K-means in Apache Spark

Mauro Sozio
Oana Balalau (TA), Maximilien Danisch (TA)
J.B. Griesner (TA), Raphael Charbey (TA)
name.lastname@telecom-paristech.fr

November 8, 2016

In this lab we will use the framework Apache Spark and its machine learning module, MLlib. Spark is used for large-scale data processing and allows the users to easily execute algorithms on clusters of machines. The framework will partition the data across the cluster and use the computational power and main memory of the nodes to perform parallel computation.

For more details about the concepts behind Apache Spark, you can read the original research article:
`http://www-bcf.usc.edu/~minlanyu/teach/csci599-fall12/papers/nsdi_spark.pdf`.

Our task

As in the previous lab, our purpose is to implement a k-means algorithm to cluster the collection of documents stored in text.txt (one document per line):

- use TF-IDF to turn the collection of documents into vectors;
- consider all values in [1, 10] for k. Plot the SSE as a function of k and choose the optimal value for k.

Using Databricks

Create a community edition account at `https://databricks.com/` to use Spark with Python (or Java or Scala). You will be able to use only one machine for free on Databricks. However, the same code can be run on several machines without changes. After creating a Databricks account, you need to create a cluster using the tab *Clusters* → *Create Cluster*. After the creation you will see the cluster in the Active Cluster. The next step consists in loading the dataset *text.txt*, tab *Tables* → *Create Table*, and remember the path returned, it should be in the format : `/FileStore/tables/path/text.txt`. For the following steps create a notebook, tab *databricks* → *New* → *Notebook*.

Preprocessing the data

```
from pyspark.sql import Row

docs = sc.textFile("/FileStore/tables/path/text.txt")
row = Row("text")
df = docs.map(row).toDF()
```

This code will load the text into main memory in the form of a distributed collection of data (dataframe), organized as a table with named columns. In the example we have just one column named *text*.

After loading the data into main memory, we need to tokenize the words and remove the stopwords:

```
from pyspark.ml.feature import Tokenizer
from pyspark.ml.feature import StopWordsRemover

tokenizer = Tokenizer(inputCol="text", outputCol="words")
wordsData = tokenizer.transform(df)
remover = StopWordsRemover(inputCol="words", outputCol="filtered")
filteredData = remover.transform(wordsData)
```

We note that to create a Tokenizer/StopWordsRemover we need to give an input and output column, that is the column on which we will run the tokenizing and removal of stopwords and the column which will store the result. The method transform is the one that triggers the computation.

TF-IDF

After tokenizing and filtering the stopwords, we compute the importance of a word for a document, more precisely the TF-IDF.

```
from pyspark.ml.feature import HashingTF, IDF
from pyspark.mllib.linalg import DenseVector

hashingTF = HashingTF(inputCol="filtered", outputCol="rawFeatures", numFeatures
    =500)
featurizedData = hashingTF.transform(filteredData)
idfModel = IDF(inputCol="rawFeatures", outputCol="features")
tfidf = idfModel.fit(featurizedData).transform(featurizedData).select("features
    ").rdd.map(lambda r : r[0]).map(lambda x : DenseVector(x.toArray()))
```

K-means

We can then proceed to train the classifier:

```
from math import sqrt

clusters = []
for k in range(1,11):
  clustersk = KMeans.train(tfidf, k)
  clusters.append(clustersk)

# error function for a given point and given K clusters models
```

```
def error(point, k):
    center = clusters[k].centers[clusters[k].predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

# Evaluate clustering by computing Within Set Sum of Squared Errors
wssses = []
for k in range(0,10):
  WSSSE = tfidf.map(lambda point: error(point, k)).reduce(lambda x, y: x + y)
  wssses.append(1.0 / WSSSE)
  print("Within Set Sum of Squared Error = " + str(WSSSE) + ", for k = " + str(k
      ))
```

At the end, we will obtain the sum of squared errors for values of $k \in [1, 15]$. The initialization of K-means is not random, but with selected seeds (K-means++) and the efficient parallel implementation is described in this research article:

`http://vldb.org/pvldb/vol5/p622_bahmanbahmani_vldb2012.pdf`

Visualizing the results

In order to better understand the results we can use the command display.

```
# Plot the errors for different values of K
row = Row("errs")
err = sc.parallelize(wssses).map(row).toDF()
err.registerTempTable("err")
display(sqlContext.sql("select * from err"))
```

This code will produce a table which can be visualized using bars, lines or other visualizing modes. To have access to this, check the three buttons at the end of the output table.

We can also visualize a K-mean model:

```
row = Row("features")
data = tfidf.map(row).toDF()
clusters5 = KMeans.train(tfidf, 5)
display(clusters5,data)
```

For a better visualization, reduce the number of features to 10 or 5.

Databricks resources

Databricks offers a training session in order to get familiar with Spark. It is accessible via *Workspace* $\rightarrow$ *Training*.