Deep Project - DBSCAN

Wai Leuk Lo
Student ID: 20153384
Group 3
wlloaa@connect.ust.hk

Abstract

We choose DBSCAN implementation as our deep project, introduce the problem and the algorithms, describe our algorithm and the implementation, and finally test it on the blobs datasets and analyze the results.

1 Introduction

Given a set of points in some space, we would like to cluster them based on their closeness in the space, where this closeness is quantified by some distance metric (Euclidean distance, Manhattan distance, etc.). Clustering methods such as k-means clustering have two short-comings, namely the need to specify the number of clusters ahead of time and not handling irregular-shaped clusters well. Density-based spatial clustering of applications with noise (DBSCAN) is a clustering algorithm that aims to solve these two problems. It clusters points that are densely packed (points with many neighbors) and marks points in low-density regions (points with not enough important neighbors) as noise (Schubert et al., 2017). Consider a set of points to be clustered, and let ϵ be the search radius of neighborhood of a point and MinPts be the threshold number of neighborhood points. Then,

- A point p is a *core point* if there are at least MinPts points within distance ϵ of it (including p).
- A point p is a *border point* if it is not a core point but is within distance ϵ of a core point.
- A point p is a *noise point* if it is neither a core point nor a border point.

DBSCAN then clusters the points based on the following four principles (Wong, 2024):

- Principle 1: Each cluster contains at least one core point.
- Principle 2: Given any two core points p and q, if they are in the neighborhood of each other, they are in the same cluster.
- Principle 3: Consider a border point p to be assigned to one of the clusters formed by Principle 1 and Principle 2. Suppose the neighborhood of p contains multiple core points, then it is arbitrarily assigned to one of the clusters containing these core points.
- Principle 4: All noise points do not belong to any cluster.

An algorithm for DBSCAN satisfying these four principles starts from a core point and its neighbors, and transitively expands the cluster using its core neighbors. A detailed algorithm is given by Alg. 1, where RangeQuery returns the ϵ-neighborhood of a point (Schubert et al., 2017).

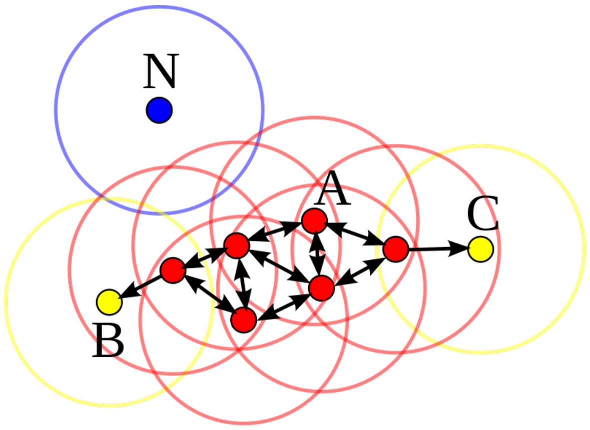


Figure 1: Illustration of DBSCAN.

2 Methodology

Alg. 1 loops over all points in the database and expands a cluster to its maximum extent when it encounters a core point. At any given moment, at most all points in the neighborhood of a core point rather than all points in the database have to be

Algorithm 1: DBSCAN(DB, distFunc, ϵ, MinPts)

```
Input: Database DB, distance function
       distFunc, neighborhood radius $\epsilon$,
       threshold number of neighborhood
       points MinPts;
Output: Cluster label for each point in DB;
Initialize C := 0;
foreach point P in database DB do
    if label(P) $\neq$ undefined then continue;
    Neighbors N := RangeQuery(DB,
      distFunc, P, $\epsilon$);
    if |N| < MinPts then
        label(P) := Noise;
        continue
    end
    C := C + 1;
    label(P) := C;
    SeedSet S := N \{P};
    foreach point Q in S do
        if label(Q) = Noise then label(Q) :=
          C;
        if label(Q) $\neq$ undefined then
          continue;
        label(Q) := C;
        Neighbors N := RangeQuery(DB,
          distFunc, Q, $\epsilon$);
        if |N| $\geq$ MinPts then S := S $\cup$ N;
    end
end
```

Algorithm 2: RangeQuery(DB, distFunc, P, ϵ)

```
Input: Database DB, distance function
       distFunc, point P, neighborhood
       radius $\epsilon$;
Output: Neighborhood of point P;
Neighbors N := empty list;
foreach point Q in database DB do
    if distFunc(P, Q) $\leq \epsilon$ then
        N := N $\cup$ {Q};
    end
end
return N
```

saved in memory. Therefore, it is memory-efficient at the cost of dealing with only one cluster at a time.

An alternative algorithm instead finds all neighbors for each point, identifies core points and merge core points into clusters, as shown in Alg. 3 (Schubert et al., 2017). This algorithm is more memory consuming but more suitable for parallel computing, so we will follow this line of idea for implementation in this project. In the first step, we calculate the distance between every two points in the dataset. Then, we build a neighborhood set for each point based on the distances and ϵ and count the number of points in the neighborhood of each point. We will then use treeReduce to merge the sets of core points (i.e. those points whose number of neighborhood points is greater than or equal to MinPts) to obtain the clusters. We apply the idea of divide and conquer, and perform the following five steps:

Algorithm 3: Abstract DBSCAN Algorithm

```
Compute neighbors of each point and
  identify core points;
Join neighboring core points into clusters;
foreach non-core points do
    Add to neighboring core points if
      possible;
    Otherwise, add to noise
end
```

1. Divide the points into square grids.
2. Perform DBSCAN on each grid as described in Alg. 3.
3. Send the maximum cluster id from each grid and compute the prefix sum to shift cluster ids in each grid.
4. Aggregate the cluster ids of those core boundary points to form an id map.
5. Map the cluster ids in each grid to new cluster ids according to merged ids in Step 4.

We now elaborate the steps above. In Step 1, the points are divided with a binary search in each dimension. Furthermore, to enable connections from different grids later on, the grids have an ϵ overlap with each other (in practice, we just send the boundary points to the neighboring grids). At this step, when using groupByKey to send points to different workers based on their grids, we set the partition function to be the sum of two grid indices mod the number of partitions to alleviate the partition imbalance a bit.

In Step 2, a k-d tree (k-dimensional tree) from scikit-learn is built for the points in each grid for spatial indexing. This takes some computations but enable us to perform DBSCAN more efficiently as the distance query for a specific point is conducted in its proximity and not all points in the grid. Note that the noise point is labeled as 0 at this step, and the cluster id starts from 1.

In Step 3, we send the maximum cluster id from each grid to the driver. The driver then performs a prefix sum and send the sums back to the workers for them to shift the cluster ids on each grid. This is done so that there will be no id overlap between different grids. Note that the same boundary points that reside in more than one grid will now have different ids.

In Step 4, we aggregate the cluster ids of those *core* boundary points to construct an id map, so that the originally disconnected points on different grids can now be merged together, if they belong in the same cluster. As the number of core boundary points is still too large, $treeReduce$ is used for the aggregation. The reduce action cannot merge all the cluster ids since it is performed in a sequential fashion, and thus a depth-first search is performed on the driver side to further merge all possible cluster ids.

In Step 5, we send this id map (a small dictionary) to each worker and the cluster ids are changed one last time, and the algorithm terminates.

3 Experiments, Results and Analysis

We produce some synthetic data using *make_blobs* provided by the scikit-learn library with varying number of points to test the performance of our model. The results are summarized in 1. We observe that the amount of time needed to perform a DBSCAN scales somewhere between N and N^2, where N is the number of points. Furthermore, as expected from a parallelized algorithm, the number of time decreases as the number of workers are added in a roughly linear manner.

Upon an analysis on the complexity of the algorithm, we observe that the most of the time is consumed to build the k-d tree and perform DBSCAN on the tree. Building a k-d tree costs $O(N \log N)$ time, while querying a point costs $O(\log N)$ time on average, which means querying all points costs $O(N \log N)$ time on average[1]. This explains the scaling behaviour of the running time with respect to varying number of points.

	Dataset	Blobs			
	Number of points	1500	25,000	100,000	200,000
	1	7.02	19.66	337.07	-
Number of workers	3	5.39	13.94	125.87	349.12
	5	4.37	15.03	92.12	244.28

Table 1: Comparison of running time (in seconds) for DBSCAN on datasets across different number of points and number of workers. There are no data for 200,000 points for the one worker case, as Spark stops the program after too many crashes.

4 Conclusion

In this project, we implement a parallelized algorithm to perform DBSCAN based on the idea of divide and conquer, and test it on the blobs datasets with varying number of points. During the implementation, we alleviate the partition imbalance using a suitable partition function, and merge cluster ids using depth-first search. We observe that the running time scales between N and N^2, and propose a possible explanation based on the complexity analysis. There are some short-comings of our implementation, and the most prominent one is that the program crashes when there are more than 200,000 points, which indicates that the implementation is probably still not optimal.

Acknowledgments

In fact, our first attempt to implement the parallelized algorithm for DBSCAN requires sending all core boundary points back to the driver, and the query search is performed on all points in the grid rather than in the proximity of a point. After listening to the presentations of other groups doing the same project, discussion with the fellow students from those groups, as well as hearing the feedback of Prof. Yi, we are able to implement an alternative algorithm presented here, which utilizes k-d tree for the query search, and uses a small overlap between different grids for cluster id aggregation to avoid sending points back to the driver. We would like to thank Prof. Yi and fellow students from the two other groups doing the deep project on DBSCAN.

[1]In our implementation we do not query all points, but stop when we have visited all points

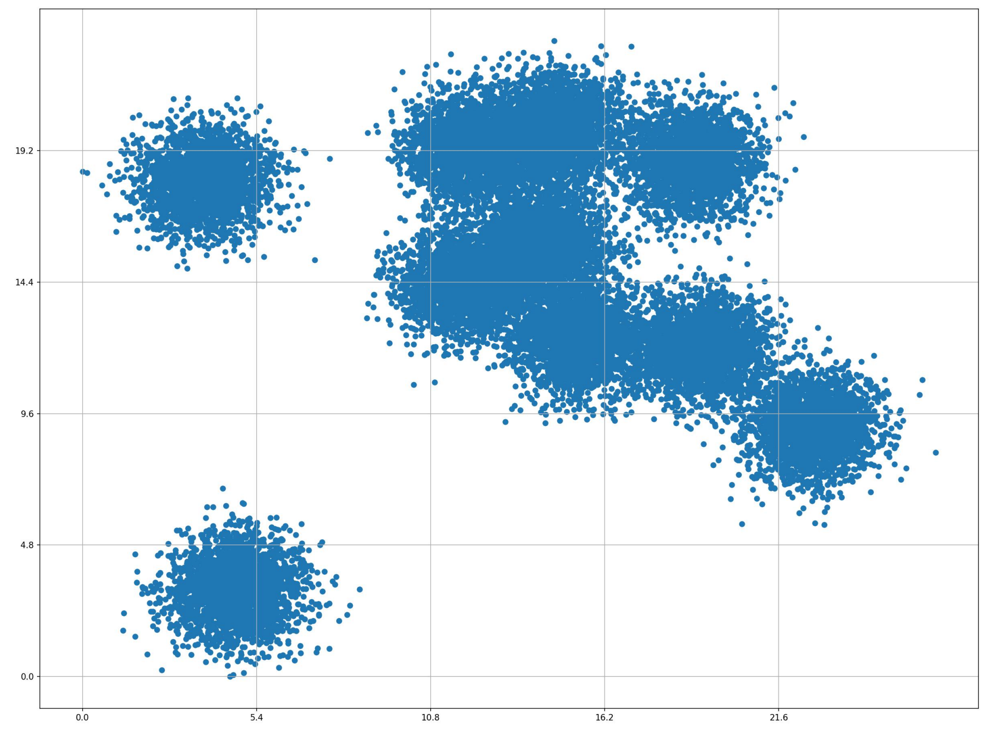
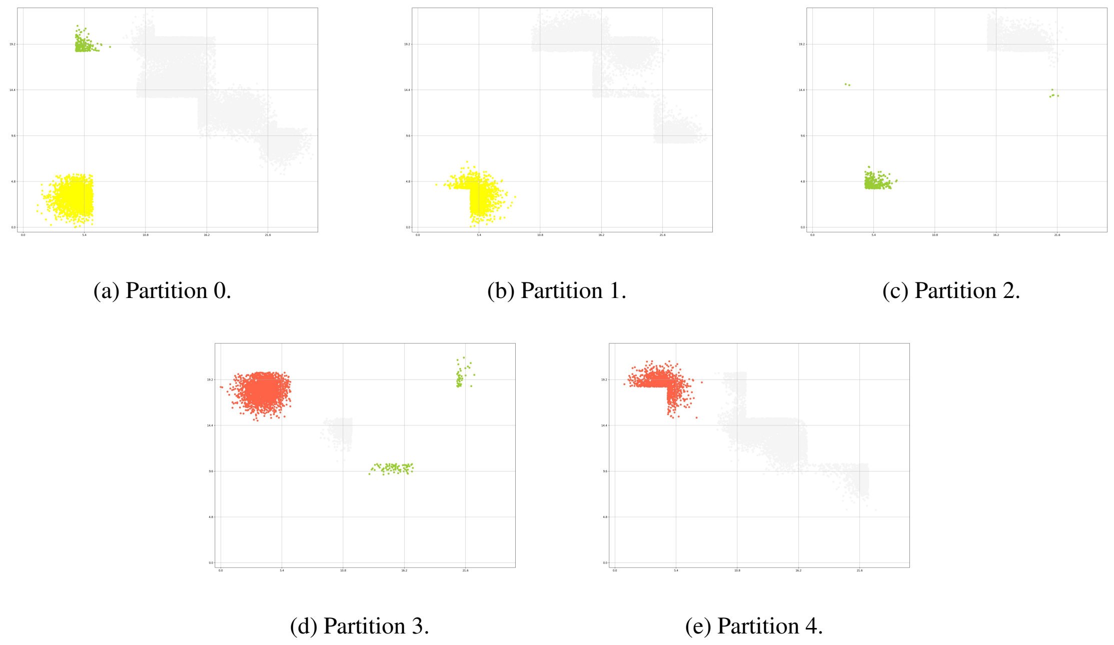


Figure 2: Blobs with 25,000 points.



(a) Partition 0.

(b) Partition 1.

(c) Partition 2.

(d) Partition 3.

(e) Partition 4.

Figure 3: Clusters formed on each of the 5 partitions. Notice that the grids are sent to the partitions based on grid index 1 + grid index 2 mod the number of partitions, which is 5 in this case. Also notice that the points divided into different grids are still able to form a single cluster in the dense area in the middle and in the lower left corner, thanks to the core boundary points that act as bridges between different grids.

The depth-first search algorithm for further merging the cluster ids is referenced from https://stackoverflow.com/a/4843380/22821896.

References

Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. Dbscan revisited, revisited: Why and how you should (still) use dbscan. *ACM Trans. Database Syst.*, 42(3).

Raymond Wong. 2024. Other clustering techniques. *MSBD5002 Lecture Notes*.