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
Fabio Malloggi
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 out 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

	N. of Clusters	N. of Dimensions	N. of Samples	Hadoop			Hadoop with combiner			Spark		
				Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter
1			1000	264,899	13	20,377	237,716	11	21,611			
2		3	10000	347,624	17	20,448	435,251	20	21,763			
3	7		100000	1216,789	57	21,347	1267,473	57	22,236			
4			1000	285,961	14	20,426	216,650	10	21,665			
5		7	10000	831,144	40	20,779	546,979	25	21,879			
6			100000	1858,952	86	21,616	1671,549	73	22,898			
7			1000	202,597	10	20,260	150,745	7	21,535			
8		3	10000	369,463	18	20,526	394,275	18	21,904			
9	13		100000	465,263	22	21,148						
10			1000									
11		7	1000	468,667	23	20,377	482,523	22	21,933			
12			10000	938,895	44	21,339	888,659	39	22,786			

Mean values

N. of Samples	Hadoop			Hadoop with combiner			Spark		
	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter
1000	251,152	12,33	20,354	201,704	9,33	21,604	#DIV/0!	#DIV/0!	#DIV/0!
10000	504,225	24,5	20,532	464,757	21,25	21,870	#DIV/0!	#DIV/0!	#DIV/0!
100000	1119,975	52,25	21,362	1275,894	56,33	22,640	#DIV/0!	#DIV/0!	#DIV/0!

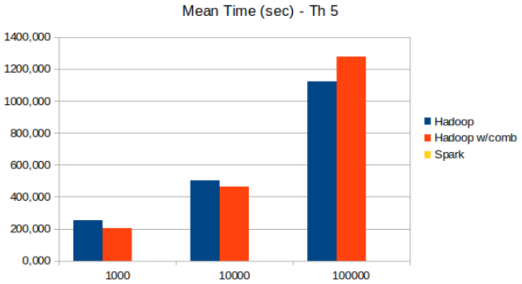
Th. 0.5

	N. of Clusters	N. of Dimensions	N. of Samples	Hadoop			Hadoop with combiner			Spark		
				Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter
1			1000				372,263	17	21,898	9,779	8	1,222
2		3	10000				62,673	3	20,891	10,753	3	3,584
3	7		100000				1257,901	57	22,068	60,694	3	20,231
4			1000				195,308	9	21,701	12,131	8	1,516
5		7	10000				64,471	3	21,490	16,551	3	5,517
6			100000				1950,553	85	22,948	154,456	4	38,614
7			1000				173,317	8	21,665	14,548	13	1,119
8		3	10000				678,533	31	21,888	32,278	8	4,035
9	13		100000				533,127	24	22,214	606,455	19	31,919
10			1000							15,572	9	1,730
11		7	1000				569,245	26	21,894	80,700	11	7,336
12			10000				917,246	40	22,931	854,504	25	34,180

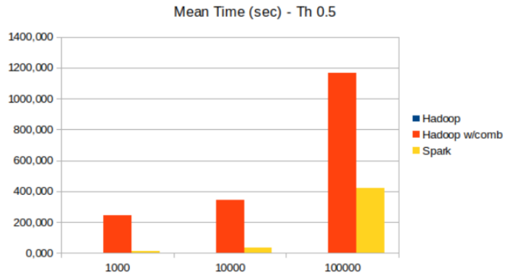
Mean values

N. of Samples	Hadoop			Hadoop with combiner			Spark		
	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter	Time (sec)	Iterations	Time per iter
1000	#DIV/0!	#DIV/0!	#DIV/0!	246,963	11,33	21,754	13,007	9,50	1,397
10000	#DIV/0!	#DIV/0!	#DIV/0!	343,731	15,75	21,541	35,071	6,25	5,118
100000	#DIV/0!	#DIV/0!	#DIV/0!	1164,707	51,50	22,540	419,027	12,75	31,236

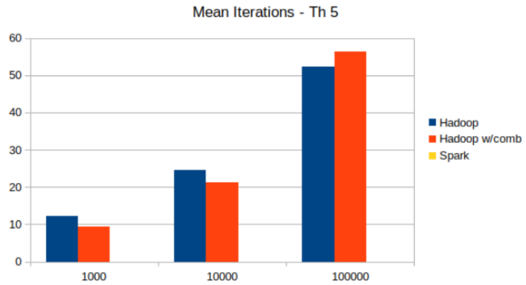
For each input file, the metrics we took into account to evaluate the performance are: the total execution time of the algorithms, the number of total iterations, and the average execution time per iteration.
We have noticed that, by introducing the Combiner class, performance has not significantly improved. Probably, this lack of improvement is due to the unreachability of some of our virtual machine during the execution of the tests with the Combiner.


Mean Time (sec) - Th 5
1400,000
1200,000
1000,000
800,000
600,000
400,000
200,000
0,000
1000
10000
100000
Hadoop
Hadoop w/comb
Spark

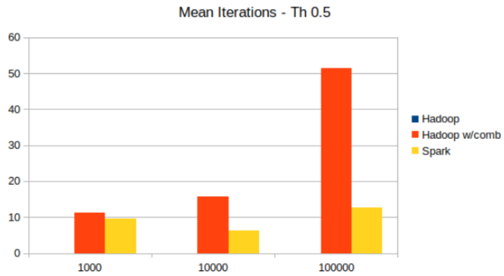
(a)


Mean Time (sec) - Th 0.5
1400,000
1200,000
1000,000
800,000
600,000
400,000
200,000
0,000
1000
10000
100000
Hadoop
Hadoop w/comb
Spark

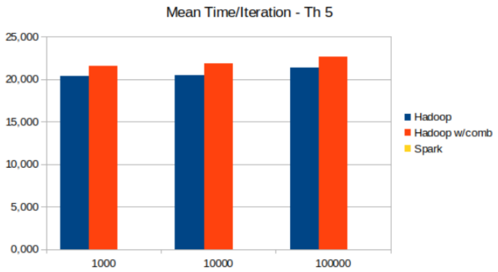
(b)


Mean Iterations - Th 5
60
50
40
30
20
10
0
1000
10000
100000
Hadoop
Hadoop w/comb
Spark

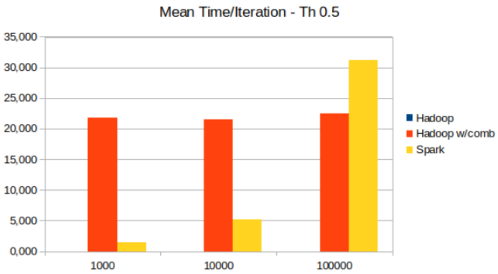
(c)


Mean Iterations - Th 0.5
60
50
40
30
20
10
0
1000
10000
100000
Hadoop
Hadoop w/comb
Spark

(d)


Mean Time/Iteration - Th 5
25,000
20,000
15,000
10,000
5,000
0,000
1000
10000
100000
Hadoop
Hadoop w/comb
Spark

(e)


Mean Time/Iteration - Th 0.5
35,000
30,000
25,000
20,000
15,000
10,000
5,000
0,000
1000
10000
100000
Hadoop
Hadoop w/comb
Spark

(f)