UNIVERSITÀ DI PISA

Cloud Computing

K-means algorithm in Map-Reduce

Group Paolo Bitta

Iacopo Canetta

Francesco Berti

Alessandro Versari

ANNO ACCADEMICO 2022/2023

Contents

1 Problem Analysis

K-Means is a simple but efficient clustering algorithm used for data analysis in different fields. This clustering operation however, if run on a large set of data, might require a high time to be performed. The goal of this project is to solve this issue by using the power and functionalities of cloud computing. To achieve this result, the algorithm is implemented using the framework *Apache Hadoop* and, thanks to it, distributed on a cluster of three nodes.

2 Main Idea

K-Means is an algorithm that runs on a set of n data samples, each having d features; the goal of the algorithm is to group these samples into k clusters, with the value of k chosen arbitrarily.

2.1 Classic Algorithm

The algorithm starts by choosing k starting points named centroids, which at the end of the algorithm will be the center of the k clusters. There are many ways to choose the starting centroids but we will assume that the centroids are chosen randomly among the datasaset for the purpose of this study. After the starting setup, the algorithm consists of two steps:

- Assigning every point to the closest centroid. The distance used is the euclidean distance;
- Updating the centroids' coordinates by averaging of the points assigned each centroid.

These steps are repeated until the centroids converge (they remains the same or they change by a factor smaller then a setted treshold), or after an arbitrary number of max-iterations.

2.2 Parallelized Algorithm

2.2.1 Hadoop introduction

The *Hadoop* framework uses the *MapReduce* paradigm which consist on having three two steps:

- a *Mapper* class which given a value links it to a key by forming a pair *[Key, Value]*
- a *Reducer* class which receives all the *Value*s linked to a *Key* and "reduces" them to a single value

An additional step can be added in between, it's named *Combiner* and it is used to speed up the execution by "combining" some of the *Value*s into groups, which then will be "reduced" by the *Reducer* class.

2.2.2 Implementation

The basic implementation of *parallelized K-means* in *Hadoop* (without the *Combiner* class) is to:

- build the starting centroids in the setup
- use the *Mapper* class to link each point to the index of the nearest cluster
- calculate the new centroids by averaging all the points linked to a cluster (with the key being the index of the centroids)
- repeat until a stopping condition is met

Then there are two possible ways to include a *Combiner* class:

- use the *Combiner*s to do the sum and use the *Reducer*s to calculate the division of the average
- introduce the concept of *weighted* point and make a partial averaging in the combiners, then do the final average in the reducers

In this implementation the second approach was chosen since it allows to use the same class for the *Combiners* and *Reducers* which results in a more even distribution of the load between reducers and combiners; in this way *Hadoop* has the capabilities to scale better regardless of the input dataset.

2.2.3 Pseudo Code

```
function Map(Point)
    Key ← argmin_j distance(Point, Centroids[j]) with j ← 1 to k
    return [Key, Point]
end function

function Reduce(Key, Points)
    NewCentroid ← weightedAverage(Points)
    return Key, NewCentroid
end function

function Combine(Key, Points)
    Key, NewCentroid ← weightedAverage(Points)
    return Key, Point
end function


function Kmeans(Data[n, d], k, Threshold, MaxIterations)
    Iteration ← 0
    Centroids[k, d] ← subset of k random elements from Data

    while Iteration < MaxIterations do
        Points[k]
        Distance ← 0
        for i ← 1 to n do
            Key, Point ← Map(Data[i])
            Points[Key] ← Points[Key] + [Point]
        end for

        // this section is skipped if combiners are not used
        CombinerPoints[c] ← c sublists of Points[i] (performed by Hadoop)
        PointsCombined[k]
        for i ← 1 to c do
            Point, Key ← Combine(Points[i])
            PointsCombined[Key] ← PointsCombined[Key] + [Point]
        end for

        //Hadoop is responsible to send the points to the reducer (combined or not)
        for i ← 1 to k do
            //If combiners are not used, Reduce receives Points[i]
            Key, NewCentroid ← Reduce(PointsCombined[i])
            Distance ← Distance + distance(Centroid[i], NewCentroid)
            Centroid[i] ← NewCentroid
        end for

        if Distance < Threshold then
            break
        end if

        Iteration++
    end while
    return Centroids
end function
```

The algorithm has as input an array *Data nperd* of data samples and an arbitrary number K of clusters. If *Combiners* are used, after the *Map()*, *Hadoop* divides the Points in c batches and assigns them to the c *Combiners*; the points received by the *Reduce()* are mapped and/or combined points, depending on *Hadoop*'s choices

3 Java implementation

The code has been implemented with *Hadoop* by expanding the *Mapper* and *Reducer* classes and by implementing a class *Point* which extends the *Writable* interface.

3.1 Main class

The class *Main* includes the *Main()* function and two utility *log()* functions, to print the logs of what is happening, using different colors to make it more readable.

The *Main()* function retrieves from the user:

- the path of the starting points (from which both the dimensions of points and K the number of cluster is inferable)
- the input folder
- the output folder

This function has a *for* loop, iterating until any of the algorithm's stopping conditions is reached. In every iteration, the *Hadoop* job is set up and receives the configuration containing the current centroids for the *Mapper* class; the *Hadoop* job is launched and, if it has finished correctly, the result is extracted and the new centroids are set.

3.2 Point class

The *Point* class is responsible to store a set of *Coordinates* and provide the following methods:

- *parsePoint()* : given a string, returns a *Point*
- *toString()* : returns the representation of the *Point*'s instance
- *distance()*: returns the distance between a *Point* and the provided *other Point*
- *nearest()*: given an array of *Point*s, returns the index of the nearest one
- *average()*: averages the provided set of *Point*s

while overriding this methods:

- *write* and *readFields* which are used from *Hadoop* serialized and *de-serialize* the *Point* class

To be compliant with the usage of the *Combiner*s, the *Point* class also stores an integer value named *weight*, that keeps track of how many points the current instance represents.

3.3 HadoopUtil

This class provides two utility functions to be used while working with hadoop:

- *extractResult()* : given the configuration, the output path, and the used K, returns the array of new *Point*s which are the result of an *Hadoop* run
- *createKMeansJob()* : given the *Hadoop* job options, returns a job instance

3.4 KMeansMapper

the *KMeansMapper* extends the *Mapper* class and has:

- a *setup()* method that takes the configuration provided from the *Main* and sets up the initial centroids
- a *map()* methods (overridden from the *Mapper* class) that, given a *Point*, returns the *Point* itself together with the index of the nearest centroid in a *[key, value]* pair

3.5 KMeansReducer

the *KMeansReducer* extends the *Reducer* class and overrides:

- the *reduce()* function that, given a *key* and a *Iterable¡Point¿* linked to it, writes down to the context the *key* and the center of the provided points

this class is suitable to be used both as *Reducer* and *Combiner* because inside the *Point* class is store the number of the point used to create it .

3.6 What if a centroid is empty?

The algorithm is made to create an arbitrary number of centroids k. However, during the iterations, a centroid might not be associated with any point, resulting in an error. To reduce the probability of such an event, the starting centroids are chosen among the dataset points. This doesn't completely solve the problem, there are lots of suggested ways to deal with it, the one chosen in this implementation is to leave the *empty* cluster as it is; This solution, which isn't obviously optimal, makes the implementation of *K-means* predictable (other solution such as "re-set the centroid of the empty cluster randomly" are, by design, random and don't give us control to make predictable benchmarks).

3.7 Configuration Tweaks

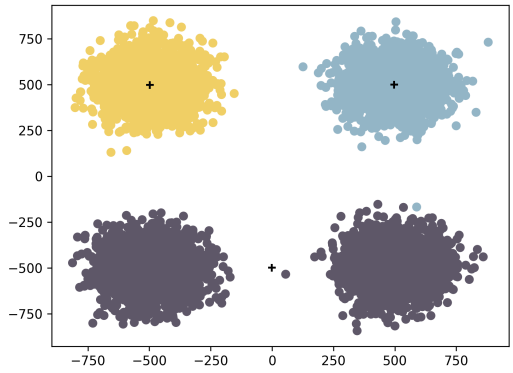
After setting up the cluster's machines with the provided configurations, the machines's allocated memory was not enough to work properly with our implementation and datasets. For this reason, the configurations were modified as shown below, increasing the *Value* fields inside the *yarn-site.xml* file.

```
<configuration>
        [...]
        <property>
                <name>yarn.nodemanager.resource.memory-mb</name>
                <value>1536</value>
        </property>
        <property>
                <name>yarn.scheduler.maximum-allocation-mb</name>
                <value>1536</value>
        </property>
        [...]
</configuration>
```

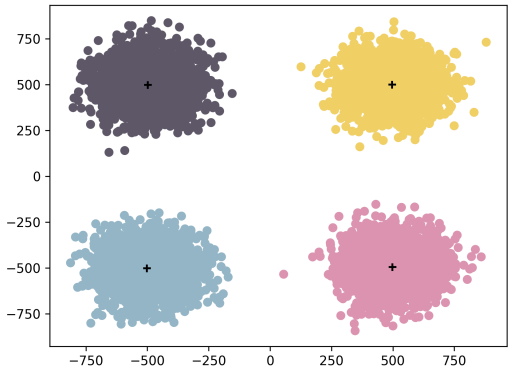
4 Correctness

For testing the algorithm, multiple datasets have been processed with different input parameters. To have a validation of the results, every test was first run with the *K-Means* function of the *Python* library: *SKLearn*; The datasets, in particular, are generated with an apposite *Python* code, *dataset_generator.py*, which calls the function *Makeblobs* with custom parameters; thanks to this code, the creation process is quick, easy to use, and can satisfy specific requirements.

The images below are the output of the test runs of the algorithm on 2 features datasets, showing the same results of *SKLearn* and proving the correctness of the *Hadoop* code.

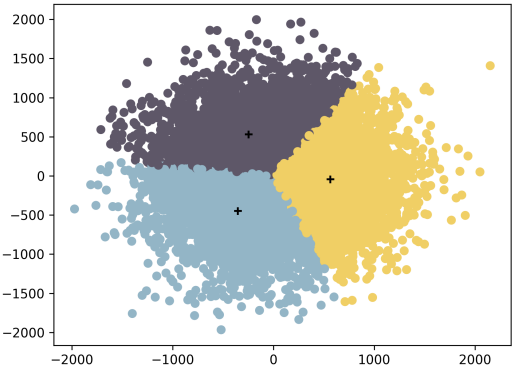


(a) $K = 3$

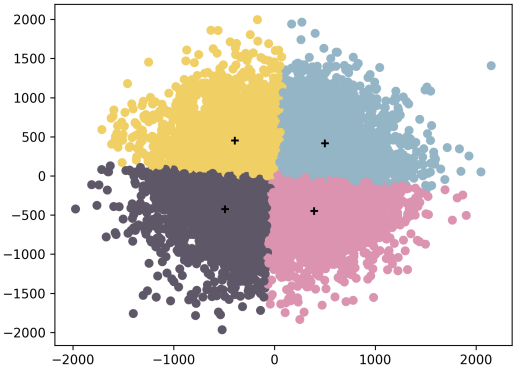


(b) $K = 4$

Figure 1: Dataset with distinct clusters with different values of K



(a) $K = 3$



(b) $K = 4$

Figure 2: Dataset with overlapping clusters with different values of K

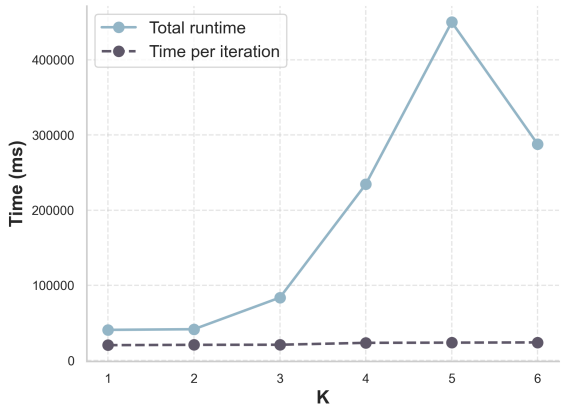
5 Performance Analysis

5.1 Increasing the number of clusters

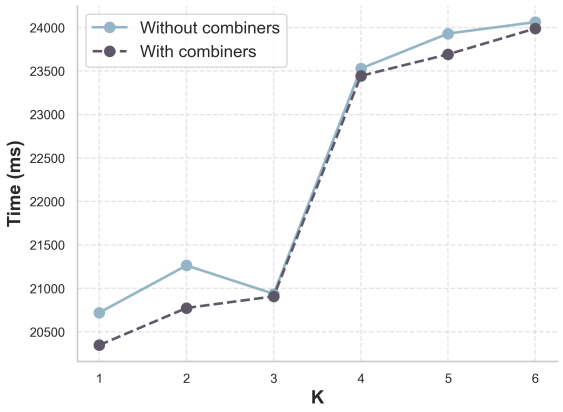
The algorithm has also been tested on a dataset regarding housing in Boston (it can be found at https://www.kaggle.com/code/prasadperera/the-boston-housing-dataset), to check how it performs on a real dataset. For this case study, *KMeans* has been run with different values of k. As shown in the graphs below, the way the value of k affects the runtime depends on how the datapoints cluster around the centroids; increasing the value of k (*graph a*), however, as expected the complexity of the algorithm increases, until the clusters become small enough to simplify it and make the runtimes smaller.

As shown in the *graph b*, the iteration runtimes increase with k and the number of keys, but the change is still small if compared to the change in the total runtime; this information proves that the total runtime depends mostly on the number of iterations rather than the single iteration's runtime, so depending on the value of k, the dataset, and the starting centroids.

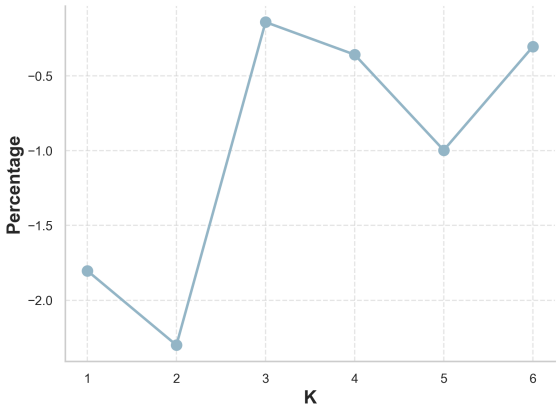
For what concerns the combiners, as shown in *graph c*, it is evident how adding them to the algorithm always decreases the runtimes, even if the decrease is in percentage really low.



(a) Boston Housing Dataset Runtimes



(b) Boston Housing Dataset Iteration Runtimes



(c) Percentage decrease of runtimes obtained by adding Combiners

Figure 3: Boston Housing Dataset Performances

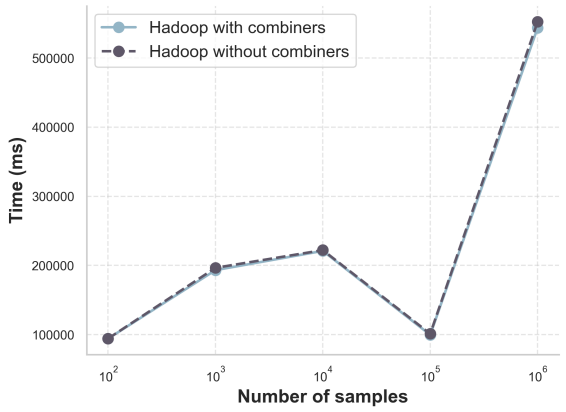
5.2 Datasets growing in samples

For the last performance analysis, the algorithm has been tested with datasets having the number of samples growing exponentially. In order to give the datasets a similar data distribution, 5 distinct centers have been created, with constant small standard deviation from them. The results of these runs have been analysed and the interesting graphs are shown below.

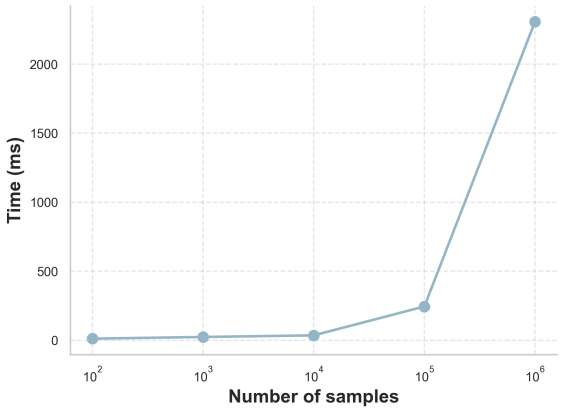
An important remark before analysing the results is that the accuracy of the runtimes during the runs was really low, having a high variation; since every run takes a lot of time to run, it was not possible to run the algorithm for long enough to get statistically relevant data.

In these graphs it is possible to analyse the differences in runtimes of both *SKLearn* implementation and the *Hadoop* with and without combiners when the dataset increases. In *graph a*, it is possible to see how the runtimes have a decrease, which is however justified by a low number of iterations for that specific number of elements; to prove this there is *graph c*, showing the iteration times and offering a better view of the increase in time with the increase of the dataset size.

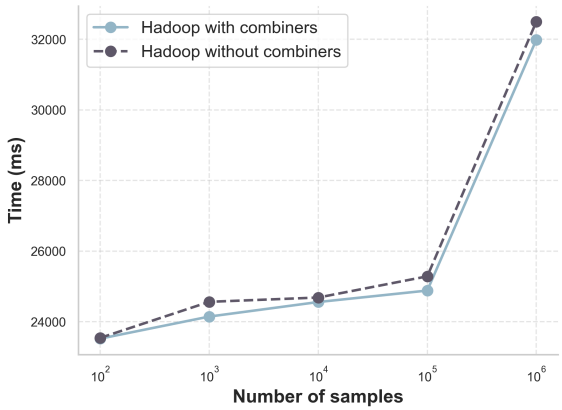
As also shown below, the runtimes with and without using combiners with *Hadoop* are really close, although the combiner version still have lower runtimes. The reason why the use of combiners makes little to no difference is probably because the code is executed on a little number of replicas; this is still relevant since it shows how this solution could scale with more respources available.



(a) Hadoop runtimes



(b) Python runtimes



(c) Hadoop iteration runtimes

Figure 4: Comparison of runtimes increasing number of samples

To have a better point of view, the variations of runtimes have been also analysed in percentage. To allow a better comparison, the runtimes are normalized to the iteration times, since *SKLearn* has optimisation to lower the number of iterations. As shown below, in fact, switching from Python to *Hadoop* has a percentual increase in runtimes of more than 10000%, which results in a critical increase in the runtimes. This percentage, however, decreases with the number of samples; this is another proof that shows how *Hadoop* with combiners could scale well.

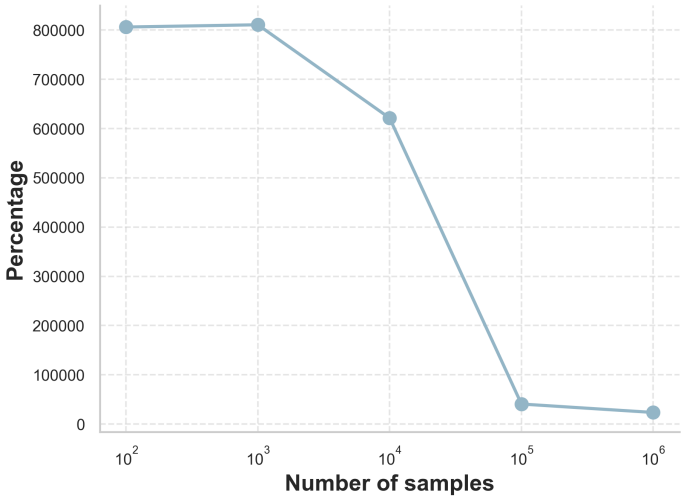


Figure 5: Iteration runtime's percentage increase using Hadoop with combiners instead of Python

6 Conclusions

The *Kmeans* algorithm has been analysed in depth and with different methods and technologies, studying the performances in every case. *Hadoop* has shown the potentialities of using a cluster of machines for computing data, even if *SKLearn* was still faster. However, it is evident how it is possible to have a well scaling solution using multiple cluster considering that *Hadoop* was launched inside virtual environment which had limited resources.