Scalable K-means++ Python implementation

Yuhao Liang

April 30, 2015

1 Background

So far, k-means still remains one of the most popular data processing algorithms. It is a widely used technique for statistical data analysis in many fields, such as machine learning, pattern recognition, image analysis and bioinformatics. However, general k-means algorithm with random initialization is not a good clustering algorithm in terms of efficiency and quality, which means it needs a long time to converge when the data set is large and it may just converge to the local optimum. In order to improve the quality, we need to improve the initialization part of the k-means algorithm first, selecting the right centers to do clustering. Recently people have proposed k-means++ initialization algorithm, obtaining the initial centers which can be provably close to the optimal solution, largely improves the quality of k-means algorithm but the problem of inefficiency is still unsolved. Now, there is a new algorithm, k-means——, obtaining a nearly optimal solution after a logarithmic number of passes.

Basically, k-means—— is based on k-means++ and the largest difference between these two algorithm is the initialization part of the algorithm. Since the initialization of k-means++ is deterministic (the previous choices that affect which points are choosed in the current solution), it is nearly impossible to use parallelized computation to improve efficiency. Instead, k-means—— algorithm samples each point independently in each round and repeat the process for approximately O(log ϕ) rounds, which can be easily implemented by means of parallel computation. Besides, ϕ is the cluster cost of initial randomly picked center, which can be viewed as sum of the distances between initial center and other points.

2 Implementation

k-means—— is a parallel version for initializing the centers of k-means clustering method. This algorithm is somehow intuitively similar to k-means++ while the main difference is the usage of an oversampling factor $l = \Omega(k)$, some linear function of k. In the first step, we sample a point C uniformly at random from data set as an initial center and compute the cost of this clustering center $\psi = \phi_X(C)$, where

$$\phi_X(C) = \sum_{x \in X} d^2(x, C) = \sum_{x \in X} min_{c_i \in C} ||x - c_i||^2 \tag{1}$$

Then we run a logφ iterations for loop, where in each iteration, we samples each point x in data set X with probability $\frac{ld^2(x,C)}{\phi_X(C)}$, given current cluster center set C. We update C with adding the sampled points and update the cluster cost quantity $\phi_X(C)$. Then we run next iteration until the for loop is completed. According to the sampling probability of each data points in each iteration in for loop, the expected number of points sampled in each iteration is l. Thus, finally the expected number of points we sampled is $l\log\psi$, which is expected to be more than k. Therefore we have to reduce the number of centers and we assigns weights to the points in C according to number of points in data set X which the center are closer to them than any other centers.

$$\omega(c_i) = \frac{\sum_{x\in X} 1_{(d^2(x,c_i)<d^2(x,C/\{c_i\}))}}{|X|} \tag{2}$$

With the weight, we can reclusters these potential cluster centers to obtain k centers. In this step we combine the idea of k-means ++ with weight we obtain in last step. In other words, we sample k cluster centers by means of k-means ++ initialization method but we combine weight and distance when we compute the cost of the cluster centers. Firstly, we sample a center, C_{final}, from C randomly with probability as weight $\vec{\omega}$. Then we run a $k-1$ iteration for loop, to obtain the rest of cluster centers, where in each iteration, we samples each point c_i in set C with probability $\frac{d^2(c_i,C)\omega(c_i)}{\phi_C(C_{final})}$, given current cluster center set C. We update C_{final} with adding the sampled points and update the cluster cost quantity $\phi_C(C_{final})$. Then we run next iteration until the for loop is completed.

$$\phi_C(C_{final}) = \sum_{c_i\in C} d^2(c_i, C_{final})\omega(c_i) \tag{3}$$

where C_{final} is the final cluster center set.

Notice that the size of C is significantly smaller than the input size which means the reclustering can be done quickly.

3 Test

As to guarantee the program works as I want, I do some unit test for different functions I used in the algorithm:

- cost function - non$_negativitycostfunction - ifchasmorepointscostshouldbesmaller$

- cost function - if c has all points cost should be 0
- probability in sampling is non negative
- sum of probability in sampling is l (oversampling factor)
- point in C of probability in sampling is 0
- find number of closet points function - non negative integer
- sum of the weight from weight function should be one
- the weight from weight function should be non-negative
- total levels of labels of KmeansParallel function should be the same as the number of cluster we want
- number of labels should be equal to the number of data

4 Optimization

For optimization, I use Cython and multiple processing to improve the efficiency of the algorithm rather than MapReduce as the author suggested, since it is hard to implement in iPython Notebook server if we need to use the result from MapReduce to MapReduce in a for loop. Overall, I have 3 version of optimization code:

- Just Cythonize the code directly - version C
- Separate some tasks in the algorithm to sse multiple coresvfor multiple processing - version mc

5 Results

In order to compare the efficiency between different algorithms and implementations, I first simulate some data following a mixture of normal distribution. With this data, I plot the clustering result and compare the time that different algorithms used.

See Table 1

	X	Y
0.0000	-7.7854	-0.5804
1.0000	3.2171	3.4884
2.0000	2.7436	5.8117
3.0000	-0.8711	1.5127
4.0000	-2.2518	2.6744
5.0000	-6.7972	-0.4494
6.0000	-5.1354	-1.3787
7.0000	-8.6354	-2.1112
8.0000	-4.3819	0.1750
9.0000	-3.0208	3.9011
10.0000	-2.0080	3.1556

Table 1: Simulated Data.

See Figure 1.
See Figure 2.
See Figure 3.

6 Conclusions

From the cluster map, we can see both K-means++ and scalable K-means have very similar initial centers which are very close to the true centers and make good clustering. In other words, we can consider both algorithms have nearly the same performance. As to the efficiency, when the input data set is small, just 6000 2-Dim points,

However, when the input data set gets larger and larger, the scalable K-means algorithm implemented with multiple processing has higher efficiency than others, even though I can only use 2 CPU in the iPython Notebook server.

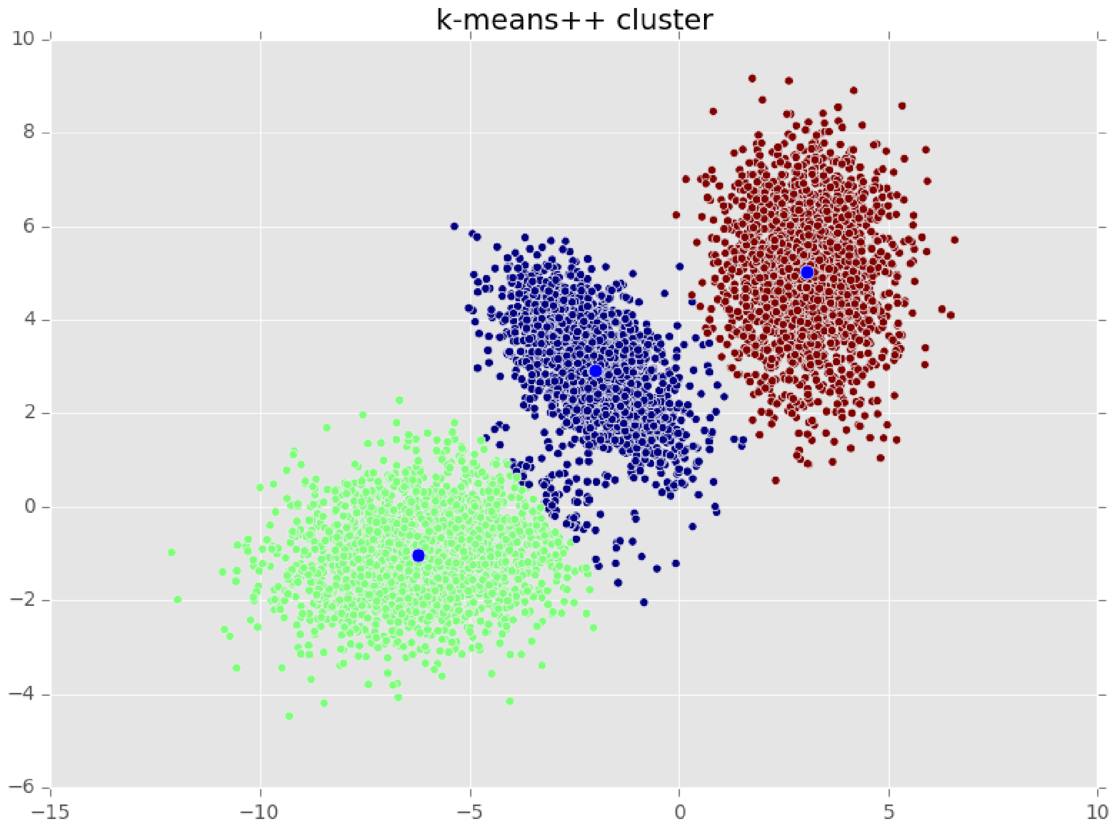

Figure 1: Cluster from K-means++

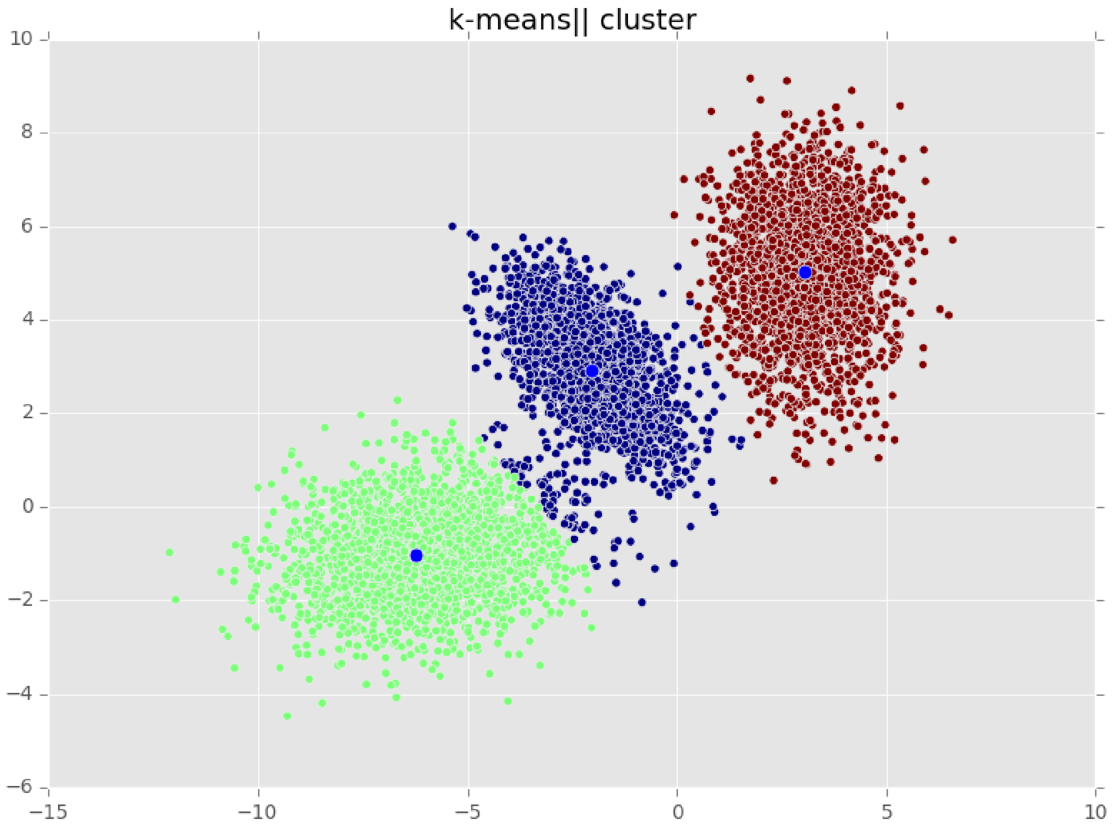

Figure 2: Cluster from K-means——

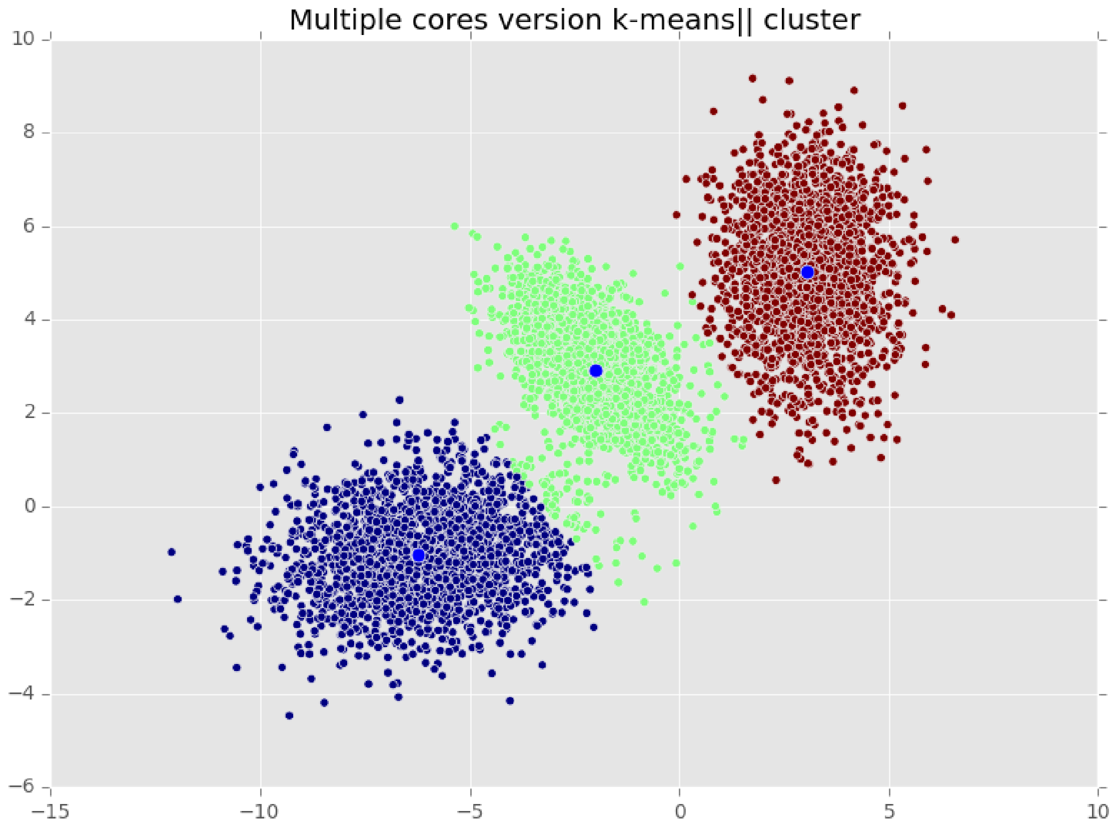

Figure 3: Cluster from K-means—— by means of Multiple Processing