Data Science Assignment #3

DBSCAN: Density-based Clustering Method

2015004857 이영수

1) Summary of algorithm

- Iterate all points in dataset, and then check whether the length of its neighbors is more than *MinPts*.
- If the point satisfies, put it to the current cluster, and look for other points that belongs to same cluster (called expansion).
  - Expansion is started with a seed 'neighbors', and iterate the seed checking state of each point that is in seed.
    - If the point is NOISE, put it to current cluster.
    - Else if the point is UNCLASSIFIED, put it to cluster and put it's neighbors to **seed** for expanding search area.
- If expansion is finished, find another cluster with unclassified points.
- If all points are classified, stop clustering and return the result.

2) Code Description

Codes are written by Python3.

```
~/ITE4005/dbscan  master  tree
.
├── README.md
├── clustering.py
├── data
│   ├── input0.txt
│   ├── input1.txt
│   ├── input2.txt
│   └── input3.txt
└── tests
    └── test_dbscan.py
```

Project consists of python file that implements clustering by DBSCAN method, test files, and dataset for testing.

clustering.py

There are two libraries used, `sys` for getting program arguments, and `math` for calculating distance, etc.

```
"""
@author Prev (prevdev@gmail.com)
"""
import sys
import math
```

Firstly, the node to cluster is called `Point` in this program. Each `Point` has `id`, `x`, `y`, and `cluster` attribute. Attribute `cluster` has initial value as UNCLASSIFIED. After clustering, this attribute would be set to integer value n (n > 0) means ID of the cluster that point belongs to. Method `dist` in this class returns the distance between two points by Euclidean metric.

```
class Point:
    class Label:
        UNCLASSIFIED = 0
        NOISE = -1

    def __init__(self, id, x, y):
        """ Point Constructor
        :param id: ID of point
        :param x: X-axis position
        :param y: Y-axis position
        """
        self.id = id
        self.x = x
        self.y = y
        self.cluster = Point.Label.UNCLASSIFIED

    def dist(self, target):
        """ Euclidean distance with other point
        :param target: Point instance
        :return: Numeric distance
        """
        return math.sqrt(math.pow(self.x - target.x, 2) + math.pow(self.y - target.y, 2))
```

Secondly, there is `Cluster` class that executing the "clustering". It has several methods, and `file2points` method that read input file and interpret it's content to list of `Point`.

```
class Cluster:

    @staticmethod
    def file2points(input_file):
        """ Read file and interpret to list of Points
        :param input_file: Opened file
        :return: List of Point
        """
        ret = []
        for line in input_file.readlines():
            if line[-1] == '\n': line = line[0:-1]
            t = line.split('\t')

            ret.append(Point(
                id=int(t[0]),
                x=float(t[1]),
                y=float(t[2])
            ))

        return ret
```

In the constructor of Cluster, it gets list of points, number of clusters, epsilon of DBSCAN, and MinPts of DBSCAN.

```
52      def __init__(self, points, n, eps, minpts):
53          """ Initialize Cluster
54          :param points: List of Point class
55          :param n: Number of clusters
56          :param eps: Epsilon of DBScan
57          :param minpts: MinPts of DBScan
58          """
59          self.points = points
60          self.n = n
61          self.eps = eps
62          self.minpts = minpts
```

The `cluster` method is main function to find clusters by dataset. This method returns list of clusters, where cluster means list of IDs of `Point`. The first logic of it is to iterate all points in dataset, and then check whether length of neighbors is more than *MinPts*. If the point satisfies, put it to the current cluster, and look for other points that belongs to same cluster (Line 86). This job (finding additional neighbors) called *expansion*, and it is implemented at `_expand` method. After finding all points that belong to current cluster, add 1 to *cluster_id* for moving to next cluster (Line 90). In the `_expand` function, it finds all nodes belonging to the cluster by *DFS*, so points that have already been classified are ignored on next iteration (Line 73).

```
64      def clusters(self):
65          """ Get clusters by running DBScan
66          :return: List of clusters (Cluster is list of point ids)
67          """
68          print('Start clustering...')
69
70          cluster_id = 1
71          for point in self.points:
72              if point.cluster != Point.Label.UNCLASSIFIED:
73                  continue
74
75              # Get neighbors of current point
76              neighbors = self._neighbors(point)
77
78              # If it has not enough neighbors, set it as noise
79              if len(neighbors) < self.minpts:
80                  point.cluster = Point.Label.NOISE
81                  continue
82
83              # Else if it is core point, make cluster and expand
84              # to find more points that belong to same cluster
85              point.cluster = cluster_id
86              self._expand(
87                  seeds=neighbors,
88                  cluster_id=cluster_id
89              )
90              cluster_id += 1
91
92          # Make cluster list from points
93          clusters = [[] for _ in range(0, cluster_id-1)]
94          for point in self.points:
95              if point.cluster == Point.Label.NOISE:
96                  continue
97              clusters[point.cluster - 1].append(point.id)
98
99          clusters.sort(key=lambda l: len(l), reverse=True)
100         return clusters[0:self.n]
```

When labeling *cluster_id* to points is all finished, tie points together which have same *cluster_id*. Then pick up `n` clusters that contain the most points, and returns them. (Line 93~100)

The `_neighbors` method is implemented simply, iterating all point and picking up whose distance with counter is less than *epsilon*.

```
102     def _neighbors(self, point):
103         """ Get neighbors of point
104         :param point: The point to find out neighbors
105         :return: List of points
106         """
107         return [p for p in self.points if p != point and point.dist(p) <= self.eps]
```

The `_expand` method is used in `cluster` method, to expand candidate points for clustering, It gets seeds, the initial candidates of expansion, and the *cluster_id* for labeling newly discovered points. If a point of seeds is NOISE, just labeling it to current cluster, and else if the point is UNCLASSIFIED, performing the labeling and put it's neighbors to seed for expanding search area, if length of new neighbors is more than *MinPts*.

```
109     def _expand(self, seeds, cluster_id):
110         """ Expand candidate points for clustering
111         :param seeds: Seed for performing expansion
112         :param cluster_id: If new point is same cluster to seed, set its cluster to this
113         """
114         for point in seeds:
115             if point.cluster == Point.Label.NOISE:
116                 point.cluster = cluster_id
117 
118             if point.cluster == Point.Label.UNCLASSIFIED:
119                 point.cluster = cluster_id
120                 n_neighbors = self._neighbors(point)
121                 if len(n_neighbors) >= self.minpts:
122                     seeds.extend(n_neighbors)
123 
```

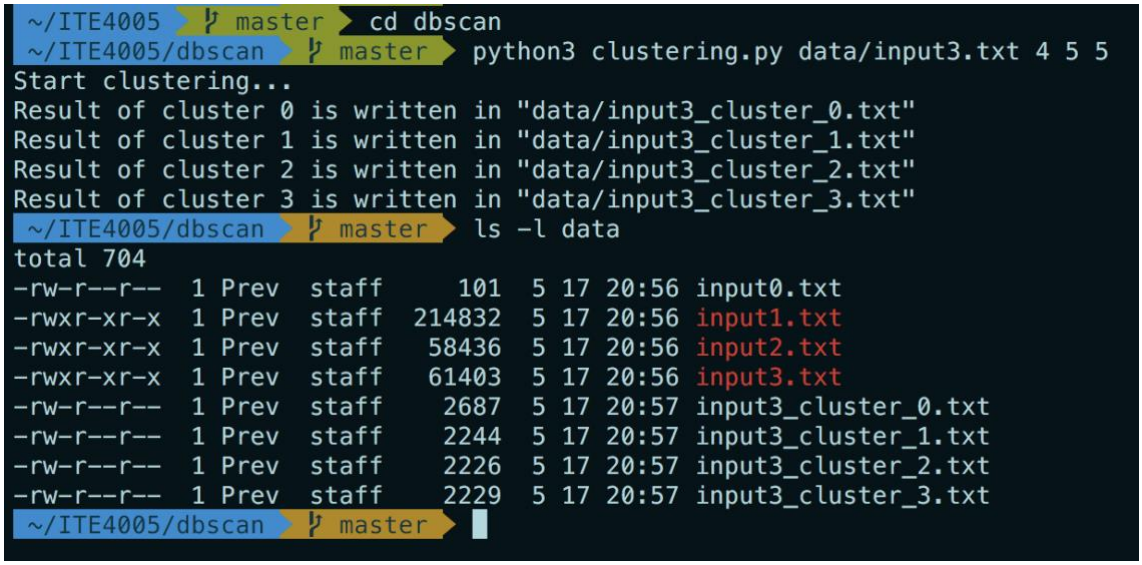
This is end of class `Cluster`, and process for getting program arguments, constructing class, and run clustering is implemented in the last of code. Cause `clusters` method returns list of clusters, logic for iterating its result and write output files is on here. The name of output file is generated by the requirements in the specification.

```
125 ▶ if __name__ == '__main__':
126       if len(sys.argv) != 5:
127           print("Usage: python cluster.py <input_file> <n> <eps>, <minpts>")
128           sys.exit(-1)
129
130       input_file_name = sys.argv[1]
131
132       c = Cluster(
133           Cluster.file2points(open(input_file_name, 'r')),
134           int(sys.argv[2]),
135           int(sys.argv[3]),
136           int(sys.argv[4]),
137       )
138
139       for index, output in enumerate(c.clusters()):
140           output_filename = input_file_name.split('.')[0] + '_cluster_%d.txt' % index
141
142           with open(output_filename, 'w') as output_file:
143               for object_id in output:
144                   output_file.write('%d\n' % object_id)
145
146           print('Result of cluster %d is written in "%s"' % (index, output_filename))
147
```

3) Instruction for compiling

Codes are written by python3, so please not to run with python2.

```
$ python3 clustering.py data/input3.txt 4 5 5
```



You can also run tests by below instruction. (pytest is required)