UNIVERSITÀ DI PISA

K-Means Implementation in Hadoop

-

Cloud Computing Project

Lorenzo Mazzei, Matteo Manni, Alice Petrillo

GitHub Repository: https://github.com/MatteoManni99/K-Means-in-Hadoop-Cloud-Computing-Project.git

1 Introduction

In this project the Hadoop framework was used to provide an implementation of the K-means algorithm clustering algorithm, meaning that the Map and Reduce approach was used with different datasets to test the correct functioning of the program.

2 Hadoop

Command Line Parameter

We passed five parameters through the command line upon program launch:

- `input`: the input path to the dataset file.
- `output`: the output path, meaning the output for the reducer.
- `k`: the number of centroids we want the k-means to perform. As said in the beginning, we tested this parameter with different values for each dataset to analyze the results produced
- `maxIteration`: numerical value, sets the maximum number of iterations the program should went through before stopping. This is one of the stopping criterions in case the algorithm doesn't converge.
- `threshold`: This is the other stopping criterion; it sets the threshold for the difference between the centroids in consecutive iterations. If all centroids do not change to much, we meet a local optimum of the problem and so we can stop with the iterations.

HDFS

We used the Hadoop filesystem to handle the output results of the reducers: Normally we would only be able to see the final result of the program, but we wanted to keep the middle computations of the program so that the behavior of the centroids could be observed over time and to keep track of the differences between the centroids during the consecutive iterations. This also allowed for a simpler debugging process since we could understand at which iteration and for which centroid the program didn't work.

We decided to create a directory for each iteration of the program, each of these directories contained k-number of files, where each file represented the computation done for one of the centroid.

Initial Centroids

We set as initial centroids k points by picking random records from the dataset file and store them in the configuration file. We actually tried to generate random points in the space in the first tests, but this led to a frequent error in the form of a NullPointerException. By checking the output files, we noticed that one of the centroid in the last iteration the algorithm got to was empty.

We initially thought that the problem was in the dataset because of it being randomly generated by us and therefore could have contained format errors in some of the records. Instead, we understood that the problem was in the Mapper: in some occasions it is possible that the mapper doesn't generate any pair of key-values for one of the centroids, this leads to the Reducer not being able to run and compute any output for that centroid and therefore not writing anything in its correspondent output file.

So, at the next iteration when all the centroids need to be retrieved from the files, the NullPointerException is generated, leading to the error. To solve this issue, we set the initial centroids by picking random points from the dataset file as said before, this guaranteed the existence of at least one key-value pair for every centroid.

Anyway, the same error occurs if a centroid remains without any point awarded. We handled these cases with a list of banned k. If no points are awarded to a centroid, then the number corresponding to that centroid is added to a list of banned centroids. Whenever there will be a cycle involving centroids, the iterations corresponding to the banned centroids will be skipped. For example, when we set the centroids, calculated in the previous iteration, into configuration it is impossible to set a centroid that does not exist, so the iteration is skipped.

3 Implementation

In this section the classes and methods used will be explained and the pseudo-code of each of them will be shown.

Mapper

`Input | Key → LongWritable , Value → Text`

`Output | Key → IntWritable , Value → Point`

The Map function takes in input one of the points of the dataset, then it computes the Euclidean distance from the point to each one of the centroids to detect the one the point is the nearest to. The Map function then emits as output key the id of the nearest centroid and as output value the point itself. The input value arrives to the Mapper as a Text type, meaning that we have to transform it first in a String type so that we can pass it to the constructor of the Point class, that will be described in detail in the following section.

Regarding the centroids, they are retrieved from the Configuration file as discussed previously: this is done in the `'setup'` function where also the number of centroids is initialized. We decided to handle the centroids though an array and not through a list since we know their dimensions. Here below is reported the pseudo code of the class.

```
PARAMETERS
centroids: Point[]
keyToReducer: IntWritable
k: int
bannedK: ArrayList<Integer>
```

SETUP METHOD
```
setup(context):
      conf ← context.getConfiguration()
      k ← conf.get(‘k’)
      bannedK ← conf.get(‘bannedK’)
      for i = 0 to k:
            if bannedK contains i, continue
            centroids[i] ← new Point(conf.get(‘"centroid” + i’))
```

MAP METHOD
```
map(key, value, context):
      valueToReducer ← new Point(value.toString())
      min_distance ← distance between: centroids[0], valueToReducer
      new_distance ← 0.0
      keyToReducer ← 0

      for i from 1 to k:
            if bannedK contains i, continue
            new_distance ← distance between: centroids[i], valueToReducer
            if new_distance < min_distance:
                  min_distance ← new_distance
                  keyToReducer ← i
      context.write(keyToReducer, valueToReducer)
```

Combiner

`Input | Key → IntWritable, Value → Point`

`Output | Key → IntWritable , Value → Point`

The combine function is used to optimize the quantity of data passed from the mappers to the reducers. It takes in input the key and a list of Points produced by a mapper. Then it computes the sum for that points on a new Point with all features initialized to 0. The combiner emits in output the key passed from the Mapper and the new summed Point. Takes in input the id of one centroid as the key and the list of Points associated to it as the value. The list of Points is actually handled as `Iterable<Point>`. Since we need to set the dimension of the partial centroid that will be emitted as output value, when we scan the list of points through an enhanced for, we retrieve that dimension by getting the one of the first Point of the list (this is done though an ‘if’ check so that we’ll avoid doing a redundant operation later).

Since we use the same class for Combiner and Reducer, we also perform an ‘if’ check on each of the Points of the list to see if some of them contained features on which the sum was already performed, if this is not the case then it means that the point has still to be processed by the Combiner which proceeds to perform the sum on the new Point (‘temporayCentroid’) that represents the centroid to emit in output.

Reducer

`Input | Key → IntWritable, Value → Point`

`Output | Key → IntWritable , Value → Point`

The reducer takes in input the key and a list of Points produced by the Combiner. The Combiner should emit in output the key passed from the Combiner and the new Centroid: the final centroid was already partially completed in the Combiner in the form of the summed Point, so the task of the Reducer is basically the same, just computing the weighted sum on the remaining features.

The Reducer uses the same class of the Combiner, so the only difference stands on the ‘if’ check on each of the Points of the list to see if some of them contained features on which the sum was already performed: if this was the case, we had to multiply again those features for the same coefficient they were divided for in the Combiner step. In order to perform this, we decided to store the value of the coefficient as a field of the Point class.

Here below is reported the pseudo code of the class that we exploit for both combiner and reducer:

REDUCE METHOD

```
reduce(key, values, context):
    tempSumCounter ← 0
    tempFeatures = []
    temporaryCentroid ← new Point()

    for point in values:
        if firstIteration:
            dimension ← point.getDimension()
            temporaryCentroid.setDimension(dimension)
            temporaryCentroid.setSumCounter(0)
            initialize tempFeatures with 0

        pointSumCounter = point.getSumCounter()
        if pointSumCounter > 1:
            multiply all point features by pointSumCounter

        tempSumCounter += pointSumCounter
        sum to all tempFeatures the point Features

    divide all tempFeatures by tempSumCounter

    temporaryCentroid.setSumCounter(tempSumCounter)
    temporaryCentroid.setFeatures(tempFeatures)

    context.write(key, temporaryCentroid)
```

Design Choices

The implementation of the Map and Reduce section is done in the ‘Kmeans’ class. The control and handling of the HDFS is done in the ‘main’ function of the class: we decided to create a different directory for each iteration of the algorithm, with the following format:

`outputPath/iteration`***`(iter)`***`/part-r-0000`***`(k)`***

where:

- ***`(iter)`*** represents the number of the iteration the program is currently in.
- ***`(k)`*** represents the id of the centroid stored in that path.

In the 'main' function we also set the centroids and the parameters we need in the Configuration file, like the number of reducers, that we decided to set as the same number of the centroids through the command `'setNumReduceTasks'`. In fact, it had no sense to do in another way, since in parallel they cannot run more than k reducers since there are no more than k keys. Even setting the number of reducers smaller than k would make little sense because in any case there would be no improvement in execution times. The Combiner and the Reducer are implemented as the same class so also the `'setCombinerClass'` and `'setReducerClass'` are set with that same class.

All these actions are done in a `while` cycle in which a new job instance is created every time and set accordingly.

The `while` cycle doesn't stop until the stop condition is reached: as a stop condition, we decided to use a simple number of max iterations, which corresponds to the parameter `max_iteration` we pass through the command line when the program is started.

Since at every iteration we have to retrieve the centroids from their respective files, we had to come up with a format that would have allowed us to easily get all the features of the centroid avoiding every other information like the id or the punctuation. For these reasons we decided to use the following format:

Id   ,feature1,feature2,feature3,…

Where 'id' is the centroid id and 'feature1,feature2,feature3…' are values for each feature of the centroid. The most important thing of this format is the comma before the first feature: in this way we can use the 'split' method for Strings to retrieve in a clean way all the values of the centroid.

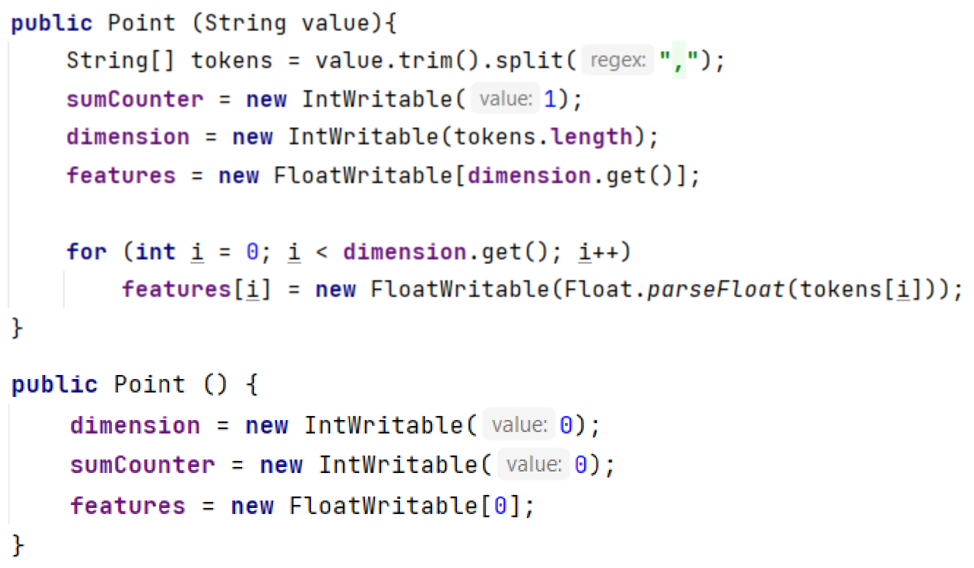
Point Class

This class was used to represent both the points of the datasets we passed in input to the program as well as the computed centroids. The fields for this class are the following ones:

```
private FloatWritable[] features;
private final IntWritable dimension;
private final IntWritable sumCounter;
```

- `FloatWritable features` → array of FloatWritable where the values of the features of the point are stored.
- `IntWritable dimension` → an IntWritable, represents the dimension of the point.
- `IntWritable sumCounter` → represents the coefficient used to perform multiplications and divisions on the features to compute the centroid.

The class also contains 2 constructors: one that takes in input a String that represents a point or a centroid and in which the method is used to the values of its features; the other constructor is used to create a new blank Point to work on from scratch.



```java
public Point (String value){
    String[] tokens = value.trim().split( regex: ",");
    sumCounter = new IntWritable( value: 1);
    dimension = new IntWritable(tokens.length);
    features = new FloatWritable[dimension.get()];

    for (int i = 0; i < dimension.get(); i++)
        features[i] = new FloatWritable(Float.parseFloat(tokens[i]));
}

public Point () {
    dimension = new IntWritable( value: 0);
    sumCounter = new IntWritable( value: 0);
    features = new FloatWritable[0];
}
```

The Point class implements the Writable class, this ensure that the class can be placed as type in input/output in the mapper, combiner, and reducer functions. In fact, we have implemented the two methods that allow the serialization of the class: `write` and `readFields`.

The values we pass for the serialization are all the fields of the class defined before, note that they are all writable types.

Utils Class

This class contains utility static methods to handle the files:

- `setInitialCentroid`: this method retrieves random points from the dataset file, those points will be set as the centroids for the first iteration of the program.

- `readOutputFile`: this method reads the centroid file and retrieves its information as a String on which the 'split' method is applied to return the values of the centroid.

4 Results

In the end we run our implementation with different datasets to compare the results and report the performances. In order to test the application, we used different datasets setting input parameters in this way: `maxIteration = 24, threshold = 0.005`

Datasets

The first dataset we used for this project was custom made by us through a python script. In it we utilized the `sklearn` library and the `make_blobs` method, in this way we were able to create a dataset easily. So, we started with one composed by 1000 records and 3 features to keep it simple, since the first goal was to check if the output of the K-means given by our reducer was the same as the one given by `sklearn` KMeans() method.

We then went on to test the program on larger and larger datasets, both in the number of records and in dimension. In the following picture all the datasets tested are shown:

```
#                 0,     1,      2,      3,      4,       5,       6,       7,       8
n_samples_  = [1000, 5000, 10000, 10000, 50000, 100000, 100000, 100000, 100000]
n_features_  = [  5,    5,      5,     10,      5,       5,      10,      15,       5]
centers_  = [     4,    4,      4,      8,      4,       4,       4,       4,       8]
```

All the datasets were saved as .txt files since each one of them was given as input to the program.

Performance

We saw that for each k-means iteration (a job execution) our implementation takes more or less 26 seconds. Obviously, things change at each restart even with the same dataset in input, this happens because the initial centroids are picked from the dataset randomly. Furthermore, in a distributed implementation of K-means it is impossible to reach each time the global optimal solution, this entails that sometimes the algorithm ends with a centroid set different from that provided by the sklearn's Kmeans. Here below are reported all the results in terms of iterations, threshold condition reached, global optimum founded, execution time (in ms), and cluster removed (banned K).

dataset0.txt – record: 1000 | centroid: 4 | dim: 5

iter.	thresh.	time	optimum	banned k
3	true	94198	true	none
9	true	251465	false	none
3	true	100643	true	none

dataset1.txt – record: 5000 | centroid: 4 | dim: 5

iter.	thresh.	time	optimum	banned k
4	true	121833	true	none
3	true	100476	true	none
3	true	76115	true	none

dataset2.txt – record: 10 000 | centroid: 4 | dim: 5

iter.	thresh.	time	optimum	banned k
3	true	95989	true	none
5	true	149892	true	none
24	true	624941	false	none

dataset3.txt – record: 10 000 | centroid: 8 | dim: 10

iter.	thresh.	time	optimum	banned k
24	false	700123	false	none
24	false	697644	false	none
24	false	693547	false	none

dataset4.txt record: 50 000 | centroid: 4 | dim: 5

iter.	thresh.	time	optimum	banned k
4	true	122088	true	none
12	true	332492	false	none
3	true	99634	true	none

dataset5.txt – record: 100 000 | centroid: 4 | dim: 5

iter.	thresh.	time	optimum	banned k
2	true	78901	false	1
3	true	100091	true	none
16	false	441955	false	none
4	true	137493	true	none

dataset6.txt – record: 100 000 | centroid: 4 | dim: 10

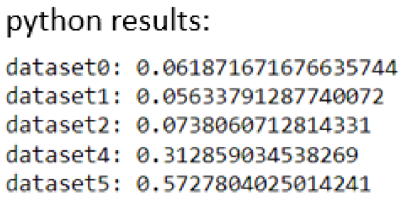
iter.	thresh.	time	optimum	banned k
1	true	61078	true	none
21	true	588331	false	none

dataset7.txt – record: 100 000 | centroid: 4 | dim: 15

iter.	thresh.	time	optimum	banned k
2	true	94138	true	none
24	false	686363	false	1

dataset8.txt – record: 100 000 | centroid: 8 | dim: 5

iter.	thresh.	time	optimum	banned k
5	true	172501	true	none
24	false	720206	false	none
24	false	699817	false	none



python results:

```
dataset0: 0.061871671676635744
dataset1: 0.05633791287740072
dataset2: 0.0738060712814331
dataset4: 0.312859034538269
dataset5: 0.5727804025014241
```

From these results we see that for datasets with 8 centroids the algorithm hardly converges before 24 iterations. In other cases, less than 10 iterations are sufficient. Note that in the tables shown the number corresponding to 'iter.' is the number that identifies it, but we must also consider iteration number 0, so the actual number of total iterations is the one reported +1.

We also reported the time it takes for the sklearn algorithm to do the same job with datasets: 0, 1, 2, 4 and 5, which have the same size and same number of centroids and increase in number of records. We repeated the performance evaluation 30 times for each dataset (initial centroids are chosen randomly) and averaged them. The increase from 1000 to 10·000 records in terms of time is not excessive, things change with 50·000 and 100·000. In fact, from dataset0 to dataset5 there is a 793.14 percent increase in computation time.

Doing the average of the time it takes Hadoop to complete an iteration on the same dataset we get these results:

- dataset0 → 24·794 ms
- dataset1 → 24·868 ms
- dataset2 → 24·880 ms
- dataset4 → 25·191 ms
- dataset5 → 26·153 ms

In the case of the Hadoop implementation, we have a percentage change in the computation time for one iteration of the k means of + 5.48%. Much more scalable than its undistributed counterpart.