Hadoop Spark Project

Authors:

Sina Gholami

Lorenzo Giorgi

Claudia Gabriela Sánchez Mendiola

Course Details:

Cloud Computing (Cod. 866ll) 2019-20

Prof. Nicola Tonellotto

Prof. Carlo Vallati

Contents

1. Introduction

In this project we implemented the well-known K-means algorithm by using the MapReduce framework in Hadoop and Spark. First, we depict our pseudo code which is the fundamental base for our implementation. Then we explain in detail our codes and experiments in the following sections.

2. Pseudo Code

2.1. Random Selection MapReduce

```
Class Mapper
    Method Map(pid n, point p)
        Emit (n, p)
Class Reducer
    Method Setup()
        randomPids ← [Selects randomly k pids without replacement]
    Method Reducer(pid, [$p_1$, $p_2$, …])
        if pid in randomPids then
            Emit(pid, $p_1$)
```

2.2. K-means MapReduce

```
Class Mapper
    Method Map(pid n, point p)
        minDist = MAX INTEGER
        for all c in Centroids do
            dist ← distance(p, c)
            if dist < minDist then
                index ← index of c in Centroids
        Emit(index, p)


Class Combiner
    Method Combiner(index, [$p_1$, $p_2$, …])
```

```
        sum ← [0, …, $p_1$.dimensions.length – 1]
    for all p in [$p_1$, $p_2$, …] do
        for all d in range p.dimensions do
            $sum_d$ ← $sum_d$ + p.dimensions(d)
        Emit(index, sum)

Class Reducer
    Method Reducer(index, [$p_1$, $p_2$, …])
        sum ← [0, …, $p_1$.dimensions.length – 1]
    for all p in [$p_1$, $p_2$, …] do
        for all d in range p.dimensions do
            $sum_d$ ← $sum_d$ + p.dimensions(d)
        sum ← sum / (count [$p_1$, $p_2$, …] + combined elements)
        Emit(index, sum)
```

3. Hadoop

The k-means implementation in Hadoop comprised of two MapReduce parts:

1. Random centroid selection (*Centroid MapReduce*)
2. Finding the optimal value of centroids based on the selected centroids (*K-means MapReduce*)

Since Hadoop suffers from poor inter-communication capability and inadequacy for in-memory computation, we decided to consider a map reduce section for selecting k points randomly. Our code contains of four Java classes:

1. DataPoint.java which is our data structure of input data in our application,
2. Main.java which is responsible for running the main method class,
3. Centroids.java in which the mapper and reducer of random centroid selection is implemented,
4. KMeans.java in which the mapper and reducer of k-means algorithm is implemented.

3.1. Implementation

3.1.1. DataPoint.java

The data point is vector of values. Each record in the vector represents the value of that data point in corresponding dimension. For instance, for data point in two dimensions, we have:

```
[123, 22]
```

The type of the vector is `ArrayList<Double>`. The `write` and `readFields` methods of `Writable Interface` are based on the string.

Method public int `findNearestCentroid(ArrayList<DataPoint> centroids)` finds the minimum distance of the point to the list of input centroids. To find the minimum distance we implement:

1. `public double distance(DataPoint that, int n)` and
2. `public double norm(int n)`.

The norm method calculates the norm of the vector. If n is 2, it calculates the Euclidean norm as follow:

$$\|x\|_2 := \sqrt{x_1^2 + \cdots + x_n^2}$$

By having the norm, we can calculate the Euclidean distance between two arbitrary data points p and q as follow:

$$\|q - p\| = \sqrt{\|p\|^2 + \|q\|^2 - 2p \cdot q}$$

Absolute distance is not sufficient for the calculating the distance. In addition, higher order distance implementations are more time consuming, thus we only consider Euclidean distance.

We also implemented the equals and `compareTo` methods.

3.1.2. Main.java

The Main class brings together all the implementation of the k-means algorithm. In this class, we declare the inputs and output paths. The inputs of our code are as follow:

1. k: number of centroids,
2. rows: matrix row length or the total number of considered data point,
3. columns: matrix column length or number of dimensions each point can have,
4. threshold: a threshold to converge the algorithm,
5. maxItr: maximum number of iterations,

6. reducers: number of reducers,
7. combiner: a boolean to set the combiner true or false,
8. input: the path of input file in Hadoop distributed file system,
9. output: the path of output files in Hadoop distributed file system.

3.1.2.1.1. Maximum Iteration (maxItr)

We consider this input to limit the number of iterations of the algorithm and provide more features for our implementation.

3.1.2.1.2. Threshold

When an iteration finished, the value of the centroids compares with the previous iteration in order to find out if the algorithm converged or not. Since in the final iterations, the changes are negligible we can omit these small differences. Therefore, we consider a threshold by which the distance of new calculated centroid to the previous one should be bigger than the threshold in order to allow the algorithm to proceed another iteration.

$$preCentroid.euclideanDistance(newCentroid) > threshold$$

3.1.2.1.3. Input

Note that the input file must have the following format:

$Row\ 1,$	$dimention\ 1,$	$dimention\ 2,$	… ,	$dimention\ m$
$Row\ 2,$	$dimention\ 1,$	$dimention\ 2,$	… ,	$dimention\ m$
… ,	$dimention\ 1,$	$dimention\ 2,$	… ,	$dimention\ m$
$Row\ n,$	$dimention\ 1,$	$dimention\ 2,$	… ,	$dimention\ m$

In order to provide such an input, we run the following python script. are generating random numbers between 0 and 500000:

```
>>> import numpy as np
>>> k = 13
>>> m = 7
>>> n = 1000000
>>> A = np.random.randint(n/2, size=(n, m))
>>> B = np.zeros((n,m+1))
>>> B[:,1:] = A
>>> B[:,0:1] = np.arange(0, n).reshape((n,1))
>>> np.savetxt('k-means_input.txt',B,fmt='%.0f',delimiter=",")
```

But why do we need the row number? Because in *Centroid MapReduce* we must check the k generated random numbers with all the row numbers to find out which data point is selected. However, we do not need the row numbers or indices for Spark.

3.1.2.1.4. Output

The program generates three output files:

1. Path to output/new/part-r-00000: Final Values of Centroid,
2. Path to output/temp/part-r-00000: Latest value of centroid before convergence,
3. Path to output/pre/part-r-00000: Random selected data points for starting the algorithm. It is generated by *Centroid MapReduce* of random centroid selection.

Example) For k = 3, m = 2 the output file has the following format:

```
1;	31.54,663.10
2;	211.21,192.19
3;	6799.03,1671.54
```

In all three files, the format is the same.

After defining the inputs and outputs of the application, the *Centroid MapReduce* is run by:
`Centroid.run(conf, k, n, m, path to input, path to output + "/pre");`

Note that n and m are the rows and columns of the input data matrix. The output of this method is stored in path to output/pre/part-r-00000, as we mentioned above.

After the *Centroid MapReduce* finished, the *K-means MapReduce* is called by:

```
KMeans.run(maxItr, conf, combiner, reducers, k, m, n, threshold, path to
input, path to output);
```

We will explain these two MapReduce in their corresponding classes.

3.1.3. Centroid.java

The centroid class contains the mapper and reducer subclasses and two other methods. Here is some explanation about the methods:

`public static int run(Configuration conf, int k, int n, int m, String input, String output)` is responsible for Hadoop configuration and we mentioned it before in Main.java.

`public static ArrayList<DataPoint> readCentroids(int k, String path, FileSystem hdfs)` reads the stored data point from the path file. We use this method in order to allow the mapper of the *K-means MapReduce* to read the centroids in each iteration.

The mapper subclass `CentroidMapper`, reads from the input data file and maps each row of the data to the reducer. The key to the read data in reducer is the row number and the value is vector of dimensions of respected data point which is restricted according to the number of columns. In summary, mappers pass n row numbers and m column number of the input data to the reducer.

In reducer subclass `CentroidReducer`, first we generate k random numbers between 0 and n in `setup` method. Then compare each key (row index) with the generated random number in `reduce` method. If they are equal, then the reducer writes that data point to the output.

3.1.4. KMeans.java

Three subclasses are implemented in this class: mapper, combiner and reducer. First, in mapper setup, we read the centroids (output of *Centroid MapReduce*) from the file by using the `Centroid.readCentroids` method. Then the mapper maps each data point to the nearest centroid by using `findNearestCentroid` method in `DataPoint`. The key is index of the nearest centroid for the corresponding data point and the value is the line of data.

In combiner, all the values that have the same key are aggregated. It means element wise aggregation of the values in `ArrayList<Double> vector`. The combiner writes the key as the received key, and then aggregation results plus number of data point that are aggregated together in the following format:

```
context.write(key, new Text(Arrays.toString(sumVectors).replace("[",
"").replace("]", "") + ";" + n));
```

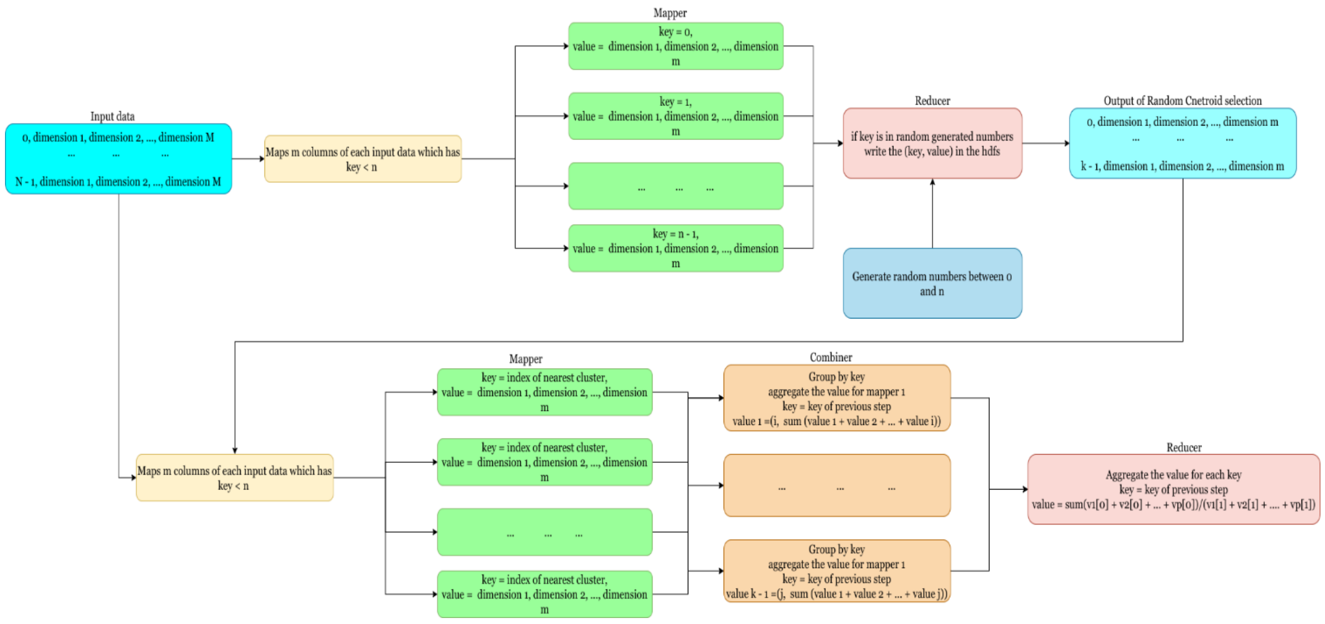
In reducer we do the same procedure with slight change, because we need to update the new value of the centroid. Thus, after we aggregate all the vectors, we divide them by the total of the data that are aggregated together.

The method run:

```
run(maxItr, conf, combiner, reducers, k, m, n, threshold, path to input, path
to output);
```

has the configuration of the *K-means MapReduce* job. It contains a while loop mainly. At the end of each iteration, when *K-means MapReduce* job finished, we compare the previous centroids with the new ones. If there is any change with respect to the threshold and the number of current iterations is less than the maximum iteration (maxItr), the *K-means MapReduce* job is executed one more time.

3.2. MapReduce Flow Chart


Input data
0, dimension 1, dimension 2, ..., dimension M
... ...
N - 1, dimension 1, dimension 2, ..., dimension M
Maps m columns of each input data which has key < n
Mapper
key = 0, value = dimension 1, dimension 2, ..., dimension m
key = 1, value = dimension 1, dimension 2, ..., dimension m
... ...
key = n - 1, value = dimension 1, dimension 2, ..., dimension m
Reducer
if key is in random generated numbers write the (key, value) in the hdfs
Generate random numbers between 0 and n
Output of Random Cnetroid selection
0, dimension 1, dimension 2, ..., dimension m
... ...
k - 1, dimension 1, dimension 2, ..., dimension m
Maps m columns of each input data which has key < n
Mapper
key = index of nearest cluster, value = dimension 1, dimension 2, ..., dimension m
key = index of nearest cluster, value = dimension 1, dimension 2, ..., dimension m
... ...
key = index of nearest cluster, value = dimension 1, dimension 2, ..., dimension m
Combiner
Group by key aggregate the value for mapper 1 key = key of previous step value 1 =(i, sum (value 1 + value 2 + ... + value i))
... ...
Group by key aggregate the value for mapper 1 key = key of previous step value k - 1 =(j, sum (value 1 + value 2 + ... + value j))
Reducer
Aggregate the value for each key key = key of previous step value = sum(v1[0] + v2[0] + ... + vp[0])/(v1[1] + v2[1] + ... + vp[1])

3.3. Benchmarks

The input data is random numbers between 0 and 500000 in our all experiments. The written time in the following tables are calculated by Java `System.currentTimeMillis();`. The time interval starts before the while loop of the *K-means MapReduce* and ends when no more job is available to be executed. Each test run twice to give better intuition about the time.

In the first one, maximum number of iterations (maxItr) and number of reducers are 1.

Input		Output			
# of datapoint	**# of dimension**	**# of Cluster**	**Combiner**	**Time (s) Experiment 1**	**Time (s) Experiment 2**
1000	3	7	No	21.583	25.776
1000	3	13	No	24.731	27.726
1000	7	7	No	23.055	22.539
1000	7	13	No	20.563	21.895
10000	3	7	No	20.572	20.734
10000	3	13	No	27.671	23.612
10000	7	7	No	26.907	20.791
10000	7	13	No	23.737	21.626
100000	3	7	No	22.641	25.635
100000	3	13	No	21.684	21.672

100000	7	7	No	22.622	24.648
100000	7	13	No	22.578	21.971
1000000	3	7	No	28.711	27.666
1000000	3	13	No	30.656	32.713
1000000	7	7	No	30.767	29.641
1000000	7	13	No	37.787	34.696
1000	3	7	Yes	22.641	25.635
1000	3	13	Yes	21.541	22.708
1000	7	7	Yes	21.594	24.743
1000	7	13	Yes	20.584	21.586
10000	3	7	Yes	20.627	23.532
10000	3	13	Yes	21.044	21.617
10000	7	7	Yes	21.638	24.583
`10000	7	13	Yes	22.649	20.592
100000	3	7	Yes	23.857	21.749
100000	3	13	Yes	22.793	20.821
100000	7	7	Yes	23.631	21.629
1000000	3	7	Yes	28.682	30.748
1000000	3	13	Yes	31.697	32.663
1000000	7	7	Yes	32.672	39.096
1000000	7	13	Yes	37.737	33.676

The results show that the combiner did not increase our implementation efficiency. We changed the maximum number of iterations to 10.

Input		Output			
# of datapoint	**# of dimension**	**# of Cluster**	**Combiner**	**Time (s) Experiment 1**	**Time (s) Experiment 2**
1000000	3	7	No	296.390	309.742
1000000	3	13	No	320.194	340.337
1000000	7	7	No	336.423	339.350
1000000	7	13	No	361.586	382.344
1000000	3	7	Yes	338.960	362.136
1000000	3	13	Yes	355.627	389.598
1000000	7	7	Yes	338.450	344.113
1000000	7	13	Yes	395.544	391.156

As we expected there is a linear dependency between consuming time and number of iterations. Finally, we changed to number of reducers to 3, number of iteration to 1 and test it again.

Input		Output			
# of datapoint	**# of dimension**	**# of Cluster**	**Combiner**	**Time (s) Experiment 1**	**Time (s) Experiment 2**
1000000	3	7	No	30.830	27.618
1000000	3	13	No	34.689	32.725
1000000	7	7	No	30.675	33.704
1000000	7	13	No	36.097	40.011
1000000	3	7	Yes	30.729	34.726
1000000	3	13	Yes	35.931	35.723
1000000	7	7	Yes	32.022	31.665
1000000	7	13	Yes	34.697	34.769

Mainly, the performance of our algorithm depends on the Unipi cloud service. Thus, it is somehow impossible to measure the efficiency of our algorithm.

4. Spark

Our Spark implementation is mainly divided into two part:

- Selection of the first, random, centroid
- A *while* loop in which K-means is implemented.

4.1. Implementation

Datapoints are loaded from a file stored on the Hadoop Distributed File System (HDFS) using the Spark Context function *textFile*. This function loads the input data, in a distributed way, into RDD.

Those RDD are RDD of Strings for now, so we must convert them into numeric form. This task is accomplished by *create_vector*: it returns each datapoint in the form:
[c_1,c_2, c_3, …, c_n], in which c stands for component.

takeSample takes the datapoints and return k points, without duplicates, into a list. The first k centroids are now inside the list *centroids*.

Now is possible to assign each datapoint to the nearest centroid: *assign_to_cluster* does it. The "nearness" is computed using Euclidian distance. a*ssign_to_cluster* returns a tuple (index_of_centroid, datapoint).

Using groupByKey all datapoints are grouped by the cluster to which they belong, the output of this step is (index_of_centroid, list of datapoint).

For each centroid, *vector_sum* computes the sum of each component of each point and divide it for the number of elements of the list of datapoint of a specific cluster. The output is a list of *new_centroids.* If they are the same (or the distance is under a specified threshold) of the last iteration (kept in *centroids*), the algorithm is finished and we can output the result on HDFS (computed centroids and datapoints with the index of the centroid). Otherwise, we must assign *new_centroids* to *centroids* and we restart the loop from *assign_to_cluster*.

In this project we do not defined a max number of iterations of the algorithm, but it is possible to enter a threshold between two iteration to stop the algorithm before the full convergence.

4.2. Input

4.2.1. Command Line

To run the file, you must run the following command:

spark-submit k-means.py <k> <rows> <dimension> <threshold> <master> <input file> <output file>

In detail:

- k: number of desired clusters.
- rows: number of datapoints provided as input.
- dimension: the number of dimensions of the input points.
- threshold: if the distance between all corresponding centroids between two consecutive iteration is under this value, the algorithm stops. If the user wants the full convergence of the algorithm, he can put threshold equal to 0.
- master: accepted values are the same of master argument of SparkContext constructor. To run the application on the cluster, write *yarn.*
- Input file: path of the input file. If the user work on large dataset is recommended to use HDFS storage.
- Output file: path to the output file. If the user work on large dataset is recommended to use HDFS storage. In that case, a folder will be created with the name specified in output file. Inside that folder you will find the output of the program.

4.2.2. Input Data

The input data are generated in the same way described in the Hadoop section.

4.3. Output data

4.3.1. Centroids

The application will output the centroids with the following structure:

[c_1,c_2, c_3, …, c_n]

This is an example of output with k=7 and d=3

```
[118960.2251223491, 119865.94208809135, 366962.9339314845]
[136610.018922853, 369201.77219796216, 379940.02838427946]
[140679.39316811788, 125475.21835231078, 125001.1862022773]
[154890.72641509434, 384903.0626684636, 124797.0128032345]
[372954.08378746593, 126568.67506811989, 359696.2813351499]
[386599.03364352183, 391027.3350035791, 333596.73013600573]
[393405.9704749679, 224008.5815147625, 101956.59563543004]
```

4.3.2. Datapoints

The application will output the datapoints with the following structure:

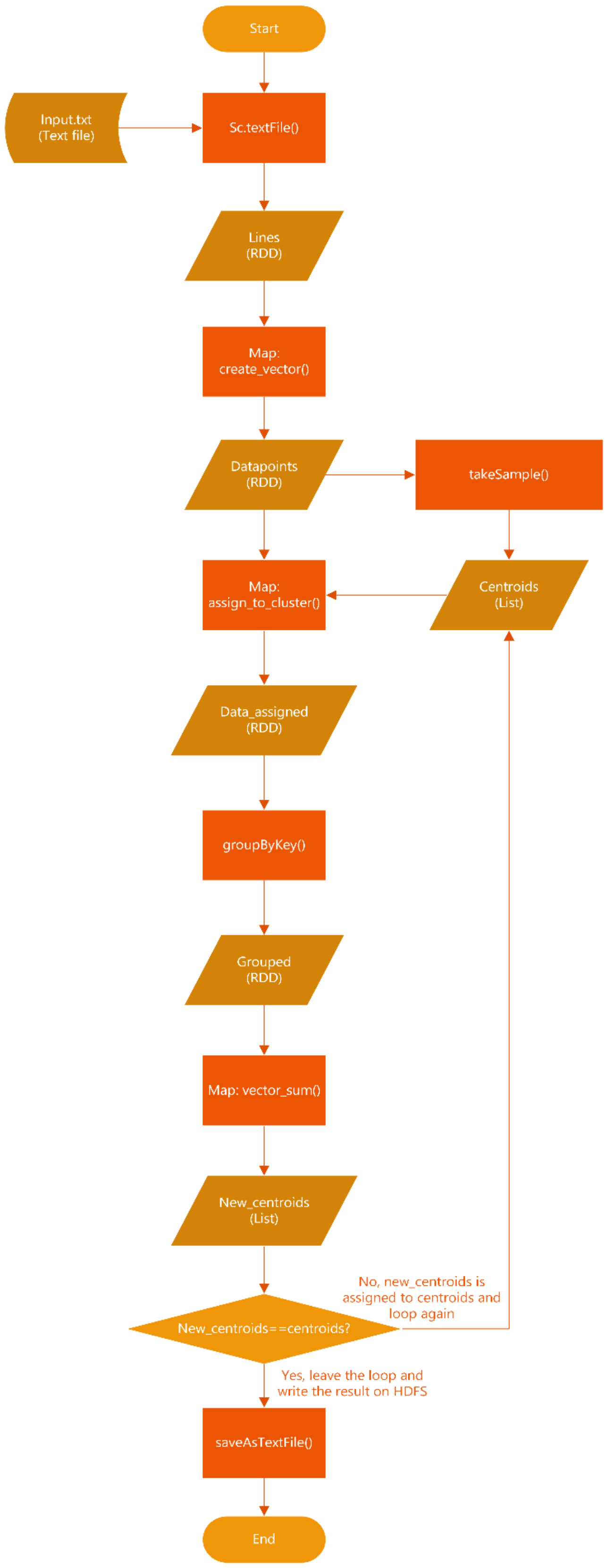
(cluster_index, datapoint)

This is an example of the output with k=7 and d=3

```
(6, [349132.0, 350641.0, 163768.0])
(3, [90474.0, 415332.0, 199090.0])
(4, [382924.0, 88852.0, 234960.0))
(3, [189203.0, 292601.0, 36963.0])
(3, [312933.0, 454573.0, 35687.0])
(0, [163962.0, 56454.0, 372429.0])
(3, [81510.0, 464053.0, 136366.0])
(4, [423968.0, 176455.0, 418867.0])
(4, [388750.0, 23888.0, 488985.0])
(5, [447126.0, 483289.0, 413386.0])
```

N.B. using HDFS as output the user will obtain a folder with some part-**** file. This is because each node output a part of the total data

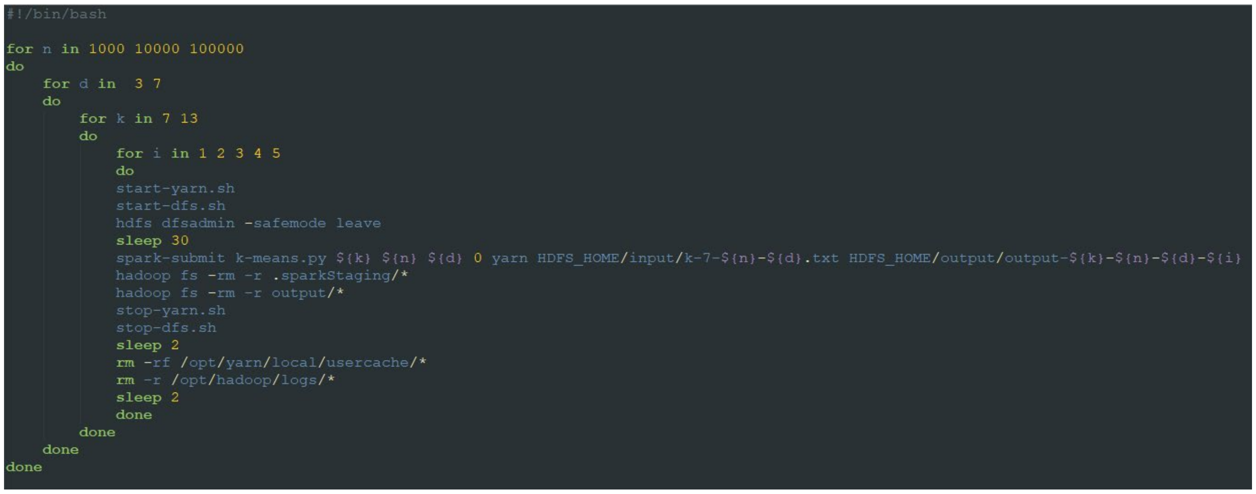
4.4. Spark Flow Chart



4.5. Benchmarks

4.5.1. Script

This is the script that we run on our namenode to make the tests:



```bash
#!/bin/bash

for n in 1000 10000 100000
do
    for d in  3 7
    do
        for k in 7 13
        do
            for i in 1 2 3 4 5
            do
            start-yarn.sh
            start-dfs.sh
            hdfs dfsadmin -safemode leave
            sleep 30
            spark-submit k-means.py ${k} ${n} ${d} 0 yarn HDFS_HOME/input/k-7-${n}-${d}.txt HDFS_HOME/output/output-${k}-${n}-${d}-${i}
            hadoop fs -rm -r .sparkStaging/*
            hadoop fs -rm -r output/*
            stop-yarn.sh
            stop-dfs.sh
            sleep 2
            rm -rf /opt/yarn/local/usercache/*
            rm -r /opt/hadoop/logs/*
            sleep 2
            done
        done
    done
done
```

We repeat each combination for 5 times, with a total of 60 tests. HDFS_home stands for hdfs://hadoop-namenode:9820/user/hadoop, we changed for graphic needs (the actual script report all the path correctly).

To reduce problems with the cloud infrastructure, every test restart both yarn and HDFS on all nodes. During our test we encounter also low storage issue, therefore we delete all the produced file after every test. At the beginning, we also had to raise the upper limit of occupied memory above which a node is considered unhealthy to 98% (using the property *yarn.nodemanager.disk-health-checker.max-disk-utilization-per-disk-percentage*). We use the sleep functions to be sure that, despite the slowness of the platform, all services are correctly started (N.B. this is theoretically unnecessary).

We ran our test during night to avoid the load on the cloud platform due to smart-working daily load, in order to obtain more reliable results.

4.5.2. Results

Input		Output			
# of datapoint	**# of dimension**	**# of Cluster**	**# Iteration**	**Time (s)**	**avg. time for iteration**
1000	3	13	16	10,194	0,64
1000	3	7	13	9,454	0,73
1000	7	13	27	13,343	0,49
1000	7	7	21	10,159	0,48
10000	3	13	44	23,576	0,54
10000	3	7	30	13,505	0,45

10000	7	13	56	31,948	0,57
10000	7	7	109	45,321	0,42
100000	3	13	81	104,453	1,29
100000	3	7	47	68,658	1,46
100000	7	13	269	498,584	1,85
100000	7	7	220	269,775	1,23

Table reports the best result (minimum time of execution) from the 5 tests. We can see that the total time of execution increases a lot with 100.000 datapoint input. We tried to run also 1-million-point test, it works perfectly but it takes 16973 seconds (4 hours and 42 minutes)

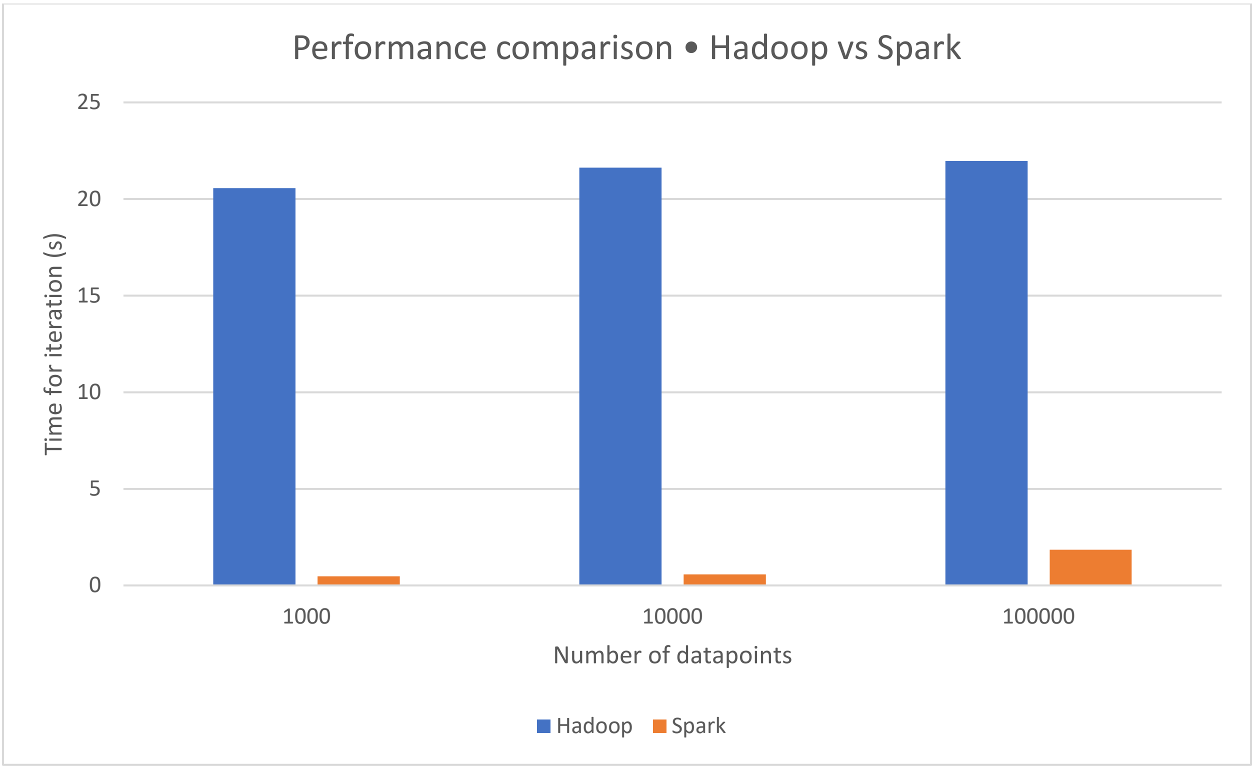
It is not possible to draw further conclusions from the data analysis, as each algorithm execution draws different starting centroids, and this affects the total execution time.

The load on the University's cloud infrastructure is also variable and this significantly affects the time needed to execute each algorithm iteration.

5. Performance Comparison

Although our implementation in both Spark and Hadoop follows a similar pseudo code, yet there is a huge difference in computations. Since Spark carried out the computations in memory, until the user actively persists them.
Therefore, Spark runs the algorithm much faster than Hadoop MapReduce according to the following reasons:

- Hadoop does not have any cyclical connection between MapReduce steps and Spark's DAGs (Directed Acyclic Graph) enables optimizations between steps.
- Spark is not bound by input-output concerns every time it runs a selected part of a MapReduce task.
- Spark is relatively faster than Hadoop, since it caches most of the input data in memory by the RDD.
- Spark starts evaluating only when it's absolutely needed, so this contributes directly to its speed.



In this chart we used k=13, d=7 and we chose the best time obtained. As we see, Spark provides about 20-30x better performance than Hadoop.

6. Conclusion

As we expected, the results shows that the Spark works faster than Hadoop due to the mentioned results.

7. Bibliography

- [cam-san] "Cloud Computing" by Sandeep Bhowmik. Cambridge University Press; 1 edition (July 4, 2017). ISBN-13: 978-1316638101
- Material provided by instructor (slides)
- TECHVIDVAN (December 17, 2019). *" Apache Spark DAG: Directed Acyclic Graph"*. Obtained from: https://techvidvan.com/tutorials/apache-spark-dag-directed-acyclic-graph/