UNIVERSITÀ DI PISA

Department of Information Engineering

The *k*-means Clustering Algorithm in MapReduce

Cloud Computing

F. Barbarulo, D. Casu, B.T. Gurmesa, G.B. Rolandi

A.Y. 2019/2020

Contents

1 Introduction

The problem of partitioning a set of unlabeled points into clusters appears in a wide variety of applications. One of the most well-known and widely used clustering algorithms is *k-means*, thanks to its simplicity and speed.

Given a set of n data points with dimension d, the *k-means problem* seeks to find a set of k points, called *means*, which minimizes a certain objective function. Following the iterative process of a standard heuristic algorithm that approximates well the solution, an implementation based on the MapReduce paradigm is presented, specifically employing the Hadoop and Spark frameworks, allowing the distributed processing of large data sets across clusters of computers.

The document is organized as follows: Section 2 illustrates the design choices to adapt the iterative algorithm to the MapReduce paradigm, Section 3 discusses the Hadoop implementation, Section 4 introduces the Spark one. Finally, Section 5 and 6 report the experimental results.

2 Design

The k-means algorithm can be decomposed in three stages, suitable for a MapReduce implementation:

1. Sampling, where k initial means are chosen at random starting from the n points of the data set;
2. Clustering, where the means are recomputed and updated to be the centroids of the data points;
3. Convergence, where the stop criterion is evaluated, based on the achievement of a local optimum.

The first stage is performed once, at the beginning of the algorithm, while the other two subsequent stages are executed multiple times.

2.1 Sampling stage

The first stage aims at sampling the initial k means, selecting them uniformly at random starting from the data points. Since the data set will be split among the mappers, the idea is to assign a priority to each parsed point in isolation, by means of randomly extracted integers, and exploit the *shuffle&sort* stage to obtain a random permutation of them. At the end, only the k points with the highest priorities will be kept as the initial means.

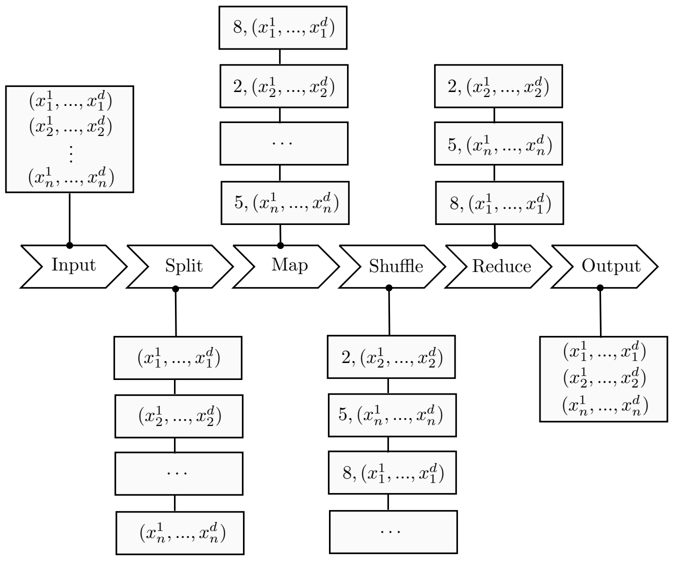


Figure 1: Sampling stage of 3 d-dimensional points out of n.

An example of this process is depicted in Figure 1.

More in details:

1. each mapper receives several points as input and assigns a priority to each of them, emitting the priority as key and the point as value;
2. a single reducer receives, for each priority value, a list of points. Points are emitted if the current number of selected points is less than k. The reducer must be one, to ensure that exactly k means are chosen.

This basic solution leads to a network overload, since all points are emitted and transmitted over the network, even if the parameter k is small. The performance bottleneck can be lessened using combiners: in this specific case, in-mapper combiners can be used, being a very high number of clusters unlikely to be chosen, as shown in Algorithm 1 and Algorithm 2.

Algorithm 1 Sampling stage Mapper

```
1: class Mapper
2:     method Initialize()
3:         $P \longleftarrow$ new PriorityQueue
4:     method Map(docid $id$, doc $d$)
5:         for all point $p \in$ doc $d$ do
6:             $priority \longleftarrow$ Random()
7:             $P$.Push($priority$, $p$)
8:             if $P$.Size > k then                    ▷ Keep at most k points in the queue
9:                 $P$.Pop()
10:    method Close()
11:        for all pairs $priority_i$, $p_i \in P$ do
12:            Emit(priority $priority_i$, point $p_i$)
```

Algorithm 2 Sampling stage Reducer

```
1: class Reducer
2:     method Initialize()
3:         $counter \longleftarrow 0$
4:     method Reduce(priority $prio$, points $[p_1, p_2, p_3, \dots]$)
5:         for all point $p \in$ points $[p_1, p_2, p_3, \dots]$ do
6:             if $counter < k$ then                    ▷ $k$ is the number of clusters
7:                 $counter \longleftarrow counter + 1$
8:                 Emit($null$, point $p$)
```

The in-mapper combiner leverages a priority queue to store at most k points, sorted by priority, so that the mappers can perform a first skimming of points, avoiding the transmission of all of them. For instance, assuming that 1,000,000 data points are provided and the system can count on 3 mappers having their own priority queues, at most $k \times 3$ points out of 1,000,000 will flow in the network. The single reducer behaves as explained before and the output of the sampling stage is a document containing the k sampled points, representing the starting means.

2.2 Clustering stage

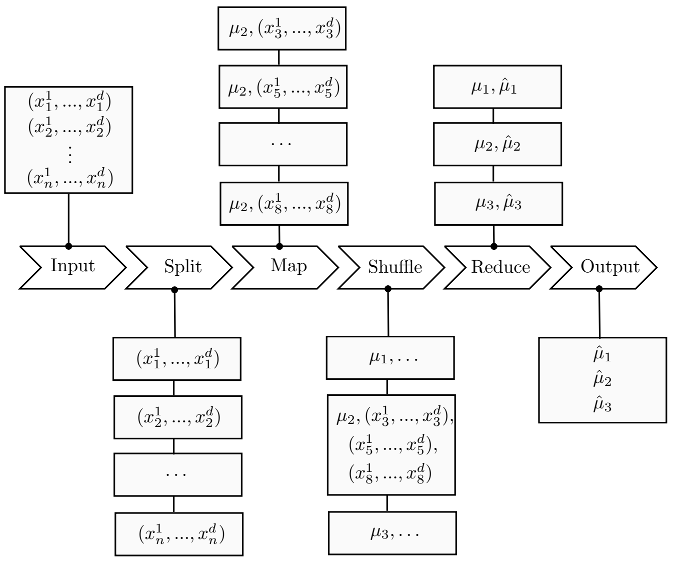
In the clustering stage, the k clusters are built: each data point is assigned to the cluster whose current centroid is the closest among all the other $k-1$ centroids, in terms of Euclidean distance.

The mappers map a point in a cluster, emitting key-value pairs $\langle mean, point \rangle$, so that, by means of *shuffle&sort*, the reducers receive lists of points belonging to the same cluster and can compute the new centroid. The latter is done summing all the points in the same cluster and dividing the result by the cardinality of the cluster itself. This is possible if the mappers have at their disposal the current means (the sampled means or the ones computed in the previous iteration).

An example is illustrated in Figure 2: the mappers map, for instance, points x_3, x_5, x_8 to the cluster whose current mean is μ_2. The reducer computes the new mean $\hat{\mu}_2$ according to the general formula

$$\hat{\mu}_j = \left(\frac{\sum_{i \in c_j} x_i^1}{|c_j|}, ..., \frac{\sum_{i \in c_j} x_i^d}{|c_j|} \right) \tag{1}$$

where $j = 1, ..., k$; c_j is the cluster with mean $\hat{\mu}_j$ and $|c_j|$ is the cardinality of the cluster.



$(x_1^1, ..., x_1^d)$ $(x_2^1, ..., x_2^d)$ ⋮ $(x_n^1, ..., x_n^d)$
$\mu_2, (x_3^1, ..., x_3^d)$
$\mu_2, (x_5^1, ..., x_5^d)$
...
$\mu_2, (x_8^1, ..., x_8^d)$
$\mu_1, \hat{\mu}_1$
$\mu_2, \hat{\mu}_2$
$\mu_3, \hat{\mu}_3$
Input
Split
Map
Shuffle
Reduce
Output
$(x_1^1, ..., x_1^d)$
$(x_2^1, ..., x_2^d)$
...
$(x_n^1, ..., x_n^d)$
$\mu_1, \dots$
$\mu_2, (x_3^1, ..., x_3^d), (x_5^1, ..., x_5^d), (x_8^1, ..., x_8^d)$
$\mu_3, \dots$
$\hat{\mu}_1$ $\hat{\mu}_2$ $\hat{\mu}_3$

Figure 2: Clustering stage computing the 3 new means.

Even in this case, this basic solution leads to performance issues mainly due to the transmission of all points in the network. Again, the in-mapper combiner represents a good trade-off between memory and network usage.

Algorithm 3 and Algorithm 4 show the Mapper and Reducer pseudocode implementing the in-mapper solution.

Algorithm 3 Clustering stage Mapper

```
1: class Mapper
2:     method Initialize()
3:         $M \leftarrow \{\mu_1, \dots, \mu_k\}$                ▷ Intermediate means
4:         for all mean $\mu_i \in M$ do
5:             $c_i \leftarrow 0$
6:             $v_i \leftarrow 0$
7:     method Map(docid $id$, doc $d$)
8:         for all point $p_i \in$ doc $d$ do
9:             $w \leftarrow \text{argmin}_j \|p_i - \mu_j\|_2^2$
10:            $c_w \leftarrow c_w + p_i$
11:            $v_w \leftarrow v_w + 1$
12:    method Close()
13:        for all mean $\mu_i \in M$ do
14:            Emit(mean $\mu_i$, pairs $(c_i, v_i)$)
```

Algorithm 4 Clustering stage Reducer

```
1: class Reducer
2:     method Reduce(mean $\mu$, pairs $[(c_1, v_1), (c_2, v_2), \dots]$)
3:         $sum \leftarrow 0$
4:         $values \leftarrow 0$
5:         for all $(c_i, v_i) \in$ pairs $[(c_1, v_1), (c_2, v_2), \dots]$ do
6:             $sum \leftarrow sum + c_i$
7:             $values \leftarrow values + v_i$
8:         $centroid \leftarrow sum/values$
9:         Emit($null$, $centroid$)
```

Instead of emitting all points, the mappers exploit a data structure that links each mean to a sum of points identified as part of the cluster and to the number of points currently summed. In this way, every mapper will emit only k points, one for each mean.

At the initialization, a mapper gets the current means $M = \{\mu_1, ..., \mu_k\}$ and associates an accumulator c_i and a counter v_i to each mean μ_i (lines 3-6). For each received point, the Map function finds the closest mean, updates the accumulator and increases by one the counter related to it (lines 9-11). At the end, it emits every mean with its actual accumulator and counter values.

The reducers receive a list of $\langle accumulator, counter \rangle$ pairs, where the key represents the corresponding mean: in the Reduce function, according to Eq. 1, they sum up all the partial sums dividing the result by the number of points represented by the sum of all counters (lines 6-7). In this stage, the new means will be produced employing multiple reducers: the best choice is to adopt k reducers, namely one per cluster.

2.3 Convergence stage

The convergence stage regards the computation of the stop condition, in order to state whether the means have converged and, in case, stop the algorithm. Given $X = \{x_1, ..., x_n\}$ the set of n data points, $M = \{\mu_1, ..., \mu_k\}$ the set of the current final means, the following objective function must be computed:

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

In other words, the distances between each point and its closest mean must be computed. This is done by the mapper, where each minimum distance is emitted as value with the unique textual key "key". At the end, all the distances are summed according to Eq. 2 to find the value of the objective function at the current step.

An example of convergence stage is illustrated in Figure 3. The minimum distance is represented by $||x_i - \mu_c||$, where μ_c is the current centroid of the cluster to which x_i belongs.

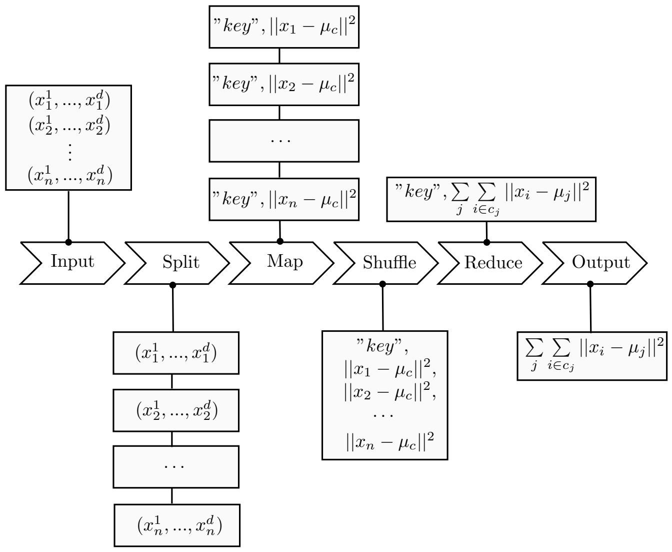

$"key", ||x_1 - \mu_c||^2$
$"key", ||x_2 - \mu_c||^2$
$\cdots$
$"key", ||x_n - \mu_c||^2$
$(x_1^1, ..., x_1^d)$
$(x_2^1, ..., x_2^d)$
$\vdots$
$(x_n^1, ..., x_n^d)$
$"key", \sum_j \sum_{i \in c_j} ||x_i - \mu_j||^2$
Input
Split
Map
Shuffle
Reduce
Output
$(x_1^1, ..., x_1^d)$
$(x_2^1, ..., x_2^d)$
$\cdots$
$(x_n^1, ..., x_n^d)$
$"key", ||x_1 - \mu_c||^2, ||x_2 - \mu_c||^2, \cdots ||x_n - \mu_c||^2$
$\sum_j \sum_{i \in c_j} ||x_i - \mu_j||^2$

Figure 3: Convergence stage: computation of the objective function.

In order not to flood the network with all the distances computed in the mappers, an in-mapper combiner is used. Algorithm 5 and Algorithm 6 show the Mapper and Reducer pseudocode respectively. To implement the in-mapper combiner, the mappers maintain a variable that accumulates the value of the distances between each point and the centroid of its cluster. In the Map function, for each point, the closest mean is found and the distance is summed to the current value of the accumulator variable (lines 7-8). The single reducer simply sums up all the distances emitting the final value.

Algorithm 5 Convergence stage Mapper

```
1: class Mapper
2:     method Initialize()
3:         $M \leftarrow \{\mu_1, \dots, \mu_k\}$        ▷ Actual final means got from cache
4:         $s \leftarrow 0$
5:     method Map(docid $id$, doc $d$)
6:         for all point $p_i \in$ doc $d$ do
7:             $c \leftarrow \text{argmin}_j \|p_i - \mu_j\|_2^2$
8:             $s \leftarrow s + \|p_i - \mu_c\|_2^2$
9:     method Close()
10:        Emit("$key$", $s$)
```

Algorithm 6 Convergence stage Reducer

```
1: class Reducer
2:     method Reduce(key "$key$", sums $[s_1, s_2, \dots]$)
3:         $objectiveFunction \leftarrow 0$
4:         for all sum $s \in$ sums $[s_1, s_2, \dots]$ do
5:             $objectiveFunction \leftarrow objectiveFunction + s$
6:         Emit($null$, $objectiveFunction$)
```

Then, in the driver function, the error is evaluated as the difference between two consecutive values of the objective function. The convergence criterion is typically met when the total error stops changing between consecutive steps, in which case a local optimum of the objective function has been reached.

3 Hadoop

In this section the k-means Hadoop Java implementation, based on the design explained in Section 2, is presented. The three stages are implemented in the relative classes: `Sampling`, `Clustering`, `Convergence`. Each of them has its own driver function, the `Mapper` and `Reducer` classes. Moreover, some utility classes have been implemented in order to manage the data points and their serialization and deserialization performed by Hadoop: `Point`, `PriorityPoint`, `AccumulatorPoint`.

3.1 Class Point

The `Point` class represents the point object with its coordinates. Coordinates are stored in an `ArrayList` of doubles and the class provides three constructors:

- `Point()`
  Used for instantiating a point for deserialization:

```java
public Point(){
    this.coordinates = new ArrayList<>();
}
```

- `Point(String value)`
  Used for instantiating a point with coordinates specified in the string, separated by the comma (`","`) character:

```java
public Point(String value){
    this();

    String[] indicesAndValues = value.split(",");
    for (String v: indicesAndValues) {
        coordinates.add(Double.parseDouble(v));
    }
}
```

- `Point(int d)`
  Used for instantiating a d-dimensional point with all coordinates initialized to zero:

```java
public Point(int d){
    this();

    for (int i = 0; i < d; i++)
        coordinates.add(0.0);
}
```

It implements the `WritableComparable` interface provided by Hadoop, overriding the methods:

- `write(DataOutput out)`
  Serializes the Point object:

```java
public void write(DataOutput out) throws IOException {
    out.writeInt(coordinates.size());
    for (Double c: coordinates)
```

```
        out.writeDouble(c);
}
```

- `readFields(DataInput in)`
  Deserializes the Point object:

```
public void readFields(DataInput in) throws IOException {
    int size = in.readInt();

    coordinates = new ArrayList<>();
    for (int i = 0; i < size; i++)
        coordinates.add(in.readDouble());
}
```

- `compareTo(Object o)`
  Makes possible the comparison when the Point is used as key:

```
public int compareTo(Object o) {
    ArrayList<Double> thisCoordinates = this.getCoordinates();
    ArrayList<Double> thatCoordinates = ((Point)o).getCoordinates();

    for (int i = 0; i < thisCoordinates.size(); i++){
        if (thisCoordinates.get(i) < thatCoordinates.get(i))
            return -1;

        if (thisCoordinates.get(i) > thatCoordinates.get(i))
            return 1;
    }

    return 0;
}
```

- `hashcode()`
  Exploits the corresponding `Text` method to guarantee a correct partition of keys and its consistency when employing heterogeneous machines:

```
public int hashCode(){
    return new Text(toString()).hashCode();
}
```

Three more methods have been implemented in order to accomplish the *k*-means algorithm:

- `getSquaredDistance(Point that)`
  Returns the squared distance between Point `this` and `that`:

```
public double getSquaredDistance(Point that){
    double sum = 0;
    ArrayList<Double> thisCoordinates = this.getCoordinates();
    ArrayList<Double> thatCoordinates = that.getCoordinates();

    for (int i = 0; i < thisCoordinates.size(); i++){
        sum += (thisCoordinates.get(i) - thatCoordinates.get(i))*(thisCoordinates.get(i)
            - thatCoordinates.get(i));
    }

    return sum;
}
```

- `add(Point that)`
  Adds Point `that` to Point `this`:

```
public void add(Point that){
    ArrayList<Double> thisCoordinates = this.getCoordinates();
    ArrayList<Double> thatCoordinates = that.getCoordinates();

    for (int i = 0; i < thisCoordinates.size(); i++){
        thisCoordinates.set(i, thisCoordinates.get(i) + thatCoordinates.get(i));
    }
}
```

- `div(int n)`
  Divides all the coordinates by `n`:

```
public void div(int n){
    for (int i = 0; i < coordinates.size(); i++){
        coordinates.set(i, coordinates.get(i)/n);
    }
}
```

3.2 Class PriorityPoint

The `PriorityPoint` class represents a point with an associated priority. The PriorityPoint object is used in the sampling stage when k initial points are selected based on their randomly assigned priority. It extends the `Point` class, adding the priority integer field. Since it is managed by the `PriorityQueue` data structure, it needs to implement the `Comparable` interface overriding the `compareTo(Object o)` method. In this case, the lower the priority value, the higher the priority.

- `compareTo(Object o)`

```
    public int compareTo(Object o) {
        return Integer.compare(((PriorityPoint)o).priority, this.priority);
    }
```

3.3 Class AccumulatorPoint

The `AccumulatorPoint` class represents a sum of points with the associated size, i.e. the number of summed points. The `AccumulatorPoint` object is used in the clustering stage for managing the partial sums in the in-mapper combiner. Since they are emitted by the clustering mapper, it needs to implement the `Writable` interface. The emitted object corresponds to the pair (c_i, v_i) in line 14 of Algorithm 3.

3.4 Class Sampling

The `Sampling` class wraps the mapper class, the reducer class and the driver code for the sampling stage.

- `class SamplingMapper`
  Parses the data points from file, one line at a time, and assigns to them a randomly generated priority. The points are stored in a priority queue declared at class level, i.e. there is one priority queue for each mapper, whose size is at most equal to the number of clusters required for the job. After parsing all the points in the assigned splits, it emits the ones stored in the queue, along with their priorities.

```java
public static class SamplingMapper extends Mapper<LongWritable, Text, IntWritable, Point>
    {
        static int K;
        final static Random rand = new Random();
        final static IntWritable outputKey = new IntWritable();
        final static Point outputValue = new Point();
        final static PriorityQueue<PriorityPoint> pq = new PriorityQueue<>();

        public void setup(Context context){
            Configuration conf = context.getConfiguration();
            K = Integer.parseInt(conf.get("k"));
            rand.setSeed(Long.parseLong(conf.get("seed")));
        }

        public void map(LongWritable key, Text value, Context context) {
            pq.add(new PriorityPoint(rand.nextInt(), value.toString()));

            if (pq.size() > K)
                pq.poll();
        }

        public void cleanup(Context context) throws IOException, InterruptedException {
            for (PriorityPoint pp: pq){
                outputKey.set(pp.getPriority());
                outputValue.set(pp.getCoordinates());
                context.write(outputKey, outputValue);
            }
        }
    }
```

- `class SamplingReducer`
  Receives the mapped points aggregated by key (the random priority) and, if instantiated only once, it emits a number of sampled means equal to the number of clusters set for the job, either obtained in a single execution of `reduce()` or in multiple ones.

```java
public static class SamplingReducer extends Reducer<IntWritable, Point,
                                                    NullWritable, Point>{
        static int K;
        static int meansCount;

        public void setup(Context context){
            Configuration conf = context.getConfiguration();
            K = Integer.parseInt(conf.get("k"));

            meansCount = 0;
        }

        public void reduce(IntWritable key, Iterable<Point> values, Context context)
            throws IOException, InterruptedException {
            for (Point p: values){
                if (meansCount > K)
                    return;

                context.write(null, p);
                meansCount++;
            }
        }
    }
```

- `main(Job job)`
  Prepares the execution of the MapReduce job, setting the necessary classes and input/output types. It deploys a single reducer to ensure the correctness of the sampling.

```java
public static boolean main(Job job) throws IOException, ClassNotFoundException,
                                                        InterruptedException {
        Configuration conf = job.getConfiguration();

        job.setJarByClass(Sampling.class);

        job.setMapperClass(SamplingMapper.class);

        job.setReducerClass(SamplingReducer.class);

        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(Point.class);

        job.setOutputKeyClass(NullWritable.class);
        job.setOutputValueClass(Point.class);

        FileInputFormat.addInputPath(job, new Path(conf.get("input")));
        FileOutputFormat.setOutputPath(job, new Path(conf.get("sampledMeans")));

        return job.waitForCompletion(true);
    }
```

3.5 Class Clustering

The `Clustering` class wraps the mapper class, the reducer class and the driver code for the clustering stage.

- `class ClusteringMapper`
  Parses the data points from file and finds for each of them the closest mean in terms of Euclidean distance. The class is responsible for recovering the means from the distributed cache and for making them available in memory within a HashMap (line 13-27): the latter links a mean with a partial sum of points, where the number of summed elements is tracked. During every `map()` execution, the closest mean to a point is found and the point itself is summed to the specific accumulator in the HashMap (line 36 - 47); at the end of a Mapper execution, the `cleanup()` method emits all the couples `(mean, accumulator)`.

```
 1  public static class ClusteringMapper extends Mapper<LongWritable, Text,
 2                                                      Point, AccumulatorPoint> {
 3
 4          static int D;
 5          final static Map<Point, AccumulatorPoint> centroidSummation = new HashMap<>();
 6
 7          protected void setup(Context context) throws IOException {
 8              Configuration conf = context.getConfiguration();
 9              D = Integer.parseInt(conf.get("d"));
10              centroidSummation.clear();
11
12              // Get the means from cache, either sampled or computed in the previous step
13              URI[] cacheFiles = context.getCacheFiles();
14              FileSystem fs = FileSystem.get(conf);
15
16              for (URI f: cacheFiles) {
17                  InputStream is = fs.open(new Path(f));
18                  BufferedReader br = new BufferedReader(new InputStreamReader(is));
19
20                  String line;
21                  while ((line = br.readLine()) != null) {
22                      Point mean = new Point(line);
23                      centroidSummation.put(mean, new AccumulatorPoint(D));
24                  }
25
26                  br.close();
27              }
28          }
29
30          public void map(LongWritable key, Text value, Context context) {
31              double minDistance = Double.POSITIVE_INFINITY;
32              Point closestMean = null;
33
34              Point p = new Point(value.toString());
35
36              for (Point m: centroidSummation.keySet()){
37                  double d = p.getSquaredDistance(m);
```

```java

                if (d < minDistance){
                    minDistance = d;
                    closestMean = m;
                }
            }

            AccumulatorPoint ap = centroidSummation.get(closestMean);
            ap.add(p);
            centroidSummation.put(closestMean, ap);
        }

        public void cleanup(Context context) throws IOException, InterruptedException {
            for (Map.Entry<Point, AccumulatorPoint> entry: centroidSummation.entrySet()){
                context.write(entry.getKey(), entry.getValue());
            }
        }
    }
```

- `class ClusteringReducer`
  Receives, for each mean, the list of partial sums of points collected by the Mappers; then, it computes the new centroid of a cluster as the ratio between the total sum and the total number of summed points, namely the cardinality of the cluster (line 16-20).

```java
1   public static class ClusteringReducer extends Reducer<Point, AccumulatorPoint,
2                                                         NullWritable, Point> {
3           static int D;
4   
5           public void setup(Context context){
6               Configuration conf = context.getConfiguration();
7               D = Integer.parseInt(conf.get("d"));
8           }
9   
10          public void reduce(Point key, Iterable<AccumulatorPoint> values, Context context)
11                  throws IOException, InterruptedException {
12  
13              Point centroid = new Point(D);
14              int n = 0;
15  
16              for (AccumulatorPoint ap: values){
17                  centroid.add(ap);
18                  n += ap.getSize();
19              }
20              centroid.div(n);
21  
22              context.write(null, centroid);
23          }
24      }
```

- `main(Job job)`
  Prepares the execution of the MapReduce job, setting the necessary classes and input/output types. It deploys multiple reducers to ensure better performances: in particular, the number of reducers is set according to the configuration file, but limited to be at most equal to the number of means, granting that no idle reducers are deployed.

```java
public static boolean main(Job job) throws IOException, ClassNotFoundException,
                                                        InterruptedException {

        Configuration conf = job.getConfiguration();
        int K = Integer.parseInt(conf.get("k"));
        int numReduceTasks = Math.min(K,
                                    Integer.parseInt(conf.get("maxNumberOfReduceTasks")));

        job.setJarByClass(Clustering.class);

        job.setMapperClass(ClusteringMapper.class);
        job.setReducerClass(ClusteringReducer.class);

        job.setNumReduceTasks(numReduceTasks);

        job.setMapOutputKeyClass(Point.class);
        job.setMapOutputValueClass(AccumulatorPoint.class);

        job.setOutputKeyClass(NullWritable.class);
        job.setOutputValueClass(Point.class);

        FileInputFormat.addInputPath(job, new Path(conf.get("input")));
        FileOutputFormat.setOutputPath(job, new Path(conf.get("finalMeans")));

        return job.waitForCompletion(true);
    }
```

3.6 Class Convergence

The `Convergence` class wraps the mapper class, the reducer class and the driver code for the convergence stage.

- `class ConvergenceMapper`
  Parses the data points from file and computes the value of the objective function using the new means. The class is responsible for retrieving the means from the distributed cache and for making them available in memory within an ArrayList (line 14-27); moreover, it stores a `distanceAccumulator` variable (line 7) at Mapper level to keep track of the partial value of the objective function, namely the partial sum of the closest distances between means and points. During every `map()` execution, the closest mean to a point is found and their squared distance is summed to the accumulator (line 35 - 42); at the end of a Mapper execution, the `cleanup()` method emits the `distanceAccumulator` as value, identified by a constant textual key, so that the partial sums are received by a single reducer.

```
1  public static class ConvergenceMapper extends Mapper<LongWritable, Text,
2                                                      Text, DoubleWritable> {
3
4          final static List<Point> means = new ArrayList<>();
5          final static Text outputKey = new Text();
6          final static DoubleWritable outputValue = new DoubleWritable();
7          static double distanceAccumulator;
8
9          public void setup(Context context) throws IOException {
10             Configuration conf = context.getConfiguration();
11             distanceAccumulator = 0.0;
12             means.clear();
13
14             URI[] cacheFiles = context.getCacheFiles();
15             FileSystem fs = FileSystem.get(conf);
16
17             for (URI f: cacheFiles) {
18                 InputStream is = fs.open(new Path(f));
19                 BufferedReader br = new BufferedReader(new InputStreamReader(is));
20
21                 String line;
22                 while ((line = br.readLine()) != null) {
23                     Point mean = new Point(line);
24                     means.add(mean);
25                 }
26                 br.close();
27             }
28         }
29
30         public void map(LongWritable key, Text value, Context context) {
31             double minDistance = Double.POSITIVE_INFINITY;
32
33             Point p = new Point(value.toString());
34
35             for (Point m: means){
```

```
                double d = p.getSquaredDistance(m);
                if (d < minDistance){
                    minDistance = d;
                }
            }

            distanceAccumulator += minDistance;
        }

        public void cleanup(Context context) throws IOException, InterruptedException {
            outputKey.set("key");
            outputValue.set(distanceAccumulator);
            context.write(outputKey, outputValue);
        }
    }
```

- `class ConvergenceReducer`
  Receives the list of partial sums of distances collected by the Mappers; then, it computes the value of the objective function summing them. The `reduce()` method is executed only once by a single reducer, since the Mappers emit a constant key: this ensures that the sum can be computed and emitted collecting the values of a single Iterator.

```
public static class ConvergenceReducer extends Reducer<Text, DoubleWritable,
                                                        NullWritable, DoubleWritable> {

        static double objFunction;
        final static DoubleWritable outputValue = new DoubleWritable();

        public void reduce(Text key, Iterable<DoubleWritable> values, Context context)
            throws IOException, InterruptedException {
            /*
                The single reducer sums up all the distances that it got from the mappers
             */
            objFunction = 0.0;
            for (DoubleWritable value: values)
                objFunction += value.get();

            outputValue.set(objFunction);
            context.write(null, outputValue);
        }
    }
```

- `main(Job job)`
  Prepares the execution of the MapReduce job, setting the necessary classes and input/output types. It deploys a single reducer to avoid idle workers, since the computation is done by a single instance by design.

```java
public static boolean main(Job job) throws IOException, ClassNotFoundException,
                                                        InterruptedException {
        Configuration conf = job.getConfiguration();

        job.setJarByClass(Convergence.class);

        job.setMapperClass(ConvergenceMapper.class);

        job.setReducerClass(ConvergenceReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(DoubleWritable.class);

        job.setOutputKeyClass(NullWritable.class);
        job.setOutputValueClass(DoubleWritable.class);

        FileInputFormat.addInputPath(job, new Path(conf.get("input")));
        FileOutputFormat.setOutputPath(job, new Path(conf.get("convergence")));

        return job.waitForCompletion(true);
    }
```

3.7 Class kMeans

The `kMeans` class is responsible for the orchestration of the program: it coordinates the iterative execution of the jobs representing the subsequent stages of the algorithm, checking if the stop condition is met.

- `cleanWorkspace()`
  Removes the output directories and files of a previous iteration of the algorithm. This is done to avoid runtime exceptions when writing results in directories already used as output.

```java
public static void cleanWorkspace() throws IOException {
        fs.delete(new Path(conf.get("finalMeans")), true);
        fs.delete(new Path(conf.get("convergence")), true);
    }
```

- `copy(Path srcPath, Path dstPath)`
  Utility method used to copy files to the Hadoop Distributed File System, deleting the source ones.

```java
public static void copy(Path srcPath, Path dstPath) throws IOException {
        RemoteIterator<LocatedFileStatus> fileIter = fs.listFiles(srcPath, true);
        while (fileIter.hasNext()){
            FileUtil.copy(fs, fileIter.next(), fs, dstPath, false, true, conf);
        }}
```

- `addCacheDirectory(Path dir, Job job)`
  Adds the files stored in a path of the Hadoop Distributed File System to the distributed cache of a job, so that they can be retrieved efficiently by a Mapper/Reducer.

```java
public static void addCacheDirectory(Path dir, Job job) throws IOException {
        RemoteIterator<LocatedFileStatus> fileIter = fs.listFiles(dir, true);
        while(fileIter.hasNext()) {
            job.addCacheFile(fileIter.next().getPath().toUri());
        }
    }
```

- `getOjectiveFunction()`
  Retrieves the value of the objective function computed in the Convergence stage of the current iteration from the Hadoop Distributed File System, returning it to the caller. Since it is a single value emitted by a single reducer, the path can point directly to a single file `part-r-00000`.

```java
public static double getObjectiveFunction() throws IOException {
        double objFunction;
        InputStream is = fs.open(new Path(conf.get("convergence") + "/part-r-00000"));
        BufferedReader br = new BufferedReader(new InputStreamReader(is));

        String line;
        if ((line = br.readLine()) == null ) {
            br.close();
            fs.close();
            System.exit(1);
        }

        objFunction = Double.parseDouble(line);
        br.close();
        return objFunction;
    }
```

- `main(String[] args)`
  It is responsible for parsing the local configuration file (line 4), which contains the user settings, and use it to initialize the distributed configuration (line 12-21); after the initial random sampling of the means (line 26-30), it enters a loop where the Clustering and Convergence stages are executed until a stop condition is met (line 37-71). The stop condition is achieved when one of the following events happens:

  1. the objective function changes less than a certain threshold percentage with respect to the previous iteration;
  2. a maximum number of iterations is reached.

  The threshold and the maximum number of iterations are set by the user inside the configuration file.

```java
1   public static void main(String[] args) throws InterruptedException, IOException,
2                                                 ClassNotFoundException {
3          conf = new Configuration();
4          LocalConfiguration localConfig = new LocalConfiguration("config.ini");
```

```java
        localConfig.printConfiguration();

        maxNumberOfIterations = localConfig.getMaxNumberOfIterations();
        errorThreshold = localConfig.getErrorThreshold();

        String BASE_DIR = localConfig.getOutputPath() + "/";

        conf.setLong("seed", localConfig.getSeedRNG());
        conf.setInt("d", localConfig.getNumberOfDimensions());
        conf.setInt("k", localConfig.getNumberOfClusters());
        conf.setInt("maxNumberOfReduceTasks",
                    localConfig.getClusteringNumberOfReducers());
        conf.set("input", localConfig.getInputPath());
        conf.set("sampledMeans", BASE_DIR + "sampled-means");
        conf.set("intermediateMeans", BASE_DIR + "intermediate-means");
        conf.set("finalMeans", BASE_DIR + "final-means");
        conf.set("convergence", BASE_DIR + "convergence");

        fs = FileSystem.get(conf);
        fs.delete(new Path(BASE_DIR), true);

        Job sampling = Job.getInstance(conf, "sampling means");
        if ( !Sampling.main(sampling) ) {
            fs.close();
            System.exit(1);
        }

        int step = 0;
        double objFunction = Double.POSITIVE_INFINITY;
        double prevObjFunction;
        double variation;

        do {
            prevObjFunction = objFunction;

            Path srcPath = (step == 0) ? new Path(conf.get("sampledMeans")) :
                                         new Path(conf.get("finalMeans"));
            Path dstPath = new Path(conf.get("intermediateMeans"));

            fs.mkdirs(dstPath);
            copy(srcPath, dstPath);

            cleanWorkspace();

            Job clustering = Job.getInstance(conf, "clustering");
            addCacheDirectory(new Path(conf.get("intermediateMeans")), clustering);
            if ( !Clustering.main(clustering) ) {
                fs.close();
                System.exit(1);
            }

            Job convergence = Job.getInstance(conf, "convergence");
            addCacheDirectory(new Path(conf.get("finalMeans")), convergence);
            if ( !Convergence.main(convergence) ) {
                fs.close();
                System.exit(1);
            }

            objFunction = getObjectiveFunction();

            variation = (prevObjFunction - objFunction)/prevObjFunction * 100;

            System.out.printf("\nSTEP: %d - PREV_OBJ_FUNCTION: %f - OBJ_FUNCTION: %f -
                CHANGE: %.2f%%\n\n", step, prevObjFunction, objFunction, variation);

            step++;
        } while (prevObjFunction == Double.POSITIVE_INFINITY
                || (variation > errorThreshold && step < maxNumberOfIterations));

        fs.close();

    }
}
```

4 Spark

The Spark implementation of *k*-means is written in Python and is based on the design explained in Section 2. Since Spark allows to execute more complex tasks in few lines of code compared to Hadoop, the core of the implementation is a collection of simple functions in `PointUtility.py` and `main.py`, where no explicit serialization/deserialization support is required. A point is represented by means of the *ndarray* class offered by the NumPy library, that grants an efficient and powerful representation of an n-dimensional entity.

4.1 PointUtility.py

The file is a simple wrapper for methods that are exploited within RDD transformations on the data set.

- `parse_point(row)`
  Parses an n-dimensional point given in string format, where the coordinates are separated by the comma (`","`) character, and converts it to an ndarray of float elements:

```
def parse_point(row):
    return np.array(row.split(','), dtype=np.float64)
```

- `get_closest_mean(point, means)`
  The method takes as input an ndarray representing a point and a list of ndarrays representing the means at a certain iteration, and computes the closest mean to that point in terms of Euclidean distance. It returns the mean wrapped inside a tuple, so that it can be hashed by Spark in the following steps, and a tuple `(point, 1)`, where the integer is useful to keep track of the number of points summed in the subsequent stages of the algorithm.

```
def get_closest_mean(point, means):
    squared_distance = np.sum(((np.array(means) - point) ** 2), axis=1)

    # Take the minimum distance (first one in case of multiple equal distances).
    closest_mean_index = np.where(squared_distance == squared_distance.min())[0][0]

    # Tuple used to make the ndarray hashable in reduceByKey()
    return tuple(means[closest_mean_index]), (point, 1)
```

- `sum_point_accumulators(accumulator_1, accumulator_2):`
  It takes as input two tuples with the format `(sum_of_points, number_of_points)`, where `sum_of_points` is an ndarray, and returns a tuple representing the sum member-wise.

```
def sum_point_accumulators(accumulator_1, accumulator_2):
    return accumulator_1[0] + accumulator_2[0], accumulator_1[1] + accumulator_2[1]
```

- `compute_new_mean(accumulator)`
  Computes the new mean of a cluster as the ratio between the sum of its points and its cardinality, where `accumulator` is a tuple with the format `((old_mean), (sum_of_points, number_of_points))`.

```
def compute_new_mean(accumulator):
    new_mean = accumulator[1][0]/accumulator[1][1]
    return new_mean
```

- `compute_min_squared_distance(point, means)`
  The method takes as input an ndarray representing a point and a list of ndarrays representing the means at a certain iteration, and computes the minimum squared distance between them.

```python
def compute_min_squared_distance(point, means):
    squared_distance = np.sum((np.array(means) - point) ** 2), axis=1)
    return squared_distance.min()
```

- `to_string(point)`
  Converts an ndarray representing a point into a string, where the coordinates are separated by the comma (`","`) character.

```python
def to_string(point):
    return np.array2string(point, separator=',')[1:-1].replace(' ', '')
```

4.2 main.py

The file contains the Spark implementation of the *k*-means algorithm: the parsing of the data points, the execution of the algorithm and the result storage are performed with simple functions. The `PointUtility` wrapper is exploited to perform transformations on RDDs.

- `delete_output_file(output_file, spark_context)`
  Deletes the output file of a previous run, if present, to avoid conflicts during the current one.

```python
def delete_output_file(output_file, spark_context):
    Path = spark_context._gateway.jvm.org.apache.hadoop.fs.Path
    FileSystem = spark_context._gateway.jvm.org.apache.hadoop.fs.FileSystem

    fileSystem = FileSystem.get(spark_context._jsc.hadoopConfiguration())
    fileSystem.delete(Path(output_file))
```

- `stop_condition(objective_function, last_objective_function, iteration_number, error_threshold)`
  Checks if the stop condition is met, namely if the value of the objective function changed less than a given threshold with respect to the previous iteration. The threshold is expressed as a percentage.

```python
def stop_condition(objective_function, last_objective_function, iteration_number,
                   error_threshold):
    print("*** Iteration number: " + str(iteration_number + 1) + " ***")
    print("*** Last objective function value: " + str(last_objective_function) + " ***")
    print("*** Current objective function value: " + str(objective_function) + " ***")

    if iteration_number == 0:
        print("*** First iteration: stop condition not checked. ***\n")
        return False

    error = 100*((last_objective_function - objective_function)/last_objective_function)

    print("*** Current error: " + str(error) + "% ***")
    print("*** Error threshold: " + str(error_threshold) + "% ***\n")

    if error <= error_threshold:
        print("*** Stop condition met: error " + str(error) + "% ***\n")
        return True

    return False
```

- `main()`
  Performs the main operations required to execute the *k*-means algorithm. In particular, it parses the local configuration file and initializes the Spark context (line 2-6), it parses the points using a custom format (line 10) and it samples the initial random means, saving them in a local list (line 13). Then, it performs a cycle implementing the Clustering and Convergence stages described in Section 2 (line 19-38). The stop condition is achieved when one of the following events happens:

  1. the objective function changes less than a certain threshold percentage with respect to the previous iteration;
  2. a maximum number of iterations is reached.

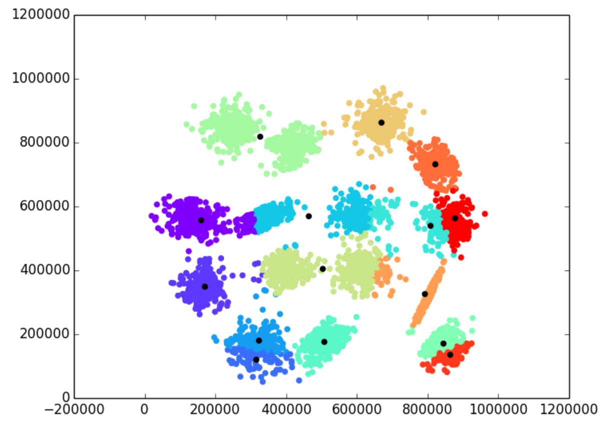
  The threshold and the maximum number of iterations are set by the user inside the configuration file. The data points are cached directly after the initial parsing, since they will be used throughout all the iterations, and the means are broadcast to the workers at every update, so that they can store and use them directly in memory to compute the Euclidean distances when needed.

```
1  def main():
2      config = LocalConfiguration("config.ini")
3      config.print()
4  
5      spark_context = SparkContext(appName="K-Means")
6      spark_context.setLogLevel(config.get_log_level())
7  
8      delete_output_file(config.get_output_path() + "/final-means", spark_context)
9  
10     points_rdd = spark_context.textFile(config.get_input_path())
11                                   .map(PointUtility.parse_point).cache()
12 
13     sampled_means = points_rdd.takeSample(False, config.get_number_of_clusters(),
                                              config.get_seed_RNG())
14 
15     iteration_means = spark_context.broadcast(sampled_means)
16     last_objective_function = float("inf")
17     completed_iterations = 0
18 
19     while completed_iterations < config.get_maximum_number_of_iterations():
20         new_means = points_rdd.map(lambda point: PointUtility.get_closest_mean(point,
                                                                  iteration_means.value))
21                                   .reduceByKey(lambda x,y:
22                                                    PointUtility.sum_point_accumulators(x,y))
23                                   .map(lambda accumulator:
24                                                 PointUtility.compute_new_mean(accumulator))
25                                   .collect()
```

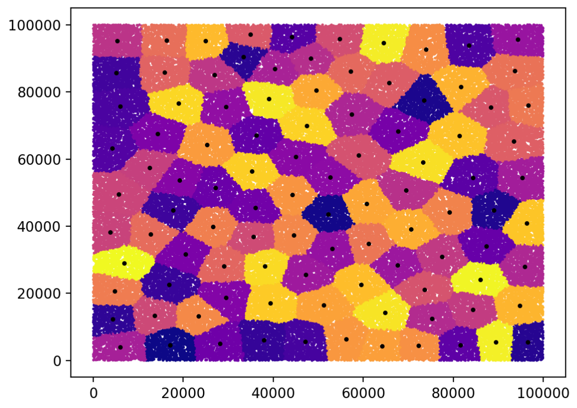
```

        iteration_means = spark_context.broadcast(new_means)
        objective_function = points_rdd.map(lambda point: PointUtility.compute_min_squared_distance(point, iteration_means.value)).sum()

        if stop_condition(objective_function, last_objective_function,
                          completed_iterations, config.get_error_threshold()):
            spark_context.parallelize(new_means).map(PointUtility.to_string)
                         .saveAsTextFile(config.get_output_path() + "/final-means")
            spark_context.stop()
            return

        last_objective_function = objective_function
        completed_iterations += 1

    print("Maximum number of iterations reached: " + str(completed_iterations))
    spark_context.stop()
```

5 Validation

Hadoop and Spark implementations have been validated using 2-dimensional synthetic data sets, both randomly generated and downloaded from the University of Eastern Finland website[1], in order to visualize the results, which are shown in Figure 4. Being a heuristic approach, sometimes, as highlighted in Figure 4a, the results do not resemble the expected ones. The results depend a lot on the initial sampled points and, if the initial choice is not optimal, the algorithm struggles in recovering, especially when the data set exhibits points already split in clusters. Anyhow, if the points are clusterable, subsequent runs with different starting means should exhibit a mean distribution of the centroids that does not suffer from this problem.



(a) Synthetic 2-d data with N=5000 vectors and k=15 Gaussian clusters with different degree of cluster overlap.[1]



(b) Randomly-generated 2-d data with N=100,000 vectors and k=100 clusters.

Figure 4: Visual results of k-means clustering.

[1]`http://cs.joensuu.fi/sipu/datasets/`

6 Performance results

Both the implementations of the k-means algorithm have been tested on a cluster composed of four machines and under different workloads, in order to measure their execution times. In particular, the data sets were randomly generated varying the factors:

1. number of points = $\{1000, 10000, 100000\}$;
2. number of dimensions = $\{3, 7\}$;
3. number of clusters = $\{7, 13\}$.

The experiments have been calibrated so that:

1. each scenario was repeated 30 times with different seeds for the initial random sampling of the means;
2. the error threshold was set to 1%, i.e. the algorithm stopped as soon as the value of objective function changed less than 1% with respect to the previous iteration;
3. for the Hadoop version, the number of reduce tasks for the Clustering stage has been set to be equal to the number of clusters.

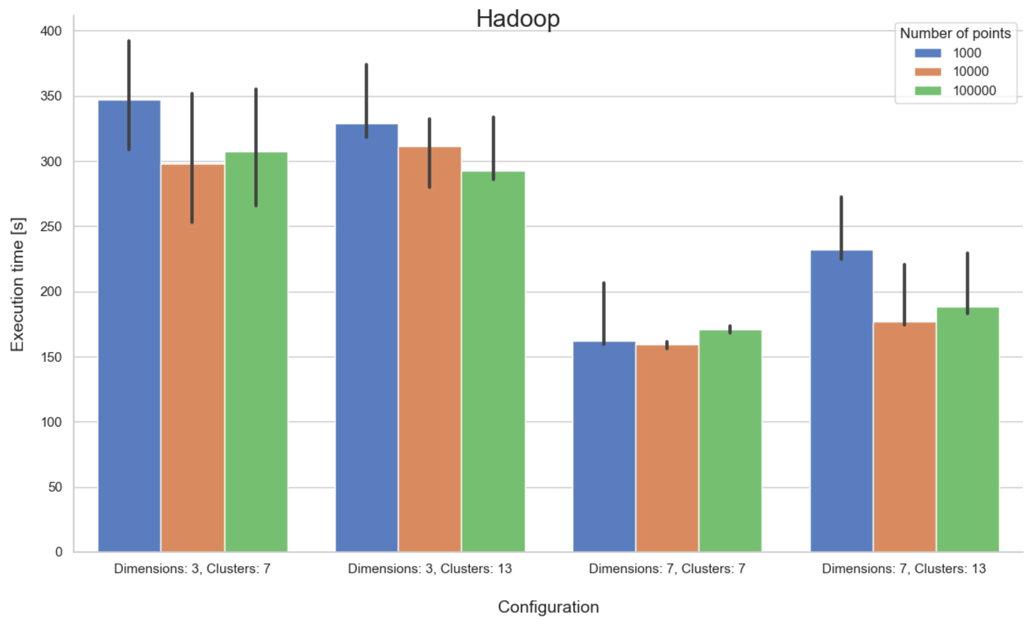


Figure 5: Median of the execution times of Hadoop k-means. The confidence level is 95%.

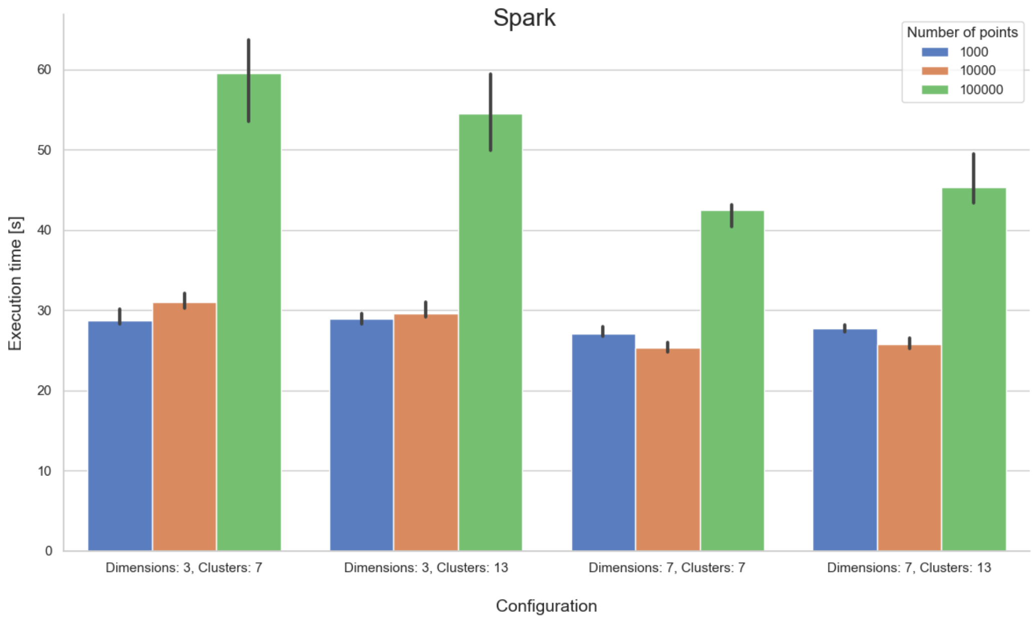


Figure 6: Median of the execution times of Spark *k*-means. The confidence level is 95%.

The two graphs above show clearly how Spark outperforms Hadoop in all the scenarios: the possibility to exploit in-memory computations is fundamental in an algorithm like *k*-means, where multiple iterations are necessary. Since Hadoop leverages the disk to store and retrieve the intermediate results, each iteration will be prolonged by I/O operations, which become a more relevant slow-down factor than the number of points inside the same configuration: the execution times in the Hadoop version don't change greatly with the size of the data set as occurs with Spark. Moreover, this causes a higher variability of the execution times and more uncertainty, as denoted by the larger confidence intervals.

Both the implementations perform better when the dimension of the points increases: in fact, the value of the objective function in these cases tends to be higher and the variation between consecutive iterations quickly drops under the error threshold, even if it is set to 1%. This phenomenon suggests that lower error thresholds should be used in high dimensional scenarios.