# K-means in spark

# +

# PROJECT description

Thanks to Paul-Henri PERRIN, PhD student at Data Science team @ Dauphine

# Basic notions

- Borrowed from Francis Back's nice notes at ENS ( https://www.di.ens.fr/~fbach/courses/fall2013/lecture3.pdf )

$K$- means clustering is a method of vector quantization. $K$-means clustering is an algorithm of alternate minimization that aims at partitioning $n$ observations into $K$ clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype to the cluster (see Figure 3.1).

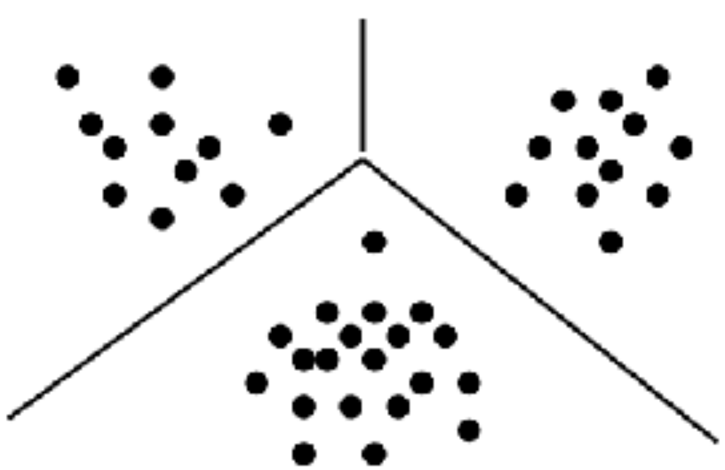

**Figure 3.1.** Clustering on a 2D point data set with 3 clusters.

# Notations

We will use the following notations:

- $x_i \in \mathbb{R}^p$, $i \in \{1, ..., n\}$ are the observations we want to partition.
- $\mu_k \in \mathbb{R}^p$, $k \in \{1, ..., K\}$ are the means where $\mu_k$ is the center of the cluster $k$. We will denote $\mu$ the associated matrix.
- $z_i^k$ are indicator variables associated to $x_i$ such that $z_i^k = 1$ if $x_i$ belongs to the cluster $k$, $z_i^k = 0$ otherwise. $z$ is the matrix which components are equal to $z_i^k$.

Finally, we define the *distortion* $J(\mu, z)$ by:

$$J(\mu, z) = \sum_{i=1}^{n} \sum_{k=1}^{K} z_i^k \|x_i - \mu_k\|^2.$$

# The K-means algorithm

The aim of the algorithm is to minimize $J(\mu, z)$. To do so we proceed with an alternating minimization :

- Step 0 : We choose a vector $\mu$
- Step 1 : we minimize $J$ with respect to $z$ : $z_i^k = 1$ if $\| x_i - \mu_k \|^2 = \min_s \| x_i - \mu_s \|^2$, in other words we associate to $x_i$ the nearest center $\mu_k$.
- Step 2 : we minimize $J$ with respect to $\mu$ : $\mu_k = \frac{\sum_i z_i^k x_i}{\sum_i z_i^k}$.
- Step 3 : we come back to step 1 until convergence.

# Data

- We will use a simple (classical) data set describing features of flowers (available at https://www.dropbox.com/s/9kits2euwawcsj0/iris.data.txt)

## Le jeu de données [ modifier | modifier le code ]

**Fisher's *Iris* Data**

| **longueur des sépales (en cm)** *(Sepal length)* | **largeur des sépales (en cm)** *(Sepal width)* | **longueur des pétales (en cm)** *(Petal length)* | **largeur des pétales (en cm)** *(Petal width)* | **Espèce** *(Species)* |
|---|---|---|---|---|
| 5.1 | 3.5 | 1.4 | 0.2 | *I. setosa* |
| 4.9 | 3.0 | 1.4 | 0.2 | *I. setosa* |
| 4.7 | 3.2 | 1.3 | 0.2 | *I. setosa* |
| 4.6 | 3.1 | 1.5 | 0.2 | *I. setosa* |
| 5.0 | 3.6 | 1.4 | 0.2 | *I. setosa* |
| 5.4 | 3.9 | 1.7 | 0.4 | *I. setosa* |
| 4.6 | 3.4 | 1.4 | 0.3 | *I. setosa* |
| 5.0 | 3.4 | 1.5 | 0.2 | *I. setosa* |

source : https://fr.wikipedia.org/wiki/Iris_(jeu_de_données)

# How do we proceed

- I will give you the code soon
- To launch a Python program from the master node of the cluster.

```
spark-submit --master yarn \
             --executor-memory 7g \
             --num-executors 5 \
              --executor-cores 1 \
             kmeans-dario-x.py
```

- Step-by-step presentation of the code
- Then you can launch the job and eventually modify it.

# K-means clustering

## Initialising variables and RDDs

```
data = lines.map(lambda x: x.split(','))\
            .map(lambda x: [float(i) for i in x[:4]] + [x[4]])\
            .zipWithIndex()\
            .map(lambda x: (x[1], x[0]))

# zipWithIndex allows us to give a specific index to each point
# (0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa'])

clusteringDone = False

number_of_steps = 0
```

# K-means clustering

Initialising the centroids

# In the same manner, zipWithIndex gives an id to each cluster

```
centroides =
sc.parallelize(data.takeSample('withoutReplacment',nb_clusters
))\
               .zipWithIndex()\
               .map(lambda x: (x[1],x[0][1][:-1]))

# (0, [4.4, 3.0, 1.3, 0.2])
```

# K-means clustering

## Points repartition

```
joined = data.cartesian(centroides)
# ((0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']), (0, [4.4, 3.0, 1.3, 0.2]))
```

# We compute the distance between the points and each cluster

```
dist = joined.map(lambda x: (x[0][0],(x[1][0], computeDistance(x[0][1][:-1], x[1][1]))))

def computeDistance(x,y):
    return sqrt(sum([(a - b)**2 for a,b in zip(x,y)]))

# (0, (0, 0.866025403784438))


dist_list = dist.groupByKey().mapValues(list)

# (0, [(0, 0.866025403784438), (1, 3.7), (2, 0.5385164807134504)])
```

# K-means clustering

```
def closestCluster(dist_list):
    cluster = dist_list[0][0]
    min_dist = dist_list[0][1]
    for elem in dist_list:
        if elem[1] < min_dist:
            cluster = elem[0]
            min_dist = elem[1]
    return (cluster,min_dist)
```

## Points repartition

# We keep only the closest cluster to each point.

```
min_dist = dist_list.mapValues(closestCluster)

# (0, [(0, 0.866), (1, 3.7), (2, 0.538)])
# (0, (2, 0.538))
```

# assignment will be our return value : It contains the datapoint,
# the id of the closest cluster and the distance of the point to the centroid

```
assignment = min_dist.join(data)

# (0, ((2, 0.538), [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']))
```

# K-means clustering

## New centroids

```
# (0, ((2, 0.538), [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']))

clusters = assignment.map(lambda x: (x[1][0][0], x[1][1]
[:-1]))

# (2, [5.1, 3.5, 1.4, 0.2])

count = clusters.map(lambda x: (x[0],1)).reduceByKey(lambda
x,y: x+y)
somme = clusters.reduceByKey(sumList)


newCentroides = somme.join(count).map(lambda x :
(x[0],moyenneList(x[1][0],x[1][1])))

def moyenneList(x,n):
    return [x[i]/n for i in range(len(x))]
```

# K-means clustering

## End Condition

# Based on counting the number of point moves

```
if number_of_steps > 0:
       switch = prev_assignment.join(min_dist)\
                                   .filter(lambda x: x[1][0][0] != x[1][1][0])\
                                   .count()
       else:
            switch = 150
```

# K-means clustering

Spark
Lightning-Fast Cluster Computing

## End Condition

```
if switch == 0 or number_of_steps == 100:
    clusteringDone = True
    error = sqrt(min_dist.map(lambda x: x[1][1]).reduce(lambda x,y: x + y))/
nb_elem.value

else:
    centroides = centroidesCluster
    prev_assignment = min_dist
    number_of_steps += 1
```

# The whole program

- Available at https://www.dropbox.com/s/tm9lk6vffci5yzr/kmeans-dario-x.py
- ATTENTION : you need to
    - load the iris data on your HDFS home
    - change the program by indicating where to load the data and to store the result - this is left as an exercice
    - On AWS run the job with

```
spark-submit --master yarn \
             --executor-memory 7g \
             --num-executors 5 \
             --executor-cores 1 \
             kmeans-dario-x.py
```

# Cluster Dauphine

```
spark-submit \
--master yarn --deploy-mode cluster \
--executor-cores 4 \
--num-executors 11 \
--executor-memory 5g \
--conf spark.yarn.executor.memoryOverhead=2g \
--conf spark.driver.memory=5g \
--conf spark.driver.cores=1 \
--conf spark.yarn.jars="file:///home/cluster/shared/spark/jars/*.jar" \
kmeans-dario-x.py
```

# Project on Kmeans

- ***Deadline Febraury 3, 2020***

- Make experimental analysis of the basic Python version, eventually by lowering the number of iterations
- Find, describe, and implement otpimizations
- Use bigger input data instead of Iris, or bigger Iris instances, and perform experiments
- Switch to Scala
    - RDD
    - Dataframes
    - Datasets
    - Pick a sufficiently large input and make experiments to compare Scala implementations
- Write a technical report, around 20 pages max, with main points about implementations and experiments.
- Send the report + the code by email by the indicated deadline.

# Second PROJECT

- ***Deadline Febraury 3, 2020***

- Consider the book http://www.iro.umontreal.ca/~nie/IFT6255/Books/MapReduce.pdf

- Implement in Spark algorithms presented in Chapter 6 about EM and HMM and perform experiments, by usisng indicated data sets + another data set you will find in the web

- Discuss eventual efficiency problems (at least one) and propose some solutions (at least one) supported by experiment.

- Write a report, max 20 pages, clearly introducing the problem and solutions and experiments.

- Send the report + the code by email by the indicated deadline.