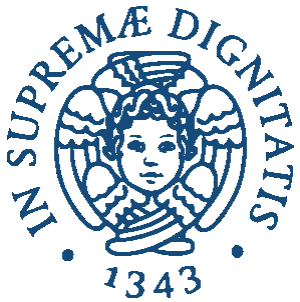

Cloud Computing

The k-means Clustering Algorithm in MapReduce

Ayoub El Ourrak , Jacopo Carlon , Nicola Riccardi

Academic Year 2022-2023

Contents

1 Design

1.1 Clustering : K-Means Algorithm

Clustering is a method of organizing comparable unlabeled data points into separate groups, called *clusters*, using data patterns. The goal is to ensure that the data points in each cluster are highly similar to one another.

The **K-Means algorithm** is a clustering algorithm that tries to partition a dataset of **n** elements into **K** pre-defined distinct and non-overlapping subgroups (**clusters**, where each data point belongs to only one of them).

In order to ensure that elements withing a group are "as similar as possible", the points are assigned to a cluster in such a way to **minimize** the *Euclidean distance* between the data points and the cluster's **centroid** (which is the arithmetic mean of all the data points that belong to its cluster).

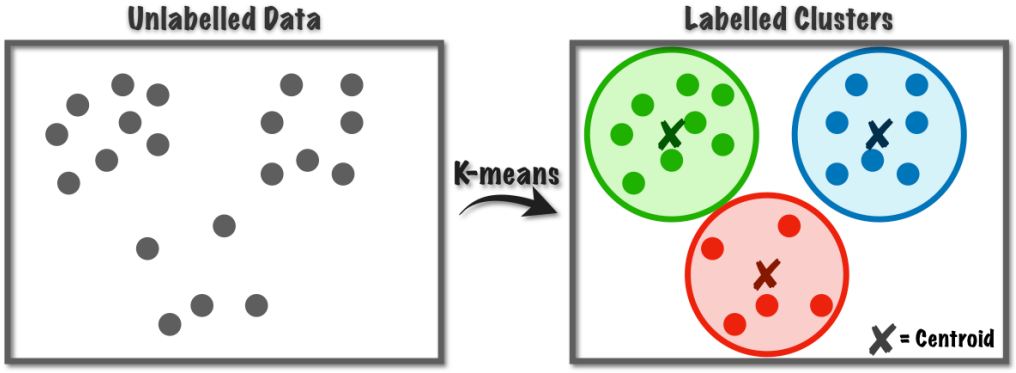

Figure 1: K-Means clustering algorithm

The K-Means clustering algorithm logically works as follows :

Algorithm 1: K-Means clustering algorithm

Input set X of n data in d dimensions : $X = x_1, ..., x_n$
number K of clusters to find
Output: set of M of K means in d dimensions $M = \mu_1, ..., \mu_k$
K-Means(X,K) :
The means $\mu_1, ..., \mu_k$ are randomly sampled from X
while a stopping condition has not been met **do**
  for each mean μ_i in M **do**
    $\omega_i \leftarrow \{\}$
  for each data point x_i in X **do**
    c $\leftarrow argmin_j \mid\mid x_i - \mu_j \mid\mid_2^2$
    $\omega_c \leftarrow \omega_c \cup \{x_i\}$
  for each mean μ_i in M **do**
    $\mu_i \leftarrow (\sum_{X \in \omega_i} x) * \frac{1}{|\omega_i|}$
return $\mu_1, ..., \mu_k$

In our implementation, we iterated the algorithm using the following stopping conditions :

- the sum of distances between centroids of consecutive iterations is below a fixed threshold;
- number of iterations above a maximum;
- time expired over a limit.

The K-Means algorithm has several advantages, such as its simplicity of implementation and scaling. It also garantees convergence, and allows to warm-start the position of the centroids, and can easily be adapted to new sets, or be generalized to clusters of different shapes or sizes.

1.2 K-Means Parallelization : Apache Hadoop

The **Apache Hadoop** software library is an open-software framework that allows for the distributed processing of large data sets across clusters of computers, using simple programming models.

It works with a `divide et impera` approach, by breaking data into smaller workloads and distributing them across *nodes* in a computing cluster. This approach is referred to as the "**MapReduce**" programming module, which allows **parallel processing of huge datasets**.

In order to parallelized the execution of K-Means, we applied the MapReduce paradigm as follows :

- we use the ***map*** function to assign a point to the centroid closest to it (min-square-distance)
- we use the ***reduce*** function to calculate the coordinates of a new centroid

With this approach, the data are initially divided into sub-portions, more manageable by the nodes. The nodes then performed calculations, and we merged the obtained results.

1.2.1 Mapper

In order to implement the parallelized version of the K-Means algorithm, we implemented a map function. The aim of this function is to associate each point in the starting dataset with the nearest centroid from a list of centroid. The map functions therefore iterates over the list of centroids for each data point, calculating the squared distance (SD) between the point and the centroid.

The list of centroids is initialized in the setup method of the mapper, by reading the coordinates of each centroid in the job configuration, in which they are loaded before starting the job.

Algorithm 2: Setup - Mapper

```
Procedure SETUP(Configuration)
    centroidsList ← GetCentroidsFromConfiguration(Configuration)
```

Algorithm 3: `Map` Function - Mapper

```
Procedure MAP(key, point)
    nearest_centroid ← null
    squaredDistanceMin ← Double.MAX_VALUE
    for all centroid in centroidsList do
      squaredDistance ← DISTANCE(centroid, point)
      if squaredDistance < squaredDistanceMin then
        squaredDistanceMin ← squaredDistance
        nearest_centroid ← centroid.getID()
    Emit(nearest_centroid, point)
```

1.2.2 Reducer

The output of the Mapper is given as input to the Reducer, which will process the clusters and produce a new set of outputs, which will be stored in **HDFS**(Hadoops Distributed File System).

The Reducer is responsible for aggregating and processing all data with the same key. The ***reduce*** method takes as input a centroidID and the list of the partial sums of the points associated to that centroid. It iterates over the list, summing the coordinates of the points. It then agerages this sum to extract the position of the new centroid, and finally emits a new pair of <ID of the centroid, istance of the new centroid with the updated coordinates>.

Algorithm 4: `Reduce` Function - Reducer

```
Procedure REDUCE(centroidID, partialSumsList)
    finalSums ← sum(partialSumsList)
    newCentroid ← finalSums.average()
    Emit(centroidId, newCentroid)
```

1.2.3 Combiner

We also implemented a Combiner. Its primary job is to process the output data of a Mapper, before passing it to a Reducer, and is used to minimize the data that has been shuffled between Map and Reduce.

Its *Reduce* function calculates the partial sum of all data points associated with a given centroidID, and does so by iterating over the points and calling the add method on partialSum. Finally, it emits a new pair of <centroidId, partialSum >.
By performing this local aggregation before passing the data, we lessen the amount of data that needs to be transferred between the mappers and reducers, thus improving performance and reducing network congestion amongst the interested computing nodes (this is increasingly important as the amount of data increases).

Algorithm 5: `Reduce` Function - Combiner

```
Procedure REDUCE(centroidID, partialSums)
    partialSum ← sum(partialSums)
    Emit(centroidId, partialSum)
```

1.2.4 In-Mapper Combining

We also implemented an alternative Mapper with **stateful in-mapper combining**, to compare performances of the two combining methods.

This version of the Mapper keeps track locally of the partial sums of data points assigned to each centroid, by using an `HashMap` initialized in the `setup` method.

The `map` method finds the nearest centroid for each data point in the same way as the other Mapper, but instead of emitting the pair {*centroid, point*}, the coordinates of that point are added to the partial sum corresponding to that centroid.

After all points have been processed, the **cleanup** method will emit a pair {*centroid, partialSum*} for each element in `centroidList`. By performing this local aggregation before passing the data, each Mapper will emit only one *key-value* pair for each centroid, instead of one couple for each data point it has processed.

Algorithm 6: `Map` Function - Mapper&Combiner

```
Procedure MAP(key, point)
    nearest_centroid ← null
    squaredDistanceMin ← Double.MAX_VALUE
    for all centroid in centroidsList do
        squaredDistance ← DISTANCE(centroid, point)
        if squaredDistance < squaredDistanceMin then
            squaredDistanceMin ← squaredDistance
            nearest_centroid ← centroid.getID()
    partialSumsMap{nearest_centroid} ← SUM(partialSumsMap{nearest_centroid}, point)
```

Algorithm 7: Setup - Mapper&Combiner

```
Procedure SETUP(Configuration)
    centroidsList ← GetCentroidsFromConfiguration(Configuration)
    partialSumsMap ← new HashMap
```

Algorithm 8: Cleanup - Mapper&Combiner

```
Procedure CLEANUP(Configuration)
    for all (centroidID, partialSum) in partialSumsMap do
        Emit(centroidID, partialSum)
```

2 Implementation

In our implementation of the K-Means algorithm with the Hadoop MapReduce, we have utilized different classes:

- **KMeans**,
- **Utils**,
- **Cluster**,
- **Centroid**,
- **KMeansMapper** (described in section 1.2.1),
- **KMeansCombiner** (described in section 1.2.3),
- **KMeansReducer** (described in section 1.2.2),
- **KMeansMapAndCombine** (described in section 1.2.4).

2.1 KMeans

The **KMeans** class covers the logic for running the K-Means algorithm with the Hadoop Mapreduce, and contains the entry point for the program execution.

It has a single method, the **main** :

- public static void **main(String[] args)**

  This is the entry point of our K-Means program. It takes arguments from the command line in the following format : *[INPUT_PATH], [OUTPUT_PATH], [CONFIG_FILE_PATH].*

  By accessing `CONFIG_FILE_PATH`, it **extracts the Hadoop Configuration** needed to initialize the system: the number **n** of points in the dataset, the number **K** of target cluster, the dimensions **d** of the points (d must be at the least 2), the iteration-stopping-threshold, the maximum number of iterations (maxIter), the maximum execution time (in seconds), the desired number of reducers, and if in-Mapper Combining should be used.

  It then randomly extracts from the dataset **X** a number K of initial centroids, needed for the first step of the algorithm.

  Finally, it begins iterating the algorithm until maxIter is reached, or until another stopping condition is met.

  - First it configures and submits the MapReduce job using the *configureJob* method of the Util class, and then waits for its completion.
  - when the job is concluded, it **gets the new list of centroids**, which is obtained by reading the current and previous centroids from multiple or a single file (depending on the number of reducers) and by then calculating the centroid shift, using the *getCentroidsFromFiles* method of the Utils class.
  - **If the shift of centroids is below the configurated threshold**, we consider that **the algorithm has converged** and the iteration is then stopped.

    Otherwise, the centroids are updated and the iterations continues until a stopping condition is met (shift below threshold, time limit or iteration limit).

2.2 Utils

This is an **utility class** called **Utils**, and offers helper methods to the K-Means algorithm. The methods provided by this class are:

- public static boolean **cleanFile(Configuration config, Path filePath)**

  This method clears the filePath if it exists.

- public static List<Centroid> **getRandomCentroids(Configuration config, Path inputPath, int nCentroids, int nPoints, int nCoordinates)**

  This method accesses the *inputPath* file and extract from the *nPoints* points inside the dataset, a number *nCentroids* of initial points, which are then returned as first random choice used for the algorithm initialization.

  If the dataset did not cointain the correct number of points nPoints, or if any other operation fails, then the function returns an empty list, otherwise it returns the extracted list of Centroids.

- public static List<Centroid> **getCentroidsFromFile(Configuration config, Path inputPath)**

  This method extracts all centrods from the given inputPaht file, and adds them to a newCentroind List<Centroid>, which is then returned.

- public static float **calculateDistanceSum(List<Centroid> previousCentroids, List<Centroid> newCentroids)**

  This method iterates over the centroids (ordered by ID), and calculates how much they shifted from the previous iteration to the current one, then sums all these shifts and returns the result.

  Since the centroids have coordinares in d dimensions, we first get the squareDistance as sum of the square of the deltas for each dimension between two centroids with the same id, and then we extract the corresponding square root.

  The effective forumla is therefore:

$$distanceSum = \sum_{i=1}^{k} \sqrt{\sum_{j=1}^{d} (x_{ij} - y_{ij})^2} \tag{1}$$

  which is

$$distanceSum = \sum_{i=1}^{k} |x_i - y_i| \tag{2}$$

  where:

  - k is the number of centroids
  - d is the number of dimensions of the points in the dataset (that is therefore the dimension of the centroids)
  - x_{ij} is the j-th coordinate of the i-th centroid in the previous iteration
  - y_{ij} is the j-th coordinate of the i-th centroid in the current iteration
  - x_i is the i-th centroid of the previous iteration considered as a point in a $\mathbb{R}^d$ space
  - y_i is the i-th centroid of the current iteration considered as a point in a $\mathbb{R}^d$ space
  - $||$ represents the Euclidean Norm on a $\mathbb{R}^d$ space.

  The argument of the square root is obtained by calling the Cluster method "getSquareDistance" on the `previousCentroids`'s i-th centroid, and as argument the `newCentroids`'s i-th centroid.

- public static Job **configureJob(Configuration config, int nCentroids, List<Centroid> iterationCentroids, int iteration, Path inputPath, Path outputPath, boolean inMapperCombining)**

  This method **configures a MapReducer job** for a K-Means iteration. It takes as input the Hadoop configuration, the input and output paths, the number of centroids and the current iteration number.

  It returns the configured job.

- public static String **newLog(int nPoints, int nCoordinates, int nCentroids, int nReducers, boolean inMapperCombining)**

  This method **creates a new log file**. Its name will be created using the arguments of the function and will be returned.

- public static void **logInfo(String logFile, String string)**

  This method **logs generic informations** to the specified log file.

2.3 Cluster

The **Cluster** class represent a data point, or the sum of data points, in the K-Means algorithm, and provides a number of operations to ease the manipulation of points, and to calculate distances between them. The Cluster has the fields:

- private **int size** This is the size of the cluster (the number of data points that were aggregated inside it)
- private **List<Float> coordinates** This is the list of sums of the coordinates of all the data points that have been aggregated inside this cluster. If size is 1, they are the coordinates of a single point.

The methods provided by this class are:

- public **Cluster()**

  This constructor **creates a new empty Cluster object**. It initializes the coordinates field to null and sets the size field to 0.

- public **Cluster(List<Float> coordinates)**

  This constructor **creates a new Cluster object using the specified coordinates**. It initializes the coordinates field and sets the size field to 1.

- public **Cluster(String text)**

  This constructor **creates a new Cluster object from a semicolon-separated string** containing the coordinates (text). It splits the string and uses the add method to add them to the coordinate field of the Cluster object.

- public List<Float> **getCoordinates()**

  This method **returns the list of coordinates** of the cluster (this means it returns one value per dimension of the space).

- public int **getSize()**

  This method **returns the size field** of the cluster, which is the number of points this cluster has aggregated.

- public void **addPoint(Cluster point)**

  This method **adds the points of the specified cluster** to the current Cluster object, meaning that it adds the j-th coordinate sum of the parameter object to the j-th of the current object. The size of the current Cluster object is increased accordingly.

- public Float **getSquaredDistance(Cluster point)**

  This method **calculates the squared Euclidean distance**(since we don't extract here the square root) between the current Cluster object and the specified cluster `point` passed as parameter.

  The formula is:

  $$squaredDistance = \sum_{i=1}^{d} (x_i - y_i)^2 \tag{3}$$

  Where

  - d is the number of dimensions of the data points,
  - x_i is the i-th coordinate (which means the coordinate for dimension i) of the current Cluster.
  - y_i is the i-th coordinate (which means the coordinate for dimension i) of the parameter `point` Cluster.

- public Cluster **getMean()**

  This method **calculates the average of the Cluster's sum** by dividing each coordinate by `size`, and returns a newly generated cluster.

  This means that effectively we have added point to a cluster by summing their coordinates (respective per each dimension), and now we divide by the number of points we have added together, and we generate the mean of these points (each coordinate is the mean of the coordinates, with respect to each dimension of the space).

- public void **write(DataOutput out)**

  This method writes the Cluster object to the specified `DataOutput` stream.

- public void **readFields(DataInput in)**

  This method reads the Cluster object from the specified `DataInput` stream.

- public String **toString()**

  This method returns a **string representaton of the Cluster object** by converting its coordinates to a string and concatenating them **separated by a semicolon**.

2.4 Centroid

The Centroid class represent a centroid in the K-Means algorithm. The fields of the Centroid class are:

- private final **IntWritable centroidId**: This is the id of the centroid, and is unique
- private **Cluster point**: This is the Cluster associated to the centroid, which is used to save its coordinates (in the d dimension of the space we are working in) in this iteration.

The methods provided by this Centroid class are:

- public **Centroid(int centroidId, Cluster point)**

  This constructor **initializes the Centroid object** with the given **centroidId and Cluster** `point`.

- public **Centroid(String string)**

  This constructor **initializes the Centroid object** from a string split with `'\t'`. The first element of the split string is expected to be the **centroidID** the remaining other (one) is used to generate a Cluster object (therefore it should be a string of d semicolon-separated coordinates).

- public IntWritable **getCentroidId()**

  This method returns the `centroidID` field of this centroid.

- public Cluster **getPoint()**

  This method provides the `point` field of this centroid, which is a Cluster object.

- public int **compareTo(Centroid other)**

  This method compares the Centroid object with the parameter Centroid `other` for sorting purposes.

- public void **write(DataOutput out)**

  This method writes the Centroid object to the indicated `DataOutput` stream.

- public void **readFields(DataInput in)**

  This method reads the Centroid object from the indicated `DataInput` stream.

- public String **toString()**

  This method converts the Centroid object to a string composed by the string-concatenation of

  - the stringification of this Centroid's IntWritable field `centroidId`,
  - a separator "`\t`" ,
  - the stringification of this Centroid's Cluster field `point`.

3 Testing Phase

This section presents the resultsobtained from implementing the KMeans algorithm using Hadoop MapReduce. The algorithm was evaluated on diverse datasets, with variations in the number of points (N), number of centroids (K), and dimensionality of points (D). Datasets were generated using a python script based on `make_blobs` module from *scikit-learn*. Each dataset consists of a specified number of samples, represented as points in a specified dimensional space. Each point has normalized coodinates between 0 and 1, so that each dataset has the same scale.
The testing phase was divided in 3 sub-phases:

- Assessing the **algorithm accuracy**,
- Comparing the two implemented **combining methods**,
- Evaluating the performances with a **varying number of reducers**.

3.1 Accuracy Tests

The primary goal of this phase is to access the performance of the developed algorithm in terms of **efficiency and accuracy**.

Specifically, we employed a **silhouette analysis** using the `silhouette_score` module [1] from *scikit-learn* to evaluate the centroids found by our algorithm.

This technique assesses the similarity of each data point to the nearest centroid in comparison to the other centroids. The analysis yields a **silhouette score**, which is a value within the range `[-1,1]`. A score near to 1 means that most of the data points are well assigned to a cluster centered on their nearest centroid, whereas a low score suggests that the data points might be better suited to another cluster.

Additionally, each test involving a dataset having **2 or 3 dimensional points** was evaluated graphically, by plotting its data points and centroids.

Silhuette Score Results			
Number of Points	Dimensionality	Number of Clusters	Silhouette Score
1000	2	3	0.79306
1000	3	5	0.81860
10000	2	5	0.75437
10000	3	7	0.71404
50000	2	5	0.75509
50000	3	5	0.52363
100000	5	7	0.61045
100000	7	7	0.59835

Table 1: Silhouette Score Table

In light of the possible range of the silhouette score, usually results **above 0.5** are usually considered to be reasonably good.

The algorithm did not manifest noticeable trends with regards to dimensionality, but the results did worsen as the number of points increased.

The following images are the plots of the datasets with the computed centroids, where the number of dimensions is 2 or 3.

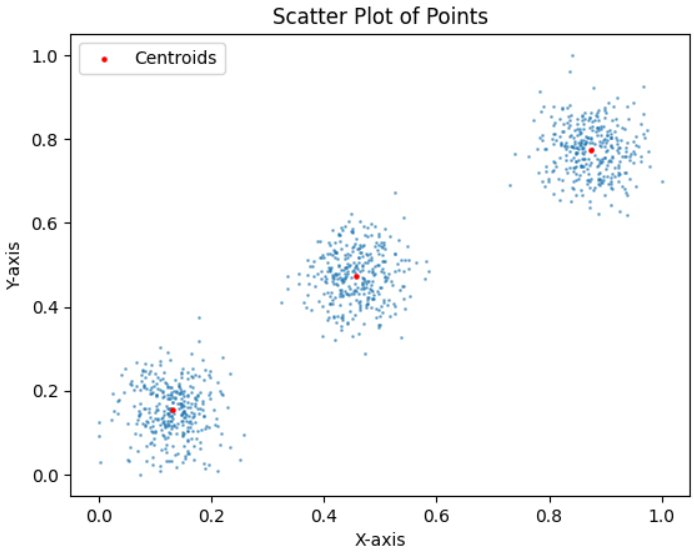

Figure 2: N=1000, D=2, K=3

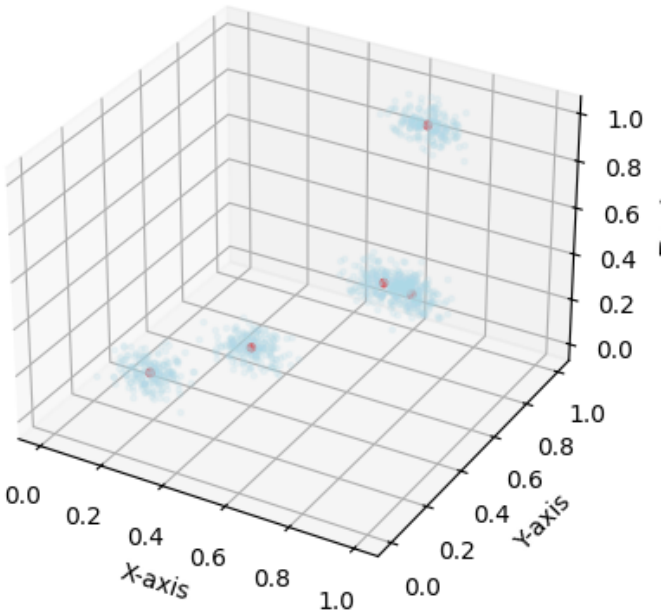

Figure 3: N=1000, D=3, K=5

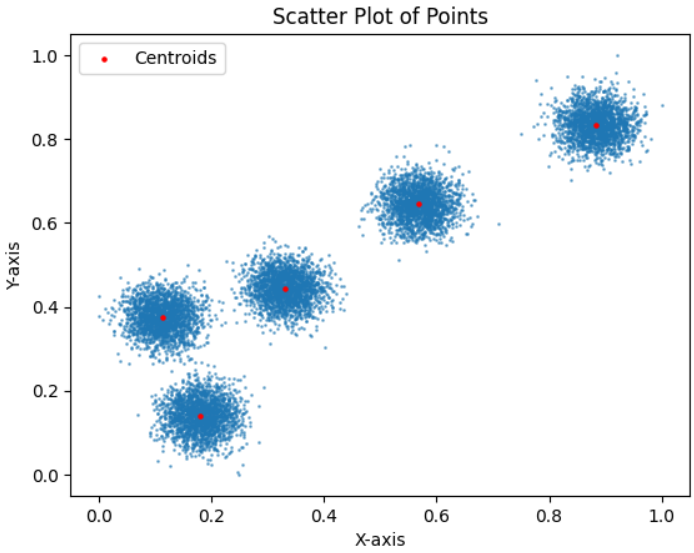

Figure 4: N=10000, D=2, K=5

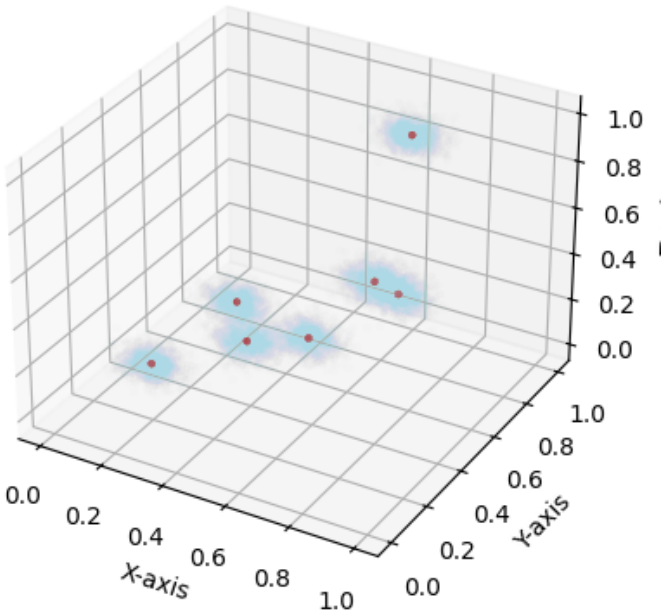

Figure 5: N=10000, D=3, K=7

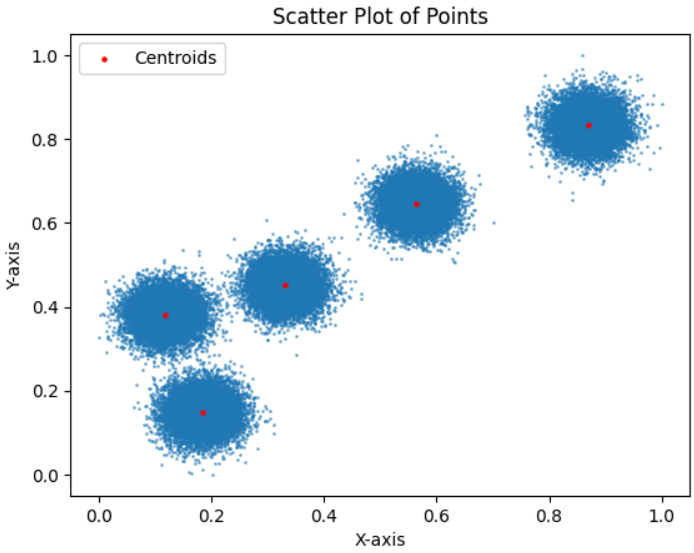

Figure 6: N=50000, D=2, K=5

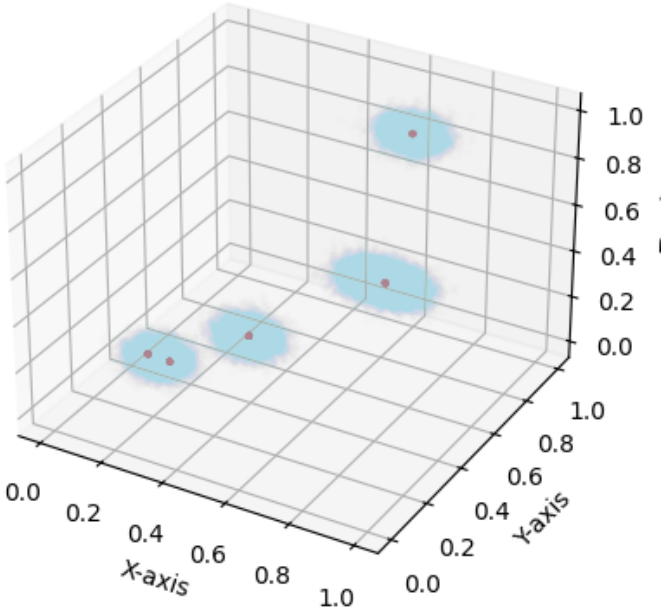

Figure 7: N=50000, D=3, K=5

3.2 Combiner and in-Mapper Combining comparison

In the second phase of testing we compared the performance of two MapReduce configurations that optimize the data processing: the **Combiner** and **in-Mapper Combining**. The Combiner is a built-in feature that aggregates intermediate key-value pairs at the mapper nodes before sending them to the reducer. This helps reducing the volume of data shuffled over the network. On the other hand, in-Mapper Combining involves a custom implementation where the mapper function itself performs local aggregation before emitting key-value pairs.

Specifically, we tested both configuration against the same datasets and measured the average computational time per iteration.

Mean Iteration Time (in seconds)		
Dataset	External Combiner	In-Mapper Combining
N=100000,D=5,K=7	26.4146	26.8089
N=100000,D=7,K=7	27.2023	27.4608
N=150000,D=7,K=10	28.8177	27.4098
N=200000,D=10,K=10	28.8388	26.5278
N=1000000,D=50,K=10	44.7934	40.5422

Table 2: Performance comparison of combining approaches

As can be seen in the table above, there is almost no difference between the two approaches until the amount of data becomes huge. This difference can be attributed to the **cost of creating and distributing intermediate data**, that are much less when using In-Mapper Combining.

3.3 Number of Reducers

Finally, we examined the impact of the number of reducers on algorithm execution. Hadoop's documentation suggests selecting the number of reducers equal to the **number of nodes multiplied by 0.95 or 1.75**, so, since we have 3 nodes, we decided to run the algorithm over the same dataset varying the number of reducers between 1 and 6.

The following results were obtained on a dataset of 200000 points of 10 dimensions, divided in 10 clusters, using the in-Mapper combiner.

Iteration Time Results	
Number of Reducers	Mean Iteration Times (in seconds)
1	26.5278
2	27.6018
3	26.8480
4	29.6937
5	38.7256
6	43.7086

Table 3: Performance with different number of reducers

These results suggest that, in our particular case, the **framework overhead** and the **cost of transferring data** have a greater impact on performance than the benefits of a better load balancing and resource utilization. However, the results obtained for different numbers of reducers can be influenced by **other several factors**, including the hardware configuration, network bandwidth, and the characteristics of the dataset.

4 Conclusions

The implementation of the K-Means algorithm in Hadoop, following the MapReduce paradigm, yielded acceptable outcomes. Especially, the program demonstrated effectiveness in achieving its objectives across diverse datasets, varying point dimensions, and different cluster numbers. The resulting centroids where for the most part accurate even if there are opportunities for improvement.

One aspect in particular is the exploration of the initialization of centroids; relying solely on the first random set led to sub-optimal solution, so considering other alternative approaches could probably further enhance the results. Additionally experimenting with datasets featuring larger point dimensions and numbers could have proven more insight on the advantages of increasing the number of reducers.

5 Bibliography

References

[1] *Silhouette_score*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html.