Spark Project : Optimizing Kmeans algorithm

Authors :

Amine El Iraki
Hadrien Mariaccia
Hippolyte Mayard

2 mars 2020

Table des matières

1 Introduction

In this project, we propose an optimization of the naive implementation of the K-means algorithm using pyspark. From a initial version of the algorithm, we have been implementing a two-fold optimization. On the one hand, we tried by optimizing the algorithm theoretically, on the other hand by finding better ways to implement actions and transformations using Spark.

2 The data sets

In order to test our optimized implementation of the K-means algorithm, we have tested it on different data sets. In this section, we present these data sets. We tried to select data sets that differ significantly in scale (number of observation and number of features) and in the number of clusters.

2.1 Data set 1 : Iris data set

We have been using first the standard Iris data set which is composed of 150 observations of 4 features and 3 classes.
In the following lines, we present a sample of the data set and its structure.

5.7,2.8,4.1,1.3,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica

2.2 Data set 2 : Balance Scale Data Set

Once we succeeded in optimizing the standard K-means algorithm, we decided to check the performances of the code on a bigger data set, with more observation. In this matter, we used the Balance Scale Data Set (source : http ://archive.ics.uci.edu/ml/datasets/balance+scale), which has exactly the same structure as the previous Iris data set (4 features and 3 classes) but has 625 observations.
A sample of the data set is given below :

L,1,4,2,1
B,1,4,2,2
R,1,4,2,3
R,1,4,2,4

2.3 Data set 3 : load digits

The third data set that we used is the load-digits data set from sklearn.datasets. This data set is composed of images embedded as vectors (8*8 pixel images have been reshaped in vectors of size 64). Hence, these vectors represent a digit beteween 0 and 9. In a nutshell, this data set contains 1797 observations of 64 features that are divided into 10 classes. This data is a good way to check the scalability of our code.

A sample of the data set is given below :

[0., 0., 5., 13., 9., 1., 0., 0., 0., 0., 13., 15., 10., 15., 5., 0., 0., 3., 15., 2., 0., 11., 8., 0., 0., 4., 12., 0., 0., 8., 8., 0., 0., 5., 8., 0., 0., 9., 8., 0., 0., 4., 11., 0., 1., 12., 7., 0., 0., 2., 14., 5., 10., 12., 0., 0., 0., 0., 6., 13., 10., 0., 0., 0., 0]
[0., 0., 0., 12., 13., 5., 0., 0., 0., 0., 11., 16., 9., 0., 0., 0., 0., 3., 15., 16., 6., 0., 0., 0., 7., 15., 16., 16., 2., 0., 0., 0., 0., 1., 16., 16., 3., 0., 0., 0., 0., 1., 16., 16., 6., 0., 0., 0., 0., 1., 16., 16., 6., 0., 0., 0., 0., 0., 11., 16., 10., 0., 0., 1]

The last element of each list represent class (the digit associated to each vector).

2.4 Data set 4 : Breast Cancer Wisconsin

source : https ://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

The last data set we used is the Breast Cancer Wisconsin data set composed of 699 observations of 10 medical features and 2 classes (the patient has a cancer or not). We used this data set to check whether the performances of our optimized algorithm would be better with only 2 clusters (K=2).
A sample of the data set is given below (2 indicates a benign cell and 4 a malignant cell) :

4,1,1,3,2,1,3,1,1,2
8,10,10,8,7,10,9,7,1,4
1,1,1,1,2,10,3,1,1,2

We presented the data sets that we have been using. These sets have been chosen carefully accordingly to there characteristics. Indeed, in this project, we tried to optimize the running of the K-means algorithm using spark. But the optimization can be twofold : we can either optimize the running time (e.g. memory allocation) or the accuracy of the algorithm. This is why we have been selecting data with different features, number of observations or number of clusters.

3 RDD

3.1 Optimizing centroids initialization

The usual k-means algorithm initializes centroids randomly, what can sometimes lead to poor clustering results. An alternative have been proposed to initialize the centroids according to their respective distance from the centroids already chosen, by using pounded probabilities : this algorithm is called K-means++. We describe the initialization bellow :

- a) We choose one centroid uniformly at random among the data points
- b) We compute distances d_i between the last centroid choosen and each data points i with $1 < i < n$.

— c) We compute $\frac{d_i}{\sum_{i=1}^{n} d_i^2}$ for each point i so that we can simulate a probability distribution over the points depending on their distance to the closest centroid.

— d) We draw uniformly a number between 0 and 1 and we sum over each $\frac{d_i}{\sum_{i=1}^{n} d_i^2}$ until the sum is over the number we draw before. We chose as a new centroid the last point from which weighted value was added to the sum and made it over the point we draw before.

— e) We go back to b) until we initialize all centroids.

Here is the code :

```
def choice(p):
    #This function allow to choose uniformly an index from a list of pounded probabilit
    random = np.random.random()
    r = 0.0
    for idx in range(len(p)):
        r = r + p[idx]
        if r > random:
            return idx
    assert(False)

def new_dist(vec, dist, k):
    new = norm(vec - centers[k], axis=0)**2
    return np.min([dist, new], axis=0)

def centroide_init(data, K) :
  n_data = data.count()
  shape = np.shape(data.take(1)[0][1][0])[0]
  centers = np.zeros((K, shape))

  new_data = (data
    .map(lambda x : x[1])
    .map(lambda p: (p, [np.inf])) \
    .cache())

  local_data = (data
            .map(lambda x : x[1][0])
            .collect())

  rand = np.random.randint(0, len(local_data))
  sample = local_data[rand]
  centers[0] = sample


  for idx in range(K-1):

    d = (new_data.map(lambda x : (x[0][1],x[0][0],new_dist(x[0][0],x[1],idx)))\
          .cache())
    d1 = d.map(lambda x : ((x[0],x[1]),x[2])).map(lambda x: (1,x[1]))
    d2 = d1.reduceByKey(lambda x,y: np.sum([x,y], axis=0))
    sum_Distance = d2.collect()[0][1]


    prob = (d1
          .map(lambda x : np.divide(x[1],sum_Distance))\
          .collect())
    #prob = np.reshape(prob,(len(local_data), RUNS))


    data_id = choice(prob)
    sample = local_data[data_id]
    centers[idx+1] = sample


  return centers
```

We implemented this other method of initialization and we were expecting to observe a significant decrease of the mean of number of iterations before convergence. But unfortunately, this number stayed the same. We can easily explain this with the small size of the dataset. In fact, Even with random initialization, the mean number of iterations is low (6). So it's hard to do better.

3.2 Optimizing functions

3.2.1 computeDistance

Instead of hard coding the distance computing. We use the euclidian function from scipy.distance library. This function is faster because optimized.

```
def computeDistance_opti(x,y):
    return distance.euclidean(x, y)
```

3.2.2 closestCluster

Instead of hardcoding the function and pass over each element of the list of distances, we used the min function from python which gave better computation times.

```
def closestCluster_opti(dist_list):
    """
    dist_list : a list distances to each centroid for a given point
    returns : the closest cluster and the distance to it
    """
    best = min(dist_list, key = lambda t: t[1])
    return (best[0], best[1])
```

3.3 Optimizing memory

3.3.1 caching RDDs

As we seen it in course, Spark is lazy. It means that when you define RDD and do map operations, nothing is calculated. Calculations are only operating when we compute a collect() or a count() on the RDD.

Since the Kmeans algorithm is executing a while loop until stop condition is reached. We observed that each iteration execution time is the double of the previous iteration execution time. As if at each iteration, we aren't only executing calculations for this iteration but also of all for previous ones.

In order to solve this problem, we had the idea to cache each RDD used in the Kmeans algorithm. But in the same manner than with map operations, when we cache a RDD, it's only when a collect() or count() is first called on this RDD that the RDD is effectively cached. The problem in the Kmeans algorithm is that a lot of RDDs are used successively and no count() or collect() is called before the end of each iteration. So we unpersist() those after their utilisation in order to free the memory for the next iteration.

3.3.2 Managing join operations

First modification

In the code provided, we understood that the computation time was very high because of the use of cartesian and join operations. In fact, in this code, the dataset is partitioned before the call of Kmeans function; Then in the Kmeans function, centroids are partitioned on their side. Then we compute a cartesian operation between centroids and data. This computation is naturally very long because it computes a join between RDDs in which respective keys are on different partitions. Moreover, at each iteration, we're redoing a cartesian product with the new centroids causing an exponentially increasing computation time between dependant transformations on different partitions presents from the very beginning of the execution.

In order to solve this problem, we had the idea to broadcast centroids and then map a function which takes as argument the centroids and which computes distance between each point and each centroid. This is saving a lot of time because we get rid of the cartesian product and we get rid of the groupByKey.

Here is the code :

```
centroides_par = sc.parallelize(data.takeSample('withoutReplacment',nb_clusters))\
                .zipWithIndex()\
                .map(lambda x: (x[1],x[0][1][:-1]))
centroides = sc.broadcast(centroides_par.collect())
centroides = centroides.value
```

And the function we mapped :

```
def join_dist(x, centroides):
    dist = []
    for i in range(len(centroides)):
        dist.append([centroides[i][0], computeDistance(centroides[i][1], x[1][:4])])
    return dist
```

And then in the while loop :

```
dist_list = data.map(lambda x: (x[0], join_dist(x, centroides)))
```

Second modification

We figured out that in the following lines of code, we could avoid the use of a join operator. Indeed, each execution of a join operation is highly costly in memory, hence, in time. In the following code, we figured out, that the join(data) could have been avoided, by mapping it previously and keeping it in memory.

```
    dist_list = data.map(lambda x: (x[0], join_dist(x, centroides_list)))

    min_dist = dist_list.mapValues(closestCluster)
    min_dist.cache()

    assignment = min_dist.join(data)
```

The operation assignment = mindist.join(data) could be optimize by using the data.map in line 1. In this matter, we optimized these lines by using the first data.map and keeping data in memory. The following code is an alternative to the join which is very costly in memory and time.

```
    dist_list = data.map(lambda x: (x[1],(x[0], join_dist(x, centroides_list))))

    min_dist = dist_list.mapValues(lambda x : (x[0],(closestCluster(x[1]))))
    min_dist.cache()

    assignment = min_dist.map(lambda x : (x[1],x[0]))
```

Hence, this two modification have been added to the code and we compared both of the execution times. The results are given in the next section.

3.4 Stop condition

In the code provided, the k-means algorithm stops if either the number of steps is over 100 or if each point belong to the same cluster as the previous step. The switch variable is heavy to calculate because it computes a join between all points.

We found smarter to change this condition to a lighter one in term of computation but with the same result. In order to do so, instead of computing the number of points which have changed cluster, we compute the number of centroids which have been updated. We're doing that by just comparing two lists, the one with the last iteration centroids and the one with current iteration centroids.

This modification of stop condition dropped significantly the computation time.

```
"""
centroides_list = centroids from previous step
list_centroidesCluster = centroids from current step
"""
set_list_centroidesCluster = set(tuple(list_centroidesCluster))
set_centroides_list = set(tuple(centroides_list))
if set_centroides_list == set_list_centroidesCluster:
    switch = False
```

Therefore, the stop condition becomes :

```
if switch == False or number_of_steps == 100:
    clusteringDone = True
```

3.5 Results

In this section, we compare the results for each data set. We ran each of our modification 500 times for the different modifications we added to the initial model. We compared different indicators for the performance of the code we have optimized. The following results compare the error, the running time and the number of steps for the different data sets.

1st Dataset

	Mean of Error	Mean of Number of Steps	Running Time
Modification 1	0.0676	5.934	5.938
Modification 1 + initialisation Kmeans++	0.0668	5.736	6.097
Modification 2	0.0669	5.792	3.243
Modification 2 + initialisation Kmeans++	0.0670	5.832	3.254
initial code	0.0681	7.232	244.847

2nd Dataset

	Mean of Error	Mean of Number of Steps	Running Time
Modification 1	0.0604	12.714	11.476
Modification 1 + initialisation Kmeans++	0.0604	13.824	12.749
Modification 2	0.060	13.182	6.217
Modification 2 + initialisation Kmeans++	0.060	13.178	6.225
initial code	0.0609	16.445	367.453

3rd Dataset

	Mean of Error	Mean of Number of Steps	Running Time
Modification 1	0.0298	11.5	10.641
Modification 1 + initialisation Kmeans++	0.0298	11.37	12.587
Modification 2	0.0299	11.43	7.728
Modification 2 + initialisation Kmeans++	0.0298	11.44	7.732
initial code	0.0302	13.869	654.021

4th Dataset

	Mean of Error	Mean of Number of Steps	Running Time
Modification 1	0.066	5.728	5.739
Modification 1 + initialisation Kmeans++	0.065	5.242	5.520
Modification 2	0.0653	5.222	2.858
Modification 2 + initialisation Kmeans++	0.0653	5.26	2.879
initial code	0.0622	7.923	260.341

4 Conclusion

To conclude, the optimization that we implemented on the K-means algorithm in Pyspark are twofold. On the one hand, we succeeded in optimizing the running time for each data set. The tables above show the considerable decrease of the running time. For instance, in the case of the Iris data set, the running time of the initial K-means algorithm is around 245 seconds in mean, while after optimization the clusters are computed after around 3 seconds in mean. Hence, in terms of running time, it is a non negligible optimization.
Our results have been tested on data sets with different shapes. Whatever the dimension of the data sets, the number of features or the number of clusters, the optimizations we have done increase the speed of the algorithm in terms of running time.
On the other hand, the implementation of the K-means++, with a probabilistic initialization of the centroids, give better results in terms of number of steps. However, we do not find any significant difference regarding the error, that is, the accuracy of the algorithm. There are two hints regarding these results. We have been using very classical data sets. The use of a more difficult data set may lead to a optimization in terms of error and accuracy.

Moreover, we would say that this project was very challenging because we had to understand deeply how spark is built to find optimizations. We spent a lot of times on optimizations of RDDs and we understood at the very end what was essential to actually optimize the algorithm : Limit joins operations, caching RDDs that are called several times and unpersist when we can.
The spark User Interface has been very helpful in this matter.
Finally, it would have been interested to compare the RDD implementation with Dataframes implementation and also compare the implementation in Python with the implementation in Scala, as we know that Spark is optimal with Scala.