DB scan report

2015004248
Jo Hyung Jung

1. Overview

DB scan is cluster analysis using density-based method. Density-based clustering method uses density rather than just a distance. So, it can discover clusters of arbitrary shape, handle noise, and only requires one scan. However, it has drawback that needs density parameters as termination condition. DB scan is announced in 1996 KDD. Then ‘OPTICS’, which can visualize DB scan’s parameter settings, is announced in 1999 SIGMOD.

2. Run-time environment

- OS: Ubuntu 18.04
- Language: Python 3.6
- Libraries:
    Python argparse module: for getting command-line arguments.

3. Summary of algorithm

DB scan method uses some basic concepts. There are two parameters named ‘Eps’, and ‘Minpts’. ‘Eps’ means maximum radius of the neighborhood. ‘Minpts’ means minimum number of points in an Eps-neighborhood of a given point.

Two objects can be ‘neighbor’ when two objects’ distance is less or equal than ‘Eps’.

There are two basic notation in DB scan.

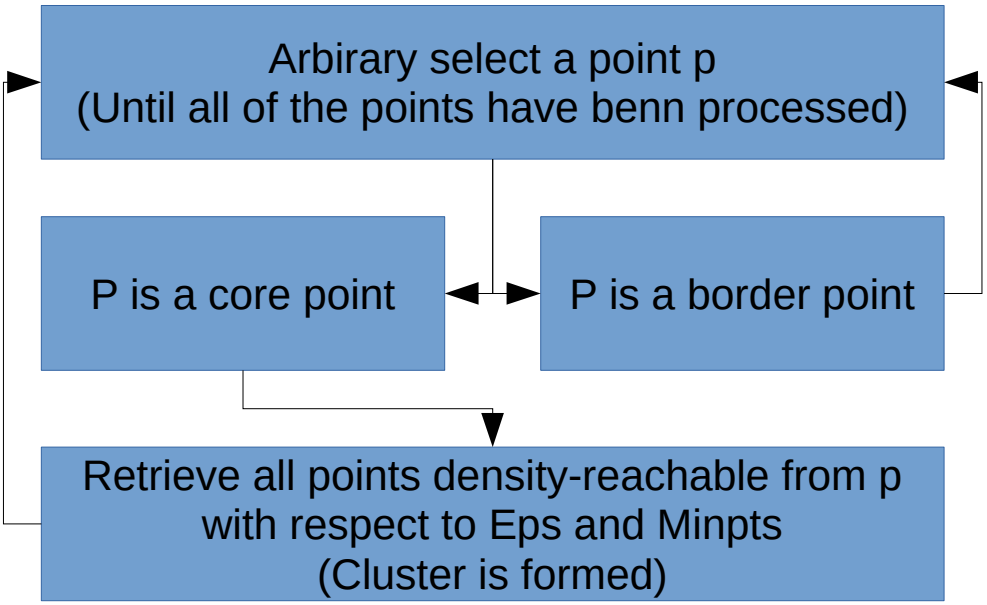
i) Directly density-reachable

A point q is density-reachable from a point p with respect to ‘Eps’ and ‘Minpts’ if q belongs to ‘neighbor’ of p, and p meets ‘core point’ condition.

To meet ‘core point’ condition, the number of ‘neighbors’ of an object should be greater or equal than ‘Minpts’.

ii) Density-reachable

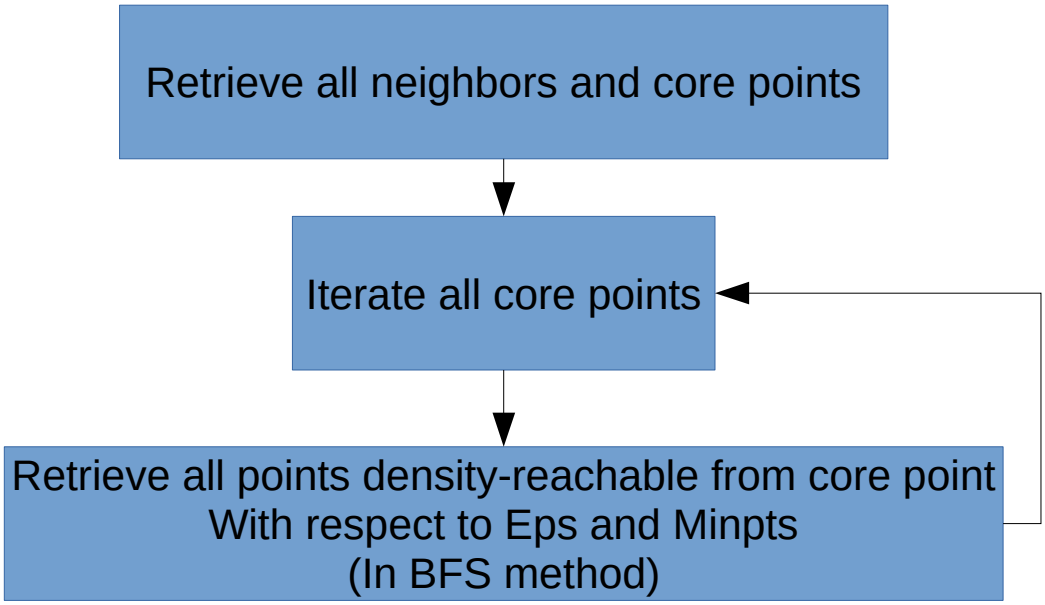
A point q is density-reachable from a point p with respect to ‘Eps’ and ‘Minpts’ if there is a chain of points p1,p2,…,pn, p1 = p, pn = q such that pi+1 is directly density-reachable from pi.



[DB scan flow chart]

4. Implementation details

I slightly modified original DB scan. First, retrieve all neighbors and core points. Second, by iterating all core points, form clusters. In DB scan, only core points are used in forming clusters. So I think it is reasonable modification. When forming clusters, I used BFS method to find all points density reachable from certain core point. Below is the flow chart.



[Flow chart]

5. Analysis of code

i) Data structures

I use one class for representing **‘Cluster’**.

```
class Cluster:
    def __init__(self, args):
        # Initialize values
        self.input_file = args.input_file
        self.n = args.n
        self.eps = args.eps
        self.minpts = args.minpts

        # Save output file names
        title = self.input_file.split('.')[0]
        self.output_file = [title+"_cluster_{}.txt".format(i) for i in range(self.n)]

        # Extract data
        with open(self.input_file, 'r') as f:
            data = [line.split() for line in f.readlines()]
            self.data = {int(e[0]): (float(e[1]), float(e[2])) for e in data}
```

There are 6 attributes in ‘cluster’ class.
1. input_file: represents input file name.
2. n: represents number of clusters.
3. eps: represents epsilon value for clustering.
4. minpts: represents number of minimum points for clustering.
5. output_file: represents output file name.
6. data: represents each point’s information. Dictionary: {point id: (point x value, point y value)}

ii) Functions

I use 6 functions in ‘Cluster’ class.

- __init__

```
class Cluster:
    def __init__(self, args):
        # Initialize values
        self.input_file = args.input_file
        self.n = args.n
        self.eps = args.eps
        self.minpts = args.minpts

        # Save output file names
        title = self.input_file.split('.')[0]
        self.output_file = [title+"_cluster_{}.txt".format(i) for i in range(self.n)]

        # Extract data
        with open(self.input_file, 'r') as f:
            data = [line.split() for line in f.readlines()]
            self.data = {int(e[0]): (float(e[1]), float(e[2])) for e in data}

        # Get each data's neighbors & corepoints
        self._get_neighbors()
        self._get_corepoints()

        # Set visited array
        self.visited = []
        # Get clusters()
        self._get_clusters()

        # Write result
        self._write_result()
```

Feature: Initialize attributes and make clusters.
Arguments: args → python 'Argument Parser' object
Return value: None

In this function, attributes are initialized. Then, cluster is formed by certain routine.

-_get_distance

```
@staticmethod
def _get_distance(x1, y1, x2, y2):
    return ((x1-x2)**2 + (y1-y2)**2)**(1/2)
```

Feature: Calculate two points' distance.
Arguments: x1→ float, y1→ float, x2 → float, y2 → float
Return value: ((x1-x2)2 + (y1-y2)**2)**(1/2) → Euclidean distance(float)**

In this function, calculate two points' distance.

-_get_neighbors

```
def _get_neighbors(self):
    self.neighbors = {key: [] for key in self.data.keys()}
    for e1_key, (e1_x, e1_y) in self.data.items():
        for e2_key, (e2_x, e2_y) in self.data.items():
            if e1_key == e2_key:
                continue
            dist = self._get_distance(e1_x, e1_y, e2_x, e2_y)
            if dist <= self.eps:
                self.neighbors[e1_key].append(e2_key)
```

Feature: Create all points' neighbors
Arguments: None
Return value: None

In this function, all neighbors from all points are created. By iterating all points,
first calculate two points' distance. Then check if distance is less or equal than epsilon value.
If so, two points are saved as neighbor.

-_get_corepoints

```
def _get_corepoints(self):
    self.core_pts = [key for key, val in self.neighbors.items() if len(val) >= self.minpts]
```

Feature: Create all core points
Arguments: None
Return value: None

In this function, all core points are created. By iterating all neighbors, check if number of neighbors is greater or equal than minimum points. If so, that point is saved as core point.

-_get_clusters

```
def _get_clusters(self):
    self.clusters = []
    # Visit core points first
    # BFS
    for cpt in self.core_pts:
        if cpt in self.visited:
            continue
        queue = [cpt]
        cluster = [cpt]
        self.visited.append(cpt)
        while len(queue)>0:
            cur = queue.pop(0)
            if cur not in self.core_pts:
                continue
            for nxt in self.neighbors[cur]:
                if nxt not in self.visited:
                    self.visited.append(nxt)
                    cluster.append(nxt)
                    queue.append(nxt)

        self.clusters.append(cluster)

    # Sort clusters
    self.clusters.sort(key=len, reverse=True)
    # Select n clusters
    self.clusters = self.clusters[:self.n]
```

Feature: Create all clusters
Arguments: None
Return value: None

In this function, create all clusters. By iterating all core points, retrieve all density-reachable points from that core point. It is done by BFS process. After creating all clusters, sort clusters by descending order(# of points in certain cluster). Then select top-n clusters.

-_write_result

```
def _write_result(self):
    for i, file_name in enumerate(self.output_file):
        with open(file_name, 'w') as f:
            for e in self.clusters[i]:
                f.write("{}\n".format(e))
```

Feature: Write result files
Arguments: None
Return value: None

In this function, write all result files. By iterating top-n clusters, write all point ids in certain cluster.

iii) Code flow

```
if __name__ == "__main__":
    # Parse arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("input_file", type=str)
    parser.add_argument("n", type=int)
    parser.add_argument("eps", type=int)
    parser.add_argument("minpts", type=int)
    args = parser.parse_args()

    # Create cluster object
    dbscan = Cluster(args)
```

[In main routine]

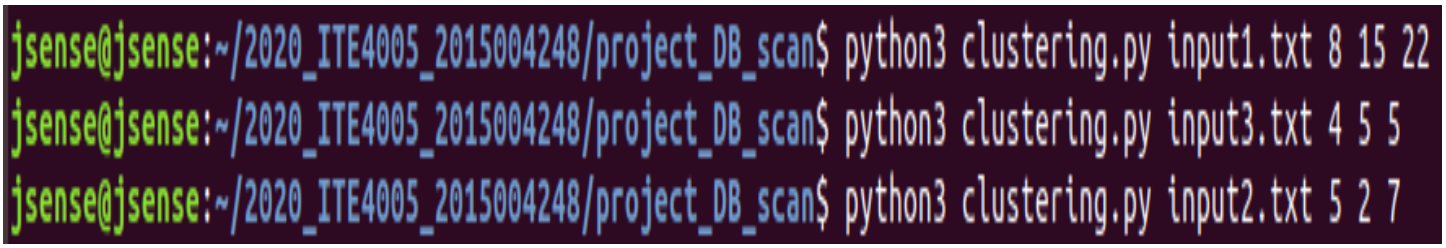
1. Parse command-line arguments.
2. Construct ‘Cluster’ class

```
class Cluster:
    def __init__(self, args):
        # Initialize values
        self.input_file = args.input_file
        self.n = args.n
        self.eps = args.eps
        self.minpts = args.minpts

        # Save output file names
        title = self.input_file.split('.')[0]
        self.output_file = [title+"_cluster_{}.txt".format(i) for i in range(self.n)]

        # Extract data
        with open(self.input_file, 'r') as f:
            data = [line.split() for line in f.readlines()]
            self.data = {int(e[0]): (float(e[1]), float(e[2])) for e in data}

        # Get each data's neighbors & corepoints
        self._get_neighbors()
        self._get_corepoints()

        # Set visited array
        self.visited = []
        # Get clusters()
        self._get_clusters()

        # Write result
        self._write_result()
```

[In __init__ function]

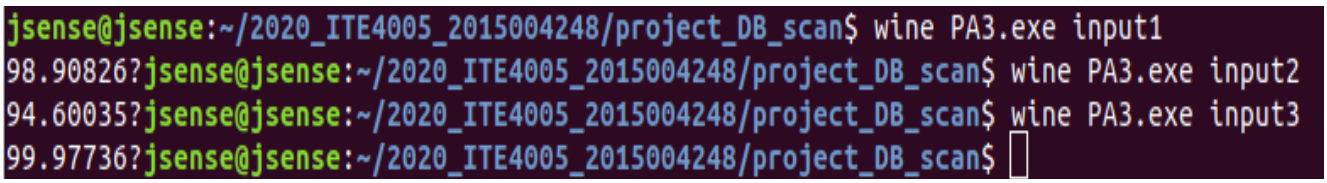
1. Save arguments
2. Retrieve output file names then save them.
3. Extract data from input file.
4. Retrieve all neighbors.
5. Retrieve all cores points.
6. Create all clusters.
7. Write result files.

6. Instructions for execution

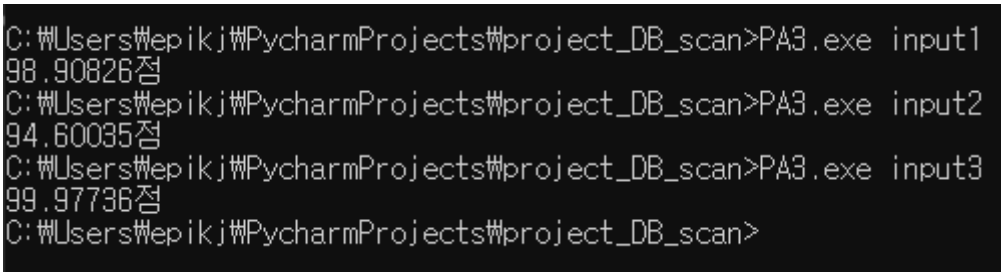


Type ‘python3 clustering.py “input file name” “n” “epsilon” “minimum points”’ in terminal

7. Test result



[Test result in Ubuntu]



[Test result in Windows]

8. Reference

[1] Jiawei Han. Data mining: concepts and techniques(3rd edition), 2000.