Implementation of DBSCAN

2018008395 박정호

1. Algorithm

이번 과제는 clustering algorithm 중 하나인 DBSCAN을 구현하는 것이었다. 강의에서 설명된 Logical Flow를 그대로 반영하려고 노력했으며, 실행 시간을 줄이기 위해 주어진 데이터를 정렬하고, BFS 알고리즘을 통해 density-reachable한 점을 최단 거리로 구하려 했다.

기본적으로 BFS 알고리즘을 쓰기 위해서는 주어진 데이터를 adjacency matrix 혹은 adjacency list로 바꿀 필요가 있었다. 그러나, 필연적으로 O(N^2)가 되는 이 과정을 naive하게 구현할 경우 너무 긴 시간이 걸렸다. (input1에 대해서 대략 2분에서 3분 정도 소요된다.) 이 시간이 실행시간의 대부분을 차지하기 때문에, 최대한 빨리 adjacency list를 만들 수 있도록 최적화를 했다. 주어진 데이터를 X 좌표 순으로 정렬하고, 현재 보고 있는 점과의 X 좌표 차이가 EPS 보다 클 경우, 이후의 점에 대해서는 adjacent 여부를 판단하지 않도록 했다. 이런 식으로 adjacent point 탐색의 범위를 줄이는 것으로 비약적인 성능 상승을 볼 수 있었다. (input1에 대해서 대략 10초 정도 소요된다.)

Adjacency list를 구성하는데 성공했다면, clustering을 할 단계이다. DBSCAN은 core condition을 만족하는 점에 대해서 density reachable한 점을 모두 찾아 한 cluster에 할당하고, 다음 core condition을 만족하는 점을 찾는 과정을 반복한다. Core condition check는 단순히 adjacency list의 해당 행의 길이를 보면 된다. Adjacency list 자체가 해당 점에서 EPS 이내에 존재하는 점들로 만들어져 있기 때문이다. Density reachable한 점을 찾는 것 또한 간단하다. Adjacency list를 가지고 BFS를 수행하면서, 만나는 모든 점이 같은 cluster에 속하도록 하면 된다. 물론 별도의 자료구조 없이 재귀함수만으로 구현되는 DFS로도 이는 구현할 수 있으나, 그래프의 구조에 따라서 필요이상으로 전체 탐색에 오래 걸릴 수 있어, 최단 거리의 탐색을 보장하는 BFS를 사용했다. 한번의 BFS가 끝나면, 방문하지 않은 점 중에서 core condition을 만족하는 점을 다시 찾아서 BFS를 수행하고, 이 과정을 모든 점에 방문할 때까지 반복하면 DBSCAN의 수행이 끝나게 된다.

다만 이 과제에서는 정확하게 N개의 cluster를 구성하는 것을 요구했기에, DBSCAN의 결과에 따라서 너무 많은 cluster를 구성했을 수도, 너무 적은 cluster를 구성했을 수도 있었다. 따라서, 이에 대한 예외 처리를 추가했다. N개보다 많은 cluster를 찾았을 경우, 크기가 큰 순서대로 N개의 cluster만 남기고 나머지는 outlier로 처리했고, N개보다 적은 cluster를 찾았을 경우, 남은 개수를 빈 cluster로 채우도록 처리하는 방식을 사용했다.

2. Implementation

1. DBSCAN Class

1. 기본 구성

```
# DBSCAN clustering algorithm class
class DBSCAN:
    # Constructor
    def __init__(self, N, EPS, MinPTS):
        self.N = N
        self.EPS = EPS
        self.MinPTS = MinPTS

        self.adjacent_list = {}

    # read data and make adjacency list with this data
    def set_data(self, filename):
        ...

    # Get cluster
    # with BFS algorithm
    def get_clusters(self):
        ...
```

좀 더 가독성 있는 구현을 위해 class를 구현했다. 생성자는 단순히 주어진 parameter를 인스턴스에 저장하는 동작과, 빈 adjacency list를 만들고 끝난다. 이 빈 adjacency list는 후술할 set_data에서 데이터를 읽으면서 구성하게 된다.

2. Data Handling

```
    # read data and make adjacency list with this data
    def set_data(self, filename):
        start_time = time.time()
        data_set = pd.read_csv(filename,
                               sep='\t',
                               header=None).values

        sort_idx = data_set.T[1].argsort()
        data_set = data_set[sort_idx]

        points = []
        for obj in data_set:
            obj_id = int(obj[0])
```

```
            obj_point = np.array([obj[1], obj[2]])
            self.adjacent_list[obj_id] = []

            for neighbor_obj in reversed(points):
                neighbor_id = neighbor_obj[0]
                neighbor_point = neighbor_obj[1]
                if obj_point[0] - neighbor_point[0]  > self.EPS:
                    break

                if distance(obj_point, neighbor_point) <= self.EPS:
                    self.adjacent_list[obj_id].append(neighbor_id)
                    self.adjacent_list[neighbor_id].append(obj_id)

            points.append((obj_id, obj_point))

        end_time = time.time()
        print('Data Handling Time :', end_time - start_time)
```

주어진 filename을 가지고 읽어온 데이터 셋을 이용해서 adjacency list를 구성하는 함수이다. 위에서 언급했듯이, 이 과정이 전체 프로그램의 실행 시간의 대부분을 차지하기 때문에, 실행 시간의 단축을 위해 데이터를 X좌표에 대해서 정렬했다. 이후, 각 점에 대해서 자신과 거리가 EPS이하인 점을 찾아서 adjacency list에 추가하도록 했으며, 이때 X 좌표의 차이가 EPS보다 커진 시점에는 더 이상의 탐색을 하지 않도록 했다. Reversed가 쓰인 이유는, 두번째 for 문의 iteration 대상은 이전에 지나온 점들인데, 현재 점과 가까운 순으로 보기 위해서는 역순으로 반복문을 돌리는 것이 필요했기 때문이다.

3. Training

```
    # Get cluster
    # with BFS algorithm
    def get_clusters(self):
        clusters = []

        visited = set()
        queue = deque([])
        start_time = time.time()
        for obj_id in self.adjacent_list:
            if obj_id in visited or len(self.adjacent_list[obj_id]) + 1 < self
.MinPTS:
                continue

            now_cluster = [obj_id]
            visited.add(obj_id)
            queue.append(obj_id)

            while queue:
                now = queue.popleft()
```

```
                for nxt in self.adjacent_list[now]:
                    if not (nxt in visited):
                        visited.add(nxt)
                        now_cluster.append(nxt)
                        if len(self.adjacent_list[nxt]) + 1 >= self.MinPTS:
                            queue.append(nxt)

            clusters.append(now_cluster)

        while len(clusters) < self.N:
            clusters.append([])

        clusters.sort(key=lambda x:len(x), reverse=True)

        while len(clusters) > self.N:
            del clusters[-1]

        end_time = time.time()

        print('Traning Time :', end_time - start_time)
        return clusters
```

단순히 각 점마다 BFS 알고리즘을 수행해서 cluster를 찾는 함수이다. 점이 탐색 대상이 될 조건은 아직 방문하지 않았고, core condition을 만족할 경우이다. 만약 core condition을 만족하지 못한 border point를 만날 경우, cluster에만 추가하고, 그 점을 탐색의 대상으로 하지는 않았다.

3. 출력 처리

```
# write clustering result to output file
def output_process(file_prefix, clusters, n):
    for cluster_id in range(n):
        cluster = []
        if cluster_id < len(clusters):
            cluster = clusters[cluster_id]
        filename = file_prefix+'_cluster_'+str(cluster_id)+'.txt'
        file = open(filename, mode='w')

        for obj_id in cluster:
            file.write(str(obj_id))
            file.write('\n')

        file.close()
```

명세에 주어진 조건에 맞도록 파일에 clustering 결과를 출력하는 함수이다.

4. main 함수

```
def main(argv):
    if len(argv)<5:
        print('PLEASE, give 4 arguments (input filename, N, EPS, MinPTS)')
        return

    file_prefix = Path(argv[1]).stem

    dbscan = DBSCAN(int(argv[2]), float(argv[3]), int(argv[4]))

    start_time = time.time()
    dbscan.set_data(argv[1])
    clusters = dbscan.get_clusters()
    end_time = time.time()

    print('DBSCAN Time :', end_time - start_time)

    output_process(file_prefix, clusters, int(argv[2]))
```

위에서 구현한 모든 함수를 사용하는 함수이다. file prefix는 output file을 출력할 때 사용하는 것으로, 파일명에서 확장자만 제거한 것이다.

3. Usage

우선 전체 코드는 python으로 작성되었으며, 버전은 3.8.2이다.

numpy 와 pandas를 활용했으며, 버전은 각각 1.18.2, 1.2.3이다.

또한 python은 인터프리터 언어이기에 별도의 컴파일 명령어나 Makefile을 쓰지 않았으며, 아래의 밑줄 그어진 부분의 명령어로 실행할 수 있다. 필자는 .py 파일에 대한 환경변수 설정이 되어있지 않았기 때문에 파일명만으로 실행하는 것이 불가능했음을 밝힌다.

```
PS C:\Users\pch68\2021_ite4005_2018008395\assignment3> python .\clustering.py .\input1.txt 8 15 22
Data Handling Time : 9.931488275527954
Traning Time : 0.020974397659301758
DBSCAN Time : 9.956418514251709
```

4. Result

```
PS C:\Users\pch68\2021_ite4005_2018008395\assignment3> python .\clustering.py .\input1.txt 8 15 22
Data Handling Time : 9.066762685775757
Traning Time : 0.025935649871826172
DBSCAN Time : 9.094728469848633
PS C:\Users\pch68\2021_ite4005_2018008395\assignment3> python .\clustering.py .\input2.txt 5 2 7
Data Handling Time : 1.0592074394226074
Traning Time : 0.00794076919555664
DBSCAN Time : 1.0681462287902832
PS C:\Users\pch68\2021_ite4005_2018008395\assignment3> python .\clustering.py .\input3.txt 4 5 5
Data Handling Time : 2.591104030609131
Traning Time : 0.033911943435668945
DBSCAN Time : 2.6250159740448
PS C:\Users\pch68\2021_ite4005_2018008395\assignment3> .\PA3.exe input1
98.97037점
PS C:\Users\pch68\2021_ite4005_2018008395\assignment3> .\PA3.exe input2
94.89474점
PS C:\Users\pch68\2021_ite4005_2018008395\assignment3> .\PA3.exe input3
99.97736점
```

전체적으로 10초 이내에 결과가 모두 나오는 것을 볼 수 있으며, 정확도는 대략 95~100% 정도임을 알 수 있다. Clustering 결과는 아래와 같다. 검은 색 점은 outlier임을 밝힌다.