UNIVERSITÀ DI PISA

Project report of K-means Algorithm Hadoop and Spark

Dipartimento di Ingegneria dell'Informazione
Cloud Computing Project
2020

F. Ritorti
F. Payes
L. Arena
M. Gomez

July 22, 2020

Contents

Chapter 1

Introduction

1.1 The basic Idea

K Means is one of the most popular "clustering" algorithms. K means stores k centroids that it uses to define clusters. A point is considered to be in particular cluster if it is closer to that cluster's centroid than any other centroid.

K Means finds the best centroids by alternating between (1) assigning data points to clusters based on the current centroids (2) choosing centroids (points which are the center of a cluster) based on the current assignment of data points to clusters.

Chapter 2

Design

2.1 Dataset

We use a Kaggle Weather Dataset that contains weather data captured for a one-minute interval. This data comes from a weather station located in San Diego, California. The weather station is equipped with sensors that capture weather-related measurements such as air temperature, air pressure, and relative humidity. Data was collected for a period of three years, from September 2011 to September 2014, to ensure that sufficient data for different seasons and weather conditions is captured.

This weather dataset contains more than one million and a half of registers with thirteen fields that consists of the following variables:

General:

1. **rowID:** Unique ID number for each row (Unit: Numeric).
2. **hpwren_timestamp:** Timestamp of measure (Unit: year-month-day hour:minute:second).

Air:

3. **air_pressure:** Air Pressure at the timestamp (Unit: hectopascals)
4. **air_temp:** Air Temperature at the timestamp (Unit: degrees Fahrenheit)

Wind:

5. **avg_wind_direction:** Wind Direction Average over the minute before the timestamp (Unit: degrees, with 0 means coming from the North, and increasing clockwise)
6. **max_wind_direction:** Highest Velocity Wind Direction (Unit: degrees, with 0 being North and increasing clockwise)
7. **min_wind_direction:** Smallest Velocity Wind Direction (Unit: degrees, with 0 being North and increasing clockwise)

8. **avg_wind_speed:** Wind Speed Average over the minute before the timestamp (Unit: meters per second)
9. **max_wind_speed:** Highest Velocity Wind Speed (Unit: meters per second)
10. **min_wind_speed:** Smallest Velocity Wind Speed (Unit: meters per second)

Rain:

11. **rain_accumulation:** Amount of Accumulated Rain measured at the timestamp (Unit: millimeters)
12. **rain_duration:** Rain Duration (Unit: seconds)

Humidity:

13. **relative_humidity:** Relative Humidity measure at the timestamp (Unit: percent)

2.2 Preprocessing

For the Preprocessing of the data, we use the Tool Weka. The data that is collected from the field contains many unwanted things that leads to wrong analysis. We decide to remove the rows:

- **rowID:** no usefull for the analysis evaluation
- **hpwren_timestamp:** no numeric value

The second step for clean the dataset were the remove of the rows with the number of the missing values on zero (*rain_accumulation and rain_duration*). Then, we normalize all numeric values in the given dataset ignoring the nominal class (*ignoreclass = true*) in the default range [0, 1].

We will use the preprocessing weather data result file with nine fields:

Air:

1. **air_pressure**
2. **air_temp**

Wind:

3. **avg_wind_direction**
4. **max_wind_direction**
5. **min_wind_direction**

6. **avg_wind_speed**

7. **max_wind_speed**

8. **min_wind_speed**

Humidity:

9. **relative_humidity**

For the respective file text, we create three different files with the first *1.000 observations* (**point_1k.txt**), the first *10.000 observations* (**point_10k.txt**) and *100.000 observations* (**point_100k.txt**).

2.3 Input Data

Input Data	
Variables	**Description**
Dataset	File name of the collection of data
k	Total number of dimensions
d	Distance function
n	Number of Observations
Threshold	Value for flexibility of the convergence

2.3.1 Change to the memory

To run our algorithms, we have used more virtual memory than our current limit of 2.1 GB and so we made two changes in the *yarn-site.xml* file:

- Disable virtual memory limit checking

- Increase virtual memory to physical memory ratio

2.3.2 Environment

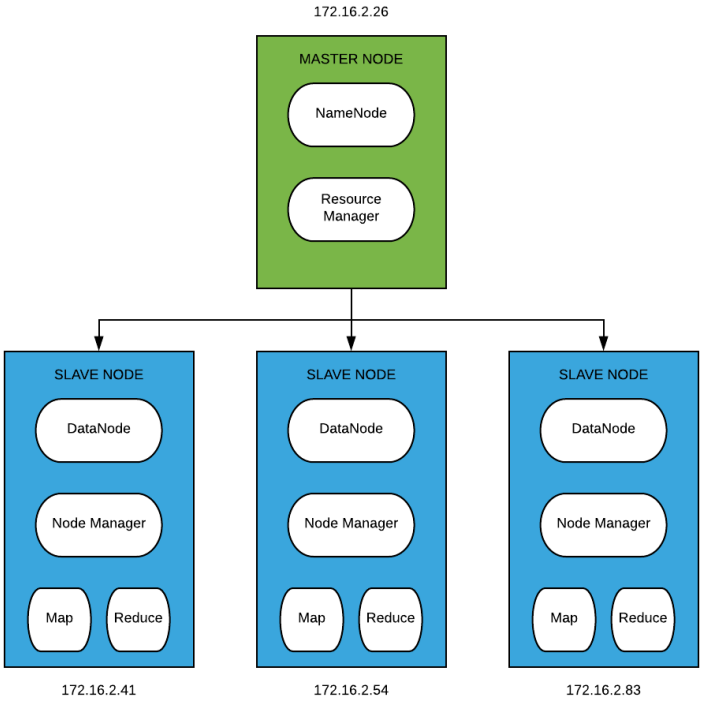


Figure 2.1: Hadoop Diagram

Chapter 3

Implementation

3.1 Pseudocode

Algorithm 1: $kMeans(dataset, k, dimension, threshold, centroids, output)$

Result: The k-means algorithm using distributed computation
input :
dataset: Dataset File name with the complete path;
k: Number of Clusters;
dimension: Number of coordinates to work with;
threshold: Value for flexibility of the convergence;
centroids: Centroids File Name with the complete path;
output: Output File Name to save the clusters;
output: The set of mean *centroids* that make the algorithm converged

```
parse the input arguments;
convergedCentroids ← 0;
iterations ← 0;
centroids ← set the initial random centroids from the whole dataset;
while convergedCentroids < k do
    if iteration > 0 then
        centroids ← previous mean centroids;
    end
    MAP every point and assign the closest centroid to it
    REDUCE to find the mean centroid of each cluster
    convergedCentroids ← count of mean centroids that converged;
    iteration ← iteration + 1;
end
return convergedCentroids;
```

Algorithm 2: $MAP(line)$

Result: Map every point and assign the closest centroid to it
input :
line: one row on the Dataset text file;

```
point ← new empty array;
coordinates: split line string by ",";
coordinateCounter ← 0 ;
foreach coordinate in coordinates do
    if coordinateCounter == dimension then
        break;
    end
    value: convert coordinate to double;
```

3.2 K-means in Hadoop

3.2.1 How to select random centroids in Hadoop?

To choose initial random centroids, we used a MapReduce solution that exploits the sorter functionalities.

Initially, the mapper receives as input a text line and outputs each line with a random integer as key.

Algorithm 4: $MAP(line)$

Result: Map every point and assign the closest centroid to it
input :
line: one row on the Dataset text file;

```
1  id: set random number;
2  EMIT(id, line);
```

Algorithm 5: $REDUCE(id, values)$

Result: Reduce to find the mean centroid of each cluster
input :
id: is the id of the current centroid;
value: is an array of points assigned to the centroid ;

```
1  parse the input arguments;
2  centroidsList: set empy list;
3  setup:
4  dimension, k ← get values from configuration;
5  reduce(id, values);
6  foreach element in values do
7  |  point ← get a slice from element's array of the same size of
   |    dimension;
8  |  add point to centroidsList;
9  |  EMIT(null, point);
10 |  if centroidsList size == k then
11 |  |  break;
12 |  end
13 end
14 cleanup:
15 centroidsFile: get value from configuration;
16 foreach element in centroidsList do
17 |  write element in centroidsFile;
18 end
```

Finally, the reducer outputs the first K values, throwing away the keys.

3.2.2 Job

The steps below explains how a MapReduce job is created for processing a kmeans iteration:

```
107     while (convergedCentroids < k) {
108       if (fs.exists(output)) {
109         fs.delete(output, true);
110       }
111
112       Job job = Job.getInstance(conf, "Kmean JAVA");
113
114       job.getConfiguration().set("k", otherArgs[1]);
115       job.getConfiguration().set("dimension", otherArgs[2]);
116       job.getConfiguration().set("threshold", otherArgs[3]);
117       job.getConfiguration().set("centroidsFilename", otherArgs[4]);
118
119       job.setInputFormatClass(TextInputFormat.class);
120       job.setOutputFormatClass(TextOutputFormat.class);
121
122       job.setJarByClass(Kmeans.class);
123       job.setMapperClass(KMeansMapper.class);
124       job.setReducerClass(KMeansReducer.class);
125       job.setMapOutputKeyClass(Centroid.class);
126       job.setMapOutputValueClass(Point.class);
127       job.setOutputKeyClass(Text.class);
128       job.setOutputValueClass(Text.class);
129
130       FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
131       FileOutputFormat.setOutputPath(job, new Path(otherArgs[5]));
132
133       job.waitForCompletion(true);
134
135       convergedCentroids = job.getCounters().findCounter(KMeansReducer.Counter.CONVERGED_COUNT).getValue();
136       iterations++;
137     }
138
```

Figure 3.1: Kmeans iteration

1. **From line 114 to 117:** we pass the required configuration to the job.

2. **From line 119 to 131:** we configure various job-specific parameters.

3. **Line 133:** Submits the job, then polls for progress until the job is complete.

4. **Line 135:** is very important to evaluate the continuity of the process. If the number of converged centroids is less than k, all the job will be started again.

3.2.3 Map

At the beginning of the task we use the setup function as a preliminary phase where we retrieve the previous centroids in a file.

```java
public class KMeansMapper extends Mapper<Object, Text, Centroid, Point> {
  private final List<Centroid> centroids = new ArrayList<>();
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  private int configurationDimension;
  private final Point point = new Point();

  @Override
  protected void setup(Context context) throws IOException, InterruptedException {
      Configuration conf = context.getConfiguration();
      Path centersPath = new Path(conf.get("centroidsFilename"));
      SequenceFile.Reader reader = new SequenceFile.Reader(conf, SequenceFile.Reader.file(centersPath));
      IntWritable key = new IntWritable();
      Centroid value = new Centroid();
      configurationDimension = Integer.parseInt(conf.get("dimension"));

      while (reader.next(key, value)) {
          Centroid c = new Centroid(key, value.getCoordinates());

          centroids.add(c);
      }

      reader.close();
```

Figure 3.2: KmeansMapper function in Hadoop

The filename where the centroids are located is taken from the centroids-Filename variable in the configuration passed to the job.

```java
40      @Override
41      public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
42          StringTokenizer itr = new StringTokenizer(value.toString(), ",");
43          List<DoubleWritable> pointsList = new ArrayList<DoubleWritable>();
44          int count = 0;
45
46          while (itr.hasMoreTokens()) {
47              word.set(itr.nextToken());
48              Double coordinate = Double.valueOf(word.toString());
49              pointsList.add(new DoubleWritable(coordinate));
50              count = count + 1;
51
52              if (count == configurationDimension) {
53                  break;
54              }
55          }
56          point.setCoordinates(pointsList);
57
58          Centroid closestCentroid = null;
59          Double minimumDistance = Double.MAX_VALUE;
60          for (Centroid c1 : centroids) {
61              Double distance = c1.findEuclideanDistance(point);
62
63              if (distance < minimumDistance) {
64                  minimumDistance = distance;
65                  closestCentroid = Centroid.copy(c1);
66              }
67          }
68
69          context.write(closestCentroid, point);
70      }
```

Figure 3.3: Map function in Hadoop

1. **From line 42 to line 56:** we splits a text line read from the file text to

create a list, which is later used as coordinates for a point Object.

2. **From line 58 to line 69:** we compare the euclidean distance between the point and each centroid to find the closest centroid.

3.2.4 Reduce

The setup function *(line 26)* in the reducer, similarly to the mapper function, preprocesses data needed for the following steps with the difference that it retrieves the dimension and the threshold from the Configuration.

```
16  public class KMeansReducer extends Reducer<Centroid, Point, NullWritable, Text> {
17      private Text result = new Text("");
18      private static int dimension;
19      private static double threshold;
20      private final List<Centroid> centroids = new ArrayList<Centroid>();
21      public static enum Counter {
22          CONVERGED_COUNT
23      }
24
25      @Override
26      protected void setup(Context context) throws IOException, InterruptedException {
27          Configuration conf = context.getConfiguration();
28
29          dimension = Integer.parseInt(conf.get("dimension"));
30          threshold = Double.parseDouble(conf.get("threshold"));
31      }
32
```

Figure 3.4: KmeansReducer function in Hadoop

```
32
33      @Override
34      public void reduce(Centroid key, Iterable<Point> values, Context context) throws IOException, InterruptedException {
35          Centroid meanCentroid = new Centroid(dimension);
36          long numElements = 0;
37
38          for (Point currentPoint : values) {
39              meanCentroid.add(currentPoint);
40              numElements++;
41          }
42
43          meanCentroid.setId(key.getId());
44          meanCentroid.calculateMean(numElements);
45
46          Centroid copy = Centroid.copy(meanCentroid);
47          centroids.add(copy);
48          Double distance = key.findEuclideanDistance((Point) meanCentroid);
49
50          result.set(meanCentroid.toString() + " - " + copy.toString() + " - Distance: " + distance);
51          context.write(null, result);
52
53          if (distance <= threshold) {
54              context.getCounter(Counter.CONVERGED_COUNT).increment(1);
55          }
56      }
```

Figure 3.5: Reduce function in Hadoop

1. **From line 35 to line 44:** we calculate the mean centroid using the points received in the list of values of the reducer.

2. **From line 46 to line 55:** in the second part of the code, we calculate the distance between the mean centroid and the current centroid to check if the convergence respects the threshold. Therefore, if the condition of the convergence is respected we will increment the counter of the convergence.

The following code explains the cleanup function, where the program loops the mean centroids and writes them into a file for the next iteration.

```
58      @Override
59      protected void cleanup(Context context) throws IOException, InterruptedException {
60          super.cleanup(context);
61
62          Configuration conf = context.getConfiguration();
63          Path centersPath = new Path(conf.get("centroidsFilename"));
64          FileSystem fs = FileSystem.get(conf);
65
66          if (fs.exists(centersPath)) {
67              System.out.println("Delete old output folder: " + centersPath.toString());
68              fs.delete(centersPath, true);
69          }
70
71          SequenceFile.Writer centroidWriter = SequenceFile.createWriter(conf,
72                  SequenceFile.Writer.file(centersPath),
73                  SequenceFile.Writer.keyClass(IntWritable.class),
74                  SequenceFile.Writer.valueClass(Centroid.class));
75
76          for (Centroid c : centroids) {
77              centroidWriter.append(c.getId(), c);
78          }
79
80          centroidWriter.close();
81      }
```

Figure 3.6: Cleanup function

3.3 K-means in Spark

3.3.1 Spark Driver

In **line 60** we initialize the Spark Context that represents a connection with the **master** system. At the master argument we specifying *ÿarn¨* because we want to connect to the yarn cluster. In the **lines 62 to line 64** we parse the argument values.

```
54  if __name__ == "__main__":
55      if len(sys.argv) < 5:
56          print("Usage: kmeans <input file> <k> <dimension> <threshold> [<centersFilename>]", file=sys.stderr)
57          sys.exit(-1)
58
59      master = "yarn"
60      sc = SparkContext(master, "Kmeans SPARK")
61
62      k = int(sys.argv[2])
63      dimension = int(sys.argv[3])
64      threshold = float(sys.argv[4])
65      lines = sc.textFile(sys.argv[1])
66      convergence_count = 0
67      iteration_count = 0
68      points = lines.map(lambda x: parsePoint(x, dimension))
69      random_centroids = []
70
71      while (convergence_count < k):
72          convergence_count = 0
73
74          if iteration_count == 0:
75              random_centroids = points.takeSample(False, k)
76          else:
77              random_centroids = [mean_centroid[1] for mean_centroid in mean_centroids]
78
79          clusters_points = points.map(lambda x: assign_nearest_centroid(x, random_centroids))
80          clusters = clusters_points.groupByKey()
81          mean_centroids = clusters.map(lambda cluster: get_mean_centroids(cluster)).collect()
82
83          for index in range(len(mean_centroids)):
84              mean_centroid_index = mean_centroids[index][0]
85              mean_centroid_points = mean_centroids[index][1]
86
87              distance = calculate_euclidean_distance(mean_centroid_points, random_centroids[mean_centroid_index])
88
89              if (distance <= threshold):
90                  convergence_count+=1
91
92          iteration_count+=1
93
```

Figure 3.7: Kmeans Spark

In the last section of the program (lines 71-92) we run the job and evaluate if we need another iteration. In line 77 we replaced the centroid with the mean previous one.

3.3.2 How to select random centroids in Spark?

```
84      while (convergence_count < k):
85          convergence_count = 0
86
87          if iteration_count == 0:
88              random_centroids = points.takeSample(False, k)
89
```

Figure 3.8: Kmeans Spark

In the command line 88 above we want to create a subset of RDD with fixed-size. The code has two parameters:

- The first parameter specifies if we want to use replacement, In this case not, because we want unrepeated items.
- The second parameter is the number of elements that we want, in our case k represents the number of initial centroids needed.

3.3.3 Map

The Map function in Spark works in a different way. In the below picture (line 108), We mapped the RDD of points taken from the file and called function assign_nearest_centroid to each point.

```
99
100         print("======================")
101         print("RANDOM CENTROIDS::")
102         print("======================")
103         print(random_centroids)
104         print("======================")
105     else:
106         random_centroids = [mean_centroid[1] for mean_centroid in mean_centroids]
107
108     clusters_points = points.map(lambda x: assign_nearest_centroid(x, random_centroids))
109     clusters = clusters_points.groupByKey()
110     mean_centroids = clusters.map(lambda cluster: get_mean_centroids(cluster)).collect()
111
```

Figure 3.9: Map function in Spark

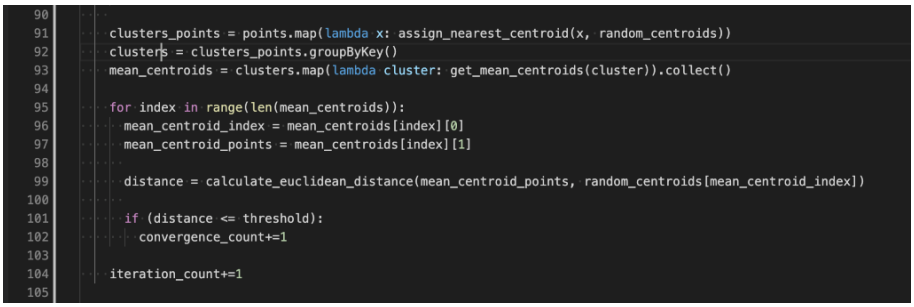
The following function is the same process as line 58 to line 69 in the java solution.

```
22  def assign_nearest_centroid(point, centroids):
23      centroid_index = -1
24      minimum_distance = float('inf')
25
26      for index in range(len(centroids)):
27          distance = calculate_euclidean_distance(centroids[index], point)
28
29          if (distance < minimum_distance):
30              centroid_index = index
31              minimum_distance = distance
32
33      return (centroid_index, point)
34
```

Figure 3.10: Nearest Centroid function in Spark

3.3.4 Reduce

The reduce function is in the line 92 and basically it groups the points by the key values, which are the nearest cluster.



```
90
91      clusters_points = points.map(lambda x: assign_nearest_centroid(x, random_centroids))
92      clusters = clusters_points.groupByKey()
93      mean_centroids = clusters.map(lambda cluster: get_mean_centroids(cluster)).collect()
94
95      for index in range(len(mean_centroids)):
96          mean_centroid_index = mean_centroids[index][0]
97          mean_centroid_points = mean_centroids[index][1]
98
99          distance = calculate_euclidean_distance(mean_centroid_points, random_centroids[mean_centroid_index])
100
101         if (distance <= threshold):
102             convergence_count+=1
103
104     iteration_count+=1
105
```

Figure 3.11: Reduce function in Spark

In the line 93 we process the mean centroid of each cluster using the map function.

Eventually, we compute the euclidian distance between the mean centroid and the previous centroid using a threshold. The condition of the convergence happens when the distance is less or equal of the threshold. If the condition is true the process continues to compute the distance by adding a unit to the convergence counter. Afterwards the process continue iteratively (iteration_count+=1).

Chapter 4

Experimental Results

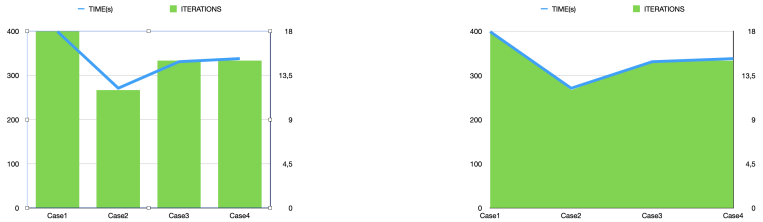
Before to apply the k-means algorithms to our data, we have to fix the first centroids. They could be chosen in two different ways:

- randomly way
- prefixed way

To understand better how **HADOOP** and **SPARK** works, we analyze them first, in a singular way choosing randomly centroids and then we will do a comparison between them choosing prefixed centroids which are the same for both the configuration.

4.1 Hadoop

4.1.1 MapReduce with 1.000 points



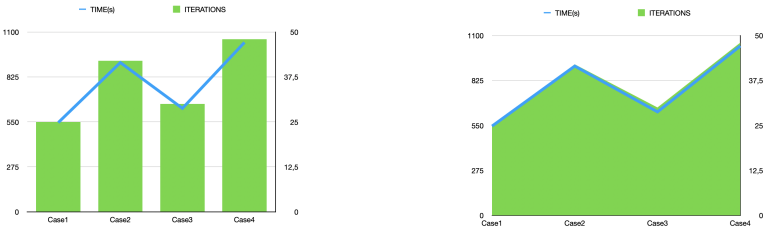
(a) (b)

points-1k.txt

	ITERATIONS	TIME(s)
Case1	18	399
Case2	12	271
Case3	15	331
Case4	15	338

Figure 4.1: **Case1:** d=3, k=7 **Case2:** d=3, k=13 **Case3:** d=7, k=7 **Case4:** d=7, k=13

4.1.2 MapReduce with 10.000 points



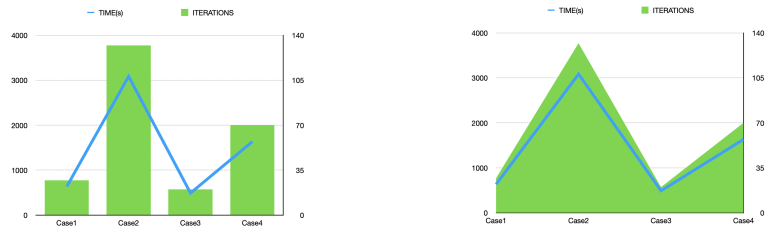
(a) (b)

points-10k.txt

	ITERATIONS	TIME(s)
Case1	25	546
Case2	42	913
Case3	30	633
Case4	48	1036

Figure 4.2: **Case1:** d=3, k=7 **Case2:** d=3, k=13 **Case3:** d=7, k=7 **Case4:** d=7, k=13

4.1.3 MapReduce with 100.000 points



(a) (b)

points-100k.txt

	ITERATIONS	TIME(s)
Case1	27	635
Case2	132	3084
Case3	20	490
Case4	70	1633

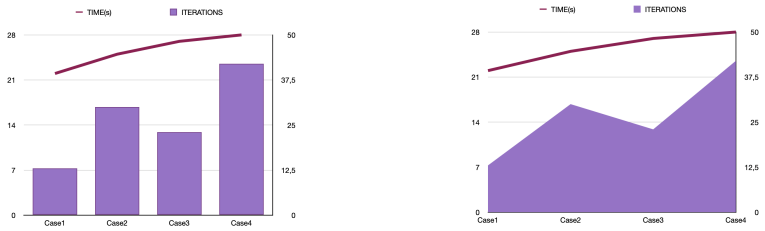
Figure 4.3: **Case1:** d=3, k=7 **Case2:** d=3, k=13 **Case3:** d=7, k=7 **Case4:** d=7, k=13

4.1.4 Results on Hadoop

These are our experimental results of the Hadoop Map reduce where we have picked our centroids in a randomly way. As we can see, the times and the iterations are like linearly independent. In fact, they present the same gait: if the iterations grow, time also increase. We could analyze better the last case where we have the maximum dataset. In the last scenario time grows as soon as the number of iterations increase but, the time (line in blue) is below the iterations (area in green) due to the fact that Hadoop works better with a huge amount of data.

4.2 Spark

4.2.1 MapReduce with 1.000 points



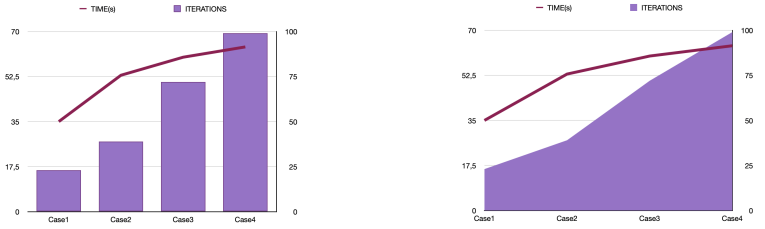
(a) (b)

points-1k.txt

	ITERATIONS	TIME(s)
Case1	13	22
Case2	30	25
Case3	23	27
Case4	42	28

Figure 4.4: **Case1:** d=3, k=7 **Case2:** d=3, k=13 **Case3:** d=7, k=7 **Case4:** d=7, k=13

4.2.2 MapReduce with 10.000 points



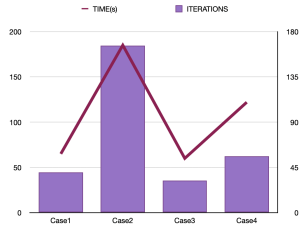
(a) (b)

points-10k.txt

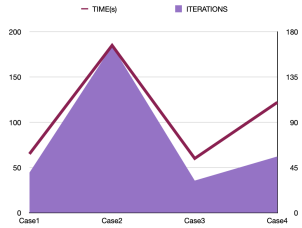
	ITERATIONS	TIME(s)
Case1	23	35
Case2	39	53
Case3	72	60
Case4	99	64

Figure 4.5: **Case1:** d=3, k=7 **Case2:** d=3, k=13 **Case3:** d=7, k=7 **Case4:** d=7, k=13

4.2.3 MapReduce with 100.000 points



(a)



(b)

points-100k.txt

	ITERATIONS	TIME(s)
Case1	40	65
Case2	166	185
Case3	32	60
Case4	56	122

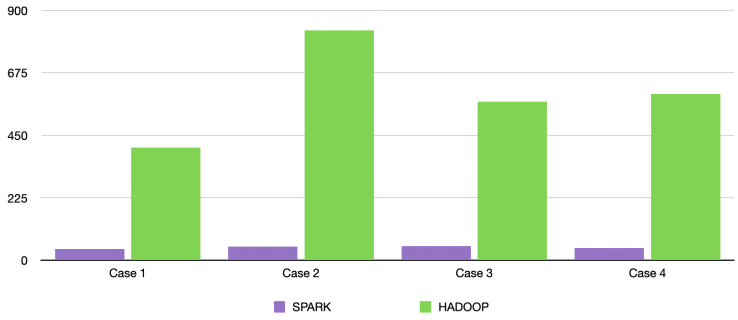
Figure 4.6: **Case1:** d=3, k=7 **Case2:** d=3, k=13 **Case3:** d=7, k=7 **Case4:** d=7, k=13

4.2.4 Results on Spark

These are our experimental results of the Spark Map reduce where we have picked our centroids in a randomly way. As we can see, the time grows in base of the number of iterations but in a constant way for the first two case because in this case the algorithm is too faster. Instead in the third case the time and the iterations present the same gait: if the iterations grow, time also increase and the time (line in brown) is above the iterations (area in purple), hence Spark works better with a small amount of data.

4.3 Comparison of Spark and Hadoop

4.3.1 Time in seconds comparison with 1.000 points

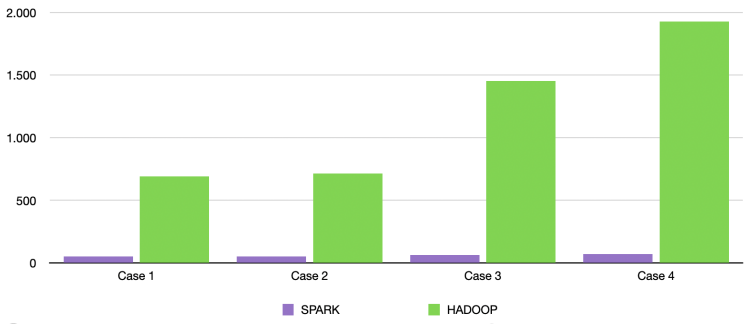


points-1k.txt

	SPARK	HADOOP
Case 1	41	406
Case 2	49	827
Case 3	50	571
Case 4	44	599

Figure 4.7: **Case1:** d=3, k=7 **Case2:** d=3, k=13 **Case3:** d=7, k=7 **Case4:** d=7, k=13

4.3.2 Time in seconds comparison with 10.000 points



points-10k.txt

	SPARK	HADOOP
Case 1	50	692
Case 2	52	713
Case 3	64	1.453
Case 4	69	1.928

Figure 4.8: **Case1:** d=3, k=7 **Case2:** d=3, k=13 **Case3:** d=7, k=7 **Case4:** d=7, k=13

4.3.3 Time in seconds comparison with 100.000 points

points-100k.txt

	SPARK	HADOOP
Case 1	69	1.128
Case 2	67	728
Case 3	545	1.018
Case 4	122	1.129

Figure 4.9: **Case1:** d=3, k=7 **Case2:** d=3, k=13 **Case3:** d=7, k=7 **Case4:** d=7, k=13

4.3.4 Results of the comparison of Hadoop and Spark

To make a time comparison between Hadoop and Spark we need to fix the initial centroids to guarantee tests in same conditions; in fact, both the algorithms converge with the same number of iterations but different times.

In all three cases we could notice that the spark algorithm takes less time than Hadoop to converge and so it could be consider better in terms of time; but, if we analyze better the situation in which we have 100.000 points, in the case 3, Spark algorithms takes more time than the other cases and it's because we have more iterations than others and that Spark does harder work with a huge amount of data.

Chapter 5

Conclusions

5.1 Key Difference between Hadoop MapReduce and Spark

The key difference between Hadoop MapReduce and Spark lies in the approach to processing: Spark can do it in-memory, while Hadoop MapReduce must read from and write to a disk. As a result, the speed of processing differs significantly – Spark may be up to 100 times faster. However, the volume of data processed also differs Hadoop MapReduce can work with far larger datasets than Spark.

The results clearly showed that the performance of Spark turn out to be considerably higher in terms of time, where each of the dataset size results in a decrease in the processing time as compared to that of Map Reduce.