UNIVERSITÀ DI PISA

MSc in Computer Engineering

Cloud Computing

K-Means clustering algorithm using Hadoop

TEAM MEMBERS:

Fabrizio Lanzillo
Federico Montini
Lorenzo Mirabella

https://github.com/FedericoMontini98/Hadoop_K-means

Academic Year 2022-2023

Contents

Chapter 1

Algorithms Definition

1.1 Clustering - K-Means algorithm

Clustering is a method of assigning comparable data points to groups using data patterns. It is used to find groups that have not been explicitly labeled in the data and to find patterns and make better decisions. Its goal is to group unlabeled data so that the data objects whose characteristics and attributes are similar are together in a cluster.
One popular clustering algorithm is **K-means**. The K-means clustering algorithm divides a set of n observations into k clusters.

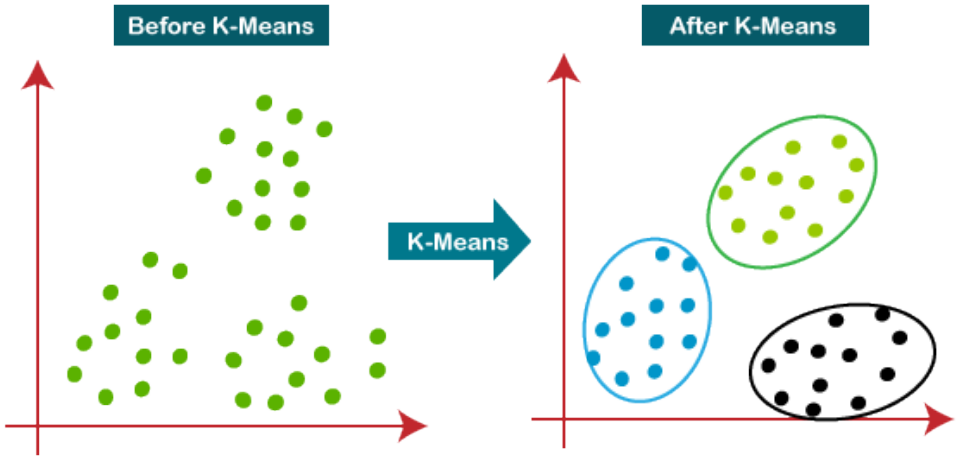

Figure 1.1: K-Means clustering algorithm in action

K-means clustering has several advantages. It is relatively simple to implement and scales to large data sets. It guarantees convergence and can warm-start the positions of centroids. It easily adapts to new examples and generalizes to clusters of different shapes and sizes. The simplicity of k-means makes it easy to explain the results.
However, there are also some disadvantages to using K-means clustering:

1. Choosing k manually can be difficult

2. The algorithm is dependent on the initial value of k: you can mitigate this dependence by running k-means several times and taking the best results
3. As k increases, you need advanced versions of k-means to pick better values of the initial centroids, i.e. K-Means++
4. Troubles clustering data where clusters are of varying sizes and density: to cluster such data, you need to generalize k-means
5. Centroids can be dragged by outliers, or outliers might get their own cluster

1.2 K-Means Parallelization

Hadoop is an open-source framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. By breaking the data into smaller chunks, Hadoop can process the data in parallel across multiple nodes, which can significantly reduce processing time.
K-means can be parallelized using Hadoop following a Map-Reduce paradigm. The idea is to perform the following steps:

1. Map each point assigning them to a specific centroid (the nearest)
2. Recalculate centroid coordinates based on the assigned points

The idea is to divide the dataset into pieces allowing each machine to work on a subset of it, thus reducing computation time, and merge the obtained results. In the following sections and chapters the pseudo-code and the implementation of this project will be analyzed.

1.3 Mapper

In order to deploy a parallelized version of the K-means algorithm, a map function has been implemented. The aim of the map function is to associate each point in the starting dataset with the nearest centroid from a list of centroids.
To accomplish this task, the map function iterates over the list of centroids for each point, calculating the distance between the point and each centroid. If this distance is lower than the one found in previous iterations, the current centroid becomes the new candidate for the nearest centroid.
After checking the entire list of centroids, the pair composed of the nearest centroid ID and the point instance is returned.

To load the initial set of centroids, the setup function has been redefined. Assuming that the initial set has been loaded into the configuration file before the job is submitted, we could implement a 'ReadCentroidsFromConfiguration' function that unloads centroids from the configuration file, instantiates a 'Centroid' instance for each centroid, and loads them into a specific parameter of the class, such as an ArrayList of centroids.
This function has been implemented externally since it could be reused to extract new centroids after the current iteration to calculate the shift of centroids.
The pseudo-code can be seen below:

Algorithm 1 Map Function - Mapper

```
1: procedure MAP(key, point)
2:     centroid_id ← null
3:     distanceFromCentroid ← Double.MAX_VALUE
4:     for all centroid in centroids do
5:         distance ← DISTANCE(centroid, point)
6:         if centroid_id = null or distance < distanceFromCentroid then
7:             centroid_id ← centroid.Centroid_id
8:             distanceFromCentroid ← distance
9:     Emit(centroid_id, point)
```

Algorithm 2 Setup Function - Mapper

```
1: procedure SETUP(configuration)
2:     centroids ← READCENTROIDSFROMCONFIGURATION(configuration)
```

1.4 Reducer

The output of the Mapper is given as input for the Reducer, which processes and produces a new set of output that will be stored in the HDFS.
This method calculates the new centroid points by averaging the partial sums of the assigned points. It takes as input the ID of the centroid, an iterable collection of partial sums of points assigned to the centroid, and a context object for accessing Hadoop services. The method iterates over the partial sums and adds every coordinate of the points. Then it calculates the average of the coordinates to obtain the new centroid point. Finally, it emits the centroid ID and the string representation of the new centroid point using the context object.
The pseudo-code can be seen below:

Algorithm 3 Reduce function - Reducer

```
1: procedure REDUCE(centroidId, partialSums)
2:     finalSum ← sum(partialSums)
3:     nextCentroidPoint ← finalSum.average()
4:     Emit(centroidId, nextCentroidPoint)
```

Since there is no need to initialize data or release resources, clean-up and setup functions are not required.

1.5 Combiner

A combiner has also been implemented. The primary job of Combiner is to process the output data from the Mapper, before passing it to Reducer and is used to minimize the data that got shuffled between Map and Reduce.
Reduce function takes in three arguments: *centroidId*, *points*, and *context*. The function calculates the partial sum of the coordinates of the points associated with a specific centroid. It does this by iterating over the points and calling the add method on the partialSum object. Finally, it writes the centroidId and the calculated partialSum to the context.
Performing this local aggregation before shuffling the data to the reducers, the amount of data that needs to be transferred between the mappers and reducers is reduced, thus improving performance and reducing network congestion among compute nodes, especially as the amount of data increases.
The pseudo-code can be seen below:

Algorithm 4 Reduce function - Combiner

```
1: function REDUCE(centroidId, points)
2:     partialSum ← sum(points)
3:     Emit(centroidId, partialSum)
```

Since there is no need to initialize data or release resources, clean-up and setup functions are not required.

Chapter 2

Implementation

In our implementation of the K-means algorithm using the Hadoop MapReduce, we have utilized different classes such as:

- **KMeans**
- **KMeansUtil**
- **Point**
- **Centroid**
- **KMeansMapper** (already explained in the previous chapter)
- **KMeansCombiner** (already explained in the previous chapter)
- **KMeansReducer** (already explained in the previous chapter)

2.1 KMeans

The **KMeans class** encapsulates the logic for running the K-means algorithm using the Hadoop MapReduce and provides an entry point for the program execution. It consists of two methods: **main** and **KMeansIterations**.

- **main(String[] args)**

  This is the main entry point for the K-means program.
  It expects arguments from the command line in the following format:
  <inputPath> <outputPath> <centroidPath> [<numReducers>].
  The method first checks if the number of arguments is valid and then parses the arguments, including the *inputPath*, *outputPath*, *centroidPath*, and optional *numReducers*.
  A **Hadoop configuration** is created, and the **initial centroids** are read from the specified centroidPath using the readCentroids method from the KMeansUtil class and they **are set in the configuration**.
  Finally, the KMeansIterations method is called, which executes the iterations of the K-means algorithm.

- **KMeansIterations(Configuration conf, Path outputPath, Path inputPath, int numReducers)**
  This method performs iterations of the K-means algorithm taking as input the *configuration file*, the *input path*, the *number of reducers*, and the *output path*.
  It first **obtains the hadoop configuration** and initializes the file system. Then, it configures and submits the MapReduce job using the *configureJob* method from the KMeansUtil class, and waits for its completion.
  When the job has concluded, it **reads the current and previous centroids**, either from multiple files or a single file, depending on the number of reducers and calculates the centroid shift, using the *calculateCentroidShift* method from the KMeansUtil class.
  If the **shift is below a configurable threshold threshold**, the **algorithm** is considered **converged** and the iteration is stopped. Otherwise the centroids are updated and the iteration continues until convergence or the maximum number of iterations is reached.
  At the end, it logs the iteration information using the *logIterationInfo* method from the KMeansUtil class.

2.2 KMeansUtil

This is an **utility class** called **KMeansUtil**, which offers different helper methods for the K-means algorithm.
These are the methods provided by this class:

- **generateCentroidFromArray(String[] fields)**
  This method **generates a Centroid object**. In order to do so it extracts the centroid ID from the first element of the array passed as input and creates a Point object using the remaining elements.
  The Centroid object is then returned.
- **readCentroidFromMultipleFiles(Configuration conf, Path dirPath)**
  This method **reads centroids from multiple files** in a directory and is called **when we have more than one reducer**.
  In order to generate the Centroid objects the generateCentroidFromArray method is used. These objects are stored in an ArrayList and returned.
- **readCentroids(String pathString, Configuration conf, boolean csvReading)**
  This method **reads the centroids from a file**.
  It takes the path to the file, the Hadoop configuration, and a flag indicating whether the file is in CSV format as input.
  In order to generate the Centroid objects the generateCentroidFromArray method is used. These objects are stored in an ArrayList and returned.
- **configureJob(Configuration conf, Path inputPath, Path outputPath, int numReducers, int iteration)**
  This method **configures a MapReduce job** for a K-means iteration. It takes the Hadoop configuration, the input and output paths, the number of reducers, and the current iteration number as input.
  It creates a new job instance, and return the configured job.

- **readCentroidsFromConfiguration(Configuration conf)**
  This method **reads centroids from the Hadoop configuration**.
  It retrieves the string representations of the centroids from the configuration, converts them into lists of doubles, and creates Centroid objects.
  The resulting centroids are stored in an ArrayList and returned.

- **calculateCentroidShift(ArrayList currentCentroids, Configuration conf)**
  This method **calculates the shift of centroids between the current and previous iterations**.
  It reads the previous centroids from the configuration, iterates over the centroids, compares their coordinates, and **calculates the shift for each dimension**. The total shift of all centroids is then returned. This is the formula used to calculate the shift:

$$\text{shift} = \sum_{i=1}^{n} \sum_{j=1}^{m} |x_{ij} - y_{ij}|$$

  n is the number of centroids.
  m is the number of dimensions for each centroid.
  x_{ij} represents the j-th coordinate of the i-th centroid in the previous iteration.
  y_{ij} represents the j-th coordinate of the i-th centroid in the current iteration.

- **setCentroidsToConf(String key, ArrayList centroids, Configuration conf)**
  This method **sets the centroids in the Hadoop configuration**.
  It converts the centroids into an array of strings, representing their points and sets them as the value of the configuration property specified by the key.

- **logIterationInfo(int iteration, double shift, int numReducers)**
  This method **logs iteration information** to a log file.

2.3 Point

The Point class represents a point or a sum of points in the KMeans algorithm and provides various operations for manipulating and calculating distances between points.
These are the methods provided by this class:

- **Point(ArrayList coordinates)**
  This constructor **creates a new Point object with the specified coordinates**.
  It initializes the coordinates field with the provided coordinates and sets the instances field to 1.

- **Point(String text)**
  This constructor **creates a new Point object from a comma-separated string representation** of the coordinates (text).
  It splits the string by commas, converts each substring to a double value, and adds them to the coordinates field of the Point object.

- **getDistance(Point point)**
  This method **calculates and return the Euclidean distance** between the current

Point object and the specified point.
This is the formula used to calculate the Euclidean distance:

$$\text{distance} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

n is the number of coordinates for each point.
x_i and y_i represent the corresponding coordinates of the points to be compared.

- **add(Point point)**
  This method adds **the coordinates of the specified point** to the current Point object.
- **average()**
  This method **calculates the average of the Points' sum** by dividing each coordinate by the number of instances (instances field).
- **getCoordinates()**
  This method returns the coordinates field of the Point object.
- **getInstances()**
  This method returns the instances field of the Point object.
- **write(DataOutput dataOutput)**
  This method writes the Point object to the specified dataOutput stream.
- **readFields(DataInput dataInput)**
  This method reads the Point object from the specified dataInput stream.
- **toString()**
  This method returns a **string representation of the Point object**.
  It converts each coordinate to a string and concatenates them, **separated by spaces**, to form the string representation.

2.4 Centroid

The Centroid class represents a centroid in the KMeans algorithm.
Each centroid is associated with a centroid ID and a Point object representing its coordinates.

- **Centroid(int centroid_id, ArrayList coords)**
  This constructor **initializes a Centroid** object with the given **centroid ID and coordinates**.
  It takes an integer value centroid_id and an ArrayList of Double values coords.
  The centroid ID is wrapped in an IntWritable object, and the coordinates are used in order to create a Point object representing the centroid's coordinates.
- **write(DataOutput dataOutput)**
  This method writes the Centroid object to a DataOutput stream.
- **compareTo(Centroid other)**
  This method compares the Centroid object with another Centroid object for sorting purposes.

- **readFields(DataInput dataInput)**
  This method reads the Centroid object from a DataInput stream.
- **setPoint(Point point)**
  This method sets the Point object representing the centroid.
- **getPoint()**
  This method provides the Point object representing the centroid.
- **getCentroid_id()**
  This method provides the centroid ID of the centroid.

Chapter 3

Tests

This section presents the comprehensive results obtained from implementing the K-Means algorithm using Hadoop MapReduce. The algorithm was evaluated on seven diverse datasets, with variations in the number of points (n), number of centroids (k), and dimensionality of points (d). The objective was to analyze the algorithm's performance and accuracy under different conditions.

Dataset generation A python script was designed to generate a dataset using the ‘make_blobs‘ module from ‘scikit-learn‘. The generated dataset will consist of a specified number of samples, represented as points in a specified dimensional space. The generated points are then normalized so that each dimension ranges from 0 to 1 using the ‘apply‘ function of pandas, this is to ensure that the variables of the dataset have the same scale and to provide more meaningful shift values.

Test 1

- Dimension (d): 2
- Number of Points (n): 200
- Number of Centroids (k): 2

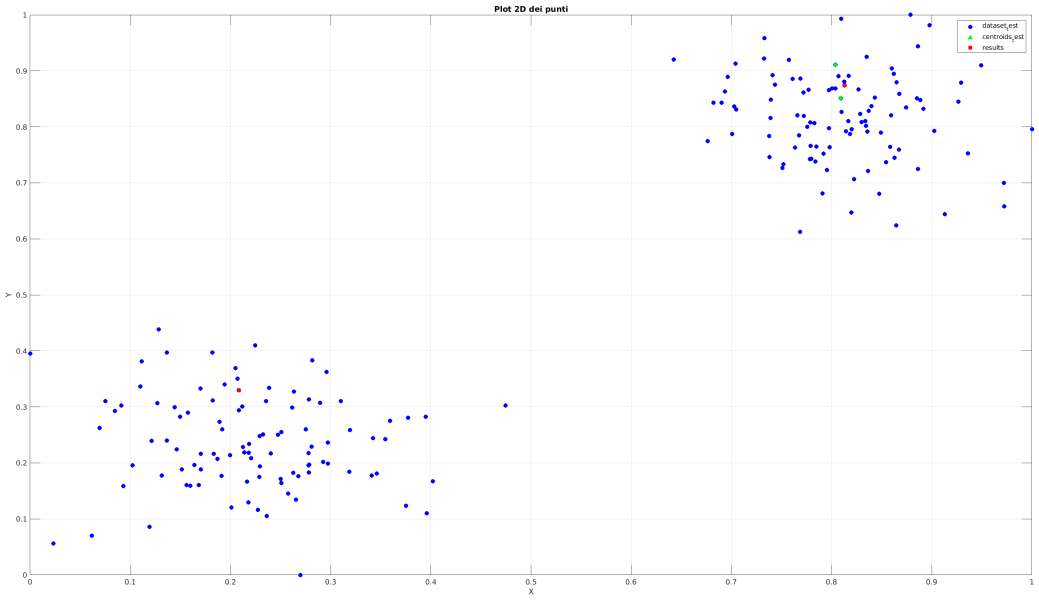

Figure 3.1: Test 1

Test 2

- Dimension (d): 3
- Number of Points (n): 500
- Number of Centroids (k): 2

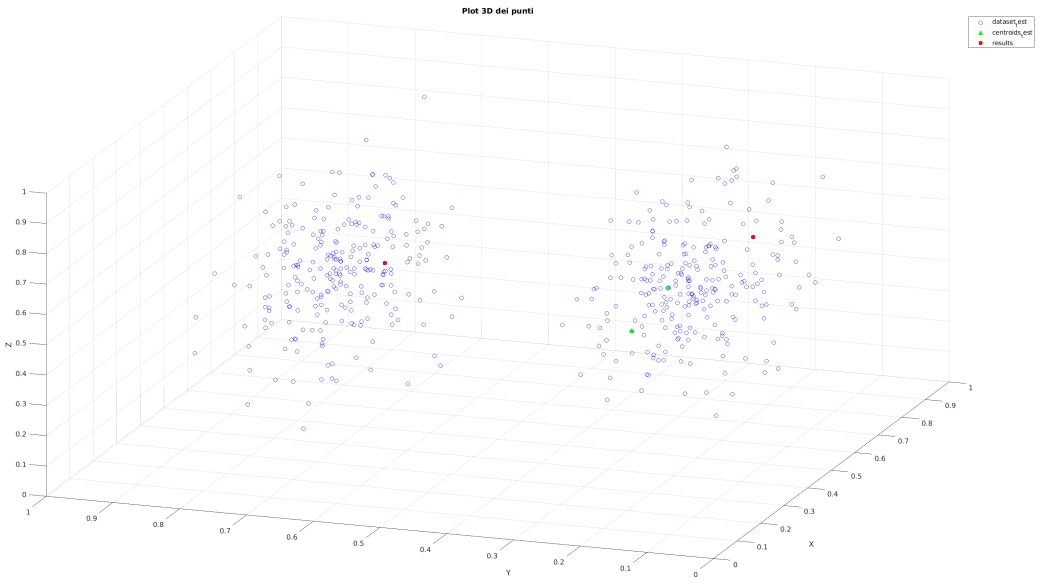

Figure 3.2: Test 2

Test 3

- Dimension (d): 3
- Number of Points (n): 2500
- Number of Centroids (k): 3

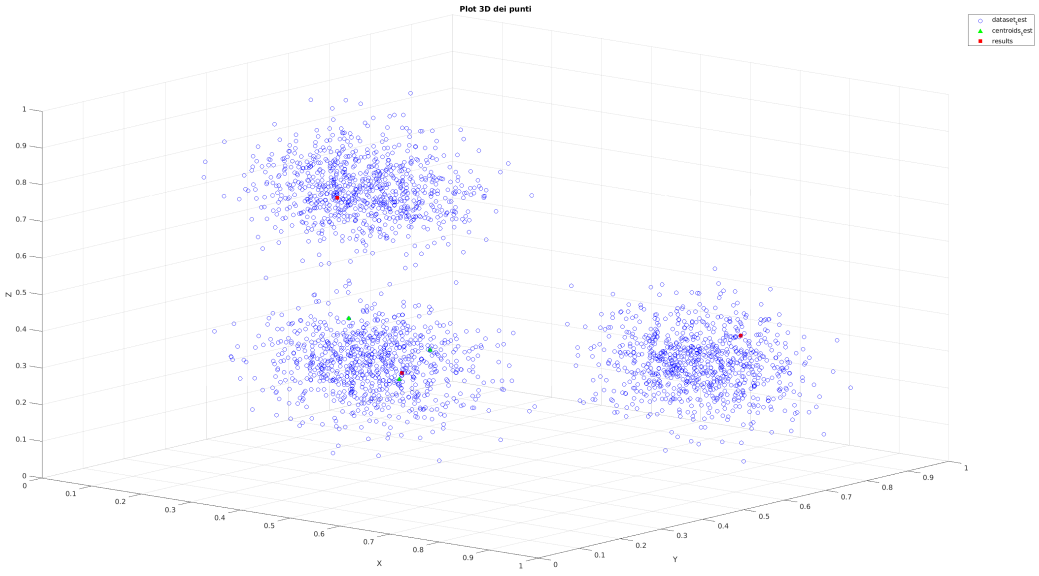

Figure 3.3: Test 3

Test 4

- Dimension (d): 3
- Number of Points (n): 5000
- Number of Centroids (k): 4

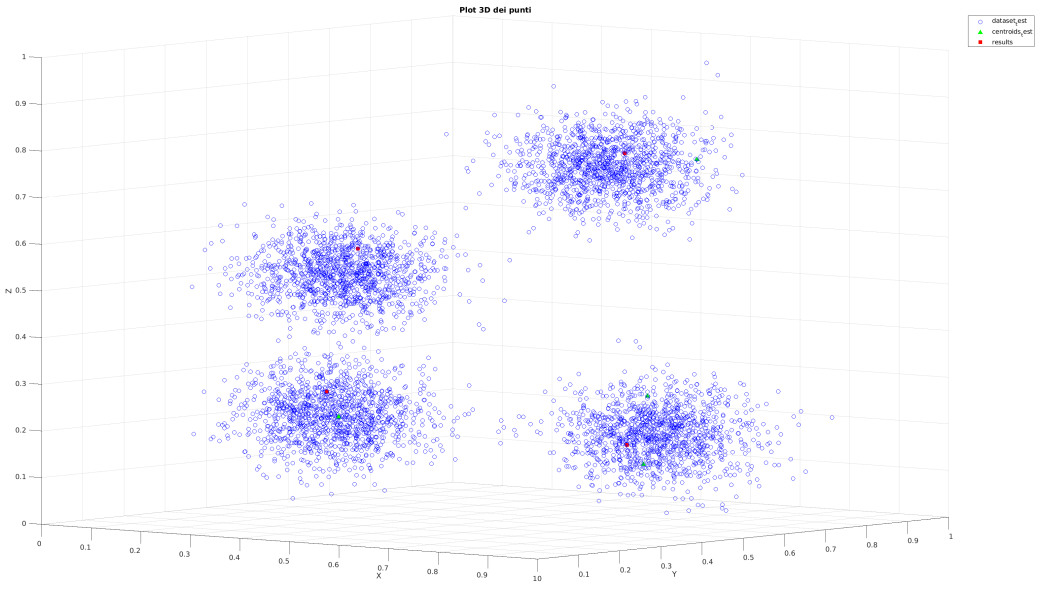

Figure 3.4: Test 4

Test 5

- Dimension (d): 3
- Number of Points (n): 10000
- Number of Centroids (k): 5

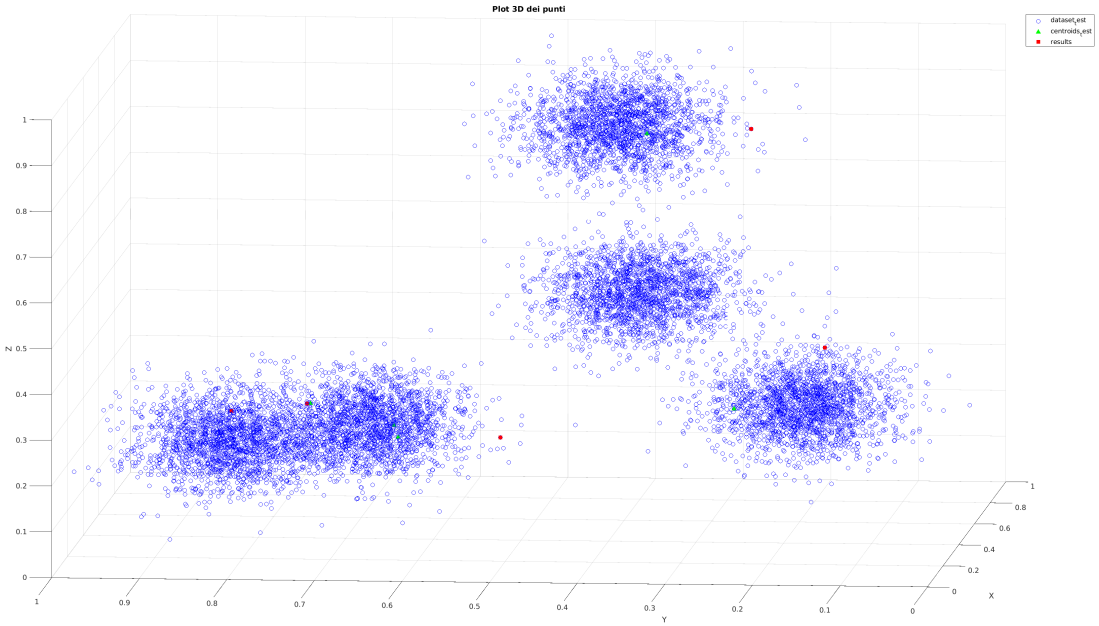

Figure 3.5: Test 5

Test 6

- Dimension (d): 4
- Number of Points (n): 10000
- Number of Centroids (k): 2

For datasets with dimensionality up to 3 (Test 1 to Test 5), the evaluation was performed using a plot generated with MATLAB. For datasets with higher dimensionality, the Silhouette score was calculated. The Silhouette score is a number between -1 and 1 which measures the quality of clustering by computing the average distance between each point and points within its cluster, relative to points in other clusters. A higher Silhouette score indicates better-defined clusters. In general, a score greater than 0.5 is considered a good result.

The Silhouette score for test 6 is: **0.5349359789769187**.

Test 7

- Dimension (d): 15
- Number of Points (n): 50000
- Number of Centroids (k): 10

The Silhouette score for test 7 is: **0.4575820224967407**.

3.1 KMeans++

The initial centroid selection affects significantly the algorithm accuracy. For example, the results in test 5 and 7 aren't good. To address this problem, two approaches were compared: random initialization and k-means++ initialization. K-means++ works by choosing the first centroid randomly from the data points, and then iteratively selecting the next centroids based on a probability distribution. The probability of a data point being chosen as the next centroid is proportional to its distance squared to the nearest existing centroid. This means that points that are further away from existing centroids have a higher chance of being chosen as the next centroid. This process is repeated until all the centroids have been chosen. The idea behind this method is to spread out the initial centroids so that they are not too close to each other, which can help speed up convergence of the k-means algorithm. In figure 3.6 it's possible to see the better-defined clusters.

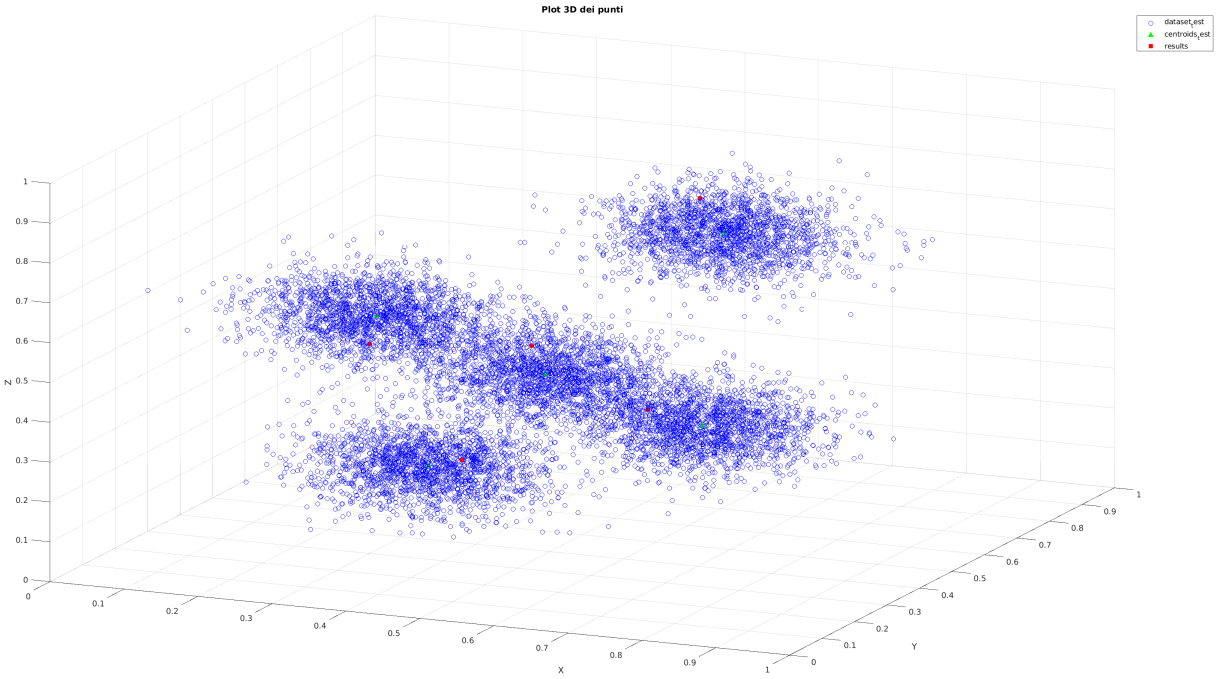

Figure 3.6: Test 5 - KMeans++

In test 7, incredible results were obtained: the number of convergence iterations decreased from the maximum of 20 to just 2, and the new Silhouette score is **0.7469265908**.

3.2 Number of Reducers

The impact of the number of reducers on algorithm execution time was examined. Hadoop's documentation suggests selecting the number of reducers by multiplying the number of nodes by 0.95 or 1.75 and in our case we have one namenode and two datanodes. The average execution time for one iteration of Test 6, using different numbers of reducers, was as follows:

The results obtained for different numbers of reducers can be influenced by several factors, including the hardware configuration, network bandwidth, and the characteristics of the dataset. When running MapReduce jobs, the number of reducers affects load balancing and

Table 3.1: Evaluation of Reducer Configuration

Number of Reducers	Execution Time (ms)
1	18460.0
2	17268.0
3	22352.0
4	20321.0
5	46593.0

resource utilization. In the scenario with one reducer, the workload is concentrated on a single node. This can result in longer execution times, as the processing power of one node may not be fully utilized, and data transfers may be limited. With multiple reducers (two, three, or four), the workload is divided across the nodes, allowing for parallel processing. This can lead to better load balancing and improved resource utilization, resulting in reduced execution times. However, in the case of five reducers, the workload may not be distributed optimally. The limited number of nodes (two Datanodes) compared to the number of reducers can lead to increased data transfers and potential congestion, causing longer execution times.

3.3 Conclusion

The implementation of the K-Means algorithm using Hadoop MapReduce provided valuable insights into its performance and behavior. Key conclusions from the results are as follows:

1. The accuracy of the K-Means algorithm is significantly influenced by the selection of initial centroids, highlighting the importance of employing appropriate initialization techniques. The k-means++ initialization method generally yielded better results in terms of silhouette score and required fewer iterations compared to random initialization.
2. Datasets with higher dimensionality exhibited lower silhouette scores, indicating increased complexity and the challenge of clustering.
3. The number of reducers had a discrete impact, especially with a high value, on execution affecting load balancing and framework overhead.

Future Improvements In order to enhance the scalability and performance of the K-Means algorithm implemented using Hadoop MapReduce, a potential future improvement could be the utilization of Hadoop's caching mechanism. Currently, the algorithm passes centroid information to each iteration through the MapReduce configuration, which can become a scalability bottleneck as the number of iterations and the size of the centroid information increase.

By leveraging Hadoop's caching capabilities, we can store and access the centroid information in a more efficient and scalable manner. Here, we discuss the potential benefits and effects this approach could have:

1. Improved Scalability: As the algorithm scales to larger datasets or higher-dimensional spaces, the size of the centroid information grows. Utilizing Hadoop's caching

mechanism ensures that the centroid data is efficiently distributed across the cluster's nodes, allowing for seamless scalability without overwhelming the network or storage resources.

2. Faster Convergence: With centroid information readily available in the cache, each iteration of the K-Means algorithm can access the data directly from the cache instead of retrieving it from the configuration. This reduces the time spent on data retrieval, resulting in faster convergence of the algorithm.

3. Enhanced Load Balancing: By caching the centroid information, the algorithm can leverage Hadoop's built-in load balancing capabilities. The caching mechanism distributes the centroid data across the cluster's nodes, ensuring that the computational workload is evenly distributed. This can lead to better resource utilization and improved overall performance.

It is important to note that incorporating Hadoop's caching mechanism into the K-Means algorithm requires careful consideration of data consistency and cache management strategies. However, by leveraging the caching capabilities, the algorithm can benefit from improved scalability, faster convergence and enhanced load balancing, ultimately leading to more efficient and scalable clustering on large-scale datasets.