UNIVERSITÀ DI PISA

K-Means Clustering Algorithm on Map and Reduce

Berti Nicola

Imbelli Cai Marco

Picchi Nicolò

Università di Pisa, DII

Master degree in AIDE, Computer Engineering

Course of Cloud Computing

Academic Year 2022-2023

Table of contents

1. Introduction: the MapReduce paradigm

The **MapReduce** paradigm is used worldwide to design and implement computer systems capable of handling large amounts of data. The framework is based on parallelizing the operations that need to be performed using four fundamental components:

Mapper: in this first phase, conditional logic filters the data through all the nodes in key-value pairs. The "key" refers to the offset address for each record, while the "value" contains the entire content of the record.

Shuffle: during the second phase, the output values from mapping are sorted and consolidated. Values are grouped based on similar keys, and duplicates are discarded. The output of the Shuffle phase is also organized into key-value pairs, but this time the values indicate a range rather than the content of a record.

Reducer: in the third phase, the consolidated output of the Shuffle phase is aggregated, with all the values added to the corresponding keys. All of this is then combined into a single output directory.

Combiner: executing this phase can optimize the performance of MapReduce processes by accelerating their flow. To perform this operation, MapReduce retrieves the outputs from the Mapper phase and examines them at the node level to identify duplicates, which are combined into a single key-value pair, thereby reducing the work that the Shuffle phase needs to complete.

2.1 The K-Means algorithm on MapReduce

The K-means algorithm is a computationally expensive algorithm for clustering data.

It involves grouping data points based on common characteristics. Given a set of n points in d dimensions and a number k of clusters, each iteration of the algorithm has a cost of $O(n \cdot d \cdot k)$.

The objective of the process is finding a set of k points (the so called "clusters") $M = \{\mu_1, \dots, \mu_k\}$ that minimizes the following objective function:

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

where $X = \{x_1, \dots, x_n\}$ is the set of n data points.

The algorithm consists of two main phases:

Choosing the initial centroids for each cluster

Calculating the new centroids for the clusters

The choice of the initial centroids can be made using specific algorithms such as K-Means++, Hierarchical K-Means, Density-Based Initialization, or Spectral Splitting Initialization, which allow for a more accurate selection of centroids and faster convergence of the algorithm.

In each step, the algorithm determines, for each point, to which centroid is closest to in order to determine the point's cluster membership. This operation is performed by calculating the Euclidean distance between each point and the various existing clusters.

Such distance is computed, considering two dimensions data points, as follows:

$$dist(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

After performing this operation, the algorithm calculates each cluster's new (updated) centroid by computing the average of the points within each cluster.

The algorithm can terminate according to various stopping criterions, such as reaching the convergence to a local optimum, reaching a predefined number of iterations, or if the cluster centroids' coordinates do not change of more than a certain threshold between the iterations.

2.2 Algorithm’s structure

The algorithm’s structure can be understood very easily by considering its pseudo-code.

Choice of the initial centroids

```
class KMeans
     method startCentroids(conf, inputFile, k, n)
          centroids[] <- {}
          for i in 1 to k do:
               extracted_index = Random(1, n)
               centroids[i] <- inputFile[extracted_index]
          return centroids
```

The choice of the starting centroids is performed by the simple method above. Random indexes in the input file containing all the data points are extracted and the array containing the initial centroids is initialized.

Mapper

```
class KMeansMapper
     method setup(context)
          centroids[] <- list of extracted centroids

     method Map(pointId, pointCoordinates, context)
          point <- init Point(pointCoordinates)
          minDistance <- POSITIVE_INFINITY
          closestCentroidIndex <- -1
          for i in 1 to centroids.length do:
               currentDistance <- distance(point, centroids[i])
               if currentDistance < minDistance then:
                    minDistance <- currentDistance
                    closestCentroidIndex <- i
          emit(closestCentroidIndex, point)
```

The Mapper is the element whose objective is determining, for each point of the dataset, its closest centroid. The `setup()` method is used to initialize the centroids array with the points extracted by the method described in the previous paragraph. The Mapper will then, for each point, compute the distance from it to the received centroids, finding the smallest one. At the end, it will emit a series of key-value pairs consisting of a point and the Id of the closest centroid.

Combiner

```
class KMeansCombiner
     method Reduce(centroidId, pointsList[], context)
          partialTotalPoint <- pointsList[0]
          for i in 1 to pointsList.length do:
               partialTotalPoint <- partialTotalPoint + pointsList[i]

          emit(centroidId, partialTotalPoint)
```

The Combiner class is used to reduce the workload received by the reducer and to minimize the network's data flow. In this case, the Combiner's task is to compute a partial sum of the points belonging to a cluster. The emitted values will then be the Id of each centroid along with the up-to-now summed points.

Reducer

```
class Reducer
     method Reduce(centroidId, pointsList[], context)
          total <- pointsList[0]
          for i in 1 to pointsList.length do:
               total <- total + pointsList[i]
          totalCoordinates[] <- total.coordinates
          for i in 0 to total.dimensions do:
               totalCoordinates[i] <- totalCoordinates[i]
                                      / total.aggregatedPoints
          total.coordinates <- totalCoordinates
          emit(centroidId, total)
```

The Reducer's goal is updating each centroid's coordinates. The Reducer receives key-value pairs, which are respectively the Id of a centroid and the list of points belonging to its cluster.

The execution of the method is simplified by the use of the Combiner that, if executed, has already computed some partial sums among points coordinates. In the Reducer, the sum is brought to the end, and the new centroid's coordinates are set as can be seen above. The emitted values will then be each centroid's Id and its new coordinates.

Driver code

```
class KMeans
	method main()
		threshold <- get threashold parameter
		clusters <- get #clusters parameter
		inputFile <- get input file parameter
		output <- get output location parameter
		iterationLimit <- get max # iterations parameter
		datasetSize <- get dataset size parameter
		currentCentroids[] <- {}
		previousCentroids[] <- {}
		for i in 1 clusters do:
			configuration.set(currentCentroids[i])
		iterationCounter <- 0
		while iterationCounter < iterationLimit do:
			job <- create Job
			setMapperClass(KMeansMapper)
			setReducerClass(KMeansReducer)
			setCombinerClass(KMeansCombiner)
			setNumReduceTasks(clusters)
			if iteration fails then:
				exit(error)
			currentCentroids <- getNewCentroids(configuration,
						clusters, output)
			stopCondition <- checkThreshold(currentCentroids,
						previousCentroids, threshold)
			iterationCounter <- iterationCounter + 1
			if stopCondition = true OR iterationCounter = iterationLimit
			then:
				break
			else:
				For i in 1 to clusters do:
					configuration.set(currentCentroids[i])
		print(currentCentroids[])
```

The driver code for this MapReduce application gets its parameters from command line and from a configuration file. These parameters are described in the following paragraph. An algorithm iteration takes place if and only if the stopping criterion is not met. This means that if the current centroids' coordinates differ from the previous ones by an amount larger than the threshold level, or if the number of completed iterations is smaller than the maximum number of possible algorithm runs, another iteration will take place.

Once the stopping criterion will be reached, the centroids will be printed in the console.

3. Implementation on Hadoop

The MapReduce application runs on a Hadoop cluster that has been set up as follows:

IP	Name-node	Data-Node	Host-name
10.1.1.68	Yes	Yes	Hadoop-namenode
10.1.1.98	No	Yes	Hadoop-datanode-1
10.1.1.103	No	Yes	Hadoop-datanode-2

The process is executed with a single MapReduce Job at a time.

Using the command line interface, the system must be provided with the necessary parameters to configure the job: the input filename (in the format datasetSize_pointDimensions.txt), the number of desired clusters, and the folder that will contain the outputs.

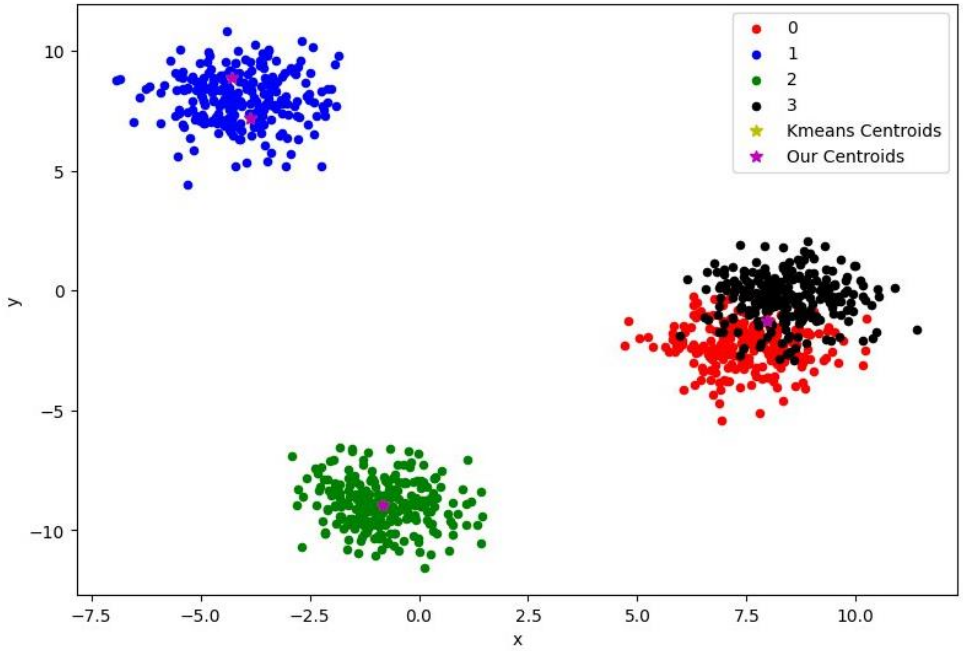
The maximum number of iterations and the threshold at which the algorithm will terminate are directly written and retrieved by the program inside a configuration file. As mentioned in the pseudocode section, such file is read by the job during the job's configuration phase, which takes place in the main.

4. Performance evaluation

Accuracy

The algorithm's accuracy has been visually estimated by comparing the result of the MapReduce implementation with the results of a python implementation of the same algorithm, which can be found in the `sklearn.cluster` library. This is an already extensively tested algorithm, so it can be considered a good choice to evaluate the MapReduce implementation's precision.

The following image shows an example of the results. As it can be seen, the golden stars (which represent the centroids computed via the Python's algorithm) are not visible, since they are entirely covered by the purple ones (i.e., the representation of the centroids computed via the MapReduce algorithm): this proves the high accuracy of the implementation.



Each algorithm run stopped after 8 iterations, and were provided the same initial centroids.

Execution time

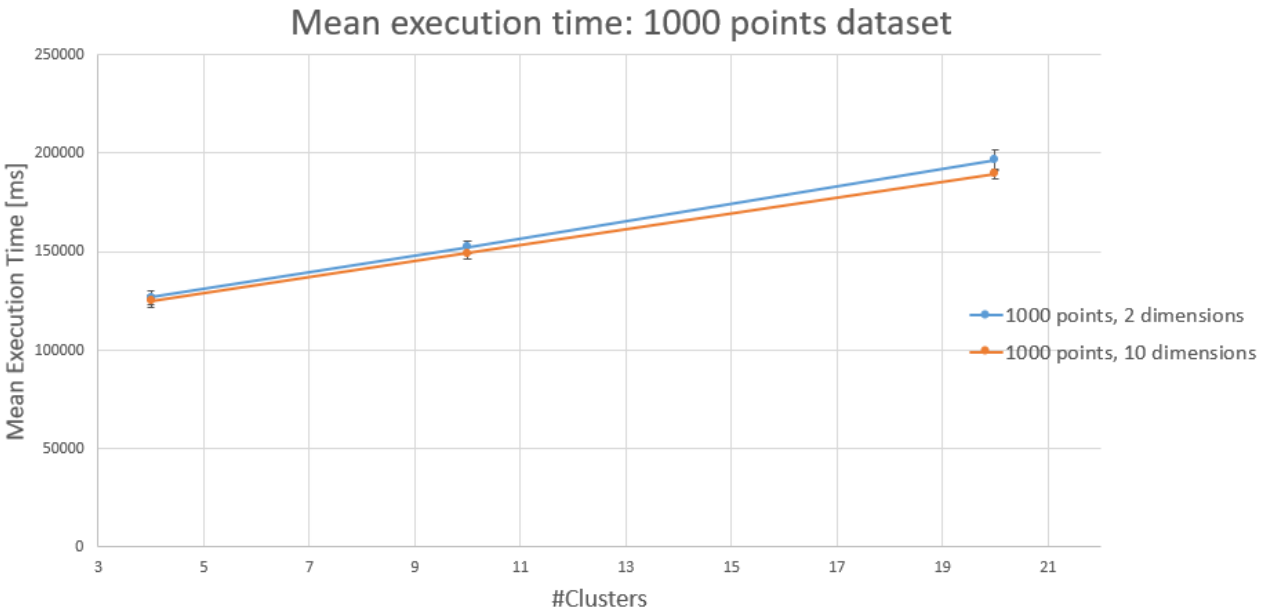
The measurement of the algorithm's execution time has been done by using java timers.

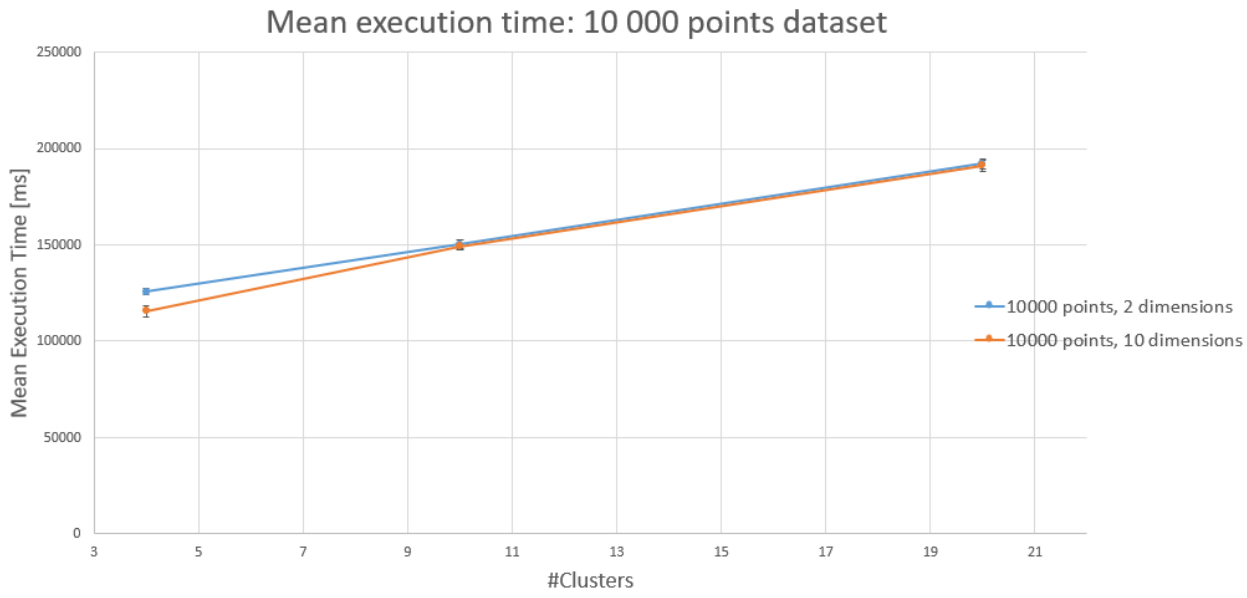
A study on the mean execution time has been brought up by varying the input parameters of the algorithm:

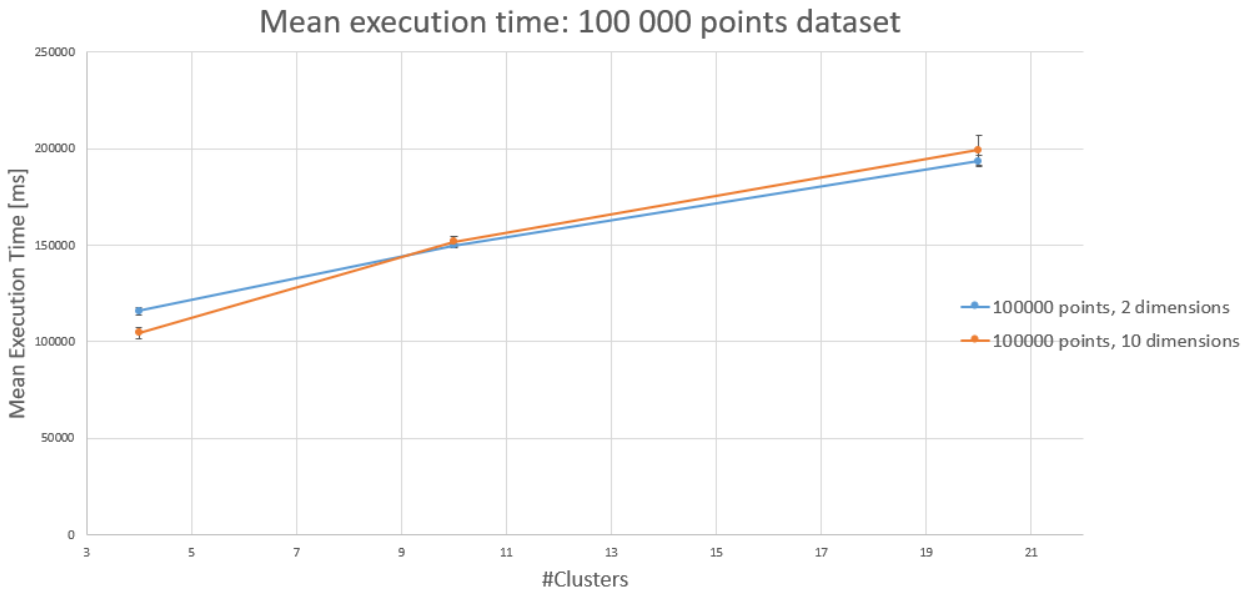
Clusters	4, 10, 20 clusters
Dataset size	1000, 10000, 100000 points
Points' dimensions	2, 10 dimensions
Use of the combiner	yes, no

For all the computations, a 95% confidence interval has been considered.

The results for the execution time without the aid of the combiner are shown in the following graphs:







The above data has been collected by setting the maximum number of iterations of each algorithm run to 5. This explains why the execution time does not increase when increasing the size of the dataset. On real world datasets, way bigger than the sample used ones, and with a higher number of iterations available, it is clear that the execution time will increase accordingly.

The study, anyway, shows that there is not a great dependence between the dimensionality of the points belonging to the used datasets and the increase in execution time. This can be explained by taking into account the very high speed at which mathematical computations and cycles are performed.

The parameter that makes the metric increase is clearly the number of clusters that needs to be determined by the algorithm: as this number increases, the algorithm will take more and more time to determine them all in a precise way. It has to be taken into account also another aspect: the initial centroids have been chosen, every time, at random. Surely, by using a more intelligent method to determine the starting points for the algorithm, the execution time can be greatly reduced. Some details about this issue are also reported in the following paragraphs.

About the use of the combiner to speed up the process

Since the mapper does not divide the input into splits, the performance measured using the combiner are almost the same as those measured without the combiner. For this reason, no graphs of execution time using the combiner have been included in this report, and considerations made in this paragraph and in the previous one apply to both scenarios. Among these, we can see that in some cases, we get lower execution times than others. This behavior is due to the random choice of the initial centroids that in some cases may have a position in space that will not differ much from the final one. So, after a few iterations the algorithm stops because the minimum threshold set in the configuration file is reached between the centroid components of the previous iteration and the centroid components of the next one.

5. Limitations and problems

As evidenced by the performance analysis above, we can observe that there are no significant differences in execution time between using a combiner and not using one, while it is well known that using one can increase the performance of the algorithm, in terms for example of speed and accuracy. This is due to an important limitation characterized by the fact that the mapper does not split the input data but considers the entire input file as a unique input split.

Consequently, the i-th combiner will only receive values with the exact key i, meaning values belonging to cluster i. As a result, the combiner will perform the partial sums instead of the reducer. Since the i-th reducer only receives values with the same key, it only needs to perform the division operation to calculate the new value of the i-th centroid, as the combiner has sent only partial sums with the same key.

This explains why there are no substantial performance differences between the version without a combiner and the version with a combiner.

Below, an example of the running code that communicates the number of splits the input dataset has been divided into, which in this case, is just one:

```
hadoop@hadoop-namenode:~/kmeans_project$ hadoop jar target/kmeans_project-1.0-SNAPSHOT.jar it.unipi.hadoop.KMeans 1939048_10.txt 4 output
[INFO] Configuration file 'configuration_file.xml' read
[INFO] Reading data from file '/user/hadoop/kmeans_datasets/1939048_10.txt'
[INFO] Running KMeans algorithm with parameters:
       -> points: 1939048
       -> clusters: 4
       -> stopping threshold: 0.01 (or when reached 5 iterations)
2023-06-27 14:17:00,100 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
[INFO] Initial centroids:
       (5.5126, -0.7697, -0.2978, -0.4285, -4.3147, -7.4156, -7.5701, -6.4947, -3.256, -10.6264)
       (6.8119, 4.9877, -0.8265, 4.5649, -3.5874, 2.3557, -5.7654, -9.6903, 6.6756, -1.7937)
       (10.1707, 3.5216, -5.5456, 5.873, -5.9711, 3.4004, -3.3924, 9.5811, -4.8107, -3.0213)
       (7.2165, 7.0192, 1.2983, 5.6427, -4.1599, 3.4627, -6.29, -9.4427, 6.3597, -3.2496)
[INFO] Iteration 1 started
2023-06-27 14:17:01,038 INFO input.FileInputFormat: Total input files to process : 1
Number of input splits: 1
2023-06-27 14:17:01,162 INFO client.RMProxy: Connecting to ResourceManager at hadoop-namenode/10.1.1.68:8032
2023-06-27 14:17:01,614 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/hadoop/.staging/job_1687874120732_0010
2023-06-27 14:17:01,792 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2023-06-27 14:17:01,899 INFO input.FileInputFormat: Total input files to process : 1
2023-06-27 14:17:01,930 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2023-06-27 14:17:01,961 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2023-06-27 14:17:01,976 INFO mapreduce.JobSubmitter: number of splits:1
2023-06-27 14:17:02,135 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2023-06-27 14:17:02,182 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1687874120732_0010
2023-06-27 14:17:02,183 INFO mapreduce.JobSubmitter: Executing with tokens: []
2023-06-27 14:17:02,423 INFO conf.Configuration: resource-types.xml not found
2023-06-27 14:17:02,424 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2023-06-27 14:17:02,507 INFO impl.YarnClientImpl: Submitted application application_1687874120732_0010
2023-06-27 14:17:02,559 INFO mapreduce.Job: The url to track the job: http://hadoop-namenode:8088/proxy/application_1687874120732_0010/
2023-06-27 14:17:02,561 INFO mapreduce.Job: Running job: job_1687874120732_0010
2023-06-27 14:17:09,791 INFO mapreduce.Job: Job job_1687874120732_0010 running in uber mode : false
2023-06-27 14:17:09,792 INFO mapreduce.Job:  map 0% reduce 0%
2023-06-27 14:17:16,899 INFO mapreduce.Job: Task Id : attempt_1687874120732_0010_m_000000_0, Status : FAILED
[2023-06-27 14:17:15.758]Container [pid=2213157,containerID=container_1687874120732_0010_01_000002] is running 58695688B beyond the 'PHYSICAL' memory limit. Current usage: 261
.6 MB of 256 MB physical memory used; 1.7 GB of 537.6 MB virtual memory used. Killing container.
```

Considering that the chunk size on the HDFS is 128 MB, we attempted to load a dataset of 180 MB, expecting it to be split by the mapper during the execution. However, this did not happen.