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

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 language, using the **org.apache.hadoop** library.

The first version consists in a Mapper class and a Reducer class. Each dataset has been tested initially with a single reducer, and then with k reducers. The results of our tests are shown below.

In the following version, a Combiner class is added to test improvements in the performance.

Finally, we tryed 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. This transfomation is executed with different number of reduce tasks, to test improvements in the performance.

The results of our tests are shown below.

6 Conclusions