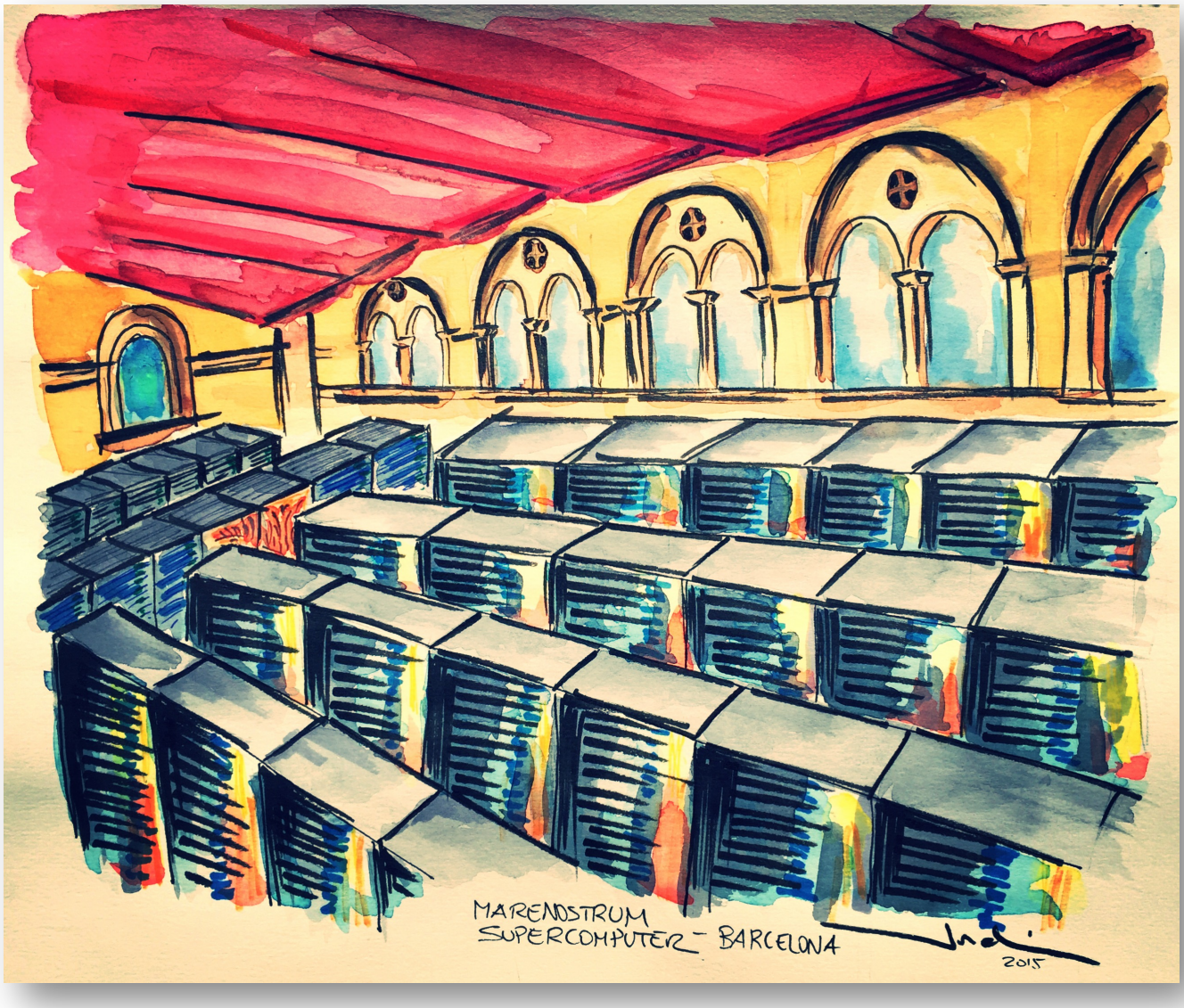
Hands-on 9:

Getting Started with Apache Spark



SUPERCOMPUTERS ARCHITECTURE

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS

Barcelona, Fall 2015

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hands-on 9

Getting Started with Apache Spark

SUPERCOMPUTERS ARCHITECTURE
Facultat d'Informàtica de Barcelona (FIB)
UPC Barcelona Tech & Barcelona Supercomputing Center

Version 1.0 - 28 February 2015

Version 2.0 – 10 November 2015

Professor: Jordi Torres

Contributors: Mauro Gómez, Rubèn Tous, Dani Cea and Mario Macías.

Table of Contents

1 Hands-on content

This hands-on is intended to help the student to get started with Apache Spark, a powerful open source framework to process big data with high speed and easy to use.

In this hands-on we use Scala sbt v0.13.7, Python 2.6, and Spark v1.3.0). All of the examples are explained in both programming languages.

For SA-MIRI students it is important to practice Scala in order to do the next hands-on in the Marenostrum.

Enjoy!

2 Run a Linux OS in a Virtual Machine (optional)

The goal of this hands-on is to learn and experiment without fear of breaking anything in your main operating system of your computer. The way is use virtualization that allows us to run Linux directly atop our primary OS. We suggest to use VirtualBox in order to run a Linux in a separate virtual machine.

As installation of VirtualBox varies depending on your host operating system, we provide installation instructions for MAC OS X host. In VirtualBox Manual you can find detailed information (fairly easy to set up) for Windows, Linux or Solaris.

For Mac OS X hosts, you can Download VirtualBox (free) from its downloads website. VirtualBox ships in a disk image (dmg) file. As usual, perform the following steps:

1. Double-click on that file to have its contents mounted.
2. A window will open telling you to double click on the VirtualBox.mpkg installer file displayed in that window.
3. This will start the installer, which will allow you to select where to install VirtualBox to.

After installation, you can find a VirtualBox icon in the “Applications” folder in the Finder.

Ubuntu is a Debian-based Linux operating system, with Unity as its default desktop environment. It is based on free software. Download a 64-bit Ubuntu Linux ISO (free). Once VirtualBox is installed and the Ubuntu ISO is downloaded, you can proceed with set up and installation:

1. Launch VirtualBox and create a new virtual machine (“New” button) , name it and set the Operating System to “Linux” and version to “Ubuntu (64 bits)” and set the base memory to at least 2048MB.

2. Create a new “virtual hard disk now”. Click “Continue” and choose VDI as the disk type. Set the storage details to be “Dynamically allocated” to preserve some initial disk space. Name the virtual disk something obvious or use the default suggested name and make it at least 8GB and click “Create” to produce the virtual hard drive.

3. At the VirtualBox Manager screen, select the newly created virtual machine and click on the “Settings” button.
4. Click on the “Storage” tab and next to “Controller: IDE” click the + icon that looks like a CD to add a new IDE Controller. Click “Choose Disk” and locate the Ubuntu ISO (named something like ” ubuntu-11.10-desktop-amd64.iso”) that you downloaded earlier, then click “OK” and close out of Settings.

Back at the VirtualBox Manager screen, select the Linux virtual machine and click on “Start” to begin booting the VM. Let the VM boot and choose “Install Ubuntu” at the welcome screen, create a login and user name, set time zones, and click through the simple installation process until it begins. When finished, Ubuntu Linux will boot into the desktop.

3 Introduction to Spark

Spark is an open source data analytics cluster computing framework and one of the faster complex event processing engines for large-scale data processing which runs very fast on-memory.

Spark was written in Scala and currently support four programming apis: Scala, Java, R and Python.

Spark is a fast and general processing engine compatible with Hadoop data. It can run in Hadoop clusters through YARN or Spark's standalone mode, and it can process data in HDFS, HBase, Cassandra, Hive, and any Hadoop InputFormat. It's designed to perform batch processing, streaming and interactive queries.

From architecture perspective Apache Spark is based on two key concepts; Resilient Distributed Datasets (RDD) and directed acyclic graph (DAG) execution engine. Latest versions include the dataframe data type that plays an important role. For a comprehensive introduction text we suggest to read the book "INTRODUCCIÓN A APACHE SPARK" (www.SparkBarcelona.es).

4 Setup Tools

First of all, we need to download the Spark environment. To do that, we can go to https://spark.apache.org/ and download the files. In all of this hands-on we will work with Spark v1.3.0. Important: you have to download one of pre-built version in "Choose a package type" section (for instance we tested this hands-on with spark-1.3.0-bin-hadoop2.4).

Once we have the tarball file, we need to uncompressing it and navigate to the bin folder. It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python.

In order to show only errors in the shell, and omit all the warnings and info messages, we need to edit the file located in $SPARK_DIR/conf/log4j.properties and replace the below line:

```
log4j.rootCategory=INFO, console
```

to

```
log4j.rootCategory=ERROR, console
```

If you use a public ip address (e.g. labs at FIB), probably Spark not works well. To solve this error, you need to edit the file conf/spark-env.sh and set up the SPARK_LOCAL_IP variable to LOCALHOST

Now, we can execute the shell:

```
## Python
$./bin/pyspark

## Scala
$./bin/spark-shell
```

Both shells provide a Spark Context available in **sc** variable, which provides all of the Spark Context functions (described in theory class). Try to execute this simple WordCount to check that everything is working fine. The wc_out directory will contain the result Be sure that wc_out directory doesn't exist before execute this simple WordCount.

```
## Python

>>>from operator import add
>>>f = sc.textFile("README.md")
>>>wc = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x,
1)).reduceByKey(add)

>>>wc.saveAsTextFile("wc_out")

## Scala

scala> val file = sc.textFile(“README.md”)
scala> val counts = file.flatMap( line => line.split(“
“)).map(word => (word, 1)).reduceByKey(_ + _)

scala> counts.saveAsTextFile("wc_out")
```

Inspect wc_out directory.

5 Simple clustering example

In order to warm-up, we will start with a simple clustering example. Clustering is an unsupervised learning problem whereby we aim to group subsets of entities with one another based on some notion of similarity. Clustering is often used for exploratory analysis and/or as a component of a hierarchical supervised learning pipeline (in which distinct classifiers or regression models are trained for each cluster).

5.1 MLlib

Machine learning has quickly emerged as a critical piece in mining Big Data for actionable insights. Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and blazing speed. The library is usable in Java, Scala, and Python as part of Spark applications, so that you can include it in complete workflows.

5.2 K-Means

MLlib supports k-means clustering, one of the most commonly used clustering algorithms that clusters the data points into predefined number of clusters. The MLlib implementation includes a parallelized variant of the k-means++ method called kmeans||. The implementation in MLlib has the following parameters:

- *k* is the number of desired clusters.
- *maxIterations* is the maximum number of iterations to run.
- *initializationMode* specifies either random initialization or initialization via k-means||.
- *runs* is the number of times to run the k-means algorithm (k-means is not guaranteed to find a globally optimal solution, and when run multiple times on a given dataset, the algorithm returns the best clustering result).
- *initializationSteps* determines the number of steps in the k-means|| algorithm.
- *epsilon* determines the distance threshold within which we consider k-means to have converged.

In the following example, after loading and parsing data, we use the K-Means object to cluster the data into five clusters. The number of desired clusters is passed to the algorithm. We then compute Within Set Sum of Squared Error (WSSSE). You can reduce this error measure by increasing de parameter k.

5.3 Loading data as a RDD

Create a file `example1.txt` with the following content:

```
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

Now we can load the data in a RDD with name `data`

```
## Python
>>>data = sc.textFile("example1.txt")

## Scala
>>>val data = sc.textFile(“example1.txt”)
```

We suggest to check the content before to continue.

```
## Python
>>>def show (x): print x
>>>data.foreach(show)

## Scala
scala> data.foreach(println)
```

5.4 Convert text to float

Now we suggest to parse the data and convert it to float and push into array.

```
## Python
>>>from numpy import array
>>>parsedData = data.map(lambda line: array([float(x) for x
in line.split(' ')])).cache()
>>>parsedData.foreach(show)
```

```
## Scala
scala> import org.apache.spark.mllib.linalg.Vectors
scala> val parsedData = data.map(s =>
Vectors.dense(s.split(“ “).map(_.toDouble))).cache()
scala> parsedData.foreach(println)
```

At this point we have an array with the parsed data as:

```
0.0 0.0 0.0                    [0.0,0.0,0.0]
0.1 0.1 0.1                    [9.1,9.1,9.1]
0.2 0.2 0.2                    [0.1,0.1,0.1]
9.0 9.0 9.0                    [0.2,0.2,0.2]
9.1 9.1 9.1                    [9.2,9.2,9.2]
9.2 9.2 9.2                    [9.0,9.0,9.0]
```

5.5 Train the algorithm

With this data, we will train the k-means algorithm and compute the cost. To do that, we need to import some libraries like KMeans and sqrt. We use the KMeans object to cluster the data into two clusters. The number of desired clusters is passed to the algorithm.

We then compute `Within Set Sum of Squared Error` (WSSSE). The standard KMeans algorithm aims at minimizing the sum of squares of the distance between the points of each set: the squared Euclidean distance . . Once you have computed the “`val result = KMeans.train(<trainData>, <clusterNumber>, <Iterations>)`” you can evaluate the result by using WSSSE (something like the sum of the distances of each observation in each K partition). You can reduce this error measure by increasing k.

Python

```
>>>from pyspark.mllib.clustering import KMeans
>>>from numpy import array
>>>from math import sqrt

>>># Build the model (cluster the data)
>>>clusters = KMeans.train(parsedData, 2, maxIterations=10,
   runs=10, initializationMode="random")
>>># Evaluate clustering by computing Within Set Sum of
   Squared  Errors
>>>def error(point):
    center = clusters.centers[clusters.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

>>>WSSSE = parsedData.map(lambda point:
error(point)).reduce(lambda x, y: x + y)
>>>print("Within Set Sum of Squared Error = " + str(WSSSE))
```

Scala

```
scala> import org.apache.spark.mllib.clustering.KMeans

scala> // Build the model (cluster the data)
scala> val clusters = KMeans.train(parsedData, 5, 10)

scala> # Evaluate clustering by computing Within
scala> # Set Sum of Squared  Errors
scala> val WSSSE = clusters.computeCost(parsedData)
scala> print("Within Set Sum of Squared Error = " + WSSSE)
```

6 TF with plain text

TF is the acronym to Term Frequency is a feature vectorization method widely used in text mining to count. TF is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus.

If you want to know more about TF algorithm, the following link explain more about the implementation in Spark and how the algorithm works. https://spark.apache.org/docs/latest/mllib-feature-extraction.html#tf-idf

The TF it's implemented in HashingTF Spark library and the method take an RDD of list as the input, and return an SparseVector with the frequencies using the hashing tricks.

6.1 Plain text example

Assume that we have a set of tagged pictures from Instagram that must be grouped in two groups:

Picture 1: #messi #barcelona #madrid #instagram #barcelona

Picture 2: #barcelona #paella #playa #sangria #instagram

Picture 3: #paella #barcelona #instagram

Picture 4: #ramblas #paella #ramblas #instagram

Picture 5: #messi #madrid #instagram

Picture 6: #barcelona #messi #instagram

Picture 7: #messi #gol #barcelona #instagram

Picture 8: #sangria #paella #instagram

First create a file, `example2.txt`, with the following content:

```
messi barcelona madrid instagram barcelona
barcelona paella playa sangria instagram
paella barcelona instagram
ramblas paella ramblas instagram
messi madrid instagram
barcelona messi instagram
messi gol barcelona instagram
sangria paella instagram
```

6.2 Data preprocessing

The first step is preprocessing the data in order to obtain a vector of words for each line.

Python

```
>>>documents = sc.textFile("example2.txt").map(lambda line:
>>>line.split(" "))
>>>documents.foreach(show)

...

[u'messi',u'barcelona',u'madrid',u'instagram',u'barcelona']
...
```

Scala

```
scala>val documents=
sc.textFile("example2.txt").map(_.split(“”).toSeq)

scala> documents.foreach(println)

...

WrapedArray(messi, barcelona, madrid, instagram, barcelona)
...
```

6.3 Text vectorization

Now we have the file stored in a variable RDD vectors. We need to create a HashingTF instance to transform the data in order to find the frequency in vector space.

Python

```
>>>from pyspark.mllib.feature import HashingTF
>>>hashingTF = HashingTF()
>>>tf = hashingTF.transform(documents)
>>>def show (x): print x
>>>tf.foreach(show)

...

(1048576,[388745,503816,618478,929925],[1.0,1.0,2.0,1.0])
...
```

```
## Scala

scala> import org.apache.spark.rdd.RDD
scala> import org.apache.spark.mllib.feature.HashingTF
scala> val hashingTF = new HashingTF()
scala> val tf = hashingTF.transform(documents)
scala> tf.foreach(println)

...

(1048576,[388745,503816,618478,929925],[1.0,1.0,2.0,1.0])
...
```

At this point, we have the tf variable that contains the term frequencies of the document (e.g. barcelona represented by the value 618478 with frequency 2, or Instagram represented by 929925 with frequency 1).

6.4 Clustering (kmeans)

Now, we have a vectors with the frequency of each word. We could create two clusters with the KMeans algorithm with the frequencies:

```
## Python

>>>from pyspark.mllib.clustering import KMeans
>>>clusters = KMeans.train(tf, 2, 1, 1)
>>>results = documents.map(lambda x : array([x,
>>>clusters.predict(hashingTF.transform(x))]))
>>>results.foreach(show)


...
[[u'barcelona', u'messi', u'instagram'] 0]
[[u'barcelona', u'paella', u'playa', u'sangria', u'instagram'] 1]
...
```

Scala

```
scala> import org.apache.spark.mllib.clustering.KMeans
scala> val clusters = KMeans.train(tf, 2, 1)
scala> val results = documents.map(x => Vector(x,
clusters.predict(hashingTF.transform(x))))
scala> results.collect.foreach(println)


...
Vector(WrappedArray(barcelona,messi,instagram),  0)
Vector(WrappedArray(barcelona,paella,playa,sangria,instagram] 1]
...
```

The last component of the output vector indicates the group of each picture (be a membership of the first group is indicated by 0). In this case we can imagine that one group are related with futbol and the second one related with tourism in Barcelona.

7 Exercise: Classify your own dataset

Find a dataset that contains information about Barcelona (or your home town) and classify it repeating the previous steps presented in this hands-on.

You have one week to deliver a report with the results, including a description of the input, the code and the output.