Implement Density Based Spatial Clustering Application with Noise (DBSCAN) and its variants on Husky

LI Changji

The Chinese University of Hong Kong

Introduction

With the development of social network and the increase of human beings, how to process big data to make it fast, efficient and easy to use become an importance problem. Husky[5] , which developed by the group of Prof. James CHENG, is an efficient, expressive and yet intuitive programming models for data-parallel computing system. With Husky, the data will be processed on many machines simultaneously with many threads in one machine such that it will consume less time and memory. Therefore, we can use Husky to realize many different machine learning algorithms such as PCA, SVM and Regression. What I tried to implement is DBSCAN which is a clustering algorithm.

Clustering is the task of partitioning a set of objects into groups which are called cluster so that the objects in the same cluster are similar to each other. DBSCAN is a density-based clustering algorithm which is widely used for clustering set of points because it can detect clusters with arbitrary shape and does not need to know the number of clusters beforehand. Theoretically, its worst case of running time complexity is O(n^2). Therefore, it will consume too much time and memory to process large scale of dataset with normal computer. Using Husky will reduce lots of running time. Nevertheless, there are also some problems for using parallel computing. On the basis of Husky, the algorithm can be modified to make DBSCAN faster and smarter. This report will introduce how to implement DBSCAN with basic version and modified version on Husky.

Method

There are two basic parameters for DBSCAN including *MinPts* which means the minimum number of points and eps(ε) which will be introduced later. What the DBSCAN should do is to partition the data into different based on density. For each data represented by point, if there are some other points whose distance to point $\mathcal{p}$ is less than eps value, these points are the density-approached to point $\mathcal{p}$. If two points are density-approached to each other, these two points must be in the same cluster. By using this basic rule, DBSCAN can find all points

which are density-approached to each other in the same cluster. Before introducing the producer of DBSCAN, there are three more basic concepts. If two points can be density-approached directly, these two points are called neighbors to each other. Meanwhile, if the number of neighbors of one point is larger or equal to *MinPts*, this point can be called *Core Point*. Otherwise, if there is a core point in the neighbor, this point can be called *Border Point* and otherwise for *Noise*.

To realize DBSCAN, the first step is choosing a point arbitrarily and check which type this point belongs to. If the point is core point and the program will check all of the neighbors of this point. If not, choose another point and execute the same procedure. However, the program will choose every point simultaneously but only one on parallel computing system which will result in that the program cannot distinguish different cluster. To solve this problem, it is very efficient and concise to use Minhash[4]:

MinHash (p)
1: {p is the object of data including coordinates, cluster id and neighbors}
2: send the cluster id of p to the neighbors
3: **do**
4: msg ← get cluster id from last sent
5: **If** msg < cluster id
6: cluster id = msg
7: send the new cluster id to neighbors
8: **while** (there is no message get)

By MinHash, the cluster id of point will be assigned to the smallest one in the cluster.

The program must calculate the Euclidean distance between two points to check whether these two points are density-approached. For each point, the program must calculate every other point with this point that bring about the running time must be worse than O(n^2). To make it faster, there is a variant of DBSCAN which can be called pDBSCAN. The core thought of pDBSCAN is the dataset will be partition into grid with the length of cell is eps/$\sqrt{d}$ (d is the number of features) before processing:

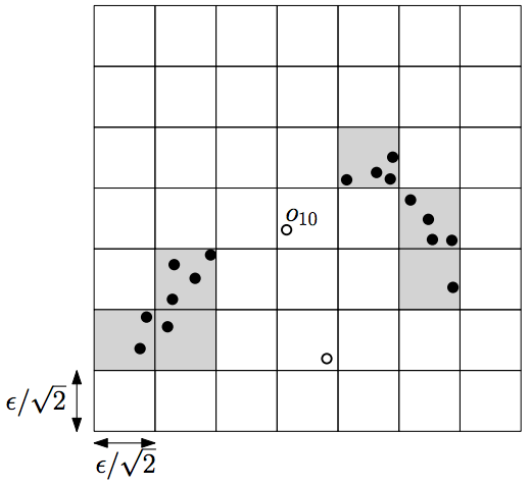


Figure 1. DBSCAN in a grid(2D)

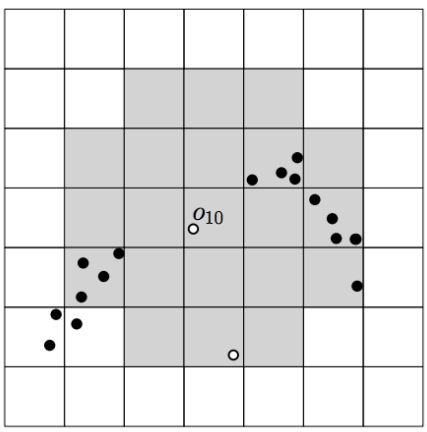
From the figure above, we can find that the points in one cell are all density-approached to each other such that the Euclidean distance between these point is not necessary to calculate. Similarly, the cell which contains more than *MinPts* points is called the *core cell*. It only differs in that if the cell contains at least one core point, it can be called *core cell.* After partitioning the data, there is another modification that is each non-core cell should only calculate nearby twenty cells to determine type:

Cells in gray are what we should calculate for point o_{10}. The last step of this algorithm is merging these cells into cluster. If there are two points p, q, and p belongs to c_1 and q belongs to c_2, and the distance between p and q is larger than eps, c_1 and c_2 are neighbors to each other. After finding all neighbors, we can use MinHash to merge these cells into relevant clusters.

Figure 2.Neighbor cells (in gray) of the cell of o_{10}

Result

This chapter will show some result of DBSCAN and its variant. The datasets we choose is **Seed Spreader[2]** and **HIGGS[1]** which are from UCI.



Seed Spreader(SS):

SS is a small and low features dataset for testing the feasibility and precision which is as follows:

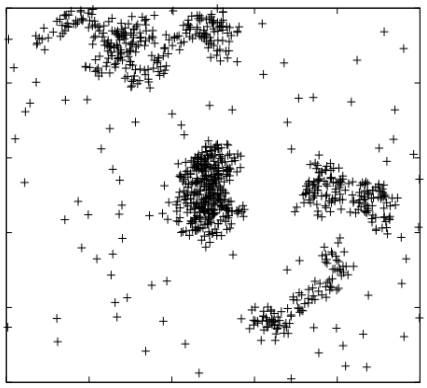


Figure 3. 2D seed spreader dataset

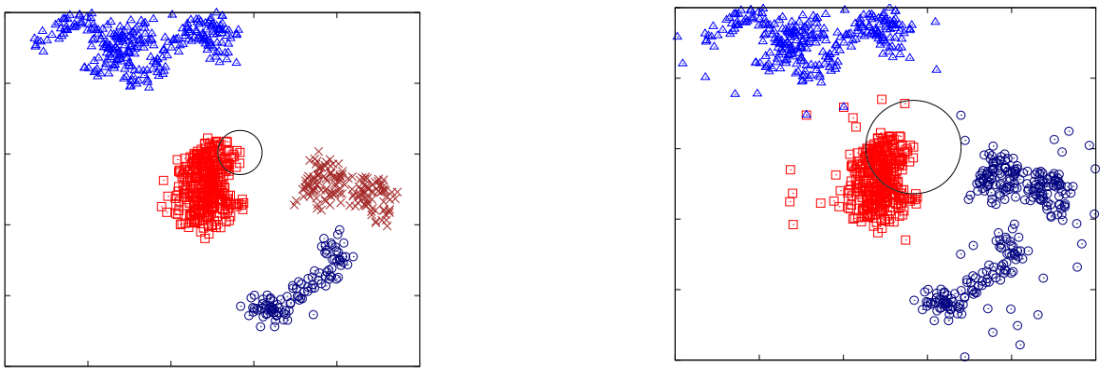


Figure 4. eps = 5000

Run the DBSCAN algorithm and mark cluster by different color, we can get:

Figure.4 is applied DBSCAN with eps equals 5000 and Figure.5 with 11300.

HIGGS:

HIGGS contains ten million data and twenty-seven features. For testing the DBSCAN and pDBSCAN, the datasets will be loaded partly to get more information.

Figure 5.eps = 11300

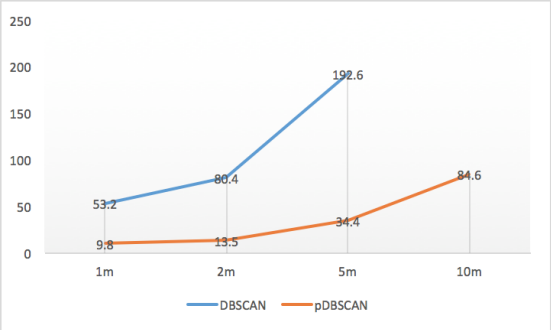


Figure 6.Running time vs data size

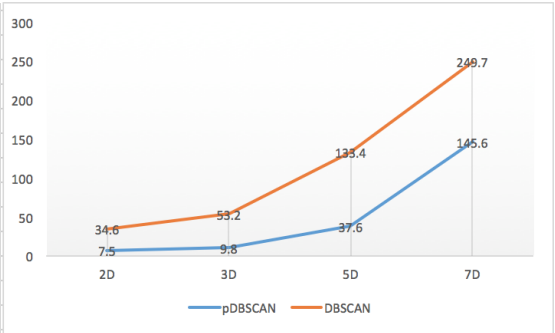


Figure 7. Running time vs number of features

The x-axis of figure.6 represents the size of data, e.g. 1m for one million and the y-axis represents the time of algorit1hm use with second. The number of features is always three. This figure shows that pDBSCAN is faster than DBSCAN and can handle larger data size. The same with figure.6, the y-axis is the running time. Meanwhile, the x-axis represents the number of features, e.g. 2D for two features. This figure shows the pDBSCAN can handle high dimensional data easier.

Compared with Spark[3]

Spark is another data-parallel computing system and there is implementation of DBSCAN[]. Compare the performance of DBSCAN on Husky and on Spark is like:

```
Data size: 2,000,000   Number of features: 2
Spark: around 20 minutes with 4 nodes
Husky: 9.5 seconds with 20 workers
       12.6 seconds with 4 workers

Data size: 3,000,000   Number of features: 2
Spark: around 50 minutes with 4 nodes
Husky: 13.9 seconds with 20 workers
       14.9 seconds with 4 workers

Data size: 10,000,000   Number of features: 2
Spark: No Data
Husky: 49.6 seconds with 20 workers

Data size: 2,000,000   Number of features: 3
Spark: No Data
Husky: 13.2 seconds with 20 workers
```

Table 1. Performance of DBSCAN on Husky and on Spark

Table 1 presents that DBSCAN on Husky is faster than on Spark and it can handle larger and higher dimensional dataset.

Discussion

From the result of testing DBSCAN and comparison with Spark, the basic DBSCAN can perform well because of the Husky and it also has many restrictions. And the pDBSCAN can handle more and perform better. There are also many other variants of DBSCAN which also can improve the performance such as approximate version of DBSCAN. For approximate DBSCAN, the procedure of merge can be modified to O(n). However, because the parallel computing requires message communication a lot between each object, the running time complexity will be increased to O(2^D) (D for number of dimensions). Therefore, implementing the approximate DBSCAN on Husky and reaching the expected result is still a problem and will be the future work.

References

[1] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5, 2014.

[2] M. Charytanowicz, J. Niewczas, P. Kulczycki, P. A. Kowalski, S. ukasik, and S. Zak. Complete gradient clustering algorithm for features analysis of x-ray images. In *Information technologies in biomedicine*, pages 15–24. Springer, 2010.

[3] DBSCAN on Spark. https://github.com/alitouka/spark_dbscan.

[4] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[5] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.