UNIVERSITÀ DI PISA

Master's degree in Artificial Intelligence and Data Engineering

Cloud Computing project documentation

K-MEANS ALGORITHM IN HADOOP

Domenico D'Orsi, Denny Meini, Matteo Razzai

A.Y. 2022/2023

GitHub repository: https://github.com/dominicofthebears/KMeans-mapReduce

Index

1. Introduction

In this work we exploit the functionalities of the Hadoop framework to obtain a MapReduce program which performs the Kmeans clustering algorithm.

The Kmeans algorithm is a centroid-based partitioning technique that uses the centroid of a cluster, Ci, to represent that cluster.

The algorithm proceeds as follows: first of all, it chooses randomly k objects from the the dataset D, each of this point represents a cluster centroid. Then, for the other objects, it calculates the Euclidian distance between itself and the cluster, assigning it to the closest centroid.

At the end of this phase, it will calculate the new centroids of all the resulted clusters, taking them as the cluster mean point, using them for the next iteration.

The iterations continue until the centroids assignment is stable, that means that the difference between the new centroids and the old ones is below a certain threshold.

2. Hadoop

Command Line Parameter

To launch the program, we used the following command line:

Es*:* *hadoop jar KMeans-mapReduce-1.0-SNAPSHOT.jar it.unipi.hadoop.kmeans.Kmeans 8 3features_10Krows.txt 50 0.01 outputK92*

The passed parameters are, in the order:

- k → the number of centroids (8 in the example above)
- input_path → the input dataset's path
- iteration → maximum number of iterations before the termination of the Kmeans algorithm
- threshold → distance value we want to get below to stop the algorithm.
- output_path → the output folder's path, where all the outputs of the reducer iterations are saved.

HDFS

The HDFS is used to store the dataset input file and then to store the reduce function output of each iteration of the MapReduce program.

For the output of the reducer, we decided to save all the iteration's output in a single file named "centroids_numberOfIteration_iteration.txt" and the final output inside the file "centroids_final.txt". All these files are saved inside the output path specified in the command line.

In this way the debugging process is easier and is possible to check the intermediate results of each iteration.

The write of the output is handled by the *writeFinalOutput* function, used for saving the output of each iteration of the MapReduce program, which is the output of the reducer.

Initial Centroids

We initialize the starting centroids using the *InizializeCentroids* function: with it we take k random points from the input dataset, using them as centroids for the first iteration of the Kmeans.

The core part of this function is visible in the following image, where we save the chosen line in the vector indexes, that contains the lines of the record that we want to use as centroid. When a random line is chosen, the following one will be chosen among the lines in the interval between the (chosen line +1) and the (totalLines – k +i), where **k** are the centroids that we want the k-means to perform and **i** the centroid already chosen randomly.

This process continues until we have k randomly chosen centroids.

```
while(i < k){
    line = reader.readLine(); //read each line of the dataset
    if(j == indexes[i]){
        centroids[i] = new DataPoint(line); //the i-th centroids take the value on that line
        if(i < k-1) {
            //the next value of the vector will contain a value on the interval between the value of the precedent
            //element of the vector and the last one possible - k + i
            indexes[i + 1] = random.nextInt( bound: (numLines - k + i) - indexes[i] + 1) + indexes[i] + 1;
        }
        i++;
    }
    j++;
}
```

Read Centroids

At the end of each job we have to read the new centroids in order to verify the stop condition, which calculates the distance between the new centroids and the old ones. To read the centroids coming from the output of the last reduce function executed, it is necessary to read the k files produced by each reduce task (set by us to the number of centroids). We make this on the *readCentroids* function, where we read the k centroids of that iteration, we save them on the Configuration setting object and we return them in a variable *newCentroids,* which will be used to be compared with the variable *centroids*, containing the ones of the previous iteration.

During the read of the centroids from the output file of the last job, it is possible that one of them had zero points associated and that the output from the last reducer may be k-1 file, related to the k-1 centroids obtained.

A key won't be emitted, and this will generate a EmptyCentroidException, handled in the main with a try-catch during the call of the readCentroids function. If the exception is raised, we reinitialized the centroids and the corresponding data structures and, at last, we restart the program from the job 1.

We had different options, such as terminating the execution or selecting a newly random centroid: we went for our solution since the first option would have been too oversimplistic while the second one would have inquinated the algorithm results.

3. Implementation

3.1. Mapper

In the Mapper class we have two functions: the setup function and the map function.

The setup function is used to retrieve the centroids of the last iteration and set the corresponding field with the retrieved ones. To perform this operation we use the Configuration class, where we can store key-value pairs of configuration settings.

The map function takes a DataPoint as input and computes the squared norm distance between this point and all the resulting centroids of the last iteration, assigning the closest label to the point, which represents the cluster to which this point belongs, that is, the cluster of the centroid which the input DataPoint is closest. So, the output of the Mapper will be the label as key and the DataPoint as value. In this way the reducer will receive all the points inside a single cluster.

```
Input:  key → LongWritable , value → Text
Output:  key → IntWritable , value → DataPoint

class MAPPER
    centroids: DataPoint[]
    method setup(context)
        k ← context.getConfiguration().get("k")
        conf ← context.getConfiguration()
        centroids ← new DataPoint[k]
        for all integer j ∈ [1,k] do
            s ← conf.get("centroid"+j)
            centroids[j] ← new DataPoint(s)

    method map(key, value, context)
        dataPoint ← new DataPoint(value.toString())
        minDistance ← dataPoint.squareNorm2Distance(centroids[0])
        closestLabel ← 0
        for all integer i ∈ [1,centroids.length] do
            distance ← dataPoint.squareNorm2Distance(centroids[i])
            if distance < minDistance do
                                minDistance ← distance
               ClosestLabel ←  i
        Emit(closestLabel, dataPoint)
```

```
class DataPoint
method squareNorm2Distance(p)
    sum ← 0
        for all integer i ∈ [1,p.getCoordinates().size()] do
        difference ← this.coordinates.get(i) - p.getCoordinates().get(i)
        sum ← sum + (difference*difference)
    return sum
```

3.2. Combiner

The Combiner function is used to optimize the MapReduce program shrinking the intermediate data from the mapper to the reducer. It takes the output of each mapper, operating on that.

In our case it performs a first cumulation of coordinates and weights of the DataPoints coming from the Mapper output. It takes all the key-value pairs from a single mapper and cumulate all points belonging to the same cluster into a single DataPoint object. Its output will be the same as that of the mapper, a label as key and a DataPoint as value. In this way, however, the total number of intermediate DataPoints will be fewer, and thus there will be fewer DataPoints as input to the reducer.

The reduce function of the Combiner is very similar to the one in the Reducer, with the difference that here we don't have to calculate the final centroid, but only to cumulate more points as possible inside a single point, to reduce the number of points as intermediate data before the Reducer.

```
Input:  key → LongWritable , value → DataPoint
Output:  key → IntWritable , value → DataPoint
```

```
class KMeansCombiner extends Reducer
method reduce(key,values,context)
    cumulator ← new DataPoint(values.iterator().next())
    for all dataPoint ∈ values do
        cumulator.cumulatePoints(values.iterator().next())
    Emit(key, cumulator)
```

3.3. Reducer

The reducer takes in input the key and a list of DataPoint, which contains a list of point already assigned to the centroid indicated by the key. Its goal is to obtain the final result, which are all the points present into the list of values cumulated. The first part of this task consisted into the cumulation of the point (in the same way we saw for the combiner), and finally it replaces all the cumulated coordinates with their average, in order to find the coordinates of the new centroid: these coordinates will be parsed into a Text (as like as the key) and finally written in output.

Below we have the pseudo-code of the reduce function:

```
Input: Key → IntWritable, Value → DataPoint
Output: Key → Text, Value → Text

CLASS Reducer
    METHOD reduce(key, values, context)
    finalResult ← new DataPoint(values.iterator().next())
    for all dataPoint ∈ values do
        finalResult.cumulatePoints(values.iterator().next())
    for all coordinate ∈ finalResult.getCoordinates() do
        coordinate ← coordinate/finalResult.getWeight()
    EMIT(key, finalResult)

 CLASS DataPoint
    METHOD cumulatePoints(point)
    for all integers ∈ [0, point.getCoordinates().getSize()] do
          this.coordinates[i] ← this.coordinates[i] + point.coordinates[i]
    this.weight ← this.weight + point.weight
```

3.4. Design choices

For what regards the “main” function, contained into the Kmeans class, the first thing which is done is to set the value of K into the configuration file, then we could initialize the centroid calling the “initializeCentroids” method. After that we saved the centroids into the configuration file as like as the values of threshold and max_iteration, that are used in the stop condition

Subsequently there’s a cycle in which we set the number of reducer (which is equal to K), the mapper, the combiner and the reducer, the cycle stops when we reach one of the stop conditions, that are:

- A value for the variation of the centroids lower than the threshold
- A number of executed iteration equal to max_iteration.

3.5. DataPoint

We used a class called “DataPoint” in order to represent a point in the dataset and the centroids as well. The class fields are the following ones:

```
18 usages
protected LinkedList<Float> coordinates;
8 usages
private int weight;
10 usages
private int numDimensions;
```

- **coordinates** is a LinkedList of Float that contains the coordinates of the point. We used a list so we could add an object for every coordinate, even if we don’t know before the number of coordinates and also since we had to move through subsequent items several times.
- **weight** is an int which has the goal of distinguish the normal points and the centroids, the normal point has 1 as value of weight, while the centroids have the number of points that are associated with their cluster.
- **numDimensions** is an int which purpose is to indicate the number of dimensions of the points.

There are 3 constructors in this class:

- One with no parameters, which creates a DataPoint with an empty LinkedList for the coordinates, 1 as weight and 0 as numDimensions

```
public DataPoint(){
        this.coordinates=new LinkedList<>();
        this.numDimensions=0;
        this.weight=1;
}
```

- One is a copy constructor, so it receives as a parameters another DataPoint and create an identical one, it’s used in both the combiner and the reducer, because they receive as input value a DataPoint

```
public DataPoint(DataPoint d){

    this.coordinates = new LinkedList<>();
    this.numDimensions = d.getNumDimensions();
    this.coordinates.addAll(d.coordinates);
    weight = d.getWeight();
    numDimensions = d.getNumDimensions();
}
```

- One takes a String as an input and parses it into a DataPoint, we used it in the setup (where we read the first centroids from the conf file) and in the mapper (because the value we receive in input is a Text)

```
public DataPoint(String s) {
    this.coordinates = new LinkedList<>();
    weight = 1;
    numDimensions = 0;
    for (String s2 : s.split( regex: ",")){
        this.coordinates.add(Float.parseFloat(s2));
        numDimensions += 1;
    }
}
```

Other two interesting methods are **squaredNorm2Distance**, which calculates the squared distance between the current point and a DataPoint that the method receives as a parameter, and **cumulatePoints**, which cumulates Datapoints' coordinates and weight (we use this method in the combiner and in the reducer in order to have a representation of the points assigned to a centroid).

The class implements the Writable class, in that way we could use the DataPoint type in input/output for mapper, combiner and reducer. In order to do this we had to implement the methods **readFields** and **write**, so that the class could be serialized.

The remaining methods are the get and set of the attributes and the toString, which parses a DataPoint into a String.

4. Description of the results

In this last section, we finally provide the results of various experiments and a comparison to check whether our version of the K-Means algorithm performed in a proper way, both from a required time and final error point of view. The comparison term has been the K-Means algorithm implementation that can be found in the "*SK-Learn*" python library, taken as state-of-the-art implementation.

We did not consider it for the time-consumption evaluation, since of course it takes just few seconds, but we wanted to check if the final results of our implementation were attendible considering the final centroids and error.

We also wanted to see how much difference the random centroid initialization causes to the results, so we run the SK-Learn algorithm both using the same starting centroids randomly chosen by our Hadoop implementation and using the optimal initialization performed by default in the python version

4.1. Datasets

In order to perform those experiments, we created several different datasets which varies in size and dimensionality.

In particular, we used the "*make_blobs*" method (still from SK-Learn library) to produce those dataset, where the selected number of features was **3, 5,** and **7** and for each dimensionality the selected dataset sizes were **1000 rows, 10000 rows** and **100000 rows.**

Moreover, we avoided to specify a number of centers to the method and we set the **cluster_std=5**: the combination of these two parameters allowed us to generate more realistic datasets to lead our experiments, otherwise they would have been already divided in a given number of well compacted and separeted clusters.

4.2. Test results

All the following experiments were perfomed with the following parameters:

- **Tolerance**: 0.01
- **Max iterations**: 50

For each of them, we report several values regarding our implementation and two values for comparison:

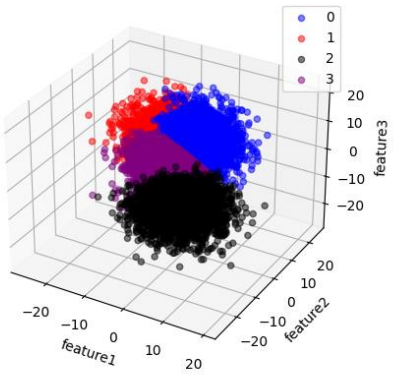
- **The *K* selected value**
- **Number of rows of the dataset**
- **Stop condition triggered**
- **Number of iterations**
- **Mean job time**
- **Total time**
- **Total error for our implementation**
- **Total error for the SK-Learn implementation (same starting centroids as our version)**
- **Total error for the SK-Learn implementation (centroids optimally initialized)**

Where by “error” we mean the norm 2 distance between each point and its assigned centroid.

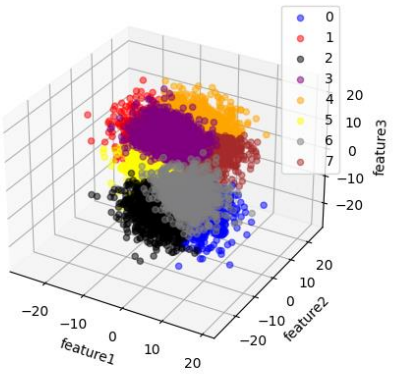
3 FEATURES

K-value	Rows	Stop condition	Iterations	Mean time (s)	Total time (s)	Hadoop error	SK-Learn error (same initialization)	SK-Learn error (optimal initialization)
4	1K	threshold	14	24,830	347,770	581274.910	582115.658	581547.477
4	10K	threshold	12	25,292	278,338	583017.373	583734.508	584375.771
4	100K	threshold	9	25,948	207,872	5854551.678	5861883.630	5851332.143
8	1K	threshold	15	27,707	332,611	411180.208	406033.836	403019.498
8	10K	threshold	46	27,450	1262,897	399890.385	409108.926	400709.676
8	100K	iterations	50	28,751	1437,855	403834.317	4180378.315	4060324.681

Hadoop result: 3 features - 4 centroids - 1K rows



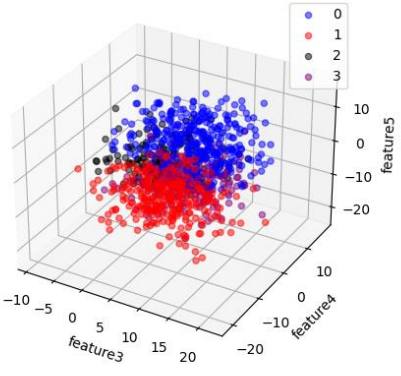
Hadoop result: 3 features - 8 centroids - 1K rows



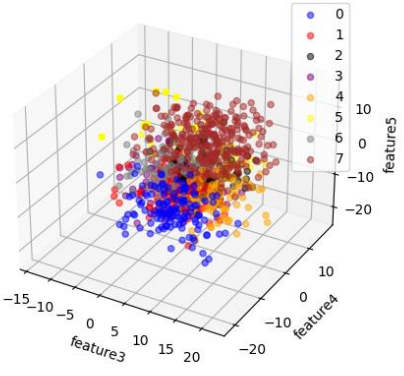
5 FEATURES

K-value	Rows	Stop condition	Iterations	Mean time (s)	Total time (s)	Hadoop error	SK-Learn error (same initialization)	SK-Learn error (optimal initialization)
4	1K	threshold	14	24,922	349,021	109403.359	109481.612	107978.370
4	10K	threshold	14	24,893	348,671	1120904.365	1121607.486	1113711.476
4	100K	threshold	12	25,378	304,918	11234152.980	11237637.151	11228208.320
8	1K	threshold	42	27,158	1140,760	89163.211	91785.410	88262.389
8	10K	threshold	46	27,732	1275,849	923570.781	929832.705	928041.237
8	100K	iterations	50	29,219	1461,231	9334636.450	9387332.508	9350069.959

Hadoop result: 5 features - 4 centroids - 1K rows

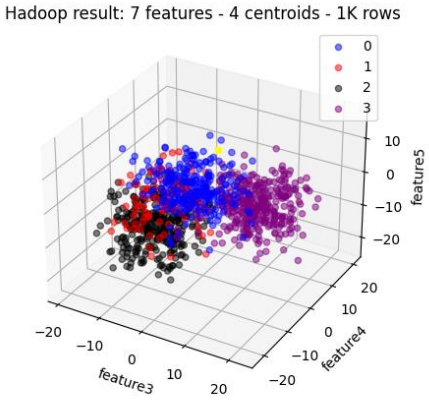


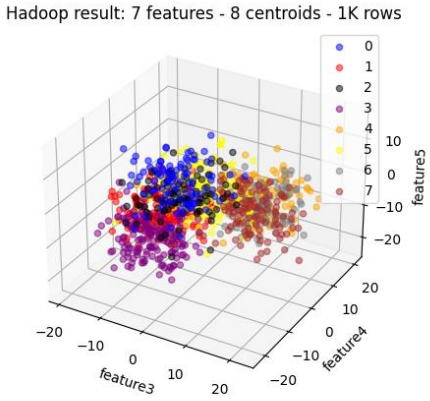
Hadoop result: 5 features - 8 centroids - 1K rows



7 FEATURES

K-value	Rows	Stop condition	Iterations	Mean time (s)	Total time (s)	Hadoop error	SK-Learn error (same initialization)	SK-Learn error (optimal initialization)
4	1K	threshold	13	24,282	315,789	167084.579	167413.886	166875.892
4	10K	threshold	10	24,722	247,384	1686809.606	1688103.889	1683081.503
4	100K	threshold	18	25,101	452,149	16828379.191	16851053.582	16839021.154
8	1K	threshold	16	26,820	429,227	143051.575	143274.687	143055.680
8	10K	iterations	50	27,551	1377,756	1509473.636	1517600.292	1492517.215
8	100K	iterations	50	28,674	1434,113	15011478.342	15127298.802	15047844.127





As we can observe from the results above, we got a good performance for the time required by jobs, in a range that goes from 24 to 29 seconds for the highest dimensionality-numerosity combination.

For what concerns the error, instead, we can see that the difference between the three version of the algorithm is really minimal, and sometimes our implementation also works better than the other two.

It's also interesting to notice that only few times the algorithm stops because of the number of iterations, so most of the times we get to a local minima and then the algorithm stops.