Università di Pisa

SCUOLA DI INGEGNERIA
Corso di Laurea in Artificial Intelligence and Data Engineering

Cloud Computing project: The *k*-means Clustering Algorithm in MapReduce

Candidati
Alice Nannini
Marco Parola
Stefano Poleggi

Relatore
Dr. Nicola Tonellotto

Anno Accademico 2019–2020

Contents

1 Introduction 1

2 Dataset 1

3 MapReduce pseudo-code 2

4 Hadoop Implementation and Tests 3

5 Spark Implementation and Tests 3

6 Tests' Results 4

1 Introduction

This project presents the implementation of the k-means algorithm based on a MapReduce version, using both the Hadoop framework both the Spark framework.
The two implementations of the k-means algorithm developed must be performed with the following inputs:

- Name of the input file containing the dataset
- Number of centroids/clusters
- Output directory
- Number of total samples in the input dataset (the algorithm can be run assuming that you know this value)

The algorithm exit can occur due to two events:

- The maximum number of possible iteration has been reached
- The centroids calculated at i-th step and i+1-th step do not deviate beyond a certain threshold (Euclidean norm)

2 Dataset

The datasets for the final tests were generated with a python script, shown below and having the following format: *‘dataset_ numPoints_ kClusters_ dimPoints’*.
We tested our implementations with different number of dimensions for the point (d), different number of clusters to find (k), and different dimension of the input datasets (n).

```
import random
# inputs: n (records), k (clusters), d (dimensions)
numPoints = [1000,10000,100000]
kClusters = [7,13]
dimPoints = [3,7]

for n in numPoints:
    for k in kClusters:
        for d in dimPoints:
            # open a new file
            f = open("data/dataset_"+str(n)+"_"+str(k)+"_"+str(d)+".txt", "a")

            # compute the interval for creating the clusters
            interval = round(n/k)
            count = 0
            print("dataset_"+str(n)+"_"+str(k)+"_"+str(d)+"; int: "+str(interval))

            # compute each point
            for i in range(n):
                if( (i%interval)==0 and i!=0):
                    count = count + 4

                x = ""
                for j in range(d):
```

```
        x = x + str( interval*count + random.random()*interval )
        if(j < d-1):
          x = x + " "
      x = x + "\n"
      # write the new point coordinates in the file
      f.write(x)

    f.close()
```

List of files generated from the previous code:

- dataset_100000_13_3.txt
- dataset_100000_13_7.txt
- dataset_100000_7_3.txt
- dataset_100000_7_7.txt
- dataset_10000_13_3.txt
- dataset_10000_13_7.txt
- dataset_10000_7_3.txt
- dataset_10000_7_7.txt
- dataset_1000_13_3.txt
- dataset_1000_13_7.txt
- dataset_1000_7_3.txt
- dataset_1000_7_7.txt

3 MapReduce pseudo-code

The following pseudo-code shows the basic functioning of the Mapper and Reducer, implemented in this project:

```
class MAPPER
  method MAP(sample_id id, sample_list l)
    for all sample s in sample_list l do
      dist <- MAX_VALUE
      for all center c in cluster_centers cc do
        newDist <- computeDistance(s, c)
        if (newDist < dist) then
          dist <- newDist
          clusterIndex <- c.index
      EMIT(index clusterIndex, sample s)



class REDUCER
  method REDUCE(index clusterIndex, samples [s1, s2,...])
    count <- 0
    center <- cluster_centers[clusterIndex]
```

```
for all sample s in samples [s1, s2,...] do
  count <- count + 1
  for i in [0:size(s)] do
    newCenter[i] <- newCenter[i] + s[i]
for i in [0:size(newCenter)] do
  newCenter[i] <- newCenter[i] / count
EMIT(index clusterIndex, sample newCenter)
```

4 Hadoop Implementation and Tests

This implementation of the k-means algorithm is developed in a Maven project written in Java programming language, using the **org.apache.hadoop** library.
Initially, we did the tests on a first version, consisting simply in a Mapper class and a Reducer class. Then, we added a Combiner class to test improvements in the performance. For our Combiner, we decided to use the same implementation of the Reducer, that though will execute locally after the *mapping* part, to minimize both the input and the computation of the *reduce* function.
Finally, we tried different values for the threshold of the stop condition.

5 Spark Implementation and Tests

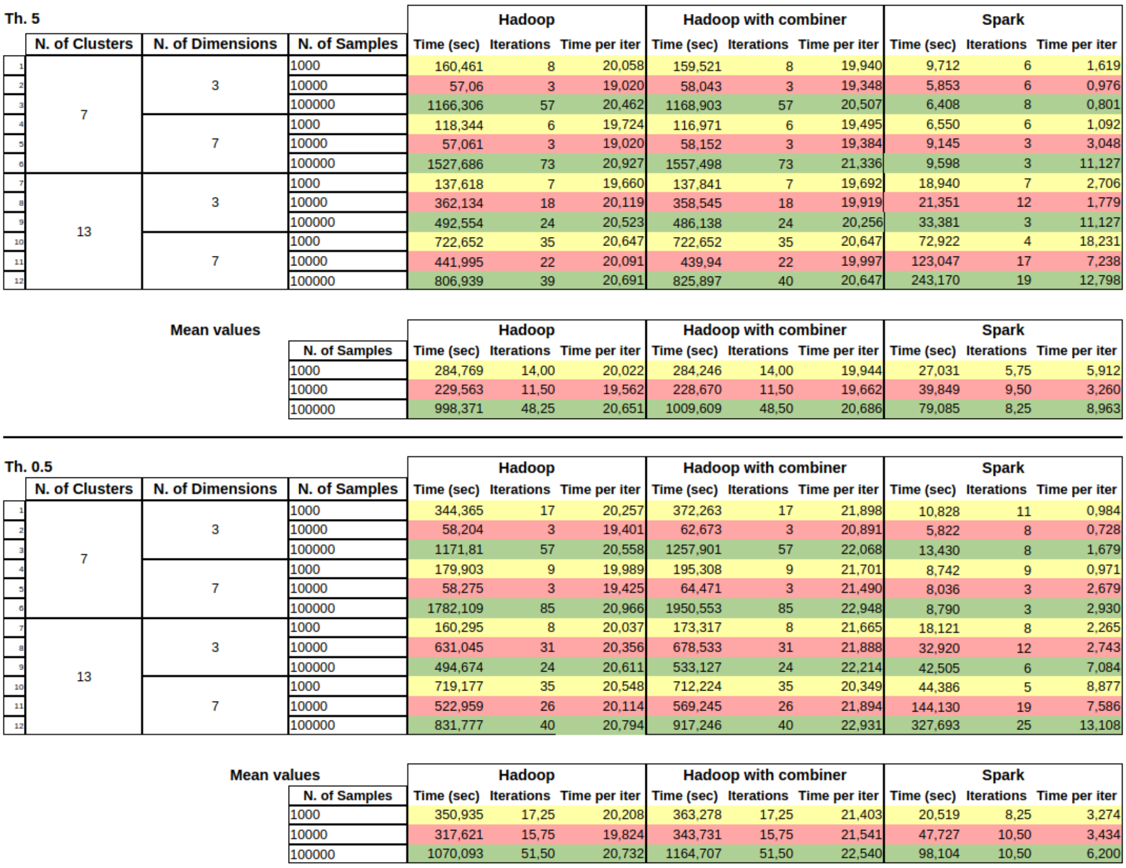
The Spark implementation is developed in Python language, using the **pyspark** library.
The exploited transformations are:

- *map*: takes as input the RDD created from the dataset file and gives as output an RDD of (K, V), in which the key is a cluster index and the value is the coordinates of a point.
- *reduceByKey*: takes the (K, V) pairs RDD, created by the mapping, and returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *computeNewCenter*, which is of type (V,V)=>V.

Also in this case, we tested our datasets with different threshold values of the stop condition.

6 Tests' Results

We have collected the results of our tests and created summary tables.



Th. 5				Hadoop			Hadoop with combiner			Spark		
	N. of Clusters	N. of Dimensions	N. of Samples	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter
1	7	3	1000	160,461	8	20,058	159,521	8	19,940	9,712	6	1,619
2			10000	57,06	3	19,020	58,043	3	19,348	5,853	6	0,976
3			100000	1166,306	57	20,462	1168,903	57	20,507	6,408	8	0,801
4		7	1000	118,344	6	19,724	116,971	6	19,495	6,550	6	1,092
5			10000	57,061	3	19,020	58,152	3	19,384	9,145	3	3,048
6			100000	1527,686	73	20,927	1557,498	73	21,336	9,598	3	11,127
7	13	3	1000	137,618	7	19,660	137,841	7	19,692	18,940	7	2,706
8			10000	362,134	18	20,119	358,545	18	19,919	21,351	12	1,779
9			100000	492,554	24	20,523	486,138	24	20,256	33,381	3	11,127
10		7	1000	722,652	35	20,647	722,652	35	20,647	72,922	4	18,231
11			10000	441,995	22	20,091	439,94	22	19,997	123,047	17	7,238
12			100000	806,939	39	20,691	825,897	40	20,647	243,170	19	12,798

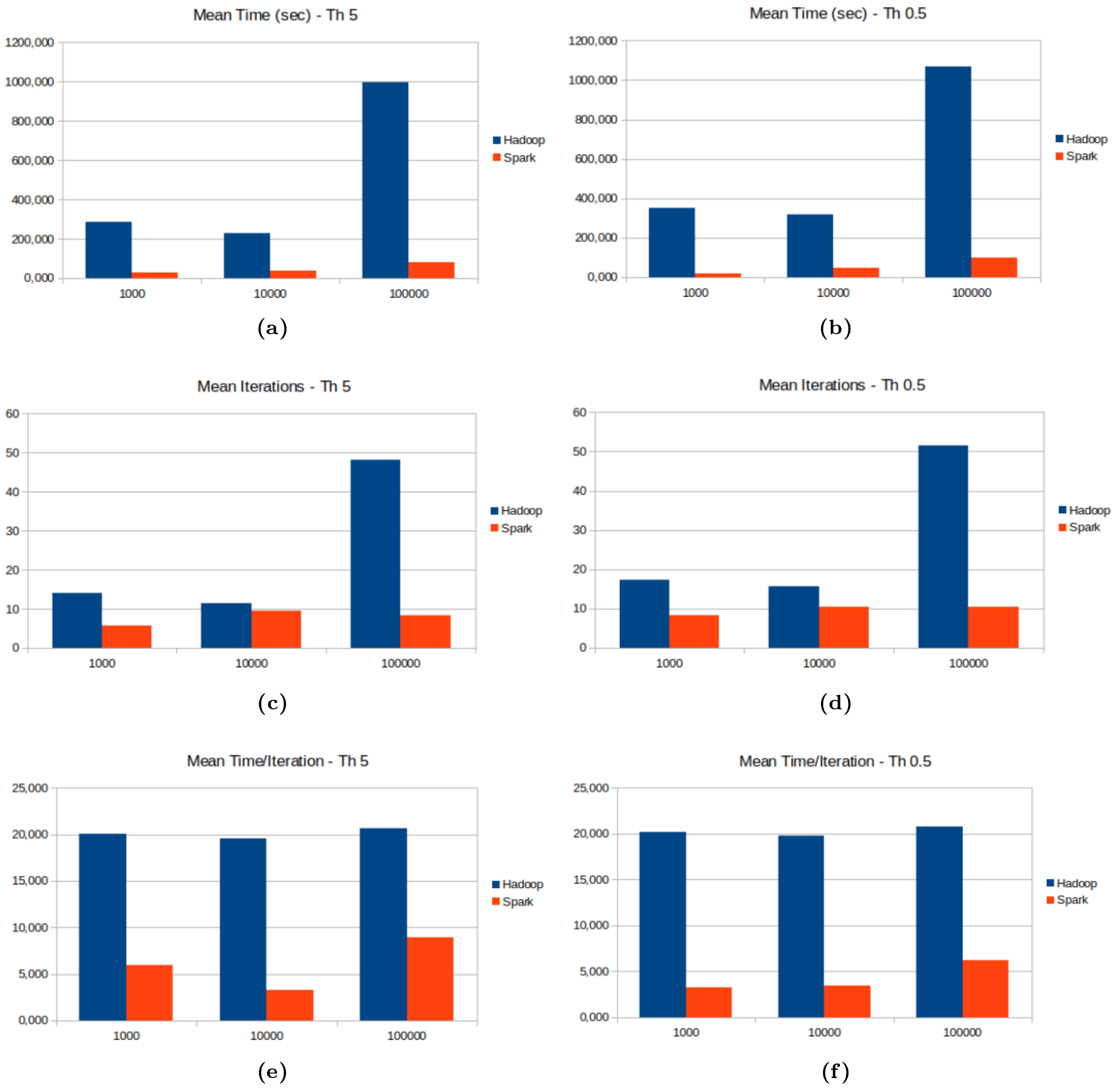
Mean values	Hadoop			Hadoop with combiner			Spark		
N. of Samples	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter
1000	284,769	14,00	20,022	284,246	14,00	19,944	27,031	5,75	5,912
10000	229,563	11,50	19,562	228,670	11,50	19,662	39,849	9,50	3,260
100000	998,371	48,25	20,651	1009,609	48,50	20,686	79,085	8,25	8,963

Th. 0.5				Hadoop			Hadoop with combiner			Spark		
	N. of Clusters	N. of Dimensions	N. of Samples	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter
1	7	3	1000	344,365	17	20,257	372,263	17	21,898	10,828	11	0,984
2			10000	58,204	3	19,401	62,673	3	20,891	5,822	8	0,728
3			100000	1171,81	57	20,558	1257,901	57	22,068	13,430	8	1,679
4		7	1000	179,903	9	19,989	195,308	9	21,701	8,742	9	0,971
5			10000	58,275	3	19,425	64,471	3	21,490	8,036	3	2,679
6			100000	1782,109	85	20,966	1950,553	85	22,948	8,790	3	2,930
7	13	3	1000	160,295	8	20,037	173,317	8	21,665	18,121	8	2,265
8			10000	631,045	31	20,356	678,533	31	21,888	32,920	12	2,743
9			100000	494,674	24	20,611	533,127	24	22,214	42,505	6	7,084
10		7	1000	719,177	35	20,548	712,224	35	20,349	44,386	5	8,877
11			10000	522,959	26	20,114	569,245	26	21,894	144,130	19	7,586
12			100000	831,777	40	20,794	917,246	40	22,931	327,693	25	13,108

Mean values	Hadoop			Hadoop with combiner			Spark		
N. of Samples	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter
1000	350,935	17,25	20,208	363,278	17,25	21,403	20,519	8,25	3,274
10000	317,621	15,75	19,824	343,731	15,75	21,541	47,727	10,50	3,434
100000	1070,093	51,50	20,732	1164,707	51,50	22,540	98,104	10,50	6,200

For each input file, the metrics we took into account to evaluate the performance are: the total execution time of the algorithms, the number of total iterations, and the average execution time per iteration.

We have noticed that, by introducing the Combiner class, performance has not improved. This means that the Combiner never run during our tests.



(a)
(b)
(c)
(d)
(e)
(f)

Mean Time (sec) - Th 5
1200,000
1000,000
800,000
600,000
400,000
200,000
0,000
1000
10000
100000
Hadoop
Spark
Mean Time (sec) - Th 0.5
Mean Iterations - Th 5
60
50
40
30
20
10
0
Mean Iterations - Th 0.5
Mean Time/Iteration - Th 5
25,000
20,000
15,000
10,000
5,000
0,000
Mean Time/Iteration - Th 0.5