Programming Assignment #3

Index

학번: 2017041930

이름: 김선동

1. Environment & Requirements

OS는 Windows, language는 python을 사용하였다. Editor는 visual studio code를 사용하였다. 사용한 python의 버전은 3.10.4이다.
별도의 패키지를 사용하였는데 requirements.txt를 첨부하였으니 아래 명령어로 설치하면 된다.

> pip3 install -r requirements.txt

2. How to run?

> ./clustering.py input1.txt 8 15 22

> ./clustering.py input2.txt 5 2 7

> ./clustering.py input3.txt 4 5 5

위 명령어를 실행하면 input의 object들을 클러스터링한 결과가 txt파일에 저장된다.

3. Summary of my implementation

```
if __name__ == '__main__':
    if len(sys.argv) != 5:
        print("Execute the program with 4 arguments: input file name, n, Eps, MinPts")

    input_file_name = sys.argv[1]
    n = int(sys.argv[2])          # Num of clusters
    eps = float(sys.argv[3])      # Epsilon
    minPts = int(sys.argv[4])     # Minimum number of points

    # Read Data
    objects = pd.read_csv(input_file_name, sep='\t', header=None, names=['id', 'x_coordinate', 'y_coordinate'])
    objects = objects.sort_values('x_coordinate').values
```

Input file을 읽고 기타 파라미터들을 처리해주는 부분이다. 클러스터의 수와 Eps, MinPts를 입력받고 오브젝트들의 정보를 데이터프레임으로 저장한다. 이후에 한 점으로부터 다른 점들의 거리를 계산해서 Eps보다 같거나 작은 지를 알아야 하는데 이를 빨리 연산하기 위해 x좌표를 오름차순으로 먼저 정렬한다.

```python
# Get all neighbors per points w.r.t. Eps
coordinates_list = []
neighbor_list = {}
for object in objects:
    object_id = int(object[0])
    object_coordinate = np.array([object[1], object[2]])
    neighbor_list[object_id] = []

    # 가까운 점 먼저 이웃 리스트에 추가
    for neighbor in reversed(coordinates_list):
        neighbor_id = neighbor[0]
        neighbor_coordinate = neighbor[1]
        # x 좌표의 차이가 eps 보다 크다면 무조건 eps를 초과하기 때문에 검사할 필요 없음
        if object_coordinate[0] - neighbor_coordinate[0] > eps:
            break
        # x 좌표의 차이가 eps보다 같거나 작다면 eps 보다 작거나 같을 수 있기 때문에 검사할 필요가 있음
        if getDist(object_coordinate, neighbor_coordinate) <= eps:
            # eps보다 작거나 같다면 그 두 점의 이웃 리스트에 각각 추가
            neighbor_list[object_id].append(neighbor_id)
            neighbor_list[neighbor_id].append(object_id)
    # 전체 좌표 집합에 추가하기
    coordinates_list.append((object_id, object_coordinate))
```

앞서 정렬한 오브젝트들을 순차적으로 살펴보면서 거리를 구할 기본 조건을 먼저 따져본다. 만약 x좌표끼리의 거리가 Eps보다 크다면 더 이상 구할 이유가 없으므로 break하고 그렇지 않다면 거리를 구해서 eps와 비교 후 작거나 같다면 각각의 point의 neighbor에 추가해준다. 이 때, neighbor_list는 딕셔너리로 구성하여 value는 리스트로 이웃들을 저장해준다. Coordinates_list는 결국 각 오브젝트의 정보를 저장하게 되는데, x좌표 순으로 오브젝트를 정렬했으므로 가장 최근에 추가한 것이 x좌표가 가장 가까운 점이 된다. 그렇기 때문에 이를 역으로 조회하면서 Eps와 비교하며 이웃들을 구하게 된다.

```python
# Generate clusters
cluster_id = 0
cluster_list = []
visited = []
q = []

for id in neighbor_list:
    # 이미 방문했으면 continue
    if id in visited:
        continue
    # (현재 점 + 이웃의 수)가 minPts보다 작다면 continue
    if len(neighbor_list[id]) + 1 < minPts:
        continue

    visited.append(id)
    cur_cluster = [id]

    q.append(id)
    while q:
        cur_id = q.pop(0)
        # 해당 오브젝트의 이웃들 탐색
        for sub_id in neighbor_list[cur_id]:
            if sub_id in visited:
                continue
            else:
                visited.append(sub_id)      # 방문 mask
                cur_cluster.append(sub_id)  # 현재 cluster에 추가
                # minPts 이상이라면 해당 오브젝트의 이웃들도 조사
                if len(neighbor_list[sub_id]) + 1 >= minPts:
                    q.append(sub_id)
    # 탐색할 수 있는 모든 오브젝트 탐색했다면 cluster_list에 추가
    cluster_list.append(cur_cluster)

cluster_list.sort(key=lambda x : len(x))
```

이제 모든 점들에 대해 Eps를 만족하는 이웃들을 구해 놓았으니 MinPts를 만족하는 클러스터를 만들어줄 차례이다. Visited 리스트를 만들어서 방문한 점들은 이 리스트에 추가하여 후에 in으로 방문한 오브젝트인지 아닌지를 확인해준다. 만약 MinPts를 만족한다면 core condition을 만족한 오브젝트로, 해당 오브젝트의 이웃들을 전부 탐색하며 방문하지 않았던 점이라면 masking을 다 해주고 현재 클러스터에 추가 해준다. 이 때 사용된 알고리즘은 BFS로 하나의 점이 갖고 있는 이웃들을 전부 방문해주고 다시 다음 점으로 넘어가는 그런 형태이다. 그렇게 얻은 cluster_list를 갖고 있는 길이의 순서대로 정렬해준다.

```
# If your algorithm finds m clusters for an input data and m is greater than n (n = the
# number of clusters given), you can remove (m-n) clusters based on the number of objects
# within each cluster. In order to remove (m-n) clusters, for example, you can select (m-n)
# clusters with the small sizes in ascending order
# You can remove outlier. In other words, you don't need to include outlier in a specific cluster
while len(cluster_list) > n:
    cluster_list.pop(0)

# Save file
printFiles(input_file_name, n, cluster_list)
```

정렬해주는 이유는, 만약 주어진 클러스터의 수보다 크다면 더 많은 이웃들을 갖고 있는 클러스터를 우선해야 하기 때문이다. 때문에 n보다 크다면 오름차순이므로 첫번째 클러스터를 계속 pop해주면 된다.

```
def getDist(a, b):
    return np.sqrt(np.dot(a-b, a-b))

def printFiles(input_file_name, n, clusters):
    for cluster_id in range(n):
        cluster = []
        if cluster_id < len(clusters):
            cluster = clusters[cluster_id]
        cluster.sort()
        output_file_name = input_file_name.split('.')[0] + '_cluster_' + str(cluster_id)+'.txt'
        np.savetxt(output_file_name, cluster, fmt='%d')
```

위 두 함수는 거리를 구하는 함수와 마지막에 결과물을 저장하는 함수이다. 거리는 그냥 두 벡터의 차이끼리의 내적과 다름 없다. 결과물은 저장한 cluster_list에 저장된 것들끼리 저장해주면 된다.

4. Testing

```
C:₩Users₩mok03₩GitHub₩ITE4005_Data-Science₩DBSCAN₩test-2>PA3.exe input1
98.97037점
C:₩Users₩mok03₩GitHub₩ITE4005_Data-Science₩DBSCAN₩test-2>PA3.exe input2
94.89474점
C:₩Users₩mok03₩GitHub₩ITE4005_Data-Science₩DBSCAN₩test-2>PA3.exe input3
99.97736점
```