

Master degree in Computer Engineering

Cloud Computing



UNIVERSITÀ DI PISA

K-means implementation using Hadoop and Spark frameworks

September 2020

Federico Cappellini, Andrea Lelli, Alberto Lunghi, Lorenzo Susini

0. Introduction	3
0.1 Description of the problem and the algorithm	3
0.2 Fitting the algorithm into MapReduce	3
1. File input generator	4
2. Hadoop Implementation	7
2.1 KMeansMapper	7
2.2 KMeansReducer	8
2.3 KMeansCombiner, CentroidWritable and PointWritable	8
2.4 Main	9
3. Spark Implementation	11
3.1 main	12
3.2 kmeans_map	13
3.3 kmeans_reduce	13
3.4 divide_coords	14
4. Correctness of the algorithm	15
5. Test results	16
5.1 Hadoop VS Spark	16
5.1.1 Scaling on multi-core architectures	17
5.2 Hadoop: combiners VS no combiners	18

0. Introduction

0.1 Description of the problem and the algorithm

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 Lloyd's algorithm, commonly referred to simply as the K-means clustering algorithm. The aim of this project is to implement this algorithm using MapReduce, both on the Hadoop and Spark frameworks. A definition of the problem and the algorithm to be implemented are described as follows:

Let $X = \{x_1, \dots, x_n\}$ be a set of n data points, each with dimension d . The K-means problem seeks to find a set of k points (called means) $M = \{\mu_1, \dots, \mu_k\}$ which minimizes the function

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

In other words, we wish to choose k means so as to minimize the sum of the squared distances between each point in the data set and the mean closest to that point. Finding an exact solution to this problem is NP-hard. However, there are a number of heuristic algorithms which yield good approximate solutions. The standard algorithm for solving the K-means problem uses an iterative process which guarantees a decrease in total error (value of the objective function $f(M)$) on each step. The algorithm can be described in this way:

1. Choose the k initial means uniformly at random from the set X .
2. For each point x belonging to the set X , find the closest mean μ_c and add x to the set ω_c , initially empty.
3. Update the values of $\{\mu_1, \dots, \mu_k\}$ by computing the centroid of the data point belonging to each set ω_i .

These steps are repeated until a convergence criterion has been met. The convergence criterion is typically met when the total error stops changing between steps, in which case a local optimum of the objective function has been reached.

0.2 Fitting the algorithm into *MapReduce*

Of course the problem has to be fitted using the *MapReduce paradigm*, by describing what the *mappers* and the *reducers* have to do. In particular, the general idea can be described as follows:

- The **map function** is executed by the mappers. It receives as input a point. Then, it computes the *euclidean distance* between the point and all the other actual means. Then, it chooses the closest mean and it emits the key-value pair (**closest mean index, (point,weight)**).
- The **reduce function** is executed by the reducers. It receives as input a centroid and the list of its closest points. Then, it computes the new centroid by taking the mean on each dimension of the points. It emits the pair (**index, new mean**).

For the sake of clarity, in the figures below are shown the map and reduce functions in pseudo-code:

```
k-meansMap(x):
    emit (argmini ||x - μi||22, (x, 1))

k-meansReduce(i, [(x, s), (y, t)]):
    return (i, (x + y, s + t))
```

This *MapReduce* job has to be repeated until the convergence criterion has not been met, so **the number of jobs executed is equal to the number of iterations of the algorithm**. At each iteration, we have to take care of using the new means as centroids, i.e the means that were computed by the previous job.

1. File input generator

We generate our dataset using a dataset generation algorithm written by us.

It takes as input:

- The number of coordinates for each centroid and point (*dim*)
- The number of centroids (*k*)
- The number of points (*n*)
- The minimum and the maximum value for the coordinates for the centroids used as generators (*min*, *max*)

```

87 try:
88     dim = int(sys.argv[1])
89     k = int(sys.argv[2])
90     n = int(sys.argv[3])
91     min_value = float(sys.argv[4])
92     max_value = float(sys.argv[5])
93 except:
94     print('Error usage: <dimension> <centroids number> <number of points> <min coordinate value for generator '
95           '<centroids> <max coordinate value for generator centroids> ')
96
97     centroid_dimension_list = []
98     for j in range(dim):
99         centroid_dim = []
100         centroid_dimension_list.append(centroid_dim)
101
102     for i in range(k):
103         for current_dim in range(dim):
104             centroid_dimension_list[current_dim].append(np.random.uniform(min_value, max_value))
105
106     points_list = []
107     for j in range(dim):
108         points_dim_list = []
109         points_list.append(points_dim_list)
110
111     for i in range(k):
112         for j in range(int(n / k)):
113             for current_dim in range(dim):
114                 dim_centroid = (centroid_dimension_list[current_dim])[i]
115                 (points_list[current_dim]).append(np.random.normal(dim_centroid, 1.0))

```

The first piece of the algorithm: the generation of the generator centroids and the dataset points.

In the first piece of code this algorithm produces the centroids with coordinate values chosen between *min* and *max* using a uniform random distribution function. Then for each centroid this algorithm creates a pool of points with coordinate values chosen by a normal random function with a standard deviation of *1.0* and the current centroid as center.

```

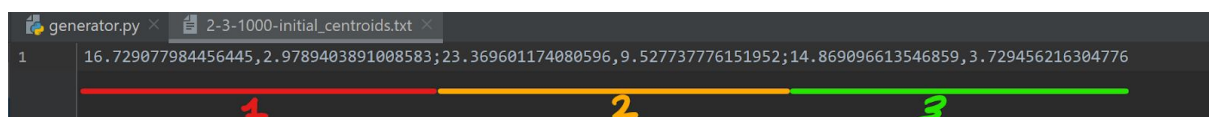
117 filename_dataset = str(dim) + '-' + str(k) + '-' + str(n) + '-dataset'
118 print_dataset_file(filename_dataset, dim, points_list)
119 filename_initial_centroids = str(dim) + '-' + str(k) + '-' + str(n) + '-initial_centroids'
120 print_initial_centroid_file(filename_initial_centroids, dim, k, points_list)

```

The second piece of the algorithm: the creation of the initial_centroids file and the points_file.

In this second piece of code the algorithm creates two files:

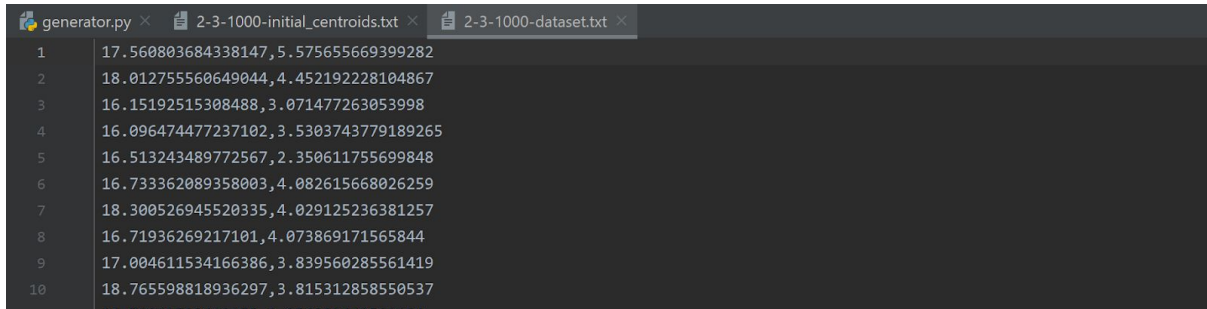
- **the initial centroids file:**
 - The file contains the initial centroids dataset for the K-Means algorithm.
 - The file is composed by a single line with all the centroids separated by a ','
 - The initial centroids are chosen randomly from the dataset points



The initial centroids file with 3 initial centroids

- **the points file:**

- The file contains the input points dataset for the K-Means algorithm.
- Each line of the file is a point described by the list of its coordinate values separated by a ','



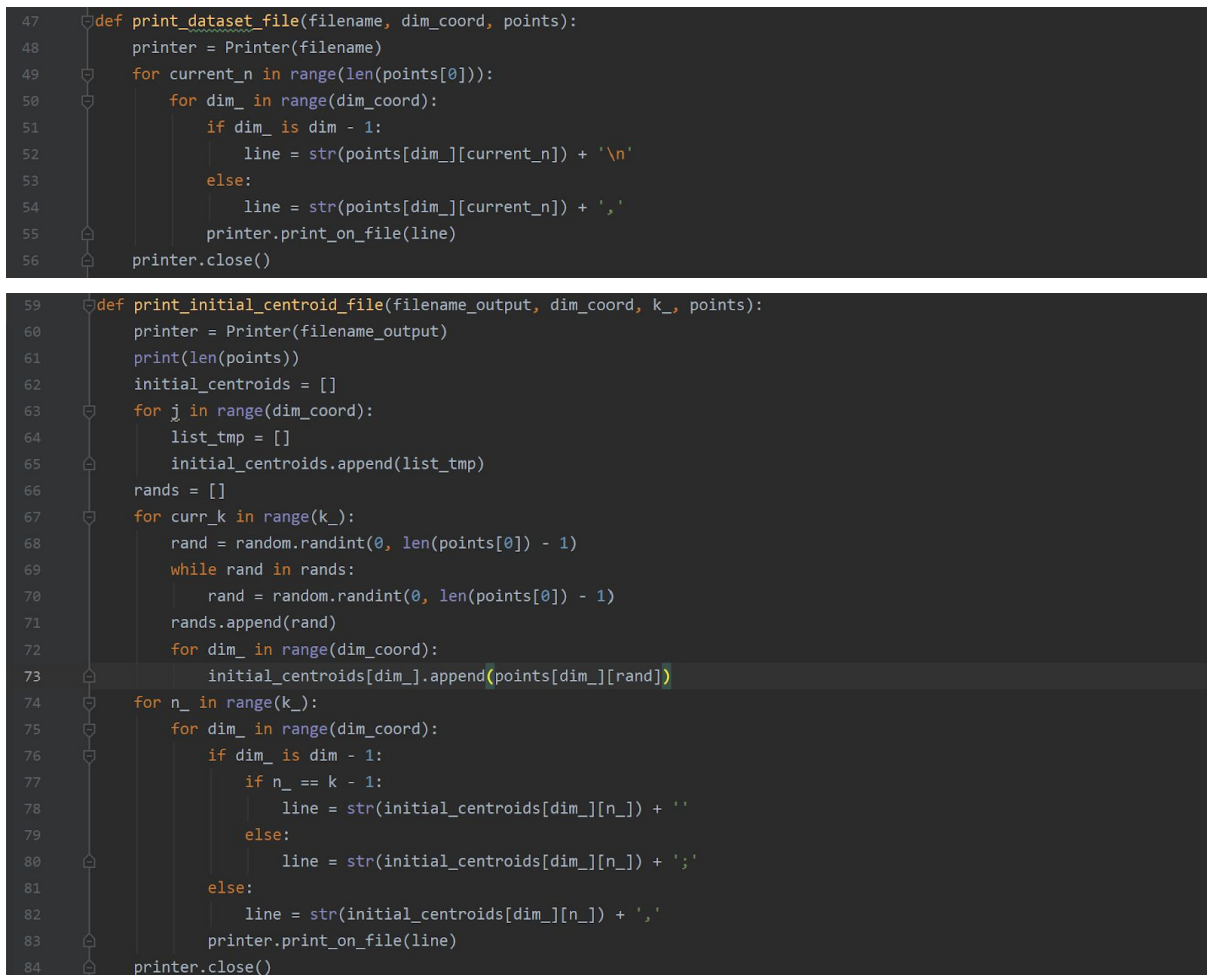
```

1 17.560803684338147,5.575655669399282
2 18.012755560649044,4.452192228104867
3 16.15192515308488,3.071477263053998
4 16.096474477237102,3.5303743779189265
5 16.513243489772567,2.350611755699848
6 16.733362089358003,4.082615668026259
7 18.300526945520335,4.029125236381257
8 16.71936269217101,4.073869171565844
9 17.004611534166386,3.839560285561419
10 18.765598818936297,3.815312858550537

```

An overview of the points file

The two files are written with the two functions below:



```

47 def print_dataset_file(filename, dim_coord, points):
48     printer = Printer(filename)
49     for current_n in range(len(points[0])):
50         for dim_ in range(dim_coord):
51             if dim_ is dim - 1:
52                 line = str(points[dim_][current_n]) + '\n'
53             else:
54                 line = str(points[dim_][current_n]) + ','
55             printer.print_on_file(line)
56     printer.close()

59 def print_initial_centroid_file(filename_output, dim_coord, k_, points):
60     printer = Printer(filename_output)
61     print(len(points))
62     initial_centroids = []
63     for j in range(dim_coord):
64         list_tmp = []
65         initial_centroids.append(list_tmp)
66     rands = []
67     for curr_k in range(k_):
68         rand = random.randint(0, len(points[0]) - 1)
69         while rand in rands:
70             rand = random.randint(0, len(points[0]) - 1)
71         rands.append(rand)
72         for dim_ in range(dim_coord):
73             initial_centroids[dim_].append(points[dim_][rand])
74     for n_ in range(k_):
75         for dim_ in range(dim_coord):
76             if dim_ is dim - 1:
77                 if n_ == k - 1:
78                     line = str(initial_centroids[dim_][n_]) + ''
79                 else:
78                     line = str(initial_centroids[dim_][n_]) + ';'
81             else:
82                 line = str(initial_centroids[dim_][n_]) + ','
83             printer.print_on_file(line)
84     printer.close()

```

The two functions used in order to write the two datasets file

2. Hadoop Implementation

In this chapter is explained the Hadoop implementation of the *K-means* algorithm. The code that was developed is mainly summarizable by describing the main classes that were used, such as:

- KMeansMapper
- KMeansCombiner
- KMeansReducer
- PointWritable
- CentroidWritable
- Main

2.1 KMeansMapper

The KMeansMapper class is the class implementing the mapper. First of all:

- The input is a (*LongWritable*, *Text*) key-value pair. In particular, the key of the input is unused. Instead, the *Text* value is an input split containing the points of the dataset generated by the dataset generator, written as strings in multiple lines.
- The output is a key-value pair made up of two *CentroidWritable*. The reason behind this choice will be explained in a bit.

The relevant function of this class is of course the map function, shown in the image below.

```
@Override
protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
    String record = value.toString();
    if(record == null || record.length() == 0)
        return;

    String[] lines = record.split("\n");

    for(String line : lines) {
        point.updateFromLine(line);
        int minIndex = 0;
        float minDistance = Float.MAX_VALUE;
        for(int i = 0; i < distances.length; ++i) {
            distances[i] = centroids[i].distanceTo(point);
            if(distances[i] < minDistance) {
                minDistance = distances[i];
                minIndex = i;
            }
        }
        pointAsCentroid.linkTo(point);
        context.write(centroids[minIndex], pointAsCentroid);
    }
}
```

The function casts each line of the input split into a *PointWritable* object. Then, it computes the distance between that point to all the actual centroids, in order to find the closest one. Then, after casting the point object into a *CentroidWritable* instance, it emits the pair

(*centroid[minIndex]*, *pointAsCentroid*) by writing into the context. The *PointWritable* object has to be casted in the corresponding *CentroidWritable* instance with a *weight* equal to one. This allows us to make use of the *KMeansCombiner* class, that acts like a local reducer, in order to reduce the amount of data needed to be exchanged during the *Shuffle And Sort phase*, taking place between the map and the reduce phases, as better explained in the *KMeansCombiner* paragraph.

2.2 KMeansReducer

The *KMeansReducer* class is the class implementing the reducer. It takes

- as input a *CentroidWritable* and a list of *CentroidWritables*
- as output a *CentroidWritable* and a null *Object*

The key of the input is the centroid that has to be updated using the corresponding list of *CentroidWritables*. Those are all the points that are closest to that actual centroid in consideration. All the reduce function has to do is to simply compute a weighted average on each dimension of the points. In this way, the new centroid for the next iteration is computed and emitted by writing it as key in the context, followed by a null value, as it can be seen in the image below.

```
@Override
protected void reduce(CentroidWritable originalCentroid, Iterable<CentroidWritable> localAvgCentroids, Context context) throws IOException, InterruptedException {
    Iterator<CentroidWritable> iterator = localAvgCentroids.iterator();
    globalAvgCentroid.resetToZero();
    int weightSum = 0;
    float[] coords = globalAvgCentroid.getCoords();
    while(iterator.hasNext()) {
        CentroidWritable localAvgCentroid = iterator.next();
        float[] lvcCoords = localAvgCentroid.getCoords();
        for(int i = 0; i < coords.length; ++i) {
            coords[i] += lvcCoords[i] * (float)localAvgCentroid.getWeight();
        }
        weightSum += localAvgCentroid.getWeight();
    }

    if(weightSum == 0)
        return;

    globalAvgCentroid.divideCoordsBy((float)weightSum);
    context.write(globalAvgCentroid, null);
}
```

2.3 KMeansCombiner, CentroidWritable and PointWritable

As already introduced in the previous sections, the *KMeansCombiner* is the class implementing the combiner. It is a reducer working locally and directly on the output of one of the mappers. The other two classes are used in order to represent a point and to make it being used by the Hadoop framework. In particular, these representations of points are used as keys and values, and for this reason they have to be serializable, because they need to be exchanged through the network. They also have to be comparable, because they have to be sorted by the framework before delivering the keys to reducers. For this reason, they implement the *WritableComparable* interface provided by the Hadoop framework. This is a common practice and it is confirmed also by the documentation of the interface: “*WritableComparables can be compared to each other, typically via Comparators. Any type which is to be used as a key in the Hadoop Map-Reduce framework should implement this interface*” and “*A (Writable is a) serializable object which implements a simple, efficient,*

serialization protocol, based on *DataInput* and *DataOutput*. Any key or value type in the Hadoop Map-Reduce framework implements this interface”.

First, we have implemented the *PointWritable* class. It has a member called *coords* that is an array of floats and a member called *dim*, used to describe the dimension of the point. The initial idea was to use this class as key and value of the output and the input of mappers and reducers, respectively, but then, after facing the problem of implementing a combiner, we soon realized that the *PointWritable* class had to be extended with the concept of weight. In fact, *CentroidWritable* extends the *PointWritable* class and adds a *weight* as a member field. In this way, the combiner can compute the intermediate new mean values by considering only the local points. Depending on the number of points that were evaluated, a different weight has to be used in the final reducer. This consideration leads to the correct implementation of the overall algorithm.

2.4 Main

Lastly, the Main class is responsible for configuring the jobs. The K-Means algorithm is intrinsically iterative. Instead, Hadoop was developed to analyse and process lots of data but in a one-pass fashion. As a consequence of this, the Main class has to launch many jobs one after the other and to pass the output of one job into the input of the next one by writing the new centroids in the configuration, as shown in the image below.

```
while(true) {
    CentroidWritable[] oldCentroids = Utils.readCentroidsFromConf(conf);

    Job job = Job.getInstance(conf, "Main");
    job.setJarByClass(Main.class);
    // set mapper/combiner/reducer
    job.setMapperClass(KMeansMapper.class);
    job.setCombinerClass(KMeansCombiner.class);
    job.setReducerClass(KMeansReducer.class);

    // define mapper's output key-value
    job.setMapOutputKeyClass(CentroidWritable.class);
    job.setMapOutputValueClass(CentroidWritable.class);

    // define reducer's output key-value
    job.setOutputKeyClass(CentroidWritable.class);
    job.setOutputValueClass(Object.class);

    // define I/O
    FileInputFormat.addInputPath(job, inPath);
    FileOutputFormat.setOutputPath(job, outPath);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    if(!job.waitForCompletion(true)) {
        System.err.println("Job failed!!!");
        System.exit(1);
    }

    //Set the new centroids
    newCentroids = Utils.readCentroidsFromHdfs(conf, args[4], dim);
    Utils.writeCentroidsToConf(conf, newCentroids);

    if(converging(oldCentroids, newCentroids)) {
        break;
    }

    Utils.deleteFromHdfs(conf, outPath);
}
```

Moreover, it also takes care of checking if the convergence criterion has been met. In particular, in our implementation we have used a configurable threshold. If the error between two consecutive iterations is below that threshold, the algorithm ends and no other jobs are launched. The check of convergence is implemented in the *converging* function, as shown

below. In practice, the distances between the centroids of the previous iteration and the new ones are computed (one by one), and a simple check is performed using the threshold.

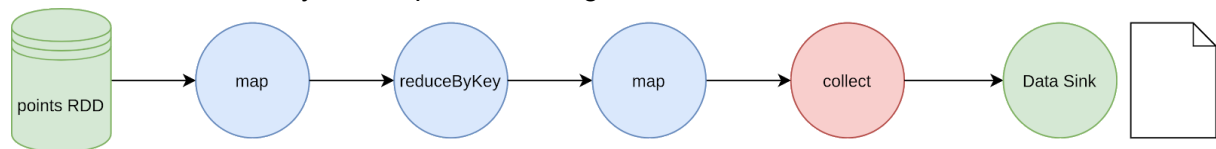
```
private static boolean converging(CentroidWritable[] a, CentroidWritable[] b) {
    if(a.length != b.length) {
        throw new RuntimeException(
            "converging(...) got a different number of centroid dims\n" +
            "a.length = " + a.length + " | b.length = " + b.length
        );
    }
    for(int i = 0; i < a.length; ++i) {
        if(a[i].distanceTo(b[i]) > THRESHOLD) {
            return false;
        }
    }
    return true;
}
```

3. Spark Implementation

For the Spark implementation of the K-means algorithm using the MapReduce paradigm, we proceeded following these steps:

1. usage of an input file containing all the points of the dataset
2. initial centroids given from command line when launching the Spark application, and write them into a *broadcast variable*
3. iterate until not converging:
 - a. read old centroids from the broadcast variable declared by the driver process
 - b. perform three transformations on the points' RDD: *map()*, *reduceByKey()*, *map()*
 - c. trigger the transformations using the action *collect()*
 - d. retrieve new centroids (and sort them)
 - e. compute the distance from the old centroids to the new ones
 - f. if not converging, update the broadcast variable containing the centroids
4. show final centroids of the k clusters in the console

This is the Directed Acyclic Graph of one single iteration:



Now we are going to analyze in detail the code of the Spark application.

3.1 main

This is the code of the main function:

```
56 def main():
57     global centroids_broadcast
58     #default values
59     centroids_string = "13.26573242704478,18.07303339406258;11.965183977426816,17.908125592420962;13.322344279783637,16.
60     points_file = "2-4-10000-hadoop.txt"
61     argc = len(sys.argv)
62     if argc > 1:
63         centroids_string = sys.argv[1]
64     if argc > 2:
65         points_file = sys.argv[2]
66     master = "local[*]"
67     sc = SparkContext(master, "K-means")
68     centroids = from_str_to_cent(centroids_string)
69     start_time = time.time()
70     centroids.sort()
71     centroids_broadcast = sc.broadcast(centroids)
72     points_rdd = sc.textFile(points_file).cache()
73     iteration_count = 0
74     while True:
75         iteration_count += 1
76         print("Iteration count: " + str(iteration_count))
77         old_centroids = centroids_broadcast.value
78         print("=== Centroidi in ingresso all'iterazione {} ===".format(iteration_count))
79         for old_centroid in old_centroids:
80             print(" Centroide: " + str(old_centroid))
81         points_rdd_map = points_rdd.map(lambda x: kmeans_map(x))
82         points_rdd_reduce = points_rdd_map.reduceByKey(lambda x, y: kmeans_reduce(x, y))
83         new_centroids_rdd = points_rdd_reduce.map(lambda x: divide_coords(x))
84         new_centroids = new_centroids_rdd.collect()
85         for i in range(len(new_centroids)):
86             new_centroids[i] = new_centroids[i][0]
87         new_centroids.sort()
88         print("=== Centroidi in uscita all'iterazione {} ===".format(iteration_count))
89         for new_centroid in new_centroids:
90             print(" Centroide: " + str(new_centroid))
91         if converge(old_centroids, new_centroids):
92             break
93         centroids_broadcast = sc.broadcast(new_centroids)
94     end_time = time.time()
95     print("Completed after {} seconds".format(end_time-start_time))
```

In the main function we initialize a string value that contains all the initial centroids generated by the python script shown in chapter 1. The centroids are converted into a list of centroids, written into the broadcast variable *centroids_broadcast*.

Then we initialize the Spark context for the application using the constructor *SparkContext* of the python library *pyspark*. Of course, the *master* option is “yarn”, so as to connect the context to a YARN cluster. For testing purposes we also run the application into a personal VM at our disposal, and in that case the *master* option used was “local[*]”.

Furthermore the input text file containing the points of the dataset is read, and Spark automatically creates the respective RDD. At this point, the *cache()* function is invoked, in this way, the workers will store the partitions of the RDD in their local RAM in order to access the data faster.

After these initialization steps the loop begins. In this loop the list of the old centroids is retrieved from the broadcast variable previously described. After this, the transformations already shown are performed. The functions used as input for the transformation operations are, respectively, *kmeans_map()*, *kmeans_reduce()*, *divide_coords()*, that we will describe in a bit.

At the end the *collect()* action is invoked, and the results of the current iteration are returned to the driver process. In this way the driver can update the broadcast variable of the centroids for the next iterations.

When the driver sees the new results, it runs the *converge()* function, that simply computes the euclidean distance between each old and new centroid. The loop is broken if the distance between all the old and new centroids is below or equal 0.1. Before checking the convergence, a sorting procedure of the list of the new centroids is made. This is done in order to always compare the correct tuple of old centroid and new centroid.

When the loop breaks, the final results are shown in the console.

3.2 kmeans_map

This is the function called in the first *map* transformation:

```
35 def kmeans_map(point):
36     point = from_str_to_point(point)
37     centroids = centroids_broadcast.value
38     min_distance = 10000000
39     min_index = -1
40     for i in range(len(centroids)):
41         distance = euclidean_distance(centroids[i], point)
42         if min_distance > distance:
43             min_distance = distance
44             min_index = i
45     return min_index, (point, 1)
```

In this function the worker takes one point at a time and computes the distance from the point to all the current centroids. At the end it retrieves a key-value pair of this format: *closest_centroid_index*, (*[point_x, point_y, point_z,...]*, 1). The last number is the weight associated with that point, that will be used for the computation of the new centroid. Of course each point has a weight of 1.

3.3 kmeans_reduce

This is the function called in the *reduceByKey* transformation:

```
17 def kmeans_reduce(point_a, point_b):
18     total_weight = point_a[1] + point_b[1]
19     point_sum = []
20     dim = len(point_a[0])
21     for i in range(dim):
22         point_sum.append(float(point_a[0][i]) + float(point_b[0][i]))
23
24     return point_sum, total_weight
```

In this function the worker takes two points at a time, expressed as key-value pairs, as shown in the *kmeans_map*.

The function sums up the coordinates and the weights of the points, and outputs them as a key-value pair of this format: *[point_sum_x, point_sum_y, point_sum_z,...]*, *weight_sum*.

The *reduceByKey* transformation applies this function to the values with the same key, and so, to the same centroids.

3.4 divide_coords

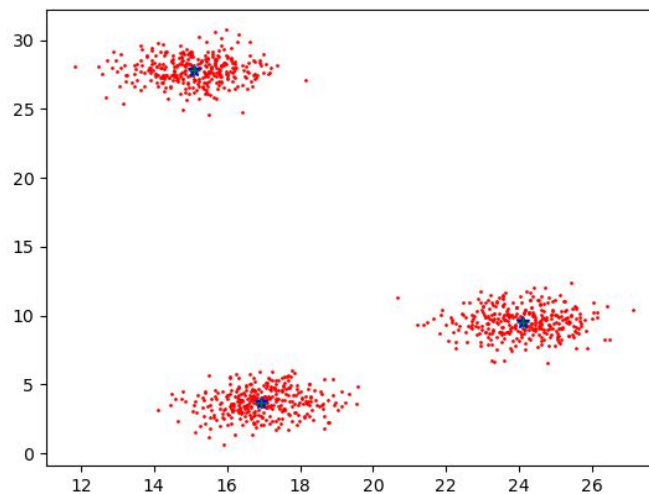
This is the function called in the second *map* transformation:

```
9 def divide_coords(point):
10     centroid = point[1]
11     weight = centroid[1]
12     coords = centroid[0]
13     coords = [coord / float(weight) for coord in coords]
14     return coords, None
```

Clearly, this function is simply taking as input a key-listOfValues pair. The list is made of all the points that have as closest centroid the centroid corresponding to the considered key. The function computes the mean value of the coordinates in the list, based on their weights (weighted average). The result will be used to compute the new coordinates of that centroid.

4. Correctness of the algorithm

Several tests were performed in order to check the correctness of the algorithm, i.e that the output centroids were actually a suboptimal solution to the problem. In particular, we have used the K-Means algorithm offered by the *sklearn* library as a reference. A plot was produced in order to visualize the results of the sklearn K-Means, the results of our implementation of the algorithm and the dataset in consideration. This is a heuristic way to check if the algorithm was working properly. For instance, considering the first dataset (2-3-1000-dataset.txt) this plot was obtained:



Our centroids are plotted using the green crosses, while the outputs of sklearn are plotted using the blue asterisks. The red points represent the dataset. As you can see, the results are pretty overlapped, meaning that our algorithm works properly. Of course the results cannot perfectly match. This is because we have set a threshold in the error between iterations that isn't a perfect match with the one used by sklearn. Moreover, the two algorithms may start from different initial conditions, but the overall result is heuristically identical (reference for [sklearn](#))

5. Test results

5.1 Hadoop VS Spark

Apache Hadoop is a collection of open-source softwares that provide a software framework for distributed storage and processing of big data using the MapReduce programming model on commodity hardware.

Apache Spark is an open-source distributed general-purpose cluster-computing framework that we already know to be better (in most cases) than Hadoop for several reasons.

The most important ones are:

- Support for more computing paradigms (other than Map-Reduce)
- Much better iterative performance thanks to
 - the capability to in-memory cache data as much as requested by the programmer (with hardware limitations)
 - the fact that Spark does not need to re-launch a new job every iteration with a new environment

Due to the fact that the k-means algorithm is an iterative algorithm, our expectation was that Spark could lead to higher performances. Moreover in each preliminary test, Spark showed less overhead delays during the first launch of the application and between an iteration and the next one.

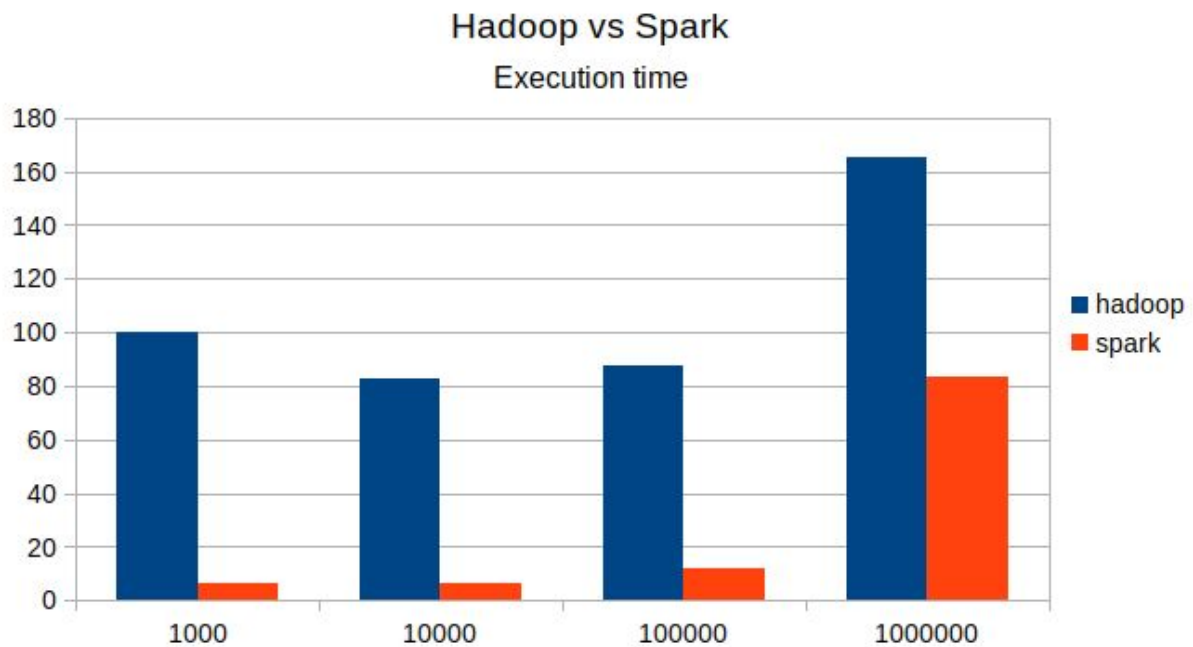
A benchmark was built composed by varying some input conditions (a total of 16 experiments were planned):

- the number of points is 1000, 10000, 100000 or 1000000 (with a larger dataset, computational times are much more relevant with respect to overhead delays)
- in some cases the points are bi-dimensional and in some others are three-dimensional
- the number of centroids is 3, 7 or 13

A Bash script was created to simplify and to automate the benchmark.

```
root@osboxes:~# ./avvia.sh
1) 1000 points, dim=2, centroids=3
2) 10000 points, dim=2, centroids=3
3) 100000 points, dim=2, centroids=3
4) 1000000 points, dim=2, centroids=3
5) 1000 points, dim=3, centroids=3
6) 10000 points, dim=3, centroids=3
7) 100000 points, dim=3, centroids=3
8) 1000000 points, dim=3, centroids=3
9) 1000 points, dim=3, centroids=7
10) 10000 points, dim=3, centroids=7
11) 100000 points, dim=3, centroids=7
12) 1000000 points, dim=3, centroids=7
13) 1000 points, dim=3, centroids=13
14) 10000 points, dim=3, centroids=13
15) 100000 points, dim=3, centroids=13
16) 1000000 points, dim=3, centroids=13
17) Quit
Choose benchmark: |
```


The aggregated results are shown here (results are grouped by the dataset size):



It is clear that Spark is faster than Hadoop in every condition. In particular, the difference is greater in tests with fewer points due to higher overhead times by Hadoop and in tests where the number of iterations required is greater.

Indeed, for instance, the test number 1 requires 4 iterations and the test number 2 requires 3 iterations and, although the test number 2 has less points, it requires more time both for Spark and Hadoop, but Spark slows down only a little (3.7% or 0.232 seconds) while Hadoop slows down a lot (55.95% or 47.72 seconds). This is of course due to the fact that Hadoop is forced on saving results to disk between consecutive iterations, leading to a performance slow down.

5.1.1 Scaling on multi-core architectures

Since all the tests were executed on a pseudo-distributed installation, it was interesting to study how the two frameworks behave in relation to vertical scaling.

The virtual machine was configured to perform the test number 16 (1 million three-dimensional points with 13 centroids) two times: at first with only one core allocated and then with four cores. All the other characteristics remained constant.

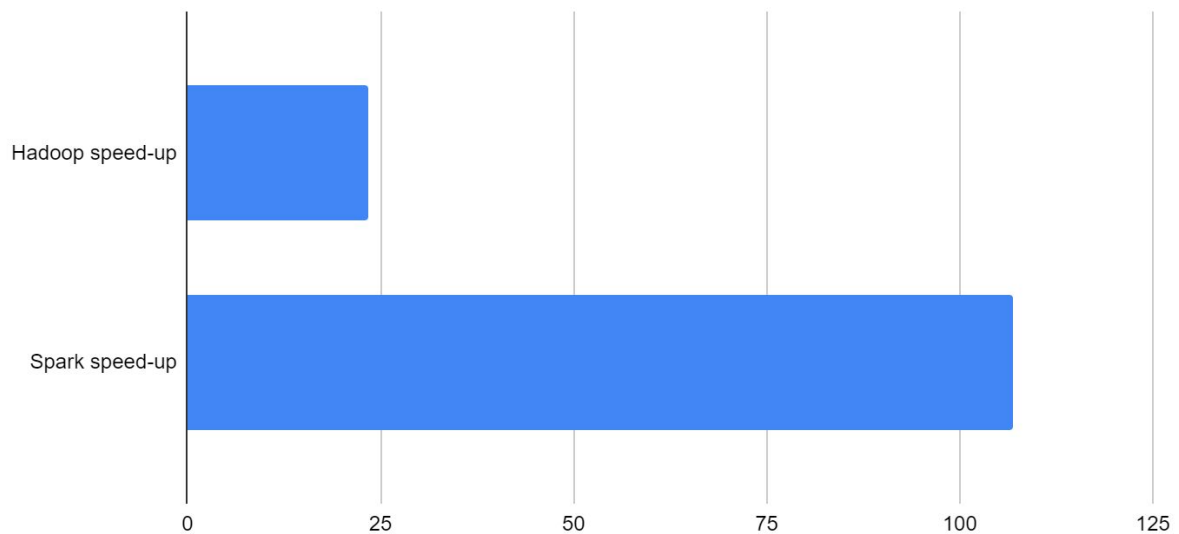
The main specifications of the physical server are:

- Intel Core i7 2600K
 - frequency: 4 GHz
 - cache: 8MB
 - core count: 4 with hyper-threading (1/4 allocated to the VM)
- 8GB RAM DDR3 1600MHz (2GB dedicated to the VM)

It was interesting to notice that, while Hadoop has no significant advantages in relation to vertical scaling, Spark benefits greatly from it. More precisely, Hadoop demonstrated a

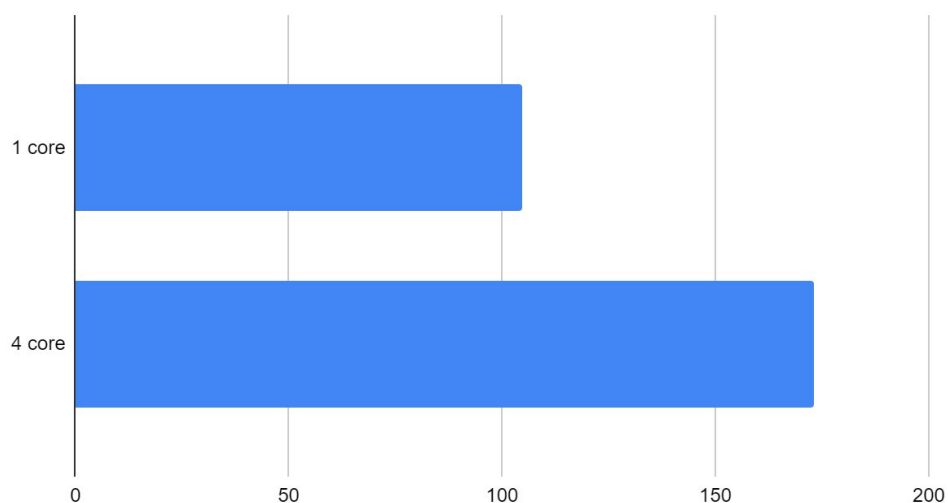
relative speed-up of 25.3% (from 2108.5 points per second to 2642.7) while Spark of 106.9% (from 2209.9 to 4573.4).

Relative speed-up comparison in relation to vertical scaling (1 to 4 cores)



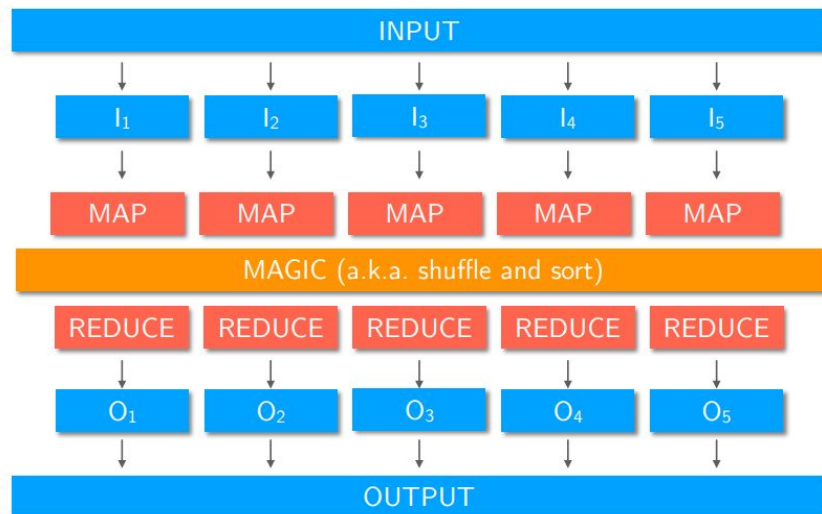
This test therefore demonstrates another of the advantages of using the Spark framework in comparison with Hadoop: with a single machine execution, the two frameworks are still comparable in the presence of a not excessive number of iterations and a very low number of cores, but as the computational capabilities increase, Spark represents in any case the most correct choice for the implementation of a distributed algorithm.

Spark speed relative to Hadoop



5.2 Hadoop: combiners VS no combiners

The Map-Reduce distributed computing paradigm can be summarized by the following diagram:

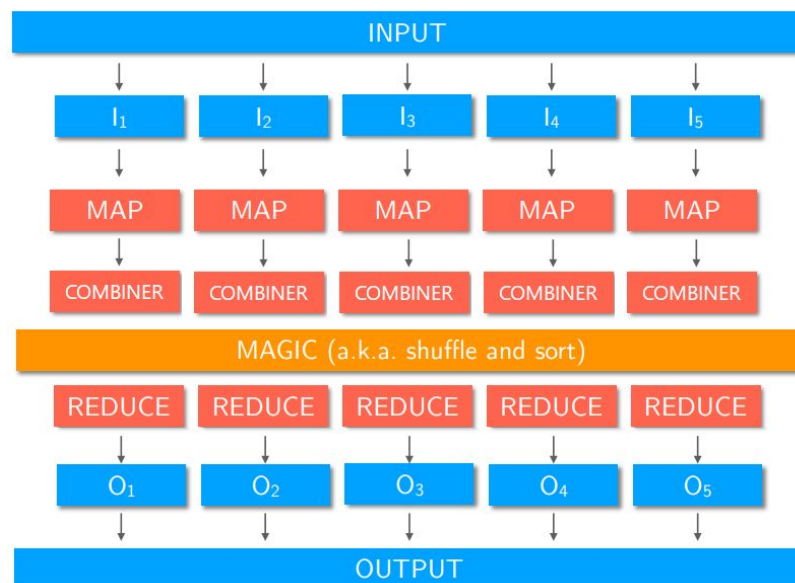


Each input split is computed separately by a MAP instance and the intermediate data produced in this stage must be “shuffled and sorted” to be processed by the right REDUCE instance.

A common “problem” of this approach is that the intermediate data could be very large and their movement across the network (to reach the right REDUCE instance) could lead to very large delay. Moreover, in iterative application, this problematic step can occur many times.

In this implementation of the k-means algorithm, the output of the MAP instances is a series of key value pairs, one for each input point, so the intermediate data are $O(N)$ with respect to the number of input points.

To reduce the network traffic, in our application combiners are used:



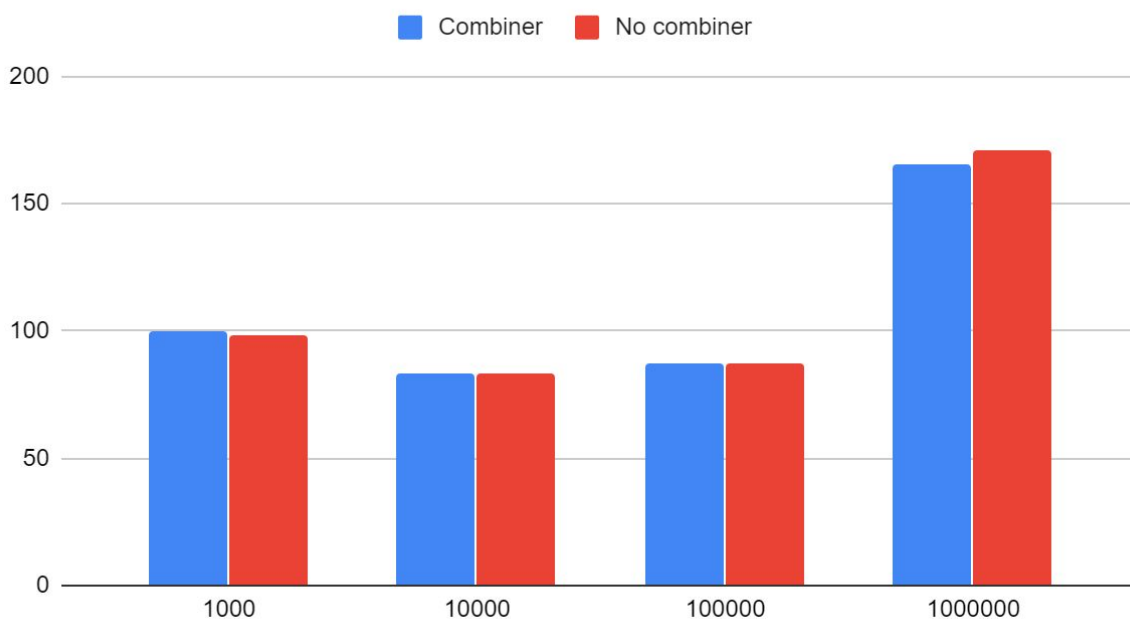
Combiners are very similar to a local copy of the reducer (local means that there is a combiner for each map instance that runs on the same node) and perform a local aggregation before the shuffle phase.

In this k-means implementation, indeed, each combiner performs a local average for each centroid. Since each combiner emits a series of k key-value pairs (the key is the original centroid and the value is the local average), given m the number of map instances, the intermediate data are $O(k * m)$ that is much less than $O(N)$.

This optimization leads to much better performance when the algorithm is performed by a compute intensive cluster.

Anyway, our tests were executed on a pseudo-distributed installation of Hadoop and the benefits of the combiners were eliminated due to the fact that a pseudo-distributed installation involves only a single machine. This is confirmed by the results of the tests which report almost identical execution times in the two cases:

Combiner approach VS No combiner approach



With the smallest dataset (1000 points), the solution with no combiners is about 1.5% faster, but with the largest dataset (1 million points), the solution with combiners is about 3.5% faster.

We noticed also that the “no combiner” approach could lead to small overhead delays, this is due to the fact that the application master will instantiate not 1, but 4 mappers and 4 combiners.