UNIVERSITÀ DI PISA

Computer Engineering

Cloud Computing

K-Means Clustering Algorithm in MapReduce

Team members:

Francesco Martoccia

Salvatore Lombardi

Luca Tartaglia

ACADEMIC YEAR 2022/2023

Contents

1 - Design

1.1 Introduction

Clustering is a method that allows organizing a set of unlabeled data points into separate groups, called clusters. The idea consists of ensuring that the data points in each cluster are highly similar to one another.
K-Means algorithm is an iterative algorithm that tries to partition the dataset into **K** pre-defined distinct non-overlapping subgroups (**clusters**) where each data point belongs to only one of them. To make the elements within a group as similar as possible, each of them is assigned to a cluster in such a way that the Euclidean distance between the data points and the cluster's **centroid** (arithmetic mean of all the data points that belong to that cluster) is at the minimum.

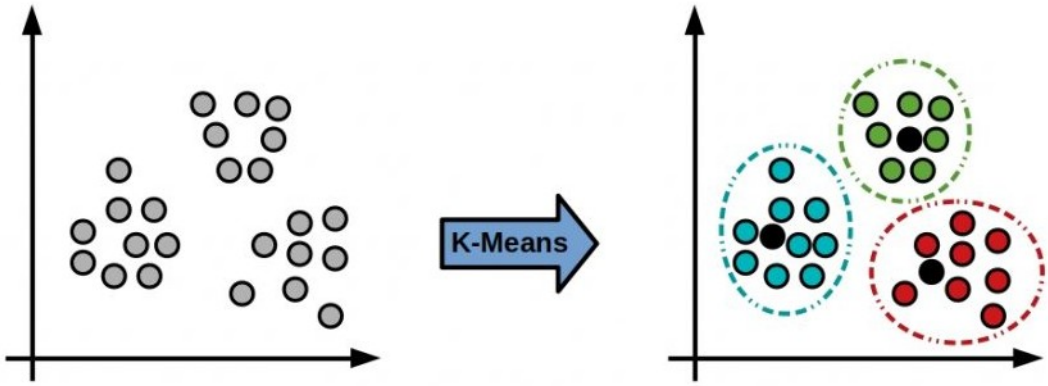

Figure 1.1: Basic principle of the K-Means clustering

1.2 MapReduce Approach

The **Apache Hadoop** software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is based on the **MapReduce** programming model, which allows for the

parallel processing of huge datasets. Hadoop distributes data across nodes in a computing cluster, breaking them down into smaller workloads that can be run in parallel.
To parallelize the execution of K-Means, the MapReduce paradigm was applied during the algorithm design phase.
Specifically, for each **MapReduce job**, considering a list of initially computed centroids, the following steps were to be performed:

1 - Mapping a point with the centroid considered closest to it (using the ***map*** function)

2 - Calculating the coordinates of a new centroid (using the ***reduce*** function)

Through this approach, the data are initially divided into portions that are more manageable by the nodes, which perform calculations on them, and, finally, by combining the results produced, it is possible to reach the goal.
In addition, this leads to better performance, as it brings about a reduction in computation time.

1.2.1 Mapper

To follow the MapReduce paradigm, we started by defining the behaviour of the map function.
Before that, for instantiating the initial list of centroids to be able to compare with each point, the ***setup*** method was implemented, which allows its loading from the Hadoop configuration.

```
1 class Mapper {
2     method Setup(config) {
3         centroids ← loadCentroids(config)
4     }
5 }
```

Mapper: Setup function pseudocode

For each data point, it is necessary to identify which centroid in the list is **closest** to it.
Given a point, the ***map*** function iterates through the list of centroids and calculates the **Euclidean distance** it has to each of them. Whenever the latter is smaller than the one previously computed, the information of the nearest centroid is **updated** and saved.
Once the function has determined the nearest centroid for the data point, it emits a key-value pair (**ID of the closest centroid**, **point instance**).

```
1 class Mapper {
```

```
    method Map(key,value) {
        coordinates ← splitInCoordinates(value)
        centroidID ← null
        point ← new Point(coordinates)
        distanceFromCentroid ← Double.MAX_VALUE
        for all centroid ∈ centroids do
            currentDistance ← distance(point,centroid)
            if currentDistance < distanceFromCentroid then
                centroidID ← centroid.centroidID
                distanceFromCentroid ← currentDistance
        emit(centroidID,point)
    }
}
```

Mapper: Map function pseudocode

1.2.2 Combiner

We decided to optimize the overall operations by designing the behaviour of a Combiner, which is in charge of doing **local aggregation of the data** before they are shuffled and transferred from the Mapper to the Reducer.
Since the reduce function in our case is **commutative** and **associative**, the combiner is very useful, as it allows the same operation to be performed locally on the mapper side, **minimizing** the amount of data to be sent over the network.
Given a centroid, the ***reduce*** function iteratively calculates and updates the **partial sum of the coordinates** of the points associated with it.
At the end of this computation, the method emits a key-value pair (**ID of the considered centroid**, **partial sum of the coordinates**). This output is the aggregated result for a particular centroid.

```
class Combiner {
    method Reduce(centroidID,associatedPoints) {
        partialSum ← sumCoordinates(associatedPoints)
        emit(centroidID,partialSum)
    }
}
```

Combiner: Reduce function pseudocode

1.2.3 Reducer

The Reducer is responsible for aggregating and processing data with the same key. The ***reduce*** method takes as input a centroid ID and the list of all the partial sum of the points associated with it.
Iterating over this list, it sums all the coordinates of the points. Having done so,

it calculates the new centroid position by averaging over the previously obtained sum. Finally, the method emits a key-value pair (**ID of the considered centroid**, **instance of a new Centroid object with its updated coordinates**).

```
class Reducer {
    method Reduce(centroidID,partialSumsList) {
        sum ← sumCoordinates(partialSumsList)
        newCentroid.coordinates ← averageCoordinates(sum)
        emit(centroidID,newCentroid)
    }
}
```

Reducer: Reduce function pseudocode

2 - Implementation

This chapter offers an in-depth exploration of the organization and structure of our **K-Means clustering** hadoop application. The program's architecture is characterized by a set of well-defined classes, categorized into packages to maintain modularity and clarity.
The chapter is divided into three sections, each focusing on a distinct aspect of the program:

- **Model**: examines the core data model of the application, outlining the essential classes used for representing **points** and **centroids**.
- **Execution and Control**: describes the control flow and orchestration of the K-Means program.
- **MapReduce**: provides details related to the implementation of the MapReduce-specific components of the program.

2.1 Model

The classes described here are ***Point*** and ***Centroid***, and both are part of the package *it.unipi.dii.hadoop.model*.

2.1.1 Point

The ***Point*** class represents individual data points in the K-Means program. It implements the ***Writable*** interface, allowing Point objects to be **serialized** and **deserialized** when reading and writing data within the Hadoop framework.
The class has the following **properties**:

- *List<Double> coordinates*:

  A list of coordinates in multi-dimensional space.

- *int partialPointsCounter*:

  A counter that maintains the number of points from which the current coordinates were derived.

The main **methods** implemented in this class are the following:

- *public void sumCoordinates(Point point)*:

  Sums the coordinates of the current point with those of another point, passed as parameter.

- *public void averageCoordinates()*:

  Computes the **average value** of each coordinate based on the number of points used to make the sum (*partialPointsCounter*, which is finally set to 1), and updates the *coordinates* property accordingly.

Since the class implements the ***Writable*** interface, it was necessary to override and implement the following methods:

- *public void write(DataOutput dataOutput)*:

  Is responsible for the **serialization** of a *Point* object into a format that can be written to an output data stream. It writes the size of the *coordinates* list, followed by the individual coordinates and the *partialPointsCounter*.

- *public void readFields(DataInput dataInput)*:

  Allows the **deserialization** of a Point object from the input data stream. It reads data from the stream and reconstructs the Point object, populating its *coordinates* and *partialPointsCounter* fields.

Additionally, the *toString* method was also overridden to enable object representation as a string, considering only its coordinates.

2.1.2 Centroid

The ***Centroid*** class is responsible for representing the cluster centroids in the K-Means algorithm.
The class possesses the following **properties**:

- *IntWritable centroidID*:

  An ID associated with the centroid.

- *Point point*:

  A Point object representing the coordinates of the centroid.

Since the class implements ***WritableComparable***, the **methods** to implement are the following:

- *public int compareTo(Centroid centroid)*:

  Facilitates the efficient **comparison** and **sorting** of Centroid objects. In the context of the MapReduce **shuffle and sort** phase, this method allows Centroid objects to be sorted based on their *centroidID*, which is critical for grouping and processing data points within K-Means clusters.

- *public void write(DataOutput dataOutput)*:

  It's responsible of a Centroid object's **serialization** into a format that is writable to an output data stream. It calls the *write* methods of the *centroidID* and *point* properties. In this way, both of them are written as bytes to the output stream.

- *public void readFields(DataInput dataInput)*:

  Allows the **deserialization** of a Centroid object from the input data stream. It calls the *readFields* methods of the *centroidID* and *point* properties. In this way, it reconstructs the Centroid object from the bytes read from the input stream.

2.2 Execution and Control

This section focuses on two key classes, ***Application*** and ***KMeans***, that manage the orchestration of the program. Both of them are part of the package *it.unipi.dii.hadoop*.

2.2.1 KMeans

The ***KMeans*** class contains a collection of utility functions that support various aspects of the K-Means algorithm's execution and control.
The implemented methods are the following:

- *public static List<Centroid> generateInitialCentroids(Configuration conf, int clustersNumber, int pointsNumber, Path inputPath)*:

  Generates an **initial** set of **centroids** by randomly selecting data points from the input data.

- *public static void setCentroidsInConfiguration(List<Centroid> initialCentroids, Configuration conf)*:

  Stores the centroids passed as parameter in the **Hadoop Configuration**, allowing easy access during MapReduce iterations.

- *public static Job configureJob(int iteration, Configuration conf, int reducersNumber, Path inputPath, Path outputPath)*:

  Sets up a MapReduce **job** with a specific **configuration** for a given iteration. It defines mapper, combiner, reducer, and other job-related settings.

- *public static double computeCentroidsShift(List<Centroid> computedCentroids, Configuration conf)*:

  Calculates the **shift** (change) in **centroids** between iterations, a crucial measure for determining **convergence**.

- *public static List<Centroid> readComputedCentroids(Configuration conf, Path outputPath)*:

  Reads the **computed centroids** from output files and returns them as a **list**.

- *public static List<Centroid> readCentroidsInConfiguration(Configuration conf)*:

  Reads and retrieves **centroids** stored in the **Hadoop Configuration** and returns them as a **list**.

- *public static List<Double> splitInCoordinates(String text)*:

  Splits a string representation of **coordinates** into a **list of doubles**, making it easier to handle coordinate extraction.

- *public static double computeEuclideanDistance(Point point, Point centroid)*:

  Calculates the **Euclidean distance** between two data points, a vital step in determining **similarity**.

- *public static void clearOutputPath(Configuration conf, Path outputPath)*:

  Ensures that the specified **output path** in **HDFS** (Hadoop Distributed FileSystem) is cleared, allowing it to be used in the next **iteration**.

- *public static boolean isConverged(double shift, int currentIteration, Float threshold, int maxIterations)*:

  Checks if the **convergence criterion** for the K-Means algorithm is satisfied, allowing the program to halt based on a **threshold** (on the centroid shift between iterations) or **maximum iteration** count.

- *public static void addLogInfo(int currentIteration, double centroidShift, List<Centroid> computedCentroids, boolean isIteration, double executionTime)*:

  Appends information about each iteration (shift values, final centroids, timestamps, and execution times) to a **log file**.

2.2.2 Application

The ***Application*** class serves as the **entry point** for our K-Means clustering application, overseeing the **execution** and **configuration** of the program.
The only method implemented in this case is the *main* method:

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

  It starts by **validating** the command-line **arguments** to ensure the correct usage format and then loads **configuration settings** from an **XML** file. After initializing various program parameters and checking for correct iteration settings, it generates **initial centroids**, sets them in the Hadoop Configuration, and begins a loop for multiple MapReduce **iterations**. In each iteration, it **configures** and executes a MapReduce **job**, reads **computed centroids**, computes the centroids' **shift**, and logs relevant information. The program continues iterating until it either **converges** or reaches the **maximum** number of **iterations**.

2.3 MapReduce

This section discusses the behavior of the MapReduce components in the application. It comprises three classes: ***KMeansMapper***, ***KMeansCombiner***, and ***KMeansReducer***, all of which belong to the *it.unipi.dii.hadoop.mapreduce* package.

2.3.1 KMeansMapper

KMeansMapper extends the MapReduce **Mapper** class, so it processes input data and produces key-value pairs in output.
The class has the following **property**:

- *private List<Centroid> centroids*:

  A list of centroids that is initialized during the *setup* and used in the *map* method.

Its key **functionalities** are implemented through the following **methods**:

- *public void setup(Context context)*:

  Initializes the local list of **centroids** with the ones saved in the Hadoop configuration, which will be used in the *map* function. This is done with the *readCentroidsInConfiguration* function of the class ***KMeans***.

- *public void map(Object key, Text value, Context context)*:

  Processes input **key-value pairs**, where the key is an arbitrary *Object* (not used), and the value is of type *Text* and represents point **coordinates**.

  It performs the following steps:

  – Creates a ***Point*** object from the **text** received. The *splitInCoordinates* function is called to **parse** the input **text**, obtaining coordinates as a list of *Double* values. This facilitates the creation of the *Point* object, that can be built starting from that list.

- It iterates through the loaded centroids, calculating the **distance** between the point and each centroid using the *computeEuclideanDistance* function to determine the **closest centroid**.
- Emits a key-value pair containing the chosen centroid's ID and the Point object. So the output pairs are of type (*IntWritable*, *Point*).

2.3.2 KMeansCombiner

KMeansCombiner contains the **Combiner** implementation, so it allows partial **aggregation** of data produced by a **Mapper** before sending it to the **Reducer**.
Its key functionalities are implemented through the following method:

- *public void reduce(IntWritable centroidID, Iterable<Point> pointsList, Context context)*:

  Takes as input **key-value** pairs where the key is the *centroidID* (*IntWritable*), and the value is the **list of all points** (*Iterable<Point>*) that have been assigned to the cluster of the **centroid** with that id.

  Iterates over all points associated with a given centroid and calls the *sumCoordinates* function from the ***Point*** class to sum their coordinates.

  Emits pairs containing the *centroidID* along with the newly created *point*.

2.3.3 KMeansReducer

KMeansReducer extends the **Reducer** class, performing the **final aggregation** and calculating the new centroid position.
This is done through the *reduce* function, defined as follows:

- *public void reduce (IntWritable centroidID, Iterable<Point> partialSumPointsList, Context context)*:

  The **input types** are the same as for the **combiner**, but in this case, the reducer will receive **all points** associated with a given **centroid**, not just those produced by a single mapper.

  The method iterates over each of the **points** in the list and **sums** their **coordinates** by exploiting the *sumCoordinates* function, as in the Combiner. In addition, after doing so, it calculates the new centroid position by **averaging** the coordinates (considering the number of summed points, maintained by the *partialPointsCounter*) using the *averageCoordinates* function of the ***Point*** class.

  Finally, it **emits** a pair containing the *centroidID* and a ***Centroid*** object with the updated coordinates.

3 - Experimental Results

This chapter will present the **test design**, the **datasets** used and the techniques for their creation, and the **results** obtained after running the K-Means algorithm described in the previous chapters.
Finally, some **concluding notes** on the work and the quality of the results obtained will be provided.

3.1 Test Design

The primary **goal of the experimental phase** is to **assess the performance** of the developed algorithm in terms of both **result accuracy** and **efficiency**.
For this reason, it was determined to employ various datasets, each tailored to a specific objective.

Specifically, it was decided to produce **datasets having 2 and 3 dimensional points** to evaluate the results accuracy graphically (by producing 2D and 3D plots).

Meanwhile, for datasets containing data with a **higher number of dimensions**, a **silhouette analysis** was employed.
This technique assesses the similarity of each data point to its cluster in comparison to other clusters. **The analysis yields the silhouette score**, which is a value within the range of -1 to 1, a high silhouette score signifies that the data point is well assigned to its cluster, whereas a low silhouette score suggests that the data point might be better suited to another cluster.

Finally, another type of test involved employing **varying numbers of reducers** on the same dataset to evaluate performance from the execution time perspective.

The following table presents the values of **number of points (n)**, **number of dimensions (d)** and **number of clusters (k)** for each generated dataset.

Chosen Values for Datasets		
Points (n)	**Dimensions (d)**	**Clusters (k)**
1.000	2	3
1.000	3	3
10.000	2	5
10.000	3	3
50.000	2	7
50.000	3	5
100.000	5	7
100.000	7	10

3.2 Datasets Generation

An ad hoc Python program, called *dataset_ generator.py* was devised for dataset generation, employing the *make_ blobs* function provided by the *sklearn* library.

This specific function accepts various parameters, including the number of points to generate, the number of centers, and the size for each generated point. It also provides the option to set the standard deviation, enabling control over the variability and the dispersion of the points within each cluster.

The program in question employs the desired number of points to generate, their dimensions, and the total number of clusters to produce the dataset, saving it in a .txt file, and generating a graph if the selected dimension is 2 or 3.
An example of 2D and 3D graphs generated can be observed in the figure 3.1

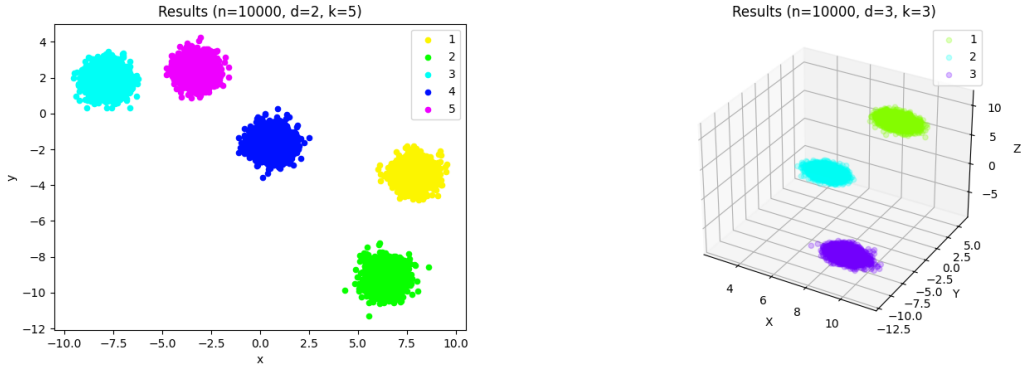

Figure 3.1: 2D and 3D dataset plots

3.3 K-means-mapreduce Results

3.3.1 Run the K-means-mapreduce Project

Shell scripts have been written to simplify the execution of the ***K-means-mapreduce*** Java program; the following are the various steps for executing the program on the virtual machines representing the Namenode and the two available Datanodes.

Once the Namenode, the two Datanodes, the YARN resource manager and the node managers have been started via the appropriate shell scripts provided by Hadoop (*start-dfs.sh* and *start-yarn.sh*) the following steps must be performed:

1. **Run the *load_config.sh* script** to load the *config.xml* file on the HDFS after appropriately editing it according to the parameters to be set (n, d, k, number of reducers, threshold and number of maximum iterations).

2. **Run the *load_jar.sh* script** to package the project into a JAR file and upload it to the virtual machine hosting the Hadoop cluster.

3. **Run the *run_mapreduce.sh* script**, providing the **parameters <n> <d> <k> <r>** according to the dataset to use (which should already be loaded on the HDFS) and the desired number of reducers. This script handles the tasks of executing the program and saving the results in a generated log file named *k-means-log.txt* directly in the results folder.

```
Iteration 1, Shift Value: 13.867711358746645
Iteration 2, Shift Value: 1.626342940079656
Iteration 3, Shift Value: 0.0

FINAL CENTROIDS:
0.7843 7.4937 7.1175
-6.7008 -5.8788 -9.1981
0.9659 4.7664 -3.6117

Timestamp: 2023-10-15 16:10:48.813

Execution Time: 60.68 s

Average Iteration Time: 20.226666666666667 s
```

Figure 3.2: k-means-log.txt File Example

3.3.2 Test Results

As previously mentioned, several tests were conducted on different datasets in alignment with the study's objectives.
For each test performed, a random set of initial centroids was generated by taking different seeds each. This approach ensured both the creation of different sets for each test and the reproducibility of the experiments.
The results for each type of study are presented below.

Tests with 2 and 3 Dimensions - Results Accuracy

Test 1 : $n = 1.000,\ d = 2,\ k = 3$

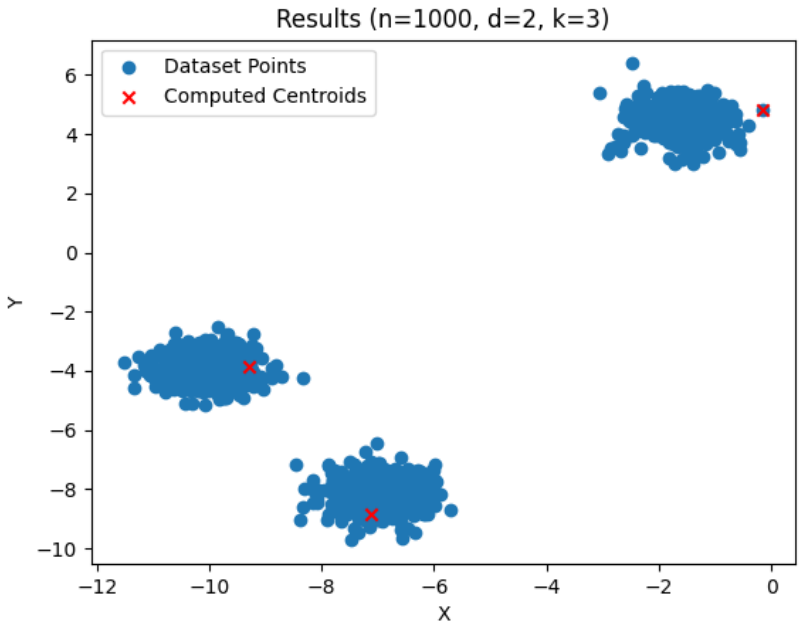

Figure 3.3: Test 1 Results Plot

Test 2 : $n = 1.000,\ d = 3,\ k = 3$

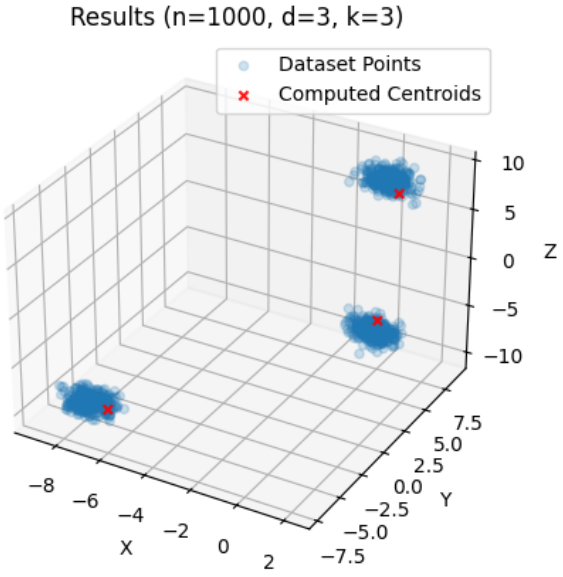

Figure 3.4: Test 2 Results Plot

Test 3 : $n = 10.000,\ d = 2,\ k = 5$

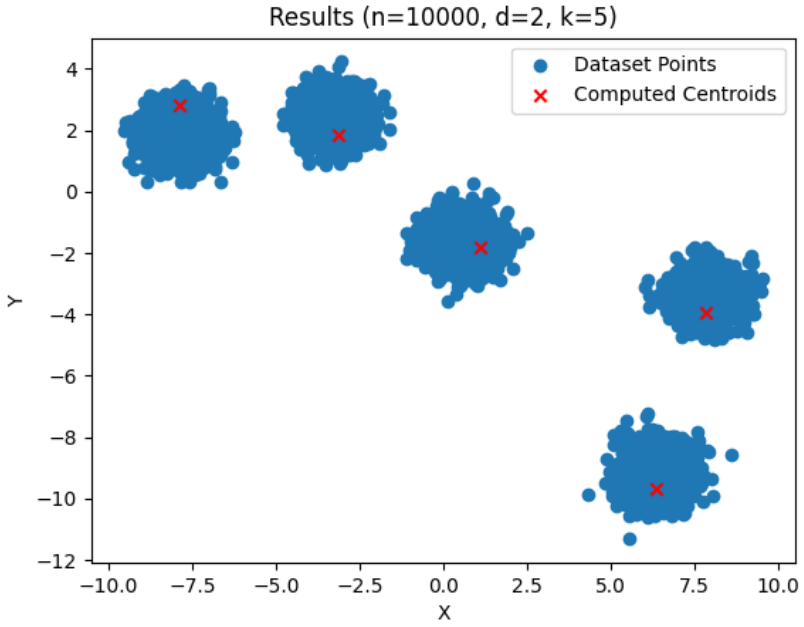

Figure 3.5: Test 3 Results Plot

Test 4 : $n = 10.000,\ d = 3,\ k = 3$

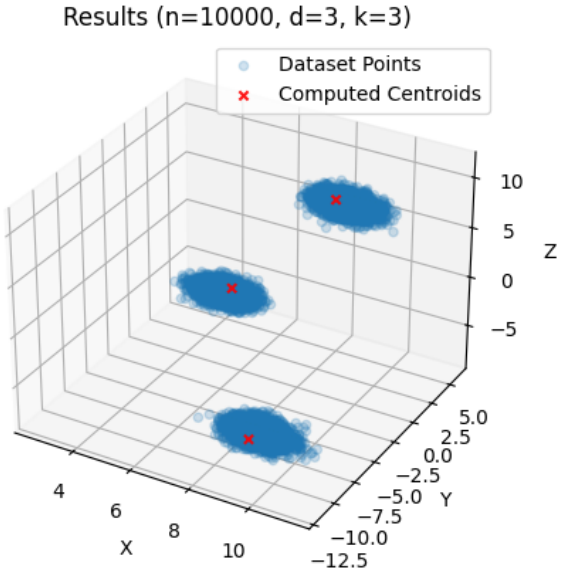

Figure 3.6: Test 4 Results Plot

Test 5 : $n = 50.000,\ d = 2,\ k = 7$

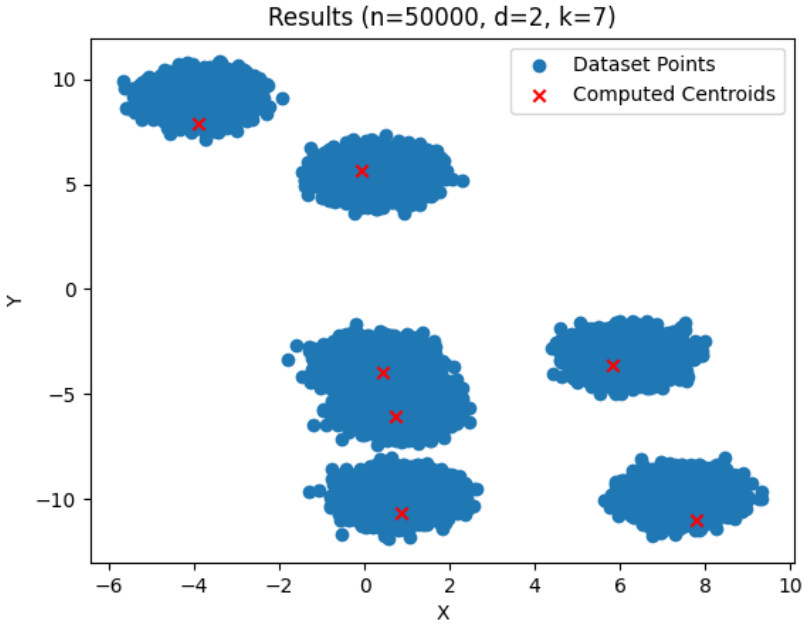

Figure 3.7: Test 5 Results Plot

Test 6 : $n = 50.000,\ d = 3,\ k = 5$

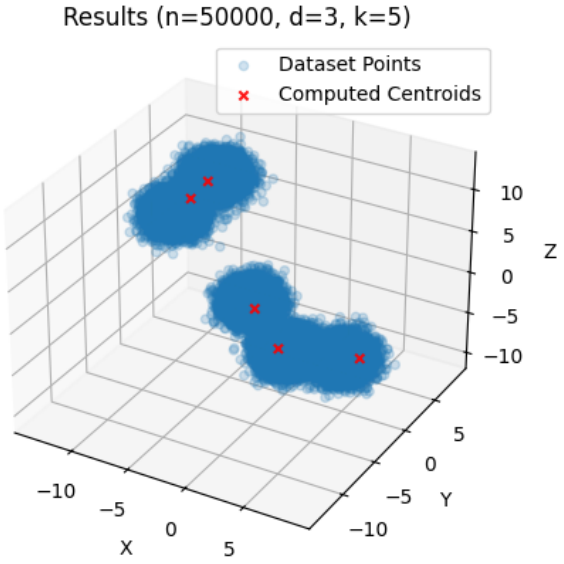

Figure 3.8: Test 6 Results Plot

Silhouette Score Results		
Test	**Dataset parameters**	**Silhouette Score**
Test 1	$n = 1.000,\ d = 2,\ k = 3$	**0.8589146548001182**
Test 2	$n = 1.000,\ d = 3,\ k = 3$	**0.9140977219549759**
Test 3	$n = 10.000,\ d = 2,\ k = 5$	**0.8322751838395809**
Test 4	$n = 10.000,\ d = 3,\ k = 3$	**0.926821500184782**
Test 5	$n = 50.000,\ d = 2,\ k = 7$	**0.7538788002387206**
Test 6	$n = 50.000,\ d = 3,\ k = 5$	**0.7084571660799758**

As can be seen from both the graphs produced and the silhouette scores obtained, the results can be considered very satisfactory, as the list of final centroids seems to be well assigned to the clusters used for the different tests, and the silhouette scores assigned have quite high values (a silhouette score is considered good when it exceeds the 0.5 threshold).

Tests with Higher Number of Dimensions - Silhouette Score

Test 7 : $\boldsymbol{n = 100.000,\ d = 5,\ k = 7}$

Test 8 : $\boldsymbol{n = 100.000,\ d = 7,\ k = 10}$

Silhouette Score Results		
Test	**Dataset parameters**	**Silhouette Score**
Test 7	$n = 100.000,\ d = 5,\ k = 7$	**0.6974096802446764**
Test 8	$n = 100.000,\ d = 7,\ k = 10$	**0.7528198056912023**

In this scenario, with an increase in both the number of dimensions (5 and 7) and the number of clusters (7 and 10) the silhouette score obtained, as might be expected, is lower but still acceptable.

Tests employing More Reducers - Execution Time

Regarding the use of multiple reducers, the Hadoop documentation recommends selecting a **number of reducers** that is **0.95** (all of the reduces can launch immediately and start transferring map outputs as the maps finish) **or 1.75** (the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing) **multiplied by the product between the number of nodes and the number of maximum containers per node**.

However, the results achieved by varying the number of reducers can **also be influenced by other factors**, such as the available hardware and its configuration, network bandwidth and dataset characteristics.
The increase of the reducers number has a **particular impact on load balancing and resource utilization** and so, **increasing the number of reduces increases load balancing** and **lowers the cost of failures**, but **increases the framework overhead**.

In our specific case, where we have one Namenode and two Datanodes it was decided to test the program using a number of reducers ranging from 1 up to 6.

Test 9 : $n = 100.000,\ d = 7,\ k = 10,\ r = [1, 6]$

Execution Time Results		
Reducers	**Execution Time**	**Average Iteration Time**
1	**437.129 s**	**21.85645 s**
2	**429.901 s**	**21.49505 s**
3	**438.27 s**	**21.9135 s**
4	**479.578 s**	**23.9789 s**
5	**489.197 s**	**24.45985 s**
6	**488.851 s**	**24.44255 s**

Examining the results table, it is evident that the execution times are quite similar across the range. This suggests that the difference between the execution times achieved may not be related to the parallel execution of multiple reducers, but rather appears to be linked to the transmission of data across the network.

Indeed, in our case, the use of many reducers leads to excessive processing of small files generated as output, potentially causing a performance overhead for the application without gaining the benefits in terms of load balancing and cost of failures.

3.4 Conclusions

The design and implementation through Hadoop of the K-Means algorithm following the Map Reduce paradigm led to satisfactory results.
Specifically, it was observed that the program effectively met its goals by using datasets of different quantities, point dimensions and clusters number, delivering accurate results and maintaining relatively consistent execution times across different scenarios.

During the testing phase, it was also noted that several aspects could be enhanced. For instance, instead of solely relying on the first random set of initial centroids, it might be helpful to explore different approaches and assess how the results can be enhanced even more.
Another potential enhancement concerns the use of reducers. Testing datasets with considerably larger points dimensions and numbers of clusters could help determine if increasing the number of reducers would yield the advantages discussed earlier.