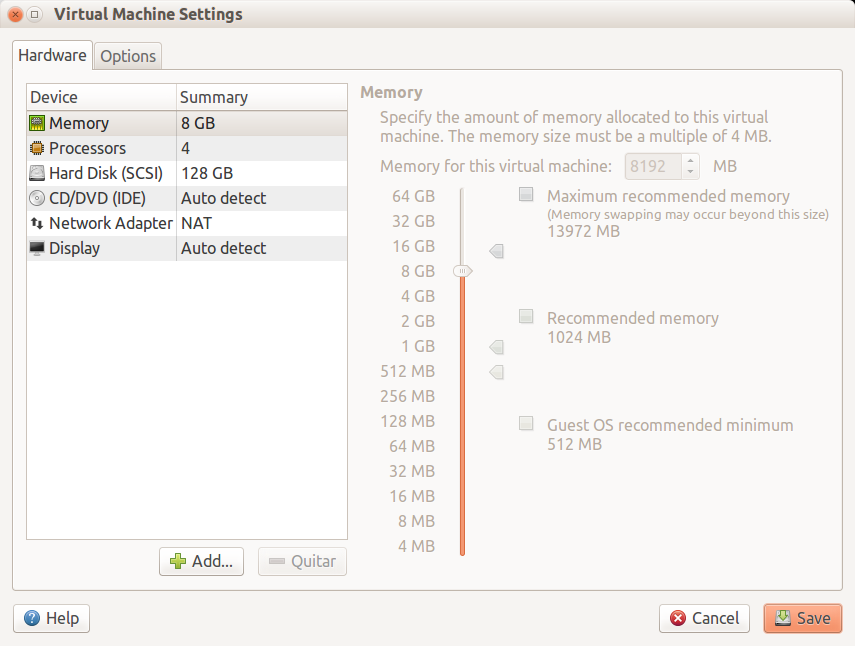
# Exercise 0: Prepare Environment

Download Hortonworks Data Platform – HDP 2.2.4 (VMWare or Virtual Box):

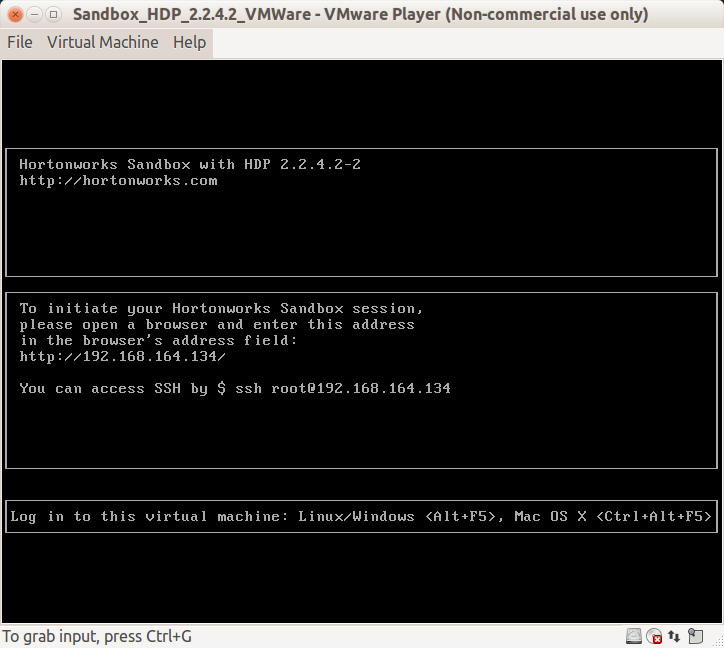
<http://hortonworks.com/hdp/downloads/>

Once downloaded start the VM.

Ensure that VM has at least 4 GB of RAM



When finish take note of a:ssigned ip address



* User: root
* Pass: Hadoop

Once logged in the VM:

1. Using FileZilla or similar (Winscp) upload Gft-Spark-Exercices.zip. You can also use scp from your local machine:

$ scp Gft-Spark-Exercices.zip [root@192.168.164.134](mailto:root@192.168.164.134):/root

1. in VM machine uncompress the file

$ unzip Gft-Spark-Exercices.zip

$ cd Gft-Spark-Exercices

$ ls

bin datasets exercise1-1 exercise2-1 exercise2-2 exercise3-1 exercise3-2 exercise4

Part of the Spark MLlib library is based on the numpy library, so install it:

$ yum install numpy

HDP 2.2.4 comes with Spark 1.2 we need to upgrade to version 1.3.1 to do that:

Download the latest RPM repo that has Spark 1.3.1

$ wget -nv http://public-repo-1.hortonworks.com/HDP/centos6/2.x/updates/2.2.4.4/hdp.repo -O /etc/yum.repos.d/HDP-TP.repo

Install Spark Package

$ yum install spark\_2\_2\_4\_4\_16-master

This will download Spark 1.3.1 RPM and setup on your HDP 2.2 cluster. As part of this RPM install it will also download necessary core Hadoop dependencies.

If you want to use PySpark you also need to run

$ yum install spark-python

Use HDP-Select to point to this Spark package

$ hdp-select set spark-historyserver 2.2.4.4-16

$ hdp-select set spark-client 2.2.4.4-16

This will set the /usr/hdp/current/spark-client & /usr/hdp/current/spark-historyserver dirs to point to the version selected with hdp-select.

# Exercise 1: WordCount MapReduce

Upload the “El Quixote” book dataset to HDFS.

$ hadoop fs –put datasets/quixote.txt /user/spark

Use HUE to explore it: In web browser type 192.168.164.134:8000/ and go to file browser

Run the *wordcount* program and check the output:

$ hadoop jar exercise1/hadoop-mapreduce-examples-2.5.0-cdh5.3.0.jar wordcount /user/spark/quixote.txt /user/spark/quixote-wordcount

$ hadoop fs –ls /user/spark/quixote-wordcount

$ hadoop fs –cat /user/spark/Quixote-wordcount/part-r-00000

There are other programs also: 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.cache

scala> rdd.take(10)

scala> rdd.take(10).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

Change the user to hdfs (hadoop super user) and Start the Spark shell

$ su hdfs

$ spark-shell

Execute wordcount and check the output:

scala> val data = sc.textFile("hdfs://sandbox.hortonworks.com/user/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(“hdfs://sandbox.hortonworks.com/user/spark/quixote-wordcount-spark”)

# Exercise 4: Top-10 WordCount

Get 10 most used words in “The Quigfte” 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("hdfs://sandbox.hortonworks.com/user/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 []

| hdfs://sandbox.hortonworks.com/user/spark/quixote.txt MappedRDD[1] at textFile at <console>:12 []

| hdfs://sandbox.hortonworks.com/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: 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 6: Data Mining with MovieLens Dataset

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

There are two datasets:

* **movilens-1M** which contains 1M of ratings (for fast testing purposes)
* **movilens-10M** which contains 10M of ratings

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:

[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

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:

[root@sandbox datasets]# head -10 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 check 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:///root/ Gft-Spark-Exercices/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:///root/Gft-Spark-Exercices/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 7: 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("hdfs://sandbox.hortonworks.com/user/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 8: Data Mining Using SparkSQL

Do the same as in exercise 6 but this time using SparkSQL.

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:///root/Gft-Spark-Exercices/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:///root/Gft-Spark-Exercices/datasets/movielens-1M/ratings.dat](file:///\\root\Spark_Training_2015_exercices\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 9: 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 / check 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 := "2.10.5"

//add this code

libraryDependencies ++= Seq(

"org.apache.spark" %% "spark-core" % "1.4.0",

"org.apache.spark" %% "spark-mllib" % "1.4.0",

"org.apache.spark" %% "spark-sql" % "1.4.0",

"com.databricks" %% "spark-csv" % "1.2.0"

)

1. Wait for the project will be created / At the level **[sparktraining/sr/main/scala]** create **package com.gft.sparktraining** / and a new scala class (like an object) called **TestSpark** (suffix .scala is added for you)
2. 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)

}

}

1. 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
2. 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

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

-Dspark.master=local

1. Repeat step 6

# Exercise 10: 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 11: 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 12: 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\_0.0.5"

"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 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.{RDDComparisions, 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 ===

RDDComparisions.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 13: 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 14: Recommender Algorithms

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

There are two datasets:

* **movilens-1M** which contains 1M of ratings (for fast testing purposes)
* **movilens-10M** which contains 10M of ratings

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** -**mínimos cuadrados alternados**-(**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

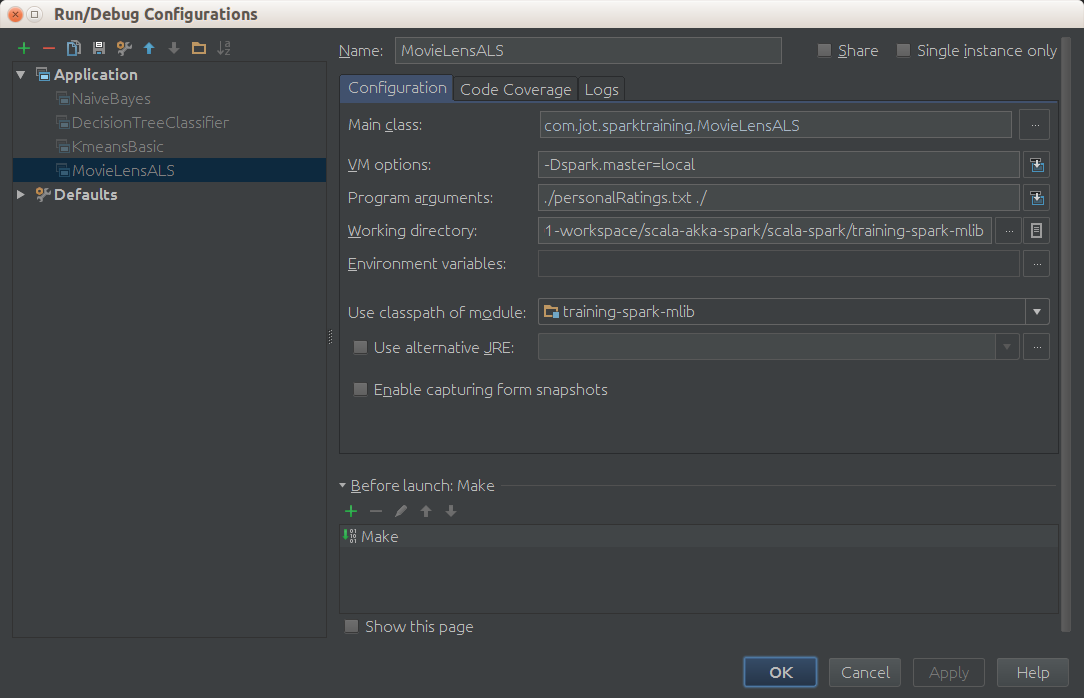
val numUsers = ratings.values.map(r => r.user).distinct().count

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** (**Root Mean Squared Error, -Raíz del error cuadrático medio-**), 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.

…

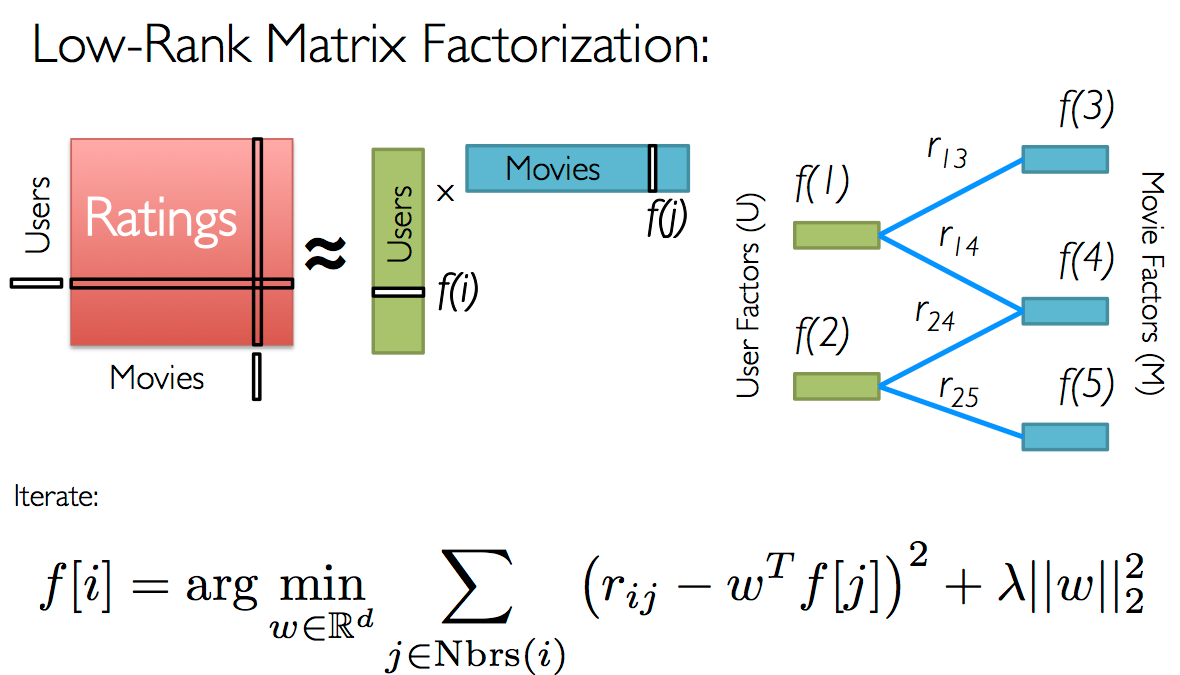
Training: 602251, Validation: 198919, numTest: 199049

…

## Training using ALS

In this section, we will use **ALS.train[[1]](#footnote-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.



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) {

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]](#footnote-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)

.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)

# Exercise 15: Deploy program in a Spark Cluster

Once solved the exercise lets deploy in a Hortonworks VM Sandbox cluster. (If previous exercise is not done or not finished don’t worry solution is in Gft-Spark-Exercices\exercise14 directory.

## Step 1. Creating a Jar from SBT

From IntelliJ open file **build.sbt** and add the following commented lines in order to create a fully auto executable **jar** that we can deploy in spark cluster. Note that spark dependencies are declared as provided:

name := "MovieLensALS"

version := "1.0"

scalaVersion := "2.10.5"

libraryDependencies ++= Seq(

// Spark dependency

"org.apache.spark" %% "spark-core" % "1.3.1" % "provided",

"org.apache.spark" %% "spark-mllib" % "1.3.1" % "provided"

)

// Configure JAR used with the assembly plug-in

assemblyJarName in assembly := "test-spark.jar"

// A special option to exclude Scala itself form our assembly JAR, since Spark

// already bundles Scala.

assemblyOption in assembly :=

(assemblyOption in assembly).value.copy(includeScala = false)

//Class with main function that we want execute

mainClass in assembly := Some("com.gft.sparktraining.MovieLensALS")

Open a Terminal in the same directory where build.sbt is and with sbt installed in its last version ([www.scala-sbt.org](http://www.scala-sbt.org/)) type **sbt** to enter in interactive mode

**Note!!!** :if you make changes to build.sbt file you'll need to type **reload** in order to consider the new changes.

After that type the following commands, **clean** for cleaning target directory, and **assembly** for making jar

May be occurs that assembly command will not be recognized , if that go to <HOME\_DIR>/.sbt/0.13/plugins and create or edit plugins.sbt and add (with extra lines):

addSbtPlugin (“com.eed3si9n” % “sbt-assembly” % “0.13.0”)

(0.13 version can be different, take the version installed in your computer)

$ sbt

[info] Loading global plugins from /home/chicochica10/.sbt/0.13/plugins

[info] Loading project definition from /home/chicochica10/01-workspace/scala-akka-spark/scala-spark/test-assembly/project

[info] Set current project to test-assembly (in build file:/home/chicochica10/01-workspace/scala-akka-spark/scala-spark/test-assembly/)

> **reload**

[info] Loading global plugins from /home/chicochica10/.sbt/0.13/plugins

[info] Loading project definition from /home/chicochica10/01-workspace/scala-akka-spark/scala-spark/test-assembly/project

[info] Set current project to test-assembly (in build file:/home/chicochica10/01-workspace/scala-akka-spark/scala-spark/test-assembly/)

> **clean**

[success] Total time: 0 s, completed 11-jun-2015 22:49:28

> **assembly**

[info] Updating {file:/home/chicochica10/01-workspace/scala-akka-spark/scala-spark/test-assembly/}test-assembly...

[info] Resolving com.sun.jersey.jersey-test-framework#jersey-test-framework-griz[info] Resolving com.fasterxml.jackson.module#jackson-module-scala\_2.10;2.4.4 ..[info] Resolving org.fusesource.jansi#jansi;1.4 ...

[info] Done updating.

[info] Compiling 7 Scala sources to /home/chicochica10/01-workspace/scala-akka-spark/scala-spark/test-assembly/target/scala-2.10/classes...

[warn] Multiple main classes detected. Run 'show discoveredMainClasses' to see the list

[info] Checking every \*.class/\*.jar file's SHA-1.

[info] Merging files...

[info] SHA-1: 4ede9a7a9acf5ec62411ad481d900329a330425b

[info] Packaging /home/chicochica10/01-workspace/scala-akka-spark/scala-spark/test-assembly/target/scala-2.10/test-spark.jar ...

[info] Done packaging.

[success] Total time: 12 s, completed 11-jun-2015 22:50:12

## Step 2. Uploading jar and data to VM and HDF

Now you are ready to upload to de VM sandbox the new created jar: **target/scala-2.10/test-spark.jar**. For doing that you can use winscp, filezilla or in linux use the command:

$ scp target/scala-2.10/test-spark.jar root@192.168.164.134:/root

you also will need upload the datasets to HDFS for doing that we are going to launch “**ambari server**” which is a visual hadoop manager shipped with HDP needed later for reviewing **YARN**.

$ **ssh root@192.168.164.134**

root@192.168.164.134's password:

Last login: Thu Jun 11 20:58:21 2015

# ./start\_ambari.sh

Starting Ambari...

/bin/sh: /etc/init.d/hdp-gmetad: No such file or directory

Starting Ganglia [ OK ]

Starting Nagios [WARNINGS]

/bin/sh: /etc/init.d/nagios: No such file or directory

Starting Ambari server [ OK ]

Starting Ambari agent [ OK ]

====================================

Ambari autostart enabled

To disable auto-start of Ambari do

# chkconfig ambari off

====================================

in your local **etc/hosts** file (or the hosts file in Windows) add the line:

$ 192.168.164.134 sandbox.hortonworks.com sandbox ambari.hortonworks.com

In your browser open: the address <http://sandbox.hortonworks.com:8080/#/login> user: admin password: admin

Press the and select Files, use upload button for uploading to hdfs /tmp movielens 1M datasets movies and ratings

There is a bug in some uploader versions and may be your files are uploaded with mistakes, if you get exceptions after launching the spark job like Integer exceptions or limits out of bounds, delete the files in hdsf and upload them from VM :

$ cd Gft-Spark-Exercices/datasets/movielens-1M/

$ hadoop fs –put ratings.dat /tmp/ratings.dat

$ hadoop fs –put movies.dat /tmp/movies.dat

## Step 3. Configure VM for launching the jar

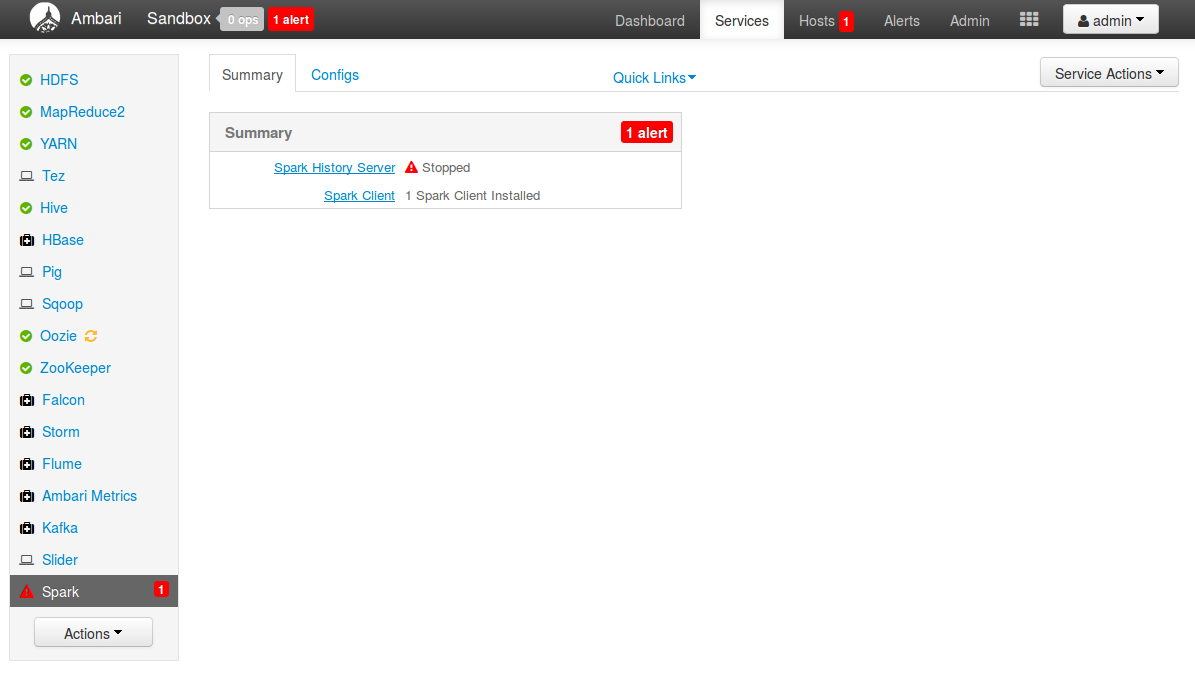
In VM let's set 2 environment variables, official documentation only requires the first one <http://hortonworks.com/hadoop-tutorial/using-apache-spark-hdp/> but is a good idea set also the other (Learning Spark Book, Chapter 7: Running on a Cluster).

$ export YARN\_CONF\_DIR=/etc/hadoop/conf

$ export HADOOP\_CONF\_DIR=/etc/hadoop/conf

Now it's time to start Spark History Server from Ambari Server:

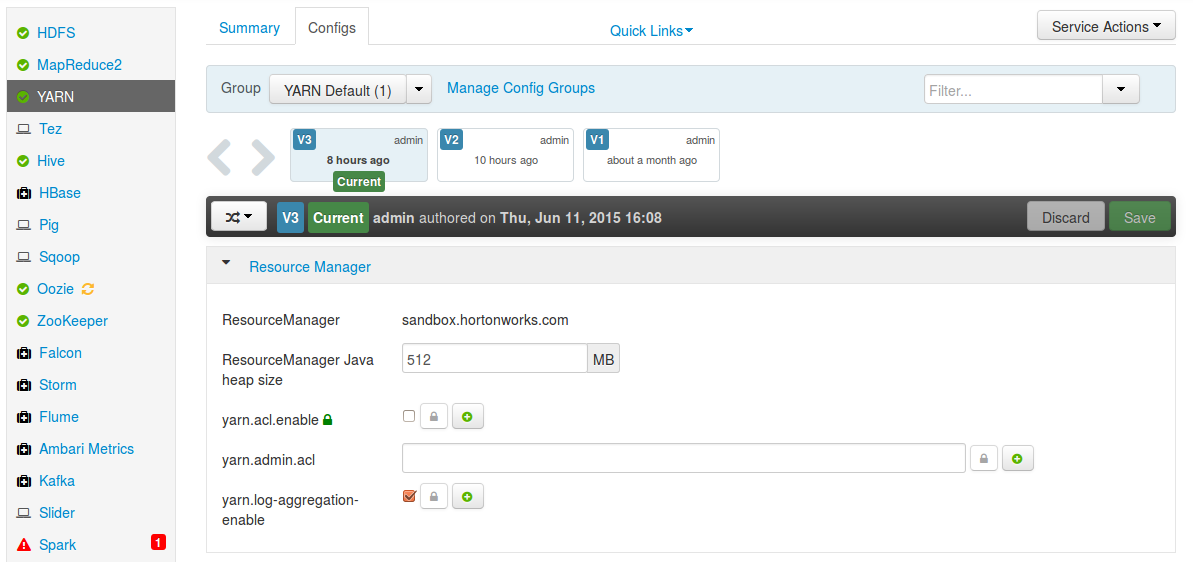
Choose from the left List Spark Service / Click on the right Summary Spark History Server

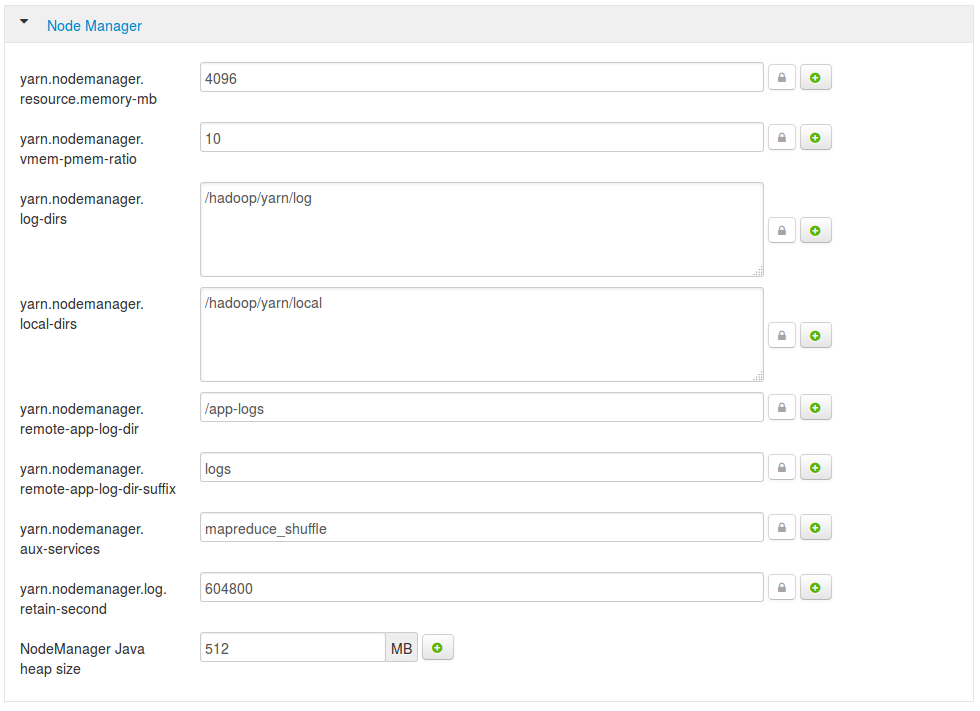


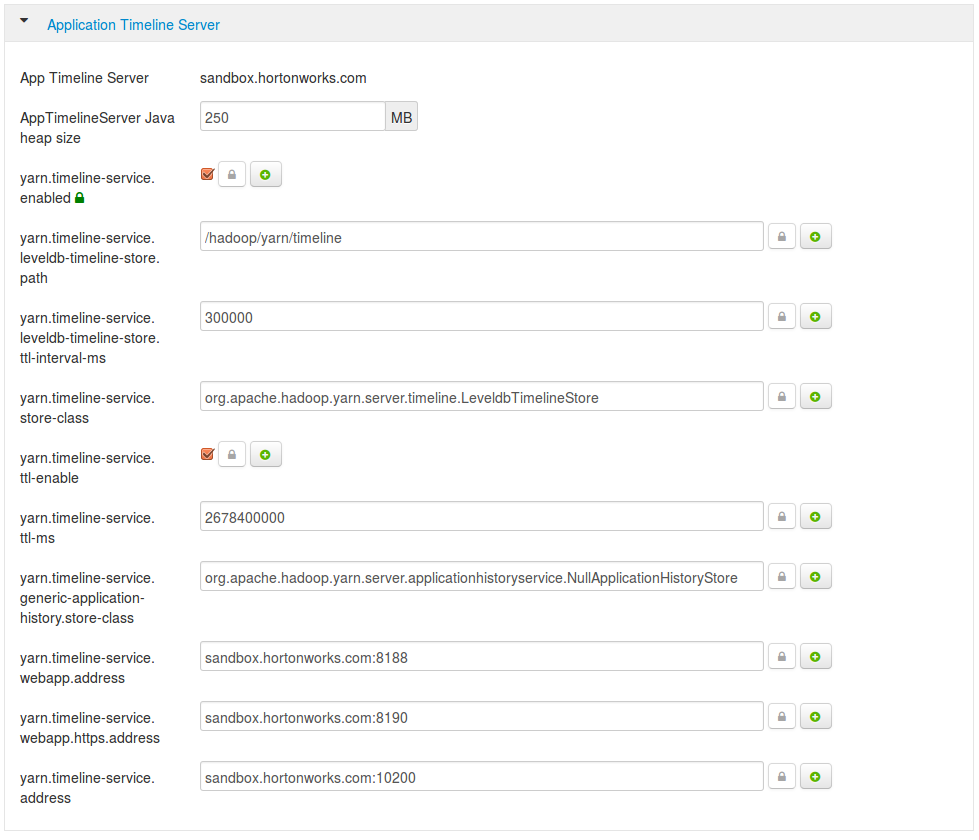


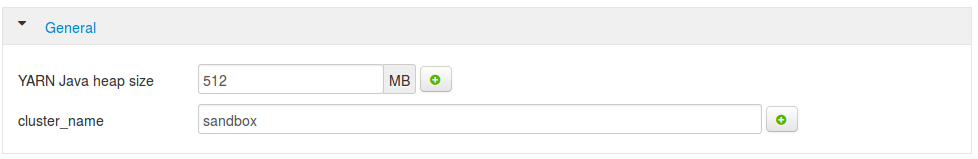
And from the list of the left choose **Start** Spark History Server:

Now let's review YARN configuration: From the Ambari main menu, choose **YARN** from the left list of services and click over **Configs** label, check the following configurations (Pay special attention to scheduler **yarn.scheduler.minimun-allocation-mb and yar.scheduler.maximum-allocation-mb**)

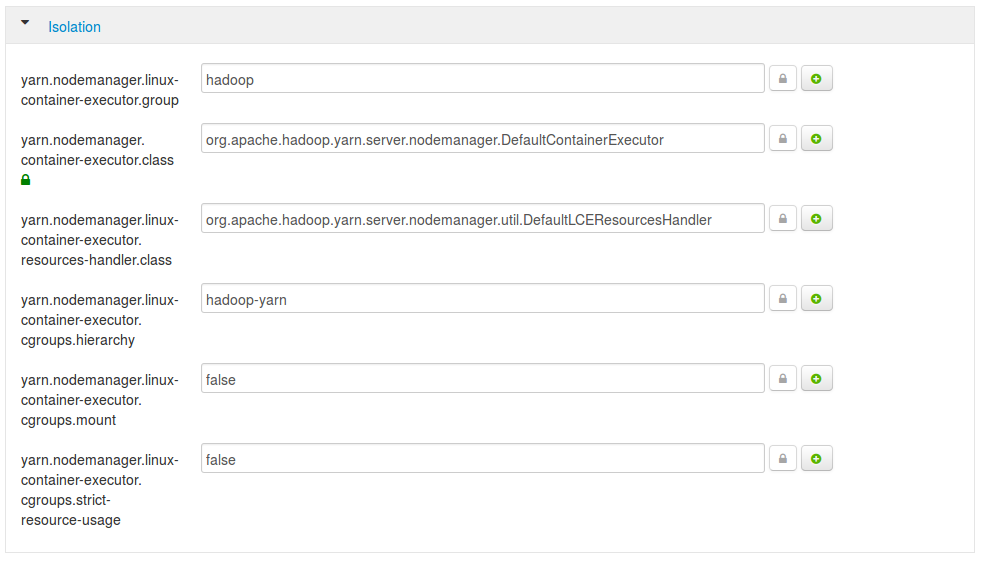


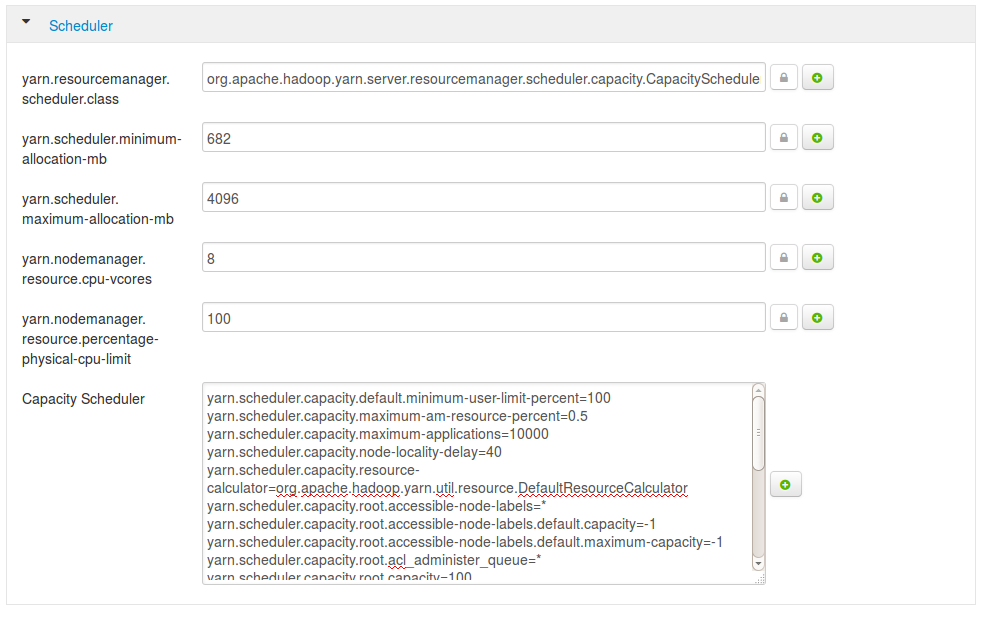












Not important or not changed:



## Step 4. Launching spark-submit

Depending of the size of your VM (memory and cores) you can increase some of the next parameter used in the spark-submit command inside of VM sandbox:

# cd

# spark-submit --verbose --master yarn test-spark.jar ./Gft-Spark-Exercices/datasets/personalRatings.txt hdfs://sandbox.hortonworks.com:8020/tmp/ --executor-memory 1G --driver-memory 1G --total-executor-cores 2

**Explanation**: Due we are using a YARN cluster to synchronize spark internal process we use option **yarn** as master (the url was set up using environment variables).The jar file was create as auto executable with a main class so is not necessary add the main class as parameter, we need to add a two path as arguments, the first is local personalRatings.txt (small file that will be parallelized) and the second is an hdfs path to movies.dat and ratings.dat because we are in a Hadoop cluster. The rest of the parameters are memory assigned to executors daemons and driver and the total cores that will run the job (depending of your cluster).

For more commands visit Learning Spark, Chapter 7: Running on a Cluster.

If you want the process gracefully stops after computation remember add: **sc.stop()** in your code.

Useful links:

YARN Resource manager web: <http://sandbox.hortonworks.com:8088/cluster>

Spark UI: <http://sandbox.hortonworks.com:4040/> (while in execution)

1. [https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.recommendation.ALS$](https://spark.apache.org/docs/1.2.0/api/scala/index.html#org.apache.spark.mllib.recommendation.ALS$) [↑](#footnote-ref-1)
2. <https://spark.apache.org/docs/1.2.0/api/scala/index.html#org.apache.spark.mllib.recommendation.MatrixFactorizationModel> [↑](#footnote-ref-2)