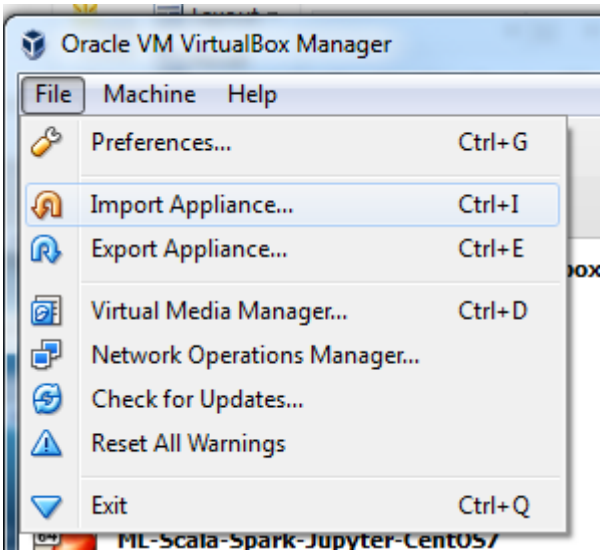
Spark Introduction – Exercises

Contents

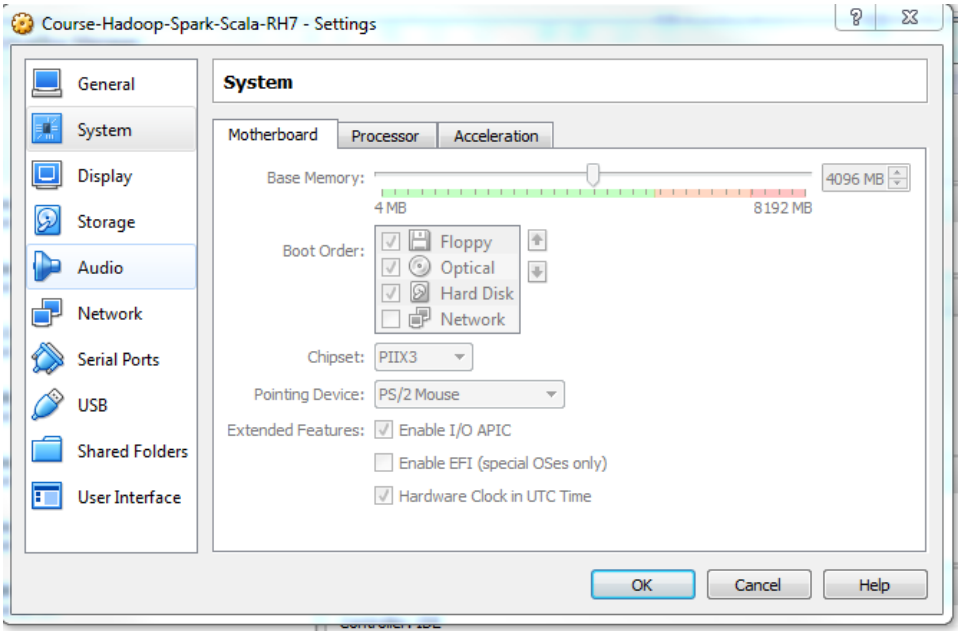
Exercise 0: Prepare Environment

The virtual machine should be already installed in your PC. If not install it following the steps:

- Import the *.ova file as an appliance



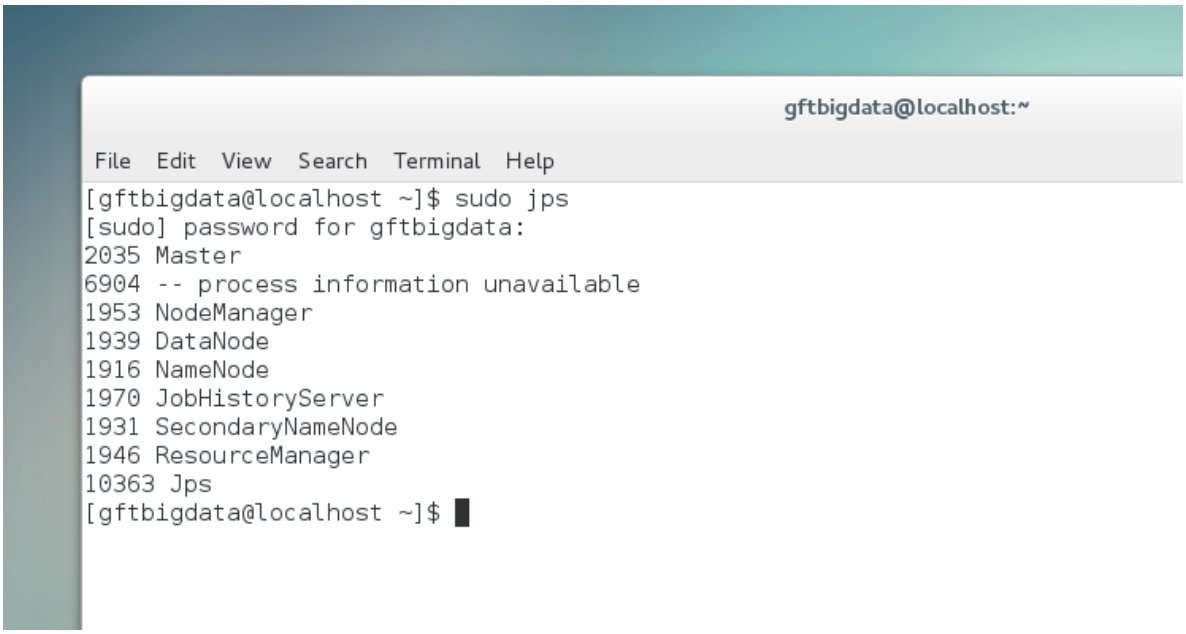
- When importing make sure you have at least **4096 MB** of memory and at least **2 cores**:



- Once the machine is up, log on:
  USER: cloudera
  PASSW: cloudera

  The user **cloudera** has sudo privileges.

- Open a terminal and check that all the required services are running:

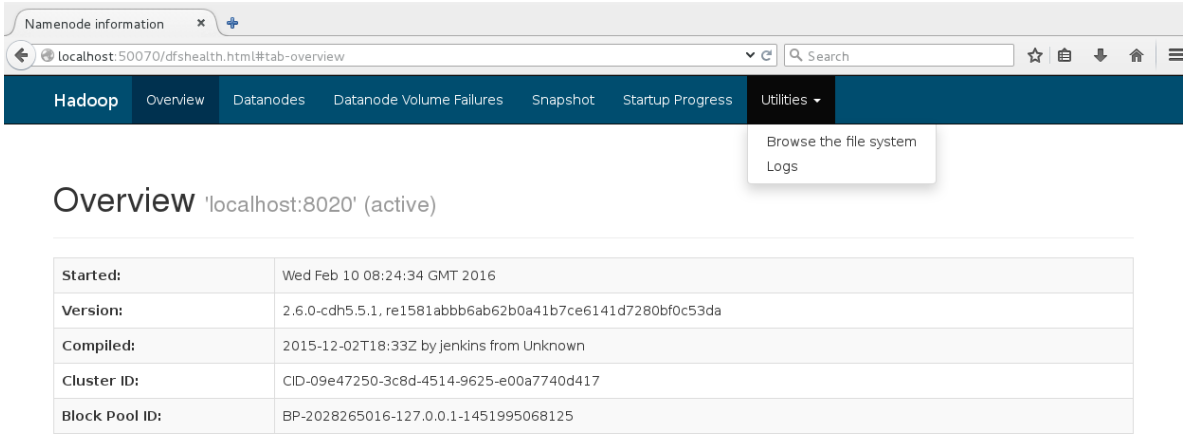


Exercise 1: WordCount MapReduce

Upload the “El Quixote” book dataset to HDFS.

```
$ cd
$ hadoop fs –mkdir input-spark
$ hadoop fs –put ./course-spark/datasets/quixote.txt /user/cloudera/input-spark
```

In the virtual machine open a web browser and go to the address: **http://localhost:50070** Browse the file system. Look for the file and blocks



Run the *wordcount* program and check the output:

```
$ cd
$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.6.0-cdh5.5.1.jar
wordcount /user/cloudera/input-spark/quixote.txt /user/cloudera/quixote-wordcount
```

Check in the console output or in the browser the job progress: **http://localhost:8088**

Check the output:

```
$ hadoop fs –ls /user/cloudera/quixote-wordcount
$ hadoop fs –cat /user/cloudera/quixote-wordcount/part-r-00000
```

There are also other pre-compiled programs that can serve as examples: wordmean, wordmedian, wordstandarddetivation, pi, teragen, terasort, etc.

Exercise 2: Using Spark Shell

Start the Spark shell

```
$ spark-shell
```

Play with the shell. It is a complete Scala interpreter:

```
scala> val data = (1 to 100).toList
scala> val rdd = sc.parallelize(data)
scala> rdd.count
scala> rdd.cache
scala> rdd.take(10)
scala> rdd.take(10).foreach(println(_))
```

```
scala> rdd.foreach(println(_))
scala> rdd.map(_ * 2).foreach(println(_))
scala> rdd.map(i => (i % 10, 1)).reduceByKey(_ + _).foreach(println(_))
scala> …
```

Do you prefer python? Try the python-based Spark shell:

```
$ pyspark
```

Exercise 3: WordCount Spark

```
$ cd
$ spark-shell
```

Execute wordcount and check the output:

```
scala> val data = sc.textFile("/user/cloudera/input-spark/quixote.txt”)
scala> data.cache
scala> data.take(10).foreach(println(_))
scala> val counts = data.flatMap(line => line.split(“ “)).map(word =>
(word,1)).reduceByKey(_+_)
scala> counts.saveAsTextFile(“/user/cloudera/quixote-wordcount-spark”)
```

Exercise 4: Top-10 WordCount

Get 10 most used words in “The Quixote” book. Note this would require 2 MapReduce jobs!

Start the Spark shell, this time with 2 local workers.

```
$ spark-shell --master local[2]
```

Execute a wordcount, sort the output and get the top 10 (except “Quixote” and “Sancho”. Also, do some kind of filtering (*e.g.* remove words with length <=4), try yourself before check the solution

Solution

```
scala> val quixote = sc.textFile("/user/cloudera/input-spark/quixote.txt")
scala> val result = quixote.flatMap(line => line.split(" ")).
| filter(word => word.length>4 && !List("Quixote","Sancho").contains(word)).map(word => (word,1)).
| reduceByKey(_ + _).
| map(tuple => tuple.swap).
| sortByKey(false)
scala> result.toDebugString
res7: String =
(2) ShuffledRDD[45] at sortByKey at <console>:14 []
+-(2) MappedRDD[42] at map at <console>:14 []
| ShuffledRDD[41] at reduceByKey at <console>:14 []
+-(2) MappedRDD[40] at map at <console>:14 []
| FilteredRDD[39] at filter at <console>:14 []
```

```
| FlatMappedRDD[38] at flatMap at <console>:14 []
| /user/spark/quixote.txt MappedRDD[1] at textFile at <console>:12 []
| /user/spark/quixote.txt HadoopRDD[0] at textFile at <console>:12 []
scala> result.take(10)
(1725,which)
(1218,would)
(1014,there)
(999,their)
(719,without)
(652,those)
(649,himself)
(643,Sancho,)
(641,should)
(634,could)
```

Exercise 5: Aggregation by key

In this short exercise we will get some practice with the expressiveness of spark key-based reduce operations.

Problem: suppose that we have the following list of key-value tuples:

{ (“a”,3), (“a” 1), (“b”, 7), (“a”, 5)}

Find:

1. The sum of all the values aggregated by key. *i.e.* { (“a”,9), (“b”, 7) }
2. The aggregation by key of all the values grouped in sets. *i.e.* { (“a”, Set(1,3,5)), (“b”, Set(7)) }

Notice that in 1. a simple reduce operation will be enough. However, in 2, the type of the result of the reduce operation and the type of the elements to be reduced is not the same. Look in the Spark documentation for *reduceByKey* and *aggregateByKey*.

Start the Spark shell

```
$ spark-shell
```

Create the RDD from the in-memory object

```
$ val pairs = sc.parallelize(Array(("a", 3), ("a", 1), ("b", 7), ("a", 5)))
```

Obtain the sum of all the values aggregated by key;

```
$val resReduce = pairs./**complete 1 **/
$resReduce.collect
```

Import the **mutable** Set Scala class (in this case is better to use **mutable** structures because for long lists, the overhead of creating new Sets every time can be substantial).

```
$ import scala.collection.mutable.HashSet
```

Obtain the aggregation by key of all the values grouped in sets

```
$val resAgg = pairs./**complete 2 **/
$resAgg.collect
```

Solution

Complete 1:

```
$val resReduce = pairs.reduceByKey(_+_)
```

Complete 2:

```
$val resAgg = pairs.aggregateByKey(new HashSet[Int])(_+_, _++_)
```

Exercise 6: Combine RDDs

Start the Spark shell

```
$ spark-shell
```

Create some RDDs and combine them.

```
scala> val A = sc.parallelize((0 to 100).toList)

scala> val B = sc.parallelize((0 to 1000 by 10).toList)

scala> A.collect.sortWith(_<_)

...

scala> B.collect.sortWith(_<_)

...

scala> A.intersection(B).collect.sortWith(_<_)

res10: Array[Int] = Array(0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100)

scala> A.union(B).collect.sortWith(_<_)

res11: Array[Int] = Array(0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39, 40, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 50, 51, 52, 53, 54, 55, 56, 57, 58,
59, 60, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
80, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
100, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 260,
270, 280, 290, 300, 310, 320, 330, 340, 350, 360, 370, 380, 390, 400, 410, 420, 430, 440,
450, 460, 470, 480, 490, 500, 510, 520, 530, 540, 550, 560, 570, 580, 590, 600, 610, 620,
630, 640, 650, 660, 670, 680, 690, 700, 710, 720, 730, 740, 750, 760, 77...

scala> A.union(B).distinct.collect.sortWith(_<_)

res12: Array[Int] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64,
65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 110, 120, 130, 140, 150, 160, 170,
180, 190, 200, 210, 220, 230, 240, 250, 260, 270, 280, 290, 300, 310, 320, 330, 340, 350,
360, 370, 380, 390, 400, 410, 420, 430, 440, 450, 460, 470, 480, 490, 500, 510, 520, 530,
```

```
540, 550, 560, 570, 580, 590, 600, 610, 620, 630, 640, 650, 660, 670, 680, 690, 700, 710,
720, 730, 740, 750, 760, 770, 780, 790, 800, 810, 820, 830, 840, 850, 8...

scala> A.cartesian(B).collect

res13: Array[(Int, Int)] = Array((0,0), (0,10), (0,20), (0,30), (0,40), (0,50), (0,60),
(0,70), (0,80), (0,90), (0,100), (0,110), (0,120), (0,130), (0,140), (0,150), (0,160),
(0,170), (0,180), (0,190), (0,200), (0,210), (0,220), (0,230), (0,240), (0,250), (0,260),
(0,270), (0,280), (0,290), (0,300), (0,310), (0,320), (0,330), (0,340), (0,350), (0,360),
(0,370), (0,380), (0,390), (0,400), (0,410), (0,420), (0,430), (0,440), (0,450), (0,460),
(0,470), (0,480), (0,490), (1,0), (1,10), (1,20), (1,30), (1,40), (1,50), (1,60), (1,70),
(1,80), (1,90), (1,100), (1,110), (1,120), (1,130), (1,140), (1,150), (1,160), (1,170),
(1,180), (1,190), (1,200), (1,210), (1,220), (1,230), (1,240), (1,250), (1,260), (1,270),
(1,280), (1,290), (1,300), (1,310), (1,320), (1,330), (1,340), (1,350), (1,360), (1,...
```

Now we can emulate some *user* and *account* datasets.

```
scala> val users = sc.parallelize(List((1,"John"),(2,"Mary"),(3,"Mark"))) // (UserId,Name)
pairs

scala> users.collect()

res14: Array[(Int, String)] = Array((1,John), (2,Mary), (3,Mark))

scala> val accounts = sc.parallelize(List((1,("#1",1000)),(1,("#2",500)),(2,("#3",750))))
// (UserId, (AccountId,Salary)) pairs

scala> accounts.collect()

res15: Array[(Int, (String, Int))] = Array((1,(#1,1000)), (1,(#2,500)), (2,(#3,750)))

scala> users.join(accounts).foreach(println)

(1, (John,(#1,1000)) )

(1,(John,(#2,500)))

(2,(Mary,(#3,750)))

scala> users.cogroup(accounts).foreach(println)

(1,(CompactBuffer(John),CompactBuffer((#1,1000), (#2,500))))

(3,(CompactBuffer(Mark),CompactBuffer()))

(2,(CompactBuffer(Mary),CompactBuffer((#3,750))))
```

Exercise 7: Data Mining with MovieLens Dataset

This exercise will be performed using the MovieLens dataset which contains movie, user and rating records.

The datasets that we are going to use is:

- **movilens-1M** which contains 1M of ratings (for fast testing purposes)

We will use two files from this MovieLens dataset: “ratings.dat” and “users.dat”.

All ratings are contained in the “ratings.dat” file and are in the following format:

```
UserID::MovieID::Rating::Timestamp
```

Rating are in the scale 1-5 being 5 the best; 0 if not seen.

Sample:

```
$ cd
$ head -10 course-spark/datasets/movielens-1M/ratings.dat
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
1::2804::5::978300719
1::594::4::978302268
1::919::4::978301368
```

All users are contained in the “users.dat” file and are in the following format:

```
UserID::Gender::Age::Occupation::Zip-code
```

Age references an age range and occupation is an enumeration. Details on the values for each column can be found in the “README” file.

Sample:

```
$ cd
$ head -10 course-spark/datasets/movielens-1M/users.dat
==> movielens-1M/users.dat <==
1::F::1::10::48067
2::M::56::16::70072
3::M::25::15::55117
4::M::45::7::02460
5::M::25::20::55455
6::F::50::9::55117
7::M::35::1::06810
8::M::25::12::11413
9::M::25::17::61614
10::F::35::1::95370
```

The goal of this exercise is answering the following query:

```
select age, avg(rating)
from users u, ratings r
where u.id = r.userId
group by age
```

Start the Spark shell, this time with 2 local workers.

```
$ spark-shell --master local[2]
```

copy the following process (you also can write in separate file and later paste using :paste and ctrl-d for finish), try /* complete */ before checking the solution

```
scala> case class User(id:Int, gender:String, age:Int, occupation:String, zip:String)

scala> case class Rating(userId:Int, movieId:Int, rating:Int, tm:Long)

scala> val users = sc.textFile "file:///home/cloudera/course-spark/datasets/movielens-
1M/users.dat").

| map(_.split("::")).

| map(row => User(row(0).toInt, row(1), row(2).toInt, row(3), row(4))).cache

scala> val ratings = sc.textFile "file:///home/cloudera/course-spark/datasets/movielens-
1M/ratings.dat").

| map(_.split("::")).

| map(row => Rating(row(0).toInt, row(1).toInt, row(2).toInt, row(3).toLong)).cache

scala> users.take(5)

res1: Array[User] = Array(User(1,F,1,10,48067), User(2,M,56,16,70072),
User(3,M,25,15,55117), User(4,M,45,7,02460), User(5,M,25,20,55455))

scala> ratings.take(5)

res2: Array[Rating] = Array(Rating(1,1193,5,978300760), Rating(1,661,3,978302109),
Rating(1,914,3,978301968), Rating(1,3408,4,978300275), Rating(1,2355,5,978824291))

scala> val usersByUserId = users.map(user => (user.id, user))

scala> val ratingsByUserId = ratings.map(rating => (rating.userId, rating))

scala> usersByUserId.join(ratingsByUserId).first

res3: (Int, (User, Rating)) =
(4904,(User(4904,M,50,15,63121),Rating(4904,2054,4,962683594)))

scala> usersByUserId.join(ratingsByUserId).

| map(tuple => (tuple._1, tuple._2._1.age, tuple._2._2.rating)).

| map{case(userId, age, rating) => (age, rating)}.

| groupByKey.take(5)

res4: Array[(Int, Iterable[Int])] = Array((56,CompactBuffer(4, 2, 3, 5, 4, 3, 4, 4, 4, 3,
4, 4, 3, 4, 4, 5, 2, 4, 5, 3, 3, 4, 5, 4, 4, 4, 3, 3, 4, 4, 4, 3, 4, 3, 4, 4, 4, 3, 4, 4,
4, 5, 4, 3, 3, 4, 4, 4, 3, 5, 5, 4, 4, 5, 5, 5, 3, 4, 4, 4, 5, 3, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 3, 4, 2, 3, 4, 4, 3, 4, 3, 4, 4, 3, 4, 3, 4, 4, 3, 4, 4, 4, 4, 4, 3, 4,
5, 3, 2, 3, 5, 3, 3, 4, 3, 3, 4, 4, 3, 4, 4, 3, 4, 4, 4, 4, 2, 4, 4, 3, 4, 4, 3, 4, 4, 4,
4, 4, 4, 2, 4, 4, 4, 4, 4, 4, 4, 4, 3, 4, 2, 3, 4, 3, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 2, 5, 3, 2, 5, 4, 4, 4, 3, 4, 3, 4, 4, 4, 4, 3, 4, 3, 4, 4, 5, 4, 2, 4, 3, 4, 4, 4, 4,
4, 3, 2, 5, 4, 2, 4, 4, 4, 5, 3, 3, 4, 4, 4, 3, 4, 4, 3, 4, 4, 4, 4, 4, 3, 4, 2, 4, 5, 3,
4, 3, 2, 3, 4, 3, 4, 4, 4, 5, 4, 4, 3, 3, 3, 4, 4, 3, 3, 2, 4, 4, 3, 4, 4, 4,...

scala> usersByUserId.join(ratingsByUserId).

| map(tuple => (tuple._1, tuple._2._1.age, tuple._2._2.rating)).

| map{case(userId, age, rating) => (age, rating)}.

| groupByKey./* complete */

res5: Array[(Int, Double)] = Array((56,3.766632284682826), (50,3.714512346530556),
(18,3.5075734460814227), (25,3.5452350615336385), (35,3.6181615352532375),
(45,3.638061530735475), (1,3.549520414538238))
```

Solution

```
usersByUserId.join(ratingsByUserId).

| map(tuple => (tuple._1, tuple._2._1.age, tuple._2._2.rating)).
```

```
| map{case(userId, age, rating) => (age, rating)}.
| groupByKey.map{case(age, ratings) => (age, ratings.sum.toDouble/ratings.toList.size)}.
| collect
```

Exercise 8: Transformations per partition

Review the data in */home/cloudera/course-spark/datasets/activations*. In each folder, each XML file contains data for all the devices of a telecommunications company that have been activated by customers during a specific month. The **goal** is to go through this set of activation files and find the **most common type of activated device**.

```
<activations>
  <activation>
    <activation timestamp="1225499258" type="phone">
      <account-number>316</account-number>
      <device-id>
        d61b6971-33e1-42f0-bb15-aa2ae3cd8680
      </device-id>
      <phone-number>5108307062</phone-number>
      <model>iFruit 1</model>
  </activation>
 …
</activations>
```

The **output** should be something like:

```
iFruit 1 (400)
Sorrento FOOL (301)
```

The input data will be automatically partitioned by file. The processing should be defined per partition.

Copy the data to HDFS

```
$ cd
$ hadoop fs –put course-spark/datasets/activations input-spark
```

Start the spark-shell

```
$ spark-shell
```

Import the scala libraries to process xml files

```
import scala.xml._
```

Load the XML files. Each XML file will constitute a partition

```
//get the activation XML files
```

```
var filename = “/user/cloudera/input-spark/activations”
var activations = sc.textFile(filename)
```

Define an utility function to get from each file the tree of the xml and then apply the map per partitions:

```
def getactivations(fileiterator: Iterator[String]): Iterator[Node] = {
val nodes = XML.loadString(fileiterator.mkString) \\ “activation”
nodes.toIterator
}
var activationTrees = activations.mapPartitions(getactivations)
```

Define an utility function to get the model name from the activation record and map each activation record to a device model name

```
def getmodel(activation: Node): String = {
(activation \ “model”).text
}
var models = activationTrees.map(getmodel)
```

Show the partitioning and count the activations per model

```
println(models.toDebugString)
var modelcounts = models.map(model => (model,1)).reduceByKey((v1,v2) => v1+v2)
println(modelcounts.toDebugString)
```

Display the top 10 models

```
for (pair <- modelcounts.map(_.swap).top(10))
println(“model %s (%s)”.format(pair._2, pair._1))
```

Exercise 9: Viewing Stages in the Spark Application UI

This exercise we will examine the stages, tasks, resources and graph of a Spark job in the Spark Application UI.

Start the spark-shell. In the Spark-shell run the job of the previous exercise with the help of the command **:load activations.scala**. The file **activations.scala** must be in the same directory where you invoke spark-shell.

```
> spark-shell
..
$ :load activations.scala
…
```

Open a browser **inside the VM**. Go to the address

http://localhost:4040

Click on the Tab Jobs and then explore the Job information: Stages, Tasks, DAG, etc…

Exercise 10: Using Broadcast variables

In this exercise we will show a practical use of broadcast variables. A company wants to do some analysis on the web traffic produce from specific devices. The goal is to filter the web server logs to include only those requests from devices specified in a certain list.

- Upload to HDFS the weblog files

```
> hadoop fs –put weblogs input-spark
```

- Start the spark-shell

```
$ spark-shell
```

- Define the location of the log files in HDFS and the target models in a local file.

```
$ val logfile = “input-spark/weblogs”
$ val targetfile = “/home/cloudera/targetmodels.txt”
```

- Read the target models in a variable and define the broadcast variable

```
$ import scala.io.Source
$ val targetlist = Source.fromFile(targetfile).getLines.toList
$ //broadcast the target list to all workers
$ val targetlistbc = /* fill here */
```

- Filter out requests from devices not in the target list and print some of them

```
$ //filter out requests from devices not in the target list
val targetregs = sc.textFile(logfile)
.filter(line => targetlistbc.value.count(line.contains(_)) > 0)
$ targetregs.take(5).foreach(println)
```

Solution

```
$ //broadcast the target list to all workers
$ val targetlistbc = sc.broadcast(targetlist)
```

Exercise 11: Using Accumulators

Using accumulators, count the number of each type of files (HTML, CSS or JPG) requested in a web as found in the web server log files. To determine the type of request use the file extension: .html, .css or .jpg.

- It is assumed that the weblogs folder is already in HDFS (see previous exercise).

- Start the spark-shell and define as accumulators the counters for each type of file

```
> spark-shell
$ val jpgcount = /* fill here */
```

```
$ val htmlcount = /* fill here */
$ val csscount = /* fill here */
```

- Define the input file and load the files

```
$ val filename = “input-spark/weblogs”
$ val logs = sc.textFile(filename)
```

Filter the logs and increment the counters

```
$ logs.foreach(line => {
if (line.contains(“.html”)) htmlcount += 1
else if(line.contains(“.jpg”)) jpgcount += 1
else if(line.contains(“.css”)) csscount += 1
})
```

Print out the totals

```
println(“Request Totals: “)
println(“.css requests: “ + csscount.value)
println(“.html requests: “ + htmlcount.value)
println(“.jpg requests: “ + jpgcount.value)
```

Solution

```
$ val jpgcount = sc.accumulator(0)
$ val htmlcount = sc.accumulator(0)
$ val ccscount = sc.accumulator(0)
```

Exercise 12: Average WordLength

This exercise will be performed using “El Quixote” book dataset and consists of determining the average length of the words of the book.

Although this problem can be solved in different ways, the goal is do it using Spark’ shared variables.

Start the Spark shell

```
$ spark-shell
```

Copy the following process and **/ * complete */**

```
scala> val quixote = sc.textFile("/user/cloudera/input-spark/quixote.txt"")
scala> val wordLengths = quixote.flatMap(line => line.split(" ")).
| filter(word => word.length > 4 && !List("Quixote", "Sancho").contains(word)).
| map(_.length).cache
scala> val accum = sc.accumulator(0, "AccumLength")
scala> wordLengths.foreach(length => accum += length)
scala> /* complete */
res7: Double = 6.8908771410364515
```

Solution

```
accum.value.toDouble / wordLengths.count
```

Exercise 13: Data Mining Using SparkSQL

Do the same as in exercise 6 but this time using SparkSQL. We will not use SparkHive because this requires the installation of Hive, a bit cumbersome for a short course. From a programming point of view to use Hive you should change SQL → Hive

Start the Spark shell

```
$ spark-shell
```

Copy the following process and / * complete */:

```
val sqlContext = new org.apache.spark.sql.SQLContext (sc)

// this is used to implicitly convert an RDD to a DataFrame.

import sqlContext.implicits._

case class User (id: Int, gender: String, age: Int, occupation: String, zip: String)

case class Rating (userId: Int, movieId: Int, rating: Int, tm: Long)

val users = sc.textFile ("file:// /home/cloudera/course-spark/datasets/movielens-
1M/users.dat").map (_.split("::")).map(row => User (row(0).toInt, row(1), row(2).toInt,
row(3), row(4))).toDF

val ratings = sc.textFile ("file:///home/cloudera/course-spark/datasets/movielens-
1M/ratings.dat").

map (_.split("::")).map(row => Rating (row (0).toInt, row(1).toInt, row(2).toInt,
row(3).toLong)).toDF

users.show

ratings.show

users.registerTempTable("USERS")

ratings.registerTempTable("RATINGS")

val resultSet = sqlContext.sql ("/* complete */")

resultSet.collect

res12: Array[org.apache.spark.sql.Row] = Array([35,3.6181615352532375],
[45,3.638061530735475], [50,3.714512346530556], [56,3.766632284682826],
[1,3.549520414538238], [18,3.5075734460814227], [25,3.5452350615336385])
```

Solution

```
SELECT age, AVG(rating) FROM USERS JOIN RATINGS ON (USERS.id = RATINGS.userId) GROUP BY age
```

Exercise 14: Running spark jobs inside IntelliJ IDE

Setting the environment

In order to run exercises let's prepare intelliJ IDE for running the code inside the IDE

1. Inside the IDE go to: **File / Project / Scala (left list) / SBT (right list) / next / name: sparktraining / scala version 2.10.5 / Use auto-import / Finish**

2. in build.sbt file add next code (leave a blank space inter lines, that's the way that sbt parses commands):

```
name := "sparktraining"


version := "1.0"


scalaVersion := "2.10.0"


//add this code
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.5.0",
  "org.apache.spark" %% "spark-mllib" % "1.5.0",
  "org.apache.spark" %% "spark-sql" % "1.5.0",
  "com.databricks" %% "spark-csv" % "1.3.0"
)
```

2. Wait for the project will be created / At the level **[sparktraining/src/main/scala]** create **package com.gft.sparktraining** / and a new scala class (like an object) called **TestSpark** (suffix .scala is added for you)
3. Add the following code for testing in TestSpark:

```
package com.gft.sparktraining
import org.apache.spark._
import org.apache.spark.rdd.RDD
object TestSpark {
def main (args: Array[String]): Unit = {
val conf = new SparkConf().setAppName ("TestSpark")
val sc = new SparkContext (conf)
val rdd = sc.parallelize(List (1,2,3,4,5))
rdd.collect().foreach (println)
}
}
```

4. If syntax highlight is not recognized close & open IntelliJ or install SBT console plugin and type reload or show SBT project and click refresh SBT project
5. In the project tree, select the Node **TestSpark / click right / select 'Run TestSpark'** / program will fail due Spark Master is not selected

```
Exception in thread "main" org.apache.spark.SparkException: A master URL must be set in
your configuration
```

6. Go to **Run / Edit configurations / VM Options** / type:

```
-Dspark.master=local
```

7. Repeat step 5

Exercise 15: Dataframes

Use the csv format reader, dataframes and OlympicAthletes.csv data set to **/* complete */** the code:

```
package com.gft.sparktraining


import org.apache.log4j.{Level, Logger}
import org.apache.spark._
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.functions._
object OlympicAthletes extends App{
  //Suppress Spark output
  Logger.getLogger("org").setLevel(Level.ERROR)
  Logger.getLogger("akka").setLevel(Level.ERROR)
  val conf = new SparkConf().setAppName ("OlympicAthletes").
    set("spark.executor.memory", "1g")
  val sc = new SparkContext(conf)
  val sqlContext = new org.apache.spark.sql.SQLContext(sc)
  //Athlete,Age,Country,Game,Date,Sport,Gold,Silver,Bronze,Total


  import com.gft.sparktraining.OlympicAthletes.sqlContext.implicits._
  val df:DataFrame = sqlContext.read.format("com.databricks.spark.csv").option("header",
"true")
    .load("datasets/OlympicAthletes.csv")
  df.registerTempTable("Records")
  //doing some agregations
  df.select("Country", "Total").groupBy("Country").agg(sum($"Total"), avg($"Total"))
    .foreach(println)
  //How many athletes by country are older than 40
  sqlContext.sql("/* complete 1 */")
    .foreach(println)
  sqlContext.udf.register("strLen", (s: String) => s.length())
  //use the registered udf to print country with its number or letters
  sqlContext.sql("/* complete 2 */").foreach(println)
  System.in.read()
}
```

Solution

```
/* complete 1 */
SELECT Country,count(1) from Records where Age > 40 group by Country
/* complete 2 */
SELECT Country, strLen(Country) from Records
```

Exercise 16: Dataframes

In the exercise we will create a dataframe with the content of a tweeter JSON file.

We want to:

- print the dataframe
- print the schema of the dataframe
- find people who are located in Paris
- find the user who tweets the more

We will use a dataset with 8198 tweets, where a tweet looks like that:

```
{"id":"572692378957430785",
    "user":"Srkian_nishu :)",
    "text":"@always_nidhi @YouTube no i dnt understand bt i loved of this mve is rocking",
    "place":"Orissa",
    "country":"India"}
```

Use the DataFrameOnTweetsTest to implement the code and fill the // TODOS with reduced-tweets.json dataset

```
package com.gft.sparktraining
import org.apache.spark._
import org.apache.spark.sql._


object DataFrameOnTweets extends App{


  val pathToFile = "datasets/reduced-tweets.json"

  /**
   *  Here the method to create the contexts (Spark and SQL) and
   *  then create the dataframe.
   */
  def loadData(): DataFrame = {
    // create spark configuration and spark context
    val conf = new SparkConf()
        .setAppName("Dataframe")
        .setMaster("local[*]")

    val sc = new SparkContext(conf)

    // Create a sql context: the SQLContext wraps the SparkContext, and is specific to
Spark SQL.
    // It is the entry point in Spark SQL.
    // TODO write code here
    val sqlcontext = null

    // Load the data regarding the file is a json file
    // Hint: use the sqlContext and apply the read method before loading the json file
    // TODO write code here
    null
  }


  /**
   *  See how looks the dataframe
   */
  def showDataFrame() = {
    val dataframe = loadData()

    // Displays the content of the DataFrame to stdout
```

```
    // TODO write code here
  }

  /**
   * Print the schema
   */
  def printSchema() = {
    val dataframe = loadData()

    // Print the schema
    // TODO write code here
  }

  /**
   * Find people who are located in Paris
   */
  def filterByLocation(): DataFrame = {
    val dataframe = loadData()

    // Select all the persons which are located in Paris
    // TODO write code here
    null
  }


  /**
   *  Find the user who tweets the more
   */
  def mostPopularTwitterer(): (Long, String) = {
    val dataframe = loadData()

    // First group the tweets by user
    // Then sort by descending order and take the first one
    // TODO write code here
    null
  }

}
```

Solution

```
package com.gft.sparktraining

import org.apache.spark._
import org.apache.spark.sql._

object DataFrameOnTweets extends App {

  val pathToFile = "datasets/reduced-tweets.json"

  /**
   *  Here the method to create the contexts (Spark and SQL) and
   *  then create the dataframe.
   */
  def loadData(): DataFrame = {
    // create spark configuration and spark context
    val conf = new SparkConf()
        .setAppName("Dataframe")
        .setMaster("local[*]")

    val sc = new SparkContext(conf)

    // Create a sql context: the SQLContext wraps the SparkContext, and is specific to
Spark SQL.
    // It is the entry point in Spark SQL.
    val sqlcontext = new SQLContext(sc)
```

```
    // Load the data regarding the file is a json file
    sqlcontext.read.json(pathToFile)
  }

  /**
   *  See how looks the dataframe
   */
  def showDataFrame() = {
    val dataframe = loadData
    dataframe.show
  }

  /**
   * Print the schema
   */
  def printSchema() = {
    val dataframe = loadData
    dataframe.printSchema
  }

  /**
   * Find people who are located in Paris
   */
  def filterByLocation(): DataFrame = {
    val dataframe = loadData

      dataframe.filter(dataframe.col("place").equalTo("Paris")).toDF()
  }

  /**
   *  Find the user who tweets the more
   */
  def mostPopularTwitterer(): (Long, String) = {
    val dataframe = loadData

    // First group the tweets by user
    // Then sort by descending order and take the first one
    dataframe.groupBy(dataframe.col("user"))
        .count
        .rdd
        .map(x => (x.get(1).asInstanceOf[Long], x.get(0).asInstanceOf[String]))
        .sortBy(_._1, false, 1)
        .first
  }

  showDataFrame

  //printSchema

  //filterByLocation.show()

  //println (mostPopularTwitterer)
}
```

Exercise 17: Spark RDD testing

You've written an awesome program in Spark and now its time to write some tests (ok we are not doing TDD). We are going to use spark-testing-base a library for spark testing that is being developing by Holden Karau one of the main Spark Contributors. The library still in process but makes easy testing RDDs and DataFrames avoiding boilerplate, so we're going to use

First point is adding dependencies to your project in order to use it. For example if you are using Spark 1.5.0 add to your sbt dependencies (looking the best fit in maven central repository):

```
"com.holdenkarau" %% "spark-testing-base" % "1.5.0_0.3.0"
"org.scalatest" %% "scalatest" % "2.2.1",
"junit" % "junit" % "4.10",
```

But at the moment of writing this exercises book release version has some bugs so after compile master branch of project a jar file is provided for you in Gft-Spark-Exercices directory for adding to intelliJ project so in File / Project Structure / Libraries, click in New Project Library / Java and add spark-testing-base_2.10-0.1.3.jar file

Now we are going to test a RDD

Our test classes have to live in src/test/scala/ and extend of FunSuite for assertions checks and use the trait SharedSparkContext for using a test Spark context ready for us and called sc

Replace /*complete x*/ with valid spark code for compiling and passing the tests

```
import com.holdenkarau.spark.testing.{RDDComparisons, SharedSparkContext}
import org.scalatest.FunSuite

class SampleRDDTest extends FunSuite with SharedSparkContext {
  test("really simple transformation") {
    val input = List("hi", "hi holden", "bye")
    val expected = List(List("hi"), List("hi", "holden"), List("bye"))
    assert(sc.parallelize(input)./*complete 1*/ === expected)
  }

  test("really simple transformation with rdd - rdd comparision") {
    val input = List("hi", "hi holden", "bye")
    val expected = List(List("hi"), List("hi", "holden"), List("bye"))
    assert(None ===
      RDDComparisons.compare(sc.parallelize(expected), /* complete2 */)
  }
}
```

Solution

```
/*complete 1*/
assert(sc.parallelize(input).map(_.split(" ").toList).collect().toList === expected)
/*complete2 */
RDDComparisions.compare(sc.parallelize(expected), sc.parallelize(input).map(_.split("
").toList))
```

Exercise 18: Spark DataFrame testing exercise

For testing DataFrames we have to add a new trait to our test classes: DataFrameSuitBase

```
import com.holdenkarau.spark.testing.{DataFrameSuiteBase, SharedSparkContext}
import org.scalatest.FunSuite
```

```
class SampleDataFrameTest extends FunSuite with SharedSparkContext with DataFrameSuiteBase
{

  val diffByteArray = Array[Byte](192.toByte)
  val inputList = List(Magic("panda", 9001.0),
    Magic("coffee", 9002.0))
  val inputList2 = List(Magic("panda", 9001.0 + 1E-6),
    Magic("coffee", 9002.0))

  test("dataframe should be equal to its self") {
    val sqlCtx = sqlContext
    import sqlCtx.implicits._
    val input = sc.parallelize(inputList).toDF
    equalDataFrames(input, /* complete 1 */)
  }
  test("unequal dataframes should not be equal") {
    val sqlCtx = sqlContext
    import sqlCtx.implicits._
    val input = sc.parallelize(inputList).toDF
    val input2 = sc.parallelize /* complete 2.1 */
    intercept[org.scalatest.exceptions.TestFailedException] {
    equalDataFrames(input, /* complete 2.2 */)
    }
  }
  test("unequal dataframes should not be equal when length differs") {
    val sqlCtx = sqlContext
    import sqlCtx.implicits._
    val input = /* complete 3 */
    val input2 = sc.parallelize(inputList.headOption.toSeq).toDF
    intercept[org.scalatest.exceptions.TestFailedException] {
    equalDataFrames(input, input2)
    }
  }
}

case class Magic(name: String, power: Double)
```

Solution

```
/*complete 1*/

equalDataFrames(input, input)

/*complete 2.1 2.2*/

val input2 = sc.parallelize(inputList2).toDF
    intercept[org.scalatest.exceptions.TestFailedException] {
      equalDataFrames(input, input2)
    }

/*complete 3*/

sc.parallelize(inputList).toDF
```

Exercise 19: Recommender Algorithms

Exercises in this chapter will be performed using the MovieLens dataset which contains movie and rating records.

We will use the dataset:

- **movilens-1M** which contains 1M of ratings (for fast testing purposes)

We will use two files from this MovieLens dataset: “ratings.dat” and “movies.dat”.

All ratings are contained in the “ratings.dat” file and are in the following format:

```
UserID::MovieID::Rating::Timestamp
```

Rating are in the scale 1-5 being 5 the best; 0 if not seen.

Sample:

```
[root@sandbox datasets]# head -10 movielens-1M/ratings.dat
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
1::2804::5::978300719
1::594::4::978302268
1::919::4::978301368
```

Movie information is in the “movies.dat” file and is in the following format:

```
MovieID::Title::Genres
```

Sample:

```
[root@sandbox datasets]# head -10 movielens-1M/movies.dat
1::Toy Story (1995)::Animation|Children's|Comedy
2::Jumanji (1995)::Adventure|Children's|Fantasy
3::Grumpier Old Men (1995)::Comedy|Romance
4::Waiting to Exhale (1995)::Comedy|Drama
5::Father of the Bride Part II (1995)::Comedy
6::Heat (1995)::Action|Crime|Thriller
7::Sabrina (1995)::Comedy|Romance
8::Tom and Huck (1995)::Adventure|Children's
9::Sudden Death (1995)::Action
10::GoldenEye (1995)::Action|Adventure|Thriller
```

Collaborative filtering is commonly used for recommender systems. These techniques aim to fill in the missing entries of a user-item association matrix. Spark MLlib currently supports model-based collaborative filtering, in which users and items are described by a small set of latent factors that can be used to predict missing entries. In particular, we implement the **alternating least squares** (**ALS**) algorithm to learn these latent factors.

Create a new SBT project **MovieLensALS** (Scala 2.10.5, jdk 1.7), add spark core and mlib dependencies and copy **MovieLensALS.scala** in IntelliJ inside of **com.gft.sparktraining** package,

datasets directory should be at the same level that build.sbt file in order to use **movies.dat** and **ratings.dat**.

```
name := "MovieLensALS"


version := "1.0"


scalaVersion := "2.10.5"


libraryDependencies ++= Seq(
// Spark dependency
"org.apache.spark" %% "spark-core" % "1.3.1",
"org.apache.spark" %% "spark-mllib" % "1.3.1"
)
```

The MovieLensALS.scala it is the base file and it will be completed in the following exercises. Have a look to the code:

```
package com.gft.sparktraining


/**
* Created by chicochica10 on 6/06/15.
*/


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


object MovieLensALS {
def main (args: Array[String]): Unit = {
val conf = new SparkConf().setAppName ("MovieLensALS").
set("spark.executor.memory", "1g")
val sc = new SparkContext (conf)


//load personal ratings
val myRatings = loadRatings (args(0))
val myRatingsRDD = sc.parallelize(myRatings)
//load ratings and movie titles
val movieLensHomeDir = args (1)
// ratings is an RDD of (last digit of timestamp, (userId, movieId, rating))
```

```
val ratings = sc.textFile(movieLensHomeDir + "ratings.dat").map(parseRating)
// movies is an RDD of (movieId, movieTitle)
val movies = sc.textFile(movieLensHomeDir + "movies.dat").map(parseMovie)
//used later in recomendation
val moviesMap = movies.collect.toMap
//your code here
//Shut down the SparkContext.
sc.stop()
}
//load ratings from file
def loadRatings (ratingsFile: String) = ???
//Parses a rating record in MovieLens format userId::movieId::rating::timestamp .
def parseRating (line: String) = ???
//Parses a movie record in MovieLens format movieId::movieTitle .
def parseMovie (line: String) = ???
//Compute RMSE (Root Mean Squared Error).
def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating], n: Long): Double = {
val predictions: RDD[Rating] = model.predict(data.map(x => (x.user, x.product)))
val predictionsAndRatings = predictions.map(x => ((x.user, x.product), x.rating))
.join(data.map(x => ((x.user, x.product), x.rating)))
.values
math.sqrt(predictionsAndRatings.map(x => (x._1 - x._2) * (x._1 - x._2)).reduce(_ + _) / n)
}
}
```

Rate Movies

To make recommendation for you, we are going to learn your taste by asking you to rate a few movies. We have selected a small set of movies that have received the most ratings from users in the MovieLens dataset. You can rate those movies by running “rateMovies” script (if python is not installed on your computer, a **personalRatings.txt** is provided in datasets directory)

```
python bin/rateMovies.py
```

When you run the script, you should see prompt similar to the following:

```
Please rate the following movie (1-5 (best), or 0 if not seen):
Toy Story (1995):
```

After you’re done rating the movies, we save your ratings in **personalRatings.txt** in the MovieLens format, where a special **user id 0** is assigned to you.

Once executed **personalRatings.txt** will look like as follows. **Copy** the file in IntelliJ at the same level that build.sbt.

```
$ cat personalRatings.txt
0::1::5::1433610750
```

```
0::780::2::1433610750
0::590::4::1433610750
0::1210::5::1433610750
0::648::3::1433610750
0::344::2::1433610750
0::165::3::1433610750
0::597::3::1433610750
0::1580::4::1433610750
0::231::1::1433610750
```

Setup

Complete the parseRating function.

```
//Parses a rating record in MovieLens format userId::movieId::rating::timestamp .
def parseRating (line: String) = {
val fields = line.split("::")
//to make later the partition easy for training, validation and test
(fields(3).toLong %10, Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble))
}
```

Complete the parseMovie function.

Implement the *parseMovie* method. It should return two values, the movie id and the movie title.

```
//Parses a movie record in MovieLens format movieId::movieTitle .
def parseMovie (line: String) = {
val fields = line.split("::")
(fields(0).toInt, fields(1))
}
```

Write code to load personal ratings file.

```
///load ratings from file
def loadRatings (ratingsFile: String) = {
try {
val fileContents = Source.fromFile(ratingsFile).getLines.toList
val ratings = fileContents.map { line =>
val fields = line.split("::")
Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
}.filter(_.rating > 0.0)
if (ratings.isEmpty)
sys.error("No ratings provided.")
else
ratings
} catch {
case ex: Exception => println(ex)
```

```
sys.exit(-1)
}
}
```

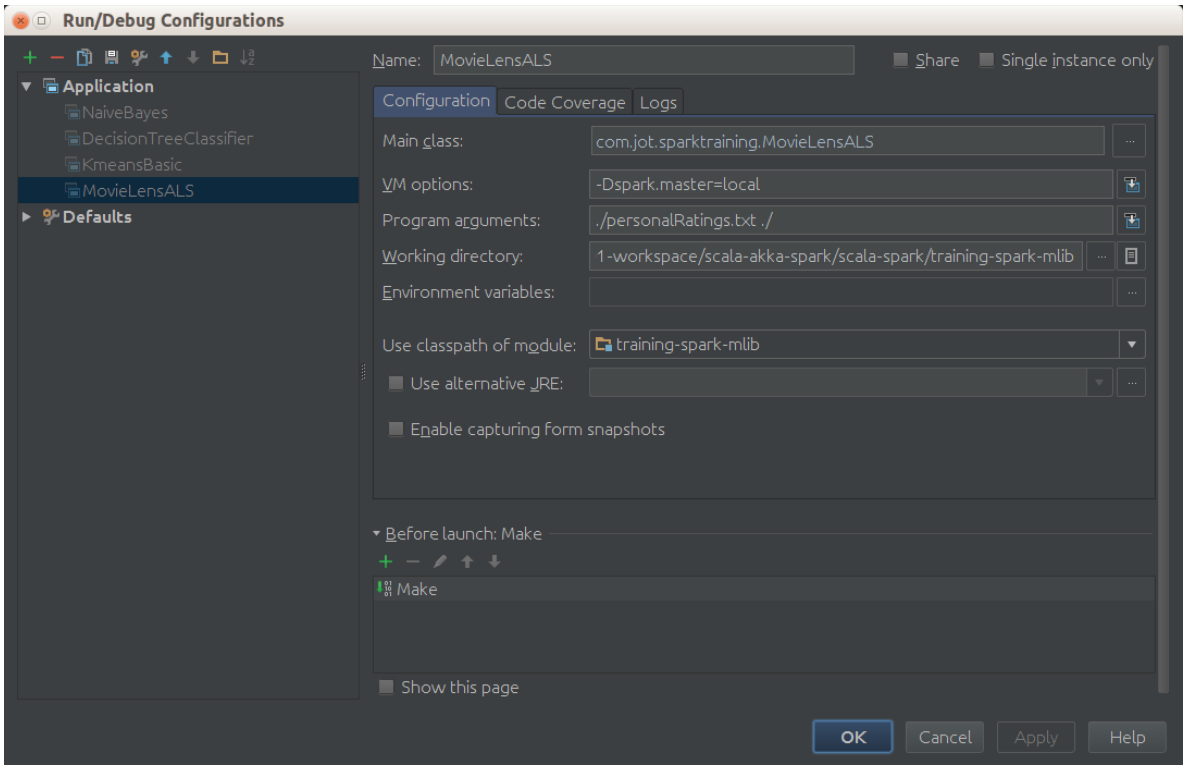
Write code to print a summary of the ratings.

Add code to write the number of ratings, the number of (unique) users and the number of (unique) movies.

```
//your code here
val numRatings = ratings.count
val numUsers = ratings.values.map(r => r.user).distinct().count
val numMovies = ratings.values.map(r => r.product).distinct().count
println(s"Got ${numRatings} ratings from ${numUsers} users on ${numMovies} movies.")
```

Execute.

Complete the Run/Debug Configurations form adding VM options: -Dspark.master=local and Program arguments: **./datasets/personalRatings.txt ./datasets/movielens-1M/** and Run the program




```
...
Got 1000209 ratings from 6040 users on 3706 movies.
...
```

Splitting Training Data

ALS has training parameters such as rank for matrix factors and regularization constants. To determine a good combination of the training parameters, we split the data into three non-overlapping subsets:

- trainging set
- test set
- and validation set

The split is based on the last digit of the timestamp. We will train multiple models based on the training set, select the best model on the validation set based on **RMSE**, and finally evaluate the best model on the test set. We also add your ratings to the training set to make recommendations for you. We hold the training, validation, and test sets in memory by calling cache because we need to visit them multiple times.

Write code to split training data.

```
// split ratings into train (60%), validation (20%), and test (20%) based on the
// last digit of the timestamp, add myRatings to train, and cache them
// training, validation, test are all RDDs of (userId, movieId, rating)
// val numPartitions = 4 //<-- not used because executed in local
val training = ratings.filter(pair => pair._1 < 6).values.union(myRatingsRDD).
/*repartition(numPartitions).*/ cache()
val validation = ratings.filter(pair => pair._1 >= 6 && pair._1 < 8).values.
/*repartition(numPartitions).*/ cache()
val test = ratings.filter(pair => pair._1 >= 8).values. /*repartition(numPartitions).*/
cache()
val numTraining = training.count
val numValidation = validation.count
val numTest = test.count
println(s"Training: ${numTraining}, Validation: ${numValidation}, numTest: ${numTest}")
```

Note: The repartition() operator will randomly shuffle an RDD into the desired number of partitions. Useful in a cluster because for training model the level of parallelism is determined automatically based on the number of partitions.

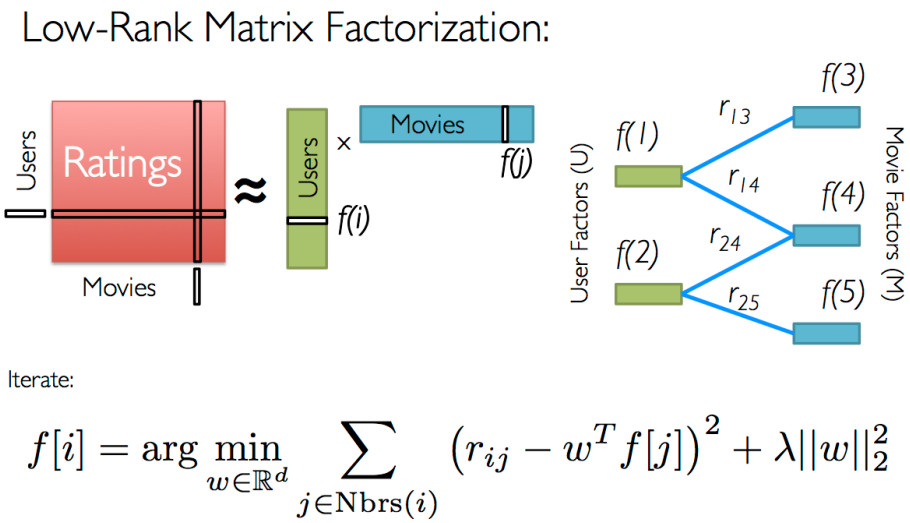
Execute again.

```
Got 1000209 ratings from 6040 users on 3706 movies.
...
Training: 602251, Validation: 198919, numTest: 199049
...
```

Training using ALS

In this section, we will use **ALS.train**[1] to train a bunch of models, and select and evaluate the best. Among the training paramters of ALS, the most important ones are :

- rank,
- lambda (regularization constant),
- number of iterations.



Iterate:

$$f[i] = \arg\min_{w \in \mathbb{R}^d} \sum_{j \in \mathrm{Nbrs}(i)} \left(r_{ij} - w^T f[j]\right)^2 + \lambda ||w||_2^2$$

The idea of the algorithm is build a factorization matrix model as a product of a **rank** (numbers of ratings fixed by parameter) of user ratings multiplied by transposed movies ratings and later apply the rest of algorithm that is iterative and needs a **number of iterations to** converge

Ideally, we want to try a large number of combinations of those parameters in order to find the best one. Due to time constraint, we will test only 8 combinations resulting from the cross product of 2 different ranks (8 and 12), 2 different lambdas (0.1 and 10.0), and 2 different numbers of iterations (10 and 20). We use the provided method computeRmse to compute the RMSE on the **validation** set for each model. The model with the smallest RMSE on the validation set becomes the one selected and its RMSE on the test set is used as the final metric.

Write the code to choose the best model.

```
// train models and evaluate them on the validation set
val ranks = List(8, 12)
val lambdas = List(0.1, 10.0)
val numIters = List(10, 20)
var bestModel: Option[MatrixFactorizationModel] = None
var bestValidationRmse = Double.MaxValue
var bestRank = 0
var bestLambda = -1.0
var bestNumIter = -1
for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
```

[1] https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.recommendation.ALS$

```
val model = ALS.train(training, rank, numIter, lambda)
val validationRmse = computeRmse(model, validation, numValidation)
println("RMSE (validation) = " + validationRmse + " for the model trained with rank = "
+ rank + ", lambda = " + lambda + ", and numIter = " + numIter + ".")
if (validationRmse < bestValidationRmse) {
bestModel = Some(model)
bestValidationRmse = validationRmse
bestRank = rank
bestLambda = lambda
bestNumIter = numIter
}
}
// evaluate the best model on the test set
val testRmse = computeRmse(bestModel.get, test, numTest)
println("The best model was trained with rank = " + bestRank + " and lambda = " +
bestLambda
+ ", and numIter = " + bestNumIter + ", and its RMSE on the test set is " + testRmse + ".")
```

Execute again.

```
...
Got 1000209 ratings from 6040 users on 3706 movies.
...
Training: 602252, validation: 198919, test: 199049
...
The best model was trained with rank = 12 and lambda = 0.1, and numIter = 20, and its RMSE
on the test set is 0.8687702604687833.
```

Recommend Movies for You

As the last exercise of this chapter, let's take a look at what movies our model recommends for you. This is done by generating (0, movieId) pairs for all movies you haven't rated and calling the **MatrixFactorizationModel#predict**[2] method to get predictions. Note 0 is the special user id assigned to you.

Write the code to list the top 50 recommendations.

```
// make personalized recommendations
val myRatedMovieIds = myRatings.map(_.product)
val moviesNotSeen = movies.keys.filter(!myRatedMovieIds.contains(_)).collect()
val candidates = sc.parallelize(moviesNotSeen)
val recommendations = bestModel.get
.predict(candidates.map((0, _)))
.collect()
.sortBy(- _.rating)
```

[2] https://spark.apache.org/docs/1.2.0/api/scala/index.html#org.apache.spark.mllib.recommendation.MatrixFactorizationModel

```
.take(50)
var i = 1
println("Movies recommended for you:")
recommendations.foreach { r =>
println("%2d".format(i) + ": " + moviesMap(r.product))
i += 1
}
```

Execute again.

```
Movies recommended for you:
1: Circus, The (1928)
2: Specials, The (2000)
3: Bandits (1997)
4: Nobody Loves Me (Keiner liebt mich) (1994)
5: General, The (1927)
6: Trouble in Paradise (1932)
7: Bewegte Mann, Der (1994)
8: For All Mankind (1989)
9: Casablanca (1942)
10: Wrong Trousers, The (1993)
11: Yojimbo (1961)
12: Leather Jacket Love Story (1997)
13: Third Man, The (1949)
14: Philadelphia Story, The (1940)
15: Man of the Century (1999)
16: Close Shave, A (1995)
17: Double Happiness (1994)
18: Time of the Gypsies (Dom za vesanje) (1989)
19: Lodger, The (1926)
20: It Happened One Night (1934)
21: Gold Rush, The (1925)
22: To Kill a Mockingbird (1962)
23: Grand Day Out, A (1992)
24: Mr. Smith Goes to Washington (1939)
25: Rear Window (1954)
26: Wallace & Gromit: The Best of Aardman Animation (1996)
27: Maltese Falcon, The (1941)
28: Lady Vanishes, The (1938)
29: I Confess (1953)
30: World of Apu, The (Apur Sansar) (1959)
31: Gay Divorcee, The (1934)
32: Before the Rain (Pred dozhdot) (1994)
```

```
33: Freedom for Us (□ nous la libert□ ) (1931)
34: African Queen, The (1951)
35: Anatomy of a Murder (1959)
36: Double Indemnity (1944)
37: Singin' in the Rain (1952)
38: All About Eve (1950)
39: Solas (1999)
40: Bridge on the River Kwai, The (1957)
41: Strangers on a Train (1951)
42: City Lights (1931)
43: Creature Comforts (1990)
44: Lawrence of Arabia (1962)
45: Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)
46: Seven Samurai (The Magnificent Seven) (Shichinin no samurai) (1954)
47: Spiral Staircase, The (1946)
48: Fantasia (1940)
49: You Can't Take It With You (1938)
50: Rich and Strange (1932)
```

(while in execution)