Report: Assignment #3

ITE4005, **Data Science.**
2016025305, **Jihun Kim<**jihunkim@hanyang.ac.kr**>.**

Perform clustering on a given data set by using DBSCAN.

Getting Started

Development Environment

* **OS:** macOS 10.15.5
* **Language:** Python 3.6.8

Run

```
$ cd /path/to/repo/assignment3
$ pip install -r requirements.txt
$ python main.py ./data/input1.txt 8 15 22
```

Implementation

Data Representation

`DataObject` represnets a single data object, and **`DBSCAN_Object`** is a wrapper class for helping DBSCAN algorithm. **`get_neighbors()`** method finds neighbors, based on **`eps`** value. It prevents computing neighbors again by storing neighbors in it.

```
class DataObject(object):
    def __init__(self, id, x, y):
        self.id = int(id)
        self.x = float(x)
        self.y = float(y)


class DBSCAN_Object(DataObject):

    @dispatch(int, float, float)
    def __init__(self, id: int, x: float, y: float):
        super().__init__(id, x, y)
        self._init()

    @dispatch(DataObject)
    def __init__(self, obj: DataObject):
        super().__init__(obj.id, obj.x, obj.y)
        self._init()

    def _init(self):
        self.cluster = -1
        self.is_core = False

    def get_neighbors(self, objects, eps):
        if hasattr(self, "neighbors") and (self.objects is objects):
            return self.neighbors
        self.objects = objects
        self.neighbors = []
        for obj in objects:
            if distance(self, obj) <= eps:
                self.neighbors.append(obj)
        return self.neighbors
```

DBSCAN

```
def DBSCAN(data_objects: List[DataObject], eps: float, min_pts: int, n: int) \
        -> List[List[DBSCAN_Object]]:

    clusters = []
    objects = []
    for obj in data_objects:
        objects.append(DBSCAN_Object(obj))

    for idx, obj in enumerate(objects):
        # If a object belongs to another cluster already, continue.
        if obj.cluster != -1:
            continue

        # Try to form cluster.
        obj.cluster = len(clusters)
        new_cluster = form_cluster(objects, obj, eps, min_pts, t)
        # If a cluster formed, append it.
        if new_cluster:
            clusters.append(new_cluster)
        else:
            obj.cluster = -1

    # post-process for unclustered.
    expand_cluster(objects, clusters, eps, min_pts)

    # Append empty cluster.
    clusters.append([])
    for obj in objects:
        if obj.cluster == -1:
            clusters[-1].append(obj)

    # Merge some clusters.
    merge(objects, clusters, n)

    return clusters
```

`DBSCAN` runs the main algorithm. It iterates through all objects and pick an unclustered object. This object is sent to **`form_cluster`** function and be tried to form a cluster.

When all objects are iterated, it does the post-processing. It will be explained next.

All the unclustered objects, in other word, outliers, goes to to the last cluster.

Form a Cluster

This is another important part of the algorithm. First, the function checks the seed's number of neighbors and if it's not sufficient, the seed fails to form a cluster. Then the function returns nothing indicating fail of forming cluter. In case of success, it runs BFS (Breadth-First Search) to find desenly-connected points. For all the points found, it forms a single cluster.

```python
def form_cluster(objects: List[DBSCAN_Object], seed: DBSCAN_Object,
                 eps: float, min_pts: int, t: tqdm = None) \
        -> List[DBSCAN_Object]:
    # It cannot be seed unless it is dense enough.
    if len(seed.get_neighbors(objects, eps)) <= min_pts:
        return []

    # Run BFS
    cluster = []
    queue = [seed]
    while queue:
        cluster.append(queue.pop())
        if t is not None:
            t.update(1)
        neighbors = cluster[-1].get_neighbors(objects, eps)
        if len(neighbors) <= min_pts:
            continue
        cluster[-1].is_core = True
        for n in neighbors:
            if n.cluster == -1:
                n.cluster = seed.cluster
                cluster.append(n)
                queue.append(n)

    return cluster
```

Post-Processing: Expand Cluster

After running DBSCAN, it post-processes its result. By running **`form_cluster()`** again, it make clusters to cover more objects, previously regarded as outliers. In this step, minimum points restriction is relaxed by half.

```python
def expand_cluster(objects: List[DBSCAN_Object], clusters: List[List[DBSCAN_Object]],
                   eps: float, min_pts: int):
    for obj in objects:
        if obj.cluster != -1:
            new_cluster = form_cluster(objects, obj, eps, min_pts // 2)
            if len(new_cluster) > 1:
                new_cluster.pop(0)
                clusters[obj.cluster] += new_cluster
```

Post-Processing: Merge

Futhermore, my algorithm merges some clusters to meet the number of clusters restriction, given my program parameter. For pair of objects which are regarded as core objects, the function computes distance between them and merge clusters via single-link fashion.

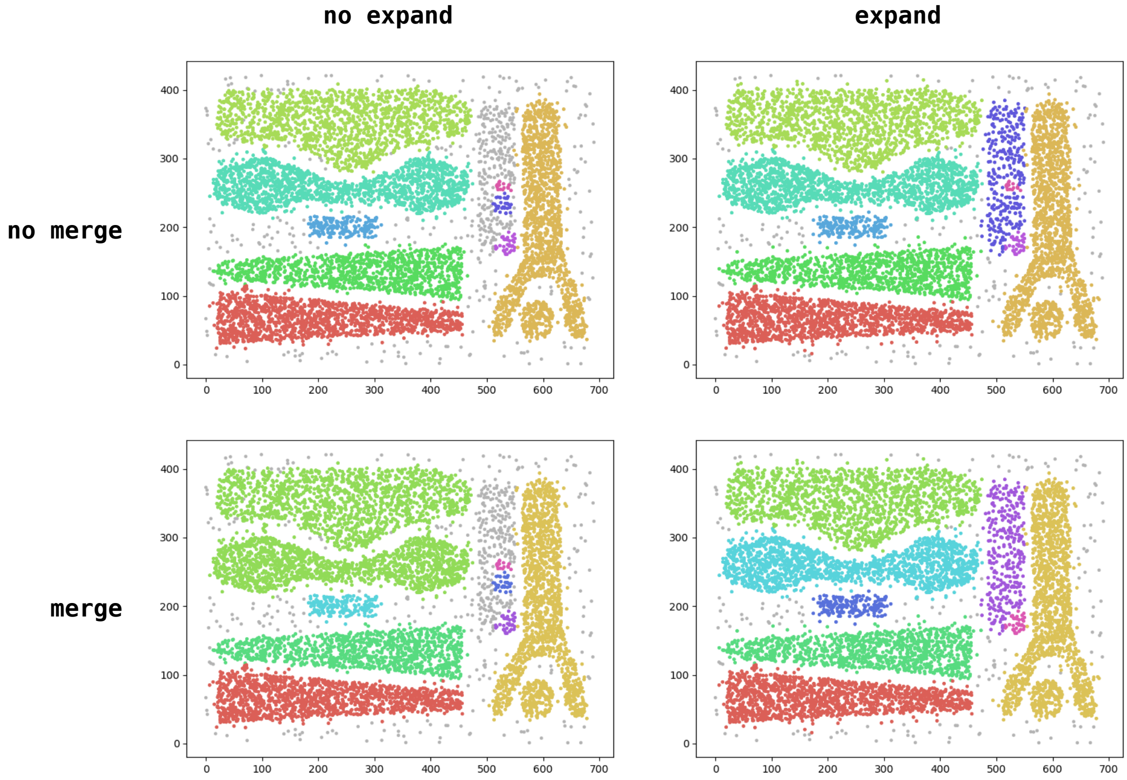
```python
def merge(objects: List[DBSCAN_Object], clusters: List[List[DBSCAN_Object]], n: int) \
        -> None:
    n_clst = len(clusters)
    t = tqdm(desc="Merging clusters",
             total=int((n_clst - 1 - n) * (len(objects) - 1) * len(objects) / 2))

    # While number of clusters exceeds the target number, `n`:
    while len(clusters) - 1 > n:
        # Merge via single link method between core points.
        min_single_link = (0, 0, 0)
        for i in range(len(objects) - 1):
            for j in range(i + 1, len(objects)):
                if objects[i].cluster != objects[j].cluster \
                        and objects[i].is_core and objects[j].is_core:
                    dist = distance(objects[i], objects[j])
                    if min_single_link[0] > dist or min_single_link[0] == 0:
                        min_single_link = (dist, objects[i].cluster, objects[j].cluster)
                t.update(1)
        clusters[min_single_link[1]] += clusters[min_single_link[2]]
        clusters.pop(min_single_link[2])
    t.close()
```

Result

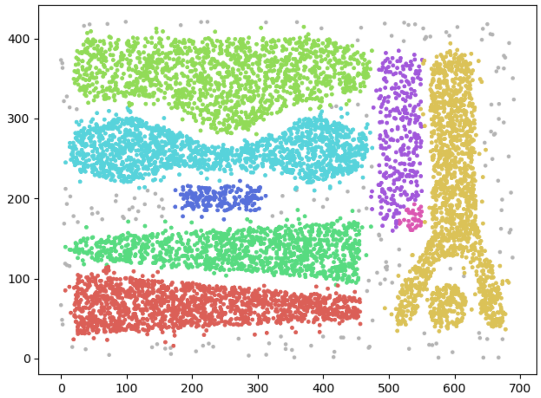
Effectiveness of Post-Processing

Images below are obtained by running my DBSCAN implementation, with `data=input1.txt, n=8, eps=15, min_pts=22.` Light grey pointes indicates outliers.

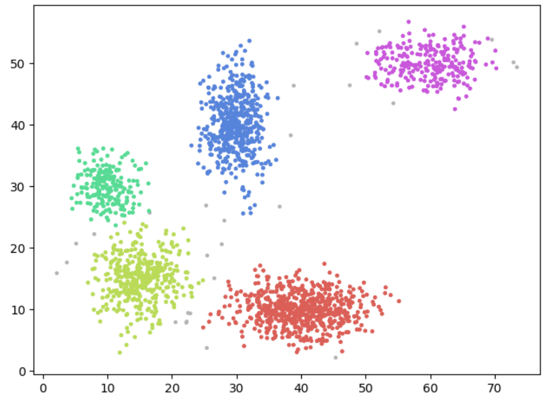
Compared to raw result, post-processing seems to improve the quality of result. However, employing merge without expand can lead mistakes.



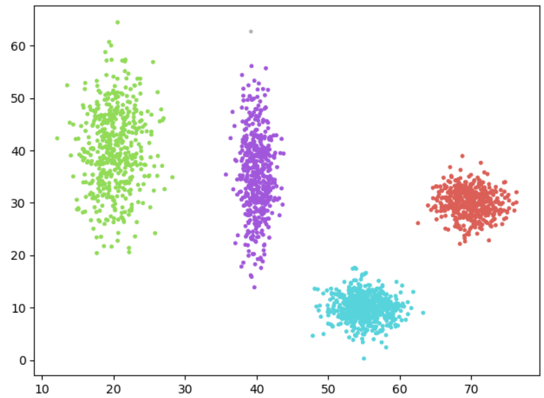
Final Results



input1.txt
n = 8
eps = 15
min_pts = 22



input2.txt
n = 5
eps = 2
min_pts = 7



input3.txt
n = 4
eps = 5
min_pts = 5