UNIVERSITÀ DI PISA

MSc Computer Engineering
Cloud Computing Project

The k-means Clustering Algorithm in MapReduce

(Hadoop Framework)

Group Members:
Francesco Bruno
Gaetano Sferrazza
Leonardo Bargiotti

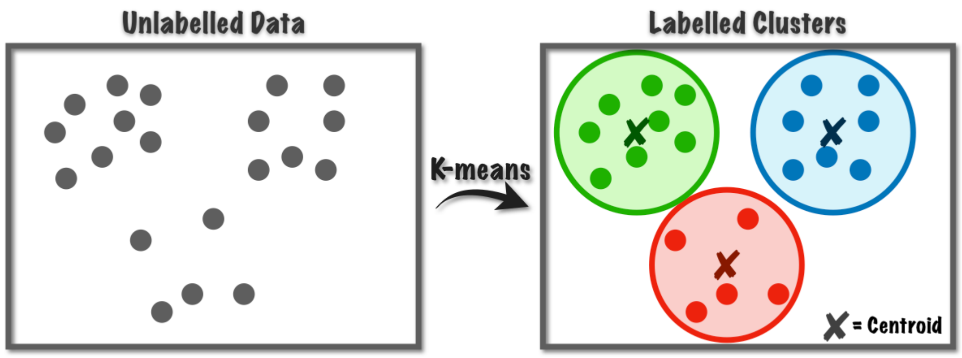
Academic Year: 2022/2023

Contents

Introduction

The Basic Idea

K Means is one of the most popular clustering algorithms. K means stores k centroids that it uses to define clusters. A point is considered to be in particular cluster if it is closer to that cluster's centroid than any other centroid. K Means finds the best centroids by alternating between (1) assigning data points to clusters based on the current centroids (2) choosing centroids (points which are the center of a cluster) based on the current assignment of data points to clusters.



K-Means Pro and Cons

The KMeans Algorithm is easy to use, and its steps are not to much. If your data has no labels (class values or targets), K-Means will still successfully cluster your data. K-Means returns clusters which can be easily interpreted and even visualized. The algorithm has linear time complexity, and it can be used with large datasets conveniently.

However, the KMeans has several drawbacks:
One of the inconsistencies of K-Means algorithm is that results will differ based due to random centroid initialization. Unless you pick the centroids at fixed positions, which is not a common practice, K-Means can come up with different clusters after its iterations.
It can be very difficult to find the number of K clusters.
Since k-means clustering works by calculating the distance between your data points and the size of centers of your clusters, it can be thrown off by situations where your variables have different scales.

In order to implement the K-Means algorithm in the Hadoop environment, a MapReduce approach was adopted.

Design

Apache Hadoop software is an open-source framework that allows for the distributed storage and processing of large datasets across clusters of computers using simple programming models. By splitting the data file into chunks (64MB or 128MB), hadoop can process each chunk on to a different node, creating a parallelized execution. This can result in crucial reduction of processing time.

In particular, the K-Means algorithm can be implemented using the MapReduce programming approach. The core of that approach is the Map and Reduce method.

- The first, will map the dataset input data into a set of value with a common feature (represented by the key).
- The latter will receive for each key generated by the mapper, a list of *value* associated with that *key.*

In the following paragraph, it will be explained the Mapper, Reducer and Main function with relative pseudo-code.

K-Means Mapper

This function, automatically invoked by Hadoop, has the purpose of map each point that compose the input dataset with the relative closest centroid.
To implement this job, the map function iterates, for each dataset point, over the list of all *k* centroids to find the closest in term of Euclidean Distance.
In details, when the map function is invoked, it will receive a single dataset point and in the loop over the centroids, if the actual distance from that point to the actual centroid is less than the distance in the before loop, then it will be saved the ID of that centroid in order to say "This passed point is closest to the centroid with that specific *ID".*
At the end, for that point,it will be found its closest centroid and the pair *< key-value >* will be *< IDClosestCentroid , Point >.*

Notes that before the *Mapper* execution, there will be the *Setup* execution. That function takes care of load from the context, the centroid's list stored by KMeansMain.

Below it’s shown the mapper pseudo-code:

Mapper

```
Algorithm 2: MAP (LongWritable key, Text line)
  Result: map each point and assign it to the closest centroid
  Input:
          key: record number
          line: inputSplit’s record line, that represents the key-th Point
  Output:
          centroidID: ID closest point’s centroid
          point: parsed point from line
1 point ← parsePointFromLine(line)
2 centroid ← new Centroid()
3 minimumDistance ← MAX_DOUBLE_VALUE
4 for all Centroid c in centroidPassedFromKMeans do
5     distance ← c.computeEuclideanDistanceFromPoint(point)
6     If distance < minimumDistance then
7         minimumDistance ←distance
8         closestCentroid ← c
9 EMIT(closestCentroid.ID, point)
```

K-Means Reducer

The output of mapper function, after *Shuffle & Sort* phase, is given in input to the Reduce function. This method, will receive in input a < *key, value* > pair formed by < *IDClosestCentroid , list of point* > , so for each cetroid there will be a partition that contains all the point near to it. So, every reducer will receive the list of all the dataset point closest to the centroid with that specific *ID.*

In details, for each dataset point in that list, it will be computed the mean point (by calculating the average for each component) that represents a better approximation of the real centroid (*mean point)* of that cluster.
After, the reduce will emit the pair < *IDClosestCentroid , meanPoint* >

The reducer pseudo-code is shown below:

Reducer

```
Algorithm 3: REDUCE (LongWritable key, points [p1, p2, … ] )
  Result: find the mean centroid from the iterable point list
  Input:
          key: centroid's ID assigned to this reducer
          points: list of all point nearest to the assigned centroid
  Output:
          key: ID centroid
          point: point that represents the meanCentroid
meanCentroid ← new Centroid()
pointCounter ← 0
for all Point p in points do
    meanCentroid.sum(p)
meanCentroid.calculateMean()
meanPoint ← parseCentroidToPoint(meanCentroid)
EMIT(key, meanPoint)
```

K-Means Main Function

The main function is not so complicated. It consists of:

- Parse input arguments into relative object.
- Generate the initial centroids by use a random method.
- Initialize the "stop loop" variables.
    - Instantiate and configure the Hadoop Job.
    - Prepare the actual centroid to the map function by write them in the context.
    - Launch the Hadoop Job.
    - Retrieve the results.
    - Check if is necessary to loop again.
- Write the results of file.

Below it’s shown the K-Means main pseudo-code:

K-Means

```
Algorithm 3: kMeans(dataset, numberRedcuers, k, dimension, epsilon, iteration, output)
Result: The k-means algorithm using distributed computation
Input :
        dataset: Dataset File name with the complete path
        numberReducer: Number of Reducers
        k: Number of Clusters
        dimension: Number of coordinates to work with
        epsilon: error threshold for convergence
        iteration: maximum number of k-means iteration
        output: Output Directory
Output:
        The list of all mean centroids.
1  recover the input arguments by allocation the conf object
2  newCentroids ← randomCentroidGenerator()
3  stopIteration ← false
4  iteration ← 0
5  while !stop do
6      oldCentroids ← new ArrayList()
7      iteration ← iteration + 1
8      for all Centroid c in newCentroids do
9          storeCentroidInContext (c)
10     MAP each point and assign it to the closest centroid
11     REDUCE to find the mean centroid for each cluster
12     for all Centroid c in newCentroids do
13         oldCentroids.addCentroid(c)
14     newCentroids ← retrieveIterationResults()
15     stop ← checkConditions()
16 end
17 writeFinalCentroidOnFile(newCentroids)
18 writeComputationalStatistics()
```

Implementation

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

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

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 several methods:

- *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> <numberReducers>* and it takes the number of centroids (K), dimensions, threshold and the maximum number of interations from the configuration file *config.propierties*. The method first checks if the number of arguments is valid.

  The initial centroids are read randomly from the input file using the method *randomCentroidGenerator* of Centroid class and written in the file *initialRand_Centroids.txt*.

  It first creates the hadoop configuration and initializes the file system. Then, it configures and submits the MapReduce job and checks the completion using the method *checkConditions*.

  For each iteration retrieves the new centroid from the current interaction result file using the method *retrieveResults.*

  When the check condition is satisfied, it writes in *finalCentroids.txt* the final centroids and prints in the terminal same information (total execution time, total iterations, number of converged centroids).

- checkConditions (List <Centroid> newCentroids, List <Centroid> oldCentroids, int K, double EPS, int MAX_ITER, int iterations)

  It calculates the stop condition, in particular returns true if:

  - the iterations are greater or equal to MAX_ITER
  - the Euclidean distance between *oldCentroids* and *newCentroids* is less op equal to EPS

  otherwise returns false.

  It also counts the number of converged centroids.

- retrieveResults (String OUT_FILE, Configuration conf)

  It reads the centroids returned by reducer from the file of the previous iteration

  - writeCentroids (Configuration conf, List <Centroid> centroids, String output)

    It writes centroids (*ID tab Coordinates*) into output file.

  - writeInfo (Configuration conf, String[] printInfo, String outputFile)
    It writes execution info (*printInfo*) into file (*outputFile*)

Point

- Point()

  This constructor creates a new Point object without specified coordinates. It initializes the coordinates field as an empty ArrayList.

- Point (final int n)

This constructor creates a new Point object with n coordinates, initialized to 0.

- Point (final List<DoubleWritable> coordinatesList)

This constructor creates a new Point object with the specified coordinates.

- Point (String coordinatesReceived, int configurationDimension)

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 until the dimension is respected.

- write (final Data0utput out)

This method writes the Point object to the specified data Output stream.

- readFields(final DataInput in)

This method reads the Point object from the specified dataInput stream.

- compareTo (Point o)

This method checks if 'this' and 'o' Points objects are the same point so it returns 0 if equals or else 1.

- toString

This method returns a string representation of the Point object. It converts each coordinate to a string and concatenates them, separated by comma, to form the string representation.

- getCoordinates()

This method returns the coordinates field of the Point object.

Centroid

- Centroid()

This constructor calls the empty constructor of the superclass Point and set id field to -1.

- Centroid (IntWritable id, List<DoubleWritable> coordinates)

This constructor calls constructor of the superclass Point specifying the coordinates and set id field to id.

- Centroid (String coordinates, int configurationDimension, int ID)

This constructor calls constructor of the superclass Point specifying the coordinates as string and the number of dimensions, and set id field to ID.

- write (Data0utput out)

This method writes the Centroid object to the specified data Output stream.

- readFields (DataInput in)

This method reads the Centroid 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 comma, to form the string representation.

- getId()

This method provides the centroid ID of the centroid.

- setId (IntWritable newId)

This method sets the centroid ID of the centroid.

- copy()

This method returns a new Centroid object with the same id and coordinates.

- findEuclideanDistance(Point point)

This method returns the Euclidean distance between the Centroid and the Point passed as argument.

- add (Point currentPoint)

This method adds the coordinates of the specified Centroid to the current Point object passed.

- calculateMean (long numberPoints)

This method updates coordinates of the Centroid, calculate as the coordinates divided by the number of points belong to its cluster

- randomCentroidGenerator(String INPUT_FILE, String k, String DIM, Configuration conf)

This method returns a List of k (passed as argument) Centroids randomly chosen from the file passed as argument. First it calculates k random numbers that correspond to the number of the line and then it takes the centroids that correspond to the lines.

- getLineNumber (String INPUT_FILE, Configuration conf)

This method returns the number of Lines of input file passed as argument.

Experimental Results

This section presents the results obtained from the K-Means algorithm using Hadoop MapReduce.

The algorithm was evaluated for 3 main datasets with different ***number of points** (n)*, for each of them 4 different scenarios characterized by the variation of the following parameters was evaluated:

- number of clusters (k)
- dimensionality of points (d).

The objective was to analyze the algorithm's performance and accuracy under different conditions.

Before to apply the k-means algorithms to our data, we have to fix the first centroids. To do so we could have chosen between different techniques, we have chosen the *random way* approach.

Generation of Synthetic Data

We decided to write a simple Python program that generates syntethic data with clusterable distribution; the code creates a dataset using the *make_blobs* function from *Scikit-learn* which allows us to specify number of samples, features, and clusters (it retrieves these parameters as arguments from the command line using the *sys.argv* list.

It creates a *pandas DataFrame* to store the data, where each row corresponds to a data point with its features. 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. The code then saves the data to a *.txt* file.

Tests

Test 1

Performed on dataset with ***n = 1000*** points.

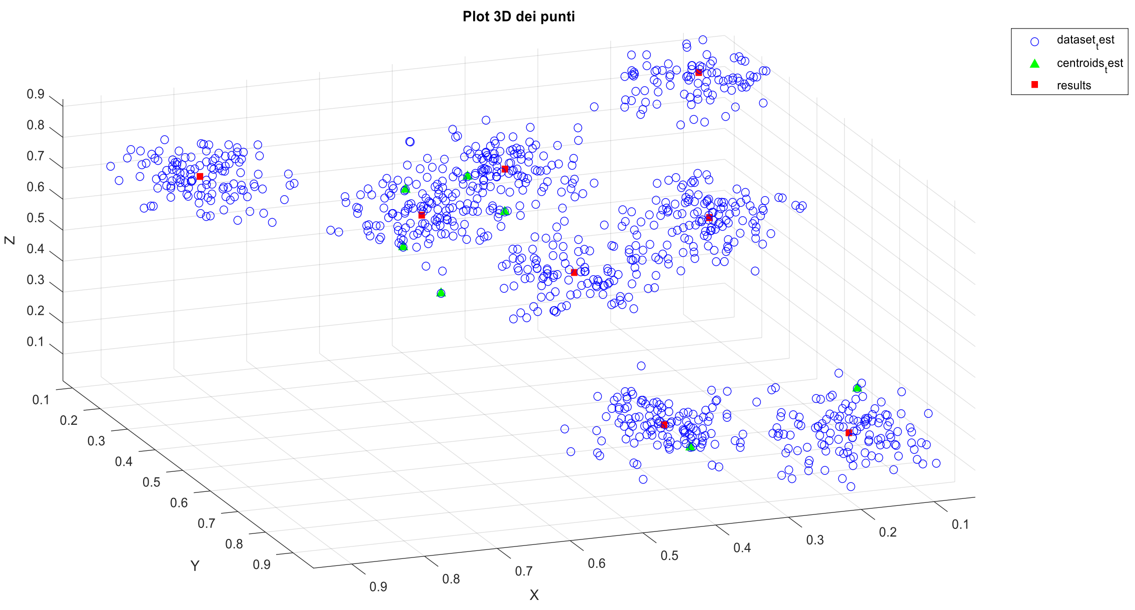
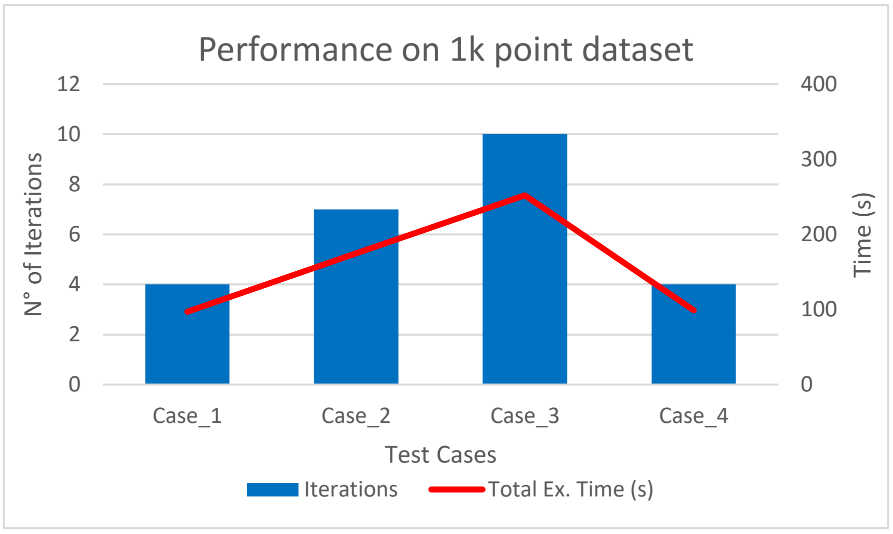


Figure 3.1: **Plot Test 1 - Case1:** d=3, k=8



CASES	***ITERATIONS***	***TOT. EX. TIME (s)***
Case 1	4	97
Case 2	7	175
Case 3	10	252
Case 4	4	99

Figure 3.2: **Case1:** d=3, k=8 **Case2:** d=3, k=15 **Case3:** d=8, k=8 **Case4:** d=8, k=15

Test 2

Performed on dataset with ***n = 10.000*** points.

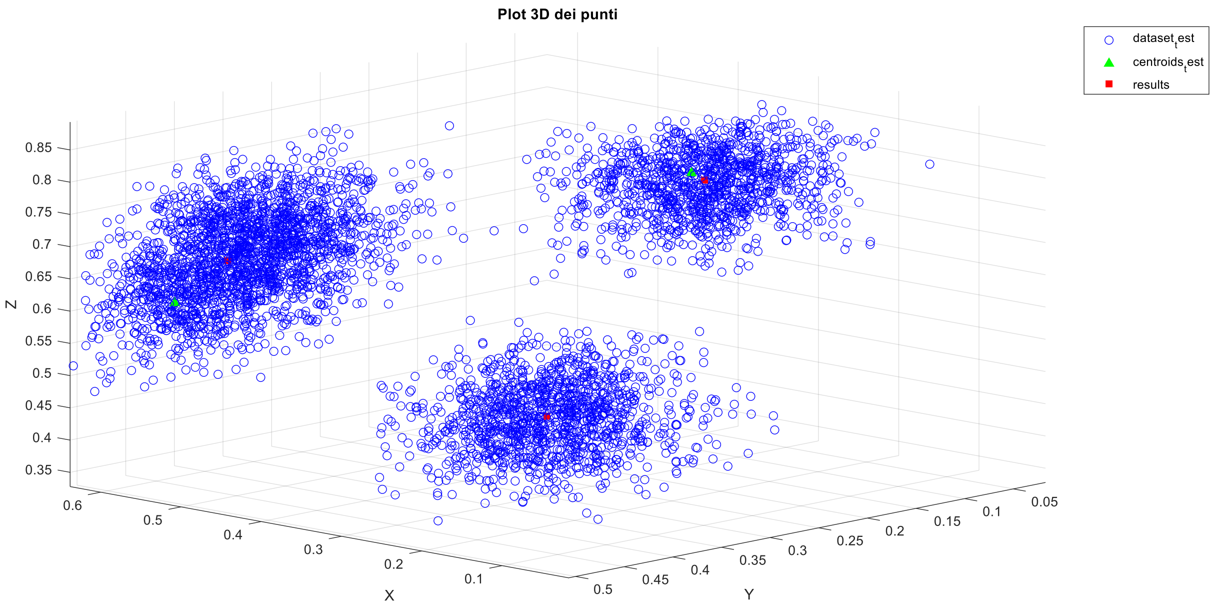
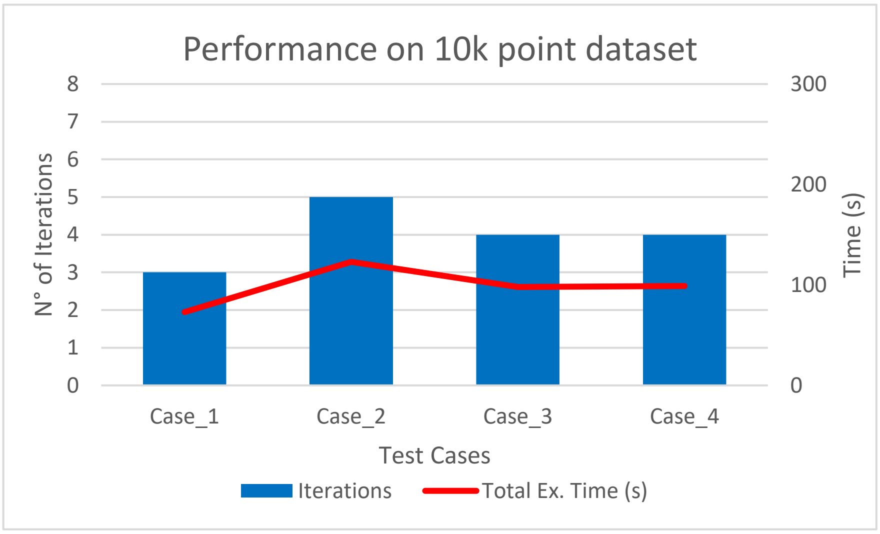


Figure 3.3: **Plot Test 2 - Case1:** d=3, k=8



CASES	***ITERATIONS***	***TOT. EX. TIME (s)***
Case 1	3	73
Case 2	5	123
Case 3	4	98
Case 4	4	99

Figure 3.4: **Case1:** d=3, k=8 **Case2:** d=3, k=15 **Case3:** d=8, k=8 **Case4:** d=8, k=15

Test 3

Performed on dataset with ***n = 100.000*** points.

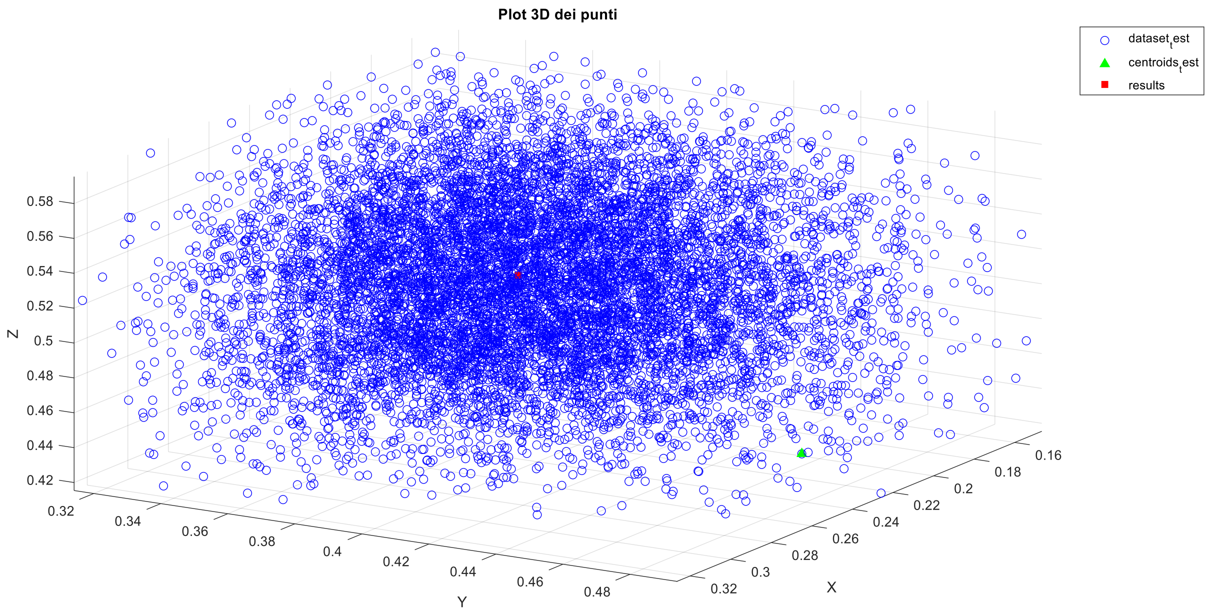
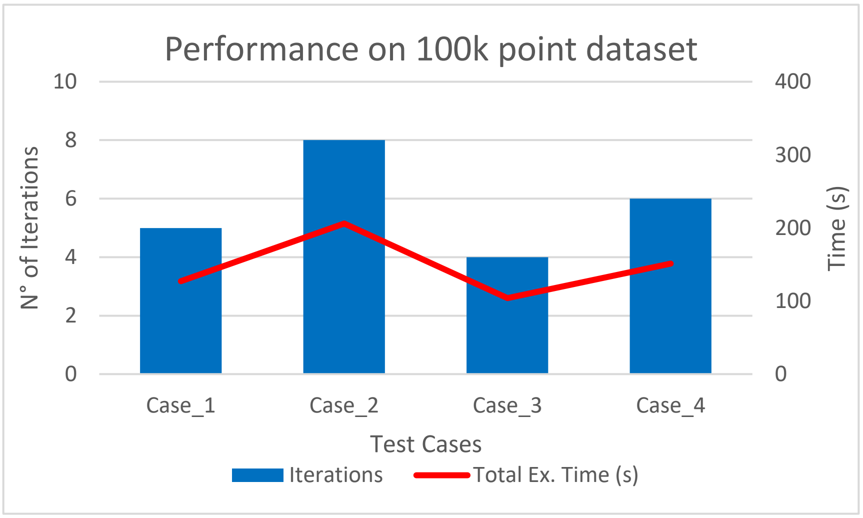


Figure 3.5: **Plot Test 3 - Case1:** d=3, k=8



CASES	***ITERATIONS***	***TOT. EX. TIME (s)***
Case 1	5	127
Case 2	8	206
Case 3	4	104
Case 4	6	151

Figure 3.6: **Case1:** d=3, k=8 **Case2:** d=3, k=15 **Case3:** d=8, k=8 **Case4:** d=8, k=15

These are our experimental results of the Hadoop Map reduce where we have picked our initial centroids in a randomly way. As we can see, the times and the iterations are like linearly independent. In fact, they present the same gait, so if the iterations grow time also increase. In the last scenario, where we have the maximum dataset, time grows as soon as the number of iterations increase but, the time (line in red) is below the iterations (area in blue) due to the fact that Hadoop works better with a huge amount of data.

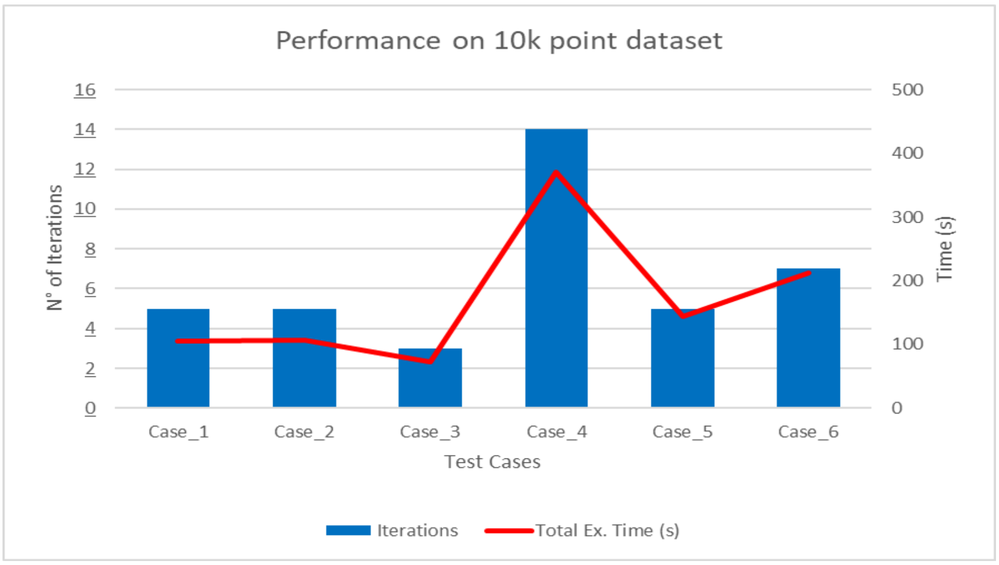
Furthermore, it can be seen that compared to the initial centroids, the final centroids calculated by the algorithm are quite centered in the relevant clusters, but for sure makes change to the initial centroids can influence the final result and the improvement in terms of grouping the different clusters.

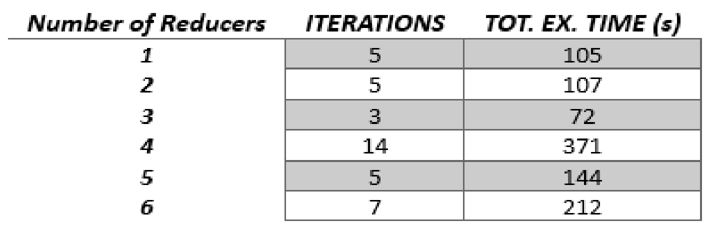
Number of reducers

The impact of the number of reducers on the K-Means algorithm execution time was examined in order to assess how the latter can be influenced related to the variation of reducers. Moreover is important to evaluate because when running MapReduce jobs, the number of reducers affects load balancing and resource utilization.

Hadoop's documentation suggests selecting the number of reducers by a specific formula that states: multiply the number of nodes by 0.95 or 1.75. This allows us to better understand what is the better number of reducers to achieve the best performance of the application. In our case we have one namenode and two datanodes for a total of 3 active nodes in the cluster, for this reason the best configuration should be in a range about between 2 or 5 reducers.

The average execution time for one iteration of Test 2, using different numbers of reducers, was conduct with following results:





Number of Reducers	*ITERATIONS*	*TOT. EX. TIME (s)*
1	5	105
2	5	107
3	3	72
4	14	371
5	5	144
6	7	212

Figure 3.7: **Case1:** d=3, k=8

The results obtained for different numbers of reducers, although can be influenced by several factors including the hardware configuration, network bandwidth, and the characteristics of the dataset, shows what we expected. More precisely 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 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. As we can see, as soon as the number of reducers begins to exceed the number 5, the workload may not be distributed optimally. The limited number of nodes, that we have in our case, compared to the number of reducers can lead to increased data transfers and potential congestion, causing longer execution times.

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: The accuracy of the K-Means algorithm is significantly influenced by the selection of initial centroids.

1. The accuracy of the K-Means algorithm is significantly influenced by the selection of initial centroids.
2. Datasets with higher dimensionality indicate greater complexity.
3. The greater the amount of data submitted to hadoop is, better hadoop works by increasing utilization.
4. 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 improve the scalability and performance of the KMeans algorithm implemented using Hadoop MapReduce, a potential future improvement could be the utilization of Hadoop's ***caching mechanism*** and the usage of different ***techniques to initialization*** of the algorithm. Considering the current behavior of the algorithm, where its passes centroid information to each iteration through the MapReduce configuration, we can state that this can lead to a scalability bottleneck as the number of iterations and the size of the centroid information increase. The Hadoop's caching capabilities approach assess this situation storing and access the centroid information in a more efficient and scalable manner. In fact, using caching capabilities, the algorithm can benefit from improved ***scalability***, ensuring that the centroid data is efficiently distributed across the cluster's nodes for seamless scalability without overwhelming the network or storage resources, faster ***convergence*** due to the fact that for each iteration of the K-Means algorithm it is possible access the data directly from the cache instead of retrieving it from the configuration, and enhanced ***load balancing*** thanks to the cache of the centroid information that leads the algorithm can leverage Hadoop's built-in load balancing capabilities ensuring that the computational workload is evenly distributed improving overall performance.

Regarding the various initialization techniques of the K-Means algorithm that could improve the final accuracy, it can take in consideration the ***K-Means++ algorithm*** to replace the standard random approach. This algorithm 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.