PISA UNIVERSITY

CLOUD COMPUTING

HADOOP AND SPARK PROJECTS DOCUMENTATION

ACADEMIC YEAR 2019-2020

STEFANO PETROCCHI, ANDREA TUBAK, FRANCESCO RONCHIERI, ALESSANDRO MADONNA

SUMMARY

DESIGN

DESCRIPTION

The design of the *k-means* algorithm that has been chosen uses a *loop* composed of several executions of the following *map-reduce* scheme, until the **stop criterion** is satisfied:

- **Mapper:** The *mappers* take as ***input*** the **dataset's points** and the initial set of **centroids**. Considering one point of the dataset at a time, the mapper identifies the **closest centroid** to that point and ***outputs*** a *key-value* pair with the **centroid** (or index) as ***key*** and the **point** assigned as ***value***.
- **Combiner:** Each *combiner* receives as ***input*** a **centroid** (or index) with a **subset** of the **points assigned** to it by the mapper, calculates the **partial sum** of these points and outputs a key-value pair with the **centroid** as ***key*** and the **partial sum** as ***value***.
- **Reducer:** Each *reducer* receives a **centroid** (or index) as ***input*** and the set of **all** the **partial sums** of the points assigned to it, calculates the **total sum** of the components of all the points, recalculates the **components** of the **centroid** as the **average of those of the assigned points** and in ***output*** provides simply the **new centroid coordinates**, ready to be used in the next step.

The centroids are **initially randomly selected** from the dataset. At each iteration of the algorithm, the centroids calculated in the previous step are taken as input in the following one.
The **stop criteria** is the cumulative measure of the centroids' movement with respect to the previous step. Generally, the algorithm ends directly with zero movement, but it is possible to set less restrictive stop criteria if desired (although it is important to highlight that the decreasing monotonicity of the value is not guaranteed). For safety, the algorithm also stops if the maximum number of iterations is reached. This criterion was chosen instead of the difference between the mean square distance of the points from the centroids between the iterations as it is simpler to implement and perfectly equivalent in cases of convergence at local minimum.

PSEUDOCODE

MAPPER FUNCTION

Input: The **Centroids** as ***global variable***, the **point's offset** as ***key*** and the **point** as ***value***
Output: The **nearest centroid** as ***key***, the **point** as ***value***
Pseudocode:

```
mapper(in_key, in_value):
    The point is constructed from the in_value;
    The centroids_list is constructed from the global variable;
    min_distance = MAX_VALUE;
    ∀ centroid in centroids_list do:
        distance = compute_distance(centroid, point)
        if distance < min_distance then:
            min_distance = distance;
            out_key = centroid;
```

out_value = *point* ;
output <*out_key*, *out_value*> ;

end

→ **compute_distance**(*p1*, *p2*) returns the distance between the two points

COMBINER FUNCTION

Input: The **centroid** as ***key*** and a **subset of points** as ***values list***
Output: The **centroid** as ***key***, the **partial sum** as ***value***
Pseudocode:

combiner(*in_key*, *in_values*)**:**

out_key = *in_key* ;
The *points_list* is constructed from the *in_values* ;
partial_sum is initialized ;
∀ *point* **in** *points_list* **do**:
    partial_sum.***sum(****point****)*** ;
out_value = *partial_sum* ;
output <*out_key*, *out_value*> ;

end

→ *partial_sum*.***sum(****p****)*** *sum the components of p to partial_sum and increase properly the counter of the summed points*

REDUCER FUNCTION

Input: The **centroid** as ***key*** and the **set of partial sums** as ***values list***
Output: The **new centroid** as ***output*** (key and value depend on the implementation)
Pseudocode:

reducer(*in_key*, *in_values*)**:**

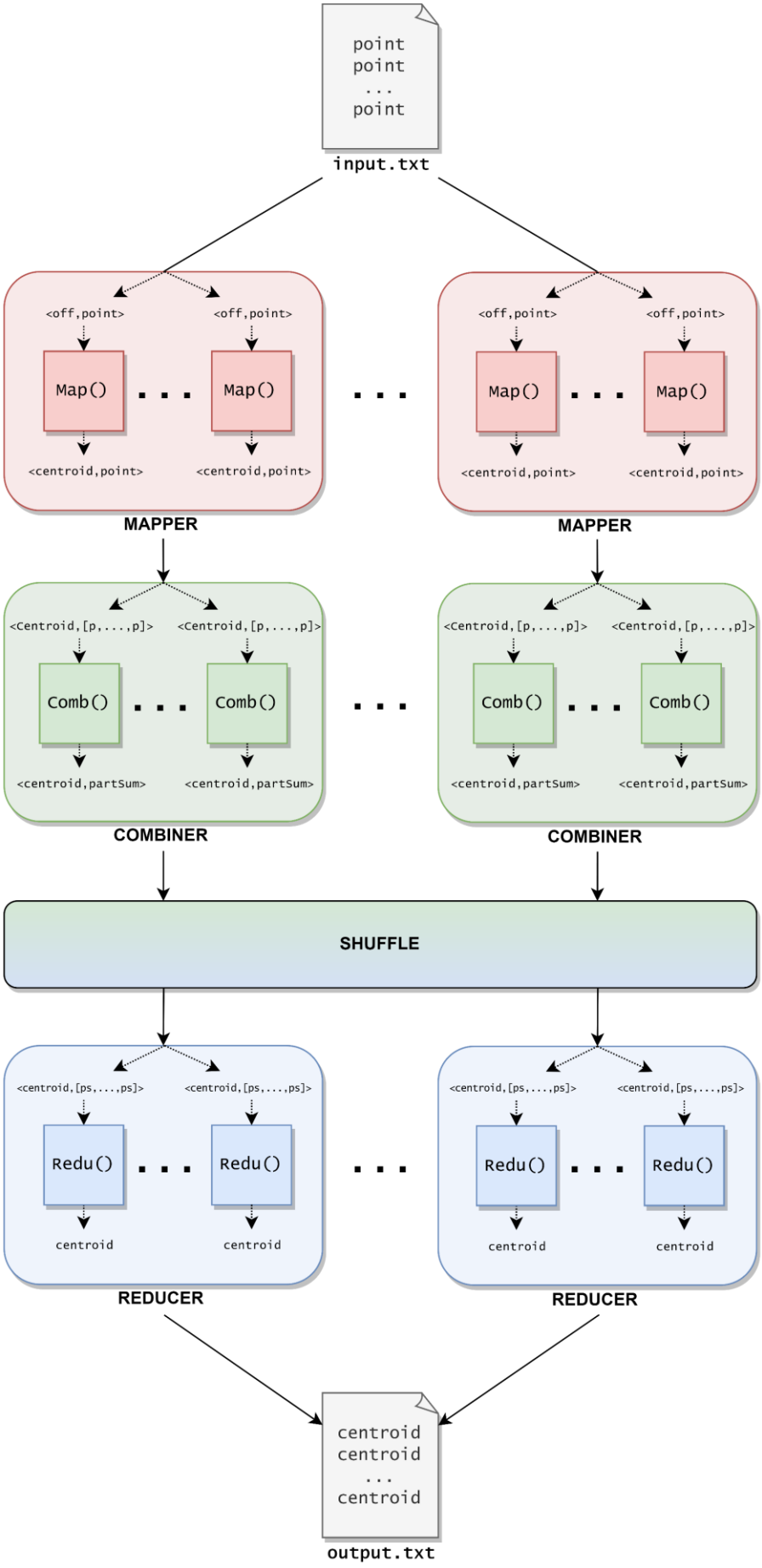
old_centroid = *in_key* ;
The *partial_sums_list* is constructed from the *in_values* ;
new_centroid is initialized ;
∀ *partial_sum* **in** *partial_sums_list* **do**:
    new_centroid.***sum(****partial_sum****)*** ;
new_centroid.idex = *old_centroid.index* ;
new_centroid.***calc_barycenter()*** ;
output *new_centroid* ;

end

→ *new_centroid*.***sum(****p****)*** *sum the components of p to those of new_centroid and increase properly the summed points' counter*
→ *new_centroid*.***calc_barycenter()*** *divides the value of the new_centroid components by the value of the summed points' counter*

HADOOP IMPLEMENTATION

JOB SCHEMA



DESCRIPTION

The *Hadoop* implementation of the *k-means* algorithm was developed in *Java* using version 3.1.3 of the *Hadoop* library for *Java.*

The code is composed by two classes, ***Point*** and ***KMeans***:

- **Point:** is the object class that represents **points** and **centroids**, implements both ***WritableComparable*** and ***Serializable*** in order to allow its serialization both to be passed as a *configuration field*, and as a *key*, in the case of centroids, or *value*, in the case of points. It contains a list of doubles which represent the *coordinates* of the point and a series of utility functions and fields.
- **KMeans:** it contains the subclasses of ***mapper***, ***combiner*** and ***reducer*** as well as the ***driver*** necessary to start the algorithm and the *map-reduce* jobs.

POINT

Below is the complete and commented code:

```
1.  public class Point implements WritableComparable, Serializable {
2.
3.      public ArrayList<Double> components = new ArrayList<>();
4.      public int summedPoints = 1;
5.      public int index = -1;
6.      public int dimensions = 1;
```

line 3 → `ArrayList<Double> components` is the list of coordinates that identify the point.
line 4 → `summedPoints` is the counter of the number of points of which the components are the sum.
line 5 → `index` contains the centroid index or "-1" in the case of a regular point.
line 6 → `dimensions` is the number of coordinates that identify the point.

```
7.      public static double distance(int distanceType, Point P1, Point P2) {
8.          double sum = 0;
9.          int index = 0;
10.         for (double componentP1 : P1.components) {
11.             double componentP2 = P2.components.get(index);
12.             sum += Math.pow((componentP1 - componentP2), distanceType);
13.             index++;
14.         }
15.         return Math.pow(sum, 1.0 / distanceType);
16.     }
```

lines[7-16]→ `double distance(int distanceType, Point P1, Point P2)` calculate the distance of type `distanceType` (Euclidean, Manhattan ...) between the point `P1` and `P2`, it is a *static* function. For efficiency reasons it can be thought to return only `sum` instead of the root.

```
17.     public void sum(Point P) {
18.         int indexP = 0;
19.         for (double componentToSum : P.components) {
20.             components.set(indexP, components.get(indexP) + componentToSum);
21.             indexP++;
22.         }
23.         summedPoints += P.summedPoints;
24.     }
```

lines[17-24]→ `void sum(Point P)` sums all the components of point `P` to the components of the point on which this function is called. On line **23** it is not just an increment by one operation because the point `P` may be a point used to express the partial sum of many other points.

```
25.     void set(Point P) {
26.         components.clear();
27.         P.components.forEach((component) -> {
28.             components.add(component);
29.         });
30.         index = P.index;
31.         dimensions = P.dimensions;
32.         summedPoints = P.summedPoints;
33.     }
```

lines[25-33]→ `void set(Point P)` implements a *deep copy* of point `P` on the parameters of the point on which the function is called.

```
34.     public void computeAndSetBarycenter() {
35.         int pointIndex = 0;
36.         for (double component : components) {
37.             components.set(pointIndex, component / summedPoints);
38.             pointIndex++;
39.         }
40.         summedPoints = 1;
41.     }
```

lines[34-41]→ `void computeAndSetBarycenter()` replaces the components of the point with their average calculated on the number of points that have been added together given by `summedPoints`.

```
42.     public static Point deserialize(String s) throws IOException, ClassNotFoundException {

43.         byte[] data = Base64.getDecoder().decode(s);
44.         Point point;
45.         try (ObjectInputStream ois = new ObjectInputStream(
46.                 new ByteArrayInputStream(data))) {
47.             point = (Point) ois.readObject();
48.         }
49.         return point;
50.     }
```

lines[42-50]→ `Point deserialize(String s)` starting from a *string* which is the serialized version of the class, it reconstructs an instance of `Point` and returns it. It is a *static* function.

```
51.     public String serialize() {
52.         ByteArrayOutputStream baos = new ByteArrayOutputStream();
53.         try (ObjectOutputStream oos = new ObjectOutputStream(baos)) {
54.             oos.writeObject(this);
55.         } catch (IOException ex) {
56.             System.err.println("Error in converting to string: " + ex.getMessage());
57.         }
58.         return Base64.getEncoder().encodeToString(baos.toByteArray());
59.     }
```

lines[51-59]→ `String serialize()` serializes the current instance of the class in the form of a string.

```
60.     @Override
61.     public String toString() {
62.         return "idex:" + index + "-" + "summedPoints:" + summedPoints + "-"
                   + "dimensions:" + dimensions + "-"
                   + components.toString().replaceAll(" ", "");
63.     }
```

lines[60-63]→ `String toString()` *human-readable* representation of the point.

```
64.     @Override
65.     public void write(DataOutput out) throws IOException {
66.         out.writeInt(index);
67.         out.writeInt(summedPoints);
68.         out.writeInt(dimensions);
69.         for (int i = 0; i < dimensions; i++) {
70.             out.writeDouble(components.get(i));
71.         }
72.     }
```

lines[65-72]→ `void write(DataOutput out)` allows the *framework* to *serialize* the information contained in this point instance, override requested by `WritableComparable` in order to pass the object as *value*.

```
73.     @Override
74.     public void readFields(DataInput in) throws IOException {
75.         components.clear();
76.         index = in.readInt();
77.         summedPoints = in.readInt();
78.         dimensions = in.readInt();
79.         for (int i = 0; i < dimensions; i++) {
80.             components.add(in.readDouble());
81.         }
82.     }
```

lines[74-82]→ `void readFields(DataInput in)` deserializes and reconstructs the point instance, override requested by `WritableComparable`.

```
83.     @Override
84.     public int compareTo(Object o) {
85.         Point p = (Point) o;
86.         return (this.index < p.index ? -1 : (this.index == p.index ? 0 : 1));
87.     }
```

lines[84-87]→ `int compareTo(Object o)` override requested by `WritableComparable` in order pass this object as a *key* and be able to compare it with the other *keys*.

```
88.     @Override
89.     public int hashCode() {
90.         return hash(components, index, summedPoints, dimensions);
91.     }
```

lines[89-91]→ `int hashCode()` is often used by the *framework* for some operations on *keys* and *values*, it calculates the combined *hash* of the instance fields.

```
92.     @Override
93.     public boolean equals(Object obj) {
94.         if (this == obj) {
95.             return true;
96.         }
97.         if (obj == null) {
98.             return false;
99.         }
100.        if (getClass() != obj.getClass()) {
101.            return false;
102.        }
103.        final Point other = (Point) obj;
104.        if (this.summedPoints != other.summedPoints) {
105.            return false;
106.        }
```

```
107.         if (this.index != other.index) {
108.             return false;
109.         }
110.         if (this.dimensions != other.dimensions) {
111.             return false;
112.         }
113.         return this.components.equals(other.components);
114.     }
115.
116. }
```

lines[93-114]→ `boolean equals(Object obj)` allows to check if this object is the same as another object.

KMEANS

The *driver*, *mapper*, *combiner* and *reducer* will be described one by one.

DRIVER

Below is described the *driver* code, some irrelevant parts have been cut for the sake of brevity.

```
1.  public static void main(String[] args) throws Exception {
2.
3.      Configuration conf = new Configuration();
4.      conf.addResource(new Path("/opt/hadoop/etc/hadoop/core-site.xml"));
5.      conf.addResource(new Path("/opt/hadoop/etc/hadoop/hdfs-site.xml"));
6.      String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
```

lines[3-5]→ Initialization of the configuration with the information necessary to access the *HDFS.*
line 6 → Retrieval of the parameters.

```
7.      int k = Integer.parseInt(otherArgs[2]);
8.      int d = Integer.parseInt(otherArgs[1]);
9.      double stopCriteria = Double.valueOf(otherArgs[3]);
10.     int maxIterations = Integer.parseInt(otherArgs[4]);
11.     double centroidsMovementFactor = stopCriteria + 1;
```

lines[7-8]→ Parameters `k` (clusters number) and `d` (point dimensions) are initialized.
lines[9-11]→ `stopCriteria` is the movement value of the *centroids* under which the iterations are stopped, `maxIterations` instead, is the maximum number of iterations for safety reasons. `centroidsMovementFactor` collects the movement value of the *centroids* from the previous iteration and is initialized as greater than `stopCriteria`.

```
12.     ArrayList<Point> centroids = new ArrayList<>();
13.     Path path = new Path(otherArgs[0]);
14.     FileSystem fs = path.getFileSystem(conf);
15.     String currentLine = null;
16.     ArrayList<String> reservoirList = new ArrayList<>(k);
17.     int count = 0;
18.     Random ra = new Random();
19.     int randomNumber = 0;
20.     Scanner sc = new Scanner(fs.open(path)).useDelimiter("\n");
21.
22.     while (sc.hasNext()) {
23.         currentLine = sc.next();
24.         count++;
25.         if (count <= k) {
26.             reservoirList.add(currentLine);
27.         } else if ((randomNumber = (int) ra.nextInt(count)) < k) {
28.             reservoirList.set(randomNumber, currentLine);
```

```
29.         }
30.     }
31.
32.     for (int centroidIndex = 0; centroidIndex < k; centroidIndex++) {
33.         Point centroid = new Point();
34.         centroid.index = centroidIndex;
35.         centroid.dimensions = d;
36.         for (String component : reservoirList.get(centroidIndex).split(",")) {
37.             centroid.components.add(Double.valueOf(component));
38.         }
39.         centroids.add(centroid);
40.     }
```

lines[12-40]→ Code based on the *reservoir sampling* to randomly select *k* lines from a large file and thus initialize the *centroids* with their values. The file containing the *dataset* is opened from the *HDFS* and is read one line at a time, the first *k* lines are initially entered in the *reservoir list*, the subsequent lines are selected with a gradually decreasing probability in a random position of the list, the result is a uniform sampling.

```
41.     for (int jobIndex = 0; jobIndex < maxIterations &&
             centroidsMovementFactor > stopCriteria; ++jobIndex) {
```

line 41 → The algorithm is executed by looping several *map-reduce* iterations for the calculation of the new coordinates of the *centroids*, the loop stops when the maximum number of iterations has been reached or the cumulative movement of the *centroids* from the previous step is below the established threshold.

```
42.         Job job = Job.getInstance(conf, "KMeans");
43.         job.getConfiguration().set("d", otherArgs[1]);
44.         job.getConfiguration().set("k", otherArgs[2]);
45.         centroids.forEach((centroid) -> {
46.             job.getConfiguration().set("centroid-" + centroid.index,
                                           centroid.serialize());
47.         });
```

lines[42-47]→ The *configuration* object is passed to the *job* together with the parameters and the *centroids* calculated in the previous step or randomly chosen from the *dataset* if it is the first iteration. To be passed as configuration fields, *centroids* must be *serialized* in *strings* using the appropriate function.

```
48.         job.setJarByClass(KMeans.class);
49.         job.setMapperClass(KMeansMapper.class);
50.         job.setCombinerClass(KMeansCombiner.class);
51.         job.setReducerClass(KMeansReducer.class);
52.         job.setMapOutputKeyClass(Point.class);
53.         job.setMapOutputValueClass(Point.class);
54.         job.setOutputKeyClass(Point.class);
55.         job.setOutputValueClass(Text.class);
56.         FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
57.         FileOutputFormat.setOutputPath(job, new Path(otherArgs[5] + "_" + jobIndex));
58.         job.setInputFormatClass(TextInputFormat.class);
59.         job.setOutputFormatClass(TextOutputFormat.class);
```

lines[48-59]→ The *map-reduce* iteration is started by passing the appropriate fields to the *job*.

```
60.         System.out.println("INFO | Job " + (jobIndex + 1) + " completato con: "
                               + job.waitForCompletion(true));
```

line 60 → The *driver* waits for the execution of the job to finish.

```
61.         path = new Path(otherArgs[5] + "_" + jobIndex);
62.         fs = path.getFileSystem(conf);
63.         FileStatus[] fss = fs.listStatus(path);
64.         boolean first = true;
65.
66.         for (FileStatus status : fss) {
67.             if (!first) {
68.                 path = status.getPath();
69.                 BufferedReader bufRead = new BufferedReader(
                                                new InputStreamReader(fs.open(path)));
70.                 String line = "";
71.                 while (line != null) {
72.                     line = bufRead.readLine();
73.                     if (line != null && !"null".equals(line)) {
74.                         Point newCentroid = Point.deserialize(line.split("\\s+")[1]);
75.                         centroids.set(newCentroid.index, newCentroid);
76.                     }
77.                 }
78.             } else {
79.                 first = false;
80.             }
81.         }
```

lines[61-81]→ The *centroids* resulting from the concluded iteration are recovered by accessing the *job* output folder and *deserializing* the string that represent them with the appropriate function. To do this, all the files present in the *output* folder in the *HDFS* minus the first must be opened and read line by line.

```
82.         fs.delete(new Path(otherArgs[5]), true);
83.         fs.rename(new Path(otherArgs[5] + "_" + jobIndex), new Path(otherArgs[5]));
```

lines[82-83]→ To avoid creating too many output folders, the output of the previous step is deleted while the current one is renamed to correspond to the parameter passed from the command line.

```
84.         centroidsMovementFactor = 0;
85.         for (int centroidIndex = 0; centroidIndex < k; centroidIndex++) {
86.             Point oldCentroid = Point.deserialize(job.getConfiguration().get(
                                        "centroid-" + centroidIndex));
87.             centroidsMovementFactor += Point.distance(distanceType,
                                            oldCentroid, centroids.get(centroidIndex));
88.         }
89.     }
90. }
```

lines[84-90]→ The cumulative movement of the *centroids* from the previous step is calculated in order to be used as *stop criterion*.

MAPPER

Below is described the *mapper* code, some irrelevant parts have been cut for the sake of brevity.

```
1.  public static class KMeansMapper extends Mapper<LongWritable, Text, Point, Point> {
2.
3.      private final Point outputKey = new Point();
4.      private final Point outputValue = new Point();
```

lines[3-4]→ **IMPORTANT:** the `Point` class is used also as a *key* as an exercise on the implementation of a `WritableComparable` class, otherwise the only use of a `Point` class that extended

`Writable` and the passage of the *centroids* as index, would be enough, given that the other information are superfluous for the *combiner* and *reducer*.

```
5.      @Override
6.      public void map(LongWritable key, Text value, Context context)
                    throws IOException, InterruptedException {
```

`line 6` → Each *mapper* takes as input a single line of the input file which is equivalent to a single point of the *dataset*.

```
7.          Configuration conf = context.getConfiguration();
8.          ArrayList<Point> centroids = new ArrayList<>();
9.          int k = Integer.parseInt(conf.get("k"));
10.         int d = Integer.parseInt(conf.get("d"));
11.
12.         for (int index = 0; index < k; index++) {
13.             try {
14.                 centroids.add(index, Point.deserialize(conf.get("centroid-" + index)));
15.             } catch (ClassNotFoundException ex) {
16.                 System.err.println("A problem occurred in passing the centroids: "
                                       + ex.getMessage());
17.             }
18.         }
```

`lines[8-11]`→ The parameters `k` and `d` are taken from the configuration.
`lines[13-18]`→ *Centroids* are retrieved as *strings* from the configuration and deserialized into `Point` objects.

```
19.         String line = value.toString();
20.         Point point = new Point();
21.         for (String component : line.split(",")) {
22.             point.components.add(Double.valueOf(component));
23.         }
24.         point.dimensions = d;
```

`lines[19-24]`→ The dataset point is reconstructed from the input file line passed to the mapper.

```
25.         double minDistance = Double.MAX_VALUE;
26.         for (Point centroid : centroids) {
27.             double distance = Point.distance(distanceType, centroid, point);
28.             if (distance < minDistance) {
29.                 minDistance = distance;
30.                 outputKey.set(centroid);
31.             }
32.         }
```

`lines[25-32]`→ The distance between the point and each *centroid* is calculated by using the appropriate function and keeping the closest one saved.

```
33.         outputValue.set(point);
34.         context.write(outputKey, outputValue);
35.
36.     }
37. }
```

`lines[33-34]`→ The *centroid* is written in *context* as *key* and the point assigned to that *centroid* as *value*.

COMBINER

Below is described the *combiner* code, some irrelevant parts have been cut for the sake of brevity.

```
1.  public static class KMeansCombiner extends Reducer<Point, Point, Point, Point> {
2.
3.      private final Point outputKey = new Point();
4.      private final Point outputValue = new Point();
5.
6.      @Override
7.      public void reduce(Point key, Iterable<Point> points, Context context)
                    throws IOException, InterruptedException {
8.
9.          outputKey.set(key);
```

lines[7-9]→ The *combiner* receives in input a subset of the set of points assigned to the *centroid* that receives as *key*, the latter is passed directly to the output.

```
10.         boolean first = true;
11.         Point point = new Point();
12.         Point sumPoint = new Point();
13.         for (Point p : points) {
14.             point.set(p);
15.             if (first == true) {
16.                 sumPoint.set(point);
17.                 first = false;
18.             } else {
19.                 sumPoint.sum(point);
20.             }
21.         }
```

lines[10-21]→ We scan through the set of points provided as input while simultaneously calculating the sum of the components using the appropriate function. The partial sum is saved in a `Point` object which keeps also the information on the number of points that have been summed.

```
22.         outputValue.set(sumPoint);
23.         context.write(outputKey, outputValue);
24.     }
25. }
```

lines[22-23]→ In output is given the *centroid* as *key* and the partial sum of the points as *value*.

REDUCER

Below is described the *reducer* code, some irrelevant parts have been cut for the sake of brevity.

```
1.  public static class KMeansReducer extends Reducer<Point, Point, Point, Text> {
2.
3.      private final Point outputKey = new Point();
4.      private final Text outputValue = new Text();
5.
6.      @Override
7.      public void reduce(Point key, Iterable<Point> points, Context context)
                    throws IOException, InterruptedException {
8.
9.          outputKey.set(key);
10.
11.         Point newCentroid = new Point();
12.         Point point = new Point();
13.         boolean first = true;
14.         for (Point p : points) {
15.             point.set(p);
16.             if (first == true) {
```

```
17.                 newCentroid.set(point);
18.                 newCentroid.index = key.index;
19.                 first = false;
20.             } else {
21.                 newCentroid.sum(point);
22.             }
23.         }
```

lines[1-23]→ Like the *combiner*, in the first part of the reducer the sum of the points passed as input is calculated, the difference lies in the fact that all the partial sums of the points assigned to the centroid are added in order to obtain the total sum of the components, in addition, at the point containing the sum is assigned the *index* of the centroid passed as a key.

```
24.         newCentroid.computeAndSetBarycenter();
```

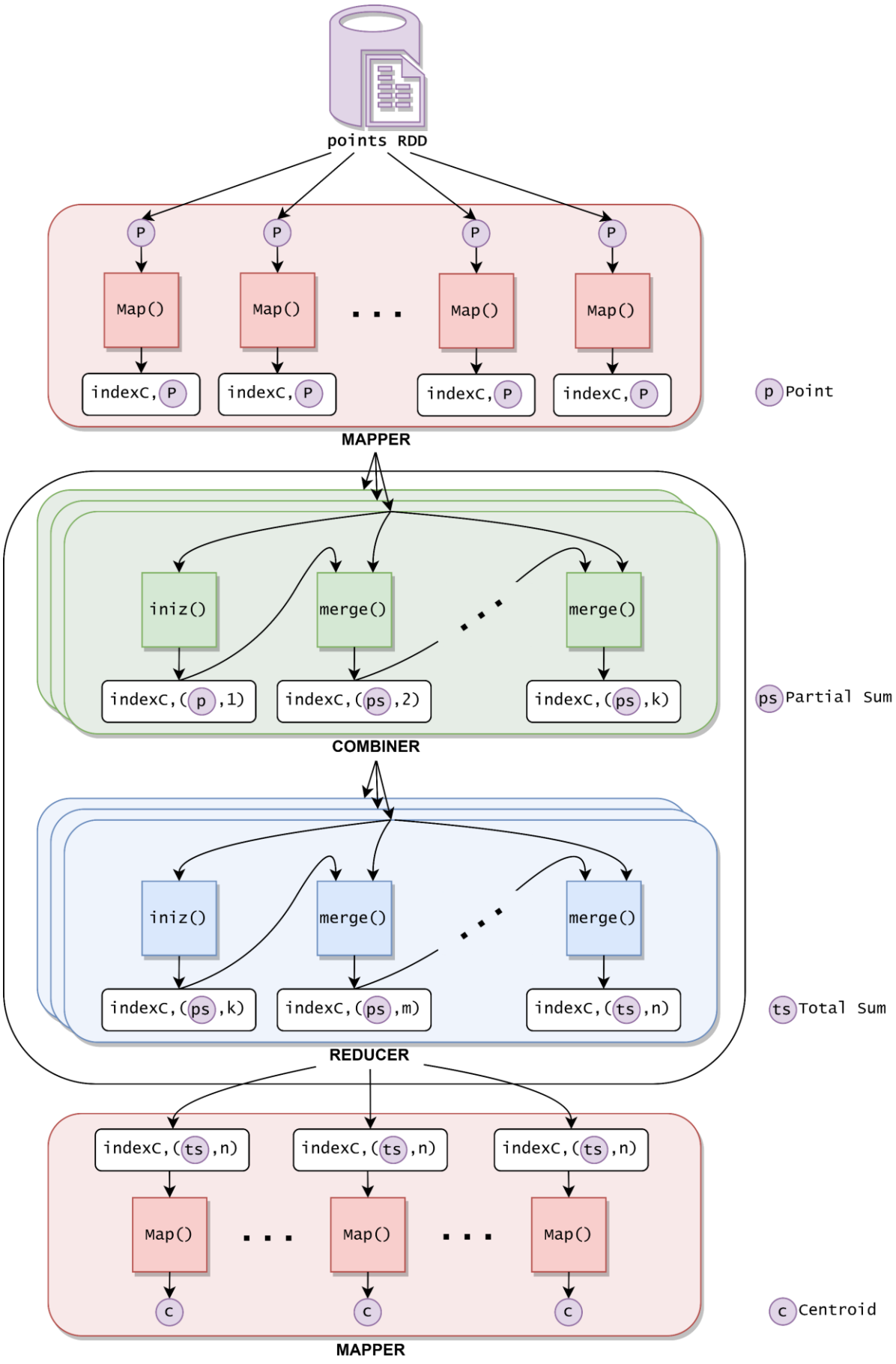
line 24 → Using the sum of the components of the points assigned to the *centroid* and the information on the number of summed points, the function calculates the new coordinates of the centroid.

```
25.         outputValue.set(newCentroid.serialize());
26.         context.write(outputKey, outputValue);
27.     }
28. }
```

lines[25-26]→ In output is given the *centroid* as *key* and its *serialized* representation in form of *string* as *value*, in this way the centroid can be easily reconstructed by the *driver* and used for the following iteration.

SPARK IMPLEMENTATION

The *Spark* implementation of the *k-means* algorithm was developed in *Python* using *PySpark,* the code consists of a single script*: kmeans.py.* Below is a schema of an iteration:



CODE

Below is described the *driver* code, some irrelevant parts have been cut for sake of brevity.

FUNCTIONS

Description of the functions following the structure of the code.

```
1.  import random
2.  import numpy as np
3.  from pyspark import SparkContext
4.
5.  def createPoint(line):
6.      lineSplit = line.split(",")
7.      if len(lineSplit) != pointsDimensions:
8.          raise ValueError("Wrong Point Dimension")
9.      point = np.array(lineSplit).astype(np.float)
10.     return point
```

lines[5-10]→ `createPoint(line)` takes a text line input, divides the line into a list of strings on "," and ensures that the right number of components are present, after which it transforms the list into a *NumPy* array to facilitate the next calculations.

```
11. def mapFunction(point):
12.     minDistance = float("inf")
13.     nearestCentroidIndex = 0
14.     index = 0
15.     for centroid in broadcastCentroids.value:
16.         distance = np.linalg.norm(point - centroid)
17.         if distance < minDistance:
18.             minDistance = distance
19.             nearestCentroidIndex = index
20.         index += 1
21.     return (nearestCentroidIndex, point)
```

lines[11-21]→ `mapFunction(point)` is the function that implements the mapper. It receives one point (at a time), for each *centroid* it calculates the distance between the point and it, memorizing the closest one, returns a tuple with the *centroid index* and the associated point.

```
22. def createCombiner(point):
23.     return (point, 1)
```

lines[22-23]→ `createCombiner(point)` is the first of the functions used by the *combiner*, receives in input a point of the subset of points associated with a *centroid* and transforms it into the first value of the **accumulator** that will be used in `mergeValue(partialSum, point)` function to calculate the *partial sum* of the subset.

```
24. def mergeValue(partialSum, point):
25.     return (partialSum[0] + point, partialSum[1] + 1)
```

lines[24-25]→ `mergeValue(partialSum, point)` is the second function used by the combiner, receives in input the accumulator (`partialSum`) and a point that will be added to it in order to calculate the *partial sum*, the first value of the returned tuple is the *partial sum* of those points assigned to a same centroid, the second is the number of points currently added up.

```
26. def mergeCombiner(partialSum1, partialSum2):
27.     return (partialSum1[0] + partialSum2[0], partialSum1[1] + partialSum2[1])
```

lines[26-27]→ `mergeCombiner(partialSum1, partialSum2)` is the last function used by the *combiner* and actually implements part of the *reducer*. It takes in input two *partial sums* of points assigned to the same *centroid* and returns the union, adding the two fields of the tuples together. The first input value is used as an *accumulator*, at the end of the *combiner's* operations all the *partial sums* of points belonging to the same *cluster* will be added, obtaining the total sum of the components and the total number of points of that *cluster*.

```
28. def totalDistanceOldNewCentroids(oldCentroids, newCentroids):
29.     totalDistance = 0.0
30.     for i in range(0,len(oldCentroids)):
31.         distance = np.linalg.norm(oldCentroids[i] - newCentroids[i])
32.         totalDistance += distance
33.     return totalDistance
```

lines[28-33]→ `totalDistanceOldNewCentroids(oldCentroids, newCentroids)` takes as input the two lists respectively of the centroids of the previous step and of those resulting from the current step, calculates the distances in pairs of centroids with the same index and returns the distances' sum.

MAIN

Description of the *main* following the structure of the code.

```
34. if __name__ == "__main__":
35.
36.     inputFilePath = sys.argv[1]
37.     pointsDimensions = int(sys.argv[2])
38.     numberOfCentroids = int(sys.argv[3])
39.     stopCriteriaMargin = float(sys.argv[4])
40.     maxIterations = int(sys.argv[5])
41.     outputPath = sys.argv[6]
```

lines[36-41]→ The input parameters are received from the command line; their meaning is the same of those of the *Hadoop* code.

```
42.     master = "yarn"
43.     sc = SparkContext(master, "KMeans")
```

lines[42-43]→ *Yarn* cluster management technology is selected and the *context* is created.

```
44.     points = sc.textFile(inputFilePath).map(createPoint).cache()
```

line 44 → The input file is loaded and transformed line by line in *NumPy* arrays representing the points, these are then made **persistent** in memory in order to reuse them in the various iterations.

```
45.     oldCentroids  = points.takeSample(withReplacement = False,
                          num = numberOfCentroids, seed = random.randrange(sys.maxsize))
46.     broadcastCentroids = sc.broadcast(oldCentroids)
```

lines[45-46]→ The *centroids* are chosen through a sampling without replacement of the points and then loaded into the *context* as *broadcast* variables.

```
47.     centroidsMovementMargin = sys.maxsize
48.
49.     for iteration in range(0, maxIterations):
50.
51.         mappedPoints = points.map(mapFunction)
```

line 51 → Runs the mapper on all dataset's points.

```
52.         combinedPoints = mappedPoints.combineByKey(createCombiner, mergeValue,
                                                        mergeCombiner)
```

line 52 → Use the three functions defined above to calculate the components' total sum of the points assigned to each *centroid* and the number of points added together. **Important**: The choice of `combineByKey` over `reduceByKey` is done because the input *type* and output *type* are not the same.

```
53.         indexesAndCentroids = combinedPoints.map(lambda x:(x[0],x[1][0]/x[1][1]))
```

line 53 → Concludes the operations to be performed by the *reducer* dividing the components of the *total sums* (`x[1][0]`) by the number of points added together (`x[1][1]`), thus obtaining the *coordinates* of the *new centroid* and providing the centroid index (`x[0]`) unchanged.

```
54.         newCentroids = [centroid for centroid in oldCentroids]
55.         for indexAndCentroid in indexesAndCentroids.collect():
56.             newCentroids[indexAndCentroid[0]] = indexAndCentroid[1]
```

lines[54-56]→ The centroids are sorted to respect the order given by the indices and allow the subsequent comparison with the old coordinates. To avoid ignoring any centroids not taken into consideration by the *mapper* (because no point was close to them) a list containing the old coordinates is used for inserting the new values.

```
57.         centroidsMovementMargin = totalDistanceOldNewCentroids(oldCentroids,
                                                                   newCentroids)
58.         if centroidsMovementMargin <= stopCriteriaMargin:
59.             break
```

lines[57-59]→ The movement margin of the *centroids* from the values in the previous step is calculate by the function `totalDistanceOldNewCentroids(oldCentroids, newCentroids)`, if it is less than or equal to the threshold (`stopCriteriaMargin`) the iterations are concluded.

```
60.         broadcastCentroids.destroy()
61.         broadcastCentroids = sc.broadcast(newCentroids)
62.         oldCentroids = newCentroids
```

lines[60-62]→ Old broadcasted *centroids* are destroyed and the new ones are loaded into context.

```
63.     indexesAndCentroids.saveAsTextFile(outputPath)
```

line 63 → Clusters *centroids* are saved in the output file at the end of the algorithm execution.

IMPLEMENTATION TEST

DESIGN

In order to properly test the implementations of the *K-Means* clustering algorithm, we decided to run different times the algorithm with some datasets and to collect some data about the performances. In particular, the datasets were made up of points organized in a number of clusters equal to that provided as input to the algorithm, in order to have greater realism instead of using datasets with points evenly distributed. This procedure was executed **10 times** for all possible combination of the following parameters:

- **K** = 7, or 13
- **D** = 3, or 7
- **N** = 1 000, or 10 000, or 100 000

Where **N** is the number of *samples*, **K** is the number of *clusters*, and **D** is the *dimensions* of the points. To generate the datasets for each batch of ten executions, we used a *python* script implemented through the *SKLearn* library.

From each execution we collected three values:

- **Number** of **seconds** elapsed to complete **all** the **iterations**
- **Number** of **iterations** required
- **Average time** required to complete a single **iteration**

We then proceeded to calculate the **average** and the **standard deviation** of the values obtained in the ten executions performed with the same parameters (*average* ± *standard deviation*).

HADOOP

RESULTS

The following tables summarize the results:

D= 3, K = 7

N° samples	Avg. Total time (s)	Avg. N° of jobs	Avg. of Avg. Job time (s)
1 000	184.3 ± 54.4	7.6 ± 2	24.2 ± 2.2
10 000	461.4 ± 128.5	19 ± 5.1	24.9 ± 0.6
100 000	2764.4 ± 862.6	79.6 ± 24.6	34.7 ± 0.4

D= 3, K = 13

N° samples	Avg. Total time (s)	Avg. N° of jobs	Avg. of Avg. Job time (s)
1 000	320.3 ± 118.3	13.6 ± 5.3	23.7 ± 0.7
10 000	743.5 ± 215.7	29 ± 9.1	25.9 ± 1.2
100 000	2857.8 ± 899.5	65.4 ± 20.8	43.8 ± 0.8

D= 7, K = 7

N° samples	Avg. Total time (s)	Avg. N° of jobs	Avg. of Avg. Job time (s)
1 000	196.1 ± 91.5	8.8 ± 4.2	22.4 ± 0.5
10 000	809.9 ± 538.3	31.1 ± 20	26.3 ± 3.2
100 000	2742 ± 1410	73.9 ± 38.4	38.1 ± 2.1

D= 7, K = 13

N° samples	Avg. Total time (s)	Avg. N° of jobs	Avg. of Avg. Job time (s)
1 000	211.5 ± 46.6	9.3 ± 1.9	22.7 ± 0.7
10 000	731.4 ± 321.6	28.7 ± 12.3	25.4 ± 0.6
100 000	4523.3 ± 824	87.9 ± 16.2	51.5 ± 1

OBSERVATIONS

By observing the aggregated data, we can see how the convergence of the algorithm gets affected. As we increase the number of points, it becomes more and more probable that some points will switch cluster multiple times, even if the movement factor of the centroids is tiny. This increases dramatically the number of jobs required to reach the convergence, especially in the *100 000* samples cases.

The average time required to execute a single *job* (aka iteration) remains basically the same in all the *1 000* and *10 000* samples results, meanwhile the *100 000* samples cases present always a considerable increase. This indicates that **N** is by far the most important parameter for the computational complexity, while **K** and **D** play a more marginal role. From this result is also possible to understand the effects of the *overhead* produced by the *Hadoop* jobs setup, since the execution times remain the same both in the case with *1 000* and *10 000* points.

SPARK

RESULTS

The following tables summarize the results:

D= 3, K = 7

N° samples	Avg. Total time (s)	Avg. N° of iterations	Avg. of Avg. Iteration time (s)
1 000	57,8 ± 15,3	9.2 ± 4.3	6.8 ± 1.5
10 000	54,9 ± 14,8	29.9 ± 16.9	2.2 ± 0.9
100 000	243,7 ± 61,2	40 ± 12.2	6.3 ± 1.1

D= 3, K = 13

N° samples	Avg. Total time (s)	Avg. N° of iterations	Avg. of Avg. Iteration time (s)
1 000	55.2 ± 17.2	13 ± 4.1	4.6 ± 1.9
10 000	85.5 ± 20.6	27.2 ± 14.4	3.9 ± 1.8
100 000	619.1 ± 231.1	79.3 ± 24	7.7 ± 1.3

D= 7, K = 7

N° samples	Avg. Total time (s)	Avg. N° of iterations	Avg. of Avg. Iteration time (s)
1 000	44.1 ± 6.1	10.7 ± 5	5.1 ± 2.4
10 000	61.7 ± 18.9	22.3 ± 17.7	6.5 ± 5.1
100 000	504.3 ± 167.2	83.5 ± 28.1	7.5 ± 4.6

D= 7, K = 13

N° samples	Avg. Total time (s)	Avg. N° of iterations	Avg. of Avg. Iteration time (s)
1 000	43.9 ± 7.9	9.6 ± 3.2	4.9 ± 1.3
10 000	73.6 ± 14.6	29.5 ± 10.3	3.2 ± 2.1
100 000	658 ± 216.1	86.3 ± 17.9	7.5 ± 1.6

OBSERVATIONS

Most of the observations made in the previous paragraph are still valid. The aggregated results show the impressive difference in performance between *Hadoop* and S*park*. As described earlier: **N** is still the most important parameter for the computational complexity, however the step between *10 000* samples and *100 000* samples didn't cause a significant raise in the average iteration time, while in *Hadoop* caused a double avg. *job* time. Another interesting fact is that the avg. iteration time is lower in the *10 000* samples executions with respect to the *1 000* samples executions. This is due to the fact that the first iteration is definitely the slowest, ranging around *35-40* seconds in each execution. The *1 000* samples cases tend to converge with only a few iterations, so the avg. iteration time stays higher than the *10 000* samples cases.