UNIVERSITÀ DI PISA

Master of science in Artificial Intelligence and Data Engineering

Cloud Computing project

K-Means Algorithm for the Hadoop MapReduce Framework

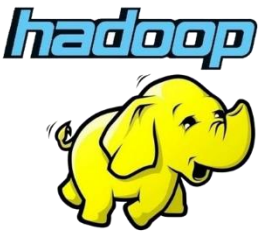
Giulio Capecchi

Federico Frati

Stefano Micheloni

Academic year 2022/2023

Index



1. Introduction

The aim of this project is to implement the K-means algorithm using the Hadoop framework, specifically by creating a MapReduce model. The K-means algorithm is a widely used clustering technique that tries to group a set of data points into a specified number of groups called clusters; it just needs as input the set of data points $X_i = \{x_1, x_2, \dots, x_n\}$ and k, the desired clusters number to individuate. This is done by selecting at first some random *centroids* and iteratively assigning each point to the closest one, then updating their positions based on the mean of the assigned data. This process continues until a stopping criterion is met, which was selected in this work as the movement of the centroids from one iteration to another below a predefined threshold. Additionally, a max iteration counter was utilized to avoid excessive execution times. By leveraging the Hadoop framework, which provides distributed processing capabilities, we aimed to implement algorithm in a scalable and efficient manner.

2. Solution Design

2.1 Pseudocode

Start by choosing K casual points from dataset. These will be the initial centroids and will be put into the Centroids array ("Centroids[]").

Algorithm 1: Map function

1: ***For each*** element $\boldsymbol{P}(\boldsymbol{x}, \boldsymbol{y}, \dots, \boldsymbol{z})$ ϵ Dataset ***do*:**

2: **For each** C_i belonging to *Centroids[]*, calculate the distance between P and C_i

3: Assign P to $Cluster_i$ by checking the nearest centroid C_i

4: *EMIT (i, P)*

Algorithm 2: Combiner function

Input: (*key*, *values*)

1: Create Point *agglomerate* and initialize its coordinates to *0*

2: *agglomerate.weight* <- *0*

3: ***For each*** point *p* in *values*:

4: *agglomerate* <- *agglomerate* + *p*

5: *agglomerate.weight* <- *agglomerate.weight* + *p.weight*

6: *EMIT (key, agglomerate)*

Algorithm 3: Reducer function

Input: (*key*, *values*)

1: Create Point *new_centroid* and initialize its components to *0*

2: *new_centroid.weight* <- *0*

3: **For each** point *p* in *values*:

4: *new_centroid* <- *new_centroid* + *p*

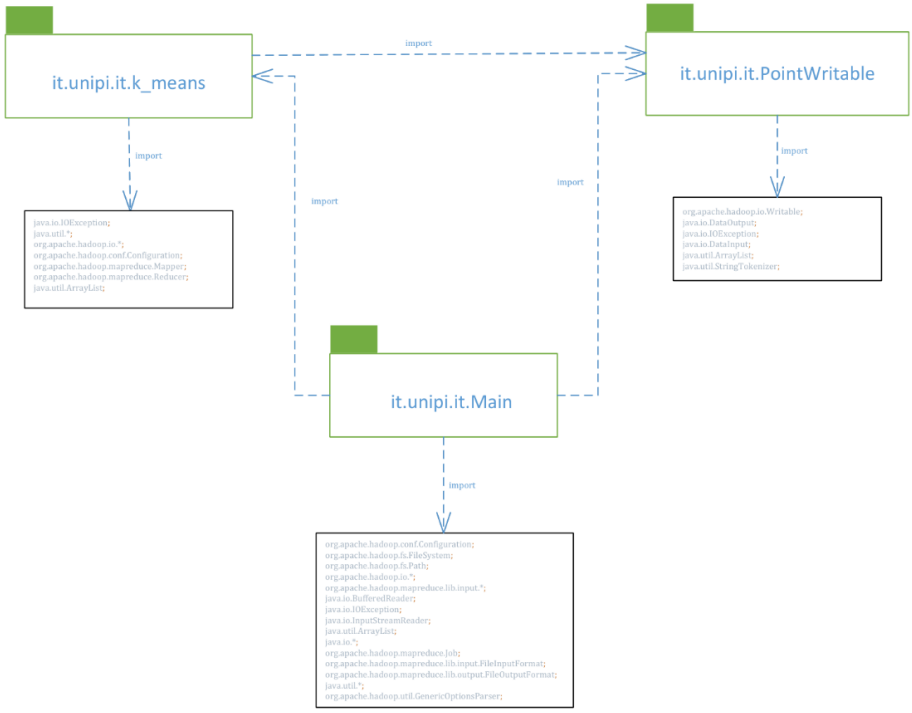
5: *new_centroid.weight* <- *new_centroid.weight* + *p.weight*

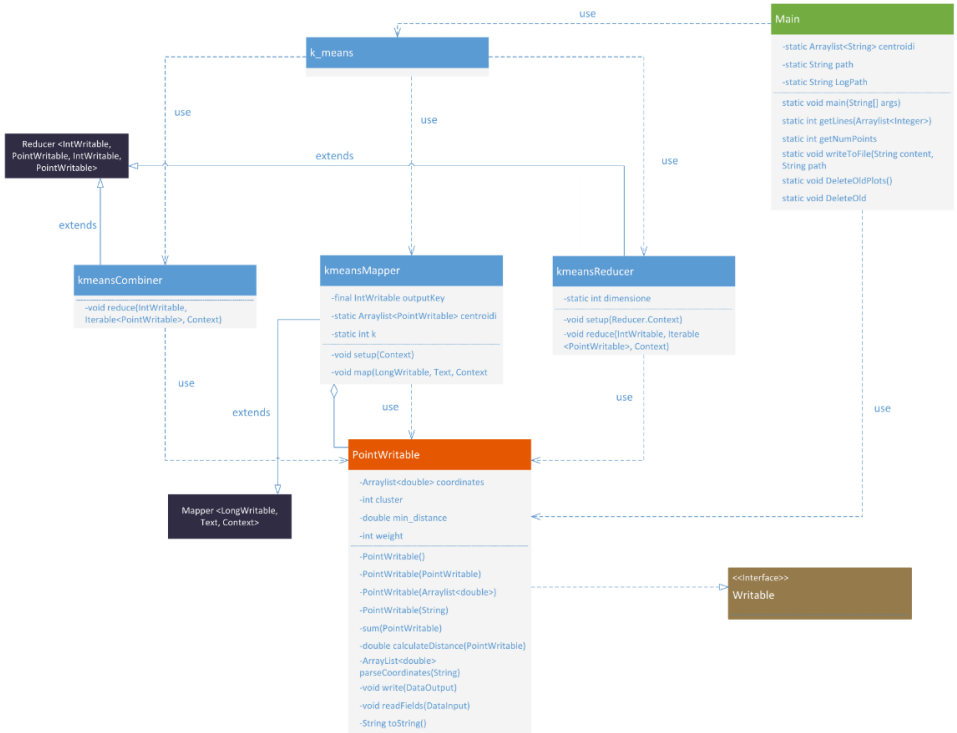
6: calculate new centroid <- *new_centroid* / *new_centroid.weight*

7: *EMIT (key, new centroid)*

2.2 UML diagrams

- *UML package diagram*





- *UML class diagram*

2.3 Flowchart

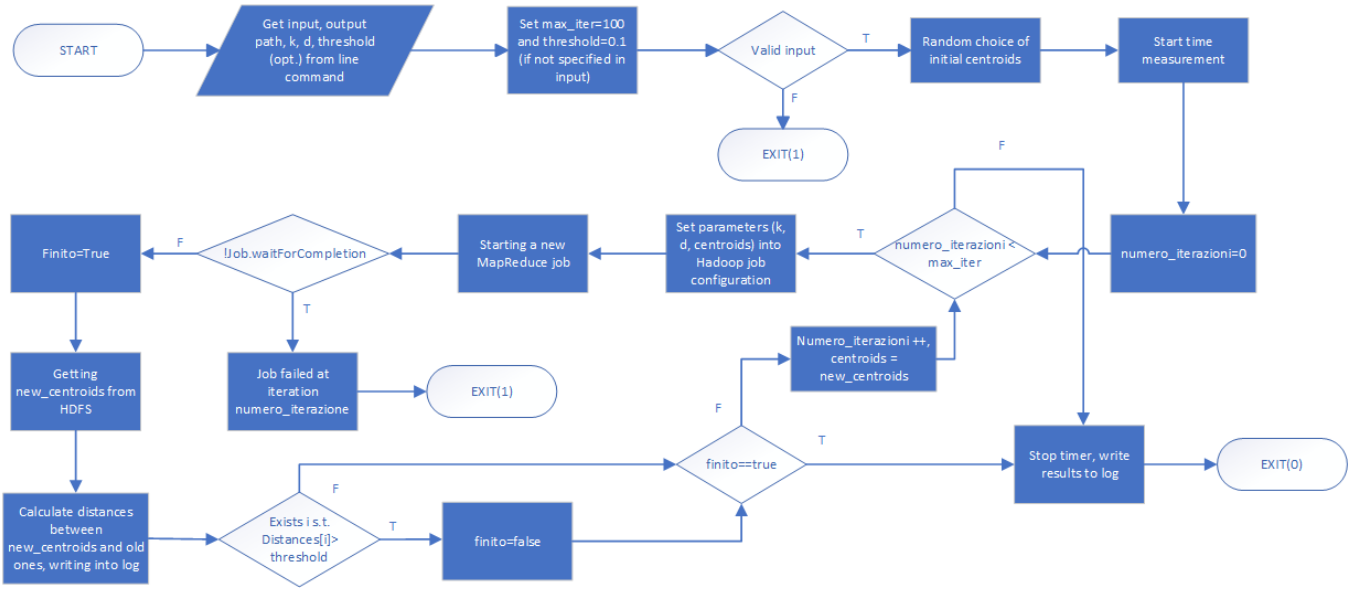


Image 1: Flowchart diagram Figure 1: Flowchart diagram

2.4 Data representation

We have developed a versatile object that serves as a representation of a point. In this implementation, it can be used to describe three different scenarios. Firstly, it can represent a point from the original dataset. Secondly, it can be used to represent the point emitted by the Combiner, which is the partial sum of a subset of points involved in the computation. Lastly, it can also be employed to represent the final centroid emitted by the Reducer. This provided a unified approach to represent points throughout the workflow, without having to use various data structures. Each point object is composed by the following attributes:

- **Coordinates:** It's an arrayList of Doubles where each element is a coordinate of the point; its size is *d.*
- **Cluster** (default value: -1, this means the point is not associated with any cluster yet): It's an integer that represents the index of the cluster the point is associated with.
- **Min_distance** (default value: -1, that means the distance between a point and a cluster is not computed yet): This attribute is used to store the minimum point-to-cluster distance, in order to perform the association.

- **Weight:** This attribute is used when it's needed to compute the sum of multiple points, in order to save the information about their quantity.

2.5 Dataset generation

To carry out the K-means algorithm, we opted to generate a synthetic dataset; to accomplish this, we employed a Python script specifically designed for this purpose. PointGenerator.py, requires two input parameters: *n* and *d*. It utilizes these parameters to generate *n* random points, each consisting of *d* dimensions. These points are represented as vectors of double numbers and are generated within the interval of [-100, 100]. The dataset created is a text file structured as follows:

- each row represents an element of the set *X*.
- each component of the point is separated from the next component by a comma.

The name of the file is '**coordinates.txt'.**

3. Development and design choices

3.1PointWritable Class

The PointWritable Class is designed to handle points in a multidimensional space and perform various operations on them. It implements the **Writable** interface, which enables efficient serialization and deserialization of objects for storage and transmission across cluster nodes. It provides constructors to initialize the point from an array of doubles, an array of string or another point itself. It also supports methods for calculating the Euclidean distance between two points, setting and getting the point's attributes and performing point summation. The class also implement some methods for parsing variables. The number of elements processed by the Combiner class is maintained with the *weight* attribute. This field is incremented immediately after the call of the summation method. This is necessary in order to make the Reducer

aware of the points that have been added during the partial sum performed by the Combiner.

3.2 MapReduce design

3.1.1 Mapper

The MapperClass accepts in input a file containing the dataset and three initial centroids, which are randomly chosen from the elements of set X. The framework may decide to use multiple mappers depending on the size of the input data. It's called for each line of the input data (so for every element of X) and assigns to each element of the input set the ID of the nearest cluster. The output key-value pair in our case is <cluster ID, Point>.

3.1.2 Reducer

The ReducerClass accepts in input the output of the MapperClass after the shuffle and sort phase. After this, data is structured as follows <cluster ID, list of Points>; where the list of Points represents the Points that belong to that cluster. The Reducer is executed for each value of the key (so, for each cluster) and it computes the new position of the centroid. The output key-value pair is <cluster ID, new centroid>.

3.1.3 Combiner

The CombinerClass is not executed necessarily; in fact, it's not required for the proper flow of the algorithm. Assuming it is always utilized, the Combiner runs locally for each Mapper, and its function is to reduce the data transmission over the network. It's as structured as the Reducer, so the same concepts seen in the previous paragraph for their input still fits. The Combiner computes the partial sum of the Points belonging to a cluster and upload the weight attribute consequently. The output key-value pair is <cluster ID, partial sum of Points>.

4. Tests and results obtained

4.1 Tabular representation

Dataset – num. Points, dimension	Parameters	Seconds elapsed	Number of iterations
100 points, 2 dimensions	K = 3	209	9
	K = 5	153	6
	K = 10	213	10
100 points, 5 dimensions	K = 3	116	5
	K = 5	152	6
	K = 10	277	10
100 points, 10 dimensions	K = 3	138	6
	K = 5	232	9
	K = 10	210	7
100k points, 2 dimensions	K = 3	660	35
	K = 5	420	20
	K = 10	908	35
100k points, 5 dimensions	K = 3	899	48
	K = 5	2067	100*
	K = 10	2496	100*
100k points, 10 dimensions	K = 3	1143	61
	K = 5	2107	100*
	K = 10	2526	100*
1 million points, 2 dimensions	K = 3	450	22
	K = 5	497	22
	K = 10	1208	44
1 million points, 5 dimensions	K = 3	474	21
	K = 5	2458	100*
	K = 10	2951	100
1 million points, 10 dimensions	K = 3	748	32
	K = 5	2378	68
	K = 10	2869	100*

All the results were obtained with the following parameters:

- Threshold = 0.1
- Max_iter = 100

NOTE: The experiments that concluded without converging are indicated with an '' in the number of iterations field.*

4.2 Plot Representation

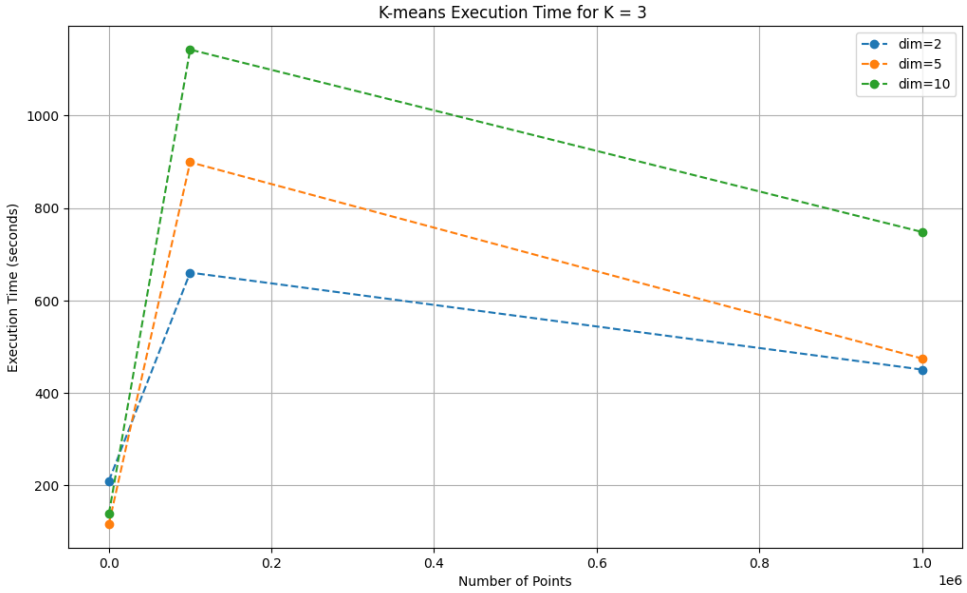


Image 2: K-Means execution time for K=3

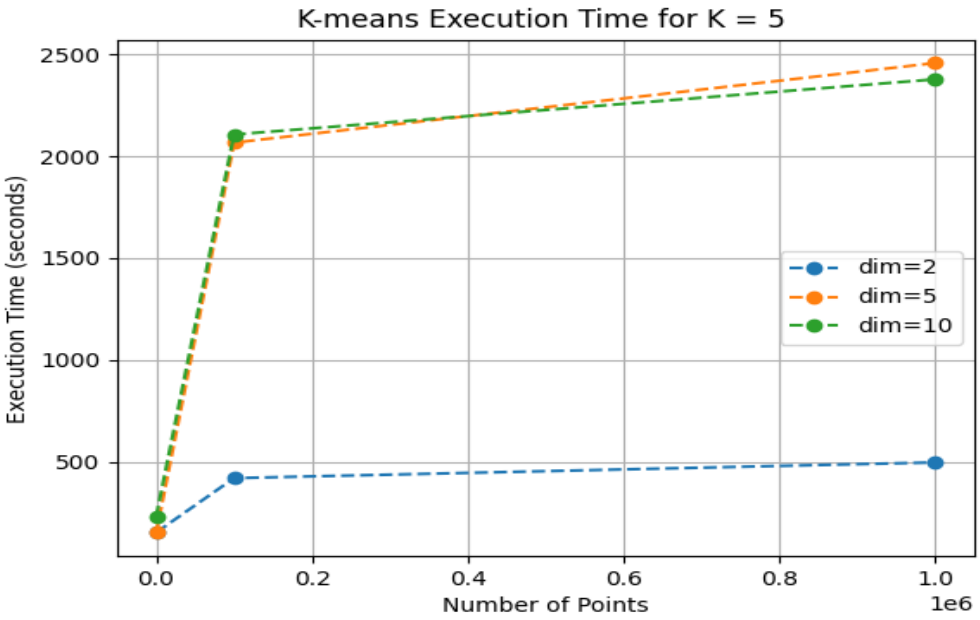


Image 3: K-Means execution time for K=5

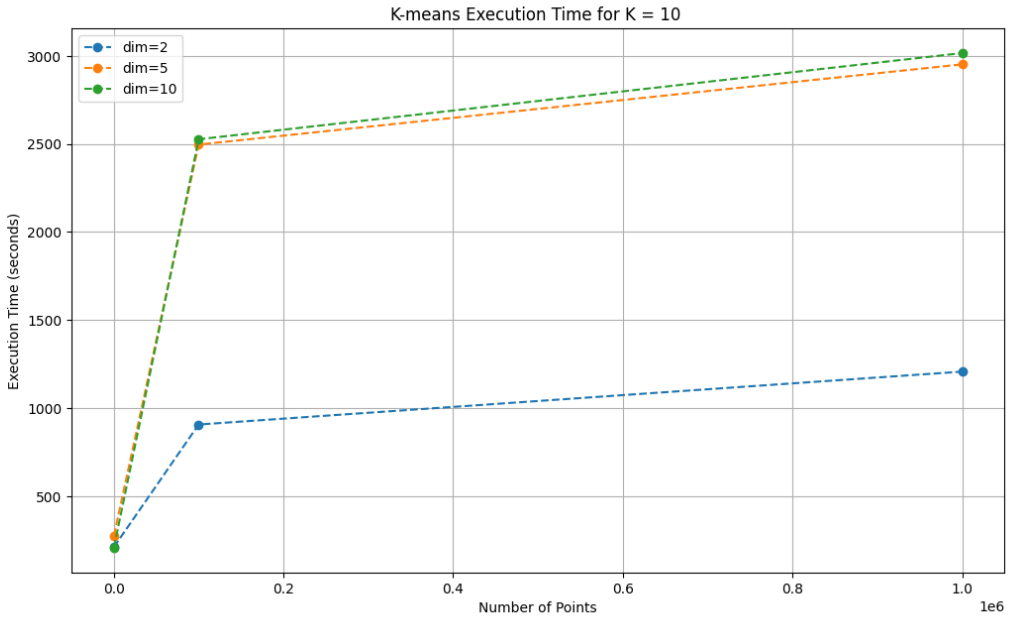


Image 4: K-Means execution time for K=10

The graphs show how the elapsed times changes for increasing number of points, by keeping the K input parameter constant. Particularly, inside each plot we can see three different polylines: these represent the shifting in time duration by keeping the "*dimension*" parameter fixed.

4.3 Visual evolution of the algorithm

To enhance the understanding of the algorithm's behavior, we utilized the capabilities of Matplotlib, a widely used Python library for mathematical analysis. Our goal was to provide a visual representation of the K-means algorithm's evolution, and to do this we implemented a script to generate a graphical plot of the situation after each iteration. It was called if the number of points was not excessively large ($num\_points < 1000$), in order to avoid significant impacts on the algorithm's duration. Additionally, this visualization feature was specifically designed for points in a two-dimensional space (R^2). To better visualize the evolution, all the generated plots were saved in a designated folder and then eventually combined into a GIF file, to also offer an animated representation of the algorithm's progression.

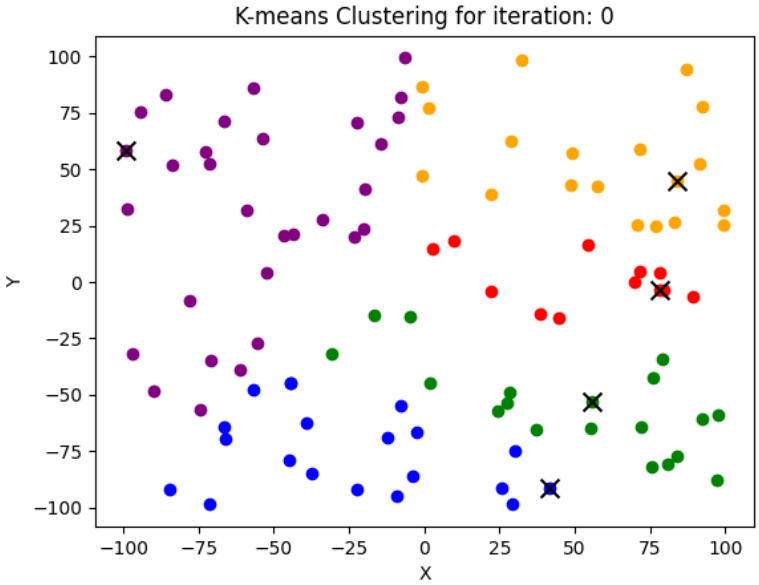
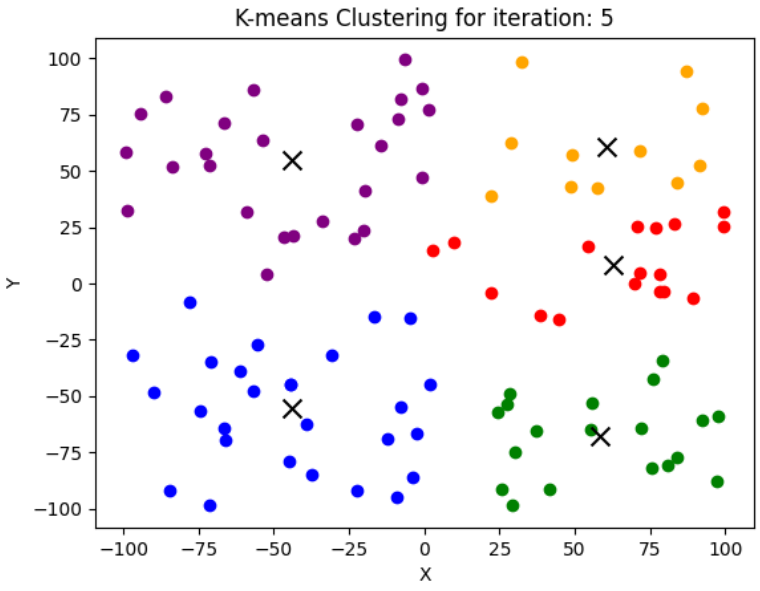


Image 5: Starting situation of the K-Means algorithm for a dataset made up of 100 points. K=5

Image 6: Last iteration of the K-Means algorithm for the state provided in Image 4



Examples of the *gifs* produced by this visual production scripts can be found at the following:

- Evolution of K-Means with 100 points, *K*=3 and *d*=2
- Evolution of K-Means with 100 points, *K*=5 and *d*=2
- Evolution of K-Means with 100 points, *K*=10 and *d*=2
- Evolution of K-Means with 100k points, *K*=3 and *d*=2
- Evolution of K-Means with 900 CLUSTERIZED points, *K*=4, *d*=2
- Evolution of K-Means with 900 CLUSTERIZED points, *K*=5, *d*=2
- Evolution of K-Means with 900 CLUSTERIZED points, *K*=6 and *d*=2
- Evolution of K-Means with 900 CLUSTERIZED points, *K*=10 and *d*=2

They all showcase well the initial random choice of centroid as values of the dataset, then their evolution which ultimately may end with points that are not included in X.

5. Conclusion

From the obtained results, it can be noticed that the execution times are influenced by the size of the dataset, both in terms of the number of points and the dimensionality of the vector components. What is important is to underline that despite this correlation existing, it is not linear. In fact, it is not true that having a ten times bigger dataset (for example, from 100,000 to 1,000,000 points) also triggers a ten times bigger execution time. In some cases, it is even appreciable that the time remains almost constant. Let's take the example of the graphs created with k=10: the lines corresponding to d=5 and d=10 remain very close within the range of 100,000 to 1,000,000 points. This is presumably since the input file with d=10 exceeds the size of 128Mb and is therefore split, which makes the processing (performed in parallel for each split) more efficient.

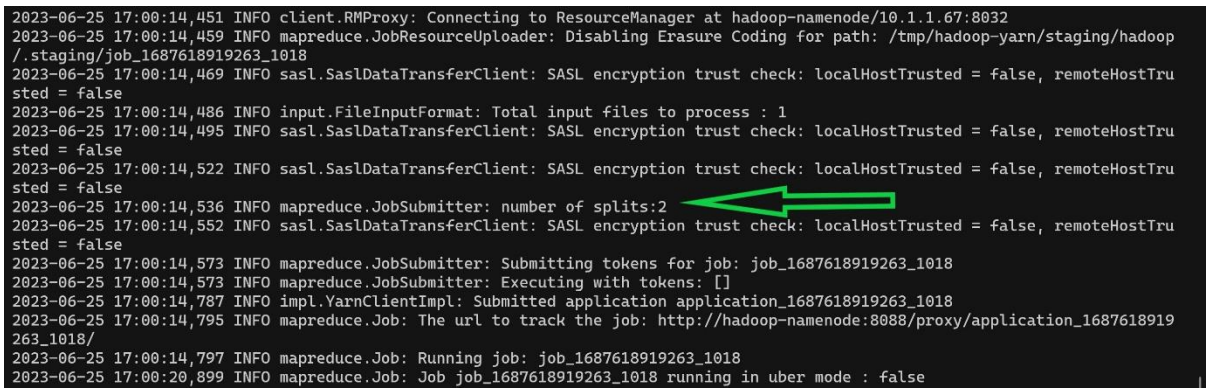


Image 7: Input is split if its dimension exceeds 128MB.

We can say that this is the main demonstration of the Hadoop's suitability in big data handling. Another aspect to consider is convergence: with the used parameters (*max_iter*=100, *threshold*=0.1), the algorithm reaches a stage where the movement of all centroids is lower than *threshold* within a number of iterations that is lower or equal than *max_iter*. This information is significant as it allows us to make proper comparisons regarding execution times. Based on literature, we know that the performances of K-means are influenced by the initial choice of centroids, and the obtained results are partly derived from this internal characteristic of the algorithm under consideration. Starting with centroids placed strategically rather than randomly, especially when we have information about the positioning of one or more clusters in certain applications, can decrease the convergence time.

6. Appendix

- GitHub repository for the project