Scalable KMeans in Python

YanyuLiu

1 Background

The paper I used is skalable K-means, which introduce a parallelizable initialization algorithm, k-means parallel for k-means clustering.

K-means is a popular method to separate data into k groups to achieve the minimum distance between each point and its cluster center. Two major theoretic (and practice) downsides of K-means are that the final result can be very bad compared to the global optimal and in the worst case running time can be exponential. K-means++ is designed to improve the performance of K-means though a better initialization approach.

K-means ++, an algorithm to choose the initial values of k-means clustering, has been proved that the initial values generated by this algorithm is close to global optimum. It is guaranteed to find a solution that is O(log k) competitive to the optimal k-means solution. However, it's not suitable for massive data due to its inherent sequential essence. We have to pass k times over the whole data set, which will dramatically slow down the speed to even when k is large in a data set.

Therefore, the author come up with a updated initialization algorithm, k-means parallel, to address this problem, which only needs logarithmic number of passes. (In fact, in practice, the number of rounds can be very small if we want to get more points in one iteration.) This really interests me, because k-means clustering is really widely used in the data analysis. It's meaningful to figure out how to apply this to large-scale data sets which are increasingly prevalent. Besides, the algorithm is parallelizable, so that I can explore more and learn more about parallelization in python programming.

1.1 Algorithm (Pseudocode)

1.1.1 k-means

Let X=$x_1, x_2, ...x_n$ be the set of points in d-dimensional Euclidean space, and k is the number of clusters we will divide X into. This starts with randomly choosing k points from X as initial values of centers $c_1, ..., c_k$. In each iteration, each point x_i is assigned to the closest cluster by calculating $argmin_j d(x_i, c_j)$. The calculate the new centroids of the observations in the new clusters as the k centers for the next iteration.

$$centroid(X) = \frac{1}{|X|}\sum_{x \in X} x$$

The iteration is repeated until a stable set of centers is obtained.

1.1.2 k-means parallel

```
1. C $<-$ smaple a point uniformly at random from X
2. $\psi <- \phi(C)$
3. for O($\log\phi$) times do:
4.     $C_1$ $<-$ sample each point x $\in$ X independently with probability $p_x = \frac{l \cdot d^2(x,c)}{\phi_x(c)}$
5.     C $<-$ C $\cup$ $C_1$
5. end for
7. For x $\in$ C, set $w_x$ to be the number of points in X closer to x than any other point in C
8. Recluster the weighted points in C into K clusters
```

1.1.3 Random

It means get k initual points randomly, each points in the data has the same probability to be chosen. We can use random.choice to get that.

2 Implementation

2.1 dataset

GAUSSMIXTURE, which is synthetic. I first sample k centers from a 10-dimensional Gaussian distribution with mean I_{10} and variance RI_{10}. Then add points from Gaussian distributions of unit variance around centers. The sample size would be 10000. To make the dataset more flexible, I define a function to generate the data, and we can change k and R to see if the result and conclusion is robust.

The codes can be seen in [Kmeans-parallel.ipynb] or [Kmeans-parallel.html].

2.2 Important functions

I designed Cost, weight and Kmeanspar to respectively calculate the cost and weight of two data sets and Kmeans clustering initialized by Kmeans parallel. I also designed two functions Random and Kmeansplus to do the Kmeans clustering initialized by Kmeans++ or Random.

The codes can be seen in [Kmeans-parallel.ipynb] or [Kmeans-parallel.html]

3 Testing

I mainly tested three functions: Cost, weight and kmeanspar (initialization part) and use 18 tests to make sure they are robust and get the right result.

I tested Cost and weight in different situation: inputs as integer arrays, input arrays with different dimensions (in case data set/center set only contain one point), some known cases to compare the results and the check to see if it get non-negative results.

I also checked if kmeanspar really return k centers, the centers are really in the data set, some known cases and if kmeanspar will recognize when number of iteration (r) times expected number of centers in each iteration (l) is smaller than k (i.e. we can't garantee to get more than k points from the for loop in kmeanspar). For the kmeanspar, I cut off the KMeans clustering part for testing because I only care about how the initialization part works so I can be more focus. KMeans is a existing command which doesn't need to be tested.

Similarly, I did not test the Random and Kmeansplus functions, because they two basically are based on KMeans command from sklearn module.

The codes are in [tests] folder.

4 Time Profile

The dataset will have 50 centers which are generated by $MVN(0, R \times I_{10})$. Expected number of generated centers in each iteration is 0.3k, I will run 5 rounds to get enough centers in kmeans parallel.

4.1 timeit and prun

I use timeit and prun for the profiling. For the first time, the running time of kmeans parallel was around 8s while kmeans++ and random was around 1-2s, which was a really bad result. By using prun, I found that weight function spent a long time and it is due to the two nested for loops. So I changed the algorithm and got $weight_{v2}$ (also the existing weight function). The running time of kmeans parallel has been improved to around 3s.

4.2 first version of Kmeans parallel

The first version of weight is defined as below, and to see the improvement by $weight_{v2}$ (i.e. weight), I also show the timeit of first version.

4.3 results and comparison

The codes and results can be seen in [Kmeans-parallel.ipynb] or [Kmeans-parallel.html]

4.4 Summary and Optimization Strategies

Even though I made some changes in the weight function and got good results (8s -> 3s), the running time (including Kmeans cluster) of Kmeans parallel was still slower than Kmeans++ using the existing module.

There are two ways I can do to do the optimization: one is using other language (C or Cython) and another one is to parallelize the Kmeans parallel.

5 Optimization - Cython

5.1 functions for Cython

The codes can be seen in [Kmeans-parallel.ipynb] or [Kmeans-parallel.html]

5.2 time profiling

The codes and results can be seen in [Kmeans-parallel.ipynb] or [Kmeans-parallel.html]

5.3 Summary

Unfortunately, the Cython seems worse. The Cython profiling shows that in fact the functions (Cost and weight) really got some improvement, but the total time for method 'reduce' of 'numpy.ufunc' objects is too long. It seems like the parallelization should be adopted and get better result. For futher developement, a Parallelized k-means in Cython should be a better choice.

6 parallelization

6.1 codes for Multi core

The codes and results can be seen in [Kmeans-parallel.ipynb] or [Kmeans-parallel.html]

6.2 time profiling

The codes and results can be seen in [Kmeans-parallel.ipynb] or [Kmeans-parallel.html]

6.3 Summary

I tried to use MapReduce and Mr. job for it but it seemed it's hard to use Mr. job in this case, so I used Multi Core. I only used two cores to run the parallized version and it got improved (from 3.7s to 1.7s)! The running time for method 'reduce' of 'numpy.ufunc' objects becomes 0.044.

7 Application Analysis

Kmeans++'s inherently sequential nature makes the biggest bottleneck especially when it's used in massive data. Its total running time is O(nkd), which is the same as that of a single Lloyds iteration. Kmeans——, instead, is better to be used for parallelization. It samples O(k) points in each round and repeat the process for approximately O(log n) rounds. And O(log n) iterations is not necessary and after as little as five rounds, the solution of k-means—— is consistently as good or better than that found by any other method.

A better way to compare them is to write down K-means++ and random by myself without using the argument in KMeans command. The functions are shown below. Besides, a parallelized Kmeans—— in Cython should performs better than others in practice, which is a good direction for future development.

However, Kmeans++ is more suitable when the data size is not that big. One of the reason is that K-means—— needs two steps: firstly we need get more than K potential centers and secondly we use K-means++ to get the final k centers. These steps apparently need more than compared to a single K-means++ for a small data set. Another drawback of Kmeans—— is that it's unstable theoretically because we can't garantee to get more than K centers in the first step. Also, the performance of Kmeans—— depends on l (expected number of generated centers in each round) and r (number of rounds). There is a tradeoff between running time and clustering cost deduction, and we might need to try several times to get what we want, which is time-consuming for a massive data set.