University of Pisa
MSc in Computer Engineering
Cloud Computing

KMeans Algorithm implementation with MapReduce Framework in Hadoop and Spark

Daniela Comola
Eugenia Petrangeli
Gerardo Alvaro
Leonardo Fontanelli

Academic Year 2019/2020

Contents

1 Introduction

In our project we have implemented the K-means algorithm using the Map Reduce framework. The K-Means is one of the simplest unsupervised learning algorithms that solve the clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The main idea is to define k centroids, one for each cluster. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more. Finally, this algorithm aims at minimizing an objective function, in this case a squared error function. The objective function

$$f(M) = \sum_{x \in X} \min_{\mu \in M} \|x - \mu\|_2^2$$

where is a chosen distance measure between a data point and the cluster centre, is an indicator of the distance of the n data points from their respective cluster centres. X is the set of n data points, each with dimension d and M is the set of k centroids.

2 Design

In our project the file containing all the samples and the centroids (randomly selected from the samples) are generated previously. In this way we can perform the performance tests for *Hadoop* and *Spark* taking as input the same data.
We use the *MapReduce* framework in order to associate each point to the nearest cluster and compute the new centroids that are the mean of the list of points associate to them. Going into more detail of each task:

- **Mapper**: assigns each point to the nearest centroid using the *Euclidean distance*. Takes as input the list of points and emits a key-value pair <`Centroid,Point`>.
- **Combiner**: calculates the partial sum of the list of points associated with a cluster and save the number of points currently summed. This operation is useful to limit traffic in the network because instead of emitting all points related to a cluster it emits at most k partial sums. Emits a key-value pair <`Centroid,Point`>
- **Reducer**: after the *shuffle&sort* receives lists of points belonging to the same cluster. It sums all the points and divides the result by the cardinality, considering also the number of points summed in the Combiner. It emits a key-value pair <`id centroid, Average`>.

Algorithm 1, Algorithm 2 and Algorithm 3 in the following show the Mapper and Reducer pseudocode.

```
class Mapper
    method INITIALIZE()
        k ← #clusters
        centroids ← {$\mu_1$,$\mu_2$, ..., $\mu_k$}
    method MAP(doc d)
        for all point $p_i \in$ doc d do
            key ← min_j distance($p_i$,$\mu_j$)
        EMIT(key,$p_i$)
```

Listing 1: Algorithm 1

At the initialization it gets the currents means centroids = μ_1,μ_2, ..., μ_k and the number of clusters.

```
class Combiner
    method REDUCE(key key, points[$p_1, p_2, ...$])
        for all point $p_i \in$ points[$p_1, p_2, ...$] do
            pointSum ← sum(pointSum,$p_i$)
        EMIT(key,pointSum)
```

Listing 2: Algorithm 2

The method "sum" puts in pointSum the partial sum related to a cluster and also the number of points summed up.

```
class Reducer
    method REDUCE(key key, points[$p_1, p_2, ...$])
```

```
        point ← $p_1$
        for all point $p_{i+1} \in$ points[$p_1, p_2, ...$] do
            point ← sum(point,$p_{i+1}$)
        point ← avg(point)
        EMIT(key,point)
```

Listing 3: Algorithm 3

”point” contains the sum of all the partial sums and also the total number of points summed up. Those information are used by the method ”avg” in order to calculate the mean and it will be the new centroid. The Reducer will emit also the key of the new centroid in order to let the Driver to compute the stop condition.
We can have multiple reducers and the best choice is to have k reducers, in this way we'll have one reducer per cluster.
The Driver compute the two stop conditions:

- Number of iterations
- The difference between the old and the new centroids, obtained as Reducer output. If this value is below a certain threshold the algorithm can stop.

3 Hadoop

3.1 Class Point

This class implements *Writable* because it is given as output in the *Mapper* and,therefore, as input to the *Reducer*. It is used also as input and output in the *Combiner*.
This class represents the point object with its coordinates and the number of points summed up in the *Combiner* and *Reducer*.

`Point()` Instantiates a point. *occurrences* default value is one because at first the coordinates aren't summed up.

```java
public Point() {
    coords = new ArrayList<Double>();
    occurrences = 1;
}
```

Listing 4: `Point()`

`Point(String s)` Instantiates a point passing a *String*, which represents the coordinates of the point.

```java
public Point(String s){
    this();
    String[] c = s.split(" ");
    for (String x : c) {
        coords.add(Double.parseDouble(x));
    }
}
```

Listing 5: `Point(Strings)`

`Point(Point p)` Instantiates a point passing another *Point* object.

```java
public Point(Point p){
    this();
    this.coords.addAll(p.coords);
    this.occurrences = p.occurrences;
}
```

Listing 6: `Point(Point p)`

`Point(double[] array)` Instantiates a point passing an array of double, which represents the coordinates of the point.

```java
public Point(double[] array){
    this();
    for (int i = 0; i < array.length; i++)
        coords.add(array[i]);
}
```

Listing 7: `Point(double[] array)`

`set(String s)` Sets the coordinates of the point.

```java
public void set(String s){
    String[] c = s.split(" ");
    coords.clear();
    for (String x : c) {
        coords.add(Double.parseDouble(x));
    }
}
```

Listing 8: `set(String s)`

`set(Point p)` Sets the coordinates and the occurrences of the point.

```java
public void set(Point p){
    this.coords = p.coords;
    this.occurrences = p.occurrences;
}
```

Listing 9: `set(Point p)`

`getNumber()` Gets the *occurrences* value.

```java
public int getNumber(){
    return occurrences;
}
```

Listing 10: `getNumber()`

`write(DataOutput out)` Serializes the *Point* object.

```java
public void write(DataOutput out) throws IOException {
    out.writeInt(coords.size());
    for(int i = 0; i < coords.size(); i++)
        out.writeDouble(coords.get(i));
    out.writeInt(occurrences);
}
```

Listing 11: `write(DataOutput out)`

`readFields(DataInput in)` Deserializes the *Point* object.

```java
public void readFields(DataInput in) throws IOException {
    int size = in.readInt();
    coords = new ArrayList<Double>();
    for (int i = 0; i < size; i++){
        double e = in.readDouble();
        coords.add(e);
    }
    occurrences = in.readInt();
}
```

Listing 12: `readFields(DataInput in)`

`add(Point p)` Sums up the coordinates of two points and increments the number of occurrences.

```java
public void add(Point p){
    for (int i = 0; i < this.coords.size(); i++){
        this.coords.set(i, this.coords.get(i) + p.coords.get(i));
        System.out.print(this.coords.get(i) + " ");
    }
    this.occurrences += p.occurrences;
}
```

Listing 13: `add(Point p)`

`avg()` Computes the mean value of the coordinates.

```java
public void avg(){
    for(int i = 0; i < this.coords.size(); i++)
        this.coords.set(i, this.coords.get(i) / this.occurrences);
}
```

Listing 14: `avg()`

`getDistance(Point p1)` Computes the Euclidean distance between two points.

```java
public double getDistance(Point p1){
    double sum = 0.0;
    for (int i = 0; i < p1.coords.size(); i++)
        sum += Math.pow(p1.coords.get(i) - this.coords.get(i), 2);
    return Math.sqrt(sum);
}
```

Listing 15: `getDistance(Point p1)`

`toString()` Redefines how to print the *Point* object.

```java
public String toString(){
    String temp = "";
    for (int i = 0; i < coords.size(); i++)
        temp += coords.get(i) + " ";
    return temp;
}
```

Listing 16: `toString()`

3.2 Class Centroid

This class implements *WritableComparable* because it is used as key in the Mapper,Reducer and Combiner classes. This class represents the centroid object with its coordinates and its id. The id is used in order to identify the centroid object.

`Centroid()` Instantiates a centroid.

```java
public Centroid() {
    id = new IntWritable(0);
    point = new Point();
}
```

Listing 17: `Centroid()`

`Centroid(int id, String s)` Instantiates a centroid passing an id and a String, this last one represents the coordinates of this object.

```java
public Centroid(int id, String s) {
    this.id = new IntWritable(id);
    this.point = new Point(s);
}
```

Listing 18: `Centroid(int id, String s)`

`getId()` Gets the centroid's id.

```java
public IntWritable getId(){
    return this.id;
}
```

Listing 19: `getId()`

`getPoint()` Gets the centroid's point with its coordinates.

```java
public Point getPoint() {
    return point;
}
```

Listing 20: `getPoint()`

`write(DataOutput out)` Serializes the *Centroid* object.

```java
public void write(DataOutput out) throws IOException {
    id.write(out);
    point.write(out);
}
```

Listing 21: `write(DataOutput out)`

`readFields(DataInput in)` Deserializes the *Centroid* object.

```java
public void readFields(DataInput in) throws IOException {
    id.readFields(in);
    point.readFields(in);
}
```

Listing 22: `readFields(DataInput in)`

`compareTo(Centroid o)` Is used in order to make the comparison among other *Centroid* objects when they are used as key.

```java
public int compareTo(Centroid o) {
    return this.id.compareTo(o.getId());
}
```

Listing 23: `compareTo(Centroid o)`

3.3 Class Utils

`class Utils` This class contains parameters useful for the application. The min shift is set to 0.05, because looking at the **KMeans** Python function of the `scikit-learn`[1] library documentation, the suggested value should be close to 0. Moreover, also for the choice of the number of iteration, we referred to the default value set in the previous algorithm.

```java
public class Utils {
    // Maximum number of iterations after which the algorithm
    terminates
    protected final static int MAX_ITERATIONS = 10;
    // Minimum centroids shitf that, if achieved, the algorithm
    terminates
    protected final static double MIN_SHIFT = 0.05;
    // String of output file common to all the reducers
    protected final static String FILE_NAME = "part-r-000";
}
```

Listing 24: `class Utils`

3.4 Class KMeans

The following is the Driver class which is responsible of initializing the centroids from file (both at first iteration and the followings), initialize the Hadoop's *Job* variable and check if the stop conditions are satisfied after every iteration.

`centroids HashMap` and `readCentroidsFile` The variable *centroids* is an HashMap where the centroids are saved at each iteration. At the first iteration the centroids are saved with the *readCentroidsFile* method in the following:

```java
    private final static HashMap<Integer,Point> centroids = new
    HashMap<Integer,Point>();

    private static void readCentroidsFile(final Configuration conf,
     final Path inputPath) {
        FileSystem fs_input = null;
        BufferedReader br = null;
        try {
            fs_input = inputPath.getFileSystem(conf);
            br = new BufferedReader(new InputStreamReader(fs_input.
    open(inputPath)));
```

[1]`https://scikit-learn.org/stable/`

```java
            int i = 0;
            String line = null;
            while((line = br.readLine()) != null) {
                centroids.put(i ,new Point(line));
                i++;
            }
            br.close();
            fs_input.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Listing 25: `Centroids`

`readOutputFile` With the following method, which is called at the end of every iteration, the application parse the output file in the *hdfs* file system, reads the new centroids, compute the shift (which is one of the two stop conditions), and update the hashmap with the new centroids.

```java
    private static double readOutputFile(final Configuration conf, final Path outputPath, final int k, final int numReducers){
        BufferedReader br = null;
        FileSystem fs = null;
        double shift = 0.0;
        Point newCoords = new Point();
        for(int fileNumber = 0; fileNumber < numReducers; fileNumber++ ){
            String name = "";
            if(fileNumber < 10)
                name = "0" + fileNumber;
            else
                name = String.valueOf(fileNumber);
            String path = outputPath + "/" + Utils.FILE_NAME + name;
            Path pt = new Path(path);
            try {
                fs = outputPath.getFileSystem(conf);
                br = new BufferedReader(new InputStreamReader(fs.open(pt)));
                String line;
                String temp = "";
                while((line = br.readLine()) != null){
                    String[] split = line.split("\\s+");
                    for(int j = 1; j < split.length; j++)
                        temp += split[j] + " ";
                    newCoords.set(temp);
                    int index = Integer.parseInt(split[0]);
                    shift += centroids.get(index).getDistance(newCoords);
                    centroids.put(index, new Point(newCoords));
                    temp = "";
                }
                br.close();
                fs.close();
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return shift;
    }
```

Listing 26: `readOutputFile`

`createJob` Initializes the job object with the parameters of *inputPath*, *outputPath* and the number of reducer selected by command line. Also indicates the mapper, combiner and reducer classes, input and output.
Also removes the previous output folder.

```
    private static Job createJob(final Configuration conf, final
    Path inputPath, final Path outputPath, final int numReducers){
        Job job = null;
        try {
            job = Job.getInstance(conf, "kmeans");
            job.setJarByClass(Kmeans.class);
            job.setMapperClass(KmeansMapper.class);
            job.setCombinerClass(KmeansCombiner.class);
            job.setMapOutputKeyClass(Centroid.class);
            job.setMapOutputValueClass(Point.class);
            job.setNumReduceTasks(numReducers);
            job.setReducerClass(KmeansReducer.class);
            job.setOutputKeyClass(IntWritable.class);
            job.setOutputValueClass(Text.class);
            FileInputFormat.addInputPath(job, inputPath);
            outputPath.getFileSystem(conf).delete(outputPath, true)
    ;
            FileOutputFormat.setOutputPath(job, outputPath);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return job;
    }
```

Listing 27: `createJob`

`main` The main is responsible of receving the input from command line, taking the arguments (input file, centroid file, number of centroids, number of reducers and output folder). If the number of reducers from command line is greater than the number of centroids, the number will be reduced at the number of centroids.
It reads the centroid from file and put them in the configuration, hence they will be retrieved by the mapper with the key ”centroids”.
Then starts the iterations (in the while condition it checks if the max iterations condition has been achieved), every time an iteration is completed, it checks if

the min shift is achieved; If it is, the algorithm is terminated and in the output folder there will be the final centroids.

```java
    public static void main(String[] args) throws Exception{
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length != 5) {
            System.err.println("Usage: kmeans <input file> <centroid file> <k> <reducers> <output folder>");
            System.exit(2);
        }
        final Path inputPath = new Path(otherArgs[0]);
        final Path centroidPath = new Path(otherArgs[1]);
        final Path outputPath = new Path(otherArgs[4]);
        final int k = Integer.parseInt(otherArgs[2]);
        final int numReducers = (Integer.parseInt(otherArgs[3]) > k) ? k : Integer.parseInt(otherArgs[3]);
        readCentroidsFile(conf, centroidPath);
        String[] coords = new String[k];
        for(Integer index : centroids.keySet()) {
            coords[index] = centroids.get(index).toString();
        }
        conf.setStrings("centroids", coords);
        conf.setInt("k", k);
        int iterations = 0;
        while (iterations < Utils.MAX_ITERATIONS){
            Job job = createJob(conf, inputPath, outputPath, numReducers);
            if(job.waitForCompletion(true) == false){
                System.err.println("Job termined with error");
                System.exit(1);
            }
            double shift = readOutputFile(conf, outputPath, k, numReducers);
            System.out.println("Iteration number: " + iterations + " Shift: " + shift);
            if(shift <= Utils.MIN_SHIFT) {
                System.out.println("Min shift achieved");
                break;
            }
            for(Integer index : centroids.keySet())
                System.out.println("Centroids " + index + ": " + centroids.get(index).toString());
            String[] result = new String[k];
            for(Integer index : centroids.keySet())
                result[index] = centroids.get(index).toString();
            conf.setStrings("centroids", result);
            iterations++;
        }
        if(iterations == Utils.MAX_ITERATIONS)
            System.out.println("Max iterations achieved");
        System.exit(0);
    }
}
```

Listing 28: `main`

3.5 Class KMeansCombiner

`class KMeansCombiner` Receives for each cluster the list of points associated with it, collected by *Mapper*. It computes the partial sum of points and saves in the *pointSum* object the number of summed elements because it will be used by the *Reducer* to calculate the new centroid. The object *pointSum* is used to avoid to initialize a new object in the *reduce* method.

```java
public class KmeansCombiner extends Reducer<Centroid, Point,
    Centroid, Point> {

    private final static Point pointSum = new Point();

    public void reduce(Centroid id, Iterable<Point> points, Context
     context) throws IOException, InterruptedException{
        final Iterator<Point> it = points.iterator();
        pointSum.set(it.next());
        // Parse every point
        while(it.hasNext()){
            // Add every coordinate of the points
            pointSum.add(it.next());
        }
        context.write(id, pointSum);
    }
}
```

Listing 29: `class KMeansCombiner`

3.6 Class KMeansMapper

`class KMeansMapper` Parses the data points and compute the Euclidean distance between each point and the centroid. In output there will be the centroid closest to the examined point (used as a key) and the point itself (used as a value). In the *setup* method the centroids are deserialized and saved in an *ArrayList* object since they will be used by the *map* method. Moreover in the *setup* method is deserialized and saved the number of centroids. The object *p* is used to avoid to initialize a new point object every time a new point is received in the *map* method.

```java
public class KmeansMapper extends Mapper<Object, Text, Centroid,
    Point>{

    private final static ArrayList<Centroid> centroids = new
    ArrayList<Centroid>();
    private static int k;
    private final static Point p = new Point();

    @Override
    protected void setup(Context context) throws IOException,
    InterruptedException{
        super.setup(context);
        String[] lines = context.getConfiguration().getStrings("
    centroids");
        // Get the number of centroids, -1 default value
        k = context.getConfiguration().getInt("k", -1);
        // Get the values of the centroids
        for(int i = 0; i < lines.length; i++)
            centroids.add(new Centroid(i,lines[i]));
    }

    public void map(final Object key, final Text value, final
    Context context) throws IOException, InterruptedException{
        // For each point, we associate the closest centroid
        p.set(value.toString());
        double min_distance = 0.0;
        int id_min_distance = -1;
        // Computing euclidean distances from the target point and
    all the centroids to associate it
        for (int i = 0; i < k; i++){
            double distance = p.getDistance(centroids.get(i).
    getPoint());
            if (i == 0 || min_distance > distance){
                min_distance = distance;
                id_min_distance = i;
            }
        }
        context.write(centroids.get(id_min_distance), p);
    }
}
```

Listing 30: `class KMeansMapper`

3.7 Class KMeansReducer

`class KMeansReducer` Receives as input a *Centroid*, which is the key, and the list of *Point* associated with it, collected by the *Mapper*. It will produce the new coordinates for the examined centroid by computing the mean among all the points in the received list. Each of them is summed up in the *pointSum* object by means of the *add()* method, which also updates the number of elements that contributed to this sum. The new coordinates are then obtained by means of the avg() method, which computes the mean for each coordinate in *pointSum*. In output there will be the *id* of the examined centroid and a *Text* with the new coordinates.

```
public class KmeansReducer extends Reducer<Centroid, Point,
    IntWritable, Text>{

    private final static Point pointSum = new Point();
    private final static Text t = new Text();

    public void reduce(Centroid id, Iterable<Point> points, Context
     context) throws IOException, InterruptedException{
        final Iterator<Point> it = points.iterator();
        pointSum.set(it.next());
        // Parse every point
        while(it.hasNext()){
            // Add every coordinate of the points
            pointSum.add(it.next());
        }
        // Get the average of the point in order to get the new
    centroid
        pointSum.avg();
        t.set(pointSum.toString());
        context.write(id.getId(), t);
    }
}
```

Listing 31: `class KMeansReducer`

4 Spark

The Spark implementation of k-means clustering algorithm is written in Python and its design is similar to the Hadoop implementation seen before. However, since Spark allows to execute complex tasks in few lines of code compared to Hadoop, its implementation is much more shorter, in fact we decided to enclose it in a single file called *main.py*.

4.1 main.py

The files contains, in addition to the main, the implementation of the following functions: `parse_point`, `compute_closest`, `delete_previous_output`.
Let's analyze them one by one.

`parse_point(line)` Parses an n-dimensional point in string format, where the coordinates are separated by a whitespace, and converts it into an *ndarray* of float elements using the *numpy* library.

```python
def parse_point(line):
    return np.array([float(coords) for coords in line.split()])
```

Listing 32: `parse_point`

`compute_closest(point, cent)` It takes as input an *ndarray* containing the coordinates of a point and a list of *ndarray* containing the coordinates of all the centroids. This method computes the euclidean distance between the point and each centroid and returns as output the index of the centroid closest to the target point and a tuple containing: an "1", that will be used after to count how many points made up a sum, and the point that had been taken as input.

```python
def compute_closest(point, cent):
    index = -1
    min_dist = math.inf
    for j in range(len(cent)):
        temp_dist = np.sqrt(np.sum((point - cent[j]) ** 2))
        if temp_dist < min_dist:
            min_dist = temp_dist
            index = j
  # in this key-value pair index is the key and (1, point) is the
    value
    return index, (1, point)
```

Listing 33: `compute_closest`

`delete_previous_output(output, sc)` This method takes as input the output folder and the spark context and deletes, if it exists, the output of a previous run to not have to do it manually.

```python
def delete_previous_output(output, sc):
    Path = sc._gateway.jvm.org.apache.hadoop.fs.Path
    FileSystem = sc._gateway.jvm.org.apache.hadoop.fs.FileSystem

    fs = FileSystem.get(sc._jsc.hadoopConfiguration())
    fs.delete(Path(output))
```
Listing 34: `delete_previous_output`

`main()` Here are performed the main operations of the application.
First, we assigne the parameters taken from command line (line 6-11), we initialize a new spark context and we set the log level to "Error" and we delete the output of the previous run, if it exists, using the `delete_previous_output` method seen before.

```
1  if __name__ == "__main__":
2      if len(sys.argv) != 7:
3          print("Usage: ./main.py <inputFile> <centroidFile> <K> <
       outputFolder> <maxIterations> <minShift>")
4          sys.exit()
5
6      input_file = sys.argv[1]
7      centroid_file = sys.argv[2]
8      k = int(sys.argv[3])
9      output = sys.argv[4]
10     maxIterations = int(sys.argv[5])
11     minShift = float(sys.argv[6])
12
13
14     # initialization of a new Spark Context
15     sc = pyspark.SparkContext(appName="KMeans", master="yarn")
16     sc.setLogLevel("ERROR")
17
18   # deletion of previous output
19     delete_previous_output(output, sc)
```
Listing 35: `main_first_part`

After that, we create two RDDs using the textFile() method.
The first one contains the points to be mapped taken from `input_file`, it is parsed using the `parse_point` method, and then, the RDD is persisted, only in memory, using the `cache()` method because we need it at every iteration.
The `initial_centroids` array contains the coordinates of the centroids that will be used for the first iteration of the algorithm.
Also in this case the `parse_point` method is performed because we want that points and centroids have the same format, instead, in this case, we don't need to persist an RDD in memory, but we materialize it by using the *colletct()* function, because from the second iteration on, the coordinates of the centroids are calculated as the average of the points that belong to the same cluster, so they will change.

```
# points is an RDD containing the points taken from input_file
points = sc.textFile(input_file).map(parse_point).cache()
# initial_centroids is an array containing all the elements of the
    RDD that contains the centroids taken from centroid_file
```

```
initial_centroids = sc.textFile(centroid_file).map(parse_point).
    collect()
```

Listing 36: `main_second_part`

In the main loop, if neither of our two stop conditions is verified, we perform the main operations of the algorithm.
First we save the values of the coordinates of the centroids in a broadcast variable using the *broadcast()* method so that all the worker nodes have it available.
Then we map each point to the closest centroid using the `compute_closest` method (line 8), we sum the coordinates of the points that belong to the same cluster counting the number of occurrences for each cluster (line 10) and we compute the coordinates of new centroids by dividing the sum of the coordinates calculated previously by the number of points that made up that sum (line 12).

```
1  shift = math.inf
2  iteration = 0
3
4  while iteration < maxIterations and shift > minShift:
5      # used a broadcast variable to save centroids values visible on
        all the worker nodes
6      broadcast_centroids = sc.broadcast(initial_centroids)
7    # mapping each point to the closest centroid
8      closest_centroid = points.map(lambda x: (compute_closest(x,
       broadcast_centroids.value)))
9    # sum of the occurrences (0) and of the coordinates (1) of each
       point reduced by key
10     summed_points = closest_centroid.reduceByKey(lambda x, y: (x[0]
        + y[0], x[1] + y[1]))
11     # new_centroids is an ordered array (by key) containing the
       coordinates of new centroids
12     new_centroids = summed_points.map(lambda v: (v[0], v[1][1] / v
       [1][0])).values().collect()
```

Listing 37: `main_third_part`

After calculating the coordinates of new centroids, we compute, always inside the main loop, the Euclidean distance between old and new centroids, we assign `new_centroids` to `initial_centroids`, we empty `new_centroids` and we increment the *iteration* variable related to the number of iterations performed.

```
    shift = 0.0
    # computation of the euclidian distance between old and new
    centroids
    for c in range(len(new_centroids)):
        shift += np.sqrt(np.sum((new_centroids[c] -
    initial_centroids[c]) ** 2))

    initial_centroids = new_centroids
    new_centroids = []
    iteration += 1
```

Listing 38: `main_fourth_part`

Finally, we save the coordinates of final centroids by creating the `to_output` RDD with the *parallelize()* method and writing its elements as a text file in the *output* folder.

```
to_output = sc.parallelize(initial_centroids)
# saving the final result in a text file
to_output.saveAsTextFile(output)
```

Listing 39: `main_final_part`

5 Validation

In order to assess the correctness of our algorithm and moreover, the correctness of the performance analysis, a validation test as been performed both with the *Hadoop* and *Spark* algorithms.
The results has been compared with a benchmark algorithm, which is a **KMeans** Python function of the `scikit-learn`[2] library.
A random input file of 100 rows and 2 dimensions have been generated, and from that file 10 random centroids have been chosen. In this way we were able to give as input the same file and the same centroids to the Python script, Hadoop and Spark to check if the resulting centroids were the same for all three.
The resulting centroids of the Python Script after 10 iterations are the following:

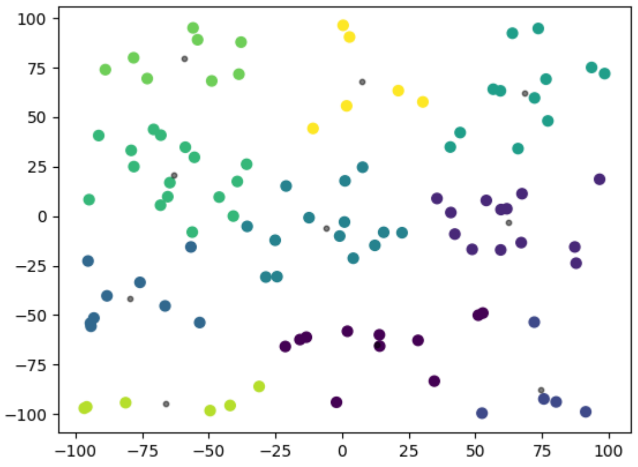


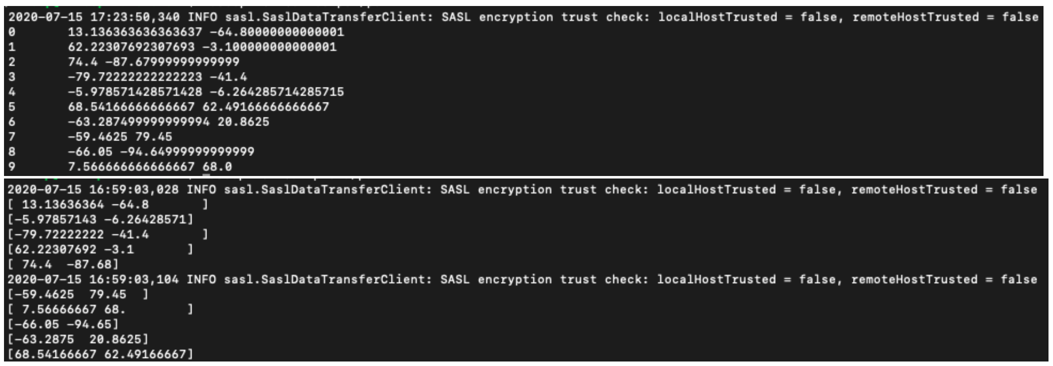
Figure 1: The scatterplot of the result in the Python script

1. $[13.136363 \quad -64.8]$
2. $[62.223072 \quad -3.0999997]$
3. $[74.4 \quad -87.68]$
4. $[-79.72223 \quad -41.4]$
5. $[-5.9785714 \quad -6.2642865]$
6. $[68.54167 \quad 62.491673]$
7. $[-63.287495 \quad 20.8625]$

[2]`https://scikit-learn.org/stable/`

8. $[-59.4625 \quad 79.450005]$
9. $[-66.05 \quad -94.65]$
10. $[7.5666676 \quad 68.00001]$

The same test has been replicated in Hadoop and Spark, the results are the following:



```
2020-07-15 17:23:50,340 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
0       13.136363636363637 -64.80000000000001
1       62.22307692307693 -3.100000000000001
2       74.4 -87.67999999999999
3       -79.72222222222223 -41.4
4       -5.978571428571428 -6.264285714285715
5       68.54166666666667 62.49166666666667
6       -63.287499999999994 20.8625
7       -59.4625 79.45
8       -66.05 -94.64999999999999
9       7.566666666666667 68.0
```

```
2020-07-15 16:59:03,028 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
[ 13.13636364 -64.8       ]
[-5.97857143 -6.26428571]
[-79.72222222 -41.4       ]
[62.22307692 -3.1       ]
[ 74.4   -87.68]
2020-07-15 16:59:03,104 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
[-59.4625  79.45  ]
[ 7.56666667 68.        ]
[-66.05 -94.65]
[-63.2875  20.8625]
[68.54166667 62.49166667]
```

Figure 2: The results for, respectively, Hadoop and Spark after 10 iterations

We can clearly see that the resulting centroids for every framework are the same, hence we can state that our algorithm for Hadoop and Spark perform well and the following results are validated.

6 Performance Analysis

In order to evaluate the performances of Hadoop and Spark frameworks for this algorithm, we prepared an experiment consisting of:

- 3 different files of input, generated at random:
  - 1000 points
  - 10000 points
  - 100000 points
- Each file has 2 different dimensions of the coordinates:
  - 3 dimensions
  - 7 dimensions
- 2 different number of clusters:
  - 7 clusters
  - 13 clusters
- Each test has been repeated 10 times, for each one a different set of centroids has been randomly chosen (with different seeds).
- On the results it has been computed a 99% confidence interval.
- The number of reducers in Hadoop has been set to 2.

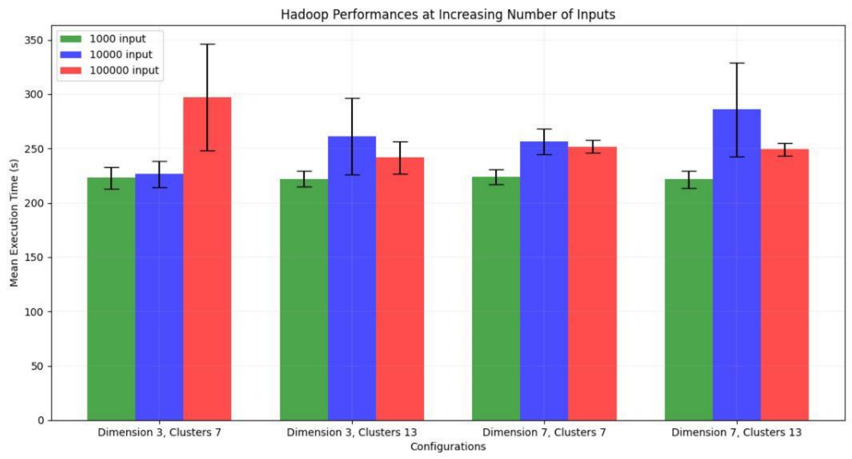


Figure 3: The Hadoop Performance graphic

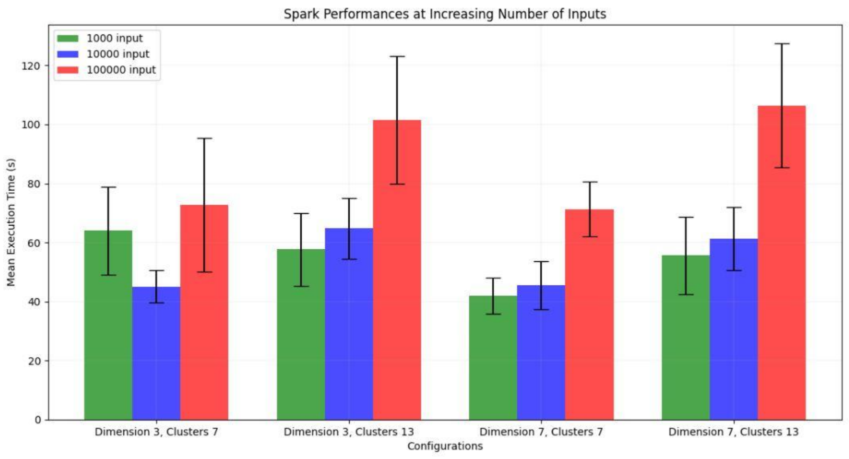


Figure 4: The Spark Performance graphic

We can clearly see from the ordinates axis, that Hadoop is much more slower than Spark.
Hadoop is slower because at each iteration it has to read the intermediate results from disk, meanwhile Spark reads it from memory.
We can notice that the confidence intervals are large, this is probably due to the choice of the initial centroids. For each repetition they were randomly chosen and this could influence the convergence of the algorithm: sometimes the algorithm could converge much earlier than other cases, hence the variance can increase.
We could try to increase the number of repetitions in order to achieve more accurate results.