Community Experience Distilled

Building a Recommendation Engine with Scala

Learn to use Scala to build a recommendation engine from scratch and empower your website users

Saleem Ansari

[PACKT] open source* PUBLISHING
community experience distilled

Building a Recommendation Engine with Scala

Learn to use Scala to build a recommendation engine from scratch and empower your website users

Saleem Ansari

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

Building a Recommendation Engine with Scala

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book 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 book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, 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 book.

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

First published: January 2016

Production reference: 1281215

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

ISBN 978-1-78528-258-4

www.packtpub.com

Credits

Author
Saleem Ansari

Reviewers
Eric Le Goff
Andrii Kravets
Loránd Szakács

Commissioning Editor
Nadeem Bagban

Acquisition Editor
Vinay Argekar

Content Development Editor
Zeeyan Pinheiro

Technical Editor
Siddhi Rane

Copy Editor
Ting Baker

Project Coordinator
Suzanne Coutinho

Proofreader
Safis Editing

Indexer
Rekha Nair

Graphics
Kirk D'Penha

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Author

Saleem Ansari is a full-stack developer with over 8 years of industry experience. He has a special interest in machine learning and information retrieval. Having implemented data ingestion and a processing pipeline in Core Java and Ruby separately, he knows the challenges faced by huge data sets in such systems. He has worked for companies such as Red Hat, Impetus Technologies, Belzabar Software, and Exzeo Software. He is also a passionate member of free and open source software (FOSS) community. He started his journey with FOSS in the year 2004. The very next year, he formed JMILUG—Linux Users Group at Jamia Millia Islamia University, New Delhi. Since then, he has been contributing to FOSS by organizing community activities and contributing code to various projects (for more information, visit `http://github.com/tuxdna`). He also mentors students about FOSS and its benefits.

In 2015, he reviewed two books related to Apache Mahout, namely *Learning Apache Mahout* and *Apache Mahout Essentials*; both the books were produced by Packt Publishing.

He blogs at `http://tuxdna.in/` and can be reached at `tuxdna@fedoraproject.org` via e-mail.

I dedicate this book to my parents.

I would like to acknowledge the amazing people who have helped me push forward while writing this book. First off, I would like to thank Vinay Argekar and Zeeyan Pinheiro from Packt Publishing, who have been of immense help and guidance right from the beginning of this book. I would like to especially thank the reviewers, Eric Le Goff and Andrii Kravets. I wouldn't have leveled up the content if I had not received their critical reviews and suggestions. So much kudos to you guys! I would like to give another special mention to Pat Ferrel from the Apache Mahout and PredictionIO project. He helped me understand the unified recommender algorithm that is mentioned in the book.

All the appreciations are due to my family and friends, who have been supportive while I was writing this book.

About the Reviewers

Eric Le Goff is a senior developer and an open source evangelist. Located in Bordeaux, France, he has more than 15 years of experience in large-scale system designs and server-side developments in both start-ups and established corporations, from digital signature solutions to financial institutions and risk management.

A former board member at the OW2 consortium (an international open source community for infrastructure), he is also a Scala enthusiast with Coursera certifications such as *Functional Programming Principles in Scala* and *Principles of Reactive Programming*.

He is also passionate about NoSql solutions (*M101J: MongoDB for Java developers* certified).

He has reviewed the book *Scala for Java Developers*, *Packt Publishing*.

First, thanks goes to my wife, Corine, who constantly supports everything that I undertake. I also would like to include all the contributors and the open source community at large. Finally, I'd like to thank Martin Odersky and his team for creating Scala.

Andrii Kravets is a highly motivated, agile-minded engineer with more than 5 years of experience in software development and software project management who wants to make the world better. He has a lot of experience with high-loaded distributed projects, big data, JVM languages, machine learning, and building complex web solutions.

He is currently making the world better at TransferWise.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

With the growth of the Internet and the widespread adoption of e-commerce and social media, a lot of new services have arrived in recent years. We shop online, we communicate online, we stay up-to-date online, and so on. We have a huge growth of data, and this has made it increasingly tough for service providers to provide only the relevant data. Recommendation engines help us provide only the relevant data to a consumer.

In this book, we will use the Scala programming language and the many tools that are available in its ecosystem, such as Apache Spark, Play Framework, Spray, Kafka, PredictionIO, to build a recommendation engine. We will reach that stage step by step with a real world dataset and a fully functional application that gives readers a hands-on experience. We have discussed the key topics in detail for readers to get started on their own. You will learn the challenges and approaches used to build a recommendation engine.

You must have some understanding of the Scala programming language, SBT, and command-line tools. An understanding of different machine learning and data processing concepts is beneficial but not required. You will learn the tools necessary for writing data-munging programs and experimenting using Scala.

What this book covers

Chapter 1, *Introduction to Scala and Machine Learning*, is a fast-paced introduction to Scala, SBT, Spark, MLlib, and other related tools. We basically set the stage for the upcoming experiments.

Chapter 2, *Data Processing Pipeline Using Scala*, explores ways to compose a data processing pipeline using Scala. We do this by taking a sample dataset from the recommendation system and then building the pipeline.

Chapter 3, *Conceptualizing an E-Commerce Store*, discusses the need for a recommendation engine. We discuss different ways in which we can present recommendations; we will also explore the architecture of our project.

Chapter 4, *Machine Learning Algorithms*, discusses some machine learning algorithms that are relevant while building different aspects of a recommender system. We will also have hands-on exercises dealing with Apache Spark's MLlib library.

Chapter 5, *Recommendation Engines and Where They Fit in?*, implements our first recommender system on a dataset for products. We will continue by populating the dataset, creating a web application, and adding recommendation pages and product/customer trends.

Chapter 6, *Collaborative Filtering versus Content-Based Recommendation Engines*, focuses on tuning the recommendations that are user-specific, rather than being global in nature. We will implement the content-based recommendation and collaborative filtering-based recommendations. Then, we will compare these two approaches.

Chapter 7, *Enhancing the User Experience*, discusses some tricks that add more spice to the overall user experience. We will add product search and recommendations listing and also discuss recommendation behavior.

Chapter 8, *Learning from User Feedback*, discusses a case study of PredictionIO. We will have a look at a hybrid recommender called unified recommender that is implemented using PredictionIO.

What you need for this book

Before you start reading this book, ensure that you have all the necessary software installed. The prerequisites for this book are as follows:

- Java: `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`
- Scala: `http://www.scala-lang.org/download/`
- SBT: `http://www.scala-sbt.org/download.html`
- MongoDB: `http://www.mongodb.org/downloads`
- Apache Spark: `https://spark.apache.org/downloads.html`

Who this book is for

This book is intended for those developers who are keen on understanding how a recommender system is built from scratch. It is assumed that you have a basic understanding of the Scala programming language and you can also handle regular data-munging tasks.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The default location for Scala and Java code is `src/main/scala` and `src/main/java` respectively."

A block of code is set as follows:

```
val tuple1 = Tuple1(1)
val tuple2 = ('a', 1) // can also be defined: ('a' -> 1)
val tuple3 = ('a', 1, "name")
```

Any command-line input or output is written as follows:

```
$ cd path/to/spark-1.3.0
$ mvn -DskipTests clean package
$ sbin/start-master.sh
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Check out the **Customer Reviews** and **Most Helpful Customer Reviews** sections."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, 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 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.

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 books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Introduction to Scala and Machine Learning

In this world of ever-growing data, we need to be quick at prototyping and at the same time we need to have a system that can handle the challenges of scalability. Scala offers a good balance between productivity and performance. In this chapter we will explore the tools, set them up and become familiar with them. In short, we will be setting the stage for your recommendation engine project. We will cover the following points:

- Setting up Scala, SBT, and Apache Spark
- Giving a quick introduction to Scala
- Discussing machine learning and recommendation engine jargon

Setting up Scala, SBT, and Apache Spark

Scala and SBT setup varies across platforms (Linux, Unix, Window, and Mac), therefore we will redirect you to the respective locations for further instructions. Please ensure you have the required versions you will need in the following steps.

For installing Scala:

1. You will need to have JDK (Java 7) already installed.
2. Next download Scala version 2.11.x: `http://www.scala-lang.org/download/`.
3. And follow the steps here: `http://www.scala-lang.org/download/install.html`.

For setting up SBT, follow the steps here: `http://www.scala-sbt.org/release/tutorial/Setup.html`.

For setting up Apache Spark, first download `spark-1.3.0.tgz` from here: `https://spark.apache.org/downloads.html`.

You can also install the popular GUI development environments mentioned in the following list. However, that is your choice:

- ScalaIDE: `http://scala-ide.org/docs/user/gettingstarted.html`
- IntelliJ: `https://www.jetbrains.com/idea/help/creating-and-running-your-scala-application.html`

Let's continue with our Apache Spark setup. Now extract the archive:

```
$ tar zxf spark-1.3.0.tgz
$ cd spark-1.3.0
```

Run Spark in local mode:

```
$ bin/spark-shell
```

A quick introduction to Scala

Scala stands for "scalable language." Scalable could be interpreted as scalability of the software application. Java developers have been programming with its nice and verbose syntax for many years now. However, compared with programming languages such as PHP, Ruby, and Python, it has often been considered to have a very restrictive syntax. While on the one hand these bunch of languages provide a very feature-rich syntax to make the code both more readable and concise, they are just not up to the power of JVM. On the other hand, Java language in itself didn't catch up with the speed of evolution in modern programming languages. One could debate over whether it is a good thing or a bad thing, but one thing is very clear "developers need a language that is concise, expressive, modern and runs fast!" Scala emerged as a language that fills this sweet spot between the power of JVM and expressiveness of modern programming languages.

Back again to the "scalable" part of the definition: scalable in this context means that Scala allows a developer to scale the "way code is written." A Scala program may begin as a quick script, then add more features, then be broken up into packages and classes. Because Scala supports both the object-oriented paradigm and the functional paradigm, it is possible for a developer to write really compact code. This allows a developer to focus more on the core logic, rather than on the boiler-plate code. Scala is a wonderful language to program with because it has:

- Operator overloading
- Generics
- Functional operators: map, filter, fold, and reduce
- Immutable collections and data structures
- Intuitive pattern matching
- Support for writing **domain-specific Languages** (**DSL**) via parser combinators

In a nutshell, Scala is a double-edged sword. On one hand you can write very elegant and concise code, and on the other hand you can write code that is too cryptic that no other developer can easily understand. That is the balance one has to maintain. Let's take a look at our first Scala application:

```
package chapter01

object HelloWorld {
  def main(args: Array[String]) {
    println("Hello World!")
  }
}
```

Here, `HelloWorld` is defined as an object, which means that in a JVM, there will be only one copy of `HelloWorld`. We can also consider it as a singleton object. `HelloWorld` contains a single method named main with parameter `args`. Note that the parameter type is defined after the parameter name. Essentially, it translates to Java's `public static void main method`. Just like in a Java application, the `main` method is the entry point, which is also the case with a Scala object with a `main` method defined in the preceding screenshot. This program will output:

```
Hello World!
```

Next up is a Scala application named `CoreConcepts`.

```
package chapter01

import scala.annotation.tailrec

object CoreConcepts {
  def main(args: Array[String]) {
    val x: Double = 10
    val y = 20.5
    val s = sum(x, y)
    println(f"$x%05.2f + $y%05.2f = $s%05.2f")
    println(s"Factorial(5) = ${factorial(5)}")
  }

  def sum(a: Double, b: Double) = a + b

  @tailrec
  def factorial(n: Int, partialProduct: Int = 1): Int = {
    if (n <= 1) partialProduct
    else factorial(n - 1, partialProduct * n)
  }
}
```

Things to notice in this code are:

- The first line defines a package.
- We have defined `x`, `y`, and `s` as `val`, which expands to values. We haven't specified the data type for `y` and `s`. Although we haven't specified the types, the Scala compiler inferred it using its type-inference system. So we save time by typing less.
- Unlike Java, there are no semicolons. If you prefer, you can still use semicolons, but they are optional.
- `def` defines a method, so apart from `main`, we have two other methods defined: `sum` and `factorial`
- We haven't specified any return type for the `sum` method. But you can check in an IDE that it returns `Double`. Again, Scala's type inference saves us from specifying that.
- There is no `return` statement. We don't need it. The value of the last expression is returned by default. Notice that some values are returned by both sum and factorial functions.
- Notice `@tailrec` (annotation for tail recursion), which tells the Scala compiler to optimize the recursive factorial method into a `for` loop. You do not need to worry about recursion or tail recursion for now. This is just an illustration of some Scala features.

- Also, look at strings in `println` statements. Take notice of `f` and `s` prefixes to those strings. Prefixing a string with `s` or `f` enables string interpolation. Prefixing with `f` also enables a format specifier for data types.

There is a lot going on in this small example. Take some time to go through each statement and refer to additional material if you need to.

Case classes

In the next example, we see how a `case` class can help drastically shorten our code size. If we were to store the name and age of an employee, we might need to create a Java bean with two members, one each for name and age. Then we would need to add two getters and two setters for each as well. However, all this and more could be achieved with just a single line of Scala code as in the next example. Check for a `case class Employee` statement:

```
package chapter01

import scala.util.Random

object ExampleCaseClasses {

  case class Employee(name: String, age: Int)

  def main(args: Array[String]) {
    val NUM_EMPLOYEES = 5
    val firstNames = List("Bruce", "Great", "The", "Jackie")
    val lastNames = List("Lee", "Khali", "Rock", "Chan")
    val employees = (0 until NUM_EMPLOYEES) map { i =>
      val first = Random.shuffle(firstNames).head
      val last = Random.shuffle(lastNames).head
      val fullName = s"$last, $first"
      val age = 20 + Random.nextInt(40)
      Employee(fullName, age)
    }

    employees foreach println

    val hasLee = """(Lee).*""".r
    for (employee <- employees) {
      employee match {
        case Employee(hasLee(x), age) => println("Found a Lee!")
        case _ => // Do nothing
      }
    }
  }
}
```

Other things to note in this example:

- Use of a range (0 until `NUM_EMPLOYEES`) coupled with the `map` functional operator, instead of a loop to construct a collection of employees.
- We use `foreach` loop to print all the employees. Technically it's not a loop, but, for simplicity, we can think of its behavior as a simple loop.
- We also use regular expression pattern matching on the full name of all the employees. This is done using a `for` loop. Take a closer look at the loop structure. The left arrow is a generator syntax, quite similar to a `foreach` loop in Java. Here, we don't have to specify any types in a `foreach` loop.

Tuples

Other than classes and case classes, Scala also provides another basic type of data container called **tuples**. Tuples are typed, fixed-size, immutable list of values. By typed we mean that each of the members has a type, and this list is of fixed size. Also once we create a tuple, we can't change its member values, that is, tuples are immutable (think of Python tuples). Each of the member values are retrieved using an underscore followed by a number (the first member is numbered as `._1`). There cannot be a zero member tuple.

Here is an example:

```
package chapter01

object ExampleTuples {
  def main(args: Array[String]) {
    val tuple1 = Tuple1(1)
    val tuple2 = ('a', 1) // can also be defined: ('a' -> 1)
    val tuple3 = ('a', 1, "name")

    // Access tuple members by underscore followed by
    // member index starting with 1
    val a = tuple1._1 // res0: Int = 1
    val b = tuple2._2 // res1: Int = 1
    val c = tuple3._1 // res2: Char = a
    val d = tuple3._3 // res3: String = name

  }
}
```

Scala REPL

Scala also has a shell that is also called a **REPL** (**read-eval-print loop**) (think of Python's shell, or Ruby IRB, or Perl's PDB). By just typing in Scala, we can access the REPL:

```
$ scala
Welcome to Scala version 2.11.2 (Java HotSpot(TM) 64-Bit Server VM, Java
1.7.0_76).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello World!")
Hello World!

scala> 3 + 4
res1: Int = 7

scala>
```

Scala REPL is very powerful and convenient to use. We just set up proper classpaths and then we can import all the Java classes we need to play with, right within the Scala REPL. Notice how we use Java's HashMap in Scala REPL:

```
scala> val m = new java.util.HashMap[String, String]
m: java.util.HashMap[String,String] = {}

scala> m.put("Language", "Scala")
res0: String = null

scala> m.put("Creator", "Martin Odersky")
res1: String = null

scala> m
res2: java.util.HashMap[String,String] = {Creator=Martin Odersky,
Language=Scala}

scala> import scala.collection.JavaConversions._
import scala.collection.JavaConversions._
```

```
scala> m.toMap
res3: scala.collection.immutable.Map[String,String] = Map(Creator ->
Martin Odersky, Language -> Scala)
```

That is complete Java interoperability. To give you an idea of how Scala looks, we have shown you the very basic Scala code. Additional reading is required to become comfortable with Scala. Hold on to any decent book on Scala, that covers Scala in more detail, and you are good to go. Next we will see how to create a sample SBT project.

SBT – Scala Build Tool

SBT is arguably the most common build tool for building and managing a Scala project and its dependencies. An SBT project consists of a `build.sbt` file or a `Build.scala` file at the project root, and optionally a project/folder with additional configuration files. However, for most parts it would suffice to have only `build.sbt`.

Here is what a sample `build.sbt` file looks like:

```
$ cat build.sbt
name := "BuildingScalaRecommendationEngine"

scalaVersion :="2.10.2"

version :="1.0"

libraryDependencies += "org.apache.spark" % "spark-mllib_2.10" %
"1.3.0"
```

Well yes, that's pretty much it. One thing to be careful with is to keep a single line spacing between the statements. We have defined a project name, project version, Scala version, and we have a complete Scala project up and running for development. Also, you need to create source code folders inside the project folder. The default location for Scala and Java code is `src/main/scala` and `src/main/java` respectively.

SBT has its own shell. Just type `sbt` and you will see:

```
$ sbt
[info] Loading global plugins from /home/tuxdna/.sbt/0.13/plugins
[info] Set current project to BuildingScalaRecommendationEngine (in build
file:/tmp/BuildingScalaRecommendationEngine/code/)
>
```

There you can type `help` to see more information on the different commands. For our purposes, we only need these commands:

- `clean`: Cleans the project
- `compile`: Compiles the Scala as well as Java code
- `package`: Creates a project archive file
- `console`: Opens a Scala console (REPL) with project dependencies already set up in the classpath

Apache Spark

Given that you have already set up Apache Spark, run Spark in local mode like so:

```
$ bin/spark-shell
```

This will give you Scala like REPL, which actually it is, but the interpreter has done Spark specific initialization for us. Let's see a simple Spark program:

```
scala> sc.textFile("README.md").flatMap( l => l.split(" ").map( _ -> 1)).
reduceByKey(_ + _).sortBy(_._2, false).take(5)

res0: Array[(String, Int)] = Array(("",158), (the,28), (to,19),
(Spark,18), (and,17))
```

The preceding code is a word count program that outputs the top five most occurring words in the file `README.md`.

Some things to note in the preceding example are:

- `sc` is the `SparkContext` that is created by the Spark shell by default
- `flatMap`, `map`, and so on are the functional programming constructs offered by Scala
- Everything here is Scala code, which means you do all the processing and database-like querying using only one language, that is, Scala

Setting up a standalone Apache Spark cluster

In the previous Spark example, we ran Spark in local mode. However, for a full blown application we would run our code on multiple machines, that is, a Spark cluster. We can also set up a single machine cluster to achieve the same effect. To do so, ensure that you have Maven version 3 installed, and then set these environment variables:

```
$ export MAVEN_OPTS="-Xmx2g -XX:MaxPermSize=512M
-XX:ReservedCodeCacheSize=512m"
```

Go to the folder where we extracted the `spark-1.3.0` archive earlier. Now build Apache Spark:

```
$ cd path/to/spark-1.3.0
$ mvn -DskipTests clean package
$ sbin/start-master.sh
```

In the output of the preceding command, you will be given a file with the extension `.out`. This file contains a string like this:

```
Starting Spark master at spark://matrix02:7077
```

Here `spark://hostname:port` part is the Spark instance you would connect to for your Spark programs. In this case, it is `spark://matrix02:7077`.

If you are using a Windows machine, you can check for instructions in this Stack Overflow question:

`https://stackoverflow.com/questions/25721781/running-apache-spark-on-windows-7`

Or, create a GNU/Linux virtual machine, and follow the steps mentioned earlier.

Apache Spark – MLlib

Starting with version 1.2.0, Spark comes packaged with a machine learning routines library called **MLlib**. Here is an example of KMeans clustering using MLlib on the Iris dataset:

```
package chapter01
import scala.collection.JavaConversions._
import scala.io.Source
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.linalg.Vectors

object IrisKMeans {
  def main(args: Array[String]) {

    val appName = "IrisKMeans"
    val master = "local"
    val conf = new
    SparkConf().setAppName(appName).setMaster(master)
```

```
    val sc = new SparkContext(conf)

    println("Loading iris data from URL...")
    val url = "https://archive.ics.uci.edu/ml/machine-learning-
    databases/iris/iris.data"
    val src = Source.fromURL(url).getLines.filter(_.size >
    0).toList
    val textData = sc.parallelize(src)
    val parsedData = textData
      .map(_.split(",").dropRight(1).map(_.toDouble))
      .map(Vectors.dense(_)).cache()

    val numClusters = 3
    val numIterations = 20
    val clusters = KMeans.train(parsedData, numClusters,
    numIterations)

    // Evaluate clustering by computing Within Set Sum of Squared
    Errors
    val WSSSE = clusters.computeCost(parsedData)
    println("Within Set Sum of Squared Errors = " + WSSSE)
  }
}
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. 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.

Run the preceding program from your IDE or use the following command from the project root folder:

```
$ sbt 'run-main chapter01.IrisKMeans'  2>/dev/null
```

Output:

```
Loading iris data from URL ...
Within Set Sum of Squared Errors = 78.94084142614648
```

In this example, we basically downloaded the Iris dataset, loaded it into Spark, and performed KMeans clustering. Finally, we displayed a WSSE metric, which indicates how much spherical and distinct, the *K* clusters were formed (for details read: `https://en.wikipedia.org/wiki/K-means_clustering#Description`). Don't worry if you don't understand what that metric means because this example is only to get your Spark environment up and running. Next we discuss some machine learning and recommendation engine jargon.

Machine learning and recommendation engines

Machine learning is a study of methods that enable computers to learn with experience. More formally, a learning task is defined as:

> *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*
>
> *– Machine Learning, Tom M. Mitchell*

In simple words, let us say that *T* is a task of playing chess. Then after you have played some games you have gained experience, which we call *E*. However, it is the performance *P* that will measure how well you have learned to play chess. A simple observation of this definition would indicate that *P* should be non-decreasing for our learning strategy to be worth investing our time. For us human beings, this learning process is natural. However, when we want to make computers learn, then it is a different game altogether. Machine learning tools and techniques allow us to enable computers to learn such strategies. Typically, we want to learn to a point where no more further learning is feasible.

In general, there are three broad categories in which we can segregate different machine learning techniques:

- **Supervised learning**: When we can make computer learn from historical data
- **Unsupervised learning**: When we just want to understand the structure of data we are presented with
- **Reinforcement learning**: When we want to maximize a reward in the learning process

Covering all the machine learning techniques is beyond the scope of this book; however we will cover some of the techniques in *Chapter 4*, *Machine Learning Algorithms*, specifically those that are relevant to creating a recommendation engine.

Our objective with this book is to build a recommendation engine using Scala. How does machine learning fit in here?

A recommendation engine is also called a **recommender system**. Given a plethora of information that is present in an information retrieval system, the task of a recommender system is to show to a user, only what is relevant. Let's take a common example of a e-commerce store. You log on to an e-commerce site, and search for a headphone. The website has thousands of headphones in its inventory. Which ones would the website show you? That's one kind of decision a recommender system would help a website with. Of course, our discussion would involve the ways and means of making that decision, so as to both keep the customer engaged, and also increase sales at the same time.

The concept of "relevant" data brings an implicit connection between the actor (the user browsing the site), and object (the headphone). We cannot just magically know how an object is relevant to an actor. We have to find some measure of relevance. To find that relevance, we need to have some data to back that relevance factor. For example:

- A popular headphone may be relevant
- A headphone that is cheaper as well as of high quality may be relevant
- A headphone owned by a user similar to the user logged in may be relevant as well
- A headphone similar to one that a user already browsed may be relevant

Do you see a pattern here? We need to make lots of decisions to come up with a good recommendation. This is where machine learning algorithms help us with understanding the data better.

Recommendation systems are not just limited to e-commerce sites. They are present at so many places we often see:

- Facebook friends suggestions
- You may know XYZ person on LinkedIn
- News you may be interested in
- Advertisements you see on your phones (ad placement)
- Movies you may want to watch (think of Netflix)
- Music you may want to listen (think of Spotify)
- Places you may love to visit
- Food you may relish at some restaurant and so on, the possibilities are endless...

We just saw how many applications are possible with recommendation engines. However, it is machine learning that helps us make our recommendations even better. Machine learning is an inter-disciplinary field that integrates scientific techniques from many fields such as information theory, cognitive sciences, mathematics, artificial intelligence to name a few. For now, let's conclude this section with the statement:

> *"Machine learning is the most exciting field of all the computer sciences. Sometimes I actually think that machine learning is not only the most exciting thing in computer science, but also the most exciting thing in all of human endeavor."*
>
> *– Andrew Ng, Associate Professor at Stanford and Chief Scientist of Baidu*

Well, if you have read this far, you are already part of this exciting field!

Summary

In this chapter, we learned how to set up Scala, SBT, and Apache Spark. We also had some hands-on experience with Scala, Spark shell, and MLlib. Next, we briefly discussed what a recommendation engine does and how machine learning is related to it. In the next chapter, we will see how to construct a data-ingestion pipeline using the tools we just learned. This will set the foundation for gathering the data for our recommendation engine, and enable us to make good recommendations for the user.

2
Data Processing Pipeline Using Scala

In *Chapter 1*, *Introduction to Scala and Machine Learning*, we gained some idea about Scala, Apache Spark, and machine learning. In this chapter, we will explore ways to compose a data processing pipeline using Scala. In particular, we will discuss:

- Entree—A sample dataset for recommendation systems
- ETL—extract transform load
- Extraction and transformation for machine learning
- Setting up MongoDB and Apache Kafka
- Data processing pipeline for Entree

And then hopefully, we will be able to compose different components of the processing pipeline.

Entree – a sample dataset for recommendation systems

In this chapter, we will focus our discussion based on a dataset that is apt for recommendation engines. We have selected the Entree dataset for this chapter. This dataset can be found at: `https://archive.ics.uci.edu/ml/machine-learning-databases/entree-mld/entree.data.html`. We have selected this dataset because:

- It is one of the classic datasets for recommendation systems (specifically case-based recommendation systems)
- The data is well formed, and it can be processed by any text processing tool
- The data has missing values, which makes the problem even more interesting

Let's take a look at the data that we are presented with. First download it:

```
$ wget -c https://archive.ics.uci.edu/ml/machine-learning-databases/
entree-mld/entree_data.tar.gz
$ tar zxf entree_data.tar.gz
$ ls -F entree
data/  README  session/
```

On a Windows, machine you have two choices: download and unzip the file manually, or use a Linux-based VM and follow the preceding steps.

So there are two folders `data/` and `session/` along with the `README` file. Your first task would be to explore different files in the dataset, and get a feel of its structure. If you read the `entree/data/README` file, you will find there are restaurants in eight cities: Atlanta, Boston, Chicago, Los Angeles, New Orleans, New York, San Francisco, and Washington DC.

Each of the restaurants has a few of the features such as bakeries, Burmese, Chinese, cafeterias, diners, early dining, and so on. These features of a particular restaurant can be encoded in terms of boolean values, that is, if a restaurant has early dining we can represent it as true if present, and false, if not. Since there are 257 features in total, we can represent each restaurant in terms of 257 Boolean values. Or, simply as an array of numbers (either 0 for false, or 1 for true):

```
val r1features = Array(0, 1, 1, 1, 0, 0, 0,...)
```

Note that the Entree system generates recommendations for Chicago city restaurants only. Additionally, we have historical data of different users. These users are identified by their IP addresses / domain names. These are stored in `session.YEAR-QUARTER` text files in `entree/session/` folder. Each row of this file has different fields, which are well described in the dataset description. Take some time to go through the description, and you will notice the following fields:

- Date/time
- IP address / domain name
- Entry point (a restaurant code)
- Some more restaurants navigated by the user during a session. These are all Chicago restaurants
- End point (a restaurant code in Chicago)

Of these fields, an entry point value of 0, and end point value of -1 means missing data. The end point value is assumed to be a restaurant that a user actually liked. In a sense this is a good indicator of a user preference to a particular restaurant, and will serve as a good data point for making potentially sensible recommendations.

Now we need to answer the following questions:

- What kind of learning algorithm should we run on this dataset to make good recommendations?
- What data representation does this algorithm expect?
- How will the data be fed into this algorithm over time?

Our focus for now should be on building a data processing pipeline. Therefore, for the sake of simplicity let us keep our discussion to recommendations based on popularity over some time period or a window. We will cover many different recommendation algorithms in the next chapters.

For *Chapter 1*, *Introduction to Scala and Machine Learning*, **learning algorithm**, we simply choose a learning algorithm based on popularity of a restaurant.

For this chapter, **data representation**, let's use the historical data to infer popularity of a restaurant. For this, we can utilize the session history provided in the dataset. In a typical browsing session, a user starts from any starting restaurant, then moves on to the next and so on until finally he/she finds one at the end (that is, in the Chicago area). Therefore, the navigation sequence gives us a base for counting the restaurant visits for data representation.

For *Chapter 3*, *Conceptualizing an E-Commerce Store*, **feeding data to algorithm**, we will consider a stream based approach. If our data is static, and it won't change for a long time, we can simply process everything in bulk and store the results. However, for an online system like Entree, this is not true and we are presented with more challenges. How should we feed data then?

Generally, there are two approaches we can take in this scenario:

- **Push**: Someone or some system, notifies our recommendation engine with the new set of data, or new transactions
- **Pull**: The recommendation engine asks for the data periodically

Of course, we would not want the recommender system to wait for an infinite amount of time to accumulate as much data as it can. So, a window-based approach makes sense here. In this case, we wait for some time period while new data is arriving. Once this, time period has elapsed we process this chunk of data and store the results (or move it to some appropriate place).

Now in practical situations, there is too much data for a single machine to handle. Typically, a set of many machine nodes form a distributed system, which together as a cluster process this huge data. We can assume that for small number of nodes in a distributed system, the push strategy performs better. Read this paper to convince yourselves: *A Fair Comparison of Pull and Push Strategies in Large Distributed Networks* at `http://www.pats.ua.ac.be/content/publications/2014/Pull_push_RR_extended.pdf`.

Next, we perform some simple analysis on the Entree dataset.

Data analysis of the Entree dataset

It goes a long way to know the dataset better. So let's write some Scala code to find out which cities have what kind of restaurants. First we count the number of restaurants in each city. Next, we find the kind of restaurant features most likely to be found in different cities.

```
package chapter02
import java.io.File
import scala.collection.immutable.SortedMap

object Stats {
  def main(args: Array[String]) {
    val entreeDataPath = args(0)
    val config = DataConfig(entreeDataPath)
    val featuresMap = Utilities.loadFeaturesMap(config.featuresFile)
    val restaurants = config.locations.flatMap { location =>
      Utilities.loadLocationData(new File(s"${config.dataPath}/" + location))
    }
    println("Cities and their restaurant count")
    println("-" * 50)
    val grouped = restaurants.groupBy(_.city).map { case (k, v) => k -> v.size }
    val citiesSorted = grouped.keys.toArray.sorted
    citiesSorted foreach { city =>
      val count = grouped(city)
      println(f"$city%20s has $count%6d restaurants")
    }
    println()

    // top 5 popular features in each city
    val cityAndTop5 = restaurants.groupBy(_.city).map { r =>
      val (city, restaurantList) = r
      val partialSum = restaurantList.flatMap(_.features).map(_ -> 1)
      val allsum = partialSum.groupBy(_._1).map {
        case (k, v) => k -> v.map(_._2).sum
      }.toSeq.sortBy(_._2).reverse
      val top5 = allsum.take(5)
      city -> top5
    }

    citiesSorted.foreach { city =>
      val top5 = cityAndTop5(city)
      println(s"City: $city")
      println("-" * 50)
      top5 foreach { entry =>
        val (f, count) = entry
        println(f"${featuresMap(f)}%25s at $count%6d restaurants")
      }
      println()
    }
  }
}
```

Here is the output of the analysis:

```
$ sbt 'run-main chapter02.Stats /home/tuxdna/work/packt/dataset/entree'
[info] Running chapter02.Stats /home/tuxdna/work/packt/dataset/entree
Cities and their restaurant count
-------------------------------------------------
             atlanta has    267 restaurants
              boston has    438 restaurants
             chicago has    676 restaurants
         los_angeles has    447 restaurants
         new_orleans has    327 restaurants
            new_york has   1200 restaurants
       san_francisco has    414 restaurants
       washington_dc has    391 restaurants

City: atlanta
-------------------------------------------------
            Parking/Valet at    225 restaurants
        Wheelchair Access at    199 restaurants
        Excellent Service at    164 restaurants
           Weekend Dining at    158 restaurants
          Private Parties at    155 restaurants

City: boston
-------------------------------------------------
           Weekend Dining at    532 restaurants
        Wheelchair Access at    238 restaurants
        Excellent Service at    209 restaurants
           Excellent Food at    202 restaurants
          Private Parties at    181 restaurants

City: chicago
-------------------------------------------------
           Weekend Brunch at    512 restaurants
        Excellent Service at    371 restaurants
           Excellent Food at    353 restaurants
```

```
            Parking/Valet at    328 restaurants
              Short Drive at    255 restaurants

City: los_angeles
--------------------------------------------------
           Weekend Brunch at    342 restaurants
            Weekend Lunch at    302 restaurants
        Excellent Service at    249 restaurants
      Weekend Jazz Brunch at    206 restaurants
   Warm spots by the fire at    206 restaurants

City: new_orleans
--------------------------------------------------
          Open on Mondays at    259 restaurants
          Open on Sundays at    259 restaurants
        Wheelchair Access at    151 restaurants
        Excellent Service at    147 restaurants
           Excellent Food at    143 restaurants

City: new_york
--------------------------------------------------
           Excellent Food at    612 restaurants
        Excellent Service at    576 restaurants
               Good Decor at    406 restaurants
          Excellent Decor at    404 restaurants
             Good Service at    369 restaurants

City: san_francisco
--------------------------------------------------
           Weekend Dining at    251 restaurants
        Excellent Service at    243 restaurants
        Wheelchair Access at    241 restaurants
          Private Parties at    193 restaurants
  Private Rooms Available at    193 restaurants
```

```
City: washington_dc
--------------------------------------------------
           Weekend Dining at    318 restaurants
            Parking/Valet at    279 restaurants
            Weekend Lunch at    222 restaurants
        Wheelchair Access at    216 restaurants
        Excellent Service at    210 restaurants
```

Note that in a city, for restaurants to be profitable, they must provide facilities/ features that are relevant to people in the area. From the preceding output, we can glean much information about different cities. For example, New York has almost double the number of restaurants as Chicago. While the people from New York seem to prefer excellent food and decor, Chicago people seem to prefer weekend brunch and also valet parking / short drive. This would make sense depending on the kind of conveyance people can afford in their cities. By analyzing our dataset, we find some important information, using which we can tune our system for some special cases. In fact, it could be a good recommendation to different restaurant owners, that they provide nice valet parking when in Chicago.

ETL – extract transform load

In any typical data-mining application, the data processing phase is broken into three stages: **extract**, **transform**, and **load**. This is an architectural pattern that helps in separating the three big concerns of a data mining project. The reason is simple: most of the effort is always spent in cleaning and organizing the data. Because garbage-in means garbage-out, it become essential to ensure that the data we are feeding to a learning algorithm is clean. Here are some brief descriptions of the three stages of an ETL pipeline.

Extract

At this stage, data is obtained from different data-sources. For example, there could be web-server logs residing on the filesystem, or the customer data residing on a database server, or products data residing on a separate application altogether. So, in this stage, we fetch data from all these sources.

In our case, every restaurant owner might have additionally uploaded a PDF/Word document to the Entree website. However, it is not well structured, since it is all free text. So we may need to extract data out of these files.

Transform

This stage involves transforming data into a common and consistent format. For example, the data from different sources might be in different metric units (meter versus feet, or USD versus INR, and so on). Also, the data storage format might be different (CSV, XML, JSON, and so on). Several rules and transformers are be applied in this stage to bring all the data sources into common format.

Another typical step in this stage is to perform de-duplication (or entity-resolution). The data from different sources might represent essentially the same product. However, their representation could be different. For example, a product might have only 10 features with slightly different description in one data source, but it may have only five features in another data sources.

Load

Finally, once all the data is processed and brought down into single format, it is stored into a data store. This data store could be a very sophisticated database, or simply a plain text file. By this time, the system should have ensured that the data presented to a learning algorithm is free from missing data or inconsistent data, and that the data is present in a format that is efficient to process.

Extraction and transformation for machine learning

To properly extract and transform the data, we need to first understand what kind of data we are dealing with, only then we can proceed with cleaning it. Now we will discuss in brief, the different kinds of data that are usually encountered in practice.

Types of data

To apply any algorithm to a dataset, we first need to map the data to a machine readable form. Let's discuss what kinds of basic features we will find in any dataset. There are three broad categories in which we can segregate the features: discreet, continuous, and categorical.

Discrete

The word discrete means *separate and distinct*, which essentially captures the essence of discreet features. Therefore, discrete simply means something that we can count. Or more technically, something that is countable. Note that the feature could have either finite or finite values. For example, the number of hair on your head is countable. Then the number of stars in the galaxy is countable, hence discrete.

Continuous

The word continuous means *going on or extending without interruptions or break*. A continuous data can have infinitely many values. Continuous values are not countable because we simply can't find intervals in between, therefore they are not distinct. Often we use continuous data to represent quantities such as temperature, time, and distance. Note that although time is continuous, we can represent dates as discrete values because dates are countable.

Categorical

Categorical data is simply finite discreet data. Categorical data takes a limited/fixed number of possible values. We can represent gender, grades, religions, and so on as categorical data.

Cleaning the data

We can apply several strategies to clean up the data. To make a data representation complete, we need to clean the data and then bring it into a common form. To do that we only discuss: missing data, normalization, and standardization.

Missing data

Sometimes the data presented to us will have some filed for which the data is empty. For example, in the Entree dataset, the entry point and end point are missing in some places. Here the missing data is marked properly. In modeling such data, we need to take extra care of how we handle such missing data. You could either ignore or discard it, or you could fill in some values yourselves. However, we need to think through what impact each approach will have on our results.

Case 1: We discard the data

In this case, we just don't discard one single feature, but the whole instance of features because, for an instance, the features are all related. For example, if we were to discard a user's session which does not have an end point, then we have to discard the whole session. And in doing so, we are likely have less data to feed to our learning algorithm.

Case 2: We fill in the data

In this case, we fill in some default or fill with an estimated value. Although we will not lack data, there will be some influence because of the data we fill in. When the data is numeric we could just use an average/mean. When the data is categorical, we could just designate an additional category, which explicitly means missing data. That is the decision one has to make when modeling the data.

Normalization

Normalization is the process of rescaling the observations to a notationally common scale. Why would we want to do that? An intuition is that, suppose you represent the age of a person as integer, and it ranges from say 0 to 100. Now, there could be another value in your data that represents salary. The salary could be represented as integer, with values ranging from 0 to 100,000. Salary also depends on the currency and job type. Now the point is, our machine learning model, or the underlying formula, would be more influenced by value of salary, rather than age because there is a magnitude of difference between their representation. To overcome this difficulty we use normalization. There are several approaches to normalization, for example the most common being scaling to the range [0,1]. The formula for such a normalization can be written as:

$$x_{new} = \frac{(x - x_{\min})}{(x_{\max} - x_{\min})}$$

Where $x_{\max}$ and $x_{\min}$ are the maximum and minimum of the feature values respectively.

We need to be very careful using such formulas though, because, in the case of missing or inadequate data, the formula many not work. For example, if all values are same, this could lead to a division by zero error.

Standardization

Standardization is another normalization approach. Standardization essentially represents how much a data point, is above or below the mean. In fact, this value represents the number of standard deviations a data point is above or below the mean value. A positive value means above, and a negative value means below the mean. This is also called as **z-score** or **normal score**, the formula for which is as follows:

$$z = \frac{(x - \mu)}{\sigma}$$

Where μ is the mean, and σ is the standard deviation, and z represents the z-score.

Standardization, or z-score can be intuitively understood like this. Consider, you have some test data that you know is a good representative sample of the actual data you will encounter. For example, historical data of student grades in a class. Now you can standardize all the attributes of this data using z-score. This will help in identifying which of the actual data encountered later, will be more or less beyond the norm (or beyond the usual grade pattern of the students). Of course, this assumes that there is some form of Gaussian/normal distribution in the given dataset.

Now that we have briefly discussed different approaches to clean and process our data, it's time to set the stage for storage of data.

Setting up MongoDB and Apache Kafka

MongoDB is a document-oriented, schema-free NoSQL database. Document oriented meaning that each of the storage units in MongoDB is a document. Think of a document being equivalent to a row in a table of a MySQL database. The documents are organized into collections, for the sake of simplifying database-structures. However, we can still keep different kinds of documents into same collection. Why? Because it is schema-free. No two documents need to have exactly the same fields. More so, the fields could be nested. So we can say it is a schema-free, document-oriented NoSQL database that even supports nested data-structures. This is pretty neat, and gives great power and flexibility to a developer.

Setting up MongoDB

Here in this chapter, we will use Scala binding for MongoDB. You should be able to install MongoDB on your system. Please look at different installation guides as per your OS and machine type: `http://docs.mongodb.org/manual/installation/`.

Let's now confirm if MongoDB is up and running:

```
$ mongo
MongoDB shell version: 2.6.3
connecting to: test
> show dbs
admin  (empty)
local  0.078GB
```

Once we have MongoDB installed on the machine, we configure `casbah` (an official Scala binding for MongoDB). Add the following line to `build.sbt` for configuring `casbah`:

```
libraryDependencies += "org.mongodb" %% "casbah" % "2.8.1"
```

MongoDB will serve as a persistent structured datastore. However, for streaming tasks we need a queuing mechanism. We could use any of Apache Kafka, RabbitMQ, Apache ActiveMQ, and so on or just roll our own. We simply chose Apache Kafka for this purpose.

Setting up Apache Kafka

Apache Kafka is an efficient distributed and persistent, producer-consumer queuing software. Let's set it up. First download the tarball, and extract it. Then start the ZooKeper in one terminal, and in another terminal start the `kafka` server as shown in the following commands:

```
$ wget -c http://apache.mesi.com.ar/kafka/0.8.2.0/kafka_2.10-0.8.2.0.tgz
$ tar zxf kafka_2.10-0.8.2.0.tgz
# Start zookeper
$ bin/zookeeper-server-start.sh config/zookeeper.properties
# In a separate terminal, start kafka server
$ bin/kafka-server-start.sh config/server.properties
```

Did you notice we just mentioned ZooKeeper? It is that core component that allows coordinating distributed components to keep in sync with each other. For our purposes, we need to make ensure that ZooKeeper is running before we start Kafka.

Additional configurations for both MongoDB and Kafka, such as replication, are beyond the scope of this book. Please read some additional material to learn about those topics.

Once we have Kafka installed on the machine, we add the following line to `build.sbt`:

```
libraryDependencies += "org.apache.kafka" % "kafka_2.11" %
"0.8.2.0"
```

This will add the Kafka library for connecting to it using its API.

Data processing pipeline for Entree

In this section, we will design a pipeline that will allow us to stream as well as persist the data. The persistence will allow us to look up the data on demand. Streaming will allow the learning algorithm to keep learning as soon as new data arrives.

We use the following technologies to achieve our goals:

- **Akka**: For message based concurrence to delegate as much as we can
- **MongoDB**: For data persistence and querying
- **Apache Kafka**: For high performance persistence queuing
- **Apache Spark**: For high throughput stream processing

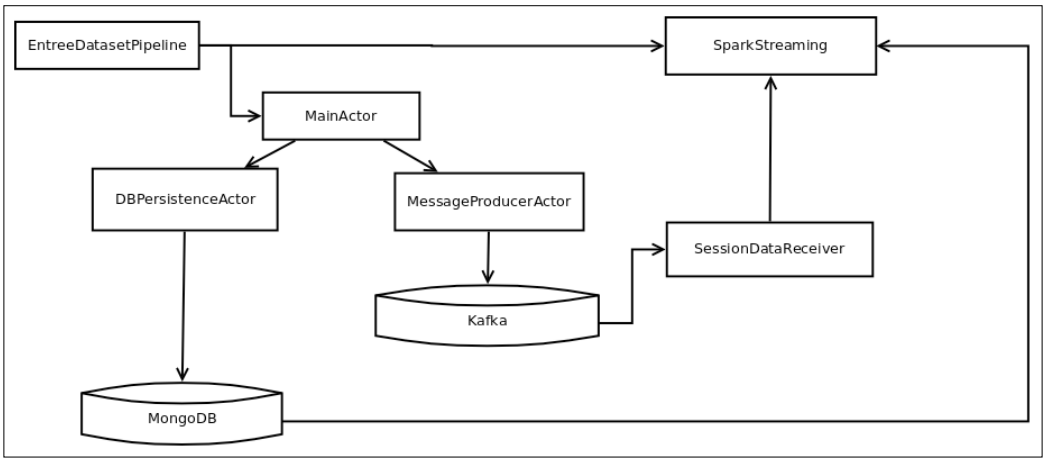
We have already covered the setup of MongoDB and Apache Kafka in this chapter, and Apache Spark in the previous chapter. We only need to add Apache Spark streaming libraries to our build file `build.sbt`:

```
libraryDependencies += "org.apache.spark" %% "spark-streaming" %
"1.3.0"
```

We are all set with the required dependencies to create a streaming example. This is essentially the flow of data across different components:

Entree dataset text files → Akka → MongoDB → Apache Kafka → Apache Spark

The entry point to this pipeline is `EntreeDatasetPipeline` Scala object's `main()` method which is defined in `EntreeDatasetPipeline.scala`. This starts the `MainActor` that reads the data from text files. In our case, we will first read from text files (Entree dataset), read and parse the data into case classes. This is done using plain Scala code.



As soon as entries from text files are read, `MainActor` delegates the task of storing those entries into MongoDB using DBPersistenceActor. MongoDB serves as a persistent datastore, for archival as well as for querying data as and when needed.

`MainActor` also delegates the task of enqueuing the user session data to the Apache Kafka queue. This queue is what will become the source of stream data processing that we will do using Apache Spark.

```
package chapter02

import java.io.{ InputStreamReader, BufferedReader, InputStream }

class SessionDataReceiver()
  extends Receiver[SessionData](StorageLevel.MEMORY_AND_DISK_2)
  with Logging {
  private def receive() = {
    // Start the thread that receives data over a connection
    val t = new Thread("Kafka Receiver") {
      override def run() {
        val consumerConfig = new ConsumerConfig(QueueConfig.consumerProps)
        val topic = QueueConfig.topic
        val topicCountMap = Map[String, Integer](topic -> 1)
        val consumer = Consumer.createJavaConsumerConnector(consumerConfig)
        val consumerMap = consumer.createMessageStreams[String, SessionData](
          topicCountMap,
          new StringDecoder,
          new SessionDataSerializer)
        val stream = consumerMap.get(topic).get(0)
        for (messageAndMetadata <- stream) {
          val key = messageAndMetadata.key()
          val message = messageAndMetadata.message()
          store(message)
        }
      }
    }
    t.start()
  }
  def onStart() { receive() }
  def onStop() {}
}
```

Now, separately in the `main()` method of `EntreeDatasePipeline`, a Spark stream is created that listens to the user session data. This is what completes the processing pipeline. A key piece to study here is the `SessionDataReceiver` class. This class implements the receiver that assists us in pushing data to the Spark stream. Finally, the Spark stream repeatedly prints the most visited restaurants in Chicago in the recent past.

```
val conf = new SparkConf(false).setMaster("local[2]").setAppName("Entree")
val ssc = new StreamingContext(conf, Seconds(2))
val receiver = new SessionDataReceiver()
val sessionDataStream = ssc.receiverStream(receiver)
val userVisit = sessionDataStream.map(sd => sd.endPoint)
val userVisitCount = userVisit.countByValue()

val pattern = """(\d\d\d)(\w)""".r
val navigations = sessionDataStream.flatMap { sd =>
  sd.navigations.filter(_.length == 4).map { code =>
    code match {
      case pattern(restuarantCode, action) => (restuarantCode, action)
    }
  }
}

val navigationCount = navigations.map(_._1).countByValue()

navigationCount.foreachRDD { rdd =>
  println("\nNext batch...\n")
  val lst = rdd.collect.toList
  val top5 = lst.sortBy(_._2).reverse.take(5)
  val top5restaurants = top5.flatMap { x =>
    val (restaurantCode, count) = x
    Database.getRestaurantByIdInChicago(restaurantCode).map { r =>
      r.name -> count
    }
  }
  top5restaurants foreach { entry =>
    println(s"${entry._1} was recently visited ${entry._2} times")
  }
}
```

Let's see some output now. First ensure that both the MongoDB and Apache Kafka servers are running, as mentioned earlier in this chapter. Then invoke the pipeline:

```
$ sbt 'run-main chapter02.EntreeDatasetPipeline /home/tuxdna/work/packt/
dataset/entree'
Number of restuarants: 4160

... OUTPUT SKIPPED ...

Next batch...

foodlife was recently visited 86 times
Anna Maria Pasteria was recently visited 71 times
Stanley's Kitchen & Tap was recently visited 59 times
Poor Phil's Oyster Bar was recently visited 54 times
Big Bowl Cafe was recently visited 53 times

Next batch...

foodlife was recently visited 88 times
Stanley's Kitchen & Tap was recently visited 63 times
Anna Maria Pasteria was recently visited 58 times
Poor Phil's Oyster Bar was recently visited 53 times
TRIO was recently visited 44 times

Next batch...

foodlife was recently visited 95 times
Anna Maria Pasteria was recently visited 80 times
Stanley's Kitchen & Tap was recently visited 72 times
Poor Phil's Oyster Bar was recently visited 51 times
Dave's Italian Kitchen was recently visited 48 times
Loading session from: /home/tuxdna/work/packt/dataset/entree/session/
session.1999-Q2
    Number of recorded sessions: 1299

... OUTPUT SKIPPED ...
```

This output is interesting. It seems the line `foodlife` has been very popular recently. Notice that we are not storing these patterns anywhere just yet. This can easily be achieved by storing an aggregate over all the past data into MongoDB. There are so many things that can be done from here on.

While the pipeline is running, you could also inspect MongoDB to ensure that the data is actually being populated there too:

```
$ mongo
MongoDB shell version: 2.6.3
connecting to: test
> use entree
switched to db entree
> db.restaurants.find()
{ "_id" : ObjectId("5549d273e4b00867355c7bcb"), "id" : "0000000", "name"
: "Tanner's", "features" : [ "100", "253", "250", "178", "174", "063",
"059", "036", "008", "074", "204", "052", "163" ], "city" : "atlanta" }
{ "_id" : ObjectId("5549d273e4b00867355c7bcc"), "id" : "0000001", "name"
: "Frijoleros", "features" : [ "250", "062", "132", "174", "063", "197",
"071", "142", "234", "243", "075", "204", "052", "162" ], "city" :
"atlanta" }
{ "_id" : ObjectId("5549d273e4b00867355c7bcd"), "id" : "0000002", "name"
: "Indian Delights", "features" : [ "253", "250", "150", "174", "083",
"059", "036", "117", "243", "076", "205", "051", "162" ], "city" :
"atlanta" }
... OUTPUT SKIPPED ...
{ "_id" : ObjectId("5549d273e4b00867355c7bdc"), "id" : "0000017", "name"
: "Jonathan Lee's", "features" : [ "253", "231", "245", "191", "192",
"174", "059", "036", "039", "235", "075", "205", "052", "163" ], "city" :
"atlanta" }
{ "_id" : ObjectId("5549d273e4b00867355c7bdd"), "id" : "0000018", "name"
: "The Country Place", "features" : [ "253", "099", "231", "250", "062",
"132", "191", "192", "174", "071", "083", "024", "215", "005", "076",
"206", "054", "165" ], "city" : "atlanta" }
{ "_id" : ObjectId("5549d273e4b00867355c7bde"), "id" : "0000019", "name"
: "Hama's", "features" : [ "253", "250", "245", "174", "128", "075",
"205", "052", "164" ], "city" : "atlanta" }
Type "it" for more
>
```

Indeed, it works! We could also see sessions data using the `db.sessions.find()` command in MongoDB shell. That is left up to the reader to explore.

That completes our task for this chapter but there are several questions worth pondering over after this exercise.

How does a pipeline relate to a recommendation engine?

In a typical recommendation system:

- We obtain the data of historical transactions along with other metadata.
- We convert these historical transactions into appropriate instances and features.
- We model the information we already have for the user, and his likes.
- We only know what a user has already liked in the past. That means we are merely predicting the future based on past experience. It is only an estimate, which we want to be as accurate as possible.
- We sometimes mean interesting as surprising, something that a user has not liked before would possibly like.

On close observation, we can clearly see that the preceding list is a sequence of steps that keep happening all the time in succession. This is exactly like a pipeline we created earlier, although we need to put much more effort into creating a good recommendation engine.

How does it relate to information retrieval?

If there were only say five restaurants, it would be easy for a user to decide, based on the cuisine, location, cost, and so on. However, this is not possible when there are many thousands of options. This problem is in a way a problem of search, relevance and taste. A user is searching for a restaurant, and only those which are relevant, according to his/her taste. Now, all we need to do is encode the problem of recommendation in terms of search, relevance, and taste.

Summary

In this chapter, we learned how to combine the power of Apache Spark, Apache Kafka, Akka, and MongoDB, which together allow us to compose a complete pipeline. We formed a minimal but complete system that gives us interesting information from a stream of data, continually. We did a hands-on exercise with Spark Streaming, and created a `SessionDataReceiver` class that listens to Kafka messages, and produces some useful output. In the next chapter, we will discuss a very common application of recommendation systems.

3
Conceptualizing an E-Commerce Store

In this chapter, we continue with the following topics:

- Introduction to e-commerce
- Importance of recommender systems
- Types of recommendation methods
- Introduction to e-commerce

E-commerce is the use of electronic communications and digital information processing technology in business transactions to create, transform, and redefine relationships for value creation between or among organizations, and between organizations and individuals.

E-commerce also pertains to any form of business transaction in which the parties interact electronically rather than by physical exchanges or direct physical contact.

In 1993, Joe Pine argued that companies need to shift from the old world of mass production where *standardized products, homogeneous markets, long product life, and development cycles were the rule* to the new world where *variety and customization supplant standardized products*. He also argued that building one product is not adequate anymore. This is called **mass customization**, which at its core allows for a tremendous increase in variety and customization without a corresponding increase in cost. Just think of the different kinds of mobile phones and features we can get these days.

Today most companies need to develop multiple products that meet the multiple needs of multiple kinds of consumers. Mass customization has made it possible, and at the same time e-commerce has allowed companies to provide customers with more options, to showcase these customized products. However, this is a massive change to the way business is done now, compared to just a couple of decades ago. With this change, arrives a new challenge—the amount of information that a customer has to process to select a desired product has increased dramatically—a challenge of information overload. One solution to this information overload problem is the use of recommender systems.

Now we know that e-commerce is essentially the trading of products or services electronically. It is important to know that it requires other technologies such as a computer network, mobile technology, electronic fund transfers, supply chain management, and automated data collection so as to complete a feature packed e-commerce store. Most popular forms of e-commerce can be identified as:

- Online shopping: Retail sales direct to customers. Examples: Amazon and Flipkart.
- Online marketplaces: Business to consumer or consumer to consumer sales. Example: eBay.
- Business to business: Direct sales from a business to another business. Examples: Salesforce and Alibaba.com.
- Pre-retail surveys to gather product feasibility. Example: SurveyMonkey.
- E-mail marketing using promos and newsletters. Example: MailChimp.
- Demographic surveys using web contacts and social media.

The following are some sites and their primary form of e-commerce engagement:

	Shopping	**Marketplace**	**B2B**	**Surveys**	**E-mail marketing**
Amazon	Yes	Yes			
Flipkart	Yes	Yes			
eBay		Yes			
Alibaba.com			Yes		
Salesforce			Yes		
SurveyMonkey				Yes	
MailChimp					Yes
NetFlix	Yes	Yes			
YouTube					

YouTube is a typical example where the use of recommender systems is very prominent; however the actual product is not obvious. It is not a shopping site, but it is kind of a marketplace for advertisers, where the user is in fact the product. Similarly, Facebook, LinkedIn, and other social networking websites have recommender systems in place.

While we are at it, we should also think about channels for communicating with the customer. These are primarily mobile apps, SMS, e-mails, phone calls, TV ads, streaming video advertisements and newspapers, exposition, and so on. Different kinds of consumers are better reached by different kinds of communication channels. For example:

> *"Whereas more B2C marketers are using social channels to connect with their customers," the report said, "more B2B marketers use traditional digital channels like email."*
>
> *The only channel more widely used than email among B2Bers is the corporate website, which is very or somewhat effective for 87 percent.*

After all, our task at hand is to help a customer narrow down his/her search for a desired product, so we need to provide recommendations using an appropriate channel. Needless to say, today we have online services for any basic human need. We can now find entertainment, travel, food, news, and other products with just a click of a button or a phone call. This brings us to the question of the value addition by using a recommender system.

Importance of recommender systems in e-commerce

Before we decide that we want to really use a recommender system in our e-commerce site, we must be convinced that it is something that will provide valuable add-ons. In this section, we will discuss the ways in which a recommender system helps us in enhancing customer experience. A recommender system enhances the customer experience in three basic ways:

- Browsers into buyers
- Cross-sell
- Loyalty

Converting browsers into buyers

First let's define what browsers and buyers mean. Browsers are those customers who simply browse through different products on a website. Buyers are those customers who end up buying any product or service on the website. A recommender system increases the chance by which a browser, who is just simply viewing some products, actually encounters one which he/she is really interested in buying. This converts a browser into a buyer.

Making cross-sell happen

A recommender system can provide product suggestions; given the fact that a customer has already selected a product for purchase. The suggestions given by recommender systems can be interesting or of the nature that those products be better bought together, with a selected product. This increases the chance of a customer buying more products from this site, rather than from a competitor's site, for example.

Increased loyalty time

Loyalty time is an important factor in every decision a customer makes. Because of previous purchases or interaction with the website, a recommender system can tailor the product recommendations better. Also from a customer's perspective, it is still time consuming to visit and browse many different sites in order to find the desired product. When a customer has built trust with an e-commerce site, then it is more likely that a customer will visit that site again for a purchase. To make that happen, a recommender system can enhance the user experience by providing better recommendations than other sites.

Next, we discuss a few other terminologies that help us identify the kind of recommender system we can implement in our site. For a user to be able to find a recommendation, he/she needs to have a way to reach it. This is defined using a recommendation interface. A recommendation interface is populated using some recommendation technology, which in turn is fed using some form of input from the whole system. This is essentially how it looks:

Input → recommendation technology → recommendation interface → means to reach to a user

A recommendation interface is the kind of information that we display to the user. This can be of the following types:

- Similar items: This shows similar items
- Top *N* list: This shows top *N* items by some ordering criteria

- Average rating (or an estimate of rating): This shows items based on user ratings
- Comments or reviews
- Search results
- Ordered search results
- Browsing (a catalog for example)

Recommendation technology is the mechanism that populates the recommendation interface. All recommendation technology requires some form of input. The following are the different techniques:

- **Attribute-based**: Filter items based on some attributes provided by the user
- **Item to item correlation**: Filter items based on items similar to the selected item
- **People to people correlation**: Filter items based on choices of users who have similar tastes to the current user
- **Aggregated rating**: Order items based on user ratings

Each type of recommendation technology is fed with some data from the system. For example, in case of *Amazon.com Delivers* (a service by Amazon), an e-mail subscription can be set up by a user. This allows a user to receive recommendations based on some attributes of a product. The recommendation technology in this case is attribute-based, and the recommendation interface is via e-mail.

Let's see some more examples of recommendation mechanisms on a few popular sites. We will cover the following sites:

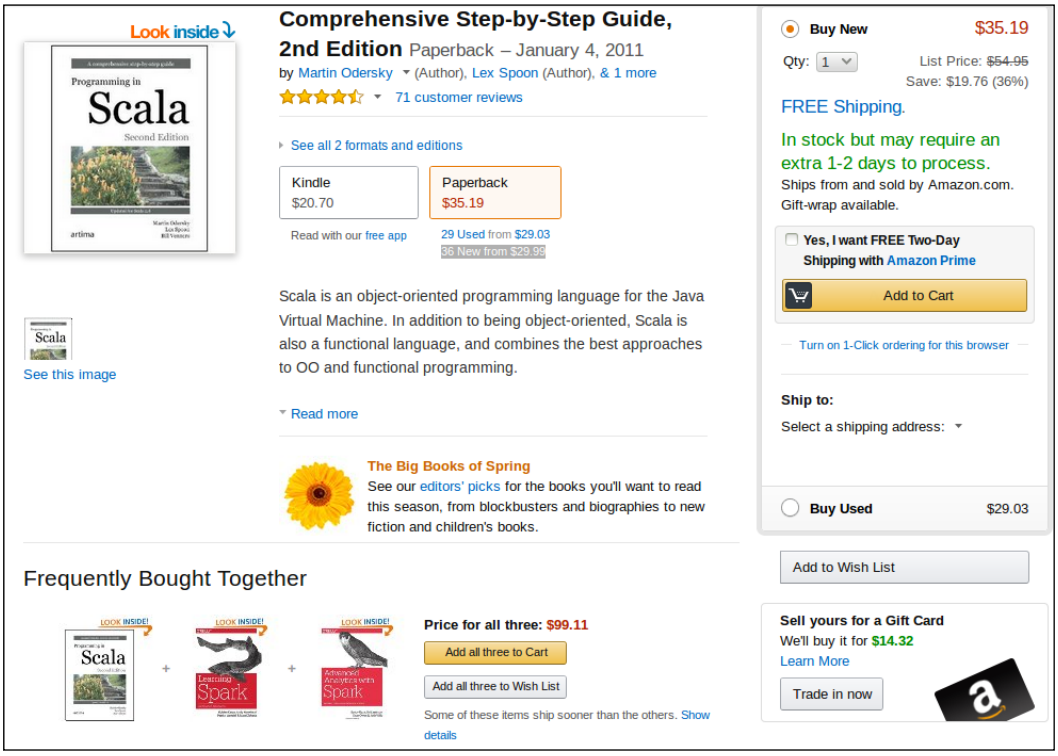
- Amazon
- Flipkart
- IMDb
- eBay
- Google News
- Times of India

Types of recommendation methods

In the following pictures we show you different real-life examples of recommendations on some e-commerce and other sites such as Amazon, Flipkart, Times Of India, and eBay.

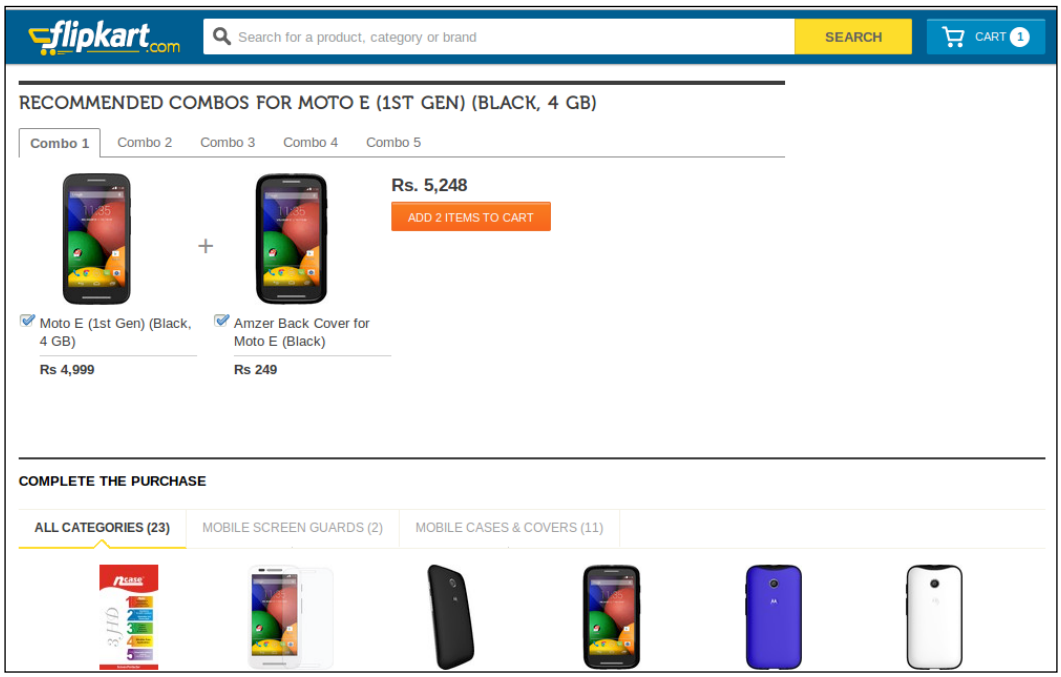
Frequently bought together

In the following picture, when we select the book *Programming in Scala* at Amazon, we see suggestions (or recommendations) for those books that were brought together with the selected book:



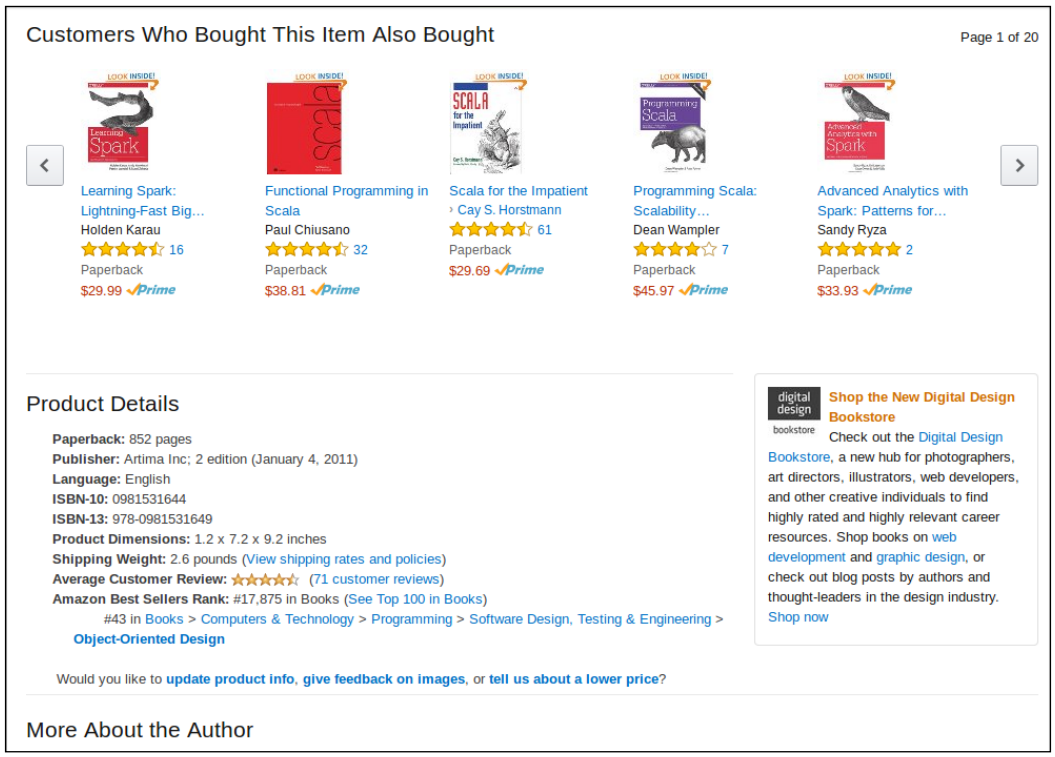
An example of frequent patterns

The following is another example of product recommendations based on the frequent patterns, that is, the products that are bought together by customers. Here Moto E mobile phone and Moto E back cover combo are recommended for the user to buy.



People to people correlation

We can also determine that if two people like similar items, then if one person liked/bought something that other person didn't, we can recommend it. The following are some examples based on people to people correlation. See the header **Customers Who Bought This Item Also Bought**.



Customers Who Bought This Item Also Bought

Page 1 of 20

Learning Spark: Lightning-Fast Big…
Holden Karau
16
Paperback
$29.99 Prime

Functional Programming in Scala
Paul Chiusano
32
Paperback
$38.81 Prime

Scala for the Impatient
› Cay S. Horstmann
61
Paperback
$29.69 Prime

Programming Scala: Scalability…
Dean Wampler
7
Paperback
$45.97 Prime

Advanced Analytics with Spark: Patterns for…
Sandy Ryza
2
Paperback
$33.93 Prime

Product Details

Paperback: 852 pages
Publisher: Artima Inc; 2 edition (January 4, 2011)
Language: English
ISBN-10: 0981531644
ISBN-13: 978-0981531649
Product Dimensions: 1.2 x 7.2 x 9.2 inches
Shipping Weight: 2.6 pounds (View shipping rates and policies)
Average Customer Review: (71 customer reviews)
Amazon Best Sellers Rank: #17,875 in Books (See Top 100 in Books)
#43 in Books > Computers & Technology > Programming > Software Design, Testing & Engineering > Object-Oriented Design

Would you like to update product info, give feedback on images, or tell us about a lower price?

digital design bookstore

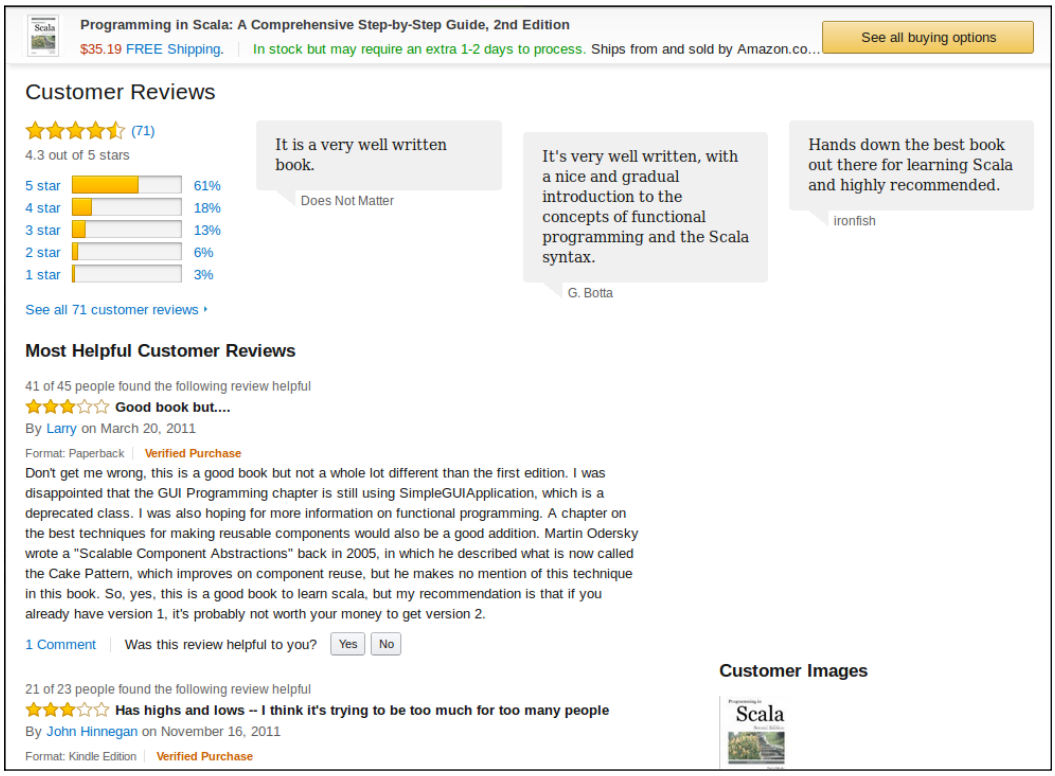
Shop the New Digital Design Bookstore

Check out the Digital Design Bookstore, a new hub for photographers, art directors, illustrators, web developers, and other creative individuals to find highly rated and highly relevant career resources. Shop books on web development and graphic design, or check out blog posts by authors and thought-leaders in the design industry. Shop now

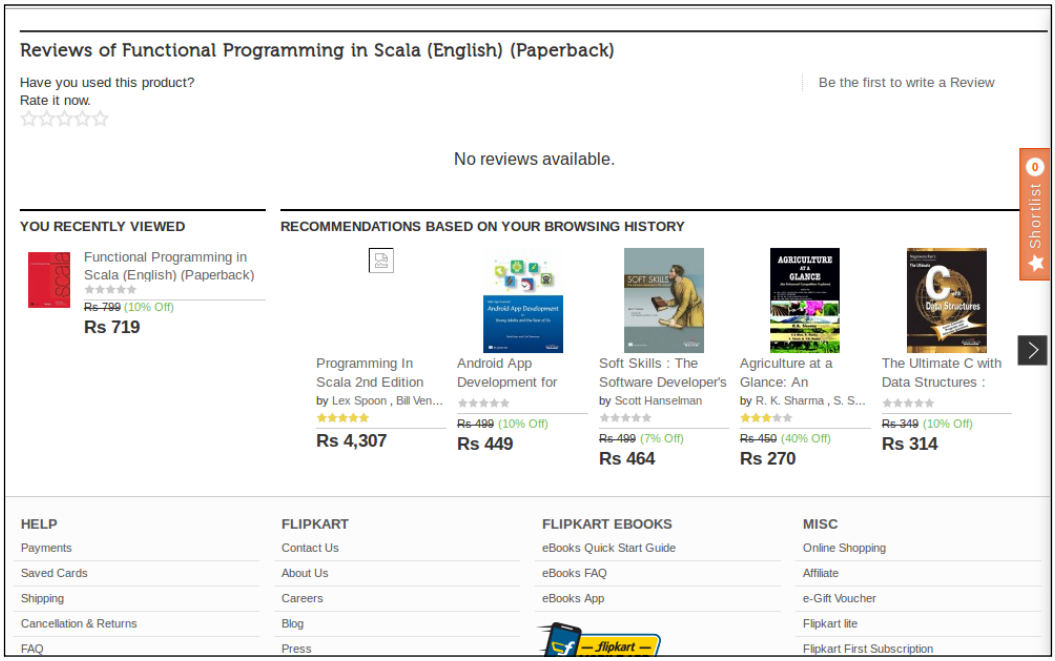
More About the Author

Customer reviews and ratings

Today e-commerce sites are so popular that we tend to buy as much as we can online because it saves us time and energy. However, our decision to buy a product greatly depends on the reviews and ratings provided by other customers. Check out the **Customer Reviews** and **Most Helpful Customer Reviews** sections in the following screenshot:



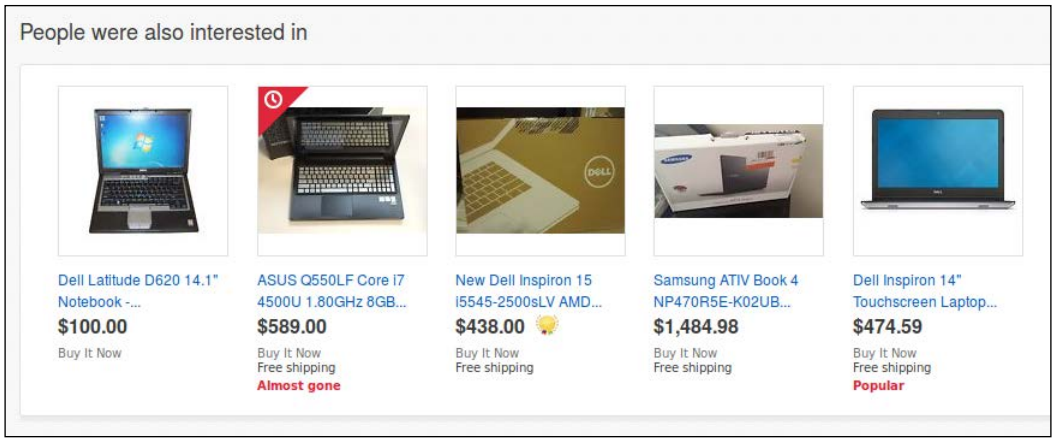
Notice how the customers' aggregated rating (stars) and their distribution across five different ratings is shown. Also note that customer reviews are listed together, making it very easy for a user to make decisions.



In the preceding screenshot, we see no reviews for a given product; however, that space is complemented by recommendations based on the user's recent history. Basically, we fill in the space with some information that could help a user find relevant items.

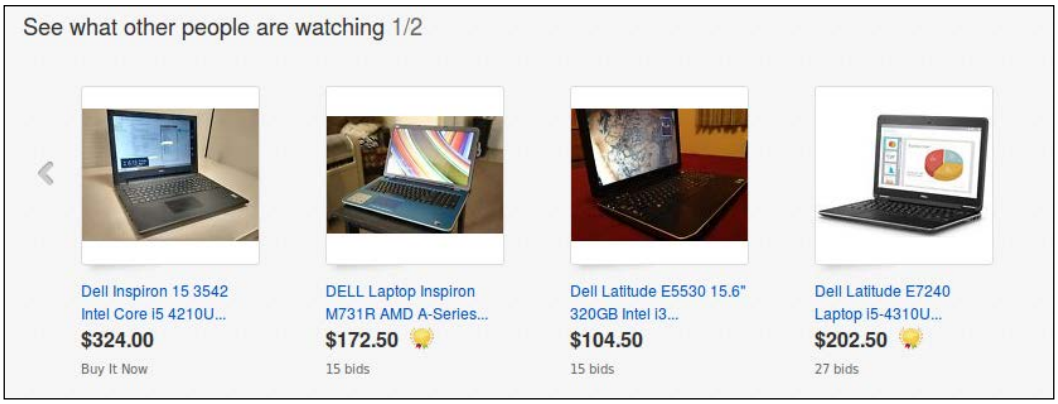
People who were also interested in other similar items

We can use the search results of other users who have also searched for similar products and provide it to our current user. This will give the user a larger variety of options in products and also keep him/her interested in the site. The following screenshot illustrates this:



Recommendation from others' views

We can also add another dimension to the mix, that is we can recommend based on the current activity of all the users. The following picture demonstrates that on selecting a Dell laptop on eBay we see what laptops people are currently watching. This makes sense in those situations where we have a lot of people currently making transactions, and the lifetime of a particular item on the website is pretty short. This is true of consumer to consumer sites. On such sites, people auction an item, and it is sold in a very short time of maybe hours, days, or weeks. However, on many other e-commerce sites, the items would be available as long as they are available across the markets.



Example of similar items

In the following screenshot, we see similar news items, which are placed right beside the current news item. This is an example of item to item correlation:

In the following table, we can summarize the three components of a recommender system. Notice how the recommendation technology is annotated with appropriate input data, below it. Also note that this is only a selective coverage of these websites, not an exhaustive coverage. There may be many more features that are not immediately obvious.

Site	Recommendation interface	Recommendation technology	How user finds/reaches a recommendation
YouTube	Similar item	Item to item correlation **Input**: Previous videos watched	Organic navigation
Amazon	Similar item Top *N* list Average rating Comments/reviews E-mail	Item to item correlation People to people correlation **Input**: Purchase history, aggregated rating, and user reviews	Keyword product search Organic navigation E-mail subscription
Flipkart	Similar item Top *N* list Average ratings Comments/reviews	Item to item correlation People to people correlation **Input**: Purchase history, aggregated rating, user reviews	Keyword product search Organic navigation
eBay	Comments/reviews Average ratings Similar items	Aggregated rating People to people correlation **Input**: Likes / text / navigation history	Organic navigation
IMDb	Similar items Ordered search results Average rating	Item to item correlation People to people correlation **Input**: Aggregated rating	Organic navigation Keyword search Categories
TOI News	Similar item Browsing	Item to item correlation Attribute based **Input**: Editor's choice	Organic navigation Keyword search Categories

There are two more metrics we need to discuss. First is the effort put by a user in actually reaching to a recommendation on a scale of manual to automatic. Second is the persistence of recommendations on the scale of ephemeral to persistent.

- **User effort**: Manual versus automatic
- **Persistence**: Ephemeral versus persistent

Manual

Manual recommendation involves an active effort on the part of a user to locate the items he/she is actually interested in. For example, in the case of a news site, a user might need to browse or search for a news item before interesting news is reached.

Automatic

Automatic recommendation is displayed to a user as soon as he/she visits the website, for example, when a user is browsing for USB storage, after buying a mobile phone the last time. If the website actually provides recommendations for the latest USB storage devices at this time, then it is a real time saver. The site will find a loyal customer in this user.

Ephemeral

Ephemeral recommendations are generated for a single user session, and it doesn't really utilize a previous transaction history of a user. For example, if a news site simply displays the most popular news story of the week, then it is true for all the users on this site.

Persistent

Persistent recommendations require that the likes and dislikes of the user are known in advance so as to make personalized recommendations to a user. The USB storage example that we saw earlier is a good example of persistent recommendation.

The optimization problem in a large e-commerce store is the maximization of profit and at the same time it is minimization of the in-store wait time for a product. The longer a product stays in a warehouse, the more its profitability decreases because other easy-selling products cannot be stored in the warehouse. It is easy to guess that frequent pattern suggestions work best when provided with a combined discount and customers prefer to see ratings and read reviews before buying products on-line. We can look at different sites from different angles and think about how to better serve the user with good recommendations.

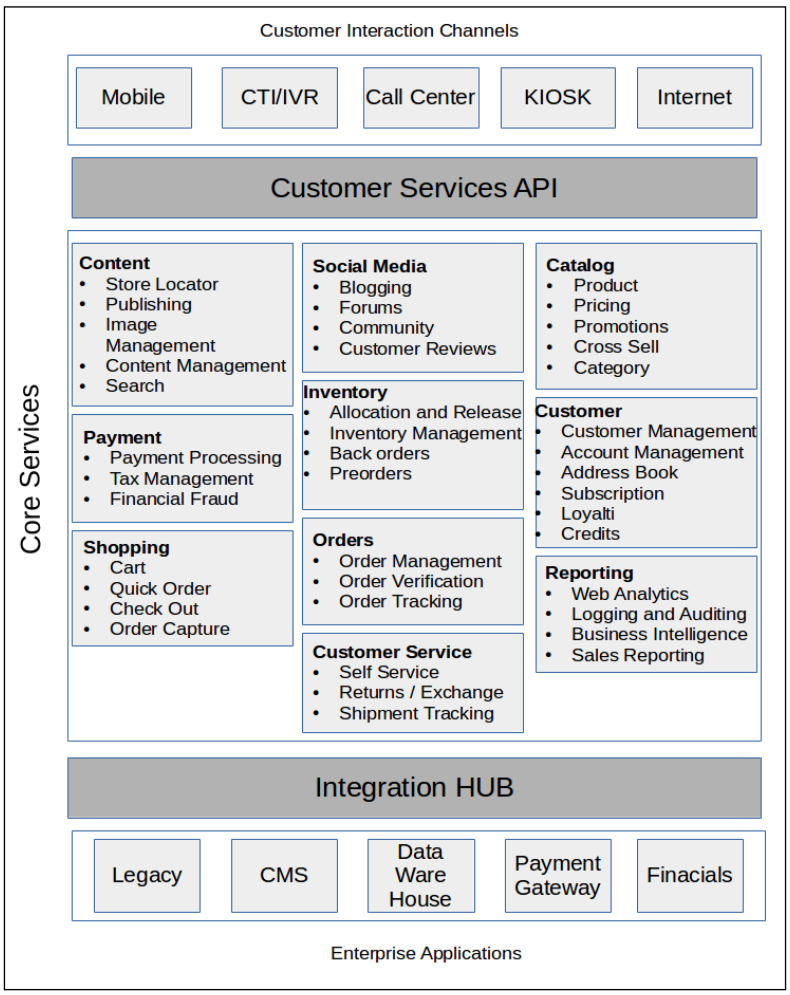
Now that we have an understanding of which concepts of a recommender system we can leverage, in order to make the customer experience better, let's look at the architecture of a typical project.

The architecture of the project

For a simple e-commerce site these days, the following entities are obvious:

- Products/items and their metadata
- Customers/users and their metadata
- News items/blog posts related to the products or from editorial
- User reviews associated with products
- User ratings associated with products

Other than that, there are many more systems required to effectively build a complete e-commerce enterprise. The following diagram highlights them:



Because our objective is to build a recommender system and not a complete e-commerce site, we will narrow our focus to a minimum set of requirements. So here are different software components that we will need:

- Persistent/structured data storage
- A queuing mechanism
- Search support

For data storage, we can use MongoDB, and we have already covered it in a previous chapter. Because MongoDB is NoSQL storage, we need to be careful in designing a schema that allows us to join different entities such as a user and reviews to form a complete product profile.

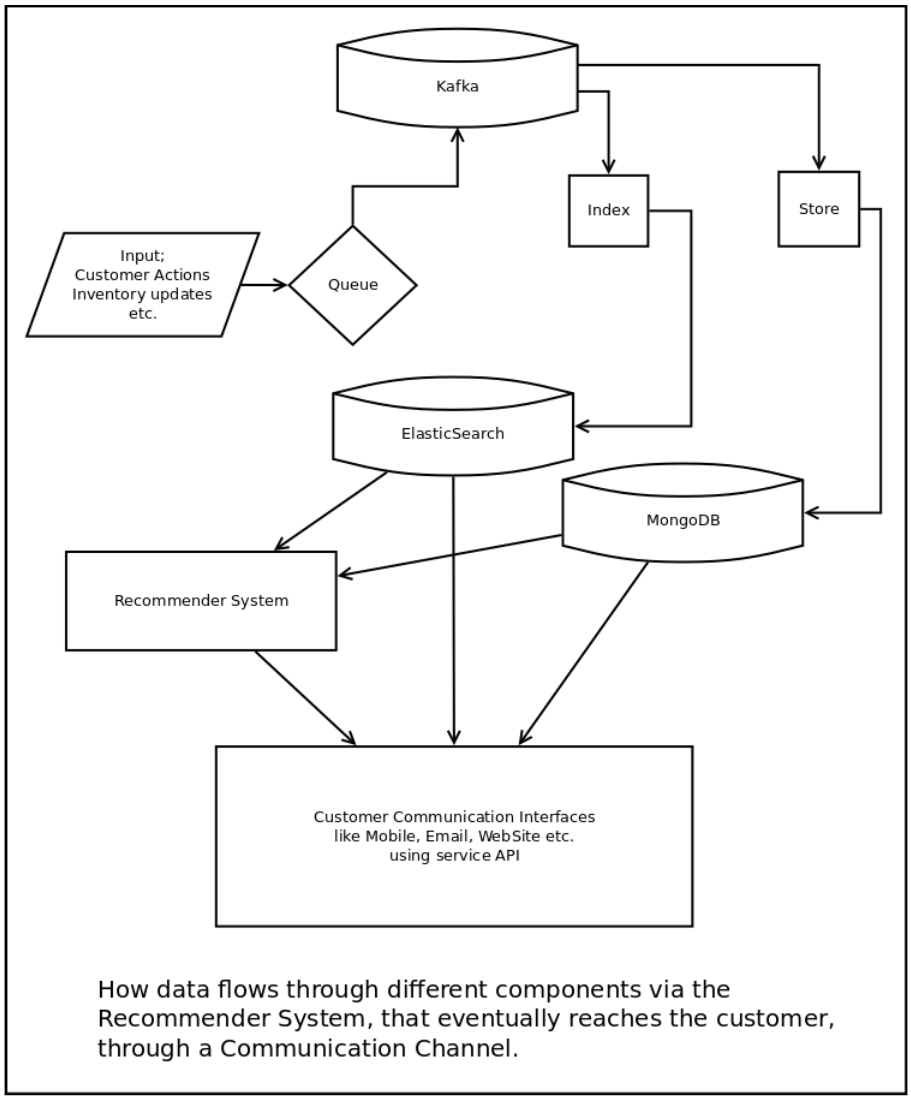
A queuing mechanism is used to process the data as it arrives in a streaming fashion. This is also important if different independent components such as a search indexer, recommendation engine, e-mailer service, and so on, all want to process the data in parallel. We can use Apache Kafka for this purpose. Since we have already covered Apache Kafka in a previous chapter let's stick to that.

For search, we can use a popular search technology such as Elasticsearch/Apache Solr, or just plain Apache Lucene. Although MongoDB also supports search queries, it is not as extensive as Elasticsearch or Apache Solr. In order to set up Elasticsearch (or Apache Solr) you can refer to their project pages:

- Elasticsearch: `https://www.elastic.co/`
- Apache Solr: `https://lucene.apache.org/solr/`

We will also go through Elasticsearch setup in *Chapter 7*, *Enhancing the User Experience*.

The following is the architecture of the application that we will build:



How data flows through different components via the Recommender System, that eventually reaches the customer, through a Communication Channel.

Batch versus online

As shown in the architecture diagram of your application, the input data of different interactions happening in the system are captured as soon as they take place. They are routed via the queuing mechanism to storage and indexing components (we have ignored other components such as e-mail and payment for now).

When this data finally reaches the recommender system component, then either the recommender system will learn instantly, that is, online recommendations, or it will wait for some specified time (maybe hours or days) and then re-generate recommendations, that is, batched processing. The recommender system can wait for some time before generating new recommendations. This delayed approach is also called batching. This can be due to the fact that either enough data is not yet available so it makes no sense to run a recommender algorithm or the recommender algorithm is itself very expensive.

Summary

In this chapter, we did a quick survey of an e-commerce site, its variations, and the importance of having recommender systems. We can now say that a customer will only buy products that are visible. A product that is hiding down below the pages and links is not likely to be bought by a customer. This is where a recommender system helps an e-commerce site, that is, with such a huge number of items to search through, a customer is now able to narrow his/her search with ease. In the next chapter, we will learn about different machine learning algorithms that will help us pick and evaluate the techniques for building a recommender system.

4
Machine Learning Algorithms

In this chapter, we continue with the following topics:

- Hands on with Spark/MLlib
- Implement classification and regression algorithms using Spark/MLlib
- Implement clustering algorithms using Spark/MLlib
- Using Python for plotting our results

Hands on with Spark/MLlib

In the first chapter, we set up Apache Spark and also discussed an example using Spark/MLlib. Now is the time to explore Spark/MLlib in detail.

The following are the different features provided by Spark/MLlib:

- **Data types**: `vector`, `LabeledPoint`, `matrix`, `DistributedMatrix` (`BlockMatrix`, `RowMatrix`, `IndexedRowMatrix`, and `CoordinateMatrix`)
- **Statistics**: Summary statistics, correlations, stratified sampling, hypothesis testing, random data generation
- **Feature extraction**: TF-IDF (HashingTF and IDF), scaling, normalization, and Word2Vec
- **Classification/regression**: Linear regression (`LinearRegressionWithSGD`, `LassoWithSGD`, and `RidgeRegressionWithSGD`), logistic regression (`SGD`, `LBFGS`), SVM, Naive Bayes, decision tree, ensembles (`RandomForests` and `GradientBostedTrees`)
- **Clustering**: K-Means, Gaussian mixture model/expectation-maximization, power iteration clustering, LDA (`EMLDAOptimizer`/`OnlineLDAOptimizer`), streaming K-Means
- **Association analysis**: Frequent pattern mining (FPGrowth)

- **Dimensionality reduction**: SVD and PCA
- **Recommendation**: Collaborative filtering
- **Pipeline API**

Data types

Recall that in the Entree dataset (*Chapter 2*, *Data Processing Pipeline Using Scala*), we represented each restaurant feature with an array of Boolean values. We could also think of that as a vector of feature values. This is exactly what we can do using the vector data type in Spark.

Vector

There are two variations of a vector: dense vector and sparse vector. For a vector of length *N*, a dense vector will take space equal to *N* double values. So, even if a restaurant has only a few features present, the vector will occupy a lot of space. However, with sparse representation, we only need to store the non-zero values. So, if there are *M* non-zero values where $M << N$, then the total space requirements are equal to *M* integers and *M* doubles. Before deciding whether to use a sparse or dense representation, you may want to think about the sparsity of your feature vectors overall. Also, you may want to think about the operations to be performed on the vectors. A dense vector allows fast random lookups, however, that is not the case with sparse vectors.

Matrix

Another data type is a **matrix**. A matrix can also be both dense and sparse. Sparse implementation in Spark is done using a coordinate-list format—`CoordinateMatrix`. `IndexedRowMatrix` is another sparse implementation where we can only have sparsity at the row level, that is, each of the row vectors is a sparse vector; however, there can be no missing rows. There are two other variations of a matrix, that is, a matrix can either be located locally or distributed across a Spark cluster. You may want to read Spark documentation to decide which ones to choose for your problem.

Labeled point

We can represent an instance and its label using a data type called **labeled point**. Labeled point has two members: a real label and a feature vector. By convention in Spark, a binary label is represented as—`0` for negative and `1` for positive. Multi-class labels should start from 0 with a step increment of 1. For example, if there are 4 classes, the labels should be 0, 1, 2, and 3.

Let me at this point introduce you to **RDD** (**Resilient Distributed Dataset**). Essentially it is an abstraction over a collection of objects (which could be a Scala object or Scala tuples, or just primitive data). In simple terms, RDD is a distributed collection of elements, spread across multiple nodes in a cluster. This distribution is taken care of by Spark.

For more information about RDD, you can read the documentation:
`https://spark.apache.org/docs/1.4.1/quick-start.html`

So finally, we have multiple options to encode our original dataset into different Spark data types. We can use an RDD of vector, or RDD of labeled point or a distributed matrix. Before proceeding with the **machine learning** (**ML**) algorithms we may also want to understand our dataset better.

Statistics

Statistic modules in Apache Spark provide different utility methods and classes for computing: statistical summary, sampling, random data generation, and so on. Here, we discuss them in brief.

Summary statistics

Using the `Statistics.colStats()` method, we can obtain `MultivariateStatisticalSummary` that provides us column-wise (or per feature): max, min, mean, variance, non-zero count, and total count for each feature.

Correlation

We can also find a metric that tells us how correlated two series of data are. For example in a dataset, if *X1* represents age and *X2* represents number of doctor-visits, then we can find the correlation between *X1* and *X2* using the `Statistics.corr()` method. However, only two kinds of correlation metrics are currently supported: **Pearson** correlation and **Spearman** correlation. The Pearson correlation is the default, unless specified otherwise. The Pearson correlation of two series is the ratio of their covariance to the product of their variances, which means it is a value between: 1.0 and 1.0. Similarly, Spearman correlation is defined for *X1* and *X2* but using relative rank of the different values of *X1* and *X2*.

Pearson correlation $\rho_{X_1,X_2} = \frac{\text{cov}(x_1, x_2)}{\sigma x_1 \sigma x_2}$ where $\text{cov}(X_1, X_2)$ is the covariance and $\sigma(X)$ is the standard deviation of *X*.

Spearman correlation $\rho x_1, x_2 = \frac{6 \sum d_i^2}{n\left(n^2 - 1\right)}$ where n is the number of samples, $d_i = x_i - y_i$ is the difference between ranks.

These correlation values can help you identify which features can be ignored. If any two features have a very high correlation, then we may just use only one of these two features. On the other hand, we may also do a correlation between a feature and a label. However, in this case we may want to keep those features which are highly correlated with a class label. This brings us to a question of bias versus variance and also of over-fitting and under-fitting a ML algorithm. If we keep only those features that are highly correlated with the class labels, then we are likely to over-fit to the given set of data, because we are now biased towards a specific set of features. However, to keep a balance between over-fitting and under-fitting, we should follow an empirical approach.

Sampling

Usually for huge amounts of data, we first want to build a model on a subset of that data. So first we need to sample the data; select a subset from the whole dataset. Also, sometimes it is not feasible to obtain the whole dataset, but only a subset of it. This makes it feasible and faster to compare and contrast from among different ML models. However, we also need to make sure that the sampled data is a good representative of the whole data. Spark provides stratified sampling: a method of sampling from a population. Things to note here are that in Spark it works only on key-value data. Also, we specify the portion distribution to be sampled using fractions per key. However, this is only an expected value and may not be an exact fraction of the whole dataset being sampled (to get exact distribution, you can use `sampleByKeyExact` function instead of `sampleByKey`).

Hypothesis testing

Hypothesis testing is an approach in statistics that allows us to accept or reject a hypothesis. For example, in terms of ML, two hypotheses could be two different classification models or patterns extracted from two different algorithms. Using hypothesis testing we decide which one of the models or algorithms to reject. The only statistic test for performing hypothesis testing available in Spark at the moment is Pearson's **Chi-squared** test. An intuitive way of understanding a Chi-squared test is like this. Let's consider we are told some distribution of sales over the seven days of week. However, this distribution could change because of some random factors or just due to chance. A Chi-squared test allows us to find the best fit based on the observations and expected distribution. This enables us to identify whether or not there is some significant underlying reason for the variation in observations that is not merely due to chance. We can use the `Statistics.chiSqTest()` method to obtain statistics on the vector, matrix and RDD of a labeled point. So, of course first we need to encode our results into one of `vector`, `matrix`, or `RDD [LabeledPoint]` to perform Chi-squared tests.

Random data generation

Spark also provides utility methods to generate random data—`RDD [Double]` and `RDD [Vector]`—which are **independent and identically distributed** (**i.i.d.**); for more information, refer to `https://en.wikipedia.org/wiki/Independent_and_identically_distributed_random_variables`). The `RandomRDDs` object has different methods that can generate i.i.d values from gamma, Poisson, log normal, normal, uniform, and from a random generator.

Feature extraction and transformation

In *Chapter 2*, *Data Processing Pipeline Using Scala*, we discussed different kinds of data types – continuous, discrete, and so on – and a couple of data cleaning methods. Now is the time to see what else we can do for cleaning and extraction of data. Spark provides many approaches for feature extraction and transformation: TF-IDF, Word2Vec, `StandardScaler`, normalizer, and feature selection.

Term frequency-inverted document frequency (TF-IDF)

TF (short for **term frequency**) and **IDF** (short for **inverted document frequency**). TF-IDF is specifically suited for text documents where we determine the discriminating power of a term in a document using this score. In Spark, TF can be calculated using `HashingTF`, which is based on a hashing trick and it uses less space when compared with the total possible number of terms present in all documents. On the other hand, IDF is calculated after we have determined TFs for all the documents. So essentially, it captures the essence of word count statistics among all the documents and also preserves the discriminating power of different words across all documents. Mathematically, TF-IDF for a term *t*, occurring in a document *d*, can be written as:

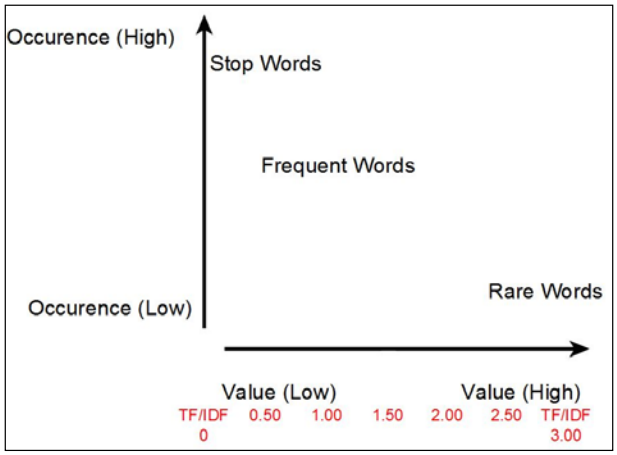
$$TFIDF(t,d,D) = TF(t,d) \times IDF(t,D)$$

Where $IDF(t,D) = \log \frac{|D|+1}{DF(t,D)+1}$:

- D is the set of documents
- $|D|$ is the total number of documents
- $TF(t,d)$ is the TF

There can be many variations of a TF-IDF formula, however this is the one used in Spark implementation right now.

The following graph depicts the power of TF-IDF, such that we can now use it to distinguish among stop-words, frequent-words, and rare-words:



Word2Vec

One of the approaches to turn text data into vector representation is to use Word2Vec, which creates a vector representation of words in a text corpus. In Spark, a skip-gram model is used using the hierarchical `softmax` method to train these vectors. This vector representation can be used in performing **Natural Language Processing** (**NLP**) and for ML algorithms.

Word2Vec basically creates vectors for all words in a text corpus based on the context in which it occurs. Given these vectors you can make queries for a given word, and the Word2Vec model will return words which are synonymous in meaning. This is amazing because the algorithm doesn't really understand any grammar or meanings of words but is still capable of giving pretty sensible results. For example, you can look at an example on Spark documentation here: `https://spark.apache.org/docs/1.4.1/mllib-feature-extraction.html#word2vec`. When you generate a model for `text8` corpus, you can make a query for India and this is what you will get:

```
scala> val synonyms = sameModel.findSynonyms("india", 5)

synonyms: Array[(String, Double)] = Array((pakistan,1.6807150073221284),
(nepal,1.5295662756996018), (tibet,1.5212651063129028),
(indonesia,1.5115830206114467), (gujarat,1.4644601715558132))


scala> for((synonym, cosineSimilarity) <- synonyms) {println(s"$synonym
$cosineSimilarity")}

pakistan 1.6807150073221284

nepal 1.5295662756996018

tibet 1.5212651063129028

indonesia 1.5115830206114467

gujarat 1.4644601715558132
```

That is the power of Word2Vec. For more extensive details and use cases of Word2Vec please watch this video: `https://www.youtube.com/watch?v=vkfXBGnDplQ`.

StandardScaler

In *Chapter 2, Data Processing Pipeline Using Scala*, we discussed what standardization means. Standardizing all the features on a common scale avoids the influence of features with high variance on the rest of the features, thereby also influencing the behavior of the final model. Spark provides `StandardScaler` and `StandardScalerModel` classes for performing standardization on the dataset.

Normalizer

We also discussed normalization in *Chapter 2*, *Data Processing Pipeline Using Scala*. Normalization is specifically more suited for vector space models, where we are only interested in the angle between two vectors. Spark provides a normalizer class for this task.

Feature selection

As the number of features increases, the requirement for the size of data to backup quality results also increases. This is also called the **curse of dimensionality**. So by feature selection we choose which of the features we want to select to create a NL model. Feature selection is best done by having some domain knowledge. For example, if you have two features "total price" and "total sales tax," then one of them is redundant because the "percentage sales tax" is always fixed. So we can infer one from the other. Also, we perform correlation analysis on these two features; we will get a high correlation score. Spark provides Chi-squared feature selection to help us with this task. Also, note that as we decrease the number of features via feature selection, we are not degrading the quality of results. We are only selecting the relevant features for the given ML task – in a sense we are reducing the dimensionality of the dataset. Chi-squared is done using `ChiSquaredSelector` class in Spark. There is another approach called dimensionality reduction that we will see next.

Dimensionality reduction

Spark provides us with two ways of performing dimensionality reduction: SVD and PCA. **SVD** (short for **Singular Values Decomposition**) is a matrix factorization approach where a given matrix can be decomposed into three matrices that satisfy the following equation:

$$A_{m\times n} = U \sum V^T$$

Where:

- U contains left singular vectors
- V contains right singular vectors
- Σ is a diagonal matrix with singular values on the diagonal

Now, after we perform SVD, and we choose only top k singular values, we can reconstruct the original matrix $A_{m\times n} = U_{m\times k} \sum_{k\times k} V^T{}_{k\times n}$.

PCA (short for **Principal Component Analysis**), which is perhaps the most popular form of dimensionality reduction. To appreciate the power of PCA, we must first recall that we should never over-fit our models. The reason is that if we over-fit a model, it will likely produce bad results on an unknown dataset. To overcome this problem, we have to find a training dataset that has most variance, which in turn increases the chances of covering some unknown dataset, however at the same time we want to do that in a reduced dimension. The goal of PCA is to find those projections of the dataset which maximize the variance, and these projections are found in decreasing order of variance. Once we find the projections using PCA, our task is to select the first few projections and use them instead of the original dataset. While PCA is based on statistics, SVD is a numerical method. Although PCA and SVD are excellent tools for dimensionality reduction, one should be careful running them on huge datasets as they are themselves very resource intensive. Therefore, it is best to first apply manual feature selection and then run these algorithms.

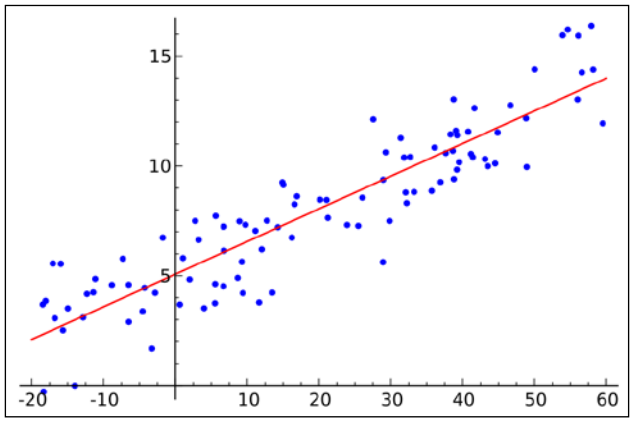
Once we have extracted, transformed, and reduced our dataset, it is time to run some ML algorithms on the dataset. In the following sections, we will see the different algorithms provided by Spark.

Classification/regression

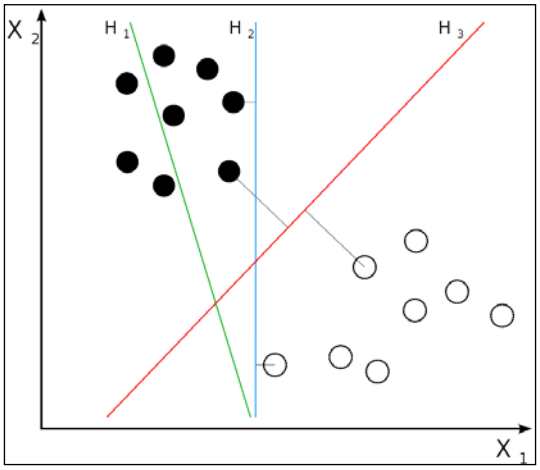
We very briefly discussed supervised learning in *Chapter 1*, *Introduction to Scala and Machine Learning*. Supervised learning is a ML technique used when we have some historical data. We will discuss two specific approaches to supervised learning: classification and regression. In both these approaches, we ultimately create a learning model, which allows us to assign a label to an unknown dataset. We say a label assignment is classification when the *label is categorical* and regression when the *label is continuous*. For example, prediction of the stock price for the next day is a regression problem and determining whether or not it will rain tomorrow is a classification problem (a binary classification problem). Let us now see different algorithms provided by Spark.

Linear methods

In Spark, the different linear methods use convex optimization to find the objective function. There are two types of optimization methods available in Spark—**Stochastic Gradient Descent** (**SGD**) and **Limited memory Broyden Fletcher Goldfarb Shanno** (**L-BFGS**)—of which SGD is more prominent in different algorithms. Which brings us to these different Linear algorithms—linear SVM, logistic regression, linear regression (using ordinary least squares, ridge regression, or Lasso). Note that there are variations of these algorithms with either SGD or L-BFGS. Spark also provides a streaming data variant of regression using the class `StreamingLinearRegressionWithSGD`. The following illustration shows how the linear regression algorithm forms a straight line:



The following illustration shows how the SVM algorithm finds the hyperplane that best separates points in a plane:



Naive Bayes

The Naive Bayes model is based on the Bayes rule, which finds the probability of a class label when we present it with the feature values of a data instance. When we derive this calculation using the Bayes rule, we will see that there are conditional dependencies among features, which are eliminated with a naive assumption; that probability of a class given a feature does not depend on any other feature value. This *naive assumption* defines the Naive Bayes model. This assumption also makes the Naive Bayes model a fast learning model and we can perform multiclass classification with high accuracy in practice.

Bayes rule:

$$p(C_k|x_1,\ldots,x_n) = p(C_k)p(x_1,\ldots,x_n|C_k) = p(C_k)p(x_1|C_k)p(x_2|C_k,x_1)\ldots p(x_n|C_k,x_1,x_2,x_3,\ldots,x_{n-1})$$

The Bayes rule under naive assumption:

$$p(C_k|x_1,\ldots,x_n) = p(C_k)p(x_1,\ldots,x_n|C_k) = p(C_k)p(x_1|C_k)p(x_2|C_k)\ldots p(x_n|C_k) = p(C_k)\Pi p(x_i|C_k)$$

Decision trees

Decision trees are different to other models in the sense that they do not form a mathematical model for regression or classification. Rather, decision trees form an actual representation of yes/no decisions with a tree structure. We can perform both classification and regression using decision trees.

Ensembles

Spark also provides ensembles of decision trees—random forests and gradient boosted trees. An ensemble method is an approach to creating learning models where a model is composed of other similar models. In the case of a decision tree ensemble, majority voting is used to generate the final instance label.

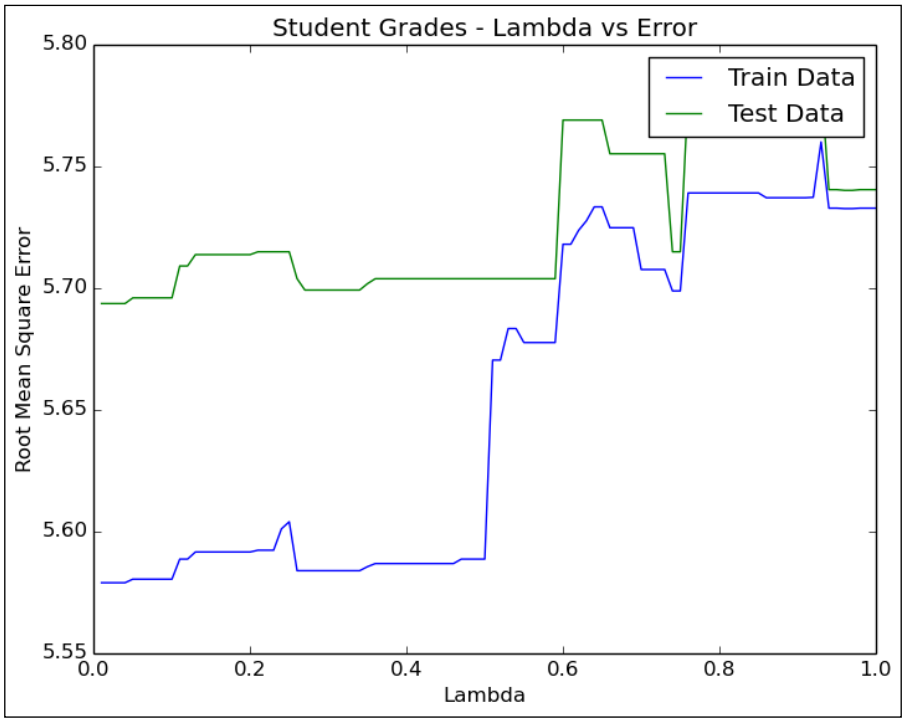
Let's now see some examples of classification/regression algorithms in action. The first dataset we will consider is student grades datasets: `https://archive.ics.uci.edu/ml/datasets/Student+Performance`.

The student performance dataset can be termed as both a classification and a regression problem. Only one of 21 grades (0 to 20) is assigned to each student. Since the class label or the grade is discreet, countable and finite, we can use Naive Bayes multi-class classifier for this problem. We can also model this as regression problem by considering the target label as a real number.

First, let's take a look at the Naive Bayes implementation. Naive Bayes takes a lambda parameter with a value from 0 to 1. We try to vary this parameter in increasing order and then observe its effect on the predictions. Its code is as follows:

```
object StudentGradesNaiveBayes {
  def main(args: Array[String]) {
    val dataPath = "datasets/student"
    val mathFileName = "student-mat.csv"
    val fileName2 = "student-por.csv"
    val mathInstances = Utils.readAndEncodeData(dataPath, mathFileName)
    val portugeseInstances = Utils.readAndEncodeData(dataPath, mathFileName)
    val allInstances = mathInstances ++ portugeseInstances
    // now that we have encoded features into numeric values,
    // lets create RDD of labeled points for classification
    val conf = new SparkConf(false).setMaster("local[2]").setAppName("StudentGrades")
    val sc = new SparkContext(conf)
    val data = sc.parallelize(allInstances)
    // test / train split
    val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
    val trainData = splits(0).cache()
    val testData = splits(1)
    val NLambdas = 100
    val output = (1 to NLambdas) map { i =>
      val lambda = 1.0 * i / NLambdas
      val model = NaiveBayes.train(trainData, lambda = lambda)
      val trainResult = Utils.evaluate(model, trainData)
      val testResult = Utils.evaluate(model, testData)
      (lambda, trainResult, testResult)
    }

    Utils.writeToFile("output/StudentOutput-NaiveBayes.csv", output)
  }

}
```

Here are the results, which show the effect of the lambda parameter on **Root Mean Square Error** (**RMSE**):



Next, we will implement a `DecisionTree` classifier as well as a `DecisionTree` regressor for the same problem and see how they perform on this problem.

```
val nbClassifier = {
  val lambda = 0.5
  val model = NaiveBayes.train(trainData, lambda = lambda)
  val trainResult = Utils.evaluate(model, trainData)
  val testResult = Utils.evaluate(model, testData)
  (0.0, trainResult, testResult)
}

val dtreeClassifier = {
  val numClasses = 21
  val categoricalFeaturesInfo = Map[Int, Int]()
  val impurity = "gini"
  val maxDepth = 5
  val maxBins = 32
  val model = DecisionTree.trainClassifier(trainData, numClasses, categoricalFeaturesInfo, impurity, maxDepth, maxBins)
  val trainResult = Utils.evaluate(model, trainData)
  val testResult = Utils.evaluate(model, testData)
  (1.0, trainResult, testResult)
}

val dtreeRegressor = {
  val categoricalFeaturesInfo = Map[Int, Int]()
  val impurity = "variance"
  val maxDepth = 5
  val maxBins = 32
  val model = DecisionTree.trainRegressor(trainData, categoricalFeaturesInfo, impurity, maxDepth, maxBins)
  val trainResult = Utils.evaluate(model, trainData)
  val testResult = Utils.evaluate(model, testData)
  (2.0, trainResult, testResult)
}
```

Here are the results of the three different algorithms:

Algorithm	**Train accuracy**	**Train RMSE**	**Test accuracy**	**Test RMSE**
Naive Bayes	0.361963190184049	5.58847777526052	0.249169435215947	5.70364457709227
D tree regression	0.049079754601227	2.05902978183439	0.026578073089701	1.9848078258
D tree classification	0.3865030675	3.9249086582	0.3621262458	4.5724147607

As you can observe, the train accuracy for the regression tree is very low but that is because it is doing a one-to-one comparison. Since the regression tree will generate a real valued output, it is likely that the label will be a mismatch by a very small value but will still fail the equals comparison. However, when we look at the RMSE for both train and test data, we observe a significant decrease in the overall error. So for this dataset, it seems like regression is the correct algorithm to choose. Also, we should notice that the grades have a default ordering, that is a grade with higher value means a bigger grade. However, for problems where there is no implicit ordering among different class labels, we should go for a classification algorithm.

The next dataset that we will explore is *TV News Channel Commercial Detection Dataset Data Set* from here: `https://archive.ics.uci.edu/ml/datasets/TV+News+Channel+Commercial+Detection+Dataset`.

Let's download the dataset into the `datasets` folder inside the codebase using the following commands:

```
$ cd code/datasets/
$ wget -c https://archive.ics.uci.edu/ml/machine-learning-
databases/00326/TV_News_Channel_Commercial_Detection_Dataset.zip
$ mkdir TVData/
$ cd TVData/
$ unzip ../TV_News_Channel_Commercial_Detection_Dataset.zip
```

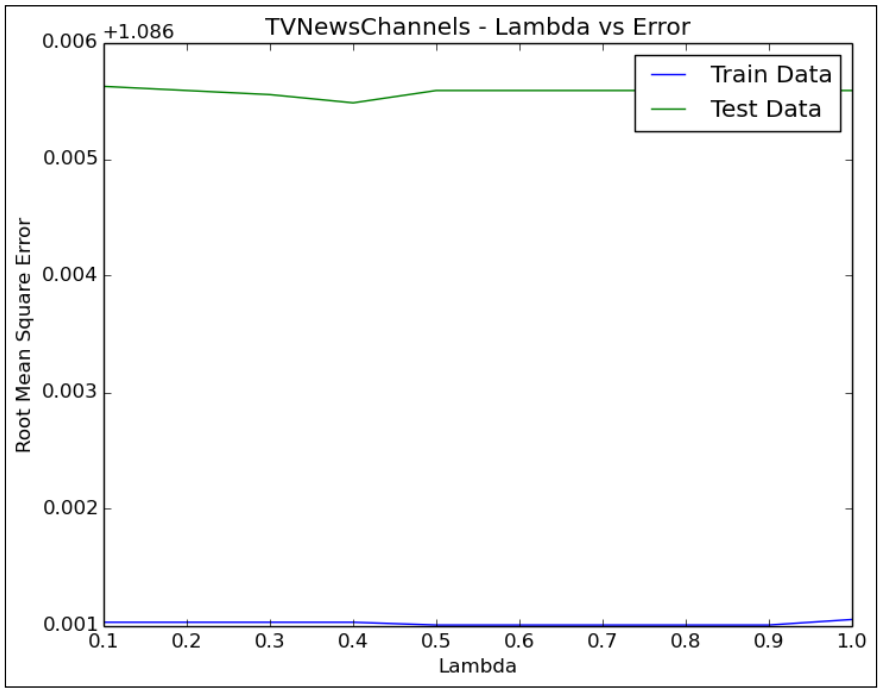
This dataset is available to us in the `LibSVM` format so it is fairly easy to load into Spark. We have data for five news channels—*CNN IBN*, *NDTV 24X7*, *Times Now*, *BBC*, and *CNN*. The data for each of the channels is present in a separate text file, so in total we have five text files (`LibSVM` format). We don't need to do any processing except to concatenate them. The class labels are binary so we can use any binary classification algorithm: SVM, Naive Bayes, logistic regression, and so on. The total dataset size is 191 MB uncompressed.

For this problem, we will first use the Naive Bayes binary classifier. The code is exactly the same as the one we used for the student performance dataset.

Its code is as follows:

```
$ cd code/
$ sbt "run-main chapter04.TVNewsChannelsNaiveBayes datasets/TVData/"
... OUTPUT SKIPPED ...
$ python scripts/ch04/student-grade-plots.py
```

Here are the results with varying lambda values:



The change in error is not significant. The train error drops in the range 0.4 to 0.9 and then rises again from 0.9 to 1.0. The test error consistently drops for values of lambda from 0.1 to 0.4 and then increases. So, if we were to choose a lambda value, we could choose it somewhere near 0.4.

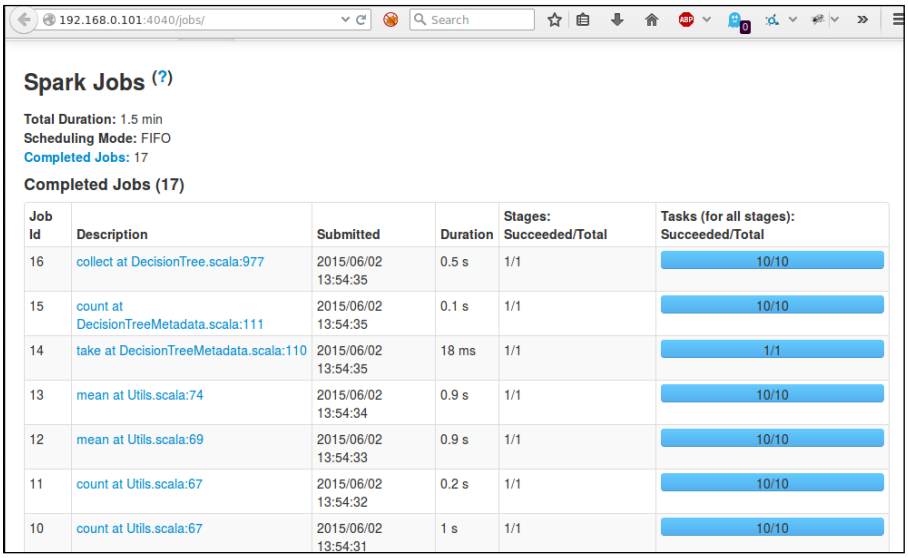
Now let's look at how this problem behaves when using a decision tree regression algorithm.

```
val nbClassifier = {
  val lambda = 0.5
  val model = NaiveBayes.train(trainData, lambda = lambda)
  val trainResult = Utils.evaluate(model, trainData)
  val testResult = Utils.evaluate(model, testData)
  (0.0, trainResult, testResult)
}


val dtreeRegressor = {
  val categoricalFeaturesInfo = Map[Int, Int]()
  val impurity = "variance"
  val maxDepth = 5
  val maxBins = 32
  val model = DecisionTree.trainRegressor(trainData, categoricalFeaturesInfo, impurity, maxDepth, maxBins)
  val trainResult = Utils.evaluate(model, trainData)
  val testResult = Utils.evaluate(model, testData)
  (2.0, trainResult, testResult)
}
```

Its code is as follows:

```
$ cd code/
$ sbt "run-main chapter04.TVNewsChannelComparison datasets/TVData/"
... OUTPUT SKIPPED …
$ python scripts/ch04/tv-news-plots.py
```

This code was run on a single machine so it took some time. We can also take a look at the progress using Spark UI:



Here are the final results:

Algorithm	Train accuracy	Train RMSE	Test accuracy	Test RMSE
Naive Bayes classifier	0.7046054071	1.0870043108	0.7021088278	1.0915881498
Decision tree classifier	0.00018015	0.6698337317	9.62056491957208E-005	0.6739810325

As we observed with the student dataset, the train/test accuracy is very low (the reason was also mentioned earlier). Again, we can see a significant increase in the RMSE values.

Clustering

Clustering is a form of unsupervised learning where the task of the learning algorithm is to find some structure in the given dataset. In particular, a notion of similarity or distance among different instances of a dataset is used to learn such clusters. Spark provides K-Means, **expectation-maximization** (**EM**), **power iteration clustering** (**PIC**), **Latent Dirichlet Allocation** (**LDA**), and streaming K-Means.

K-Means

K-Means is one of the most popular clustering algorithms in which we pre-determine the parameter K—the number of clusters. Or in a more formal way we can define K-Means as a prototype-based, partitional clustering technique that attempts to find a user-specified number of clusters (*K*) which are represented by their centroids. In K-Means, the centroid of a cluster is a prototype that best distinguishes the whole cluster. Since K-Means partitions the points based on the distance between them, we must have a data representation in which it is easy to calculate the distance between the two instances; there must be a distance metric. Apache Spark uses Lloyd's K-Means algorithm with Euclidean distance metric, however, we cannot specify any other distance function at the moment. We must also note that K-Means always produces clusters in a way that an instance will be part of only one of the *K* clusters; clusters are distinct.

Expectation-maximization

The **Gaussian Mixture Model** (**GMM**) allows us to model the clustering in a way that we can produce fuzzy clusters. In fuzzy clusters an instance of a data point can be part of multiple clusters. Apache Spark provides the **expectation-maximization** (**EM**) algorithm for this fuzzy clustering, with a fixed *K* number of clusters. However, note that the EM algorithm does a **maximum-likelihood estimation** (**MLE**) of model parameters so as to maximize the likelihood of a cluster membership given a data point.

Power iteration clustering

In contrast with other clustering algorithms, **power iteration clustering** (**PIC**) works on graphs where for the graph vertices; we have to provide a similarity score between vertices (or the edge weights). The higher the edge weight, the stronger the similarity; edge weight must be non-negative and an edge must not repeat. This algorithm takes a parameter *K* for the number of clusters to generate.

Latent Dirichlet Allocation

Latent Dirichlet Allocation (**LDA**) is a topic modeling algorithm for text documents. This algorithm also takes a parameter *K* for the number of topics to generate. This algorithm is suitable for clustering text documents like news articles, research papers, books, and so on.

LDA example

All the previous clustering algorithms we saw earlier, take a whole set of data points and then generate clusters. However, for the cases where we want our clusters to be updated as soon as the new data arrives, dynamically, we can use streaming K-Means algorithm. It is essentially the same K-Means algorithm, with two additions: it performs clustering on mini-batches and also has an update rule to update cluster centers with newly learned results.

We will take one example of clustering, and for this we will see the BBC news dataset at `http://mlg.ucd.ie/datasets/bbc.html`:

- Consists of 2225 documents from the *BBC* news website corresponding to stories in five topical areas from 2004-2005
- There are five class labels (business, entertainment, politics, sport, and tech)

Since this is a text clustering problem, we first need to extract features from text data. We can do this by performing tokenizing and then some cleaning on the terms. This can also include advanced NLP techniques (NER, stemming, acronyms, and so on), but we will use simple rules and regular expressions to perform tokenization using the following code:

```
def loadData(dataPath: File): Array[(String, Array[String])] = {
  val categories = Array("business", "entertainment", "politics", "sport", "tech")
  val allData = (categories).flatMap { category =>
    val folder = new File(dataPath, category)
    println(folder.getAbsolutePath())
    folder.listFiles.map { docf =>
      val docid = docf.getName().replace(".txt", "")
      println(docf.getAbsolutePath())
      val lines = Source.fromFile(docf, "latin1").getLines.mkString(" ")
      val llines = lines.toLowerCase()
      val words = llines
        .split(" ")
        .flatMap(_.split("(\"|\\'|\\,|\\.|\\?|\\!)+"))
        .filter(_.length() > 1)

      val content = words
      docid -> content
    }
  }
  // allData foreach println
  allData
}
```

After we have mapped our text documents into tokens, we need to calculate TFs. For this we will use the `HashingTF` class provided by Apache Spark. `HashingTF` expects a RDD of `Seq [String]`, which we can obtain from the tokenized documents. Its code is as follows:

```
val defaultPath = "datasets/bbc"
val dataPath = new File(if (args.length > 0) args(0) else defaultPath)
val textData = loadData(dataPath)

val conf = new SparkConf(false).setMaster("local[2]").setAppName("BBCNews")
val sc = new SparkContext(conf)

val documents: RDD[Seq[String]] = sc.parallelize(textData.map(_._2.toSeq).toSeq)
val hashingTF = new HashingTF()
val tf: RDD[Vector] = hashingTF.transform(documents)
```

Finally, we run the LDA algorithm on the term-frequency vectors and obtain a topic-term distribution. In the following code, we only write to 10 terms IDs with their scores:

```
val tf: RDD[Vector] = hashingTF.transform(documents)

// Load and parse the data
// Index documents with unique IDs
val corpus = tf.zipWithIndex.map(_.swap).cache()
// Cluster the documents into three topics using LDA
val ldaModel = new LDA().setK(5).run(corpus)
println("Learned topics (as distributions over vocab of " + ldaModel.vocabSize + " words):")
val topics = ldaModel.topicsMatrix
val N = 5
for (topic <- 0 until 5) {
  print("Topic " + topic + ": ")
  val treeMap = new TreeMap[Double, Int]()
  for (word <- 0 until ldaModel.vocabSize) {
    val score = topics(word, topic)
    if (treeMap.keySet().size() < N) treeMap.put(score, word)
    else {
      val low = treeMap.firstKey()
      if (score > low) {
        treeMap.remove(low); treeMap.put(score, word)
      }
    }
  }
  treeMap.keySet() foreach { k =>
    print(s" (${treeMap(k)},$k)")
  }
  println()
}
```

Here is the output with the top five terms with their score calculated by the LDA algorithm. These scores are drawn from the term distribution for each topic:

```
$ sbt "run-main chapter04.TextLDA"

... OUTPUT SKIPPED ...

Topic 0: (3365,3913.65720852964) (3543,4326.574428888365)
(3707,4535.491948332051) (96727,4778.215907939703)
(114801,11964.705438792551)

Topic 1: (3365,2396.035833693271) (96727,3298.3282136942335)
(3543,3527.026301333385) (3707,4862.134970235111)
(114801,8261.274505002908)

Topic 2: (96727,2850.4724248980783) (3543,2952.916293771327)
(3707,3792.9462669647196) (3365,3998.609192912204)
(114801,9039.540687387185)

Topic 3: (96727,4008.45003946005) (3365,4435.302735745526)
(3543,5767.906047678759) (3707,6886.8060016365625)
(114801,14050.285613710528)

Topic 4: (3365,2877.395029119364) (3543,3382.5769283281697)
(96727,3627.5334140079353) (3707,4913.620812831554)
(114801,9254.193755106819)
```

Association analysis

Association analysis is the process of discovering interesting relationships hidden in large datasets. This is an interesting relationship that can be discovered in the form of association rules or frequent items that occur together. Spark provides the FPGrowth algorithm for finding frequent itemsets.

Frequent pattern mining (FPGrowth)

The FPGrowth algorithm creates a compact data structure called an FPtree and then it can extract frequent items from this data structure. Let's see an example of how to use the FPGrowth algorithm:

```
package chapter04

import scala.io.Source
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.mllib.fpm.FPGrowth

object MarketBasketAnalysis {

  def loadData(dataFile: String): Seq[Array[String]] = {

  def main(args: Array[String]): Unit = {
    val defaultFile = "datasets/marketbasket.csv"
    println(args.toList)
    val dataFile = if (args.length > 0) args(0) else defaultFile
    val data = loadData(dataFile)
    val conf = new SparkConf(false).setMaster("local[2]").setAppName("MarketBasket")
    val sc = new SparkContext(conf)
    val transactions = sc.parallelize(data).cache()
    val fpg = new FPGrowth().setMinSupport(0.05)
    val model = fpg.run(transactions)
    // only those itemsets which have 2 or more items together
    model.freqItemsets.filter(_.items.size > 1).collect().foreach { itemset =>
      println(itemset.items.mkString("[", ",", "]") + ", " + itemset.freq)
    }
  }
}
```

Download the dataset into the `datasets` folder from the following website:

`https://sites.google.com/a/nu.edu.pk/tariq-mahmood/teaching-1/fall-12---dm/marketbasket.csv?attredirects=0&d=1`

Now invoke the following program:

```
$ sbt "set fork := true" "run-main chapter04.MarketBasketAnalysis"
... OUTPUT SKIPPED ...
[info] [2pct. Milk,White Bread], 70
[info] [2pct. Milk,Eggs], 71
[info] [White Bread,Eggs], 75
[info] [Potato Chips,White Bread], 70
```

From the preceding output we can see, that this makes perfect sense. Anyone who buys milk will also likely buy either bread or eggs and in this case we have the data to back up our conclusions.

Summary

In this chapter, we discussed several ML algorithms, especially in the context of Apache Spark. We also saw a few examples of how to use Apache Spark to run these algorithms on different datasets. This will be a good foundation for us to create our first recommendation algorithm in the next chapter, where we will create our first implementation of a recommender system.

5
Recommendation Engines and Where They Fit in?

In the earlier chapters, we set the stage for creating our implementations of recommendation engines. In this chapter, we will implement our first recommender system on a products dataset. Here we continue with the following topics:

- Populate an Amazon dataset
- Create a web app with user/product pages
- Add recommendation pages
- Add product and customer trends

Populating the Amazon dataset

Let's start by downloading the SNAP Amazon dataset from this page `https://snap.stanford.edu/data/amazon-meta.html`. First go to the `datasets` folder, download the file, and decompress it. Note that this file is about 2002 MB compressed and 933 MB uncompressed; it could take some time to download:

```
$ cd datasets/
$ wget -c https://snap.stanford.edu/data/bigdata/amazon/amazon-meta.txt.gz
$ gunzip amazon-meta.txt.gz
```

A single product entry in this file looks like this:

```
Id:  1
ASIN: 0827229534
  title: Patterns of Preaching: A Sermon Sampler
  group: Book
```

```
salesrank: 396585
similar: 5 0804215715 156101074X 0687023955 0687074231
082721619X
categories: 2
|Books[283155]|Subjects[1000]|Religion &
Spirituality[22]|Christianity[12290]|Clergy[12360]|
Preaching[12368]
|Books[283155]|Subjects[1000]|Religion &
Spirituality[22]|Christianity[12290]|Clergy[12360]|
Sermons[12370]
reviews: total: 2 downloaded: 2 avg rating: 5
2000-7-28 cutomer: A2JW67OY8U6HHK rating: 5 votes: 10
helpful:  9
2003-12-14 cutomer: A2VE83MZF98ITY rating: 5 votes: 6
helpful:  5
```

The fields to note are `Id`, `ASIN`, `title`, `salesrank`, `group`, `similar`, `categories` and `reviews`. We have already created code using Scala libraries to parse this file to make it easy for you to process it. We use the ReactiveMongo driver, and Play JSON library, and a custom parser script to scan through the data file:

```
package chapter05

import play.api.libs.json.Json

import reactivemongo.bson.Macros

case class Review(date: String, customer: String,
  rating: Int, votes: Int, helpful: Int)

object Review {
  implicit val reviewHandler = Macros.handler[Review]
  implicit val reviewFormat = Json.format[Review]
}

case class OverallReview(total: Int,
  downloaded: Int, averageRating: Double)

object OverallReview {
  implicit val categoryHandler = Macros.handler[OverallReview]
  implicit val categoryFormat = Json.format[OverallReview]
}

case class Category(name: String, code: Int)

object Category {
  implicit val categoryHandler = Macros.handler[Category]
  implicit val categoryFormat = Json.format[Category]
}
```

```
case class AmazonMeta(
  var id: Int,
  var asin: String,
  var title: String,
  var group: String,
  var salesrank: Int,
  var similar: List[String],
  var categories: List[List[Category]],
  var reviews: List[Review],
  var overallReview: OverallReview)

object AmazonMeta {
  implicit val amazonRatingHandler = Macros.handler[AmazonMeta]
  implicit val amazonRatingFormat = Json.format[AmazonMeta]
}
```

Notice how we have used Play JSON API, ReactiveMongo macros and implicit values in companion objects. This allows us to transform JSON to `Case` class and vice-versa seamlessly, without writing any extra code.

Next we put all these entries into a MongoDB collection. We will call the database `amazon_dataset` and the collection products. As you can see in the following code (see `LoadAmazonDataset` Scala object in this chapter), we are using ReactiveMongo to utilize its JSON macros. This saves us from typing a lot of boilerplate code for JSON conversion. With a bunch of regular expressions and a simple iteration we can process the full text file. We only need to be careful with the end of file. The following is a snippet of regular expressions that have been used:

```
val driver = new MongoDriver
val connection = driver.connection(List("localhost"))

val db = connection("amazon_dataset")
val ratingCollection = db[BSONCollection]("products")

val src = Source.fromFile(file)
val idR = "\\s*Id:\\s+(.+)".r
val asinR = "\\s*ASIN: (.+)".r
val titleR = "\\s*title: (.+)".r
val groupR = "\\s*group: (.+)".r
val salesrankR = "\\s*salesrank: (.+)".r
val similarR = "\\s*similar: (.+)".r
val categoriesR = "\\s*categories: (.+)".r
val reviewsR = "\\s*reviews: (.+)".r
val reviewCountR = "\\s*total:\\s*(\\d+)\\s*downloaded:\\s*(\\d+)\\s*avg rating:\\s*(.*)".r
val reviewLineR = "\\s*(.*?)\\s+cutomer:\\s+(.*?)\\s+rating:\\s+(.*?)\\s+votes:\\s+(.*?)\\s+helpful:\\s+(.*?)".r
val lIter = src.getLines
```

Before executing the following command, make sure that you have started a MongoDB server on your local machine:

```
$ sbt "run-main chapter05.LoadAmazonDataset datasets/amazon-dataset/
amazon-meta.txt"
```

It will take some time to load all the product entries into MongoDB. Meanwhile, you can also check your MongoDB instance:

```
$ mongo
> use amazon_dataset
> db.products.findOne()
{
    "_id" : ObjectId("5579f5bd1d00001d002109dd"),
    "id" : 1,
    "asin" : "0827229534",
    "title" : "Patterns of Preaching: A Sermon Sampler",
    "group" : "Book",
    "salesrank" : 396585,
    "similar" : ["0804215715", "156101074X", "0687023955",
    "0687074231", "082721619X"],
    "categories" : [
        [
            {"name" : "Books", "code" : 283155},
            {"name" : "Subjects", "code" : 1000},
            {"name" : "Religion & Spirituality", "code" : 22},
            {"name" : "Christianity", "code" : 12290},
            {"name" : "Clergy", "code" : 12360},
            {"name" : "Preaching", "code" : 12368}
        ],
        [
            {"name" : "Books", "code" : 283155},
            {"name" : "Subjects", "code" : 1000},
            {"name" : "Religion & Spirituality", "code" : 22},
            {"name" : "Christianity", "code" : 12290},
            {"name" : "Clergy", "code" : 12360},
            {"name" : "Sermons", "code" : 12370}
        ]
    ],
    "reviews" : [
        {"date" : "2000-7-28", "customer" : "A2JW67OY8U6HHK",
        "rating" : 5, "votes" : 10, "helpful" : 9},
```

```
        {"date" : "2003-12-14", "customer" : "A2VE83MZF98ITY",
        "rating" : 5, "votes" : 6, "helpful" : 5}
    ],
    "overallReview" : {"total" : 2, "downloaded" : 2, "averageRating" :
5}
}
```

You may also want to check the different product groups present in the whole dataset:

```
> db.runCommand({distinct: "products", key: "group"})
{
    "values" : [
        "Book",
        "Music",
        "DVD",
        "Video",
        "Toy",
        "Video Games",
        "Software",
        "Baby Product",
        "CE",
        "Sports"
    ],
    "stats" : {
        "n" : 533023,
        "nscanned" : 533023,
        "nscannedObjects" : 533023,
        "timems" : 614,
        "cursor" : "BasicCursor"
    },
    "ok" : 1
}
```

Hopefully, if all goes well our dataset is populated into MongoDB. Next we create a web-interface to explore this data. We will also add two specialized pages for *most popular* and *top-rated* products.

Let's also populate some fake customer data. For this we will use the Fake Name Generator website, which is free and very convenient to use. Visit this page `http://www.fakenamegenerator.com/order.php` (bulk order), then select all the fields, and ensure that you put in an order for 50,000 entries. After a while you will receive a link to download the randomly generated data. Also make sure that if you publish that data, then you must follow the guidelines of the GPL v3 / Creative Commons licenses (see `readme.txt` in the file you receive).

So if you receive the file named `FakeNameGenerator.com_data.zip`, we will proceed as follows:

```
$ unzip FakeNameGenerator.com_qw3r7y.zip
```

This will extract a CSV file and a `readme.txt` (this contains the license information I mentioned earlier). Now we will import this data directly into a MongoDB collection. Let's call the collection `customers`.

```
$ mongoimport -d amazon_dataset -c customers --type csv --file ./
FakeNameGenerator.com_qw3r7y.csv --headerline
connected to: 127.0.0.1
2015-06-27T18:34:34.462+0530 check 9 50001
2015-06-27T18:34:34.705+0530 imported 50000 objects
```

Now that the data is imported into MongoDB, we can query for some data. Let's see how `Gender` is encoded in this dataset:

```
$ mongo
> use amazon_dataset
> db.runCommand({distinct: "customers", key: "Gender"})
{
    "values" : [
        "female",
        "male"
    ],
    "stats" : {
        "n" : 50000,
        "nscanned" : 50000,
        "nscannedObjects" : 50000,
        "timems" : 39,
        "cursor" : "BasicCursor"
    },
    "ok" : 1
}
```

As we can see the encoding as `male`/`female` strings, we can explore other fields to decide how we will encode these in Scala. Now also find out the `NameSet` column, which essentially represents what kind of a name a record represents:

```
> db.runCommand({distinct: "customers", key: "NameSet"})
{
    "values" : [
        "Chinese (Traditional)",
        "Russian",
        "Danish",
        "Japanese (Anglicized)",
             ... OUTPUT SKIPPED ...
        "Norwegian",
        "Dutch",
        "Thai"
    ],
    "stats" : {
        "n" : 50000,
        "nscanned" : 50000,
        "nscannedObjects" : 50000,
        "timems" : 38,
        "cursor" : "BasicCursor"
    },
    "ok" : 1
}
```

Our customers' data is set up, however we also need to find a way to map these random names to Amazon dataset customers. Remember that the Amazon dataset has customer reviews with customer IDs. So first, we need to figure out how many customers there are. For that we will split the reviews data into a separate collection called `reviews`.

Create a separate collection for `reviews`:

```
> db.reviews.drop()
true
> db.products.find({}).forEach(  function(doc) {   var rs = doc.reviews;
for(r in rs) {    var elem = rs[r];    elem.asin = doc.asin;    elem.
productId = doc.id;    elem.date = new Date(elem.date);    db.reviews.
insert(elem);   }  }  );
```

```
> db.reviews.count()
6874336
```

As we can see in the above output, there are 6,874,336 reviews in total. We will also add an index to the `customer` column:

```
> db.reviews.createIndex( { "customer": 1 } )
```

All right, so now we have a separate collection `reviews`, and we will also create a mappings collection between the Amazon customer ID and a random user. So first we extract a `customer_mapping` collection with the actual customer IDs and then assign a random number between 0 and 50,000 to each of these customers. This is because in our random users' dataset, we have only 50,000 users but the actual customers are far more than that as we will see now:

```
> db.reviews.aggregate({$group: {_id: "customer"}}, {$out: "customer_
mapping"})
> db.customer_mapping.count()
1510224
> db.customer_mapping.find().forEach(
...  function(doc) {
...   var custNum = Math.floor(Math.random() * db.customers.count());
...   db.customer_mapping.update(
...      {_id: doc._id},
...      {$set: {customer_number: custNum}}
...     )
...  })
```

Finally, we have the mapping complete. Although not perfect, it will be good enough for us to create a nice web application for our first recommendation project. That's exactly what we will do next.

Creating a web app with user/product pages

Now that we have our data ready, let's create a simple UI to display it to the user. In this section, we will be creating a web application using the Play Scala framework. Check out a Play Scala tutorial at `https://www.typesafe.com/activator/template/play-scala-intro` before you continue.

Creating a Play framework application

From a Scala developer, who knows how to use SBT, a Play framework application is no different than a normal SBT project. Essentially either you can use SBT or you can use Activator (for recent Play projects).

We have already prepared the application for you, located in the `webapp-recommender` folder in this book's source code. Let's go to that folder and see what we have in the project:

```
$ cd webapp-recommender/
$ ls -ng
total 44
drwxrwxr-x 7 1000 4096 Jun 22 19:52 app
-rw-rw-r-- 1 1000 1028 Jun 16 16:12 build.sbt
drwxrwxr-x 2 1000 4096 Jun 16 14:23 conf
drwxrwxr-x 4 1000 4096 Jun 16 01:48 project
drwxrwxr-x 4 1000 4096 Jun 21 20:09 public
-rw-rw-r-- 1 1000  27 Jun 16 01:42 README.md
drwxrwxr-x 2 1000 4096 Jun 11 10:44 test
```

Notice that there is a `build.sbt` file, which is the build definition file. There are other folders `app`, `conf`, `public`, and so on which define how the web application will behave. To run the web application you need to invoke the following command (we are not covering production level setup here):

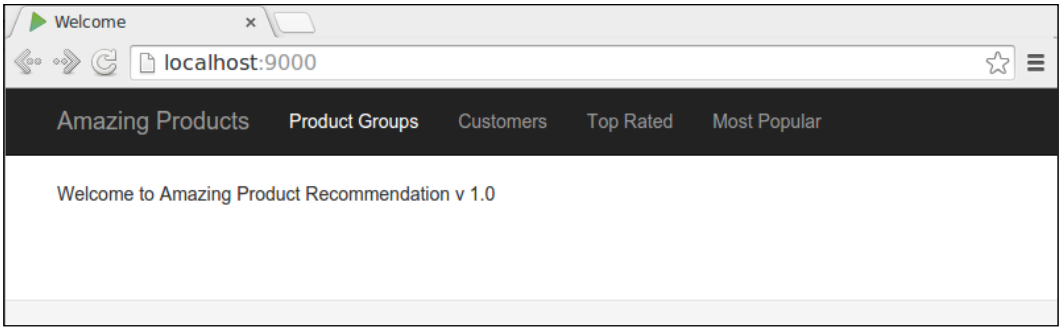
```
$ sbt run
[info] Loading global plugins from ...
[info] Loading project definition from ...
[info] Set current project to webapp-recommender ...
--- (Running the application from SBT, auto-reloading is enabled) ---
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:0:9000
(Server started, use Ctrl+D to stop and go back to the console...)
```

It could take some time to download all the dependencies so please bear with the network speed.

This will start a HTTP server on your machine running on port 9000. Open this link: `http://localhost:9000/`. What do you see?

The home page

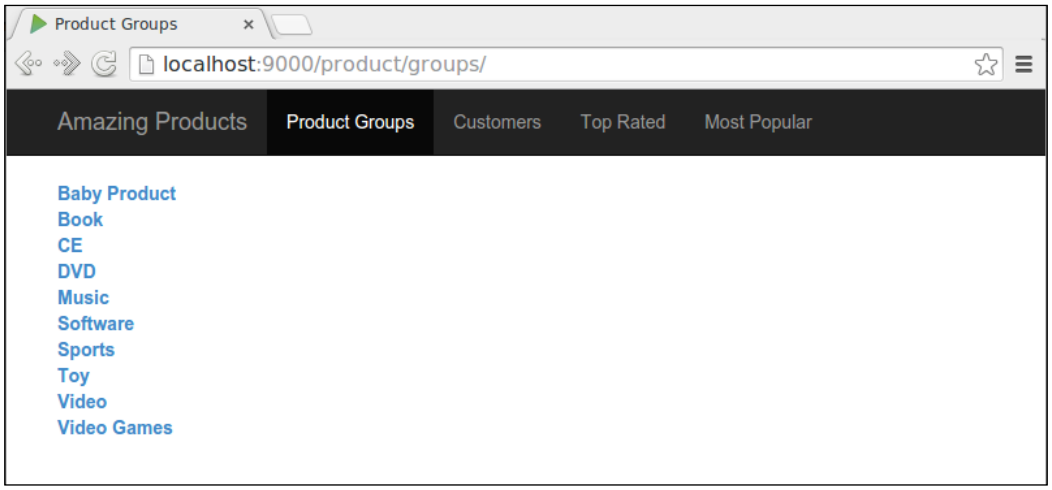
Well first up is the home page, which should be self explanatory:



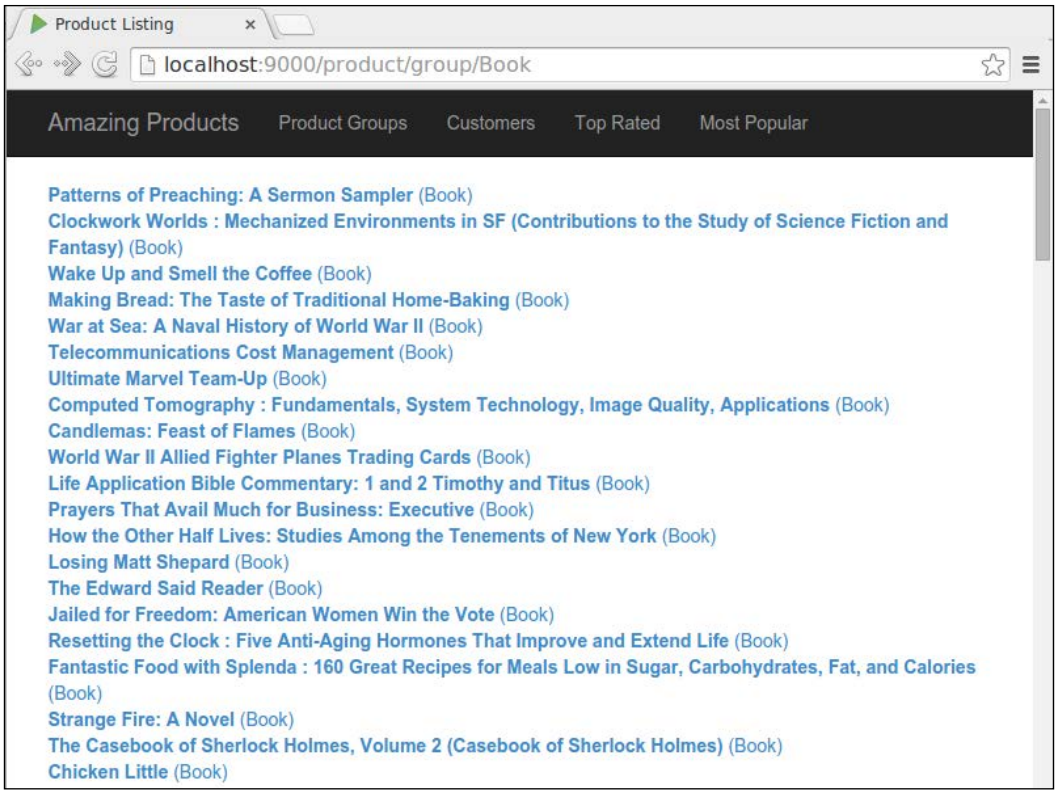
Go back to your MongoDB console and notice that every product is assigned to a group such as DVD, Book, and so on. So the next view we have is for the different product groups.

Product Groups

Click on **Product Groups** from the navigation bar:



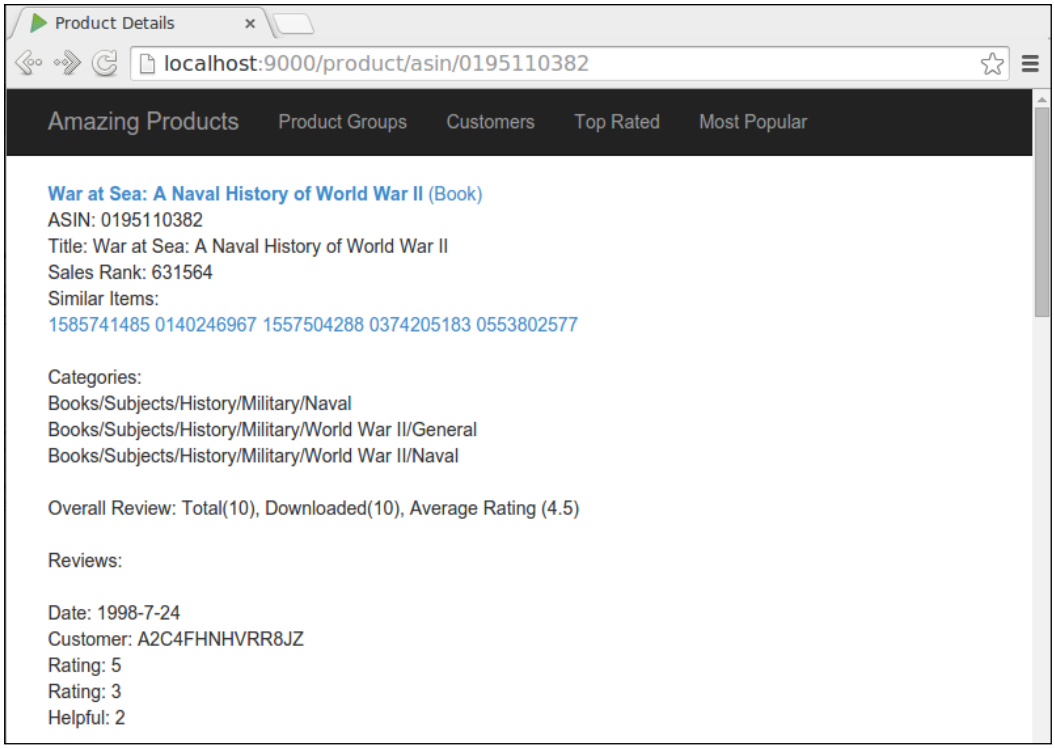
Next if you click on a group, say **Book**, you will land on **Product Listing** for that group. This shows only the first 100 products as shown in the following screenshot:



Further down if you click on a book named **War at Sea: A Naval History of World War II (Book)**, we have what we call the product view.

Product view

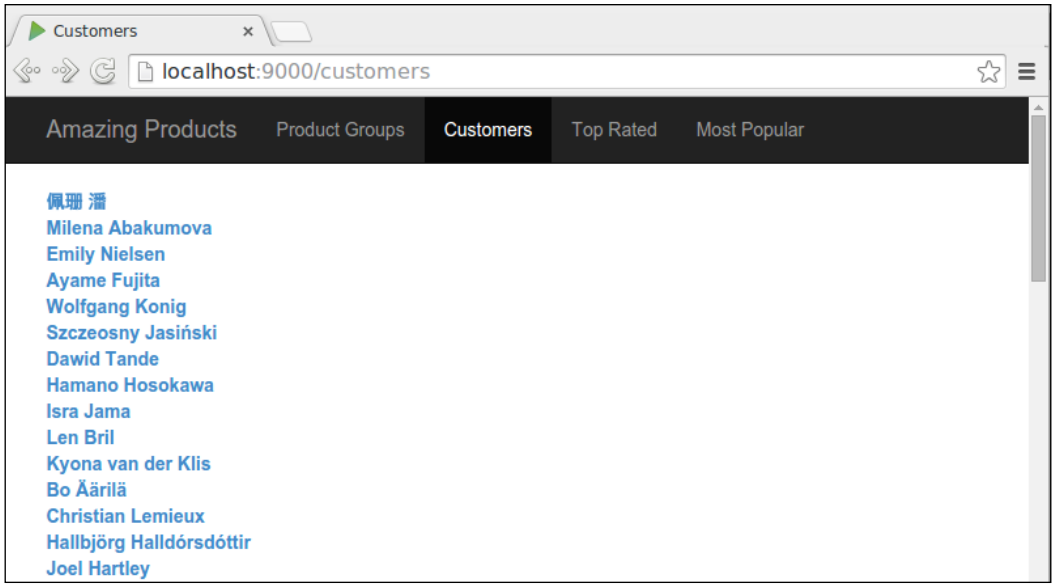
In this view, we see the details of a product along with its overall rating and reviews:



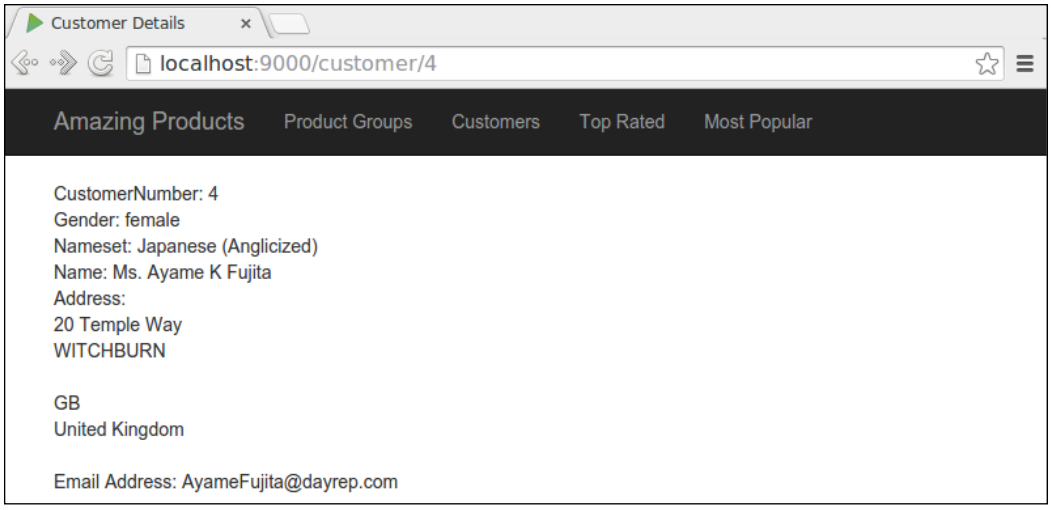
Finally, for the sake of completeness we have created a basic user listing, which only enlists the customers. Again, this shows only 100 customers.

Customer views

The following view shows the **Customers** listing:



The following view shows selected customer details:



Until now we have used a very simple web app to display selective data on the browser. Now it's time to add two additional views for persistent and automatic user recommendations. Recall from our discussion of the recommender interface and recommendation technology, that we discussed the top *N* list and aggregated rating, and so on. We will use those in the following sections.

Adding recommendation pages

All the recommendations we will add are global recommendations, that is, they will be the same for any user arriving on the website. Next we will add these views: **Top Rated**, **Most Popular**, and **Monthly Trends**.

The Top Rated view

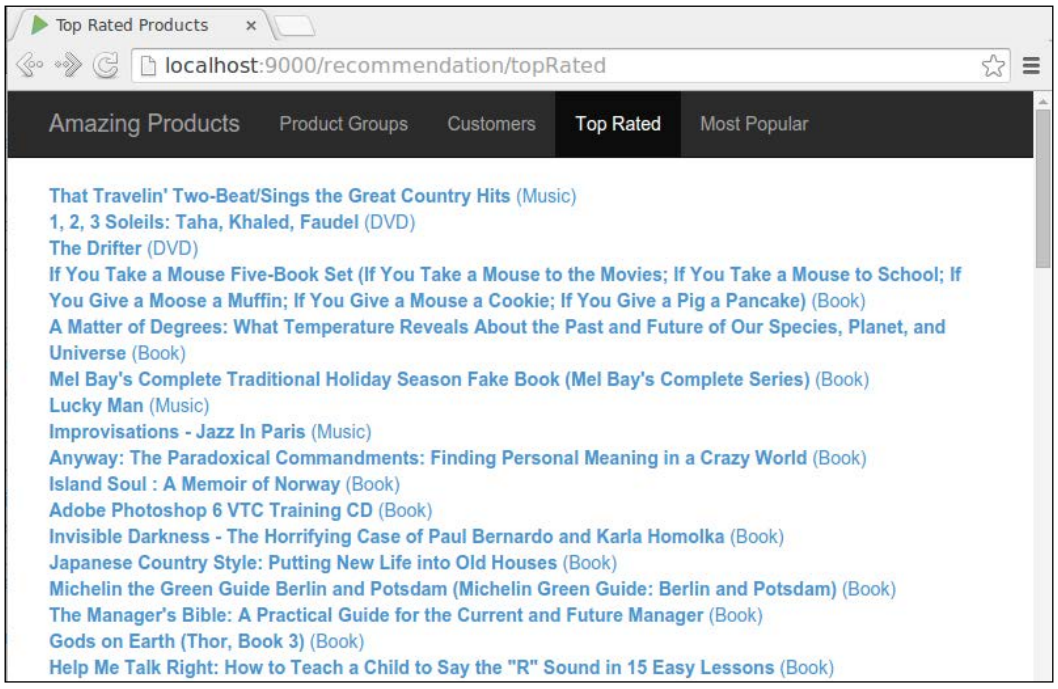
This view will show only the first 100 products with the highest average ratings. Since we already have the average rating available in our dataset, we can make a simple database query. This is exactly what we will do. Check out the file named `webapp-recommender/app/util/ReactiveDB.scala`, which contains the following code:

```
def topRatedProducts() = {
  val query = BSONDocument.empty
  val sortCriteria = BSONDocument("overallReview.averageRating" -> -1)
  val collection = productCollection()
  val cursor = collection.find(query).sort(sortCriteria).cursor[AmazonMeta]
  cursor
}

def mostPopularProducts() = {
  val query = BSONDocument.empty
  val sortCriteria = BSONDocument("overallReview.total" -> -1)
  val collection = productCollection()
  val cursor = collection.find(query).sort(sortCriteria).cursor[AmazonMeta]
  cursor
}
```

Corresponding to that, we have code in our Play framework's `controller` code (`webapp-recommender/app/controllers/RecommendationController.scala`), which uses it to retrieve top rated products:

```
package controllers

import scala.concurrent.ExecutionContext.Implicits.global

object RecommendationController extends Controller {

  def topRated() = Action.async {
    val itemsF = ReactiveDB.topRatedProducts.collect[Seq](100, true)
    itemsF map { items =>
      Ok(views.html.top_rated_products(items))
    }
  }

  def mostPopular() = Action.async {
    val itemsF = ReactiveDB.mostPopularProducts.collect[Seq](100, true)
    itemsF map { items =>
      Ok(views.html.most_popular_products(items))
    }
  }


}
```

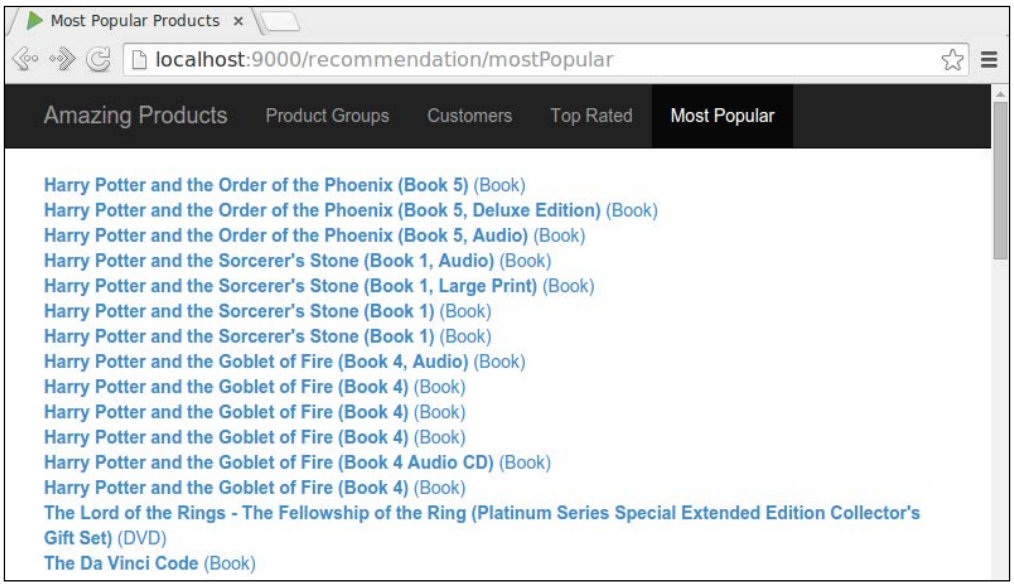
Notice that in the controller we fetch the top rated products and generate corresponding HTML to be displayed in the web browser. This is how the view will look:



Next up we will follow a similar approach to find the most popular products.

The Most Popular view

We just saw the code for top rated products, and for the most popular products it is the same story. We find those products that had the highest number of reviews. This is the strongest indicator that a product is the most popular. In this case, we also only list the first 100 products as show in the following screenshot:



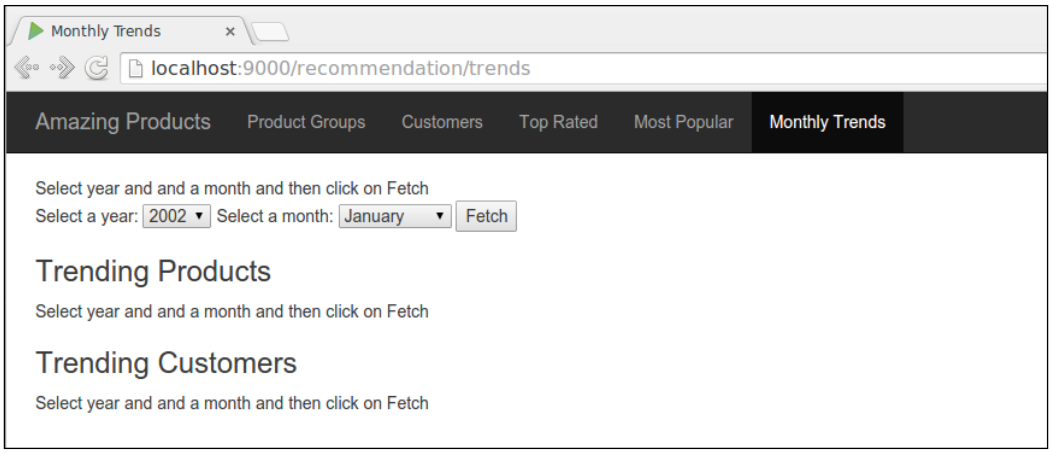
Well, there is a clear difference between top rated and most popular products. While **That Travelin' Two-Beat/Sings the Great Country Hits (Music)** may be very highly rated, it is the *Harry Potter* series that is most popular. The numbers make it clear.

So far we have only populated the dataset, and the data for recommendation was readily available. However, from here on things will not be so easy. For example, let's ask this question "which of the top 10 books have been trending higher in the past 6 months?"

The Monthly Trends view

How do we figure out which products are trending? Well, the idea is very simple. We maintain a sliding window of time *T*, which we will keep as one month in this example. In this window, we maintain a count of all the items during this time period. Whichever items have the highest hits are most likely trending.

Let's first create a view that will allow a user to select the year and a month and then we can proceed with the actual implementation of the trends logic.



We will add three HTTP URLs to the web application (see the `conf/routes` file):

```
GET   /recommendation/trends
GET   /recommendation/trending/product/:year/:month/
GET   /recommendation/trending/customer/:year/:month/
```

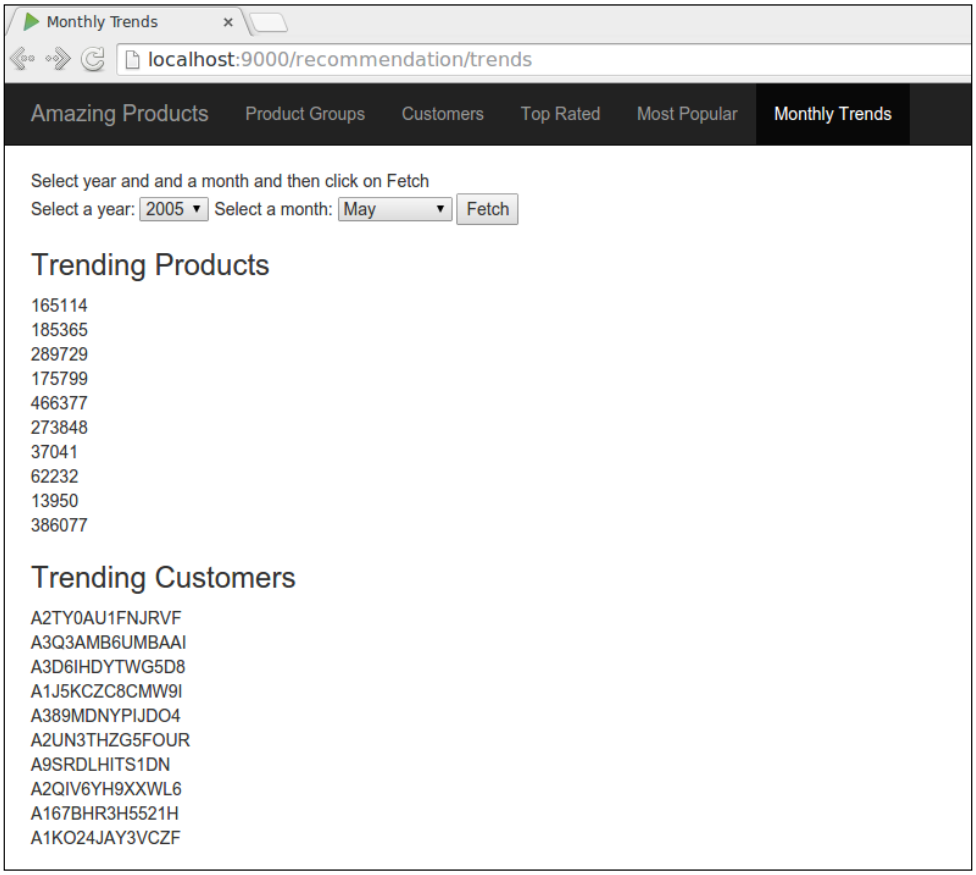
These three URLs will be mapped to `RecommendationController` to fetch the data:

```
controllers.RecommendationController.trends
controllers.RecommendationController.trendingProduct(year: Int, month:
Int)
controllers.RecommendationController.trendingCustomer(year: Int,
month: Int)
```

The algorithm we will use to find the top items for each month is simple. Our main data source is going to be reviews data (which we have already separated into reviews collection). Now we filter out all the reviews for the given year and month. For each of the reviews, we retrieve the product ID and number of votes received by the review. Once we have a list of such entries, we can sort the products by their vote counts.

1. Filter reviews by given year and month for a one month duration.
2. Retrieve (product ID and votes) from each review.
3. Count total votes for each Product ID.
4. Sort them by decreasing order of total votes.
5. Take top 10 of entries as the top trending items.

We do the same for customers as well. Since we have customers making reviews, we can also find the top trending customers from the reviews data. Finally, once we have implemented the algorithm, it works pretty well.



Simply counting values for all the products may not be a feasible option in some situations. The reason could be that either you cannot afford to store all the counts in-memory, or you only receive the data as a stream. In this situation, a clever data-structure called Count Min Sketch could be used to estimate counts of the top *K* items. We have also provided an example of TopCMS from Twitter's algebird library for product trends. The code for this is as follows:

```
def findTrendingCustomersFor(year: Int, month: Int): Future[List[String]] = {
  val cursor = ReactiveDB.oneMonthReviewsFrom(year, month)
  val enumerator = cursor.enumerate()
  val itf: Iteratee[BSONDocument, Map[String, Long]] =
    Iteratee.fold(Map[String, Long]()) {
      (a, doc: BSONDocument) =>
        val itemId = doc.get("customer").map {
          _.asInstanceOf[BSONString].value
        }.get

        val voteCount = doc.get("votes").map {
          _.asInstanceOf[BSONDouble].value.toLong
        }.get

        val newVotes = a.get(itemId).getOrElse(0L)
        a + (itemId -> newVotes)
    }

  val finalCounts = enumerator |>>> itf
  finalCounts.map { cms =>
    cms.toList.sortBy(-_._2).map(_._1).take(10)
  }
}

def findTrendingProductsFor(year: Int, month: Int): Future[List[Long]] = {
  val cursor = ReactiveDB.oneMonthReviewsFrom(year, month)
  if (useCMS) {
    val DELTA = 1E-3; val EPS = 0.01; val SEED = 1; val PERC = 0.001
    val TOPK = 10 // K highest frequency elements to take
    val cms = TopPctCMS.monoid[Long](EPS, DELTA, SEED, PERC)
    val enumerator = cursor.enumerate()
    val itf: Iteratee[BSONDocument, TopCMS[Long]] =
      Iteratee.fold(cms.zero) {
        (a, doc: BSONDocument) =>
          val itemId = doc.get("productId").map {
            _.asInstanceOf[BSONDouble].value.toLong
          }.get
          val voteCount = doc.get("votes").map {
            _.asInstanceOf[BSONDouble].value.toLong
          }.get
          val b = cms.create(itemId)
          val c = b + (itemId, voteCount)
          a ++ c
      }
    val finalCMS = enumerator |>>> itf
    finalCMS.map { cms =>
      cms.heavyHitters.toList
    }
```

Summary

Finally, we come to the conclusion of our first recommender implementation, which only uses the data and statistics readily available from the data. We created a web application that demonstrates the integration of both frontend as well as the backend components. All the recommendations that we generated were global, and any user visiting the site will see the same recommendations. In the next chapter, we will infer more meaning from user activities and engagement with products and reviews and fine-tune the recommendations for each user.

6
Collaborative Filtering versus Content-Based Recommendation Engines

In the last chapter, we created our first implementations of a recommender system. That sets the stage for further improvements. We only provided recommendations that were global in nature. Now we will focus on recommendations that are tuned for the current user. We will discuss the following topics in this chapter:

- Content-based recommendation
- Collaborative filtering based recommendation
- Comparison of the two approaches

Content-based recommendation

The main idea in the content-based recommendation system is to recommend items to a customer *X* similar to previous items rated highly by the same customer *X*. Notice in this definition that we find "similar" items, which means we need to have a measure of similarity between items. To measure similarity of two items we decode item features and then apply a similarity function.

What is a similarity function? A **similarity function** takes two items (or their feature representations) and returns a value that indicates degree of similarity of two items. Now there are many different kinds of similarity functions because there are different ways we can represent an item.

We can represent an item as a set of features. For example, an item has the color red, is square shaped, and is made in India. Another item could be colored red, square shaped, and made in Italy. So we can represent these items as two sets:

- $item_1$ = {color red, square-shaped, made in India}
- $item_2$ = {color red, square-shaped, made in Italy}

So how similar are these two items? We will see that later in this chapter.

We need to have a common representation that is well understood by a similarity function. Also, there can be thousands or millions of features that we can extract for an item. For this, we can use a vector-based representation for an item. Notice that this is different from a set-based representation. So for $item_1$ and $item_2$, with vector representation how do we now calculate the similarity?

Similarity measures

To answer these questions, we have different similarity (or dissimilarity/distance) metrics. Some of these we have already discussed earlier, but let's visit them again here. We will discuss the following similarity metrics:

- Pearson correlation
- Euclidean distance
- Cosine measure
- Spearman correlation
- Tanimoto coefficient
- Log likelihood test

Pearson correlation

The Pearson correlation value is in the range -1.0 to 1.0, where 1.0 indicates a very high correlation or high similarity and -1.0 indicates the opposite or high dissimilarity. As we had seen in an earlier chapter, the Pearson correlation of two series is the ratio of their covariance to the product of their variances. For a detailed review of Pearson correlation, read this Wikipedia article (`https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient`).

Challenges with Pearson correlation

Pearson correlation expects the series to have the same length, and it doesn't take into account the number of features in which two series preferences overlap. Pearson correlation is also undefined if either of the series of feature values are identical because that makes the variance zero. Therefore, the result is undefined.

Euclidean distance

Euclidean distance is the geometric distance between two *n*-dimensional points. So given two series of feature values, we can consider both as vectors and calculate the geometric distance. Note that this is a distance metric but we want a similarity measure. So to convert it into a similarity metric, we can use the following formulas:

$$D(p,q) = D(q,p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

$$S_{Euclidean}(p,q) = 1/(1 + D_{Euclidean}(p,q))$$

Where $S_{Euclidean}$ is Euclidean similarity and $D_{Euclidean}$ = Euclidean distance.

Challenges with Euclidean distance

It only works when the two feature vectors are of the same length. Features with larger values affect the similarity measure more than the features with smaller values.

Cosine measure

Cosine measure or cosine similarity is the cosine of angle between two vectors (or points with respect to origin). The value of cosine ranges from -1.0 to 1.0.

Also note that the meaning of the cosine measure value is exactly similar to Pearson correlation. So we can also use Pearson correlation to the same effect. Therefore the same challenges apply to cosine measure too.

Spearman correlation

Spearman correlation, also called Spearman rank correlation, has already been covered in an earlier chapter. Essentially, Spearman correlation is applicable when we have features in a vector that can be ranked somehow (for example, time and rating both have an implicit ordering). Well, this measure works only if there is an implicit ordering of features. This free video gives a very simple and concise description of calculating Spearman correlation.

Tanimoto coefficient

Tanimoto coefficient, also known as **Jaccard coefficient**, is the measure of overlap between two sets.

TC = intersection of preferences/union of preferences

$$J(A,B) = \frac{n(A \cap B)}{n(A \cup B)}$$

Where $n(A \cap B)$ is number of feature overlap and $n(A \cup B)$ is the joint number of features. The value of this measure is always from 0 to 1. So it is easy to convert into a distance measure using the following formula:

$$D_{jaccard}(A,B) = 1 - J(A,B)$$

Note that this measure is defined for sets, so we need to make an appropriate representation for our item features to make it work. For example, we can have an item vector with 1s wherever a feature is present. So for $item_1$ and $item_2$ examples, which we saw earlier, we can have the following representation:

- $item_1$ = `[1,1,1,0]`
- $item_2$ = `[1,1,0,1]`

Where the universal set would be `[1,1,1,1]` representing [color red, square-shaped, made in India, made in Italy]. So for $item_1$ and $item_2$ the similarity measure is: $J(item_1, item_2) = 2/4 = 0.5$.

Log likelihood test

This is quite similar to the Tanimoto coefficient, which measures an overlap. However, the log likelihood test is an expression of how unlikely users will have so much overlap, given the total number of items and the number of items each user has a preference for.

Two similar users are likely to rate a movie common to them. However, two dissimilar users are unlikely to rate a common movie. Therefore, the more unlikely the two users would rate the same movie, and still rate the same movie, the more similar two users should be. The resulting value can be interpreted as a probability that an overlap isn't just due to chance. You may also want to refer to this video for a nice explanation.

Content-based recommendation steps

We follow these steps to arrive at a mode to make content-based recommendations:

1. Compute vectors to describe items.
2. Building profiles of user preferences.
3. Predicting user interest in items.

First we take our items dataset and identify the features we want to encode for each item. Next we generate a **pretend item** for a user, based on a user's interaction with items. We can use a user's activity with items such as clicks, likes, purchases, and reviews. So essentially each user now encoded has the same features, and is represented the same as other items in the dataset. Therefore we have a set of feature vectors for all items, and also a pretend feature vector for the target user:

```
User -> Likes -> Item profile
```

Now the idea is simple. Apply a similarity function for the user features with all the items, and sort them by the most similar at the top. For example:

Let item feature vectors be $items = v_1, v_2, v_3, \ldots, v_n$, and let the pretend item for user be u. Also, let our similarity function be fs. So in a pseudo-code form, finding top *K* similar items would be:

$items.sortBy(v => -f_s(v,u)).take(K)$ or $items.sortBy(v => f_s(v,u)).reverse.take(K)$

This gives us top *K* items, which are most similar to the pretend item we created for our target user. And that is exactly what we would recommend to the user. However, this is an expensive operation. If you can afford to perform these matches, then it is perfectly fine. However, when the data size is huge, this won't be an option.

To enhance it, we should understand what exactly is happening here. We will discuss that in the next section. Can you think about it on your own for now?

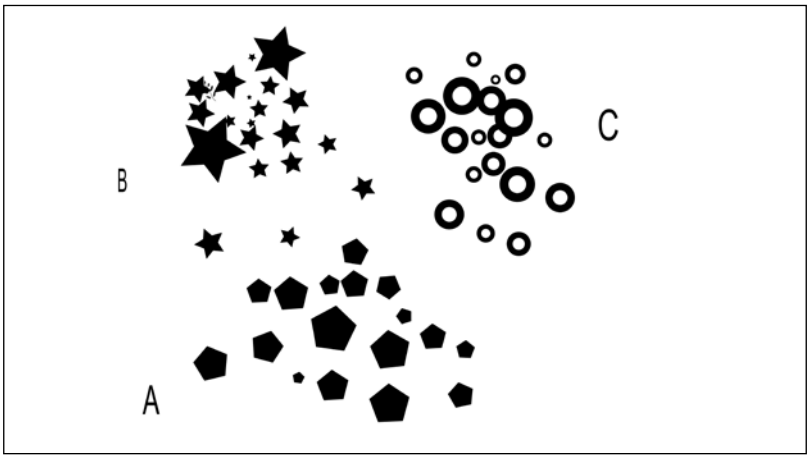
Note here that the key ideas are:

- To model items according to relevant attributes
- To infer user likes by modeling the user as a set of item features, and then use the model built above to make recommendations

Let's now discuss how clustering can give us some performance benefits.

Clustering for performance

In our items and preceding user example, we have to scan all the items every time we find top *K* recommendations for a user. First, we need to choose a similarity function. Then we are essentially finding the items nearest to the user. From these items, we are choosing only the *K* nearest items. But we really don't need to perform similarity calculation of a user with all the items every time. We can pre-process items and cluster them together such that the most similar items are always grouped together. This is where we will use the K-Means clustering algorithm. The K-Means algorithm gives us a nice model to find the closest set of points to a given point using cluster centroids. Here is how it will look:



In this figure, there are three users labeled as **A**, **B**, and **C**. These three points are the pretend points based on their interaction with the system. First, we cluster all the items into three clusters hexagon, star, and ring. Once we have done that, we can find which cluster a user most likely belongs to and only recommend items from that cluster.

We will run our implementation of this recommendation approach on the Amazon dataset. So first let's extract the data for all the items from MongoDB into a CSV file.

```
$ mongo amazon_dataset --quiet < ../scripts/item-features.js > datasets/
content-based-dataset-new.csv
```

Since we need to extract each item as a set of features, we have the following attributes for each item:

- Title
- Group

- Sales rank
- Average rating
- Categories

Let's look at a sample entry:

- **ASIN**: 078510870X
- **Title**: Ultimate Marvel Team-Up
- **Group**: Book
- **Sales rank**: 612475
- **Average rating**: 3.5
- **Categories**: Books::Subjects::Children's Books::Literature::Science Fiction, Fantasy, Mystery & Horror::Comics & Graphic Novels::Books::Subjects::Comics & Graphic Novels::Publishers::Marvel::Books::Subjects::Teens::Science Fiction & Fantasy::Fantasy::Books::Subjects::Comics & Graphic Novels::Graphic Novels::Superheroes

Our first task is to convert each item into a feature vector. For the preceding text attributes, we will use a HashingTF transformer. We set it up with a cardinality of 1024 (number of features) in the following code:

```
val dim = math.pow(2, 10).toInt
val hashingTF = new HashingTF(dim)
```

Notice how we create a user defined function to extract values and create a vector:

```
val featureVectorsUDF = udf[Vector, String, String, String, String, String] {
  (title, group, salesrank, averageRating, categories) =>
    val cats = categories.split("::").map(_.trim).filter(_.length > 0)
    val row = (title, group, cats.toList)

    val titleTerms = titleToTerms(title)
    val allTerms = titleTerms ++ Array(group) ++ cats
    val t1 = allTerms.toSeq
    val vector1 = hashingTF.transform(t1)
    var sr = 0.0
    var ar = 0.0
    try { sr = salesrank.toDouble } catch { case _: Exception => }
    try { ar = averageRating.toDouble } catch { case _: Exception => }
    val aVec = vector1.asInstanceOf[SparseVector]
    val vector2 = Vectors.sparse(dim + 2,
      aVec.indices ++ Array(dim, dim + 1),
      aVec.values ++ Array(sr, ar))
    vector2
}
```

Now based on the limited and mixed data (both numeric and textual), we would still like to be able to extract numeric features, the reason being that K-Means (or any distance based clustering) works only with numeric data. Here is what we will do:

1. We will extract title terms, group and categories terms.
2. We will calculate term frequencies of all these terms using HashingTF.
3. We will create a vector out of these term frequencies and append two features sales rank and average rating to this vector.

For this, we will create a Spark **UDF** (**user defined function**) that will operate very nicely on a dataframe. Once we have converted every item into a vector representation, we learn a K-Means model, evaluate it, update its hyper parameters, and so on. So, finally, when we have obtained a good K-Means model, we will label each of the items with the cluster ID to which they belong.

```
def main(args: Array[String]): Unit = {
  val conf = new SparkConf(false).
    setMaster("local[2]").
    setAppName("ContentBasedRSExample").
    set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  val fileName = "datasets/content-based-dataset-new.csv"
  val df = sqlContext.load("com.databricks.spark.csv",
    Map("path" -> fileName,
      "header" -> "true", "delimiter" -> colSep,
      "quote" -> "\0"))
  df.printSchema()
  val df2 = df.withColumn("features",
    featureVectorsUDF(
      df("title"),
      df("group"),
      df("salesrank"),
      df("averageRating"),
      df("categories")))
  df2.printSchema()
  val itemsRDD: RDD[Vector] = df2.select("features").rdd.
    map(x => x(0).asInstanceOf[Vector])
```

The full code for this implementation is in the `src/main/scala/chapter06/ContentBasedRSExample.scala` file.

First, we load the CSV data into a dataframe, and then transform it into feature vectors using the UDF we defined earlier. Next, we train the model and assign cluster IDs to all the items. Later to make recommendations, we would also need to store the K-Means model to disk. The bad news is this feature is only available in Spark 1.4 and we are using Spark 1.3 (see SPARK-5986 at `https://issues.apache.org/jira/browse/SPARK-5986`). But don't worry, we only need to save the K centroids (see spark/pull/4951 at `https://github.com/apache/spark/pull/4951/files`).

```
val stdScaler = new StandardScaler(withMean = true, withStd = true).
  fit(itemsRDD)
val finalRDD = itemsRDD.map(x => stdScaler.transform(Vectors.dense(x.toArray)))
finalRDD.cache

val seed = 42
val weights = Array(0.1, 0.7, 0.2)
val Array(trainRDD, valRDD, testRDD) = finalRDD.randomSplit(weights, seed)
val numClusters = 10
val numIterations = 20

val kmeansModel = KMeans.train(trainRDD, numClusters, numIterations)
val WSSSE = kmeansModel.computeCost(trainRDD)
val cc = kmeansModel.clusterCenters

val clusterIdUDF = udf { (x: Vector) =>
  val vector = stdScaler.transform(Vectors.dense(x.toArray))
  kmeansModel.predict(vector)
}
df2.select(col("asin"), clusterIdUDF(col("features")).as("clusterID")).show()
println("Within Set Sum of Squared Errors = " + WSSSE)
```

Once we have a K-Means model, and the many clusters of items, we are only left with one task—picking a user and making item recommendations.

```
$ sbt 'set fork := true' 'run-main chapter06.ContentBasedRSExample'
[info] asin   clusterID
[info] 0827229534 4
[info] 0313230269 1
[info] B00004W1W1 4
[info] 1559362022 4
[info] 1859677800 3
[info] B000051T8A 4
...
[info] 1577943082 4
```

```
[info] 0486220125 9
[info] B00000AU3R 0
[info] 0231118597 5
[info] 0375709363 9
[info] 0939165252 5
[info] Within Set Sum of Squared Errors = 2.604849376963733E14
[success] Total time: 28 s, completed 31 Jul, 2015 9:36:54 PM
```

Some points to note here are:

- Spark 1.3 doesn't have a mechanism to store the K-Means model to disk. This feature is available in Spark 1.4 and is pretty easy to use.
- To convert users to pretend items, you have many options like:
    1. Take an aggregate sum of user's items feature vectors.
    2. Take an average of user's item feature vectors.
    3. Take a weighted sum (using average ratings).
    4. Take a weighted sum (using distance of an item from its cluster centroid).

However, keep in mind that if you do scaling/normalization on feature vectors while learning the models, you also need to perform same scaling/normalization operation on pretend item vectors too.

Collaborative filtering based recommendation

Collaborative filtering is another approach to providing recommendation to users. A content-based recommender works well when we have a common set of features across all the items. But if we have a very diverse set of items, then we are stuck. Why? Because we would try to map all features to all the items. For example: a book doesn't have a director or singer or music-director, but a movie does. So it doesn't make sense to describe a book that way. In summary, the content-based recommender makes sense when we have items of a similar kind.

A collaborative filter takes a very different approach. Instead of looking into item details (or features), we see how two users are related just by the fact that how many items they both have in common. The more similar they are, the more likely they have similar taste. That is the basis of collaborative filtering. More specifically, this is called **user-user CF** or **user-based collaborative filtering**.

This argument goes with item-item similarity too, in which case we call it item-based collaborative filtering.

For example, let's say we have some historical data for $user_1$ and $user_2$. Basically we know which movies these two users like.

$user_1$ = `[1,1,0,0,1,1,0,1]`

$user_2$ = `[1,0,1,0,1,1,1,0]`

Where there are eight movies in total ["Star Wars," "Terminator," "Jurassic Park," "Inception," "Gravity," "Gone with the Wind," "Transformer," "Megamind"], we can find a similarity score between any pair of users by using a similarity metric, for example, using *Jaccard distance = 3 / 7 = 0.428*. From a set of all users, if we determine $user_1$ and $user_2$ are most similar, we can cross recommend items to them. To $user_1$, we would recommend ["Jurassic Park," "Transformer"] and to $user_2$, we would recommend ["Terminator," "Megamind"].

Movie names have been shortened in the following table to save space:

users	SW	TMR	JP	IC	GR	GWW	TFR	MGM
user1	1	1	0	0	1	1	0	1
user2	1	0	1	0	1	1	1	0
user3	1	0	0	0	1	0	1	0
user4	0	0	1	0	0	1	0	0
user5	1	0	0	0	0	1	1	0

Let's get back to our user and movies representation. It is essentially a 2D matrix of user and movie preferences. In this case, the preference was either 0 (not watched) and 1 (watched). Traditionally, these values are rating values (such as IMDb ratings, or Amazon product ratings), which range from 0 to 5 or 1 to 10 numeric values. However, gathering these product ratings is tough because very few users actually rate items. However, using a user's browsing history, clicks, views, and so on, we can gather a rich set of data. However, in this case we don't consider these as ratings, rather we consider them as implicit feedback, that is, either 0 for not clicked or 1 for clicked.

Now back to the 2D matrix representation of users and items (movies in this case). Here we represent the same matrix as a transpose of original matrix:

Items	user1	user2	user3	user4	user5
SW	1	1	1	0	1
TMR	1	0	0	0	0
JP	0	1	0	1	0
IC	0	0	0	0	0
GR	1	1	1	0	0
GWW	1	1	0	1	1
TFR	0	1	1	0	1
MGM	1	0	0	0	0

All of the data is the same except now we can view this problem as item-item collaborative filtering. So instead of determining similar users, we now determine similar items, based on the users' interaction with these items. Once we have identified similar items, we can recommend them to users who have not yet interacted with these items.

So far, we have only discussed how to find similarity between two users, decide the most similar ones and then make recommendations. However, essentially what we are trying to do is make a prediction of whether or not a user may be interested in a particular item. Now there are many 1s and many 0s in the ratings matrix above. Note that we are only going to recommend the items that have 0s in the ratings matrix. So, another way of thinking about this problem of recommendation is to fill in those zeros with some meaningful values. Once we replace these zeros, we can simply scan over the matrix and pick those items that have non-zero values in descending order. This is achievable using a matrix factorization algorithm called **ALS**, which we will discuss next.

What is ALS?

What is ALS and how does it work? It is a matrix factorization algorithm, so it essentially breaks down a matrix M into two matrices *X* and *Y* such that:

$A_{M\times N} = X_{M\times K} Y^{T}_{K\times N}$ where $A_{M\times N}$ is the user rating matrix, and $X_{M\times K}$ and $Y_{N\times K}$ are the two factors.

Since there could be millions of users and many thousands of items, *M* and *N* are typically very large values. We typically choose the value of *k* to be small. This is also called rank (rank k approximation). **ALS** stands for **alternating least squares**, so it works by keeping one of *X* or *Y* as a fixed value, with pre-filled values. Since *A* is now known and let's say we fixed *X* with some values, then we can solve *Y*: *Y = solve(A, X)*.

Now that we can solve Y, we can check the squared differences from original matrix A:

`squaredDifference(A,` $X \times Y^T$ $A - X \times Y^T$`)` = $| A - X \times Y^T |$

After calculating `squaredDifference` of A with $X \times Y^T$, we can check it against a threshold, and continue with fixing Y this time and solving X. In the ALS algorithm, this process keeps alternating between either X or Y. In every step the squared difference checked with a threshold value, which makes this an iterative algorithm. Fortunately, Apache Spark provides a distributed version ALS algorithm in MLlib package, which we will use.

ALS in Apache Spark

We will go through two code examples of ALS. First, with implicit ratings on a very small dataset we just discussed. Second, with our Amazon products dataset with explicit ratings. The ALS example on the preceding sample dataset is implemented in `src/main/scala/chapter06/ALSExample.scala`.

In the following snippet, we first load the sample CSV data, which has item ratings. Then we read into a RDD, and learn `ALSModel` from it:

```
val fileName = "movies-data.csv"

val df = sqlContext.load("com.databricks.spark.csv",
  Map("path" -> fileName,
    "header" -> "true", "delimiter" -> ",",
    "quote" -> "\""))

df.printSchema()
df.show()

val ratingsRDD = df.rdd.zipWithIndex.flatMap { entry =>
  val (row, userId) = entry
  val rowLen = row.length
  (1 until rowLen).flatMap { i =>
    val uid = userId.toInt
    val r = row.getString(i).toDouble
    if (r == 0) None else Some(Rating(uid, i, r))
  }
}.cache()

// train with implicit ratings
// because ratings are either 0 or 1
val rank = 2
val numIterations = 3
val model = ALS.trainImplicit(ratingsRDD, rank, numIterations)
```

Once we learn an `ALSModel`, we can then save it to the disk, as shown in the next snippet:

```
val MSE = ratesAndPreds.map {
  case ((user, product), (r1, r2)) =>
    val err = (r1 - r2)
    err * err
}.mean()

println("Mean Squared Error = " + MSE)

// Save and load model
val modelPath = "models/ALSExampleModel"
if (new File(modelPath).exists()) {
  FileUtils.deleteDirectory(new File(modelPath));
}
model.save(sc, modelPath)
val sameModel = MatrixFactorizationModel.load(sc, modelPath)
```

As shown in the preceding code, we can also save the trained model and load it back from the disk. Here is a sample run of the example:

```
$ sbt 'set fork := true' 'run-main chapter06.ALSExample'
...OUTPUT SKIPEED...
[info] Mean Squared Error = 0.05451631367213185
```

Next, we will see how to run the ALS algorithm on a decently sized dataset with explicit ratings.

ALS on Amazon ratings

We will extract the ratings data from MongoDB into a CSV file, load the ratings from CSV and then run ALS.

To extract the ratings, use the `scripts/amazon-ratings.js` script shown as follows:

```
$ mongo amazon_dataset --quiet < scripts/amazon-ratings.js > amazon-ratings.csv
```

The following commands give us the count of 6.8 million ratings, with over 1.5 million distinct customers, and over 398 thousand distinct products:

```
$ wc -l amazon-ratings.csv
6874337 amazon-ratings.csv
$ cut -f1 amazon-ratings.csv | sort | uniq | wc -l
```

```
1510225
$ cut -f2 amazon-ratings.csv | sort | uniq | wc -l
394828
```

This recommender is implemented in `src/main/scala/chapter06/AmazonRatingALS.scala`. First, we set 2 GB for the configuration parameter `spark.executor.memory`:

```
conf.set("spark.executor.memory", "2g")
```

Also we have to convert the string representation of customer IDs and item IDs, into numeric values first. For this, we have to make mapping tables as Scala Maps:

```
val customerToId =
rawData.map(_._1).distinct().zipWithUniqueId().collectAsMap
val asinToId =
rawData.map(_._2).distinct().zipWithUniqueId().collectAsMap
```

The preceding two lines are added in the code as follows:

```
def main(args: Array[String]): Unit = {
  val conf = new SparkConf(false)
    .setMaster("local[4]")
    .setAppName("AmazonRatingsALS")

  conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
  conf.set("spark.executor.memory", "2g")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)
  import sqlContext.implicits._

  val fileName = "datasets/amazon-ratings.csv"

  val rawData = sc.textFile(fileName, 100).
    map(_.split("\t")).
    flatMap { arr =>
      if (arr.length == 3) {
        val Array(c, a, r) = arr
        if (r.equals("rating")) None
        else Some((c, a, r.toDouble))
      } else None
    }

  val customerToId = rawData.map(_._1).distinct().zipWithUniqueId().collectAsMap
  val asinToId = rawData.map(_._2).distinct().zipWithUniqueId().collectAsMap
```

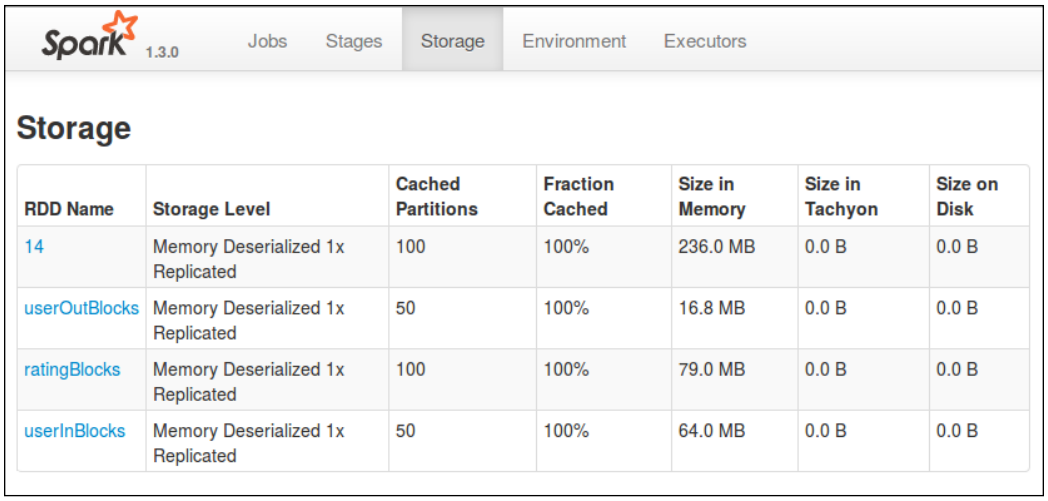
Next we convert all the rating entries into rating objects, train the model, and finally calculate the root mean squared error of all the predictions.

```
val ratingsRDD = rawData.map { t =>
  val (c, a, r) = t
  val cid = customerToId(c).toInt
  val aid = asinToId(a).toInt
  Rating(cid, aid, r)
}.cache()

val rank = 2
val numIterations = 3
val model = ALS.train(ratingsRDD, rank, numIterations, 0.01)

// Evaluate the model on rating data
val usersProducts = ratingsRDD.map {
  case Rating(user, product, rate) => (user, product)
}
val predictions = model.predict(usersProducts).map {
  case Rating(user, product, rate) => ((user, product), rate)
}
val ratesAndPreds = ratingsRDD.map {
  case Rating(user, product, rate) => ((user, product), rate)
}.join(predictions)
val MSE = ratesAndPreds.map {
  case ((user, product), (r1, r2)) =>
    val err = (r1 - r2)
    err * err
}.mean()
println("Mean Squared Error = " + MSE)
```

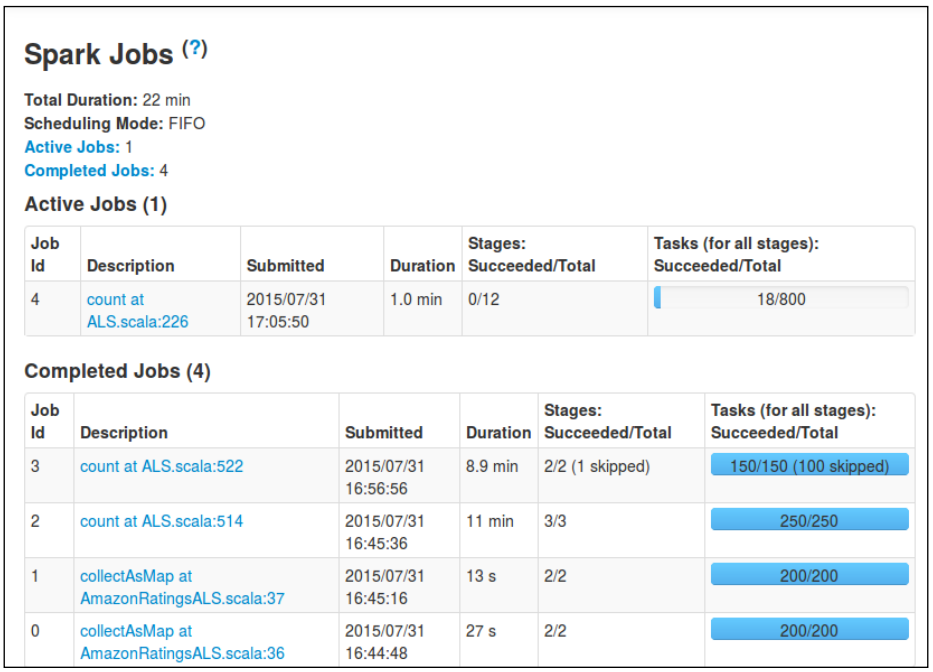
Finally, we save the model to disk.

Let's train the recommender on a full dataset using 6 GB of heap for JVM set using SBT configuration.

The storage view of Spark UI while the model is being trained is available at: `http://localhost:4040/storage/`.



Tasks in progress while the model is being trained are available at: `http://localhost:4040/jobs/`.



Invoke the training of the ALS model using the following command:

```
$ sbt ';set javaOptions += "-Xmx6114m"' ';set fork := true' 'run-main
chapter06.AmazonRatingsALS'
...OUTPUT SKIPEED...
[info] Mean Squared Error = 1.7120005414027863
...OUTPUT SKIPEED...
[info] Model saved to: models/AmazonRatingsALSModel
[success] Total time: 3445 s, completed 31 Jul, 2015 6:10:13 PM
```

It took about an hour to run on a single machine with the following configuration:

```
OS: Ubuntu 14.10
Linux Kernel: 3.17.6-031706-generic
Processor: Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz x 2
Memory: 8GB
```

That's not bad considering the millions of users and thousands of products we are dealing with.

Content-based versus collaborative filtering

We have covered a lot of ground for both content-based and collaborative filtering approaches. We even did some hands-on examples. Now let's do a fair comparison of both these approaches. First we discuss content-based approach, followed by the collaborative-filtering approach to recommender systems.

Content-based recommends items that are similar to those that the user liked in the past. In content-based recommendation, we really don't need user ratings. We can start with global recommendations, and as the user interacts more with the system, we can make recommendations even without the user rating any items. A content-based recommender will not generalize to different kinds of items as we discussed earlier. Content-based recommender systems therefore are implemented as case-based recommender systems, which take into account items specific to a particular category. Also note that in content-based recommendation, the items don't change their features frequently. So once we have prepared a model, we can re-use until more items are added, or removed.

To summarize:

- We do not need ratings, or data on other users.
- We are able to tailor recommendations according to a user's taste using the content-based approach.
- We can recommend an item regardless of an item being new or unpopular. The only criteria is that the feature vector for that item comes in a close neighborhood of a user's pretend item.
- We can explain the reason why an item was recommended to a user because of his/her history.

Collaborative filtering recommends items that similar users like collaboratively. Filtering assumes that we need item ratings before we can proceed. Since the algorithm depends on ratings (either implicit or explicit), there are no notion item features. Therefore, collaborative filtering algorithms are applicable to datasets containing diverse categories of items.

Because collaborative filtering algorithms require ratings, we face a road-block. Since the interaction of users and items is very dynamic, that is users keep visiting and rating items all the time, we need to learn or update the models in real time. This poses a significant challenge in engineering a good recommender system.

To summarize:

- The key assumptions in collaborative filtering algorithms are:
    - Past activity of a user will predict tastes of that same user in the future
    - The tastes of two similar users now, will likely be the same in future too
- The new or unpopular items will likely be recommended later, when their ratings improve.
- Explaining recommendations is also very challenging because now the recommendations are based on a collaborative effort of many users, and not just one user.

Summary

We implemented two new recommendation algorithms in this chapter—content-based recommender using K-Means clustering and collaborative filtering using the ALS algorithm. We also saw a comparison between the two approaches. In the next chapter, we will add more features to our application and complete the full application.

7
Enhancing the User Experience

We saw the implementation of both content-based recommendation and also recommendation based on collaborative filtering. In this chapter, we will discuss some more tricks that add more product search feature spice to the overall user experience. We will:

- Add a product search feature
- Add a recommendation listing
- Understand recommendation behavior

Adding product search

You may have noticed that with millions of users, and thousands of products, MongoDB is considerably slow for querying data on a single machine. Another point to note is that it stores most of the data which we may not want to query. And, to top it all, we want to be able to make free-text queries. For such a use case, an inverted-index based search engine is most appropriate. Please see this discussion for more details: `https://stackoverflow.com/questions/12723239/elasticsearch-v-s-mongodb-for-filtering-application`.

Elasticsearch is one of those search engines that we can conveniently set up, code and use in just a matter of hours. So, let's set it up.

Setting up Elasticsearch

First we download the ZIP archive:

```
$ wget -c https://download.elastic.co/elasticsearch/elasticsearch/
elasticsearch-1.4.4.tar.gz
$ tar zxf elasticsearch-1.4.4.tar.gz
$ cd elasticsearch-1.4.4
```

We chose version 1.4.4 here, however you could choose a later version too.

Next, we will install a very handy plugin called `head` that will help us visualize the search indices and make queries.

```
$ bin/plugin -i mobz/elasticsearch-head
```

We are all set. Let's start the Elasticsearch server:

```
$ bin/elasticsearch
```

Now visit `http://localhost:9200/_plugin/head/` to monitor your Elasticsearch collections and indices:

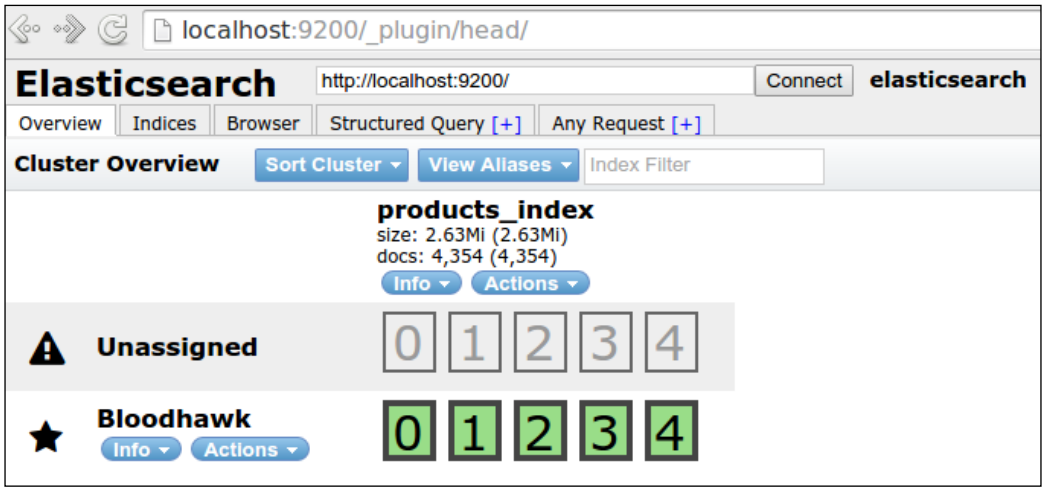
We have set up the Elasticsearch server, but it doesn't do us any good until we feed it some data. The product search feature will allow us to search through the title, group or categories of a product. This means we need to be able to add all these attributes into an Elasticsearch index. Once we have done that, we can query for products matching a criteria.

First, let's write some Scala code to push products data into the search server. Recall that we created a file called `datasets/content-based-dataset-new.csv`, for our content based recommender example. We will reuse that file to load product data. We can also read from MongoDB, but that's not the point. We are trying to quickly populate data into the search server. This code is implemented in the `chapter07.ImportElasticSearch` Scala object.

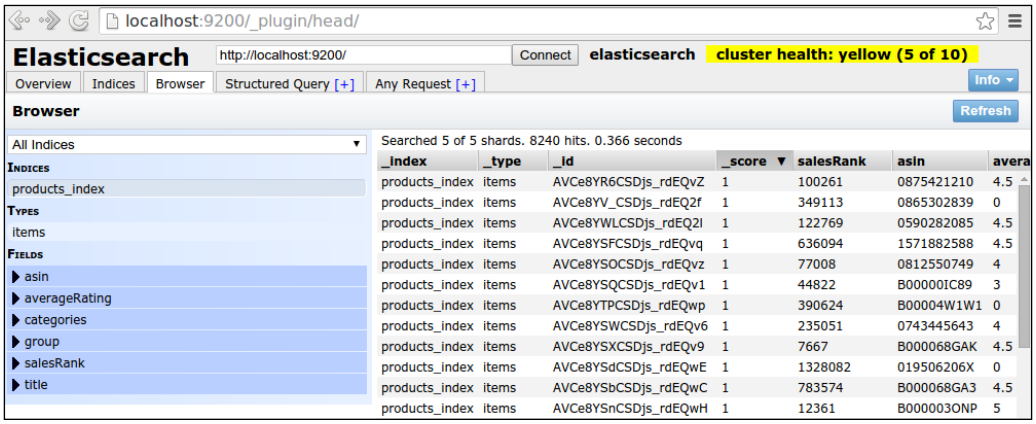
Now let's run the importer:

```
$ sbt "set fork := true" "run-main chapter07.ImportElasticSearch"
```

Once the import is complete, you can open Elasticsearch web UI, and explore the index. Open the UI:



Then go to the **Browser** tab and select **products_index**.



So far, so good. Now we must integrate the search server from your web application. For that we will use `elastic4s` Scala library to make queries. For that, we have to follow these steps:

1. Update `build.sbt` for library dependencies.
2. Create a helper class for forwarding queries to Elasticsearch server.
3. Add a route for search and update home page for products search.

Update `build.sbt` with the following line:

```
libraryDependencies += "com.sksamuel.elastic4s" %% "elastic4s" %
"1.4.14"
```

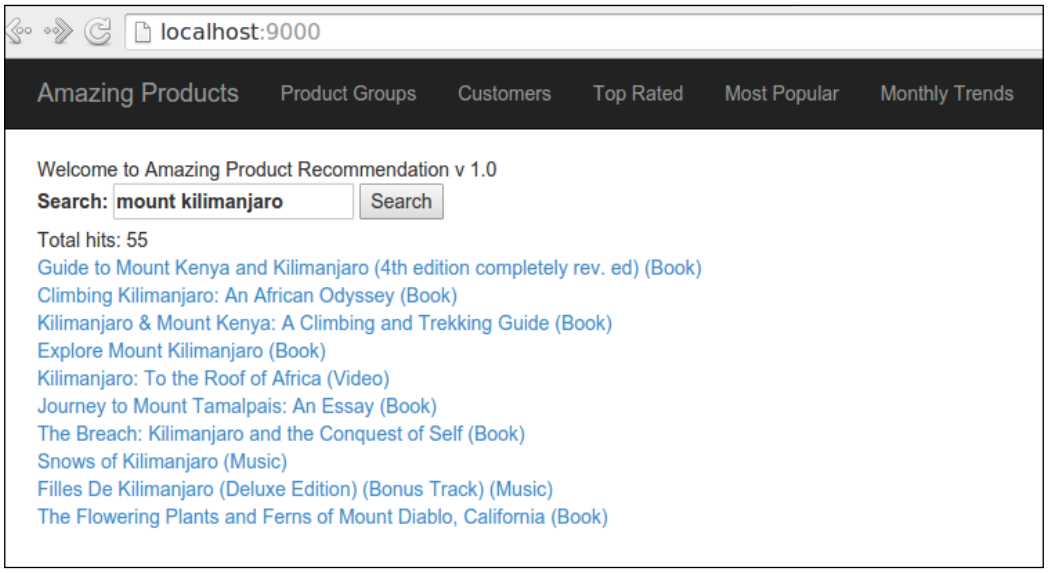
Next we create Scala object ESClient, with a utility method to search for user queries:

```
package util

import com.sksamuel.elastic4s.ElasticClient
import com.sksamuel.elastic4s.ElasticDsl._
import configuration.AppConfig
import scala.concurrent.Future
import org.elasticsearch.action.search.SearchResponse

object ESClient {
  val ESConfig = AppConfig.ES

  /**
   * Search for an item in the index
   * @param q
   * @return
   */
  def searchItem(q: String): Future[SearchResponse] = {
    val client = ElasticClient.remote(ESConfig.server, ESConfig.port)
    val productsIndex = ESConfig.productsIndex
    val rs = client.execute { search in productsIndex / "items" query q }
    rs
  }

}
```

Next we write update HTML, and some JavaScript in templates, and fetch search results using the `SearchController` class, which is routed to `/search/` path (see the `webapp-recommender/conf/routes` file):

```
package controllers

import scala.concurrent.ExecutionContext.Implicits.global

object SearchController extends Controller {
  def searchProducts() = Action.async { implicit request =>
    val query = request.getQueryString("q").getOrElse("")
    val searchResponseFuture = ESClient.searchItem(query)
    searchResponseFuture.map { searchResponse =>
      val hitlist = searchResponse.getHits().map { hit =>
        hit.getSource().toMap
      }.toList

      Ok(searchResponse.toString())
        .withHeaders("Content-Type" -> "application/json")
    }
  }
}
```

Finally, our search feature is working! Try searching for `mount Kilimanjaro`, which will probably show your following output:



This implementation is far from what you will use in a real website. However, it is an essential feature of most of the inventory-based websites, where you need to provide some way of narrowing down the search. Of course, this is minimal, but it will give you a head start on what technologies and glue code you will probably need. You may also want to make it much prettier. Next up, we will use our ALS model we created in the previous chapter. We will add recommendations to an individual customer's page.

Adding recommendation listings

Recall that in *Chapter 6*, *Collaborative Filtering versus Content-Based Recommendation Engines*, we trained a collaborative filtering based model using an ALS implementation of Apache Spark. Now that we already have it persisted we will re-use it to provide recommendations on a customer page.

Remember that when we trained that model, we had to create a rating object. This object had `userId`, `productId`, and a rating value. However, these values are specific to the algorithm, and are not related to our products, and customer IDs. So first, we need to recover those mappings. The following change to `AmazonRatingALS` does the job for us:

```
val customerToIdRDD = rawData.map(_._1).distinct().zipWithUniqueId()
val asinToIdRDD = rawData.map(_._2).distinct().zipWithUniqueId()
val custFile = "mappings/customerToId"
val asinFile = "mappings/asinToId"
removePathIfExists(custFile)
removePathIfExists(asinFile)
customerToIdRDD.map { case (x, y) => s"$x,$y" }.saveAsTextFile(custFile)
asinToIdRDD.map { case (x, y) => s"$x,$y" }.saveAsTextFile(asinFile)
```

With this change, we will have both customer and product mappings, which we will use to map between our data and the model representation of items and customers. After retraining the ALS model, now we will have the mappings written to disk which we can load in-memory.

```
$ sbt ';set javaOptions += "-Xmx6114m"' ';set fork := true' 'run-main
chapter06.AmazonRatingsALS'

[success] Total time: 3359 ...
```

Since our web application is written in the Play 2 framework, we have a problem with integrating Apache Spark into the same codebase. We need Apache Spark, to read the model and those mapping files. The integration problem is because both of them use different versions of Akka, which are incompatible. To overcome that limitation, we will create a simple recommender web service using the Spray framework. This can all be implemented in one single file. Again, update `build.sbt` for library dependencies:

```
libraryDependencies += "io.spray" %% "spray-can" % "1.3.3"

libraryDependencies += "io.spray" %% "spray-routing" % "1.3.3"

libraryDependencies += "io.spray" %% "spray-json" % "1.3.2"
```

Now, create a simple web service using the Spray framework (`http://spray.io/`) for which our web application will use forwarding recommendation queries. This is implemented in `src/main/scala/chapter07/RecommendationServer.scala`. The following snippets show the route `recommendations/foruser/<userID>`, that then returns at most 10 recommendations for a particular user. All the ID mappings to and from our representation are handled in this snippet:

```
path("recommendations" / "foruser" / Segment) { user =>
  get {
    respondWithMediaType(`application/json`) {
      complete {
        val m = SharedData.alsModel
        val userId = SharedData.customerToIdMap.get(user)
        val rr = userId.map { id =>
          val recommendations: Array[Rating] = try {
            m.recommendProducts(id, 10)
          } catch {
            case t: Throwable => Array()
          }
          val recs: Array[(String, Double)] = recommendations.flatMap { r =>
            val asinOpt = SharedData.idToAsinMap.get(r.product)
            asinOpt.map(asin => asin -> r.rating)
          }

          RecResults(s"Found results for user: ${user}", recs.toMap)
        }.getOrElse(RecResults(s"No such user found: ${user}"))

        rr.toJson.prettyPrint

      }
    }
  }
}
```

The full implementation is present in `src/main/scala/chapter07/RecommendationServer.scala`. We need to make sure that the `RecommendationServer` is running for our web application to be able to fetch recommendations for a customer.

```
$ sbt "set fork := true" "run-main chapter07.RecommendationServer"
```

This server will run on localhost (or `127.0.0.1`), listening on HTTP port `8082`. We will use this same host and port to configure our web application. All such configurations in our project are controlled using `conf/application.conf`, parts of which are shown in the following code:

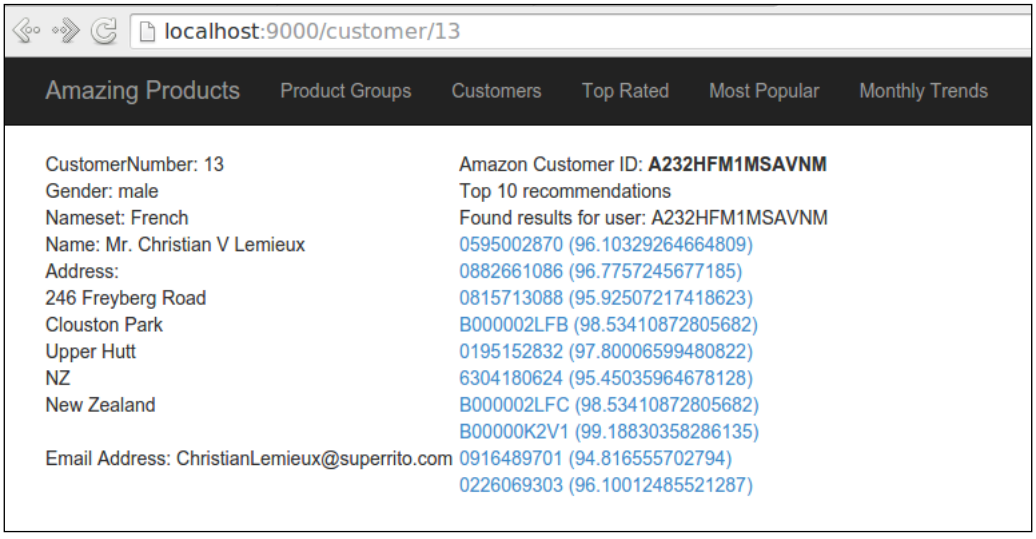
```
elasticsearch {
  server = "127.0.0.1"
  port = 9300
  products_index = "products_index"
}

mongodb {
  host = "localhost"
  port = 27017
```

```
  database = "amazon_dataset"
  products_collection = "products"
}

recommenderService {
  host = "localhost"
  port = 8082
}
```

Next, we will update our customer view to fetch recommendations asynchronously using AJAX. We added the following method to `util.Recommender`, and created a route `/recommendation/for/customer/:id/` which merely acts as a proxy to `RecommendationServer` running on port `8082`.

```
def findForCustomer(customerId: String): Future[JsValue] = {
  val host = AppConfig.RS.host
  val port = AppConfig.RS.port
  val url = s"http://$host:$port/recommendations/foruser/$customerId"
  for (response <- WS.url(url).get()) yield {
    response.json
  }
}
```

The rest of the changes are mostly cosmetic. Let's take a look at how the recommendations are displayed:



The left side shows the customer information we already had earlier. The right side shows that for this customer, we have 10 recommendations, which shows the product IDs (with recommender score in parentheses, the higher the better).

Understanding recommendation behavior

In this section, we will discuss how a recommender system impacts the user's behavior and also how the user's behavior should impact the recommender system.

So far we discussed global recommendations, content-based recommendations, and recommendations based on user-user / item-item collaborative filtering. The impact of these recommendations is that they always try to find something that is based on the likes of the user. However, these recommenders fail to find something a user's history doesn't indicate but the user will like. A recommender should be all about finding new stuff that a user will like. Yes, our recommender will find something new but that will be similar to what a user has already liked.

Why is that so?

To understand that, let's look back and recall that from a whole set of items that we have, we typically do not have any of the user's preferences initially. So we put up global recommendations. Since these global recommendations are the only ones a new user will see, she/he will get bored. Why?

This is because a user wants to be surprised and delighted with new and interesting items from our recommender system, which our recommender will fail at terribly.

However as time passes, and hopefully while we still have active users, the recommender will adapt itself and will have affinity toward a few sets of popular items. This means a vast majority of items will always have a very low to negligible hit-count. And that is what we do not want. So we need to be systematic in our approach towards building a recommender system. To make this a systematic process, we will discuss some themes now, which are:

- Logging
- Ranking
- Diversification
- Justification
- Evaluation

Logging

As we have seen, our recommendations are totally based on knowledge of the past. This knowledge is what lets us best predict the future choices of the users. So the first thing that we need to ensure is that we log (record) the choices that users make. Since, in collaborative filtering and content-based recommendations, we need to model a user's profile before we can make any recommendations; we need to keep a log of a user's activity. A detailed user profile will only increase the confidence with which a recommendation can be made.

Ranking

In many cases, we have more than one recommendation to provide to a user. In such a situation, we have to ensure that the most appropriate recommendation is provided at the top. That also gives us a chance to put some "interesting" items somewhere near the middle and the bottom positions.

Diversification

Suppose you went on a site to search for restaurants, and searched for "Biryani." Now there could be thousands of places where you could find such an item, but the system only shows "Chicken Biryani," it would be boring. How about we add items like "Veg Biryani," "Raita," "Mughlai Food," "Hyderabadi Biryani", and so on to the mix? That would make the recommendations more diversified. Sometimes a customer is not decided on a single item, so the recommendations can, do and should help the customer decide what she/he really wants.

Justification

Finally, when we make a recommendation, it could be possible that the recommendation is wrong. It could be interesting but not clear as to why the system is generating such a recommendation. We always provide recommendations to a user based on historical data, and the knowledge of a user's choices and preferences. So based on that data explaining why we recommended something is crucial. The user deserves, and in some case expects, an explanation.

Evaluation

Once we have built a recommendation we also need to have a way to evaluate how it will actually perform on a production system. Evaluation is by far the toughest part. Evaluation has its own challenges, which we will break down into three parts:

- Engagement:
    - How do we measure that the users are returning or churning?
    - How do we measure that the recommendations are always up to date?
- Data:
    - How is the system responding to new users and new items?
    - How does the performance change when we receive new ratings on same and new items?
- Experience:
    - So over time, do the recommendations really improve?
    - How do we measure improvement?

Out of all the steps involved in creating a recommender system, evaluation is the toughest part. It involves continuous effort. There are so many possible ways a minor change could affect the whole system. Just creating a recommender algorithm and deploying is not the end of story. Evaluation is a continuous process, which must be integrated with the whole development process of the application. That also means that just discussing evaluation strategies of different algorithms will require an entire book to be written. It requires one to understand the data science. However, we will discuss a mature project called PredictionIO in the next chapter, to help us with the tooling.

Summary

In this chapter, we added free-text search, and also added recommendations based on the collaborative filtering model we learned about. We also discussed how complex it is to understand the behavior of recommender systems. In the next chapter, we will discuss a case study of a real-world, Scala based open source machine learning server called PredictionIO. It covers most of the steps we discussed for setting up data ingestion and a processing pipeline. We will also discuss a hybrid recommender called unified recommender.

8
Learning from User Feedback

PredictionIO is an open source machine learning server written in Scala, which under the hood uses Apache Spark and/or Apache Mahout for its machine learning algorithms. It uses HBase, Elasticsearch, and other different databases for data storage. PredictionIO provides a developer API where we can even create our own custom algorithms, evaluate them, and deploy in a fraction of time.

We will discuss the following topics:

- Case study: PredictionIO
- Hybrid recommender: PredictionIO unified recommender

Introducing PredictionIO

PredictionIO is targeted for developers and data scientists, specifically because it can help with the following tasks:

- Connecting different pieces in the complete data processing pipeline into one coherent system
- Prototyping predictive models
- Training, persisting, and deploying predictive models in a distributed environment
- Configuration changes and updates to the models without downtime

How does PredictionIO achieve all these things you may ask? Well, we get the answer from the way it is architected, which is called **DASE**. It consists of the following components:

- Data includes data source and DataPreparator
- Algorithm(s)

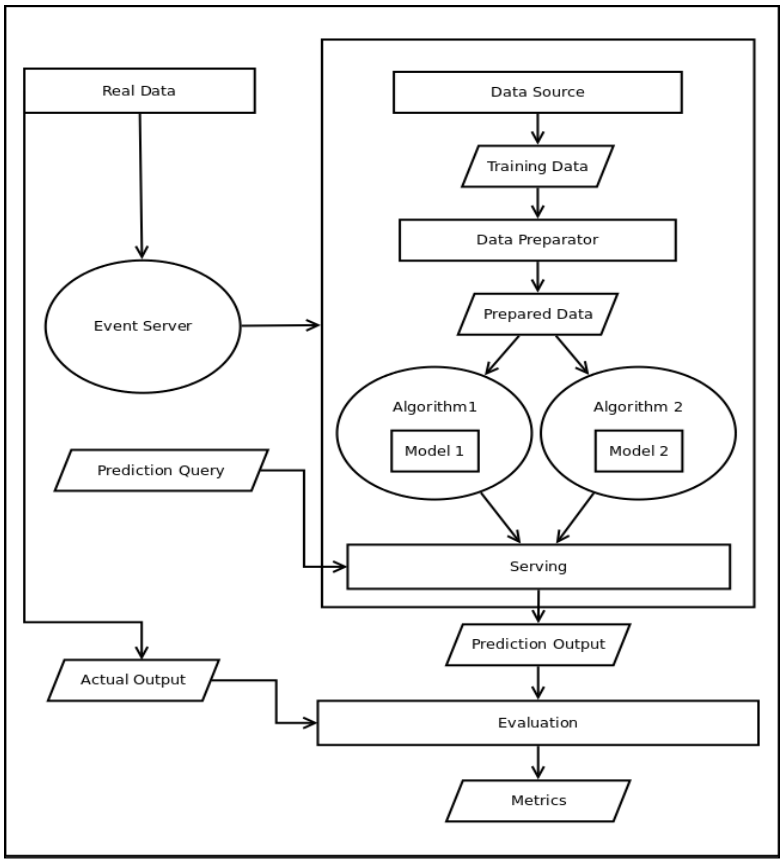
- Serving
- Evaluator

In the next figure, we illustrate how data, algorithms, serving and evaluator components fit together in the whole data processing pipeline. Event server passes events to the data source. These events are actually the real data, which we want to learn from. DataPreparator transforms it into a representation that can then be passed on to different learning algorithms.

Once the models are trained, they are persisted into storage, and can be used for either experimental evaluation, or for actual production usage. Serving layer queries different algorithms and produces a prediction for a prediction query.

For evaluation though, we need the evaluation layer, which uses the serving layer to generate predictions for test queries.

The architectural diagram of DASE components in PredictionIO:

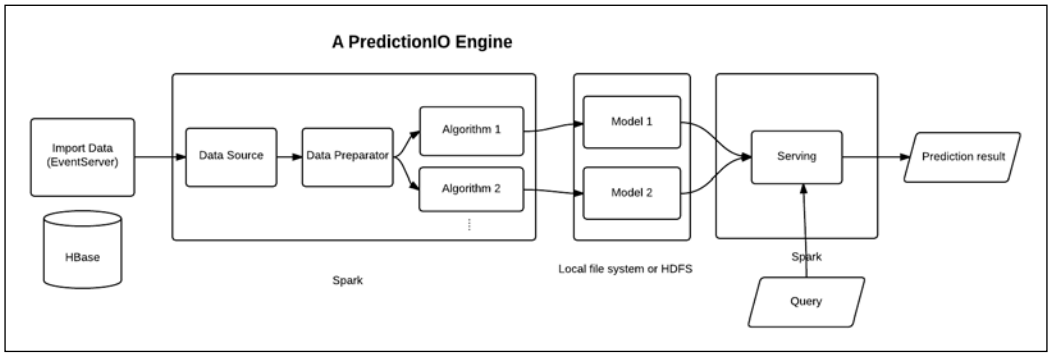


You may want to look at its documentation (`https://prediction.io/whatispredictionio`) for more details. However, we will present you with a summary here:

- Based on the event source: The events are a source of inputs to the learning algorithms
- DASE: This gives us a complete pipeline
- Swap and evaluate: The algorithms evaluation step is integrated with the pipeline
- Template Gallery: There are many ready-to-use algorithms in the Template Gallery

Also, there is a broad range of pre-build engines that you can use from PredictionIO's Template Gallery. This will definitely get you started on your next scalable machine learning project.

Here is a pipeline view of the PredictionIO engine:



Source: `https://docs.prediction.io/system/`

Let's now move on to some real examples. We first begin by installing PredictionIO and setting it up.

Installing PredictionIO

Installing PredictionIO is very straightforward. We can use a shell script that does all the installation for us, and using cURL we don't even need to download that script. Here is how you do it:

```
$ bash -c "$(curl -s https://install.prediction.io/install.sh)"
Welcome to PredictionIO 0.9.4!
Installation path (/home/tuxdna/PredictionIO):
```

```
Vendor path (/home/tuxdna/PredictionIO/vendors):
Please choose between the following sources (1, 2 or 3):
1) PostgreSQL
2) MySQL
3) Elasticsearch + HBase
#? 3
Receive updates? [Y/n] n
--------------------------------------------------------------------------
-------
OK, looks good!
You are going to install PredictionIO to: /home/tuxdna/PredictionIO
Vendor applications will go in: /home/tuxdna/PredictionIO/vendors

Spark: /home/tuxdna/PredictionIO/vendors/spark-1.4.1
Elasticsearch: /home/tuxdna/PredictionIO/vendors/elasticsearch-1.4.4
HBase: /home/tuxdna/PredictionIO/vendors/hbase-1.0.0
ZooKeeper: /home/tuxdna/PredictionIO/vendors/zookeeper
Select your linux distribution:
1) Debian/Ubuntu
2) Other
#? 1
Would you like to install Java? [Y/n] n
Locating JAVA_HOME...
Found: /usr/lib/jvm/java-7-oracle
--------------------------------------------------------------------------
-------
Installation of PredictionIO 0.9.4 complete!
... OUTPUT SKIPPED ...
```

Now that we have installed all the required dependencies, we also need to make sure that PredictionIO tools are on the `PATH` variable so that we can use them. To do that we add them to the command path:

```
$ echo 'export PATH=$PATH:'"$HOME/PredictionIO/bin" >> ~/.bashrc
```

Now, open a new terminal and then start all the services required:

```
$ pio-start-all
Starting Elasticsearch...
```

```
Starting HBase...
starting master, logging to /home/tuxdna/PredictionIO/vendors/
hbase-1.0.0/bin/../logs/hbase-tuxdna-master-matrix02.out
Waiting 10 seconds for HBase to fully initialize...
Starting PredictionIO Event Server...
```

Note that you should not have any incompatible versions of Hadoop tools on the path. If there is an incompatible version of Hadoop tools, then HBase may not start up properly.

Unified recommender

Now let's create a unified recommender using the template provided by PredictionIO project.

```
$ mkdir ~/tmp
$ cd ~/tmp
$ pio template get PredictionIO/template-scala-parallel-universal-
recommendation urec-app
Please enter the template's Scala package name (e.g. com.mycompany): com.
example.urec
Author's name:    My Name
Author's e-mail:   MyEmail.Example@gmail.com
Author's organization: com.example.urec
... OUTPUT SKIPPED ...
Engine template PredictionIO/template-scala-parallel-universal-
recommendation is now ready at urec-app
```

Create a Git repository for the app we just created. This will help us track our local changes:

```
$ cd urec-app
$ git init .
$ git add .
```

Update the `appName` in `engine.json` to `urec-app` as shown in the following Git diff:

```
$ git diff
diff --git a/engine.json b/engine.json
index 495de5c..f371d40 100644
--- a/engine.json
+++ b/engine.json
```

```
@@ -5,7 +5,7 @@
 "datasource": {
  "params" : {
   "name": "sample-handmade-data.txt",
-   "appName": "handmade",
+   "appName": "urec-app",
   "eventNames": ["purchase", "view"]
  }
 },
@@ -22,7 +22,7 @@
   "comment": "setup for example 'handmade' data, make sure to change for
your needs",
   "name": "ur",
   "params": {
-    "appName": "handmade",
+    "appName": "urec-app",
     "indexName": "urindex",
     "typeName": "items",
     "eventNames": ["purchase", "view"]
```

Now start all Prediction IO services:

```
$ pio-start-all
Starting Elasticsearch...
Starting HBase...
starting master, logging to /home/tuxdna/PredictionIO/vendors/
hbase-1.0.0/bin/../logs/hbase-tuxdna-master-matrix02.out
Waiting 10 seconds for HBase to fully initialize...
Starting PredictionIO Event Server...
```

From the application template, we ask PredictionIO to initialize the app:

```
$ pio app new urec-app
[INFO] [HBLEvents] The table pio_event:events_1 doesn't exist yet.
Creating now...
[INFO] [App$] Initialized Event Store for this app ID: 1.
[INFO] [App$] Created new app:
[INFO] [App$]   Name: urec-app
[INFO] [App$]    ID: 1
```

```
[INFO] [App$] Access Key:
fwqjaPTyhzPScEBP8kl05EEQCExrRZ2zds0EeDpA23MwnAyPvTlw9ANQSKZ2k1Wl
```

You must note the access key generated in the preceding output, as it is used for all subsequent operations. In the template the examples data is populated using PredictionIO Python API, we also install them:

```
$ sudo pip install PredictionIO
```

Now import sample data using the access key we received for our app:

```
~/tmp/urec-app$ python examples/import_handmade_eventserver.py --access_
key fwqjaPTyhzPScEBP8kl05EEQCExrRZ2zds0EeDpA23MwnAyPvTlw9ANQSKZ2k1Wl

Namespace(access_key='fwqjaPTyhzPScEBP8kl05EEQCExrRZ2zds0EeDpA23MwnAyPv
Tlw9ANQSKZ2k1Wl', file='./data/sample-handmade-data.txt', url='http://
localhost:7070')

Importing data...

Event: purchase entity_id: u1 target_entity_id: iphone

... OUTPUT SKIPPED ...

Event: view entity_id: u1 target_entity_id: phones

... OUTPUT SKIPPED ...

Event: $set entity_id: iphone properties/catagory: phones

... OUTPUT SKIPPED ...

22 events are imported.
```

Now build the engine, train and then finally deploy it shown as follows:

```
$ pio build

[INFO] [Console$] Using existing engine manifest JSON at /home/tuxdna/
tmp/urec-app/manifest.json

... OUTPUT SKIPPED ...

[INFO] [RegisterEngine$] Registering engine
BqyQGKT8gmWjn9cNY0r9MJMVsF5cNemM f1a7b261ef8ab87d2bd107c4214fc648f2db4617

[INFO] [Console$] Your engine is ready for training.
```

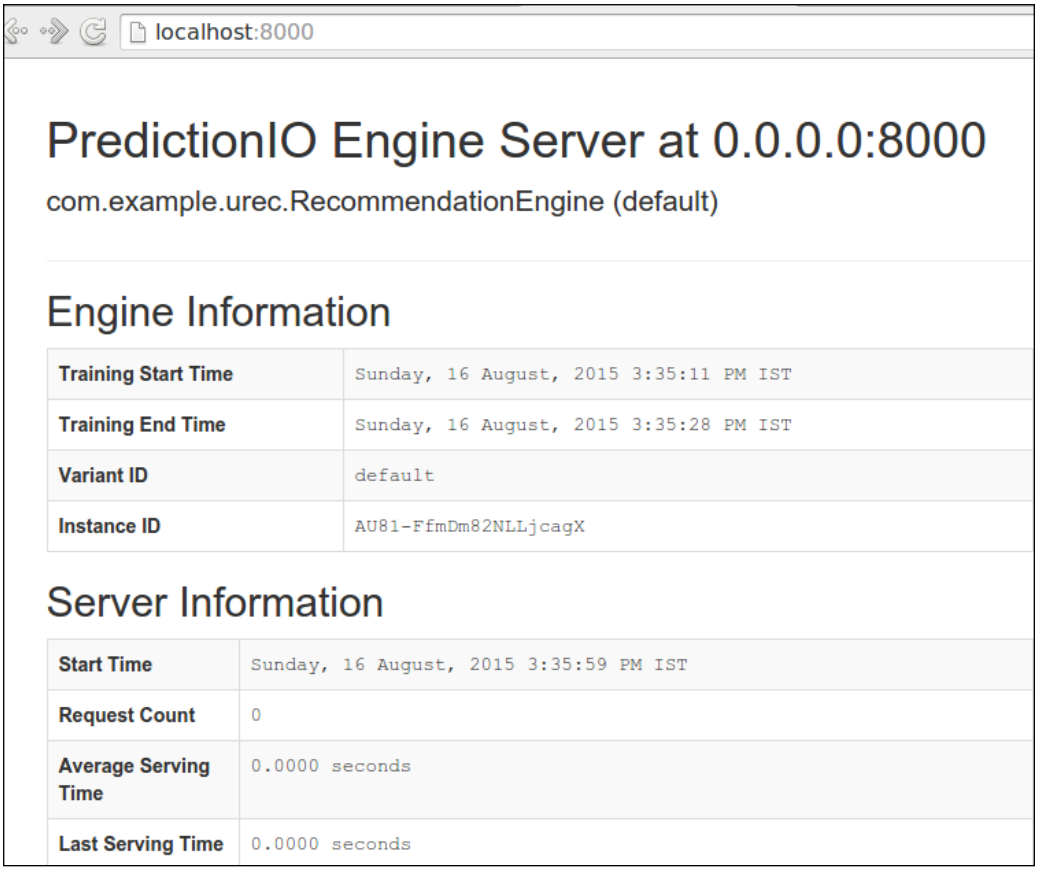
The training commands are as follows:

```
$ pio train -- --driver-memory 8G --executor-memory 8G

[INFO] [Console$] Using existing engine manifest JSON at /home/tuxdna/
tmp/urec-app/manifest.json

... OUTPUT SKIPPED ...

[INFO] [CoreWorkflow$] Updating engine instance

[INFO] [CoreWorkflow$] Training completed successfully.
```

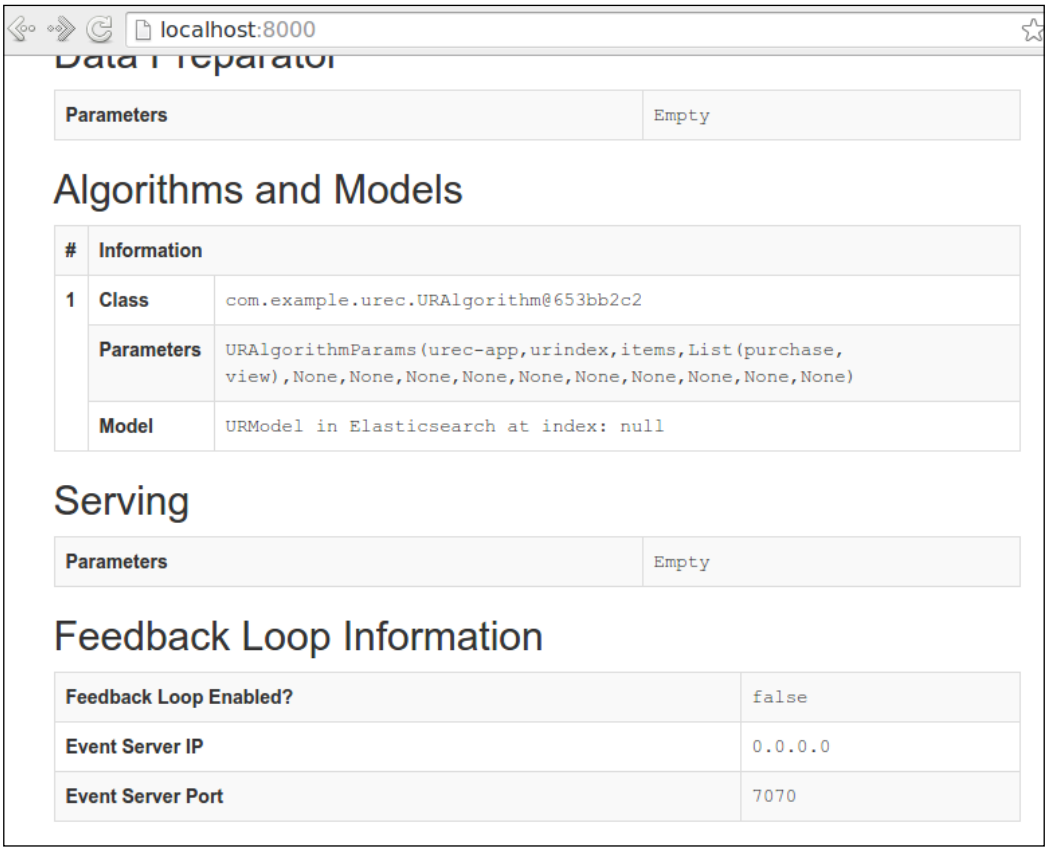
Finally, deploy it to the PredictionIO cluster:

```
$ pio deploy
... OUTPUT SKIPPED ...
[INFO] [MasterActor] Engine is deployed and running. Engine API is live
at http://0.0.0.0:8000.
```

Once it is deployed, you can see it running on the web browser:



In the following figure, we can see that our app using URAlgorithm is deployed, along with its parameters:



Please note here that we are using PredictionIO version 0.9.4, and in an upcoming release some changes may break the commands we have used. Now let's query the engine for recommendations:

```
$ sh ./examples/query-handmade.sh


Recommendations for user: u1
{"itemScores":[{"item":"galaxy","score":0.7635629773139954},{"it
em":"nexus","score":0.06365098059177399},{"item":"surface","sco
re":0.0482502318918705}]}
 ... OUTPUT SKIPPED ...
```

The preceding command makes a query to PredictionIO app where our recommendation algorithm is running. The query returned some recommendations for user `u1`. With this approach all the development and deployment of a recommendation becomes much simpler, and easy to maintain and evaluate. Using a server like PredictionIO we can now focus on building our core piece, that is, the recommendation engine.

Summary

We saw a PredictionIO setup and a unified recommender example. That brings us to the end of *Building a Recommendation Engine with Scala*. We hope you have gained enough ground, so that you can now start building your own recommendation engines using the Scala programming language. Creating real-world recommendation engines is a complex task, and rightly so, we can't possibly cover each and every detail in a single book. We gave you a bird's eye view of what's available and what's possible. It has been a great journey and we hope that this book helps you explore advanced tools and techniques using Scala programming language. Good luck!

Index

D

E

F

G

H

Q

R

S

T

U

V

W

Z