connections = zipWithShift1(layers, 1).map{case(src, dst) =>  
    new MLPConnection(config, src, dst, model)} //4  
  
  def trainEpoch(x: DblArray, y: DblArray): Double //5  
  def getModel: MLPModel //6  
  def predict(x: DblArray): DblArray //7  
}
```

A MLP network has the following components, which are derived from the topology array:

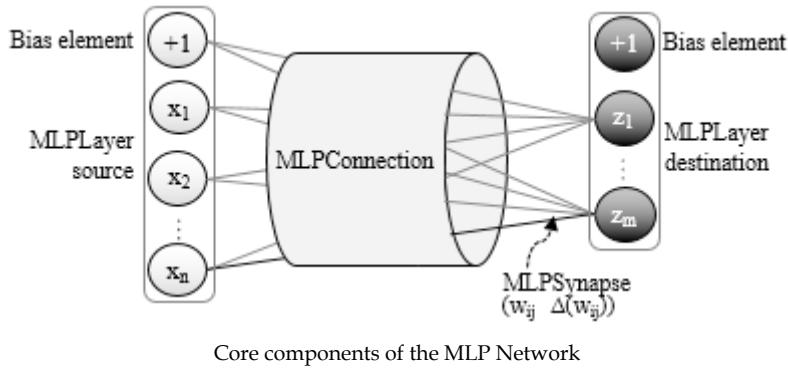
- Multiple layers of the `MLPLayers` class (line 3)
- Multiple connections of the `MLPConnection` class (line 4)

The topology is defined as an array of number of nodes per layer, starting with the input nodes. The array indices follow the forward path within the network. The size of the input layer is automatically generated from the observations as the size of the features vector. The size of the output layer is automatically extracted from the size of the output vector (line 3).

The constructor for `MLPNetwork` creates a sequence of layers by assigning and ordering an `MLPLayer` instance to each entry in the topology (line 3). The constructor creates *number of layers - 1* interlayer connections of the `MLPConnection` type (line 4). The `zipWithShift1` method of the `XTSeries` object zips a time series with its duplicated shift by one element.

The `trainEpoch` method (line 5) implements the training of this network for a single pass of the entire set of observations (refer to the *Putting it all together* section under *The training epoch*). The `getModel` method retrieves the model (synapses) generated through training of the MLP (line 6). The `predict` method computes the output value generated from the network using the forward propagation algorithm (line 7).

The following diagram visualizes the interaction between the different components of a model: `MLPLayer`, `MLPConnection`, and `MLPSynapse`:



Input and hidden layers

First, let's start with the definition of the `MLPLayer` layer class, which is completely specified by its position (or rank) `id` in the network and the number of nodes, `numNodes`, it contains:

```
class MLPLayer(val id: Int, val numNodes: Int,
    val activation: Double => Double) //8
    (implicit mode: MLPMode){ //9

    val output = Array.fill(numNodes)(1.0) //10

    def setOutput(xt: DblArray): Unit =
        xt.copyOf(output, 1) //11
    def activate(x: Double): Double = activation(x) //12
    def delta(loss: DblArray, srcOut: DblArray,
        synapses: MLPConnSynapses): Delta //13
    def setInput(_x: DblArray): Unit //14
}
```

The `id` parameter is the order of the layer (0 for input, 1 for the first hidden layer, and $n - 1$ for the output layer) in the network. The `numNodes` value is the number of elements or nodes, including the bias element, in this layer. The `activation` function is the last argument of the layer given a user-defined mode or objective (line 8). The `operatingMode` has to be provided implicitly prior to the instantiation of a layer (line 9).

The output vector for the layer is an uninitialized array of values updated during the forward propagation. It initializes the bias value with the value 1.0 (line 9). The matrix of difference of weights, `deltaMatrix`, associated with the output vector (line 10) is updated using the error backpropagation algorithm, as described in the *Step 2 – error back propagation* section under *The training epoch*. The `setOutput` method initializes the output values for the output and hidden layers during the backpropagation of the error on the output of the network (*expected – predicted*) values (line 11).

The `activate` method invokes the activation method (*tanh*, *sigmoid*, ...) defined in the configuration (line 12).

The `delta` method computes the correction to be applied to each weight or synapses, as described in the *Step 2 – error back propagation* section under *The training epoch* (line 13).

The `setInput` method initializes the `output` values for the nodes of the input and hidden layers, except the bias element, with the value `x` (line 14). The method is invoked during the forward propagation of input values:

```
def setInput(x: DblVector): Unit =
  x.copyToArray(output, output.length -x.length)
```

The methods of the `MLPLayer` class for the input and hidden layers are overridden for the output layer of the `MLPOutLayer` type.

The output layer

Contrary to the hidden layers, the output layer does not have either an activation function or a bias element. The `MLPOutLayer` class has the following arguments: the order `id` in the network (as the last layer of the network) and the number, `numNodes`, of the output or nodes (line 15):

```
class MLPOutLayer(id: Int, numNodes: Int)
  (implicit mode: MLP.MLPMode) //15
  extends MLPLayer(id, numNodes, (x: Double) => x) {

  override def numNonBias: Int = numNodes
  override def setOutput(xt: DblArray): Unit =
    obj(xt).copyToArray(output)
```

```
override def delta(loss: DblArray, srcOut: DblArray,
  synapses: MLPConnSynapses): Delta
...
}
```

The `numNonBias` method returns the actual number of output values from the network. The implementation of the `delta` method is described in the *Step 2 – error back propagation* section under *The training epoch*.

Synapses

A synapse is defined as a pair of real (a floating point) values:

- The weight w_{ij} of the connection from the neuron i of the previous layer to the neuron j
- The weights' adjustment (or gradient of weights) Δw_{ij}

Its type is defined as `MLPSynapse`, as shown here:

```
type MLPSynapse = (Double, Double)
```

Connections

The connections are instantiated by selecting two consecutive layers of an index n (with respect to $n + 1$) as a source (with respect to destination). A connection between two consecutive layers implements the matrix of synapses as the $(w_{ij}, \Delta w_{ij})$ pairs. The `MLPConnection` instance is created with the following parameters (line 16):

- Configuration parameters, `config`
- The source layer, sometimes known as the ingress layer, `src`
- The `dst` destination (or egress) layer
- A reference to the `model` if it has already been generated through training or `None` if the model has not been trained
- An implicitly defined operating `mode` or objective `mode`

The `MLPConnection` class is defined as follows:

```
type MLPConnSynapses = Array[Array[MLPSynapse]]

class MLPConnection(config: MLPConfig,
  src: MLPLayer,
  dst: MLPLayer,
  model: Option[MLPModel]) //16
```

```
(implicit mode: MLP.MLPMode) {  
  
    var synapses: MLPConnSynapses //17  
    def connectionForwardPropagation: Unit //18  
    def connectionBackpropagation(delta: Delta): Delta //19  
    ...  
}
```

The last step in the initialization of the MLP algorithm is the selection of the initial (usually random) values of the weights (synapse) (line 17).

The MLPConnection methods implement the forward propagation of weights' computation for this connectionForwardPropagation connection (line 18) and the backward propagation of the delta error during training connectionBackpropagation (line 19). These methods are described in the next section related to the training of the MLP model.

The initialization weights

The initialization values for the weights depends is domain specific. Some problems require a very small range, less than $1e-3$, while others use the probability space $[0, 1]$. The initial values have an impact on the number of epochs required to converge toward an optimal set of weights [9:6].

Our implementation relies on the sigmoid activation function and uses the range $[0, BETA/sqrt(numOutputs + 1)]$ (line 20). However, the user can select a different range for random values, such as $[-r, +r]$ for the *tanh* activation function. The weight of the bias is obviously defined as $w_0=+1$, and its weight adjustment is initialized as $\Delta w_0 = 0$, as shown here (line 20):

```
var synapses: MLPConnSynapses = if(model == None) {  
    val max = BETA/Math.sqrt(src.output.length+1.0) //20  
    Array.fill(dst.numNonBias) (  
        Array.fill(src.numNodes) ((Random.nextDouble*max, 0.00))  
    )  
} else model.get.synapses(src.id) //21
```

The connection derives its weights or synapses from a model (line 21) if it has already been created through training.

The model

The `MLPNetwork` class defines the topological model of the multilayer perceptron. The weights or synapses are the attributes of the model of the `MLPModel` type generated through training:

```
case class MLPModel(
  val synapses: Vector[MLPConnSynapses]) extends Model
```

The model can be stored in a simple key-value pair JSON, CVS, or sequence file.

Encapsulation and the model factory

The network components: connections, layers, and synapses are implemented as top-level classes for the sake of clarity. However, there is no need for the model to expose its inner workings to the client code. These components should be declared as an inner class to the model. A factory design pattern would be perfectly appropriate to instantiate an `MLPNetwork` instance dynamically [9:7].

Once initialized, the MLP model is ready to be trained using a combination of forward propagation, output error back propagation, and iterative adjustment of weights and gradients of weights.

Problem types (modes)

There are three distinct types of problems or operating modes associated with the multilayer perceptron:

- The **binomial classification** (binary) with two classes and one output
- The **multinomial classification** (multiclass) with n classes and output
- Regression

Each operating mode has distinctive error, hidden layer, and output layer activation functions, as illustrated in the following table:

Operating modes	Error function	Hidden layer activation function	Output layer activation function
Binomial classification	Cross-entropy	Sigmoid	Sigmoid
Multinomial classification	Sum of squares error or mean squared error	Sigmoid	Softmax
Regression	Sum of squares error or mean squared error	Sigmoid	Linear

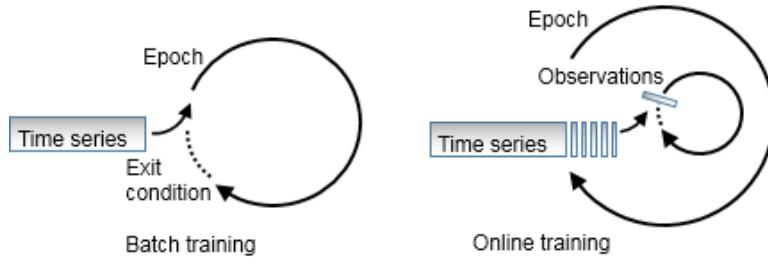
A table for operating modes of the multilayer perceptron

The cross-entropy is described by the mathematical expressions **M6** and **M7** and the softmax uses the formula **M8** in the *Step 1 – input forward propagation* section under *The training epoch*.

Online training versus batch training

One important issue is to find a strategy to conduct the training of a time series as an ordered sequence of data. There are two strategies to create an MLP model for a time series:

- **Batch training:** The entire time series is processed at once as a single input to the neural network. The weights (synapses) are updated at each epoch using the sum of the squared errors on the output of the time series. The training exits once the sum of the squared errors meets the convergence criteria.
- **Online training:** The observations are fed to the neural network one at a time. Once the time series has been processed, the total of the sum of the squared errors (sse) for the time series for all the observations are computed. If the exit condition is not met, the observations are reprocessed by the network.



An online training is faster than batch training because the convergence criterion has to be met for each data point, possibly resulting in a smaller number of epochs [9:12]. Techniques such as the momentum factor, which is described earlier, or any adaptive learning scheme improves the performance and accuracy of the online training methodology.

The online training strategy is applied to all the test cases of this chapter.

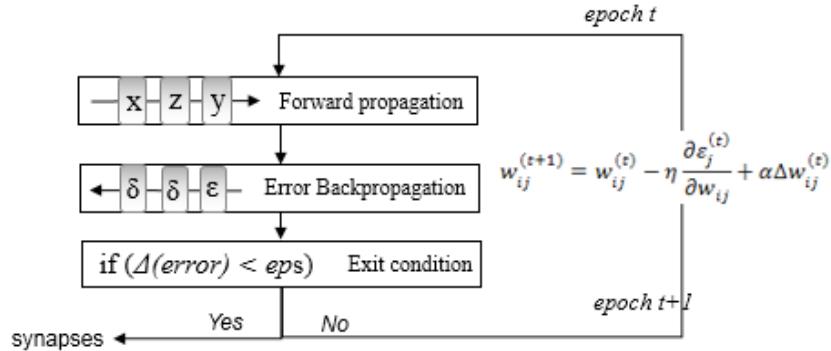
The training epoch

The training of the model processes the training observations iteratively multiple times. A training cycle or iteration is known as an **epoch**. The order of observations is shuffled for each epoch. The three steps of the training cycle are as follows:

1. Forward the propagation of the input value for a specific epoch.
2. Computation and backpropagation of the output error.
3. Evaluate the convergence criteria and exit if the criteria is met.

The computation of the network weights during training can use the difference between labeled data and actual output for each layer. But this solution is not feasible because the output of the hidden layers is actually unknown. The solution is to propagate the error on the output values (predicted values) backward to the input layer through the hidden layers, if an error is defined.

The three steps of the training cycle or training epoch are summarized in the following diagram:



An iterative implementation of the training for MLP

Let's apply the three steps of a training epoch in the `trainEpoch` method of the `MLPNetwork` class using a simple `foreach` Scala higher order function, as shown here:

```

def trainEpoch(x: DblArray, y: DblArray): Double = {
  layers.head.setInput(x) //22
  connections.foreach(_.connectionForwardPropagation) //23

  val err = mode.error(y, layers.last.output)
  val bckIterator = connections.reverseIterator

  var delta = Delta(zipToArray(y, layers.last.output)(diff)) //24
  bckIterator.foreach(iter =>
    delta = iter.connectionBackpropagation(delta)) //25
  err //26
}
  
```

You can certainly recognize the first two stages of the training cycle: the forward propagation of the input and the backpropagation of the error of the online training of a single epoch.

The execution of the training of the network for one epoch, `trainEpoch`, initializes the input layer with observations, `x` (line 22). The input values are propagated through the network by invoking `connectionForwardPropagation` for each connection (line 23). The `delta` error is initialized from the values in the output layer and the expected values, `y` (line 24).

The training method iterates through the connections backward to propagate the error through each connection by invoking the `connectionBackpropagation` method on the backward iterator, `bckIterator` (line 25). Finally, the training method returns the cumulative error, mean square error, or cross entropy, according to the operating mode (line 26).

This approach is not that different than the beta (or backward) pass in the hidden Markov model, which was covered in the *Beta – the backward pass* section in *Chapter 7, Sequential Data Models*.

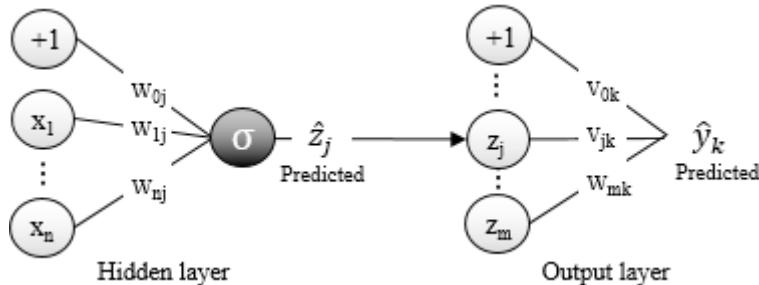
Let's take a look at the implementation of the forward and backward propagation algorithm for each type of connection:

- An input or hidden layer to a hidden layer
- A hidden layer to an output layer

Step 1 – input forward propagation

As mentioned earlier, the output values of a hidden layer are computed as the sigmoid or hyperbolic tangent of the dot product of the weights w_{ij} and the input values x_i .

In the following diagram, the MLP algorithm computes the linear product of the weights w_{ij} and input x_i for the hidden layer. The product is then processed by the activation function σ (the sigmoid or hyperbolic tangent). The output values z_j are then combined with the weights v_{jk} of the output layer that doesn't have an activation function:



The distribution of weights in MLP hidden and output layers

The mathematical formulation of the output of a neuron j is defined as a composition of the activation function and the dot product of the weights w_{ij} and input values x_i :

M3: The computation (or prediction) of the output layer from the output values z_j of the preceding hidden layer and the weights v_{kj} is defined as:

$$\tilde{y}_k = v_{0j} + \sum_{j=1}^m v_{kj} z_j$$



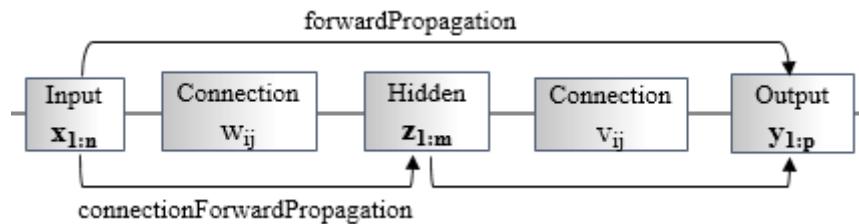
M4: The estimation of the output values for a binary classification with an activation function σ is defined as:

$$\tilde{z}_j = \sigma\left(w_{0j} + \sum_{i=1}^m w_{ij} x_i\right) = \frac{1}{1 + e^{-w_{0j} - \sum_{i=1}^m w_{ij} x_i}}$$

As seen in the network architecture section, the output values for the multinomial (or multiclass) classification with more than two classes are normalized using an exponential function, as described in the following *Softmax* section.

The computational flow

The computation of the output values y from the input x is known as the input forward propagation. For the sake of simplicity, we represent the forward propagation between layers with the following block diagram:



A computation model of the input forward propagation

The preceding diagram conveniently illustrates a computational model for the input forward propagation, as the programmatic relation between the source and destination layers and their connectivity. The input x is propagated forward through each connection.

The `connectionForwardPropagation` method computes the dot product of the weights and the input values and applies the activation function, in the case of hidden layers, for each connection. Therefore, it is a member of the `MLPConnection` class.

The forward propagation of input values across the entire network is managed by the MLP algorithm itself. The forward propagation of the input value is used in the classification or prediction $y = f(x)$. It depends on the value weights w_{ij} and v_{ij} that need to be estimated through training. As you may have guessed, the weights define the model of a neural network similar to the regression models. Let's take a look at the `connectionForwardPropagation` method of the `MLPConnection` class:

```
def connectionForwardPropagation: Unit = {
    val _output = synapses.map(x => {
        val dot = inner(src.output, x.map(_.value)) //27
        dst.activate(dot) //28
    })
    dst.setOutput(_output) //29
}
```

The first step is to compute the linear inner (or dot) product (refer to the *Time series in Scala* section in *Chapter 3, Data Preprocessing*) of the output, `_output`, of the current source layer for this connection and the synapses (weights) (line 27). The activation function is computed by applying the `activate` method of the destination layer to the dot product (line 28). Finally, the computed value, `_output`, is used to initialize the output for the destination layer (line 29).

Error functions

As mentioned in the *Problem types (modes)* section, there are two approaches to compute the error or loss on the output values:

- The sum of the squared errors between expected and predicted output values, as defined in the **M5** mathematical expression
- Cross-entropy of expected and predicted values described in the **M6** and **M7** mathematical formulas

M5: The sum of the squared errors ε and mean square error for predicted values \tilde{y} and expected values y are defined as:

$$\varepsilon = \frac{1}{2} \sum_{i=0}^{n-1} \sum_{j=0}^{k-1} (y_{ij} - \tilde{y}_{ij})^2 \quad \bar{\varepsilon} = \frac{\varepsilon}{n}$$

M6: Cross entropy for a single output value y is defined as:



$$ce = - \sum_{i=0}^{n-1} \{y_i \log(\tilde{y}_i) + (1 - y_i) \cdot \log(1 - \tilde{y}_i)\}$$

M7: Cross entropy for a multivariable output vector y is defined as:

$$ce = - \sum_{i=0}^{n-1} \sum_{j=0}^{k-1} \tilde{y}_{ij} \log(y_{ij})$$

The sum of squared errors and mean squared error functions have been described in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*.

The `crossEntropy` method of the `XTSeries` object for a single variable is implemented as follows:

```
def crossEntropy(x: Double, y: Double): Double =
  -(x * Math.log(y) + (1.0 - x) * Math.log(1.0 - y))
```

The computation of the cross entropy for multiple variable features as a signature is similar to the single variable case:

```
def crossEntropy(xt: DblArray, yt: DblArray): Double =
  yt.zip(xt).aggregate(0.0)({ case (s, (y, x)) =>
    s - y * Math.log(x), _ + _ })
```

Operating modes

In the *network architecture* section, you learned that the structure of the output layer depends on the type of problems that need to be resolved, also known as operating modes. Let's encapsulate the different operating modes (binomial, multinomial classification, and regression) into a class hierarchy, implementing the `MLPMode` trait. The `MLPMode` trait has two methods that are specific to the type of the problem:

- `apply`: This is the transformation applied to the output values
- `error`: This is the computation of the cumulative error for the entire observation set

The code will be as follows:

```
trait MLPMode {
    def apply(output: DblArray): DblArray //30
    def error(labels: DblArray, output: DblArray): Double =
        mse(labels, output) //31
}
```

The `apply` method applies a transformation to the output layer, as described in the last column of the operating modes table (line 30). The `error` function computes the cumulative error or loss in the output layer for all the observations, as described in the first column of the operating modes table (line 31).

The transformation in the output layer of the `MLPBinClassifier` binomial (two-class) classifier consists of applying the `sigmoid` function to each output value (line 32). The cumulative error is computed as the cross entropy of the expected output, labels, and the predicted output (line 33):

```
class MLPBinClassifier extends MLPMode {
    override def apply(output: DblArray): DblArray =
        output.map(sigmoid(_)) //32
    override def error(labels: DblArray,
                      output: DblArray): Double =
        crossEntropy(labels.head, output.head) //33
}
```

The regression mode for the multilayer perceptron is defined according to the operating modes table in the *Problem types (modes)* section:

```
class MLPRegression extends MLPMode {
    override def apply(output: DblArray): DblArray = output
}
```

The multinomial classifier mode is defined by the `MLPMultiClassifier` class. It uses the `softmax` method to boost the output with the highest value, as shown in the following code:

```
class MLPMultiClassifier extends MLPMode {
    override def apply(output: DblArray): DblArray = softmax(output)
}
```

The `softmax` method is applied to the actual output value, not the bias. Therefore, the first node $y(0) = +1$ has to be dropped before applying the `softmax` normalization.

Softmax

In the case of a classification problem with K classes ($K > 2$), the output has to be converted into a probability $[0, 1]$. For problems that require a large number of classes, there is a need to boost the output y_k with the highest value (or probability). This process is known as **exponential normalization** or softmax [9:8].

M8: The softmax formula for the multinomial ($K > 2$) classification is as follows:

$$\hat{y}_k = \frac{e^{-\hat{y}_k}}{\sum_i e^{-\hat{y}_i}}$$

Here is the simple implementation of the softmax method of the `MLPMultiClassifier` class:

```
def softmax(y: DblArray): DblArray = {
    val softmaxValues = new DblArray(y.size)
    val expY = y.map( Math.exp(_) ) //34
    val expYSum = expY.sum //35

    expY.map( _ /expYSum) .copyToArray(softmaxValues, 1) //36
    softmaxValues
}
```

The softmax method implements the M8 mathematical expression. First, the method computes the `expY` exponential values of the output values (line 34). The exponentially transformed outputs are then normalized by their sum, `expYSum`, (line 35) to generate the array of the `softmaxValues` output (line 36). Once again, there is no need to update the bias element $y(0)$.

The second step in the training phase is to define and initialize the matrix of delta error values to be back propagated between layers from the output layer back to the input layer.

Step 2 – error backpropagation

The error backpropagation is an algorithm that estimates the error for the hidden layer in order to compute the change in weights of the network. It takes the sum of squared errors of the output as the input.



The convention for computing the cumulative error

Some authors refer to the backpropagation as a training methodology for an MLP, which applies the gradient descent to the output error defined as either the sum of squared errors, or the mean squared error for multinomial classification or regression. In this chapter, we keep the narrower definition of the backpropagation as the backward computation of the sum of squared errors.

Weights' adjustment

The connection weights Δv and Δw are adjusted by computing the sum of the derivatives of the error, over the weights scaled with a learning factor. The gradient of weights are then used to compute the error of the output of the source layer [9:9].

The simplest algorithm to update the weights is the gradient descent [9:10]. The batch gradient descent was introduced in *Let's kick the tires in Chapter 1, Getting Started*.

The gradient descent is a very simple and robust algorithm. However, it can be slower in converging toward a global minimum than the conjugate gradient or the quasi-Newton method (refer to the *Summary of optimization techniques* section in the *Appendix A, Basic Concepts*).

There are several methods available to speed up the convergence of the gradient descent toward a minimum, such as the momentum factor and adaptive learning coefficient [9:11].

Large variations of the weights during training increase the number of epochs required for the model (connection weights) to converge. This is particularly true for a training strategy known as online training. The training strategies are discussed in the next section. The momentum factor α is used for the remaining section of the chapter.

M9: The learning rate

The computation of neural network weights using the gradient descent is as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \frac{\partial \varepsilon_j^{(t)}}{\partial w_{ij}}$$



M10: The learning rate and momentum factor

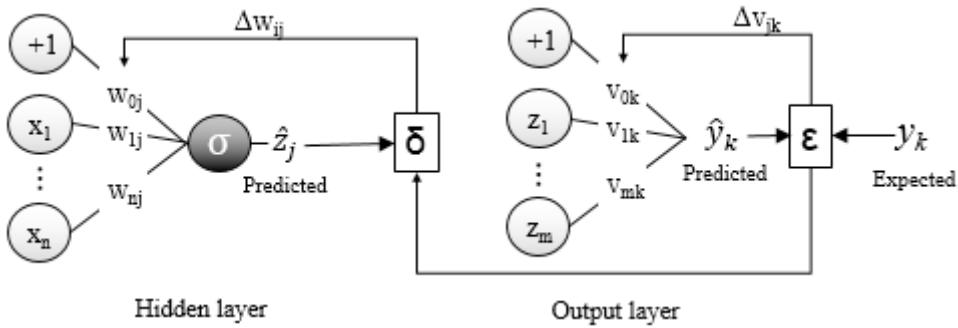
The computation of neural network weights using the gradient descent method with the momentum coefficient α is as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \frac{\partial \varepsilon_j^{(t)}}{\partial w_{ij}} + \alpha \Delta w_{ij}^{(t)}$$

The simplest version of the gradient descent algorithm (**M9**) is selected by simply setting the momentum factor α to zero in the generic (**M10**) mathematical expression.

The error propagation

The objective of the training of a perceptron is to minimize the loss or cumulative error for all the input observations as either the sum of squared errors or the cross entropy as computed at the output layer. The error ε_k for each output neuron y_k is computed as the difference between a predicted output value and label output value. The error cannot be computed on output values of the hidden layers z_j because the label values for those layers are unknown:



An illustration of the back-propagation algorithm

In the case of the sum of squared errors, the partial derivative of the cumulative error over each weight of the output layer is computed as the composition of the derivative of the square function and the derivative of the dot product of weights and the input z .

As mentioned earlier, the computation of the partial derivative of the error over the weights of the hidden layer is a bit tricky. Fortunately, the mathematical expression for the partial derivative can be written as the product of three partial derivatives:

- The derivative of the cumulative error ε over the output value y_k
- The derivative of the output value y_k over the hidden value z_j , knowing that the derivative of a sigmoid σ is $\sigma(1 - \sigma)$
- The derivative of the output of the hidden layer z_j over the weights w_{ij}

The decomposition of the partial derivative produces the following formulas for updating the synapses' weights for the output and hidden neurons by propagating the error (or loss) ε .

Output weights' adjustment

M11: The computation of delta δ and weight adjustment Δv for the output layer with the predicted value \tilde{y} and expected value y , and output z of the hidden layer is as follows:

$$\delta_{ih} = (\tilde{y}_i - y_i) \cdot z_h \quad \Delta v_{ih} = -\eta \cdot \delta_{ih}$$

Hidden weights' adjustment

M12: The computation of delta δ and weight adjustment Δw for the hidden layer with the predicted value \tilde{y} and expected value y , output z of the hidden layer, and the input value x is as follows:

$$\delta_{hi} = \sum_{j=0}^{k-1} \{(\tilde{y}_j - y_j) \cdot v_{jh}\} \cdot z_h (1 - z_h) \cdot x_i \quad \Delta w_{hi} = -\eta \delta_{hi}$$

The matrix δ_{ij} is defined by the `delta` matrix in the `Delta` class. It contains the basic parameters to be passed between layers, traversing the network from the output layer back to the input layer. The parameters are as follows:

- Initial loss or error computed at the output layer
- Matrix of the `delta` values from the current connection
- Weights or synapses of the downstream connection (or connection between the destination layer and the following layer)

The code will be as follows:

```
case class Delta(val loss: DblArray,
    val delta: DblMatrix = Array.empty[DblArray],
    val synapses: MLPConnSynapses = Array.empty[Array[MLPSynapse]] )
```

The first instance of the `Delta` class is generated for the output layer using the expected values y , then propagated to the preceding hidden layer in the `MLPNetwork.trainEpoch` method (line 24):

```
val diff = (x: Double, y: Double) => x - y
Delta(zipToArray(y, layers.last.output)(diff))
```

The **M11** mathematical expression is implemented by the `delta` method of the `MLPOutLayer` class:

```
def delta(error: DblArray, srcOut: DblArray,
    synapses: MLPConnSynapses): Delta = {

    val deltaMatrix = new ArrayBuffer[DblArray] //34
    val deltaValues = error.:(deltaMatrix) ( (m, l) => {
        m.append( srcOut.map( _*l) )
        m
    }) //35
    new Delta(error, deltaValues.toArray, synapses) //36
}
```

The method generates the matrix of delta values associated with the output layer (line 34). The **M11** formula is actually implemented by the fold over the `srcOut` output value (line 35). The new delta instances are returned to the `trainEpoch` method of `MLPNetwork` and backpropagated to the preceding hidden layer (line 36).

The `delta` method of the `MLPLayer` class implements the **M12** mathematical expression:

```
def delta(oldDelta: DblArray, srcOut: DblArray,
    synapses: MLPConnSynapses): Delta = {

    val deltaMatrix = new ArrayBuffer[(Double, DblArray)]
    val weights = synapses.map(_.map(_.-_1))
        .transpose.drop(1) //37

    val deltaValues = output.drop(1)
        .zipWithIndex.:(deltaMatrix){ // 38
        case (m, (zh, n)) => {
```

```

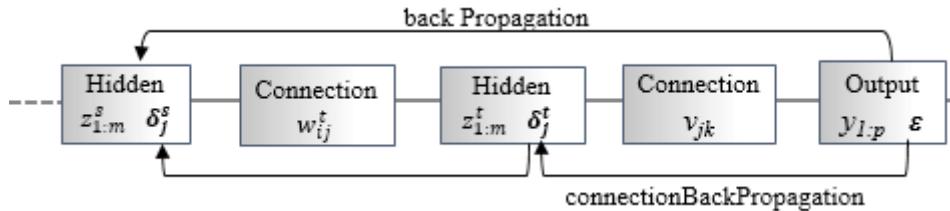
    val newDelta = inner(oldDelta, weights(n))*zh*(1.0 - zh)
    m.append((newDelta, srcOut.map(_ * newdelta) ))
    m
}
}.unzip
new Delta(deltaValues._1.toArray, deltaValues._2.toArray)//39
}

```

The implementation of the `delta` method is similar to the `MLPOutLayer.delta` method. It extracts the weights `v` from the output layer through transposition (line 37). The values of the delta matrix in the hidden connection is computed by applying the **M12** formula (line 38). The new delta instance is returned to the `trainEpoch` method (line 39) to be propagated to the preceding hidden layer if one exists.

The computational model

The computational model for the error backpropagation algorithm is very similar to the forward propagation of the input. The main difference is that the propagation of δ (delta) is performed from the output layer to the input layer. The following diagram illustrates the computational model of the backpropagation in the case of two hidden layers z_s and z_t :



An illustration of the backpropagation of the delta error

The `connectionBackPropagation` method propagates the error back from the output layer or one of the hidden layers to the preceding layer. It is a member of the `MLPConnection` class. The backpropagation of the output error across the entire network is managed by the `MLP` class.

It implements the two set of equations where `synapses(j)(i)._1` are the weights w_{ji} , `dst.delta` is the vector of the error derivative in the destination layer, and `src.delta` is the error derivative of the output in the source layer, as shown here:

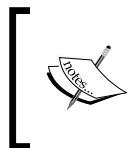
```

def connectionBackpropagation(delta: Delta): Delta = { //40
  val inSynapses = //41
    if( delta.synapses.length > 0) delta.synapses
    else synapses
}

```

```
    val delta = dst.delta(delta.loss, src.output,inSynapses) //42
    synapse = synapses.zipWithIndex.map{ //43
      case (synapsesj, j) => synapsesj.zipWithIndex.map{
        case ((w, dw), i) => {
          val ndw = config.eta*connectionDelta.delta(j)(i)
          (w + ndw - config.alpha*dw, ndw)
        }
      }
    }
    new Delta(connectionDelta.loss,
              connectionDelta.delta, synapses)
  }
```

The `connectionBackPropagation` method takes `delta` associated with the destination (`output`) layer as an argument (line 40). The output layer is the last layer of the network, and therefore, the synapses for the following connection is defined as an empty matrix of length zero (line 41). The method computes the new `delta` matrix for the hidden layer using the `delta.loss` error and output from the source layer, `src.output` (line 42). The weights (synapses) are updated using the gradient descent with the momentum factor as in the **M10** mathematical expression (line 43).



The adjustable learning rate

The computation of the new weights of a connection for each new epoch can be further improved by making the learning adjustable.

Step 3 – exit condition

The convergence criterion consists of evaluating the cumulative error (or loss) relevant to the operating mode (or problem) against a predefined `eps` convergence. The cumulative error is computed using either the sum of squares error formula (**M5**) or the cross-entropy formula (**M6** and **M7**). An alternative approach is to compute the difference of the cumulative error between two consecutive epochs and apply the `eps` convergence criteria as the exit condition.

Putting it all together

The `MLP` class is defined as a data transformation of the `ITransform` type using a model implicitly generated from a training set, `xt`, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 44).

The `MLP` algorithm takes the following parameters:

- `config`: This is the configuration of the algorithm
- `hidden`: This is an array of the size of the hidden layers if any
- `xt`: This is the time series of features used to train the model
- `expected`: This is the labeled output values for training purpose
- `mode`: This is the implicit operating mode or objective of the algorithm
- `f`: This is the implicit conversion from feature from type `T` to `Double`

The `V` type of the output of the prediction or classification method `|>` of this implicit transform is `DblArray` (line 45):

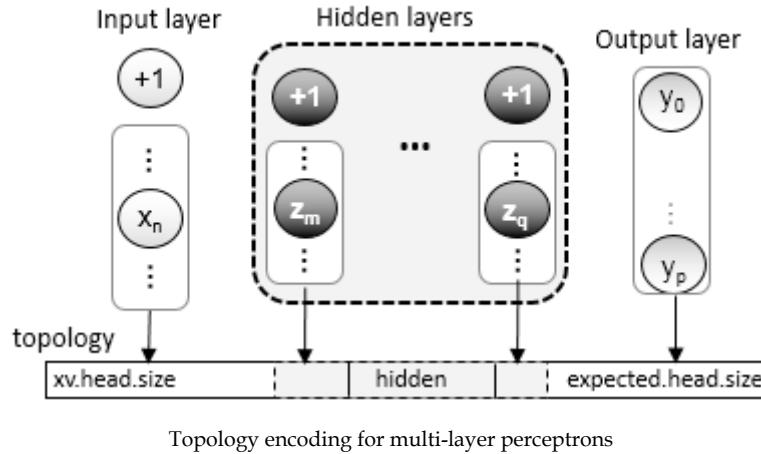
```
class MLP[T <: AnyVal] (config: MLPConfig,
  hidden: Array[Int] = Array.empty[Int],
  xt: XVSeries[T],
  expected: XVSeries[T])
  (implicit mode: MLPMode, f: T => Double)
extends ITransform[Array[T]](xt) with Monitor[Double] { //44

  type V = DblArray //45

  lazy val topology = if(hidden.length == 0)
    Array[Int](xt.head.size, expected.head.size)
  else Array[Int](xt.head.size) ++ hidden ++
    Array[Int](expected.head.size) //46

  val model: Option[MLPModel] = train
  def train: Option[MLPModel] //47
  override def |> : PartialFunction[Array[T], Try[V]]
}
```

The topology is created from the `xt` input variables, the expected values, and the configuration of hidden layers, if any (line 46). The generation of the topology from parameters of the `MLPNetwork` class is illustrated in the following diagram:



For instance, the topology of a neural network with three input variables: one output variable and two hidden layers of three neurons each is specified as `Array[Int] (4, 3, 3, 1)`. The model is generated through training by invoking the `train` method (line 47). Finally, the `|>` operator of the `ITransform` trait is used for classification, prediction, or regression, depending on the selected operating mode (line 48).

Training and classification

Once the training cycle or epoch is defined, it is merely a matter of defining and implementing a strategy to create a model using a sequence of data or time series.

Regularization

There are two approaches to find the most appropriate network architecture for a given classification or regression problem, which are follows:

- **Destructive tuning:** Starting with a large network, and then removing nodes, synapses, and hidden layers that have no impact on the sum of squared errors
- **Constructive tuning:** Starting with a small network, and then incrementally adding the nodes, synapses, and hidden layers that reduce the output error

The destructive tuning strategy removes the synapses by zeroing out their weights. This is commonly accomplished using regularization.

You have seen that regularization is a powerful technique to address overfitting in the case of the linear and logistic regression in the *Ridge regression* section in *Chapter 6, Regression and Regularization*. Neural networks can benefit from adding a regularization term to the sum of squared errors. The larger the regularization factor is, the more likely some weights will be reduced to zero, thus reducing the scale of the network [9:13].

The model generation

The `MLPModel` instance is created (trained) during the instantiation of the multilayer perceptron. The constructor iterates through the training cycles (or epochs) over all the data points of the `xt` time series, until the cumulative is smaller than the `eps` convergence criteria, as shown in the following code:

```
def train: Option[MLPModel] = {
    val network = new MLPNetwork(config, topology) //48
    val zi = xt.toVector.zip(expected.view) // 49

    Range(0, config.numEpochs).find( n => { //50
        val cumulErr = fisherYates(xt.size)
            .map(zi(_))
            .map{ case(x, e) => network.trainEpoch(x, e) }
            .sum/st.size //51
        cumulErr < config.eps //52
    }).map(_ => network.getModel)
}
```

The `train` method instantiates an MLP network using the configuration and topology as the input (line 48). The method executes multiple epochs until either the gradient descent with a momentum converges or the maximum number of allowed iterations is reached (line 50). At each epoch, the method shuffles the input values and labels using the Fisher-Yates algorithm, invokes the `MLPNetwork`.`trainEpoch` method, and computes the `cumulErr` cumulative error (line 51). This particular implementation compares the value of the cumulative error against the `eps` convergence criteria as the exit condition (line 52).

Tail recursive training of MLP

The training of the multilayer is implemented as an iterative process. It can be easily substituted with a tail recursion using weights and the cumulative error as the argument of the recursion.

Lazy views are used to reduce the unnecessary creation of objects (line 49).

The exit condition

In this implementation, the training initializes the model as `None` if it does not converge before the maximum number of epochs are reached. An alternative would be to generate a model even in the case of nonconvergence and add an accuracy metric to the model, as in our implementation of the support vector machine (refer to the *Training* section under *Support vector classifiers – SVC* in *Chapter 8, Kernel Models and Support Vector Machines*).

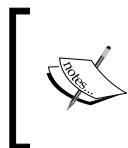
Once the model is created during the instantiation of the multilayer perceptron, it is available to predict or classify the class of a new observation.

The Fast Fisher-Yates shuffle

The *Step 5 – implementing the classifier* section under *Let's kick the tires* in *Chapter 1, Getting Started*, describes a home grown shuffling algorithm as an alternative to the `scala.util.Random.shuffle` method of the Scala standard library. This section describes an alternative shuffling mechanism known as the Fisher-Yates shuffling algorithm:

```
def fisherYates(n: Int): IndexedSeq[Int] = {  
  
    def fisherYates(seq: Seq[Int]): IndexedSeq[Int] = {  
        Random.setSeed(System.currentTimeMillis)  
        (0 until seq.size).map(i => {  
            var randomIdx: Int = i + Random.nextInt(seq.size-i) //53  
            seq(randomIdx) ^= seq(i) //54  
            seq(i) = seq(randomIdx) ^ seq(i)  
            seq(randomIdx) ^= (seq(i))  
            seq(i)  
        })  
    }  
  
    if( n <= 0) Array.empty[Int]  
    else  
        fisherYates(ArrayBuffer.tabulate(n)(n => n)) //55  
    }  
}
```

The Fisher-Yates algorithm creates an ordered sequence of integers (line 55), and swaps each integer with another integer, randomly selected from the remaining of the initial sequences (line 52). This implementation is particularly fast because the integers are swapped in place using the bit operator, also known as **bitwise swap** (line 54).



Tail recursive implementation of Fisher-Yates

The Fisher-Yates shuffling algorithm can be implemented using a tail recursion instead of an iteration.

Prediction

The `| >` data transformation implements the runtime classification/prediction. It returns the predicted value that is normalized as a probability if the model was successfully trained and `None` otherwise. The methods invoke the forward prediction function of `MLPNetwork` (line 53):

```
override def |> : PartialFunction[Array[T], Try[V]] = {
  case x: Array[T] if(isModel && x.size == dimension(xt)) =>
    Try(MLPNetwork(config, topology, model).predict(x)) //56
}
```

The `predict` method of `MLPNetwork` computes the output values from an input `x` using the forward propagation as follows:

```
def predict(x: DblArray): DblArray = {
  layers.head.set(x)
  connections.foreach(_.connectionForwardPropagation)
  layers.last.output
}
```

Model fitness

The fitness of a model measures how well the model fits the training set. A model with a high-degree of fitness will likely overfit. The `fit` fitness method computes the mean squared errors of the predicted values against the labels (or expected values) of the training set. The method returns the percentage of observations for which the prediction value is correct, using the higher order `count` method:

```
def fit(threshold: Double): Option[Double] = model.map(m =>
  xt.map(MLPNetwork(config, topology, Some(m)).predict(_))
    .zip(expected)
```

```
.count{case (y, e) =>mse(y, e.map(_.toDouble))< threshold }  
/xt.size.toDouble  
)
```

Model fitness versus accuracy

The fitness of a model against the training set reflects the degree the model fit the training set. The computation of the fitness does not involve a validation set. Quality parameters such as accuracy, precision, or recall measures the reliability or quality of the model against a validation set.



Our `MLP` class is now ready to tackle some classification challenges.

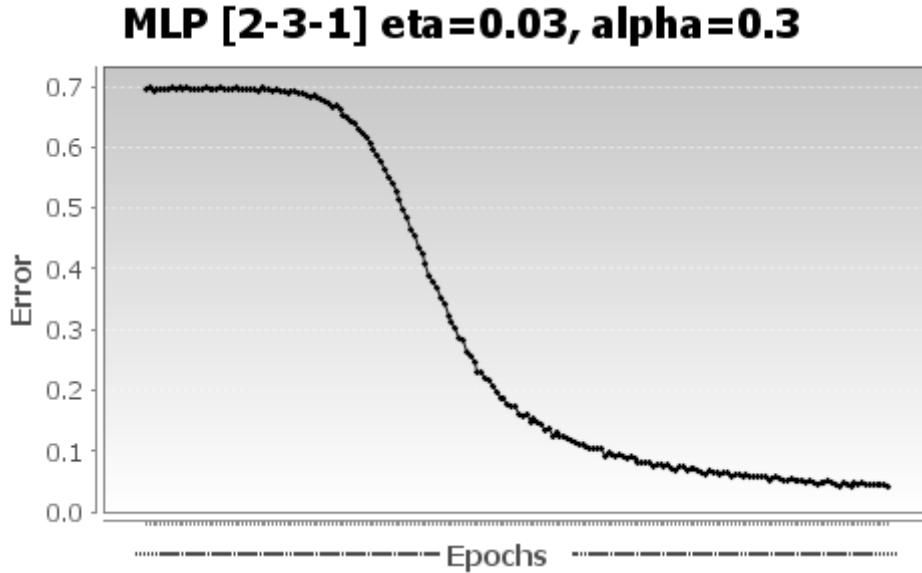
Evaluation

Before applying our multilayer perceptron to understand fluctuations in the currency market exchanges, let's get acquainted with some of the key learning parameters introduced in the first section.

The execution profile

Let's take a look at the convergence of the training of the multiple layer perceptron. The monitor trait (refer to the *Monitor* section under *Utility classes* in the *Appendix A, Basic Concepts*) collects and displays some execution parameters. We select to extract the profile for the convergence of the multiple layer perceptron using the difference of the backpropagation errors between two consecutive episodes (or epochs).

The test profiles the convergence of the MLP using a learning rate of $\eta = 0.03$ and a momentum factor of $\alpha = 0.3$ for a multilayer perceptron with two input values: one hidden layer with three nodes and one output value. The test relies on synthetically generated random values:



The execution profile for the cumulative error for MLP

Impact of the learning rate

The purpose of the first exercise is to evaluate the impact of the learning rate η on the convergence of the training epoch, as measured by the cumulative error of all output variables. The observations xt (with respect to the labeled output yt) are synthetically generated using several noisy patterns such as f_1 (line 57) and f_2 functions (line 58), as follows:

```

def f1(x: Double): DblArray = Array[Double]( //57
  0.1+ 0.5*Random.nextDouble, 0.6*Random.nextDouble)
def f2(x: Double): DblArray = Array[Double]( //58
  0.6 + 0.4*Random.nextDouble, 1.0 - 0.5*Random.nextDouble)

val HALF_TEST_SIZE = (TEST_SIZE>>1)
val xt = Vector.tabulate(TEST_SIZE)(n => //59
  if( n <HALF_TEST_SIZE) f1(n) else f2(n -HALF_TEST_SIZE))
val yt = Vector.tabulate(TEST_SIZE)(n =>
  if( n < HALF_TEST_SIZE) Array[Double](0.0)
  else Array[Double](1.0) ) //60
  
```

The input values, `xt`, are synthetically generated by the `f1` function for half of the dataset and by the `f2` function for the other half (line 59). The data generator for the expected values `yt` assigns the label 0.0 for the input values generated with the `f1` function and 1.0 for the input values created with `f2` (line 60).

The test is run with a sample of size `TEST_SIZE` data points over a maximum of `NUM_EPOCHS` epochs, a single hidden layer of `HIDDEN`.head neurons with no `softmax` transformation, and the following MLP parameters:

```
val ALPHA = 0.3
val ETA = 0.03
val HIDDEN = Array[Int] (3)
val NUM_EPOCHS = 200
val TEST_SIZE = 12000
val EPS = 1e-7

def testEta(eta: Double,
           xt: XVSeries[Double],
           yt: XVSeries[Double]): Option[(ArrayBuffer[Double], String)] = {

  implicit val mode = new MLPBinClassifier //61
  val config = MLPConfig(ALPHA, eta, NUM_EPOCHS, EPS)
  MLP[Double](config, HIDDEN, xt, yt)
    .counters("err").map( (_, s"eta=$eta")) //62
}
```

The `testEta` method generates the profile or errors given different values of `eta`.

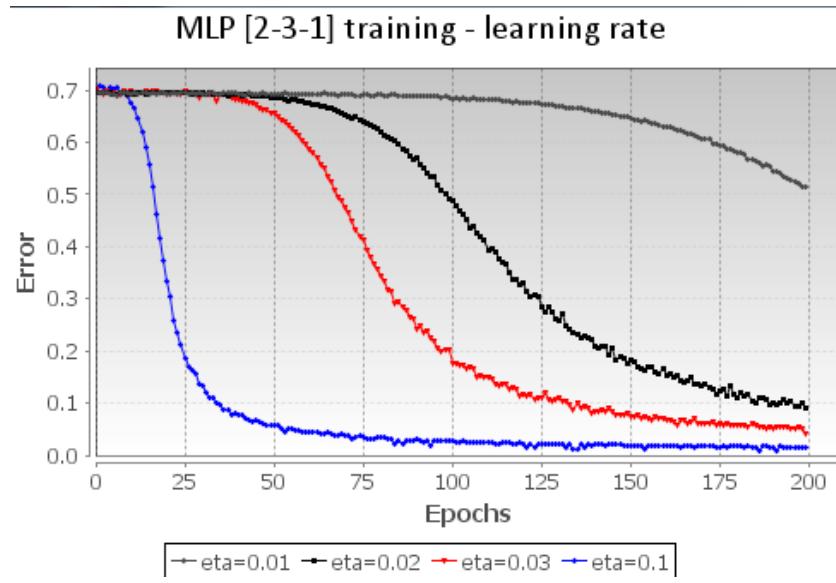
The operating mode has to be implicitly defined prior to the instantiation of the `MLP` class (line 61). It is set as a binomial classifier of the `MLPBinClassifier` type. The execution profile data is collected by the `counters` method of the `Monitor` trait (line 62) (refer to the *Monitor* section under *Utility classes* in the *Appendix A, Basic Concepts*).

The driver code for evaluating the impact of the learning rate on the convergence of the multilayer perceptron is quite simple:

```
val etaValues = List[Double] (0.01, 0.02, 0.03, 0.1)
val data = etaValues.flatMap( testEta(_, xt, yt))
  .map{ case(x, s) => (x.toVector, s) }

val legend = new Legend("Err",
  "MLP [2-3-1] training - learning rate", "Epochs", "Error")
LinePlot.display(data, legend, new LightPlotTheme)
```

The profile is created with the JFreeChart library and displayed in the following chart:



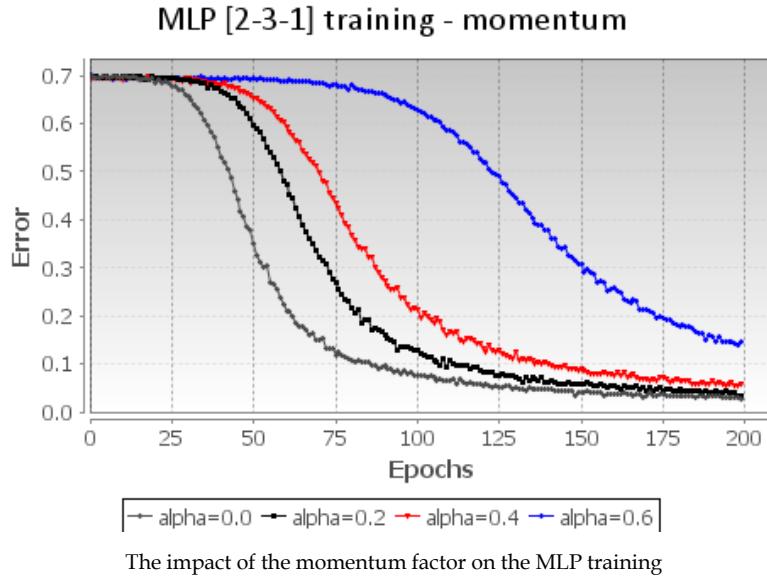
The impact of the learning rate on the MLP training

The chart illustrates that the MLP model training converges a lot faster with a larger value of learning rate. You need to keep in mind, however, that a very steep learning rate may lock the training process into a local minimum for the cumulative error, generating weights with lesser accuracy. The same configuration parameters are used to evaluate the impact of the momentum factor on the convergence of the gradient descent algorithm.

The impact of the momentum factor

Let's quantify the impact of the momentum factor a on the convergence of the training process toward an optimal model (synapse weights). The testing code is very similar to the evaluation of the impact of the learning rate.

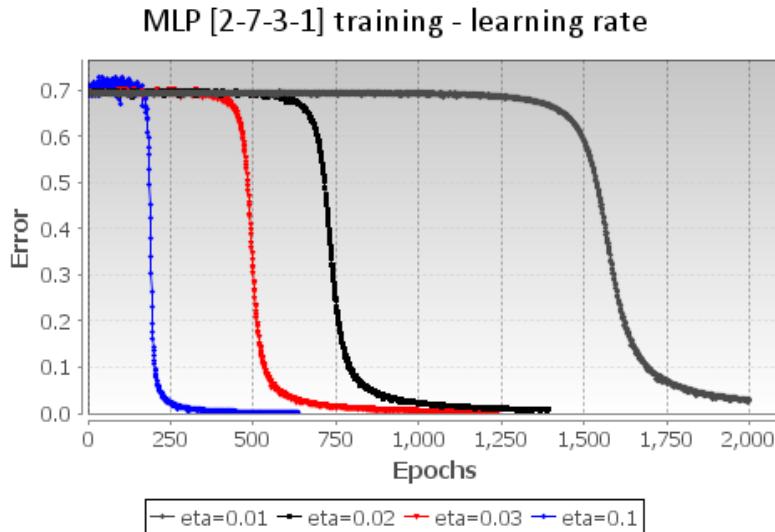
The cumulative error for the entire time series is plotted in the following graph:



The preceding graph shows that the rate of the mean square error decreases as the momentum factor increases. In other words, the momentum factor has a positive although limited impact on the convergence of the gradient descent.

The impact of the number of hidden layers

Let's consider a multilayer perceptron with two hidden layers (7 and 3 neurons). The execution profile for the training shows that the cumulative error of the output converges abruptly after several epochs for which the descent gradient failed to find a direction:



The execution profile of training of an MLP with two hidden layers

Let's apply our newfound knowledge regarding neural networks and the classification of variables that impact the exchange rate of a certain currency.

Test case

Neural networks have been used in financial applications from risk management in mortgage applications and hedging strategies for commodities pricing, to predictive modeling of the financial markets [9:14].

The objective of the test case is to understand the correlation factors between the exchange rate of some currencies, the spot price of gold, and the S&P 500 index. For this exercise, we will use the following **exchange-traded funds (ETFs)** as proxies for the exchange rate of currencies:

- **FXA:** This is the rate of an Australian dollar in US dollar
- **FXB:** This is the rate of a British pound in US dollar
- **FXE:** This is the rate of an Euro in US dollar
- **FXC:** This is the rate of a Canadian dollar in US dollar
- **FXF:** This is the rate of a Swiss franc in US dollar
- **FXY:** This is the rate of a Japanese yen in US dollar
- **CYB:** This is the rate of a Chinese yuan in US dollar
- **SPY:** This is the S&P 500 index
- **GLD:** This is the price of gold in US dollar

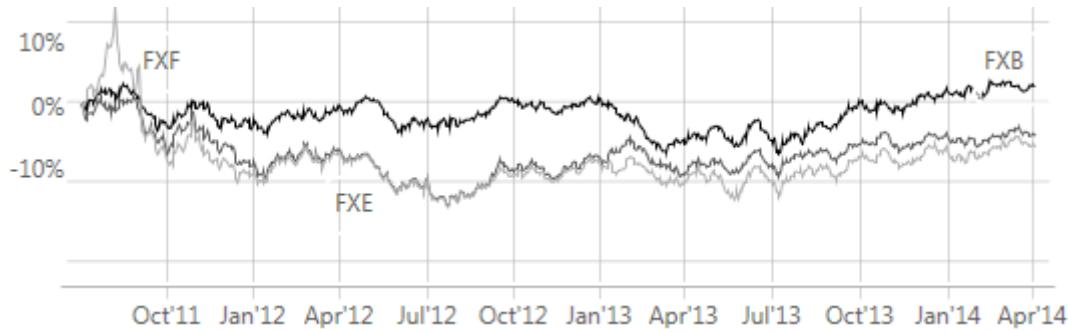
Practically, the problem to solve is to extract one or more regressive models that link one ETFs y with a basket of other ETFs $\{x_i\}$ $y=f(x_i)$. For example, is there a relation between the exchange rate of the Japanese yen (FXY) and a combination of the spot price for gold (GLD), exchange rate of the Euro in US dollar (FXE), the exchange rate of the Australian dollar in US dollar (FXA), and so on? If so, the regression f will be defined as $FXY = f(GLD, FXE, FXA)$.

The following two charts visualize the fluctuation between currencies over a period of two and a half years. The first chart displays an initial group of potentially correlated ETFs:



An example of correlated currency-based ETFs

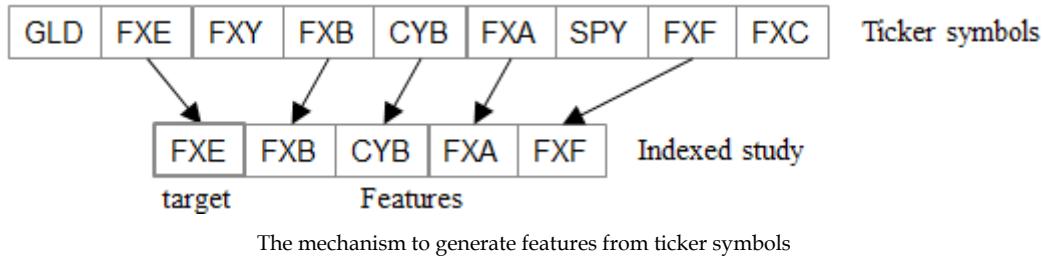
The second chart displays another group of currency-related ETFs that shares a similar price action behavior. Neural networks do not provide any analytical representation of their internal reasoning; therefore, a *visual* correlation can be extremely useful to novice engineers to validate their models:



An example of correlated currency-based ETFs

A very simple approach for finding any correlation between the movement of the currency exchange rates and the gold spot price is to select one ticker symbol as the target and a subset of other currency-based ETFs as features.

Let's consider the following problem: finding the correlation between the price of FXE and a range of currencies FXB, CYB, FXA, and FXC, as illustrated in the following diagram:



Implementation

The first step is to define the configuration parameter for the MLP classifier, as follows:

```
val path = "resources/data/chap9/"
val ALPHA = 0.8;
val ETA = 0.01
val NUM_EPOCHS = 250
val EPS = 1e-3
val THRESHOLD = 0.12
val hiddens = Array[Int](7, 7) //59
```

Besides the learning parameters, the network is initialized with multiple topology configurations (line 59).

Next, let's create the search space of the prices of all the ETFs used in the analysis:

```
val symbols = Array[String](
  "FXE", "FXA", "SPY", "GLD", "FXB", "FXF", "FXC", "FXY", "CYB"
)
val STUDIES = List[Array[String]]( //60
  Array[String]("FXY", "FXC", "GLD", "FXA"),
  Array[String]("FXE", "FXF", "FXB", "CYB"),
  Array[String]("FXE", "FXC", "GLD", "FXA", "FXY", "FXB"),
  Array[String]("FXC", "FXY", "FXA"),
  Array[String]("CYB", "GLD", "FXY"),
  symbols
)
```

The purpose of the test is to evaluate and compare seven different portfolios or studies (line 60). The closing prices of all the ETFs over a period of 3 years are extracted from the Google Financial tables, using the `GoogleFinancials` extractor for a basket of ETFs (line 61):

```
val prices = symbols.map(s => DataSource(s"$path$s.csv"))
    .flatMap(_.get(close).toOption) //61
```

The next step consists of implementing the mechanism to extract the target and the features from a basket of ETFs or studies introduced in the previous paragraph. Let's consider the following study as the list of ETF ticker symbols:

```
val study = Array[String] ("FXE", "FXF", "FXB", "CYB")
```

The first element of the study, `FXE`, is the labeled output; the remaining three elements are observed features. For this study, the network architecture has three input variables (`FXF`, `FXB`, and `CYB`) and one output variable, `FXE`:

```
val obs = symbols.flatMap(index.get(_))
    .map(prices(_).toArray) //62
val xv = obs.drop(1).transpose //63
val expected = Array[DblArray](obs.head).transpose //64
```

The set of observations, `obs`, is built using an index (line 62). By convention, the first observation is selected as the label data and the remaining studies as the features for training. As the observations are loaded as an array of time series, the time features of the series is computed using `transpose` (line 63). The single target output variable has to be converted into a matrix before transposition (line 64).

Ultimately, the model is built through instantiation of the `MLP` class:

```
implicit val mode = new MLPBinClassifier //65
val classifier = MLP[Double](config, hiddenLayers, xv, expected)
classifier.fit(THRESHOLD)
```

The objective or operating `mode` is implicitly defined as an `MLP` binary classifier, `MLPBinClassifier` (line 65). The `MLP`.`fit` method is defined in the *Training and classification* section.

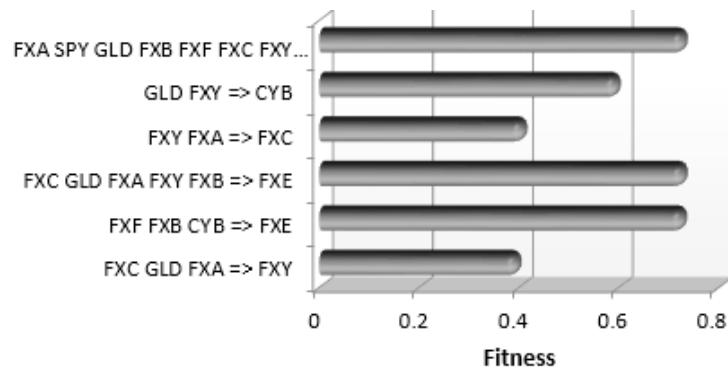
Evaluation of models

The test consists of evaluating six different models to determine which ones provide the most reliable correlation. It is critical to ensure that the result is somewhat independent of the architecture of the neural network. Different architectures are evaluated as part of the test.

The following charts compare the models for two architectures:

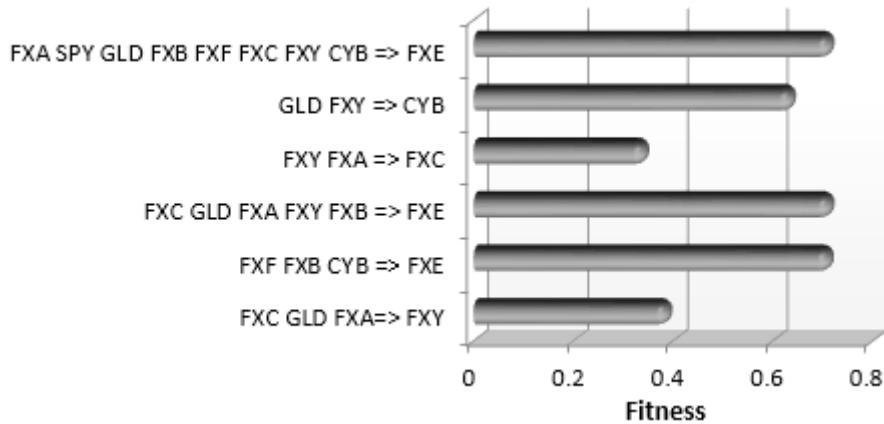
- Two hidden layers with four nodes each
- Three hidden layers with eight (with respect to five and six) nodes

This first chart visualizes the fitness of the six regression models with an architecture consisting of a variable number of inputs (2, 7): one output variable and two hidden layers of four nodes each. The features (ETF symbols) are listed on the left-hand side of the arrow \Rightarrow along the y axis. The symbol on the right-hand side of the arrow is the expected output value:



The accuracy of MLP with two hidden layers of four nodes each

The following chart displays the fitness of the six regression models for an architecture with three hidden layers of eight, five, and six nodes, respectively:



The accuracy of MLP with three hidden layers with 8, 5, and 6 nodes, respectively

The two network architectures shared a lot of similarity; in both cases, the fittest regression models are as follows:

- $FXE = f(FXA, SPY, GLD, FXB, FXF, FXD, FXY, CYB)$
- $FXE = g(FXC, GLD, FXA, FXY, FXB)$
- $FXE = h(FXF, FXB, CYB)$

On the other hand, the prediction of the Canadian dollar to US dollar's exchange rate (FXC) using the exchange rate for the Japanese yen (FXY) and the Australian dollar (FXA) is poor with both the configurations.

The empirical evaluation

These empirical tests use a simple accuracy metric. A formal comparison of the regression models will systematically analyze every combination of input and output variables. The evaluation will also compute the precision, the recall, and the F1 score for each of those models (refer to the *Key quality metrics* section under *Validation* in the *Assessing a model* section in *Chapter 2, Hello World!*!).

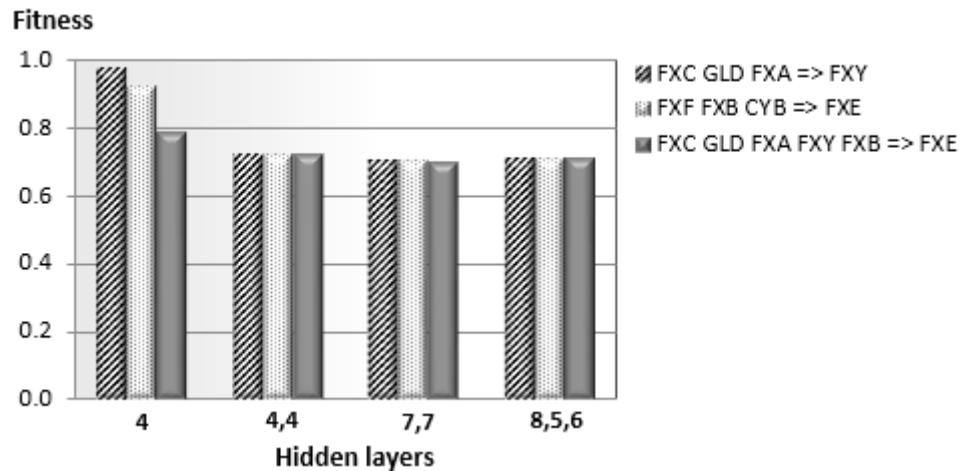
Impact of the hidden layers' architecture

The next test consists of evaluating the impact of the hidden layer(s) of configuration on the accuracy of three models: $FXF, FXB, CYB \Rightarrow FXE$, $FCX, GLD, FXA \Rightarrow FXY$, and $FXC, GLD, FXA, FXY, FXB \Rightarrow FXE$. For this test, the accuracy is computed by selecting a subset of the training data as a test sample, for the sake of convenience. The objective of the test is to compare different network architectures using some metrics, and not to estimate the absolute accuracy of each model.

The four network configurations are as follows:

- A single hidden layer with four nodes
- Two hidden layers with four nodes each
- Two hidden layers with seven nodes each
- Three hidden layers with eight, five, and six nodes

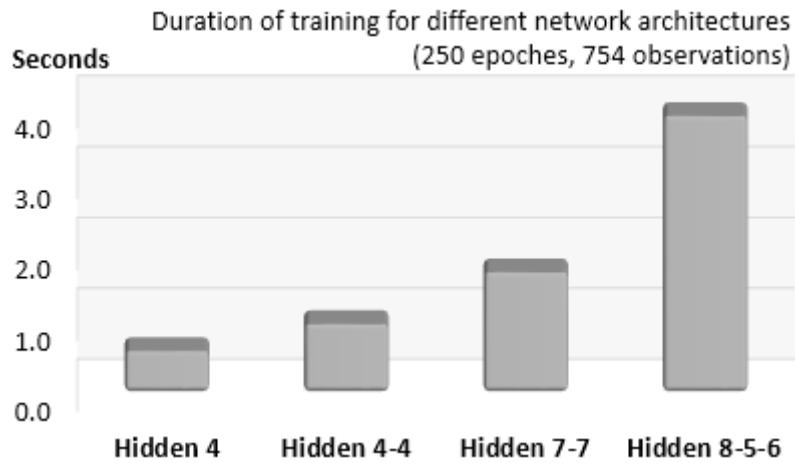
Let's take a look at the following graph:



The impact of the hidden layers' architecture on the MLP accuracy

The complex neural network architecture with two or more hidden layers generates weights with similar accuracy. The four-node single hidden layer architecture generates the highest accuracy. The computation of the accuracy using a formal cross-validation technique would generate a lower accuracy number.

Finally, we take a look at the impact of the complexity of the network on the duration of the training, as shown in the following graph:



The impact of the hidden layers' architecture on the duration of training

Not surprisingly, the time complexity increases significantly with the number of hidden layers and number of nodes.

Convolution neural networks

This section is provided as a brief introduction to convolution neural networks without the Scala implementation.

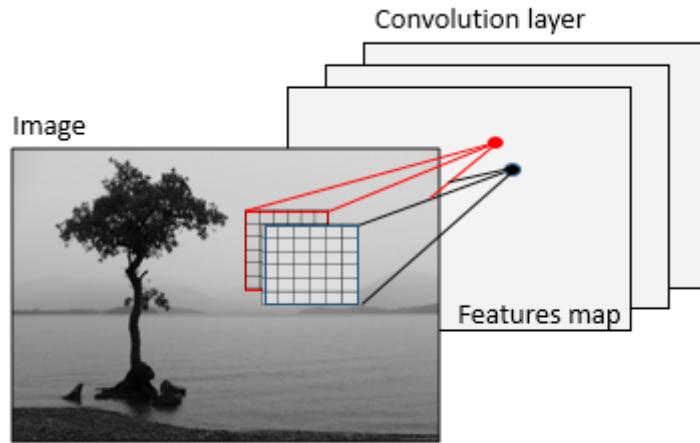
So far, the layers of perceptrons were organized as a fully connected network. It is clear that the number of synapses or weights increases significantly as the number and size of hidden layers increases. For instance, a network for a features set of dimension 6, 3 hidden layers of 64 nodes each, and one output value requires $7*64 + 2*65*64 + 65*1 = 8833$ weights!

Applications such as image or character recognition require very large features set, making training a fully connected layered perceptron very computational intensive. Moreover, these applications need to convey spatial information such as the proximity of pixels as part of the features vector.

A recent approach, known as **convolution neural networks**, consists of limiting the number of nodes in the hidden layers a input node is connected to. In other words, the methodology leverages spatial localization to reduce the complexity of connectivity between the input and the hidden layer [9:15]. The subset of input nodes connected to a single neuron in the hidden layer is known as the **local receptive fields**.

Local receptive fields

The neuron of the hidden layer learns from the local receptive fields or subimage of n by n pixels, each of those pixels being an input value. The next local receptive field, which is shifted by one pixel in any direction, is connected to the next neuron in the first hidden layer. The first hidden layer is known as the **convolution layer**. An illustration of the mapping between the input (image) and the first hidden layer (convolution layer) is as follows:



The generation of a convolution layer from an image

It would make sense that each n by n local receptive field has a bias element (+1) that connects to the hidden neuron. However, the extra complexity does not lead to a more accurate model, and therefore, the bias is shared across the neurons of the convolution layer.

Sharing of weights

The local receptive fields representing a small section of the image are generated by shifting the fields by one pixel (up, down, left, or right). Therefore, the weights associated with the local receptive fields are also shared across the neurons in the hidden layer. As a matter of fact, the same feature such as an image color or edge can be detected in many pixels across the image. The maps between the input features and neurons in the hidden layer, known as **features maps**, share weights across the convolution layer. The output is computed using the activation function.

Tanh versus the sigmoid activation

The sigmoid is predominately used in the examples related to the multilayer perceptron as the activation function for the hidden layer. The hyperbolic tangent function is commonly used for convolution networks.

Convolution layers

The output computed from the features maps is expressed as a convolution that is similar to the convolution used in a discrete Fourier transformed-based filter (refer to **M11** mathematical expression in the *DFT-based filtering* section under *Fourier analysis* in *Chapter 3, Data Preprocessing*). The activation function that computes the output in the hidden layer has to be modified to take into account the local receptive fields.

The activation of a convolution neural network

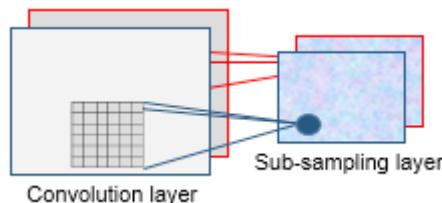
M13: The output value z_j for a shared bias w_0 , an activation function σ , a local receptive field of n by n pixels, input values x_{ij} and weights w_{uv} associated with a features map is given by:

$$\tilde{z}_j = \sigma \left(w_0 + \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} w_{u,v} x_{j+u,i+v} \right)$$

The next step in building the neural network would be to use the output of the convolution layer to a full connected hidden layer. However, the features maps in the convolution layer are usually similar so that they can be reduced to a smaller set of outputs using an intermediate layer known as subsampling layer [9:16].

Subsampling layers

Each features map in the convolution layer is reduced or condensed into a smaller features map. The layer composed of these smaller features map is known as the subsampling layer. The purpose of the sampling is to reduce the sensitivity of the weights to any minute changes in the image between adjacent pixels. The sharing of weights reduces the sensitivity to any nonsignificant changes in the image:

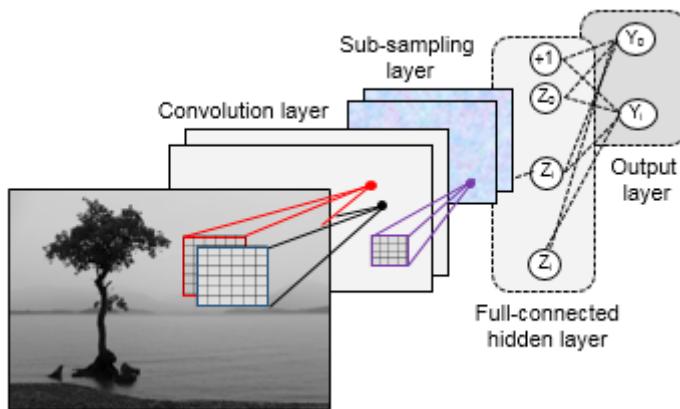


The connectivity between features map from a convolution to a subsampling layer

The subsampling layer is sometimes referred to as the **pooling layer**.

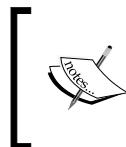
Putting it all together

The last layer of the convolution neural network is the fully connected hidden layer and output layer, subjected to the same transformative formulas as the traditional multilayer perceptron. The output values can be computed using a linear product or a softmax function:



An overview of a convolution neural network

The error backpropagation algorithm described in the *Step 2 – error back propagation* section has to be modified to support the features map [9:17].



The architecture of convolution networks

Deep convolution neural networks have multiple sequences of convolution layers and subsampling layers and may have more than one fully connection hidden layer.



Benefits and limitations

The advantages and disadvantages of neural networks depend on which other machine learning methods they are compared to. However, neural-network-based classifiers, particularly the multilayer perceptron using the error backpropagation, have some obvious advantages, which are as follows:

- The mathematical foundation of a neural network does not require expertise in dynamic programming or linear algebra, beyond the basic gradient descent algorithm.
- A neural network can perform tasks that a linear algorithm cannot.

- An MLP is usually reliable for highly dynamic and nonlinear processes. Contrary to the support vector machines, they do not require us to increase the problem dimension through kernelization.
- An MLP does not make any assumption on linearity, variable independence, or normality.
- The execution of training of an MLP lends itself to concurrent processing quite well for online training. In most architecture, the algorithm can continue even if a node in the network fails (refer to the *Apache Spark* section in *Chapter 12, Scalable Frameworks*).

However, as with any machine learning algorithm, neural networks have their detractors. The most documented limitations are as follows:

- MLP models are black boxes for which the association between features and classes may not be easily described and understood.
- An MLP requires a lengthy training process, especially using the batch training strategy. For example, a two-layer network has a time complexity (number of multiplications) of $O(n.m.p.N.e)$ for n input variables, m hidden neurons, p output values, N observations, and e epochs. It is not uncommon that a solution emerges after thousands of epochs. The online training strategy using a momentum factor tends to converge faster and requires a smaller number of epochs than the batch process.
- Tuning the configuration parameters, such as optimization of the learning rate and momentum factors, selection of the most appropriate activation method, and the cumulative error formula can turn into a lengthy process.
- Estimating the minimum size of the training set required to generate an accurate model and limiting the computation time is not obvious.
- A neural network cannot be incrementally retrained. Any new labeled data requires the execution of several training epochs.

 **Other types of neural networks**

This chapter covers the multilayer perceptron and introduces the concept of a convolution neural network. There are many more types of neural networks, such as recurrent networks and mixture density networks.

Summary

This concludes not only the journey inside the multilayer perceptron, but also the introduction of the supervised learning algorithms. In this chapter, you learned:

- The components and architecture of artificial neural networks
- The stages of the training cycle (or epoch) for the backpropagation multilayer perceptron
- How to implement an MLP from the ground up in Scala
- The numerous configuration parameters and options available to create the MLP classification or regression model.
- To evaluate the impact of the learning rate and the gradient descent momentum factor on the convergence of the training process.
- How to apply a multilayer perceptron to the financial analysis of the fluctuation of currencies
- An overview of the convolution neural network

The next chapter will introduce the concept of genetic algorithms with a complete implementation in Scala. Although, strictly speaking, genetic algorithms do not belong to the family of machine learning algorithms, they play a crucial role in the optimization of nonlinear, nondifferentiable problems, and the selection of strong classifiers within ensembles.

10

Genetic Algorithms

This chapter introduces the concept of evolutionary computing. Algorithms derived from the theory of evolution are particularly efficient in solving large combinatorial or **NP problems**. Evolutionary computing has been pioneered by John Holland [10:1] and David Goldberg [10:2]. Their findings should be of interest to anyone eager to learn about the foundation of **genetic algorithms (GA)** and **artificial life**.

This chapter covers the following topics:

- The origin of evolutionary computing
- The theoretical foundation of genetic algorithms
- Advantages and limitations of genetic algorithms

From a practical perspective, you will learn how to:

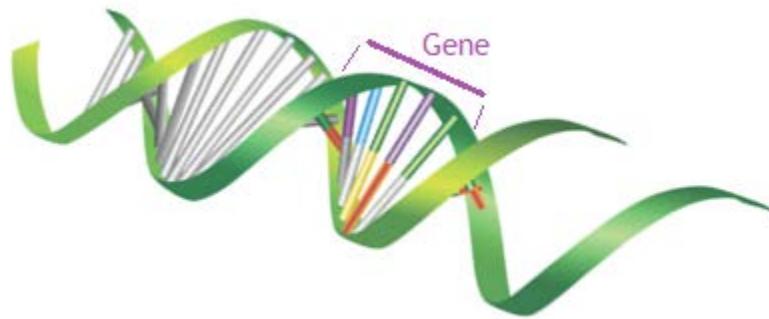
- Apply genetic algorithms to leverage technical analysis of market price and volume movement to predict future returns
- Evaluate or estimate the search space
- Encode solutions in the binary format using either hierarchical or flat addressing
- Tune some of the genetic operators
- Create and evaluate fitness functions

Evolution

The **theory of evolution**, enunciated by Charles Darwin, describes the morphological adaptation of living organisms [10:3].

The origin

The **Darwinian** process consists of optimizing the morphology of organisms to adapt to the harshest environments—hydrodynamic optimization for fishes, aerodynamic for birds, or stealth skills for predators. The following diagram shows a gene:

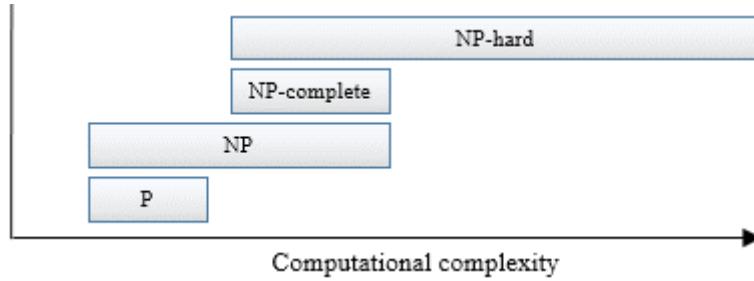


The **population** of organisms varies over time. The number of individuals within a population changes, sometimes dramatically. These variations are usually associated with the abundance or lack of predators and prey as well as the changing environment. Only the fittest organisms within the population can survive over time by adapting quickly to sudden changes in living environments and new constraints.

NP problems

NP stands for nondeterministic polynomial time. The NP problems' concept relates to the theory of computation and more precisely, time and space complexity. The categories of NP problems are as follows:

- **P-problems** (or P decision problems): For these problems, the resolution on a deterministic Turing machine (computer) takes a deterministic polynomial time.
- **NP problems**: These problems can be resolved in a polynomial time on nondeterministic machines.
- **NP-complete problems**: These are NP-hard problems that are reduced to NP problems for which the solution takes a deterministic polynomial time. These types of problems may be difficult to solve but their solutions can be validated.
- **NP-hard problems**: These problems have solutions that may not be found in polynomial time.



The categorization of NP problems using computational complexity

Problems such as the traveling salesman, floor shop scheduling, the computation of a graph K-minimum spanning tree, map coloring, or cyclic ordering have a search execution time that is a nondeterministic polynomial, ranging from $n!$ to 2^n for a population of n elements [10:4].

NP problems cannot always be solved using analytical methods because of the computation overhead—even in the case of a model, it relies on differentiable functions. Genetic algorithms were invented by John Holland in the 1970s, and they derived their properties from the theory of evolution of Darwin to tackle NP and NP-complete problems.

Evolutionary computing

A living organism consists of cells that contain identical chromosomes.

Chromosomes are strands of **DNA** and serve as a model for the whole organism. A chromosome consists of **genes** that are blocks of DNA and encode a specific protein.

Recombination (or crossover) is the first stage of reproduction. Genes from parents generate the whole new chromosome (**offspring**) that can be mutated. During mutation, one or more elements, also known as individual bases of the DNA strand or chromosomes, are changed. These changes are mainly caused by errors that occur when the genes from parents are being passed on to their offspring. The success of an organism in its life measures its fitness [10:5].

Genetic algorithms use reproduction to evolve a population of possible solutions to a problem.

Genetic algorithms and machine learning

The practical purpose of a genetic algorithm as an optimization technique is to solve problems by finding the most relevant or fittest solution among a set or group of solutions. Genetic algorithms have many applications in machine learning, which are as follows:

- **Discrete model parameters:** Genetic algorithms are particularly effective in finding the set of discrete parameters that maximizes the log likelihood. For example, the colorization of a black and white movie relies on a large but finite set of transformations from shades of grey to the RGB color scheme. The search space is composed of the different transformations and the objective function is the quality of the colorized version of the movie.
- **Reinforcement learning:** Systems that select the most appropriate rules or policies to match a given dataset rely on genetic algorithms to evolve the set of rules over time. The search space or population is the set of candidate rules, and the objective function is the credit or reward for an action triggered by these rules (refer to *Chapter 11, Reinforcement Learning*).
- **The neural network architecture:** A genetic algorithm drives the evaluation of different configurations of networks. The search space consists of different combinations of hidden layers and the size of those layers. The fitness or objective function is the sum of the squared errors.
- **Ensemble learning [10:6]:** A genetic algorithm can weed out the weak learners among a set of classifiers in order to improve the quality of the prediction.

Genetic algorithm components

Genetic algorithms have the following three components:

- **Genetic encoding (and decoding):** This is the conversion of a solution candidate and its components into the binary format (an array of bits or a string of 0 and 1 characters)
- **Genetic operations:** This is the application of a set of operators to extract the best (most genetically fit) candidates (chromosomes)
- **Genetic fitness functions:** This is the evaluation of the fittest candidate using an objective function

Encodings and the fitness function are problem dependent. Genetic operators are not.

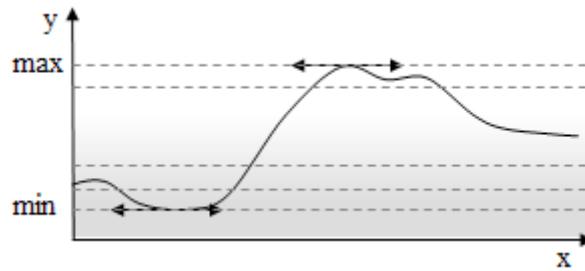
Encoding

Let's consider the optimization problem in machine learning that consists of maximizing the log likelihood or minimizing the loss function. The goal is to compute the parameters or weights, $w=\{w_i\}$, that minimize or maximize a function $f(w)$. In the case of a nonlinear model, variables may depend on other variables, which make the optimization problem particularly challenging.

Value encoding

The genetic algorithm manipulates variables as bits or bit strings. The conversion of a variable into a bit string is known as encoding. In the case where the variable is continuous, the conversion is known as **quantization** or **discretization**. Each type of variable has a unique encoding scheme, as follows:

- Boolean values are easily encoded with 1 bit: 0 for false and 1 for true.
- Continuous variables are quantized or discretized in a fashion similar to the conversion of an analog to a digital signal. Let's consider the function with a maximum **max** (similarly **min** for minimum) over a range of values, encoded with $n = 16$ bits:



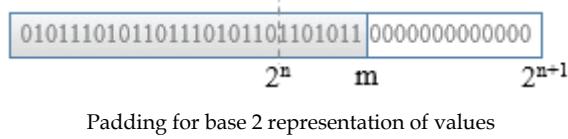
An illustration of quantization of a continuous variable $y = f(x)$

The step size of the discretization is computed as (M1):

$$\text{step} = \frac{\text{max} - \text{min}}{2^n}$$

The step size of the quantization of the $\sin e y = \sin(x)$ in 16 bits is 1.524e-5.

Discrete or categorical variables are a bit more challenging to encode to bits. At a minimum, all the discrete values have to be accounted for. However, there is no guarantee that the number of variables will coincide with the bits boundary:



In this case, the next exponent, $n+1$, defines the minimum number of bits required to represent the set of values: $n = \log_2(m).toInt + 1$. A discrete variable with 19 values requires 5 bits. The remaining bits are set to an arbitrary value (0, NaN, and so on) depending on the problem. This procedure is known as **padding**.

Encoding is as much art as it is science. For each encoding function, you need a decoding function to convert the bits representation back to actual values.

Predicate encoding

A predicate for a variable x is a relation defined as a x operator [target]; for instance, *unit cost < [9\$]*, *temperature = [82F]*, or *Movie rating is [3 stars]*.

The simplest encoding scheme for predicates is as follows:

- **Variables** are encoded as a category or type (for example, temperature, barometric pressure, and so on) because there are a finite number of variables in any model
- **Operators** are encoded as discrete types
- **Values** are encoded as either discrete or continuous values



Encoding format for predicates

There are many approaches for encoding a predicate in a bits string. For instance, the format {operator, left-operand, and right-operand} is useful because it allows you to encode a binary tree. The entire rule, *IF predicate THEN action*, can be encoded with the action being represented as a discrete or categorical value.

Solution encoding

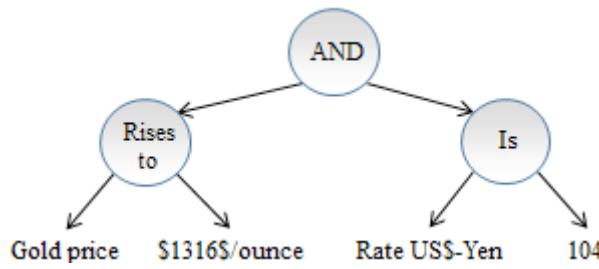
The solution encoding approach describes the solution to a problem as an unordered sequence of predicates. Let's consider the following rule:

```
IF {Gold price rises to [1316$/ounce]} AND
    {US$/Yen rate is [104]}).
THEN {S&P 500 index is [UP]}
```

In this example, the search space is defined by two levels:

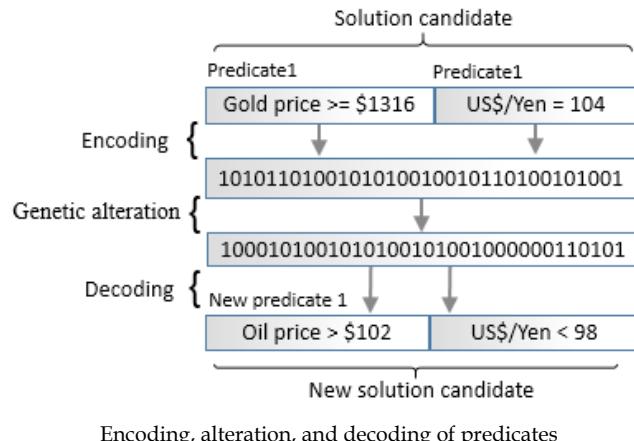
- Boolean operators (for example, AND) and predicates
- Each predicate is defined as a tuple (a variable, operator, target value)

The tree representation for the search space is shown in the following diagram:



A graph representation of encoded rules

The bits string representation is decoded back to its original format for further computation:



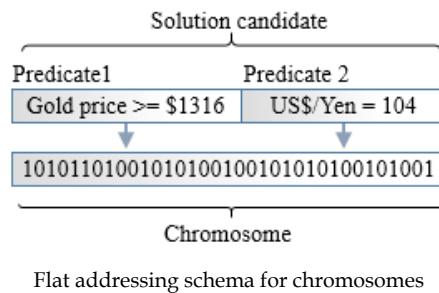
The encoding scheme

There are two approaches to encode such a candidate solution or chain of predicates:

- Flat coding of a chromosome
- Hierarchical coding of a chromosome as a composition of genes

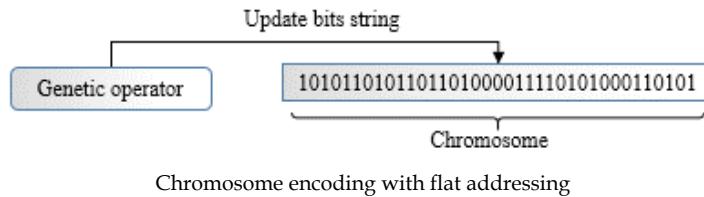
Flat encoding

The flat encoding approach consists of encoding the set of predicates into a single chromosome (bits string), representing a specific solution candidate to the optimization problem. The identity of the predicates is not preserved:



Flat addressing schema for chromosomes

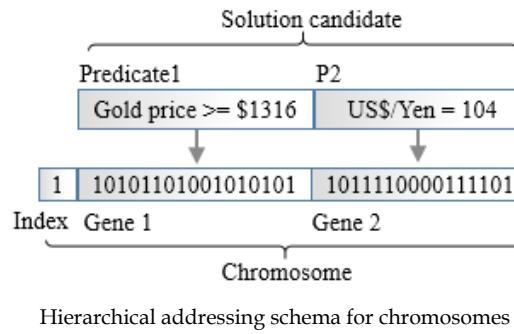
A genetic operator manipulates the bits of the chromosome regardless of whether the bits refer to a particular predicate:



Chromosome encoding with flat addressing

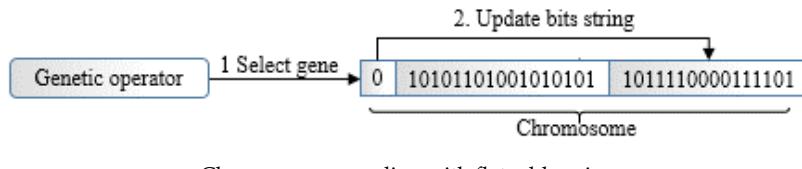
Hierarchical encoding

In this configuration, the characteristic of each predicate is preserved during the encoding process. Each predicate is converted into a gene represented by a bit string. The genes are aggregated to form the chromosome. An extra field is added to the bits string or chromosome for the selection of the gene. This extra field consists of the index or the address of the gene:



Hierarchical addressing schema for chromosomes

A generic operator selects the predicate it needs to first manipulate. Once the target gene is selected, the operator updates the bits string associated with the gene, as follows:



Chromosome encoding with flat addressing

The next step is to define the genetic operators that manipulate or update the bits string representing either a chromosome or individual genes.

Genetic operators

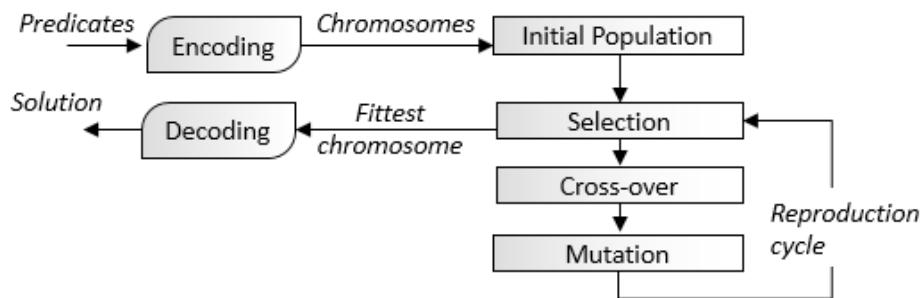
The implementation of the reproduction cycle attempts to replicate the natural reproduction process [10:7]. The reproduction cycle that controls the population of chromosomes consists of three genetic operators:

- **Selection:** This operator ranks chromosomes according to a fitness function or criteria. It eliminates the weakest or less-fit chromosomes and controls the population growth.
- **Crossover:** This operator pairs chromosomes to generate offspring chromosomes. These offspring chromosomes are added to the population along with their parent chromosomes.
- **Mutation:** This operator introduces a minor alteration in the genetic code (bits string representation) to prevent the successive reproduction cycles from electing the same fittest chromosome. In optimization terms, this operator reduces the risk of the genetic algorithm converging quickly toward a local maximum or minimum.

The transposition operator

Some implementations of genetic algorithms use a fourth operator, genetic transposition, in case the fitness function cannot be very well defined and the initial population is very large. Although additional genetic operators could potentially reduce the odds of finding a local maximum or minimum, the inability to describe the fitness criteria or the search space is a sure sign that a genetic algorithm may not be the most suitable tool.

The following diagram gives an overview of the genetic algorithm workflow:



A basic workflow for the execution of genetic algorithms

Initialization

The initialization of the search space (a set of potential solutions to a problem) in any optimization procedure is challenging and genetic algorithms are no exception. In the absence of biases or heuristics, the reproduction initializes the population with randomly generated chromosomes. However, it is worth the effort to extract the characteristics of a population. Any well-founded bias introduced during initialization facilitates the convergence of the reproduction process.

Each of these genetic operators has at least one configurable parameter that has to be estimated and/or tuned. Moreover, you will likely need to experiment with different fitness functions and encoding schemes in order to increase your odds of finding a fittest solution (or chromosome).

Selection

The purpose of the genetic selection phase is to evaluate, rank, and weed out the chromosomes (that is, the solution candidates) that are not a good fit for the problem. The selection procedure relies on a fitness function to score and rank candidate solutions through their chromosomal representation. It is a common practice to constrain the growth of the population of chromosomes by setting a limit to the size of the population.

There are several methodologies to implement the selection process from scaled relative fitness, Holland roulette wheel, and tournament selection to rank-based selection [10:8].

Relative fitness degradation

 As the initial population of chromosomes evolves, the chromosomes tend to get more and more similar to each other. This phenomenon is a healthy sign that the population is actually converging. However, for some problems, you may need to scale or magnify the relative fitness to preserve a meaningful difference in the fitness score between the chromosomes [10:9].

The following implementation relies on rank-based selection.

The selection process consists of the following steps:

1. Apply the fitness function to each chromosome j in the population f_j .
2. Compute the total fitness score for the entire population $\sum f_j$.
3. Normalize the fitness score of each chromosome by the sum of the fitness scores of all the chromosomes $f'_j = f_j / \sum f_j$.
4. Sort the chromosomes by their descending fitness score $f'_j < f'_j - 1$.
5. Compute the cumulative fitness score for each chromosome j $f_j = f_j + \sum f_k$.
6. Generate the selection probability (for the rank-based formula) as a random value $p \in [0,1]$.
7. Eliminate the chromosome k that has a low unfitness score $f_k < p$ or high fitness cost $f_k > p$.
8. Reduce the size of the population if it exceeds the maximum allowed number of chromosomes.

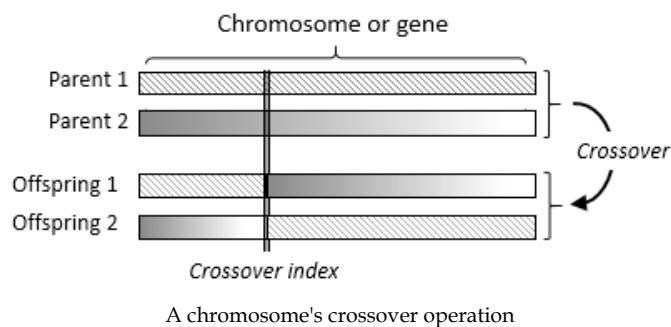
Natural selection



You should not be surprised by the need to control the size of the population of chromosomes. After all, nature does not allow any species to grow beyond a certain point in order to avoid depleting natural resources. The predator-prey process modeled by the **Lotka-Volterra** equation [10:10] keeps the population of each species in check.

Crossover

The purpose of the genetic crossover is to expand the current population of chromosomes in order to intensify the competition among the solution candidates. The crossover phase consists of reprogramming chromosomes from one generation to the next. There are many different variations of crossover techniques. The algorithm for the evolution of the population of chromosomes is independent of the crossover technique. Therefore, the case study uses the simpler one-point crossover. The crossover swaps sections of the two-parent chromosomes to produce two offspring chromosomes, as illustrated in the following diagram:



An important element in the crossover phase is selecting and pairing of parent chromosomes. There are different approaches for selecting and pairing the parent chromosomes that are the most suitable for reproduction:

- Selecting only the n fittest chromosomes for reproduction
- Pairing chromosomes ordered by their fitness (or unfitness) value
- Pairing the fittest chromosome with the least-fit chromosome, the second fittest chromosome with the second least-fit chromosome, and so on

It is a common practice to rely on a specific optimization problem to select the most appropriate selection method as it is highly domain dependent.

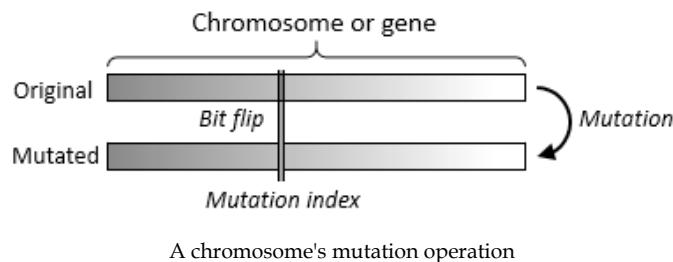
The crossover phase that uses hierarchical addressing as the encoding scheme consists of the following steps:

1. Extract pairs of chromosomes from the population.
2. Generate a random probability $p \in [0,1]$.
3. Compute the index r_i of the gene for which the crossover is applied as $r_i = p.\text{num_genes}$, where num_genes are the number of genes in a chromosome.
4. Compute the index of the bit in the selected gene for which the crossover is applied as $x_i = p.\text{gene_length}$, where gene_length is the number of bits in the gene.
5. Generate two offspring chromosomes by interchanging strands between parents.
6. Add the two offspring chromosomes to the population.

 **Preserving parent chromosomes**
You may wonder why the parents are not removed from the population once the offspring chromosomes are created. This is because there is no guarantee that any of the offspring chromosomes are a better fit.

Mutation

The objective of genetic mutation is to prevent the reproduction cycle from converging toward a local optimum by introducing a pseudo-random alteration to the genetic material. The mutation procedure inserts a small variation in a chromosome to maintain some level of diversity between generations. The methodology consists of flipping one bit in the bits string representation of the chromosome, as illustrated in the following diagram:



The mutation is the simplest of the three phases in the reproduction process. In the case of hierarchical addressing, the steps are as follows:

1. Select the chromosome to be mutated.
2. Generate a random probability $p \in [0,1]$.
3. Compute the index m_i of the gene to be mutated using the formula $m_i = p.\text{num_genes}$.
4. Compute the index of the bit in the gene to be mutated $x_i = p.\text{genes_length}$.
5. Perform a flip XOR operation on the selected bit.

The tuning issue



The tuning of a genetic algorithm can be a daunting task. A plan including a systematic design experiment for measuring the impact of the encoding, fitness function, crossover, and mutation ratio is necessary to avoid lengthy evaluation and self-doubt.

The fitness score

The fitness function is the centerpiece of the selection process. There are three categories of fitness functions, which are as follows:

- **The fixed fitness function:** In this function, the computation of the fitness value does not vary during the reproduction process
- **The evolutionary fitness function:** In this function, the computation of the fitness value morphs between each selection according to predefined criteria
- **An approximate fitness function:** In this function, the fitness value cannot be computed directly using an analytical formula [10:11]

Our implementation of the genetic algorithm uses a fixed fitness function.

Implementation

As mentioned earlier, the genetic operators are independent of the problem to be solved. Let's implement all the components of the reproduction cycle. The fitness function and the encoding scheme are highly domain specific.

In accordance with the principles of object-oriented programming, the software architecture defines the genetic operators using a top-down approach: starting with the population, then each chromosome, and down to each gene.

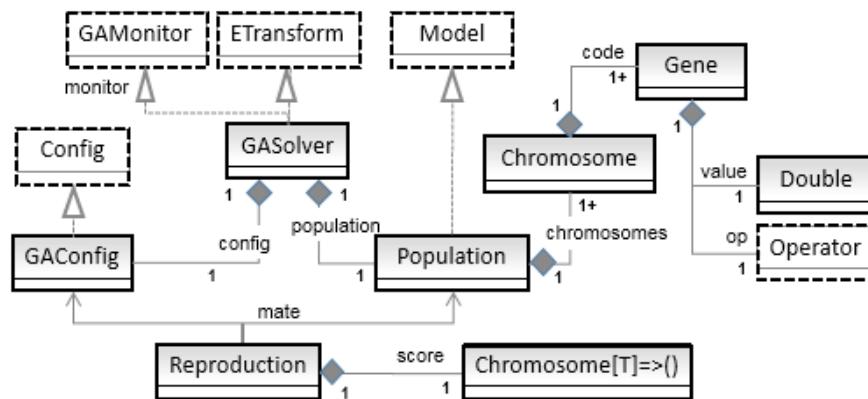
Software design

The implementation of the genetic algorithm uses a design that is similar to the template for classifiers (refer to the *Design template for classifier* section in the *Appendix A, Basic Concepts*).

The key components of the implementation of the genetic algorithm are as follows:

- The `Population` class defines the current set of solution candidates or chromosomes.
- The `GASolver` class implements the GA solver and has two components: a configuration object of the `GAConfig` type and the initial population. This class implements an explicit monadic data transformation of the `ETransform` type.
- The `GAConfig` configuration class consists of the GA execution and reproduction configuration parameters.
- The reproduction (of the `Reproduction` type) controls the reproduction cycle between consecutive generations of chromosomes through the `mate` method.
- The `GAMonitor` monitoring trait tracks the progress of the optimization and evaluates the exit condition for each reproduction cycle.

The following UML class diagram describes the relation between the different components of the genetic algorithm:



The UML class diagram of genetic algorithm components

Let's start with defining the key classes that control the genetic algorithm.

Key components

The `Population` parameterized class (with the `Gene` subtype) contains the set or pool of chromosomes. A population contains chromosomes that are a sequence or list of elements of the type inherited from `Gene`. A `Pool` is a mutable array used in order to avoid excessive duplication of the `Chromosome` instances associated with immutable collections.

The case for mutability

 It is a good Scala programming practice to stay away from mutable collections. However, in this case, the number of chromosomes can be very large. Most implementations of genetic algorithms update the population potentially three times per reproduction cycle, generating a large number of objects and taxing the Java garbage collector.

Population

The `Population` class takes two arguments:

- `limit`: This is the maximum size of the population
- `chromosomes`: This is the pool of chromosomes that define the current population

A reproduction cycle executes the following sequence of three genetic operators on a population: `select` for the selection across all the chromosomes of the population (line 1), `+-` for crossover of all the chromosomes (line 2), and `^` for the mutation of each chromosome (line 3). Consider the following code:

```
type Pool[T <: Gene] = mutable.ArrayBuffer[Chromosome[T]]  
  
class Population[T <: Gene] {  
    limit: Int, val chromosomes: Pool[T]) {  
        def select(score: Chromosome[T] => Unit, cutOff: Double) //1  
        def +- (xOver: Double) //2  
        def ^ (mu: Double) //3  
        ...  
    }  
}
```

The `limit` value specifies the maximum size of the population during optimization. It defines the hard limit or constraints on the population growth.

Chromosomes

The chromosome is the second level of containment in the genotype hierarchy. The Chromosome class takes a list of genes as parameter (code). The signature of the crossover and mutation methods, `+-` and `^`, are similar to their implementation in the Population class except for the fact that the crossover and mutable parameters are passed as indices relative to the list of genes and each gene. The section dedicated to the genetic crossover describes the `GeneticIndices` class:

```
class Chromosome[T <: Gene](val code: List[T]) {
    var cost: Double = Random.nextDouble //4
    def +- (that: Chromosome[T], idx: GeneticIndices):
        (Chromosome[T], Chromosome[T])
    def ^ (idx: GeneticIndices): Chromosome[T]
    ...
}
```

The algorithm assigns the (un)fitting score or a `cost` value to each chromosome to enable the ranking of chromosomes in the population, and ultimately, the selection of the fittest chromosomes (line 4).

Fitness versus cost

 The machine learning algorithms use the loss function or its variant as an objective function to be minimized. This implementation of the GA uses `cost` scores in order to be consistent with the concept of the minimization of the cost, loss, or penalty function.

Genes

Finally, the reproduction process executes the genetic operators on each gene:

```
class Gene(val id: String,
          val target: Double,
          op: Operator)
          (implicit quantize: Quantization, encoding: Encoding){//5
    lazy val bits: BitSet = apply(target, op)

    def apply(value: Double, op: Operator): BitSet //6
    def unapply(bitSet: BitSet): (Double, Operator) //7
    ...
    def +- (index: Int, that: Gene): Gene //8
    def ^ (index: Int): Unit //9
    ...
}
```

The Gene class takes three arguments and two implicit parameters, which are as follows:

- `id`: This is the identifier of the gene. It is usually the name of the variable represented by the gene.
- `target`: This is the target value or threshold to be converted or discretized into a bit string.
- `op`: This is the operator that is applied to the target value.
- `quantize`: This is the quantization or discretization class that converts a double value to an integer to be converted into bits and vice versa (line 5).
- `encoding`: This is the encoding or bits layout of the gene as a pair of values and operators.

The `apply` method encodes a pair of value and operator into a bit set (line 6). An `unapply` method is the reverse operation of `apply`. In this case, it decodes a bit set into a pair of value and operator (line 7).

 **unapply()**
The `unapply` method reverses the state transition performed by the `apply` method. For example, if the `apply` method populates a collection, the `unapply` method clears the collection from its elements.

The implementation of the crossover (line 8) and mutation (line 9) operators on a gene is similar to the operations on the container chromosome.

The quantization is implemented as a case class:

```
case class Quantization(toInt: Double => Int,
   toDouble: Int => Double) {
    def this(R: Int) = this((x: Double) =>
        (x*R).floor.toInt, (n: Int) => n/R)
}
```

The first `toInt` function converts a real value to an integer and `toDouble` converts the integer back to a real value. The `discretization` and `inverse` functions are encapsulated into a class to reduce the risk of inconsistency between the two opposite conversion functions.

The instantiation of a gene converts the predicate representation into a bit string (bits of the `java.util.BitSet` type) using the quantization function, `Quantization.toInt`.

The layout of a gene is defined by the `Encoding` class as follows:

```
class Encoding(nValueBits: Int, nOpBits: Int) {
    val rValue = Range(0, nValueBits)
    val length = nValueBits + nOpBits
    val rOp = Range(nValueBits, length)
}
```

The `Encoding` class specifies the bits layout of the gene as a number of bits, `nValueBits`, to encode the value and the number of bits, `nOpBits`, to encode the operator. The class defines the `rValue` range for the value and the `rOp` range for the operator. The client code has to be supplied to the implicit instance of the `Encoding` class.

The bit set, `bitset`, of the gene (encoding) is implemented by using the `apply` method:

```
def apply(value: Double, op: Operator): BitSet = {
    val bitset = new BitSet(encoding.length)
    encoding.rOp foreach(i => //10
        if((op.id>>i) & 0x01)==0x01) bitset.set(i))
    encoding.rValue foreach(i => //11
        if( ((quant.toInt(value)>>i) & 0x01)==0x01) bitset.set(i))
    bitset
}
```

The bits layout of the gene is created using `java.util.BitSet`. The `op` operator is encoded first through its identifier, `id` (line 10). The `value` is quantized by invoking the `toInt` method and then encoded (line 11).

The `unapply` method decodes the gene from a bit set or bit string to a pair of values and operators. The method uses the quantization instance to cover bits into values and a convert auxiliary function that is described along with its implementation in the source code, accompanying the book (line 12):

```
def unapply(bits: BitSet): (Double, Operator) =
    (quant.toDouble(convert(encoding.rValue, bits)),
     op(convert(encoding.rOp, bits))) //12
```

The `Operator` trait defines the signature of any operator. Each domain-specific problem requires a unique set of operations: Boolean, numeric, or string manipulation:

```
trait Operator {
    def id: Int
    def apply(id: Int): Operator
}
```

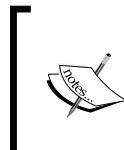
The preceding operator has two methods: an identifier `id` and an `apply` method that converts an index to an operator.

Selection

The first genetic operator of the reproduction cycle is the selection process. The `select` method of the `Population` class implements the steps of the selection phase to the population of chromosomes in the most efficient manner, as follows:

```
def select(score: Chromosome[T] => Unit, cutOff: Double): Unit = {  
    val cumul = chromosomes.map(_.cost).sum/SCALING_FACTOR //13  
    chromosomes foreach(_ /= cumul) //14  
  
    val _chromosomes = chromosomes.sortWith(_.cost < _.cost)//15  
    val cutOffSize = (cutOff*_chromosomes.size).floor.toInt //16  
    val popSize = if(limit < cutOffSize) limit else cutOffSize  
  
    chromosomes.clear //17  
    chromosomes ++= _chromosomes.take(popSize) //18  
}
```

The `select` method computes the `cumul` cumulative sum of the `cost` (line 13) for the entire population. It normalizes the cost of each chromosome (line 14), orders the population by decreasing the value (line 15), and applies a `cutOff` soft limit function on the population growth (line 16). The next step reduces the size of the population to the lowest of the two limits: the hard limit, `limit`, or the soft limit, `cutOffSize`. Finally, the existing chromosomes are cleared (line 17) and updated with the next generation (line 18).



Even population size

The next phase in the reproduction cycle is the crossover, which requires the pairing of parent chromosomes. It makes sense to pad the population so that its size is an even integer.

The `score` scoring function takes a chromosome as a parameter and returns the `cost` value for this chromosome.

Controlling the population growth

The natural selection process controls or manages the growth of the population of species. The genetic algorithm uses the following two mechanisms:

- The absolute maximum size of the population (the hard limit).
- The incentive to reduce the population as the optimization progresses (the soft limit). This incentive (or penalty) on the population growth is defined by the `cutoff` value used during selection (the `select` method).

The `cutoff` value is computed using a `softLimit` user-defined function of the `Int => Double` type, which is provided as a configuration parameter (`softLimit(cycle: Int) => a.cycle + b`).

The GA configuration

The four configurations and tuning parameters required by the genetic algorithm are as follows:

- `xOver`: This is the crossover ratio (or probability) and has a value in the interval $[0, 1]$
- `mu`: This is the mutation ratio
- `maxCycles`: This is the maximum number of reproduction cycles
- `softLimit`: This is the soft constraint on the population growth

Consider the following code:

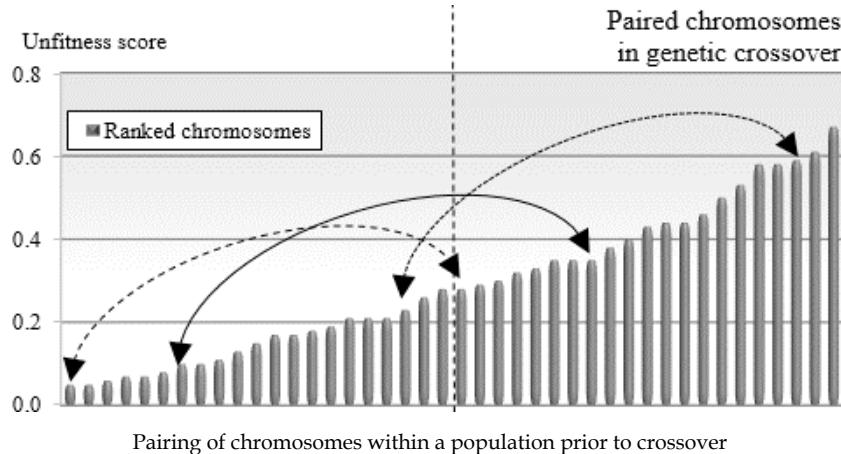
```
class GAConfig(val xover: Double,
               val mu: Double,
               val maxCycles: Int,
               val softLimit: Int => Double) extends Config {
    val mutation = (cycle : Int) => softLimit(cycle)
}
```

Crossover

As mentioned earlier, the genetic crossover operator couples two chromosomes to generate two offspring chromosomes that compete with all the other chromosomes in the population, including their own parents, in the selection phase of the next reproduction cycle.

Population

We use the `+-` notation as the implementation of the crossover operator in Scala. There are several options to select pairs of chromosomes for crossover. This implementation ranks the chromosomes by their *fitness* (or inverse *cost*) value and then divides the population into two halves. Finally, it pairs the chromosomes of identical rank from each half, as illustrated in the following diagram:



The crossover implementation, `+-`, selects the parent chromosome candidates for crossover using the pairing scheme described earlier. Consider the following code:

```
def +- (xOver: Double): Unit =
  if( size > 1) {
    val mid = size>>1
    val bottom = chromosomes.slice(mid, size) //19
    val gIdx = geneticIndices(xOver) //20

    val offSprings = chromosomes.take(mid).zip(bottom)
      .map{ case (t, b) => t +- (b, gIdx) }.unzip //21
    chromosomes ++= offSprings._1 ++ offSprings._2 //22
  }
```

This method splits the population into two subpopulations of equal size (line 19) and applies the Scala `zip` and `unzip` methods to generate the set of pairs of offspring chromosomes (line 20). The `+-` crossover operator is applied to each chromosome pair to produce an array of pairs of `offSprings` (line 21). Finally, the `crossover` method adds offspring chromosomes to the existing population (line 22). The `xOver` crossover value is a probability randomly generated over the interval `[config.xOver, 1]`.

The `GeneticIndices` case class defines two indices of the bit whenever a crossover or a mutation occurs. The first `chOpIdx` index is the absolute index of the bit affected by the genetic operation in the chromosome (line 23). The second `geneOpIdx` index is the index of the bit within the gene subjected to crossover or mutation (line 24):

```
case class GeneticIndices(val chOpIdx: Int, //23
                           val geneOpIdx: Int) //24
```

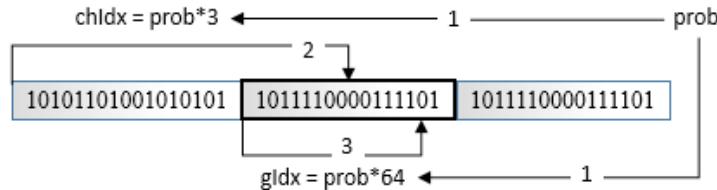
The `geneticIndices` method computes the relative indices of the crossover bit in the chromosomes and genes:

```
def geneticIndices(prob: Double): GeneticIndices = {
    var idx = (prob*chromosomeSize).floor.toInt //25
    val chIdx = if(idx == chromosomeSize) chromosomeSize-1
                  else idx //25

    idx = (prob*geneSize).floor.toInt
    val gIdx = if(idx == geneSize) geneSize-1 else idx //26
    GeneticIndices(chIdx, gIdx)
}
```

The first `chIdx` indexer is the index or rank of the gene within the chromosome to be affected by the genetic operator (line 25). The second `gIdx` indexer is the relative index of the bit within the gene (line 26).

Let's consider a chromosome composed of 2 genes with 63 bits/elements each, as illustrated in the following diagram:



The `geneticIndices` method computes the following:

- The `chIdx` index of the gene within the chromosome and the `gIdx` index of the bit within the gene
- The genetic operator selects the gene of the `chIdx` index (that is the second gene) to be altered
- The genetic operator alters the chromosome at the bit of the `gIdx` index (that is $chIdx*64 + gIdx$)

Chromosomes

First, we need to define the Chromosome class, which takes a list of genes, code, (for genetic code) as the parameter:

```
val QUANT = 500
class Chromosome[T <: Gene] (val code: List[T]) {
    var cost: Double = QUANT*(1.0 + Random.nextDouble) //27

    def +- (that: Chromosome[T], indices: GeneticIndices): //28
        (Chromosome[T], Chromosome[T])
    def ^ (indices: GeneticIndices): Chromosome[T] //29
    def /= (normalizeFactor: Double): Unit = //30
        cost /= normalizeFactor
    def decode(implicit d: Gene=>T): List[T] = //31
        code.map( d(_))
    ...
}
```

The cost (or unfitness) of a chromosome is initialized as a random value between QUANT and 2*QUANT (line 27). The genetic +- crossover operator generates a pair of two offspring chromosomes (line 28). The genetic ^ mutation operator creates a slightly modified (1 or 2 bits) clone of this chromosome (line 29). The /= method normalizes the cost of the chromosome (line 30). The decode method converts the gene to a logic predicate or rule using an implicit conversion, d, between a gene and its subclass (line 31).

Cost initialization

There is no absolute rule to initialize the cost of the chromosomes from an initial population. However, it is recommended that you differentiate a chromosome using nonzero random values with a large range as their cost.

The implementation of the crossover for a pair of chromosomes using hierarchical encoding follows two steps:

1. Find the gene on each chromosome that corresponds to the indices chOpIdx crossover index and then swap the remaining genes.
2. Split and splice the gene crossover at xoverIdx.

Consider the following code:

```
def +- (that: Chromosome[T], indices: GeneticIndices):  
    (Chromosome[T], Chromosome[T]) = {  
    val xoverIdx = indices.chOpIdx //32  
    val xGenes = spliceGene(indices, that.code(xoverIdx)) //33  
  
    val offSprng1 = code.slice(0, xoverIdx) :::  
        xGenes._1 :: that.code.drop(xoverIdx+1) //34  
    val offSprng2 = that.code.slice(0, xoverIdx) :::  
        xGenes._2 :: code.drop(xoverIdx+1)  
    (Chromosome[T](offSprng1), Chromosome[T](offSprng2)) //35  
}
```

The crossover method computes the index `xoverIdx` of the bit that defines the crossover in each parent chromosome (line 32). The `this.code(xoverIdx)` and `that.code(xoverIdx)` genes are swapped and spliced by the `spliceGene` method to generate a spliced gene (line 33):

```
def spliceGene(indices: GeneticIndices, thatCode: T): (T,T) ={  
    ((this.code(indices.chOpIdx) +- (thatCode,indices)),  
     (thatCode +- (code(indices.chOpIdx),indices)) )  
}
```

The offspring chromosomes are gathered by collating the first `xOverIdx` genes of the parent chromosome, the crossover gene, and the remaining genes of the other parent (line 34). The method returns the pair of offspring chromosomes (line 35).

Genes

The crossover is applied to a gene using the `+-` method of the `Gene` class. The exchange of bits between the `this` and `that` genes uses the `BitSet` Java class to rearrange the bits after the permutation:

```
def +- (that: Gene, indices: GeneticIndices): Gene = {  
    val clonedBits = cloneBits(bits) //36  
  
    Range(indices.geneOpIdx, bits.size).foreach(n =>  
        if( that.bits.get(n) ) clonedBits.set(n)  
        else clonedBits.clear(n) //37  
    )  
    val valOp = decode(clonedBits) //38  
    new Gene(id, valOp._1, valOp._2)  
}
```

The bits of the gene are cloned (line 36) and then spliced by exchanging their bits along with the `indices.geneOpIdx` crossover point (line 37). The `cloneBits` function duplicates a bit string, which is then converted into a (target value, operator) tuple using the `decode` method (line 38). We omit these two methods because they are not critical to the understanding of the algorithm.

Mutation

The mutation of the population uses the same algorithmic approach as the crossover operation.

Population

The `^` mutation operator invokes the same operator for all the chromosomes in the population and then adds the mutated chromosomes to the existing population, so that they can compete with the original chromosomes. We use the `^` notation to define the mutation operator to remind you that the mutation is implemented by flipping one bit:

```
def ^ (prob: Double): Unit =
    chromosomes += chromosomes.map(_ ^ geneticIndices(prob))
```

The `prob` mutation parameter is used to compute the absolute index of the mutating gene, `geneticIndices(prob)`.

Chromosomes

The implementation of the `^` mutation operator on a chromosome consists of mutating the gene of the `indices.chOpIdx` index (line 39) and then updating the list of genes in the chromosome (line 40). The method returns a new chromosome (line 41) that will compete with the original chromosome:

```
def ^ (indices: GeneticIndices): Chromosome[T] = { //39
    val mutated = code(indices.chOpIdx) ^ indices
    val xs = Range(0, code.size).map(i =>
        if(i== indices.chOpIdx) mutated
        else code(i)).toList //40
    Chromosome[T](xs) //41
}
```

Genes

Finally, the mutation operator flips (XOR) the bit at the `indices.geneOpIdx` index:

```
def ^ (indices: GeneticIndices): Gene = {
    val idx = indices.geneOpIdx
    val clonedBits = cloneBits(bits) //42

    clonedBits.flip(idx) //43
    val valOp = decode(clonedBits) //44
    new Gene(id, valOp._1, valOp._2) //45
}
```

The `^` method mutates the cloned bit string, `clonedBits`, (line 42) by flipping the bit at the `indices.geneOpIdx` index (line 43). It decodes and converts the mutated bit string by converting it into a (target value, operator) tuple (line 44). The last step creates a new gene from the target-operator tuple (line 45).

Reproduction

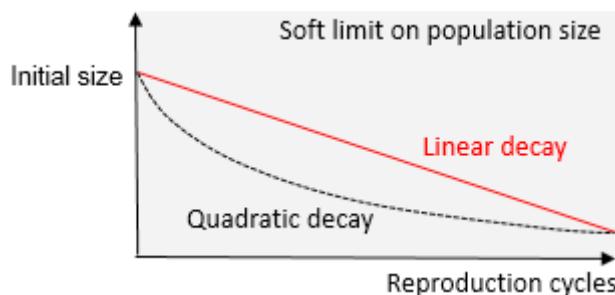
Let's wrap the reproduction cycle into a `Reproduction` class that uses the scoring function, `score`:

```
class Reproduction[T <: Gene](score: Chromosome[T] => Unit)
```

The `mate` reproduction function implements the sequence or workflow of the three genetic operators: `select` for the selection, `+-` (xover) for the crossover, and `^` (mu) for the mutation:

```
def mate(population: Population[T], config: GAConfig,
        cycle: Int): Boolean = (population.size: @switch) match {
    case 0 | 1 | 2 => false //46
    case _ => {
        rand.setSeed(rand.nextInt + System.currentTimeMillis)
        population.select(score, config.softLimit(cycle)) //47
        population +- rand.nextDouble * config.xover //48
        population ^ rand.nextDouble * config.mu //49
        true
    }
}
```

The `mate` method returns false (that is, the reproduction cycle aborts) if the population size is less than 3 (line 46). The chromosomes in the current population are ranked by the increasing cost. The chromosomes with the high cost or low fitness are discarded to comply with the soft limit, `softLimit`, on the population growth (line 47). The randomly generated probability is used as an input to the crossover operation on the entire remaining population (line 48) and as an input to the mutation of the remaining population (line 49):



An illustration of the linear and quadratic soft limit for the population growth

Solver

The `GASolver` class manages the reproduction cycles and the population of chromosomes. The solver is defined as a data transformation of the `ETransform` type using an explicit configuration of the `GAConfig` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* (line 50).

The `GASolver` class implements the `GAMonitor` trait to monitor the population diversity, manage the reproduction cycle, and control the convergence of the optimizer (line 51).

The genetic algorithm-based solver has the following three arguments:

- `config`: This is the configuration of the execution of the genetic algorithm
- `score`: This is the scoring function of a chromosome
- `tracker`: This is the optional tracking function to initialize the monitoring function of `GAMonitor`

The code will be as follows:

```
class GASolver[T <: Gene](config: GAConfig,  
    score: Chromosome[T] => Unit,
```

```

tracker: Option[Population[T] => Unit] = None)
  extends ETransform[GAConfig](config) //50
    with GAMonitor[T] { //51

  type U = Population[T] //52
  type V = Population[T] //53

  val monitor: Option[Population[T] => Unit] = tracker
  def |>(initialize: => Population[T]): Try[Population[T]] =
    this.|> (initialize()) //54
  override def |> : PartialFunction[U, Try[V]] //55
}

```

This explicit data transformation has to initialize the **U** type of an input element (line 52) and the **V** type of an output element (line 53) for the prediction or optimization method, **|>**. The optimizer takes an initial population as the input and generates a very small population of the fittest chromosomes from which the best solution is extracted (line 55).

The population is generated by the **|>** method (**=> Population[T]**) that takes the constructor of the **Population** class as an argument (line 54).

Let's briefly take a look at the **GAMonitor** monitoring trait assigned to the genetic algorithm. The trait has the following two attributes:

- **monitor**: This is an abstract value to be initialized by classes that implement this trait (line 55).
- **state**: This is the current state of the execution of the genetic algorithm. The initial state of the genetic algorithm is **GA_NOT_RUNNING** (line 56).

The code will be as follows:

```

trait GAMonitor[T <: Gene] extends Monitor {
  self: {
    def |> : PartialFunction[Population[T], Try[Population[T]]]
  } => //55
  val monitor: Option[Population[T] => Unit] //56
  var state: GAState = GA_NOT_RUNNING //57

  def isReady: Boolean = state == GA_NOT_RUNNING
  def start: Unit = state = GA_RUNNING
}

```

```
def isComplete(population: Population[T] ,  
               remainingCycles: Int): Boolean = { ... } //58  
}
```

The state of the genetic algorithm can only be updated in the `|>` method through an instance of the `GAMonitor` class. (line 55).

Here is a subset of the possible state of the execution of the genetic algorithm:

```
sealed abstract class GAState(description: String)  
case class GA_FAILED(description: String)  
  extends GAState(description)  
object GA_RUNNING extends GAState("Running")
```

The solver invokes the `isComplete` method to test the convergence of the optimizer at each reproduction cycle (line 58).

There are two options for estimating that the reproducing cycle is converging:

- **Greedy:** In this approach, the objective is to check whether the n fittest chromosomes have not changed in the last m reproduction cycles
- **Loss function:** This approach is similar to the convergence criteria for the training of supervised learning

Let's consider the following implementation of the genetic algorithm solver:

```
override def |> : PartialFunction[U, Try[V]] = {  
  case population: U if(population.size > 1 && isReady) => {  
    start //59  
    val reproduction = Reproduction[T](score) //60  
  
    @tailrec  
    def reproduce(population: Population[T],  
                  n:Int): Population[T] = { //61  
      if( !reproduction.mate(population, config, n) ||  
          isComplete(population, config.maxCycles -n) )  
        population  
      else  
        reproduce(population, n+1)  
    }  
    reproduce(population, 0)  
    population.select(score, 1.0) //62  
    Try(population)  
  }  
}
```

The optimizing method initializes the state of execution (line 59) and the components of the reproduction cycle (line 60). The reproduction cycle (or an epoch) is implemented as a tail recursion that tests whether the last reproduction cycle has failed or whether the optimization has converged toward a solution (line 61). Finally, the remaining fittest chromosomes are reordered by invoking the `select` method of the `Population` class (line 62).

GA for trading strategies

Let's apply our expertise in genetic algorithms to evaluate different strategies to trade securities using trading signals. Knowledge in trading strategies is not required to understand the implementation of a GA. However, you may want to get familiar with the foundation and terminology of technical analysis of securities and financial markets, as described briefly in the *Technical analysis* section in the *Appendix A, Basic Concepts*.

The problem is to find the best trading strategy to predict the increase or decrease of the price of a security given a set of trading signals. A trading strategy is defined as a set of trading signals ts_j that are triggered or fired when a variable $x = \{x_j\}$, derived from financial metrics such as the price of the security or the daily or weekly trading volume, either exceeds or equals or is below a predefined target value a_j (refer to the *Trading signals and strategy* section in the *Appendix A, Basic Concepts*).

The number of variables that can be derived from price and volume can be very large. Even the most seasoned financial professionals face two challenges, which are as follows:

- Selecting a minimal set of trading signals that are relevant to a given dataset (minimize a cost or unfitness function)
- Turning those trading signals with heuristics derived from personal experience and expertise

Alternative to GA

 The problem described earlier can certainly be solved using one of the machine learning algorithms introduced in the previous chapters. It is just a matter of defining a training set and formulating the problem as minimizing the loss function between the predictor and the training score.

The following table lists the trading classes with their counterpart in the genetic world:

Generic classes	Corresponding securities trading classes
Operator	SOperator
Gene	Signal
Chromosome	Strategy
Population	StrategiesFactory

Definition of trading strategies

A chromosome is the genetic encoding of a trading strategy. A factory class, `StrategyFactory`, assembles the components of a trading strategy: operators, unfitness function, and signals

Trading operators

Let's extend the `Operator` trait with the `SOperator` class to define the operations that we need to trigger the signals. The `SOperator` instance has a single parameter: its identifier, `_id`. The class overrides the `id()` method to retrieve the ID (similarly, the class overrides the `apply` method to convert an ID into an `SOperator` instance):

```
class SOperator(_id: Int) extends Operator {
    override def id: Int = _id
    override def apply(idx: Int): SOperator = new SOperator(idx)
}
```

The operators used by trading signals are the logical operators: < (`LESS_THAN`), > (`GREATER_THAN`), and = (`EQUAL`), as follows:

```
object LESS_THAN extends SOperator(1)
object GREATER_THAN extends SOperator(2)
object EQUAL extends SOperator(3)
```

Each operator of the `SOperator` type is associated with a scoring function by the `operatorFuncMap` map. The scoring function computes the cost (or unfitness) of the signal against a real value or a time series:

```
val operatorFuncMap = Map[Operator, (Double, Double) => Double] (
    LESS_THAN -> ((x: Double, target: Double) => target - x),
    ... )
```

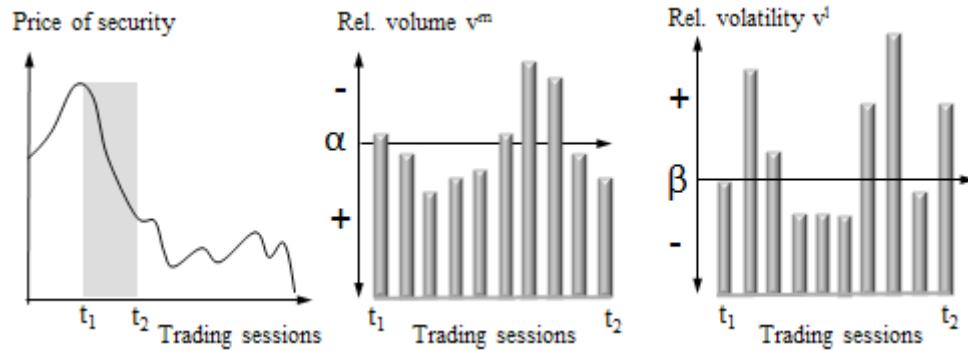
The select method of Population computes the cost value of a signal by quantifying the truthfulness of the predicate. For instance, the unfitness value for a trading signal, $x > 10$, is penalized as $5 - 10 = -5$ for $x = 5$ and credited as $14 - 10 = 4$ if $x = 14$. In this case, the unfitness value is similar to the cost or loss in a discriminative machine learning algorithm.

The cost function

Let's consider the following trading strategy defined as a set of two signals to predict the sudden relative decrease Δp of the price of a security:

- Relative volume v_m with a condition $v_m < \alpha$
- Relative volatility v_l with the condition $v_l > \beta$

Let's take a look at the following graphs:



A chart of the price, relative volume, and relative volatility of a security

As the goal is to model a sudden crash in the stock price, we should reward the trading strategies that predict the steep decrease in the stock price and penalize the strategies that work well only with a small decrease or increase in the stock price. In the case of the trading strategy with two signals, relative volume v_m and relative volatility v_l , n trading sessions, the cost or unfitness function C , and given a relative variation of the stock price and a penalization $w = -\Delta p$ (M2):

$$w_t = -\Delta p_t$$

$$C(p, v^m, v^l | \alpha, \beta) = \sum_{t=0}^{n-1} (\alpha - v_t^m) w_t + (v_t^l - \beta) w_t$$

Trading signals

Let's subclass the `Gene` class to define the trading signal of the `Signal` type as follows:

```
class Signal(id: String, target: Double, op: Operator,
            xt: DblVector, weights: Option[DblVector] = None)
  (implicit quantize: Quantization, encoding: Encoding)
  extends Gene(id, target, op)
```

The `Signal` class requires the following arguments:

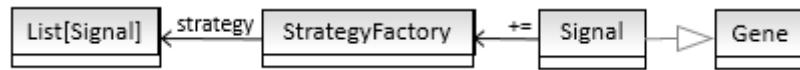
- An identifier `id` for the feature
- A target value
- An `op` operator
- An `xt` time series of the `DblVector` type
- The optional `weights` associated with each data point of the time series, `xt`
- An implicit quantization instance, `quantize`
- An implicit encoding scheme

The main purpose of the `Signal` class is to compute its `score` as a chromosome. The chromosome updates its cost by summing the score or weighted score of the signals it contains. The score of the trading signal is simply the summation of the penalty or truthfulness of the signal for each entry of the time series, `ts`:

```
override def score: Double =
  if(!operatorFuncMap.contains(op)) Double.MaxValue
  else {
    val f = operatorFuncMap.get(op).get
    if( weights != None ) xt.zip(weights.get)
      .map{case(x, w) => w*f(x,target)} .sum
    else xt.map( f(_, target)).sum
  }
```

Trading strategies

A trading strategy is an unordered list of trading signals. It makes sense to create a factory class to generate the trading strategies. The `StrategyFactory` class creates strategies of the `List[Signal]` type from an existing pool of signals of the subtype, `Gene`:



A factory pattern for trading signals

The `StrategyFactory` class has two arguments: the number of signals, `nSignals`, in a trading strategy and the implicit Quantization and Encoding instances (line 63):

```
class StrategyFactory(nSignals: Int) //63
  (implicit quantize: Quantization, encoding: Encoding) {
  val signals = new ListBuffer[Signal]
  lazy val strategies: Pool[Signal] //64
  def +=(id: String, target: Double, op: SOperator,
         xt: DblVector, weights: DblVector)
  ...
}
```

The `+=` method takes five arguments: the identifier `id`, the target value, the `op` operation to qualify the class as Gene, the `xt` times series for scoring the signals, and the `weights` associated with the overall cost function. The `StrategyFactory` class generates all possible sequences of signals as trading strategies as lazy values to avoid unnecessary regeneration of the pool on demand (line 64), as follows:

```
lazy val strategies: Pool[Signal] = {
  implicit val ordered = Signal.orderedSignals //70

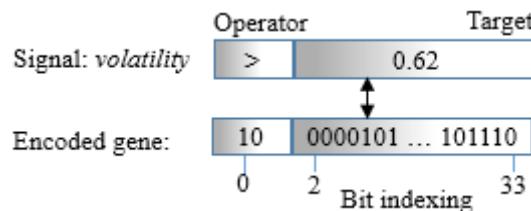
  val XSS = new Pool[Signal] //65
  val treeSet = new TreeSet[Signal] ++= signals.toList //66
  val subsetsIterator = treeSet.subsets(nSignals) //67

  while( subsetsIterator.hasNext) {
    val signalList = subsetsIterator.next.toList //68
    XSS.append(Chromosome[Signal](signalList)) //69
  }
  XSS
}
```

The implementation of the `strategies` value creates a pool of signals `Pool` (line 65) by converting the list of signals to `treeset` (line 66). It breaks down the tree set into unique subtrees of `nSignals` nodes each. It instantiates a `subsetsIterator` iterator to traverse the sequence of subtrees (line 67) and converts them into a list (line 68) as arguments of the new chromosome (trading strategy) (line 69). The procedure to order the signals, `orderedSignals`, in the tree set has to be implicitly defined (line 70) as `val orderedSignals = Ordering.by((signal: Signal) => signal.id)`.

Trading signal encoding

The encoding of trading predicates is the most critical element of the genetic algorithm. In our example, we encode a predicate as a tuple (target value, operator). Let's consider the simple predicate *volatility > 0.62*. The discretization converts the value 0.62 into 32 bits for the instance and a 2-bit representation for the operator:



Encoding of the trading signal: *volatility > 0.62*

IEEE-732 encoding

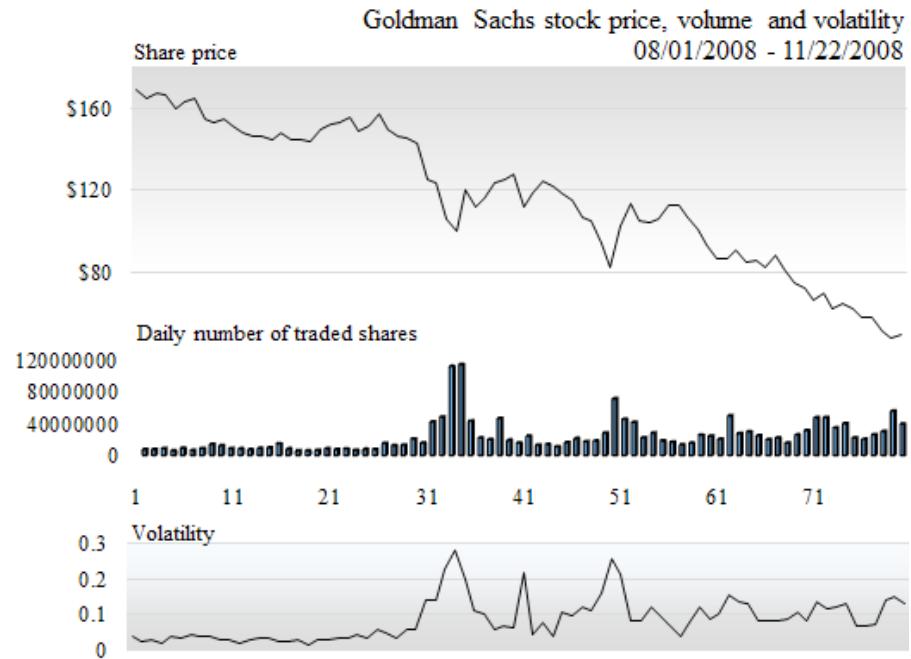
The threshold value for predicates is converted into an integer (the `Int` type or `Long`). The IEEE-732 binary representation of floating point values makes the bit addressing required to apply genetic operators quite challenging. A simple conversion consists of the following:

- encoding e: `(x: Double) => (x*100000).toInt`
- decoding d: `(x: Int) => x*1e-5`

All values are normalized, so there is no risk of overflowing the 32-bit representation.

A test case

The goal is to evaluate which trading strategy was the most relevant (fittest) during the crash of the stock market in fall 2008. Let's consider the stock price of one of the financial institutions, Goldman Sachs, as a proxy of the sudden market decline:



Besides the variation of the price of the stock between two consecutive trading sessions (`dPrice`), the model uses the following parameters (or trading signals):

- `dVolume`: This is the relative variation of the volume between two consecutive trading sessions
- `dVolatility`: This is the relative variation of volatility between two consecutive trading sessions
- `volatility`: This is the relative volatility within a trading session
- `vPrice`: This is the relative difference of the stock opening and closing price

The naming convention for the trading data and metrics is described in the *Trading data* section under *Technical analysis* in the *Appendix A, Basic Concepts*.

The execution of the genetic algorithm requires the following steps:

1. Extraction of model parameters or variables.
2. Generation of the initial population of trading strategies.
3. Setting up the GA configuration parameters with the maximum number of reproduction cycles allowed, the crossover and mutation ratio, and the soft limit function for the population growth.
4. Instantiating the GA algorithm with the scoring/unfitness function.
5. Extracting the fittest trading strategy that can best explain the sharp decline in the price of Goldman Sachs stocks.

Creating trading strategies

The input to the genetic algorithm is the population of trading strategies. Each strategy consists of the combination of three trading signals and each trading signal is a tuple (signal ID, operator, and target value).

The first step is to extract the model parameters as illustrated for the variation of the stock price volume, volatility, and relative volatility between two consecutive trading sessions (line 71):

```
Import YahooFinancials._  
val NUM_SIGNALS = 3  
  
def createStrategies: Try[Pool[Signal]] = {  
    val src = DataSource(path, false, true, 1) //71  
    for { //72  
        price <- src.get(adjClose)  
        dPrice <- delta(price, -1.0)  
        volume <- src.get(volume)  
        dVolume <- delta(volume, 1.0)  
        volatility <- src.get(volatility)  
        dVolatility <- delta(volatility, 1.0)  
        vPrice = src.get(vPrice)  
    } yield { //72  
        val factory = new StrategyFactory(NUM_SIGNALS) //73  
  
        val weights = dPrice //74  
        factory += ("dvolume", 1.1, GREATER_THAN, dVolume, weights)  
        factory += ("volatility", 1.3, GREATER_THAN,  
                    volatility.drop(1), weights)  
        factory += ("vPrice", 0.8, LESS_THAN,  
                    vPrice.drop(1), weights)
```

```

factory += ("dVolatility", 0.9, GREATER_THAN,
            dVolatility, weights)
factory.strategies
}
}

```

The purpose is to generate the initial population of strategies that compete to become relevant to the decline of the price of stocks of Goldman Sachs. The initial population of trading strategies is generated by creating a combination from four trading signals weighted by the variation in the stock price: $\Delta(volume) > 1.1$, $\Delta(volatility) > 1.3$, $\Delta(close-open) < 0.8$, and $volatility > 0.9$.

The delta method computes the variation of a trading variable between consecutive trading sessions. It invokes the `XTSeries.zipWithShift` method, which was introduced in the *Time series in Scala* section in *Chapter 3, Data Preprocessing*:

```

def delta(xt: DblVector, a: Double): Try[DblVector] = Try {
    zipWithShift(xt, 1).map{case (x, y) => a*(y/x - 1.0)}
}

```

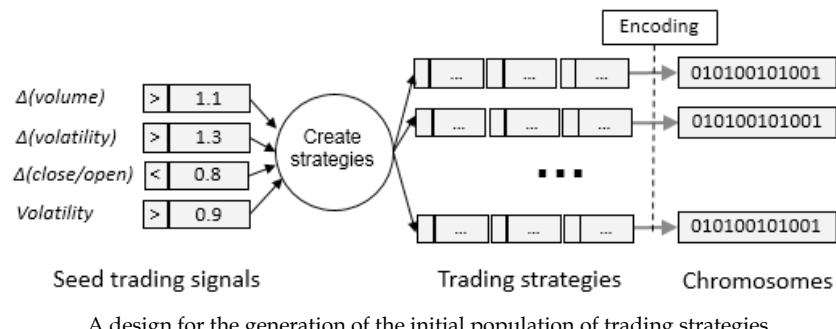
The trading strategies are generated by the `StrategyFactory` class introduced in the previous section (line 73). The weights for the trading strategies are computed as the `dPrice` difference of the price of the stock between two consecutive trading sessions (line 74). The option of unweighted trading strategies is selected by replacing the weights by the average price variation as follows:

```

val avWeights = dPrice.sum/dPrice.size
val weights = Vector.fill(dPrice.size)(avWeights)

```

The generation of the initial population of trading strategies is illustrated in the following diagram:



Configuring the optimizer

The configuration parameters for the execution of the genetic algorithm is categorized as follows:

- Tuning parameters such as crossover, mutation ratio, or soft limit on the population growth
- Data representation parameters such as quantization and encoding
- A scoring scheme

The four configuration parameters for the GA are the maximum number of reproduction cycles (**MAX_CYCLES**) allowed in the execution, the crossover (**XOVER**), the mutation ratio (**MU**), and the soft limit function (**softLimit**) to control the population growth. The soft limit is implemented as a linearly decreasing function of the number of cycles (**n**) to retrain the growth of the population as the execution of the genetic algorithm progresses:

```
val XOVER = 0.8 //Probability(ratio) for cross-over
val MU = 0.4 //Probability(ratio) for mutation
val MAX_CYCLES = 400 //Max. number of optimization cycles

val CUTOFF_SLOPE = -0.003 //Slope linear soft limit
val CUTOFF_INTERCEPT = 1.003 //Intercept linear soft limit
val softLimit = (n: Int) => CUTOFF_SLOPE*n +CUTOFF_INTERCEPT
```

The trading strategies are converted into chromosomes through encoding (line 75). A digitize quantization scheme has to be implicitly defined in order to encode the target value in each trading signal (line 76):

```
implicit val encoding = defaultEncoding //75
val R = 1024 //Quantization ratio
implicit val digitize = new Quantization(R) //76
```

The scoring function computes the cost or unfitness of a trading strategy (chromosome) by applying the score function to each of the three trading signals (genes) it contains (line 77):

```
val scoring = (chr: Chromosome[Signal]) => {
    val signals: List[Gene] = chr.code
    chr.cost = signals.map(_.score).sum //77
}
```

Finding the best trading strategy

The trading strategies generated by the factory in the `createStrategies` method are fed to the genetic algorithm as the initial population (line 79). The upper limit to the population growth is set at eight times the size of the initial population (line 78):

```
createStrategies.map(strategies => {
    val limit = strategies.size <<3 //78
    val initial = Population[Signal](limit, strategies) //79

    val config = GAConfig(XOVER, MU, MAX_CYCLES,softLimit) //80
    val solver = GASolver[Signal](config,scoring,Some(tracker)) //81
        (solver |> initial)
        .map(_.fittest.map(_.symbolic).getOrElse("NA")) match {
            case Success(results) => show(results)
            case Failure(e) => error("GAEval: ", e)
        } //82
    })
})
```

The configuration, `config` (line 80), the scoring function, and optionally a tracker function are all that you need to create and execute the `solver` genetic algorithm (line 81). The partial function generated by the `|>` operator transforms the initial population of trading strategies into the two `fittest` strategies (line 82).

The documented source code for the monitoring function, tracker, and miscellaneous methods is available online.

Tests

The cost function C (or unfitness) score of each trading strategy are weighted for the rate of decline of the price of the Goldman Sachs stock. Let's run the following two tests:

- Evaluation of the configuration of the genetic algorithm with the score weighted by the price variation
- Evaluation of the genetic algorithm with an unweighted scoring function

The weighted score

The score is weighted by the variation of the price of the stock GS. The test uses three different sets of crossover and mutation ratios: (0.6, 0.2), (0.3, 0.1), and (0.2, 0.6). The best trading strategy for each scenario is as follows:

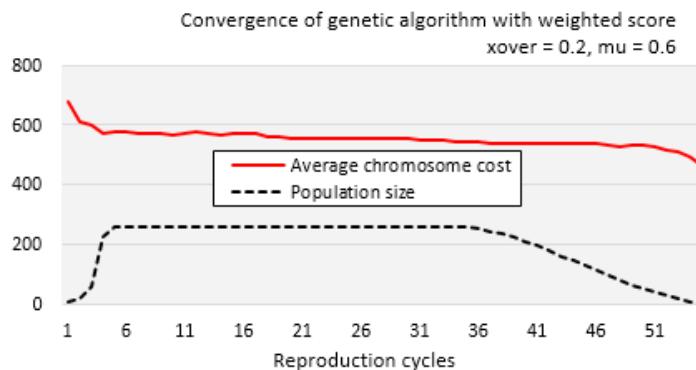
- **0.6-0.2:** $change < 0.82$ $dVolume > 1.17$ $volatility > 1.35$ $cost = 0.0$ $fitness: 1.0E10$
- **0.3-0.1:** $change < 0.42$ $dVolume > 1.61$ $volatility > 1.08$ $cost = 59.18$ $fitness: 0.016$
- **0.2-0.6:** $change < 0.87$ $dVolume < 8.17$ $volatility > 3.91$ $cost = 301.3$ $fitness: 0.003$

The fittest trading strategy for each case does not differ much from the initial population for one or several of the following reasons:

- The initial guess for the trading signals was good
- The size of the initial population is too small to generate genetic diversity
- The test does not take into account the rate of decline of the stock price

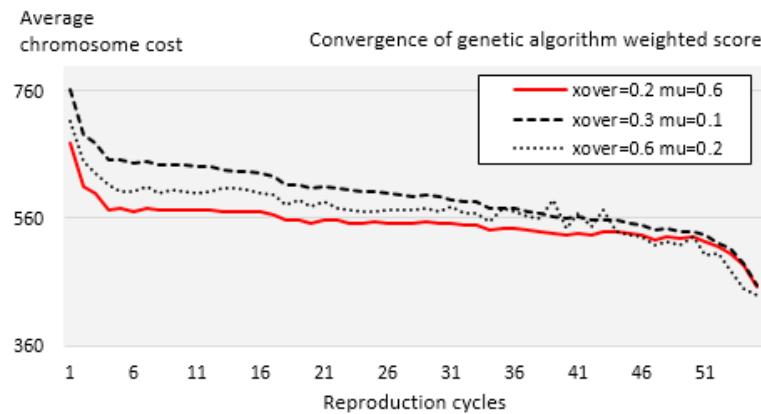
The execution of the genetic algorithm with $crossover = 0.2$ and $mutation = 0.6$ produces a trading strategy that is inconsistent with the first two cases. One possible explanation is the fact that the crossover is applied always to the first of the three genes, forcing the optimizer to converge toward a local minimum.

Let's examine the behavior of the genetic algorithm during execution. We are particularly interested in the convergence of the average chromosome unfitness score. The average chromosome unfitness is the ratio of the total unfitness score for the population over the size of the population. Let's take a look at the following graph:



The convergence of a genetic algorithm for the crossover ratio 0.2 and mutation 0.6 with a weighted score

The GA converges quite quickly and then stabilizes. The size of the population increases through crossover and mutation operations until it reaches the maximum of 256 trading strategies. The soft limit or constraint on the population size kicks in after 23 trading cycles. The test is run again with different values of crossover and mutation ratios, as shown in the following graph:



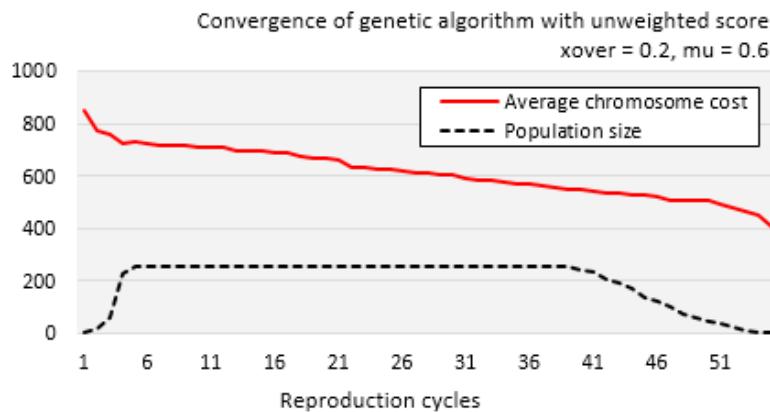
The impact of the crossover and mutation ratio on the convergence of a genetic algorithm with a weighted score

The profile of the execution of the genetic algorithm is not overly affected by the different values of crossover and mutation ratios. The chromosome unfitness score for the high crossover ratio (0.6) oscillates as the execution progresses. In some cases, the unfitness score between chromosomes is so small that the GA recycles the same few trading strategies.

The quick decline in the unfitness of the chromosomes is consistent with the fact that some of the fittest strategies were part of the initial population. It should, however, raise some concerns that the GA locked on a local minimum early on.

The unweighted score

The execution of a test that is similar to the previous one with the unweighted trading strategies (trading strategies that use the average price variation) scoring formula produces some interesting results, as shown in the following graph:



The convergence of a genetic algorithm for the crossover ratio 0.4 and mutation 0.4 with an unweighted score

The profile for the size of the population is similar to the test using weighted scoring. However, the chromosome average cost pattern is somewhat linear. The unweighted (or averaging) adds the rate of decline of the stock price to the score (cost).

The complexity of a scoring function

The complexity of the scoring (or computation of the cost) formula increases the odds of the genetic algorithm not converging properly. The possible solutions to the convergence problem are as follows:

- Make the weighting function additive (less complex)
- Increase the size and diversity of the initial population

Advantages and risks of genetic algorithms

Now, it should be clear that genetic algorithms provide scientists with a powerful toolbox with which to optimize problems that:

- Are poorly understood.
- May have more than one good enough solution.

- Have discrete, discontinuous, and nondifferentiable functions.
- Can be easily integrated with the rules engine and knowledge bases (for example, learning classifiers systems).
- Do not require deep domain knowledge. The genetic algorithm generates new solution candidates through genetic operators. The initial population does not have to contain the fittest solution.
- Do not require knowledge of numerical methods such as the **Newton-Raphson**, **conjugate gradient**, or **BFGS** as optimization techniques, which frighten those with little inclination for mathematics.

However, evolutionary computation is not suitable for problems for which:

- A fitness function cannot be clearly defined
- Finding the global (absolute) minimum or maximum is essential to the problem
- The execution time has to be predictable
- The solution has to be provided in real time or pseudo-real time (streaming data)

Summary

Are you hooked on evolutionary computation, genetic algorithms in particular, and their benefits, limitations as well as some of the common pitfalls? If the answer is yes, then you may find learning classifier systems, introduced in the next chapter, fascinating. This chapter dealt with the following topics:

- Key concepts in evolutionary computing
- The key components and operators of genetic operators
- The pitfalls in defining a fitness or unfitness score using a financial trading strategy as a backdrop
- The challenge of encoding predicates in the case of trading strategies
- Advantages and risks of genetic algorithms
- The process for building a genetic algorithm forecasting tool from the bottom up

The genetic algorithm is an important element of a special class of reinforcement learning, which is introduced in the *Learning classifier systems* section in the next chapter.

11

Reinforcement Learning

This chapter presents the concept of **reinforcement learning**, which is widely used in gaming and robotics. The second part of this chapter is dedicated to **learning classifier systems**, which combine reinforcement learning techniques with evolutionary computing introduced in the previous chapter. Learning classifiers are an interesting breed of algorithms that are not commonly included in literature dedicated to machine learning. I highly recommend that you to read the seminal book on reinforcement learning by R. Sutton and A. Barto [11:1] if you are interested to know about the origin, purpose, and scientific foundation of reinforcement learning.

In this chapter, you will learn the following topics:

- Basic concepts behind reinforcement learning
- A detailed implementation of the Q-learning algorithm
- A simple approach to manage and balance an investment portfolio using reinforcement learning
- An introduction to learning classifier systems
- A simple implementation of extended learning classifiers

The section on **learning classifier systems (LCS)** is mainly informative and does not include a test case.

Reinforcement learning

The need of an alternative to traditional learning techniques arose with the design of the first autonomous systems.

The problem

Autonomous systems are semi-independent systems that perform tasks with a high degree of autonomy. Autonomous systems touch every facet of our life, from robots and self-driving cars to drones. Autonomous devices react to the environment in which they operate. The reaction or action requires the knowledge of not only the current state of the environment but also the previous state(s).

Autonomous systems have specific characteristics that challenge traditional methodologies of machine learning, as listed here:

- Autonomous systems have poorly defined domain knowledge because of the sheer number of possible combinations of states.
- Traditional nonsequential supervised learning is not a practical option because of the following:
 - Training consumes significant computational resources, which are not always available on small autonomous devices
 - Some learning algorithms are not suitable for real-time prediction
 - The models do not capture the sequential nature of the data feed
- Sequential data models such as hidden Markov models require training sets to compute the emission and state transition matrices (as explained in *The hidden Markov model* section in *Chapter 7, Sequential Data Models*), which are not always available. However, a reinforcement learning algorithm benefits from a hidden Markov model if some of the states are unknown. These algorithms are known as behavioral hidden Markov models [11:2].
- Genetic algorithms are an option if the search space can be constrained heuristically. However, genetic algorithms have unpredictable response time, which makes them impractical for real-time processing.

A solution – Q-learning

Reinforcement learning is an algorithmic approach to understanding and ultimately automating goal-based decision making. Reinforcement learning is also known as control learning. It differs from both supervised and unsupervised learning techniques from the knowledge acquisition standpoint: **autonomous**, automated systems, or devices learn from direct and real-time interaction with their environment. There are numerous practical applications of reinforcement learning from robotics, navigation agents, drones, adaptive process control, game playing, and online learning, to scheduling and routing problems.

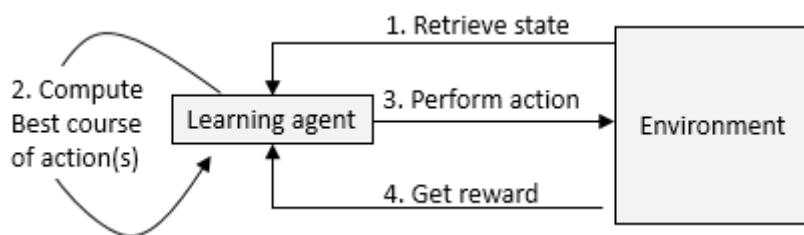
Terminology

Reinforcement learning introduces new terminologies as listed here, which are quite different from that of older machine learning techniques:

- **Environment:** This is any system that has states and mechanisms to transition between states. For example, the environment for a robot is the landscape or facility it operates.
- **Agent:** This is an automated system that interacts with the environment.
- **State:** The state of the environment or system is the set of variables or features that fully describe the environment.
- **Goal or absorbing state or terminal state:** This is the state that provides a higher discounted cumulative reward than any other state. A high cumulative reward prevents the best policy from being dependent on the initial state during training.
- **Action:** This defines the transition between states. The agent is responsible for performing or at least recommending an action. Upon execution of the action, the agent collects a reward (or punishment) from the environment.
- **Policy:** This defines the action to be selected and executed for any state of the environment.
- **Best policy:** This is the policy generated through training. It defines the model in Q-learning and is constantly updated with any new episode.
- **Reward:** This quantifies the positive or negative interaction of the agent with the environment. Rewards are essentially the training set for the learning engine.
- **Episode:** This defines the number of steps necessary to reach the goal state from an initial state. Episodes are also known as trials.
- **Horizon:** This is the number of future steps or actions used in the maximization of the reward. The horizon can be infinite, in which case the future rewards are discounted in order for the value of the policy to converge.

Concepts

The key component in reinforcement learning is a **decision-making agent** that reacts to its environment by selecting and executing the best course of actions and being rewarded or penalized for it [11:3]. You can visualize these agents as robots navigating through an unfamiliar terrain or a maze. Robots use reinforcement learning as part of their reasoning process after all. The following diagram gives the overview architecture of the reinforcement learning agent:



The four state transitions of reinforcement learning

The agent collects the state of the environment, selects, and then executes the most appropriate action. The environment responds to the action by changing its state and rewarding or punishing the agent for the action.

The four steps of an episode or learning cycle are as follows:

1. The learning agent retrieves or is notified of a new state of the environment.
2. The agent evaluates and selects the action that may provide the highest reward.
3. The agent executes the action.
4. The agent collects the reward or penalty and applies it to calibrate the learning algorithm.

 **Reinforcement versus supervision**
The training process in reinforcement learning rewards features that maximize a value or return. Supervised learning rewards features that meet a predefined labeled value. Supervised learning can be regarded as forced learning.

The action of the agent modifies the state of the system, which in turn notifies the agent of the new operational condition. Although not every action will trigger a change in the state of the environment, the agent collects the reward or penalty nevertheless. At its core, the agent has to design and execute a sequence of actions to reach its goal. This sequence of actions is modeled using the ubiquitous Markov decision process (refer to the *Markov decision processes* section in *Chapter 7, Sequential Data Models*).

Dummy actions



It is important to design the agent so that actions may not automatically trigger a new state of the environment. It is easy to think about a scenario in which the agent triggers an action just to evaluate its reward without affecting the environment significantly.

A good metaphor for such a scenario is the *rollback* of the action. However, not all environments support such a *dummy* action, and the agent may have to run Monte-Carlo simulations to try out an action.

Value of a policy

Reinforcement learning is particularly suited to problems for which long-term rewards can be balanced against short-term rewards. A policy enforces the trade-off between short-term and long-term rewards. It guides the behavior of the agent by mapping the state of the environment to its actions. Each policy is evaluated through a variable known as the **value of a policy**.

Intuitively, the value of a policy is the sum of all the rewards collected as a result of the sequence of actions taken by the agent. In practice, an action over the policy farther in the future obviously has a lesser impact than the next action from a state S_t to a state S_{t+1} . In other words, the impact of future actions on the current state has to be discounted by a factor, known as the *discount coefficient for future rewards* < 1 .

Transition and rewards matrices



The transition and emission matrices have been introduced in the *The hidden Markov model* section in *Chapter 7, Sequential Data Models*.

The optimum policy π^* is the agent's sequence of actions that maximizes the future reward discounted to the current time.

The following table introduces the mathematical notation of each component of reinforcement learning:

Notation	Description
$S = \{s_i\}$	These are the states of the environment
$A = \{a_j\}$	These are the actions on the environment
$\Pi_t = p(a_t s_t)$	This is the policy (or strategy) of the agent
$V^\pi(s_t)$	This is the value of the policy at a state
$p_t = p(s_{t+1} s_t, a_t)$	These are the state transition probabilities from the state s_t to the state s_{t+1}
$r_t = p(r_{t+1} s_t, s_{t+1}, a_t)$	This is the reward of an action a_t for a state s_t
R_t	This is the expected discounted long-term return
γ	This is the coefficient to discount the future rewards

The purpose is to compute the maximum expected reward R_t from any starting state s_k as the sum of all discounted rewards to reach the current state s_t . The value V^π of a policy π at the state s_t is the maximum expected reward R_t given the state s_t .



M1: The cumulative reward R_t and value function $V^\pi(st)$ for the state st given a policy π and a discount rate γ is defined as:

$$R_t = \sum_{k=0}^{+\infty} \gamma^k r_{t+1+k}$$

$$V^\pi(s_t) = E\{R_t | s_t\}$$

The Bellman optimality equations

The problem of finding the optimal policies is indeed a nonlinear optimization problem whose solution is iterative (dynamic programming). The expression of the value function V^π of a policy π can be formulated using the Markovian state transition probabilities p_t .

M2: The value function $V^\pi(s_t)$ for a state s_t and future state s_k with a reward r_k using the transition probability $p_{k'}$ given a policy π and a discount rate γ is defined as:

$$V^\pi(s_t) = \sum_{a \in A} \pi_a \sum_k \{p_k(r_k + \gamma \cdot V^\pi(s_k))\}$$

$$V^*(s_t) = \max_\pi V^\pi(s_t)$$

$V^*(s_t)$ is the optimal value of the state s_t across all the policies. The equations are known as the Bellman optimality equations.

The curse of dimensionality

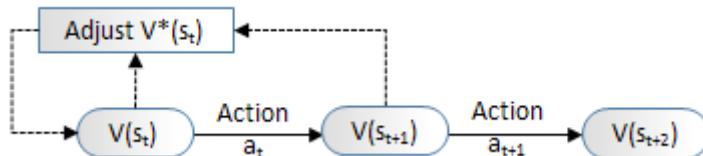
The number of states for a high-dimension problem (large-feature vector) becomes quickly unsolvable. A workaround is to approximate the value function and reduce the number of states by sampling. The application test case introduces a very simple approximation function.

If the environment model, state, action, and rewards, as well as transition between states, are completely defined, the reinforcement learning technique is known as model-based learning. In this case, there is no need to explore a new sequence of actions or state transitions. Model-based learning is similar to playing a board game in which all combinations of steps that are necessary to win are completely known.

However, most practical applications using sequential data do not have a complete, definitive model. Learning techniques that do not depend on a fully defined and available model are known as model-free techniques. These techniques require exploration to find the best policy for any given state. The remaining sections in this chapter deal with model-free learning techniques, and more specifically, the temporal difference algorithm.

Temporal difference for model-free learning

Temporal difference is a model-free learning technique that samples the environment. It is a commonly used approach to solve the Bellman equations iteratively. The absence of a model requires a discovery or **exploration** of the environment. The simplest form of exploration is to use the value of the next state and the reward defined from the action to update the value of the current state, as described in the following diagram:



An illustration of the temporal difference algorithm

The iterative feedback loop used to adjust the value action on the state plays a role similar to the backpropagation of errors in artificial neural networks or minimization of the loss function in supervised learning. The adjustment algorithm has to:

- Discount the estimate value of the next state using the discount rate γ
- Strike a balance between the impact of the current state and the next state on updating the value at time t using the learning rate α

The iterative formulation of the first Bellman equation predicts $V^\pi(s_t)$, the value function of state s_t from the value function of the next state s_{t+1} . The difference between the predicted value and the actual value is known as the temporal difference error abbreviated as δ_t .

M3: The formula for tabular temporal difference δ_t for a value function $V(s_t)$ at state s_t , a learning rate α , a reward r_t , and a discount rate γ is defined as:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

$$\hat{V}^\pi(s_t) = V^\pi(s_t) + \alpha \delta_t$$

An alternative to evaluating a policy using the value of the state $V^\pi(s_t)$ is to use the value of taking an action on a state s_t known as the value of action (or action-value) $Q^\pi(s_t, a_t)$.

 M4: The definition of the value Q of action at a state s_t as the expectation of a reward R_t for an action a_t on a state s_t is defined as:

$$Q_t^\pi = Q^\pi(s_t, a_t) = E(R_t | s_t, a_t)$$

There are two methods to implement the temporal difference algorithm:

- **On-policy:** This is the value for the next best action that uses the policy
- **Off-policy:** This is the value for the next best action that does not use the policy

Let's consider the temporal difference algorithm using an off-policy method and its most commonly used implementation: Q-learning.

Action-value iterative update

Q-learning is a model-free learning technique using an off-policy method. It optimizes the action-selection policy by learning an action-value function. Like any machine learning technique that relies on convex optimization, the Q-learning algorithm iterates through actions and states using the quality function, as described in the following mathematical formulation.

The algorithm predicts and discounts the optimum value of action $\max\{Q_i\}$ for the current state s_t and action a_t on the environment to transition to the state s_{t+1} .

Similar to genetic algorithms that reuse the population of chromosomes in the previous reproduction cycle to produce offspring, the Q-learning technique strikes a balance between the new value of the quality function Q_{t+1} and the old value Q_t using the learning rate α . Q-learning applies temporal difference techniques to the Bellman equation for an off-policy methodology.

 M5: The Q-learning action-value updating formula for a given policy π , set of states $\{s_t\}$, a set of actions $\{a_t\}$ associated with each state s_t , a learning rate α , and a discount rate γ is given by:

$$\tilde{Q}_t^\pi = Q_t^\pi + \alpha \left[r_{t+1} + \gamma \max_{a_{t+1}} Q_{t+1}^\pi - Q_t^\pi \right] \quad Q_t^\pi = Q^\pi(s_t, a_t)$$

- A value 1 for the learning rate α discards the previous state, while a value 0 discards learning
- A value 1 for the discount rate γ uses long-term rewards only, while a value 0 uses the short-term reward only

Q-learning estimates the cumulative reward discounted for future actions.



Q-learning as reinforcement learning

Q-learning qualifies as a reinforcement learning technique because it does not strictly require labeled data and training. Moreover, the Q-value does not have to be a continuous, differentiable function.

Let's apply our hard-earned knowledge of reinforcement learning to management and optimization of a portfolio of exchange-traded funds.

Implementation

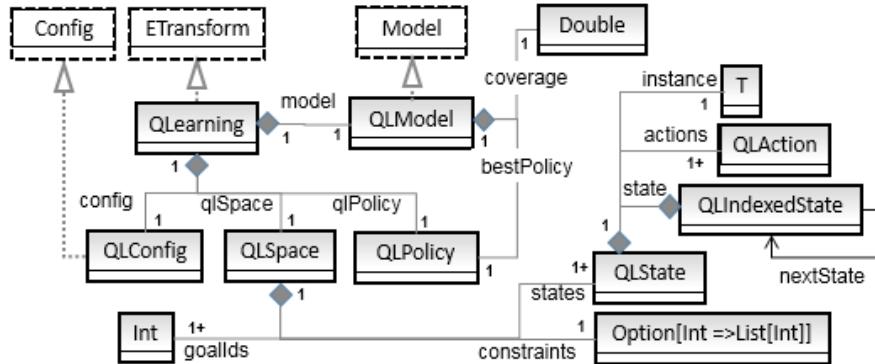
Let's implement the Q-learning algorithm in Scala.

Software design

The key components of the implementation of the Q-learning algorithm are defined as follows:

- The `QLearning` class implements training and prediction methods. It defines a data transformation of the `ETransform` type using an explicit configuration of the `QLConfig` type.
- The `QLSpace` class has two components: a sequence of states of the `QLState` type and the identifier `id` of one or more goal states within the sequence.
- A state, `QLState`, contains a sequence of `QLAction` instances used in its transition to another state and a reference to the object or `instance` for which the state is to be evaluated and predicted.
- An indexed state, `QLIndexedState`, indexes a state in the search toward the goal state.
- An optional constraint function that limits the scope of the search for the next most rewarding action from the current state.
- The model of the `QLModel` type is generated through training. It contains the best policy and the accuracy for a model.

The following diagram shows the key components of the Q-learning algorithm:



The UML components diagram of the Q-learning algorithm

The states and actions

The `QLAction` class specifies the transition of one state with a `from` identifier to another state with the `to` identifier, as shown here:

```
class QLAction(val from: Int, val to: Int)
```

Actions have a Q value (or action-value), a reward, and a probability. The implementation defines these three values in three separate matrices: Q for the action values, R for rewards, and P for probabilities, in order to stay consistent with the mathematical formulation.

A state of the `QLState` type is fully defined by its identifier, `id`, the list of `actions` to transition to some other states, and a `prop` property of the parameterized type, as shown in the following code:

```
class QLState[T](val id: Int,
                 val actions: List[QLAction] = List.empty,
                 val instance: T)
```

The state might not have any actions. This is usually the case of the goal or absorbing state. In this case, the list is empty. The parameterized `instance` is a reference to the object for which the state is computed.

The next step consists of creating the graph or search space.

The search space

The search space is the container responsible for any sequence of states. The `QLSpace` class takes the following parameters:

- The sequence of all the possible states
- The ID of one or several states that have been selected as goals

 **Why multiple goals?**
There is absolutely no requirement that a state space must have a single goal. You can describe a solution to a problem as reaching a threshold or meeting one of the several conditions. Each condition can be defined as a state goal.

The `QLSpace` class is implemented as follows:

```
class QLSpace[T](states: Seq[QLState[T]], goals: Array[Int]) {  
    val statesMap = states.map(st =>(st.id, st)) //1  
    val goalStates = new HashSet[Int] () ++ goals //2  
  
    def maxQ(state: QLState[T],  
             policy: QLPolicy): Double //3  
    def init(state0: Int) //4  
    def nextStates(st: QLState[T]): Seq[QLState[T]] //5  
    ...  
}
```

The constructor of the `QLSpace` class generates a map, `statesMap`. It retrieves the state using its `id` value (line 1) and the array of goals, `goalStates` (line 2). Furthermore, the `maxQ` method computes the maximum action-value, `maxQ`, for a state given a policy (line 3). The implementation of the `maxQ` method is described in the next section.

The `init` method selects an initial state for training episodes (line 4). The state is randomly selected if the `state0` argument is invalid:

```
def init(state0: Int): QLState[T] =  
    if(state0 < 0) {  
        val seed = System.currentTimeMillis+Random.nextLong  
        states((new Random(seed)).nextInt(states.size-1))  
    }  
    else states(state0)
```

Finally, the `nextStates` method retrieves the list of states resulting from the execution of all the actions associated with the `st` state (line 5).

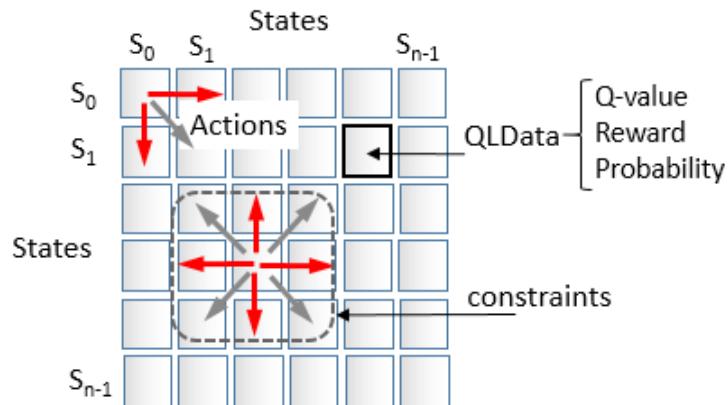
The QLSpace search space is actually created by the apply factory method defined in the QLSpace companion object, as shown here:

```
def apply[T] (goals: Array[Int], instances: Seq[T],
  constraints: Option[Int => List[Int]]): QLSpace[T] = { //6
  val r = Range(n, instances.size)

  val states = instances.zipWithIndex.map{ case(x, n) => {
    val validStates = constraints.map( _(n)).getOrElse(r)
    val actions = validStates.view
      .map(new QLAction(n, _)).filter(n != _.to) //7
    QLState[T](n, actions, x)
  }}
  new QLSpace[T](states, goals)
}
```

The apply method creates a list of states using the instances set, the goals, and the constraints constraining function as inputs (line 6). Each state creates its list of actions. The actions are generated from this state to any other states (line 7).

The search space of states is illustrated in the following diagram:



The state transition matrix with QLData (Q-value, reward, and probability)

The constraints function limits the scope of the actions that can be triggered from any given state, as illustrated in the preceding diagram.

The policy and action-value

Each action has an action-value, a reward, and a potentially probability. The probability variable is introduced to simply model the hindrance or adverse condition for an action to be executed. If the action does not have any external constraint, the probability is 1. If the action is not allowed, the probability is 0.

Dissociating a policy from states

The action and states are the edges and vertices of the search space or search graph. The policy defined by the action-values, rewards, and probabilities is completely dissociated from the graph. The Q-learning algorithm initializes the reward matrix and updates the action-value matrix independently of the structure of the graph.

The `QLData` class is a container for three values: reward, probability, and a value variable for the Q-value, as shown here:

```
class QLData(val reward: Double, val probability: Double) {  
    var value: Double = 0.0  
    def estimate: Double = value*probability  
}
```

Reward and punishment

The probability in the `QLData` class represents the hindrance or difficulty to reach one state from another state. Nothing prevents you from using the probability as a negative reward or punishment. However, its proper definition is to create a soft constraint of a state transition for a small subset of a state. For most applications, the overwhelming majority of state transitions have a probability of 1.0, and therefore, rely on the reward to guide the search toward the goal.

The `estimate` method adjusts the Q-value, `value`, with the probability to reflect any external condition that can impede the action.

Mutable data

You might wonder why the `QLData` class defines a value as a variable instead of a value as recommended by the best Scala coding practices [11:4]. The reason being that an instance of an immutable class can be created for each action or state transition that requires you to update the `value` variable.

The training of the Q-learning model entails iterating across several episodes, each episode being defined as a multiple iteration. For instance, the training of a model with 400 states for 10 episodes of 100 iterations can potentially create 160 million instances of `QLData`. Although not quite elegant, mutability reduces the load on the JVM garbage collector.

Next, let's create a simple schema or class, `QLInput`, to initialize the reward and probability associated with each action as follows:

```
class QLInput(val from: Int, val to: Int,
            val reward: Double, val probability: Double = 1.0)
```

The first two arguments are the identifiers for the `from` source state and the `to` target state for this specific action. The last two arguments are the `reward`, collected at the completion of the action, and its `probability`. There is no need to provide an entire matrix. Actions have a reward of 1 and a probability of 1 by default. You only need to create an input for actions that have either a higher reward or a lower probability.

The number of states and a sequence of input define the policy of the `QLPolicy` type. It is merely a data container, as shown here:

```
class QLPolicy(input: Seq[QLInput]) {
    val numStates = Math.sqrt(input.size).toInt //8

    val qlData = input.map(qlIn =>
        new QLData(qlIn.reward, qlIn.prob)) //9

    def setQ(from: Int, to: Int, value: Double): Unit =
        qlData(from*numStates + to).value = value //10

    def Q(from: Int, to: Int): Double =
        qlData(from*numStates + to).value //11
    def EQ(from: Int, to: Int): Double =
        qlData(from*numStates + to).estimate //12
    def R(from: Int, to: Int): Double =
        qlData(from*numStates + to).reward //13
    def P(from: Int, to: Int): Double =
        qlData(from*numStates + to).probability //14
}
```

The number of states, `numStates`, is the square root of the number of elements of the initial input matrix, `input` (line 8). The constructor initializes the `qlData` matrix of the `QLData` type with the input data, `reward`, and `probability` (line 9). The `QLPolicy` class defines the shortcut methods to update (line 10) and retrieve (line 11) the value, the estimate (line 12), the reward (line 13), and the probability (line 14).

The Q-learning components

The `QLearning` class encapsulates the Q-learning algorithm, and more specifically, the action-value updating equation. It is a data transformation of the `ETransform` type with an explicit configuration of the `QLConfig` type (line 16) (refer to the *Monadic data transformation* section in *Chapter 2, Hello World!*):

```
class QLearning[T] (conf: QLConfig,
  qlSpace: QLSpace[T], qlPolicy: QLPolicy) //15
  extends ETransform[QLConfig](conf) { //16

  type U = QLState[T] //17
  type V = QLState[T] //18

  val model: Option[QLModel] = train //19
  def train: Option[QLModel]
  def nextState(iSt: QLIndexedState[T]): QLIndexedState[T]

  override def |> : PartialFunction[U, Try[V]]
  ...
}
```

The constructor takes the following parameters (line 15):

- `config`: This is the configuration of the algorithm
- `qlSpace`: This is the search space
- `qlPolicy`: This is the policy

The `model` is generated or trained during the instantiation of the class (refer to the *Design template for classifier* section in the *Appendix A, Basic Concepts*) (line 19). The Q-learning algorithm is implemented as an explicit data transformation; therefore, the `U` type of the input element and the `V` type of the output element to the `|>` predictor are initialized as `QLState` (lines 17 and 18).

The configuration of the Q-learning algorithm, `QLConfig`, specifies the learning rate, `alpha`, the discount rate, `gamma`, the maximum number of states (or length) of an episode, `episodeLength`, the number of episodes (or epochs) used in training, `numEpisodes`, and the minimum coverage, `minCoverage`, required to select the best policy as follows:

```
case class QLConfig(val alpha: Double,
    val gamma: Double,
    val episodeLength: Int,
    val numEpisodes: Int,
    val minCoverage: Double) extends Config
```

The `QLearning` class has two constructors defined in its companion object that initializes the policy either from an input matrix of states or from a function that compute the reward and probabilities:

- The client code specifies the `input` function to initialize the state of the Q-learning algorithm from the input data
- The client code specifies the functions to generate the `reward` and `probability` for each action or state transition

The first constructor for the `QLearning` class passes the initialization of states => `Seq[QLInput]`, the sequence of references of `instances` associated with the states, and the `constraints` scope constraining function as an argument, besides the configuration and the goals (line 20):

```
def apply[T](config: QLConfig, //20
    goals: Array[Int],
    input: => Seq[QLInput],
    instances: Seq[T],
    constraints: Option[Int => List[Int]] = None): QLearning[T] = {
    new QLearning[T](config,
        QLSpace[T](goals, instances, constraints),
        new QLPolicy(input))
}
```

The second constructor passes the input data, `xt` (line 21), the `reward` function (line 22), and the `probability` function (line 23) as well as the sequence of references of `instances` associated with the states and the `constraints` scope constraining function as arguments:

```
def apply[T](config: QLConfig,
    goals: Array[Int],
    xt: DblVector, //21
```

```
    reward: (Double, Double) => Double, //22
    probability: (Double, Double) => Double, //23
    instances: Seq[T].
    constraints: Option[Int =>List[Int]] =None) : QLearning[T] ={

        val r = Range(0, xt.size)
        val input = new ArrayBuffer[QLInput] //24
        r.foreach(i =>
            r.foreach(j =>
                input.append(QLInput(i, j, reward(xt(i), xt(j)),
                    probability(xt(i), xt(j)))))

        )
    )
    new QLearning[T](config,
        QLSpace[T](goals, instances, constraints),
        new QLPolicy(input))
}
```

The reward and probability matrices are used to initialize the `input` state (line 24).

The Q-learning training

Let's take a look at the computation of the best policy during training. First, we need to define a `QLModel` model class with the `bestPolicy` optimum policy (or path) and its coverage as parameters:

```
class QLModel(val bestPolicy: QLPolicy,
             val coverage: Double) extends Model
```

The creation of `model` consists of executing multiple episodes to extract the best policy. The training is executed in the `train` method: Each episode starts with a randomly selected state, as shown in the following code:

```
def train: Option[QLModel] = Try {
    val completions = Range(0, config.numEpisodes)
        .map(epoch => if(train(-1)) 1 else 0).sum //25
    completions.toDouble/config.numEpisodes //26
}
.map( coverage => {
    if(coverage > config.minCoverage)
        Some(new QLModel(qlPolicy, coverage)) //27
    else None
}).get
```

The `train` method iterates through the generation of the best policy starting from a randomly selected state `config.numEpisodes` times (line 25). The state coverage is calculated as the percentage of times the search ends with the goal state (line 26). The training succeeds only if the coverage exceeds a threshold value, `config.minAccuracy`, specified in the configuration.

The quality of the model

The implementation uses the accuracy to measure the quality of the model or best policy. The F_1 measure (refer to the *Assessing a model* section in *Chapter 2, Hello World!*), is not appropriate because there are no false positives.

The `train(state0: Int)` method does the heavy lifting at each episode (or epoch). It triggers the search by selecting either the `state0` initial state or a `r` random generator with a new seed, if `state0` is `< 0`, as shown in the following code:

```
case class QLIndexedState[T] (val state: QLState[T],
    val iter: Int)
```

The `QLIndexedState` utility class keeps track of the `state` at a specific iteration, `iter`, within an episode or epoch:

```
def train(state0: Int): Boolean = {
    @tailrec
    def search(iSt: QLIndexedState[T]): QLIndexedState[T]

    val finalState = search(
        QLIndexedState(qlSpace.init(state0), 0)
    )
    if( finalState.index == -1) false //28
    else qlSpace.isGoal(finalState.state) //29
}
```

The implementation of `search` for the goal state(s) from a `state0` predefined or random is a textbook implementation of the Scala tail recursion. Either the recursive search ends if there are no more states to consider (line 28) or the goal state is reached (line 29).

Tail recursion to the rescue

Tail recursion is a very effective construct to apply an operation to every item of a collection [11:5]. It optimizes the management of the function stack frame during the recursion. The annotation triggers a validation of the condition necessary for the compiler to optimize the function calls, as shown here:

```
@tailrec
def search(iSt: QLIndexedState[T]): QLIndexedState[T] = {
    val states = qlSpace.nextStates(iSt.state) //30

    if( states.isEmpty || iSt.iter >= config.episodeLength) //31
        QLIndexedState(iSt.state, -1)

    else {
        val state = states.maxBy(s =>
            qlPolicy.R(iSt.state.id, s.id)) //32
        if( qlSpace.isGoal(state) )
            QLIndexedState(state, iSt.iter) //33

        else {
            val fromId = iSt.state.id
            val r = qlPolicy.R(fromId, state.id)
            val q = qlPolicy.Q(fromId, state.id) //34

            val nq = q + config.alpha*(r +
                config.gamma * qlSpace.maxQ(state, qlPolicy)-q) //35
            qlPolicy.setQ(fromId, state.id, nq) //36
            search(QLIndexedState(state, iSt.iter+1))
        }
    }
}
```

Let's dive into the implementation for the Q action-value updating equation. The `search` method implements the **M5** mathematical expression for each recursion.

The recursion uses the `QLIndexedState` utility class (state, iteration number in the episode) as an argument. First, the recursion invokes the `nextStates` method of `QLSpace` (line 30) to retrieve all the states associated with the `st` current state through its actions, as shown here:

```
def nextStates(st: QLState[T]): Seq[QLState[T]] =
    if( st.actions.isEmpty )
        Seq.empty[QLState[T]]
    else
        st.actions.map(ac => statesMap.get(ac.to) )
```

The search completes and returns the current `state` if the length of the episode (maximum number of states visited) is reached or the `goal` is reached or there is no further state to transition to (line 31). Otherwise, the recursion computes the state to which the transition generates the higher reward `R` from the current policy (line 32). The recursion returns the state with the highest reward if it is one of the goal states (line 33). The method retrieves the current `q` action-value (line 34) and `r` reward matrices from the policy, and then applies the equation to update the action-value (line 35). The method updates the action-value `Q` with the new value `nq` (line 36).

The action-value updating equation requires the computation of the maximum action-value associated with the current state, which is performed by the `maxQ` method of the `QLSpace` class:

```
def maxQ(state: QLState[T], policy: QLPolicy): Double = {
    val best = states.filter(_ != state) //37
        .maxBy(st => policy.EQ(state.id, st.id)) //38
    policy.EQ(state.id, best.id)
}
```

The `maxQ` method filters out the current state (line 37) and then extracts the best state, which maximizes the policy (line 38).

Reachable goal

The algorithm does not require the goal state to be reached for every episode. After all, there is no guarantee that the goal will be reached from any randomly selected state. It is a constraint on the algorithm to follow a positive gradient of the rewards when transitioning between states within an episode. The goal of the training is to compute the best possible policy or sequence of states from any given initial state. You are responsible for validating the model or best policy extracted from the training set, independent from the fact that the goal state is reached for every episode.

The validation

A commercial application may require multiple types of validation mechanisms regarding the states transition, reward, probability, and Q-value matrices.

One critical validation is to verify that the user-defined constraints function does not create a dead end in the search or training of Q-learning. The constraints function establishes the list of states that can be accessed from a given state through actions. If the constraints are too tight, some of the possible search paths may not reach the goal state. Here is a simple validation of the constraints function:

```
def validateConstraints(numStates: Int,  
    constraints: Int => List[Int]): Boolean =  
  Range(0, numStates).exists( constraints(_).isEmpty )
```

The prediction

The last functionality of the QLearning class is the prediction using the model created during training. The `|>` method predicts the optimum state transition (or action) from a given state, `state0`:

```
override def |> : PartialFunction[U, Try[V]] = {  
  case state0: U if(isModel) => Try {  
    if(state0.isGoal) state0 //39  
    else nextState(QLIndexedState[T](state0, 0)).state //40  
  }  
}
```

The `|>` data transformation returns itself if the `state0` input state is the goal (line 39) or computes the best outcome, `nextState`, (line 40) using another tail recursion, as follows:

```
@tailrec  
def nextState(iSt: QLIndexedState[T]): QLIndexedState[T] = {  
  val states = qlSpace.nextStates(iSt.state) //41  
  
  if( states.isEmpty || iSt.iter >= config.episodeLength)  
    iSt //42  
  else {  
    val fromId = iSt.state.id  
    val qState = states.maxBy(s => //43  
      model.map(_.bestPolicy.R(fromId, s.id)).getOrElse(-1.0))  
    nextState(QLIndexedState[T](qState, iSt.iter+1)) //44  
  }  
}
```

The `nextState` method executes the following sequence of invocations:

1. Retrieve the eligible states that can be transitioned to from the current state, `ist.state` (line 41).
2. Return the states if there are no more states or if the method does not converge within the maximum number of allowed iterations, `config.episodeLength` (line 42).
3. Extracts the state, `qState`, with the most rewarding policy (line 43).
4. Increment the `ist.iter` iteration counter (line 44).

The exit condition

The prediction ends when no more states are available or the maximum number of iterations within the episode is exceeded. You can define a more sophisticated exit condition. The challenge is that there is no explicit error or loss variable/function that can be used except the temporal difference error.

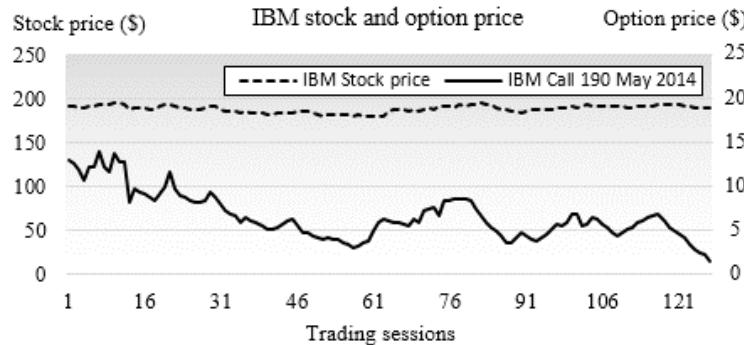
The `|>` prediction method returns either the best possible state or `None` if the model cannot be created during training.

Option trading using Q-learning

The Q-learning algorithm is used in many financial and market trading applications [11:6]. Let's consider the problem of computing the best strategy to trade certain types of options given some market conditions and trading data.

The **Chicago Board Options Exchange (CBOE)** offers an excellent online tutorial on options [11:7]. An option is a contract that gives the buyer the right but not the obligation to buy or sell an underlying asset at a specific price on or before a certain date (refer to the *Options trading* section under *Finances 101* in the *Appendix A, Basic Concepts*.) There are several option pricing models, the Black-Scholes stochastic partial differential equations being the most recognized [11:8].

The purpose of the exercise is to predict the price of an option on a security for N days in the future according to the current set of observed features derived from the time to expiration, price of the security, and volatility. Let's focus on the call options of a given security, IBM. The following chart plots the daily price of the IBM stock and its derivative call option for May 2014 with a strike price of \$190:



The IBM stock and Call \$190 May 2014 pricing in May-Oct 2013

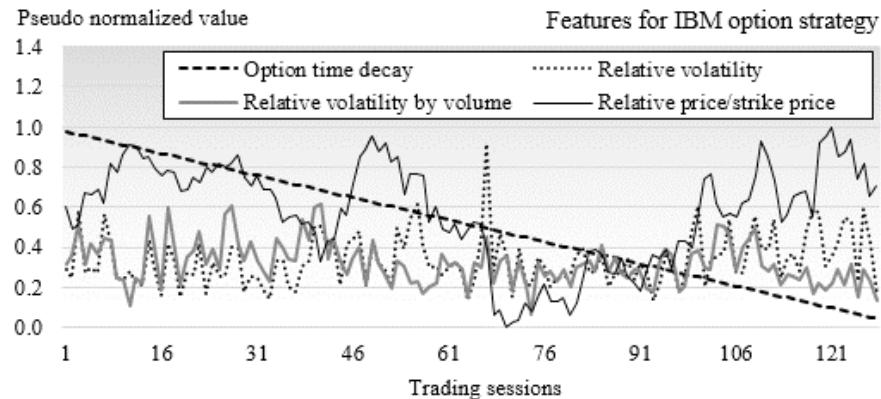
The price of an option depends on the following parameters:

- Time to expiration of the option (time decay)
- The price of the underlying security
- The volatility of returns of the underlying asset

The pricing model usually does not take into account the variation in trading volume of the underlying security. Therefore, it would be quite interesting to include it in our model. Let's define the state of an option using the following four normalized features:

- **Time decay** (`timeToExp`): This is the time to expiration once normalized over $[0, 1]$.
- **Relative volatility** (`volatility`): This is the relative variation of the price of the underlying security within a trading session. It is different from the more complex volatility of returns defined in the Black-Scholes model, for example.
- **Volatility relative to volume** (`vltyByVol`): This is the relative volatility of the price of the security adjusted for its trading volume.
- **Relative difference between the current price and strike price** (`priceToStrike`): This measures the ratio of the difference between price and strike price to the strike price.

The following graph shows the four normalized features for the IBM option strategy:



Normalized relative stock price volatility, volatility relative to trading volume, and price relative to strike price for the IBM stock

The implementation of the option trading strategy using Q-learning consists of the following steps:

1. Describing the property of an option
2. Defining the function approximation
3. Specifying the constraints on the state transition

The OptionProperty class

Let's select $N = 2$ as the number of days in the future for our prediction. Any longer-term prediction is quite unreliable because it falls outside the constraint of the discrete Markov model. Therefore, the price of the option two days in the future is the value of the reward – profit or loss.

The OptionProperty class encapsulates the four attributes of an option (line 45) as follows:

```
class OptionProperty(timeToExp: Double, volatility: Double,
    vltyByVol: Double, priceToStrike: Double) { //45

    val toArray = Array[Double](
        timeToExp, volatility, vltyByVol, priceToStrike
    )
}
```

A modular design



The implementation avoids subclassing the `QLState` class to define the features of our option pricing model. The state of the option is a parameterized prop parameter for the state class.

The OptionModel class

The `OptionModel` class is a container and a factory for the properties of the option. It creates the list of `propsList` option properties by accessing the data source of the four features introduced earlier. It takes the following parameters:

- The symbol of the security.
- The strike price for the `strikePrice` option.
- The source of data, `src`.
- The minimum time decay or time to expiration, `minTDecay`. Out-of-the-money options expire worthless and in-the-money options have very different price behavior as they get closer to the expiration date (refer to the *Options trading* section in the *Appendix A, Basic Concepts*). Therefore, the last `minTDecay` trading sessions prior to the expiration date are not used in the training of the model.
- The number of steps (or buckets), `nSteps`. It is used in approximating the values of each feature. For instance, an approximation of four steps creates four buckets [0, 25], [25, 50], [50, 75], and [75, 100].

The implementation of the `OptionModel` class is as follows:

```
class OptionModel(symbol: String,  strikePrice: Double,
                 src: DataSource, minExpT: Int, nSteps: Int) {

    val propsList = (for {
        price <- src.get(adjClose)
        volatility <- src.get(volatility)
        nVolatility <- normalize(volatility)
        vltyByVol <- src.get(volatilityByVol)
        nvltbyVol <- normalize(vltyByVol)
        priceToStrike <- normalize(price.map(p =>
            (1.0 - strikePrice/p)))
    } yield {
        nVolatility.zipWithIndex./:(List[OptionProperty]()) { //46
            case(xs, (v,n)) => {
                val normDecay = (n + minExpT).toDouble/
                    (price.size + minExpT) //47
                OptionProperty(priceToStrike, v, n, normDecay)
            }
        }
    })
}
```

```

        new OptionProperty(normDecay, v, nVltyByVol(n),
            priceToStrike(n)) :: xs
    }
}.drop(2).reverse //48
})
.getOrElse(List.empty[OptionProperty].)
}

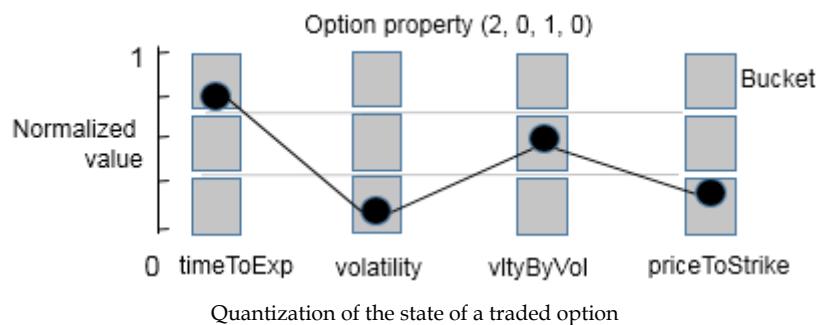
def quantize(o: DblArray): Map[Array[Int], Double]
}

```

The factory uses the `zipWithIndex` Scala method to represent the index of the trading sessions (line 46). All feature values are normalized over the interval [0, 1], including the time decay (or time to expiration) of the `normDecay` option (line 47). The instantiation of the `OptionModel` class generates a list of `OptionProperty` elements if the constructor succeeds (line 48), an empty list otherwise.

Quantization

The four properties of the option are continuous values, normalized as a probability [0, 1]. The states in the Q-learning algorithm are discrete and require a quantization or categorization known as a **function approximation**; although a function approximation scheme can be quite elaborate [11:9]. Let's settle for a simple linear categorization, as illustrated in the following diagram:



The function approximation defines the number of states. In this example, a function approximation that converts a normalized value into three intervals or buckets generates $3^4 = 81$ states or potentially $3^8 \cdot 3^4 = 6480$ actions! The maximum number of states for l buckets function approximation and n features is l^n with a maximum number of $l^n \cdot l^n$ actions.

Quantization or function approximation guidelines

The design of the function to approximate the state of options has to address the following two conflicting requirements:



- Accuracy demands a fine-grained approximation
- Limited computation resources restrict the number of states, and therefore, level of approximation

The `quantize` method of the `OptionModel` class converts the normalized value of each option property of features into an array of bucket indices. It returns a map of profit and loss for each bucket keyed on the array of bucket indices, as shown in the following code:

```
def quantize(o: DblArray): Map[Array[Int], Double] = {
    val mapper = new mutable.HashMap[Int, Array[Int]] //49
    val _acc = new NumericAccumulator[Int] //50

    val acc = propsList.view.map(_.toArray)
        .map( toArrayInt(_)) //51
        .map(ar => {
            val enc = encode(ar) //52
            mapper.put(enc, ar)
            enc
        }).zip(o)
        .:/(_acc) {
            case (acc, (t, y)) => { //53
                acc += (t, y)
                acc
            }
        }
    acc.map {case (k, (v, w)) => (k, v/w)} //54
        .map {case (k, v) => (mapper(k), v)}.toMap
}
```

The method creates a `mapper` instance to index the array of buckets (line 49). An `acc` accumulator of the `NumericAccumulator` type extends `Map[Int, (Int, Double)]` and computes the tuple (number of occurrences of features on each buckets and the sum of increase or decrease of the option price) (line 50). The `toArrayInt` method converts the value of each option property (`timeToExp`, `volatility`, and so on) into the index of the appropriate bucket (line 51). The array of indices is then encoded (line 52) to generate the id or index of a state. The method updates the accumulator with the number of occurrences and the total profit and loss for a trading session for the option (line 53). It finally computes the reward on each action by averaging the profit and loss on each bucket (line 54).

A view is used in the generation of the list of `OptionProperty` to avoid unnecessary object creation.

The source code for the `toArrayInt` and `encode` methods and `NumericAccumulator` is documented and available online.

Putting it all together

The final piece of the puzzle is the code that configures and executes the Q-learning algorithm on one or several options on a security, IBM:

```
val STOCK_PRICES = "resources/data/chap11/IBM.csv"
val OPTION_PRICES = "resources/data/chap11/IBM_O.csv"
val QUANTIZER = 4
val src = DataSource(STOCK_PRICES, false, false, 1) //55

val model = for {
    option <- Try(createOptionModel(src)) //56
    oPrices <- DataSource(OPTION_PRICES, false).extract //57
    _model <- createModel(option, oPrices) //58
} yield _model
```

The preceding implementation creates the Q-learning model with the following steps:

1. Extract the historical prices for the IBM stock by instantiating a data source, `src` (line 55).
2. Create an `option` model (line 56).
3. Extract the historical prices `oPrices` for option call \$190 May 2014 (line 57).
4. Create the model, `_model`, with a `goalStr` predefined goal (line 58).

The code is as follows:

```
val STRIKE_PRICE = 190.0
val MIN_TIME_EXPIRATION = 6

def createOptionModel(src: DataSource): OptionModel =
    new OptionModel("IBM", STRIKE_PRICE, src,
        MIN_TIME_EXPIRATION, QUANTIZER)
```

Let's take a look at the `createModel` method that takes the option pricing model, option, and the historical prices for `oPrices` options as arguments:

```
val LEARNING_RATE = 0.2
val DISCOUNT_RATE = 0.7
val MAX_EPISODE_LEN = 128
val NUM_EPISODES = 80

def createModel(option: OptionModel, oPrices: DblArray,
               alpha: Double, gamma: Double): Try[QLModel] = Try {

    val qPriceMap = option.quantize(oPrices) //59
    val numStates = qPriceMap.size

    val qPrice = qPriceMap.values.toVector //60
    val profit = zipWithShift(qPrice, 1).map{case(x,y) => y - x} //61
    val maxProfitIndex = profit.zipWithIndex.maxBy(_._1)._2 //62

    val reward = (x: Double, y: Double)
                 => Math.exp(30.0*(y - x)) //63
    val probability = (x: Double, y: Double) =>
        if(y < 0.3*x) 0.0 else 1.0 //64

    if( !validateConstraints(profit.size, neighbors)) //65
        throw new IllegalStateException(" ... ")

    val config = QLConfig(alpha, gamma,
                          MAX_EPISODE_LEN, NUM_EPISODES) //66
    val instances = qPriceMap.keySet.toSeq.drop(1)
    QLearning[Array[Int]](config, Array[Int](maxProfitIndex),
                          profit, reward, probability,
                          instances, Some(neighbors)).getModel //67
}
```

The method quantizes the option prices map, `oPrices` (line 59), extracts the historical option prices, `qPrice` (line 60), computes the profit as the difference in the price of the option between two consecutive trading sessions (line 61), and computes the index, `maxProfitIndex`, of the trading session with the highest profit (line 62). The state with the `maxProfitIndex` index is selected as the goal.

The input matrix is automatically generated using the reward and probability functions. The reward function rewards the state transition proportionally to the profit (line 63). The probability function punishes the state transition for which the loss $y - x$ is greater than $0.3*x$ by setting the probability value to 0 (line 64).



Initialization of rewards and probabilities

In our example, the reward and probability matrices are automatically generated through two functions. An alternative approach consists of initializing these two matrices using either historical data or educated guesses.

The validateConstraints method of the QLearning companion object validates the neighbors constraints function, as described in the *The validation* section (line 65).

The last two steps consists of creating a configuration, config, for the Q-learning algorithm (line 66) and training the model by instantiating the QLearning class with the appropriate parameters, including the neighbors method that defines the neighboring states for any given state (line 67). The neighbors method is described in the documented source code available online.

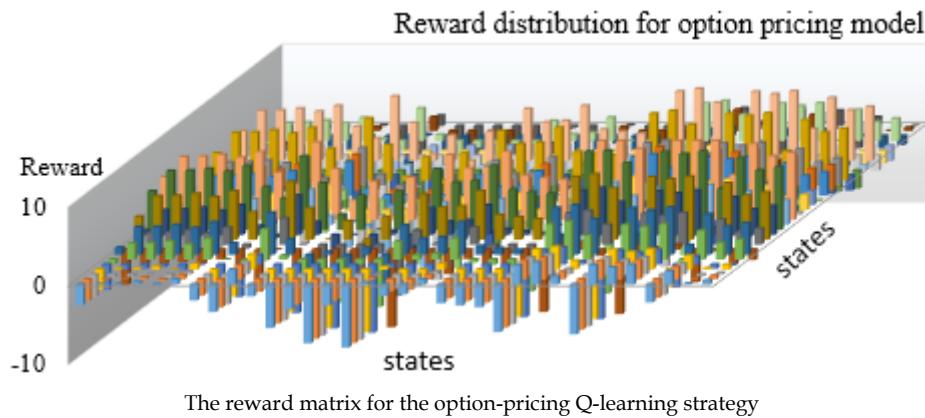


The anti-goal state

The goal state is the state with the highest assigned reward. It is a heuristic to reward a strategy for a good performance. However, it is conceivable and possible to define an anti-goal state with the highest assigned penalty or the lowest assigned reward to guide the search away from some condition.

Evaluation

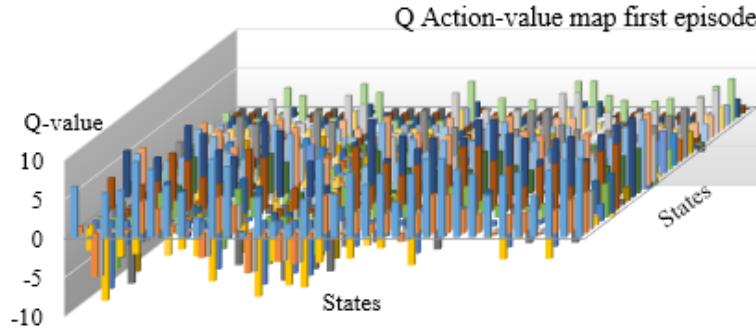
Besides the function approximation, the size of the training set has an impact on the number of states. A well-distributed or large training set provides at least one value for each bucket created by the approximation. In this case, the training set is quite small and only 34 out of 81 buckets have actual values. As result, the number of states is 34. The initialization of the Q-learning model generates the following reward matrix:



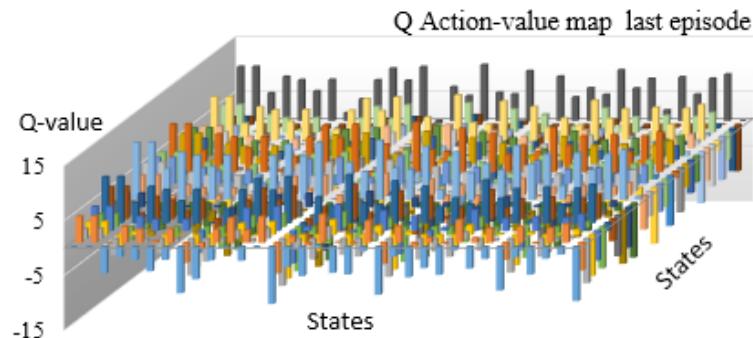
The graph visualizes the distribution of the rewards computed from the profit and loss of the option. The xy plane represents the actions between states. The states' IDs are listed on x and y axes. The z axis measures the actual value of the reward associated with each action.

The reward reflects the fluctuation in the price of the option. The price of an option has a higher volatility than the price of the underlying security.

The xy reward matrix R is rather highly distributed. Therefore, we select a small value for the learning rate 0.2 to reduce the impact of the previous state on the new state. The value for the discount rate 0.7 accommodates the fact that the number of states is limited. There is no reason to compute the future discounted reward using a long sequence of states. The training of the policies generates the following action-value matrix Q of 34 states by 34 states after the first episode:



The distribution of the action-values between states at the end of the first episode reflects the distribution of the reward across state-to-state action. The first episode consists of a sequence of nine states from an initial randomly selected state to the goal state. The action-value map is compared to the map generated after 20 episodes in the following graph:



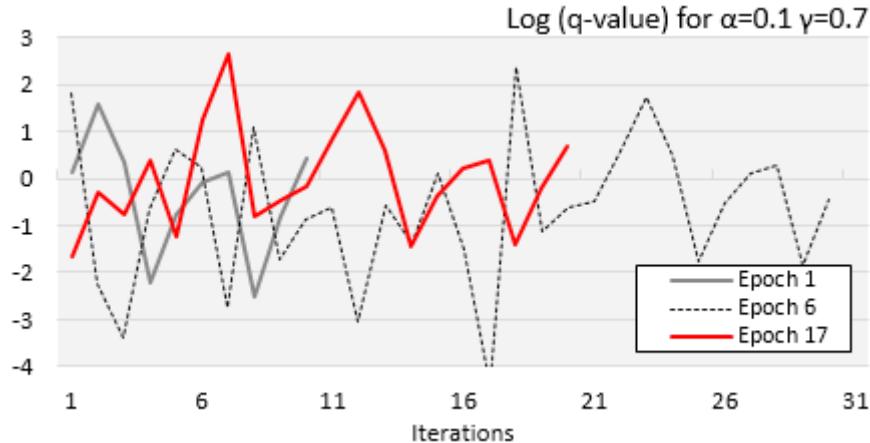
The Q Action-Value matrix for the last episode (epoch)

The action-value map at the end of the last episode shows some clear patterns. Most of the rewarding actions transition from a large number of states (X axis) to a smaller number of states (Y axis). The chart illustrates the following issues with the small training sample:

- The small size of the training set forces us to use an approximate representation of each feature. The purpose is to increase the odds that most buckets have, that is, at least one data point.
- However, a loose function approximation or quantization tends to group quite different states into the same bucket.
- The bucket with a very low number can potentially mischaracterize one property or feature of a state.

Reinforcement Learning

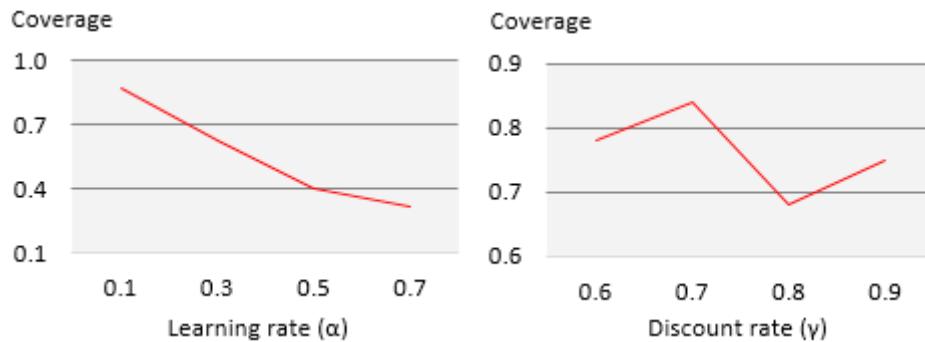
The next test is to display the profile of the log of the Q-value (`QLData.value`) as the recursive search (or training) progress for different episodes or epochs. The test uses a learning rate $\alpha = 0.1$ and a discount rate $\gamma = 0.9$.



The profile of the log (Q-Value) for different epochs during Q-learning training

The preceding chart illustrates the fact that the Q-value for each profile is independent of the order of the epochs during training. However, the length of the profile (or number of iterations to reach the goal state) depends on the initial state, which is selected randomly, in this example.

The last test consists of evaluating the impact of the learning rate and discount rate on the coverage of the training:



Training coverage versus learning rate and discount rate

The coverage (percentage of an episode or epoch for which the goal state is reached) decreases as the learning rate increases. The result confirms the general rule of using learning rate < 0.2 . The similar test to evaluate the impact of the discount rate on the coverage is inconclusive.

Pros and cons of reinforcement learning

Reinforcement learning algorithms are ideal for the following problems:

- Online learning
- The training data is small or nonexistent
- A model is nonexistent or poorly defined
- Computation resources are limited

However, these techniques perform poorly in the following cases:

- The search space (number of possible actions) is large because the maintenance of the states, action graph, and rewards matrix becomes challenging
- The execution is not always predictable in terms of scalability and performance

Learning classifier systems

J. Holland introduced the concept of **learning classifier systems (LCS)** more than 30 years ago as an extension to evolutionary computing [11:10].

Learning classifier systems are a kind of rule-based system with general mechanisms for processing rules in parallel, for adaptive generation of new rules, and for testing the effectiveness of new rules.

However, the concept started to get the attention of computer scientists only a few years ago, with the introduction of several variants of the original concept, including **extended learning classifiers (XCS)**. Learning classifier systems are interesting because they combine rules, reinforcement learning, and genetic algorithms.

Disclaimer

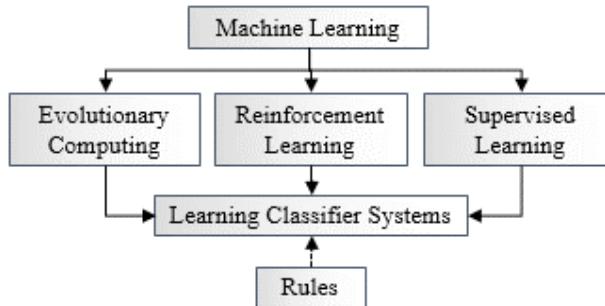
The implementation of the extended learning classifier is presented for informational purposes only. Validating XCS against a known and labeled population of rules is a very significant endeavor. The source code snippet is presented only to illustrate the different components of the XCS algorithm.

Introduction to LCS

Learning classifier systems merge the concepts of reinforcement learning, rule-based policies, and evolutionary computing. This unique class of learning algorithms represents the merger of the following research fields [11:11]:

- Reinforcement learning
- Genetic algorithms and evolutionary computing
- Supervised learning
- Rule-based knowledge encoding

Let's take a look at the following diagram:



A diagram of the scientific disciplines required for learning classifier systems

Learning classifier systems are an example of **complex adaptive systems**. A learning classifier system has the following four components:

- **A population of classifiers or rules:** This evolves over time. In some cases, a domain expert creates a primitive set of rules (core knowledge). In other cases, the rules are randomly generated prior to the execution of the learning classifier system.
- **A genetic algorithm-based discovery engine:** This generates new classifiers or rules from the existing population. This component is also known as the **rules discovery module**. The rules rely on the same pattern of evolution of organisms introduced in the previous chapter. The rules are encoded as strings or bit strings to represent a condition (predicate) and action.
- **A performance or evaluation function:** This measures the positive or negative impact of the actions from the fittest classifiers or policies.

- **A reinforcement learning component:** This rewards or punishes the classifiers that contribute to the action, as seen in the previous section. The rules that contribute to an action that improves the performance of the system are rewarded, while those that degrade the performance of the system are punished. This component is also known as the credit assignment module.

Why LCS?

Learning classifier systems are particularly appropriate to problems in which the environment is constantly changing and are the combinations of a learning strategy and an evolutionary approach to build and maintain a knowledge base [11:12].

Supervised learning methods alone can be effective on large datasets, but they require either a significant amount of labeled data or a reduced set of features to avoid overfitting. Such constraints may not be practical in the case of ever-changing environments.

The last 20 years have seen the introduction of many variants of learning classifier systems that belong to the following two categories:

- Systems for which accuracy is computed from the correct predictions and that apply the discovery to a subset of those correct classes. They incorporate elements of supervised learning to constrain the population of classifiers. These systems are known to follow the **Pittsburgh approach**.
- Systems that explore all the classifiers and apply rule accuracy to the genetic selection of the rules. Each individual classifier is a rule. These systems are known to follow the **Michigan approach**.

The rest of this section is dedicated to the second type of learning classifiers – more specifically, extended learning classifier systems. In a context of LCS, the term *classifier* refers to the predicate or rule generated by the system. From this point on, the term *rule* replaces the term *classifier* to avoid confusion with the more common definition of classification.

Terminology

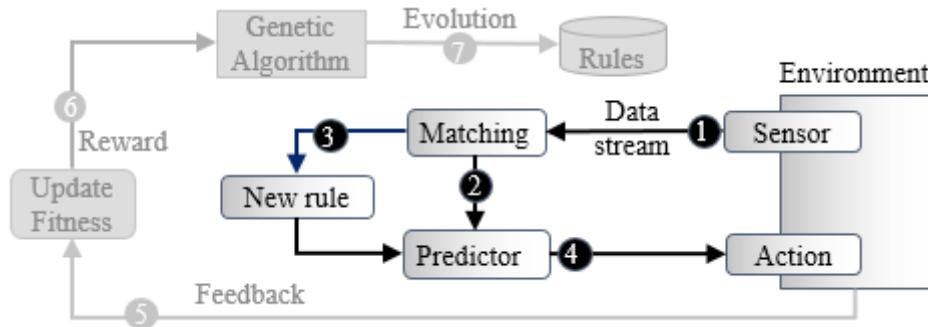
Each domain of research has its own terminology and LCS is no exception. The terminology of LCS consists of the following terms:

- **Environment:** These are the environment variables in the context of reinforcement learning.
- **Agent:** An agent used in reinforcement learning.

- **Predicate:** A clause or fact using the format, *variable-operator-value*, and usually implemented as (operator, variable value); for example, *Temperature-exceeds-87F* or ('Temperature', 87F), *Hard drive-failed* or ('Status hard drive', FAILED), and so on. It is encoded as a gene in order to be processed by the genetic algorithm.
- **Compound predicate:** This is the composition of several predicates and Boolean logic operators, which is usually implemented as a logical tree (for example, ((*predicate1 AND predicate2*) OR *predicate3*) is implemented as OR (AND (*predicate1, predicate2*), *predicate3*)). It uses a chromosome representation.
- **Action:** This is a mechanism that alters the environment by modifying the value of one or several of its parameters using a format (*type of action, target*); for example, *change thermostat settings, replace hard drive*, and so on.
- **Rule:** This is a formal first-order logic formula using the format *IF compound predicate THEN sequence of action*; for example, *IF gold price < \$1140 THEN sell stock of oil and gas producing companies*.
- **Classifier:** This is a rule in the context of an LCS.
- **Rule fitness or score:** This is identical to the definition of the fitness or score in the genetic algorithm. In the context of an LCS, it is the probability of a rule to be invoked and fired in response to the change in environment.
- **Sensors:** These are environment variables monitored by an agent; for example, the temperature and hard drive status.
- **Input data stream:** This is the flow of data generated by sensors. It is usually associated with online training.
- **Rule matching:** This is a mechanism to match a predicate or compound predicate with a sensor.
- **Covering:** This is the process of creating new rules to match a new condition (sensor) in the environment. It generates the rules by either using a random generator or mutating existing rules.
- **Predictor:** This is an algorithm to find the action with the maximum number of occurrences within a set of matching rules.

Extended learning classifier systems

Similar to reinforcement learning, the XCS algorithm has an **exploration** phase and an **exploitation** phase. The exploitation process consists of leveraging the existing rules to influence the target environment in a profitable or rewarding manner:

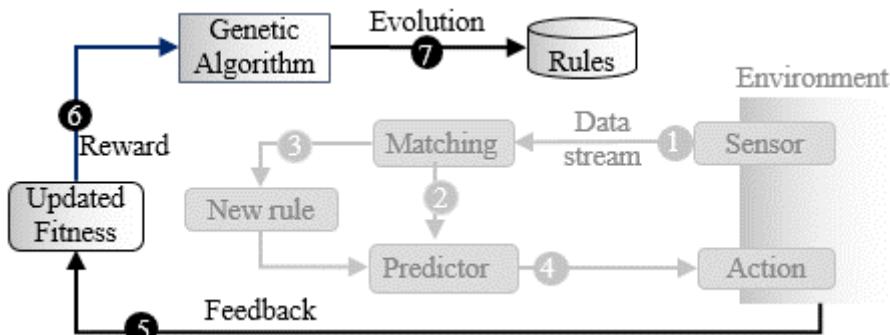


The exploitation component of the XCS algorithm

The following list describes each numbered block:

1. Sensors acquire new data or events from the system.
2. Rules for which the condition matches the input event are extracted from the current population.
3. A new rule is created if no match is found in the existing population. This process is known as covering.
4. The chosen rules are ranked by their fitness values, and the rules with the highest predicted outcome are used to trigger the action.

The purpose of exploration components is to increase the rule base as a population of the chromosomes that encode these rules.



Exploration components of the XCS algorithm

The following list describes each numbered block of the block diagram:

1. Once the action is performed, the system rewards the rules for which the action has been executed. The reinforcement learning module assigns credit to these rules.
2. Rewards are used to update the rule fitness, applying evolutionary constraints to the existing population.
3. The genetic algorithm updates the existing population of classifiers/rules using operators such as crossover and mutation.

XCS components

This section describes the key classes of the XCS. The implementation leverages the existing design of the genetic algorithm and the reinforcement learning. It is easier to understand the inner workings of the XCS algorithm with a concrete application.

Application to portfolio management

Portfolio management and trading have benefited from the application of extended learning classifiers [11:13]. The use case is the management of a portfolio of exchange-traded funds in an ever-changing financial environment. Contrary to stocks, exchange-traded funds are representative of an industry-specific group of stocks or the financial market at large. Therefore, the price of these ETFs is affected by the following macroeconomic changes:

- Gross domestic product
- Inflation
- Geopolitical events
- Interest rates

Let's select the value of the 10-year Treasury yield as a proxy for the macroeconomic conditions, for the sake of simplicity.

The portfolio has to be constantly adjusted in response to any specific change in the environment or market condition that affects the total value of the portfolio, and this can be done by referring to the following table:

XCS component	Portfolio management
Environment	This is the portfolio of securities defined by its composition, total value, and the yield of the 10-year Treasury bond
Action	This is the change in the composition of the portfolio

XCS component	Portfolio management
Reward	This is the profit and loss of the total value of the portfolio
Input data stream	This is the feed of the stock and bond price quotation
Sensor	This is the trading information regarding securities in the portfolio such as price, volume, volatility, yield, and the yield of the-10 year Treasury bond
Predicate	This is the change in the composition of the portfolio
Action	This rebalances a portfolio by buying and selling securities
Rule	This is the association of trading data with the rebalancing of a portfolio

The first step is to create an initial set of rules regarding the portfolio. This initial set can be created randomly, like the initial population of a genetic algorithm or defined by a domain expert.

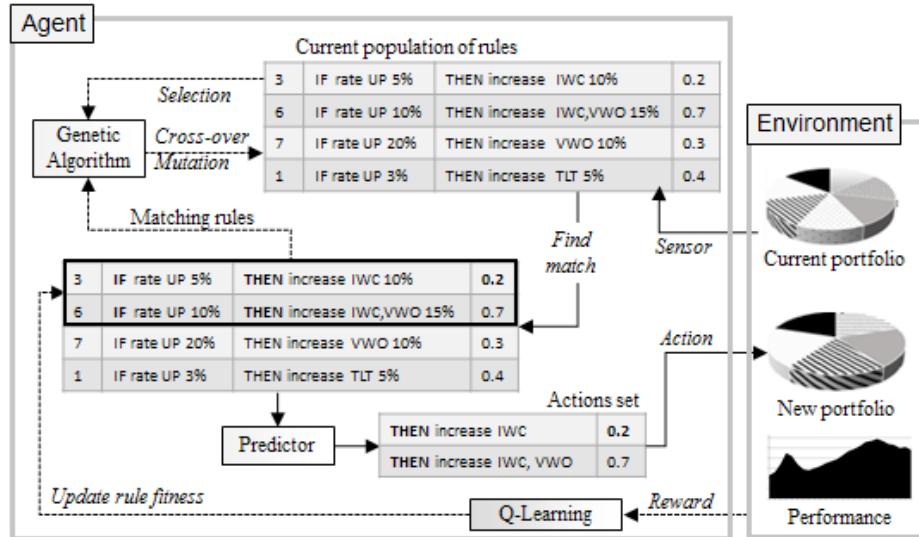
The XCS initial population

Rules or classifiers are defined and/or refined through evolution. Therefore, there is no absolute requirement for the domain expert to set up a comprehensive knowledge base. In fact, rules can be randomly generated at the start of the training phase. However, seeding the XCS initial population with a few relevant rules improves the odds of having the algorithm converge quickly.

You are invited to initialize the population of rules with as many relevant and financially sound trading rules as possible. Over time, the execution of the XCS algorithm will confirm whether or not the initial rules are indeed appropriate. The following diagram describes the application of the XCS algorithm to the composition of a portfolio of ETFs, such as VWO, TLT, IWC, and so on, with the following components:

- The population of trading rules
- An algorithm to match rules and compute the prediction
- An algorithm to extract the actions sets
- The Q-learning module to assign a credit or reward to the selected rules
- The genetic algorithm to evolve the population of rules

Let's take a look at the following diagram:



An overview of the XCS algorithm to optimize the portfolio allocation

The agent responds to the change in the allocation of ETFs in the portfolio by matching one of the existing rules.

Let's build the XCS agent from the ground.

The XCS core data

There are three types of data that are manipulated by the XCS agent:

- **Signal:** This is the trading signal.
- **XcsAction:** This is the action on the environment. It subclasses a Gene defined in the genetic algorithm.
- **XcsSensor:** This is the sensor or data from the environment.

The Gene class was introduced for the evaluation of the genetic algorithm in the *Trading signals* section in *Chapter 10, Genetic Algorithms*. The agent creates, modifies, and deletes actions. It makes sense to define these actions as mutable genes, as follows:

```
class XcsAction(sensorId: String, target: Double)
  (implicit quantize: Quantization, encoding: Encoding) //1
  extends Gene(sensorId, target, EQUAL)
```

The quantization and encoding of the `XcsAction` into a `Gene` has to be explicitly declared (line 1). The `XcsAction` class has the identifier of the `sensorId` sensor and the target value as parameters. For example, the action to increase the number of shares of ETF, VWO in the portfolio to 80 is defined as follows:

```
val vwoTo80 = new XcsAction("VWO", 80.0)
```

The only type of action allowed in this scheme is setting a value using the `EQUAL` operator. You can create actions that support other operators such as `+=` used to increase an existing value. These operators need to implement the `operator` trait, as explained in the *Trading operators* section in *Chapter 10, Genetic Algorithms*.

Finally, the `XcsSensor` class encapsulates the `sensorId` identifier for the variable and value of the sensor, as shown here:

```
case class XcsSensor(val sensorId: String, val value: Double)
val new10ytb = XcsSensor("10yTBYield", 2.76)
```

Setters and getters

In this simplistic scenario, the sensors retrieve a new value from an environment variable. The action sets a new value to an environment variable. You can think of a sensor as a get method of an environment class and an action as a set method with variable/sensor ID and value as arguments.

XCS rules

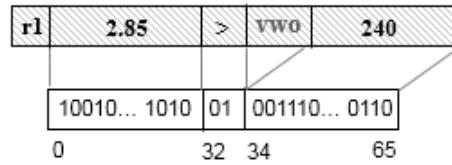
The next step consists of defining a rule of the `XcsRule` type as a pair of two genes: a signal and an action, as shown in the following code:

```
class XcsRule(val signal: Signal, val action: XcsAction)
```

The rule: *r1: IF(yield 10-year TB > 2.84%) THEN reduce VWO shares to 240* is implemented as follows:

```
val signal = new Signal("10ytb", 2.84, GREATER_THAN)
val action = new XcsAction("vwo", 240)
val r1 = new XcsRule(signal, action)
```

The agent encodes the rule as a chromosome using 2 bits to represent the operator and 32 bits for values, as shown in the following diagram:



In this implementation, there is no need to encode the type of action as the agent uses only one type of action—set. A complex action requires encoding of its type.

Knowledge encoding

This example uses very simple rules with a single predicate as the condition. Real-world domain knowledge is usually encoded using complex rules with multiple clauses. It is highly recommended that you break down complex rules into multiple basic rules of classifiers.

Matching a rule to a new sensor consists of matching the sensor to the signal. The algorithm matches the new `new10ytb` sensor (line 2) against the signal in the current population of `s10ytb1` (line 3) and `s10ytb2` (line 4) rules that use the same sensor or the `10ytb` variable as follows:

```
val new10ytb = new XcsSensor("10ytb", 2.76) //2
val s10ytb1 = Signal("10ytb", 2.5, GREATER_THAN) //3
val s10ytb2 = Signal("10ytb", 2.2, LESS_THAN) //4
```

In this case, the agent selects the `r23` rule but not `r34` in the existing population. The agent then adds the `act12` action to the list of possible actions. The agent lists all the rules that match the `r23`, `r11`, and `r46` sensors, as shown in the following code:

```
val r23: XcsRule(s10yTB1, act12) //5
val r11: XcsRule(s10yTB6, act6)
val r46: XcsRule(s10yTB7, act12) //6
```

The action with the most references, `act12`, (lines 5 and 6) is executed. The Q-learning algorithm computes the reward from the profit or loss incurred by the portfolio following the execution of the selected `r23` and `r46` rules. The agent uses the reward to adjust the fitness of `r23` and `r46`, before the genetic selection in the next reproduction cycle. These two rules will reach and stay in the top tier of the rules in the population, until either a new genetic rule modified through crossover and mutation or a rule created through covering, triggers a more rewarding action on the environment.

Covering

The purpose of the covering phase is to generate new rules if no rule matches the input or sensor. The `cover` method of an `XcsCover` singleton generates a new `XcsRule` instance given a sensor and an existing set of actions, as shown here:

```
val MAX_NUM_ACTIONS = 2048

def cover(sensor: XcsSensor, actions: List[XcsAction])
  (implicit quant: Quantization, encoding: Encoding): List[XcsRule] =
  actions./:(List[XcsRule]()) ((xs, act) => {
    val signal = Signal(sensor.id, sensor.value,
      new SOperator(Random.nextInt(Signal.numOperators)))
    new XcsRule(signal, XcsAction(act, Random)) :: xs
  })
}
```

You might wonder why the `cover` method uses a set of actions as arguments knowing that covering consists of creating new actions. The method mutates (^ operator) an existing action to create a new one instead of using a random generator. This is one of the advantages of defining an action as a gene. One of the constructors of `XcsAction` executes the mutation, as follows:

```
def apply(action: XcsAction, r: Random): XcsAction =
  (action ^ r.nextInt(XCSACTION_SIZE))
```

The index of the operator `r` type is a random value in the interval [0, 3] because a signal uses four types of operators: None, >, <, and =.

An implementation example

The `Xcs` class has the following purposes:

- `gaSolver`: This is the selection and generation of genetically modified rules
- `qlLearner`: This is the rewarding and scoring the rules
- `Xcs`: These are the rules for matching, covering, and generation of actions

The extended learning classifier is a data transformation of the `ETransform` type with an explicit configuration of the `XcsConfig` type (line 8) (refer to the *Monadic data transformation* section in *Chapter 2, Hello World!*):

```
class Xcs(config: XcsConfig,
  population: Population[Signal],
  score: Chromosome[Signal] => Unit,
  input: Array[QLInput]) //7
  extends ETransform[XcsConfig](config) { //8

  type U = XcsSensor //9
  type V = List[XcsAction] //10

  val solver = GASolver[Signal](config.gaConfig, score)
  val features = population.chromosomes.toSeq
  val qLearner = QLearning[Chromosome[Signal]]( //11
    config.qlConfig, extractGoals(input), input, features)
  override def |> : PartialFunction[U, Try[V]] =
  ...
}
```

The XCS algorithm is initialized with a configuration `config`, an initial set of rules `population`, a fitness function `score`, and an `input` to the Q-learning policy generate reward matrix for `qLearner` (line 7). Being an explicit data transformation, the `U` type of an input element and the `V` type of the output element to the `|>` predictor are initialized as `XcsSensor` (line 9) and `List[XcsAction]` (line 10).

The goals and number of states are extracted from the input to the policy of the Q-learning algorithm.

In this implementation, the `solver` generic algorithm is mutable. It is instantiated along with the `Xcs` container class. The Q-learning algorithm uses the same design, as any classifier, as immutable. The model of Q-learning is the best possible policy to reward rules. Any changes in the number of states or the rewarding scheme require a new instance of the learner.

Benefits and limitations of learning classifier systems

Learning classifier systems and XCS in particular, hold many promises, which are listed as follows:

- They allow nonscientists and domain experts to describe the knowledge using familiar Boolean constructs and inferences such as predicates and rules

- They provide analysts with an overview of the knowledge base and its coverage by distinguishing between the need for exploration and exploitation of the knowledge base

However, the scientific community has been slow to recognize the merits of these techniques. The wider adoption of learning classifier systems is hindered by the following factors:

- The large number of parameters used in both exploration and exploitation phases adds to the sheer complexity of the algorithm.
- There are too many competitive variants of learning classifier systems
- There is no clear unified theory to validate the concept of evolutionary policies or rules. After all, these algorithms are the merger of standalone techniques. The accuracy and performance of the execution of many variants of the learning classifier systems depend on each component as well as the interaction between components.
- An execution that is not always predictable in terms of scalability and performance.

Summary

The software engineering community sometimes overlooks reinforcement learning algorithms. Let's hope that this chapter provides adequate answers to the following questions:

- What is reinforcement learning?
- What are the different types of algorithms that qualify as reinforcement learning?
- How can we implement the Q-learning algorithm in Scala?
- How can we apply Q-learning to the optimization of option trading?
- What are the pros and cons of using reinforcement learning?
- What are learning classifier systems?
- What are the key components of the XCS algorithm?
- What are the potentials and limitations of learning classifier systems?

This concludes the introduction of the last category of learning techniques. The ever-increasing amount of data that surrounds us requires data processing and machine learning algorithms to be highly scalable. This is the subject of the next and the final chapter.

12

Scalable Frameworks

The advent of social networking, interactive media, and deep analysis has caused the amount of data processed daily to skyrocket. For data scientists, it's no longer just a matter of finding the most appropriate and accurate algorithm to mine data; it is also about leveraging multi-core CPU architectures and distributed computing frameworks to solve problems in a timely fashion. After all, how valuable is a data mining application if the model does not scale?

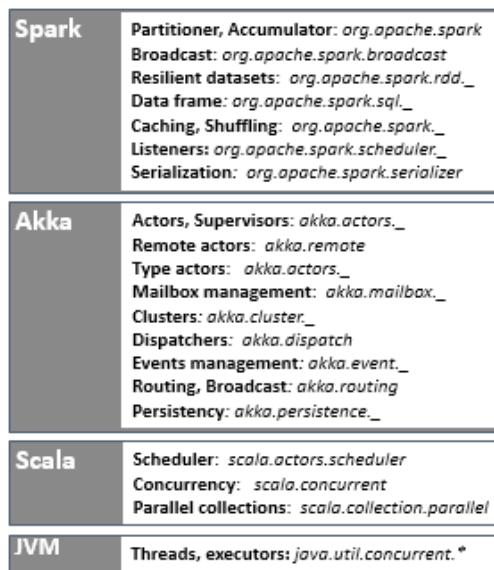
There are many options available to Scala developers to build classification and regression applications for very large datasets. This chapter covers the Scala parallel collections, Actor model, Akka framework, and Apache Spark in-memory clusters. The following topics are covered in this chapter:

- An introduction to Scala parallel collections
- Evaluation of performance of a parallel collection on multi-core CPUs
- The actor model and reactive systems
- Clustered and reliable distributed computing using Akka
- A design of the computational workflow using Akka routers
- An introduction to Apache Spark clustering and its design principles
- Using Spark MLlib for clustering
- Relative performance tuning and evaluation of Spark
- Benefits and limitations of the Apache Spark framework

An overview

The support for distributing and concurrent processing is provided by different stacked frameworks and libraries. Scala concurrent and parallel collections' classes leverage the threading capabilities of the Java virtual machine. **Akka.io** implements a reliable action model originally introduced as part of the Scala standard library. The Akka framework supports remote actors, routing, load balancing protocols, dispatchers, clusters, events, and configurable mailbox management. This framework also provides support for different transport modes, supervisory strategies, and typed actors. Apache Spark's resilient distributed datasets with advanced serialization, caching, and partitioning capabilities leverage Scala and Akka libraries.

The following stack representation illustrates the interdependencies between frameworks:



The Stack representation of scalable frameworks using Scala

Each layer adds a new functionality to the previous one to increase scalability. The Java virtual machine runs as a process within a single host. Scala concurrent classes support effective deployment of an application by leveraging multicore CPU capabilities without the need to write multithreaded applications. Akka extends the Actor paradigm to clusters with advanced messaging and routing options. Finally, Apache Spark leverages Scala higher-order collection methods and the Akka implementation of the Actor model to provide large-scale data processing systems with better performance and reliability, through its resilient distributed datasets and in-memory persistency.

Scala

The Scala standard library offers a rich set of tools, such as parallel collections and concurrent classes to scale number-crunching applications. Although these tools are very effective in processing medium-sized datasets, they are unfortunately quite often discarded by developers in favor of more elaborate frameworks.

Object creation

Although code optimization and memory management is beyond the scope of this chapter, it is worthwhile to remember that a few simple steps can be taken to improve the scalability of an application. One of the most frustrating challenges in using Scala to process large datasets is the creation of a large number of objects and the load on the garbage collector.

A partial list of remedial actions is as follows:

- Limiting unnecessary duplication of objects in an iterated function using a mutable instance
- Using lazy values and **Stream** classes to create objects as needed
- Leveraging efficient collections such as **bloom filters** or **skip lists**
- Running `javap` to decipher the generation of byte code by the JVM

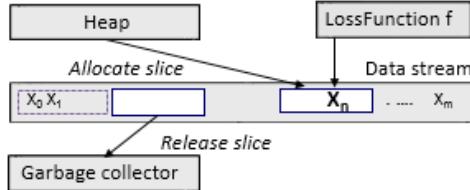
Streams

Some problems require the preprocessing and training of very large datasets, resulting in significant memory consumption by the JVM. Streams are list-like collections in which elements are instantiated or computed lazily. Streams share the same goal of postponing computation and memory allocation as views.

Let's consider the computation of the loss function in machine learning. An observation of the `DataPoint` type is defined as a features vector, `x`, and a labeled or expected value, `y`:

```
case class DataPoint(x: DblVector, y: Double)
```

We can create a loss function, `LossFunction`, that processes a very large dataset on a platform with limited memory. The optimizer responsible for the minimization of the loss or error invokes the loss function at each iteration or recursion, as described in the following diagram:



An illustration of Scala streams allocation and release

The constructor of the `LossFunction` class has the following three arguments (line 2):

- The computation `f` of the loss for each data point
- The weights of the model
- The size of the entire stream `dataSize`

The code is as follows:

```

type StreamLike = WeakReference[Stream[DataPoint]] //1
class LossFunction(
    f: (DblVector, DblVector) => Double,
    weights: DblVector,
    dataSize: Int) { //2

    var nElements = 0
    def compute(stream: () => StreamLike): Double =
        compute(stream().get, 0.0) //3

    def _loss(xs: List[DataPoint]): Double = xs.map(
        dp => dp.y - f(weights, dp.x)).map( sqr(_)).sum //4
}

```

The loss function for the stream is implemented as the `compute` tail recursion (line 3). The recursive method updates the reference of the stream. The type of reference of the stream is `WeakReference` (line 1), so the garbage collection can reclaim the memory associated with the slice for which the loss has been computed. In this example, the loss function is computed as a sum of squared errors (line 4).

The `compute` method manages the allocation and release of slices of stream:

```
@tailrec
def compute(stream: Stream[DataPoint], loss: Double): Double = {
  if( nElements >= dataSize)  loss
  else {
    val step = if(nElements + STEP > dataSize)
      dataSize - nElements else STEP
    nElements += step
    val newLoss = _loss(stream.take(step).toList) //5
    compute( stream.drop(STEP), loss + newLoss ) //6
  }
}
```

The dataset is processed in two steps:

- The driver allocates (that is, `take`) a slice of the stream of observations and then computes the cumulative loss for all the observations in the slice (line 5)
- Once the computation of the loss for the slice is completed, the memory allocated to the weak reference is released (that is, `drop`) (line 6)

An alternative to weak references

There are alternatives to weak references in order for the stream to force the garbage collector to reclaim the memory blocks associated with each slice of observations, which are as follows:



- Define the stream reference as `def`
- Wrap the reference into a method; the reference is then accessible to the garbage collector when the wrapping method returns
- Use a `List` iterator

The average memory allocated during the execution of the loss function for the entire stream is the memory needed to allocate a single slice.

Parallel collections

The Scala standard library includes parallelized collections, whose purpose is to shield developers from the intricacies of concurrent thread execution and race condition. Parallel collections are a very convenient approach to encapsulate concurrency constructs to a higher level of abstraction [12:1].

There are two ways to create parallel collections in Scala, which are as follows:

- Converting an existing collection into a parallel collection of the same semantic using the `par` method; for example, `List[T].par: ParSeq[T]`, `Array[T].par: ParArray[T]`, `Map[K,V].par: ParMap[K,V]`, and so on
- Using the collection classes from the `collection.parallel`, `parallel.immutable`, or `parallel.mutable` packages; for example, `ParArray`, `ParMap`, `ParSeq`, `ParVector`, and so on

Processing a parallel collection

A parallel collection does lend itself to concurrent processing until a pool of threads and a task scheduler are assigned to it. Fortunately, Scala parallel and concurrent packages provide developers with a powerful toolbox to map partitions or segments of collection to tasks running on different CPU cores. The components are as follows:

- `TaskSupport`: This trait inherits the generic `Tasks` trait. It is responsible for scheduling the operation on the parallel collection. There are three concrete implementations of `TaskSupport`.
- `ThreadPoolTaskSupport`: This uses the threads pool in an older version of the JVM.
- `ExecutionContextTaskSupport`: This uses `ExecutorService` that delegates the management of tasks to either a thread pool or the `ForkJoinTasks` pool.
- `ForkJoinTaskSupport`: This uses the fork-join pools of the `java.util.concurrent.ForkJoinPool` type introduced in the Java SDK 1.6. In Java, a **fork-join pool** is an instance of `ExecutorService` that attempts to run not only the current task but also any of its subtasks. It executes the `ForkJoinTask` instances that are lightweight threads.

The following example implements the generation of a random exponential value using a parallel vector and `ForkJoinTaskSupport`:

```
val rand = new ParVector[Float]
Range(0,MAX).foreach(n => rand.updated(n, n*Random.nextFloat()))//1
rand.tasksupport = new ForkJoinTaskSupport(new ForkJoinPool(16))
val randExp = vec.map( Math.exp(_) ) //2
```

The `rand` parallel vector of random probabilities is created and initialized by the main task (line 1), but the conversion to a vector of a `randExp` exponential value is executed by a pool of 16 concurrent tasks (line 2).

Preserving the order of elements

 Operations that traverse a parallel collection using an iterator preserve the original order of the element of the collection. Iterator-less methods such as `foreach` or `map` do not guarantee that the order of the elements that are processed will be preserved.

The benchmark framework

The main purpose of parallel collections is to improve the performance of execution through concurrency. The first step is to either select an existing benchmark or create our own benchmark.

Scala library benchmark

 The Scala standard library has a `testing.Benchmark` trait used to test using the command line [12:2]. All you need to do is insert your function or code in the `run` method:

```
object test with Benchmark { def run { /* ... */ } }
```

Let's create a `ParBenchmark` parameterized class to evaluate the performance of operations on parallel collections:

```
abstract class ParBenchmark[U] (times: Int) {
  def map(f: U => U) (nTasks: Int): Double //1
  def filter(f: U => Boolean) (nTasks: Int): Double //2
  def timing(g: Int => Unit ): Long
}
```

The user has to supply the data transformation `f` for the `map` (line 1) and `filter` (line 2) operations of parallel collections as well as the number of concurrent tasks `nTasks`. The `timing` method collects the duration of the `times` execution of a given operation `g` on a parallel collection:

```
def timing(g: Int => Unit ): Long = {
  var startTime = System.currentTimeMillis
  Range(0, times).foreach(g)
  System.currentTimeMillis - startTime
}
```

Let's define the mapping and reducing operation for the parallel arrays for which the benchmark is defined as follows:

```
class ParArrayBenchmark[U] (u: Array[U], //3
                           v: ParArray[U], //4
                           times:Int) extends ParBenchmark[T] (times)
```

The first argument of the benchmark constructor is the default array of the Scala standard library (line 3). The second argument is the parallel data structure (or class) associated with the array (line 4).

Let's compare the parallelized and default array on the `map` and `reduce` methods of `ParArrayBenchmark` as follows:

```
def map(f: U => U) (nTasks: Int): Unit = {
    val pool = new ForkJoinPool(nTasks)
    v.tasksupport = new ForkJoinTaskSupport(pool)
    val duration = timing(_ => u.map(f)).toDouble //5
    val ratio = timing(_ => v.map(f))/duration //6
    show(s"$numTasks, $ratio")
}
```

The user has to define the mapping function `f` and the number of concurrent tasks `nTasks` available to execute a `map` transformation on the array `u` (line 5) and its parallelized counterpart `v` (line 6). The `reduce` method follows the same design, as shown in the following code:

```
def reduce(f: (U,U) => U) (nTasks: Int): Unit = {
    val pool = new ForkJoinPool(nTasks)
    v.tasksupport = new ForkJoinTaskSupport(pool)
    val duration = timing(_ => u.reduceLeft(f)).toDouble //7
    val ratio = timing(_ => v.reduceLeft(f))/duration //8
    show(s"$numTasks, $ratio")
}
```

The user-defined function `f` is used to execute the `reduce` action on the array `u` (line 7) and its parallelized counterpart `v` (line 8).

The same template can be used for other higher Scala methods, such as `filter`.

The absolute timing of each operation is completely dependent on the environment. It is far more useful to record the ratio of the duration of execution of the operation on the parallelized array, over the single thread array.

The benchmark class `ParMapBenchmark` used to evaluate `ParHashMap` is similar to the benchmark for `ParArray`, as shown in the following code:

```
class ParMapBenchmark[U] (val u: Map[Int, U],
    val v: ParMap[Int, U],
    times: Int) extends ParBenchmark[T] (times)
```

For example, the `filter` method of `ParMapBenchmark` evaluates the performance of the parallel map `v` relative to a single-threaded map `u`. It applies the filtering condition to the values of each map, as follows:

```
def filter(f: U => Boolean) (nTasks: Int): Unit = {
    val pool = new ForkJoinPool(nTasks)
    v.tasksupport = new ForkJoinTaskSupport(pool)
    val duration = timing(_ => u.filter(e => f(e._2))).toDouble
    val ratio = timing(_ => v.filter(e => f(e._2)))/duration
    show(s"$nTasks, $ratio")
}
```

Performance evaluation

The first performance test consists of creating a single-threaded and a parallel array of random values and executing the `map` and `reduce` evaluation methods, on using an increasing number of tasks, as follows:

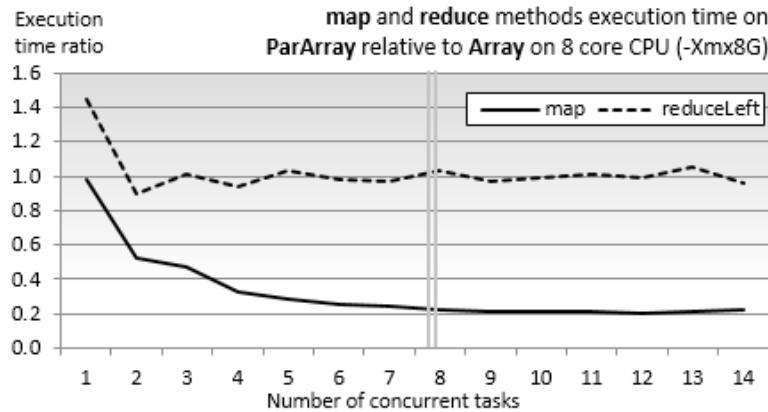
```
val sz = 1000000; val NTASKS = 16
val data = Array.fill(sz) (Random.nextDouble)
val pData = ParArray.fill(sz) (Random.nextDouble)
val times: Int = 50

val bench = new ParArrayBenchmark[Double] (data, pData, times)
val mapper = (x: Double) => Math.sin(x*0.01) + Math.exp(-x)
Range(1, NTASKS).foreach(bench.map(mapper) (_))
val reducer = (x: Double, y: Double) => x+y
Range(1, NTASKS).foreach(bench.reduce(reducer) (_))
```

Measuring performance

The code has to be executed within a loop and the duration has to be averaged over a large number of executions to avoid transient actions such as initialization of the JVM process or collection of unused memory (GC).

The following graph shows the output of the performance test:



The impact of concurrent tasks on the performance on Scala parallelized map and reduce

The test executes the mapper and reducer functions 1 million times on an 8-core CPU with 8 GB of available memory on the JVM.

The results are not surprising in the following respects:

- The reducer doesn't take advantage of the parallelism of the array. The reduction of `ParArray` has a small overhead in the single-task scenario and then matches the performance of `Array`.
- The performance of the `map` function benefits from the parallelization of the array. The performance levels off when the number of tasks allocated equals or exceeds the number of CPU core.

The second test consists of comparing the behavior of the `ParArray` and `ParHashMap` parallel collections, on the `map` and `filter` methods, using a configuration identical to the first test as follows:

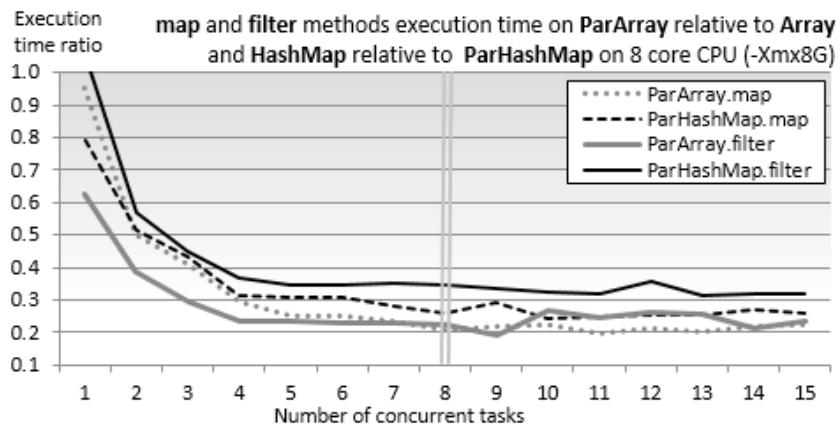
```

val sz = 10000000
val mData = new HashMap[Int, Double]
Range(0, sz).foreach( mData.put(_, Random.nextDouble()) ) //9
val mParData = new ParHashMap[Int, Double]
Range(0, sz).foreach( mParData.put(_, Random.nextDouble()) )

val bench = new ParMapBenchmark[Double](mData, mParData, times)
Range(1, NTASKS).foreach(bench.map(mapper)(_)) //10
val filterer = (x: Double) => (x > 0.8)
Range(1, NTASKS).foreach( bench.filter(filterer)(_) ) //11

```

The test initializes a `HashMap` instance and its `ParHashMap` parallel counter with 1 million random values (line 9). The benchmark `bench` processes all the elements of these hash maps with the `mapper` instance introduced in the first test (line 10) and a filtering function `filterer` (line 11) with `NTASKS` equal to 6. The output is shown in the following diagram:



The impact of concurrent tasks on the performance on Scala parallelized array and hash map

The impact of the parallelization of collections is very similar across methods and collections. It's important to notice that the performance of the parallel collections levels off at around four times the single thread collections for five concurrent tasks and above. **Core parking** is partially responsible for this behavior. Core parking disables a few CPU cores in an effort to conserve power, and in the case of a single application, it consumes almost all CPU cycles.

Further performance evaluation

The purpose of the performance test was to highlight the benefits of using Scala parallel collections. You should experiment further with collections other than `ParArray` and `ParHashMap` and other higher-order methods to confirm the pattern.

Clearly, a four times increase in performance is nothing to complain about. Having said that, parallel collections are limited to single-host deployments. If you cannot live with such a restriction and still need a scalable solution, the Actor model provides a blueprint for highly distributed applications.

Scalability with Actors

Traditional multithreaded applications rely on accessing data located in shared memory. The mechanism relies on synchronization monitors such as locks, mutexes, or semaphores to avoid deadlocks and inconsistent mutable states. Even for the most experienced software engineer, debugging multithreaded applications is not a simple endeavor.

The second problem with shared memory threads in Java is the high computation overhead caused by continuous context switches. Context switching consists of saving the current stack frame delimited by the base and stack pointers into the heap memory and loading another stack frame.

These restrictions and complexities can be avoided using a concurrency model that relies on the following key principles:

- Immutable data structures
- Asynchronous communication

The Actor model

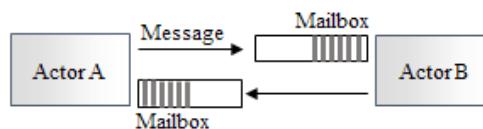
The Actor model, originally introduced in the **Erlang** programming language, addresses these issues [12:3]. The purpose of using the Actor model is twofold as follows:

- It distributes the computation over as many cores and servers as possible
- It reduces or eliminates race conditions and deadlocks, which are very prevalent in the Java development

The model consists of the following components:

- Independent processing units known as Actors. They communicate by exchanging messages asynchronously instead of sharing states.
- Immutable messages are sent to queues, known as mailboxes, before being processed by each actor one at a time.

Let's take a look at the following diagram:



The representation of messaging between actors

There are two message-passing mechanisms, which are as follows:

- **Fire-and-forget or tell:** This sends the immutable message asynchronously to the target or receiving Actor and immediately returns without blocking. The syntax is `targetActorRef ! message`.
- **Send-and-receive or ask:** This sends a message asynchronously, but returns a `Future` instance that defines the expected reply from the `val future = targetActorRef ? message target actor`.

The generic construct for the Actor message handler is somewhat similar to the `Runnable.run()` method in Java, as shown in the following code:

```
while( true ){
    receive { case msg1: MsgType => handler }
}
```

The `receive` keyword is, in fact, a partial function of the `PartialFunction[Any, Unit]` type [12:4]. The purpose is to avoid forcing developers to handle all possible message types. The Actor consuming messages may very well run on a separate component or even application, from the Actor producing these messages. It is not always easy to anticipate the type of messages an Actor has to process in a future version of an application.

A message whose type is not matched is merely ignored. There is no need to throw an exception from within the Actor's routine. Implementations of the Actor model strive to avoid the overhead of context switching and creation of threads [12:5].

I/O blocking operations

 Although it is highly recommended that you do not use Actors to block operations, such as I/O, there are circumstances that require the sender to wait for a response. You need to be keep in mind that blocking the underlying threads might starve other Actors from CPU cycles. It is recommended that you either configure the runtime system to use a large thread pool or allow the thread pool to be resized by setting the `actors.enableForkJoin` property as `false`.

Partitioning

A dataset is defined as a Scala collection, for example, `List`, `Map`, and so on. Concurrent processing requires the following steps:

1. Breaking down a dataset into multiple subdatasets.
2. Processing each dataset independently and concurrently.
3. Aggregating all the resulting datasets.

These steps are defined through a monad associated with a collection in the *Abstraction* section under *Why Scala?* in *Chapter 1, Getting Started*.

1. The `apply` method creates the sub-collection or partitions for the first step, for example, `def apply[T] (a: T): List[T]`.
2. A map-like operation defines the second stage. The last step relies on the monoidal associativity of the Scala collection, for example, `def ++ (a: List[T] . b: List[T]): List[T] = a ++ b`.
3. The aggregation, such as `reduce`, `fold`, `sum`, and so on, consists of flattening all the subresults into a single output, for example, `val xs: List(...) = List(List(...), List(...)).flatten`.

The methods that can be parallelized are `map`, `flatMap`, `filter`, `find`, and `filterNot`. The methods that cannot be completely parallelized are `reduce`, `fold`, `sum`, `combine`, `aggregate`, `groupBy`, and `sortWith`.

Beyond actors – reactive programming

The Actor model is an example of the reactive programming paradigm. The concept is that functions and methods are executed in response to events or exceptions. Reactive programming combines concurrency with event-based systems [12:6].

Advanced functional reactive programming constructs rely on composable futures and **continuation-passing style (CPS)**. An example of a Scala reactive library can be found at <https://github.com/ingoem/scala-react>.

Akka

The Akka framework extends the original Actor model in Scala by adding extraction capabilities such as support for typed Actor, message dispatching, routing, load balancing, and partitioning, as well as supervision and configurability [12:7].

The Akka framework can be downloaded from the <http://akka.io/> website or through the Typesafe Activator at <http://www.typesafe.com/platform>.

Akka simplifies the implementation of the Actor model by encapsulating some of the details of Scala Actor in the `akka.actor.Actor` and `akka.actor.ActorSystem` classes.

The three methods you want to override are as follows:

- `prestart`: This is an optional method that is invoked to initialize all the necessary resources such as file or database connection before the Actor is executed
- `receive`: This method defines the Actor's behavior and returns a partial function of the `PartialFunction[Any, Unit]` type
- `postStop`: This is an optional method to clean up resources such as releasing memory, closing database connections, and socket or file handles

Typed and untyped actors

Untyped actors can process messages of any type. If the type of the message is not matched by the receiving actor, it is discarded. Untyped actors can be regarded as contract-less actors. They are the default actors in Scala.

Typed actors are similar to Java remote interfaces. They respond to a method invocation. The invocation is declared publicly, but the execution is delegated asynchronously to the private instance of the target actor [12:8].

Akka offers a variety of functionalities to deploy concurrent applications. Let's create a generic template for a master Actor and worker Actors to transform a dataset using any preprocessing or classification algorithm inherited from an explicit or implicit monadic data transformation, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*

The master Actor manages the worker actors in one of the following ways:

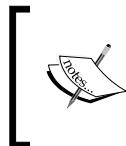
- Individual actors
- Clusters through a **router** or a **dispatcher**

The router is a very simple example of Actor supervision. Supervision strategies in Akka are an essential component to make the application fault-tolerant [12:9]. A supervisor Actor manages the operations, availability, and life cycle of its children, known as **subordinates**. The supervision among actors is organized as a hierarchy. Supervision strategies are categorized as follows:

- **One-for-one strategy:** This is the default strategy. In case of a failure of one of the subordinates, the supervisor executes a recovery, restart, or resume action for that subordinate only.
- **All-for-one strategy:** The supervisor executes a recovery or remedial action on all its subordinates in case one of the Actors fails.

Master-workers

The first model to evaluate is the traditional **master-slaves** or **master-workers** design for the computation workflow. In this design, the worker Actors are initialized and managed by the master Actor, which is responsible for controlling the iterative process, state, and termination condition of the algorithm. The orchestration of the distributed tasks is performed through message passing.



The design principle

It is highly recommended that you segregate the implementation of the computation or domain-specific logic from the actual implementation of the worker and master actors.

Exchange of messages

The first step in implementing the master-worker design is to define the different classes of messages exchanged between the master and each worker in order to control the execution of the iterative procedure. The implementation of the master-worker design is as follows:

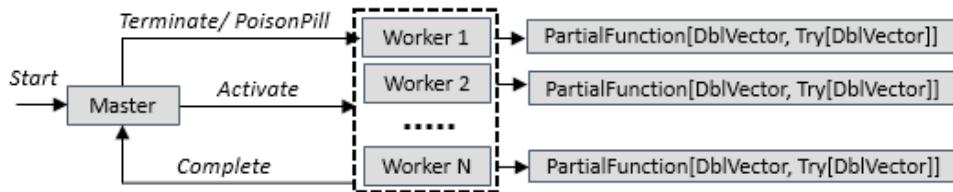
```
sealed abstract class Message(val i: Int)
case class Terminate(i: Int) extends Message(i)
case class Start(i: Int = 0) extends Message(i) //1
case class Activate(i: Int, x: DblVector) extends Message(i) //2
case class Completed(i: Int, x: DblVector) extends Message(i)//3
```

Let's define the messages that control the execution of the algorithm. We need at least the following message types or case classes:

- **Start:** This is sent by the client code to the master to start the computation (line 1).
- **Activate:** This is sent by the master to the workers to activate the computation. This message contains the time series x to be processed by the worker Actors. It also contains the reference to `sender` (master actor). (line 2).
- **Completed:** This is sent by each worker back to `sender`. It contains the variance of the data in the group (line 3).

The master stops a worker using a `PoisonPill` message. The different approaches to terminate an actor are described in the *The master actor* section.

The hierarchy of the `Message` class is sealed to prevent third-party developers from adding another message type. The worker responds to the activate message by executing a data transformation of the `ITransform` type. The messages exchanged between master and worker actors are shown in the following diagram:



A sketch design of the master-slave communication in an actor framework

Messages as case classes

The actor retrieves the messages queued in its mailbox by managing each message instance (copying, matching, and so on). Therefore, the message type has to be defined as a case class. Otherwise, the developer will have to override the `equals` and `hashCode` methods.



Worker actors

The worker actors are responsible for transforming each partitioned datasets created by the master Actor, as follows:

```
type PfnTransform = PartialFunction[DblVector, Try[DblVector]]\n\nclass Worker(id: Int,\n    fct: PfnTransform) extends Actor { //1\n    override def receive = {\n        case msg: Activate => //2\n            sender ! Completed(msg.id+id, fct(msg.xt).get)\n    }\n}
```

The Worker class constructor takes the `fct` (the partial function as an argument) (line 1). The worker launches the processing or transformation of the `msg.xt` data on arrival of the `Activate` message (line 2). It returns the `Completed` message to the master once the `fct` data transformation is completed.

The workflow controller

In the *Scalability* section in *Chapter 1, Getting Started*, we introduced the concepts of workflow and controller to manage the training and classification process as a sequence of transformation on a time series. Let's define an abstract class for all controller actors, `Controller`, with the following three key parameters:

- A time series `xt` to be processed
- A `fct` data transformation implemented as a partial function
- The number of partitions `nPartitions` to break down a time series for concurrent processing

The `Controller` class can be defined as follows:

```
abstract class Controller (\n    val xt: DblVector,\n    val fct: PfnTransform,\n    val nPartitions: Int) extends Actor with Monitor { //3\n\n    def partition: Iterator[DblVector] = { //4\n        val sz = (xt.size.toDouble/nPartitions).ceil.toInt\n        xt.grouped(sz)\n    }\n}
```

The controller is responsible for splitting the time series into several partitions and assigning each partition to a dedicated worker (line 4).

The master actor

Let's define a master actor class `Master`. The three methods to override are as follows:

- `prestart`: This is a method invoked to initialize all the necessary resources such as a file or database connection before the actor executes (line 9)
- `receive`: This is a partial function that dequeues and processes the messages from the mail box
- `postStop`: This cleans up resources such as releasing memory and closing database connections, sockets, or file handles (line 10)

The `Master` class can be defined as follows:

```
abstract class Master( //5
    xt: DblVector,
    fct: PfnTransform,
    nPartitions: Int) extends Controller(xt, fct, nPartitions) {

    val aggregator = new Aggregator(nPartitions) //6
    val workers = List.tabulate(nPartitions)(n =>
        context.actorOf(Props(new Worker(n, fct)),
            name = s"worker_$n")) //7
    workers.foreach( context.watch( _ ) ) //8

    override def preStart: Unit = /* ... */ //9
    override def postStop: Unit = /* ... */ //10
    override def receive
}
```

The `Master` class has the following parameters (line 5):

- `xt`: This is the time series to transform
- `fct`: This is the transformation function
- `nPartitions`: This is the number of partitions

An aggregating class `aggregator` collects and reduces the results from each worker (line 6):

```
class Aggregator(partitions: Int) {
    val state = new ListBuffer[DblVector]

    def += (x: DblVector): Boolean = {
        state.append(x)
        state.size == partitions
    }

    def clear: Unit = state.clear
    def completed: Boolean = state.size == partitions
}
```

The worker actors are created through the `actorOf` factory method of the `ActorSystem` context (line 7). The worker actors are attached to the context of the master actor, so it can be notified when the workers terminate (line 8).

The receive message handler processes only two types of messages: `Start` from the client code and `Completed` from the workers, as shown in the following code:

```
override def receive = {
    case s: Start => start //11

    case msg: Completed => //12
        if( aggregator += msg.xt) //13
            workers.foreach( context.stop(_) ) //14

    case Terminated(sender) => //15
        if( aggregator.completed ) {
            context.stop(self) //16
            context.system.shutdown
        }
}
```

The `Start` message triggers the partitioning of the input time series into partitions (line 11):

```
def start: Unit = workers.zip(partition.toVector)
    .foreach {case (w, s) => w ! Activate(0,s)} //16
```

The partitions are then dispatched to each worker with the `Activate` message (line 16).

Each worker sends a `Completed` message back to master on the completion of their task (line 12). The master aggregates the results from each worker (line 13). Once all the workers have completed their task, they are removed from the master's context (line 14). The master terminates all the workers through a `Terminated` message (line 15), and finally, terminates itself through a request to its `context` to stop it (line 16).

The previous code snippet uses two different approaches to terminate an actor. There are four different methods of shutting down an actor, as mentioned here:

- `actorSystem.shutdown`: This method is used by the client to shut down the parent actor system
- `actor ! PoisonPill`: This method is used by the client to send a poison pill message to the actor
- `context.stop(self)`: This method is used by the Actor to shut itself down within its context
- `context.stop(childActorRef)`: This method is used by the Actor to shut itself down through its reference

Master with routing

The previous design makes sense only if each worker has a unique characteristic that requires direct communication with the master. This is not the case in most applications. The communication and internal management of the worker can be delegated to a router. The implementation of the master routing capabilities is very similar to the previous design, as shown in the following code:

```
class MasterWithRouter(
    xt: DblVector,
    fct: PfnTransform,
    nPartitions: Int) extends Controller(xt, fct, nPartitions) {

    val aggregator = new Aggregator(nPartitions)
    val router = { //17
        val routerConfig = RoundRobinRouter(nPartitions, //18
            supervisorStrategy = this.supervisorStrategy)
        context.actorOf(
            Props(new Worker(0, fct)).withRouter(routerConfig) )
    }
    context.watch(router)

    override def receive
}
```

The only difference is that the `context.actorOf` factory creates an extra actor, router, along with the workers (line 17). This particular implementation relies on round-robin assignment of the message by the router to each worker (line 18). Akka supports several routing mechanisms that select a random actor, or the actor with the smallest mailbox, or the first to respond to a broadcast, and so on.

Router supervision



The router actor is a parent of the worker actors. It is by design a supervisor of the worker actors, which are its children actors. Therefore, the router is responsible for the life cycle of the worker actors, which includes their creation, restarting, and termination.

The implementation of the `receive` message handler is almost identical to the message handler in the master without routing capabilities, with the exception of the termination of the workers through the router (line 19):

```
override def receive = {
  case Start => start
  case msg: Completed =>
    if( aggregator += msg.xt) context.stop(router) //19
  ...
}
```

The `start` message handler has to be modified to broadcast the `Activate` message to all the workers through the router:

```
def start: Unit =
  partition.toVector.foreach {router ! Activate(0, _)}
```

Distributed discrete Fourier transform

Let's select the **discrete Fourier transform** (DFT) on a time series `xt` as our data transformation. We discussed this in the *Discrete Fourier transform* section in *Chapter 3, Data Preprocessing*. The testing code is exactly the same, whether the master has routing capabilities or not.

First, let's define a master controller `DFTMaster` dedicated to the execution of the distributed discrete Fourier transform, as follows:

```
type Reducer = List[DblVector] -> immutable.Seq[Double]
class DFTMaster(
  xt: DblVector,
  nPartitions: Int,
  reducer: Reducer) //20
  extends Master(xt, DFT[Double].|>, nPartitions)
```

The reducer method aggregates or reduces the results of the discrete Fourier transform (frequencies distribution) from each worker (line 20). In the case of the discrete Fourier transform, the fReduce reducer method transposes the list of frequencies distribution and then sums up the amplitude for each frequency (line 21):

```
def fReduce(buf: List[DblVector]): immutable.Seq[Double] =
  buf.transpose.map(_.sum).toSeq //21
```

Let's take a look at the test code:

```
val NUM_WORKERS = 4
val NUM_DATAPOINTS = 1000000
val h = (x: Double) => 2.0 * Math.cos(Math.PI * 0.005 * x) +
  Math.cos(Math.PI * 0.05 * x) + 0.5 * Math.cos(Math.PI * 0.2 * x) +
  0.3 * Random.nextDouble //22

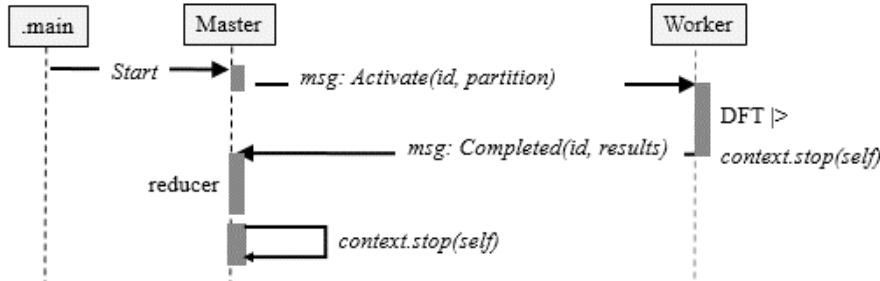
val actorSystem = ActorSystem("System") //23
val xt = Vector.tabulate(NUM_DATA_POINTS)(h(_))
val controller = actorSystem.actorOf(
  Props(new DFTMasterWithRouter(xt, NUM_WORKERS,
    fReduce)), "MasterWithRouter") //24
controller ! Start(1) //25
```

The input time series is synthetically generated by the noisy sinusoidal function h (line 22). The function h has three distinct harmonics: 0.005, 0.05, and 0.2, so the results of the transformation can be easily validated. The Actor system, `ActorSystem`, is instantiated (line 23) and the master Actor is generated through the Akka `ActorSystem.actorOf` factory (line 24). The main program sends a `start` message to the master to trigger the distributed computation of the discrete Fourier transform (line 25).

The action instantiation

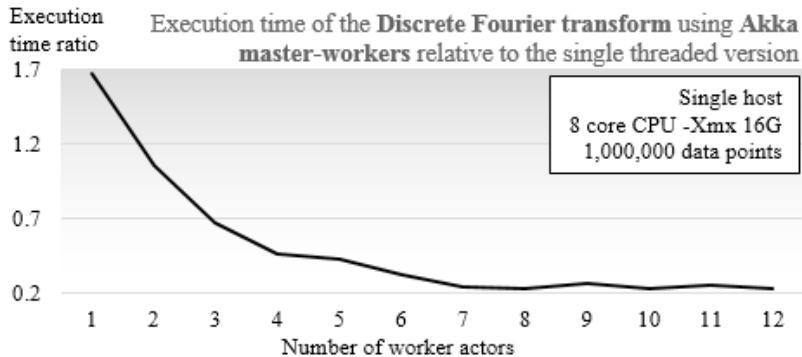
Although the `scala.actor.Actor` class can be instantiated using the constructor, `akka.actor.Actor` is instantiated using an `ActorSystem` context, an `actorOf` factory, and a `Props` configuration object. This second approach has several benefits, including decoupling the deployment of the actor from its functionality and enforcing a default supervisor or parent for the Actor; in this case, `ActorSystem`.

The following sequential diagram illustrates the message exchange between the main program, master, and worker Actors:



A sequential diagram for the normalization of cross-validation groups

The purpose of the test is to evaluate the performance of the computation of the discrete Fourier transform using the Akka framework relative to the original implementation, without actors. As with Scala parallel collections, the absolute timing for the transformation depends on the host and the configuration, as shown in the following graph:



The impact of the number of worker (slave) actors on the performance of the discrete Fourier transform

The single-threaded version of the discrete Fourier transform is significantly faster than the implementation using the Akka master-worker model with a single worker actor. The cost of partitioning and aggregating (or reducing) the results adds a significant overhead to the execution of the Fourier transform. However, the master worker model is far more efficient with three or more worker actors.

Limitations

The master-worker implementation has a few problems, which are as follows:

- In the message handler of the master Actor, there is no guarantee that the poison pill will be consumed by all the workers before the master stops.
- The main program has to sleep for a period of time long enough to allow the master and workers to complete their tasks. There is no guarantee that the computation will be completed when the main program awakes.
- There is no mechanism to handle failure in delivering or processing messages.

The culprit is the exclusive use of the fire-and-forget mechanism to exchange data between master and workers. The send-and-receive protocol and futures are remedies to these problems.

Futures

A future is an object, more specifically a monad, used to retrieve the results of concurrent operations, in a nonblocking fashion. The concept is very similar to a callback supplied to a worker, which invokes it when the task is completed. Futures hold a value that might or might not become available in the future when a task is completed, whether successful or not [12:10].

There are two options to retrieve results from futures:

- Blocking the execution using `scala.concurrent.Await`
- The `onComplete`, `onSuccess`, and `onFailure` callback functions



Which future?

A Scala environment provides developers with two different Future classes: `scala.actor.Future` and `scala.concurrent.Future`.

The `actor.Future` class is used to write continuation-passing style workflows in which the current actor is blocked until the value of the future is available. Instances of the `scala.concurrent.Future` type used in this chapter are the equivalent of `java.concurrent.Future` in Scala.

The Actor life cycle

Let's reimplement the normalization of cross-validation groups by their variance, which we introduced in the previous section, using futures to support concurrency. The first step is to import the appropriate classes for execution of the main actor and futures, as follows:

```
import akka.actor.{Actor, ActorSystem, ActorRef, Props} //26
import akka.util.Timeout    //27
import scala.concurrent.{Await, Future}   //28
```

The Actor classes are provided by the `akka.actor` package, instead of the `scala.actor._` package because of Akka's extended actor model (line 26). The future-related classes, `Future` and `Await`, are imported from the `scala.concurrent` package, which is similar to the `java.concurrent` package (line 28). The `akka.util.Timeout` class is used to specify the maximum duration the actor has to wait for the completion of the futures (line 27).

There are two options for a parent actor or the main program to manage the futures it creates, which are as follows:

- **Blocking:** The parent actor or main program stops the execution until all futures have completed their tasks.
- **Callback:** The parent actor or the main program initiates the futures during the execution. The future tasks are performed concurrently with the parent actor, and it is then notified when each future task is completed.

Blocking on futures

The following design consists of blocking the actor that launches the futures until all the futures have been completed, either returning with a result or throwing an exception. Let's modify the master actor into a `TransformFutures` class that manages futures instead of workers or routing actors, as follows:

```
abstract class TransformFutures(
  xt: DblVector,
  fct: PfnTransform,
  nPartitions: Int)
  (implicit timeout: Timeout) //29
  extends Controller(xt, fct, nPartitions) {
  override def receive = {
    case s: Start => compute(transform) //30
  }
}
```

The `TransformFutures` class requires the same parameters as the `Master` actor: a time series, `xt`, a data transformation, `fct`, and the number of partitions, `nPartitions`. The `timeout` parameter is an implicit argument of the `Await.result` method, and therefore, needs to be declared as an argument (line 29). The only message, `Start`, triggers the computation of the data transformation of each future, and then the aggregation of the results (line 30). The `transform` and `compute` methods have the same semantics as those in the master-workers design.

The generic message handler

You may have read or even written examples of actors that have generic `case _ =>` handlers in the message loop for debugging purposes. The message loop takes a partial function as an argument. Therefore, no error or exception is thrown if the message type is not recognized. There is no need for such a handler apart from the one for debugging purposes. Message types should inherit from a sealed abstract class or a sealed trait in order to prevent a new message type from being added by mistake.

Let's take a look at the `transform` method. Its main purpose is to instantiate, launch, and return an array of futures responsible for the transformation of the partitions, as shown in the following code:

```
def transform: Array[Future[DblVector]] = {
    val futures = new Array[Future[DblVector]](nPartitions) //31

    partition.zipWithIndex.foreach { case (x, n) => { //32
        futures(n) = Future[DblVector] { fct(x).get } //33
    }}
    futures
}
```

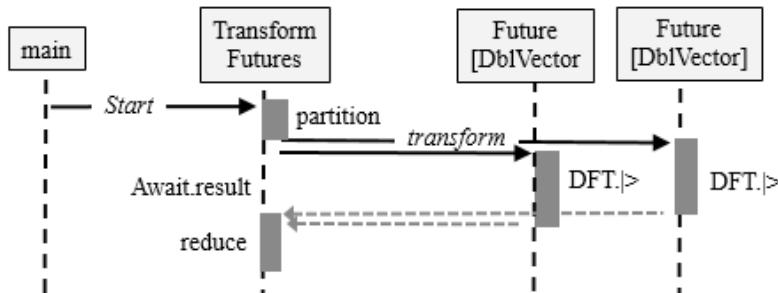
An array of futures (one future per partition) is created (line 31). The `transform` method invokes the partitioning method `partition` (line 32) and then initializes the future with the `fct` partial function (line 33):

```
def compute(futures: Array[Future[DblVector]]): Seq[Double] =
    reduce(futures.map(Await.result(_, timeout.duration))) //34
```

The compute method invokes a user-defined reduce function on the futures. The execution of the Actor is blocked until the `Await` class' `scala.concurrent.Await.result` method (line 34) returns the result of each future computation. In the case of the discrete Fourier transform, the list of frequencies is transposed before the amplitude of each frequency is summed (line 35), as follows:

```
def reduce(data: Array[DblVector]): Seq[Double] =
    data.view.map(_.toArray)
        .transpose.map(_.sum) //35
        .take(SPECTRUM_WIDTH).toSeq
```

The following sequential diagram illustrates the blocking design and the activities performed by the Actor and the futures:



The sequential diagram for actor blocking on future results

Handling future callbacks

Callbacks are an excellent alternative to having the actor blocks on futures, as they can simultaneously execute other functions concurrently with the future execution.

There are two simple ways to implement the callback function, as follows:

- `Future.onComplete`
- `Future.onSuccess` and `Future.onFailure`

The `onComplete` callback function takes a function of the `Try[T] => U` type as an argument with an implicit reference to the execution context, as shown in the following code:

```
val f: Future[T] = future { execute task } f onComplete {
    case Success(s) => { ... }
    case Failure(e) => { ... }
}
```

You can surely recognize the {Try, Success, Failure} monad.

An alternative implementation is to invoke the `onSuccess` and `onFailure` methods that use partial functions as arguments to implement the callbacks, as follows:

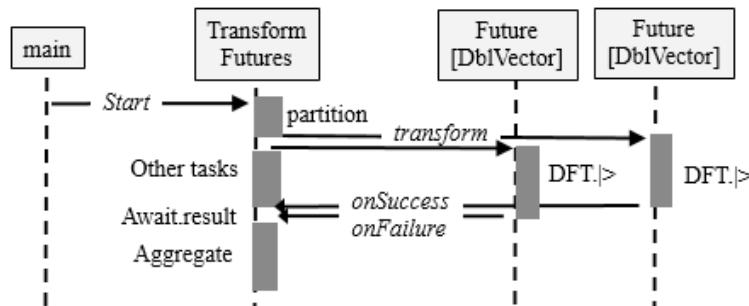
```
f onFailure { case e: Exception => { ... } }
f onSuccess { case t => { ... } }
```

The only difference between blocking one future data transformation and handling callbacks is the implementation of the `compute` method or reducer. The class definition, message handler, and initialization of futures are identical, as shown in the following code:

```
def compute(futures: Array[Future[DblVector]]): Seq[Double] = {
    val buffer = new ArrayBuffer[DblVector]

    futures.foreach( f => {
        f onSuccess { //36
            case data: DblVector => buffer.append(data)
        }
        f onFailure { case e: Exception => /* .. */ } //37
    })
    buffer.find( _.isEmpty).map( _ => reduce(buffer)) //38
}
```

Each future calls the master actor back with either the result of the data transformation, the `onSuccess` message (line 36), or an exception, the `OnFailure` message (line 37). If every future succeeds, the values of all frequencies for all the partitions are summed (line 38). The following sequential diagram illustrates the handling of the callback in the master actor:



A sequential diagram for actor handling future result with callbacks

The execution context

The application of futures requires that the execution context is implicitly provided by the developer. There are three different ways to define the execution context:



- Import the context: `import ExecutionContext.Implicits.global`
- Create an instance of the context within the actor (or actor context): `implicit val ec = ExecutionContext.fromExecutorService(...)`
- Define the context when instantiating the future: `val f = Future[T] = { } (ec)`

Putting it all together

Let's reuse the discrete Fourier transform. The client code uses the same synthetically created time series as in the master-worker test model. The first step is to create a transform future for the discrete Fourier transform, `DFTTransformFuture`, as follows:

```
class DFTTransformFutures(
    xt: DblVector,
    partitions: Int) (implicit timeout: Timeout)
    extends TransformFutures(xt, DFT[Double].|>, partitions) {

    override def reduce(data: Array[DblVector]): Seq[Double] =
        data.map(_.toArray).transpose
            .map(_.sum).take(SPECTRUM_WIDTH).toSeq
}
```

The only purpose of the `DFTTransformFuture` class is to define the `reduce` aggregation method for the discrete Fourier transform. Let's reuse the same test case as in the *Distributed discrete Fourier transform* section under *Master-workers*:

```
import akka.pattern.ask

val duration = Duration(8000, "millis")
implicit val timeout = new Timeout(duration)
val master = actorSystem.actorOf( //39
    Props(new DFTTransformFutures(xt, NUM_WORKERS)),
    "DFTTransform")
val future = master ? Start(0) //40
Await.result(future, timeout.duration) //41
actorSystem.shutdown //42
```

The master actor is initialized as of the `TransformFutures` type with the input time series `xt`, the discrete Fourier transform `DFT`, and the number of workers or partitions `nPartitions` as arguments (line 39). The program creates a future instance by sending (ask) the `Start` message to the master (line 40). The program blocks until the completion of the future (line 41), and then shuts down the Akka actor system (line 42).

Apache Spark

Apache Spark is a fast and general-purpose cluster computing system, initially developed as AMPLab/UC Berkeley as part of the **Berkeley Data Analytics Stack (BDAS)** (http://en.wikipedia.org/wiki/UC_Berkeley). It provides high-level APIs for the following programming languages that make large and concurrent parallel jobs easy to write and deploy [12:11]:

- **Scala:** <http://spark.apache.org/docs/latest/api/scala/index.html>
- **Java:** <http://spark.apache.org/docs/latest/api/java/index.html>
- **Python:** <http://spark.apache.org/docs/latest/api/python/index.html>



The link to the latest information

The URLs as any reference to Apache Spark may change in future versions.



The core element of Spark is a **resilient distributed dataset (RDD)**, which is a collection of elements partitioned across the nodes of a cluster and/or CPU cores of servers. An RDD can be created from a local data structure such as a list, array, or hash table, from the local filesystem or the **Hadoop distributed file system (HDFS)**.

The operations on an RDD in Spark are very similar to the Scala higher-order methods. These operations are performed concurrently over each partition. Operations on RDDs can be classified as follows:

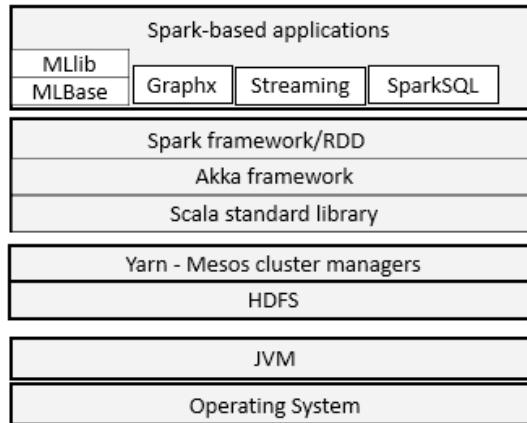
- **Transformation:** This operation converts, manipulates, and filters the elements of an RDD on each partition
- **Action:** This operation aggregates, collects, or reduces the elements of the RDD from all partitions

An RDD can be persisted, serialized, and cached for future computation.

Spark is written in Scala and built on top of Akka libraries. Spark relies on the following mechanisms to distribute and partition RDDs:

- Hadoop/HDFS for the distributed and replicated filesystem
- Mesos or Yarn for the management of a cluster and shared pool of data nodes

The Spark ecosystem can be represented as stacks of technology and framework, as seen in the following diagram:



The Apache Spark framework ecosystem

The Spark ecosystem has grown to support some machine learning algorithms out of the box, such as **MLlib**, a SQL-like interface to manipulate datasets with relational operators, **SparkSQL**, a library for distributed graphs, **GraphX**, and a streaming library [12:12].

Why Spark?

The authors of Spark attempt to address the limitations of Hadoop in terms of performance and real-time processing by implementing in-memory iterative computing, which is critical to most discriminative machine learning algorithms. Numerous benchmark tests have been performed and published to evaluate the performance improvement of Spark relative to Hadoop. In the case of iterative algorithms, the time per iteration can be reduced by a ratio of 1:10 or more.

Spark provides a large array of prebuilt transforms and actions that go well beyond the basic map-reduce paradigm. These methods on RDDs are a natural extension of the Scala collections, making code migration seamless for Scala developers.

Finally, Apache Spark supports fault-tolerant operations by allowing RDDs to persist both in memory and in the filesystem. Persistence enables automatic recovery from node failures. The resiliency of Spark relies on the supervisory strategy of the underlying Akka actors, the persistency of their mailboxes, and the replication schemes of the HDFS.

Design principles

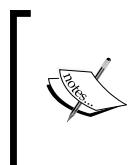
The performance of Spark relies on the following five core design principles [12:13]:

- In-memory persistency
- Laziness in scheduling tasks
- Transform and actions applied to RDDs
- Implementation of shared variables
- Support for data frames (SQL-aware RDDS)

In-memory persistency

The developer can decide to persist and/or cache an RDD for future usage. An RDD may persist in memory only or on disk only—in memory if available, or on disk otherwise as deserialized or serialized Java objects. For instance, an RDD, `rdd`, can be cached through serialization through a simple statement, as shown in the following code:

```
rdd.persist(StorageLevel.MEMORY_ONLY_SER).cache
```



Kryo serialization

Java serialization through the `Serializable` interface is notoriously slow. Fortunately, the Spark framework allows the developer to specify a more efficient serialization mechanism such as the Kryo library.

Laziness

Scala supports lazy values natively. The left-hand side of the assignment, which can either be a value, object reference, or method, is performed once, that is, the first time it is invoked, as shown in the following code:

```
class Pipeline {
    lazy val x = { println("x"); 1.5}
    lazy val m = { println("m"); 3}
    val n = { println("n"); 6}
```

```
def f = (m <<1)
def g(j: Int) = Math.pow(x, j)
}
val pipeline = new Pipeline //1
pipeline.g(pipeline.f) //2
```

The order of the variables printed is `n`, `m`, and then `x`. The instantiation of the `Pipeline` class initializes `n` but not `m` or `x` (line 1). At a later stage, the `g` method is called, which in turn invokes the `f` method. The `f` method initializes the `m` value it needs, and then `g` initializes `x` to compute its power to `m <<1` (line 2).

Spark applies the same principle to RDDs by executing the transformation only when an action is performed. In other words, Spark postpones memory allocation, parallelization, and computation until the driver code gets the result through the execution of an action. The cascading effect of invoking all these transformations backward is performed by the direct acyclic graph scheduler.

Transforms and actions

Spark is implemented in Scala, so you should not be too surprised to know that the most relevant Scala higher methods on collections are supported in Spark. The first table describes the transformation methods using Spark, as well as their counterparts in the Scala standard library. We use the (K, V) notation for (key, value) pairs:

Spark	Scala	Description
<code>map(f)</code>	<code>map(f)</code>	This transforms an RDD by executing the <code>f</code> function on each element of the collection
<code>filter(f)</code>	<code>filter(f)</code>	This transforms an RDD by selecting the element for which the <code>f</code> function returns <code>true</code>
<code>flatMap(f)</code>	<code>flatMap(f)</code>	This transforms an RDD by mapping each element to a sequence of output items
<code>mapPartitions(f)</code>		This executes the <code>map</code> method separately on each partition
<code>sample</code>		This samples a fraction of the data with or without a replacement using a random generator
<code>groupByKey</code>	<code>groupBy</code>	This is called on (K, V) to generate a new $(K, Seq(V))$ RDD
<code>union</code>	<code>union</code>	This creates a new RDD as an union of this RDD and the argument
<code>distinct</code>	<code>distinct</code>	This eliminates duplicate elements from this RDD
<code>reduceByKey(f)</code>	<code>reduce</code>	This aggregates or reduces the value corresponding to each key using the <code>f</code> function
<code>sortByKey</code>	<code>sortWith</code>	This reorganizes (K, V) in an RDD by ascending, descending, or otherwise specified order of the keys, K
<code>join</code>		This joins an RDD (K, V) with an RDD (K, W) to generate a new RDD $(K, (V, W))$

Spark	Scala	Description
coGroup		This implements a join operation but generates an RDD ($K, Seq(V), Seq(W)$)

Action methods trigger the collection or the reduction of the datasets from all partitions back to the driver, as listed here:

Spark	Scala	Description
reduce(f)	reduce(f)	This aggregates all the elements of the RDD across all the partitions and returns a Scala object to the driver
collect	collect	This collects and returns all the elements of the RDD across all the partitions as a list in the driver
count	count	This returns the number of elements in the RDD to the driver
first	head	This returns the first element of the RDD to the driver
take(n)	take(n)	This returns the first n elements of the RDD to the driver
takeSample		This returns an array of random elements from the RDD back to the driver
saveAsTextFile		This writes the elements of the RDD as a text file in either the local filesystem or HDFS
countByKey		This generates an (K, Int) RDD with the original keys, K , and the count of values for each key
foreach	foreach	This executes a $T \Rightarrow$ Unit function on each elements of the RDD

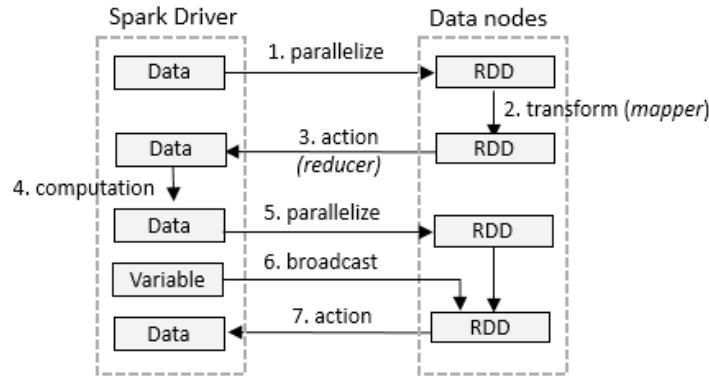
Scala methods such as `fold`, `find`, `drop`, `flatten`, `min`, `max`, and `sum` are not currently implemented in Spark. Other Scala methods such as `zip` have to be used carefully, as there is no guarantee that the order of the two collections in `zip` is maintained between partitions.

Shared variables

In a perfect world, variables are immutable and local to each partition to avoid race conditions. However, there are circumstances where variables have to be shared without breaking the immutability provided by Spark. To this extent, Spark duplicates shared variables and copies them to each partition of the dataset. Spark supports the following types of shared variables:

- **Broadcast values:** These values encapsulate and forward data to all the partitions
- **Accumulator variables:** These variables act as summations or reference counters

The four design principles can be summarized in the following diagram:



An interaction between the Spark driver and RDDs

The preceding diagram illustrates the most common interaction between the Spark driver and its workers, as listed in the following steps:

1. The input data, residing in either the memory as a Scala collection or HDFS as a text file, is parallelized and partitioned into an RDD.
2. A transformation function is applied to each element of the dataset across all the partitions.
3. An action is performed to reduce and collect the data back to the driver.
4. The data is processed locally within the driver.
5. A second parallelization is performed to distribute computation through the RDDs.
6. A variable is broadcast to all the partitions as an external parameter of the last RDD transformation.
7. Finally, the last action aggregates and collects the final result back in the driver.

If you take a look at it closely, the management of datasets and RDDs by the Spark driver is not very different from that by the Akka master and worker actors of futures.

Experimenting with Spark

Spark's in-memory computation for iterative computing makes it an excellent candidate to distribute the training of machine learning models, implemented with dynamic programming or optimization algorithms. Spark runs on Windows, Linux, and Mac OS operating systems. It can be deployed either in local mode for a single host or master mode for a distributed environment. The version of the Spark framework used is 1.3.

JVM and Scala compatible versions

 At the time of writing, the version of Spark 1.3.0 required Java 1.7 or higher and Scala 2.10.2 or higher. Spark 1.5.0 supports Scala 2.11 but requires the framework to be reassembled with the flag D-scala2.11.

Deploying Spark

The easiest way to learn Spark is to deploy a localhost in standalone mode. You can either deploy a precompiled version of Spark from the website, or build the JAR files using the **simple build tool (sbt)** or Maven [12:14] as follows:

1. Go to the download page at <http://spark.apache.org/downloads.html>.
2. Choose a package type (Hadoop distribution). The Spark framework relies on the HDFS to run in cluster mode; therefore, you need to select a distribution of Hadoop or an open source distribution such as MapR or Cloudera.
3. Download and decompress the package.
4. If you are interested in the latest functionality added to the framework, check out the newest source code at <http://github.com/apache/spark.git>.
5. Next, you need to build, or assemble, the Apache Spark libraries from the top-level directory using either Maven or sbt:

- **Maven:** Set the following Maven options to support build, deployment, and execution:

```
MAVEN_OPTS="-Xmx4g -XX:MaxPermSize=512M
           -XX:ReservedCodeCacheSize=512m"
mvn [args] -DskipTests clean package
```

The following are some examples:

- Building on Hadoop 2.4 using Yarn clusters manager and Scala 2.10 (default):

```
mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -  
DskipTests  
    clean package
```

- Building on Hadoop 2.6 using Yarn clusters manager and Scala 2.11:

```
mvn -Pyarn -Phadoop-2.6 -Dhadoop.version=2.6.0 -  
Dscala-2.11  
    -DskipTests clean package
```

- **A simple build tool:** Use the following command:

```
sbt/sbt [args] assembly
```

The following are some examples:

- Building on Hadoop 2.4 using Yarn clusters manager and Scala 2.10 (default):

```
sbt -Pyarn -pHadoop 2.4 assembly
```

- Building on Hadoop 2.6 using Yarn clusters manager and Scala 2.11:

```
sbt -Pyarn -pHadoop 2.6 -Dscala-2.11 assembly
```

Installation instructions

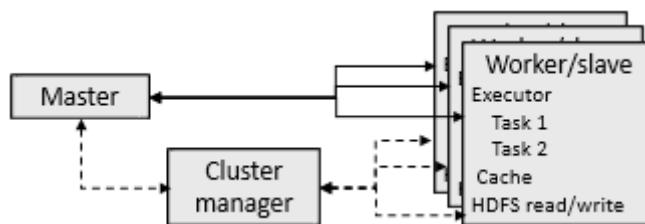
The directory and name of artifacts used in Spark will undoubtedly change over time. You can refer to the documentation and installation guide for the latest version of Spark.

Apache supports multiple deployment modes:

- **Standalone mode:** The drivers and executors run as master and slave Akka actors, bundled with the default spark distribution JAR file.
- **Local mode:** This is a standalone mode running on a single host. The slave actors are deployed across multiple cores within the same host.
- **Yarn clusters manager:** Spark relies on the Yarn resource manager running on Hadoop version 2 and higher. The Spark driver can run either on the same JVM as the client application (client mode) or on the same JVM as the master (cluster mode).

- **Apache Mesos resource manager:** This deployment allows dynamic and scalable partitioning. Apache Mesos is an open source and general-purpose cluster manager that has to be installed separately (refer to <http://mesos.apache.org/>). Mesos manages abstracted the hardware artifacts such as memory or storage.

The communication between a master node (or driver), cluster manager, and set of slave (or worker) nodes is illustrated in the following diagram:



The communication between a master, slave nodes, and a cluster manager

Installation under Windows

 Hadoop relies on some UNIX/Linux utilities that need to be added to the development environment when running on Windows. The `winutils.exe` file has to be installed and added to the `HADOOP_PATH` environment variable.

Using Spark shell

Use any of the following methods to use the Spark shell:

- The shell is an easy way to get your feet wet with Spark-resilient distributed datasets. To launch the shell locally, execute `./bin/spark-shell -master local[8]` to execute the shell on an 8-core localhost.
- To launch a Spark application locally, connect to the shell and execute the following command line:

```

./bin/spark-submit --class application_class --master local[4]
--executor-memory 12G --jars myApplication.jar
--class myApp.class

```

The command launches the application, `myApplication`, with the `myApp.main` method on a 4-core CPU localhost and 12 GB of memory.

- To launch the same Spark application remotely, connect to the shell execute the following command line:

```
./bin/spark-submit --class application_class  
    --master spark://162.198.11.201:7077  
    --total-executor-cores 80  
    --executor-memory 12G  
    --jars myApplication.jar -class myApp.class
```

The output will be as follows:

A partial screenshot of the Spark shell command line output

Potential pitfalls with the Spark shell



Depending on your environment, you might need to disable logging information into the console by reconfiguring `conf/log4j.properties`. The Spark shell might also conflict with the declaration of classpath in the profile or the environment variables' list. In this case, it has to be replaced by `ADD_JARS` as an environment variable such as `ADD_JARS = path1/jar1, path2/jar2`.

MLlib

MLLib is a scalable machine learning library built on top of Spark. As of version 1.0, the library is a work in progress.

The main components of the library are as follows:

- Classification algorithms, including logistic regression, Naïve Bayes, and support vector machines
- Clustering limited to K-means in version 1.0
- L1 and L2 regularization
- Optimization techniques such as gradient descent, logistic gradient and stochastic gradient descent, and L-BFGS
- Linear algebra such as the singular value decomposition
- Data generator for K-means, logistic regression, and support vector machines

The machine learning bytecode is conveniently included in the Spark assembly JAR file built with the simple build tool.

RDD generation

The transformation and actions are performed on RDDs. Therefore, the first step is to create a mechanism to facilitate the generation of RDDs from a time series. Let's create an `RDDSource` singleton with a `convert` method that transforms a time series `xt` into an RDD, as shown here:

```
def convert(
    xt: immutable.Vector[DblArray],
    rddConfig: RDDConfig)
  (implicit sc: SparkContext): RDD[Vector] = {

  val rdd: RDD[Vector] =
    sc.parallelize(xt.toVector.map(new DenseVector(_))) //3
  rdd.persist(rddConfig.persist) //4
  if( rddConfig.cache) rdd.cache //5
  rdd
}
```

The last `rddConfig` argument of the `convert` method specifies the configuration for the RDD. In this example, the configuration of the RDD consists of enabling/disabling cache and selecting the persistency model, as follows:

```
case class RDDConfig(val cache: Boolean,
                     val persist: StorageLevel)
```

It is fair to assume that `SparkContext` has already been implicitly defined in a manner quite similar to `ActorSystem` in the Akka framework.

The generation of the RDD is performed in the following steps:

1. Create an RDD using the `parallelize` method of the context and convert it into a vector (`SparseVector` or `DenseVector`) (line 3).
2. Specify the persistency model or the storage level if the default level needs to be overridden for the RDD (line 3).
3. Specify whether the RDD has to persist in memory (line 5).

 **An alternative for the creation of an RDD**
An RDD can be generated from data loaded from either the local filesystem or HDFS using the `SparkContext.textFile` method that returns an RDD of a string.

Once the RDD is created, it can be used as an input for any algorithm defined as a sequence of transformation and actions. Let's experiment with the implementation of the K-means algorithm in Spark/MLLib.

K-means using Spark

The first step is to create a `SparkKMeansConfig` class to define the configuration of the Apache Spark K-means algorithm, as follows:

```
class SparkKMeansConfig(k: Int, maxIters: Int,  
    numRuns: Int = 1) {  
    val kmeans: KMeans = {  
        (new KMeans).setK(k) //6  
        .setMaxIterations(maxIters) //7  
        .setRuns(numRuns) //8  
    }  
}
```

The minimum set of initialization parameters for MLLib K-means algorithm is as follows:

- The number of clusters, `k` (line 6)
- The maximum number of iterations for the reconstruction of the total errors, `maxIters` (line 7)
- The number of training runs, `numRuns` (line 8)

The `SparkKMeans` class wraps the Spark `KMeans` into a data transformation of the `ITransform` type, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!* The class follows the design template for a classifier, as explained in the *Design template for immutable classifiers* section in the *Appendix A, Basic Concepts*:

```
class SparkKMeans(      //9
    kMeansConfig: SparkKMeansConfig,
    rddConfig: RDDConfig,
    xt: Vector[DblArray])
(implicit sc: SparkContext) extends ITransform[DblArray](xt) {

    type V = Int      //10
    val model: Option[KMeansModel] = train //11

    override def |> : PartialFunction[DblArray, Try[V]] //12
    def train: Option[KMeansModel]
}
```

The constructor takes three arguments: the Apache Spark `KMeans` configuration `kMeansConfig`, the RDD configuration `rddConfig`, and the `xt` input time series for clustering (line 9). The return type of the `ITransform` trait's partial function `|>` is defined as an `Int` (line 10).

The generation of `model` merely consists of converting the time series `xt` into an RDD using `rddConfig` and invoking MLlib `KMeans.run` (line 11). Once it is created, the model of clusters (`KMeansModel`) is available for predicting a new observation, `x`, (line 12), as follows:

```
override def |> : PartialFunction[DblArray, Try[V]] = {
    case x: DblArray if(x.length > 0 && model != None) =>
        Try[V](model.get.predict(new DenseVector(x)))
}
```

The `|>` prediction method returns the index of the cluster of observations.

Finally, let's write a simple client program to exercise the `SparkKMeans` model using the volatility of the price of a stock and its daily trading volume. The objective is to extract clusters with features (volatility and volume), each cluster representing a specific behavior of the stock:

```
val K = 8
val RUNS = 16
val MAXITERS = 200
val PATH = "resources/data/chap12/CSCO.csv"
val CACHE = true
```

```
val sparkConf = new SparkConf().setMaster("local[8]")
  .setAppName("SparkKMeans")
  .set("spark.executor.memory", "2048m") //13
implicit val sc = new SparkContext(sparkConf) //14

extract.map { case (vty,vol) => { //15
  val vtyVol = zipToSeries(vty, vol)
  val conf = SparkKMeansConfig(K,MAXITERS,RUNS) //16
  val rddConf = RDDConfig(CACHE,
    StorageLevel.MEMORY_ONLY) //17

  val pfnSparkKMeans = SparkKMeans(conf,rddConf,vtyVol) |> //18
  val obs = Array[Double](0.23, 0.67)
  val clusterId = pfnSparkKMeans(obs)
}
```

The first step is to define the minimum configuration for the `sc` context (line 13) and initialize it (line 14). The `vty` and `vol` volatility variables are used as features for K-means and extracted from a CSV file (line 15):

```
def extract: Option[(DblVector, DblVector)] = {
  val extractors = List[Array[String] => Double](
    YahooFinancials.volatility, YahooFinancials.volume
  )
  val pfnSrc = DataSource(PATH, true) |>
  pfnSrc( extractors ) match {
    case Success(x) => Some((x(0).toVector, x(1).toVector))
    case Failure(e) => { error(e.toString); None }
  }
}
```

The execution creates a configuration `conf` for the K-means (line 16) and another configuration for the Spark RDD, `rddConf`, (line 17). The `pfnSparkKMeans` partial function, which implements the K-means algorithm, is created with the K-means, RDD configurations, and the input data `vtyVol` (line 18).

Performance evaluation

Let's execute the normalization of the cross-validation groups on an 8-core CPU machine with 32 GB of RAM. The data is partitioned with a ratio of two partitions per CPU core.

 **A meaningful performance test**

The scalability test should be performed with a large number of data points (normalized volatility, normalized volume), in excess of 1 million, in order to estimate the asymptotic time complexity.

Tuning parameters

The performance of a Spark application depends greatly on the configuration parameters. Selecting the appropriate value for those configuration parameters in Spark can be overwhelming – there are 54 configuration parameters as of the last count. Fortunately, the majority of those parameters have relevant default values. However, there are few parameters that deserve your attention, including the following:

- The number of cores available to execute transformation and actions on RDDs (`config.cores.max`).
- Memory available for the execution of the transformation and actions (`spark.executor.memory`). Setting the value to 60 percent of the maximum JVM heap is a generally a good compromise.
- The number of concurrent tasks to use across all the partitions for shuffle-related operations; they use a key such as `reduceByKey` (`spark.default.parallelism`). The recommended formula is $\text{parallelism} = \text{total number of cores} \times 2$. The value of the parameter can be overridden with the `spark.reducer.partitions` parameter for specific RDD reducers.
- A flag to compress a serialized RDD partition for `MEMORY_ONLY_SER` (`spark.rdd.compress`). The purpose is to reduce memory footprints at the cost of extra CPU cycles.
- The maximum size of messages containing the results of an action is sent to the `spark.akka.frameSize` driver. This value needs to be increased if a collection may potentially generate a large size array.
- A flag to compress large size broadcasted `spark.broadcast.compress` variables. It is usually recommended.

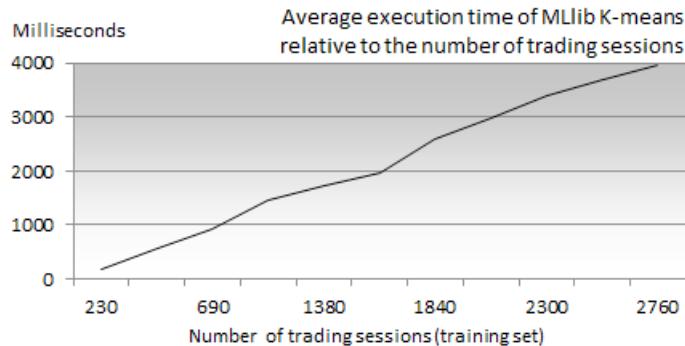
Tests

The purpose of the test is to evaluate how the execution time is related to the size of the training set. The test executes K-means from the MLlib library on the volatility and trading session volume on the **Bank of America (BAC)** stock over the following periods: 3 months, 6 months, 12 months, 24 months, 48 months, 60 months, 72 months, 96 months, and 120 months.

The following configuration is used to perform the training of K-means: 10 clusters, 30 maximum iterations, and 3 runs. The test is run on a single host with 8-CPU cores and 32 GB of RAM. The test was conducted with the following values of parameters:

- StorageLevel = MEMORY_ONLY
- spark.executor.memory = 12G
- spark.default.parallelism = 48
- spark.akka.frameSize = 20
- spark.broadcast.compress = true
- No serialization

The first step after executing a test for a specific dataset is to log in to the Spark monitoring console at http://host_name:4040/stages:



The average duration of the K-means clustering versus size of training data in months

Obviously, each environment produces somewhat different performance results but confirms that the time complexity of the Spark K-means is a linear function of the training set.

Performance evaluation in a distributed environment

A Spark deployment on multiple hosts will add latency to the overall execution time of the TCP communication. The latency is related to the collection of the results of the clustering back to the Spark driver, which is negligible and independent of the size of the training set.

Performance considerations

This test barely scratches the surface of the capabilities of Apache Spark. The following are the lessons learned from personal experience in order to avoid the most common performance pitfalls when deploying Spark 1.3+:

- Get acquainted with the most common Spark configuration parameters regarding partitioning, storage level, and serialization.
- Avoid serializing complex or nested objects unless you use an effective Java serialization library such as Kryo.
- Look into defining your own partitioning function to reduce large key-value pair datasets. The convenience of `reduceByKey` has its price. The ratio of number of partitions to number of cores has an impact on the performance of a reducer using keys.
- Avoid unnecessary actions such as `collect`, `count`, or `lookup`. An action reduces the data residing in the RDD partitions, and then forwards it to the Spark driver. The Spark driver (or master) program runs on a single JVM with limited resources.
- Rely on shared or broadcast variables whenever necessary. Broadcast variables, for instance, improve the performance of operations on multiple datasets with very different sizes. Let's consider the common case of joining two datasets of very different sizes. Broadcasting the smaller dataset to each partition of the RDD of the larger dataset is far more efficient than converting the smaller dataset into an RDD and executing a join operation between the two datasets.
- Use an accumulator variable for summation as it is faster than using a reduce action on an RDD.

Pros and cons

An increasing number of organizations are adopting Spark as their distributed data processing platform for real-time or pseudo real-time operations. There are several reasons for the fast adoption of Spark:

- It is supported by a large and dedicated community of developers [12:15]
- In-memory persistency is ideal for iterative computation found in machine learning and statistical inference algorithms
- Excellent performance and scalability that can be extended with the Streaming module

- Apache Spark leverages Scala functional capabilities and a large number of open source Java libraries
- Spark can leverage the Mesos or Yarn cluster manager, which reduces the complexity of defining fault-tolerance and load balancing between worker nodes
- Spark needs to be integrated with commercial Hadoop vendors such as Cloudera

However, no platform is perfect and Spark is no exception. The most common complaints or concerns regarding Spark are as follows:

- Creating a Spark application can be intimidating for a developer with no prior knowledge of functional programming.
- The integration with the database has been somewhat lagging, relying heavily on Hive. The Spark development team has started to address these limitations with the introduction of SparkSQL and data frame RDDs.

0xdata Sparkling Water

Sparkling Water is an initiative to integrate **0xdata H2O** with Spark and complement MLlib [12:16]. H2O from 0xdata is a very fast, open source, in-memory platform for machine learning for very large datasets (<http://0xdata.com/product/>). The framework is worth mentioning for the following reasons:

- It has a Scala API
- It is fully dedicated to machine learning and predictive analytics
- It leverages both the frame data representation of H2O and in-memory clustering of Spark

H2O has an extensive implementation of the generalized linear model and gradient boosted classification, among other goodies. Its data representation consists of hierarchical **data frames**. A data frame is a container of vectors potentially shared with other frames. Each vector is composed of **data chunks**, which themselves are containers of **data elements** [12:17]. At the time of writing, Sparkling Water is in beta version.

Summary

This completes the introduction of the most common scalable frameworks built using Scala. It is quite challenging to describe frameworks, such as Akka and Spark, as well as new computing models such as Actors, futures, and RDDs, in a few pages. This chapter should be regarded as an invitation to further explore the capabilities of those frameworks in both a single host and a large deployment environment.

In this last chapter, we learned:

- The benefits of asynchronous concurrency
- The essentials of the actor model and composing futures with blocking or callback modes
- How to implement a simple Akka cluster to squeeze performance of distributed applications
- The ease and blazing performance of Spark's resilient distributed datasets and the in-memory persistency approach

Basic Concepts

Machine learning algorithms make significant use of linear algebra and optimization techniques. Describing the concept and the implementation of linear algebra, calculus, and optimization algorithms in detail would have added significant complexity to the book and distracted the reader from the essence of machine learning.

The appendix lists a basic set of elements of linear algebra and optimization mentioned throughout the book. It also summarizes the coding practices and acquaints the reader with basic knowledge of financial analysis.

Scala programming

Here is a partial list of coding practices and design techniques used throughout the book.

List of libraries and tools

The precompiled *Scala for Machine Learning* code is `ScalaML-2.11-0.99.jar` located in the `$ROOT/project/target/scala-2.11` directory. Not all the libraries are needed for every chapter. The list is as follows:

- Java JDK 1.7 or 1.8 is required for all chapters
- Scala 2.10.4 or higher is required for all chapters
- Scala IDE for Eclipse 4.0 or higher
- IntelliJ IDEA Scala plugin 13.0 or higher
- sbt 0.13 or higher

- Apache Commons Math 3.5+ is required for *Chapter 3, Data Preprocessing*, *Chapter 4, Unsupervised Learning*, and *Chapter 6, Regression and Regularization*
- JFChart 1.0.7 is required for *Chapter 1, Getting Started*, *Chapter 2, Hello World!*, *Chapter 5, Naïve Bayes Classifiers*, and *Chapter 9, Artificial Neural Networks*
- Iitb CRF 0.2 (including the LBFGS and Colt libraries) is required for *Chapter 7, Sequential Data Models*
- LIBSVM 0.1.6 is required for *Chapter 8, Kernel Models and Support Vector Machines*
- Akka framework 2.2 or higher is required for *Chapter 12, Scalable Frameworks*
- Apache Spark/MLib 1.3 or higher is required for *Chapter 12, Scalable Frameworks*
- Apache Maven 3.3 or higher (required for Apache Spark 1.4 or higher)

 **A note for Spark developers**
The Scala library and compiler JAR files bundled with the assembly JAR file for Apache Spark contain a version of the Scala standard library and compiler JAR file that may conflict with an existing Scala library (that is, Eclipse default ScalaIDE library).

The `lib` directory contains the following JAR files related to the third-party libraries or frameworks used in the book: colt, CRF, LBFGS and LIBSVM.

Code snippets format

For the sake of readability of the implementation of algorithms, all nonessential code such as error checking, comments, exception, or import have been omitted. The following code elements are discarded in the code snippets presented in the book:

- Comments:

```
/**  
 * This class is defined as ...  
 */  
// The MathRuntime exception has to be caught here!
```

- Validation of class parameters and method arguments:

```
class BaumWelchEM(val lambda: HMMLambda ...) {  
    require( lambda != null, "Lambda model is undefined")
```

- Class qualifiers such as `final` and `private`:

```
final protected class MLP[T <% Double] ...
```

- Method qualifiers and access control (`final`, `private`, and so on):

```
final def inputLayer: MLPLayer  
private def recurse: Unit =
```

- Serialization:

```
class Config extends Serializable { ... }
```

- Validation of partial functions:

```
val pfn: PartialFunction[U, V]  
pfn.isDefinedAt(u)
```

- Validation of intermediate states:

```
assert( p != None, " ... ")
```

- Java style exceptions:

```
try { ... }  
catch { case e: ArrayIndexOutOfBoundsException => ... }  
if (y < EPS)  
    throw new IllegalStateException( ... )
```

- Scala style exceptions:

```
Try(process(args)) match {  
    case Success(results) => ...  
    case Failure(e) => ...  
}
```

- Nonessential annotations:

```
@inline def mean = { ... }  
@implicitNotFound("Conversion $T to Array[Int] undefined")  
@throws(classOf[IllegalStateException])
```

- Logging and debugging code:

```
m_logger.debug( ...)  
Console.println( ... )
```

- Auxiliary and nonessential methods

Best practices

Encapsulation

One important objective while creating an API is to reduce the access to support a helper class. There are two options to encapsulate helper classes, as follows:

- **A package scope:** The supporting classes are first-level classes with protected access
- **A class or object scope:** The supported classes are nested in the main class

The algorithms presented in this book follow the first encapsulation pattern.

Class constructor template

The constructors of a class are defined in the companion object using `apply` and the class has a package scope (`protected`):

```
protected class A[T] (val x: X, val y: Y,...) { ... }
object A {
  def apply[T] (x: X, y: Y, ...): A[T] = new A(x, y,...)
  def apply[T] (x: , ...): A[T] = new A(x, y0, ...)
}
```

For example, the `SVM` class that implements the support vector machine is defined as follows:

```
final protected class SVM[T <: AnyVal] (
  config: SVMConfig,
  xt: XVSeries[T],
  expected: DblVector)(implicit f: T => Double)
  extends ITransform[Array[T]](xt) {
```

The `SVM` companion object is responsible for defining all the constructors (instance factories) relevant to the `SVM` protected class:

```
def apply[T <: AnyVal] (
  config: SVMConfig,
  xt: XVSeries[T],
  expected: DblVector)(implicit f: T => Double): SVM[T] =
  new SVM[T](config, xt, expected)
```

Companion objects versus case classes

In the preceding example, the constructors are explicitly defined in the companion object. Although the invocation of the constructor is very similar to the instantiation of case classes, there is a major difference; the Scala compiler generates several methods to manipulate an instance as regular data (equals, copy, hash, and so on).

Case classes should be reserved for single state data objects (no methods).

Enumerations versus case classes

It is quite common to read or hear discussions regarding the relative merit of enumerations and pattern matching with case classes in Scala [A:1]. As a very general guideline, enumeration values can be regarded as lightweight case classes or case classes can be considered as heavy weight enumeration values.

Let's take an example of a Scala enumeration that consists of evaluating the uniform distribution of the `scala.util.Random` library:

```
object A extends Enumeration {
    type TA = Value
    val A, B, C = Value
}

import A._
val counters = Array.fill(A.maxId+1)(0)
Range(0, 1000).foreach(_ => Random.nextInt(10) match {
    case 3 => counters(A.id) += 1
    ...
    case _ => {}
})
```

The pattern matching is very similar to the Java's `switch` statement.

Let's consider the following example of pattern matching using case classes that selects a mathematical formula according to the input:

```
package AA {
    sealed abstract class A(val level: Int)
    case class AA extends A(3) { def f = (x:Double) => 23*x}
    ...
}

import AA._
def compute(a: A, x: Double): Double = a match {
    case a: A => a.f(x)
    ...
}
```

The pattern matching is performed using the default equals method, whose byte code is automatically set for each case class. This approach is far more flexible than the simple enumeration at the cost of extra computation cycles.

The advantages of using enumerations over case classes are as follows:

- Enumerations involve less code for a single attribute comparison
- Enumerations are more readable, especially for Java developers.

The advantages of using case classes are as follows:

- Case classes are data objects and support more attributes than enumeration IDs
- Pattern matching is optimized for sealed classes as the Scala compiler is aware of the number of cases

In short, you should use enumeration for single value constants and case classes to match data objects.

Overloading

Contrary to C++, Scala does not actually overload operators. Here is the definition of the very few operators used in code snippets:

- `+=`: This adds an element to a collection or container
- `+`: This sums two elements of the same type

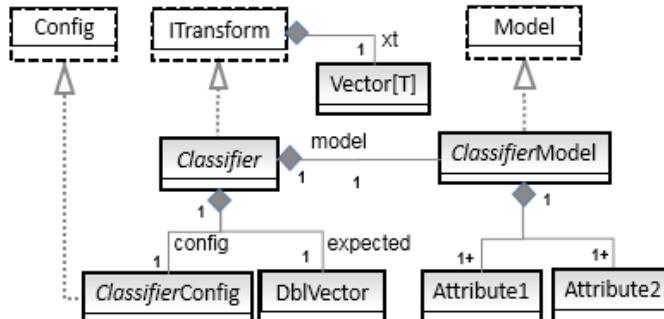
Design template for immutable classifiers

The machine learning algorithms described in this book uses the following design pattern and components:

- The set of configuration and tuning parameters for the classifier is defined in a class inheriting from `Config` (that is, `SVMConfig`).
- The classifier implements a monadic data transformation of the `ITransform` type for which the model is implicitly generated from a training set (that is, `SVM[T]`). The classifier requires at least three parameters, which are as follows:
 - A configuration for the execution of the training and classification tasks
 - An input dataset, `xt`, of the `Vector[T]` type
 - A vector of labels or expected values

- A model of type inherited from `Model`. The constructor is responsible for creating the model through training (that is, `SVMModel`).

Let's take a look at the following diagram:



A generic UML class diagram for classifiers

For example, the key components of the support vector machine package are the classifier SVMs:

```

final protected class SVM[T <: AnyVal] (
  val config: SVMConfig,
  val xt: XTSeries[Array[T]],
  val labels: DblVector)(implicit f: T => Double)
extends ITransform[Array[T]](xt) with Monitor[Double] {

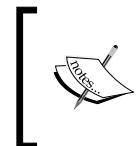
  type V =
  val model: Option[SVMModel] = { ... }
  override def |> PartialFunction[Array[T], V]
  ...
}
  
```

The training set is created by combining or zipping the input dataset `xt` with the labels or expected values `expected`. Once trained and validated, the model is available for prediction or classification.

This design has the main advantage of reducing the life cycle of a classifier: a model is either defined, available for classification, or is not created.

The configuration and model classes are implemented as follows:

```
final class SVMConfig(val formulation: SVMFormulation,  
                      val kernel: SVMKernel,  
                      val svmExec: SVMExecution) extends Config  
  
class SVMModel(val svmmode1: svm_model) extends Model
```



Implementation considerations

The validation phase is omitted in most of the practical examples throughout the book for the sake of readability.



Utility classes

Data extraction

A CSV file is the most common format used to store historical financial data. It is the default format used to import data throughout the book. The data source relies on a `DataSourceConfig` configuration class, as follows:

```
case class DataSourceConfig(pathName: String, normalize: Boolean,  
                           reverseOrder: Boolean, headerLines: Int = 1)
```

The parameters of the `DataSourceConfig` class are as follows:

- `pathName`: This is the relative pathname of a data file to be loaded if the argument is a file or the directory containing multiple input data files. Most of files are CSV files.
- `normalize`: This is the flag that is used to specify whether the data has to be normalized over [0, 1].
- `reverseOrder`: This is the flag that is used to specify whether the order of the data in the file has to be reversed (for example, a time series) if its value is true.
- `headerLines`: This specifies the number of lines for the column headers and comments.

The data source `DataSource` implements data transformation of the `ETransform` type using an explicit configuration `DataSourceConfig`, as described in the *Monadic data transformation* section in *Chapter 2, Hello World!*:

```
final class DataSource(config: DataSourceConfig,
  srcFilter: Option[Fields => Boolean] = None)
  extends ETransform[DataSourceConfig](config) {

  type Fields = Array[String]
  type U = List[Fields => Double]
  type V = XVSeries[Double]
  override def |> : PartialFunction[U, Try[V]] =
  ...
}
```

The `srcFilter` argument specifies the filter or condition of some of the row fields to skip the dataset (that is, missing data or incorrect format). Being an explicit data transformation, the constructor for the `DataSource` class has to initialize the `U` input type and the `V` output type of the `|>` extracting method. The method takes the extractor from a row of literal values to double floating point values:

```
override def |> : PartialFunction[U, Try[V]] = {
  case fields: U if(!fields.isEmpty) => load.map(data =>{ //1
    val convert = (f: Fields =>Double) => data._2.map(f(_))
    if( config.normalize) //2
      fields.map(t => new MinMax[Double](convert(t)) //3
        .normalize(0.0, 1.0).toArray ).toVector //4
    else fields.map(convert(_)).toVector
  })
}
```

The data is loaded from the file using the `load` helper method (line 1). The data is normalized if required (line 2) by converting each literal to a floating point value using an instance of the `MinMax` class (line 3). Finally, the `MinMax` instance normalizes the sequence of floating point values (line 4).

The `DataSource` class implements a significant set of methods that are documented in the source code available online.

Data sources

The examples in the book rely on three different sources of financial data using the CSV format:

- `YahooFinancials`: This is for Yahoo schema for the historical stock and ETF price
- `GoogleFinancials`: This is for Google schema for the historical stock and ETF price
- `Fundamentals`: This is for fundamental financial analysis ration (a CSV file)

Let's illustrate the extraction from a data source using `YahooFinancials` as an example:

```
object YahooFinancials extends Enumeration {  
    type YahooFinancials = Value  
    val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, ADJ_CLOSE = Value  
    val adjClose = ((s:Array[String]) =>  
        s(ADJ_CLOSE.id).toDouble) //5  
    val volume = (s: Fields) => s(VOLUME.id).toDouble  
    ...  
    def toDouble(value: Value): Array[String] => Double =  
        (s: Array[String]) => s(value.id).toDouble  
}
```

Let's take a look at an example of an application of a `DataSource` transformation: loading the historical stock data from the Yahoo finance site. The data is downloaded as a CSV formatted file. Each column is associated with an extractor function (line 5):

```
val symbols = Array[String] ("CSCO", ...) //6  
val prices = symbols  
    .map(s => DataSource(s"$path$s.csv",true,true,1))//7  
    .map(_ |> adjClose) //8
```

The list of stocks for which the historical data has to be downloaded is defined as an array of symbols (line 6). Each symbol is associated with a CSV file (that is, `csc0 => resources/CSCO.csv`) (line 7). Finally, the `YahooFinancials` extractor for the `adjClose` price is invoked (line 8).

The format for the financial data extracted from the Google financial pages are similar to the format used in the Yahoo finances pages:

```
object GoogleFinancials extends Enumeration {  
    type GoogleFinancials = Value  
    val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME = Value  
    val close = ((s:Array[String]) =>s(CLOSE.id).toDouble)//5  
    ...  
}
```

The `YahooFinancials`, `YahooFinancials`, and `Fundamentals` classes implement a significant number of methods that are documented in the source code available online.

Extraction of documents

The `DocumentsSource` class is responsible for extracting the date, title, and content of a list of text documents or text files. The class does not support HTML documents. The `DocumentsSource` class implements a monadic data transformation of the `ETransform` type with an explicit configuration of the `SimpleDateFormat` type:

```
class DocumentsSource(dateFormat: SimpleDateFormat,  
                      val pathName: String)  
extends ETransform[SimpleDateFormat](dateFormat) {  
  
  type U = Option[Long] //2  
  type V = Corpus[Long] //3  
  
  override def |> : PartialFunction[U, Try[V]] = { //4  
    case date: U if (filesList != None) =>  
      Try(if(date == None) getAll else get(date))  
  }  
  def get(t: U): V = getAll.filter(_.date == t.get)  
  def getAll: V //5  
  ...  
}
```

The `DocumentsSource` class takes two arguments: the format of the date associated with the document and the name of the path in which the documents are located (line 1). Being an explicit data transformation, the constructor of the `DocumentsSource` class has to initialize the `U` input type (line 2) as a date and convert it into a `Long` and `V` output type (line 3) as a `Corpus` to extract the `|>` method.

The `|>` extractor generates a corpus associated with a specific date and converts it into a `Long` type (line 4). The `getAll` method does the heavy lifting to extract or sort documents (line 5).

The implementation of the `getAll` method as well as other methods of the `DocumentsSource` class are described in the documented source code available online.

DMatrix class

Some discriminative learning models require operations to be performed on rows and columns of a matrix. The `DMatrix` class facilitates the read and write operations on columns and rows:

```
class DMatrix(val nRows: Int, val nCols: Int,
             val data: DblArray) {
  def apply(i: Int, j: Int): Double = data(i*nCols+j)
  def row(iRow: Int): DblArray = {
    val idx = iRow*nCols
    data.slice(idx, idx + nCols)
  }
  def col(iCol: Int): IndexedSeq[Double] =
    (iCol until data.size by nCols).map( data(_) )
  def diagonal: IndexedSeq[Double] =
    (0 until data.size by nCols+1).map( data(_) )
  def trace: Double = diagonal.sum
  ...
}
```

The `apply` method returns an element of the matrix. The `row` method returns a row array, and the `col` method returns the indexed sequence of column elements. The `diagonal` method returns the indexed sequence of diagonal elements, and the `trace` method sums the diagonal elements.

The `DMatrix` class supports normalization of elements, rows, and columns; transposition; and updation of elements, columns and rows. The `DMatrix` class implements a significant number of methods that are documented in the source code available online.

Counter

The `Counter` class implements a generic mutable counter for which the key is a parameterized type. The number of occurrences of a key is managed by a mutable hash map:

```
class Counter[T] extends mutable.HashMap[T, Int] {
  def += (t: T): type.Counter = super.put(t, getOrElse(t, 0)+1)
  def + (t: T): Counter[T] = {
    super.put(t, getOrElse(t, 0)+1); this
  }
  def ++ (cnt: Counter[T]): type.Counter = {
    cnt./:(this)((c, t) => c + t._1); this
  }
}
```

```
def / (cnt: Counter[T]): mutable.HashMap[T, Double] = map {  
    case(str, n) => (str, if( !cnt.contains(str) )  
        throw new IllegalStateException(" ... ")  
        else n.toDouble/cnt.get(str).get )  
    }  
    ...  
}
```

The `+=` operator updates the counter of the `t` key and returns itself. The `+` operator updates and then duplicates the updated counters. The `++` operator updates this counter with another counter. The `/` operator divides the count for each key by the counts of another counter.

The `Counter` class implements a significant set of methods that are documented in the source code available online.

Monitor

The `Monitor` class has two purposes:

- It stores log information and error messages using the `show` and `error` methods
- It collects and displays variables related to the recursive or iterative execution of an algorithm

The data is collected at each iteration or recursion and then displayed as a time series with iterations as *x* axis values:

```
trait Monitor[T] {  
    protected val logger: Logger  
    lazy val _counters =  
        new mutable.HashMap[String, mutable.ArrayBuffer[T]]()  
  
    def counters(key: String): Option[mutable.ArrayBuffer[T]]  
    def count(key: String, value: T): Unit  
    def display(key: String, legend: Legend)  
        (implicit f: T => Double): Boolean  
    def show(msg: String): Int = DisplayUtils.show(msg, logger)  
    def error(msg: String): Int = DisplayUtils.error(msg, logger)  
    ...  
}
```

The `counters` method returns an array associated with a specific key. The `count` method updates the data associated with a key. The `display` method plots the time series. Finally, the `show` and `error` methods send information and error messages to the standard output.

The documented source code for the implementation of the `Monitor` class is available online.

Mathematics

This section very briefly describes some of the mathematical concepts used in this book.

Linear algebra

Many algorithms used in machine learning such as minimization of a convex loss function, principal component analysis, or least squares regression invariably involves manipulation and transformation of matrices. There are many good books on the subject, from the inexpensive [A:2] to the sophisticated [A:3].

QR decomposition

The QR decomposition (or the QR factorization) is the decomposition of a matrix A into a product of an orthogonal matrix Q and upper triangular matrix R . So, $A=QR$ and $Q^TQ=I$ [A:4].

The decomposition is unique if A is a real, square, and invertible matrix. In the case of a rectangle matrix A , m by n with $m > n$, the decomposition is implemented as the dot product of two vector of matrices: $A = [Q_1, Q_2].[R_1, R_2]^T$, where Q_1 is an m by n matrix, Q_2 is an m by n matrix, R_1 is an n by n upper triangle matrix, and R_2 is an m by n null matrix.

The QR decomposition is a reliable method used to solve a large system of linear equations for which the number of equations (rows) exceeds the number of variables (columns). Its asymptotic computational time complexity for a training set of m dimensions and n observations is $O(mn^2 \cdot n^3 / 3)$.

It is used to minimize the loss function for ordinary least squares regression (refer to the *Ordinary least squares regression* section in *Chapter 6, Regression and Regularization*).

LU factorization

LU factorization is a technique used to solve a matrix equation $A.x = b$, where A is a nonsingular matrix and x and b are two vectors. The technique consists of decomposing the original matrix A as the product of a simple matrix $A = A_1 A_2 \dots A_n$.

- **Basic LU factorization:** This defines A as the product of a lower unit triangular matrix L and an upper triangular matrix U . So, $A = LU$.
- **LU factorization with a pivot:** This defines A as the product of a permutation matrix P , a lower unit triangular matrix L , and an upper triangular matrix U . So, $A = PLU$.

LDL decomposition

The **LDL decomposition** for real matrices defines a real positive matrix A as the product of a lower unit triangular matrix L , a diagonal matrix D , and the transposed matrix of L , that is, L^T . So, $A = LDL^T$.

Cholesky factorization

The **Cholesky factorization** (or the **Cholesky decomposition**) of real matrices is a special case of the LU factorization [A:4]. It decomposes a positive definite matrix A into a product of a lower triangular matrix L and its conjugate transpose L^T . So, $A = LL^T$.

The asymptotic computational time complexity for the Cholesky factorization is $O(mn^2)$, where m is the number of features (model parameters) and n is the number of observations. The Cholesky factorization is used in linear least squares Kalman filter (refer to the *The recursive algorithm* section in *Chapter 3, Data Preprocessing*) and nonlinear Quasi-Newton optimizer.

Singular Value Decomposition

The **singular value decomposition (SVD)** of real matrices defines an m by n real matrix A as the product of an m real square unitary matrix U , an m by n rectangular diagonal matrix Σ , and the transpose matrix V^T of a real matrix. So, $A = U\Sigma V^T$.

The columns of the U and V matrices are the orthogonal bases and the value of the diagonal matrix Σ is a singular value [A:4]. The asymptotic computational time complexity for the singular value decomposition for n observations and m features is $O(mn^2 - n^3)$. The singular value decomposition is used to minimize the total least squares and solve homogeneous linear equations.

Eigenvalue decomposition

The Eigen decomposition of a real square matrix A is the canonical factorization, $Ax = \lambda x$.

λ is the **eigenvalue** (scalar) corresponding to the vector x . The n by n matrix A is then defined as $A = QDQ^T$. Q is the square matrix that contains the eigenvectors and D is the diagonal matrix whose elements are the eigenvalues associated with the eigenvectors [A:5] and [A:6]. The Eigen decomposition is used in Principal Components Analysis (refer to the *Principal components analysis* section in *Chapter 4, Unsupervised Learning*).

Algebraic and numerical libraries

There are many more open source algebraic libraries available to developers as APIs besides Apache Commons Math, which is used in *Chapter 3, Data Preprocessing*, *Chapter 5, Naïve Bayes Classifiers*, *Chapter 6, Regression and Regularization*, and Apache Spark/MLLib in *Chapter 12, Scalable Frameworks*. They are as follows:

- **jBlas 1.2.3** (Java) created by Mikio Braun under the BSD revised license. This library provides Java and Scala developers a high-level Java interface to **BLAS** and **LAPACK** (<https://github.com/mikiobraun/jblas>).
- **Colt 1.2.0** (Java) is a high-performance scientific library developed at CERN under the European Organization for Nuclear Research license (<http://acs.lbl.gov/ACSSoftware/colt/>).
- **AlgeBird 2.10** (Scala) developed at Twitter under Apache Public License 2.0. It defines concepts of abstract linear algebra using monoid and monads. This library is an excellent example of high-level functional programming using Scala (<https://github.com/twitter/algebroid>).
- **Breeze 0.8** (Scala) is a numerical processing library using Apache Public License 2.0 originally created by David Hall. It is a component of the ScalaNLP suite of machine learning and numerical computing libraries (<http://www.scalanlp.org/>).

The Apache Spark/MLLib framework bundles jBlas, Colt, and Breeze. The Iitb framework for conditional random fields uses Colt linear algebra components.

 **An alternative to Java/Scala libraries**

If your application or project needs a high-performance numerical processing tool under limited resources (CPU and RAM memory), then using a C/C++ compiled library is an excellent alternative if portability is not a constraint. The binary functions are accessed through the Java Native Interface (JNI).

First order predicate logic

Propositional logic is the formulation of **axioms** or propositions. There are several formal representations of propositions:

- **Noun-VERB-Adjective:** For example, *Variance of the stock price EXCEEDS 0.76* or *Minimization of the loss function DOES NOT converge*
- **Entity-value = Boolean:** For example, *Variance of the stock price GREATER+THAN 0.76 = true* or *Minimization of the loss function converge = false*
- **Variable op value:** For example, *Variance_stock_price > 0.76* or *Minimization_loss_function != converge*

Propositional logic is subject to the rules of Boolean calculus. Let's consider three propositions: P , Q , and R and the three Boolean operators *NOT*, *AND*, and *OR*:

- $\text{NOT}(\text{NOT } P) = P$
- $P \text{ AND } \text{false} = \text{false}$, $P \text{ AND } \text{true} = P$, $P \text{ or } \text{false} = P$, and $P \text{ or } \text{true} = P$
- $P \text{ AND } Q = Q \text{ AND } P$ and $P \text{ OR } Q = Q \text{ OR } P$
- $P \text{ AND } (Q \text{ AND } R) = (P \text{ AND } Q) \text{ AND } R$

First order predicate logic, also known as **first order predicate calculus**, is the quantification of a propositional logic [A:7]. The most common formulations of the first order logic are as follows:

- Rules (for example, *IF P THEN action*)
- Existential operators

First order logic is used to describe the classifiers in the learning classifier systems (refer to the XCS rules section in *Chapter 11, Reinforcement Learning*).

Jacobian and Hessian matrices

Let's consider a function with n variables x_i and m outputs y_j such that $f: \{x_i\} \rightarrow \{y_j = f_j(x)\}$.

The **Jacobian matrix** [A:8] is the matrix of the first order partial derivatives of the output values of a continuous, differentiable function:

$$J(f) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

The **Hessian matrix** is the square matrix of the second order partial derivatives of a continuously, twice differentiable function:

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \dots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

An example is as follows:

$$f(x, y) = x^2y + e^{-y} \quad J(f) = [2xy, x^2 - e^{-y}] \quad H(f) = \begin{bmatrix} 2y & 2x \\ 2x & e^{-y} \end{bmatrix}$$

Summary of optimization techniques

The same comments regarding linear algebra algorithms apply to optimization. Treating such techniques in depth would have rendered the book impractical. However, optimization is critical to the efficiency and, to a lesser extent, the accuracy of the machine learning algorithms. Some basic knowledge in this field goes a long way to build practical solutions for large datasets.

Gradient descent methods

Steepest descent

The **steepest descent** (or gradient descent) method is one of the simplest techniques used to find a local minimum of any continuous, differentiable function F or the global minimum for any defined, differentiable, and convex function [A:9]. The value of a vector or data point x_{t+1} at iteration $t+1$ is computed from the previous value x_t using the *gradient* ∇F of function F and the slope γ :

$$x_{(t+1)} = x_{(t)} - \gamma \nabla F(a)$$

The steepest gradient algorithm is used to solve systems of nonlinear equations and minimization of the loss function in the logistic regression (refer to the *Numerical optimization* section in *Chapter 6, Regression and Regularization*), in support vector classifiers (refer to the *The nonseparable case – the soft margin* section in *Chapter 8, Kernel Models and Support Vector Machines*), and in multilayer perceptrons (refer to the *Training and classification* section in *Chapter 9, Artificial Neural Networks*).

Conjugate gradient

The **conjugate gradient** solves unconstrained optimization problems and systems of linear equations. It is an alternative to the LU factorization for positive, definite, and symmetric square matrices. The solution x^* of the equation $Ax = b$ is expanded as the weighted summation of n basis orthogonal directions p_i (or **conjugate directions**):

$$Ax = b \rightarrow \sum_{i=0}^{n-1} \alpha_i p_i x^* = b; \quad p_i \cdot p_j = 0$$

The solution x^* is extracted by computing the i^{th} conjugate vector p_i and then computing the coefficients α_i .

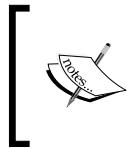
Stochastic gradient descent

The **stochastic gradient** method is a variant of the steepest descent that minimizes the convex function by defining the objective function F as the sum of differentiable, basis function f_i :

$$F(x) = \sum_{i=0}^{n-1} f_i(x), \quad x_{t+1} = x_t - \alpha \sum_{i=0}^{n-1} \nabla f_i(x)$$

The solution x_{t+1} at iteration $t+1$ is computed from the value x_t at iteration t , the step size (or the learning rate) α , and the sum of the gradient of the basis functions [A:10]. The stochastic gradient descent is usually faster than other gradient descents or quasi-Newton methods in converging toward a solution for convex functions. The stochastic gradient descent is used in logistic regression, support vector machines, and backpropagation neural networks.

Stochastic gradient is particularly suitable for discriminative models with large datasets [A:11]. Spark/Mlib makes extensive use of the stochastic gradient method.



The batch gradient descent

The batch gradient descent is introduced and implemented in the *Step 5 – implementing the classifier* section under *Let's kick the tires in Chapter 1, Getting Started*.

Quasi-Newton algorithms

Quasi-Newton algorithms are variations of Newton's method of finding the value of a vector or data point that maximizes or minimizes a function F (first order derivative is null) [A:12].

The Newton's method is a well-known and simple optimization method used to find the solution of equations $F(x) = 0$ for which F is continuous and second order differentiable. It relies on the Taylor series expansion to approximate the function F with a quadratic approximation of variable $\Delta x = x_{t+1} - x_t$ to compute the value at the next iteration using the first order F' and second order F'' derivatives:

$$F(x_t + \Delta x) - F(x_t) \approx F'(x_t) \cdot \Delta x + F''(x_t) \rightarrow x_{t+1} = x_t - \frac{F'(x_t)}{F''(x_t)}$$

Contrary to Newton's method, quasi-Newton methods do not require that the second order derivative, Hessian matrix, of the objective function be computed; it just has to be approximated [A:13]. There are several approaches to approximate the computation of the Hessian matrix.

BFGS

The **Broyden-Fletcher-Goldfarb-Shanno (BFGS)** is a quasi-Newton iterative numerical method used to solve unconstrained nonlinear problems. The hessian matrix H_{t+1} at iteration t is approximated using the value of the previous iteration t as $H_{t+1} = H_t + U_t + V_t$ applied to the Newton equation for the direction p_t :

$$H_t p_t = -\nabla F(x_t), \quad x_{t+1} = x_t + \alpha_t p_t$$

The BFGS is used in the minimization of the cost function for the conditional random field and L₁ and L₂ regressions.

L-BFGS

The performance of the BFGS algorithm is related to the caching of the approximation of the Hessian matrix in the memory (U, V) at the cost of high-memory consumption.

The **Limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS)** algorithm is a variant of BFGS that uses a minimum amount of computer RAM. The algorithm maintains the last m incremental updates of the values Δx_t and gradient ΔG_t at iteration t , and then computes these values for the next step $t+1$:

$$x_{t+1} = x_t + \Delta x_t; \quad \nabla F(x_{t+1}) = \nabla F(x_t) + \Delta G_t \text{ with } \Delta G_t = \Delta(\nabla F(x_t))$$

It is supported by the Apache Commons Math 3.3+, Apache Spark/MLlib 1.0+, Colt 1.0+, and Liitb CRF libraries. L-BFGS is used in the minimization of the loss function in conditional random fields (refer to the *Conditional random fields* section in *Chapter 7, Sequential Data Models*).

Nonlinear least squares minimization

Let's consider the classic minimization of the least squares of a nonlinear function $y = F(x, w)$ with w_i parameters for observations $\{y, x\}$. The objective is to minimize the sum of the squares of residuals r_i , which is as follows:

$$\mathcal{L}(w) = \sum_{i=0}^{m-1} r_i(w)^2 ; \quad r_i = y_i - F(x_i, w)$$

Gauss-Newton

The Gauss-Newton technique is a generalization of Newton's method. The technique solves nonlinear least squares by updating the parameters w_{t+1} at iteration $t+1$ using the first order derivative (or Jacobian):

$$w_{(t+1)} = w_{(t)} - \left\| \frac{\partial r_i(w_{(t)})}{\partial w_i} \right\|_{ij}^{-1} r(w_{(t)})$$

The Gauss-Newton algorithm is used in logistic regression (refer to the *Logistic regression* section in *Chapter 6, Regression and Regularization*).

Levenberg-Marquardt

The Levenberg-Marquardt algorithm is an alternative to the Gauss-Newton technique used to solve nonlinear least squares and curve fitting problems. The method consists of adding the gradient (Jacobian) terms to the residuals r_i to approximate the least squares error:

$$\mathcal{L}(w + \delta) \approx \sum_{i=0}^{m-1} \left(r_i(w) - \frac{\partial F(x_i, w)}{\partial w} \delta \right)^2$$

The Levenberg-Marquardt algorithm is used in the training of logistic regression (refer to the *Logistic regression* section in *Chapter 6, Regression and Regularization*).

Lagrange multipliers

The **Lagrange multipliers** methodology is an optimization technique used to find the local optima of a multivariate function, subject to equality constraints [A:14]. The problem is stated as *maximize $f(x)$ subject to $g(x) = c$, where c is a constant and x is a variable or features vector.*

This methodology introduces a new variable λ to integrate the constraint g into a function, known as the Lagrange function $\mathcal{L}(x, \lambda)$. Let's note $\nabla \mathcal{L}$, which is the gradient of \mathcal{L} over the variables x_i and λ . The Lagrange multipliers are computed by maximizing \mathcal{L} :

$$\begin{aligned}\mathcal{L}(x, \lambda) &= f(x) + \lambda(g(x) - c) \\ \nabla_{x,\lambda} \mathcal{L}(x, \lambda) &= 0 \\ \nabla \mathcal{L} &= \left[\frac{\partial \mathcal{L}}{\partial x_i}, \frac{\partial \mathcal{L}}{\partial \lambda} \right]\end{aligned}$$

An example is as follows:

$$\begin{aligned}f(x, y) &= x^2 + y^2 \text{ subject } x - y = 2 \\ \frac{\partial \mathcal{L}}{\partial x} &= 2x + \lambda, \frac{\partial \mathcal{L}}{\partial y} = 2y - \lambda, \frac{\partial \mathcal{L}}{\partial \lambda} = x - y - 2 \\ x &= 1, y = -1, \lambda = -2\end{aligned}$$

Lagrange multipliers are used to minimize the loss function in the nonseparable case of linear support vector machines (refer to the *The nonseparable case – the soft margin case* section in *Chapter 8, Kernel Models and Support Vector Machines*).

Overview of dynamic programming

The purpose of **dynamic programming** is to break down an optimization problem into a sequence of steps known as **substructures** [A:15]. There are two types of problems for which dynamic programming is suitable.

The solution of a global optimization problem can be broken down into optimal solutions for its subproblems. The solution of the subproblems is known as **optimal substructures**. Greedy algorithms or the computation of the minimum span of a graph are examples of the decomposition into optimal substructures. Such algorithms can be implemented either recursively or iteratively.

The solution of the global problem is applied recursively to the subproblems if the number of subproblems is small. This approach is known as dynamic programming using **overlapping substructures**. Forward-backward passes on hidden Markov models, the Viterbi algorithm (refer to *The Viterbi algorithm* section in *Chapter 7, Sequential Data Models*), or the backpropagation of error in a multilayer perceptron (refer to the *Step 2 – error backpropagation* section in *Chapter 9, Artificial Neural Networks*) are good examples of overlapping substructures.

The mathematical formulation of a dynamic programming solution is specific to the problem it attempts to resolve. Dynamic programming techniques are also commonly used in mathematical puzzles such as *The Tower of Hanoi*.

Finances 101

The exercises presented throughout this book are related to historical financial data and require the reader to have some basic understanding of financial markets and reports.

Fundamental analysis

Fundamental analysis is a set of techniques used to evaluate a security (stock, bond, currency, or commodity) that entails attempting to measure its intrinsic value by examining both macro and micro financial and economy reports. Fundamental analysis is usually applied to estimate the optimal price of a stock using a variety of financial ratios.

Numerous financial metrics are used throughout this book. Here are the definitions of the most commonly used metrics [A:16]:

- **Earnings per share (EPS):** This is the ratio of net earnings to the number of outstanding shares.
- **Price/earnings ratio (PE):** This is the ratio of the market price per share to earnings per share.
- **Price/sales ratio (PS):** This is the ratio of the market price per share to gross sales (or revenue).
- **Price/book value ratio (PB):** This is the ratio of the market price per share to the total balance sheet value per share.
- **Price to earnings/growth (PEG):** This is the ratio of price/earnings per share (PE) to the annual growth of earnings per share.
- **Operating income:** This is the difference between the operating revenue and operating expenses.

- **Net sales:** This is the difference between the revenue or gross sales and cost of goods or cost of sales.
- **Operating profit margin:** This is the ratio of the operating income to the net sales.
- **Net profit margin:** This is the ratio of the net profit to the net sales (or the net revenue).
- **Short interest:** This is the quantity of shares sold short and not yet covered.
- **Short interest ratio:** This is the ratio of the short interest to the total number of shares floated.
- **Cash per share:** This is the ratio of the value of cash per share to the market price per share.
- **Pay-out ratio:** This is the percentage of the primary/basic earnings per share, excluding extraordinary items paid to common stockholders in the form of cash dividends.
- **Annual dividend yield:** This is the ratio of the sum of dividends paid during the previous 12-month rolling period over the current stock price. Regular and extra dividends are included.
- **Dividend coverage ratio:** This is the ratio of the income available to common stockholders, excluding extraordinary items, for the most recent trailing 12 months to gross dividends paid to common shareholders, expressed as percent.
- **Gross Domestic Product (GDP):** This is the aggregate measure of the economic output of a country. It actually measures the sum of values added by the production of goods and delivery of services.
- **Consumer Price Index (CPI):** This is an indicator that measures the change in the price of an arbitrary basket of goods and services used by the Bureau of Labor Statistics to evaluate the inflationary trend.
- **Federal Fund rate:** This is the interest rate at which banks trade balances held at the Federal Reserve. The balances are called Federal Funds.

Technical analysis

Technical analysis is a methodology used to forecast the direction of the price of any given security through the study of the past market information derived from price and volume. In simpler terms, it is the study of price activity and price patterns in order to identify trade opportunities [A:17]. The price of a stock, commodity, bond, or financial future reflects all the information publicly known about that asset as processed by the market participants.

Terminology

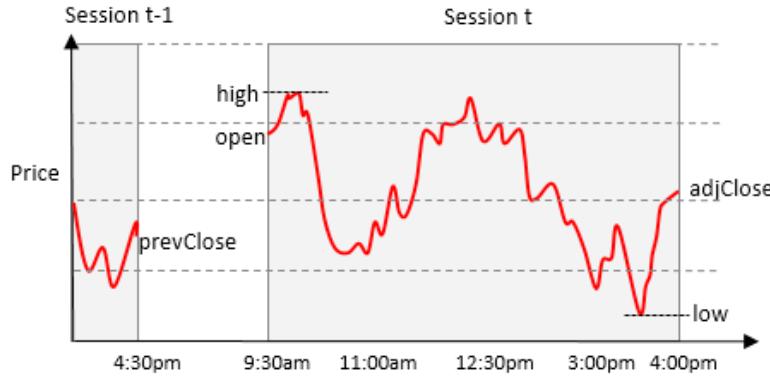
- **Bearish or bearish position:** This attempts to profit by betting that the prices of the security will fall.
- **Bullish or bullish position:** This attempts to profit by betting that the price of the security will rise.
- **Long position:** This is the same as Bullish.
- **Neutral position:** This attempts to profit by betting that the price of the security will not change significantly.
- **Oscillator:** This is a technical indicator that measures the price momentum of a security using some statistical formula.
- **Overbought:** This is a security that is overbought when its price rises too fast as measured by one or several trading signals or indicators.
- **Oversold:** This is a security that is oversold when its price drops too fast as measured by one or several trading signals or indicators.
- **Relative strength index (RSI):** This is an oscillator that computes the average of number of trading sessions for which the closing price is higher than the opening price over the average of number of trading sessions for which the closing price is lower than the opening price. The value is normalized over [0, 1] or [0, 100%].
- **Resistance:** This is the upper limit of the price range of a security. The price falls back as soon as it reaches the resistance level.
- **Short position:** This is the same as Bearish.
- **Support:** This is the lower limit of the price range of a security over a period of time. The price bounces back as soon as it reaches the support level.
- **Technical indicator:** This is a variable derived from the price of a security and possibly its trading volume.
- **Trading range:** The trading range for a security over a period of time is the difference between the highest and lowest price for this period of time.
- **Trading signal:** This is a signal that is triggered when a technical indicator reaches a predefined value, upward or downward.
- **Volatility:** This is the variance or standard deviation of the price of a security over a period of time.

Trading data

The raw trading data extracted from Google or Yahoo financials pages consists of the following:

- **adjClose (or close)**: This is the adjusted or nonadjusted price of a security at closing of the trading session
- **open**: This is the price of the security at the opening of the trading session
- **high**: This is the highest price of the security during the trading session
- **low**: This is the lowest price of the security during the trading session

Let's take a look at the following graph:



We can derive the following metrics from the raw trading data:

- Price volatility: $volatility = 1.0 - high/low$
- Price variation: $vPrice = adjClose - open$
- Price difference (or change) between two consecutive sessions: $dPrice = adjClose - prevClose = adjClose(t) - adjClose(t-1)$
- Volume difference between two consecutive sessions: $dVolume = volume(t)/volume(t-1) - 1.0$
- Volatility difference between two consecutive sessions: $dVolatility = volatility(t)/volatility(t-1) - 1.0$
- Relative price variation over the last T trading days: $rPrice = price(t)/average(price over T) - 1.0$
- Relative volume variation over the last T trading days: $rVolume = volume(t)/average(volume over T) - 1.0$
- Relative volatility variation over the last T trading days: $rVolatility = volatility(t)/average(volatility over T) - 1.0$

Trading signals and strategy

The purpose is to create a set variable x , derived from price and volume $x = f(\text{price}, \text{volume})$, and then generate predicates $x \text{ op } c$ for which op is a Boolean operator, such as $>$ or $=$ that compares the value of x to a predetermined threshold c .

Let's consider one of the most common technical indicators derived from price: the relative strength index RSI or the normalized RSI $nRSI$, whose formulation is provided here as a reference:

The relative strength index

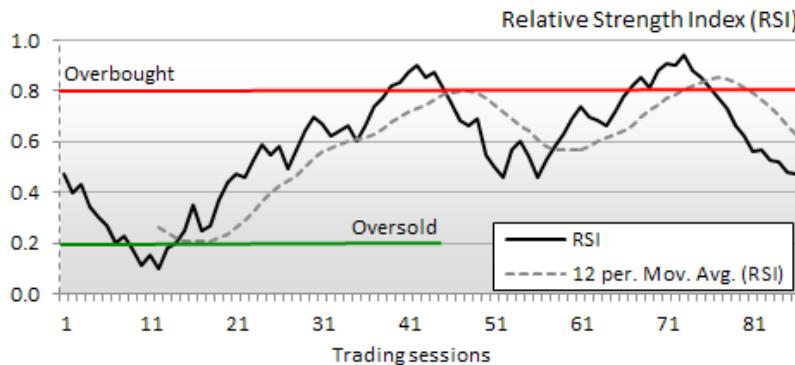
The RSI for a period of T sessions with p_o opening price and p_c closing price is defined as:



$$U_T = \sum_{t=0}^{T-1} (p_c(t) > p_o(t))$$

$$RSI_T = 100 - \frac{100}{1 + \frac{U_T}{T - U_T}} \quad nRSI_T = RSI_T / 100$$

A **trading signal** is a predicate using a technical indicator $nRSI_T(t) < 0.2$. In trading terminology, a signal is emitted for any time period t for which the predicate is true:



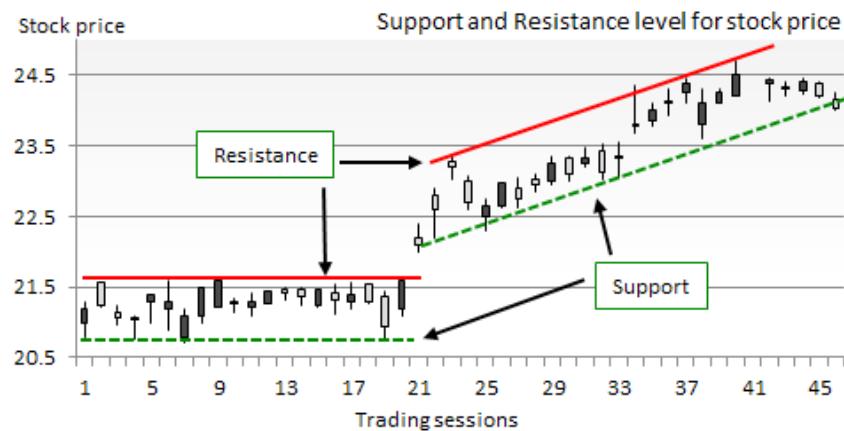
The visualization of oversold and overbought positions using the relative strength index

Traders do not usually rely on a single trading signal to make a rational decision.

For example, if G is the price of gold, I_{10} is the current rate of the 10-year Treasury bond, and RSI_{sp500} is the relative strength index of the S&P 500 index, then we can conclude that the increase in the exchange rate of US\$ to the Japanese Yen is maximized for the following trading strategy: $\{G < \$1170 \text{ and } I_{10} > 3.9\% \text{ and } RSI_{sp500} > 0.6 \text{ and } RSI_{sp500} < 0.8\}$.

Price patterns

Technical analysis assumes that historical prices contains some recurring albeit noisy, patterns that can be discovered using statistical methods. The most common patterns used in this book are the trend, support, and resistance levels [A:18], as illustrated in the following chart:



An illustration of trend, support, and resistance levels in technical analysis

Options trading

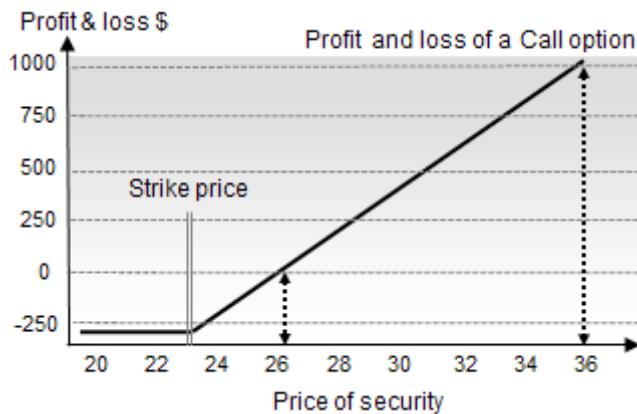
An option is a contract that gives the buyer the right, but not the obligation, to buy or sell a security at a specific price on or before a certain date [A:19].

The two types of options are calls and puts, as described here:

- A call gives the holder the right to buy a security at a certain price within a specific period of time. Buyers of calls expect that the price of the security will increase substantially over the strike price before the option expires.
- A put option gives the holder the right to sell a security at a certain price within a specific period of time. Buyers of puts expect that the price of the stock will fall below the strike price before the option expires.

Let's consider a call option contract of 100 shares at a strike price of \$23 for a total cost of \$270 (\$2.7 per option). The maximum loss the holder of the call can incur is the loss of premium or \$270 when the option expires. However, the profit can be potentially almost unlimited. If the price of the security reaches \$36 when the call option expires, the owner will have a profit of $(\$36 - \$23) * 100 - \$270 = \1030 . The return on the investment is $1030/270 = 380\%$. Buying and then selling the stock would have generated a return on the investment of $36/24 - 1 = 50\%$. This example is simple and does not take into account a transaction fee or margin cost [A:20]:

Let's take a look at the following chart:



An illustration of the pricing of a call option

Financial data sources

There are numerous sources of financial data available to experiment with machine learning and validation models [A:21]:

- Yahoo finances (stocks, ETFs, and indices): <http://finance.yahoo.com>
- Google finances (stocks, ETFs, and indices): <https://www.google.com/finance>
- NASDAQ (stocks, ETFs, and indices): <http://www.nasdaq.com>
- European Central Bank (European bonds and notes): <http://www.ecb.int>
- TrueFx (Forex): <http://www.truefx.com>
- Quandl (Economics and financials statistics): <http://www.quandl.com>
- Dartmouth University (portfolio and simulation): <http://mba.tuck.dartmouth.edu>

Suggested online courses

- *Practical Machine Learning*, J. Leek, R. Peng, B. Caffo, Johns Hopkins University (<https://www.coursera.org/jhu>)
- *Probabilistic Graphical Models*, D. Koller, Stanford University (<https://www.coursera.org/course/pgm>)
- *Machine Learning*, A. Ng, Stanford University (<https://www.coursera.org/course/ml>)

References

- [A:1] *Daily Scala: Enumeration*. J. Eichar. 2009 (<http://daily-scala.blogspot.com/2009/08/enumerations.html>)
- [A:2] *Matrices and Linear Transformations 2nd Edition*. C. Cullen. Dover Books on Mathematics. 1990
- [A:3] *Linear Algebra: A Modern Introduction*. D Poole. BROOKS/COLE CENGAGE Learning. 2010
- [A:4] *Matrix decomposition for regression analysis*. D. Bates. 2007 (<http://www.stat.wisc.edu/courses/st849-bates/lectures/Orthogonal.pdf>)
- [A:5] *Eigenvalues and Eigenvectors of Symmetric Matrices*. I. Mateev. 2013 (<http://www.slideshare.net/vanchizzle/eigenvalues-and-eigenvectors-of-symmetric-matrices>)
- [A:6] *Linear Algebra Done Right 2nd Edition* (§5 Eigenvalues and Eigenvectors) S Axler. Springer. 2000
- [A:7] *First Order Predicate Logic*. S. Kaushik. CSE India Institute of Technology, Delhi (http://www.cse.iitd.ac.in/~saroj/LFP/LFP_2013/L4.pdf)
- [A:8] *Matrix Recipes*. J. Movellan. 2005 (http://www.math.vt.edu/people/dlr/m2k_svbl1_hesian.pdf)
- [A:9] *Gradient descent*. Wikipedia (http://en.wikipedia.org/wiki/Gradient_descent)
- [A:10] *Large Scale Machine Learning: Stochastic Gradient Descent Convergence*. A. Ng. Stanford University (<https://class.coursera.org/ml-003/lecture/107>)
- [A:11] *Large-Scale Machine Learning with Stochastic Gradient Descent*. L Bottou. 2010 (<http://leon.bottou.org/publications/pdf/compstat-2010.pdf>)

- [A:12] *Overview of Quasi-Newton optimization methods.* Dept. Computer Science, University of Washington (<https://homes.cs.washington.edu/~galen/files/quasi-newton-notes.pdf>)
- [A:13] *Lecture 2-3: Gradient and Hessian of Multivariate Function.* M. Zibulevsky. 2013 (<http://www.youtube.com>)
- [A:14] *Introduction to the Lagrange Multiplier.* ediwm.com video (<http://www.noodle.com/learn/details/334954/introduction-to-the-lagrange-multiplier>)
- [A:15] *A brief introduction to Dynamic Programming (DP).* A. Kasibhatla. Nanocad Lab (http://nanocad.ee.ucla.edu/pub/Main/SnippetTutorial/Amar_DP_Intro.pdf)
- [A:16] *Financial ratios.* Wikipedia (http://en.wikipedia.org/wiki/Financial_ratio)
- [A:17] *Getting started in Technical Analysis* (§1 Charts: Forecasting Tool or Folklore?) J Schwager. John Wiley & Sons. 1999
- [A:18] *Getting started in Technical Analysis* (§4 Trading Ranges, Support & Resistance) J Schwager. John Wiley & Sons. 1999
- [A:19] *Options: a personal seminar* (§1 Options: An Introduction, What is an Option) S. Fullman, New York Institute of Finance. Simon Schuster. 1992
- [A:20] *Options: a personal seminar* (§2 Purchasing Options) S. Fullman New York Institute of Finance. Simon Schuster. 1992
- [A:21] *List of financial data feeds.* Wikipedia (http://en.wikipedia.org/wiki/List_of_financial_data_feeds)

Module 3

Mastering Scala Machine Learning

Advance your skills in efficient data analysis and data processing
using the powerful tools of Scala, Spark, and Hadoop

1

Exploratory Data Analysis

Before I dive into more complex methods to analyze your data later in the book, I would like to stop at basic data exploratory tasks on which almost all data scientists spend at least 80-90% of their productive time. The data preparation, cleansing, transforming, and joining the data alone is estimated to be a \$44 billion/year industry alone (*Data Preparation in the Big Data Era* by Federico Castanedo and *Best Practices for Data Integration*, O'Reilly Media, 2015). Given this fact, it is surprising that people only recently started spending more time on the science of developing best practices and establishing good habits, documentation, and teaching materials for the whole process of data preparation (*Beautiful Data: The Stories Behind Elegant Data Solutions*, edited by Toby Segaran and Jeff Hammerbacher, O'Reilly Media, 2009 and *Advanced Analytics with Spark: Patterns for Learning from Data at Scale* by Sandy Ryza et al., O'Reilly Media, 2015).

Few data scientists would agree on specific tools and techniques—and there are multiple ways to perform the exploratory data analysis, ranging from Unix command line to using very popular open source and commercial ETL and visualization tools. The focus of this chapter is how to use Scala and a laptop-based environment to benefit from techniques that are commonly referred as a functional paradigm of programming. As I will discuss, these techniques can be transferred to exploratory analysis over distributed system of machines using Hadoop/Spark.

What has functional programming to do with it? Spark was developed in Scala for a good reason. Many basic principles that lie at the foundation of functional programming, such as lazy evaluation, immutability, absence of side effects, list comprehensions, and monads go really well with processing data in distributed environments, specifically, when performing the data preparation and transformation tasks on big data. Thanks to abstractions, these techniques work well on a local workstation or a laptop. As mentioned earlier, this does not preclude us from processing very large datasets up to dozens of TBs on modern laptops connected to distributed clusters of storage/processing nodes. We can do it one topic or focus area at the time, but often we even do not have to sample or filter the dataset with proper partitioning. We will use Scala as our primary tool, but will resort to other tools if required.

While Scala is complete in the sense that everything that can be implemented in other languages can be implemented in Scala, Scala is fundamentally a high-level, or even a scripting, language. One does not have to deal with low-level details of data structures and algorithm implementations that in their majority have already been tested by a plethora of applications and time, in, say, Java or C++—even though Scala has its own collections and even some basic algorithm implementations today. Specifically, in this chapter, I'll be focusing on using Scala/Spark only for high-level tasks.

In this chapter, we will cover the following topics:

- Installing Scala
- Learning simple techniques for initial data exploration
- Learning how to downsample the original dataset for faster turnover
- Discussing the implementation of basic data transformation and aggregations in Scala
- Getting familiar with big data processing tools such as Spark and Spark Notebook
- Getting code for some basic visualization of datasets

Getting started with Scala

If you have already installed Scala, you can skip this paragraph. One can get the latest Scala download from <http://www.scala-lang.org/download/>. I used Scala version 2.11.7 on Mac OS X El Capitan 10.11.5. You can use any other version you like, but you might face some compatibility problems with other packages such as Spark, a common problem in open source software as the technology adoption usually lags by a few released versions.



In most cases, you should try to maintain precise match between the recommended versions as difference in versions can lead to obscure errors and a lengthy debugging process.

If you installed Scala correctly, after typing `scala`, you should see something similar to the following:

```
[akozlov@Alexanders-MacBook-Pro ~]$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

This is a Scala **read-evaluate-print-loop (REPL)** prompt. Although Scala programs can be compiled, the content of this chapter will be in REPL, as we are focusing on interactivity with, maybe, a few exceptions. The `:help` command provides a some utility commands available in REPL (note the colon at the start):

```
[15:05:20 2.6.0-cdh5.5.0 akozlov@Alexanders-MacBook-Pro chapter01(master)]$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_40).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :help
All commands can be abbreviated, e.g., :he instead of :help.
:edit <id>|<line>          edit history
:help [command]              print this summary or command-specific help
:history [num]                show the history (optional num is commands to show)
:h? <string>                 search the history
:imports [name name ...]     show import history, identifying sources of names
:implicits [-v]               show the implicits in scope
:javap <path|class>          disassemble a file or class name
:line <id>|<line>            place line(s) at the end of history
:load <path>                  interpret lines in a file
:paste [-raw] [path]          enter paste mode or paste a file
:power                        enable power user mode
:quit                         exit the interpreter
:replay [options]             reset the repl and replay all previous commands
:require <path>              add a jar to the classpath
:reset [options]              reset the repl to its initial state, forgetting all session entries
:save <path>                  save replayable session to a file
:sh <command line>            run a shell command (result is implicitly => List[String])
:settings <options>          update compiler options, if possible; see reset
:silent                       disable/enable automatic printing of results
:type [-v] <expr>            display the type of an expression without evaluating it
:kind [-v] <expr>             display the kind of expression's type
:warnings                      show the suppressed warnings from the most recent line which had any
```

Distinct values of a categorical field

Now, you have a dataset and a computer. For convenience, I have provided you a small anonymized and obfuscated sample of clickstream data with the book repository that you can get at <https://github.com/alexvk/ml-in-scala.git>. The file in the chapter01/data/clickstream directory contains lines with timestamp, session ID, and some additional event information such as URL, category information, and so on at the time of the call. The first thing one would do is apply transformations to find out the distribution of values for different columns in the dataset.

Figure 01-1 shows screenshot shows the output of the dataset in the terminal window of the `gzcat chapter01/data/clickstream/clickstream_sample.tsv.gz | less -U` command. The columns are tab (^I) separated. One can notice that, as in many real-world big data datasets, many values are missing. The first column of the dataset is recognizable as the timestamp. The file contains complex data such as arrays, structs, and maps, another feature of big data datasets.

Unix provides a few tools to dissect the datasets. Probably, `less`, `cut`, `sort`, and `uniq` are the most frequently used tools for text file manipulations. `Awk`, `sed`, `perl`, and `tr` can do more complex transformations and substitutions. Fortunately, Scala allows you to transparently use command-line tools from within Scala REPL, as shown in the following screenshot:

Figure 01-1. The clickstream file as an output of the less -U Unix command

Fortunately, Scala allows you to transparently use command-line tools from within Scala REPL:

```
[akozlov@Alexanders-MacBook-Pro] $ scala  
...  
scala> import scala.sys.process._  
import scala.sys.process.
```

```
scala> val histogram = ( "gzcat chapter01/data/clickstream/clickstream_
sample.tsv.gz" #| "cut -f 10" #| "sort" #| "uniq -c" #| "sort -k1nr"
).lineStream

histogram: Stream[String] = Stream(7731 http://www.mycompany.com/us/en_us/, ?)

scala> histogram take(10) foreach println
7731 http://www.mycompany.com/us/en_us/
3843 http://mycompanyplus.mycompany.com/plus/
2734 http://store.mycompany.com/us/en_us/?l=shop,men_shoes
2400 http://m.mycompany.com/us/en_us/
1750 http://store.mycompany.com/us/en_us/?l=shop,men_mycompanyid
1556 http://www.mycompany.com/us/en_us/c/mycompanyid?sitesrc=id_redir
1530 http://store.mycompany.com/us/en_us/
1393 http://www.mycompany.com/us/en_us/?cp=USNS_KW_0611081618
1379 http://m.mycompany.com/us/en_us/?ref=http%3A%2F%2Fwww.mycompany.
com%2F
1230 http://www.mycompany.com/us/en_us/c/running
```

I used the `scala.sys.process` package to call familiar Unix commands from Scala REPL. From the output, we can immediately see the customers of our Webshop are mostly interested in men's shoes and running, and that most visitors are using the referral code, **KW_0611081618**.

One may wonder when we start using complex Scala types and algorithms. Just wait, a lot of highly optimized tools were created before Scala and are much more efficient for explorative data analysis. In the initial stage, the biggest bottleneck is usually just the disk I/O and slow interactivity. Later, we will discuss more iterative algorithms, which are usually more memory intensive. Also note that the UNIX pipeline operations can be implicitly parallelized on modern multi-core computer architectures, as they are in Spark (we will show it in the later chapters).



It has been shown that using compression, implicit or explicit, on input data files can actually save you the I/O time. This is particularly true for (most) modern semi-structured datasets with repetitive values and sparse content. Decompression can also be implicitly parallelized on modern fast multi-core computer architectures, removing the computational bottleneck, except, maybe in cases where compression is implemented implicitly in hardware (SSD, where we don't need to compress the files explicitly). We also recommend using directories rather than files as a paradigm for the dataset, where the insert operation is reduced to dropping the data file into a directory. This is how the datasets are presented in big data Hadoop tools such as Hive and Impala.

Summarization of a numeric field

Let's look at the numeric data, even though most of the columns in the dataset are either categorical or complex. The traditional way to summarize the numeric data is a five-number-summary, which is a representation of the median or mean, interquartile range, and minimum and maximum. I'll leave the computations of the median and interquartile ranges till the Spark DataFrame is introduced, as it makes these computations extremely easy; but we can compute mean, min, and max in Scala by just applying the corresponding operators:

```
scala> import scala.sys.process._
import scala.sys.process._

scala> val nums = ("gzcat chapter01/data/clickstream/clickstream_sample.
tsv.gz" #| "cut -f 6").lineStream
nums: Stream[String] = Stream(0, ?)

scala> val m = nums.map(_.toDouble).min
m: Double = 0.0

scala> val m = nums.map(_.toDouble).sum/nums.size
m: Double = 3.6883642764024662

scala> val m = nums.map(_.toDouble).max
m: Double = 33.0
```

Grepping across multiple fields

Sometimes one needs to get an idea of how a certain value looks across multiple fields—most common are IP/MAC addresses, dates, and formatted messages. For examples, if I want to see all IP addresses mentioned throughout a file or a document, I need to replace the `cut` command in the previous example by `grep -o -E [1-9] [0-9] {0,2}(:\.\. [1-9] [0-9] {0,2}){3}`, where the `-o` option instructs `grep` to print only the matching parts—a more precise regex for the IP address should be `grep -o -E (:(:25 [0-5] | 2 [0-4] [0-9] | [01]? [0-9] [0-9]?)\.){3} (:25 [0-5] | 2 [0-4] [0-9] | [01]? [0-9] [0-9]?)`, but is about 50% slower on my laptop and the original one works in most practical cases. I'll leave it as an excursive to run this command on the sample file provided with the book.

Basic, stratified, and consistent sampling

I've met quite a few data practitioners who scorn sampling. Ideally, if one can process the whole dataset, the model can only improve. In practice, the tradeoff is much more complex. First, one can build more complex models on a sampled set, particularly if the time complexity of the model building is non-linear – and in most situations, if it is at least $N^* \log(N)$. A faster model building cycle allows you to iterate over models and converge on the best approach faster. In many situations, *time to action* is beating the potential improvements in the prediction accuracy due to a model built on complete dataset.

Sampling may be combined with appropriate filtering – in many practical situation, focusing on a subproblem at a time leads to better understanding of the whole problem domain. In many cases, this partitioning is at the foundation of the algorithm, like in decision trees, which are considered later. Often the nature of the problem requires you to focus on the subset of original data. For example, a cyber security analysis is often focused around a specific set of IPs rather than the whole network, as it allows to iterate over hypothesis faster. Including the set of all IPs in the network may complicate things initially if not throw the modeling off the right track.

When dealing with rare events, such as clickthroughs in ADTECH, sampling the positive and negative cases with different probabilities, which is also sometimes called oversampling, often leads to better predictions in short amount of time.

Fundamentally, sampling is equivalent to just throwing a coin – or calling a random number generator – for each data row. Thus it is very much like a stream filter operation, where the filtering is on an augmented column of random numbers. Let's consider the following example:

```
import scala.util.Random
import util.Properties

val threshold = 0.05

val lines = scala.io.Source.fromFile("chapter01/data/iris/in.txt") .
getLines
val newLines = lines.filter(_ =>
    Random.nextDouble() <= threshold
)

val w = new java.io.FileWriter(new java.io.File("out.txt"))
newLines.foreach { s =>
    w.write(s + Properties.lineSeparator)
}
w.close
```

This is all good, but it has the following disadvantages:

- The number of lines in the resulting file is not known beforehand – even though on average it should be 5% of the original file
- The results of the sampling is non-deterministic – it is hard to rerun this process for either testing or verification

To fix the first point, we'll need to pass a more complex object to the function, as we need to maintain the state during the original list traversal, which makes the original algorithm less functional and parallelizable (this will be discussed later):

```
import scala.reflect.ClassTag
import scala.util.Random
import util.Properties

def reservoirSample[T: ClassTag](input: Iterator[T], k: Int): Array[T] =
{
    val reservoir = new Array[T](k)
    // Put the first k elements in the reservoir.
    var i = 0
    while (i < k && input.hasNext) {
        val item = input.next()
        reservoir(i) = item
        i += 1
    }

    if (i < k) {
        // If input size < k, trim the array size
        reservoir.take(i)
    } else {
        // If input size > k, continue the sampling process.
        while (input.hasNext) {
            val item = input.next()
            val replacementIndex = Random.nextInt(i)
            if (replacementIndex < k) {
                reservoir(replacementIndex) = item
            }
            i += 1
        }
        reservoir
    }
}

val numLines=15
```

```
val w = new java.io.FileWriter(new java.io.File("out.txt"))
val lines = io.Source.fromFile("chapter01/data/iris/in.txt").getLines
reservoirSample(lines, numLines).foreach { s =>
    w.write(s + scala.util.Properties.lineSeparator)
}
w.close
```

This will output `numLines` lines. Similarly to reservoir sampling, stratified sampling is guaranteed to provide the same ratios of input/output rows for all strata defined by levels of another attribute. We can achieve this by splitting the original dataset into N subsets corresponding to the levels, performing the reservoir sampling, and merging the results afterwards. However, MLlib library, which will be covered in *Chapter 3, Working with Spark and MLlib*, already has stratified sampling implementation:

```
val origLinesRdd = sc.textFile("file://...")
val keyedRdd = origLines.keyBy(r => r.split(",")(0))
val fractions = keyedRdd.countByKey.keys.map(r => (r, 0.1)).toMap
val sampledWithKey = keyedRdd.sampleByKeyExact(fractions)
val sampled = sampledWithKey.map(_._2).collect
```

The other bullet point is more subtle; sometimes we want a consistent subset of values across multiple datasets, either for reproducibility or to join with another sampled dataset. In general, if we sample two datasets, the results will contain random subsets of IDs which might have very little or no intersection. The cryptographic hashing functions come to the help here. The result of applying a hash function such as MD5 or SHA1 is a sequence of bits that is statistically uncorrelated, at least in theory. We will use the `MurmurHash` function, which is part of the `scala.util.hashing` package:

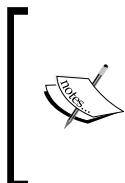
```
import scala.util.hashing.MurmurHash._

val markLow = 0
val markHigh = 4096
val seed = 12345

def consistentFilter(s: String): Boolean = {
    val hash = stringHash(s.split(" ")(0), seed) >>> 16
    hash >= markLow && hash < markHigh
}

val w = new java.io.FileWriter(new java.io.File("out.txt"))
val lines = io.Source.fromFile("chapter01/data/iris/in.txt").getLines
lines.filter(consistentFilter).foreach { s =>
    w.write(s + Properties.lineSeparator)
}
w.close
```

This function is guaranteed to return exactly the same subset of records based on the value of the first field—it is either all records where the first field equals a certain value or none—and will come up with approximately one-sixteenth of the original sample; the range of hash is 0 to 65,535.



MurmurHash? It is not a cryptographic hash!

Unlike cryptographic hash functions, such as MD5 and SHA1, MurmurHash is not specifically designed to be hard to find an inverse of a hash. It is, however, really fast and efficient. This is what really matters in our use case.

Working with Scala and Spark Notebooks

Often the most frequent values or five-number summary are not sufficient to get the first understanding of the data. The term **descriptive statistics** is very generic and may refer to very complex ways to describe the data. Quantiles, a **Pareto** chart or, when more than one attribute is analyzed, correlations are also examples of descriptive statistics. When sharing all these ways to look at the data aggregates, in many cases, it is also important to share the specific computations to get to them.

Scala or Spark Notebook <https://github.com/Bridgewater/scala-notebook>, <https://github.com/andypetrella/spark-notebook> record the whole transformation path and the results can be shared as a JSON-based *.snb file. The Spark Notebook project can be downloaded from <http://spark-notebook.io>, and I will provide a sample Chapter01.snb file with the book. I will use Spark, which I will cover in more detail in *Chapter 3, Working with Spark and MLlib*.

For this particular example, Spark will run in the local mode. Even in the local mode Spark can utilize parallelism on your workstation, but it is limited to the number of cores and hyperthreads that can run on your laptop or workstation. With a simple configuration change, however, Spark can be pointed to a distributed set of machines and use resources across a distributed set of nodes.

Here is the set of commands to download the Spark Notebook and copy the necessary files from the code repository:

```
[akozlov@Alexanders-MacBook-Pro]$ wget http://s3.eu-central-1.amazonaws.com/spark-notebook/zip/spark-notebook-0.6.3-scala-2.11.7-spark-1.6.1-hadoop-2.6.4-with-hive-with-parquet.zip
...
[akozlov@Alexanders-MacBook-Pro]$ unzip -d ~/ spark-notebook-0.6.3-scala-2.11.7-spark-1.6.1-hadoop-2.6.4-with-hive-with-parquet.zip
...
```

Exploratory Data Analysis

```
[akozlov@Alexanders-MacBook-Pro] $ ln -sf ~/spark-notebook-0.6.3-scala-2.11.7-spark-1.6.1-hadoop-2.6.4-with-hive-with-parquet ~/spark-notebook  
[akozlov@Alexanders-MacBook-Pro] $ cp chapter01/notebook/Chapter01.snb ~/spark-notebook/notebooks  
[akozlov@Alexanders-MacBook-Pro] $ cp chapter01/ data/kddcup/kddcup.parquet ~/spark-notebook  
[akozlov@Alexanders-MacBook-Pro] $ cd ~/spark-notebook  
[akozlov@Alexanders-MacBook-Pro] $ bin/spark-notebook  
Play server process ID is 2703  
16/04/14 10:43:35 INFO play: Application started (Prod)  
16/04/14 10:43:35 INFO play: Listening for HTTP on /0:0:0:0:0:0:0:9000  
...
```

Now you can open the notebook at <http://localhost:9000> in your browser, as shown in the following screenshot:

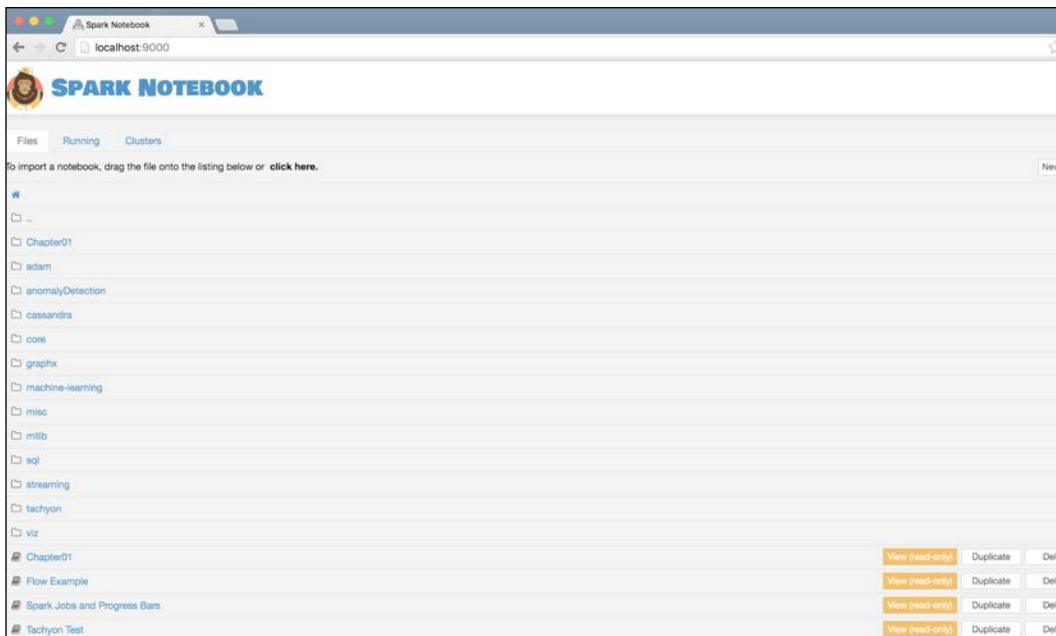


Figure 01-2. The first page of the Spark Notebook with the list of notebooks

Open the Chapter01 notebook by clicking on it. The statements are organized into cells and can be executed by clicking on the small right arrow at the top, as shown in the following screenshot, or run all cells at once by navigating to **Cell | Run All**:

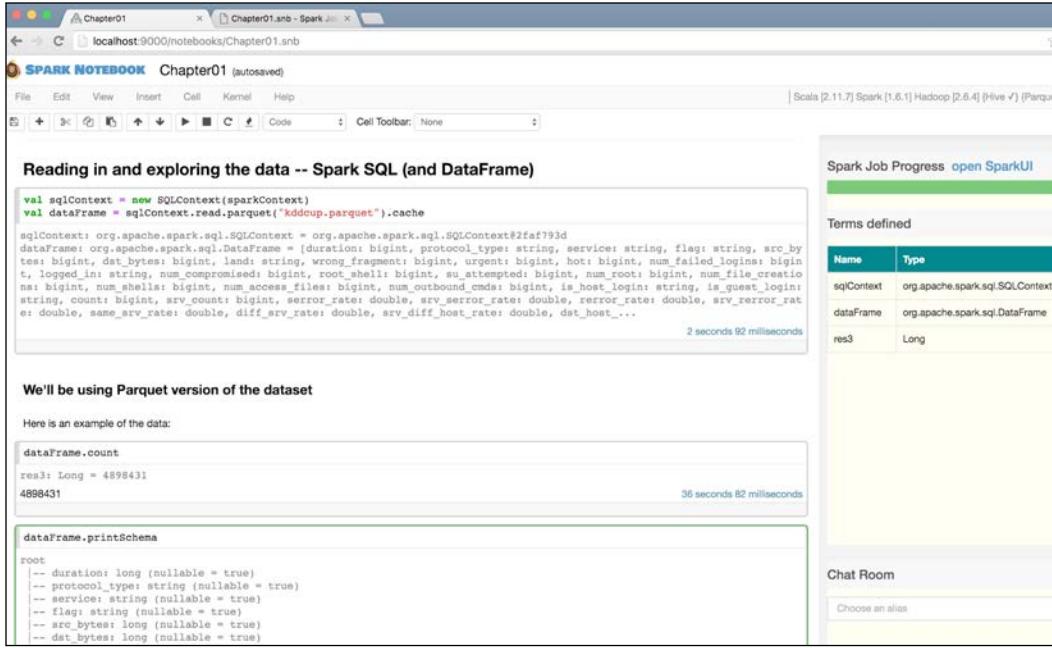


Figure 01-3. Executing the first few cells in the notebook

First, we will look at the values of all or some of discrete variables. For example, to get the distribution of the labels, issue the following code:

```
val labelCount = df.groupBy("lbl").count().collect
labelCount.toList.map(row => (row.getString(0), row.getLong(1)))
```

Exploratory Data Analysis

The first time I read the dataset, it took about a minute on MacBook Pro, but Spark caches the data in memory and the subsequent aggregation runs take only about a second. Spark Notebook provides you the distribution of the values, as shown in the following screenshot:

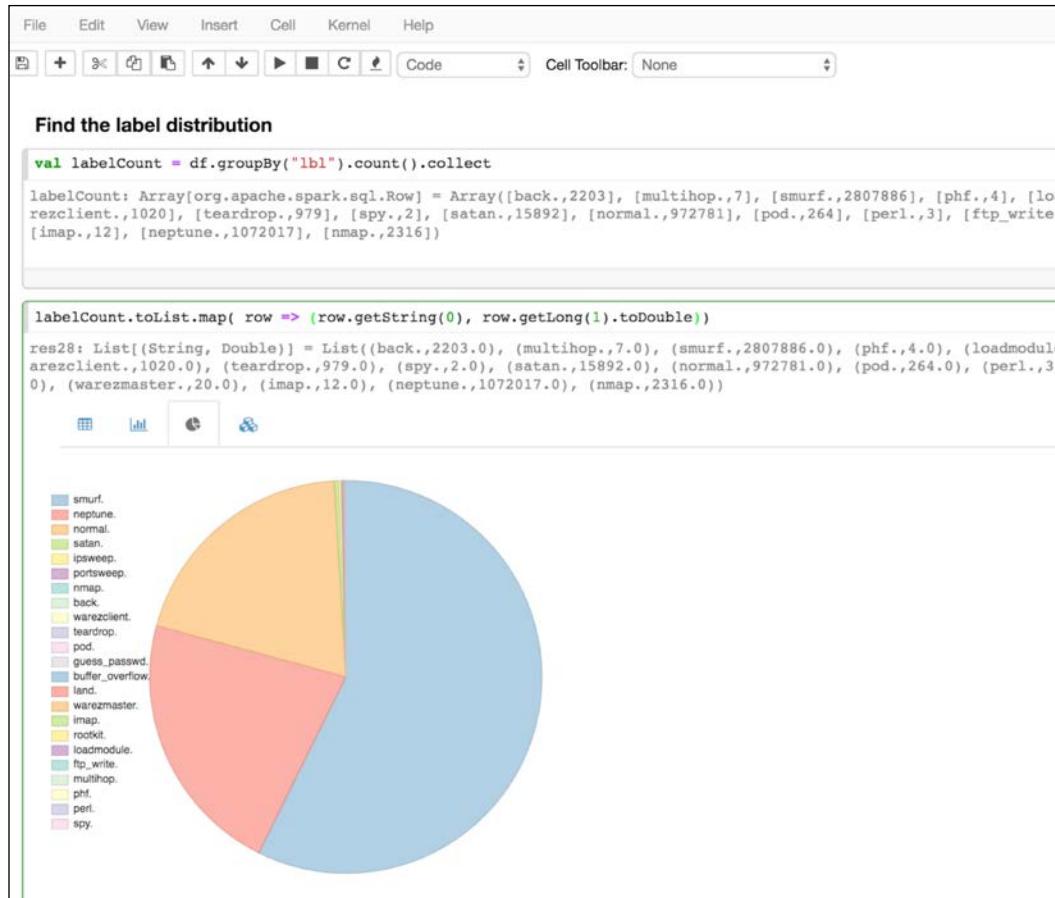


Figure 01-4. Computing the distribution of values for a categorical field

I can also look at crosstab counts for pairs of discrete variables, which gives me an idea of interdependencies between the variables using <http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameStatFunctions> – the object does not support computing correlation measures such as chi-square yet:

In [63]: <code>dataFrame.stat.crosstab("service", "flag")</code>												
Out[63]:												
service_flag	S0	RSTO	RSTR	RSTOS0	SF	SH	REJ	S1	OTH	S2	S3	
ftp	843	234	6	2	4115	1	0	10	2	1	0	
netbios_ssn	842	1	6	0	3	1	202	0	0	0	0	
hostnames	837	0	6	0	0	1	206	0	0	0	0	
printer	834	202	5	0	2	1	0	1	0	0	0	
finger	1634	212	7	2	5031	1	0	3	0	0	1	
smtp	1008	349	9	2	95111	1	4	37	2	21	10	
harvest	1	0	0	0	0	0	1	0	0	0	0	
aol	0	0	0	0	0	0	2	0	0	0	0	
name	837	0	8	1	0	1	220	0	0	0	0	
whois	843	0	8	1	0	1	220	0	0	0	0	
http_8001	1	0	0	0	0	0	1	0	0	0	0	
private	820049	1203	4703	91	76524	981	197246	1	33	0	0	
sql_net	839	0	6	0	0	1	205	0	1	0	0	
shell	834	203	5	0	7	1	0	1	0	0	0	
ftp_data	1611	0	9	1	38743	1	238	72	3	6	13	
auth	837	4	6	0	2314	1	220	0	0	0	0	
ssh	840	16	6	1	9	1	202	0	0	0	0	
telnet	1730	315	43	2	2106	1	0	73	3	0	4	
gopher	842	3	6	1	14	1	210	0	0	0	0	
pop_2	843	1	5	0	2	1	203	0	0	0	0	
domain	848	4	6	1	48	1	205	0	0	0	0	
pm_dump	0	0	0	0	5	0	0	0	0	0	0	
supdup	846	0	7	0	0	1	206	0	0	0	0	
netbios_dgm	839	0	7	0	0	1	205	0	0	0	0	
discard	841	202	8	2	1	1	4	0	0	0	0	

Figure 01-5. Contingency table or crosstab

However, we can see that the most popular service is private and it correlates well with the SF flag. Another way to analyze dependencies is to look at 0 entries. For example, the S2 and S3 flags are clearly related to the SMTP and FTP traffic since all other entries are 0.

Of course, the most interesting correlations are with the target variable, but these are better discovered by supervised learning algorithms that I will cover in *Chapter 3, Working with Spark and MLlib*, and *Chapter 5, Regression and Classification*.

Correlations

Pearson Correlation Coefficient of two columns

```
sampled.stat.corr("src_bytes", "dst_bytes")
res9: Double = 0.23256972813705676
0.23256972813705676
```

Covariance and variance

```
sampled.stat.cov("src_bytes", "dst_bytes")
res15: Double = 4.7960500298884094E8
4.7960500298884094E8
```

```
sampled.stat.cov("src_bytes", "src_bytes")
res17: Double = 6.37408697211937E9
6.37408697211937E9
```

```
sampled.stat.cov("dst_bytes", "dst_bytes")
res19: Double = 6.671800540336397E8
6.671800540336397E8
```

Figure 01-6. Computing simple aggregations using org.apache.spark.sql.DataFrameStatFunctions.

Analogously, we can compute correlations for numerical variables with the `dataFrame.stat.corr()` and `dataFrame.stat.cov()` functions (refer to *Figure 01-6*). In this case, the class supports the **Pearson correlation coefficient**. Alternatively, we can use the standard SQL syntax on the parquet file directly:

```
sqlContext.sql("SELECT lbl, protocol_type, min(duration),
    avg(duration), stddev(duration), max(duration) FROM
    parquet.`kddcup.parquet` group by lbl, protocol_type")
```

Finally, I promised you to compute percentiles. Computing percentiles usually involves sorting the whole dataset, which is expensive; however, if the tile is one of the first or the last ones, usually it is possible to optimize the computation:

```
val pct = sqlContext.sql("SELECT duration FROM
    parquet.`kddcup.parquet` WHERE protocol_type =
    'udp'").rdd.map(_.getLong(0)).cache
pct.top((0.05*pct.count).toInt).last
```

Computing the exact percentiles for a more generic case is more computationally expensive and is provided as a part of the Spark Notebook example code.

Basic correlations

You probably noticed that detecting correlations from contingency tables is hard. Detecting patterns takes practice, but many people are much better at recognizing the patterns visually. Detecting actionable patterns is one of the primary goals of machine learning. While advanced supervised machine learning techniques that will be covered in *Chapter 4, Supervised and Unsupervised Learning* and *Chapter 5, Regression and Classification* exist, initial analysis of interdependencies between variables can help with the right transformation of variables or selection of the best inference technique.

Multiple well-established visualization tools exist and there are multiple sites, such as <http://www.kdnuggets.com>, which specialize on ranking and providing recommendations on data analysis, data explorations, and visualization software. I am not going to question the validity and accuracy of such rankings in this book, and very few sites actually mention Scala as a specific way to visualize the data, even if this is possible with, say, a D3.js package. A good visualization is a great way to deliver your findings to a larger audience. One look is worth a thousand words.

For the purposes of this chapter, I will use **Grapher** that is present on every Mac OS notebook. To open **Grapher**, go to Utilities (*shift + command + U* in Finder) and click on the **Grapher** icon (or search by name by pressing *command + space*). Grapher presents many options, including the following **Log-Log** and **Polar** coordinates:

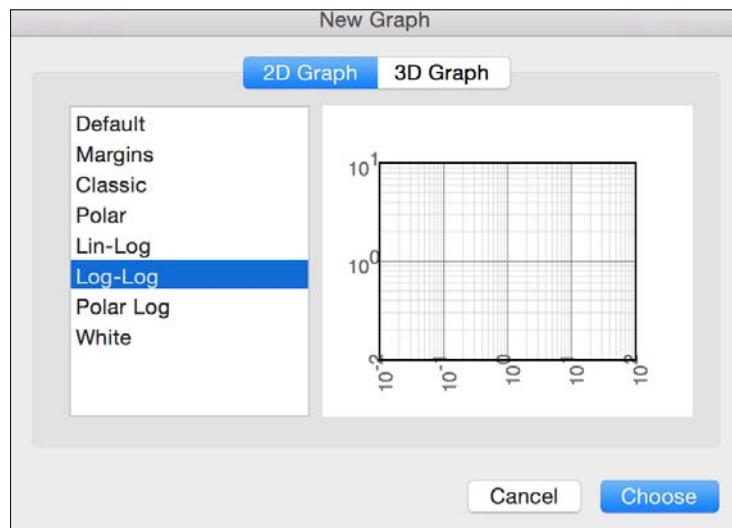


Figure 01-7. The Grapher window

Fundamentally, the amount of information that can be delivered through visualization is limited by the number of pixels on the screen, which, for most modern computers, is in millions and color variations, which arguably can also be in millions (Judd, Deane B.; Wyszecki, Günter (1975). *Color in Business, Science and Industry. Wiley Series in Pure and Applied Optics* (3rd ed.). New York). If I am working on a multidimensional TB dataset, the dataset first needs to be summarized, processed, and reduced to a size that can be viewed on a computer screen.

For the purpose of illustration, I will use the Iris UCI dataset that can be found at <https://archive.ics.uci.edu/ml/datasets/Iris>. To bring the dataset into the tool, type the following code (on Mac OS):

```
[akozlov@Alexander-MacBook-Pro] $ pbcopy < chapter01/data/iris/in.txt
```

Open the new **Point Set** in the **Grapher** (*command + alt + P*), press **Edit Points...** and paste the data by pressing *command + V*. The tools has line-fitting capabilities with basic linear, polynomial, and exponential families and provides the popular chi-squared metric to estimate the goodness of the fit with respect to the number of free parameters:

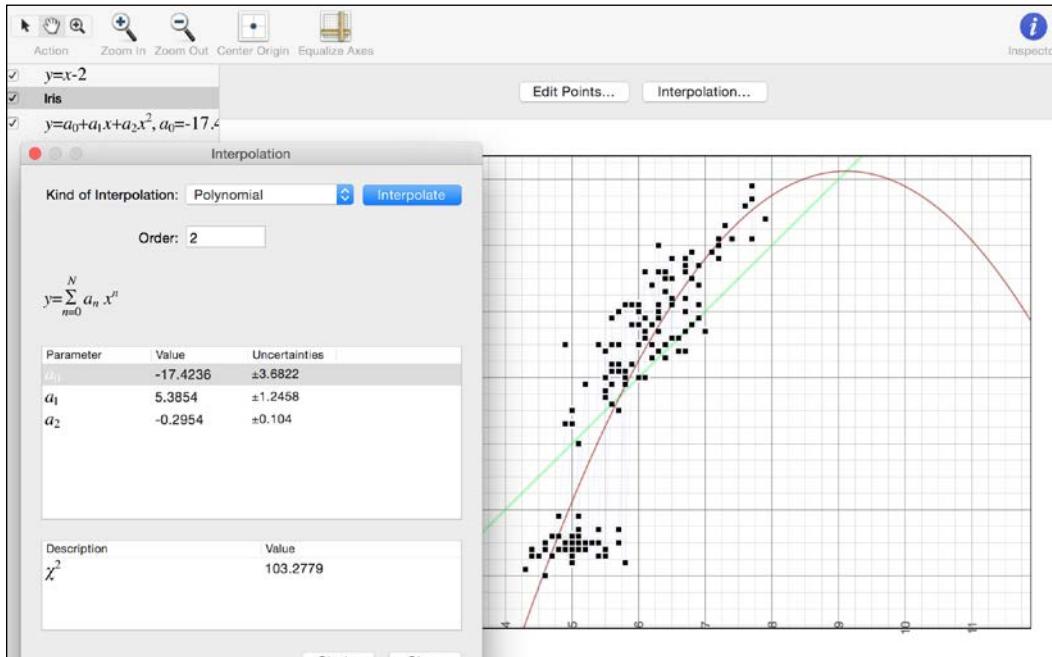


Figure 01-8. Fitting the Iris dataset using Grapher on Mac OS X

We will cover how to estimate the goodness of model fit in the following chapters.

Summary

I've tried to establish a common ground to perform a more complex data science later in the book. Don't expect these to be a complete set of exploratory techniques, as the exploratory techniques can extend to running very complex modes. However, we covered simple aggregations, sampling, file operations such as read and write, working with tools such as notebooks and Spark DataFrames, which brings familiar SQL constructs into the arsenal of an analyst working with Spark/Scala.

The next chapter will take a completely different turn by looking at the data pipelines as a part of a data-driven enterprise and cover the data discovery process from the business perspective: what are the ultimate goals we are trying to accomplish by doing the data analysis. I will cover a few traditional topics of ML, such as supervised and unsupervised learning, after this before delving into more complex representations of the data, where Scala really shows its advantage over SQL.

2

Data Pipelines and Modeling

We have looked at basic hands-on tools for exploring the data in the previous chapter, thus we now can delve into more complex topics of statistical model building and optimal control or science-driven tools and problems. I will go ahead and say that we will only touch on some topics in optimal control since this book really is just about ML in Scala and not the theory of data-driven business management, which might be an exciting topic for a book on its own.

In this chapter, I will stay away from specific implementations in Scala and discuss the problem of building a data-driven enterprise at a high level. Later chapters will address how to solve these smaller pieces of the puzzle. A special emphasis will be given to handling uncertainty. Uncertainty usually comes in several flavors: first, there can be noise in the information we are provided with. Secondly, the information can be incomplete. The system may have some degree of freedom in filling the missing pieces, which results in uncertainty. Finally, there may be variations in the interpretation of the models and the resulting metrics. The final point is subtle, as most classic textbooks assume that we can measure things directly. Not only the measurements may be noisy, but the definition of the measure may change in time – try measuring satisfaction or happiness. Certainly, we can avoid the ambiguity by saying that we can optimize only measurable metrics, as people usually do, but it will significantly limit the application domain in practice. Nothing prevents the scientific machinery from handling the uncertainty in the interpretation into account as well.

The predictive models are often built just for data understanding. From the linguistic derivation, model is a simplified representation of the actual complex buildings or processes for exactly the purpose of making a point and convincing people, one or another way. The ultimate goal for predictive modeling, the modeling I am concerned about in this book and this chapter specifically, is to optimize the business processes by taking the most important factors into account in order to make the world a better place. This was certainly a sentence with a lot of uncertainty entrenched, but at least it looks like a much better goal than optimizing a click-through rate.

Let's look at a traditional business decision-making process: a traditional business might involve a set of C-level executives making decisions based on information that is usually obtained from a set of dashboards with graphical representation of the data in one or several DBs. The promise of an automated data-driven business is to be able to automatically make most of the decisions provided the uncertainties eliminating human bias. This is not to say that we no longer need C-level executives, but the C-level executives will be busy helping the machines to make the decisions instead of the other way around.

In this chapter, we will cover the following topics:

- Going through the basics of influence diagrams as a tool for decision making
- Looking at variations of the pure decision making optimization in the context of adaptive **Markov Decision** making process and **Kelly Criterion**
- Getting familiar with at least three different practical strategies for exploration-exploitation trade-off
- Describing the architecture of a data-driven enterprise
- Discussing major architectural components of a decision-making pipeline
- Getting familiar with standard tools for building data pipelines

Influence diagrams

While the decision making process can have multiple facets, a book about decision making under uncertainty would be incomplete without mentioning influence diagrams (*Influence Diagrams for Team Decision Analysis*, Decision Analysis 2 (4): 207–228), which help the analysis and understanding of the decision-making process. The decision may be as mundane as selection of the next news article to show to a user in a personalized environment or a complex one as detecting malware on an enterprise network or selecting the next research project.

Depending on the weather she can try and go on a boat trip. We can represent the decision-making process as a diagram. Let's decide whether to take a river boat tour during her stay in Portland, Oregon:

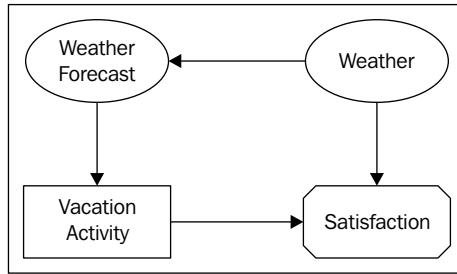


Figure 02-1. A simple vacation influence diagram to represent a simple decision-making process. The diagram contains decision nodes such as **Vacation Activity**, observable and unobservable information nodes such as **Weather Forecast** and **Weather**, and finally the value node such as **Satisfaction**

The preceding diagram represents this situation. The decision whether to participate in the activity is clearly driven by the potential to get certain satisfaction, which is a function of the decision itself and the weather at the time of the activity. While the actual weather conditions are unknown at the time of the trip planning, we believe there is a certain correlation between the weather forecast and the actual weather experienced during the trip, which is represented by the edge between the **Weather** and **Weather Forecast** nodes. The **Vacation Activity** node is the decision node, it has only one parent as the decision is made solely based on **Weather Forecast**. The final node in the DAG is **Satisfaction**, which is a function of the actual whether and the decision we made during the trip planning – obviously, *yes + good weather* and *no + bad weather* are likely to have the highest scores. The *yes + bad weather* and *no + good weather* would be a bad outcome – the latter case is probably just a missed opportunity, but not necessarily a bad decision, provided an inaccurate weather forecast.

The absence of an edge carries an independence assumption. For example, we believe that **Satisfaction** should not depend on **Weather Forecast**, as the latter becomes irrelevant once we are on the boat. Once the vacation plan is finalized, the actual weather during the boating activity can no longer affect the decision, which was made solely based on the weather forecast; at least in our simplified model, where we exclude the option of buying a trip insurance.

The graph shows different stages of decision making and the flow of information (we will provide an actual graph implementation in Scala in *Chapter 7, Working with Graph Algorithms*). There is only one piece of information required to make the decision in our simplified diagram: the weather forecast. Once the decision is made, we can no longer change it, even if we have information about the actual weather at the time of the trip. The weather and the decision data can be used to model her satisfaction with the decision she has made.

Let's map this approach to an advertising problem as an illustration: the ultimate goal is to get user satisfaction with the targeted ads, which results in additional revenue for an advertiser. The satisfaction is the function of user-specific environmental state, which is unknown at the time of decision making. Using machine learning algorithms, however, we can forecast this state based on the user's recent Web visit history and other information that we can gather, such as geolocation, browser-agent string, time of day, category of the ad, and so on (refer to *Figure 02-2*).

While we are unlikely to measure the level of dopamine in the user's brain, which will certainly fall under the realm of measurable metrics and probably reduce the uncertainty, we can measure the user satisfaction indirectly by the user's actions, either the fact that they responded to the ad or even the measure of time the user spent between the clicks to browse relevant information, which can be used to estimate the effectiveness of our modeling and algorithms. Here is an influence diagram, similar to the one for "vacation", adjusted for the advertising decision-making process:

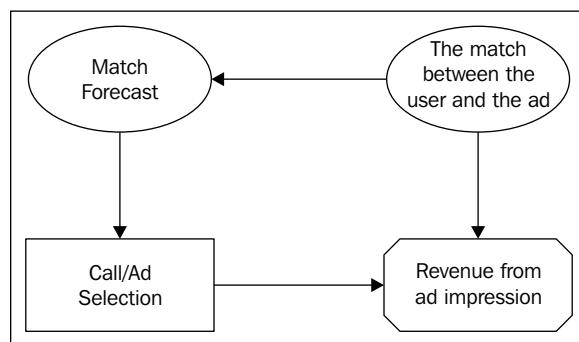


Figure 02-2. The vacation influence diagram adjusted to the online advertising decision-making case.
The decisions for online advertising can be made thousand times per second

The actual process might be more complex, representing a chain of decisions, each one depending on a few previous time slices. For example, the so-called **Markov Chain Decision Process**. In this case, the diagram might be repeated over multiple time slices.

Yet another example might be Enterprise Network Internet malware analytics system. In this case, we try to detect network connections indicative of either **command and control (C2)**, lateral movement, or data exfiltration based on the analysis of network packets flowing through the enterprise switches. The goal is to minimize the potential impact of an outbreak with minimum impact on the functioning systems.

One of the decisions we might take is to reimage a subset of nodes or to at least isolate them. The data we collect may contain uncertainty – many benign software packages may send traffic in suspicious ways, and the models need to differentiate between them based on the risk and potential impact. One of the decisions in this specific case may be to collect additional information.

I will leave it to the reader to map this and other potential business cases to the corresponding diagram as an exercise. Let's consider a more complex optimization problem now.

Sequential trials and dealing with risk

What if my preferences for making an extra few dollars outweigh the risk of losing the same amount? I will stop on why one's preferences might be asymmetric in a little while in this section, and there is scientific evidence that this asymmetry is ingrained in our minds for evolutionary reasons, but you are right, I have to optimize the expected value of the asymmetric function of the parameterized utility now, as follows:

$$\frac{\partial E(F(u(z)))}{\partial z}. \quad (2.1)$$

Why would an asymmetric function surface in the analysis? One example is repeated bets or re-investments, also known as the Kelly Criterion problem. Although originally, the Kelly Criterion was developed for a specific case of binary outcome as in a gambling machine and the optimization of the fraction of money to bet in each round (*A New Interpretation of Information Rate*, Bell System Technical Journal 35 (4): 917–926, 1956), a more generic formulation as an re-investment problem involves a probabilistic distribution of possible returns.

The return over multiple bets is a product of individual return rates on each of the bets – the return rate is the ratio between the bankroll after the bet to the original bankroll before each individual bet, as follows:

$$R = r_1 r_2 \dots r_N$$

This does not help us much to optimize the total return as we don't know how to optimize the product of *i.i.d.* random variables. However, we can convert the product to a sum using log transformation and apply the **central limit theorem** (CLT) to approximate the sum of *i.i.d.* variables (provided that the distribution of r_i is subject to CLT conditions, for example, has a finite mean and variance), as follows:

$$\begin{aligned} E(R) &= \\ E\left(\exp\left(\log(r_1) + \log(r_2) + \dots + \log(r_N)\right)\right) &= \\ \exp\left(N \times E(\log(r)) + O(\sqrt{N})\right) &= \\ \exp\left(E(\log(r))\right)^N (1 + O\left(\frac{1}{\sqrt{N}}\right)) \end{aligned}$$

Thus, the cumulative result of making N bets would look like the result of making N bets with expected return of $\exp(E(\log(r)))$, and not $E(r)$!

As I mentioned before, the problem is most often applied for the case of binary bidding, although it can be easily generalized, in which case there is an additional parameter: x , the amount of money to bid in each round. Let's say I make a profit of W with probability p or completely lose my bet otherwise with the probability $(1-p)$. Optimizing the expected return with respect to the following additional parameter:

$$E(\log(r(x))) = p \log(1+xW) + (1-p) \log(1-x) \quad (2.2)$$

$$\frac{\partial E(\log(r(x)))}{\partial x} = \frac{pW}{1+xW} - \frac{(1-p)}{1-x} = 0 \quad (2.3)$$

$$x = p - \left[\frac{1-p}{W} \right] \quad (2.4)$$

The last equation is the Kelly Criterion ratio and gives you the optimal amount to bet.

The reason that one might bet less than the total amount is that even if the average return is positive, there is still a possibility to lose the whole bankroll, particularly, in highly skewed situations. For example, even if the probability of making $10x$ on your bet is 0.105 ($W = 10$, the expected return is 5%), the combinatorial analysis show that even after 60 bets, there is roughly a 50% chance that the overall return will be negative, and there is an 11% chance, in particular, of losing $(57 - 10 \times 3) = 27$ times your bet or more:

```
akozlov@Alexanders-MacBook-Pro$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40).
Type in expressions to have them evaluated.
Type :help for more information.27

scala> def logFactorial(n: Int) = { (1 to n).map(Math.log(_)).sum }
logFactorial: (n: Int)Double

scala> def cmnp(m: Int, n: Int, p: Double) = {
    |   Math.exp(logFactorial(n) -
    |   logFactorial(m) +
    |   m*Math.log(p) -
    |   logFactorial(n-m) +
    |   (n-m)*Math.log(1-p))
    | }
cmnp: (m: Int, n: Int, p: Double)Double

scala> val p = 0.105
p: Double = 0.105

scala> val n = 60
n: Int = 60

scala> var cumulative = 0.0
cumulative: Double = 0.0

scala> for(i <- 0 to 14) {
    |   val prob = cmnp(i,n,p)
    |   cumulative += prob
```

```
|     println(f"We expect $i wins with ${prob:.6f} probability  
$cumulative:.3f cumulative (n = $n, p = $p).")  
| }  
We expect 0 wins with 0.001286 probability 0.001 cumulative (n = 60, p =  
0.105).  
We expect 1 wins with 0.009055 probability 0.010 cumulative (n = 60, p =  
0.105).  
We expect 2 wins with 0.031339 probability 0.042 cumulative (n = 60, p =  
0.105).  
We expect 3 wins with 0.071082 probability 0.113 cumulative (n = 60, p =  
0.105).  
We expect 4 wins with 0.118834 probability 0.232 cumulative (n = 60, p =  
0.105).  
We expect 5 wins with 0.156144 probability 0.388 cumulative (n = 60, p =  
0.105).  
We expect 6 wins with 0.167921 probability 0.556 cumulative (n = 60, p =  
0.105).  
We expect 7 wins with 0.151973 probability 0.708 cumulative (n = 60, p =  
0.105).  
We expect 8 wins with 0.118119 probability 0.826 cumulative (n = 60, p =  
0.105).  
We expect 9 wins with 0.080065 probability 0.906 cumulative (n = 60, p =  
0.105).  
We expect 10 wins with 0.047905 probability 0.954 cumulative (n = 60, p =  
0.105).  
We expect 11 wins with 0.025546 probability 0.979 cumulative (n = 60, p =  
0.105).  
We expect 12 wins with 0.012238 probability 0.992 cumulative (n = 60, p =  
0.105).  
We expect 13 wins with 0.005301 probability 0.997 cumulative (n = 60, p =  
0.105).  
We expect 14 wins with 0.002088 probability 0.999 cumulative (n = 60, p =  
0.105).
```

Note that to recover the $27x$ amount, one would need to play only $\log(27)/\log(1.05) = 68$ additional rounds on average with these favourable odds, but one must have something to bet to start with. The Kelly Criterion provides that the optimal is to bet only 1.55% of our bankroll. Note that if I bet the whole bankroll, I would lose all my money with 89.5% certainty in the first round (the probability of a win is only 0.105). If I bet only a fraction of the bankroll, the chances of staying in the game are infinitely better, but the overall returns are smaller. The plot of expected log of return is shown in *Figure 02-3* as a function of the portions of the bankroll to bet, x , and possible distribution of outcomes in 60 bets that I just computed. In 24% of the games we'll do worse than the lower curve, in 39% worse than the next curve, in about half—44%—a gambler we'll do the same or better than the black curve in the middle, and in 30% of cases better than the top one. The optimal Kelly Criterion value for x is 0.0155, which will eventually optimize the overall return over infinitely many rounds:

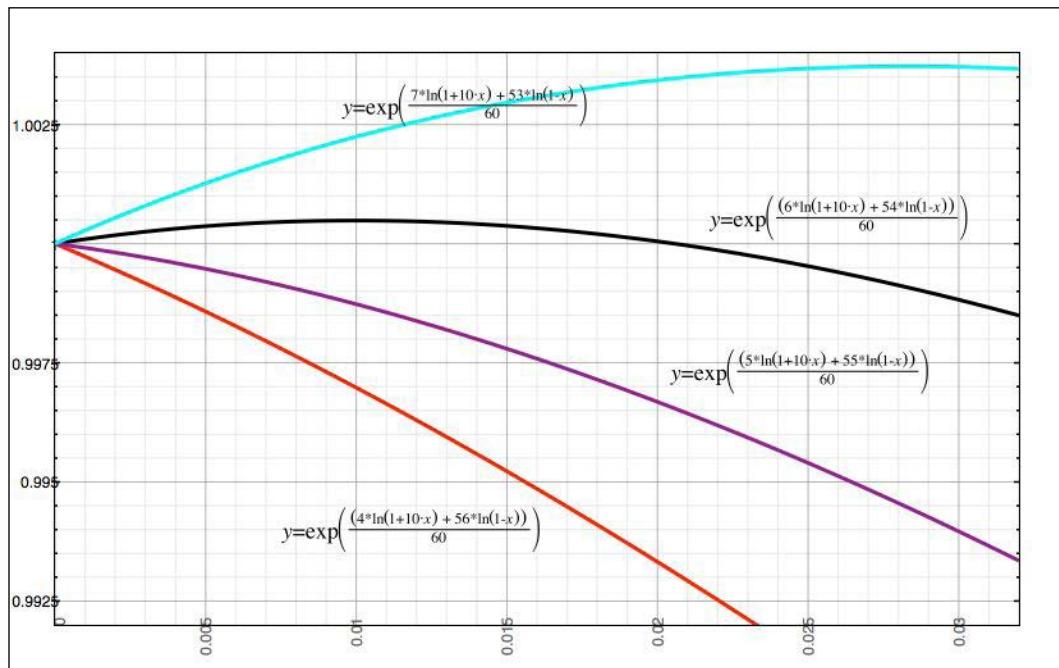


Figure 02-3. The expected log of return as a function of the bet amount and possible outcomes in 60 rounds (see equation (2.2))

The Kelly Criterion has been criticized for being both too aggressive (gamblers tend to overestimate their winning potential/ratio and underestimate the probability of a ruin), as well as for being too conservative (the value at risk should be the total available capital, not just the bankroll), but it demonstrates one of the examples where we need to compensate our intuitive understanding of the "benefit" with some additional transformations.

From the financial point of view, the Kelly Criterion is a much better description of risk than the standard definition as volatility or variance of the returns. For a generic parametrized payoff distribution, $y(z)$, with a probability distribution function, $f(z)$, the equation (2.3) can be reformulated as follows. after the substitution $r(x) = 1 + x y(z)$, where x is still the amount to bet:

$$\frac{\partial E(\log(1+xy(z)))}{\partial x} = E\left(\frac{y(z)}{1+xy(z)}\right) = 0$$
$$\int_{z=-\infty}^{\infty} \frac{y(z)f(z)}{1+xy(z)} dz = 0 \cdot (2.5)$$

It can also be written in the following manner in the discrete case:

$$= \sum_i \frac{y(z_i)p(z_i)}{1+xy(z_i)} = 0 \quad (2.6)$$

Here, the denominator emphasizes the contributions from the regions with negative payoffs. Specifically, the possibility of losing all your bankroll is exactly where the denominator $(1+xy(z))$ is zero.

As I mentioned before, interestingly, risk aversion is engrained in our intuitions and there seems to be a natural risk-aversion system of preferences encoded in both humans and primates (*A Monkey Economy as Irrational as Ours* by Laurie Santos, TED talk, 2010). Now enough about monkeys and risk, let's get into another rather controversial subject—the exploration-exploitation trade-off, where one might not even know the payoff trade-offs initially.

Exploration and exploitation

The exploration-exploitation trade-off is another problem that has its apparent origin within gambling, even though the real applications range from allocation of funding to research projects to self-driving cars. The traditional formulation is a multi-armed bandit problem, which refers to an imaginary slot machine with one or more arms. Sequential plays of each arm generate *i.i.d.* returns with unknown probabilities for each arm; the successive plays are independent in the simplified models. The rewards are assumed to be independent across the arms. The goal is to maximize the reward—for example, the amount of money won, and to minimize the learning loss, or the amount spent on the arms with less than optimal winning rate, provided an agreed upon arm selection policy. The obvious trade-off is between the **exploration** in search of an arm that produces the best return and **exploitation** of the best-known arm with optimal return:

$$r_{opt} = \max_{i=1\dots K} E(r_i)$$

The **pseudo-regret** is then the difference:

$$R_N = Nr_{opt} - \sum_{i=1}^N E(r_{s_i})$$

Here, s_i is the i^{th} arm selection out of N trials. The multi-armed bandit problem was extensively studied in the 1930s and again during the early 2000s, with the application in finance and ADTECH. While in general, due to stochastic nature of the problem, it is not possible to provide a bound on the expected regret better than the square root of N , the pseudo-regret can be controlled so that we are able to bound it by a log of N (*Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems* by Sébastien Bubeck and Nicolo Cesa-Bianchi, <http://arxiv.org/pdf/1204.5721.pdf>).

One of the most common strategies used in practice is epsilon strategies, where the optimal arm is chosen with the probability of $(1 - \varepsilon)$ and one of the other arms with the remaining probability. The drawback of this approach is that we might spend a lot of exploration resources on the arms that are never going to provide any rewards. The UCB strategy improves the epsilon strategy by choosing an arm with the largest estimate of the return, plus some multiple or fraction of the standard deviation of the return estimates. The approach needs the recomputation of the best arm to pull at each round and suffers from approximations made to estimate the mean and standard deviation. Besides, UCB requires the recomputation of the estimates for each successive pull, which might be a scalability problem.

Finally, the Thompson sampling strategy uses a fixed random sample from Beta-Bernoulli posterior estimates and assigns the next arm to the one that gives the minimal expected regret, for which real data can be used to avoid parameter recomputation. Although the specific numbers may depend on the assumptions, one available comparison for these model performances is provided in the following diagram:

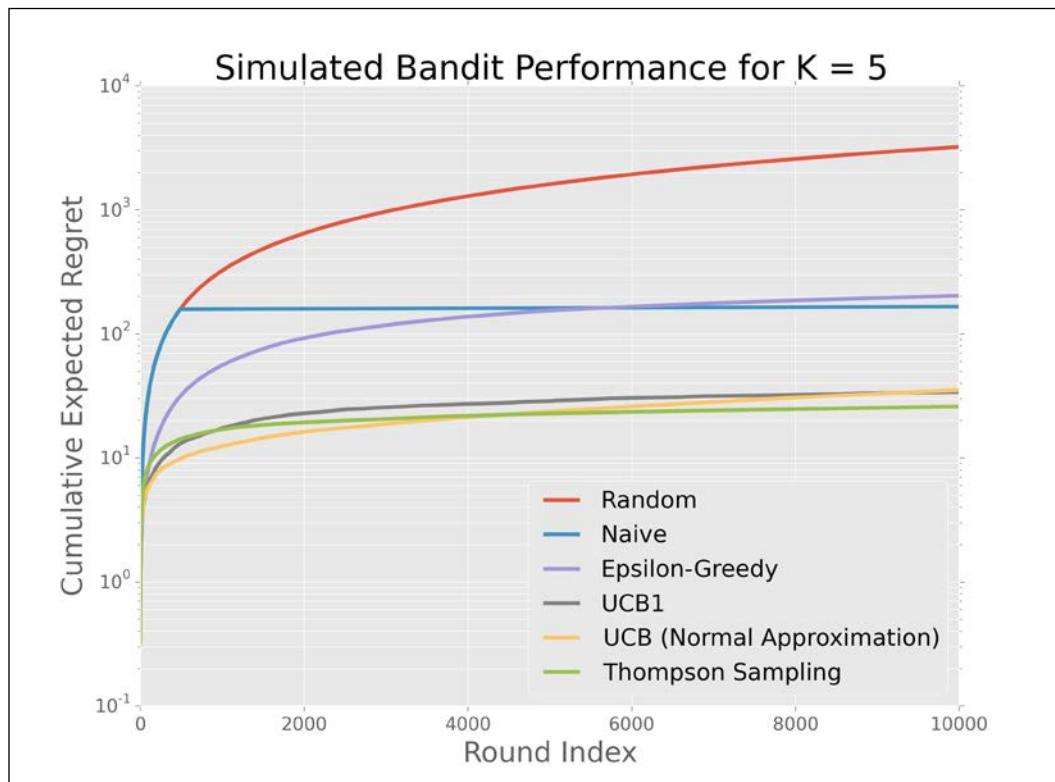


Figure 02-3. The simulation results for different exploration exploitation strategies for $K = 5$, one-armed bandits, and different strategies.

Figure 02-3 shows simulation results for different strategies (taken from the Rich Relevance website at <http://engineering.richrelevance.com/recommendations-thompson-sampling>). The **Random** strategy just allocates the arms at random and corresponds to pure exploration. The **Naive** strategy is random up to a certain threshold and then switches to pure Exploitation mode. **Upper Confidence Bound (UCB)** with 95% confidence level. UCB1 is a modification of UCB to take into account the log-normality of the distributions. Finally the Thompson sampling strategy makes a random sample from actual posterior distribution to optimize the regret.

Exploration/exploitation models are known to be very sensitive to the initial conditions and outliers, particularly on the low-response side. One can spend enormous amount of trials on the arms that are essentially dead.

Other improvements on the strategies are possible by estimating better priors based on additional information, such as location, or limiting the set of arms to explore— K —due to such additional information, but these aspects are more domain-specific (such as personalization or online advertising).

Unknown unknowns

Unknown unknowns have been largely made famous due to a phrase from a response the United States Secretary of Defense, Donald Rumsfeld, gave to a question at a United States **Department of Defense (DoD)** news briefing on February 12, 2002 about the lack of evidence linking the government of Iraq with the supply of weapons of mass destruction to terrorist groups, and books by Nassim Taleb (*The Black Swan: The Impact of the Highly Improbable* by Nassim Taleb, Random House, 2007).

Turkey paradox

Arguably, the unknown unknown is better explained by the turkey paradox. Suppose you have a family of turkeys playing in the backyard and enjoying protection and free food. Across the fence, there is another family of turkeys. This all works day after day, and month after month, until Thanksgiving comes—Thanksgiving Day is a national holiday celebrated in Canada and the United States, where it's customary to roast the turkeys in an oven. The turkeys are very likely to be harvested and consumed at this point, although from the turkey's point of view, there is no discernable signal that anything will happen on the second Monday of October in Canada and the fourth Thursday of November in the United States. No amount of modeling on the within-the-year data can fix this prediction problem from the turkey's point of view besides the additional year-over-year information.



The unknown unknown is something that is not in the model and cannot be anticipated to be in the model. In reality, the only unknown unknowns that are of interest are the ones that affect the model so significantly that the results that were previously virtually impossible, or possible with infinitesimal probability, now become the reality. Given that most of the practical distributions are from exponential family with really thin tails, the deviation from normal does not have to be more than a few sigmas to have devastating results on the standard model assumptions. While one has still to come up with an actionable strategy of how to include the unknown factors in the model—a few ways have been proposed, including fractals, but few if any are actionable—the practitioners have to be aware of the risks, and here the definition of the risk is exactly the possibility of delivering the models useless. Of course, the difference between the known unknown and unknown unknown is exactly that we understand the risks and what needs to be explored.

As we looked at the basic scope of problems that the decision-making systems are facing, let's look at the data pipelines, the software systems that provide information for making the decisions, and more practical aspects of designing the data pipeline for a data-driven system.

Basic components of a data-driven system

In short, a data-driven architecture contains the following components—at least all the systems I've seen have them—or can be reduced to these components:

- **Data ingest:** We need to collect the data from systems and devices. Most of the systems have logs, or at least an option to write files into a local filesystem. Some can have capabilities to report information to network-based interfaces such as syslog, but the absence of persistence layer usually means potential data loss, if not absence of audit information.
- **Data transformation layer:** It was also historically called **extract, transform, and load (ETL)**. Today the data transformation layer can also be used to have real-time processing, where the aggregates are computed on the most recent data. The data transformation layer is also traditionally used to reformat and index the data to be efficiently accessed by a UI component of algorithms down the pipeline.

- **Data analytics and machine learning engine:** The reason this is not part of the standard data transformation layer is usually that this layer requires quite different skills. The mindset of people who build reasonable statistical models is usually different from people who make terabytes of data move fast, even though occasionally I can find people with both skills. Usually, these unicorns are called data scientists, but the skills in any specific field are usually inferior to ones who specialize in a particular field. We need more of either, though. Another reason is that machine learning, and to a certain extent, data analysis, requires multiple aggregations and passes over the same data, which as opposed to a more stream-like ETL transformations, requires a different engine.
- **UI component:** Yes, UI stands for user interface, which most often is a set of components that allow you to communicate with the system via a browser (it used to be a native GUI, but these days the web-based JavaScript or Scala-based frameworks are much more powerful and portable). From the data pipeline and modeling perspective, this component offers an API to access internal representation of data and models.
- **Actions engine:** This is usually a configurable rules engine to optimize the provided metrics based on insights. The actions may be either real-time, like in online advertising, in which case the engine should be able to supply real-time scoring information, or a recommendation for a user action, which can take the form of an e-mail alert.
- **Correlation engine:** This is an emerging component that may analyze the output of data analysis and machine learning engine to infer additional insights into data or model behavior. The actions might also be triggered by an output from this layer.
- **Monitoring:** This is a complex system will be incomplete without logging, monitoring, and some way to change system parameters. The purpose of monitoring is to have a nested decision-making system regarding the optimal health of the system and either to mitigate the problem(s) automatically or to alert the system administrators about the problem(s).

Let's discuss each of the components in detail in the following sections.

Data ingest

With the proliferation of smart devices, information gathering has become less of a problem and more of a necessity for any business that does more than a type-written text. For the purpose of this chapter, I will assume that the device or devices are connected to the Internet or have some way of passing this information via home dialing or direct network connection.

The major purpose of this component is to collect all relevant information that can be relevant for further data-driven decision making. The following table provides details on the most common implementations of the data ingest:

Framework	When used	Comments
Syslog	Syslog is one of the most common standards to pass messages between the machines on Unix. Syslog usually listens on port 514 and the transport protocol can be configured either with UDP (unreliable) or with TCP. The latest enhanced implementation on CentOS and Red Hat Linux is rsyslog, which includes many advanced options such as regex-based filtering that is useful for system-performance tuning and debugging. Apart from slightly inefficient raw message representation – plain text, which might be inefficient for long messages with repeated strings – the syslog system can support tens of thousands of messages per second.	Syslog is one of the oldest protocols developed in the 1980s by Eric Allman as part of Sendmail. While it does not guarantee delivery or durability, particularly for distributed systems, it is one of the most widespread protocols for message passing. Some of the later frameworks, such as Flume and Kafka, have syslog interfaces as well.
Rsync	Rsync is a younger framework developed in the 1990s. If the data is put in the flat files on a local filesystem, rsync might be an option. While rsync is more traditionally used to synchronize two directories, it also can be run periodically to transfer log data in batches. Rsync uses a recursive algorithm invented by an Australian computer programmer, Andrew Tridgell, for efficiently detecting the differences and transmitting a structure (such as a file) across a communication link when the receiving computer already has a similar, but not identical, version of the same structure. While it incurs extra communication, it is better from the point of durability, as the original copy can always be retrieved. It is particularly appropriate if the log data is known to arrive in batches in the first place (such as uploads or downloads).	Rsync has been known to be hampered by network bottlenecks, as it ultimately passes more information over the network when comparing the directory structures. However, the transferred files may be compressed when passed over the network. The network bandwidth can be limited per command-line flags.

Framework	When used	Comments
Flume	Flume is one of the youngest frameworks developed by Cloudera in 2009-2011 and open sourced. Flume – we refer to the more popular flume-ng implementation as Flume as opposed to an older regular Flume—consists of sources, pipes, and sinks that may be configured on multiple nodes for high availability and redundancy purposes. Flume was designed to err on the reliability side at the expense of possible duplication of data. Flume passes the messages in the Avro format, which is also open sourced and the transfer protocol, as well as messages can be encoded and compressed.	While Flume originally was developed just to ship records from a file or a set of files, it can also be configured to listen to a port, or even grab the records from a database. Flume has multiple adapters including the preceding syslog.
Kafka	Kafka is the latest addition to the log-processing framework developed by LinkedIn and is open sourced. Kafka, compared to the previous frameworks, is more like a distributed reliable message queue. Kafka keeps a partitioned, potentially between multiple distributed machines; buffer and one can subscribe to or unsubscribe from getting messages for a particular topic. Kafka was built with strong reliability guarantees in mind, which is achieved through replication and consensus protocol.	Kafka might not be appropriate for small systems (< five nodes) as the benefits of the fully distributed system might be evident only at larger scales. Kafka is commercially supported by Confluent.

The transfer of information usually occurs in batches, or micro batches if the requirements are close to real time. Usually the information first ends up in a file, traditionally called log, in a device's local filesystem, and then is transferred to a central location. Recently developed Kafka and Flume are often used to manage these transfers, together with a more traditional syslog, rsync, or netcat. Finally, the data can be placed into a local or distributed storage such as HDFS, Cassandra, or Amazon S3.

Data transformation layer

After the data ends up in HDFS or other storage, the data needs to be made available for processing. Traditionally, the data is processed on a schedule and ends up partitioned by time-based buckets. The processing can happen daily or hourly, or even on a sub-minute basis with the new Scala streaming framework, depending on the latency requirements. The processing may involve some preliminary feature construction or vectorization, even though it is traditionally considered a machine-learning task. The following table summarizes some available frameworks:

Framework	When used	Comments
Oozie	This is one of the oldest open source frameworks developed by Yahoo. This has good integration with big data Hadoop tools. It has limited UI that lists the job history.	The whole workflow is put into one big XML file, which might be considered a disadvantage from the modularity point of view.
Azkaban	This is an alternative open source workflow-scheduling framework developed by LinkedIn. Compared to Oozie, this arguably has a better UI. The disadvantage is that all high-level tasks are executed locally, which might present a scalability problem.	The idea behind Azkaban is to create a fully modularized drop-in architecture where the new jobs/tasks can be added with as few modifications as possible.
StreamSets	StreamSets is the latest addition build by the former Informix and Cloudera developers. It has a very developed UI and supports a much richer set of input sources and output destinations.	This is a fully UI-driven tool with an emphasis on data curation, for example, constantly monitoring the data stream for problems and abnormalities.

Separate attention should be given to stream-processing frameworks, where the latency requirements are reduced to one or a few records at a time. First, stream processing usually requires much more resources dedicated to processing, as it is more expensive to process individual records at a time as opposed to batches of records, even if it is tens or hundreds of records. So, the architect needs to justify the additional costs based on the value of more recent result, which is not always warranted. Second, stream processing requires a few adjustments to the architecture as handling the more recent data becomes a priority; for example, a delta architecture where the more recent data is handled by a separate substream or a set of nodes became very popular recently with systems such as **Druid** (<http://druid.io>).

Data analytics and machine learning

For the purpose of this chapter, **Machine Learning (ML)** is any algorithm that can compute aggregates or summaries that are actionable. We will cover more complex algorithms from *Chapter 3, Working with Spark and MLlib* to *Chapter 6, Working with Unstructured Data*, but in some cases, a simple sliding-window average and deviation from the average may be sufficient signal for taking an action. In the past few years, it just works in A/B testing somehow became a convincing argument for model building and deployment. I am not speculating that solid scientific principles might or might not apply, but many fundamental assumptions such as *i.i.d.*, balanced designs, and the thinness of the tail just fail to hold for many big data situation. Simpler models tend to be faster and to have better performance and stability.

For example, in online advertising, one might just track average performance of a set of ads over a certain similar properties over times to make a decision whether to have this ad displayed. The information about anomalies, or deviation from the previous behavior, may signal a new unknown unknown, which signals that the old data no longer applies, in which case, the system has no choice but to start the new exploration cycle.

I will talk about more complex non-structured, graph, and pattern mining later in *Chapter 6, Working with Unstructured Data*, *Chapter 8, Integrating Scala with R and Python* and *Chapter 9, NLP in Scala*.

UI component

Well, UI is for wimps! Just joking...maybe it's too harsh, but in reality, UI usually presents a syntactic sugar that is necessary to convince the population beyond the data scientists. A good analyst should probably be able to figure out t-test probabilities by just looking at a table with numbers.

However, one should probably apply the same methodologies we used at the beginning of the chapter, assessing the usefulness of different components and the amount of cycles put into them. The presence of a good UI is often justified, but depends on the target audience.

First, there are a number of existing UIs and reporting frameworks. Unfortunately, most of them are not aligned with the functional programming methodologies. Also, the presence of complex/semi-structured data, which I will describe in *Chapter 6, Working with Unstructured Data* in more detail, presents a new twist that many frameworks are not ready to deal with without implementing some kind of DSL. Here are a few frameworks for building the UI in a Scala project that I find particularly worthwhile:

Framework	When used	Comments
Scala Swing	If you used Swing components in Java and are proficient with them, Scala Swing is a good choice for you. Swing component is arguably the least portable component of Java, so your mileage can vary on different platforms.	The <code>Scala.swing</code> package uses the standard Java Swing library under the hood, but it has some nice additions. Most notably, as it's made for Scala, it can be used in a much more concise way than the standard Swing.
Lift	Lift is a secure, developer-centric, scalable, and interactive framework written in Scala. Lift is open sourced under Apache 2.0 license.	The open source Lift framework was launched in 2007 by David Polak, who was dissatisfied with certain aspects of the Ruby on Rails framework. Any existing Java library and web container can be used in running Lift applications. Lift web applications are thus packaged as WAR files and deployed on any servlet 2.4 engine (for example, Tomcat 5.5.xx, Jetty 6.0, and so on). Lift programmers may use the standard Scala/Java development toolchain, including IDEs such as Eclipse, NetBeans, and IDEA. Dynamic web content is authored via templates using standard HTML5 or XHTML editors. Lift applications also benefit from native support for advanced web development techniques, such as Comet and Ajax.

Framework	When used	Comments
Play	Play is arguably better aligned with Scala as a functional language than any other platform—it is officially supported by Typesafe, the commercial company behind Scala. The Play framework 2.0 builds on Scala, Akka, and sbt to deliver superior asynchronous request handling, fast and reliable. Typesafe templates, and a powerful build system with flexible deployment options. Play is open sourced under Apache 2.0 license.	The open source Play framework was created in 2007 by Guillaume Bort, who sought to bring a fresh web development experience inspired by modern web frameworks like Ruby on Rails to the long-suffering Java web development community. Play follows a familiar stateless model-view-controller architectural pattern, with a philosophy of convention-over-configuration and an emphasis on developer productivity. Unlike traditional Java web frameworks with their tedious compile-package-deploy-restart cycles, updates to Play applications are instantly visible with a simple browser refresh.
Dropwizard	The dropwizard (www.dropwizard.io) project is an attempt to build a generic RESTful framework in both Java and Scala, even though one might end up using more Java than Scala. What is nice about this framework is that it is flexible enough to be used with arbitrary complex data (including semi-structured). This is licensed under Apache License 2.0.	RESTful API assumes state, while functional languages shy away from using state. Unless you are flexible enough to deviate from a pure functional approach, this framework is probably not good enough for you.
Slick	While Slick is not a UI component, it is Typesafe's modern database query and access library for Scala, which can serve as a UI backend. It allows you to work with the stored data almost as if you were using Scala collections, while at the same time, giving you full control over when a database access occurs and what data is transferred. You can also use SQL directly. Use it if all of your data is purely relational. This is open sourced under BSD-Style license.	Slick was started in 2012 by Stefan Zeiger and maintained mainly by Typesafe. It is useful for mostly relational data.

Framework	When used	Comments
NodeJS	Node.js is a JavaScript runtime, built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world. It is open sourced under MIT License.	Node.js was first introduced in 2009 by Ryan Dahl and other developers working at Joyent. Originally Node.js supported only Linux, but now it runs on OS X and Windows.
AngularJS	AngularJS (https://angularjs.org) is a frontend development framework, built to simplify development of one-page web applications. This is open sourced under MIT License.	AngularJS was originally developed in 2009 by Misko Hevery at Brat Tech LLC. AngularJS is mainly maintained by Google and by a community of individual developers and corporations, and thus is specifically for Android platform (support for IE8 is dropped in versions 1.3 and later).

Actions engine

While this is the heart of the data-oriented system pipeline, it is also arguably the easiest one. Once the system of metrics and values is known, the system decides, based on the known equations, whether to take a certain set of actions or not, based on the information provided. While the triggers based on a threshold is the most common implementation, the significance of probabilistic approaches that present the user with a set of possibilities and associated probabilities is emerging—or just presenting the user with the top N relevant choices like a search engine does.

The management of the rules might become pretty involved. It used to be that managing the rules with a rule engine, such as **Drools** (<http://www.drools.org>), was sufficient. However, managing complex rules becomes an issue that often requires development of a DSL (*Domain-Specific Languages* by Martin Fowler, Addison-Wesley, 2010). Scala is particularly fitting language for the development of such an actions engine.

Correlation engine

The more complex the decision-making system is, the more it requires a secondary decision-making system to optimize its management. DevOps is turning into DataOps (*Getting Data Right* by Michael Stonebraker et al., Tamr, 2015). Data collected about the performance of a data-driven system are used to detect anomalies and semi-automated maintenance.

Models are often subject to time drift, where the performance might deteriorate either due to the changes in the data collection layers or the behavioral changes in the population (I will cover model drift in *Chapter 10, Advanced Model Monitoring*). Another aspect of model management is to track model performance, and in some cases, use "collective" intelligence of the models by various consensus schemes.

Monitoring

Monitoring a system involves collecting information about system performance either for audit, diagnostic, or performance-tuning purposes. While it is related to the issues raised in the previous sections, monitoring solution often incorporates diagnostic and historical storage solutions and persistence of critical data, such as a black box on an airplane. In the Java and, thus, Scala world, a popular tool of choice is Java performance beans, which can be monitored in the Java Console. While Java natively supports MBean for exposing JVM information over JMX, **Kamon** (<http://kamon.io>) is an open source library that uses this mechanism to specifically expose Scala and Akka metrics.

Some other popular monitoring open source solutions are **Ganglia** (<http://ganglia.sourceforge.net/>) and **Graphite** (<http://graphite.wikidot.com>).

I will stop here, as I will address system and model monitoring in more detail in *Chapter 10, Advanced Model Monitoring*.

Optimization and interactivity

While the data collected can be just used for understanding the business, the final goal of any data-driven business is to optimize the business behavior by automatically making data-based and model-based decisions. We want to reduce human intervention to minimum. The following simplified diagram can be depicted as a cycle:

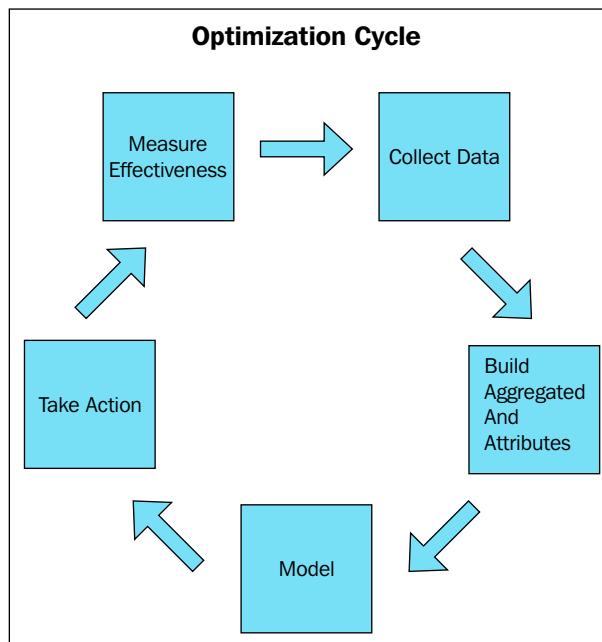


Figure 02-4. The predictive model life cycle

The cycle is repeated over and over for new information coming into the system. The parameters of the system may be tuned to improve the overall system performance.

Feedback loops

While humans are still likely to be kept in the loop for most of the systems, last few years saw an emergence of systems that can manage the complete feedback loop on their own—ranging from advertisement systems to self-driving cars.

The classical formulation of this problem is the optimal control theory, which is also an optimization problem to minimize cost functional, given a set of differential equations describing the system. An optimal control is a set of control policies to minimize the cost functional given constraints. For example, the problem might be to find a way to drive the car to minimize its fuel consumption, given that it must complete a given course in a time not exceeding some amount. Another control problem is to maximize profit for showing ads on a website, provided the inventory and time constraints. Most software packages for optimal control are written in other languages such as C or MATLAB (PROPT, SNOPT, RIOTS, DIDO, DIRECT, and GPOPS), but can be interfaced with Scala.

However, in many cases, the parameters for the optimization or the state transition, or differential equations, are not known with certainty. **Markov Decision Processes (MDPs)** provide a mathematical framework to model decision making in situations where outcomes are partly random and partly under the control of the decision maker. In MDPs, we deal with a discrete set of possible states and a set of actions. The "rewards" and state transitions depend both on the state and actions. MDPs are useful for studying a wide range of optimization problems solved via dynamic programming and reinforcement learning.

Summary

In this chapter, I described a high-level architecture and approach to design a data-driven enterprise. I also introduced you to influence diagrams, a tool for understanding how the decisions are made in traditional and data-driven enterprises. I stopped on a few key models, such as Kelly Criterion and multi-armed bandit, essential to demonstrate the issues from the mathematical point of view. I built on top of this to introduce some Markov decision process approaches where we deal with decision policies based on the results of the previous decisions and observations. I delved into more practical aspects of building a data pipeline for decision-making, describing major components and frameworks that can be used to build them. I also discussed the issues of communicating the data and modeling results between different stages and nodes, presenting the results to the user, feedback loop, and monitoring.

In the next chapter, I will describe MLlib, a library for machine learning over distributed set of nodes written in Scala.

3

Working with Spark and MLlib

Now that we are powered with the knowledge of where and how statistics and machine learning fits in the global data-driven enterprise architecture, let's stop at the specific implementations in Spark and MLlib, a machine learning library on top of Spark. Spark is a relatively new member of the big data ecosystem that is optimized for memory usage as opposed to disk. The data can be still spilled to disk as necessary, but Spark does the spill only if instructed to do so explicitly, or if the active dataset does not fit into the memory. Spark stores lineage information to recompute the active dataset if a node goes down or the information is erased from memory for some other reason. This is in contrast to the traditional MapReduce approach, where the data is persisted to the disk after each map or reduce task.

Spark is particularly suited for iterative or statistical machine learning algorithms over a distributed set of nodes and can scale out of core. The only limitation is the total memory and disk space available across all Spark nodes and the network speed. I will cover the basics of Spark architecture and implementation in this chapter.

One can direct Spark to execute data pipelines either on a single node or across a set of nodes with a simple change in the configuration parameters. Of course, this flexibility comes at a cost of slightly heavier framework and longer setup times, but the framework is very parallelizable and as most of modern laptops are already multithreaded and sufficiently powerful, this usually does not present a big issue.

In this chapter, we will cover the following topics:

- Installing and configuring Spark if you haven't done so yet
- Learning the basics of Spark architecture and why it is inherently tied to the Scala language
- Learning why Spark is the next technology after sequential implementations and Hadoop MapReduce
- Learning about Spark components

- Looking at the simple implementation of word count in Scala and Spark
- Looking at the streaming word count implementation
- Seeing how to create Spark DataFrames from either a distributed file or a distributed database
- Learning about Spark performance tuning

Setting up Spark

If you haven't done so yet, you can download the pre-build Spark package from <http://spark.apache.org/downloads.html>. The latest release at the time of writing is **1.6.1**:

The screenshot shows the 'Download Apache Spark™' section of the Apache Spark website. It includes a list of five steps for selecting a Spark release, package type, download type, and download URL. A note for Scala 2.11 users is present. To the right, there is a sidebar titled 'Latest News' with three recent articles.

Download Apache Spark™

Our latest version is Spark 1.6.1, released on March 9, 2016 ([release notes](#)) ([git tag](#))

1. Choose a Spark release:
2. Choose a package type:
3. Choose a download type:
4. Download Spark: [spark-1.6.1.tgz](#)
5. Verify this release using the [1.6.1 signatures and checksums](#).

Note: Scala 2.11 users should download the Spark source package and build with Scala 2.11 support.

Link with Spark

Latest News

- Spark Summit (Jun San Francisco) ago posted (Apr 17, 2016)
- Spark 1.6.1 release (2016)
- Submission is open for the Summit San Francisco (11, 2016)
- Spark Summit East (Mar 9, 2016, New York) ago posted (Jan 14, 2016)

Figure 03-1. The download site at <http://spark.apache.org> with recommended selections for this chapter

Alternatively, you can build the Spark by downloading the full source distribution from <https://github.com/apache/spark>:

```
$ git clone https://github.com/apache/spark.git
Cloning into 'spark'...
remote: Counting objects: 301864, done.
...
$ cd spark
$ sh ./ dev/change-scala-version.sh 2.11
...
$ ./make-distribution.sh --name alex-build-2.6-yarn --skip-java-test --tgz
-Pyarn -Phive -Phive-thriftserver -Pscala-2.11 -Phadoop-2.6
...
```

The command will download the necessary dependencies and create the `spark-2.0.0-SNAPSHOT-bin-alex-spark-build-2.6-yarn.tgz` file in the Spark directory; the version is 2.0.0, as it is the next release version at the time of writing. In general, you do not want to build from trunk unless you are interested in the latest features. If you want a released version, you can checkout the corresponding tag. Full list of available versions is available via the `git branch -r` command. The `spark*.tgz` file is all you need to run Spark on any machine that has Java JRE.

The distribution comes with the `docs/building-spark.md` document that describes other options for building Spark and their descriptions, including incremental Scala compiler, zinc. Full Scala 2.11 support is in the works for the next Spark 2.0.0 release.

Understanding Spark architecture

A parallel execution involves splitting the workload into subtasks that are executed in different threads or on different nodes. Let's see how Spark does this and how it manages execution and communication between the subtasks.

Task scheduling

Spark workload splitting is determined by the number of partitions for **Resilient Distributed Dataset (RDD)**, the basic abstraction in Spark, and the pipeline structure. An RDD represents an immutable, partitioned collection of elements that can be operated on in parallel. While the specifics might depend on the mode in which Spark runs, the following diagram captures the Spark task/resource scheduling:

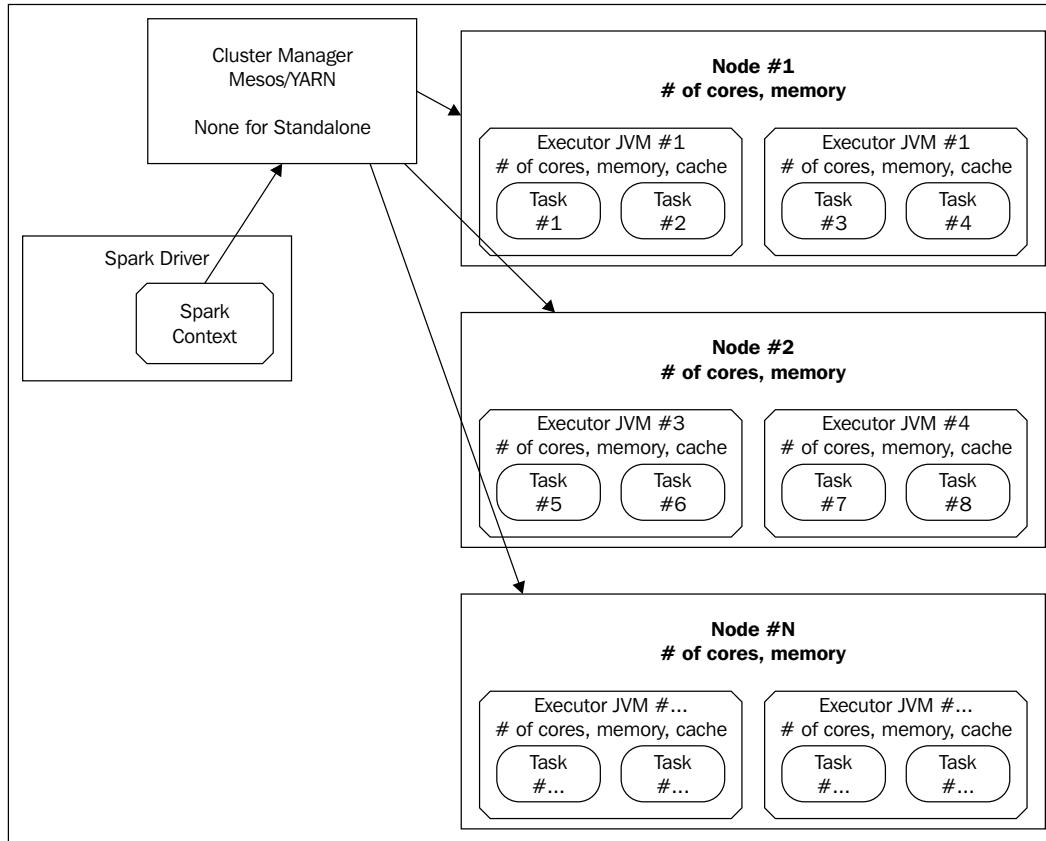


Figure 03-2. A generic Spark task scheduling diagram. While not shown explicitly in the figure, Spark Context opens an HTTP UI, usually on port 4040 (the concurrent contexts will open 4041, 4042, and so on), which is present during a task execution. Spark Master UI is usually 8080 (although it is changed to 18080 in CDH) and Worker UI is usually 7078. Each node can run multiple executors, and each executor can run multiple tasks.



You will find that Spark, as well as Hadoop, has a lot of parameters. Some of them are specified as environment variables (refer to the `$SPARK_HOME/conf/spark-env.sh` file), and yet some can be given as a command-line parameter. Moreover, some files with pre-defined names can contain parameters that will change the Spark behavior, such as `core-site.xml`. This might be confusing, and I will cover as much as possible in this and the following chapters. If you are working with **Hadoop Distributed File System (HDFS)**, then the `core-site.xml` and `hdfs-site.xml` files will contain the pointer and specifications for the HDFS master. The requirement for picking this file is that it has to be on CLASSPATH Java process, which, again, may be set by either specifying `HADOOP_CONF_DIR` or `SPARK_CLASSPATH` environment variables. As is usual with open source, you need to grep the code sometimes to understand how various parameters work, so having a copy of the source tree on your laptop is a good idea.

Each node in the cluster can run one or more executors, and each executor can schedule a sequence of tasks to perform the Spark operations. Spark driver is responsible for scheduling the execution and works with the cluster scheduler, such as Mesos or YARN to schedule the available resources. Spark driver usually runs on the client machine, but in the latest release, it can also run in the cluster under the cluster manager. YARN and Mesos have the ability to dynamically manage the number of executors that run concurrently on each node, provided the resource constraints.

In the Standalone mode, **Spark Master** does the work of the cluster scheduler—it might be less efficient in allocating resources, but it's better than nothing in the absence of preconfigured Mesos or YARN. Spark standard distribution contains shell scripts to start Spark in Standalone mode in the `sbin` directory. Spark Master and driver communicate directly with one or several Spark workers that run on individual nodes. Once the master is running, you can start Spark shell with the following command:

```
$ bin/spark-shell --master spark://<master-address>:7077
```



Note that you can always run Spark in local mode, which means that all tasks will be executed in a single JVM, by specifying `--master local [2]`, where 2 is the number of threads that have to be at least 2. In fact, we will be using the local mode very often in this book for running small examples.

Spark shell is an application from the Spark point of view. Once you start a Spark application, you will see it under **Running Applications** in the Spark Master UI (or in the corresponding cluster manager), which can redirect you to the Spark application HTTP UI at port 4040, where one can see the subtask execution timeline and other important properties such as environment setting, classpath, parameters passed to the JVM, and information on resource usage (refer to *Figure 3-3*):

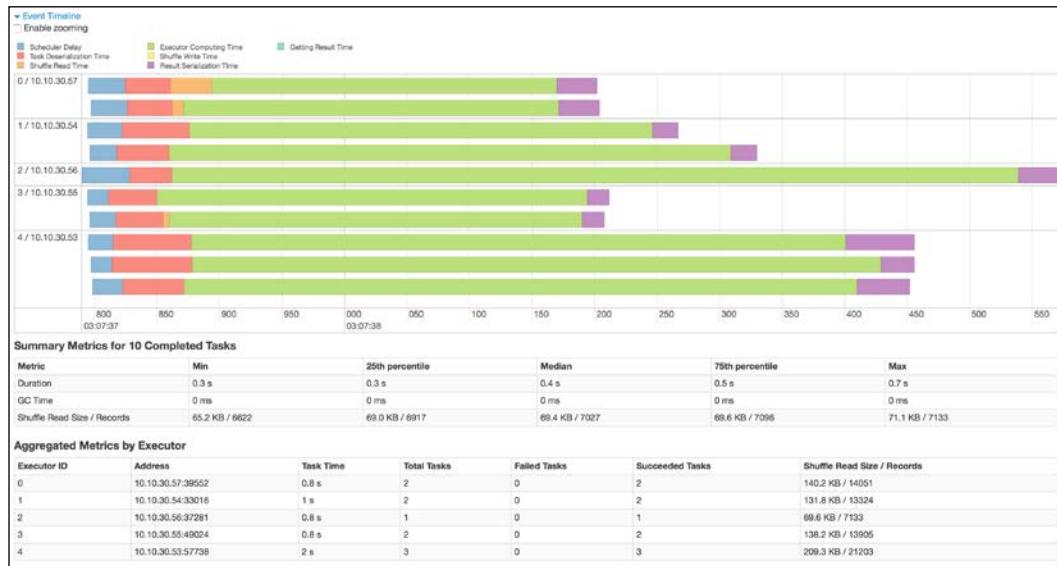


Figure 03-3. Spark Driver UI in Standalone mode with time decomposition

As we saw, with Spark, one can easily switch between local and cluster mode by providing the `--master` command-line option, setting a `MASTER` environment variable, or modifying `spark-defaults.conf`, which should be on the classpath during the execution, or even set explicitly using the `setters` method on the `SparkConf` object directly in Scala, which will be covered later:

Cluster Manager	MASTER env variable	Comments
Local (single node, multiple threads)	<code>local [n]</code>	<code>n</code> is the number of threads to use, should be greater than or equal to 2. If you want Spark to communicate with other Hadoop tools such as Hive, you still need to point it to the cluster by either setting the <code>HADOOP_CONF_DIR</code> environment variable or copying the Hadoop <code>*-site.xml</code> configuration files into the <code>conf</code> subdirectory.

Cluster Manager	MASTER env variable	Comments
Standalone (Daemons running on the nodes)	spark://master-address:>:7077	This has a set of start/stop scripts in the \$SPARK_HOME/sbin directory. This also supports the HA mode. More details can be found at https://spark.apache.org/docs/latest/spark-standalone.html .
Mesos	mesos://host:5050 or mesos://zk://host:2181 (multimaster)	Here, you need to set MESOS_NATIVE_JAVA_LIBRARY=<path to libmesos.so> and SPARK_EXECUTOR_URI=<URL of spark-1.5.0.tar.gz>. The default is fine-grained mode, where each Spark task runs as a separate Mesos task. Alternatively, the user can specify the coarse-grained mode, where the Mesos tasks persists for the duration of the application. The advantage is lower total start-up costs. This can use dynamic allocation (refer to the following URL) in coarse-grained mode. More details can be found at https://spark.apache.org/docs/latest/running-on-mesos.html .
YARN	yarn	Spark driver can run either in the cluster or on the client node, which is managed by the --deploy-mode parameter (cluster or client, shell can only run in the client mode). Set HADOOP_CONF_DIR or YARN_CONF_DIR to point to the YARN config files. Use the --num-executors flag or spark.executor.instances property to set a fixed number of executors (default). Set spark.dynamicAllocation.enabled to true to dynamically create/kill executors depending on the application demand. More details are available at https://spark.apache.org/docs/latest/running-on-yarn.html .

The most common ports are 8080, the master UI, and 4040, the application UI. Other Spark ports are summarized in the following table:

Standalone ports				
From	To	Default Port	Purpose	Configuration Setting
Browser	Standalone Master	8080	Web UI	spark.master.ui.port / SPARK_MASTER_WEBUI_PORT

Standalone ports				
From	To	Default Port	Purpose	Configuration Setting
Browser	Standalone worker	8081	Web UI	spark.worker.ui.port / SPARK_WORKER_WEBUI_PORT
Driver / Standalone worker	Standalone Master	7077	Submit job to cluster / Join cluster	SPARK_MASTER_PORT
Standalone master	Standalone worker	(random)	Schedule executors	SPARK_WORKER_PORT
Executor / Standalone master	Driver	(random)	Connect to application / Notify executor state changes	spark.driver.port
Other ports				
From	To	Default Port	Purpose	Configuration Setting
Browser	Application	4040	Web UI	spark.ui.port
Browser	History server	18080	Web UI	spark.history.ui.port
Driver	Executor	(random)	Schedule tasks	spark.executor.port
Executor	Driver	(random)	File server for files and jars	spark.filesServer.port
Executor	Driver	(random)	HTTP broadcast	spark.broadcast.port

Also, some of the documentation is available with the source distribution in the `docs` subdirectory, but may be out of date.

Spark components

Since the emergence of Spark, multiple applications that benefit from Spark's ability to cache RDDs have been written: Shark, Spork (Pig on Spark), graph libraries (GraphX, GraphFrames), streaming, MLlib, and so on; some of these will be covered here and in later chapters.

In this section, I will cover major architecture components to collect, store, and analyze the data in Spark. While I will cover a more complete data life cycle architecture in *Chapter 2, Data Pipelines and Modeling*, here are Spark-specific components:

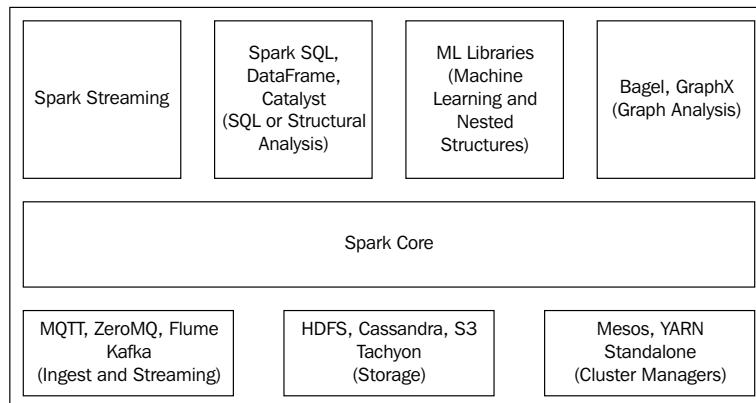


Figure 03-4. Spark architecture and components.

MQTT, ZeroMQ, Flume, and Kafka

All of these are different ways to reliably move data from one place to another without loss and duplication. They usually implement a publish-subscribe model, where multiple writers and readers can write and read from the same queues with different guarantees. Flume stands out as a first distributed log and event management implementation, but it is slowly replaced by Kafka, a fully functional publish-subscribe distributed message queue optionally persistent across a distributed set of nodes developed at LinkedIn. We covered Flume and Kafka briefly in the previous chapter. Flume configuration is file-based and is traditionally used to deliver messages from a Flume source to one or several Flume sinks. One of the popular sources is netcat — listening on raw data over a port. For example, the following configuration describes an agent receiving data and then writing them to HDFS every 30 seconds (default):

```

# Name the components on this agent
a1.sources = r1
a1.sinks = k1
  
```

```
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 4987

# Describe the sink (the instructions to configure and start HDFS are
# provided in the Appendix)
a1.sinks.k1.type=hdbs
a1.sinks.k1.hdfs.path=hdbs://localhost:8020/flume/netcat/data
a1.sinks.k1.hdfs.filePrefix=chapter03.example
a1.sinks.k1.channel=c1
a1.sinks.k1.hdfs.writeFormat = Text

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

This file is included as part of the code provided with this book in the chapter03/conf directory. Let's download and start Flume agent (check the MD5 sum with one provided at <http://flume.apache.org/download.html>):

```
$ wget http://mirrors.ocf.berkeley.edu/apache/flume/1.6.0/apache-flume-
1.6.0-bin.tar.gz
$ md5sum apache-flume-1.6.0-bin.tar.gz
MD5 (apache-flume-1.6.0-bin.tar.gz) = defd21ad8d2b6f28cc0a16b96f652099
$ tar xf apache-flume-1.6.0-bin.tar.gz
$ cd apache-flume-1.6.0-bin
$ ./bin/flume-ng agent -Dlog.dir=. -Dflume.log.level=DEBUG,console -n a1
-f ../chapter03/conf/flume.conf
Info: Including Hadoop libraries found via (/Users/akozlov/hadoop-2.6.4/
bin/hadoop) for HDFS access
Info: Excluding /Users/akozlov/hadoop-2.6.4/share/hadoop/common/lib/
slf4j-api-1.7.5.jar from classpath
Info: Excluding /Users/akozlov/hadoop-2.6.4/share/hadoop/common/lib/
slf4j-log4j12-1.7.5.jar from classpath
...
```

Now, in a separate window, you can type a `netcat` command to send text to the Flume agent:

```
$ nc localhost 4987  
Hello  
OK  
World  
OK
```

...

The Flume agent will first create a `*.tmp` file and then rename it to a file without extension (the file extension can be used to filter out files being written to):

```
$ bin/hdfs dfs -text /flume/netcat/data/chapter03.example.1463052301372  
16/05/12 04:27:25 WARN util.NativeCodeLoader: Unable to load native-  
hadoop library for your platform... using builtin-java classes where  
applicable  
1463052302380 Hello  
1463052304307 World
```

Here, each row is a Unix time in milliseconds and data received. In this case, we put the data into HDFS, from where they can be analyzed by a Spark/Scala program, we can exclude the files being written to by the `*.tmp` filename pattern. However, if you are really interested in up-to-the-last-minute values, Spark, as well as some other platforms, supports streaming, which I will cover in a few sections.

HDFS, Cassandra, S3, and Tachyon

HDFS, Cassandra, S3, and Tachyon are the different ways to get the data into persistent storage and compute nodes as necessary with different guarantees. HDFS is a distributed storage implemented as a part of Hadoop, which serves as the backend for many products in the Hadoop ecosystem. HDFS divides each file into blocks, which are 128 MB in size by default, and stores each block on at least three nodes. Although HDFS is reliable and supports HA, a general complaint about HDFS storage is that it is slow, particularly for machine learning purposes. Cassandra is a general-purpose key/value storage that also stores multiple copies of a row and can be configured to support different levels of consistency to optimize read or write speeds. The advantage that Cassandra over HDFS model is that it does not have a central master node; the reads and writes are completed based on the consensus algorithm. This, however, may sometimes reflect on the Cassandra stability. S3 is the Amazon storage: The data is stored off-cluster, which affects the I/O speed. Finally, the recently developed Tachyon claims to utilize node's memory to optimize access to working sets across the nodes.

Additionally, new backends are being constantly developed, for example, Kudu from Cloudera (<http://getkudu.io/kudu.pdf>) and **Ignite File System (IGFS)** from GridGain (<http://apacheignite.gridgain.org/v1.0/docs/igfs>). Both are open source and Apache-licensed.

Mesos, YARN, and Standalone

As we mentioned before, Spark can run under different cluster resource schedulers. These are various implementations to schedule Spark's containers and tasks on the cluster. The schedulers can be viewed as cluster kernels, performing functions similar to the operating system kernel: resource allocation, scheduling, I/O optimization, application services, and UI.

Mesos is one of the original cluster managers and is built using the same principles as the Linux kernel, only at a different level of abstraction. A Mesos slave runs on every machine and provides API's for resource management and scheduling across entire datacenter and cloud environments. Mesos is written in C++.

YARN is a more recent cluster manager developed by Yahoo. Each node in YARN runs a **Node Manager**, which communicates with the **Resource Manager** which may run on a separate node. The resource manager schedules the task to satisfy memory and CPU constraints. The Spark driver itself can run either in the cluster, which is called the cluster mode for YARN. Otherwise, in the client mode, only Spark executors run in the cluster and the driver that schedules Spark pipelines runs on the same machine that runs Spark shell or submit program. The Spark executors will talk to the local host over a random open port in this case. YARN is written in Java with the consequences of unpredictable GC pauses, which might make latency's long tail fatter.

Finally, if none of these resource schedulers are available, the standalone deployment mode starts a `org.apache.spark.deploy.worker.Worker` process on each node that communicates with the Spark Master process run as `org.apache.spark.deploy.master.Master`. The worker process is completely managed by the master and can run multiple executors and tasks (refer to *Figure 3-2*).

In practical implementations, it is advised to track the program parallelism and required resources through driver's UI and adjust the parallelism and available memory, increasing the parallelism if necessary. In the following section, we will start looking at how Scala and Scala in Spark address different problems.

Applications

Let's consider a few practical examples and libraries in Spark/Scala starting with a very traditional problem of word counting.

Word count

Most modern machine learning algorithms require multiple passes over data. If the data fits in the memory of a single machine, the data is readily available and this does not present a performance bottleneck. However, if the data becomes too large to fit into RAM, one has a choice of either dumping pieces of the data on disk (or database), which is about 100 times slower, but has a much larger capacity, or splitting the dataset between multiple machines across the network and transferring the results. While there are still ongoing debates, for most practical systems, analysis shows that storing the data over a set of network connected nodes has a slight advantage over repeatedly storing and reading it from hard disks on a single node, particularly if we can split the workload effectively between multiple CPUs.



An average disk has bandwidth of about 100 MB/sec and transfers with a few ms latency, depending on the rotation speed and caching. This is about 100 times slower than reading the data from memory, depending on the data size and caching implementation again. Modern data bus can transfer data at over 10 GB/sec. While the network speed still lags behind the direct memory access, particularly with standard TCP/IP kernel networking layer overhead, specialized hardware can reach tens of GB/sec and if run in parallel, it can be potentially as fast as reading from the memory. In practice, the network-transfer speeds are somewhere between 1 to 10 GB/sec, but still faster than the disk in most practical systems. Thus, we can potentially fit the data into combined memory of all the cluster nodes and perform iterative machine learning algorithms across a system of them.

One problem with memory, however, is that it does not persist across node failures and reboots. A popular big data framework, Hadoop, made possible with the help of the original Dean/Ghemawat paper (Jeff Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI, 2004.), is using exactly the disk layer persistence to guarantee fault tolerance and store intermediate results. A Hadoop MapReduce program would first run a `map` function on each row of a dataset, emitting one or more key/value pairs. These key/value pairs then would be sorted, grouped, and aggregated by key so that the records with the same key would end up being processed together on the same reducer, which might be running on same or another node. The reducer applies a `reduce` function that traverses all the values that were emitted for the same key and aggregates them accordingly. The persistence of intermediate results would guarantee that if a reducer fails for one or another reason, the partial computations can be discarded and the reduce computation can be restarted from the checkpoint-saved results. Many simple ETL-like applications traverse the dataset only once with very little information preserved as state from one record to another.

For example, one of the traditional applications of MapReduce is word count. The program needs to count the number of occurrences of each word in a document consisting of lines of text. In Scala, the word count is readily expressed as an application of the `foldLeft` method on a sorted list of words:

```
val lines = scala.io.Source.fromFile("...").getLines.toSeq
val counts = lines.flatMap(line => line.split("\\W+")).sorted.
  foldLeft(List[(String,Int)]()){ (r,c) =>
    r match {
      case (key, count) :: tail =>
        if (key == c) (c, count+1) :: tail
        else (c, 1) :: r
      case Nil =>
        List((c, 1))
    }
}
```

If I run this program, the output will be a list of (word, count) tuples. The program splits the lines into words, sorts the words, and then matches each word with the latest entry in the list of (word, count) tuples. The same computation in MapReduce would be expressed as follows:

```
val linesRdd = sc.textFile("hdfs://...")
val counts = linesRdd.flatMap(line => line.split("\\W+"))
  .map(_.toLowerCase)
  .map(word => (word, 1)).
  .reduceByKey(_+_)
counts.collect
```

First, we need to process each line of the text by splitting the line into words and generation (word, 1) pairs. This task is easily parallelized. Then, to parallelize the global count, we need to split the counting part by assigning a task to do the count for a subset of words. In Hadoop, we compute the hash of the word and divide the work based on the value of the hash.

Once the map task finds all the entries for a given hash, it can send the key/value pairs to the reducer, the sending part is usually called shuffle in MapReduce vernacular. A reducer waits until it receives all the key/value pairs from all the mappers, combines the values – a partial combine can also happen on the mapper, if possible – and computes the overall aggregate, which in this case is just sum. A single reducer will see all the values for a given word.

Let's look at the log output of the word count operation in Spark (Spark is very verbose by default, you can manage the verbosity level by modifying the `conf/log4j.properties` file by replacing `INFO` with `ERROR` or `FATAL`):

```
$ wget http://mirrors.sonic.net/apache/spark/spark-1.6.1/spark-1.6.1-bin-hadoop2.6.tgz  
$ tar xvf spark-1.6.1-bin-hadoop2.6.tgz  
$ cd spark-1.6.1-bin-hadoop2.6  
$ mkdir leotolstoy  
$ (cd leotolstoy; wget http://www.gutenberg.org/files/1399/1399-0.txt)  
$ bin/spark-shell  
  
Welcome to
```

Using Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_40)

Type in expressions to have them evaluated.

Type :help for more information.

Spark context available as `sc`.

SQL context available as `sqlContext`.

```
scala> val linesRdd = sc.textFile("leotolstoy", minPartitions=10)
```

```
linesRdd: org.apache.spark.rdd.RDD[String] = leotolstoy
```

```
MapPartitionsRDD[3] at textFile at <console>:27
```

At this stage, the only thing that happened is metadata manipulations, Spark has not touched the data itself. Spark estimates that the size of the dataset and the number of partitions. By default, this is the number of HDFS blocks, but we can specify the minimum number of partitions explicitly with the `minPartitions` parameter:

```
scala> val countsRdd = linesRdd.flatMap(line => line.split("\\W+")).  
|   map(_.toLowerCase).  
|   map(word => (word, 1)).  
|   reduceByKey(_+_)  
countsRdd: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[5] at  
reduceByKey at <console>:31
```

We just defined another RDD derived from the original `linesRdd`:

```
scala> countsRdd.collect.filter(_.value > 99)
res3: Array[(String, Int)] = Array((been,1061), (them,841), (found,141),
(my,794), (often,105), (table,185), (this,1410), (here,364),
(asked,320), (standing,132), ("",13514), (we,592), (myself,140),
(is,1454), (carriage,181), (got,277), (won,153), (girl,117), (she,4403),
(moment,201), (down,467), (me,1134), (even,355), (come,667),
(new,319), (now,872), (upon,207), (sister,115), (veslovsky,110),
(letter,125), (women,134), (between,138), (will,461), (almost,124),
(thinking,159), (have,1277), (answer,146), (better,231), (men,199),
(after,501), (only,654), (suddenly,173), (since,124), (own,359),
(best,101), (their,703), (get,304), (end,110), (most,249), (but,3167),
(was,5309), (do,846), (keep,107), (having,153), (betsy,111), (had,3857),
(before,508), (saw,421), (once,334), (side,163), (ough...)
```

Word count over 2 GB of text data—40,291 lines and 353,087 words—took under a second to read, split, and group by words.

With extended logging, you could see the following:

- Spark opens a few ports to communicate with the executors and users
- Spark UI runs on port 4040 on `http://localhost:4040`
- You can read the file either from local or distributed storage (HDFS, Cassandra, and S3)
- Spark will connect to Hive if Spark is built with Hive support
- Spark uses lazy evaluation and executes the pipeline only when necessary or when output is required
- Spark uses internal scheduler to split the job into tasks, optimize the execution, and execute the tasks
- The results are stored into RDDs, which can either be saved or brought into RAM of the node executing the shell with `collect` method

The art of parallel performance tuning is to split the workload between different nodes or threads so that the overhead is relatively small and the workload is balanced.

Streaming word count

Spark supports listening on incoming streams, partitioning it, and computing aggregates close to real-time. Currently supported sources are Kafka, Flume, HDFS/S3, Kinesis, Twitter, as well as the traditional MQs such as ZeroMQ and MQTT. In Spark, streaming is implemented as micro-batches. Internally, Spark divides input data into micro-batches, usually from subseconds to minutes in size and performs RDD aggregation operations on these micro-batches.

For example, let's extend the Flume example that we covered earlier. We'll need to modify the Flume configuration file to create a Spark polling sink. Instead of HDFS, replace the sink section:

```
# The sink is Spark
a1.sinks.k1.type=org.apache.spark.streaming.flume.sink.SparkSink
a1.sinks.k1.hostname=localhost
a1.sinks.k1.port=4989
```

Now, instead of writing to HDFS, Flume will wait for Spark to poll for data:

```
object FlumeWordCount {
  def main(args: Array[String]) {
    // Create the context with a 2 second batch size
    val sparkConf = new SparkConf().setMaster("local[2]")
      .setAppName("FlumeWordCount")
    val ssc = new StreamingContext(sparkConf, Seconds(2))
    ssc.checkpoint("/tmp/flume_check")
    val hostPort=args(0).split(":")
    System.out.println("Opening a sink at host: [" + hostPort(0) +
      "] port: [" + hostPort(1).toInt + "]")
    val lines = FlumeUtils.createPollingStream(ssc, hostPort(0),
      hostPort(1).toInt, StorageLevel.MEMORY_ONLY)
    val words = lines
      .map(e => new String(e.event.getBody.array()))
      .map(_.toLowerCase).flatMap(_.split("\\W+"))
      .map(word => (word, 1L))
      .reduceByKeyAndWindow(_+_ , _-_ , Seconds(6) ,
        Seconds(2)).print
    ssc.start()
    ssc.awaitTermination()
  }
}
```

To run the program, start the Flume agent in one window:

```
$ ./bin/flume-ng agent -Dflume.log.level=DEBUG,console -n a1 -f ../
chapter03/conf/flume-spark.conf
...
```

Then run the FlumeWordCount object in another:

```
$ cd ../chapter03
$ sbt "run-main org.akozlov.chapter03.FlumeWordCount localhost:4989
...

```

Now, any text typed to the netcat connection will be split into words and counted every two seconds for a six second sliding window:

```
$ echo "Happy families are all alike; every unhappy family is unhappy in
its own way" | nc localhost 4987
...
-----
Time: 1464161488000 ms
-----
(are,1)
(is,1)
(its,1)
(family,1)
(families,1)
(alike,1)
(own,1)
(happy,1)
(unhappy,2)
(every,1)
...
-----
Time: 1464161490000 ms
-----
(are,1)
(is,1)
(its,1)
(family,1)
(families,1)
(alike,1)
(own,1)
(happy,1)
(unhappy,2)
(every,1)
...
```

Spark/Scala allows to seamlessly switch between the streaming sources. For example, the same program for Kafka publish/subscribe topic model looks similar to the following:

```
object KafkaWordCount {  
    def main(args: Array[String]) {  
        // Create the context with a 2 second batch size  
        val sparkConf = new SparkConf().setMaster("local[2]").  
            .setAppName("KafkaWordCount")  
        val ssc = new StreamingContext(sparkConf, Seconds(2))  
        ssc.checkpoint("/tmp/kafka_check")  
        System.out.println("Opening a Kafka consumer at zk:  
            [" + args(0) + "] for group group-1 and topic example")  
        val lines = KafkaUtils.createStream(ssc, args(0), "group-1",  
            Map("example" -> 1), StorageLevel.MEMORY_ONLY)  
        val words = lines  
            .flatMap(_.toLowerCase.split("\\W+"))  
            .map(word => (word, 1L))  
            .reduceByKeyAndWindow(_+_ , _-_ , Seconds(6),  
                Seconds(2)).print  
        ssc.start()  
        ssc.awaitTermination()  
    }  
}
```

To start the Kafka broker, first download the latest binary distribution and start ZooKeeper. ZooKeeper is a distributed-services coordinator and is required by Kafka even in a single-node deployment:

```
$ wget http://apache.cs.utah.edu/kafka/0.9.0.1/kafka_2.11-0.9.0.1.tgz  
...  
$ tar xf kafka_2.11-0.9.0.1.tgz  
$ bin/zookeeper-server-start.sh config/zookeeper.properties  
...
```

In another window, start the Kafka server:

```
$ bin/kafka-server-start.sh config/server.properties  
...
```

Run the KafkaWordCount object:

```
$ sbt "run-main org.akozlov.chapter03.KafkaWordCount localhost:2181"  
...
```

Now, publishing the stream of words into the Kafka topic will produce the window counts:

```
$ echo "Happy families are all alike; every unhappy family is unhappy  
in its own way" | ./bin/kafka-console-producer.sh --broker-list  
localhost:9092 --topic example  
...  
  
$ sbt "run-main org.akozlov.chapter03.FlumeWordCount localhost:4989  
...  
-----  
Time: 1464162712000 ms  
-----  
(are,1)  
(is,1)  
(its,1)  
(family,1)  
(families,1)  
(alike,1)  
(own,1)  
(happy,1)  
(unhappy,2)  
(every,1)
```

As you see, the programs output every two seconds. Spark streaming is sometimes called **micro-batch processing**. Streaming has many other applications (and frameworks), but this is too big of a topic to be entirely considered here and needs to be covered separately. I'll cover some ML on streams of data in *Chapter 5, Regression and Classification*. Now, let's get back to more traditional SQL-like interfaces.

Spark SQL and DataFrame

DataFrame was a relatively recent addition to Spark, introduced in version 1.3, allowing one to use the standard SQL language for data analysis. We already used some SQL commands in *Chapter 1, Exploratory Data Analysis* for the exploratory data analysis. SQL is really great for simple exploratory analysis and data aggregations.

According to the latest poll results, about 70% of Spark users use DataFrame. Although DataFrame recently became the most popular framework for working with tabular data, it is a relatively heavyweight object. The pipelines that use DataFrames may execute much slower than the ones that are based on Scala's vector or LabeledPoint, which will be discussed in the next chapter. The evidence from different developers is that the response times can be driven to tens or hundreds of milliseconds depending on the query, from submillisecond on simpler objects.

Spark implements its own shell for SQL, which can be invoked in addition to the standard Scala REPL shell: `./bin/spark-sql` can be used to access the existing Hive/Impala or relational DB tables:

```
$ ./bin/spark-sql
...
spark-sql> select min(duration), max(duration), avg(duration) from
kddcup;
...
0 58329 48.34243046395876
Time taken: 11.073 seconds, Fetched 1 row(s)
```

In standard Spark's REPL, the same query can be performed by running the following command:

```
$ ./bin/spark-shell
...
scala> val df = sqlContext.sql("select min(duration), max(duration),
avg(duration) from kddcup")
16/05/12 13:35:34 INFO parse.ParseDriver: Parsing command: select
min(duration), max(duration), avg(duration) from alex.kddcup_parquet
16/05/12 13:35:34 INFO parse.ParseDriver: Parse Completed
df: org.apache.spark.sql.DataFrame = [_c0: bigint, _c1: bigint, _c2:
double]
scala> df.collect.foreach(println)
...
16/05/12 13:36:32 INFO scheduler.DAGScheduler: Job 2 finished: collect at
<console>:22, took 4.593210 s
[0,58329,48.34243046395876]
```

ML libraries

Spark, particularly with memory-based storage systems, claims to substantially improve the speed of data access within and between nodes. ML seems to be a natural fit, as many algorithms require multiple passes over the data, or repartitioning. MLlib is the open source library of choice, although private companies are catching up with their own proprietary implementations.

As I will show in *Chapter 5, Regression and Classification*, most of the standard machine learning algorithms can be expressed as an optimization problem. For example, classical linear regression minimizes the sum of squares of y distance between the regression line and the actual value of y :

$$\frac{\partial}{\partial X} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

Here, \hat{y}_i are the predicted values according to the linear expression:

$$\hat{y} = A^T X + B$$

A is commonly called the slope, and B the intercept. In a more generalized formulation, a linear optimization problem is to minimize an additive function:

$$C(w) = \frac{1}{N} \sum_{i=1}^N L(w|x_i, y_i) + \lambda R(w)$$

Here, $L(w|x_i, y_i)$ is a loss function and $R(w)$ is a regularization function. The regularization function is an increasing function of model complexity, for example, the number of parameters (or a natural logarithm thereof). Most common loss functions are given in the following table:

	Loss function L	Gradient
Linear	$\frac{1}{2} (y_i - A^T X)^2$	$(A^T X - y_i)X$
Logistic	$\ln(1 + \exp(-yw^T x))$	$-y \left[1 - \frac{1}{1 + \exp(-yw^T x)} \right] x$

	Loss function L	Gradient
Hinge	$\max(0, 1 - yw^T x)$	$-yx \text{ if } yw^T x < 1, 0 \text{ otherwise}$

The purpose of the regularizer is to penalize more complex models to avoid overfitting and improve generalization error: more MLlib currently supports the following regularizers:

	Regularizer R	Gradient
L2	$\frac{1}{2} \ w\ _2^2$	w
L1	$\ w\ _1$	$sign(w)$
Elastic net	$\alpha \ w\ _1 + (1-\alpha) \frac{1}{2} \ w\ _2^2$	$\alpha sign(w) + (1-\alpha) w$

Here, $sign(w)$ is the vector of the signs of all entries of w .

Currently, MLlib includes implementation of the following algorithms:

- Basic statistics:
 - Summary statistics
 - Correlations
 - Stratified sampling
 - Hypothesis testing
 - Streaming significance testing
 - Random data generation
- Classification and regression:
 - Linear models (SVMs, logistic regression, and linear regression)
 - Naive Bayes
 - Decision trees
 - Ensembles of trees (Random Forests and Gradient-Boosted Trees)
 - Isotonic regression

- Collaborative filtering:
 - **Alternating least squares (ALS)**
- Clustering:
 - k-means
 - Gaussian mixture
 - **Power Iteration Clustering (PIC)**
 - **Latent Dirichlet allocation (LDA)**
 - Bisecting k-means
 - Streaming k-means
- Dimensionality reduction:
 - **Singular Value Decomposition (SVD)**
 - **Principal Component Analysis (PCA)**
- Feature extraction and transformation
- Frequent pattern mining:
 - FP-growth
 - Association rules
 - PrefixSpan
- Optimization:
 - **Stochastic Gradient Descent (SGD)**
 - **Limited-Memory BFGS (L-BFGS)**

I will go over some of the algorithms in *Chapter 5, Regression and Classification*. More complex non-structured machine learning methods will be considered in *Chapter 6, Working with Unstructured Data*.

SparkR

R is an implementation of popular S programming language created by John Chambers while working at Bell Labs. R is currently supported by the **R Foundation for Statistical Computing**. R's popularity has increased in recent years according to polls. SparkR provides a lightweight frontend to use Apache Spark from R. Starting with Spark 1.6.0, SparkR provides a distributed DataFrame implementation that supports operations such as selection, filtering, aggregation, and so on, which is similar to R DataFrames, dplyr, but on very large datasets. SparkR also supports distributed machine learning using MLlib.

SparkR required R version 3 or higher, and can be invoked via the `./bin/sparkR` shell. I will cover SparkR in *Chapter 8, Integrating Scala with R and Python*.

Graph algorithms – GraphX and GraphFrames

Graph algorithms are one of the hardest to correctly distribute between nodes, unless the graph itself is naturally partitioned, that is, it can be represented by a set of disconnected subgraphs. Since the social networking analysis on a multi-million node scale became popular due to companies such as Facebook, Google, and LinkedIn, researches have been coming up with new approaches to formalize the graph representations, algorithms, and types of questions asked.

GraphX is a modern framework for graph computations described in a 2013 paper (*GraphX: A Resilient Distributed Graph System on Spark* by Reynold Xin, Joseph Gonzalez, Michael Franklin, and Ion Stoica, GRADES (SIGMOD workshop), 2013). It has graph-parallel frameworks such as Pregel, and PowerGraph as predecessors. The graph is represented by two RDDs: one for vertices and another one for edges. Once the RDDs are joined, GraphX supports either Pregel-like API or MapReduce-like API, where the map function is applied to the node's neighbors and reduce is the aggregation step on top of the map results.

At the time of writing, GraphX includes the implementation for the following graph algorithms:

- PageRank
- Connected components
- Triangle counting
- Label propagation
- SVD++ (collaborative filtering)
- Strongly connected components

As GraphX is an open source library, changes to the list are expected. GraphFrames is a new implementation from Databricks that fully supports the following three languages: Scala, Java, and Python, and is build on top of DataFrames. I'll discuss specific implementations in *Chapter 7, Working with Graph Algorithms*.

Spark performance tuning

While efficient execution of the data pipeline is prerogative of the task scheduler, which is part of the Spark driver, sometimes Spark needs hints. Spark scheduling is primarily driven by the two parameters: CPU and memory. Other resources, such as disk and network I/O, of course, play an important part in Spark performance as well, but neither Spark, Mesos or YARN can currently do anything to actively manage them.

The first parameter to watch is the number of RDD partitions, which can be specified explicitly when reading the RDD from a file. Spark usually errs on the side of too many partitions as it provides more parallelism, and it does work in many cases as the task setup/teardown times are relatively small. However, one might experiment with decreasing the number of partitions, especially if one does aggregations.

The default number of partitions per RDD and the level of parallelism is determined by the `spark.default.parallelism` parameter, defined in the `$SPARK_HOME/conf/spark-defaults.conf` configuration file. The number of partitions for a specific RDD can also be explicitly changed by the `coalesce()` or `repartition()` methods.

The total number of cores and available memory is often the reason for deadlocks as the tasks cannot proceed further. One can specify the number of cores for each executor with the `--executor-cores` flag when invoking `spark-submit`, `spark-shell`, or `PySpark` from the command line. Alternatively, one can set the corresponding parameters in the `spark-defaults.conf` file discussed earlier. If the number of cores is set too high, the scheduler will not be able to allocate resources on the nodes and will deadlock.

In a similar way, `--executor-memory` (or the `spark.executor.memory` property) specifies the requested heap size for all the tasks (the default is 1g). If the executor memory is specified too high, again, the scheduler may be deadlocked or will be able to schedule only a limited number of executors on a node.

The implicit assumption in Standalone mode when counting the number of cores and memory is that Spark is the only running application—which may or may not be true. When running under Mesos or YARN, it is important to configure the cluster scheduler that it has the resources available to schedule the executors requested by the Spark Driver. The relevant YARN properties are: `yarn.nodemanager.resource.cpu-vcores` and `yarn.nodemanager.resource.memory-mb`. YARN may round the requested memory up a little. YARN's `yarn.scheduler.minimum-allocation-mb` and `yarn.scheduler.increment-allocation-mb` properties control the minimum and increment request values respectively.

JVMs can also use some memory off heap, for example, for interned strings and direct byte buffers. The value of the `spark.yarn.executor.memoryOverhead` property is added to the executor memory to determine the full memory request to YARN for each executor. It defaults to max (384, .07 * `spark.executor.memory`).

Since Spark can internally transfer the data between executors and client node, efficient serialization is very important. I will consider different serialization frameworks in *Chapter 6, Working with Unstructured Data*, but Spark uses Kryo serialization by default, which requires the classes to be registered explicitly in a static method. If you see a serialization error in your code, it is likely because the corresponding class has not been registered or Kryo does not support it, as it happens with too nested and complex data types. In general, it is recommended to avoid complex objects to be passed between the executors unless the object serialization can be done very efficiently.

Driver has similar parameters: `spark.driver.cores`, `spark.driver.memory`, and `spark.driver.maxResultSize`. The latter one sets the limit for the results collected from all the executors with the `collect` method. It is important to protect the driver process from out-of-memory exceptions. The other way to avoid out-of-memory exceptions and consequent problems are to either modify the pipeline to return aggregated or filtered results or use the `take` method instead.

Running Hadoop HDFS

A distributed processing framework wouldn't be complete without distributed storage. One of them is HDFS. Even if Spark is run on local mode, it can still use a distributed file system at the backend. Like Spark breaks computations into subtasks, HDFS breaks a file into blocks and stores them across a set of machines. For HA, HDFS stores multiple copies of each block, the number of copies is called replication level, three by default (refer to *Figure 3-5*).

NameNode is managing the HDFS storage by remembering the block locations and other metadata such as owner, file permissions, and block size, which are file-specific. **Secondary Namenode** is a slight misnomer: its function is to merge the metadata modifications, edits, into fsimage, or a file that serves as a metadata database. The merge is required, as it is more practical to write modifications of fsimage to a separate file instead of applying each modification to the disk image of the fsimage directly (in addition to applying the corresponding changes in memory). **Secondary Namenode** cannot serve as a second copy of the **Namenode**. A **Balancer** is run to move the blocks to maintain approximately equal disk usage across the servers—the initial block assignment to the nodes is supposed to be random, if enough space is available and the client is not run within the cluster. Finally, the **Client** communicates with the **Namenode** to get the metadata and block locations, but after that, either reads or writes the data directly to the node, where a copy of the block resides. The client is the only component that can be run outside the HDFS cluster, but it needs network connectivity with all the nodes in the cluster.

If any of the node dies or disconnects from the network, the **Namenode** notices the change, as it constantly maintains the contact with the nodes via heartbeats. If the node does not reconnect to the **Namenode** within 10 minutes (by default), the **Namenode** will start replicating the blocks in order to achieve the required replication level for the blocks that were lost on the node. A separate block scanner thread in the **Namenode** will scan the blocks for possible bit rot—each block maintains a checksum—and will delete corrupted and orphaned blocks:

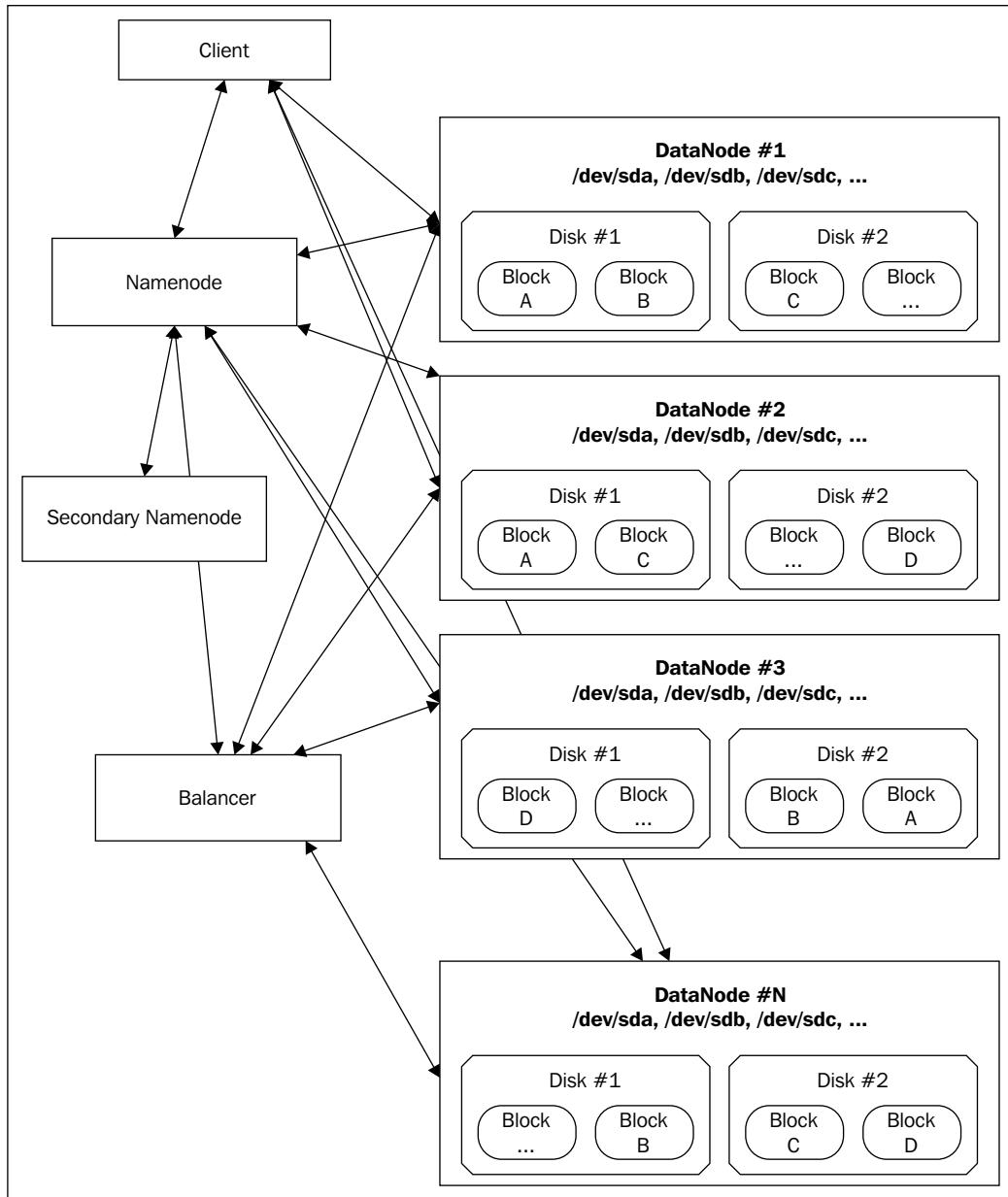


Figure 03-5. This is the HDFS architecture. Each block is stored in three separate locations (the replication level).

1. To start HDFS on your machine (with replication level 1), download a Hadoop distribution, for example, from <http://hadoop.apache.org>:

```
$ wget ftp://apache.cs.utah.edu/apache.org/hadoop/common/h/hadoop-2.6.4.tar.gz
--2016-05-12 00:10:55--  ftp://apache.cs.utah.edu/apache.org/hadoop/common/hadoop-2.6.4/hadoop-2.6.4.tar.gz
                         => 'hadoop-2.6.4.tar.gz.1'

Resolving apache.cs.utah.edu... 155.98.64.87
Connecting to apache.cs.utah.edu|155.98.64.87|:21... connected.
Logging in as anonymous ... Logged in!
==> SYST ... done.    ==> PWD ... done.
==> TYPE I ... done.  ==> CWD (1) /apache.org/hadoop/common/
hadoop-2.6.4 ... done.
==> SIZE hadoop-2.6.4.tar.gz ... 196015975
==> PASV ... done.    ==> RETR hadoop-2.6.4.tar.gz ... done.
...
$ wget ftp://apache.cs.utah.edu/apache.org/hadoop/common/
hadoop-2.6.4/hadoop-2.6.4.tar.gz.mds
--2016-05-12 00:13:58--  ftp://apache.cs.utah.edu/apache.org/
hadoop/common/hadoop-2.6.4/hadoop-2.6.4.tar.gz.mds
                         => 'hadoop-2.6.4.tar.gz.mds'

Resolving apache.cs.utah.edu... 155.98.64.87
Connecting to apache.cs.utah.edu|155.98.64.87|:21... connected.
Logging in as anonymous ... Logged in!
==> SYST ... done.    ==> PWD ... done.
==> TYPE I ... done.  ==> CWD (1) /apache.org/hadoop/common/
hadoop-2.6.4 ... done.
==> SIZE hadoop-2.6.4.tar.gz.mds ... 958
==> PASV ... done.    ==> RETR hadoop-2.6.4.tar.gz.mds ... done.
...
$ shasum -a 512 hadoop-2.6.4.tar.gz
493cc1a3e8ed0f7edee506d99bfabbe2aa71a4776e4bff5b852c6279b4c828a
0505d4ee5b63a0de0dcfecf70b4bb0ef801c767a068eaac938b8c58d8f21beec
hadoop-2.6.4.tar.gz
$ cat !$ .mds
hadoop-2.6.4.tar.gz:      MD5 = 37 01 9F 13 D7 DC D8 19  72 7B E1 58
44 0B 94 42
```

```
hadoop-2.6.4.tar.gz:   SHA1 = 1E02 FAAC 94F3 35DF A826  73AC BA3E  
7498 751A 3174  
hadoop-2.6.4.tar.gz: RMD160 = 2AA5 63AF 7E40 5DCD 9D6C  D00E EBB0  
750B D401 2B1F  
hadoop-2.6.4.tar.gz: SHA224 = F4FDFF12 5C8E754B DAF5BCFC 6735FCD2  
C6064D58  
                                36CB9D80 2C12FC4D  
hadoop-2.6.4.tar.gz: SHA256 = C58F08D2 E0B13035 F86F8B0B 8B65765A  
B9F47913  
                                81F74D02 C48F8D9C EF5E7D8E  
hadoop-2.6.4.tar.gz: SHA384 = 87539A46 B696C98E 5C7E352E 997B0AF8  
0602D239  
                                5591BF07 F3926E78 2D2EF790 BCBB6B3C  
EAF5B3CF  
                                ADA7B6D1 35D4B952  
hadoop-2.6.4.tar.gz: SHA512 = 493CC1A3 E8ED0F7E DEE506D9 9BFABBE2  
AA71A477  
                                6E4BFF5B 852C6279 B4C828A0 505D4EE5  
B63A0DE0  
                                DCFECF70 B4BB0EF8 01C767A0 68EAEAC9  
38B8C58D  
                                8F21BEEC
```



```
$ tar xf hadoop-2.6.4.tar.gz  
$ cd hadoop-2.6.4
```

2. To get the minimal HDFS configuration, modify the core-site.xml and hdfs-site.xml files, as follows:

```
$ cat << EOF > etc/hadoop/core-site.xml  
<configuration>  
    <property>  
        <name>fs.defaultFS</name>  
        <value>hdfs://localhost:8020</value>  
    </property>  
</configuration>  
EOF  
$ cat << EOF > etc/hadoop/hdfs-site.xml
```

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
EOF
```

This will put the Hadoop HDFS metadata and data directories under the /tmp/hadoop-\$USER directories. To make this more permanent, we can add the dfs.namenode.name.dir, dfs.namenode.edits.dir, and dfs.datanode.data.dir parameters, but we will leave these out for now. For a more customized distribution, one can download a Cloudera version from <http://archive.cloudera.com/cdh>.

3. First, we need to write an empty metadata:

```
$ bin/hdfs namenode -format
16/05/12 00:55:40 INFO namenode.NameNode: STARTUP_MSG:
/*****
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = alexanders-macbook-pro.local/192.168.1.68
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 2.6.4
STARTUP_MSG:   classpath =
...
...
```

4. Then start the namenode, secondarynamenode, and datanode Java processes (I usually open three different command-line windows to see the logs, but in a production environment, these are usually daemonized):

```
$ bin/hdfs namenode &
...
$ bin/hdfs secondarynamenode &
...
$ bin/hdfs datanode &
...
```

5. We are now ready to create the first HDFS file:

```
$ date | bin/hdfs dfs -put - date.txt
...
$ bin/hdfs dfs -ls
Found 1 items
-rw-r--r-- 1 akozlov supergroup 29 2016-05-12 01:02 date.txt
$ bin/hdfs dfs -text date.txt
Thu May 12 01:02:36 PDT 2016
```

6. Of course, in this particular case, the actual file is stored only on one node, which is the same node we run datanode on (localhost). In my case, it is the following:

```
$ cat /tmp/hadoop-akozlov/dfs/data/current/BP-1133284427-
192.168.1.68-1463039756191/current/finalized/subdir0/subdir0/
blk_1073741827
Thu May 12 01:02:36 PDT 2016
```

7. The Namenode UI can be found at <http://localhost:50070> and displays a host of information, including the HDFS usage and the list of DataNodes, the slaves of the HDFS Master node as follows:

The screenshot shows the HDFS NameNode UI interface. At the top, there is a navigation bar with tabs: Hadoop, Overview, Datanodes, Snapshot, Startup Progress, and Utilities. The 'Overview' tab is currently selected.

Overview section (localhost:8020 (active)):

Started:	Thu May 12 01:11:05 PDT 2016
Version:	2.6.4, r5082c73637530bb7e115f9625ed7fac69f937e6
Compiled:	2016-02-12T09:45Z by jenkins from (detached from 5082c73)
Cluster ID:	CID-d56fa807-3872-444d-be7f-1a6f8ee6fc01
Block Pool ID:	BP-1133284427-192.168.1.68-1463039756191

Summary section:

Security is off.
Safemode is off;
4 files and directories, 1 blocks = 5 total filesystem object(s).
Heap Memory used 62.97 MB of 356 MB Heap Memory. Max Heap Memory is 1.74 GB.
Non Heap Memory used 48.87 MB of 49.59 MB Committed Non Heap Memory. Max Non Heap Memory is -1 B.

Configured Capacity:	464.77 GB
DFS Used:	20 KB
Non DFS Used:	412.32 GB
DFS Remaining:	52.45 GB
DFS Used%:	0%
DFS Remaining%:	11.29%
Block Pool Used:	20 KB

Figure 03-6. A snapshot of HDFS NameNode UI.

The preceding figure shows HDFS Namenode HTTP UI in a single node deployment (usually, `http://<namenode-address>:50070`). The **Utilities | Browse the file system** tab allows you to browse and download the files from HDFS. Nodes can be added by starting DataNodes on a different node and pointing to the Namenode with the `fs.defaultFS=<namenode-address>:8020` parameter. The Secondary Namenode HTTP UI is usually at `http:<secondarynamenode-address>:50090`.

Scala/Spark by default will use the local file system. However, if the `core-site.xml` file is on the classpath or placed in the `$SPARK_HOME/conf` directory, Spark will use HDFS as the default.

Summary

In this chapter, I covered the Spark/Hadoop and their relationship with Scala and functional programming at a very high level. I considered a classic word count example and its implementation in Scala and Spark. I also provided high-level components of Spark ecosystem with specific examples of word count and streaming. I now have all the components to start looking at the specific implementation of classic machine learning algorithms in Scala/Spark. In the next chapter, I will start by covering supervised and unsupervised learning – a traditional division of learning algorithms for structured data.

4

Supervised and Unsupervised Learning

I covered the basics of the MLlib library in the previous chapter, but MLlib, at least at the time of writing this book, is more like a fast-moving target that is gaining the lead rather than a well-structured implementation that everyone uses in production or even has a consistent and tested documentation. In this situation, as people say, rather than giving you the fish, I will try to focus on well-established concepts behind the libraries and teach the process of fishing in this book in order to avoid the need to drastically modify the chapters with each new MLlib release. For better or worse, this increasingly seems to be a skill that a data scientist needs to possess.

Statistics and machine learning inherently deal with uncertainty, due to one or another reason we covered in *Chapter 2, Data Pipelines and Modeling*. While some datasets might be completely random, the goal here is to find trends, structure, and patterns beyond what a random number generator will provide you. The fundamental value of ML is that we can generalize these patterns and improve on at least some metrics. Let's see what basic tools are available within Scala/Spark.

In this chapter, I am covering supervised and unsupervised learning, the two historically different approaches. Supervised learning is traditionally used when we have a specific goal to predict a label, or a specific attribute of a dataset. Unsupervised learning can be used to understand internal structure and dependencies between any attributes of a dataset, and is often used to group the records or attributes in meaningful clusters. In practice, both methods can be used to complement and aid each other.

In this chapter, we will cover the following topics:

- Learning standard models for supervised learning – decision trees and logistic regression
- Discussing the staple of unsupervised learning – k-means clustering and its derivatives
- Understanding metrics and methods to evaluate the effectiveness of the above algorithms
- Having a glimpse of extending the above methods on special cases of streaming data, sparse data, and non-structured data

Records and supervised learning

For the purpose of this chapter, a record is an observation or measurement of one or several attributes. We assume that the observations might contain noise ε_{ij} (or be inaccurate for one or other reason):

$$x_i = \hat{x}_i + \varepsilon_{ij}$$

While we believe that there is some pattern or correlation between the attributes, the one that we are after and want to uncover, the noise is uncorrelated across either the attributes or the records. In statistical terms, we say that the values for each record are drawn from the same distribution and are independent (or *i.i.d.* in statistical terms). The order of records does not matter. One of the attributes, usually the first, might be designated to be the label.

Supervised learning is when the goal is to predict the label y_i :

$$y_i = f(x_1, \dots, x_N)$$

Here, N is the number of remaining attributes. In other words, the goal is to generalize the patterns so that we can predict the label by just knowing the other attributes, whether because we cannot physically get the measurement or just want to explore the structure of the dataset without having the immediate goal to predict the label.

The unsupervised learning is when we don't use the label – we just try to explore the structure and correlations to understand the dataset to, potentially, predict the label better. The number of problems in this latter category has increased recently with the emergence of learning for unstructured data and streams, each of which, I'll be covering later in the book in separate chapters.

Iris dataset

I will demonstrate the concept of records and labels based on one of the most famous datasets in machine learning, the Iris dataset (<https://archive.ics.uci.edu/ml/datasets/Iris>). The Iris dataset contains 50 records for each of the three types of Iris flower, 150 lines of total five fields. Each line is a measurement of the following:

- Sepal length in cm
- Sepal width in cm
- Petal length in cm
- Petal width in cm

With the final field being the type of the flower (*setosa*, *versicolor*, or *virginica*). The classic problem is to predict the label, which, in this case, is a categorical attribute with three possible values as a function of the first four attributes:

$$\text{label} = f(x_1, x_2, x_3, x_4)$$

One option would be to draw a plane in the four-dimensional space that separates all four labels. Unfortunately, as one can find out, while one of the classes is clearly separable, the remaining two are not, as shown in the following multidimensional scatterplot (we have used Data Desk software to create it):

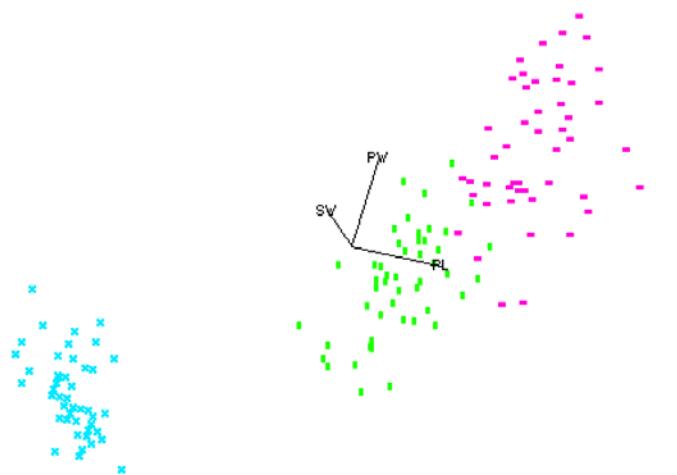


Figure 04-1. The Iris dataset as a three-dimensional plot. The Iris setosa records, shown by crosses, can be separated from the other two types based on petal length and width.

The colors and shapes are assigned according to the following table:

Label	Color	Shape
<i>Iris setosa</i>	Blue	x
<i>Iris versicolor</i>	Green	Vertical bar
<i>Iris virginica</i>	Purple	Horizontal bar

The *Iris setosa* is separable because it happens to have a very short petal length and width compared to the two other types.

Let's see how we can use MLlib to find that separating multidimensional plane.

Labeled point

The labeled datasets used to have a very special place in ML – we will discuss unsupervised learning later in the chapter, where we do not need a label, so MLlib has a special data type to represent a record with a `org.apache.spark.mllib.regression.LabeledPoint` label (refer to <https://spark.apache.org/docs/latest/mllib-data-types.html#labeled-point>). To read the Iris dataset from a text file, we need to transform the original UCI repository file into the so-called LIBSVM text format. While there are plenty of converters from CSV to LIBSVM format, I'd like to use a simple AWK script to do the job:

```
awk -F, '/setosa/ {print "0 1:\"$1" 2:\"$2" 3:\"$3" 4:\"$4;}; /versicolor/ {print "1 1:\"$1" 2:\"$2" 3:\"$3" 4:\"$4;}; /virginica/ {print "1 1:\"$1" 2:\"$2" 3:\"$3" 4:\"$4;};' iris.csv > iris-libsvm.txt
```

Why do we need LIBSVM format?

LIBSVM is the format that many libraries use. First, LIBSVM takes only continuous attributes. While a lot of datasets in the real world contain discrete or categorical attributes, internally they are always converted to a numerical representation for efficiency reasons, even if the L1 or L2 metrics on the resulting numerical attribute does not make much sense in the unordered discrete values. Second, the LIBSVM format allows for efficient sparse data representation. While the Iris dataset is not sparse, almost all of the modern big data sources are sparse, and the format allows for efficient storage by only storing the provided values. Many modern big data key-value and traditional RDBMS databases actually do the same for efficiency reasons.



The code might be more complex for missing values, but we know that the Iris dataset is not sparse – otherwise we'd complement our code with a bunch of if statements. We mapped the last two labels to 1 for our purpose now.

SVMWithSGD

Now, let's run the **Linear Support Vector Machine (SVM)** SVMWithSGD code from MLlib:

```
$ bin/spark-shell
Welcome to

    __
   / _\ \_ _ \_ / /_
  _\ \ \_ \ \_ \_ \_ / ' /
 / \_ / . \_ \_, _/ / / \_ \ version 1.6.1
 /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala> import org.apache.spark.mllib.classification.{SVMModel,
SVMWithSGD}
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
scala> import org.apache.spark.mllib.evaluation.
BinaryClassificationMetrics
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
scala> import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.util.MLUtils
scala> val data = MLUtils.loadLibSVMFile(sc, "iris-libsvm.txt")
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[6] at map at MLUtils.scala:112
scala> val splits = data.randomSplit(Array(0.6, 0.4), seed = 123L)
splits: Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.
regression.LabeledPoint]] = Array(MapPartitionsRDD[7] at randomSplit at
<console>:26, MapPartitionsRDD[8] at randomSplit at <console>:26)
scala> val training = splits(0).cache()
training: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[7] at randomSplit at <console>:26
scala> val test = splits(1)
```

```

test: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[8] at randomSplit at <console>:26
scala> val numIterations = 100
numIterations: Int = 100
scala> val model = SVMWithSGD.train(training, numIterations)
model: org.apache.spark.mllib.classification.SVMModel = org.apache.
spark.mllib.classification.SVMModel: intercept = 0.0, numFeatures = 4,
numClasses = 2, threshold = 0.0
scala> model.clearThreshold()
res0: model.type = org.apache.spark.mllib.classification.SVMModel:
intercept = 0.0, numFeatures = 4, numClasses = 2, threshold = None
scala> val scoreAndLabels = test.map { point =>
    |   val score = model.predict(point.features)
    |   (score, point.label)
    | }
scoreAndLabels: org.apache.spark.rdd.RDD[(Double, Double)] =
MapPartitionsRDD[212] at map at <console>:36
scala> val metrics = new BinaryClassificationMetrics(scoreAndLabels)
metrics: org.apache.spark.mllib.evaluation.BinaryClassificationMetrics =
org.apache.spark.mllib.evaluation.BinaryClassificationMetrics@692e4a35
scala> val auROC = metrics.areaUnderROC()
auROC: Double = 1.0

scala> println("Area under ROC = " + auROC)
Area under ROC = 1.0
scala> model.save(sc, "model")
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.

```

So, you just run one of the most complex algorithms in the machine learning toolbox: SVM. The result is a separating plane that distinguishes *Iris setosa* flowers from the other two types. The model in this case is exactly the intercept and the coefficients of the plane that best separates the labels:

```

scala> model.intercept
res5: Double = 0.0

scala> model.weights

```

```
res6: org.apache.spark.mllib.linalg.Vector = [-0.2469448809675877, -  
1.0692729424287566, 1.7500423423258127, 0.8105712661836376]
```

If one looks under the hood, the model is stored in a parquet file, which can be dumped using parquet-tool:

```
$ parquet-tools dump model/data/part-r-00000-7a86b825-569d-4c80-8796-  
8ee6972fd3b1.gz.parquet  
...  
DOUBLE weights.values.array  
-----  
-----  
*** row group 1 of 1, values 1 to 4 ***  
value 1: R:0 D:3 V:-0.2469448809675877  
value 2: R:1 D:3 V:-1.0692729424287566  
value 3: R:1 D:3 V:1.7500423423258127  
value 4: R:1 D:3 V:0.8105712661836376  
  
DOUBLE intercept  
-----  
-----  
*** row group 1 of 1, values 1 to 1 ***  
value 1: R:0 D:1 V:0.0  
...
```

The **Receiver Operating Characteristic (ROC)** is a common measure of the classifier to be able to correctly rank the records according to their numeric label. We will consider precision metrics in more detail in *Chapter 9, NLP in Scala*.

What is ROC?

ROC has emerged in signal processing with the first application to measure the accuracy of analog radars. The common measure of accuracy is area under ROC, which, shortly, is the probability of two randomly chosen points to be ranked correctly according to their labels (the 0 label should always have a lower rank than the 1 label). AUROC has a number of attractive characteristics:



- The value, at least theoretically, does not depend on the oversampling rate, that is, the rate at which we see 0 labels as opposed to 1 labels.
- The value does not depend on the sample size, excluding the expected variance due to the limited sample size.
- Adding a constant to the final score does not change the ROC, thus the intercept can always be set to 0. Computing the ROC requires a sort with respect to the generated score.

Of course, separating the remaining two labels is a harder problem since the plane that separated *Iris versicolor* from *Iris virginica* does not exist: the AUROC score will be less than 1.0. However, the SVM method will find the plane that best differentiates between the latter two classes.

Logistic regression

Logistic regression is one of the oldest classification methods. The outcome of the logistic regression is also a set of weights, which define the hyperplane, but the loss function is logistic loss instead of L2:

$$\ln\left(1 + \exp(-yw^T x)\right)$$

Logit function is a frequent choice when the label is binary (as $y = +/- 1$ in the above equation):

```
$ bin/spark-shell
Welcome to

   __
  / _\|_ _ \_ _ _ / /_
 _\ \V_ _ \V_ _ \_ / _/ ' _/
/_/_/ . _/ \_, _/_ / _/\_ \ version 1.6.1
   /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_40)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala> import org.apache.spark.SparkContext
import org.apache.spark.SparkContext
scala> import org.apache.spark.mllib.classification.{LogisticRegressionWithLBFGS, LogisticRegressionModel}
import org.apache.spark.mllib.classification.{LogisticRegressionWithLBFGS, LogisticRegressionModel}
scala> import org.apache.spark.mllib.evaluation.MulticlassMetrics
import org.apache.spark.mllib.evaluation.MulticlassMetrics
scala> import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LabeledPoint
scala> import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.Vectors
scala> import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.util.MLUtils
scala> val data = MLUtils.loadLibSVMFile(sc, "iris-libsvm-3.txt")
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] = MapPartitionsRDD[6] at map at MLUtils.scala:112
scala> val splits = data.randomSplit(Array(0.6, 0.4))
```

```

splits: Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.
regression.LabeledPoint]] = Array(MapPartitionsRDD[7] at randomSplit at
<console>:29, MapPartitionsRDD[8] at randomSplit at <console>:29)
scala> val training = splits(0).cache()
training: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[7] at randomSplit at <console>:29
scala> val test = splits(1)
test: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[8] at randomSplit at <console>:29
scala> val model = new LogisticRegressionWithLBFGS().setNumClasses(3).
run(training)
model: org.apache.spark.mllib.classification.LogisticRegressionModel =
org.apache.spark.mllib.classification.LogisticRegressionModel: intercept
= 0.0, numFeatures = 8, numClasses = 3, threshold = 0.5
scala> val predictionAndLabels = test.map { case LabeledPoint(label,
features) =>
    |   val prediction = model.predict(features)
    |   (prediction, label)
    | }
predictionAndLabels: org.apache.spark.rdd.RDD[(Double, Double)] =
MapPartitionsRDD[67] at map at <console>:37
scala> val metrics = new MulticlassMetrics(predictionAndLabels)
metrics: org.apache.spark.mllib.evaluation.MulticlassMetrics = org.
apache.spark.mllib.evaluation.MulticlassMetrics@6d5254f3
scala> val precision = metrics.precision
precision: Double = 0.9516129032258065
scala> println("Precision = " + precision)
Precision = 0.9516129032258065
scala> model.intercept
res5: Double = 0.0
scala> model.weights
res7: org.apache.spark.mllib.linalg.Vector = [10.644978886788556,-
26.850171485157578,3.852594349297618,8.74629386938248,4.288703063075211,-
31.029289381858273,9.790312529377474,22.058196856491996]

```

The labels in this case can be any integer in the range $[0, k]$, where k is the total number of classes (the correct class will be determined by building multiple binary logistic regression models against the pivot class, which in this case, is the class with the 0 label) (*The Elements of Statistical Learning* by Trevor Hastie, Robert Tibshirani, Jerome Friedman, Springer Series in Statistics).

The accuracy metric is precision, or the percentage of records predicted correctly (which is 95% in our case).

Decision tree

The preceding two methods describe linear models. Unfortunately, the linear approach does not always work for complex interactions between attributes. Assume that the label looks like an exclusive OR: 0 if $X \neq Y$ and 1 if $X = Y$:

X	Y	Label
1	0	0
0	1	0
1	1	1
0	0	1

There is no hyperplane that can differentiate between the two labels in the XY space. Recursive split solution, where the split on each level is made on only one variable or a linear combination thereof might work a bit better in these case. Decision trees are also known to work well with sparse and interaction-rich datasets:

```
$ bin/spark-shell
Welcome to

      _____
     /   /   \
    _\ \ \_ \ \_ \ \_ / / \
   /__/_ . __/ \_, _/ / / \_ \
   /_/
Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_40)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala> import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.DecisionTree
scala> import org.apache.spark.mllib.tree.model.DecisionTreeModel
```

```
import org.apache.spark.mllib.tree.model.DecisionTreeModel
scala> import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.util.MLUtils
scala> import org.apache.spark.mllib.tree.configuration.Strategy
import org.apache.spark.mllib.tree.configuration.Strategy
scala> import org.apache.spark.mllib.tree.configuration.Algo.
Classification
import org.apache.spark.mllib.tree.configuration.Algo.Classification
scala> import org.apache.spark.mllib.tree.impurity.{Entropy, Gini}
import org.apache.spark.mllib.tree.impurity.{Entropy, Gini}
scala> val data = MLUtils.loadLibSVMFile(sc, "iris-libsvm-3.txt")
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[6] at map at MLUtils.scala:112

scala> val splits = data.randomSplit(Array(0.7, 0.3), 11L)
splits: Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.
regression.LabeledPoint]] = Array(MapPartitionsRDD[7] at randomSplit at
<console>:30, MapPartitionsRDD[8] at randomSplit at <console>:30)
scala> val (trainingData, testData) = (splits(0), splits(1))
trainingData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[7] at randomSplit at <console>:30
testData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[8] at randomSplit at <console>:30
scala> val strategy = new Strategy(Classification, Gini, 10, 3, 10)
strategy: org.apache.spark.mllib.tree.configuration.Strategy = org.
apache.spark.mllib.tree.configuration.Strategy@4110e631
scala> val dt = new DecisionTree(strategy)
dt: org.apache.spark.mllib.tree.DecisionTree = org.apache.spark.mllib.
tree.DecisionTree@33d89052
scala> val model = dt.run(trainingData)
model: org.apache.spark.mllib.tree.model.DecisionTreeModel =
DecisionTreeModel classifier of depth 6 with 21 nodes
scala> val labelAndPreds = testData.map { point =>
    |   val prediction = model.predict(point.features)
    |   (point.label, prediction)
    | }
labelAndPreds: org.apache.spark.rdd.RDD[(Double, Double)] =
MapPartitionsRDD[32] at map at <console>:36
```

```
scala> val testErr = labelAndPreds.filter(r => r._1 != r._2).count.  
toDouble / testData.count()  
  
testErr: Double = 0.02631578947368421  
  
scala> println("Test Error = " + testErr)  
Test Error = 0.02631578947368421  
  
scala> println("Learned classification tree model:\n" + model.  
toDebugString)  
  
Learned classification tree model:  
DecisionTreeModel classifier of depth 6 with 21 nodes  
  If (feature 3 <= 0.4)  
    Predict: 0.0  
  Else (feature 3 > 0.4)  
    If (feature 3 <= 1.7)  
      If (feature 2 <= 4.9)  
        If (feature 0 <= 5.3)  
          If (feature 1 <= 2.8)  
            If (feature 2 <= 3.9)  
              Predict: 1.0  
            Else (feature 2 > 3.9)  
              Predict: 2.0  
          Else (feature 1 > 2.8)  
            Predict: 0.0  
        Else (feature 0 > 5.3)  
          Predict: 1.0  
      Else (feature 2 > 4.9)  
        If (feature 0 <= 6.0)  
          If (feature 1 <= 2.4)  
            Predict: 2.0  
          Else (feature 1 > 2.4)  
            Predict: 1.0  
        Else (feature 0 > 6.0)  
          Predict: 2.0  
      Else (feature 3 > 1.7)  
        If (feature 2 <= 4.9)  
          If (feature 1 <= 3.0)
```

```

Predict: 2.0
Else (feature 1 > 3.0)
Predict: 1.0
Else (feature 2 > 4.9)
Predict: 2.0
scala> model.save(sc, "dt-model")
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.

```

As you can see, the error (misprediction) rate on hold-out 30% sample is only 2.6%. The 30% sample of 150 is only 45 records, which means we missed only 1 record from the whole test set. Certainly, the result might and will change with a different seed, and we need a more rigorous cross-validation technique to prove the accuracy of the model, but this is enough for a rough estimate of model performance.

Decision tree generalizes on regression case, that is, when the label is continuous in nature. In this case, the splitting criterion is minimization of weighted variance, as opposed to entropy gain or gini in the case of classification. I will talk more about the differences in *Chapter 5, Regression and Classification*.

There are a number of parameters, which can be tuned to improve the performance:

Parameter	Description	Recommended value
maxDepth	This is the maximum depth of the tree. Deep trees are costly and usually are more likely to overfit. Shallow trees are more efficient and better for bagging/boosting algorithms such as AdaBoost.	This depends on the size of the original dataset. It is worth experimenting and plotting the accuracy of the resulting tree versus the parameter to find out the optimum.
minInstancesPerNode	This also limits the size of the tree: once the number of instances falls under this threshold, no further splitting occurs.	The value is usually 10-100, depending on the complexity of the original dataset and the number of potential labels.
maxBins	This is used only for continuous attributes: the number of bins to split the original range.	Large number of bins increase computation and communication cost. One can also consider the option of pre-discretizing the attribute based on domain knowledge.

Parameter	Description	Recommended value
minInfoGain	This is the amount of information gain (entropy), impurity (gini), or variance (regression) gain to split a node.	The default is 0, but you can increase the default to limit the tree size and reduce the risk of overfitting.
maxMemoryInMB	This is the amount of memory to be used for collecting sufficient statistics.	The default value is conservatively chosen to be 256 MB to allow the decision algorithm to work in most scenarios. Increasing maxMemoryInMB can lead to faster training (if the memory is available) by allowing fewer passes over the data. However, there may be decreasing returns as maxMemoryInMB grows, as the amount of communication on each iteration can be proportional to maxMemoryInMB.
subsamplingRate	This is the fraction of the training data used for learning the decision tree.	This parameter is most relevant for training ensembles of trees (using RandomForest and GradientBoostedTrees), where it can be useful to subsample the original data. For training a single decision tree, this parameter is less useful since the number of training instances is generally not the main constraint.
useNodeIdCache	If this is set to true, the algorithm will avoid passing the current model (tree or trees) to executors on each iteration.	This can be useful with deep trees (speeding up computation on workers) and for large random forests (reducing communication on each iteration).

Parameter	Description	Recommended value
checkpointDir:	This is the directory for checkpointing the node ID cache RDDs.	This is an optimization to save intermediate results to avoid recomputation in case of node failure. Set it in large clusters or with unreliable nodes.
checkpointInterval	This is the frequency for checkpointing the node ID cache RDDs.	Setting this too low will cause extra overhead from writing to HDFS and setting this too high can cause problems if executors fail and the RDD needs to be recomputed.

Bagging and boosting – ensemble learning methods

As a portfolio of stocks has better characteristics compared to individual equities, models can be combined to produce better classifiers. Usually, these methods work really well with decision trees as the training technique can be modified to produce models with large variations. One way is to train the model on random subsets of the original data or random subsets of attributes, which is called random forest.

Another way is to generate a sequence of models, where misclassified instances are reweighted to get a larger weight in each subsequent iteration. It has been shown that this method has a relation to gradient descent methods in the model parameter space. While these are valid and interesting techniques, they usually require much more space in terms of model storage and are less interpretable compared to bare decision tree models. For Spark, the ensemble models are currently under development – the umbrella issue is SPARK-3703 (<https://issues.apache.org/jira/browse/SPARK-3703>).

Unsupervised learning

If we get rid of the label in the Iris dataset, it would be nice if some algorithm could recover the original grouping, maybe without the exact label names – *setosa*, *versicolor*, and *virginica*. Unsupervised learning has multiple applications in compression and encoding, CRM, recommendation engines, and security to uncover internal structure without actually having the exact labels. The labels sometimes can be given base on the singularity in attribute value distributions. For example, *Iris setosa* can be described as a *Flower with Small Leaves*.

While a supervised learning problem can always be cast as unsupervised by disregarding the label, the reverse is also true. A clustering algorithm can be cast as a density-estimation problem by assigning label 1 to all vectors and generating random vectors with label 0 (*The Elements of Statistical Learning* by Trevor Hastie, Robert Tibshirani, Jerome Friedman, Springer Series in Statistics). The difference between the two is formal and it's even fuzzier with non-structured and nested data. Often, running unsupervised algorithms in labeled datasets leads to a better understanding of the dependencies and thus a better selection and performance of the supervised algorithm.

One of the most popular algorithms for clustering and unsupervised learning in k-means (and its variants, k-median and k-center, will be described later):

```
$ bin/spark-shell
Welcome to

    ___
   / _ \_  _ _ _ / /
  _\ \ \_ \ \_ `/_ / '
 /__ / .__/\_,/_/ /_/\_ \
 /_/
Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala> import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}
import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}
scala> import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.Vectors
scala> val iris = sc.textFile("iris.txt")
iris: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[4] at textFile
at <console>:23

scala> val vectors = data.map(s => Vectors.dense(s.split('\t').map(_.toDouble))).cache()
```

```
vectors: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] =  
MapPartitionsRDD[5] at map at <console>:25  
  
scala> val numClusters = 3  
numClusters: Int = 3  
scala> val numIterations = 20  
numIterations: Int = 20  
scala> val clusters = KMeans.train(vectors, numClusters, numIterations)  
clusters: org.apache.spark.mllib.clustering.KMeansModel = org.apache.  
spark.mllib.clustering.KMeansModel@5dc9cb99  
scala> val centers = clusters.clusterCenters  
centers: Array[org.apache.spark.mllib.linalg.Vector] =  
Array([5.005999999999999, 3.4180000000000006, 1.4640000000000002,  
0.2439999999999999], [6.8538461538461535, 3.076923076923076,  
5.715384615384614, 2.0538461538461537], [5.883606557377049,  
2.740983606557377, 4.388524590163936, 1.4344262295081966])  
scala> val SSE = clusters.computeCost(vectors)  
WSSSE: Double = 78.94506582597859  
scala> vectors.collect.map(x => clusters.predict(x))  
res18: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,  
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,  
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 2, 2, 1, 1, 1, 2, 1, 2, 1, 1, 2, 2, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1,  
1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2)  
scala> println("Sum of Squared Errors = " + SSE)  
Sum of Squared Errors = 78.94506582597859  
scala> clusters.save(sc, "model")  
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".  
SLF4J: Defaulting to no-operation (NOP) logger implementation  
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further  
details.
```

One can see that the first center, the one with index 0, has petal length and width of 1.464 and 0.244, which is much shorter than the other two – 5.715 and 2.054, 4.389 and 1.434). The prediction completely matches the first cluster, corresponding to *Iris setosa*, but has a few mispredictions for the other two.

The measure of cluster quality might depend on the (desired) labels if we want to achieve a desired classification result, but since the algorithm has no information about the labeling, a more common measure is the sum of distances from centroids to the points in each of the clusters. Here is a graph of WSSSE, depending on the number of clusters:

```
scala> 1.to(10).foreach(i => println("i: " + i + " SSE: " + KMeans.  
train(vectors, i, numIterations).computeCost(vectors)))  
i: 1 WSSSE: 680.8244  
i: 2 WSSSE: 152.3687064773393  
i: 3 WSSSE: 78.94506582597859  
i: 4 WSSSE: 57.47327326549501  
i: 5 WSSSE: 46.53558205128235  
i: 6 WSSSE: 38.9647878510374  
i: 7 WSSSE: 34.311167589868646  
i: 8 WSSSE: 32.607859500805034  
i: 9 WSSSE: 28.231729411088438  
i: 10 WSSSE: 29.435054384424078
```

As expected, the average distance is decreasing as more clusters are configured. A common method to determine the optimal number of clusters—in our example, we know that there are three types of flowers—is to add a penalty function. A common penalty is the log of the number of clusters as we expect a convex function. What would be the coefficient in front of log? If each vector is associated with its own cluster, the sum of all distances will be zero, so if we would like a metric that achieves approximately the same value at both ends of the set of possible values, 1 to 150, the coefficient should be $680.8244/\log(150)$:

```
scala> for (i <- 1.to(10)) println(i + " -> " + ((KMeans.train(vectors,  
i, numIterations).computeCost(vectors)) + 680 * scala.math.log(i) /  
scala.math.log(150)))  
1 -> 680.8244  
2 -> 246.436635016484  
3 -> 228.03498068120865  
4 -> 245.48126639400738  
5 -> 264.9805962616268  
6 -> 285.48857890531764  
7 -> 301.56808340425164  
8 -> 315.321639004243  
9 -> 326.47262191671723  
10 -> 344.87130979355675
```

Here is how the sum of the squared distances with penalty looks as a graph:

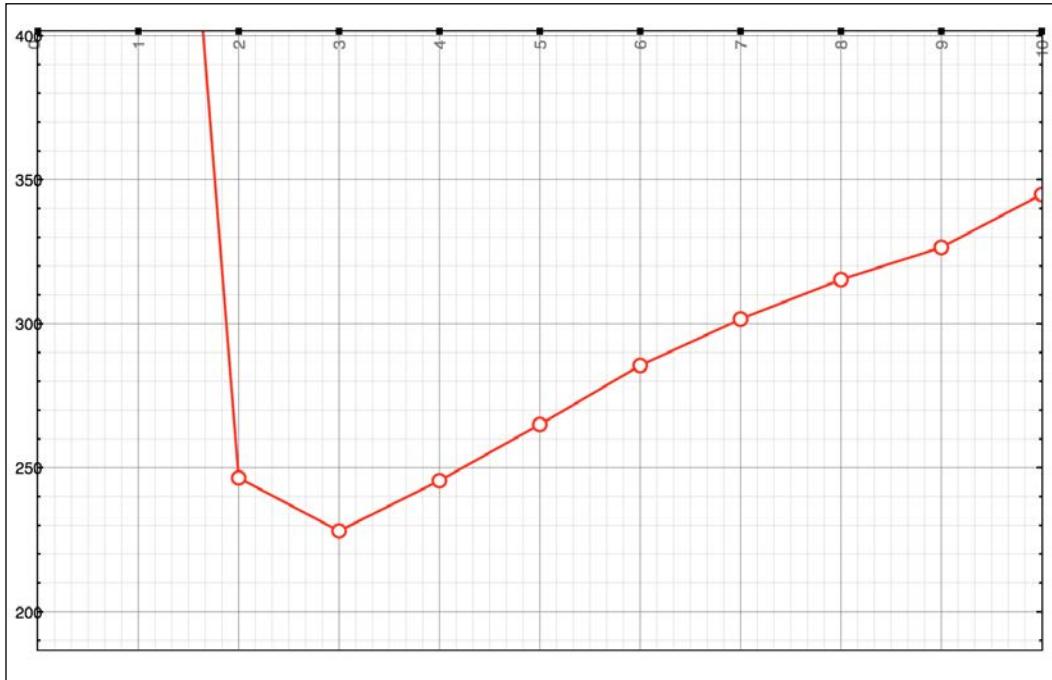


Figure 04-2. The measure of the clustering quality as a function of the number of clusters

Besides k-means clustering, MLlib also has implementations of the following:

- Gaussian mixture
- **Power Iteration Clustering (PIC)**
- **Latent Dirichlet Allocation (LDA)**
- Streaming k-means

The Gaussian mixture is another classical mechanism, particularly known for spectral analysis. Gaussian mixture decomposition is appropriate, where the attributes are continuous and we know that they are likely to come from a set of Gaussian distributions. For example, while the potential groups of points corresponding to clusters may have the average for all attributes, say **Var1** and **Var2**, the points might be centered around two intersecting hyperplanes, as shown in the following diagram:

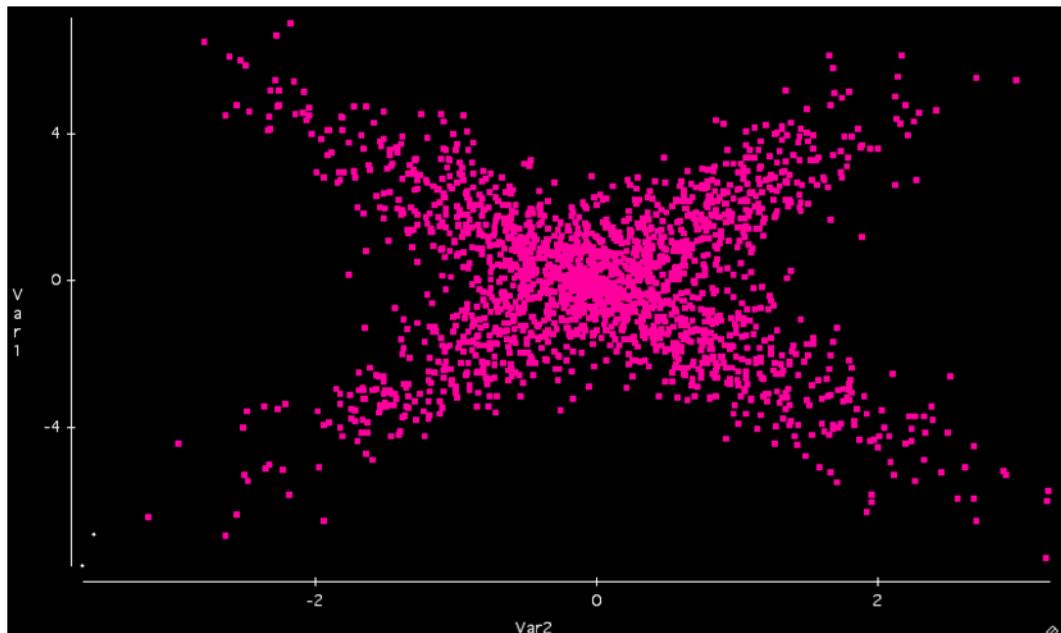


Figure 04-3. A mixture of two Gaussians that cannot be properly described by k-means clustering

This renders the k-means algorithm ineffective as it will not be able to distinguish between the two (of course a simple non-linear transformation such as a distance to one of the hyperplanes will solve the problem, but this is where domain knowledge and expertise as a data scientist are handy).

PIC is using clustering vertices of a graph provided pairwise similarity measures given as edge properties. It computes a pseudo-eigenvector of the normalized affinity matrix of the graph via power iteration and uses it to cluster vertices. MLlib includes an implementation of PIC using GraphX as its backend. It takes an RDD of (`srcId`, `dstId`, `similarity`) tuples and outputs a model with the clustering assignments. The similarities must be non-negative. PIC assumes that the similarity measure is symmetric. A pair (`srcId`, `dstId`) regardless of the ordering should appear at most once in the input data. If a pair is missing from the input, their similarity is treated as zero.

LDA can be used for clustering documents based on keyword frequencies. Rather than estimating a clustering using a traditional distance, LDA uses a function based on a statistical model of how text documents are generated.

Finally, streaming k-means is a modification of the k-means algorithm, where the clusters can be adjusted with new batches of data. For each batch of data, we assign all points to their nearest cluster, compute new cluster centers based on the assignment, and then update each cluster parameters using the equations:

$$c_{t+1} = \frac{a n_t c_t + n'_t c'_t}{a n_t + n'_t}$$

$$n_{t+1} = a n_t + n'_t$$

Here, c_t and c'_t are the centers of from the old model and the ones computed for the new batch and n_t and n'_t are the number of vectors from the old model and for the new batch. By changing the a parameter, we can control how much information from the old runs can influence the clustering—0 means the new cluster centers are totally based on the points in the new batch, while 1 means that we accommodate for all points that we have seen so far.

k-means clustering has many modifications. For example, k-medians computes the cluster centers as medians of the attribute values, not mean, which works much better for some distributions and with $L1$ target distance metric (absolute value of the difference) as opposed to $L2$ (the sum of squares). K-medians centers are not necessarily present as a specific point in the dataset. K-medoids is another algorithm from the same family, where the resulting cluster center has to be an actual instance in the input set and we actually do not need to have the global sort, only the pairwise distances between the points. Many variations of the techniques exist on how to choose the original seed cluster centers and converge on the optimal number of clusters (besides the simple log trick I have shown).

Another big class of clustering algorithms is hierarchical clustering. Hierarchical clustering is either done from the top – akin to the decision tree algorithms – or from the bottom; we first find the closest neighbors, pair them, and continue the pairing process up the hierarchy until all records are merged. The advantage of hierarchical clustering is that it can be made deterministic and relatively fast, even though the cost of one iteration in k-means is probably going to be better. However, as mentioned, the unsupervised problem can actually be converted to a density-estimation supervised problem, with all the supervised learning techniques available. So have fun understanding the data!

Problem dimensionality

The larger the attribute space or the number of dimensions, the harder it is to usually predict the label for a given combination of attribute values. This is mostly due to the fact that the total number of possible distinct combinations of attributes increases exponentially with the dimensionality of the attribute space—at least in the case of discrete variables (in case of continuous variables, the situation is more complex and depends on the metrics used), and it is becoming harder to generalize.

The effective dimensionality of the problem might be different from the dimensionality of the input space. For example, if the label depends only on the linear combination of the (continuous) input attributes, the problem is called linearly separable and its internal dimensionality is one – we still have to find the coefficients for this linear combination like in logistic regression though.

This idea is also sometimes referred to as a **Vapnik-Chervonenkis (VC)** dimension of a problem, model, or algorithm—the expressive power of the model depending on how complex the dependencies that it can solve, or shatter, might be. More complex problems require algorithms with higher VC dimensions and larger training sets. However, using an algorithm with higher VC dimension on a simple problem can lead to overfitting and worse generalization to new data.

If the units of input attributes are comparable, say all of them are meters or units of time, PCA, or more generally, kernel methods, can be used to reduce the dimensionality of the input space:

```
/_/  
  
Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java  
1.8.0_40)  
Type in expressions to have them evaluated.  
Type :help for more information.  
Spark context available as sc.  
SQL context available as sqlContext.  
  
scala> import org.apache.spark.mllib.regression.LabeledPoint  
import org.apache.spark.mllib.regression.LabeledPoint  
scala> import org.apache.spark.mllib.feature.PCA  
import org.apache.spark.mllib.feature.PCA  
scala> import org.apache.spark.mllib.util.MLUtils  
import org.apache.spark.mllib.util.MLUtils  
scala> val pca = new PCA(2).fit(data.map(_.features))  
pca: org.apache.spark.mllib.feature.PCAModel = org.apache.spark.mllib.  
feature.PCAModel@4eee0b1a  
  
scala> val reduced = data.map(p => p.copy(features = pca.transform(p.  
features)))  
reduced: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.  
LabeledPoint] = MapPartitionsRDD[311] at map at <console>:39  
scala> reduced.collect().take(10)  
res4: Array[org.apache.spark.mllib.regression.LabeledPoint] =  
Array((0.0, [-2.827135972679021, -5.641331045573367]), (0.0, [-  
2.7959524821488393, -5.145166883252959]), (0.0, [-2.621523558165053, -  
5.177378121203953]), (0.0, [-2.764905900474235, -5.0035994150569865]),  
(0.0, [-2.7827501159516546, -5.6486482943774305]), (0.0, [-  
3.231445736773371, -6.062506444034109]), (0.0, [-2.6904524156023393, -  
5.232619219784292]), (0.0, [-2.8848611044591506, -5.485129079769268]),  
(0.0, [-2.6233845324473357, -4.743925704477387]), (0.0, [-  
2.8374984110638493, -5.208032027056245]))  
  
scala> import scala.language.postfixOps  
import scala.language.postfixOps  
  
scala> pca.pc  
res24: org.apache.spark.mllib.linalg.DenseMatrix =
```

```
-0.36158967738145065 -0.6565398832858496
0.08226888989221656 -0.7297123713264776
-0.856572105290527 0.17576740342866465
-0.35884392624821626 0.07470647013502865

scala> import org.apache.spark.mllib.classification.{SVMModel,
SVMWithSGD}
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
scala> import org.apache.spark.mllib.evaluation.
BinaryClassificationMetrics
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
scala> val splits = reduced.randomSplit(Array(0.6, 0.4), seed = 1L)
splits: Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint]] = Array(MapPartitionsRDD[312] at randomSplit at
<console>:44, MapPartitionsRDD[313] at randomSplit at <console>:44)
scala> val training = splits(0).cache()
training: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[312] at randomSplit at <console>:44
scala> val test = splits(1)
test: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[313] at randomSplit at <console>:44
scala> val numIterations = 100
numIterations: Int = 100
scala> val model = SVMWithSGD.train(training, numIterations)
model: org.apache.spark.mllib.classification.SVMModel = org.apache.
spark.mllib.classification.SVMModel: intercept = 0.0, numFeatures = 2,
numClasses = 2, threshold = 0.0
scala> model.clearThreshold()
res30: model.type = org.apache.spark.mllib.classification.SVMModel:
intercept = 0.0, numFeatures = 2, numClasses = 2, threshold = None
scala> val scoreAndLabels = test.map { point =>
|   val score = model.predict(point.features)
|   (score, point.label)
| }
scoreAndLabels: org.apache.spark.rdd.RDD[(Double, Double)] =
MapPartitionsRDD[517] at map at <console>:54
scala> val metrics = new BinaryClassificationMetrics(scoreAndLabels)
```

```
metrics: org.apache.spark.mllib.evaluation.BinaryClassificationMetrics =  
org.apache.spark.mllib.evaluation.BinaryClassificationMetrics@27f49b8c  
  
scala> val auROC = metrics.areaUnderROC()  
auROC: Double = 1.0  
scala> println("Area under ROC = " + auROC)  
Area under ROC = 1.0
```

Here, we reduced the original four-dimensional problem to two-dimensional. Like averaging, computing linear combinations of input attributes and selecting only those that describe most of the variance helps to reduce noise.

Summary

In this chapter, we looked at supervised and unsupervised learning and a few examples of how to run them in Spark/Scala. We considered SVM, logistic regression, decision tree, and k-means in the example of UCI Iris dataset. This is in no way a complete guide, and many other libraries either exist or are being made as we speak, but I would bet that you can solve 99% of the immediate data analysis problems just with these tools.

This will give you a very fast shortcut on how to start being productive with a new dataset. There are many other ways to look at the datasets, but before we get into more advanced topics, let's discuss regression and classification in the next chapter, that is, how to predict continuous and discrete labels.

5

Regression and Classification

In the previous chapter, we got familiar with supervised and unsupervised learning. Another standard taxonomy of the machine learning methods is based on the label is from continuous or discrete space. Even if the discrete labels are ordered, there is a significant difference, particularly how the goodness of fit metrics is evaluated.

In this chapter, we will cover the following topics:

- Learning about the origin of the word regression
- Learning metrics for evaluating the goodness of fit in continuous and discrete space
- Discussing how to write simple code in Scala for linear and logistic regression
- Learning about advanced concepts such as regularization, multiclass predictions, and heteroscedasticity
- Discussing an example of MLlib application for regression tree analysis
- Learning about the different ways of evaluating classification models

What regression stands for?

While the word classification is intuitively clear, the word regression does not seem to imply a predictor of a continuous label. According to the Webster dictionary, regression is:

"a return to a former or less developed state."

It does also mention a special definition for statistics as *a measure of the relation between the mean value of one variable (for example, output) and corresponding values of other variables (for example, time and cost)*, which is actually correct these days. However, historically, the regression coefficient was meant to signify the hereditability of certain characteristics, such as weight and size, from one generation to another, with the hint of planned gene selection, including humans (<http://www.amstat.org/publications/jse/v9n3/stanton.html>). More specifically, in 1875, Galton, a cousin of Charles Darwin and an accomplished 19th-century scientist in his own right, which was also widely criticized for the promotion of eugenics, had distributed packets of sweet pea seeds to seven friends. Each friend received seeds of uniform weight, but with substantial variation across the seven packets. Galton's friends were supposed to harvest the next generation seeds and ship them back to him. Galton then proceeded to analyze the statistical properties of the seeds within each group, and one of the analysis was to plot the regression line, which always appeared to have the slope less than one – the specific number cited was 0.33 (Galton, F. (1894), *Natural Inheritance* (5th ed.), New York: Macmillan and Company), as opposed to either 0, in the case of no correlation and no inheritance; or 1, in the case the total replication of the parent's characteristics in the descendants. We will discuss why the coefficient of the regression line should always be less than 1 in the presence of noise in the data, even if the correlation is perfect. However, beyond the discussion and details, the origin of the term regression is partly due to planned breeding of plants and humans. Of course, Galton did not have access to PCA, Scala, or any other computing machinery at the time, which might shed more light on the differences between correlation and the slope of the regression line.

Continuous space and metrics

As most of this chapter's content will be dealing with trying to predict or optimize continuous variables, let's first understand how to measure the difference in a continuous space. Unless a drastically new discovery is made pretty soon, the space we live in is a three-dimensional Euclidian space. Whether we like it or not, this is the world we are mostly comfortable with today. We can completely specify our location with three continuous numbers. The difference in locations is usually measured by distance, or a metric, which is a function of a two arguments that returns a single positive real number. Naturally, the distance, $d^2(X, Y)$, between X and Y should always be equal or smaller than the sum of distances between X and Z and Y and Z:

$$d^2(X, Y) \leq d^2(X, Z) + d^2(Y, Z)$$

For any X , Y , and Z , which is also called triangle inequality. The two other properties of a metric is symmetry:

$$d^2(X, Y) = d^2(Y, X)$$

Non-negativity of distance:

$$d^2(X, Y) > 0 \text{ if } X \neq Y$$

$$d^2(X, Y) = 0 \text{ if } X = Y$$

Here, the metric is 0 if, and only if, $X=Y$. The L_2 distance is the distance as we understand it in everyday life, the square root of the sum of the squared differences along each of the dimensions. A generalization of our physical distance is p-norm ($p = 2$ for the L_2 distance):

$$d^p(X, Y) = \left(\sum_{i=1}^N |X_i - Y_i|^p \right)^{1/p}$$

Here, the sum is the overall components of the X and Y vectors. If $p=1$, the 1-norm is the sum of absolute differences, or Manhattan distance, as if the only path from point X to point Y would be to move only along one of the components. This distance is also often referred to as L_1 distance:

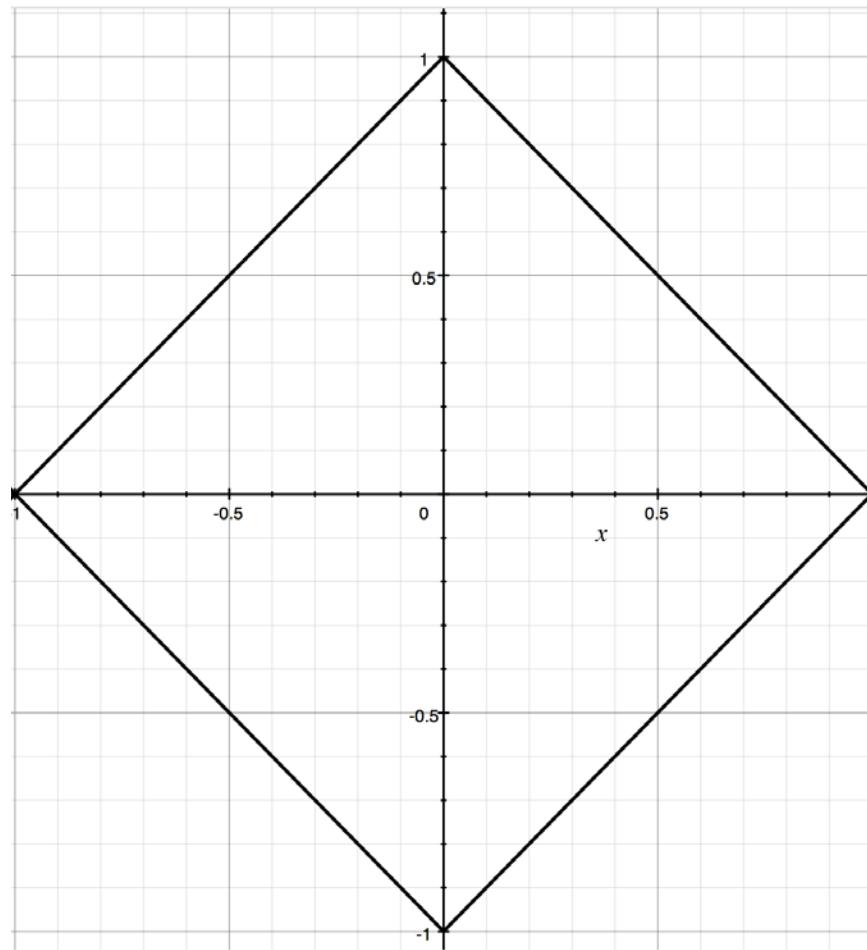


Figure 05-1. The L_1 circle in two-dimensional space (the set of points exactly one unit from the origin $(0, 0)$)

Here is a representation of a circle in a two-dimensional space:

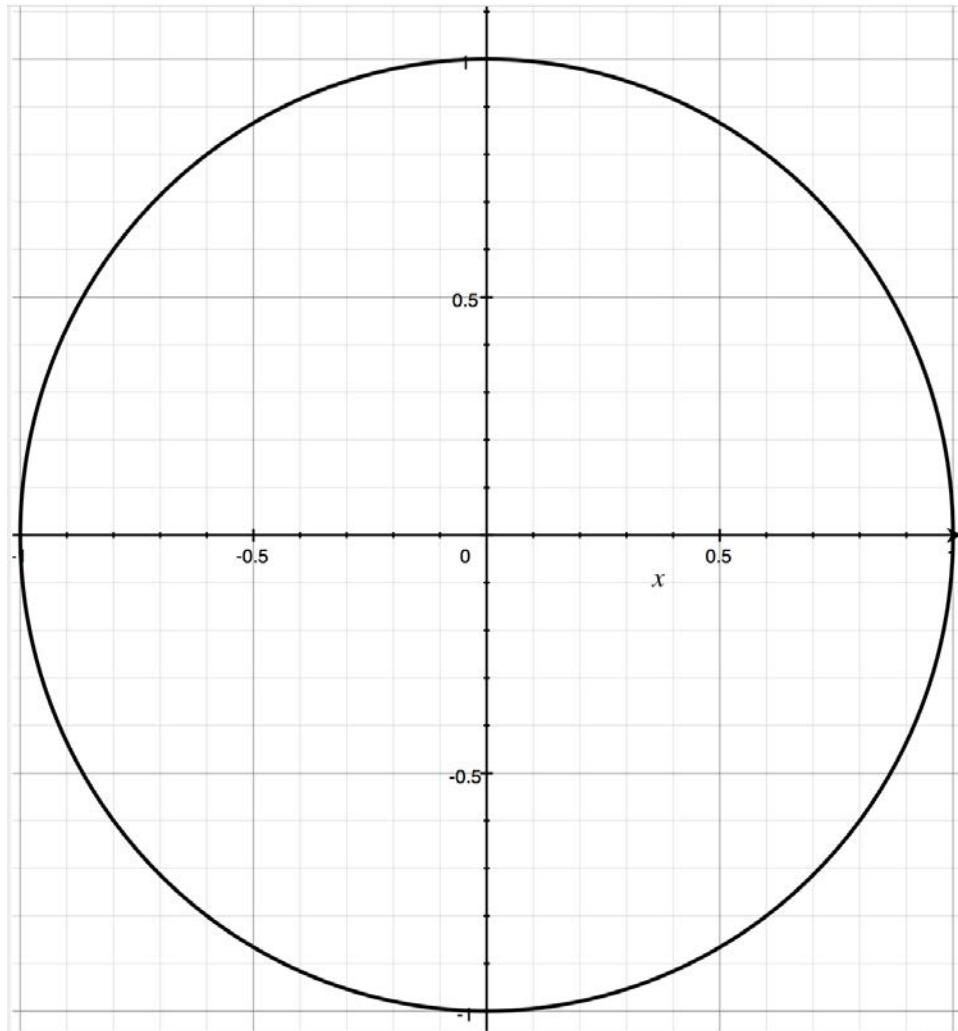


Figure 05-2. L_2 circle in two-dimensional space (the set of points equidistant from the origin $(0, 0)$), which actually looks like a circle in our everyday understanding of distance.

Another frequently used special case is L_∞ , the limit when $p \rightarrow \infty$, which is the maximum deviation along any of the components, as follows:

$$d^\infty(X, Y) = \max_i |X_i - Y_i|$$

The equidistant circle for the L_∞ distance is shown in *Figure 05-3*:

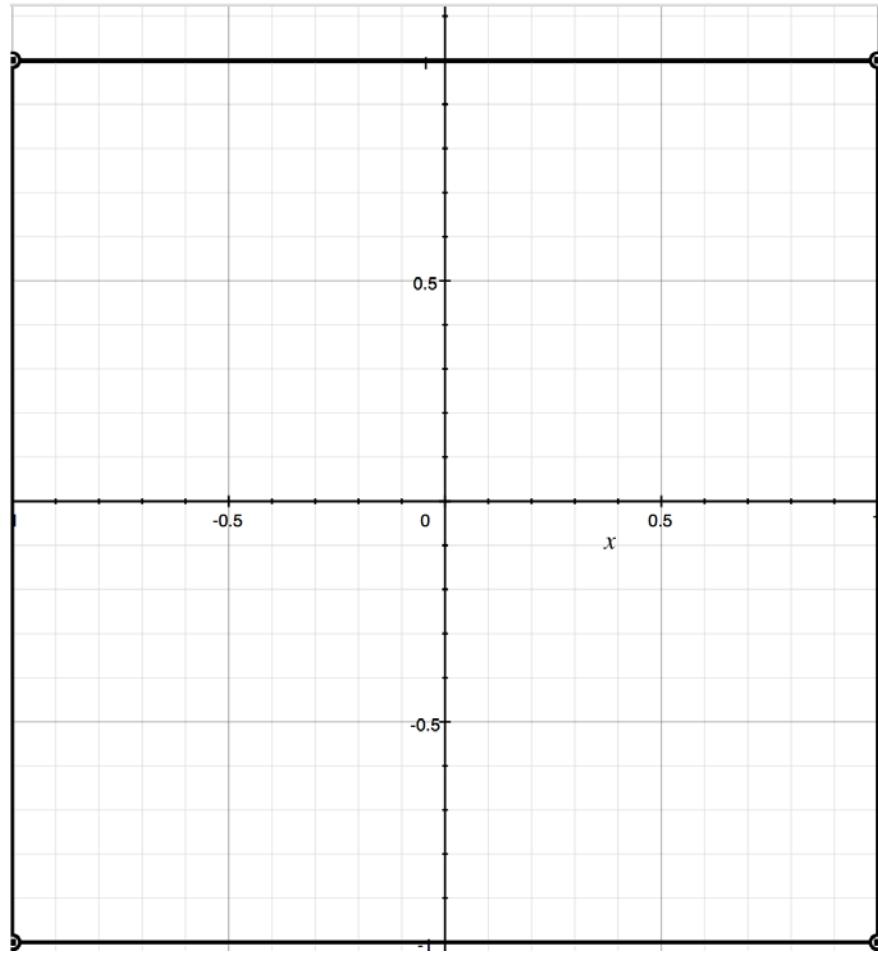


Figure 05-3. L_∞ circle in two-dimensional space (the set of points equidistant from the origin $(0, 0)$). This is a square as the L_∞ metric is the maximum distance along any of the components.

I'll consider the **Kullback-Leibler (KL)** distance later when I talk about classification, which measures the difference between two probability distributions, but it is an example of distance that is not symmetric and thus it is not a metric.

The metric properties make it easier to decompose the problem. Due to the triangle inequality, one can potentially reduce a difficult problem of optimizing a goal by substituting it by a set of problems by optimizing along a number of dimensional components of the problem separately.

Linear regression

As explained in *Chapter 2, Data Pipelines and Modeling*, most complex machine learning problems can be reduced to optimization as our final goal is to optimize the whole process where the machine is involved as an intermediary or the complete solution. The metric can be explicit, such as error rate, or more indirect, such as **Monthly Active Users (MAU)**, but the effectiveness of an algorithm is finally judged by how it improves some metrics and processes in our lives. Sometimes, the goals may consist of multiple subgoals, or other metrics such as maintainability and stability might eventually be considered, but essentially, we need to either maximize or minimize a continuous metric in one or other way.

For the rigor of the flow, let's show how the linear regression can be formulated as an optimization problem. The classical linear regression needs to optimize the cumulative L_2 error rate:

$$\text{params} = \underset{\text{params}}{\operatorname{argmin}} \sum_{i=1}^N (y_i - \hat{y})^2$$

Here, \hat{y} is the estimate given by a model, which, in the case of linear regression, is as follows:

$$\hat{y} = a x_i + b$$

(Other potential **loss functions** have been enumerated in *Chapter 3, Working with Spark and MLlib*). As the L_2 metric is a differentiable convex function of a, b , the extreme value can be found by equating the derivative of the cumulative error rate to 0:

$$0 = \frac{\partial d^2}{\partial a} = \frac{\partial d^2}{\partial b}$$

Computing the derivatives is straightforward in this case and leads to the following equation:

$$0 = \frac{\partial \sum_{i=1}^N (y_i - ax_i - b)^2}{\partial a} = 2 \sum_{i=1}^N (ax_i + b - y_i) x_i$$

$$0 = \frac{\partial \sum_{i=1}^N (y_i - ax_i - b)^2}{\partial b} = 2 \sum_{i=1}^N (ax_i + b - y_i)$$

This can be solved to give:

$$a = \frac{avg(x_i y_i) - avg(x_i) avg(y_i)}{avg(x_i^2) - avg(x_i)^2}$$

$$b = avg(y_i) - a avg(x_i)$$

Here, *avg()* denotes the average overall input records. Note that if *avg(x)=0* the preceding equation is reduced to the following:

$$a = \frac{avg(xy)}{avg(x^2)}$$

$$b = avg(y)$$

So, we can quickly compute the linear regression coefficients using basic Scala operators (we can always make *avg(x)* to be zero by performing a *x => x - avg(x)*):

akozlov@Alexanders-MacBook-Pro\$ scala

```
Welcome to Scala version 2.11.6 (Java HotSpot(TM) 64-Bit Server VM, Java  
1.8.0_40).
```

```
Type in expressions to have them evaluated.
```

```
Type :help for more information.
```

```
scala> import scala.util.Random
```

```
import scala.util.Random

scala> val x = -5 to 5
x: scala.collection.immutable.Range.Inclusive = Range(-5, -4, -3, -2, -1,
0, 1, 2, 3, 4, 5)

scala> val y = x.map(_ * 2 + 4 + Random.nextGaussian)
y: scala.collection.immutable.IndexedSeq[Double] =
Vector(-4.317116812989753, -4.4056031270948015, -2.0376543660274713,
0.0184679796245639, 1.8356532746253016, 3.2322795591658644,
6.821999810895798, 7.7977904139852035, 10.288549406814154,
12.424126535332453, 13.611442206874917)

scala> val a = (x, y).zipped.map(_ * _).sum / x.map(x => x * x).sum
a: Double = 1.9498665133868092

scala> val b = y.sum / y.size
b: Double = 4.115448625564203
```

Didn't I inform you previously that Scala is a very concise language? We just did linear regression with five lines of code, three of which were just data-generation statements.

Although there are libraries written in Scala for performing (multivariate) linear regression, such as Breeze (<https://github.com/scalanlp/breeze>), which provides a more extensive functionality, it is nice to be able to use pure Scala functionality to get some simple statistical results.

Let's look at the problem of Mr. Galton, where he found that the regression line always has the slope of less than one, which implies that we should always regress to some predefined mean. I will generate the same points as earlier, but they will be distributed along the horizontal line with some predefined noise. Then, I will rotate the line by 45 degrees by doing a linear rotation transformation in the xy -space. Intuitively, it should be clear that if anything, y is strongly correlated with x and absent, the y noise should be nothing else but x :

```
[akozlov@Alexanders-MacBook-Pro]$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40).
Type in expressions to have them evaluated.
```

Type :help for more information.

```
scala> import scala.util.Random.nextGaussian
import scala.util.Random.nextGaussian

scala> val x0 = Vector.fill(201)(100 * nextGaussian)
x0: scala.collection.immutable.IndexedSeq[Double] =
Vector(168.28831870102465, -40.56031270948016, -3.7654366027471324,
1.84679796245639, -16.43467253746984, -76.77204408341358,
82.19998108957988, -20.22095860147962, 28.854940681415442,
42.41265353324536, -38.85577931250823, -17.320873680820082,
64.19368427702135, -8.173507833084892, -198.6064655461397,
40.73700995880357, 32.36849515282444, 0.07758364225363915,
-101.74032407199553, 34.789280276495646, 46.29624756866302,
35.54024768650289, 24.7867839701828, -11.931948933554782,
72.12437623460166, 30.51440227306552, -80.20756177356768,
134.2380548346385, 96.14401034937691, -205.48142161773896,
-73.48186022765427, 2.7861465340245215, 39.49041527572774,
12.262899592863906, -118.30408039749234, -62.727048950163855,
-40.58557796128219, -23.42...

scala> val y0 = Vector.fill(201)(30 * nextGaussian)
y0: scala.collection.immutable.IndexedSeq[Double] =
Vector(-51.675658534203876, 20.230770706186128, 32.47396891906855,
-29.35028743620815, 26.7392929946199, 49.85681312583139,
24.226102932450917, 31.19021547086266, 26.169544117916704,
-4.51435617676279, 5.6334117227063985, -59.641661744341775,
-48.83082934374863, 29.655750956280304, 26.000847703123497,
-17.43319605936741, 0.8354318740518344, 11.44787080976254,
-26.26312164695179, 88.63863939038357, 45.795968719043785,
88.12442528090506, -29.829048945601635, -1.0417034396751037,
-27.119245702417494, -14.055969115249258, 6.120344305721601,
6.102779172838027, -6.342516875566529, 0.06774080659895702,
46.364626315486014, -38.473161588561, -43.25262339890197,
19.77322736359687, -33.78364440355726, -29.085765762613683,
22.87698648100551, 30.53...

scala> val x1 = (x0, y0).zipped.map((a,b) => 0.5 * (a + b) )
```

```
x1: scala.collection.immutable.IndexedSeq[Double] =
Vector(58.30633008341039, -10.164771001647015, 14.354266158160707,
-13.75174473687588, 5.152310228575029, -13.457615478791094,
53.213042011015396, 5.484628434691521, 27.51224239966607,
18.949148678241286, -16.611183794900917, -38.48126771258093,
7.681427466636357, 10.741121561597705, -86.3028089215081,
11.651906949718079, 16.601963513438136, 5.7627272260080895,
-64.00172285947366, 61.71395983343961, 46.0461081438534,
61.83233648370397, -2.5211324877094174, -6.486826186614943,
22.50256526609208, 8.229216578908131, -37.04360873392304,
70.17041700373827, 44.90074673690519, -102.70684040557,
-13.558616956084126, -17.843507527268237, -1.8811040615871129,
16.01806347823039, -76.0438624005248, -45.90640735638877,
-8.85429574013834, 3.55536787...)

scala> val y1 = (x0, y0).zipped.map((a,b) => 0.5 * (a - b) )

y1: scala.collection.immutable.IndexedSeq[Double] =
Vector(109.98198861761426, -30.395541707833143, -18.11970276090784,
15.598542699332269, -21.58698276604487, -63.31442860462248,
28.986939078564482, -25.70558703617114, 1.3426982817493691,
23.463504855004075, -22.244595517607316, 21.160394031760845,
56.51225681038499, -18.9146293946826, -112.3036566246316,
29.08510300908549, 15.7665316393863, -5.68514358375445,
-37.73860121252187, -26.924679556943964, 0.2501394248096176,
-26.292088797201085, 27.30791645789222, -5.445122746939839,
49.62181096850958, 22.28518569415739, -43.16395303964464,
64.06763783090022, 51.24326361247172, -102.77458121216895,
-59.92324327157014, 20.62965406129276, 41.37151933731485,
-3.755163885366482, -42.26021799696754, -16.820641593775086,
-31.73128222114385, -26.9...)

scala> val a = (x1, y1).zipped.map(_ * _).sum / x1.map(x => x * x).sum
a: Double = 0.8119662470457414
```

The slope is only 0.81! Note that if one runs PCA on the x1 and y1 data, the first principal component is correctly along the diagonal.

For completeness, I am giving a plot of $(x1, y1)$ zipped here:

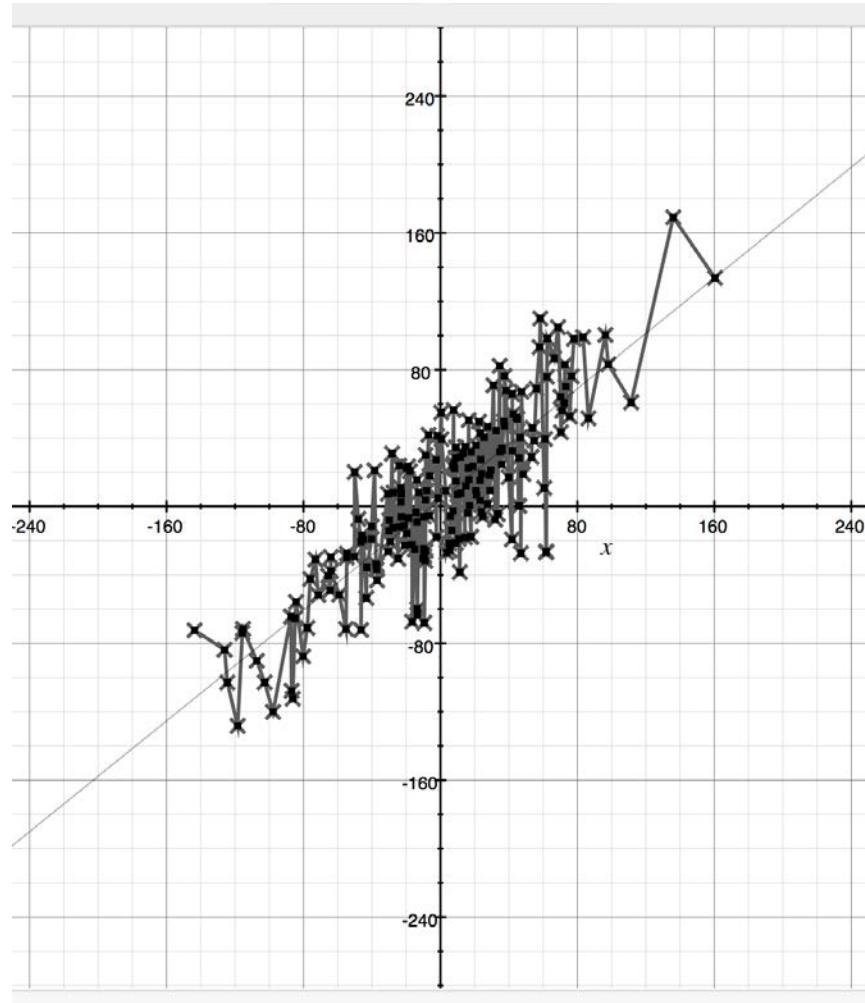


Figure 05-4. The regression curve slope of a seemingly perfectly correlated dataset is less than one. This has to do with the metric the regression problem optimizes (y-distance).

I will leave it to the reader to find the reason why the slope is less than one, but it has to do with the specific question the regression problem is supposed to answer and the metric it optimizes.

Logistic regression

Logistic regression optimizes the logit loss function with respect to w :

$$\ln\left(1 + \exp\left(-yw^T x\right)\right)$$

Here, y is binary (in this case plus or minus one). While there is no closed-form solution for the error minimization problem like there was in the previous case of linear regression, logistic function is differentiable and allows iterative algorithms that converge very fast.

The gradient is as follows:

$$\frac{\partial \ln(1 + \exp(-yw^T x))}{\partial w_j} = -\frac{\sum_{i=1}^N y_i x_{ij}}{(1 + \exp(yw^T x))}$$

Again, we can quickly concoct a Scala program that uses the gradient to converge to the value, where $\sum_{i=1}^N \ln(1 + \exp(-y_i w^T x_i)) = 0$ (we use the MLlib LabeledPoint data structure only for convenience of reading the data):

```
$ bin/spark-shell
Welcome to

   ___
  / _\|_ \  _ \  _ \| / /_ \
 _\ \ \/_ \/_ ^/ _/ ' _/
 /__/_ . __/\_._/_ / /_\ \    version 1.6.1-SNAPSHOT
 /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala> import org.apache.spark.mllib.linalg.Vector
```

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

scala> import org.apache.spark.util._
import org.apache.spark.util._

scala> import org.apache.spark.mllib.util._
import org.apache.spark.mllib.util._

scala> val data = MLUtils.loadLibSVMFile(sc, "data/iris/iris-libsvm.txt")
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.
LabeledPoint] = MapPartitionsRDD[291] at map at MLUtils.scala:112

scala> var w = Vector.random(4)
w: org.apache.spark.util.Vector = (0.9515155226069267,
0.4901713461728122, 0.4308861351586426, 0.8030814804136821)

scala> for (i <- 1.to(10)) println { val gradient = data.map(p => ( -
p.label / (1+scala.math.exp(p.label*(Vector(p.features.toDense.values)
dot w))) * Vector(p.features.toDense.values) )).reduce(_+_); w -= 0.1 *
gradient; w }
(-24.056553839570114, -16.585585503253142, -6.881629923278653,
-0.4154730884796032)
(38.56344616042987, 12.134414496746864, 42.178370076721365,
16.344526911520397)
(13.533446160429868, -4.95558550325314, 34.858370076721364,
15.124526911520398)
(-11.496553839570133, -22.045585503253143, 27.538370076721364,
13.9045269115204)
(-4.002010810020908, -18.501520148476196, 32.506256310962314,
15.455945245916512)
(-4.002011353029471, -18.501520429824225, 32.50625615219947,
15.455945209971787)
(-4.002011896036225, -18.501520711171313, 32.50625599343715,
15.455945174027184)
(-4.002012439041171, -18.501520992517463, 32.506255834675365,
15.455945138082699)
(-4.002012982044308, -18.50152127386267, 32.50625567591411,
15.455945102138333)
```

```
(-4.002013525045636, -18.501521555206942, 32.506255517153384,
15.455945066194088)
```

```
scala> w *= 0.24 / 4
w: org.apache.spark.util.Vector = (-0.24012081150273815,
-1.1100912933124165, 1.950375331029203, 0.9273567039716453)
```

The logistic regression was reduced to only one line of Scala code! The last line was to normalize the weights – only the relative values are important to define the separating plane – to compare them to the one obtained with the MLlib in previous chapter.

The **Stochastic Gradient Descent (SGD)** algorithm used in the actual implementation is essentially the same gradient descent, but optimized in the following ways:

- The actual gradient is computed on a subsample of records, which may lead to faster conversion due to less rounding noise and avoid local minima.
- The step – a fixed 0.1 in our case – is a monotonically decreasing function of the iteration as $\frac{1}{\sqrt(i)}$, which might also lead to better conversion.
- It incorporates regularization; instead of minimizing just the loss function, you minimize the sum of the loss function, plus some penalty metric, which is a function of model complexity. I will discuss this in the following section.

Regularization

The regularization was originally developed to cope with ill-poised problems, where the problem was underconstrained – allowed multiple solutions given the data – or the data and the solution that contained too much noise (*A.N. Tikhonov, A.S. Leonov, A.G. Yagola. Nonlinear Ill-Posed Problems, Chapman and Hall, London, Weinhe*). Adding additional penalty function that skews a solution if it does not have a desired property, such as the smoothness in curve fitting or spectral analysis, usually solves the problem.

The choice of the penalty function is somewhat arbitrary, but it should reflect a desired skew in the solution. If the penalty function is differentiable, it can be incorporated into the gradient descent process; ridge regression is an example where the penalty is the L_2 metric for the weights or the sum of squares of the coefficients.

MLlib currently implements L_2 , L_1 , and a mixture thereof called **Elastic Net**, as was shown in *Chapter 3, Working with Spark and MLlib*. The L_1 regularization effectively penalizes for the number of non-zero entries in the regression weights, but has been known to have slower convergence. **Least Absolute Shrinkage and Selection Operator (LASSO)** uses the L_1 regularization.

Another way to reduce the uncertainty in underconstrained problems is to take the prior information that may be coming from domain experts into account. This can be done using Bayesian analysis and introducing additional factors into the posterior probability – the probabilistic rules are generally expressed as multiplication rather than sum. However, since the goal is often minimizing the log likelihood, the Bayesian correction can often be expressed as standard regularizer as well.

Multivariate regression

It is possible to minimize multiple metrics at the same time. While Spark only has a few multivariate analysis tools, other more traditional well-established packages come with **Multivariate Analysis of Variance (MANOVA)**, a generalization of **Analysis of Variance (ANOVA)** method. I will cover ANOVA and MANOVA in *Chapter 7, Working with Graph Algorithms*.

For a practical analysis, we first need to understand if the target variables are correlated, for which we can use the PCA Spark implementation covered in *Chapter 3, Working with Spark and MLlib*. If the dependent variables are strongly correlated, maximizing one leads to maximizing the other, and we can just maximize the first principal component (and potentially build a regression model on the second component to understand what drives the difference).

If the targets are uncorrelated, building a separate model for each of them can pinpoint the important variables that drive either and whether these two sets are disjoint. In the latter case, we could build two separate models to predict each of the targets independently.

Heteroscedasticity

One of the fundamental assumptions in regression approach is that the target variance is not correlated with either independent (attributes) or dependent (target) variables. An example where this assumption might break is counting data, which is generally described by **Poisson distribution**. For Poisson distribution, the variance is proportional to the expected value, and the higher values can contribute more to the final variance of the weights.

While heteroscedasticity may or may not significantly skew the resulting weights, one practical way to compensate for heteroscedasticity is to perform a log transformation, which will compensate for it in the case of Poisson distribution:

$$y' = \log(y)$$

$$\text{var}(y') = \frac{\text{var}(y)}{y}$$

Some other (parametrized) transformations are the **Box-Cox transformation**:

$$y'_\lambda = \frac{y^\lambda - 1}{\lambda}$$

Here, λ is a parameter (the log transformation is a partial case, where $\lambda = 0$) and Tuckey's lambda transformation (for attributes between 0 and 1):

$$y'_\lambda = 0.5^L \left(y^L - (1-y)^L \right) / L$$

These compensate for Poisson binomial distributed attributes or the estimates of the probability of success in a sequence of trials with potentially a mix of n Bernoulli distributions.

Heteroscedasticity is one of the main reasons that logistic function minimization works better than linear regression with L_2 minimization in a binary prediction problem. Let's consider discrete labels in more details.

Regression trees

We have seen classification trees in the previous chapter. One can build a recursive split-and-concur structure for a regression problem, where a split is chosen to minimize the remaining variance. Regression trees are less popular than decision trees or classical ANOVA analysis; however, let's provide an example of a regression tree here as a part of MLlib:

```
akozlov@Alexanders-MacBook-Pro$ bin/spark-shell
Welcome to

    _\ _/ \
   / \ /_ \ \ /_ \ / \ / \
  / \ / . \ /_,/_/ /_/\_ \
 /_ / \_ \_ \_ \_ \_ \_ \_ \
                           version 1.6.1-SNAPSHOT

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala> import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.DecisionTree

scala> import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.tree.model.DecisionTreeModel

scala> import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.util.MLUtils

scala> // Load and parse the data file.

scala> val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
```

```
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.  
LabeledPoint] = MapPartitionsRDD[6] at map at MLUtils.scala:112  
  
scala> // Split the data into training and test sets (30% held out for  
testing)  
  
scala> val Array(trainingData, testData) = data.randomSplit(Array(0.7,  
0.3))  
trainingData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.  
LabeledPoint] = MapPartitionsRDD[7] at randomSplit at <console>:26  
testData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.  
LabeledPoint] = MapPartitionsRDD[8] at randomSplit at <console>:26  
  
scala> val categoricalFeaturesInfo = Map[Int, Int]()  
categoricalFeaturesInfo: scala.collection.immutable.Map[Int,Int] = Map()  
  
scala> val impurity = "variance"  
impurity: String = variance  
  
scala> val maxDepth = 5  
maxDepth: Int = 5  
  
scala> val maxBins = 32  
maxBins: Int = 32  
  
scala> val model = DecisionTree.trainRegressor(trainingData,  
categoricalFeaturesInfo, impurity, maxDepth, maxBins)  
model: org.apache.spark.mllib.tree.model.DecisionTreeModel =  
DecisionTreeModel regressor of depth 2 with 5 nodes  
  
scala> val labelsAndPredictions = testData.map { point =>  
|   val prediction = model.predict(point.features)  
|   (point.label, prediction)  
| }  
labelsAndPredictions: org.apache.spark.rdd.RDD[(Double, Double)] =  
MapPartitionsRDD[20] at map at <console>:36  
  
scala> val testMSE = labelsAndPredictions.map{ case(v, p) => math.pow((v  
- p), 2)}.mean()
```

```
testMSE: Double = 0.07407407407407407

scala> println(s"Test Mean Squared Error = $testMSE")
Test Mean Squared Error = 0.07407407407407407

scala> println("Learned regression tree model:\n" + model.toDebugString)
Learned regression tree model:
DecisionTreeModel regressor of depth 2 with 5 nodes
  If (feature 378 <= 71.0)
    If (feature 100 <= 165.0)
      Predict: 0.0
    Else (feature 100 > 165.0)
      Predict: 1.0
  Else (feature 378 > 71.0)
    Predict: 1.0
```

The splits at each level are made to minimize the variance, as follows:

$$var(x) = \sum_k N_k \left(\sum_{i=1}^{N_k} |x_i - avg(x)|^2 / N_k \right)^{1/2}$$

which is equivalent to minimizing the L_2 distances between the label values and their mean within each leaf summed over all the leaves of the node.

Classification metrics

If the label is discrete, the prediction problem is called classification. In general, the target can take only one of the values for each record (even though multivalued targets are possible, particularly for text classification problems to be considered in *Chapter 6, Working with Unstructured Data*).

If the discrete values are ordered and the ordering makes sense, such as *Bad*, *Worse*, *Good*, the discrete labels can be cast into integer or double, and the problem is reduced to regression (we believe if you are between *Bad* and *Worse*, you are definitely farther away from being *Good* than *Worse*).

A generic metric to optimize is the misclassification rate is as follows:

$$\text{error} = 1 - \sum_{i=1}^N \widehat{\text{if } y_i = \text{y}_i \text{ then } 1 \text{ else } 0}$$

However, if the algorithm can predict the distribution of possible values for the target, a more general metric such as the KL divergence or Manhattan can be used.

KL divergence is a measure of information loss when probability distribution P_1 is used to approximate probability distribution P_2 :

$$d^{KL}(P_1, P_2) = \sum P_1(i) \log \left(\frac{P_1(i)}{P_2(i)} \right)$$

It is closely related to entropy gain split criteria used in the decision tree induction, as the latter is the sum of KL divergences of the node probability distribution to the leaf probability distribution over all leaf nodes.

Multiclass problems

If the number of possible outcomes for target is larger than two, in general, we have to predict either the expected probability distribution of the target values or at least the list of ordered values – hopefully augmented by a rank variable, which can be used for additional analysis.

While some algorithms, such as decision trees, can natively predict multivalued attributes. A common technique is to reduce the prediction of one of the K target values to $(K-1)$ binary classification problems by choosing one of the values as the base and building $(K-1)$ binary classifiers. It is usually a good idea to select the most populated level as the base.

Perceptron

In the early days of machine learning, researchers were trying to imitate the functionality of the human brain. At the beginning of the 20th century, people thought that the human brain consisted entirely of cells that are called neurons – cells with long appendages called axons that were able to transmit signals by means of electric impulses. The AI researchers were trying to replicate the functionality of neurons by a perceptron, which is a function that is firing, based on a linearly-weighted sum of its input values:

$$y = \begin{cases} 1 & \text{if } w^T x > b \\ 0 & \text{otherwise} \end{cases}$$

This is a very simplistic representation of the processes in the human brain—biologists have since then discovered other ways in which information is transferred besides electric impulses such as chemical ones. Moreover, they have found over 300 different types of cells that may be classified as neurons (<http://neurolex.org/wiki/Category:Neuron>). Also, the process of neuron firing is more complex than just linear transmission of voltages as it involves complex time patterns as well. Nevertheless, the concept turned out to be very productive, and multiple algorithms and techniques were developed for neural nets, or the sets of perceptions connected to each other in layers. Specifically, it can be shown that the neural network, with certain modification, where the step function is replaced by a logistic function in the firing equation, can approximate an arbitrary differentiable function with any desired precision.

MLLib implements **Multilayer Perceptron Classifier (MLCP)** as an org.apache.spark.ml.classification.MultilayerPerceptronClassifier class:

```
Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_40)
Type in expressions to have them evaluated.
```

```
Type :help for more information.  
Spark context available as sc.  
SQL context available as sqlContext.  
  
scala> import org.apache.spark.ml.classification.  
MultilayerPerceptronClassifier  
import org.apache.spark.ml.classification.MultilayerPerceptronClassifier  
  
scala> import org.apache.spark.ml.evaluation.  
MulticlassClassificationEvaluator  
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator  
  
scala> import org.apache.spark.mllib.util.MLUtils  
import org.apache.mllib.util.MLUtils  
  
scala>  
  
scala> val data = MLUtils.loadLibSVMFile(sc, "iris-libsvm-3.txt").toDF()  
data: org.apache.spark.sql.DataFrame = [label: double, features: vector]  
  
scala>  
  
scala> val Array(train, test) = data.randomSplit(Array(0.6, 0.4), seed =  
13L)  
train: org.apache.spark.sql.DataFrame = [label: double, features: vector]  
test: org.apache.spark.sql.DataFrame = [label: double, features: vector]  
  
scala> // specify layers for the neural network:  
  
scala> // input layer of size 4 (features), two intermediate of size 5  
and 4 and output of size 3 (classes)  
  
scala> val layers = Array(4, 5, 4, 3)  
layers: Array[Int] = Array(4, 5, 4, 3)  
  
scala> // create the trainer and set its parameters
```

```
scala> val trainer = new MultilayerPerceptronClassifier().
setLayers(layers).setBlockSize(128).setSeed(13L).setMaxIter(100)
trainer: org.apache.spark.ml.classification.
MultilayerPerceptronClassifier = mlpc_b5f2c25196f9

scala> // train the model

scala> val model = trainer.fit(train)
model: org.apache.spark.ml.classification.
MultilayerPerceptronClassificationModel = mlpc_b5f2c25196f9

scala> // compute precision on the test set

scala> val result = model.transform(test)
result: org.apache.spark.sql.DataFrame = [label: double, features:
vector, prediction: double]

scala> val predictionAndLabels = result.select("prediction", "label")
predictionAndLabels: org.apache.spark.sql.DataFrame = [prediction:
double, label: double]

scala> val evaluator = new MulticlassClassificationEvaluator().
setMetricName("precision")
evaluator: org.apache.spark.ml.evaluation.
MulticlassClassificationEvaluator = mcEval_55757d35e3b0

scala> println("Precision = " + evaluator.evaluate(predictionAndLabels))
Precision = 0.9375
```

Generalization error and overfitting

So, how do we know that the model we have discussed is good? One obvious and ultimate criterion is its performance in practice.

One common problem that plagues the more complex models, such as decision trees and neural nets, is overfitting. The model can minimize the desired metric on the provided data, but does a very poor job on a slightly different dataset in practical deployments. Even a standard technique, when we split the dataset into training and test, the training for deriving the model and test for validating that the model works well on a hold-out data, may not capture all the changes that are in the deployments. For example, linear models such as ANOVA, logistic, and linear regression are usually relatively stable and less of a subject to overfitting. However, you might find that any particular technique either works or doesn't work for your specific domain.

Another case when generalization may fail is time-drift. The data may change over time significantly so that the model trained on the old data no longer generalizes on the new data in a deployment. In practice, it is always a good idea to have several models in production and constantly monitor their relative performance.

I will consider standard ways to avoid overfitting such as hold out datasets and cross-validation in *Chapter 7, Working with Graph Algorithms* and model monitoring in *Chapter 9, NLP in Scala*.

Summary

We now have all the necessary tools to look at more complex problems that are more commonly called the big data problems. Armed with standard statistical algorithms—I understand that I have not covered many details and I am completely ready to accept the criticism—there is an entirely new ground to explore where we do not have clearly defined records, the variables in the datasets may be sparse and nested, and we have to cover a lot of ground and do a lot of preparatory work just to get to the stage where we can apply the standard statistical models. This is where Scala shines best.

In the next chapter, we will look more at working with unstructured data.

6

Working with Unstructured Data

I am very excited to introduce you to this chapter. Unstructured data is what, in reality, makes big data different from the old data, it also makes Scala to be the new paradigm for processing the data. To start with, unstructured data at first sight seems a lot like a derogatory term. Notwithstanding, every sentence in this book is unstructured data: it does not have the traditional record / row / column semantics. For most people, however, this is the easiest thing to read rather than the book being presented as a table or spreadsheet.

In practice, the unstructured data means nested and complex data. An XML document or a photograph are good examples of unstructured data, which have very rich structure to them. My guess is that the originators of the term meant that the new data, the data that engineers at social interaction companies such as Google, Facebook, and Twitter saw, had a different structure to it as opposed to a traditional flat table that everyone used to see. These indeed did not fit the traditional RDBMS paradigm. Some of them can be flattened, but the underlying storage would be too inefficient as the RDBMSs were not optimized to handle them and also be hard to parse not only for humans, but for the machines as well.

A lot of techniques introduced in this chapter were created as an emergency Band-Aid to deal with the need to just process the data.

In this chapter, we will cover the following topics:

- Learning about the serialization, popular serialization frameworks, and language in which the machines talk to each other
- Learning about Avro-Parquet encoding for nested data
- Learning how RDBMs try to incorporate nested structures in modern SQL-like languages to work with them

- Learning how you can start working with nested structures in Scala
- Seeing a practical example of sessionization – one of the most frequent use cases for unstructured data
- Seeing how Scala traits and match/case statements can simplify path analysis
- Learning where the nested structures can benefit your analysis

Nested data

You already saw unstructured data in the previous chapters, the data was an array of **LabeledPoint**, which is a tuple (**label: Double, features: Vector**). The label is just a number of type **Double**. **Vector** is a sealed trait with two subclasses: **SparseVector** and **DenseVector**. The class diagram is as follows:

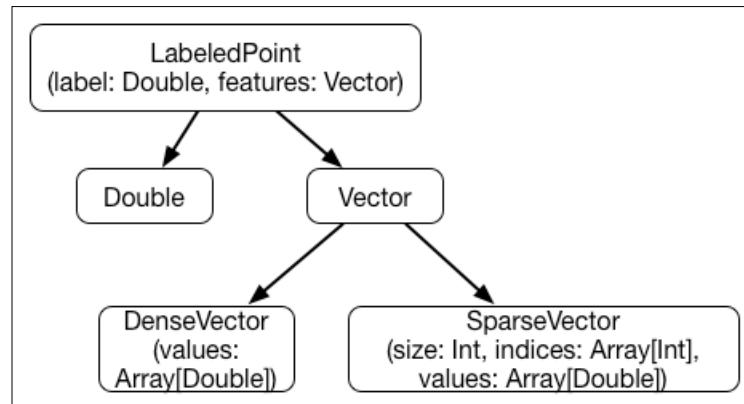


Figure 1: The **LabeledPoint** class structure is a tuple of label and features, where features is a trait with two inherited subclasses {Dense,Sparse}Vector. **DenseVector** is an array of double, while **SparseVector** stores only size and non-default elements by index and value.

Each observation is a tuple of label and features, and features can be sparse. Definitely, if there are no missing values, the whole row can be represented as vector. A dense vector representation requires $(8 \times \text{size} + 8)$ bytes. If most of the elements are missing – or equal to some default value – we can store only the non-default elements. In this case, we would require $(12 \times \text{non_missing_size} + 20)$ bytes, with small variations depending on the JVM implementation. So, the threshold for switching between one or another, from the storage point of view, is when the size is greater than $1.5 \times (\text{non_missing_size} + 1)$, or if roughly at least 30% of elements are non-default. While the computer languages are good at representing the complex structures via pointers, we need some convenient form to exchange these data between JVMs or machines. First, let's see first how Spark/Scala does it, specifically persisting the data in the Parquet format:

```
|     LabeledPoint(0.0, Vectors.sparse(3, Array(1), Array(1.0))),  
|     LabeledPoint(1.0, Vectors.dense(0.0, 2.0, 0.0)),  
|     LabeledPoint(2.0, Vectors.sparse(3, Array((1, 3.0)))),  
|     LabeledPoint.parse("(3.0,[0.0,4.0,0.0])));  
pts: Array[org.apache.spark.mllib.regression.LabeledPoint] =  
Array((0.0,(3,[1],[1.0])), (1.0,[0.0,2.0,0.0]), (2.0,(3,[1],[3.0])),  
(3.0,[0.0,4.0,0.0]))  
scala>  
  
scala> val rdd = sc.parallelize(points)  
rdd: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.  
LabeledPoint] = ParallelCollectionRDD[0] at parallelize at <console>:25  
  
scala>  
  
scala> val df = rdd.repartition(1).toDF  
df: org.apache.spark.sql.DataFrame = [label: double, features: vector]  
  
scala> df.write.parquet("points")
```

What we did was create a new RDD dataset from command line, or we could use `org.apache.spark.mllib.util.MLUtils` to load a text file, converted it to a `DataFrames` and create a serialized representation of it in the Parquet file under the `points` directory.

What Parquet stands for?

Apache Parquet is a columnar storage format, jointly developed by Cloudera and Twitter for big data. Columnar storage allows for better compression of values in the datasets and is more efficient if only a subset of columns need to be retrieved from the disk. Parquet was built from the ground up with complex nested data structures in mind and uses the record shredding and assembly algorithm described in the Dremel paper (<https://blog.twitter.com/2013/dremel-made-simple-with-parquet>). Dremel/Parquet encoding uses definition/repetition fields to denote the level in the hierarchy the data is coming from, which covers most of the immediate encoding needs, as it is sufficient to store optional fields, nested arrays, and maps. Parquet stores the data by chunks, thus probably the name Parquet, which means flooring composed of wooden blocks arranged in a geometric pattern. Parquet can be optimized for reading only a subset of blocks from disk, depending on the subset of columns to be read and the index used (although it very much depends on whether the specific implementation is aware of these features). The values in the columns can use dictionary and **Run-Length Encoding (RLE)**, which provides exceptionally good compression for columns with many duplicate entries, a frequent use case in big data.



Parquet file is a binary format, but you might look at the information in it using `parquet-tools`, which are downloadable from <http://archive.cloudera.com/cdh5/cdh/5>:

```
akozlov@Alexanders-MacBook-Pro$ wget -O - http://archive.cloudera.com/cdh5/cdh/5/parquet-1.5.0-cdh5.5.0.tar.gz | tar xzvf -
```

```
akozlov@Alexanders-MacBook-Pro$ cd parquet-1.5.0-cdh5.5.0/parquet-tools
```

```
akozlov@Alexanders-MacBook-Pro$ tar xvf xvf parquet-1.5.0-cdh5.5.0/parquet-tools/target/parquet-tools-1.5.0-cdh5.5.0-bin.tar.gz
```

```
akozlov@Alexanders-MacBook-Pro$ cd parquet-tools-1.5.0-cdh5.5.0
```

```
akozlov@Alexanders-MacBook-Pro $ ./parquet-schema ~/points/*.parquet
message spark_schema {
    optional double label;
    optional group features {
        required int32 type (INT_8);
        optional int32 size;
```

```
optional group indices (LIST) {
    repeated group list {
        required int32 element;
    }
}
optional group values (LIST) {
    repeated group list {
        required double element;
    }
}
}
```

Let's look at the schema, which is very close to the structure depicted in *Figure 1*: first member is the label of type double and the second and last one is features of composite type. The keyword optional is another way of saying that the value can be null (absent) in the record for one or another reason. The lists or arrays are encoded as a repeated field. As the whole array may be absent (it is possible for all features to be absent), it is wrapped into optional groups (indices and values). Finally, the type encodes whether it is a sparse or dense representation:

```
akozlov@Alexanders-MacBook-Pro $ ./parquet-dump ~/points/*.parquet
row group 0
-----
-----
-----
label:      DOUBLE GZIP DO:0 FPO:4 SZ:78/79/1.01 VC:4 ENC:BIT_
PACKED,PLAIN,RLE
features:
.type:      INT32 GZIP DO:0 FPO:82 SZ:101/63/0.62 VC:4 ENC:BIT_
PACKED,PLAIN_DICTIONARY,RLE
.size:      INT32 GZIP DO:0 FPO:183 SZ:97/59/0.61 VC:4 ENC:BIT_
PACKED,PLAIN_DICTIONARY,RLE
.indices:
..list:
...element: INT32 GZIP DO:0 FPO:280 SZ:100/65/0.65 VC:4 ENC:PLAIN_
DICTIONARY,RLE
.values:
..list:
```

```
...element: DOUBLE GZIP DO:0 FPO:380 SZ:125/111/0.89 VC:8 ENC:PLAIN_
DICTIONARY,RLE
```

```
label TV=4 RL=0 DL=1
```

```
-----  
-----  
-----  
page 0: DLE:RLE RLE:BIT_
PACKED VLE:PLAIN SZ:38 VC:4
```

```
features.type TV=4 RL=0 DL=1 DS: 2 DE:PLAIN_
DICTIONARY
```

```
-----  
-----  
-----  
page 0: DLE:RLE RLE:BIT_
PACKED VLE:PLAIN_DICTIONARY SZ:9 VC:4
```

```
features.size TV=4 RL=0 DL=2 DS: 1 DE:PLAIN_
DICTIONARY
```

```
-----  
-----  
-----  
page 0: DLE:RLE RLE:BIT_
PACKED VLE:PLAIN_DICTIONARY SZ:9 VC:4
```

```
features.indices.list.element TV=4 RL=1 DL=3 DS: 1 DE:PLAIN_
DICTIONARY
```

```
-----  
-----  
-----  
page 0: DLE:RLE RLE:RLE
VLE:PLAIN_DICTIONARY SZ:15 VC:4
```

```
features.values.list.element TV=8 RL=1 DL=3 DS: 5 DE:PLAIN_
DICTIONARY
```

```
page 0:                                     DLE:RLE RLE:RLE
VLE:PLAIN_DICTIONARY SZ:17 VC:8
```

```
DOUBLE label
```

```
*** row group 1 of 1, values 1 to 4 ***
value 1: R:0 D:1 V:0.0
value 2: R:0 D:1 V:1.0
value 3: R:0 D:1 V:2.0
value 4: R:0 D:1 V:3.0
```

```
INT32 features.type
```

```
*** row group 1 of 1, values 1 to 4 ***
value 1: R:0 D:1 V:0
value 2: R:0 D:1 V:1
value 3: R:0 D:1 V:0
value 4: R:0 D:1 V:1
```

```
INT32 features.size
```

```
*** row group 1 of 1, values 1 to 4 ***
value 1: R:0 D:2 V:3
value 2: R:0 D:1 V:<null>
value 3: R:0 D:2 V:3
value 4: R:0 D:1 V:<null>
```

```
INT32 features.indices.list.element
```

```
*** row group 1 of 1, values 1 to 4 ***
```

```
value 1: R:0 D:3 V:1
value 2: R:0 D:1 V:<null>
value 3: R:0 D:3 V:1
value 4: R:0 D:1 V:<null>

DOUBLE features.values.list.element
-----
-----
-----
*** row group 1 of 1, values 1 to 8 ***
value 1: R:0 D:3 V:1.0
value 2: R:0 D:3 V:0.0
value 3: R:1 D:3 V:2.0
value 4: R:1 D:3 V:0.0
value 5: R:0 D:3 V:3.0
value 6: R:0 D:3 V:0.0
value 7: R:1 D:3 V:4.0
value 8: R:1 D:3 V:0.0
```

You are probably a bit confused about the R: and D: in the output. These are the repetition and definition levels as described in the Dremel paper and they are necessary to efficiently encode the values in the nested structures. Only repeated fields increment the repetition level and only non-required fields increment the definition level. Drop in R signifies the end of the list(array). For every non-required level in the hierarchy tree, one needs a new definition level. Repetition and definition level values are small by design and can be efficiently stored in a serialized form.

What is best, if there are many duplicate entries, they will all be placed together. The case for which the compression algorithm (by default, it is gzip) are optimized. Parquet also implements other algorithms exploiting repeated values such as dictionary encoding or RLE compression.

This is a simple and efficient serialization out of the box. We have been able to write a set of complex objects to a file, each column stored in a separate block, representing all values in the records and nested structures.

Let's now read the file and recover RDD. The Parquet format does not know anything about the `LabeledPoint` class, so we'll have to do some typecasting and trickery here. When we read the file, we'll see a collection of `org.apache.spark.sql.Row`:

Personally, I think that this is pretty cool: without any compilation, we can encode and decide complex objects. One can easily create their own objects in REPL. Let's consider that we want to track user's behavior on the web:

```
akozlov@Alexanders-MacBook-Pro$ bin/spark-shell
Welcome to

    _\ _/ \
   _\ \ / _ \ \ _ \ / _ \ /
  /__/_ . __/\_,_/_/ /_/_\ \
                           version 1.6.1-SNAPSHOT
  /_/

Using Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala> case class Person(id: String, visits: Array[String]) { override
def toString: String = { val vsts = visits.mkString(","); s"($id ->
$vsts)" } }
defined class Person

scala> val p1 = Person("Phil", Array("http://www.google.com", "http://
www.facebook.com", "http://www.linkedin.com", "http://www.homedepot.
com"))
p1: Person = (Phil -> http://www.google.com,http://www.facebook.
com,http://www.linkedin.com,http://www.homedepot.com)

scala> val p2 = Person("Emily", Array("http://www.victoriasssecret.com",
"http://www.pacsun.com", "http://www.abercrombie.com/shop/us", "http://
www.orvis.com"))
p2: Person = (Emily -> http://www.victoriasssecret.com,http://www.pacsun.
com,http://www.abercrombie.com/shop/us,http://www.orvis.com)

scala> sc.parallelize(Array(p1,p2)).repartition(1).toDF.write.
parquet("history")

scala> import scala.collection.mutable.WrappedArray
```

```
import scala.collection.mutable.WrappedArray

scala> val df = sqlContext.read.parquet("history")
df: org.apache.spark.sql.DataFrame = [id: string, visits: array<string>]

scala> val rdd = df.map(x => Person(x(0).asInstanceOf[String], x(1).asInstanceOf[WrappedArray[String]]).toArray[String]))
rdd: org.apache.spark.rdd.RDD[Person] = MapPartitionsRDD[27] at map at
<console>:28

scala> rdd.collect
res9: Array[Person] = Array((Phil -> http://www.google.com,http://www.facebook.com,http://www.linkedin.com,http://www.homedepot.com), (Emily -> http://www.victoriassecret.com,http://www.pacsun.com,http://www.abercrombie.com/shop/us,http://www.orvis.com))
```

As a matter of good practice, we need to register the newly created classes with the Kryo serializer—Spark will use another serialization mechanism to pass the objects between tasks and executors. If the class is not registered, Spark will use default Java serialization, which might be up to $10 \times$ slower:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import com.esotericsoftware.kryo.Kryo
import org.apache.spark.serializer.{KryoSerializer, KryoRegistrar}

class MyKryoRegistrar extends KryoRegistrar {
    override def registerClasses(kryo: Kryo) {
        kryo.register(classOf[Person])
    }
}

object MyKryoRegistrar {
    def register(conf: org.apache.spark.SparkConf) {
        conf.set("spark.serializer", classOf[KryoSerializer].getName)
        conf.set("spark.kryo.registrator", classOf[MyKryoRegistrar].getName)
    }
}
```

```
}

^D

// Exiting paste mode, now interpreting.

import com.esotericsoftware.kryo.Kryo
import org.apache.spark.serializer.{KryoSerializer, KryoRegistrator}
defined class MyKryoRegistrator
defined module MyKryoRegistrator

scala>
```

If you are deploying the code on a cluster, the recommendation is to put this code in a jar on the classpath.

I've certainly seen examples of up to 10 level deep nesting in production. Although this might be an overkill for performance reasons, nesting is required in more and more production business use cases. Before we go into the specifics of constructing a nested object in the example of sessionization, let's get an overview of serialization in general.

Other serialization formats

I do recommend the Parquet format for storing the data. However, for completeness, I need to at least mention other serialization formats, some of them like Kryo will be used implicitly for you during Spark computations without your knowledge and there is obviously a default Java serialization.

Object-oriented approach versus functional approach

Objects in object-oriented approach are characterized by state and behavior. Objects are the cornerstone of object-oriented programming. A class is a template for objects with fields that represent the state, and methods that may represent the behavior. Abstract method implementation may depend on the instance of the class. In functional approach, the state is usually frowned upon; in pure programming languages, there should be no state, no side effects, and every invocation should return the same result. The behaviors may be expressed through additional function parameters and higher order functions (functions over functions, such as currying), but should be explicit unlike the abstract methods. Since Scala is a mix of object-oriented and functional language, some of the preceding constraints are violated, but this does not mean that you have to use them unless absolutely necessary. It is best practice to store the code in jar packages while storing the data, particularly for the big data, separate from code in data files (in a serialized form); but again, people often store data/configurations in jar files, and it is less common, but possible to store code in the data files.



The serialization has been an issue since the need to persist data on disk or transfer object from one JVM or machine to another over network appeared. Really, the purpose of serialization is to make complex nested objects be represented as a series of bytes, understandable by machines, and as you can imagine, this might be language-dependent. Luckily, serialization frameworks converge on a set of common data structures they can handle.

One of the most popular serialization mechanisms, but not the most efficient, is to dump an object in an ASCII file: CSV, XML, JSON, YAML, and so on. They do work for more complex nested data like structures, arrays, and maps, but are inefficient from the storage space perspective. For example, a Double represents a continuous number with 15-17 significant digits that will, without rounding or trivial ratios, take 15-17 bytes to represent in US ASCII, while the binary representation takes only 8 bytes. Integers may be stored even more efficiently, particularly if they are small, as we can compress/remove zeroes.

One advantage of text encoding is that they are much easier to visualize with simple command-line tools, but any advanced serialization framework now comes with a set of tools to work with raw records such as `avro-tools` or `parquet-tools`.

The following table provides an overview for most common serialization frameworks:

Serialization Format	When developed	Comments
XML, JSON, YAML	This was a direct response to the necessity to encode nested structures and exchange the data between machines.	While grossly inefficient, these are still used in many places, particularly in web services. The only advantage is that they are relatively easy to parse without machines.
Protobuf	Developed by Google in the early 2000s. This implements the Dremel encoding scheme and supports multiple languages (Scala is not officially supported yet, even though some code exists).	The main advantage is that Protobuf can generate native classes in many languages. C++, Java, and Python are officially supported. There are ongoing projects in C, C#, Haskell, Perl, Ruby, Scala, and more. Run-time can call native code to inspect/serialize/deserialize the objects and binary representations.
Avro	Avro was developed by Doug Cutting while he was working at Cloudera. The main objective was to separate the encoding from a specific implementation and language, allowing better schema evolution.	While the arguments whether Protobuf or Avro are more efficient are still ongoing, Avro supports a larger number of complex structures, say unions and maps out of the box, compared to Protobuf. Scala support is still to be strengthened to the production level. Avro files have schema encoded with every file, which has its pros and cons.

Serialization Format	When developed	Comments
Thrift	The Apache Thrift was developed at Facebook for the same purpose Protobuf was developed. It probably has the widest selection of supported languages: C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml, Delphi, and other languages. Again, Twitter is hard at work for making the Thrift code generation in Scala (https://twitter.github.io/scrooge/).	Apache Thrift is often described as a framework for cross-language services development and is most frequently used as Remote Procedure Call (RPC) . Even though it can be used directly for serialization/deserialization, other frameworks just happen to be more popular.
Parquet	Parquet was developed in a joint effort between Twitter and Cloudera. Compared to the Avro format, which is row-oriented, Parquet is columnar storage that results in better compression and performance if only a few columns are to be selected. The interval encoding is Dremel or Protobuf-based, even though the records are presented as Avro records; thus, it is often called AvroParquet .	Advances features such as indices, dictionary encoding, and RLE compression potentially make it very efficient for pure disk storage. Writing the files may be slower as Parquet requires some preprocessing and index building before it can be committed to the disk.
Kryo	This is a framework for encoding arbitrary classes in Java. However, not all built-in Java collection classes can be serialized.	If one avoids non-serializable exceptions, such as priority queues, Kryo can be very efficient. Direct support in Scala is also under way.

Certainly, Java has a built-in serialization framework, but as it has to support all Java cases, and therefore is overly general, the Java serialization is far less efficient than any of the preceding methods. I have certainly seen other companies implement their own proprietary serialization earlier, which would beat any of the preceding serialization for the specific cases. Nowadays, it is no longer necessary, as the maintenance costs definitely overshadow the converging inefficiency of the existing frameworks.

Hive and Impala

One of the design considerations for a new framework is always the compatibility with the old frameworks. For better or worse, most data analysts still work with SQL. The roots of the SQL go to an influential relational modeling paper (*Codd, Edgar F* (June 1970). *A Relational Model of Data for Large Shared Data Banks. Communications of the ACM (Association for Computing Machinery)* 13 (6): 377–87). All modern databases implement one or another version of SQL.

While the relational model was influential and important for bringing the database performance, particularly for **Online Transaction Processing (OLTP)** to the competitive levels, the significance of normalization for analytic workloads, where one needs to perform aggregations, and for situations where relations themselves change and are subject to analysis, is less critical. This section will cover the extensions of standard SQL language for analysis engines traditionally used for big data analytics: Hive and Impala. Both of them are currently Apache licensed projects. The following table summarizes the complex types:

Type	Hive support since version	Impala support since version	Comments
ARRAY	This is supported since 0.1.0, but the use of non-constant index expressions is allowed only as of 0.14.	This is supported since 2.3.0 (only for Parquet tables).	This can be an array of any type, including complex. The index is <code>int</code> in Hive (<code>bigint</code> in Impala) and access is via array notation, for example, <code>element [1]</code> only in Hive (<code>array.pos</code> and <code>item pseudocolumns</code> in Impala).
MAP	This is supported since 0.1.0, but the use of non-constant index expressions is allowed only as of 0.14.	This is supported since 2.3.0 (only for Parquet tables).	The key should be of primitive type. Some libraries support keys of the string type only. Fields are accessed using array notation, for example, <code>map ["key"]</code> only in Hive (map key and value pseudocolumns in Impala).
STRUCT	This is supported since 0.5.0.	This is supported since 2.3.0 (only for Parquet tables).	Access is using dot notation, for example, <code>struct . element</code> .

Type	Hive support since version	Impala support since version	Comments
UNIONTYPE	This is supported since 0.7.0.	This is not supported in Impala.	Support is incomplete: queries that reference UNIONTYPE fields in JOIN (HIVE-2508), WHERE, and GROUP BY clauses will fail, and Hive does not define the syntax to extract the tag or value fields of UNIONTYPE. This means that UNIONTYPEs are effectively look-at-only.

While Hive/Impala tables can be created on top of many underlying file formats (Text, Sequence, ORC, Avro, Parquet, and even custom format) and multiple serializations, in most practical instances, Hive is used to read lines of text in ASCII files. The underlying serialization/deserialization format is `LazySimpleSerDe` (**Serialization/Deserialization (SerDe)**). The format defines several levels of separators, as follows:

```
row_format
  : DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]]
    [COLLECTION ITEMS TERMINATED BY char]
    [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]
    [NULL DEFINED AS char]
```

The default for separators are '\001' or ^A, '\002' or ^B, and '\003' or ^C. In other words, it's using the new separator at each level of the hierarchy as opposed to the definition/repetition indicator in the Dremel encoding. For example, to encode the `LabeledPoint` table that we used before, we need to create a file, as follows:

```
$ cat data
0^A1^B1^D1.0$
2^A1^B1^D3.0$
1^A0^B0.0^C2.0^C0.0$
3^A0^B0.0^C4.0^C0.0$
```

Download Hive from <http://archive.cloudera.com/cdh5/cdh/5/hive-1.1.0-cdh5.5.0.tar.gz> and perform the follow:

```
$ tar xf hive-1.1.0-cdh5.5.0.tar.gz
$ cd hive-1.1.0-cdh5.5.0
$ bin/hive
...
hive> CREATE TABLE LABELED_POINT ( LABEL INT, VECTOR
UNIONTYPE<ARRAY<DOUBLE>, MAP<INT,DOUBLE>> ) STORED AS TEXTFILE;
OK
Time taken: 0.453 seconds
hive> LOAD DATA LOCAL INPATH './data' OVERWRITE INTO TABLE LABELED_POINT;
Loading data to table alexdb.labeled_point
Table labeled_point stats: [numFiles=1, numRows=0, totalSize=52,
rawDataSize=0]
OK
Time taken: 0.808 seconds
hive> select * from labeled_point;
OK
0  {1:{1:1.0}}
2  {1:{1:3.0}}
1  {0:[0.0,2.0,0.0]}
3  {0:[0.0,4.0,0.0]}
Time taken: 0.569 seconds, Fetched: 4 row(s)
hive>
```

In Spark, select from a relational table is supported via the `sqlContext.sql` method, but unfortunately the Hive union types are not directly supported as of Spark 1.6.1; it does support maps and arrays though. The supportability of complex objects in other BI and data analysis tools still remains the biggest obstacle to their adoption. Supporting everything as a rich data structure in Scala is one of the options to converge on nested data representation.

Sessionization

I will demonstrate the use of the complex or nested structures in the example of sessionization. In sessionization, we want to find the behavior of an entity, identified by some ID over a period of time. While the original records may come in any order, we want to summarize the behavior over time to derive trends.

We already analyzed web server logs in *Chapter 1, Exploratory Data Analysis*. We found out how often different web pages are accessed over a period of time. We could dice and slice this information, but without analyzing the sequence of pages visited, it would be hard to understand each individual user interaction with the website. In this chapter, I would like to give this analysis more individual flavor by tracking the user navigation throughout the website. Sessionization is a common tool for website personalization and advertising, IoT tracking, telemetry, and enterprise security, in fact anything to do with entity behavior.

Let's assume the data comes as tuples of three elements (fields 1, 5, 11 in the original dataset in *Chapter 1, Exploratory Data Analysis*):

```
(id, timestamp, path)
```

Here, `id` is a unique entity ID, `timestamp` is an event `timestamp` (in any sortable format: Unix timestamp or an ISO8601 date format), and `path` is some indication of the location on the web server page hierarchy.

For people familiar with SQL, sessionization, or at least a subset of it, is better known as a windowing analytics function:

```
SELECT id, timestamp, path
    ANALYTIC_FUNCTION(path) OVER (PARTITION BY id ORDER BY
        timestamp) AS agg
    FROM log_table;
```

Here `ANALYTIC_FUNCTION` is some transformation on the sequence of paths for a given `id`. While this approach works for a relatively simple function, such as first, last, lag, average, expressing a complex function over a sequence of paths is usually very convoluted (for example, `nPath` from Aster Data (<https://www.nersc.gov/assets/Uploads/AnalyticsFoundation5.0previewfor4.6.x-Guide.pdf>)). Besides, without additional preprocessing and partitioning, these approaches usually result in big data transfers across multiple nodes in a distributed setting.

While in a pure functional approach, one would just have to design a function—or a sequence of function applications—to produce the desired answers from the original set of tuples, I will create two helper objects that will help us to simplify working with the concept of a user session. As an additional benefit, the new nested structures can be persisted on a disk to speed up getting answers on additional questions.

Let's see how it's done in Spark/Scala using case classes:

```
akozlov@Alexanders-MacBook-Pro$ bin/spark-shell
```

Welcome to

/ _/ _ _ _ _ / /
\ \/ \/_ ^/_ / '_/_
/___/ .__/_,/_/_ /_/_\ \ version 1.6.1-SNAPSHOT
/_/_

Using Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_40)

Type in expressions to have them evaluated.

Type :help for more information.

Spark context available as `sc`.

SOL context available as `sqlContext`.

```
scala> :paste
```

```
// Entering paste mode (ctrl-D to finish)
```

```
import java.io.
```

```
// a basic page view structure
```

@SerialVersionUID(123L)

```
case class PageView(ts: String, path: String) extends Serializable with Ordered[PageView] {
```

```
override def toString: String = {
```

```
s"($t:s :$path) "
```

1

```
def compare(other: PageView) = ts.compare(other.ts)
```

3

```
// represent a session
```

```
@SerialVersionUID(456L)
case class Session[A <: PageView](id: String, visits: Seq[A]) extends
Serializable {
    override def toString: String = {
        val vsts = visits.mkString("[", ", ", ", ", "]")
        s"($id -> $vsts)"
    }
}^D
// Exiting paste mode, now interpreting.
```

```
import java.io._
defined class PageView
defined class Session
```

The first class will represent a single page view with a timestamp, which, in this case, is an ISO8601 String, while the second a sequence of page views. Could we do it by encoding both members as a String with a object separator? Absolutely, but representing the fields as members of a class gives us nice access semantics, together with offloading some of the work that we need to perform on the compiler, which is always nice.

Let's read the previously described log files and construct the objects:

```
scala> val rdd = sc.textFile("log.csv").map(x => { val z =
x.split(", ", 3); (z(1), new PageView(z(0), z(2))) } ).groupByKey.map( x =>
{ new Session(x._1, x._2.toSeq.sorted) } ).persist
rdd: org.apache.spark.rdd.RDD[Session] = MapPartitionsRDD[14] at map at
<console>:31
```

```
scala> rdd.take(3).foreach(println)
(189.248.74.238 -> [(2015-08-23 23:09:16 :mycompanycom>homepa
ge), (2015-08-23 23:11:00 :mycompanycom>homepage), (2015-08-23 23:11:02
:mycompanycom>running:slp), (2015-08-23 23:12:01 :mycompanycom>running
:slp), (2015-08-23 23:12:03 :mycompanycom>running>stories>2013>04>them
ycompanyfreestore:cdp), (2015-08-23 23:12:08 :mycompanycom>running>sto
ries>2013>04>themycompanyfreestore:cdp), (2015-08-23 23:12:08 :mycomp
anycom>running>stories>2013>04>themycompanyfreestore:cdp), (2015-08-23
23:12:42 :mycompanycom>running:slp), (2015-08-23 23:13:25 :mycompanyc
om>homepage), (2015-08-23 23:14:00 :mycompanycom>homepage), (2015-08-23
23:14:06 :mycompanycom:mobile>mycompany photoid>landing), (2015-08-23
23:14:56 :mycompanycom:men>shoes:segmentedgrid), (2015-08-23 23:15:10
:mycompanycom>homepage)])
(82.166.130.148 -> [(2015-08-23 23:14:27 :mycompanycom>homepage)])
```

```
(88.234.248.111 -> [(2015-08-23 22:36:10 :mycompanycom>plus>home) , (2015-08-23 22:36:20 :mycompanycom>plus>home) , (2015-08-23 22:36:28 :mycompanycom>plus>home) , (2015-08-23 22:36:30 :mycompanycom>plus>onepluspdp>sport band) , (2015-08-23 22:36:52 :mycompanycom>onsite search>results found) , (2015-08-23 22:37:19 :mycompanycom>plus>onepluspdp>sport band) , (2015-08-23 22:37:21 :mycompanycom>plus>home) , (2015-08-23 22:37:39 :mycompanycom>plus>home) , (2015-08-23 22:37:43 :mycompanycom>plus>home) , (2015-08-23 22:37:46 :mycompanycom>plus>onepluspdp>sport watch) , (2015-08-23 22:37:50 :mycompanycom>gear>mycompany+ sportwatch:standardgrid) , (2015-08-23 22:38:14 :mycompanycom>homepage) , (2015-08-23 22:38:35 :mycompanycom>homepage) , (2015-08-23 22:38:37 :mycompanycom>plus>products landing) , (2015-08-23 22:39:01 :mycompanycom>homepage) , (2015-08-23 22:39:24 :mycompanycom>homepage) , (2015-08-23 22:39:26 :mycompanycom>plus>whatismycompanyfuel)])
```

Bingo! We have an RDD of Sessions, one per each unique IP address. The IP 189.248.74.238 has a session that lasted from 23:09:16 to 23:15:10, and seemingly ended after browsing for men's shoes. The session for IP 82.166.130.148 contains only one hit. The last session concentrated on sports watch and lasted for over three minutes from 2015-08-23 22:36:10 to 2015-08-23 22:39:26. Now, we can easily ask questions involving specific navigation path patterns. For example, we want analyze all the sessions that resulted in checkout (the path contains checkout) and see the number of hits and the distribution of times after the last hit on homepage:

```
scala> import java.time.ZonedDateTime
import java.time.ZonedDateTime

scala> import java.time.LocalDateTime
import java.time.LocalDateTime

scala> import java.time.format.DateTimeFormatter
import java.time.format.DateTimeFormatter

scala>
scala> def toEpochSeconds(str: String) : Long = { LocalDateTime.parse(str, DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")).toEpochSecond(ZoneOffset.UTC) }
toEpochSeconds: (str: String)Long

scala> val checkoutPattern = ".*>checkout.*".r.pattern
```

```
checkoutPattern: java.util.regex.Pattern = .*>checkout.*

scala> val lengths = rdd.map(x => { val pths = x.visits.map(y => y.path);
val pchs = pths.indexWhere(checkoutPattern.matcher(_).matches); (x.id,
x.visits.map(y => y.ts).min, x.visits.map(y => y.ts).max, x.visits.
lastIndexWhere(_ match { case PageView(ts, "mycompanycom>homepage")
=> true; case _ => false }, pchs), pchs, x.visits) } ).filter(_.4>0).
filter(t => t._5>t._4).map(t => (t._5 - t._4, toEpochSeconds(t._6(t._5).
ts) - toEpochSeconds(t._6(t._4).ts)))

scala> lengths.toDF("cnt", "sec").agg(avg($"cnt"),min($"cnt"),max($"cnt"),
avg($"sec"),min($"sec"),max($"sec")).show
+-----+-----+-----+-----+-----+
-+
|      avg(cnt) |min(cnt) |max(cnt) |
avg(sec) |min(sec) |max(sec) |
+-----+-----+-----+-----+-----+
-+
| 19.77570093457944 |       1 |     121 | 366.06542056074767 |       15 |
2635 |
+-----+-----+-----+-----+-----+
-+
```



```
scala> lengths.map(x => (x._1,1)).reduceByKey(_+_).sortByKey().collect
res18: Array[(Int, Int)] = Array((1,1), (2,8), (3,2), (5,6), (6,7),
(7,9), (8,10), (9,4), (10,6), (11,4), (12,4), (13,2), (14,3), (15,2),
(17,4), (18,6), (19,1), (20,1), (21,1), (22,2), (26,1), (27,1), (30,2),
(31,2), (35,1), (38,1), (39,2), (41,1), (43,2), (47,1), (48,1), (49,1),
(65,1), (66,1), (73,1), (87,1), (91,1), (103,1), (109,1), (121,1))
```

The sessions last from 1 to 121 hits with a mode at 8 hits and from 15 to 2653 seconds (or about 45 minutes). Why would you be interested in this information? Long sessions might indicate that there was a problem somewhere in the middle of the session: a long delay or non-responsive call. It does not have to be: the person might just have taken a long lunch break or a call to discuss his potential purchase, but there might be something of interest here. At least one should agree that this is an outlier and needs to be carefully analyzed.

Let's talk about persisting this data to the disk. As you've seen, our transformation is written as a long pipeline, so there is nothing in the result that one could not compute from the raw data. This is a functional approach, the data is immutable. Moreover, if there is an error in our processing, let's say I want to change the homepage to some other anchor page, I can always modify the function as opposed to data. You may be content or not with this fact, but there is absolutely no additional piece of information in the result — transformations only increase the disorder and entropy. They might make it more palatable for humans, but this is only because humans are a very inefficient data-processing apparatus.

Why rearranging the data makes the analysis faster?



Sessionization seems just a simple rearranging of data — we just put the pages that were accessed in sequence together. Yet, in many cases, it makes practical data analysis run 10 to 100 times faster. The reason is data locality. The analysis, like filtering or path matching, most often tends to happen on the pages in one session at a time. Deriving user features requires all page views or interactions of the user to be in one place on disk and memory. This often beats other inefficiencies such as the overhead of encoding/decoding the nested structures as this can happen in local L1/L2 cache as opposed to data transfers from RAM or disk, which are much more expensive in modern multithreaded CPUs. This very much depends on the complexity of the analysis, of course.

There is a reason to persist the new data to the disk, and we can do it with either CSV, Avro, or Parquet format. The reason is that we do not want to reprocess the data if we want to look at them again. The new representation might be more compact and more efficient to retrieve and show to my manager. Really, humans like side effects and, fortunately, Scala/Spark allows you to do this as was described in the previous section.

Well, well, well...will say the people familiar with sessionization. This is only a part of the story. We want to split the path sequence into multiple sessions, run path analysis, compute conditional probabilities for page transitions, and so on. This is exactly where the functional paradigm shines. Write the following function:

```
def splitSession(session: Session[PageView]) :  
  Seq[Session[PageView]] = { ... }
```

Then run the following code:

```
val newRdd = rdd.flatMap(splitSession)
```

Bingo! The result is the session's split. I intentionally left the implementation out; it's the implementation that is user-dependent, not the data, and every analyst might have its own way to split the sequence of page visits into sessions.

Another use case to apply the function is feature generation for applying machine learning...well, this is already hinting at the side effect: we want to modify the state of the world to make it more personalized and user-friendly. I guess one cannot avoid it after all.

Working with traits

As we saw, case classes significantly simplify handling of new nested data structures that we want to construct. The case class definition is probably the most convincing reason to move from Java (and SQL) to Scala. Now, what about the methods? How do we quickly add methods to a class without expensive recompilation? Scala allows you to do this transparently with traits!

A fundamental feature of functional programming is that functions are a first class citizen on par with objects. In the previous section, we defined the two EpochSeconds functions that transform the ISO8601 format to epoch time in seconds. We also suggested the splitSession function that provides a multi-session view for a given IP. How do we associate this or other behavior with a given class?

First, let's define a desired behavior:

```
scala> trait Epoch {  
|   this: PageView =>  
|   def epoch(): Long = { LocalDateTime.parse(ts,  
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")).  
toEpochSecond(ZoneOffset.UTC) }  
| }  
defined trait Epoch
```

This basically creates a PageView-specific function that converts a string representation for datetime to epoch time in seconds. Now, if we just make the following transformation:

```
scala> val rddEpoch = rdd.map(x => new Session(x.id, x.visits.map(x =>  
new PageView(x.ts, x.path) with Epoch)))  
rddEpoch: org.apache.spark.rdd.RDD[Session[PageView with Epoch]] =  
MapPartitionsRDD[20] at map at <console>:31
```

We now have a new RDD of page views with additional behavior. For example, if we want to find out what is the time spent on each individual page in a session is, we will run a pipeline, as follows:

```
scala> rddEpoch.map(x => (x.id, x.visits.zip(x.visits.tail).map(x =>  
(x._2.path, x._2.epoch - x._1.epoch)).mkString("[", ", ", ", ", "]"))).take(3).  
foreach(println)
```

```
(189.248.74.238, [(mycompanycom>homepage,104), (mycompanycom>running:slp,2)
, (mycompanycom>running:slp,59), (mycompanycom>running>stories>2013>04>them
ycompanyfreestore:cdp,2), (mycompanycom>running>stories>2013>04>themyc
ompanyfreestore:cdp,5), (mycompanycom>running>stories>2013>04>themyc
ompanyfree
store:cdp,0), (mycompanycom>running:slp,34), (mycompanycom>homepage,43), (my
companycom>homepage,35), (mycompanycom:mobile>mycompany photoid>landing,6)
, (mycompanycom>men>shoes:segmentedgrid,50), (mycompanycom>homepage,14)])
(82.166.130.148, [])
(88.234.248.111, [(mycompanycom>plus>home,10), (mycompanycom>plus>home
,8), (mycompanycom>plus>onepluspdp>sport band,2), (mycompanycom>onsite
search>results found,22), (mycompanycom>plus>onepluspdp>sport band,27), (my
companycom>plus>home,2), (mycompanycom>plus>home,18), (mycompanycom>plus>h
ome,4), (mycompanycom>plus>onepluspdp>sport watch,3), (mycompanycom>gear>my
company+ sportwatch:standardgrid,4), (mycompanycom>homepage,24), (mycompany
com>homepage,21), (mycompanycom>plus>products landing,2), (mycompanycom>hom
epage,24), (mycompanycom>homepage,23), (mycompanycom>plus>whatismycompanyfu
el,2)])
```

Multiple traits can be added at the same time without affecting either the original class definitions or original data. No recompilation is required.

Working with pattern matching

No Scala book would be complete without mentioning the `match/case` statements. Scala has a very rich pattern-matching mechanism. For instance, let's say we want to find all instances of a sequence of page views that start with a homepage followed by a products page—we really want to filter out the determined buyers. This may be accomplished with a new function, as follows:

```
scala> def findAllMatchedSessions(h: Seq[Session[PageView]], s:
Session[PageView]) : Seq[Session[PageView]] = {
|     def matchSessions(h: Seq[Session[PageView]], id: String, p:
Seq[PageView]) : Seq[Session[PageView]] = {
|         p match {
|             case Nil => Nil
|             case PageView(ts1, "mycompanycom>homepage") :: PageView(ts2, "mycompanycom>plus>products landing") :: tail =>
|                 matchSessions(h, id, tail).:+:(new Session(id, p))
|             case _ => matchSessions(h, id, p.tail)
|         }
|     }
|     matchSessions(h, s.id, s.visits)
| }
```

```
findAllSessions: (h: Seq[Session[PageView]], s: Session[PageView])  
Seq[Session[PageView]]
```

Note that we explicitly put `PageView` constructors in the case statement! Scala will traverse the `visits` sequence and generate new sessions that match the specified two `PageViews`, as follows:

```
scala> rdd.flatMap(x => findAllMatchedSessions(Nil, x)).take(10).  
foreach(println)  
  
(88.234.248.111 -> [(2015-08-23 22:38:35 :mycompanycom>homepa  
ge),(2015-08-23 22:38:37 :mycompanycom>plus>products landing),(2015-08-23  
22:39:01 :mycompanycom>homepage),(2015-08-23 22:39:24 :mycompanycom>homepa  
ge),(2015-08-23 22:39:26 :mycompanycom>plus>whatismycompanyfuel)])  
  
(148.246.218.251 -> [(2015-08-23 22:52:09 :mycompanycom>homepa  
ge),(2015-08-23 22:52:16 :mycompanycom>plus>products landing),(2015-08-23  
22:52:23 :mycompanycom>homepage),(2015-08-23 22:52:32 :mycompanycom>homepa  
ge),(2015-08-23 22:52:39 :mycompanycom>running:slp)])  
  
(86.30.116.229 -> [(2015-08-23 23:15:00 :mycompanycom>homepa  
ge),(2015-08-23 23:15:02 :mycompanycom>plus>products landing),(2015-08-23  
23:15:12 :mycompanycom>plus>products landing),(2015-08-23  
23:15:18 :mycompanycom>language tunnel>load),(2015-08-23 23:15:23  
:mycompanycom>language tunnel>geo selected),(2015-08-23 23:15:24  
:mycompanycom>homepage),(2015-08-23 23:15:27 :mycompanycom>homepa  
ge),(2015-08-23 23:15:30 :mycompanycom>basketball:slp),(2015-08-23  
23:15:38 :mycompanycom>basketball>lebron-10:cdp),(2015-08-23 23:15:50  
:mycompanycom>basketball>lebron-10:cdp),(2015-08-23 23:16:05 :my  
companycom>homepage),(2015-08-23 23:16:09 :mycompanycom>homepa  
ge),(2015-08-23 23:16:11 :mycompanycom>basketball:slp),(2015-08-23  
23:16:29 :mycompanycom>onsite search>results found),(2015-08-23 23:16:39  
:mycompanycom>onsite search>no results)])  
  
(204.237.0.130 -> [(2015-08-23 23:26:23 :mycompanycom>homepa  
ge),(2015-08-23 23:26:27 :mycompanycom>plus>products landing),(2015-08-23  
23:26:35 :mycompanycom>plus>fuelband activity>summary>wk)])  
  
(97.82.221.34 -> [(2015-08-23 22:36:24 :mycompanycom>homepa  
ge),(2015-08-23 22:36:32 :mycompanycom>plus>products landing),(2015-08-23  
22:37:09 :mycompanycom>plus>plus activity>summary>wk),(2015-08-23  
22:37:39 :mycompanycom>plus>products landing),(2015-08-23 22:44:17  
:mycompanycom>plus>home),(2015-08-23 22:44:33 :mycompanycom>plus>ho  
me),(2015-08-23 22:44:34 :mycompanycom>plus>home),(2015-08-23 22:44:36  
:mycompanycom>plus>home),(2015-08-23 22:44:43 :mycompanycom>plus>home)])  
  
(24.230.204.72 -> [(2015-08-23 22:49:58 :mycompanycom>homepa  
ge),(2015-08-23 22:50:00 :mycompanycom>plus>products landing),(2015-08-23  
22:50:30 :mycompanycom>homepage),(2015-08-23 22:50:38 :mycompa  
nycom>homepage),(2015-08-23 22:50:41 :mycompanycom>training:c  
dp),(2015-08-23 22:51:56 :mycompanycom>training:cdp),(2015-08-23  
22:51:59 :mycompanycom>store locator>start),(2015-08-23 22:52:28  
:mycompanycom>store locator>landing)])
```

```
(62.248.72.18 -> [(2015-08-23 23:14:27 :mycompanycom>homepa
ge), (2015-08-23 23:14:30 :mycompanycom>plus>products landing), (2015-08-23
23:14:33 :mycompanycom>plus>products landing), (2015-08-23
23:14:40 :mycompanycom>plus>products landing), (2015-08-23 23:14:47
:mycompanycom>store homepage), (2015-08-23 23:14:50 :mycompanycom>store
homepage), (2015-08-23 23:14:55 :mycompanycom>men:clp), (2015-08-23
23:15:08 :mycompanycom>men:clp), (2015-08-23 23:15:15 :mycompanyco
m>men:clp), (2015-08-23 23:15:16 :mycompanycom>men:clp), (2015-08-23
23:15:24 :mycompanycom>men>sportswear:standardgrid), (2015-08-23
23:15:41 :mycompanycom>pdp>mycompany blazer low premium vintage
suede men's shoe), (2015-08-23 23:15:45 :mycompanycom>pdp>mycompany
blazer low premium vintage suede men's shoe), (2015-08-23 23:15:45
:mycompanycom>pdp>mycompany blazer low premium vintage suede
men's shoe), (2015-08-23 23:15:49 :mycompanycom>pdp>mycompany
blazer low premium vintage suede men's shoe), (2015-08-23 23:15:50
:mycompanycom>pdp>mycompany blazer low premium vintage suede men's
shoe), (2015-08-23 23:15:56 :mycompanycom>men>sportswear:standardgr
id), (2015-08-23 23:18:41 :mycompanycom>pdp>mycompany bruin low men's
shoe), (2015-08-23 23:18:42 :mycompanycom>pdp>mycompany bruin low
men's shoe), (2015-08-23 23:18:53 :mycompanycom>pdp>mycompany bruin low
men's shoe), (2015-08-23 23:18:55 :mycompanycom>pdp>mycompany bruin
low men's shoe), (2015-08-23 23:18:57 :mycompanycom>pdp>mycompany
bruin low men's shoe), (2015-08-23 23:19:04 :mycompanycom>men>sport
swear:standardgrid), (2015-08-23 23:20:12 :mycompanycom>men>sportsw
ear>silver:standardgrid), (2015-08-23 23:28:20 :mycompanycom>onsite
search>no results), (2015-08-23 23:28:33 :mycompanycom>onsite
search>no results), (2015-08-23 23:28:36 :mycompanycom>pdp>mycompany
blazer low premium vintage suede men's shoe), (2015-08-23 23:28:40
:mycompanycom>pdp>mycompany blazer low premium vintage suede
men's shoe), (2015-08-23 23:28:41 :mycompanycom>pdp>mycompany
blazer low premium vintage suede men's shoe), (2015-08-23 23:28:43
:mycompanycom>pdp>mycompany blazer low premium vintage suede men's
shoe), (2015-08-23 23:28:43 :mycompanycom>pdp>mycompany blazer low premium
vintage suede men's shoe), (2015-08-23 23:29:00 :mycompanycom>pdp>mycompan
yid>mycompany blazer low id shoe))

(46.5.127.21 -> [(2015-08-23 22:58:00 :mycompanycom>homepage), (2015-08-23
22:58:01 :mycompanycom>plus>products landing)])

(200.45.228.1 -> [(2015-08-23 23:07:33 :mycompanycom>homepa
ge), (2015-08-23 23:07:39 :mycompanycom>plus>products landing), (2015-08-23
23:07:42 :mycompanycom>plus>products landing), (2015-08-23 23:07:45
:mycompanycom>language tunnel>load), (2015-08-23 23:07:59 :mycompanyco
m>homepage), (2015-08-23 23:08:15 :mycompanycom>homepage), (2015-08-23
23:08:26 :mycompanycom>onsite search>results found), (2015-08-23
23:08:43 :mycompanycom>onsite search>no results), (2015-08-23
23:08:49 :mycompanycom>onsite search>results found), (2015-08-23
23:08:53 :mycompanycom>language tunnel>load), (2015-08-23 23:08:55
:mycompanycom>plus>products landing), (2015-08-23 23:09:04 :mycompanycom>h
omepage), (2015-08-23 23:11:34 :mycompanycom>running:slp)])
```

```
(37.78.203.213 -> [(2015-08-23 23:18:10 :mycompanycom>homepa  
ge), (2015-08-23 23:18:12 :mycompanycom>plus>products landing), (2015-08-23  
23:18:14 :mycompanycom>plus>products landing), (2015-08-23 23:18:22  
:mycompanycom>plus>products landing), (2015-08-23 23:18:25  
:mycompanycom>store homepage), (2015-08-23 23:18:31 :mycompanycom>store  
homepage), (2015-08-23 23:18:34 :mycompanycom>men:clp), (2015-08-23  
23:18:50 :mycompanycom>store homepage), (2015-08-23 23:18:51 :mycompanyc  
om>footwear:segmentedgrid), (2015-08-23 23:19:12 :mycompanycom>men>footwe  
ar:segmentedgrid), (2015-08-23 23:19:12 :mycompanycom>men>footwear:segmen  
tedgrid), (2015-08-23 23:19:26 :mycompanycom>men>footwear>new releases:st  
andardgrid), (2015-08-23 23:19:26 :mycompanycom>men>footwear>new releases  
:standardgrid), (2015-08-23 23:19:35 :mycompanycom>pdp>mycompany cheyenne  
2015 men's shoe), (2015-08-23 23:19:40 :mycompanycom>men>footwear>new  
releases:standardgrid)])
```

I leave it to the reader to write a function that also filters only those sessions where the user spent less than 10 seconds before going to the products page. The epoch trait or the previously defined to the EpochSeconds function may be useful.

The match/case function can be also used for feature generation and return a vector of features over a session.

Other uses of unstructured data

The personalization and device diagnostic obviously are not the only uses of unstructured data. The preceding case is a good example as we started from structured record and quickly converged on the need to construct an unstructured data structure to simplify the analysis.

In fact, there are many more unstructured data than there are structured; it is just the convenience of having the flat structure for the traditional statistical analysis that makes us to present the data as a set of records. Text, images, and music are the examples of semi-structured data.

One example of non-structured data is denormalized data. Traditionally the record data are normalized mostly for performance reasons as the RDBMSs have been optimized to work with structured data. This leads to foreign key and lookup tables, but these are very hard to maintain if the dimensions change. Denormalized data does not have this problem as the lookup table can be stored with each record—it is just an additional table object associated with a row, but may be less storage-efficient.

Probabilistic structures

Another use case is the probabilistic structures. Usually people assume that answering a question is deterministic. As I showed in *Chapter 2, Data Pipelines and Modeling*, in many cases, the true answer has some uncertainty associated with it. One of the most popular ways to encode uncertainty is probability, which is a frequentist approach, meaning that the simple count of when the answer does happen to be the true answer, divided by the total number of attempts – the probability also can encode our beliefs. I will touch on probabilistic analysis and models in the following chapters, but probabilistic analysis requires storing each possible outcome with some measure of probability, which happens to be a nested structure.

Projections

One way to deal with high dimensionality is projections on a lower dimensional space. The fundamental basis for why projections might work is Johnson-Lindenstrauss lemma. The lemma states that a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved. We will touch on random and other projections when we talk about NLP in *Chapter 9, NLP in Scala*, but the random projections work well for nested structures and functional programming language, as in many cases, generating a random projection is the question of applying a function to a compactly encoded data rather than flattening the data explicitly. In other words, the Scala definition for a random projection may look like functional paradigm shines. Write the following function:

```
def randomProjection(data: NestedStructure) : Vector = { ... }
```

Here, `Vector` is in low dimensional space.

The map used for embedding is at least Lipschitz, and can even be taken to be an orthogonal projection.

Summary

In this chapter, we saw examples of how to represent and work with complex and nested data in Scala. Obviously, it would be hard to cover all the cases as the world of unstructured data is much larger than the nice niche of structured row-by-row simplification of the real world and is still under construction. Pictures, music, and spoken and written language have a lot of nuances that are hard to capture in a flat representation.

While for ultimate data analysis, we eventually convert the datasets to the record-oriented flat representation, at least at the time of collection, one needs to be careful to store that data as it is and not throw away useful information that might be contained in data or metadata. Extending the databases and storage with a way to record this useful information is the first step. The next one is to use languages that can effectively analyze this information; which is definitely Scala.

In the next chapter we'll look at somewhat related topic of working with graphs, a specific example of non-structured data.

7

Working with Graph Algorithms

In this chapter, I'll delve into graph libraries and algorithm implementations in Scala. In particular, I will introduce Graph for Scala (<http://www.scala-graph.org>), an open source project that was started in 2011 in the EPFL Scala incubator. Graph for Scala does not support distributed computing yet—the distributed computing aspects of popular graph algorithms is available in GraphX, which is a part of MLlib library that is part of Spark project (<http://spark.apache.org/docs/latest/mllib-guide.html>). Both, Spark and MLlib were started as class projects at UC Berkeley around or after 2009. I considered Spark in *Chapter 3, Working with Spark and MLlib* and introduced an RDD. In GraphX, a graph is a pair of RDDs, each of which is partitioned among executors and tasks, represents vertices and edges in a graph.

In this chapter, we will cover the following topics:

- Configuring **Simple Build Tool (SBT)** to use the material in this chapter interactively
- Learning basic operations on graphs supported by Graph for Scala
- Learning how to enforce graph constraints
- Learning how to import/export graphs in JSON
- Performing connected components, triangle count, and strongly connected components running on Enron e-mail data
- Performing PageRank computations on Enron e-mail data
- Learning how to use SVD++

A quick introduction to graphs

What is a graph? A graph is a set of **vertices** where some pairs of these vertices are linked with **edges**. If every vertex is linked with every other vertex, we say the graph is a complete graph. On the contrary, if it has no edges, the graph is said to be empty. These are, of course, extremes that are rarely encountered in practice, as graphs have varying degrees of density; the more edges it has proportional to the number of vertices, the more dense we say it is.

Depending on what algorithms we intend to run on a graph and how dense is it expected to be, we can choose how to appropriately represent the graph in memory. If the graph is really dense, it pays off to store it as a square $N \times N$ matrix, where 0 in the n th row and m th column means that the n vertex is not connected to the m vertex. A diagonal entry expresses a node connection to itself. This representation is called the adjacency matrix.

If there are not many edges and we need to traverse the whole edge set without distinction, often it pays off to store it as a simple container of pairs. This structure is called an **edge list**.

In practice, we can model many real-life situations and events as graphs. We could imagine cities as vertices and plane routes as edges. If there is no flight between two cities, there is no edge between them. Moreover, if we add the numerical costs of plane tickets to the edges, we say that the graph is **weighted**. If there are some edges where only travels in one direction exist, we can represent that by making a graph directed as opposed to an undirected graph. So, for an undirected graph, it is true that the graph is symmetric, that is, if A is connected to B , then B is also connected to A – that is not necessarily true for a directed graph.

Graphs without cycles are called acyclic. Multigraph can contain multiple edges, potentially of different type, between the nodes. Hyperedges can connect arbitrary number of nodes.

The most popular algorithm on the undirected graphs is probably **connected components**, or partitioning of a graph into subgraph, in which any two vertices are connected to each other by paths. Partitioning is important to parallelize the operations on the graphs.

Google and other search engines made PageRank popular. According to Google, PageRank estimates of how important the website is by counting the number and quality of links to a page. The underlying assumption is that more important websites are likely to receive more links from other websites, especially more highly ranked ones. PageRank can be applied to many problems outside of websites ranking and is equivalent to finding eigenvectors and the most significant eigenvalue of the connectivity matrix.

The most basic, nontrivial subgraph, consists of three nodes. Triangle counting finds all the possible fully connected (or complete) triples of nodes and is another well-known algorithm used in community detection and CAD.

A **clique** is a fully connected subgraph. A strongly connected component is an analogous notion for a directed graph: every vertex in a subgraph is reachable from every other vertex. GraphX provides an implementation for both.

Finally, a recommender graph is a graph connecting two types of nodes: users and items. The edges can additionally contain the strength of a recommendation or a measure of satisfaction. The goal of a recommender is to predict the satisfaction for potentially missing edges. Multiple algorithms have been developed for a recommendation engine, such as SVD and SVD++, which are considered at the end of this chapter.

SBT

Everyone likes Scala REPL. REPL is the command line for Scala. It allows you to type Scala expressions that are evaluated immediately and try and explore things. As you saw in the previous chapters, one can simply type `scala` at the command prompt and start developing complex data pipelines. What is even more convenient is that one can press `tab` to have auto-completion, a required feature of any fully developed modern IDE (such as Eclipse or IntelliJ, `Ctrl +.` or `Ctrl + Space`) by keeping track of the namespace and using reflection mechanisms. Why would we need one extra tool or framework for builds, particularly that other builds management frameworks such as Ant, Maven, and Gradle exist in addition to IDEs? As the SBT authors argue, even though one might compile Scala using the preceding tools, all of them have inefficiencies, as it comes to interactivity and reproducibility of Scala builds (*SBT in Action* by Joshua Suereth and Matthew Farwell, Nov 2015).

One of the main SBT features for me is interactivity and the ability to seamlessly work with multiple versions of Scala and dependent libraries. In the end, what is critical for software development is the speed with which one can prototype and test new ideas. I used to work on mainframes using punch cards, where the programmers were waiting to execute their programs and ideas, sometimes for hours and days. The efficiency of the computers mattered more, as this was the bottleneck. These days are gone, and a personal laptop is probably having more computing power than rooms full of servers a few decades back. To take advantage of this efficiency, we need to utilize human time more efficiently by speeding up the program development cycle, which also means interactivity and more versions in the repositories.

Apart from the ability to handle multiple versions and REPL, SBT's main features are as follows:

- Native support for compiling Scala code and integrating with many test frameworks, including JUnit, ScalaTest, and Selenium
- Build descriptions written in Scala using a DSL
- Dependency management using Ivy (which also supports Maven-format repositories)
- Continuous execution, compilation, testing, and deployment
- Integration with the Scala interpreter for rapid iteration and debugging
- Support for mixed Java/Scala projects
- Support for testing and deployment frameworks
- Ability to complement the tool with custom plugins
- Parallel execution of tasks

SBT is written in Scala and uses SBT to build itself (bootstrapping or dogfooding). SBT became the de facto build tool for the Scala community, and is used by the **Lift** and **Play** frameworks.

While you can download SBT directly from <http://www.scala-sbt.org/download>, the easiest way to install SBT on Mac is to run MacPorts:

```
$ port install sbt
```

You can also run Homebrew:

```
$ brew install sbt
```

While other tools exist to create SBT projects, the most straightforward way is to run the `bin/create_project.sh` script in the GitHub book project repository provided for each chapter:

```
$ bin/create_project.sh
```

This will create main and test source subdirectories (but not the code). The project directory contains project-wide settings (refer to `project/build.properties`). The target will contain compiled classes and build packages (the directory will contain different subdirectories for different versions of Scala, for example, 2.10 and 2.11). Finally, any jars or libraries put into the `lib` directory will be available across the project (I personally recommend using the `libraryDependencies` mechanism in the `build.sbt` file, but not all libraries are available via centralized repositories). This is the minimal setup, and the directory structure may potentially contain multiple subprojects. The Scalastyle plugin will even check the syntax for you (<http://www.scalastyle.org/sbt.html>). Just add `project/plugin.sbt`:

```
$ cat >> project.plugin.sbt << EOF
addSbtPlugin("org.scalastyle" %% "scalastyle-sbt-plugin" % "0.8.0")
EOF
```

Finally, the SBT creates Scaladoc documentation with the `sbt doc` command.

Blank lines and other settings in `build.sbt`

Probably most of the `build.sbt` files out there are double spaced: this is a remnant of old versions. You no longer need them. As of version 0.13.7, the definitions do not require extra lines.

There are many other settings that you can use on `build.sbt` or `build.properties`, the up-to-date documentation is available at <http://www.scala-sbt.org/documentation.html>.

When run from the command line, the tool will automatically download and use the dependencies, in this case, `graph-{core,constrained,json}` and `lift-json`. In order to run the project, simply type `sbt run`.

In continuous mode, SBT will automatically detect changes to the source file and rerun the command(s). In order to continuously compile and run the code, type `~~ run` after starting REPL with `sbt`.

To get help on the commands, run the following command:

```
$ sbt
[info] Loading global plugins from /Users/akozlov/.sbt/0.13/plugins
[info] Set current project to My Graph Project (in build file:/Users/
akozlov/Scala/graph/)
> help
help                                         Displays this help message or
prints detailed help on requested commands (run 'help <command>').
```

```
For example, `sbt package` will build a Java jar, as follows:  
$ sbt package  
[info] Loading global plugins from /Users/akozlov/.sbt/0.13/plugins  
[info] Loading project definition from /Users/akozlov/Scala/graph/project  
[info] Set current project to My Graph Project (in build file:/Users/  
akozlov/Scala/graph/)  
[info] Updating {file:/Users/akozlov/Scala/graph/}graph...  
[info] Resolving jline#jline;2.12.1 ...  
[info] Done updating.  
$ ls -1 target/scala-2.11/  
classes  
my-graph-project_2.11-1.0.jar
```

While SBT will be sufficient for our use even with a simple editor such as **vi** or **Emacs**, the `sbteclipse` project at <https://github.com/typesafehub/sbteclipse> will create the necessary project files to work with your Eclipse IDE.

Graph for Scala

For this project, I will create a `src/main/scala/InfluenceDiagram.scala` file. For demo purpose, I will just recreate the graph from *Chapter 2, Data Pipelines and Modeling*:

```
import scalax.collection.Graph  
import scalax.collection.edge._  
import scalax.collection.GraphPredef._  
import scalax.collection.GraphEdge._  
  
import scalax.collection.edge.Implicits._  
  
object InfluenceDiagram extends App {  
    var g = Graph[String, LDiEdge](("Weather" ~> "Weather Forecast")  
    ("Forecast"), ("Weather Forecast" ~> "Vacation Activity")  
    ("Decision"), ("Vacation Activity" ~> "Satisfaction")  
    ("Deterministic"), ("Weather" ~> "Satisfaction") ("Deterministic"))  
    println(g.mkString(";"))  
    println(g.isDirected)  
    println(g.isAcyclic)  
}
```

The `~+>` operator is used to create a directed labeled edge between two nodes defined in `scalax/collection/edge/Implicits.scala`, which, in our case, are of the `String` type. The list of other edge types and operators is provided in the following table:

The following table shows graph edges from `scalax.collection.edge.Implicits` (from <http://www.scala-graph.org/guides/core-initializing.html>)

Edge Class	Shortcut/Operator	Description
Hyperedges		
HyperEdge	<code>~</code>	hyperedge
WHyperEdge	<code>~%</code>	weighted hyperedge
WkHyperEdge	<code>~%#</code>	key-weighted hyperedge
LHyperEdge	<code>~+</code>	labeled hyperedge
LkHyperEdge	<code>~#+</code>	key-labeled hyperedge
WLHyperEdge	<code>~%+</code>	weighted labeled hyperedge
WkLHyperEdge	<code>~%#+</code>	key-weighted labeled hyperedge
WLkHyperEdge	<code>~%+#+</code>	weighted key-labeled hyperedge
WkLkHyperEdge	<code>~%#+#</code>	key-weighted key-labeled hyperedge
Directed hyperedges		
DiHyperEdge	<code>~></code>	directed hyperedge
WDiHyperEdge	<code>~%></code>	weighted directed hyperedge
WkDiHyperEdge	<code>~%#></code>	key-weighted directed hyperedge
LDiHyperEdge	<code>~+></code>	labeled directed hyperedge
LkDiHyperEdge	<code>~#+></code>	key-labeled directed hyperedge
WLDiHyperEdge	<code>~%+></code>	weighted labeled directed hyperedge
WkLDiHyperEdge	<code>~%#+></code>	key-weighted labeled directed hyperedge
WLkDiHyperEdge	<code>~%+#+></code>	weighted key-labeled directed hyperedge
WkLkDiHyperEdge	<code>~%#+#></code>	key-weighted key-labeled directed hyperedge
Undirected edges		
UnDiEdge	<code>~</code>	undirected edge
WUnDiEdge	<code>~%</code>	weighted undirected edge
WkUnDiEdge	<code>~%#</code>	key-weighted undirected edge
LUnDiEdge	<code>~+</code>	labeled undirected edge
LkUnDiEdge	<code>~#+</code>	key-labeled undirected edge
WLUnDiEdge	<code>~%+</code>	weighted labeled undirected edge

Edge Class	Shortcut/Operator	Description
WkLUnDiEdge	$\sim\%#+$	key-weighted labeled undirected edge
WLkUnDiEdge	$\sim\%+\#$	weighted key-labeled undirected edge
WkLkUnDiEdge	$\sim\%#\#+$	key-weighted key-labeled undirected edge
Directed edges		
DiEdge	$\sim>$	directed edge
WDiEdge	$\sim\%>$	weighted directed edge
WkDiEdge	$\sim\%#\gt;$	key-weighted directed edge
LDiEdge	$\sim+>$	labeled directed edge
LkDiEdge	$\sim+\#>$	key-labeled directed edge
WLDiEdge	$\sim\%+>$	weighted labeled directed edge
WkLDiEdge	$\sim\%#\+>$	key-weighted labeled directed edge
WLkDiEdge	$\sim\%+\#>$	weighted key-labeled directed edge
WkLkDiEdge	$\sim\%#\+\#>$	key-weighted key-labeled directed edge

You saw the power of graph for Scala: the edges can be weighted and we may potentially construct a multigraph (key-labeled edges allow multiple edges for a pair of source and destination nodes).

If you run SBT on the preceding project with the Scala file in the `src/main/scala` directory, the output will be as follows:

```
[akozlov@Alexanders-MacBook-Pro chapter07 (master)]$ sbt
[info] Loading project definition from /Users/akozlov/Src/Book/ml-in-
scala/chapter07/project
[info] Set current project to Working with Graph Algorithms (in build
file:/Users/akozlov/Src/Book/ml-in-scala/chapter07/)
> run
[warn] Multiple main classes detected. Run 'show discoveredMainClasses'
to see the list
```

Multiple main classes detected, select one to run:

```
[1] org.akozlov.chapter07.ConstrainedDAG
[2] org.akozlov.chapter07.EnronEmail
[3] org.akozlov.chapter07.InfluenceDiagram
```

```
[4] org.akozlov.chapter07.InfluenceDiagramToJson

Enter number: 3

[info] Running org.akozlov.chapter07.InfluenceDiagram
'Weather';'Vacation Activity';'Satisfaction';'Weather
Forecast';'Weather'~>'Weather Forecast' 'Forecast;'Weather'~>'S
atisfaction' 'Deterministic;'Vacation Activity'~>'Satisfaction'
'Deterministic;'Weather Forecast'~>'Vacation Activity' 'Decision
Directed: true
Acyclic: true
'Weather';'Vacation Activity';'Satisfaction';'Recommend to a
Friend';'Weather Forecast';'Weather'~>'Weather Forecast' 'Forecast;'Wea
ther'~>'Satisfaction' 'Deterministic;'Vacation Activity'~>'Satisfaction'
'Deterministic;'Satisfaction'~>'Recommend to a Friend'
'Probabilistic;'Weather Forecast'~>'Vacation Activity' 'Decision
Directed: true
Acyclic: true
```

If continuous compilation is enabled, the main method will be run as soon as SBT detects that the file has changed (in the case of multiple classes having the main method, SBT will ask you which one to run, which is not great for interactivity; so you might want to limit the number of executable classes).

I will cover different output formats in a short while, but let's first see how to perform simple operations on the graph.

Adding nodes and edges

First, we already know that the graph is directed and acyclic, which is a required property for all decision diagrams so that we know we did not make a mistake. Let's say that I want to make the graph more complex and add a node that will indicate the likelihood of me recommending a vacation in Portland, Oregon to another person. The only thing I need to add is the following line:

```
g += ("'Satisfaction'" ~+> "'Recommend to a Friend'") ("Probabilistic")
```

If you have continuous compilation/run enabled, you will immediately see the changes after pressing the **Save File** button:

```
'Weather';'Vacation Activity';'Satisfaction';'Recommend to a Friend';'Weather Forecast';'Weather'~>'Weather Forecast' 'Forecast;'Weather'~>'Satisfaction' 'Deterministic;'Vacation Activity'~>'Satisfaction' 'Deterministic;'Satisfaction'~>'Recommend to a Friend' 'Probabilistic;'Weather Forecast'~>'Vacation Activity' 'Decision  
Directed: true  
Acyclic: true
```

Now, if we want to know the parents of the newly introduced node, we can simply run the following code:

```
println((g get "'Recommend to a Friend'").incoming)  
  
Set('Satisfaction'~>'Recommend to a Friend' 'Probabilistic')
```

This will give us a set of parents for a specific node—and thus drive the decision making process. If we add a cycle, the acyclic method will automatically detect it:

```
g += ("'Satisfaction'" ~+> "'Weather'")("Cyclic")  
println(g.mkString(";")) println("Directed: " + g.isDirected)  
println("Acyclic: " + g.isAcyclic)  
  
'Weather';'Vacation Activity';'Satisfaction';'Recommend to a Friend';'Weather Forecast';'Weather'~>'Weather Forecast' 'Forecast;'Weather'~>'Satisfaction' 'Deterministic;'Vacation Activity'~>'Satisfaction' 'Deterministic;'Satisfaction'~>'Recommend to a Friend' 'Probabilistic;'Satisfaction'~>'Weather' 'Cyclic;'Weather Forecast'~>'Vacation Activity' 'Decision  
Directed: true  
Acyclic: false
```

Note that you can create the graphs completely programmatically:

```
var n, m = 0; val f = Graph.fill(45){ m = if (m < 9) m + 1 else { n = if (n < 8) n + 1 else 8; n + 1 }; m ~ n }  
  
println(f.nodes)  
println(f.edges)  
println(f)  
  
println("Directed: " + f.isDirected)  
println("Acyclic: " + f.isAcyclic)  
  
NodeSet(0, 9, 1, 5, 2, 6, 3, 7, 4, 8)
```

```

EdgeSet(9~0, 9~1, 9~2, 9~3, 9~4, 9~5, 9~6, 9~7, 9~8, 1~0, 5~0, 5~1,
5~2, 5~3, 5~4, 2~0, 2~1, 6~0, 6~1, 6~2, 6~3, 6~4, 6~5, 3~0, 3~1, 3~2,
7~0, 7~1, 7~2, 7~3, 7~4, 7~5, 7~6, 4~0, 4~1, 4~2, 4~3, 8~0, 8~1, 8~2,
8~3, 8~4, 8~5, 8~6, 8~7)
Graph(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1~0, 2~0, 2~1, 3~0, 3~1, 3~2, 4~0,
4~1, 4~2, 4~3, 5~0, 5~1, 5~2, 5~3, 5~4, 6~0, 6~1, 6~2, 6~3, 6~4, 6~5,
7~0, 7~1, 7~2, 7~3, 7~4, 7~5, 7~6, 8~0, 8~1, 8~2, 8~3, 8~4, 8~5, 8~6,
8~7, 9~0, 9~1, 9~2, 9~3, 9~4, 9~5, 9~6, 9~7, 9~8)
Directed: false
Acyclic: false

```

Here, the element computation provided as the second parameter to the fill method is repeated 45 times (the first parameter). The graph connects every node to all of its predecessors, which is also known as a clique in the graph theory.

Graph constraints

Graph for Scala enables us to set constraints that cannot be violated by any future graph update. This comes in handy when we want to preserve some detail in the graph structure. For example, a **Directed Acyclic Graph (DAG)** should not contain cycles. Two constraints are currently implemented as a part of the `scalax.collection.constrained.constraints` package – connected and acyclic, as follows:

```

package org.akozlov.chapter07

import scalax.collection.GraphPredef._, scalax.collection.GraphEdge._
import scalax.collection.constrained.{Config, ConstraintCompanion,
Graph => DAG}
import scalax.collection.constrained.constraints.{Connected, Acyclic}

object AcyclicWithSideEffect extends ConstraintCompanion[Acyclic] {
    def apply [N, E[X] <: EdgeLikeIn[X]] (self: DAG[N,E]) =
        new Acyclic[N,E] (self) {
            override def onAdditionRefused(refusedNodes: Iterable[N],
refusedEdges: Iterable[E[N]], graph: DAG[N,E]) = {
                println("Addition refused: " + "nodes = " + refusedNodes
                    + ", edges = " + refusedEdges)
                true
            }
        }
    }

object ConnectedWithSideEffect extends ConstraintCompanion[Connected]
{

```

```
def apply [N, E[X] <: EdgeLikeIn[X]] (self: DAG[N, E]) =  
  new Connected[N, E] (self) {  
    override def onSubtractionRefused(refusedNodes:  
      Iterable[DAG[N, E]#NodeT],  
      refusedEdges: Iterable[DAG[N, E]#EdgeT],  
      graph: DAG[N, E]) = {  
      println("Subtraction refused: " + "nodes = " +  
        refusedNodes + ", edges = " + refusedEdges)  
      true  
    }  
  }  
  
class CycleException(msg: String) extends  
IllegalArgumentException(msg)  
object ConstrainedDAG extends App {  
  implicit val conf: Config = ConnectedWithSideEffect &&  
  AcyclicWithSideEffect  
  val g = DAG(1~>2, 1~>3, 2~>3, 3~>4) // Graph()  
  println(g ++ List(1~>4, 3~>1))  
  println(g - 2~>3)  
  println(g - 2)  
  println((g + 4~>5) - 3)  
}
```

Here is the command to run the program that tries to add or remove nodes that violate the constraints:

```
[akozlov@Alexanders-MacBook-Pro chapter07 (master)]$ sbt "run-main org.  
akozlov.chapter07.ConstrainedDAG"  
[info] Loading project definition from /Users/akozlov/Src/Book/ml-in-  
scala/chapter07/project  
[info] Set current project to Working with Graph Algorithms (in build  
file:/Users/akozlov/Src/Book/ml-in-scala/chapter07/)  
[info] Running org.akozlov.chapter07.ConstrainedDAG  
Addition refused: nodes = List(), edges = List(1~>4, 3~>1)  
Graph(1, 2, 3, 4, 1~>2, 1~>3, 2~>3, 3~>4)  
Subtraction refused: nodes = Set(), edges = Set(2~>3)  
Graph(1, 2, 3, 4, 1~>2, 1~>3, 2~>3, 3~>4)  
Graph(1, 3, 4, 1~>3, 3~>4)  
Subtraction refused: nodes = Set(3), edges = Set()  
Graph(1, 2, 3, 4, 5, 1~>2, 1~>3, 2~>3, 3~>4, 4~>5)  
[success] Total time: 1 s, completed May 1, 2016 1:53:42 PM
```

Adding or subtracting nodes that violate one of the constraints is rejected. The programmer can also specify a side effect if an attempt to add or subtract a node that violates the condition is made.

JSON

Graph for Scala supports importing/exporting graphs to JSON, as follows:

```
object InfluenceDiagramToJson extends App {

    val g = Graph[String,LDiEdge]((("Weather" ~> "Weather Forecast") "Forecast"), ("Weather Forecast" ~> "Vacation Activity") "Decision"), ("Vacation Activity" ~> "Satisfaction") "Deterministic"), ("Weather" ~> "Satisfaction") "Deterministic"), ("Satisfaction" ~> "Recommend to a Friend") ("Probabilistic"))

    import scalax.collection.io.json.descriptor.predefined.{LDi}
    import scalax.collection.io.json.descriptor.StringNodeDescriptor
    import scalax.collection.io.json._

    val descriptor = new Descriptor[String] (
        defaultNodeDescriptor = StringNodeDescriptor,
        defaultEdgeDescriptor = LDi.descriptor[String, String] ("Edge")
    )

    val n = g.toJson(descriptor)
    println(n)
    import net.liftweb.json._
    println(Printer.pretty(JsonAST.render(JsonParser.parse(n))))
}
```

To produce a JSON representation for a sample graph, run:

```
[kozlov@Alexanders-MacBook-Pro chapter07 (master)]$ sbt "run-main org.akozlov.chapter07.InfluenceDiagramToJson"
[info] Loading project definition from /Users/akozlov/Src/Book/ml-in-scala/chapter07/project
[info] Set current project to Working with Graph Algorithms (in build file:/Users/akozlov/Src/Book/ml-in-scala/chapter07/)
[info] Running org.akozlov.chapter07.InfluenceDiagramToJson
{
  "nodes": [ ["Recommend to a Friend"], ["Satisfaction"], ["Vacation Activity"], ["Weather Forecast"], ["Weather"] ],
  "edges": [
    { "source": "Weather", "target": "Satisfaction", "label": "Deterministic" },
    { "source": "Weather Forecast", "target": "Vacation Activity", "label": "Decision" },
    { "source": "Vacation Activity", "target": "Satisfaction", "label": "Deterministic" },
    { "source": "Satisfaction", "target": "Recommend to a Friend", "label": "Deterministic" },
    { "source": "Weather", "target": "Recommend to a Friend", "label": "Probabilistic" }
  ]
}
```

```
"edges": [{  
    "n1": "'Weather'",  
    "n2": "'Weather Forecast'",  
    "label": "Forecast"  
}, {  
    "n1": "'Vacation Activity'",  
    "n2": "'Satisfaction'",  
    "label": "Deterministic"  
}, {  
    "n1": "'Weather'",  
    "n2": "'Satisfaction'",  
    "label": "Deterministic"  
}, {  
    "n1": "'Weather Forecast'",  
    "n2": "'Vacation Activity'",  
    "label": "Decision"  
}, {  
    "n1": "'Satisfaction'",  
    "n2": "'Recommend to a Friend'",  
    "label": "Probabilistic"  
}]  
}  
[success] Total time: 1 s, completed May 1, 2016 1:55:30 PM
```

For more complex structures, one might need to write custom descriptors, serializers, and deserializers (refer to <http://www.scala-graph.org/api/json/api/#scalax.collection.io.json.package>).

GraphX

While graph for Scala may be considered a DSL for graph operations and querying, one should go to GraphX for scalability. GraphX is build on top of a powerful Spark framework. As an example of Spark/GraphX operations, I'll use the CMU Enron e-mail dataset (about 2 GB). The actual semantic analysis of the e-mail content is not going to be important to us until the next chapters. The dataset can be downloaded from the CMU site. It has e-mail from mailboxes of 150 users, primarily Enron managers, and about 517,401 e-mails between them. The e-mails may be considered as an indication of a relation (edge) between two people: Each email is an edge between a source (`From:`) and a destination (`To:`) vertices.

Since GraphX requires the data in RDD format, I'll have to do some preprocessing. Luckily, it is extremely easy with Scala – this is why Scala is the perfect language for semi-structured data. Here is the code:

```
package org.akozlov.chapter07

import scala.io.Source

import scala.util.hashing.{MurmurHash3 => Hash}
import scala.util.matching.Regex

import java.util.{Date => javaDateTime}

import java.io.File
import net.liftweb.json._
import Extraction._
import Serialization.{read, write}

object EnronEmail {

    val emailRe = """[a-zA-Z0-9_.+\-]+@enron.com""".r.unanchored

    def emails(s: String) = {
        for (email <- emailRe findAllIn s) yield email
    }

    def hash(s: String) = {
        java.lang.Integer.MAX_VALUE.toLong + Hash.stringHash(s)
    }
}
```

```
}

val messageRe =
    """(?:Message-ID:\s+)(<[A-Za-z0-9_.+\-@]+>) (?s) (?:.*)? (?m)
   |(?:Date:\s+)(.*?)$ (?:.*)?
   |(?:From:\s+)([a-zA-Z0-9_.+\-]+@enron.com) (?:.*)?
   |(?:Subject: )(.*)$""".stripMargin.r.unanchored

case class Relation(from: String, fromId: Long, to: String, toId: Long, source: String, messageId: String, date: javaDateTime, subject: String)

implicit val formats = Serialization.formats(NoTypeHints)

def getFileTree(f: File): Stream[File] =
    f #:: (if (f.isDirectory) f.listFiles().toStream.
flatMap(getFileTree) else Stream.empty)

def main(args: Array[String]) {
    getFileTree(new File(args(0))).par.map {
        file =>
        "\\\\".r findFirstIn file.getName match {
            case Some(x) =>
                try {
                    val src = Source.fromFile(file, "us-ascii")
                    val message = try src.mkString finally src.close()
                    message match {
                        case messageRe(messageId, date, from, subject) =>
                            val fromLower = from.toLowerCase
                            for (to <- emails(message).filter(_ != fromLower).toList.distinct)
                                println(write(Relation(fromLower, hash(fromLower),
                                    to, hash(to), file.toString, messageId, new
                                    javaDateTime(date), subject)))
                        case _ =>
                    }
                } catch {
                    case e: Exception => System.err.println(e)
                }
            case _ =>
        }
    }
}
```

First, we use the `MurmurHash3` class to generate node IDs, which are of type `Long`, as they are required for each node in GraphX. The `emailRe` and `messageRe` are used to match the file content to find the required content. Scala allows you to parallelize the programs without much work.

Note the `par` call on line 50, `getFileTree(new File(args(0))).par.map`. This will make the loop parallel. If processing the whole Enron dataset can take up to an hour even on 3 GHz processor, adding parallelization reduces it by about 8 minutes on a 32-core Intel Xeon E5-2630 2.4 GHz CPU Linux machine (it took 15 minutes on an Apple MacBook Pro with 2.3 GHz Intel Core i7).

Running the code will produce a set of JSON records that can be loaded into Spark (to run it, you'll need to put **joda-time** and **lift-json** library jars on the classpath), as follows:

```
# (mkdir Enron; cd Enron; wget -O - http://www.cs.cmu.edu/~./enron/enron_
mail_20150507.tgz | tar xzvf -)

...
# sbt --error "run-main org.akozlov.chapter07.EnronEmail Enron/maildir" >
graph.json

# spark --driver-memory 2g --executor-memory 2g
...
scala> val df = sqlContext.read.json("graph.json")
df: org.apache.spark.sql.DataFrame = [[date: string, from: string,
fromId: bigint, messageId: string, source: string, subject: string, to:
string, toId: bigint]]
```

Nice! Spark was able to figure out the fields and types on its own. If Spark was not able to parse all the records, one would have a `_corrupt_record` field containing the unparsed records (one of them is the `[success]` line at the end of the dataset, which can be filtered out with a `grep -Fv [success]`). You can see them with the following command:

```
scala> df.select("_corrupt_record").collect.foreach(println)
...
```

The nodes (people) and edges (relations) datasets can be extracted with the following commands:

```
scala> import org.apache.spark._
...
scala> import org.apache.spark.graphx._
...
```

```
scala> import org.apache.spark.rdd.RDD
...
scala> val people: RDD[(VertexId, String)] = df.select("fromId", "from").
unionAll(df.select("toId", "to")).na.drop.distinct.map( x => (x.get(0).
toString.toLong, x.get(1).toString))
people: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
String)] = MapPartitionsRDD[146] at map at <console>:28

scala> val relationships = df.select("fromId", "toId", "messageId",
"subject").na.drop.distinct.map( x => Edge(x.get(0).toString.toLong,
x.get(1).toString.toLong, (x.get(2).toString, x.get(3).toString)))
relationships: org.apache.spark.rdd.RDD[org.apache.spark.graphx.
Edge[(String, String)]] = MapPartitionsRDD[156] at map at <console>:28

scala> val graph = Graph(people, relationships).cache
graph: org.apache.spark.graphx.Graph[String, (String, String)] = org.
apache.spark.graphx.impl.GraphImpl@7b59aa7b
```

Node IDs in GraphX

As we saw in Graph for Scala, specifying the edges is sufficient for defining the nodes and the graph. In Spark/GraphX, nodes need to be extracted explicitly, and each node needs to be associated with n id of the Long type. While this potentially limits the flexibility and the number of unique nodes, it enhances the efficiency. In this particular example, generating node ID as a hash of the e-mail string was sufficient as no collisions were detected, but the generation of unique IDs is usually a hard problem to parallelize.

The first GraphX graph is ready!! It took a bit more work than Scala for Graph, but now it's totally ready for distributed processing. A few things to note: first, we needed to explicitly convert the fields to Long and String as the Edge constructor needed help in figuring out the types. Second, Spark might need to optimize the number of partitions (likely, it created too many):

```
scala> graph.vertices.getNumPartitions
res1: Int = 200

scala> graph.edges.getNumPartitions
res2: Int = 200
```

To repartition, there are two calls: repartition and coalesce. The latter tries to avoid shuffle, as follows:

```
scala> val graph = Graph(people.coalesce(6), relationships.coalesce(6))
graph: org.apache.spark.graphx.Graph[String, (String, String)] = org.
apache.spark.graphx.impl.GraphImpl@5dc7d016
```

```
scala> graph.vertices.getNumPartitions
res10: Int = 6
```

```
scala> graph.edges.getNumPartitions
res11: Int = 6
```

However, this might limit parallelism if one performs computations over a large cluster. Finally, it's a good idea to use `cache` method that pins the data structure in memory:

```
scala> graph.cache
res12: org.apache.spark.graphx.Graph[String, (String, String)] = org.
apache.spark.graphx.impl.GraphImpl@5dc7d016
```

It took a few more commands to construct a graph in Spark, but four is not too bad. Let's compute some statistics (and show the power of Spark/GraphX, in the following table:

Computing basic statistics on Enron e-mail graph.

Statistics	Spark command	Value for Enron
Total # of relations (pairwise communications)	graph.numEdges	3,035,021
Number of e-mails (message IDs)	graph.edges.map(e => e.attr._1).distinct.count	371,135
Number of connected pairs	graph.edges.flatMap(e => List((e.srcId, e.dstId), (e.dstId, e.srcId))).distinct.count / 2	217,867
Number of one-way communications	graph.edges.flatMap(e => List((e.srcId, e.dstId), (e.dstId, e.srcId))).distinct.count - graph.edges.map(e => (e.srcId, e.dstId)).distinct.count	193,183

Statistics	Spark command	Value for Enron
Number of distinct subject lines	graph.edges.map(e => e.attr._2).distinct.count	110,273
Total # of nodes	graph.numVertices	23,607
Number of destination-only nodes	graph. numVertices - graph.edges.map(e => e.srcId).distinct.count	17,264
Number of source-only nodes	graph. numVertices - graph.edges.map(e => e.dstId).distinct.count	611

Who is getting e-mails?

One of the most straightforward ways to estimate people's importance in an organization is to look at the number of connections or the number of incoming and outgoing communicates. The GraphX graph has built-in `inDegrees` and `outDegrees` methods. To rank the emails with respect to the number of incoming emails, run:

```
scala> people.join(graph.inDegrees).sortBy(_. _2 . _2 , ascending=false) .
take(10).foreach(println)
(268746271, (richard.shapiro@enron.com, 18523))
(1608171805, (steven.kean@enron.com, 15867))
(1578042212, (jeff.dasovich@enron.com, 13878))
(960683221, (tana.jones@enron.com, 13717))
(3784547591, (james.steffes@enron.com, 12980))
(1403062842, (sara.shackleton@enron.com, 12082))
(2319161027, (mark.taylor@enron.com, 12018))
(969899621, (mark.guzman@enron.com, 10777))
(1362498694, (geir.solberg@enron.com, 10296))
(4151996958, (ryan.slinger@enron.com, 10160))
```

To rank the emails with respect to the number of egressing emails, run:

```
scala> people.join(graph.outDegrees).sortBy(_. _2 . _2 , ascending=false) .
take(10).foreach(println)
(1578042212, (jeff.dasovich@enron.com, 139786))
(2822677534, (veronica.espinoza@enron.com, 106442))
(3035779314, (pete.davis@enron.com, 94666))
(2346362132, (rhonda.denton@enron.com, 90570))
(861605621, (cheryl.johnson@enron.com, 74319))
```

```
(14078526, (susan.mara@enron.com, 58797))  
(2058972224, (jae.black@enron.com, 58718))  
(871077839, (ginger.dernehls@enron.com, 57559))  
(3852770211, (lorna.brennan@enron.com, 50106))  
(241175230, (mary.hain@enron.com, 40425))  
...
```

Let's apply some more complex algorithms to the Enron dataset.

Connected components

Connected components determine whether the graph is naturally partitioned into several parts. In the Enron relationship graph, this would mean that two or several groups communicate mostly between each other:

```
scala> val groups = org.apache.spark.graphx.lib.ConnectedComponents.  
run(graph).vertices.map(_.value).distinct.cache  
groups: org.apache.spark.rdd.RDD[org.apache.spark.graphx.VertexId] =  
MapPartitionsRDD[2404] at distinct at <console>:34  
  
scala> groups.count  
res106: Long = 18  
  
scala> people.join(groups.map( x => (x, x))).map(x => (x._1, x._2._1)).  
sortBy(_.value).collect.foreach(println)  
(332133, laura.beneville@enron.com)  
(81833994, gpg.me-q@enron.com)  
(115247730, dl-ga-enron_debtor@enron.com)  
(299810291, gina.peters@enron.com)  
(718200627, techsupport.notices@enron.com)  
(847455579, paul.de@enron.com)  
(919241773, etc.survey@enron.com)  
(1139366119, enron.global.services.--.us@enron.com)  
(1156539970, shelley.ariel@enron.com)  
(1265773423, dl-ga-all_ews_employees@enron.com)  
(1493879606, chairman.ees@enron.com)  
(1511379835, gary.allen.--.safety.specialist@enron.com)  
(2114016426, executive.robert@enron.com)  
(2200225669, ken.board@enron.com)
```

```
(2914568776,ge.americas@enron.com)
(2934799198,yowman@enron.com)
(2975592118,tech.notices@enron.com)
(3678996795,mail.user@enron.com)
```

We see 18 groups. Each one of the groups can be counted and extracted by filtering the ID. For instance, the group associated with `etc.survey@enron.com` can be found by running a SQL query on DataFrame:

```
scala> df.filter("fromId = 919241773 or toId = 919241773").select("date",
"from","to","subject","source").collect.foreach(println)
[2000-09-19T18:40:00.000Z,survey.test@enron.com,etc.survey@enron.com,NO
ACTION REQUIRED - TEST,Enron/maildir/dasovich-j/all_documents/1567.]
[2000-09-19T18:40:00.000Z,survey.test@enron.com,etc.survey@enron.com,NO
ACTION REQUIRED - TEST,Enron/maildir/dasovich-j/notes_inbox/504.]
```

This group is based on a single e-mail sent on September 19, 2000, from `survey.test@enron.com` to `etc.survey@enron.com`. The e-mail is listed twice, only because it ended up in two different folders (and has two distinct message IDs). Only the first group, the largest subgraph, contains more than two e-mail addresses in the organization.

Triangle counting

The triangle counting algorithm is relatively straightforward and can be computed in the following three steps:

1. Compute the set of neighbors for each vertex.
2. For each edge, compute the intersection of the sets and send the count to both vertices.
3. Compute the sum at each vertex and divide by two, as each triangle is counted twice.

We need to convert the multigraph to an undirected graph with `srcId < dstId`, which is a precondition for the algorithm:

```
scala> val unedges = graph.edges.map(e => if (e.srcId < e.dstId)
(e.srcId, e.dstId) else (e.dstId, e.srcId)).map( x => Edge(x._1, x._2,
1)).cache
unedges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] =
MapPartitionsRDD[87] at map at <console>:48
```

```
scala> val ungraph = Graph(people, unedges).partitionBy(org.apache.spark.
graphx.PartitionStrategy.EdgePartitionID, 10).cache
```

```

ungraph: org.apache.spark.graphx.Graph[String,Int] = org.apache.spark.
graphx.impl.GraphImpl@77274fff

scala> val triangles = org.apache.spark.graphx.lib.TriangleCount.
run(ungraph).cache

triangles: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.
graphx.impl.GraphImpl@6aec6da1

scala> people.join(triangles.vertices).map(t => (t._2._2,t._2._1)).
sortBy(_._1, ascending=false).take(10).foreach(println)
(31761,sally.beck@enron.com)
(24101,louise.kitchen@enron.com)
(23522,david.forster@enron.com)
(21694,kenneth.lay@enron.com)
(20847,john.lavorato@enron.com)
(18460,david.oxley@enron.com)
(17951,tammie.schoppe@enron.com)
(16929,steven.kean@enron.com)
(16390,tana.jones@enron.com)
(16197,julie.clyatt@enron.com)

```

While there is no direct relationship between the triangle count and the importance of people in the organization, the people with higher triangle count arguably are more social—even though a clique or a strongly connected component count might be a better measure.

Strongly connected components

In the mathematical theory of directed graphs, a subgraph is said to be strongly connected if every vertex is reachable from every other vertex. It could happen that the whole graph is just one strongly connected component, but on the other end of the spectrum, each vertex could be its own connected component.

If you contract each connected component to a single vertex, you get a new directed graph that has a property to be without cycles—acyclic.

The algorithm for SCC detection is already built into GraphX:

```
scala> val components = org.apache.spark.graphx.lib.  
StronglyConnectedComponents.run(graph, 100).cache  
components: org.apache.spark.graphx.Graph[org.apache.spark.  
graphx.VertexId,(String, String)] = org.apache.spark.graphx.impl.  
GraphImpl@55913bc7  
  
scala> components.vertices.map(_.value._2).distinct.count  
res2: Long = 17980  
  
scala> people.join(components.vertices.map(_.value._2).distinct.map( x => (x,  
x))).map(x => (x.value._1, x.value._2._1)).sortBy(_.value._1).collect.foreach(println)  
(332133,laura.beneville@enron.com)  
(466265,medmonds@enron.com)  
(471258,.jane@enron.com)  
(497810,.kimberly@enron.com)  
(507806,aleck.dadson@enron.com)  
(639614,j..bonin@enron.com)  
(896860,imceanotes-hbcamp+40aep+2ecom+40enron@enron.com)  
(1196652,enron.legal@enron.com)  
(1240743,thi.ly@enron.com)  
(1480469,ofedb12a77a.a6162183-on86256988.005b6308@enron.com)  
(1818533,fran.i.mayes@enron.com)  
(2337461,michael.marryott@enron.com)  
(2918577,houston.resolution.center@enron.com)
```

There are 18,200 strongly connected components with only an average 23,787/18,200 = 1.3 users per group.

PageRank

The PageRank algorithm gives us an estimate of how important a person by analysing the links, which are the emails in this case. For example, let's run PageRank on Enron email graph:

```
scala> val ranks = graph.pageRank(0.001).vertices
ranks: org.apache.spark.graphx.VertexRDD[Double] = VertexRDDImpl[955] at
RDD at VertexRDD.scala:57

scala> people.join(ranks).map(t => (t._2._2,t._2._1)).sortBy(_._1,
ascending=false).take(10).foreach(println)

scala> val ranks = graph.pageRank(0.001).vertices
ranks: org.apache.spark.graphx.VertexRDD[Double] = VertexRDDImpl[955] at
RDD at VertexRDD.scala:57

scala> people.join(ranks).map(t => (t._2._2,t._2._1)).sortBy(_._1,
ascending=false).take(10).foreach(println)
(32.073722548483325,tana.jones@enron.com)
(29.086568868043248,sara.shackleton@enron.com)
(28.14656912897315,louise.kitchen@enron.com)
(26.57894933459292,vince.kaminski@enron.com)
(25.865486865014493,sally.beck@enron.com)
(23.86746232662471,john.lavorato@enron.com)
(22.489814482022275,jeff.skilling@enron.com)
(21.968039409295585,mark.taylor@enron.com)
(20.903053536275547,kenneth.lay@enron.com)
(20.39124651779771,gerald.nemec@enron.com)
```

Ostensibly, these are the go-to people. PageRank tends to emphasize the incoming edges, and Tana Jones returns to the top of the list compared to the 9th place in the triangle counting.

SVD++

SVD++ is a recommendation engine algorithm, developed specifically for Netflix competition by Yahuda Koren and team in 2008 – the original paper is still out there in the public domain and can be Googled as `kdd08koren.pdf`. The specific implementation comes from the .NET *MyMediaLite* library by ZenoGarther (<https://github.com/zenogantner/MyMediaLite>), who granted Apache 2 license to the Apache Foundation. Let's assume I have a set of users (on the left) and items (on the right):

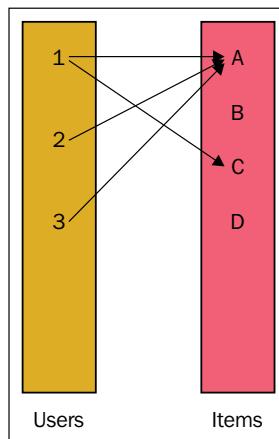


Figure 07-1. A graphical representation of a recommendation problem as a bipartite graph.

The preceding diagram is a graphical representation of the recommendation problem. The nodes on the left represent users. The nodes on the right represent items. User **1** recommends items **A** and **C**, while users **2** and **3** recommend only a single item **A**. The rest of the edges are missing. The common question is to find recommendation ranking of the rest of the items, the edges may also have a weight or recommendation strength attached to them. The graph is usually sparse. Such graph is also often called bipartite, as the edges only go from one set of nodes to another set of nodes (the user does not recommend another user).

For the recommendation engine, we typically need two types of nodes – users and items. The recommendations are based on the rating matrix of (user, item, and rating) tuples. One of the implementations of the recommendation algorithm is based on **Singular Value Decomposition (SVD)** of the preceding matrix. The final scoring has four components: the baseline, which is the sum of average for the whole matrix, average for the users, and average for the items, as follows:

$$r_{\{u,i\}} = \mu + b_u + b_i$$

Here, the μ , b_u , and b_i can be understood as the averages for the whole population, user (among all user recommendations), and item (among all the users). The final part is the Cartesian product of two rows:

$$r_{\{i,j\}} = \mu + b_u + b_i u + p_u^T q_i$$

The problem is posed as a minimization problem (refer to *Chapter 4, Supervised and Unsupervised Learning*):

$$\min_{p_*, q_*, b_*} \sum_{u,i} (r_{ui} - \mu + b_u + b_i + p_u^T q_i) + \lambda_3 (\|p_u\|^2 + \|q_i\|^2 + b_u^2 + b_i^2)$$

Here, λ_3 is a regularization coefficient also discussed in *Chapter 4, Supervised and Unsupervised Learning*. So, each user is associated with a set of numbers $((b_u, p_u))$, and each item with (b_i, q_i) . In this particular implementation, the optimal coefficients are found by gradient descent. This is the basic of SVD optimization. In linear algebra, SVD takes an arbitrary $m \times n$ matrix A and represents it as a product of an orthogonal $m \times n$ matrix U , a diagonal $m \times n$ matrix Σ , and a $m \times n$ unitary matrix V , for example, the columns are mutually orthogonal. Arguably, if one takes the largest r entries of the Σ matrix, the product is reduced to the product of a very tall $m \times r$ matrix and a wide $r \times n$ matrix, where r is called the rank of decomposition. If the remaining values are small, the new $(m+n) \times r$ numbers approximate the original $m \times n$ numbers for the relation, A . If m and n are large to start with, and in practical online shopping situations, m is the items and can be in hundreds of thousands, and n is the users and can be hundreds of millions, the saving can be substantial. For example, for $r=10$, $m=100,000$, and $n=100,000,000$, the savings are as follows:

$$\frac{m \times n}{(m+n) \times r} = \frac{10,000,000,000,000}{1,001,000,000} \sim 10,000$$

SVD can also be viewed as PCA for matrices with $m \neq n$. In the Enron case, we can treat senders as users and recipients as items (we'll need to reassign the node IDs), as follows:

```
scala> val rgraph = graph.partitionBy(org.apache.spark.graphx.  
PartitionStrategy.EdgePartition1D, 10).mapEdges(e => 1).groupEdges(_+_)  
cache  
  
rgraph: org.apache.spark.graphx.Graph[String,Int] = org.apache.spark.  
graphx.impl.GraphImpl@2c1a48d6  
  
scala> val redges = rgraph.edges.map( e => Edge(-e.srcId, e.dstId, Math.  
log(e.attr.toDouble)) ).cache  
redges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Double]] =  
MapPartitionsRDD[57] at map at <console>:36  
  
scala> import org.apache.spark.graphx.lib.SVDPlusPlus  
import org.apache.spark.graphx.lib.SVDPlusPlus  
  
scala> implicit val conf = new SVDPlusPlus.Conf(10, 50, 0.0, 10.0, 0.007,  
0.007, 0.005, 0.015)  
conf: org.apache.spark.graphx.lib.SVDPlusPlus.Conf = org.apache.spark.  
graphx.lib.SVDPlusPlus$Conf@15cdc117  
  
scala> val (svd, mu) = SVDPlusPlus.run(redges, conf)  
svd: org.apache.spark.graphx.Graph[(Array[Double], Array[Double], Double,  
Double),Double] = org.apache.spark.impl.GraphImpl@3050363d  
mu: Double = 1.3773578970633769  
  
scala> val svdRanks = svd.vertices.filter(_.value > 0).map(x => (x._2._3,  
x._1))  
svdRanks: org.apache.spark.rdd.RDD[(Double, org.apache.spark.graphx.  
VertexId)] = MapPartitionsRDD[1517] at map at <console>:31  
  
scala> val svdRanks = svd.vertices.filter(_.value > 0).map(x => (x._1,  
x._2._3))  
svdRanks: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,  
Double)] = MapPartitionsRDD[1520] at map at <console>:31  
  
scala> people.join(svdRanks).sortBy(_.value, ascending=false).map(x =>  
(x._2._2, x._2._1)).take(10).foreach(println)
```

```
(8.864218804309887, jbryson@enron.com)
(5.935146713012661, dl-ga-all_enron_worldwide2@enron.com)
(5.740242927715701, houston.report@enron.com)
(5.441934324464593, a478079f-55e1f3b0-862566fa-612229@enron.com)
(4.910272928389445, pchoi2@enron.com)
(4.701529779800544, dl-ga-all_enron_worldwide1@enron.com)
(4.4046392452058045, eligible.employees@enron.com)
(4.374738019256556, all_ena_egm_eim@enron.com)
(4.303078586979311, dl-ga-all_enron_north_america@enron.com)
(3.8295412053860867, the.mailout@enron.com)
```

The svdRanks is the user-part of the $\mu + b_i$ prediction. The distribution lists take a priority as this is usually used for mass e-mailing. To get the user-specific part, we need to provide the user ID:

```
scala> import com.github.fommil.netlib.BLAS.{getInstance => blas}

scala> def topN(uid: Long, num: Int) = {
    |   val usr = svd.vertices.filter(uid == -_._1).collect()(0)._2
    |   val recs = svd.vertices.filter(_.1 > 0).map( v => (v._1, mu +
usr._3 + v._2._3 + blas.ddot(usr._2.length, v._2._1, 1, usr._2, 1)))
    |   people.join(recs).sortBy(_.2._2, ascending=false).map(x =>
(x._2._2, x._2._1)).take(num)
    | }
topN: (uid: Long, num: Int)Array[(Double, String)]

scala> def top5(x: Long) : Array[(Double, String)] = topN(x, 5)
top5: (x: Long)Array[(Double, String)]
```



```
scala> people.join(graph.inDegrees).sortBy(_.2._2, ascending=false).
map(x => (x._1, x._2._1)).take(10).toList.map(t => (t._2, top5(t._1).
toList)).foreach(println)
richard.shapiro@enron.com,List((4.866184418005094E66,anne.
bertino@enron.com), (3.9246829664352734E66,kgustafs@enron.com),
(3.9246829664352734E66,gweiss@enron.com), (3.871029763863491E66,hill@
enron.com), (3.743135924382312E66,fraser@enron.com))

(steven.kean@enron.com,List((2.445163626935533E66,an
ne.bertino@enron.com), (1.9584692804232504E66,hill@
enron.com), (1.9105427465629028E66,kgustafs@enron.com),
(1.9105427465629028E66,gweiss@enron.com), (1.8931872324048717E66,fraser@
enron.com))
```

```
(jeff.dasovich@enron.com,List((2.8924566115596135E66,
anne.bertino@enron.com), (2.3157345904446663E66,hill@
enron.com), (2.2646318970030287E66,gweiss@enron.
com), (2.2646318970030287E66,kgustafs@enron.com),
(2.2385865127706285E66,fraser@enron.com)))

(tana.jones@enron.com,List((6.1758464471309754E66,elizabeth.
sager@enron.com), (5.279291610047078E66,tana.jones@enron.com),
(4.967589820856654E66,tim.belden@enron.com), (4.909283344915057E66,jeff.
dasovich@enron.com), (4.869177440115682E66,mark.taylor@enron.com)))

(james.steffes@enron.com,List((5.7702834706832735E66,anne.
bertino@enron.com), (4.703038082326939E66,gweiss@enron.com),
(4.703038082326939E66,kgustafs@enron.com), (4.579565962089777E66,hill@
enron.com), (4.4298763869135494E66,george@enron.com)))

(sara.shackleton@enron.com,List((9.198688613290757E67,loui
se.kitchen@enron.com), (8.078107057848099E67,john.lavorato@
enron.com), (6.922806078209984E67,greg.whalley@enron.
com), (6.787266892881456E67,elizabeth.sager@enron.com),
(6.420473603137515E67,sally.beck@enron.com)))

(mark.taylor@enron.com,List((1.302856119148208E66,anne.
bertino@enron.com), (1.0678968544568682E66,hill@enron.com),
(1.031255083546722E66,fraser@enron.com), (1.009319696608474E66,george@
enron.com), (9.901391892701356E65,brad@enron.com)))

(mark.guzman@enron.com,List((9.770393472845669E65,anne.
bertino@enron.com), (7.97370292724488E65,kgustafs@enron.com),
(7.97370292724488E65,gweiss@enron.com), (7.751983820970696E65,hill@enron.
com), (7.500175024539423E65,george@enron.com)))

(geir.solberg@enron.com,List((6.856103529420811E65,anne.
bertino@enron.com), (5.611272903720188E65,gweiss@enron.com),
(5.611272903720188E65,kgustafs@enron.com), (5.436280144720843E65,hill@
enron.com), (5.2621103015001885E65,george@enron.com)))

(ryan.slinger@enron.com,List((5.0579114162531735E65,anne.
bertino@enron.com), (4.136838933824579E65,kgustafs@enron.com),
(4.136838933824579E65,gweiss@enron.com), (4.0110663808847004E65,hill@
enron.com), (3.8821438267917902E65,george@enron.com)))

scala> people.join(graph.outDegrees).sortBy(_.value, ascending=false).
map(x => (x._1, x._2._1)).take(10).toList.map(t => (t._2, top5(t._1).
toList)).foreach(println)

(jeff.dasovich@enron.com,List((2.8924566115596135E66,
anne.bertino@enron.com), (2.3157345904446663E66,hill@
enron.com), (2.2646318970030287E66,gweiss@enron.
com), (2.2646318970030287E66,kgustafs@enron.com),
(2.2385865127706285E66,fraser@enron.com)))
```

```
(veronica.espinoza@enron.com,List((3.135142195254243E65,gw  
eiss@enron.com), (3.135142195254243E65,kgustafs@enron.com),  
(2.773512892785554E65,anne.bertino@enron.com), (2.350799070225962E65,marc  
ia.a.linton@enron.com), (2.2055288158758267E65,robert@enron.com)))  
  
(pete.davis@enron.com,List((5.773492048248794E66,louise.  
kitchen@enron.com), (5.067434612038159E66,john.lavorato@  
enron.com), (4.389028076992449E66,greg.whalley@enron.  
com), (4.1791711984241975E66,sally.beck@enron.com),  
(4.009544764149938E66,elizabeth.sager@enron.com)))  
  
(rhonda.denton@enron.com,List((2.834710591578977E68,louise.  
kitchen@enron.com), (2.488253676819922E68,john.lavorato@  
enron.com), (2.1516048969715738E68,greg.whalley@enron.com),  
(2.0405329247770104E68,sally.beck@enron.com), (1.9877213034021861E68,eliz  
abeth.sager@enron.com)))  
  
(cheryl.johnson@enron.com,List((3.453167402163105E64,ma  
ry.dix@enron.com), (3.208849221485621E64,theresa.byrne@  
enron.com), (3.208849221485621E64,sandy.olofson@enron.com),  
(3.0374270093157086E64,hill@enron.com), (2.886581252384442E64,fraser@  
enron.com)))  
  
(susan.mara@enron.com,List((5.1729089729525785E66,anne.  
bertino@enron.com), (4.220843848723133E66,kgustafs@enron.com),  
(4.220843848723133E66,gweiss@enron.com), (4.1044435240204605E66,hill@  
enron.com), (3.9709951893268635E66,george@enron.com)))  
  
(jae.black@enron.com,List((2.513139130001457E65,anne.bertino@enron.com),  
(2.1037756300035247E65,hill@enron.com), (2.0297519350719265E65,fraser@  
enron.com), (1.9587139280519927E65,george@enron.com),  
(1.947164483486155E65,brad@enron.com)))  
  
(ginger.dernehle@enron.com,List((4.516267307013845E66,anne.  
bertino@enron.com), (3.653408921875843E66,gweiss@enron.com),  
(3.653408921875843E66,kgustafs@enron.com), (3.590298037045689E66,hill@  
enron.com), (3.471781765250177E66,fraser@enron.com)))  
  
(lorna.brennan@enron.com,List((2.0719309635087482E66,anne.  
bertino@enron.com), (1.732651408857978E66,kgustafs@enron.com),  
(1.732651408857978E66,gweiss@enron.com), (1.6348480059915056E66,hill@  
enron.com), (1.5880693846486309E66,george@enron.com)))  
  
(mary.hain@enron.com,List((5.596589595417286E66,anne.bertino@enron.com),  
(4.559474243930487E66,kgustafs@enron.com), (4.559474243930487E66,gweiss@  
enron.com), (4.4421474044331
```

Here, we computed the top five recommended e-mail-to list for top in-degree and out-degree users.

SVD has only 159 lines of code in Scala and can be the basis for some further improvements. SVD++ includes a part based on implicit user feedback and item similarity information. Finally, the Netflix winning solution had also taken into consideration the fact that user preferences are time-dependent, but this part has not been implemented in GraphX yet.

Summary

While one can easily create their own data structures for graph problems, Scala's support for graphs comes from both semantic layer—Graph for Scala is effectively a convenient, interactive, and expressive language for working with graphs—and scalability via Spark and distributed computing. I hope that some of the material exposed in this chapter will be useful for implementing algorithms on top of Scala, Spark, and GraphX. It is worth mentioning that bot libraries are still under active development.

In the next chapter, we'll step down from from our flight in the the skies and look at Scala integration with traditional data analysis frameworks such as statistical language R and Python, which are often used for data munching. Later, in *Chapter 9, NLP in Scala*. I'll look at NLP Scala tools, which leverage complex data structures extensively.

8

Integrating Scala with R and Python

While Spark provides MLlib as a library for machine learning, in many practical situations, R or Python present a more familiar and time-tested interface for statistical computations. In particular, R's extensive statistical library includes very popular ANOVA/MANOVA methods of analyzing variance and variable dependencies/independencies, sets of statistical tests, and random number generators that are not currently present in MLlib. The interface from R to Spark is available under SparkR project. Finally, data analysts know Python's NumPy and SciPy linear algebra implementations for their efficiency as well as other time-series, optimization, and signal processing packages. With R/Python integration, all these familiar functionalities can be exposed to Scala/Spark users until the Spark/MLlib interfaces stabilize and the libraries make their way into the new framework while benefiting the users with Spark's ability to execute workflows in a distributed way across multiple machines.

When people program in R or Python, or with any statistical or linear algebra packages for this matter, they are usually not specifically focusing on the functional programming aspects. As I mentioned in *Chapter 1, Exploratory Data Analysis*, Scala should be treated as a high-level language and this is where it shines. Integration with highly efficient C and Fortran implementations, for example, of the freely available **Basic Linear Algebra Subprograms (BLAS)**, **Linear Algebra Package (LAPACK)**, and **Arnoldi Package (ARPACK)**, is known to find its way into Java and thus Scala (<http://www.netlib.org>, <https://github.com/fommil/netlib-java>). I would like to leave Scala at what it's doing best. In this chapter, however, I will focus on how to use these languages with Scala/Spark.

I will use the publicly available United States Department of Transportation flights dataset for this chapter (<http://www.transtats.bts.gov>).

In this chapter, we will cover the following topics:

- Installing R and configuring SparkR if you haven't done so yet
- Learning about R (and Spark) DataFrames
- Performing linear regression and ANOVA analysis with R
- Performing **Generalized Linear Model (GLM)** modeling with SparkR
- Installing Python if you haven't done so yet
- Learning how to use PySpark and call Python from Scala

Integrating with R

As with many advanced and carefully designed technologies, people usually either love or hate R as a language. One of the reason being that R was one of the first language implementations that tries to manipulate complex objects, even though most of them turn out to be just a list as opposed to struct or map as in more mature modern implementations. R was originally created at the University of Auckland by Ross Ihaka and Robert Gentleman around 1993, and had its roots in the S language developed at Bell Labs around 1976, when most of the commercial programming was still done in Fortran. While R incorporates some functional features such as passing functions as a parameter and map/apply, it conspicuously misses some others such as lazy evaluation and list comprehensions. With all this said, R has a very good help system, and if someone says that they never had to go back to the `help(...)` command to figure out how to run a certain data transformation or model better, they are either lying or just starting in R.

Setting up R and SparkR

To run SparkR, you'll need R version 3.0 or later. Follow the given instructions for the installation, depending on your operating system.

Linux

On a Linux system, detailed installation documentation is available at <https://cran.r-project.org/bin/linux>. However, for example, on a Debian system, one installs it by running the following command:

```
# apt-get update  
...  
# apt-get install r-base r-base-dev  
...
```

To list installed/available packages on the Linux repository site, perform the following command:

```
# apt-cache search "^r-.*" | sort
...
```

R packages, which are a part of r-base and r-recommended, are installed into the /usr/lib/R/library directory. These can be updated using the usual package maintenance tools such as apt-get or aptitude. The other R packages available as precompiled Debian packages, r-cran-* and r-bioc-*, are installed into /usr/lib/R/site-library. The following command shows all packages that depend on r-base-core:

```
# apt-cache rdepends r-base-core
```

This comprises of a large number of contributed packages from CRAN and other repositories. If you want to install R packages that are not provided as package, or if you want to use newer versions, you need to build them from source that requires the r-base-dev development package that can be installed by the following command:

```
# apt-get install r-base-dev
```

This pulls in the basic requirements to compile R packages, such as the development tools group install. R packages may then be installed by the local user/admin from the CRAN source packages, typically from inside R using the R> install.packages() function or R CMD INSTALL. For example, to install the R ggplot2 package, run the following command:

```
> install.packages("ggplot2")
--- Please select a CRAN mirror for use in this session ---
also installing the dependencies 'stringi', 'magrittr', 'colorspace',
'Rcpp', 'stringr', 'RColorBrewer', 'dichromat', 'munsell', 'labeling',
'digest', 'gttable', 'plyr', 'reshape2', 'scales'
```

This will download and optionally compile the package and its dependencies from one of the available sites. Sometime R is confused about the repositories; in this case, I recommend creating a ~/.Rprofile file in the home directory pointing to the closest CRAN repository:

```
$ cat >> ~/.Rprofile << EOF
r =getOption("repos") # hard code the Berkeley repo for CRAN
r["CRAN"] = "http://cran.cnr.berkeley.edu"
options(repos = r)
rm(r)

EOF
```

`~/.Rprofile` contains commands to customize your sessions. One of the commands I recommend to put in there is options (`prompt="R> "`) to be able to distinguish the shell you are working in by the prompt, following the tradition of most tools in this book. The list of known mirrors is available at <https://cran.r-project.org/mirrors.html>.

Also, it is good practice to specify the directory to install system/site/user packages via the following command, unless your OS setup does it already by putting these commands into `~/.bashrc` or system `/etc/profile`:

```
$ export R_LIBS_SITE=${R_LIBS_SITE:-/usr/local/lib/R/site-library:/usr/lib/R/site-library:/usr/lib/R/library}  
$ export R_LIBS_USER=${R_LIBS_USER:-$HOME/R/$(uname -i)-library/${( R --version | grep -o -E [0-9]+\.[0-9]+ | head -1 )}}
```

Mac OS

R for Mac OS can be downloaded, for example, from <http://cran.r-project.org/bin/macosx>. The latest version at the time of the writing is 3.2.3. Always check the consistency of the downloaded package. To do so, run the following command:

```
$ pkgutil --check-signature R-3.2.3.pkg  
Package "R-3.2.3.pkg":  
  Status: signed by a certificate trusted by Mac OS X  
  Certificate Chain:  
    1. Developer ID Installer: Simon Urbanek  
       SHA1 fingerprint: B7 EB 39 5E 03 CF 1E 20 D1 A6 2E 9F D3 17 90 26  
D8 D6 3B EF  
-----  
    2. Developer ID Certification Authority  
       SHA1 fingerprint: 3B 16 6C 3B 7D C4 B7 51 C9 FE 2A FA B9 13 56 41  
E3 88 E1 86  
-----  
    3. Apple Root CA  
       SHA1 fingerprint: 61 1E 5B 66 2C 59 3A 08 FF 58 D1 4A E2 24 52 D1  
98 DF 6C 60
```

The environment settings in the preceding subsection also apply to the Mac OS setup.

Windows

R for Windows can be downloaded from <https://cran.r-project.org/bin/windows/> as an exe installer. Run this executable as an administrator to install R.

One can usually edit the environment setting for **System/User** by following the **Control Panel | System and Security | System | Advanced system settings | Environment Variables** path from the Windows menu.

Running SparkR via scripts

To run SparkR, one needs to run install the `R/install-dev.sh` script that comes with the Spark git tree. In fact, one only needs the shell script and the content of the `R/pkg` directory, which is not always included with the compiled Spark distributions:

```
$ git clone https://github.com/apache/spark.git
Cloning into 'spark'...
remote: Counting objects: 301864, done.

...
$ cp -r R/{install-dev.sh,pkg} $SPARK_HOME/R
...
$ cd $SPARK_HOME
$ ./R/install-dev.sh
* installing *source* package 'SparkR' ...
** R
** inst
** preparing package for lazy loading
Creating a new generic function for 'colnames' in package 'SparkR'
...
$ bin/sparkR

R version 3.2.3 (2015-12-10) -- "Wooden Christmas-Tree"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
```

Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.

Type 'contributors()' for more information and

'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.

Type 'q()' to quit R.

```
Launching java with spark-submit command /home/alex/spark-1.6.1-bin-hadoop2.6/bin/spark-submit "sparkr-shell" /tmp/RtmpgdTfmU/backend_port22446d0391e8
```

Welcome to

/ _/ _ _ _ _ / /
_ \ \ / _ \ \ / _ / ' /
/ __ / . __ \ / , / / / \ \ \ version 1.6.1
/ /

```
Spark context is available as sc, SQL context is available as sqlContext
```

Running Spark via R's command line

Alternatively, we can also initialize Spark from the R command line directly (or from RStudio at <http://rstudio.org/>) using the following commands:

```
R> library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R",
"lib")))
...
R> sc <- sparkR.init(master = Sys.getenv("SPARK_MASTER"), sparkEnvir =
list(spark.driver.memory="1g"))
...
R> sqlContext <- sparkRSQl.init(sc)
```

As described previously in *Chapter 3, Working with Spark and MLlib*, the `SPARK_HOME` environment variable needs to point to your local Spark installation directory and `SPARK_MASTER` and `YARN_CONF_DIR` to the desired cluster manager (local, standalone, mesos, and YARN) and YARN configuration directory if one is using Spark with the YARN cluster manager.

Although most all of the distributions come with a UI, in the tradition of this book and for the purpose of this chapter I'll use the command line.

DataFrames

The DataFrames originally came from R and Python, so it is natural to see them in SparkR.



Please note that the implementation of DataFrames in SparkR is on top of RDDs, so they work differently than the R DataFrames.



The question of when and where to store and apply the schema and other metadata like types has been a topic of active debate recently. On one hand, providing the schema early with the data enables thorough data validation and potentially optimization. On the other hand, it may be too restrictive for the original data ingest, whose goal is just to capture as much data as possible and perform data formatting/cleansing later on, the approach often referred as schema on read. The latter approach recently won more ground with the tools to work with evolving schemas such as Avro and automatic schema discovery tools, but for the purpose of this chapter, I'll assume that we have done the schema discovery part and can start working with a DataFrames.

Let's first download and extract a flight delay dataset from the United States Department of Transportation, as follows:

```
$ wget http://www.transtats.bts.gov/Download/On_Time_On_Time_Performance_2015_7.zip
--2016-01-23 15:40:02--  http://www.transtats.bts.gov/Download/On_Time_On_Time_Performance_2015_7.zip
Resolving www.transtats.bts.gov... 204.68.194.70
Connecting to www.transtats.bts.gov|204.68.194.70|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 26204213 (25M) [application/x-zip-compressed]
```

Saving to: "On_Time_On_Time_Performance_2015_7.zip"

```
100% [=====]
=====
=====
=====] 26,204,213 966K/s in 27s
```

```
2016-01-23 15:40:29 (956 KB/s) - "On_Time_On_Time_Performance_2015_7.zip"
saved [26204213/26204213]
```

```
$ unzip -d flights On_Time_On_Time_Performance_2015_7.zip
Archive: On_Time_On_Time_Performance_2015_7.zip
  inflating: flights/On_Time_On_Time_Performance_2015_7.csv
  inflating: flights/readme.html
```

If you have Spark running on the cluster, you want to copy the file in HDFS:

```
$ hadoop fs -put flights .
```

The flights/readme.html files gives you detailed metadata information, as shown in the following image:

BACKGROUND

The data contained in the compressed file has been extracted from the On-Time Performance data table of the "On-Time" database from the TranStats data library. The time period is indicated in the name of the compressed file; for example, XXX_XXXXX_2001_1 contains data of the first month of the year 2001.

RECORD LAYOUT

Below are fields in the order that they appear on the records:

Year	Year
Quarter	Quarter (1-4)
Month	Month
DayofMonth	Day of Month
DayOfWeek	Day of Week
FlightDate	Flight Date (yyyymmdd)
UniqueCarrier	Unique Carrier Code. When the same code has been used by multiple carriers, a numeric suffix is used for earlier users, for example, PA, PA(1), PA(2). Use this field for analysis across a range of years.
AirlineID	An identification number assigned by US DOT to identify a unique airline (carrier). A unique airline (carrier) is defined as one holding and reporting under the same DOT certificate regardless of its Code, Name, or holding company/corporation.
Carrier	Code assigned by IATA and commonly used to identify a carrier. As the same code may have been assigned to different carriers over time, the code is not always unique. For analysis, use the Unique Carrier Code.
TailNum	Tail Number
FlightNum	Flight Number
OriginAirportID	Origin Airport, Airport ID. An identification number assigned by US DOT to identify a unique airport. Use this field for airport analysis across a range of years because an airport can change its airport code and airport codes can be reused.
OriginAirportSeqID	Origin Airport, Airport Sequence ID. An identification number assigned by US DOT to identify a unique airport at a given point of time. Airport attributes, such as airport name or coordinates, may change over time.
OriginCityMarketID	Origin Airport, City Market ID. City Market ID is an identification number assigned by US DOT to identify a city market. Use this field to consolidate airports serving the same city market.
Origin	Origin Airport
OriginCityName	Origin Airport, City Name
OriginState	Origin Airport, State Code
OriginStateFips	Origin Airport, State Fips
OriginStateName	Origin Airport, State Name
OriginWac	Origin Airport, World Area Code

Figure 08-1: Metadata provided with the On-Time Performance dataset released by the US Department of Transportation (for demo purposes only)

Now, I want you to analyze the delays of SFO returning flights and possibly find the factors contributing to the delay. Let's start with the R `data.frame`:

```
$ bin/sparkR --master local[8]

R version 3.2.3 (2015-12-10) -- "Wooden Christmas-Tree"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

Launching java with spark-submit command /Users/akozlov/spark-1.6.1-
bin-hadoop2.6/bin/spark-submit --master "local[8]" "sparkr-shell" /
var/folders/p1/y7ygx_4507q34vhd60q115p80000gn/T//RtmpD42eTz/backend_
port682e58e2c5db

Welcome to

   ___
  / _ \_  _ _ _ _ / / _ 
 _\ \ \_ \ \_ \_ \_ / _ \_ / 
/_ / . _ / \_ , _ / / _ / \_ \  version  1.6.1
 /_ /
```

Spark context is available as `sc`, SQL context is available as `sqlContext`

```
> flights <- read.table(unz("On_Time_On_Time_Performance_2015_7.zip",
  "On_Time_On_Time_Performance_2015_7.csv"), nrow=1000000, header=T,
  quote="", sep=",")  
> sfoFlights <- flights[flights$Dest == "SFO", ]  
> attach(sfoFlights)  
> delays <- aggregate(ArrDelayMinutes ~ DayOfWeek + Origin +
  UniqueCarrier, FUN=mean, na.rm=TRUE)  
> tail(delays[order(delays$ArrDelayMinutes), ])  
  DayOfWeek Origin UniqueCarrier ArrDelayMinutes  
220        4    ABQ          OO      67.60  
489        4    TUS          OO      71.80  
186        5    IAH          F9      77.60  
696        3    RNO          UA      79.50  
491        6    TUS          OO     168.25  
84         7    SLC          AS     203.25
```

If you were flying from Salt Lake City on Sunday with Alaska Airlines in July 2015, consider yourself unlucky (we have only done simple analysis so far, so one shouldn't attach too much significance to this result). There may be multiple other random factors contributing to the delay.

Even though we ran the example in SparkR, we still used the R `data.frame`. If we want to analyze data across multiple months, we will need to distribute the load across multiple nodes. This is where the SparkR distributed DataFrame comes into play, as it can be distributed across multiple threads even on a single node. There is a direct way to convert the R DataFrame to SparkR DataFrame (and thus to RDD):

```
> sparkDf <- createDataFrame(sqlContext, flights)
```

If I run it on a laptop, I will run out of memory. The overhead is large due to the fact that I need to transfer the data between multiple threads/nodes, we want to filter it as soon as possible:

```
sparkDf <- createDataFrame(sqlContext, subset(flights, select =
c("ArrDelayMinutes", "DayOfWeek", "Origin", "Dest", "UniqueCarrier")))
```

This will run even on my laptop. There is, of course, a reverse conversion from Spark's DataFrame to R's `data.frame`:

```
> rDf <- as.data.frame(sparkDf)
```

Alternatively, I can use the `spark-csv` package to read it from the `.csv` file, which, if the original `.csv` file is in a distributed filesystem such as HDFS, will avoid shuffling the data over network in a cluster setting. The only drawback, at least currently, is that Spark cannot read from the `.zip` files directly:

```
> $ ./bin/sparkR --packages com.databricks:spark-csv_2.10:1.3.0 --master local[8]

R version 3.2.3 (2015-12-10) -- "Wooden Christmas-Tree"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Warning: namespace 'SparkR' is not available and has been replaced
by .GlobalEnv when processing object 'sparkDf'
[Previously saved workspace restored]

Launching java with spark-submit command /home/alex/spark-1.6.1-bin-
hadoop2.6/bin/spark-submit --master "local[8]" --packages "com.
databricks:spark-csv_2.10:1.3.0" "sparkr-shell" /tmp/RtmpfhcUXX/backend_
port1b066bea5a03
Ivy Default Cache set to: /home/alex/.ivy2/cache
The jars for the packages stored in: /home/alex/.ivy2/jars
:: loading settings :: url = jar:file:/home/alex/spark-1.6.1-bin-
hadoop2.6/lib/spark-assembly-1.6.1-hadoop2.6.0.jar!/org/apache/ivy/core/
settings/ivysettings.xml
```

```
com.databricks#spark-csv_2.10 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent;1.0
  confs: [default]
    found com.databricks#spark-csv_2.10;1.3.0 in central
    found org.apache.commons#commons-csv;1.1 in central
    found com.univocity#univocity-parsers;1.5.1 in central
:: resolution report :: resolve 189ms :: artifacts dl 4ms
:: modules in use:
  com.databricks#spark-csv_2.10;1.3.0 from central in [default]
  com.univocity#univocity-parsers;1.5.1 from central in [default]
  org.apache.commons#commons-csv;1.1 from central in [default]
-----
|           |           modules           ||   artifacts   |
|   conf     |   number| search|dwnlded|evicted||   number|dwnlded|
-----
|   default  |   3   |   0   |   0   |   0   ||   3   |   0   |
-----
:: retrieving :: org.apache.spark#spark-submit-parent
confs: [default]
0 artifacts copied, 3 already retrieved (0kB/7ms)
```

Welcome to

```
   /__\
  /  \  /  _ \  /  \
 /  \  /  \  /  \  /  \
 /  \  /  \  /  \  /  \  version  1.6.1
 /  \
 /  \
```

```
Spark context is available as sc, SQL context is available as sqlContext
> sparkDf <- read.df(sqlContext, "./flights", "com.databricks.spark.csv",
header="true", inferSchema = "false")
> sfoFlights <- select(filter(sparkDf, sparkDf$Dest == "SFO"),
"DayOfWeek", "Origin", "UniqueCarrier", "ArrDelayMinutes")
> aggs <- agg(group_by(sfoFlights, "DayOfWeek", "Origin",
"UniqueCarrier"), count(sparkDf$ArrDelayMinutes),
avg(sparkDf$ArrDelayMinutes))
```

Note that we loaded the additional `com.databricks:spark-csv_2.10:1.3.0` package by supplying the `--package` flag on the command line; we can easily go distributed by using a Spark instance over a cluster of nodes or even analyze a larger dataset:

```
$ for i in $(seq 1 6); do wget http://www.transtats.bts.gov/Download/  
On_Time_On_Time_Performance_2015_$i.zip; unzip -d flights On_Time_On_  
Time_Performance_2015_$i.zip; hadoop fs -put -f flights/On_Time_On_Time_  
Performance_2015_$i.csv flights; done  
  
$ hadoop fs -ls flights  
Found 7 items  
  
-rw-r--r-- 3 alex eng 211633432 2016-02-16 03:28 flights/On_Time_On_  
Time_Performance_2015_1.csv  
  
-rw-r--r-- 3 alex eng 192791767 2016-02-16 03:28 flights/On_Time_On_  
Time_Performance_2015_2.csv  
  
-rw-r--r-- 3 alex eng 227016932 2016-02-16 03:28 flights/On_Time_On_  
Time_Performance_2015_3.csv
```

```
-rw-r--r-- 3 alex eng 218600030 2016-02-16 03:28 flights/On_Time_On_
Time_Performance_2015_4.csv
-rw-r--r-- 3 alex eng 224003544 2016-02-16 03:29 flights/On_Time_On_
Time_Performance_2015_5.csv
-rw-r--r-- 3 alex eng 227418780 2016-02-16 03:29 flights/On_Time_On_
Time_Performance_2015_6.csv
-rw-r--r-- 3 alex eng 235037955 2016-02-15 21:56 flights/On_Time_On_
Time_Performance_2015_7.csv
```

This will download and put the on-time performance data in the flight's directory (remember, as we discussed in *Chapter 1, Exploratory Data Analysis*, we would like to treat directories as big data datasets). We can now run the same analysis over the whole period of 2015 (for the available data):

```
> sparkDf <- read.df(sqlContext, "./flights", "com.databricks.spark.csv",
header="true")

> sfoFlights <- select(filter(sparkDf, sparkDf$Dest == "SFO"),
"DayOfWeek", "Origin", "UniqueCarrier", "ArrDelayMinutes")

> aggs <- cache(agg(group_by(sfoFlights, "DayOfWeek",
"Origin", "UniqueCarrier"), count(sparkDf$ArrDelayMinutes),
avg(sparkDf$ArrDelayMinutes)))

> head(arrange(aggs, c('avg(ArrDelayMinutes)'), decreasing = TRUE), 10)

  DayOfWeek Origin UniqueCarrier count(ArrDelayMinutes)
  avg(ArrDelayMinutes)
1           6    MSP          UA        1
122.00000
2           3    RNO          UA        8
79.50000
3           1    MSP          UA       13
68.53846
4           7    SAT          UA        1
65.00000
5           7    STL          UA        9
64.55556
6           1    ORD          F9       13
55.92308
7           1    MSO          OO        4
50.00000
8           2    MSO          OO        4
48.50000
9           5    CEC          OO       28
45.86957
10          3    STL          UA       13
43.46154
```

Note that we used a `cache()` call to pin the dataset to the memory as we will use it again later. This time it's Minneapolis/United on Saturday! However, you probably already know why: there is only one record for this combination of `DayOfWeek`, `Origin`, and `UniqueCarrier`; it's most likely an outlier. The average over about 30 flights for the previous outlier was reduced to 30 minutes:

```
> head(arrange(filter(filter(aggs, aggs$Origin == "SLC"),
  aggs$UniqueCarrier == "AS"), c('avg(ArrDelayMinutes)'), decreasing =
TRUE), 100)

  DayOfWeek Origin UniqueCarrier count(ArrDelayMinutes)
avg(ArrDelayMinutes)

  1       7     SLC        AS           30
32.600000

  2       2     SLC        AS           30
10.200000

  3       4     SLC        AS           31
9.774194

  4       1     SLC        AS           30
9.433333

  5       3     SLC        AS           30
5.866667

  6       5     SLC        AS           31
5.516129

  7       6     SLC        AS           30
2.133333
```

Sunday still remains a problem in terms of delays. The limit to the amount of data we can analyze now is only the number of cores on the laptop and nodes in the cluster. Let's look at more complex machine learning models now.

Linear models

Linear methods play an important role in statistical modeling. As the name suggests, linear model assumes that the dependent variable is a weighted combination of independent variables. In R, the `lm` function performs a linear regression and reports the coefficients, as follows:

```
R> attach(iris)
R> lm(Sepal.Length ~ Sepal.Width)
```

Call:

```
lm(formula = Sepal.Length ~ Sepal.Width)
```

Coefficients:

(Intercept)	Sepal.Width
6.5262	-0.2234

The summary function provides even more information:

```
R> model <- lm(Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width)
R> summary(model)
```

Call:

```
lm(formula = Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width)
```

Residuals:

Min	1Q	Median	3Q	Max
-0.82816	-0.21989	0.01875	0.19709	0.84570

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)		
(Intercept)	1.85600	0.25078	7.401	9.85e-12 ***		
Sepal.Width	0.65084	0.06665	9.765	< 2e-16 ***		
Petal.Length	0.70913	0.05672	12.502	< 2e-16 ***		
Petal.Width	-0.55648	0.12755	-4.363	2.41e-05 ***		

Signif. codes:	0 ****	0.001 ***	0.01 **	0.05 *	0.1 .	1

Residual standard error: 0.3145 on 146 degrees of freedom

Multiple R-squared: 0.8586, Adjusted R-squared: 0.8557

F-statistic: 295.5 on 3 and 146 DF, p-value: < 2.2e-16

While we considered generalized linear models in *Chapter 3, Working with Spark and MLLib*, and we will also consider the `glm` implementation in R and SparkR shortly, linear models provide more information in general and are an excellent tool for working with noisy data and selecting the relevant attribute for further analysis.

Data analysis life cycle

While most of the statistical books focus on the analysis and best use of available data, the results of statistical analysis in general should also affect the search for the new sources of information. In the complete data life cycle, discussed at the end of *Chapter 3, Working with Spark and MLlib*, a data scientist should always transform the latest variable importance results into the theories of how to collect data. For example, if the ink usage analysis for home printers points to an increase in ink usage for photos, one could potentially collect more information about the format of the pictures, sources of digital images, and paper the user prefers to use. This approach turned out to be very productive in a real business situation even though not fully automated.



Specifically, here is a short description of the output that linear models provide:

- **Residuals:** These are statistics for the difference between the actual and predicted values. A lot of techniques exist to detect the problems with the models on patterns of the residual distribution, but this is out of scope of this book. A detailed residual table can be obtained with the `resid(model)` function.
- **Coefficients:** These are the actual linear combination coefficients; the t-value represents the ratio of the value of the coefficient to the estimate of the standard error: higher values mean a higher likelihood that this coefficient has a non-trivial effect on the dependent variable. The coefficients can also be obtained with `coef(model)` functions.
- **Residual standard error:** This reports the standard mean square error, the metric that is the target of optimization in a straightforward linear regression.
- **Multiple R-squared:** This is the fraction of the dependent variable variance that is explained by the model. The adjusted value accounts for the number of parameters in your model and is considered to be a better metric to avoid overfitting if the number of observations does not justify the complexity of the models, which happens even for big data problems.
- **F-statistic:** The measure of model quality. In plain terms, it measures how all the parameters in the model explain the dependent variable. The p-value provides the probability that the model explains the dependent variable just due to random chance. The values under 0.05 (or 5%) are, in general, considered satisfactory. While in general, a high value probably means that the model is probably not statistically valid and "nothing else matters", the low F-statistic does not always mean that the model will work well in practice, so it cannot be directly applied as a model acceptance criterion.

Once the linear models are applied, usually more complex `glm` or recursive models, such as decision trees and the `rpart` function, are applied to find interesting variable interactions. Linear models are good for establishing baseline on the other models that can improve.

Finally, ANOVA is a standard technique to study the variance if the independent variables are discrete:

```
R> aov <- aov(Sepal.Length ~ Species)
R> summary(aov)

  Df Sum Sq Mean Sq F value Pr(>F)
Species      2   63.21   31.606   119.3 <2e-16 ***
Residuals  147   38.96    0.265
---
Signif. codes:  0 '****' 0.001 '***' 0.01 '**' 0.05 '*' 0.1 '.' 1
```

The measure of the model quality is F-statistics. While one can always run R algorithms with RDD using the pipe mechanism with `Rscript`, I will partially cover this functionality with respect to **Java Specification Request (JSR) 223** Python integration later. In this section, I would like to explore specifically a generalized linear regression `glm` function that is implemented both in R and SparkR natively.

Generalized linear model

Once again, you can run either R `glm` or SparkR `glm`. The list of possible link and optimization functions for R implementation is provided in the following table:

The following list shows possible options for R `glm` implementation:

Family	Variance	Link
gaussian	gaussian	identity
binomial	binomial	logit, probit or cloglog
poisson	poisson	log, identity or sqrt
Gamma	Gamma	inverse, identity or log
inverse.gaussian	inverse.gaussian	1/mu^2
quasi	user-defined	user-defined

I will use a binary target, `ArrDel15`, which indicates whether the plane was more than 15 minutes late for the arrival. The independent variables will be `DepDel15`, `DayOfWeek`, `Month`, `UniqueCarrier`, `Origin`, and `Dest`:

```
R> flights <- read.table(unz("On_Time_On_Time_Performance_2015_7.zip",
  "On_Time_On_Time_Performance_2015_7.csv"), nrow=1000000, header=T,
  quote="", sep=",")
R> flights$DoW_ <- factor(flights$DayOfWeek, levels=c(1,2,3,4,5,6,7), lab-
  ls=c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"))
R> attach(flights)
R> system.time(model <- glm(ArrDel15 ~ UniqueCarrier + DoW_ + Origin +
  Dest, flights, family="binomial"))
```

While you wait for the results, open another shell and run `glm` in the SparkR mode on the full seven months of data:

```
sparkR> cache(sparkDf <- read.df(sqlContext, "./flights", "com.
  databricks.spark.csv", header="true", inferSchema="true"))
DataFrame[Year:int, Quarter:int, Month:int, DayofMonth:int,
  DayOfWeek:int, FlightDate:string, UniqueCarrier:string, AirlineID:int,
  Carrier:string, TailNum:string, FlightNum:int, OriginAirportID:int,
  OriginAirportSeqID:int, OriginCityMarketID:int, Origin:string,
  OriginCityName:string, OriginState:string, OriginStateFips:int,
  OriginStateName:string, OriginWac:int, DestAirportID:int,
  DestAirportSeqID:int, DestCityMarketID:int, Dest:string,
  DestCityName:string, DestState:string, DestStateFips:int,
  DestStateName:string, DestWac:int, CRSDepTime:int, DepTime:int,
  DepDelay:double, DepDelayMinutes:double, DepDel15:double,
  DepartureDelayGroups:int, DepTimeBlk:string, TaxiOut:double,
  WheelsOff:int, WheelsOn:int, TaxiIn:double, CRSArrTime:int,
  ArrTime:int, ArrDelay:double, ArrDelayMinutes:double, ArrDel15:double,
  ArrivalDelayGroups:int, ArrTimeBlk:string, Cancelled:double,
  CancellationCode:string, Diverted:double, CRSElapsedTime:double,
  ActualElapsedTime:double, AirTime:double, Flights:double,
  Distance:double, DistanceGroup:int, CarrierDelay:double,
  WeatherDelay:double, NASDelay:double, SecurityDelay:double,
  LateAircraftDelay:double, FirstDepTime:int, TotalAddGTime:double,
  LongestAddGTime:double, DivAirportLandings:int, DivReachedDest:double,
  DivActualElapsedTime:double, DivArrDelay:double, DivDistance:double,
  Div1Airport:string, Div1AirportID:int, Div1AirportSeqID:int,
  Div1WheelsOn:int, Div1TotalGTime:double, Div1LongestGTime:double,
  Div1WheelsOff:int, Div1TailNum:string, Div2Airport:string,
  Div2AirportID:int, Div2AirportSeqID:int, Div2WheelsOn:int,
  Div2TotalGTime:double, Div2LongestGTime:double, Div2WheelsOff:string,
  Div2TailNum:string, Div3Airport:string, Div3AirportID:string,
  Div3AirportSeqID:string, Div3WheelsOn:string, Div3TotalGTime:string,
  Div3LongestGTime:string, Div3WheelsOff:string, Div3TailNum:string,
  Div4Airport:string, Div4AirportID:string, Div4AirportSeqID:string,
```

```
Div4WheelsOn:string, Div4TotalGTime:string, Div4LongestGTime:string,
Div4WheelsOff:string, Div4TailNum:string, Div5Airport:string,
Div5AirportID:string, Div5AirportSeqID:string, Div5WheelsOn:string,
Div5TotalGTime:string, Div5LongestGTime:string, Div5WheelsOff:string,
Div5TailNum:string, :string]

sparkR> noNulls <- cache(dropna(selectExpr(filter(sparkDf,
sparkDf$Cancelled == 0), "ArrDel15", "UniqueCarrier", "format_
string('%d', DayOfWeek) as DayOfWeek", "Origin", "Dest"), "any"))

sparkR> sparkModel = glm(ArrDel15 ~ UniqueCarrier + DayOfWeek + Origin +
Dest, noNulls, family="binomial")
```

Here we try to build a model explaining delays as an effect of carrier, day of week, and origin on destination airports, which is captured by the formula construct `ArrDel15 ~ UniqueCarrier + DayOfWeek + Origin + Dest.`

Nulls, big data, and Scala

Note that in the SparkR case of `glm`, I had to explicitly filter out the non-cancelled flights and removed the NA – or nulls in the C/Java lingo. While R does this for you by default, NAs in big data are very common as the datasets are typically sparse and shouldn't be treated lightly. The fact that we have to deal with nulls explicitly in MLlib warns us about some additional information in the dataset and is definitely a welcome feature. The presence of an NA can carry information about the way the data was collected. Ideally, each NA should be accompanied by a small `get_na_info` method as to why this particular value was not available or collected, which leads us to the `Either` type in Scala.

Even though nulls are inherited from Java and a part of Scala, the `Option` and `Either` types are new and more robust mechanism to deal with special cases where nulls were traditionally used. Specifically, `Either` can provide a value or exception message as to why it was not computed; while `Option` can either provide a value or be `None`, which can be readily captured by the Scala pattern-matching framework.



One thing you will notice is that SparkR will run multiple threads, and even on a single node, it will consume CPU time from multiple cores and returns much faster even with a larger size of data. In my experiment on a 32-core machine, it was able to finish in under a minute (as opposed to 35 minutes for R `glm`). To get the results, as in the R model case, we need to run the `summary()` method:

```
> summary(sparkModel)
$coefficients
      Estimate
(Intercept) -1.518542340
```

```
UniqueCarrier_WN  0.382722232
UniqueCarrier_DL -0.047997652
UniqueCarrier_OO  0.367031995
UniqueCarrier_AA  0.046737727
UniqueCarrier_EV  0.344539788
UniqueCarrier_UA  0.299290120
UniqueCarrier_US  0.069837542
UniqueCarrier_MQ  0.467597761
UniqueCarrier_B6  0.326240578
UniqueCarrier_AS  -0.210762769
UniqueCarrier_NK  0.841185903
UniqueCarrier_F9  0.788720078
UniqueCarrier_HA  -0.094638586
DayOfWeek_5       0.232234937
DayOfWeek_4       0.274016179
DayOfWeek_3       0.147645473
DayOfWeek_1       0.347349366
DayOfWeek_2       0.190157420
DayOfWeek_7       0.199774806
Origin_ATL        -0.180512251
...

```

The worst performer is NK (Spirit Airlines). Internally, SparkR uses limited-memory BFGS, which is a limited-memory quasi-Newton optimization method that is similar to the results obtained with R `glm` on the July data:

```
R> summary(model)

Call:
glm(formula = ArrDel15 ~ UniqueCarrier + DoW + Origin + Dest,
     family = "binomial", data = dow)

Deviance Residuals:
    Min      1Q      Median      3Q      Max 
-1.4205 -0.7274 -0.6132 -0.4510  2.9414 

Coefficients:
```

```

      Estimate Std. Error z value Pr(>|z|)

(Intercept) -1.817e+00 2.402e-01 -7.563 3.95e-14 ***
UniqueCarrierAS -3.296e-01 3.413e-02 -9.658 < 2e-16 ***
UniqueCarrierB6 3.932e-01 2.358e-02 16.676 < 2e-16 ***
UniqueCarrierDL -6.602e-02 1.850e-02 -3.568 0.000359 ***
UniqueCarrierEV 3.174e-01 2.155e-02 14.728 < 2e-16 ***
UniqueCarrierF9 6.754e-01 2.979e-02 22.668 < 2e-16 ***
UniqueCarrierHA 7.883e-02 7.058e-02 1.117 0.264066
UniqueCarrierMQ 2.175e-01 2.393e-02 9.090 < 2e-16 ***
UniqueCarrierNK 7.928e-01 2.702e-02 29.343 < 2e-16 ***
UniqueCarrierOO 4.001e-01 2.019e-02 19.817 < 2e-16 ***
UniqueCarrierUA 3.982e-01 1.827e-02 21.795 < 2e-16 ***
UniqueCarrierVX 9.723e-02 3.690e-02 2.635 0.008423 **
UniqueCarrierWN 6.358e-01 1.700e-02 37.406 < 2e-16 ***
dowTue 1.365e-01 1.313e-02 10.395 < 2e-16 ***
dowWed 1.724e-01 1.242e-02 13.877 < 2e-16 ***
dowThu 4.593e-02 1.256e-02 3.656 0.000256 ***
dowFri -2.338e-01 1.311e-02 -17.837 < 2e-16 ***
dowSat -2.413e-01 1.458e-02 -16.556 < 2e-16 ***
dowSun -3.028e-01 1.408e-02 -21.511 < 2e-16 ***
OriginABI -3.355e-01 2.554e-01 -1.314 0.188965
...

```

Other parameters of SparkR `glm` implementation are provided in the following table:

The following table shows a list of parameters for SparkR `glm` implementation:

Parameter	Possible Values	Comments
<code>formula</code>	A symbolic description like in R	Currently only a subset of formula operators are supported: ' <code>~</code> ', ' <code>.</code> ', ' <code>:</code> ', ' <code>+</code> ', and ' <code>-</code> '
<code>family</code>	gaussian or binomial	Needs to be in quotes: gaussian -> linear regression, binomial -> logistic regression
<code>data</code>	DataFrame	Needs to be SparkR DataFrame, not <code>data.frame</code>
<code>lambda</code>	positive	Regularization coefficient
<code>alpha</code>	positive	Elastic-net mixing parameter (refer to <code>glmnet</code> 's documentation for details)
<code>standardize</code>	TRUE or FALSE	User-defined

Parameter	Possible Values	Comments
solver	l-bfgs, normal or auto	auto will choose the algorithm automatically, l-bfgs means limited-memory BFGS, normal means using normal equation as an analytical solution to the linear regression problem

Reading JSON files in SparkR

Schema on Read is one of the convenient features of big data. The DataFrame class has the ability to figure out the schema of a text file containing a JSON record per line:

```
[akozlov@Alexanders-MacBook-Pro spark-1.6.1-bin-hadoop2.6]$ cat examples/src/main/resources/people.json
{"name": "Michael"}
{"name": "Andy", "age": 30}
{"name": "Justin", "age": 19}

[akozlov@Alexanders-MacBook-Pro spark-1.6.1-bin-hadoop2.6]$ bin/sparkR
...

> people = read.json(sqlContext, "examples/src/main/resources/people.json")
> dypes(people)
[[1]]
[1] "age"      "bigint"

[[2]]
[1] "name"     "string"

> schema(people)
StructType
|-name = "age", type = "LongType", nullable = TRUE
|-name = "name", type = "StringType", nullable = TRUE
> showDF(people)
+---+---+
| age | name |
+---+---+
```

```
| null|Michael|
| 30 | Andy|
| 19 | Justin|
+---+-----+
```

Writing Parquet files in SparkR

As we mentioned in the previous chapter, the Parquet format is an efficient storage format, particularly for low cardinality columns. Parquet files can be read/written directly from R:

```
> write.parquet(sparkDf, "parquet")
```

You can see that the new Parquet file is 66 times smaller than the original zip file downloaded from the DoT:

```
[akozlov@Alexanders-MacBook-Pro spark-1.6.1-bin-hadoop2.6]$ ls -l On_
Time_On_Time_Performance_2015_7.zip parquet/ flights/
-rw-r--r-- 1 akozlov staff 26204213 Sep 9 12:21 /Users/akozlov/spark/
On_Time_On_Time_Performance_2015_7.zip

flights/:
total 459088
-rw-r--r-- 1 akozlov staff 235037955 Sep 9 12:20 On_Time_On_Time_
Performance_2015_7.csv
-rw-r--r-- 1 akozlov staff 12054 Sep 9 12:20 readme.html

parquet/:
total 848
-rw-r--r-- 1 akozlov staff 0 Jan 24 22:50 _SUCCESS
-rw-r--r-- 1 akozlov staff 10000 Jan 24 22:50 _common_metadata
-rw-r--r-- 1 akozlov staff 23498 Jan 24 22:50 _metadata
-rw-r--r-- 1 akozlov staff 394418 Jan 24 22:50 part-r-00000-9e2d0004-
c71f-4bf5-aafe-90822f9d7223.gz.parquet
```

Invoking Scala from R

Let's assume that one has an exceptional implementation of a numeric method in Scala that we want to call from R. One way of doing this would be to use the `R system()` function that invokes `/bin/sh` on Unix-like systems. However, the `rscala` package is a more efficient way that starts a Scala interpreter and maintains communication over TCP/IP network connection.

Here, the Scala interpreter maintains the state (memoization) between the calls. Similarly, one can define functions, as follows:

```
R> scala <- scalaInterpreter()
R> scala %~% 'def pri(i: Stream[Int]): Stream[Int] = i.head #:: pri(i.
tail filter { x => { println("Evaluating " + x + "%" + i.head); x %
i.head != 0 } } )'
ScalaInterpreterReference... engine: javax.script.ScriptEngine
R> scala %~% 'val primes = pri(Stream.from(2))'
ScalaInterpreterReference... primes: Stream[Int]
R> scala %~% 'primes take 5 foreach println'
2
Evaluating 3%2
3
Evaluating 4%2
Evaluating 5%2
Evaluating 5%3
5
Evaluating 6%2
Evaluating 7%2
Evaluating 7%3
Evaluating 7%5
7
Evaluating 8%2
Evaluating 9%2
Evaluating 9%3
Evaluating 10%2
Evaluating 11%2
Evaluating 11%3
Evaluating 11%5
Evaluating 11%7
```

```
11
R> scala %~% 'primes take 5 foreach println'
2
3
5
7
11
R> scala %~% 'primes take 7 foreach println'
2
3
5
7
11
Evaluating 12%2
Evaluating 13%2
Evaluating 13%3
Evaluating 13%5
Evaluating 13%7
Evaluating 13%11
13
Evaluating 14%2
Evaluating 15%2
Evaluating 15%3
Evaluating 16%2
Evaluating 17%2
Evaluating 17%3
Evaluating 17%5
Evaluating 17%7
Evaluating 17%11
Evaluating 17%13
17
R>
```

R from Scala can be invoked using the ! or !! Scala operators and Rscript command:

```
[akozlov@Alexanders-MacBook-Pro ~]$ cat << EOF > rdate.R
> #!/usr/local/bin/Rscript
>
> write(date(), stdout())
> EOF
[akozlov@Alexanders-MacBook-Pro ~]$ chmod a+x rdate.R
[akozlov@Alexanders-MacBook-Pro ~]$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import sys.process._
import sys.process._

scala> val date = Process(Seq("./rdate.R")).!!
date: String =
"Wed Feb 24 02:20:09 2016
"
```

Using Rserve

A more efficient way is to use the similar TCP/IP binary transport protocol to communicate with R with Rsclient/Rserve (<http://www.rforge.net/Rserve>). To start Rserve on a node that has R installed, perform the following action:

```
[akozlov@Alexanders-MacBook-Pro ~]$ wget http://www.rforge.net/Rserve/
snapshot/Rserve_1.8-5.tar.gz

[akozlov@Alexanders-MacBook-Pro ~]$ R CMD INSTALL Rserve_1.8-5.tar.gz
...
[akozlov@Alexanders-MacBook-Pro ~]$ R CMD INSTALL Rserve_1.8-5.tar.gz

[akozlov@Alexanders-MacBook-Pro ~]$ $ R -q CMD Rserve

R version 3.2.3 (2015-12-10) -- "Wooden Christmas-Tree"
```

```
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
```

```
Natural language support but running in an English locale
```

```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
```

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

```
Rserv started in daemon mode.
```

By default, Rserv opens a connection on localhost:6311. The advantage of the binary network protocol is that it is platform-independent and multiple clients can communicate with the server. The clients can connect to Rserve.

Note that, while passing the results as a binary object has its advantages, you have to be careful with the type mappings between R and Scala. Rserve supports other clients, including Python, but I will also cover JSR 223-compliant scripting at the end of this chapter.

Integrating with Python

Python has slowly established ground as a de-facto tool for data science. It has a command-line interface and decent visualization via matplotlib and ggplot, which is based on R's ggplot2. Recently, Wes McKinney, the creator of Pandas, the time series data-analysis package, has joined Cloudera to pave way for Python in big data.

Setting up Python

Python is usually part of the default installation. Spark requires version 2.7.0+.

If you don't have Python on Mac OS, I recommend installing the Homebrew package manager from <http://brew.sh>:

```
[akozlov@Alexanders-MacBook-Pro spark(master)]$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
==> This script will install:
/usr/local/bin/brew
/usr/local/Library/...
/usr/local/share/man/man1/brew.1
...
[akozlov@Alexanders-MacBook-Pro spark(master)]$ brew install python
...
```

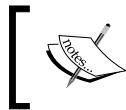
Otherwise, on a Unix-like system, Python can be compiled from the source distribution:

```
$ export PYTHON_VERSION=2.7.11
$ wget -O - https://www.python.org/ftp/python/$PYTHON_VERSION/Python-$PYTHON_VERSION.tgz | tar xzvf -
$ cd $HOME/Python-$PYTHON_VERSION
$ ./configure --prefix=/usr/local --enable-unicode=ucs4 --enable-shared
LDFLAGS="-Wl,-rpath /usr/local/lib"
$ make; sudo make altinstall
$ sudo ln -sf /usr/local/bin/python2.7 /usr/local/bin/python
```

It is good practice to place it in a directory different from the default Python installation. It is normal to have multiple versions of Python on a single system, which usually does not lead to problems as Python separates the installation directories. For the purpose of this chapter, as for many machine learning takes, I'll also need a few packages. The packages and specific versions may differ across installations:

```
$ wget https://bootstrap.pypa.io/ez_setup.py
$ sudo /usr/local/bin/python ez_setup.py
$ sudo /usr/local/bin/easy_install-2.7 pip
$ sudo /usr/local/bin/pip install --upgrade avro nose numpy scipy pandas
statsmodels scikit-learn iso8601 python-dateutil python-snappy
```

If everything compiles—SciPy uses a Fortran compiler and libraries for linear algebra—we are ready to use Python 2.7.11!



Note that if one wants to use Python with the pipe command in a distributed environment, Python needs to be installed on every node in the network.



PySpark

As `bin/sparkR` launches R with preloaded Spark context, `bin/pyspark` launches Python shell with preloaded Spark context and Spark driver running. The `PYSPARK_PYTHON` environment variable can be used to point to a specific Python version:

```
[akozlov@Alexanders-MacBook-Pro spark-1.6.1-bin-hadoop2.6]$ export  
PYSPARK_PYTHON=/usr/local/bin/python  
  
[akozlov@Alexanders-MacBook-Pro spark-1.6.1-bin-hadoop2.6]$ bin/pyspark  
Python 2.7.11 (default, Jan 23 2016, 20:14:24)  
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
Welcome to
```

```
    /__/_\ _ \ / _ / _ / _ /  
    \ \ / _ \ / _ ^ / _ / _ /  
    / _ / . _ / \ _ , _ / / _ / \ _ \   version 1.6.1  
    / _ /
```

```
Using Python version 2.7.11 (default, Jan 23 2016 20:14:24)  
SparkContext available as sc, HiveContext available as sqlContext.  
>>>
```

PySpark directly supports most of MLlib functionality on Spark RDDs (<http://spark.apache.org/docs/latest/api/python>), but it is known to lag a few releases behind the Scala API (<http://spark.apache.org/docs/latest/api/python>). As of the 1.6.0+ release, it also supports DataFrames (<http://spark.apache.org/docs/latest/sql-programming-guide.html>):

```
>>> sfoFlights = sqlContext.sql("SELECT Dest, UniqueCarrier,  
ArrDelayMinutes FROM parquet.parquet")  
>>> sfoFlights.groupBy(["Dest", "UniqueCarrier"]).agg(func.  
avg("ArrDelayMinutes"), func.count("ArrDelayMinutes")).  
sort("avg(ArrDelayMinutes)", ascending=False).head(5)
```

```
[Row(Dest=u'HNL', UniqueCarrier=u'HA', avg(ArrDelayMinutes)=53.70967741935484, count(ArrDelayMinutes)=31), Row(Dest=u'IAH', UniqueCarrier=u'F9', avg(ArrDelayMinutes)=43.064516129032256, count(ArrDelayMinutes)=31), Row(Dest=u'LAX', UniqueCarrier=u'DL', avg(ArrDelayMinutes)=39.68691588785047, count(ArrDelayMinutes)=214), Row(Dest=u'LAX', UniqueCarrier=u'WN', avg(ArrDelayMinutes)=29.704453441295545, count(ArrDelayMinutes)=247), Row(Dest=u'MSO', UniqueCarrier=u'OO', avg(ArrDelayMinutes)=29.551724137931036, count(ArrDelayMinutes)=29)]
```

Calling Python from Java/Scala

As this is really a book about Scala, we should also mention that one can call Python code and its interpreter directly from Scala (or Java). There are a few options available that will be discussed in this chapter.

Using `sys.process._`

Scala, as well as Java, can call OS processes via spawning a separate thread, which we already used for interactive analysis in *Chapter 1, Exploratory Data Analysis*: the `.!` method will start the process and return the exit code, while `.!!` will return the string that contains the output:

```
scala> import sys.process._
import sys.process._

scala> val retCode = Process(Seq("/usr/local/bin/python", "-c", "import
socket; print(socket.gethostname())")).!
Alexanders-MacBook-Pro.local
retCode: Int = 0

scala> val lines = Process(Seq("/usr/local/bin/python", "-c", """from
datetime import datetime, timedelta; print("Yesterday was {}".
format(datetime.now()-timedelta(days=1)))""")).!!
lines: String =
"Yesterday was 2016-02-12 16:24:53.161853
"
```

Let's try a more complex SVD computation (similar to the one we used in SVD++ recommendation engine, but this time, it invokes BLAS C-libraries at the backend). I created a Python executable that takes a string representing a matrix and the required rank as an input and outputs an SVD approximation with the provided rank:

```
#!/usr/bin/env python

import sys
import os
import re

import numpy as np
from scipy import linalg
from scipy.linalg import svd

np.set_printoptions(linewidth=10000)

def process_line(input):
    inp = input.rstrip("\r\n")
    if len(inp) > 1:
        try:
            (mat, rank) = inp.split("|")
            a = np.matrix(mat)
            r = int(rank)
        except:
            a = np.matrix(inp)
            r = 1
        U, s, Vh = linalg.svd(a, full_matrices=False)
        for i in xrange(r, s.size):
            s[i] = 0
        S = linalg.diagsvd(s, s.size, s.size)
        print(str(np.dot(U, np.dot(S, Vh))).replace(os.linesep, ";"))

if __name__ == '__main__':
    map(process_line, sys.stdin)
```

Let's call it `svd.py` and put in in the current directory. Given a matrix and rank as an input, it produces an approximation of a given rank:

```
$ echo -e "1,2,3;2,1,2;3,2,1;7,8,9|3" | ./svd.py
[[ 1.  2.  3.]; [ 2.  1.  2.]; [ 3.  2.  1.]; [ 7.  8.  9.]]
```

To call it from Scala, let's define the following #<<< method in our DSL:

```
scala> implicit class RunCommand(command: String) {
|   def #<<< (input: String)(implicit buffer: StringBuilder) = {
|     val process = Process(command)
|     val io = new ProcessIO (
|       in => { in.write(input.getBytes "UTF-8"); in.close},
|       out => { buffer.append scala.io.Source.fromInputStream(out).
|         getLines.mkString("\n"); buffer.append("\n"); out.close() },
|       err => { scala.io.Source.fromInputStream(err).getLines().
|         foreach(System.err.println) })
|     (process run io).exitValue
|   }
| }
defined class RunCommand
```

Now, we can use the #<<< operator to call Python's SVD method:

```
scala> implicit val buffer = new StringBuilder()
buffer: StringBuilder =

scala> if ("./svd.py" #<<< "1,2,3;2,1,2;3,2,1;7,8,9|1" == 0)
Some(buffer.toString) else None
res77: Option[String] = Some([[ 1.84716691  2.02576751  2.29557674];
[ 1.48971176  1.63375041  1.85134741]; [ 1.71759947  1.88367234
2.13455611]; [ 7.19431647  7.88992728  8.94077601]])
```

Note that as we requested the resulting matrix rank to be one, all rows and columns are linearly dependent. We can even pass several lines of input at a time, as follows:

```
scala> if ("./svd.py" #<<< """
| 1,2,3;2,1,2;3,2,1;7,8,9|0
| 1,2,3;2,1,2;3,2,1;7,8,9|1
| 1,2,3;2,1,2;3,2,1;7,8,9|2
| 1,2,3;2,1,2;3,2,1;7,8,9|3"""" == 0) Some(buffer.toString) else None
res80: Option[String] =
Some([[ 0.  0.  0.]; [ 0.  0.  0.]; [ 0.  0.  0.]; [ 0.  0.  0.]])
[[ 1.84716691  2.02576751  2.29557674]; [ 1.48971176  1.63375041
1.85134741]; [ 1.71759947  1.88367234  2.13455611]; [ 7.19431647
7.88992728  8.94077601]])
```

```
[[ 0.9905897  2.02161614  2.98849663]; [ 1.72361156  1.63488399  
1.66213642]; [ 3.04783513  1.89011928  1.05847477]; [ 7.04822694  
7.88921926  9.05895373]]  
[[ 1.  2.  3.]; [ 2.  1.  2.]; [ 3.  2.  1.]; [ 7.  8.  9.]])
```

Spark pipe

SVD decomposition is usually a pretty heavy operation, so the relative overhead of calling Python in this case is small. We can avoid this overhead if we keep the process running and supply several lines at a time, like we did in the last example. Both Hadoop MR and Spark implement this approach. For example, in Spark, the whole computation will take only one line, as shown in the following:

```
scala> sc.parallelize(List("1,2,3;2,1,2;3,2,1;7,8,9|0",  
"1,2,3;2,1,2;3,2,1;7,8,9|1", "1,2,3;2,1,2;3,2,1;7,8,9|2",  
"1,2,3;2,1,2;3,2,1;7,8,9|3"),4).pipe("./svd.py").collect.foreach(println)  
[[ 0.  0.  0.]; [ 0.  0.  0.]; [ 0.  0.  0.]; [ 0.  0.  0.]]  
[[ 1.84716691  2.02576751  2.29557674]; [ 1.48971176  1.63375041  
1.85134741]; [ 1.71759947  1.88367234  2.13455611]; [ 7.19431647  
7.88992728  8.94077601]]  
[[ 0.9905897  2.02161614  2.98849663]; [ 1.72361156  1.63488399  
1.66213642]; [ 3.04783513  1.89011928  1.05847477]; [ 7.04822694  
7.88921926  9.05895373]]  
[[ 1.  2.  3.]; [ 2.  1.  2.]; [ 3.  2.  1.]; [ 7.  8.  9.]])
```

The whole pipeline is ready to be distributed across a cluster of multicore workstations! I think you will be in love with Scala/Spark already.

Note that debugging the pipelined executions might be tricky as the data is passed from one process to another using OS pipes.

Jython and JSR 223

For completeness, we need to mention Jython, a Java implementation of Python (as opposed to a more familiar C implementation, also called CPython). Jython avoids the problem of passing input/output via OS pipelines by allowing the users to compile Python source code to Java byte codes, and running the resulting bytecodes on any Java virtual machine. As Scala also runs in Java virtual machine, it can use the Jython classes directly, although the reverse is not true in general; Scala classes sometimes are not compatible to be used by Java/Jython.

JSR 223

 In this particular case, the request is for "Scripting for the JavaTM Platform" and was originally filed on Nov 15th 2004 (<https://www.jcp.org/en/jsr/detail?id=223>). At the beginning, it was targeted towards the ability of the Java servlet to work with multiple scripting languages. The specification requires the scripting language maintainers to provide a Java JAR with corresponding implementations. Portability issues hindered practical implementations, particularly when platforms require complex interaction with OS, such as dynamic linking in C or Fortran. Currently, only a handful languages are supported, with R and Python being supported, but in incomplete form.

Since Java 6, JSR 223: Scripting for Java added the `javax.script` package that allows multiple scripting languages to be called through the same API as long as the language provides a script engine. To add the Jython scripting language, download the latest Jython JAR from the Jython site at <http://www.jython.org/downloads.html>:

```
$ wget -O jython-standalone-2.7.0.jar http://search.maven.org/
remotecontent?filepath=org/python/jython-standalone/2.7.0/jython-
standalone-2.7.0.jar
```

```
[akozlov@Alexanders-MacBook-Pro Scala]$ scala -cp jython-standalone-
2.7.0.jar
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import javax.script.ScriptEngine;
...
scala> import javax.script.ScriptEngineManager;
...
scala> import javax.script.ScriptException;
...
scala> val manager = new ScriptEngineManager();
manager: javax.script.ScriptEngineManager = javax.script.
ScriptEngineManager@3a03464

scala> val engines = manager.getEngineFactories();
```

```
engines: java.util.List[javax.script.ScriptEngineFactory] = [org.python.jsr223.PyScriptEngineFactory@4909b8da, jdk.nashorn.api.scripting.NashornScriptEngineFactory@68837a77, scala.tools.nsc.interpreter.IMain$Factory@1324409e]
```

Now, I can use the Jython/Python scripting engine:

```
scala> val engine = new ScriptEngineManager().getEngineByName("jython");
engine: javax.script.ScriptEngine = org.python.jsr223.PyScriptEngine@6094de13

scala> engine.eval("from datetime import datetime, timedelta; yesterday =
str(datetime.now() -timedelta(days=1))")
res15: Object = null

scala> engine.get("yesterday")
res16: Object = 2016-02-12 23:26:38.012000
```

It is worth giving a disclaimer here that not all Python modules are available in Jython. Modules that require a C/Fortran dynamic linkage for the library that doesn't exist in Java are not likely to work in Jython. Specifically, NumPy and SciPy are not supported in Jython as they rely on C/Fortran. If you discover some other missing modules, you can try copying the .py file from a Python distribution to a sys.path Jython directory—if this works, consider yourself lucky.

Jython has the advantage of accessing Python-rich modules without the necessity of starting the Python runtime on each call, which might result in a significant performance saving:

```
scala> val startTime = System.nanoTime
startTime: Long = 54384084381087

scala> for (i <- 1 to 100) {
|   engine.eval("from datetime import datetime, timedelta; yesterday =
str(datetime.now() -timedelta(days=1))")
|   val yesterday = engine.get("yesterday")
| }
```



```
scala> val elapsed = 1e-9 * (System.nanoTime - startTime)
elapsed: Double = 0.270837934

scala> val startTime = System.nanoTime
```

```
startTime: Long = 54391560460133

scala> for (i <- 1 to 100) {
|   val yesterday = Process(Seq("/usr/local/bin/python", "-c",
"""from datetime import datetime, timedelta; print(datetime.now() -
timedelta(days=1))""").!!
| }

scala> val elapsed = 1e-9 * (System.nanoTime - startTime)
elapsed: Double = 2.221937263

Jython JSR 223 call is 10 times faster!
```

Summary

R and Python are like bread and butter for a data scientist. Modern frameworks tend to be interoperable and borrow from each other's strength. In this chapter, I went over the plumbing of interoperability with R and Python. Both of them have packages (R) and modules (Python) that became very popular and extend the current Scala/Spark functionality. Many consider R and Python existing libraries to be crucial for their implementations.

This chapter demonstrated a few ways to integrate these packages and provide the tradeoffs of using these integrations so that we can proceed on to the next chapter, looking at the NLP, where functional programming has been traditionally used from the start.

9

NLP in Scala

This chapter describes a few common techniques of **Natural Language Processing (NLP)**, specifically, the ones that can benefit from Scala. There are some NLP packages in the open source out there. The most famous of them is probably NLTK (<http://www.nltk.org>), which is written in Python, and ostensibly even a larger number of proprietary software solutions emphasizing different aspects of NLP. It is worth mentioning Wolf (<https://github.com/wolfe-pack>), FACTORIE (<http://factorie.cs.umass.edu>), and ScalaNLP (<http://www.scalanlp.org>), and skymind (<http://www.skymind.io>), which is partly proprietary. However, few open source projects in this area remain active for a long period of time for one or another reason. Most projects are being eclipsed by Spark and MLlib capabilities, particularly, in the scalability aspect.

Instead of giving a detailed description of each of the NLP projects, which also might include speech-to-text, text-to-speech, and language translators, I will provide a few basic techniques focused on leveraging Spark MLlib in this chapter. The chapter comes very naturally as the last analytics chapter in this book. Scala is a very natural-language looking computer language and this chapter will leverage the techniques I developed earlier.

NLP arguably is the core of AI. Originally, the AI was created to mimic the humans, and natural language parsing and understanding is an indispensable part of it. Big data techniques has started to penetrate NLP, even though traditionally, NLP is very computationally intensive and is regarded as a small data problem. NLP often requires extensive deep learning techniques, and the volume of data of all written texts appears to be not so large compared to the logs volumes generated by all the machines today and analyzed by the big data machinery.

Even though the Library of Congress counts millions of documents, most of them can be digitized in PBs of actual digital data, a volume that any social websites is able to collect, store, and analyze within a few seconds. Complete works of most prolific authors can be stored within a few MBs of files (refer to *Table 09-1*). Nonetheless, the social network and ADTECH companies parse text from millions of users and in hundreds of contexts every day.

The complete works of	When lived	Size
Plato	428/427 (or 424/423) - 348/347 BC	2.1 MB
William Shakespeare	26 April 1564 (baptized) - 23 April 1616	3.8 MB
Fyodor Dostoevsky	11 November 1821 - 9 February 1881	5.9 MB
Leo Tolstoy	9 September 1828 - 20 November 1910	6.9 MB
Mark Twain	November 30, 1835 - April 21, 1910	13 MB

Table 09-1. Complete Works collections of some famous writers (most can be acquired on Amazon.com today for a few dollars, later authors, although readily digitized, are more expensive)

The natural language is a dynamic concept that changes over time, technology, and generations. We saw the appearance of emoticons, three-letter abbreviations, and so on. Foreign languages tend to borrow from each other; describing this dynamic ecosystem is a challenge on itself.

As in the previous chapters, I will focus on how to use Scala as a tool to orchestrate the language analysis rather than rewriting the tools in Scala. As the topic is so large, I will not claim to cover all aspects of NLP here.

In this chapter, we will cover the following topics:

- Discussing NLP with the example of text processing pipeline and stages
- Learning techniques for simple text analysis in terms of bags
- Learning about **Term Frequency Inverse Document Frequency (TF-IDF)** technique that goes beyond simple bag analysis and de facto the standard in **Information Retrieval (IR)**
- Learning about document clustering with the example of the **Latent Dirichlet Allocation (LDA)** approach
- Performing semantic analysis using word2vec n-gram-based algorithms

Text analysis pipeline

Before we proceed to detailed algorithms, let's look at a generic text-processing pipeline depicted in *Figure 9-1*. In text analysis, the input is usually presented as a stream of characters (depending on the specific language).

Lexical analysis has to do with breaking this stream into a sequence of words (or lexemes in linguistic analysis). Often it is also called tokenization (and the words called the tokens). **A**nother **T**ool for **L**anguage **R**ecognition (**ANTLR**) (<http://www.antlr.org/>) and Flex (<http://flex.sourceforge.net>) are probably the most famous in the open source community. One of the classical examples of ambiguity is lexical ambiguity. For example, in the phrase *I saw a bat.* *bat* can mean either an animal or a baseball bat. We usually need context to figure this out, which we will discuss next:

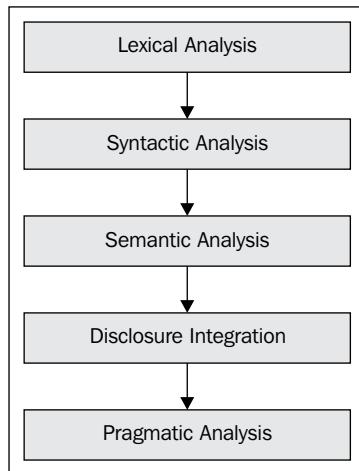


Figure 9-1. Typical stages of an NLP process.

Syntactic analysis, or parsing, traditionally deals with matching the structure of the text with grammar rules. This is relatively more important for computer languages that do not allow any ambiguity. In natural languages, this process is usually called chunking and tagging. In many cases, the meaning of the word in human language can be subject to context, intonation, or even body language or facial expression. The value of such analysis, as opposed to the big data approach, where the volume of data trumps complexity is still a contentious topic – one example of the latter is the word2vec approach, which will be described later.

Semantic analysis is the process of extracting language-independent meaning from the syntactic structures. As much as possible, it also involves removing features specific to particular cultural and linguistic contexts, to the extent that such a project is possible. The sources of ambiguity at this stage are: phrase attachment, conjunction, noun group structure, semantic ambiguity, anaphoric non-literal speech, and so on. Again, word2vec partially deals with these issues.

Disclosure integration partially deals with the issue of the context: the meaning of a sentence or an idiom can depend on the sentences or paragraphs before that. Syntactic analysis and cultural background play an important role here.

Finally, pragmatic analysis is yet another layer of complexity trying to reinterpret what is said in terms of what the intention was. How does this change the state of the world? Is it actionable?

Simple text analysis

The straightforward representation of the document is a bag of words. Scala, and Spark, provides an excellent paradigm to perform analysis on the word distributions. First, we read the whole collection of texts, and then count the unique words:

```
leotolstoy: org.apache.spark.rdd.RDD[String] = leotolstoy
MapPartitionsRDD[1] at textFile at <console>:27

scala> leotolstoy.flatMap(_.split("\\W+")).count
res1: Long = 1318234

scala> val shakespeare = sc.textFile("shakespeare").cache
shakespeare: org.apache.spark.rdd.RDD[String] = shakespeare
MapPartitionsRDD[7] at textFile at <console>:27

scala> shakespeare.flatMap(_.split("\\W+")).count
res2: Long = 1051958
```

This gives us just an estimate of the number of distinct words in the repertoire of quite different authors. The simplest way to find intersection between the two corpuses is to find the common vocabulary (which will be quite different as *Leo Tolstoy* wrote in Russian and French, while *Shakespeare* was an English-writing author):

```
scala> :silent

scala> val shakespeareBag = shakespeare.flatMap(_.split("\\W+")).map(_.toLowerCase).distinct

scala> val leotolstoyBag = leotolstoy.flatMap(_.split("\\W+")).map(_.toLowerCase).distinct
leotolstoyBag: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[27] at
map at <console>:29

scala> println("The bags intersection is " + leotolstoyBag.
intersection(shakespeareBag).count)
The bags intersection is 11552
```

A few thousands word indices are manageable with the current implementations. For any new story, we can determine whether it is more likely to be written by Leo Tolstoy or William Shakespeare. Let's take a look at *The King James Version of the Bible*, which also can be downloaded from Project Gutenberg (<https://www.gutenberg.org/files/10/10-h/10-h.htm>):

```
$ (mkdir bible; cd bible; wget http://www.gutenberg.org/cache/epub/10/pg10.txt)

scala> val bible = sc.textFile("bible").cache

scala> val bibleBag = bible.flatMap(_.split("\\W+")).map(_.toLowerCase).
distinct

scala>:silent

scala> bibleBag.intersection(shakespeareBag).count
res5: Long = 7250

scala> bibleBag.intersection(leotolstoyBag).count
res24: Long = 6611
```

This seems reasonable as the religious language was popular during the Shakespearean time. On the other hand, plays by *Anton Chekhov* have a larger intersection with the *Leo Tolstoy* vocabulary:

```
$ (mkdir chekhov; cd chekhov;
wget http://www.gutenberg.org/cache/epub/7986/pg7986.txt
wget http://www.gutenberg.org/cache/epub/1756/pg1756.txt
wget http://www.gutenberg.org/cache/epub/1754/1754.txt
wget http://www.gutenberg.org/cache/epub/13415/pg13415.txt)

scala> val chekhov = sc.textFile("c chekhov").cache
chekhov: org.apache.spark.rdd.RDD[String] = chekhov MapPartitionsRDD[61]
at textFile at <console>:27

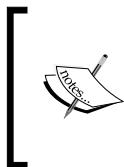
scala> val chekhovBag = chekhov.flatMap(_.split("\\W+")).map(_.toLowerCase).
distinct
```

```
chekhovBag: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[66] at
distinct at <console>:29
```

```
scala> chekhovBag.intersection(leotolstoyBag).count
res8: Long = 8263
```

```
scala> chekhovBag.intersection(shakespeareBag).count
res9: Long = 6457
```

This is a very simple approach that works, but there are a number of commonly known improvements we can make. First, a common technique is to stem the words. In many languages, words have a common part, often called root, and a changeable prefix or suffix, which may depend on the context, gender, time, and so on. Stemming is the process of improving the distinct count and intersection by approximating this flexible word form to the root, base, or a stem form in general. The stem form does not need to be identical to the morphological root of the word, it is usually sufficient that related words map to the same stem, even if this stem is not in itself a valid grammatical root. Secondly, we probably should account for the frequency of the words – while we will describe more elaborate methods in the next section, for the purpose of this exercise, we'll exclude the words with very high count, that usually are present in any document such as articles and possessive pronouns, which are usually called stop words, and the words with very low count. Specifically, I'll use the optimized **Porter Stemmer** implementation that I described in more detail at the end of the chapter.



The <http://tartarus.org/martin/PorterStemmer/> site contains some of the Porter Stemmer implementations in Scala and other languages, including a highly optimized ANSI C, which may be more efficient, but here I will provide another optimized Scala version that can be used immediately with Spark.



The Stemmer example will stem the words and count the relative intersections between them, removing the stop words:

```
def main(args: Array[String]) {
    val stemmer = new Stemmer
    val conf = new SparkConf().
        setAppName("Stemmer").
```

```
setMaster(args(0))

val sc = new SparkContext(conf)

val stopwords = scala.collection.immutable.TreeSet(
    "", "i", "a", "an", "and", "are", "as", "at", "be", "but",
    "by", "for", "from", "had", "has", "he", "her", "him", "his",
    "in", "is", "it", "its", "my", "not", "of", "on", "she",
    "that", "the", "to", "was", "were", "will", "with", "you"
) map { stemmer.stem(_) }

val bags = for (name <- args.slice(1, args.length)) yield {
    val rdd = sc.textFile(name).map(_.toLowerCase)
    if (name == "nytimes" || name == "nips" || name == "enron")
        rdd.filter(!_.startsWith("zzz_")).flatMap(_.split("_"))
            .map(stemmer.stem(_))
            .distinct.filter(!stopwords.contains(_)).cache
    else {
        val withCounts = rdd.flatMap(_.split("\\W+"))
            .map(stemmer.stem(_)).filter(!stopwords.contains(_))
            .map((_, 1)).reduceByKey(_+_)
        val minCount = scala.math.max(1L, 0.0001 *
            withCounts.count.toLong)
        withCounts.filter(_.value > minCount).map(_.key).cache
    }
}

val cntRoots = (0 until { args.length - 1 }).map(i =>
    Math.sqrt(bags(i).count.toDouble))

for(l <- 0 until { args.length - 1 }; r <- l until
    { args.length - 1 }) {
    val cnt = bags(l).intersection(bags(r)).count
    println("The intersect " + args(l+1) + " x " + args(r+1) + " "
        is: " + cnt + " (" +
        (cnt.toDouble / cntRoots(l) / cntRoots(r)) + ")")
}

sc.stop
}
```

When one runs the main class example from the command line, it outputs the stemmed bag sizes and intersection for datasets specified as parameters (these are directories in the home filesystem with documents):

```
$ sbt "run-main org.akozlov.examples.Stemmer local[2] shakespeare
leotolstoy chekhov nytimes nips enron bible"
[info] Loading project definition from /Users/akozlov/Src/Book/ml-in-
scala/chapter09/project
[info] Set current project to NLP in Scala (in build file:/Users/akozlov/
Src/Book/ml-in-scala/chapter09/)
[info] Running org.akozlov.examples.Stemmer local[2] shakespeare
leotolstoy chekhov nytimes nips enron bible
The intersect shakespeare x shakespeare is: 10533 (1.0)
The intersect shakespeare x leotolstoy is: 5834 (0.5293670391596142)
The intersect shakespeare x chekhov is: 3295 (0.4715281914492153)
The intersect shakespeare x nytimes is: 7207 (0.4163369701270161)
The intersect shakespeare x nips is: 2726 (0.27457329089479504)
The intersect shakespeare x enron is: 5217 (0.34431535832271265)
The intersect shakespeare x bible is: 3826 (0.45171392986714726)
The intersect leotolstoy x leotolstoy is: 11531 (0.9999999999999999)
The intersect leotolstoy x chekhov is: 4099 (0.5606253333241973)
The intersect leotolstoy x nytimes is: 8657 (0.47796976891152176)
The intersect leotolstoy x nips is: 3231 (0.3110369262979765)
The intersect leotolstoy x enron is: 6076 (0.38326210407266764)
The intersect leotolstoy x bible is: 3455 (0.3898604013063757)
The intersect chekhov x chekhov is: 4636 (1.0)
The intersect chekhov x nytimes is: 3843 (0.33463022711780555)
The intersect chekhov x nips is: 1889 (0.28679311682962116)
The intersect chekhov x enron is: 3213 (0.31963226496874225)
The intersect chekhov x bible is: 2282 (0.40610513998395287)
The intersect nytimes x nytimes is: 28449 (1.0)
The intersect nytimes x nips is: 4954 (0.30362042173997206)
The intersect nytimes x enron is: 11273 (0.45270741164576034)
The intersect nytimes x bible is: 3655 (0.2625720159205085)
The intersect nips x nips is: 9358 (1.0000000000000002)
The intersect nips x enron is: 4888 (0.3422561629856124)
The intersect nips x bible is: 1615 (0.20229053645165143)
```

```
The intersect enron x enron is: 21796 (1.0)
The intersect enron x bible is: 2895 (0.23760453654690084)
The intersect bible x bible is: 6811 (1.0)
[success] Total time: 12 s, completed May 17, 2016 11:00:38 PM
```

This, in this case, just confirms the hypothesis that Bible's vocabulary is closer to *William Shakespeare* than to Leo Tolstoy and other sources. Interestingly, modern vocabularies of *NY Times* articles and Enron's e-mails from the previous chapters are much closer to *Leo Tolstoy's*, which is probably more an indication of the translation quality.

Another thing to notice is that the pretty complex analysis took about 40 lines of Scala code (not counting the libraries, specifically the Porter Stemmer, which is about ~ 100 lines) and about 12 seconds. The power of Scala is that it can leverage other libraries very efficiently to write concise code.

Serialization

We already talked about serialization in *Chapter 6, Working with Unstructured Data*. As Spark's tasks are executed in different threads and potentially JVMs, Spark does a lot of serialization/deserialization when passing the objects. Potentially, I could use `map { val stemmer = new Stemmer; stemmer.stem(_) }` instead of `map { stemmer.stem(_) }`, but the latter reuses the object for multiple iterations and seems to be linguistically more appealing. One suggested performance optimization is to use *Kryo serializer*, which is less flexible than the Java serializer, but more performant. However, for integrative purpose, it is much easier to just make every object in the pipeline serializable and use default Java serialization.



As another example, let's compute the distribution of word frequencies, as follows:

```
scala> val bags = for (name <- List("shakespeare", "leotolstoy",
  "chekhov", "nytimes", "enron", "bible")) yield {
    |   sc.textFile(name).flatMap(_.split("\\W+")).map {
    |     _.toLowerCase}.map { stemmer.stem(_) }.filter { !stopwords.contains(_) }
    |   cache()
    | }
```

bags: List[org.apache.spark.rdd.RDD[String]] = List(MapPartitionsRDD[93]
at filter at <console>:36, MapPartitionsRDD[98] at filter at
<console>:36, MapPartitionsRDD[103] at filter at <console>:36,
MapPartitionsRDD[108] at filter at <console>:36, MapPartitionsRDD[113] at
filter at <console>:36, MapPartitionsRDD[118] at filter at <console>:36)

```

scala> bags reduceLeft { (a, b) => a.union(b) } map { (_, 1) }
reduceByKey { _+_ } collect() sortBy(- _.2) map { x => scala.math.
log(x._2) }

res18: Array[Double] = Array(10.27759958298627, 10.1152465449837,
10.058652004037477, 10.046635061754612, 9.999615579630348,
9.855399641729074, 9.834405391348684, 9.801233318497372,
9.792667717430884, 9.76347807952779, 9.742496866444002,
9.655474810542554, 9.630365631415676, 9.623244409181346,
9.593355351246755, 9.517604459155686, 9.515837804297965,
9.47231994707559, 9.45930760329985, 9.441531454869693, 9.435561763085358,
9.426257878198653, 9.378985497953893, 9.355997944398545,
9.34862295977619, 9.300820725104558, 9.25569607369698, 9.25320827220336,
9.229162126216771, 9.20391980417326, 9.19917830726999, 9.167224080902555,
9.153875834995056, 9.137877200242468, 9.129889247578555,
9.090430075303626, 9.090091799380007, 9.083075020930307,
9.077722847361343, 9.070273383079064, 9.0542711863262...
...

```

The distribution of relative frequencies on the log-log scale is presented in the following diagram. With the exception of the first few tokens, the dependency of frequency on rank is almost linear:

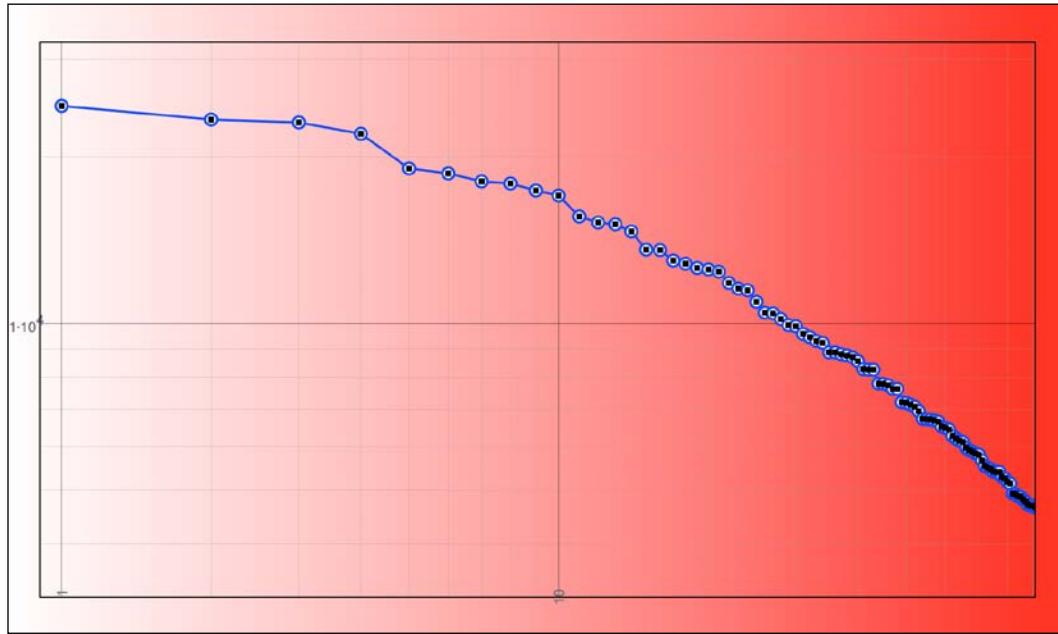


Figure 9-2. A typical distribution of word relative frequencies on log-log scale (Zipf's Law)

MLlib algorithms in Spark

Let's halt at MLlib that complements other NLP libraries written in Scala. MLlib is primarily important because of scalability, and thus supports a few of the data preparation and text processing algorithms, particularly in the area of feature construction (<http://spark.apache.org/docs/latest/ml-features.html>).

TF-IDF

Although the preceding analysis can already give a powerful insight, the piece of information that is missing from the analysis is term frequency information. The term frequencies are relatively more important in information retrieval, where the collection of documents need to be searched and ranked in relation to a few terms. The top documents are usually returned to the user.

TF-IDF is a standard technique where term frequencies are offset by the frequencies of the terms in the corpus. Spark has an implementation of the TF-IDF. Spark uses a hash function to identify the terms. This approach avoids the need to compute a global term-to-index map, but can be subject to potential hash collisions, the probability of which is determined by the number of buckets of the hash table. The default feature dimension is $2^{20}=1,048,576$.

In the Spark implementation, each document is a line in the dataset. We can convert it into to an RDD of iterables and compute the hashing by the following code:

```
scala> import org.apache.spark.mllib.feature.HashingTF
import org.apache.mllib.feature.HashingTF

scala> import org.apache.spark.mllib.linalg.Vector
import org.apache.mllib.linalg.Vector

scala> val hashingTF = new HashingTF
hashingTF: org.apache.spark.mllib.feature.HashingTF = org.apache.spark.
mllib.feature.HashingTF@61b975f7

scala> val documents: RDD[Seq[String]] = sc.textFile("shakespeare").map(_.split("\\W+").toSeq)
documents: org.apache.spark.rdd.RDD[Seq[String]] = MapPartitionsRDD[263]
at map at <console>:34

scala> val tf = hashingTF transform documents
tf: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] =
MapPartitionsRDD[264] at map at HashingTF.scala:76
```

When computing hashingTF, we only need a single pass over the data, applying IDF needs two passes: first to compute the IDF vector and second to scale the term frequencies by IDF:

```
scala> tf.cache
res26: tf.type = MapPartitionsRDD[268] at map at HashingTF.scala:76

scala> import org.apache.spark.mllib.feature.IDF
import org.apache.spark.mllib.feature.IDF

scala> val idf = new IDF(minDocFreq = 2) fit tf
idf: org.apache.spark.mllib.feature.IDFModel = org.apache.spark.mllib.
feature.IDFModel@514bda2d

scala> val tfidf = idf transform tf
tfidf: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] =
MapPartitionsRDD[272] at mapPartitions at IDF.scala:178

scala> tfidf take(10) foreach println
(1048576,[3159,3543,84049,582393,787662,838279,928610,961626,1021219,1021
273],[3.9626355004005083,4.556357737874695,8.380602528651274,8.1577369746
83708,11.513471982269106,9.316247404932888,10.666174121881904,11.51347198
2269106,8.07948477778396,11.002646358503116])
(1048576,[267794,1021219],[8.783442874448122,8.07948477778396])
(1048576,[0],[0.5688129477150906])
(1048576,[3123,3370,3521,3543,96727,101577,114801,116103,497275,504006,50
8606,843002,962509,980206],[4.207164322003765,2.9674322162952897,4.125144
122691999,2.2781788689373474,2.132236195047438,3.2951341639027754,1.92045
75904855747,6.318664992090735,11.002646358503116,3.1043838099579815,5.451
238364272918,11.002646358503116,8.43769700104158,10.30949917794317])
(1048576,[0,3371,3521,3555,27409,89087,104545,107877,552624,735790,910062
,943655,962421],[0.5688129477150906,3.442878442319589,4.125144122691999,4
.462482535201062,5.023254392629403,5.160262034409286,5.646060083831103,4.
712188947797486,11.002646358503116,7.006282204641219,6.216822672821767,11
.513471982269106,8.898512204232908])
(1048576,[3371,3543,82108,114801,149895,279256,582393,597025,838279,91518
1],[3.442878442319589,2.2781788689373474,6.017670811187438,3.840915180971
1495,7.893585399642122,6.625632265652778,8.157736974683708,10.41485969360
0997,9.316247404932888,11.513471982269106])
(1048576,[3123,3555,413342,504006,690950,702035,980206],[4.20716432200376
5,4.462482535201062,3.4399651117812313,3.1043838099579815,11.513471982269
106,11.002646358503116,10.30949917794317])
```

```
(1048576, [0], [0.5688129477150906])
(1048576, [97, 1344, 3370, 100898, 105489, 508606, 582393, 736902, 838279, 1026302]
, [2.533299776544098, 23.026943964538212, 2.9674322162952897, 0.0, 11.22578990
9817326, 5.451238364272918, 8.157736974683708, 10.30949917794317, 9.316247404
932888, 11.513471982269106])
(1048576, [0, 1344, 3365, 114801, 327690, 357319, 413342, 692611, 867249, 965170], [
4.550503581720725, 23.026943964538212, 2.7455719545259836, 1.920457590485574
7, 8.268278849083533, 9.521041817578901, 3.4399651117812313, 0.0, 6.6614417183
49489, 0.0])
```

Here we see each document represented by a set of terms and their scores.

LDA

LDA in Spark MLlib is a clustering mechanism, where the feature vectors represent the counts of words in a document. The model maximizes the probability of observing the word counts, given the assumption that each document is a mixture of topics and the words in the documents are generated based on **Dirichlet distribution** (a generalization of beta distribution on multinomial case) for each of the topic independently. The goal is to derive the (latent) distribution of the topics and the parameters of the words generation statistical model.

The MLlib implementation is based on 2009 LDA paper (<http://www.jmlr.org/papers/volume10/newman09a/newman09a.pdf>) and uses GraphX to implement a distributed **Expectation Maximization (EM)** algorithm for assigning topics to the documents.

Let's take the Enron e-mail corpus discussed in *Chapter 7, Working with Graph Algorithms*, where we tried to figure out communications graph. For e-mail clustering, we need to extract the body of the e-mail and place it as a single line in the training file:

```
$ mkdir enron
$ cat /dev/null > enron/all.txt
$ for f in $(find maildir -name '*\. -print'); do cat $f | sed
'1,/^$/d; /^$/d' | tr "\n\r" " " >> enron/all.txt; echo "" >> enron/all.
txt; done
$
```

Now, let's use Scala/Spark to construct a corpus dataset containing the document ID, followed by a dense array of word counts in the bag:

```
$ spark-shell --driver-memory 8g --executor-memory 8g --packages com.
github.fommil.netlib:all:1.1.2
Ivy Default Cache set to: /home/alex/.ivy2/cache
```

```
The jars for the packages stored in: /home/alex/.ivy2/jars
:: loading settings :: url = jar:file:/opt/cloudera/parcels/CDH-
5.5.2-1.cdh5.5.2.p0.4/jars/spark-assembly-1.5.0-cdh5.5.2-hadoop2.6.0-
cdh5.5.2.jar!/org/apache/ivy/core/settings/ivysettings.xml
com.github.fommil.netlib#all added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent;1.0
  confs: [default]
    found com.github.fommil.netlib#all;1.1.2 in central
    found net.sourceforge.f2j#arpack_combined_all;0.1 in central
    found com.github.fommil.netlib#core;1.1.2 in central
    found com.github.fommil.netlib#netlib-native_ref-osx-x86_64;1.1 in
central
    found com.github.fommil.netlib#native_ref-java;1.1 in central
    found com.github.fommil#jniloader;1.1 in central
    found com.github.fommil.netlib#netlib-native_ref-linux-x86_64;1.1 in
central
    found com.github.fommil.netlib#netlib-native_ref-linux-i686;1.1 in
central
    found com.github.fommil.netlib#netlib-native_ref-win-x86_64;1.1 in
central
    found com.github.fommil.netlib#netlib-native_ref-win-i686;1.1 in
central
    found com.github.fommil.netlib#netlib-native_ref-linux-armhf;1.1 in
central
    found com.github.fommil.netlib#netlib-native_system-osx-x86_64;1.1 in
central
    found com.github.fommil.netlib#native_system-java;1.1 in central
    found com.github.fommil.netlib#netlib-native_system-linux-x86_64;1.1 in
central
    found com.github.fommil.netlib#netlib-native_system-linux-i686;1.1 in
central
    found com.github.fommil.netlib#netlib-native_system-linux-armhf;1.1 in
central
    found com.github.fommil.netlib#netlib-native_system-win-x86_64;1.1 in
central
    found com.github.fommil.netlib#netlib-native_system-win-i686;1.1 in
central
downloading https://repo1.maven.org/maven2/net/sourceforge/f2j/arpack_
combined_all/0.1/arpack_combined_all-0.1-javadoc.jar ...
```

```
[SUCCESSFUL] net.sourceforge.f2j#arpack_combined_all;0.1!arpack_
combined_all.jar (513ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/
core/1.1.2/core-1.1.2.jar ...
[SUCCESSFUL] com.github.fommil.netlib#core;1.1.2!core.jar (18ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/
netlib-native_ref-osx-x86_64/1.1/netlib-native_ref-osx-x86_64-1.1-
natives.jar ...
[SUCCESSFUL] com.github.fommil.netlib#netlib-native_ref-osx-
x86_64;1.1!netlib-native_ref-osx-x86_64.jar (167ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/
netlib-native_ref-linux-x86_64/1.1/netlib-native_ref-linux-x86_64-1.1-
natives.jar ...
[SUCCESSFUL] com.github.fommil.netlib#netlib-native_ref-linux-
x86_64;1.1!netlib-native_ref-linux-x86_64.jar (159ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/
netlib-native_ref-linux-i686/1.1/netlib-native_ref-linux-i686-1.1-
natives.jar ...
[SUCCESSFUL] com.github.fommil.netlib#netlib-native_ref-linux-
i686;1.1!netlib-native_ref-linux-i686.jar (131ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/
netlib-native_ref-win-x86_64/1.1/netlib-native_ref-win-x86_64-1.1-
natives.jar ...
[SUCCESSFUL] com.github.fommil.netlib#netlib-native_ref-win-
x86_64;1.1!netlib-native_ref-win-x86_64.jar (210ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/
netlib-native_ref-win-i686/1.1/netlib-native_ref-win-i686-1.1-natives.jar
...
[SUCCESSFUL] com.github.fommil.netlib#netlib-native_ref-win-
i686;1.1!netlib-native_ref-win-i686.jar (167ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/
netlib-native_ref-linux-armhf/1.1/netlib-native_ref-linux-armhf-1.1-
natives.jar ...
[SUCCESSFUL] com.github.fommil.netlib#netlib-native_ref-linux-
armhf;1.1!netlib-native_ref-linux-armhf.jar (110ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/
netlib-native_system-osx-x86_64/1.1/netlib-native_system-osx-x86_64-1.1-
natives.jar ...
[SUCCESSFUL] com.github.fommil.netlib#netlib-native_system-osx-
x86_64;1.1!netlib-native_system-osx-x86_64.jar (54ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/
netlib-native_system-linux-x86_64/1.1/netlib-native_system-linux-x86_64-
1.1-natives.jar ...
```

```
[SUCCESSFUL ] com.github.fommil.netlib#netlib-native_system-linux-x86_64;1.1!netlib-native_system-linux-x86_64.jar (47ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/netlib-native_system-linux-i686/1.1/netlib-native_system-linux-i686-1.1-natives.jar ...
[SUCCESSFUL ] com.github.fommil.netlib#netlib-native_system-linux-i686;1.1!netlib-native_system-linux-i686.jar (44ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/netlib-native_system-linux-armhf/1.1/netlib-native_system-linux-armhf-1.1-natives.jar ...
[SUCCESSFUL ] com.github.fommil.netlib#netlib-native_system-linux-armhf;1.1!netlib-native_system-linux-armhf.jar (35ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/netlib-native_system-win-x86_64/1.1/netlib-native_system-win-x86_64-1.1-natives.jar ...
[SUCCESSFUL ] com.github.fommil.netlib#netlib-native_system-win-x86_64;1.1!netlib-native_system-win-x86_64.jar (62ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/netlib-native_system-win-i686/1.1/netlib-native_system-win-i686-1.1-natives.jar ...
[SUCCESSFUL ] com.github.fommil.netlib#netlib-native_system-win-i686;1.1!netlib-native_system-win-i686.jar (55ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/native_ref-java/1.1/native_ref-java-1.1.jar ...
[SUCCESSFUL ] com.github.fommil.netlib#native_ref-java;1.1!native_ref-java.jar (24ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/jniloader/1.1/jniloader-1.1.jar ...
[SUCCESSFUL ] com.github.fommil.jniloader;1.1!jniloader.jar (3ms)
downloading https://repo1.maven.org/maven2/com/github/fommil/netlib/native_system-java/1.1/native_system-java-1.1.jar ...
[SUCCESSFUL ] com.github.fommil.netlib#native_system-java;1.1!native_system-java.jar (7ms)
:: resolution report :: resolve 3366ms :: artifacts dl 1821ms
:: modules in use:
com.github.fommil.jniloader;1.1 from central in [default]
com.github.fommil.netlib.all;1.1.2 from central in [default]
com.github.fommil.netlib.core;1.1.2 from central in [default]
com.github.fommil.netlib.native_ref-java;1.1 from central in [default]
com.github.fommil.netlib.native_system-java;1.1 from central in [default]
```

```
com.github.fommil.netlib#netlib-native_ref-linux-armhf;1.1 from central
in [default]
com.github.fommil.netlib#netlib-native_ref-linux-i686;1.1 from central
in [default]
com.github.fommil.netlib#netlib-native_ref-linux-x86_64;1.1 from
central in [default]
com.github.fommil.netlib#netlib-native_ref-osx-x86_64;1.1 from central
in [default]
com.github.fommil.netlib#netlib-native_ref-win-i686;1.1 from central in
[default]
com.github.fommil.netlib#netlib-native_ref-win-x86_64;1.1 from central
in [default]
com.github.fommil.netlib#netlib-native_system-linux-armhf;1.1 from
central in [default]
com.github.fommil.netlib#netlib-native_system-linux-i686;1.1 from
central in [default]
com.github.fommil.netlib#netlib-native_system-linux-x86_64;1.1 from
central in [default]
com.github.fommil.netlib#netlib-native_system-osx-x86_64;1.1 from
central in [default]
com.github.fommil.netlib#netlib-native_system-win-i686;1.1 from central
in [default]
com.github.fommil.netlib#netlib-native_system-win-x86_64;1.1 from
central in [default]
net.sourceforge.f2j#arpack_combined_all;0.1 from central in [default]
:: evicted modules:
com.github.fommil.netlib#core;1.1 by [com.github.fommil.
netlib#core;1.1.2] in [default]

-----
|                               |           modules           ||      artifacts   |
|       conf        | number| search|dwnlded|evicted||  number|dwnlded|
-----
|       default     |   19  |   18  |   18  |    1   ||   17  |   17  |
-----
...
scala> val enron = sc.textFile("enron")
enron: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile
at <console>:21

scala> enron.flatMap(_.split("\\W+")).map(_.toLowerCase).distinct.count
```

[1208]

```

res0: Long = 529199

scala> val stopwords = scala.collection.immutable.TreeSet("", "i", "a",
"an", "and", "are", "as", "at", "be", "but", "by", "for", "from", "had",
"has", "he", "her", "him", "his", "in", "is", "it", "its", "not", "of",
"on", "she", "that", "the", "to", "was", "were", "will", "with", "you")
stopwords: scala.collection.immutable.TreeSet[String] = TreeSet(, a, an,
and, are, as, at, be, but, by, for, from, had, has, he, her, him, his, i,
in, is, it, its, not, of, on, she, that, the, to, was, were, will, with,
you)

scala>

scala> val terms = enron.flatMap(x => if (x.length < 8192) x.toLowerCase.
split("\\W+") else Nil).filterNot(stopwords).map(_,_1).reduceByKey(_+_).
collect.sortBy(- _._2).slice(0, 1000).map(_._1)
terms: Array[String] = Array(enron, ect, com, this, hou, we, s, have,
subject, or, 2001, if, your, pm, am, please, cc, 2000, e, any, me, 00,
message, 1, corp, would, can, 10, our, all, sent, 2, mail, 11, re,
thanks, original, know, 12, 713, http, may, t, do, 3, time, 01, ees, m,
new, my, they, no, up, information, energy, us, gas, so, get, 5, about,
there, need, what, call, out, 4, let, power, should, na, which, one, 02,
also, been, www, other, 30, email, more, john, like, these, 03, mark,
04, attached, d, enron_development, their, see, 05, j, forwarded, market,
some, agreement, 09, day, questions, meeting, 08, when, houston, doc,
contact, company, 6, just, jeff, only, who, 8, fax, how, deal, could, 20,
business, use, them, date, price, 06, week, here, net, 15, 9, 07, group,
california,...)

scala> def getBagCounts(bag: Seq[String]) = { for(term <- terms) yield {
bag.count(_==term) } }
getBagCounts: (bag: Seq[String])Array[Int]

scala> import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.Vectors

scala> val corpus = enron.map(x => { if (x.length < 8192) Some(x.
toLowerCase.split("\\W+").toSeq) else None } ).map(x => { Vectors.
dense(getBagCounts(x.getOrElse(Nil)).map(_.toDouble).toArray)
}).zipWithIndex.map(_._swap).cache
corpus: org.apache.spark.rdd.RDD[(Long, org.apache.spark.mllib.linalg.
Vector)] = MapPartitionsRDD[14] at map at <console>:30

scala> import org.apache.spark.mllib.clustering.{LDA,
DistributedLDAModel}

```

```
import org.apache.spark.mllib.clustering.{LDA, DistributedLDAModel}

scala> import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.Vectors

scala> val ldaModel = new LDA().setK(10).run(corpus)
...
scala> ldaModel.topicsMatrix.transpose
res2: org.apache.spark.mllib.linalg.Matrix =
207683.78495933366 79745.88417942637 92118.63972404732 ... (1000
total)
35853.48027575886 4725.178508682296 111214.8860582083 ...
135755.75666585402 54736.471356209106 93289.65563593085 ...
39445.796099155996 6272.534431534215 34764.02707696523 ...
329786.21570967307 602782.9591026317 42212.22143362559 ...
62235.09960154089 12191.826543794878 59343.24100019015 ...
210049.59592560542 160538.9650732507 40034.69756641789 ...
53818.14660186875 6351.853448001488 125354.26708575874 ...
44133.150537842856 4342.697652158682 154382.95646078113 ...
90072.97362336674 21132.629704311104 93683.40795807641 ...
```

We can also list the words and their relative importance for the topic in the descending order:

```
scala> ldaModel.describeTopics foreach { x : (Array[Int], Array[Double]) => { print(x._1.slice(0,10).map(terms(_)).mkString(":")); print("-> ");
print(x._2.slice(0,10).map(_.toFloat).mkString(":")); println } }
com:this:ect:or:if:s:hou:2001:00:we-> 0.054606363:0.024220783:0.02096761:
0.013669214:0.0132700335:0.012969772:0.012623918:0.011363528:0.010114557:
0.009587474
s:this:hou:your:2001:or:please:am:com:new-> 0.029883621:0.027119286:0.013
396418:0.012856948:0.01218803:0.01124849:0.010425644:0.009812181:0.008742
722:0.0070441025
com:this:s:ect:hou:or:2001:if:your:am-> 0.035424445:0.024343235:0.0151826
28:0.014283071:0.013619815:0.012251413:0.012221165:0.011411696:0.01028402
4:0.009559739
would:pm:cc:3:thanks:e:my:all:there:11-> 0.047611523:0.034175437:0.022914
853:0.019933242:0.017208714:0.015393614:0.015366959:0.01393391:0.01257752
5:0.011743208
```

```

ect:com:we:can:they:03:if:also:00:this-> 0.13815293:0.0755843:0.065043546
:0.015290086:0.0121941045:0.011561104:0.011326733:0.010967959:0.010653805
:0.009674695

com:this:s:hou:or:2001:pm:your:if:cc-> 0.016605735:0.015834121:0.01289918
:0.012708308:0.0125788655:0.011726159:0.011477625:0.010578845:0.010555539
:0.009609056

com:ect:we:if:they:hou:s:00:2001:or-> 0.05537054:0.04231919:0.023271963:0
.012856676:0.012689817:0.012186356:0.011350313:0.010887237:0.010778923:0.
010662295

this:s:hou:com:your:2001:or:please:am:if-> 0.030830953:0.016557815:0.0142
36835:0.013236604:0.013107091:0.0126846135:0.012257128:0.010862533:0.0102
7849:0.008893094

this:s:or:pm:com:your:please:new:hou:2001-> 0.03981197:0.013273305:0.0128
72894:0.011672661:0.011380969:0.010689667:0.009650983:0.009605533:0.00953
5899:0.009165275

this:com:hou:s:or:2001:if:your:am:please-> 0.024562683:0.02361607:0.01377
0585:0.013601272:0.01269994:0.012360005:0.011348433:0.010228578:0.0096196
28:0.009347991

```

To find out the top documents per topic or top topics per document, we need to convert this model to `DistributedLDA` or `LocalLDAModel`, which extend `LDAModel`:

```

scala> ldaModel.save(sc, "ldamodel")

scala> val sameModel = DistributedLDAModel.load(sc, "ldamodel21")

scala> sameModel.topDocumentsPerTopic(10) foreach { x : (Array[Long],
Array[Double]) => { print(x._1.mkString(":")); print("-> "); print(x._2.
map(_.toFloat).mkString(":")); println } }
59784:50745:52479:60441:58399:49202:64836:52490:67936:67938-> 0.97146696:
0.9713364:0.9661418:0.9661132:0.95249915:0.9519995:0.94945914:0.94944507:
0.8977366:0.8791358
233009:233844:233007:235307:233842:235306:235302:235293:233020:233857->
0.9962034:0.9962034:0.9962034:0.9962034:0.9962034:0.99620336:0.9954057:0.
9954057:0.9954057:0.9954057
14909:115602:14776:39025:115522:288507:4499:38955:15754:200876-> 0.839639
07:0.83415157:0.8319566:0.8303818:0.8291597:0.8281472:0.82739806:0.827251
7:0.82579833:0.8243338
237004:71818:124587:278308:278764:278950:233672:234490:126637:123664-> 0.
99929106:0.9968135:0.9964454:0.99644524:0.996445:0.99644494:0.99644476:0.
9964447:0.99644464:0.99644417

```

```
156466:82237:82252:82242:341376:82501:341367:340197:82212:82243-> 0.99716
955:0.94635135:0.9431836:0.94241136:0.9421047:0.9410431:0.94075173:0.9406
304:0.9402021:0.94014835

335708:336413:334075:419613:417327:418484:334157:335795:337573:334160->
0.987011:0.98687994:0.9865438:0.96953565:0.96953565:0.96953565:0.9588571:
0.95852506:0.95832515:0.9581657

243971:244119:228538:226696:224833:207609:144009:209548:143066:195299->
0.7546907:0.7546907:0.59146744:0.59095955:0.59090924:0.45532238:0.4506441
7:0.44945204:0.4487876:0.44833568

242260:214359:126325:234126:123362:233304:235006:124195:107996:334829->
0.89615464:0.8961442:0.8106028:0.8106027:0.8106023:0.8106023:0.8106021:0.
8106019:0.76834095:0.7570231

209751:195546:201477:191758:211002:202325:197542:193691:199705:329052->
0.913124:0.9130985:0.9130918:0.9130672:0.5525752:0.5524637:0.5524494:0.55
2405:0.55240136:0.5026157

153326:407544:407682:408098:157881:351230:343651:127848:98884:129351-> 0.
97206575:0.97206575:0.97206575:0.97206575:0.97206575:0.9689198:0.968068:0
.9659192:0.9657442:0.96553063
```

Segmentation, annotation, and chunking

When the text is presented in digital form, it is relatively easy to find words as we can split the stream on non-word characters. This becomes more complex in spoken language analysis. In this case, segmenters try to optimize a metric, for example, to minimize the number of distinct words in the lexicon and the length or complexity of the phrase (*Natural Language Processing with Python* by Steven Bird *et al*, O'Reilly Media Inc, 2009).

Annotation usually refers to parts-of-speech tagging. In English, these are nouns, pronouns, verbs, adjectives, adverbs, articles, prepositions, conjunctions, and interjections. For example, in the phrase *we saw the yellow dog*, *we* is a pronoun, *saw* is a verb, *the* is an article, *yellow* is an adjective, and *dog* is a noun.

In some languages, the chunking and annotation depends on context. For example, in Chinese, 爱江山人 literally translates to *love country person* and can mean either *country-loving person* or *love country-person*. In Russian, казнить нельзя помиловать, literally translating to *execute not pardon*, can mean *execute*, *don't pardon*, or *don't execute, pardon*. While in written language, this can be disambiguated using commas, in a spoken language this is usually it is very hard to recognize the difference, even though sometimes the intonation can help to segment the phrase properly.

For techniques based on word frequencies in the bags, some extremely common words, which are of little value in helping select documents, are explicitly excluded from the vocabulary. These words are called stop words. There is no good general strategy for determining a stop list, but in many cases, this is to exclude very frequent words that appear in almost every document and do not help to differentiate between them for classification or information retrieval purposes.

POS tagging

POS tagging probabilistically annotates each word with its grammatical function—noun, verb, adjective, and so on. Usually, POS tagging serves as an input to syntactic and semantic analysis. Let's demonstrate POS tagging on the FACTORIE toolkit example, a software library written in Scala (<http://factorie.cs.umass.edu>). To start, you need to download the binary image or source files from <https://github.com/factorie/factorie.git> and build it:

```
$ git clone https://github.com/factorie/factorie.git  
...  
$ cd factorie  
$ git checkout factorie_2.11-1.2  
...  
$ mvn package -Pnlp-jar-with-dependencies
```

After the build, which also includes model training, the following command will start a network server on port 3228:

```
$ $ bin/fac nlp --wsj-forward-pos --conll-chain-ner  
java -Xmx6g -ea -Djava.awt.headless=true -Dfile.encoding=UTF-8 -server  
-classpath ./src/main/resources:/target/classes:/target/factorie_2.11-  
1.2-nlp-jar-with-dependencies.jar  
found model  
18232  
Listening on port 3228  
...
```

Now, all traffic to port 3228 will be interpreted (as text), and the output will be tokenized and annotated:

```
$ telnet localhost 3228  
Trying ::1...  
Connected to localhost.  
Escape character is '^]'.
```

But I warn you, if you don't tell me that this means war, if you still try to defend the infamies and horrors perpetrated by that Antichrist--I really believe he is Antichrist--I will have nothing more to do with you and you are no longer my friend, no longer my 'faithful slave,' as you call yourself! But how do you do? I see I have frightened you--sit down and tell me all the news.

```
1 1 But CC O
2 2 I PRP O
3 3 warn VBP O
4 4 you PRP O
5 5 , O
6 6 if IN O
7 7 you PRP O
8 8 do VBP O
9 9 n't RB O
10 10 tell VB O
11 11 me PRP O
12 12 that IN O
13 13 this DT O
14 14 means VBZ O
15 15 war NN O
16 16 , , O
17 17 if IN O
18 18 you PRP O
19 19 still RB O
20 20 try VBP O
21 21 to TO O
22 22 defend VB O
23 23 the DT O
24 24 infamies NNS O
25 25 and CC O
26 26 horrors NNS O
27 27 perpetrated VBN O
28 28 by IN O
29 29 that DT O
30 30 Antichrist NNP O
```

31 31 -- : O
32 1 I PRP O
33 2 really RB O
34 3 believe VBP O
35 4 he PRP O
36 5 is VBZ O
37 6 Antichrist NNP U-MISC
38 7 -- : O
39 1 I PRP O
40 2 will MD O
41 3 have VB O
42 4 nothing NN O
43 5 more JJR O
44 6 to TO O
45 7 do VB O
46 8 with IN O
47 9 you PRP O
48 10 and CC O
49 11 you PRP O
50 12 are VBP O
51 13 no RB O
52 14 longer RBR O
53 15 my PRP\$ O
54 16 friend NN O
55 17 , , O
56 18 no RB O
57 19 longer RB O
58 20 my PRP\$ O
59 21 ' POS O
60 22 faithful NN O
61 23 slave NN O
62 24 , , O
63 25 ' '' O
64 26 as IN O
65 27 you PRP O
66 28 call VBP O

```
67 29 yourself PRP O
68 30 ! . O
69 1 But CC O
70 2 how WRB O
71 3 do VBP O
72 4 you PRP O
73 5 do VB O
74 6 ? . O
75 1 I PRP O
76 2 see VBP O
77 3 I PRP O
78 4 have VBP O
79 5 frightened VBN O
80 6 you PRP O
81 7 -- : O
82 8 sit VB O
83 9 down RB O
84 10 and CC O
85 11 tell VB O
86 12 me PRP O
87 13 all DT O
88 14 the DT O
89 15 news NN O
90 16 . . O
```

This POS is a single-path left-right tagger that can process the text as a stream. Internally, the algorithm uses probabilistic techniques to find the most probable assignment. Let's also look at other techniques that do not use grammatical analysis and yet proved to be very useful for language understanding and interpretation.

Using word2vec to find word relationships

Word2vec has been developed by Tomas Mikolov at Google, around 2012. The original idea behind word2vec was to demonstrate that one might improve efficiency by trading the model's complexity for efficiency. Instead of representing a document as bags of words, word2vec takes each word context into account by trying to analyze n-grams or skip-grams (a set of surrounding tokens with potential the token in question skipped). The words and word contexts themselves are represented by an array of floats/doubles u_t . The objective function is to maximize log likelihood:

$$\frac{1}{T} \sum_{t=1}^T \sum_{j=-k}^k \log p(w_{t+j} | w_t)$$

Where:

$$p(w_j | w_i) = \frac{\exp(u_j^T u_i)}{\sum_k \exp(u_k^T u_i)}$$

By choosing the optimal u_i and to get a comprehensive word representation (also called **map optimization**). Similar words are found based on cosine similarity metric (dot product) of u_i . Spark implementation uses hierarchical softmax, which reduces the complexity of computing the conditional probability to $O(\log(V))$, or log of the vocabulary size V , as opposed to $O(V)$, or proportional to V . The training is still linear in the dataset size, but is amenable to big data parallelization techniques.

Word2vec is traditionally used to predict the most likely word given context or find similar words with a similar meaning (synonyms). The following code trains in word2vec model on *Leo Tolstoy's Wars and Peace*, and finds synonyms for the word *circle*. I had to convert the Gutenberg's representation of *War and Peace* to a single-line format by running the `cat 2600.txt | tr "\n\r" " " > warandpeace.txt` command:

```
scala> val word2vec = new Word2Vec
word2vec: org.apache.spark.mllib.feature.Word2Vec = org.apache.spark.
mllib.feature.Word2Vec@58bb4dd

scala> val model = word2vec.fit(sc.textFile("warandpeace").map(_.split("\\W+").toSeq)
model: org.apache.spark.mllib.feature.Word2VecModel = org.apache.spark.
mllib.feature.Word2VecModel@6f61b9d7

scala> val synonyms = model.findSynonyms("life", 10)
synonyms: Array[(String, Double)] = Array((freedom,1.704344822168997),
(universal,1.682276637692245), (conception,1.6776193389148586),
(relation,1.6760497906519414), (humanity,1.67601036253831),
(consists,1.6637604144872544), (recognition,1.6526169382380496),
(subjection,1.6496559771230317), (activity,1.646671198014248),
(astronomy,1.6444424059160712))

scala> synonyms foreach println
(freedom,1.704344822168997)
(universal,1.682276637692245)
(conception,1.6776193389148586)
(relation,1.6760497906519414)
(humanity,1.67601036253831)
(consists,1.6637604144872544)
(recognition,1.6526169382380496)
(subjection,1.6496559771230317)
(activity,1.646671198014248)
(astronomy,1.6444424059160712)
```

While in general, it is hard to come with an objective function, and `freedom` is not listed as a synonym to `life` in the English Thesaurus, the results do make sense.

Each word in the word2vec model is represented as an array of doubles. Another interesting application is to find associations a to b is the same as c to ? by performing subtraction $\text{vector}(a) - \text{vector}(b) + \text{vector}(c)$:

```
scala> val a = model.getVectors.filter(_.1 == "monarchs").map(_.2).head
a: Array[Float] = Array(-0.0044642715, -0.0013227836, -0.011506443,
0.03691717, 0.020431392, 0.013427449, -0.0036369907, -0.013460356,
-3.8938568E-4, 0.02432113, 0.014533845, 0.004130258, 0.00671316,
-0.009344602, 0.006229065, -0.005442078, -0.0045390734, -0.0038824948,
-6.5973646E-4, 0.021729799, -0.011289608, -0.0030690092, -0.011423801,
0.009100784, 0.011765533, 0.0069619063, 0.017540144, 0.011198071,
0.026103685, -0.017285397, 0.0045515243, -0.0044477824, -0.0074411617,
-0.023975836, 0.011371289, -0.022625357, -2.6478301E-5, -0.010510282,
0.010622139, -0.009597833, 0.014937023, -0.01298345, 0.0016747514,
0.01172987, -0.001512275, 0.022340108, -0.009758578, -0.014942565,
0.0040697413, 0.0015349758, 0.010246878, 0.0021413323, 0.008739062,
0.007845526, 0.006857361, 0.01160148, 0.008595...

scala> val b = model.getVectors.filter(_.1 == "princess").map(_.2).head
b: Array[Float] = Array(0.13265875, -0.04882792, -0.08409957,
-0.04067986, 0.009084379, 0.121674284, -0.11963971, 0.06699862,
-0.20277102, 0.26296946, -0.058114383, 0.076021515, 0.06751665,
-0.17419271, -0.089830205, 0.2463593, 0.062816426, -0.10538805,
0.062085453, -0.2483566, 0.03468293, 0.20642486, 0.3129267, -0.12418643,
-0.12557726, 0.06725172, -0.03703333, -0.10810595, 0.06692443,
-0.046484336, 0.2433963, -0.12762263, -0.18473054, -0.084376186,
0.0037174677, -0.0040220995, -0.3419341, -0.25928706, -0.054454487,
0.09521076, -0.041567303, -0.13727514, -0.04826158, 0.13326299,
0.16228828, 0.08495835, -0.18073058, -0.018380836, -0.15691829,
0.056539804, 0.13673553, -0.027935665, 0.081865616, 0.07029694,
-0.041142456, 0.041359138, -0.2304657, -0.17088272, -0.14424285,
-0.0030700471, -0...

scala> val c = model.getVectors.filter(_.1 == "individual").map(_.2).
head
c: Array[Float] = Array(-0.0013353615, -0.01820516, 0.007949033,
0.05430816, -0.029520465, -0.030641818, -6.607431E-4, 0.026548808,
0.04784935, -0.006470232, 0.041406438, 0.06599842, 0.0074243015,
0.041538745, 0.0030222891, -0.003932073, -0.03154199, -0.028486902,
0.022139633, 0.05738223, -0.03890591, -0.06761177, 0.0055152955,
-0.02480924, -0.053222697, -0.028698998, -0.005315235, 0.0582403,
-0.0024816995, 0.031634405, -0.027884213, 6.0290704E-4, 1.9750209E-
4, -0.05563172, 0.023785716, -0.037577976, 0.04134448, 0.0026664822,
-0.019832063, -0.0011898747, 0.03160933, 0.031184288, 0.0025268437,
-0.02718441, -0.07729341, -0.009460656, 0.005344515, -0.05110715,
0.018468754, 0.008984449, -0.0053139487, 0.0053904117, -0.01322933,
-0.015247412, 0.009819351, 0.038043085, 0.044905875, 0.00402788...
```

```
scala> model.findSynonyms(new DenseVector((for(i <- 0 until 100) yield  
(a(i) - b(i) + c(i)).toDouble).toArray), 10) foreach println  
  
(achievement, 0.9432423663884002)  
(uncertainty, 0.9187759184842362)  
(leader, 0.9163721499105207)  
(individual, 0.9048367510621271)  
(instead, 0.8992079672038455)  
(cannon, 0.8947818781378154)  
(arguments, 0.8883634101905679)  
(aims, 0.8725107984356915)  
(ants, 0.8593842583047755)  
(War, 0.8530727227924755)
```

This can be used to find relationships in the language.

A Porter Stemmer implementation of the code

Porter Stemmer was first developed around the 1980s and there are many implementations. The detailed steps and original reference are provided at <http://tartarus.org/martin/PorterStemmer/def.txt>. It consists of roughly 6-9 steps of suffix/ endings replacements, some of which are conditional on prefix or stem. I will provide a Scala-optimized version with the book code repository. For example, step 1 covers the majority of stemming cases and consists of 12 substitutions: the last 8 of which are conditional on the number of syllables and the presence of vowels in the stem:

```
def step1(s: String) = {  
    b = s  
    // step 1a  
    processSubList(List(("sses", "ss"), ("ies", "i"),  
        ("ss", "ss"), ("s", "")), _>=0)  
    // step 1b  
    if (!(replacer("eed", "ee", _>0)))  
    {  
        if ((vowelInStem("ed") && replacer("ed", "", _>=0)) ||  
            (vowelInStem("ing") && replacer("ing", "", _>=0)))  
        {  
            if (!processSubList(List(("at", "ate"), ("bl", "ble"),  
                ("iz", "ize")), _>=0))  
            {  
                // if this isn't done, then it gets more confusing.  
            }  
        }  
    }  
}
```

```
        if (doublec() && b.last != 'l' && b.last != 's' &&
            b.last != 'z') { b = b.substring(0, b.length - 1) }
        else
            if (calcM(b.length) == 1 && cvc("")) { b = b + "e" }
    }
}
}
// step 1c
(vowelInStem("y") && replacer("y", "i", _>=0))
this
}
```

The complete code is available at <https://github.com/alexvk/ml-in-scala/blob/master/chapter09/src/main/scala/Stemmer.scala>.

Summary

In this chapter, I described basic NLP concepts and demonstrated a few basic techniques. I hoped to demonstrate that pretty complex NLP concepts could be expressed and tested in a few lines of Scala code. This is definitely just the tip of the iceberg as a lot of NLP techniques are being developed now, including the ones based on in-CPU parallelization as part of GPUs. (refer to, for example, **Puck** at <https://github.com/dlwh/puck>). I also gave a flavor of major Spark MLlib NLP implementations.

In the next chapter, which will be the final chapter of this book, I'll cover systems and model monitoring.

10

Advanced Model Monitoring

Even though this is the last chapter of the book, it can hardly be an afterthought even though monitoring in general often is in practical situations, quite unfortunately. Monitoring is a vital deployment component for any long execution cycle component and thus is part of the finished product. Monitoring can significantly enhance product experience and define future success as it improves problem diagnostic and is essential to determine the improvement path.

One of the primary rules of successful software engineering is to create systems as if they were targeted for personal use when possible, which fully applies to monitoring, diagnostic, and debugging – quite hapless name for fixing existing issues in software products. Diagnostic and debugging of complex systems, particularly distributed systems, is hard, as the events often can be arbitrary interleaved and program executions subject to race conditions. While there is a lot of research going in the area of distributed system devops and maintainability, this chapter will scratch the service and provide guiding principle to design a maintainable complex distributed system.

To start with, a pure functional approach, which Scala claims to follow, spends a lot of time avoiding side effects. While this idea is useful in a number of aspects, it is hard to imagine a useful program that has no effect on the outside world, the whole idea of a data-driven application is to have a positive effect on the way the business is conducted, a well-defined side effect.

Monitoring clearly falls in the side effect category. Execution needs to leave a trace that the user can later parse in order to understand where the design or implementation went awry. The trace of the execution can be left by either writing something on a console or into a file, usually called a log, or returning an object that contains the trace of the program execution, and the intermediate results. The latter approach, which is actually more in line with functional programming and monadic philosophy, is actually more appropriate for the distributed programming but often overlooked. This would have been an interesting topic for research, but unfortunately the space is limited and I have to discuss the practical aspects of monitoring in contemporary systems that is almost always done by logging. Having the monadic approach of carrying an object with the execution trace on each call can certainly increase the overhead of the interprocess or inter-machine communication, but saves a lot of time in stitching different pieces of information together.

Let's list the naive approaches to debugging that everyone who needed to find a bug in the code tried:

- Analyzing program output, particularly logs produced by simple print statements or built-in logback, java.util.logging, log4j, or the slf4j façade
- Attaching a (remote) debugger
- Monitoring CPU, disk I/O, memory (to resolve higher level resource-utilization issues)

More or less, all these approaches fail if we have a multithreaded or distributed system – and Scala is inherently multithreaded as Spark is inherently distributed. Collecting logs over a set of nodes is not scalable (even though a few successful commercial systems exist that do this). Attaching a remote debugger is not always possible due to security and network restrictions. Remote debugging can also induce substantial overhead and interfere with the program execution, particularly for ones that use synchronization. Setting the debug level to the DEBUG or TRACE level helps sometimes, but leaves you at the mercy of the developer who may or may not have thought of a particular corner case you are dealing with right at the moment. The approach we take in this book is to open a servlet with enough information to glean into program execution and application methods real-time, as much as it is possible with the current state of Scala and Scalatra.

Enough about the overall issues of debugging the program execution. Monitoring is somewhat different, as it is concerned with only high-level issue identification. Intersection with issue investigation or resolution happens, but usually is outside of monitoring. In this chapter, we will cover the following topics:

- Understanding major areas for monitoring and monitoring goals
- Learning OS tools for Scala/Java monitoring to support issue identification and debugging
- Learning about MBeans and MXBeans
- Understanding model performance drift
- Understanding A/B testing

System monitoring

While there are other types of monitoring dealing specifically with ML-targeted tasks, such as monitoring the performance of the models, let me start with basic system monitoring. Traditionally, system monitoring is a subject of operating system maintenance, but it is becoming a vital component of any complex application, specifically running over a set of distributed workstations. The primary components of the OS are CPU, disk, memory, network, and energy on battery-powered machines. The traditional OS-like tools for monitoring system performance are provided in the following table. We limit them to Linux tools as this is the platform for most Scala applications, even though other OS vendors provide OS monitoring tools such as **Activity Monitor**. As Scala runs in Java JVM, I also added Java-specific monitoring tools that are specific to JVMs:

Area	Programs	Comments
CPU	htop, top, sar -u	top has been the most often used performance diagnostic tool, as CPU and memory have been the most constraint resources. With the advent of distributed programming, network and disk tend to be the most constraint.
Disk	iostat, sar -d, lsof	The number of open files, provided by lsof, is often a constraining resource as many big data applications and daemons tend to keep multiple files open.
Memory	top, free, vmstat, sar -r	Memory is used by OS in multiple ways, for example to maintain disk I/O buffers so that having extra buffered and cached memory helps performance.

Area	Programs	Comments
Network	ifconfig, netstat, tcpdump, nettop, iftop, nmap	Network is how the distributed systems talk and is an important OS component. From the application point of view, watch for errors, collisions, and dropped packets as an indicator of problems.
Energy	powerstat	While power consumption is traditionally not a part of OS monitoring, it is nevertheless a shared resource, which recently became one of the major costs for maintaining a working system.
Java	jconsole, jinfo, jcmd, jmc	All these tools allow you to examine configuration and run-time properties of an application. Java Mission Control (JMC) is shipped with JDK starting with version 7u40.

Table 10.1. Common Linux OS monitoring tools

In many cases, the tools are redundant. For example, the CPU and memory information can be obtained with `top`, `sar`, and `jmc` commands.

There are a few tools for collecting this information over a set of distributed nodes. Ganglia is a BSD-licensed scalable distributed monitoring system (<http://ganglia.info>). It is based on a hierarchical design and is very careful about data structure and algorithm designs. It is known to scale to 10,000s of nodes. It consists of a gmetad daemon that collects information from multiple hosts and presents it in a web interface, and gmond daemons running on each individual host. The communication happens on the 8649 port by default, which spells Unix. By default, gmond sends information about CPU, memory, and network, but multiple plugins exist for other metrics (or can be created). Gmetad can aggregate the information and pass it up the hierarchy chain to another gmetad daemon. Finally, the data is presented in a Ganglia web interface.

Graphite is another monitoring tool that stores numeric time-series data and renders graphs of this data on demand. The web app provides a `/render` endpoint to generate graphs and retrieve raw data via a RESTful API. Graphite has a pluggable backend (although it has its own default implementation). Most of the modern metrics implementations, including scala-metrics used in this chapter, support sending data to Graphite.

Process monitoring

The tools described in the previous section are not application-specific. For a long-running process, it often necessary to provide information about the internal state to either a monitoring or graphing solution such as Ganglia or Graphite, or just display it in a servlet. Most of these solutions are read-only, but in some cases, the commands give the control to the users to modify the state, such as log levels, or to trigger garbage collection.

Monitoring, in general is supposed to do the following:

- Provide high-level information about program execution and application-specific metrics
- Potentially, perform health-checks for critical components
- Might incorporate alerting and thresholding on some critical metrics

I have also seen monitoring to include update operations to either update the logging parameters or test components, such as trigger model scoring with predefined parameters. The latter can be considered as a part of parameterized health check.

Let's see how it works on the example of a simple `Hello World` web application that accepts REST-like requests and assigns a unique ID for different users written in the Scalatra framework (<http://scalatra.org>), a lightweight web-application development framework in Scala. The application is supposed to respond to CRUD HTTP requests to create a unique numeric ID for a user. To implement the service in Scalatra, we need just to provide a `Scalate` template. The full documentation can be found at <http://scalatra.org/2.4/guides/views/scalate.html>, the source code is provided with the book and can be found in `chapter10` subdirectory:

```
class SimpleServlet extends Servlet {
    val logger = LoggerFactory.getLogger(getClass)
    var hwCounter: Long = 0L
    val hwLookup: scala.collection.mutable.Map[String, Long] =
        scala.collection.mutable.Map()
    val defaultName = "Stranger"
    def response(name: String, id: Long) = { "Hello %s! Your id
        should be %d.".format(if (name.length > 0) name else
        defaultName, id) }
    get("/hw/:name") {
        val name = params("name")
        val startTime = System.nanoTime
        val retVal = response(name, synchronized { hwLookup.get(name)
            match { case Some(id) => id; case _ => hwLookup += name -> {
                hwCounter += 1; hwCounter } ; hwCounter } })
    }
}
```

```
        logger.info("It took [" + name + "] " + (System.nanoTime -  
            startTime) + " " + TimeUnit.NANOSECONDS)  
        retVal  
    }  
}
```

First, the code gets the name parameter from the request (REST-like parameter parsing is also supported). Then, it checks the internal HashMap for existing entries, and if the entry does not exist, it creates a new index using a synchronized call to increment hwCounter (in a real-world application, this information should be persistent in a database such as HBase, but I'll skip this layer in this section for the purpose of simplicity). To run the application, one needs to download the code, start sbt, and type ~;jetty:stop;jetty:start to enable continuous run/compilation as in *Chapter 7, Working with Graph Algorithms*. The modifications to the file will be immediately picked up by the build tool and the jetty server will restart:

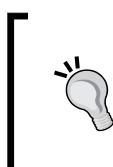
```
[akozlov@Alexanders-MacBook-Pro chapter10]$ sbt  
[info] Loading project definition from /Users/akozlov/Src/Book/ml-in-  
scala/chapter10/project  
[info] Compiling 1 Scala source to /Users/akozlov/Src/Book/ml-in-scala/  
chapter10/project/target/scala-2.10/sbt-0.13/classes...  
[info] Set current project to Advanced Model Monitoring (in build file:/  
Users/akozlov/Src/Book/ml-in-scala/chapter10/)  
> ~;jetty:stop;jetty:start  
[success] Total time: 0 s, completed May 15, 2016 12:08:31 PM  
[info] Compiling Templates in Template Directory: /Users/akozlov/Src/  
Book/ml-in-scala/chapter10/src/main/webapp/WEB-INF/templates  
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".  
SLF4J: Defaulting to no-operation (NOP) logger implementation  
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further  
details.  
[info] starting server ...  
[success] Total time: 1 s, completed May 15, 2016 12:08:32 PM  
1. Waiting for source changes... (press enter to interrupt)  
2016-05-15 12:08:32.578:INFO::main: Logging initialized @119ms  
2016-05-15 12:08:32.586:INFO:oejr.Runner:main: Runner  
2016-05-15 12:08:32.666:INFO:oejs.Server:main: jetty-9.2.1.v20140609  
2016-05-15 12:08:34.650:WARN:oeja.AnnotationConfiguration:main:  
ServletContainerInitializers: detected. Class hierarchy: empty  
2016-05-15 12:08:34.921: [main] INFO o.scalatra.servlet.ScalatraListener  
- The cycle class name from the config: ScalatraBootstrap
```

```

2016-15-05 12:08:34.973: [main] INFO o.scalatra.servlet.ScalatraListener
- Initializing life cycle class: ScalatraBootstrap
2016-15-05 12:08:35.213: [main] INFO o.f.s.servlet.ServletTemplateEngine
- Scalate template engine using working directory: /var/folders/p1/y7ygx_
4507q34vh60q115p8000gn/T/scalate-6339535024071976693-workdir
2016-05-15 12:08:35.216:INFO:oejsh.ContextHandler:main: Started o.e.j
.w.WebApplicationContext@1ef7fe8e{/file:/Users/akozlov/Src/Book/ml-in-scala/
chapter10/target/webapp/,AVAILABLE}{file:/Users/akozlov/Src/Book/ml-in-
scala/chapter10/target/webapp/}
2016-05-15 12:08:35.216:WARN:oejsh.RequestLogHandler:main: !RequestLog
2016-05-15 12:08:35.237:INFO:oejs.ServerConnector:main: Started ServerCon
nector@68df9280{HTTP/1.1}{0.0.0.0:8080}
2016-05-15 12:08:35.237:INFO:oejs.Server:main: Started @2795ms2016-15-05
12:03:52.385: [main] INFO o.f.s.servlet.ServletTemplateEngine - Scalate
template engine using working directory: /var/folders/p1/y7ygx_4507q34vh6
60q115p8000gn/T/scalate-3504767079718792844-workdir
2016-05-15 12:03:52.387:INFO:oejsh.ContextHandler:main: Started o.e.j
.w.WebApplicationContext@1ef7fe8e{/file:/Users/akozlov/Src/Book/ml-in-scala/
chapter10/target/webapp/,AVAILABLE}{file:/Users/akozlov/Src/Book/ml-in-
scala/chapter10/target/webapp/}
2016-05-15 12:03:52.388:WARN:oejsh.RequestLogHandler:main: !RequestLog
2016-05-15 12:03:52.408:INFO:oejs.ServerConnector:main: Started ServerCon
nector@68df9280{HTTP/1.1}{0.0.0.0:8080}
2016-05-15 12:03:52.408:INFO:oejs.Server:main: Started @2796mss

```

When the servlet is started on port 8080, issue a browser request:



I pre-created the project for this book, but if you want to create a Scalatra project from scratch, there is a gitter command in chapter10/bin/create_project.sh. Gitter will create a project/build.scala file with a Scala object, extending build that will set project parameters and enable the Jetty plugin for the SBT.

<http://localhost:8080/hw/Joe>.

The output should look similar to the following screenshot:



Figure 10-1: The servlet web page.

If you call the servlet with a different name, it will assign a distinct ID, which will be persistent across the lifetime of the application.

As we also enabled console logging, you will also see something similar to the following command on the console:

```
2016-15-05 13:10:06.240: [qtp1747585824-26] INFO o.a.examples.  
ServletWithMetrics - It took [Joe] 133225 NANOSECONDS
```

While retrieving and analyzing logs, which can be redirected to a file, is an option and there are multiple systems to collect, search, and analyze logs from a set of distributed servers, it is often also important to have a simple way to introspect the running code. One way to accomplish this is to create a separate template with metrics, however, Scalatra provides metrics and health support to enable basic implementations for counts, histograms, rates, and so on.

I will use the Scalatra metrics support. The `ScalatraBootstrap` class has to implement the `MetricsBootstrap` trait. The `org.scalatra.metrics.MetricsSupport` and `org.scalatra.metrics.HealthChecksSupport` traits provide templating similar to the Scalate templates, as shown in the following code.

The following is the content of the `ScalatraTemplate.scala` file:

```
import org.akozlov.examples._  
  
import javax.servlet.ServletContext  
import org.scalatra.LifeCycle  
import org.scalatra.metrics.MetricsSupportExtensions._  
import org.scalatra.metrics._  
  
class ScalatraBootstrap extends LifeCycle with MetricsBootstrap {  
    override def init(context: ServletContext) = {  
        context.mount(new ServletWithMetrics, "/")  
        context.mountMetricsAdminServlet("/admin")  
        context.mountHealthCheckServlet("/health")  
        context.installInstrumentedFilter("/*")  
    }  
}
```

The following is the content of the `ServletWithMetrics.scala` file:

```
package org.akozlov.examples  
  
import org.scalatra._  
import scalate.ScalateSupport
```

```
import org.scalatra.ScalatraServlet
import org.scalatra.metrics.{MetricsSupport, HealthChecksSupport}
import java.util.concurrent.atomic.AtomicLong
import java.util.concurrent.TimeUnit
import org.slf4j.{Logger, LoggerFactory}

class ServletWithMetrics extends Servlet with MetricsSupport with
  HealthChecksSupport {
  val logger = LoggerFactory.getLogger(getClass)
  val defaultName = "Stranger"
  var hwCounter: Long = 0L
  val hwLookup: scala.collection.mutable.Map[String, Long] =
    scala.collection.mutable.Map()  val hist =
    histogram("histogram")
  val cnt = counter("counter")
  val m = meter("meter")
  healthCheck("response", unhealthyMessage = "Ouch!") {
    response("Alex", 2) contains "Alex" }
  def response(name: String, id: Long) = { "Hello %s! Your id
    should be %d.".format(if (name.length > 0) name else
    defaultName, id) }

  get("/hw/:name") {
    cnt += 1
    val name = params("name")
    hist += name.length
    val startTime = System.nanoTime
    val retVal = response(name, synchronized { hwLookup.get(name)
      match { case Some(id) => id; case _ => hwLookup += name -> {
        hwCounter += 1; hwCounter } ; hwCounter } } )s
    val elapsedTime = System.nanoTime - startTime
    logger.info("It took [" + name + "] " + elapsedTime + " " +
      TimeUnit.NANOSECONDS)
    m.mark(1)
    retVal
  }
}
```

If you run the server again, the `http://localhost:8080/admin` page will show a set of links for operational information, as shown in the following screenshot:

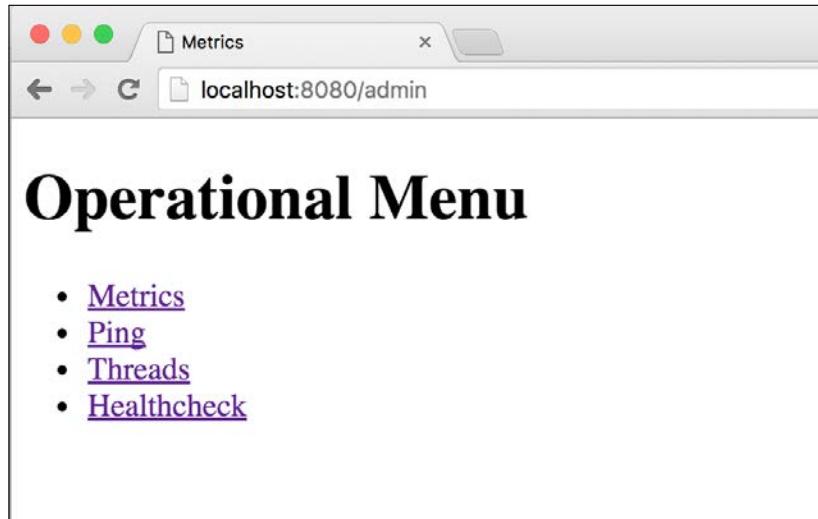
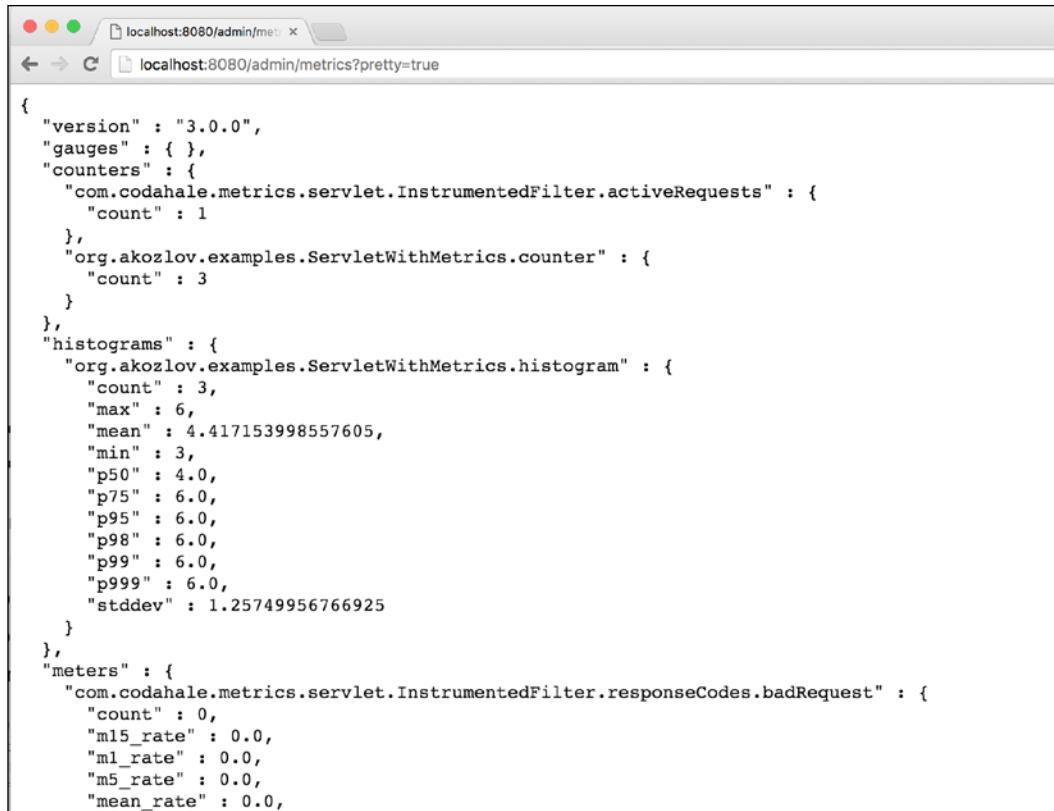


Figure 10-2: The admin servlet web page

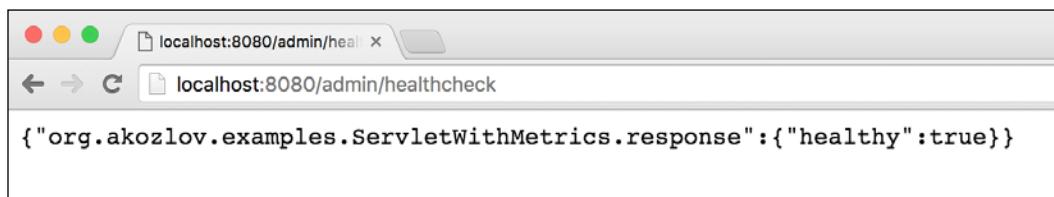
The **Metrics** link will lead to the metrics servlet depicted in *Figure 10-3*. The `org.akozlov.examples.ServletWithMetrics.counter` will have a global count of requests, and `org.akozlov.examples.ServletWithMetrics.histogram` will show the distribution of accumulated values, in this case, the name lengths. More importantly, it will compute 50, 75, 95, 98, 99, and 99.9 percentiles. The meter counter will show rates for the last 1, 5, and 15 minutes:



```
{
  "version" : "3.0.0",
  "gauges" : { },
  "counters" : {
    "com.codahale.metrics.servlet.InstrumentedFilter.activeRequests" : {
      "count" : 1
    },
    "org.akozlov.examples.ServletWithMetrics.counter" : {
      "count" : 3
    }
  },
  "histograms" : {
    "org.akozlov.examples.ServletWithMetrics.histogram" : {
      "count" : 3,
      "max" : 6,
      "mean" : 4.417153998557605,
      "min" : 3,
      "p50" : 4.0,
      "p75" : 6.0,
      "p95" : 6.0,
      "p98" : 6.0,
      "p99" : 6.0,
      "p999" : 6.0,
      "stddev" : 1.25749956766925
    }
  },
  "meters" : {
    "com.codahale.metrics.servlet.InstrumentedFilter.responseCodes.badRequest" : {
      "count" : 0,
      "m15_rate" : 0.0,
      "m1_rate" : 0.0,
      "m5_rate" : 0.0,
      "mean_rate" : 0.0,
      "max_rate" : 0.0
    }
  }
}
```

Figure 10-3: The metrics servlet web page

Finally, one can write health checks. In this case, I will just check whether the result of the response function contains the string that it has been passed as a parameter. Refer to the following *Figure 10.4*:



```
{"org.akozlov.examples.ServletWithMetrics.response": {"healthy": true}}
```

Figure 10-4: The health check servlet web page.

The metrics can be configured to report to Ganglia or Graphite data collection servers or periodically dump information into a log file.

Endpoints do not have to be read-only. One of the pre-configured components is the timer, which measures the time to complete a task—which can be used for measuring scoring performance. Let's put the code in the `ServletWithMetrics` class:

```
get("/time") {  
    val sleepTime = scala.util.Random.nextInt(1000)  
    val startTime = System.nanoTime  
    timer("timer") {  
        Thread.sleep(sleepTime)  
        Thread.sleep(sleepTime)  
        Thread.sleep(sleepTime)  
    }  
    logger.info("It took [" + sleepTime + "] " + (System.nanoTime  
        - startTime) + " " + TimeUnit.NANOSECONDS)  
    m.mark(1)  
}
```

Accessing `http://localhost:8080/time` will trigger code execution, which will be timed with a timer in metrics.

Analogously, the put operation, which can be created with the `put()` template, can be used to either adjust the run-time parameters or execute the code in-situ—which, depending on the code, might need to be secured in production environments.

JSR 110

JSR 110 is another **Java Specification Request (JSR)**, commonly known as **Java Management Extensions (JMX)**. JSR 110 specifies a number of APIs and protocols in order to be able to monitor the JVM executions remotely. A common way to access JMX Services is via the `jconsole` command that will connect to one of the local processes by default. To connect to a remote host, you need to provide the `-Dcom.sun.management.jmxremote.port=portNum` property on the Java command line. It is also advisable to enable security (SSL or password-based authentication). In practice, other monitoring tools use JMX for monitoring, as well as managing the JVM, as JMX allows callbacks to manage the system state.

You can provide your own metrics that are exposed via JMX. While Scala runs in JVM, the implementation of JMX (via MBeans) is very Java-specific, and it is not clear how well the mechanism will play with Scala. JMX Beans can certainly be exposed as a servlet in Scala though.

The JMX MBeans can usually be examined in JConsole, but we can also expose it as `/jmx servlet`, the code provided in the book repository (<https://github.com/alexvk/ml-in-scala>).

Model monitoring

We have covered basic system and application metrics. Lately, a new direction evolved for using monitoring components to monitor statistical model performance. The statistical model performance covers the following:

- How the model performance evolved over time
- When is the time to retire the model
- Model health check

Performance over time

ML models deteriorate with time, or 'age': While this process is not still well understood, the model performance tends to change with time, if even due to concept drift, where the definition of the attributes change, or the changes in the underlying dependencies. Unfortunately, model performance rarely improves, at least in my practice. Thus, it is imperative to keep track of models. One way to do this is by monitoring the metrics that the model is intended to optimize, as in many cases, we do not have a ready-labeled set of data.

In many cases, the model performance deterioration is not related directly to the quality of the statistical modeling, even though simpler models such as linear and logistic regression tend to be more stable than more complex models such as decision trees. Schema evolution or unnoticed renaming of attributes may cause the model to not perform well.

Part of model monitoring should be running the health check, where a model periodically scores either a few records or a known scored set of data.

Criteria for model retiring

A very common case in practical deployments is that data scientists come with better sets of models every few weeks. However, if this does not happen, one needs come up with a set of criteria to retire a model. As real-world traffic rarely comes with the scored data, for example, the data that is already scored, the usual way to measure model performance is via a proxy, which is the metric that the model is supposed to improve.

A/B testing

A/B testing is a specific case of controlled experiment in e-commerce setting. A/B testing is usually applied to versions of a web page where we direct completely independent subset of users to each of the versions. The dependent variable to test is usually the response rate. Unless any specific information is available about users, and in many cases, it is not unless a cookie is placed in the computer, the split is random. Often the split is based on unique userID, but this is known not to work too well across multiple devices. A/B testing is subject to the same assumptions the controlled experiments are subject to: the tests should be completely independent and the distribution of the dependent variable should be i.i.d.. Even though it is hard to imagine that all people are truly i.i.d., the A/B test has been shown to work for practical problems.

In modeling, we split the traffic to be scored into two or multiple channels to be scored by two or multiple models. Further, we need to measure the cumulative performance metric for each of the channels together with estimated variance. Usually, one of the models is treated as a baseline and is associated with the null hypothesis, and for the rest of the models, we run a t-test, comparing the ratio of the difference to the standard deviation.

Summary

This chapter described system, application, and model monitoring goals together with the existing monitoring solutions for Scala, and specifically Scalatra. Many metrics overlap with standard OS or Java monitoring, but we also discussed how to create application-specific metrics and health checks. We talked about a new emerging field of model monitoring in an ML application, where statistical models are subject to deterioration, health, and performance monitoring. I also touched on monitoring distributed systems, a topic that really deserves much more space, which unfortunately, I did not have.

This is the end of the book, but in no way is it the end of the journey. I am sure, new frameworks and applications are being written as we speak. Scala has been a pretty awesome and succinct development tool in my practice, with which I've been able to achieve results in hours instead of days, which is the case with more traditional tools, but it is yet to win the popular support, which I am pretty sure it. We just need to emphasize its advantages in the modern world of interactive analysis, complex data, and distributed processing.

Bibliography

This Learning Path is packaged keeping your journey in mind. It includes content from the following Packt products:

- *Scala for Data Science*, Pascal Bugnion
- *Scala for Machine Learning*, Patrick R. Nicolas
- *Mastering Scala Machine Learning*, Alex Kozlov



Thank you for buying
Scala: Applied Machine Learning

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Please check www.PacktPub.com for information on our titles

