UNIVERSITÀ DI PISA

Cloud Computing
Master Degree in Computer Engineering
year 2019/20

K-Means Clustering in MapReduce

Project for the Cloud Programming lab

Riccardo Polini
Marco Bongiovanni
Simone Pampaloni
Alexander De Roberto

Contents

1 Project Overview

The work, part of the Cloud Computing course, consists in implementing a distributed MapReduce application that implements the well known *K-Means Clustering* algorithm. The implementation must be carried out in *Apache Hadoop* and then in *Apache Spark*, both running on a cluster of 4 virtual machines provided by the University of Pisa.

2 Pseudocode

2.1 Mapper

Algorithm 1: Map Function

input : a point x in d dimensions
output: a key-value pair

kMeansMap(x):

1 load μ vector from distributed file system
2 $c \longleftarrow \text{argmin}_j \|x - \mu_j\|_2{}^2$
3 **emit** $(c, pair(x, 1))$

A set M of centroids is chosen uniformly at random from the input set X and saved in a distributed file system. For each point x in the input set, we call a map function which computes the Euclidean distance of the point from each centroid μ_j and selects the index c of the closest one (2).
Finally we emit a key-value pair {*key*=c, *value*=(x, 1)}.

2.2 Combiner

Algorithm 2: Combine Function

input : a cluster c
a set of key-value pairs
output: a cluster c and point with its counter

kMeansCombine(c, pairs$[(x_1, 1), (x_2, 1), ...]$):

1 sum $\longleftarrow$ 0
2 count $\longleftarrow$ 0
3 **foreach** $pair(x, 1) \in pairs[(x_1, 1), (x_2, 1), ...]$ **do**
4 $\quad sum \longleftarrow sum + x$
5 $\quad count \longleftarrow count + 1$
6 **end**
7 **emit** (c, pair($sum, count$))

Given a key-list of values pairs from the Map stage, we wish to reduce it by combining together each element in the list. To do so, we iterate each value (x_i, 1) in the list of values and we compute the sum of the coordinates x (*sum*) and the number of points in the list (*count*).
The new key-value pair will be composed by the index of the cluster c (key) and a value. The value is composed by the sum of the coordinates (*sum*) and the number of points assigned to this cluster (*count*). The combiner will emit this key-value pair.

2.3 Reducer

Algorithm 3: Reduce Function

input : a cluster c
a set of key-value pairs
output: a set of new centroids

kMeansReduce(c, pairs$[(s_1, c_1), (s_2, c_2), ...]$):

```
1 sum ⟵ 0
2 count ⟵ 0
3 foreach pair(s, c) ∈ pairs[(s_1, c_1), (s_2, c_2), ...] do
4 |   sum ⟵ sum + s
5 |   count ⟵ count + c
6 end
7 μ_c ⟵ sum/count
8 emit (c, μ_c)
```

Given a key-list of values pairs emitted by the combiners, the reduce function wants to find the new centroid μ_c. In order to do so, we iterate each value (s, c) which belongs to the same cluster c and we compute the sum of the coordinates x of the cluster (sum) and the number of points in the cluster ($count$).
Finally the reduce function will compute the new centroid μ_c as the mean value of the coordinates in the cluster c: $sum/count$. The reducer will emit a key-value pair composed by the cluster index c as key and the centroid of the cluster μ_c as value.

3 Hadoop Implementation

For the implementation of Hadoop, the classes describing a point and a centroid were defined.

The Point class represents a point or a set of points added together. It is composed of an array (`double`) of cordinates (of settable size) and an integer (`int`) which represents the number of points that have been added.

The class implements the `Writable` interface because it must be serialized and deserialized to be sent within the Hadoop cluster. For this reason, the `write` and `readFiles` methods have been defined.

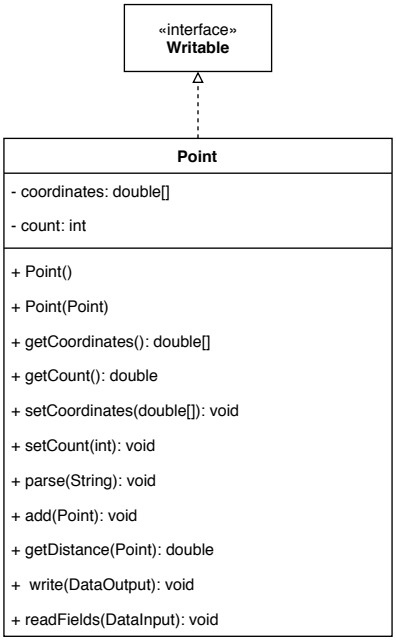


Figure 1: Point class

The Centroid class (which obviously represents a centroid) is composed of an id (`Text`), that uniquely identifies the centroid, and a Point, that specifies its coordinates.
The class implements the `WritableComparable` interface because it must be serialized and deserialized to be sent within the Hadoop cluster and at the same time it should allow the comparation of two Centroid objects.

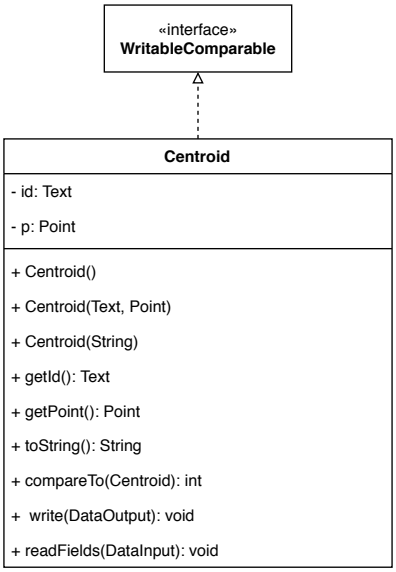


Figure 2: Centroid class

For convenience, a `CentroidList` class has also been defined. It represents the list of centroids necessary for the execution of the algorithm.

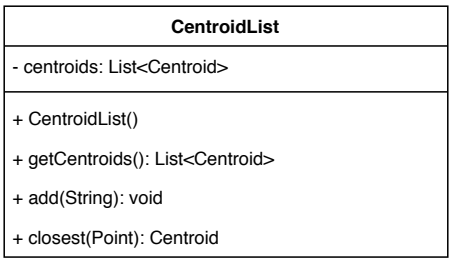


Figure 3: CentroidList class

The mapper was implemented by creating a `KMeansMapper` class that extends the `Mapper<LongWritable, Text, Centroid, Point>` class. Before the map function, a *Point* object is initialized and centroids are loaded from file. The map function takes a point as input (in String format), parses it into a `Point` object, finds the centroid closest to this point and emits the pair `<centroid,point>` via the context object.

Listing 1: Map function

```
1  public class KMeansMapper extends Mapper<LongWritable, Text, Centroid,
       Point> {
2
3      //Function to initialize Point and CentroidList before map
           function
4    @Override
5    protected void setup(Context context) throws IOException {
6      point = new Point();
7      centList = new CentroidList();
8      //retrieve the centroids of the last iteration
9      BufferedReader reader = new BufferedReader(new StringReader(
           context.getConfiguration().get("centroids")));
10     String line = reader.readLine();
11     while(line != null) {
12       centList.add(line);
13       line = reader.readLine();
```

```
    }
  }

   public void map(LongWritable key, Text value, Context context) {
     try {
       //retrieve the input point
       point.parse(value.toString());
       //emit the nearest centroid to the input point
       context.write(centList.closest(point), point);
     } catch (Exception e) {
       e.printStackTrace();
     }
   }
}
```

The combiner was implemented by creating a `KMeansCombiner` class that extends the `Reducer<Centroid, Point, Centroid, Point>` class. The "combine" function takes as input a centroid and a list of points belonging to its cluster. After an iteration on the list of points, the pair represented by the centroid and by a point obtained as the sum of the points of the list, is written in the context.

Listing 2: Combine function

```
public class KMeansCombiner extends Reducer<Centroid, Point, Centroid,
    Point>{

    ...

  protected void reduce(Centroid key, Iterable<Point> val, Context
      context) throws IOException, InterruptedException
  {
    Iterator<Point> it = val.iterator();
    Point pTot = new Point(it.next());

    while(it.hasNext()) {
      Point p = it.next();
      pTot.add(p);
    }
    //pass the data to the reducer
    context.write(key, pTot);
  }
}
```

The reducer was implemented by creating a `KMeansReducer` class that extends the `Reducer<Centroid, Point, Text, NullWritable>` class. The reduce function takes as input a centroid and a list of points belonging to its cluster. As seen before for the combiner, the list of points is summed up obtaining a single point `"pTot"`. Then, the new centroid coordinates are calculated as the mean of the coordinates of the points represented by the `"pTot"` point.

Listing 3: Reduce function

```
public class KMeansReducer extends Reducer<Centroid, Point, Text,
    NullWritable>{

    ...

    protected void reduce(Centroid key, Iterable<Point> values,
        Context context) {
        Iterator<Point> it = values.iterator();

        Point pTot = new Point(it.next());
        while(it.hasNext()){
            Point p = it.next();
            pTot.add(p);
```

```
        }

        //setting new centroid
        //computeMean() computes the new coordinates of the centroid
            as
        //the mean of the coordinates of the points belonging to the
        //centroid's cluster
        newKey.set(new Centroid(key.getId(), computeMean(pTot)).
            toString());

        try {
          //emit the index of the cluster, the new centroid (newKey),
              and a null placeholder
             context.write(newKey, NullWritable.get());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Finally, the `KMeans` class that contains the main function has been defined. First, the configuration was defined as follows:

Listing 4: Configuration Settings

```
    //conf is the Configuration istance
    Job job = new Job(conf, name);
    job.setJarByClass(KMeans.class);
    job.setMapperClass(KMeansMapper.class);
    job.setMapOutputKeyClass(Centroid.class);
    job.setMapOutputValueClass(Point.class);
    job.setCombinerClass(KMeansCombiner.class);
    job.setReducerClass(KMeansReducer.class);

    //INPUT_PATH and OUTPUT_PATH define respectively the path where
    //to find the input and the path where to write the output
    FileInputFormat.addInputPath(job, new Path(INPUT_PATH+"/"+
        inputPointsFile));
    FileSystem.get(conf).delete(new Path(OUTPUT_PATH), true);
    FileOutputFormat.setOutputPath(job, new Path(OUTPUT_PATH));
```

The number of centroids and the names of centroids' and points' input files are passed as arguments to the main function. The main function iteratively executes the MapReduce algorithm until a maximum number of iterations is reached or until the average difference between the new centroids and the ones of the previous iteration goes under a certain threshold (in this case, we consider the algorithm as converged). Obviously, the starting centroids of each new iteration are those obtained as a result from the previous iteration. The code of what is described is shown below

Listing 5: main function

```
 public static void main(String[] args) throws Exception
  {
    final Configuration conf = new Configuration();
    int iter = 0;

    //read centroids from input files (name passed as argument)
    String centroids = readCentroids(conf, INPUT_PATH+"/"+
        inputCentroidsFile);

    String oldCentroids = "";
    double var = 0.0;

        //computeVariation computes the difference between the new
        //centroids and the ones of the previous iteration.
```

```
14      while(iter < MAX_ITER && ((var = computeVariation(oldCentroids,
            centroids)) > THRESHOLD)) {
15        iter++;
16
17        //set list of centroids readable from Mapper,Combiner and
              Reducer
18        conf.set("centroids", centroids);
19
20        final Job job = createJob(conf, "k-means");
21        job.waitForCompletion(true);
22
23        oldCentroids = centroids;
24
25        //read new centroids
26        centroids = readCentroids(conf, OUTPUT_PATH+"/part-r-00000");
27      }
28    }
```

4 Spark Implementation

The KMeans algorithm for Spark has been developed with the following steps.

After initializing the Spark Context and cleaning the output directory, the points are loaded from the HDFS and the centroids are randomly selected from the set of points.

Listing 6: Initialization phase Spark implementation

```
1
2  #inputPath, outputPath, dimension, k, maxIterations, threshold,
3      #seed and master are passed as input argument of the main function
4       ...
5       #initialize context
6    sc = SparkContext(master, "k-Means")
7
8    #Clean output directory from previous execution outputs
9    subprocess.call(["hadoop", "fs", "-rm", "-r", outputPath])
10
11   #Load points from file in HDFS (in string format)
12   pointStrings = sc.textFile(inputPath)
13
14   #parseStrings converts pointString to float np.array
15   #and they are cached on RDD
16   pointsDRR = pointStrings.map(parseStrings).cache()
17
18   #K centroids are selected at random from input start points
19   # and then are send to all workers
20   centroids = np.array(pointsDRR.takeSample(False, k, seed))
21   br_centroids = sc.broadcast(centroids)
```

The MapReduce algorithm is run iteratively. At each iteration, the *map* method is performed, which finds the nearest centroid for each point of the RDD and returns the index of the centroid and the point with a "1" append at the end. The "1" stays for the number of aggregated points.

Then, *reduceByKey* method permits to sum the coordinates of the points having the same cluster index associated obtaining a single summed point and the *mapValues* method permits to divide each coordinate of the previously obtained point by the last element of the point which represents the counter of the points that have been added previously. In this way, the new coordinates of the centroids were calculated.

Listing 7: Iteration Phase Spark Implementation

```
1
2      ...
3
4    iteration = 1
5    delta = float("inf")
6
7    while True:
8      #Perform map-reduce
9      #Get the index of the centroid the point belongs to and
10     #append 1 at the end of the point's coordinates to be used as
           counter (of points) in reduce
11     centroidPointPairs = pointsDRR.map(lambda x: assignToCentroid(x,
           br_centroids.value))
12
13     #sum coordinates of all points belonging to the same cluster
14     #coordinates of summed point divided for the number of points that
            compose the summed point
15     newCentroidsRDD = centroidPointPairs.reduceByKey(lambda x, y: np.
           add(x, y)).mapValues(lambda x: (np.divide(x, x[-1]))[0:-1])
16
17     #Compute centroids' movements
```

```
18      newCentroids = np.array(newCentroidsRDD.sortByKey(ascending=True).
            values().collect())#Build ndarray from a list of key-value
            pairs
19      delta = np.linalg.norm(br_centroids.value - newCentroids, axis=1).
            mean()
20
21      #Broadcast new centroids to all workers
22      br_centroids = sc.broadcast(newCentroids)
23
24      #Check stop conditions
25      print('Delta =', delta, 'obtained at iteration', iteration)
26      if (iteration >= maxIterations) or (delta < threshold):
27        break
28
29      iteration += 1
30
31  #Iterative part ended
32  newCentroidsRDD.saveAsTextFile(outputPath)
```

Finally, the difference *delta* between the coordinates of the centroids of this iteration and the previous one is calculated. If the maximum number of iterations has been reached or *delta* is less than a certain threshold, the algorithm ends with the writing of the final centroids in the output file.

5 Running the Software

5.1 Hadoop

First of all the `jar` file of the Hadoop project must be placed in the home directory of the hadoop-namenode. Then, the HDFS must have a Resources directory containing Input and Output subdirectories. In the Input subdirectory we will put the input files for points and clusters. In the Output subdirectory we will find the output file containing the centroids at the end of the execution.
The input files must be named in the following manner:

- `Resources/Input/points_nxd.txt`, where n is the number of points and d is the dimension.
- `Resources/Input/centroids_nxd.txt`, where n is the number of centroids and d is the dimension.

The *points* file must contain a list of `double` or `float` coordinates separated by a whitespace, one point per line. The *centorids* file must contain the coordinates of the centroids and the unique IDs of the centroids as follows (example of a 3d centroid): `ID. x y z`
The ID must be separated from the coordinates by a tab.
After setting up the HDFS, the application can be launched (from the hadoop-namenode) with the following command:

```
hadoop jar jar_file package_name.mainClass k n d
```

where the parameters are:

- `package_name.mainClass`, in our case, was `it.unipi.hadoop.KMeans`
- `k` is the number of clusters
- `n` is the number of points
- `d` is the dimensionality of points

At the end of the execution the output can be found in `Resources/Output/part-r-00000`.

5.2 Spark

Given the input files configuration explained in the previous section, the Spark program can be launched with the following command:

```
spark-submit python_file -input "Resources/Input/points_nxd.txt"
-output "Resources/Output/Spark_param_test" -dimension d -num_k k
-iterations iter -threshold thr -seed s -master yarn
```

where the parameters are:

- `python_file`, in our case, was `k-means.py`
- `k` is the number of clusters
- `n` is the number of points
- `d` is the dimensionality of points
- `iter` is the maximum number of iterations (10 in our case)
- `thr` is the stopping threshold (0.03 in our case)
- `s` is the seed used in the selection of initial centroids (for repeatability)
- `-master` specifies whether we want to run the program locally (local) or on the cluster of nodes (yarn)

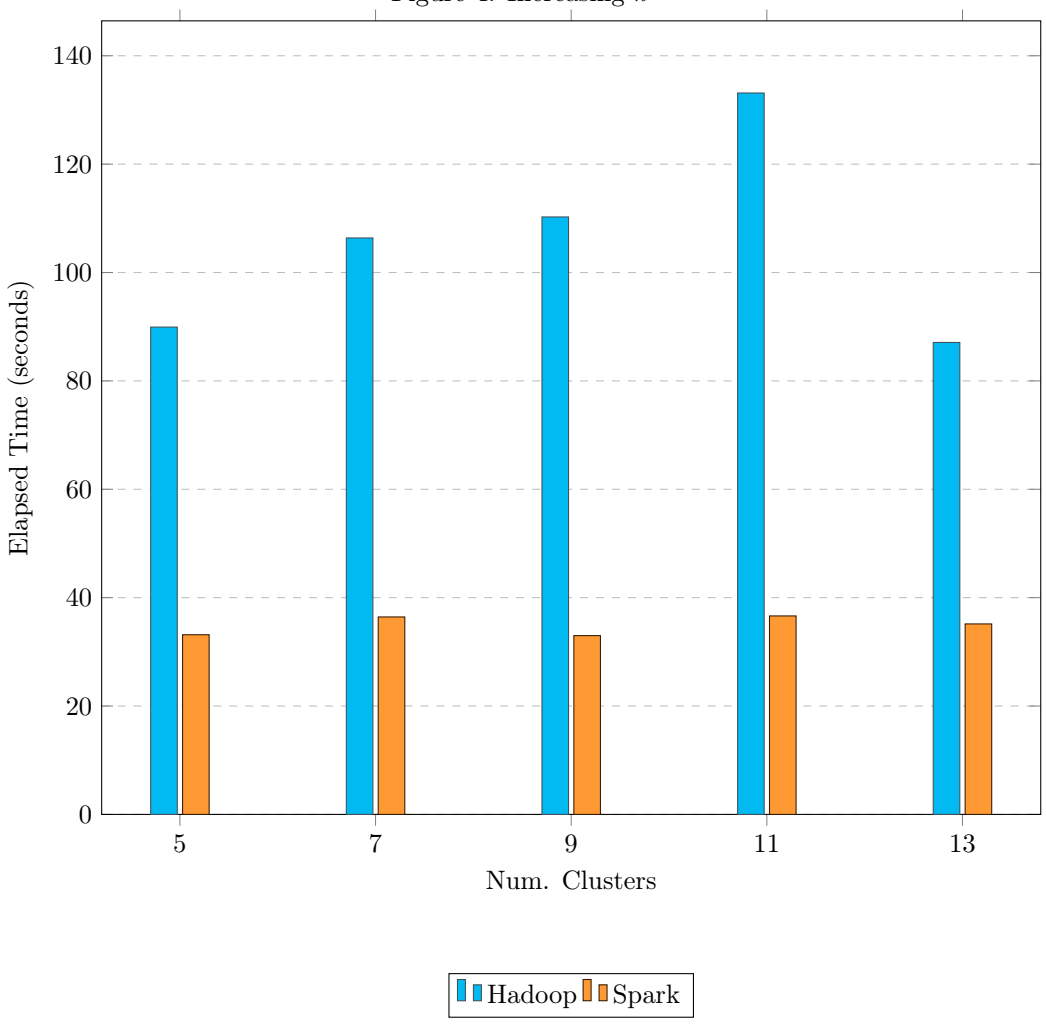
6 Conclusions

The Hadoop and Spark implementations of the algorithm has been run several times by varying, one by one, the following parameters:

- Number of Points n: the number of points that have been given as input to the algorithms range from 10^3 to 10^7 (with a step of an order of magnitude).
- Dimension d: the dimensionality of each point varies from 3 to 7.
- Number of Cluster k: the number of clusters to be identified varies from 5 to 13.
- The elapsed time and the elapsed time per iteration of the algorithm was measured for each run.

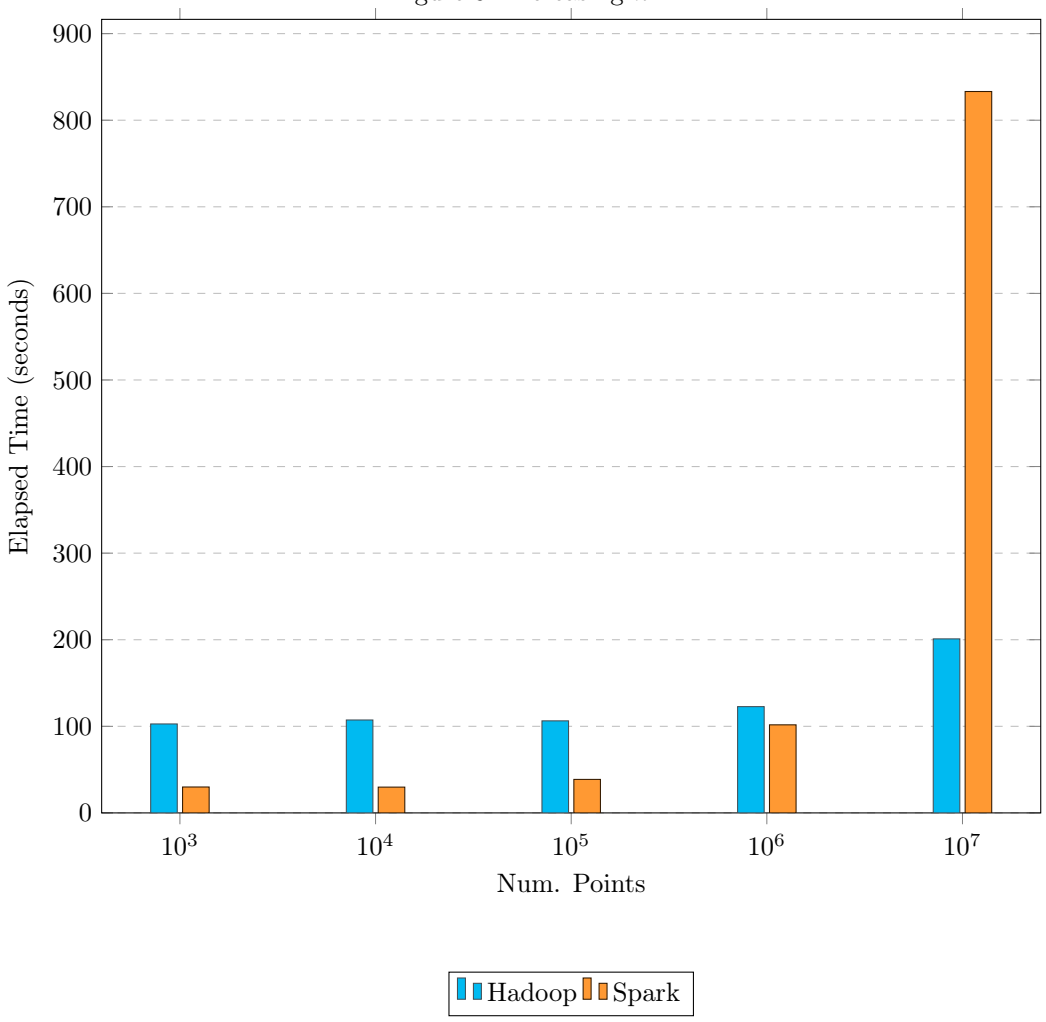
The algorithm was run with `THRESHOLD = 0.03` and `MAX_ITER = 10`. Each experiment was measured with 5 independent repetitions.

Figure 4: Increasing k



The above figure represents a series of simulations in which we had 100000 points in 3 dimensions and an increasing number of clusters. We can see that Spark is significantly faster than Hadoop.

Figure 5: Increasing n



In this case we fixed 7 clusters to find and the dimensionality of the points to 3. We increased each time the number of points to classify starting from 1000 up to 10 million points. We can see that with 10 million points the Spark program spikes in execution time. This happens because Spark runs out of RAM memory for intermediate data and starts swapping in and out of the hard disks introducing a significant overhead. Hadoop, instead, stays relatively constant with a slight increase in execution time in the last case.

Figure 6: Increasing d