Building a Recommendation Engine with Scala

With the increase of data in online e-commerce systems, the challenges in assisting users to narrow down their search have grown dramatically. The various tools available in the Scala ecosystem enable developers to build a processing pipeline to meet those challenges and create a recommendation system to accelerate business growth and leverage brand advocacy for your clients.

This book provides you with the Scala knowledge you need to build a recommendation engine.

You'll be introduced to Scala and other related tools to set the stage for the project and familiarise yourself with the different stages in the data processing pipeline, including at which stages you can leverage the power of Scala and related tools. You'll also discover different machine learning algorithms using MLlib.

As the book progresses, you will gain detailed knowledge of what constitutes a collaborative filtering based recommendation and explore different methods to improve users' recommendation.

Who this book is written for

This book is written for those who want to learn the different tools in the Scala ecosystem to build a recommendation engine. No prior knowledge of Scala or recommendation engines is assumed.

What you will learn from this book

- Discover the tools in the Scala ecosystem
- Understand the challenges faced in e-commerce systems and learn how you can solve those challenges with a recommendation engine
- Familiarise yourself with machine learning algorithms provided by the Apache Spark framework
- Build different versions of recommendation engines from practical code examples
- Enhance the user experience by learning from user feedback
- Dive into the various techniques of recommender systems, such as collaborative, content-based, and cross-recommendations

Who this book is written for

This book is written for those who want to learn the different tools in the Scala ecosystem to build a recommendation engine. No prior knowledge of Scala or recommendation engines is assumed.

What you will learn from this book

- Discover the tools in the Scala ecosystem
- Understand the challenges faced in e-commerce systems and learn how you can solve those challenges with a recommendation engine
- Familiarise yourself with machine learning algorithms provided by the Apache Spark framework
- Build different versions of recommendation engines from practical code examples
- Enhance the user experience by learning from user feedback
- Dive into the various techniques of recommender systems, such as collaborative, content-based, and cross-recommendations

Saleem Ansari

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction to Scala and Machine Learning'
- A synopsis of the book’s content
- More information on Building a Recommendation Engine with Scala
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.
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.
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.
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:

- **Scal IDE**: http://scala-ide.org/docs/user/gettingstarted.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`

```scala
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(s"x\03205.2f + \$y\03205.2f = \$s\03205.2f")
    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:

```scala
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:

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

scala> 3 + 4
res1: Int = 7

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

```scala
$ 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:

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

---

[8]
Chapter 1

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:

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

```scala
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:

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

```bash
dependency管理器
```
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:

```scala
package chapter01
import scala.collection.JavaConversions._
import scala.io.Source
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 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)
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 an 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.
Where to buy this book

You can buy Building a Recommendation Engine with Scala from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.