

# Programming Assignment #3

2017030328 조지훈

## Compilation method and environment

사용 에디터: Visual Studio Code & VIM editor

개발 OS: Ubuntu 20.04.1 LTS (Windows 10 PRO 21H1 버전에서 WSL2 이용)

언어: Python 3.9.7

장치 사양

프로세서 Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz


RAM 16.0GB

시스템 종류 64 비트 운영 체제, x64 기반 프로세서

## 실행 방법

```
> python3 clustering.py input2.txt 5 2 7
```

리눅스 우분투 기준으로 과제 명세서와 동일하게 4개의 arguments를 받아 작동하고 1번째는 input 텍스트 파일의 이름을, 그 이후에는 조건을 입력하는데, 2번째, 4번째 arguments는 자연수를 값으로 입력해야 하고, 3번째 arguments는 실수를 입력 받을 수 있다. 결과물은 주어진 명세서처럼 파일로 결과가 출력된다. 다음은 과제에서 주어진 테스트에 대한 프로그램 실행 결과물의 일부이다.

 input2\_cluster\_1.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
1
8
9
13
14
18
21
38
```

## Algorithm 설명

DBSCAN (밀도 기반 클러스터링)을 구현한 과제이다. DBSCAN은 Density-based spatial clustering of applications with noise의 약자로 말 그대로 data들의 밀도를 이용해서 분류하는 방법이다. 기본 원리는 어느 점을 기준으로 정해진 반경 내에 자신을 포함한 점이 정해진 개수 이상 있으면 하나의 동일한 군집으로 인식하는 방식이다.

알고리즘을 설명하기 앞서 개념을 설명하면 data를 3개로 나눌 수 있다. core point, border point, noise point로 나눌 수 있는데, core point는 조건을 만족하는 조밀한 중심의 점, border point는 조건을 만족하지 못해 조밀하지 않지만 주변 점들 중에서 core point가 존재하는 점, noise point 둘을 제외한 나머지 점을 의미한다.

core point에 대하여 근처에 있는 점들은 해당 core point에 대하여 directly density-reachable하다고 하고, 한 core point에 대하여 어떤 점이 그 사이에 있는 점들에 대해 directly density-reachable하다면 core point와 어떤 점은 density-reachable하다.

DBSCAN Algorithm은 어떤 점 p가 core point 여부를 확인하고 core point라면 점 p에 대해 모든 density-reachable을 찾는다. 이 때 찾아진 점들은 모두 mark를 하고, 같은 cluster로 분류한다. 그리고 선택되지 않은 점들 중에서 선택하여 다시 작업을 진행한다. 만약에 더 이상 선택할 점이 없다면 알고리즘을 종료하게 된다.

## Code 설명

해당 코드는 대략적으로 분류할 data를 읽고, 초기화하는 부분, 읽은 data를 분류하는 부분, 분류한 data를 출력하는 부분으로 구분할 수 있다.

input data는 [object\_id\_i], [x\_coordinate], [y\_coordinate]로 이루어져 있다. 이 때 [object\_id\_i]는 i번째 object를 의미하는 것으로, 같은 data set에서 각각의 object마다 고유한 값을 가지고 있고, 순차적으로 구성이 되어있다.

```
# argument parser
parser = argparse.ArgumentParser()
parser.add_argument("input", help="input file name", type=str)
parser.add_argument("clust_num", help="number of clusters for the corresponding input data", type=int)
parser.add_argument("esp", help="maximum radius of the neighborhood", type=float)
parser.add_argument("minpts", help="minimum number of points in an Eps-neighborhood of a given point", type=int)
args = parser.parse_args()
```

우선적으로 argument parser이다. 4개의 argument를 받고 각각의 type을 정해주었다.

명세서의 입력과 동일한 입력을 받을 수 있게 구현하였다.

```

for point in data:
    point[1] = float(point[1])
    point[2] = float(point[2])
    point_type.append(-1) # -1 = 'outlier' , 0 <= 'cluster'
    processed.append(False)

```

위 코드는 읽은 data를 초기화하는 코드이다. 기본적으로 txt를 readlines()를 이용해서 읽으면 해당 변수의 타입은 string의 형태로 되어 있다. 그렇기에 추후에 계산이 용이할 수 있도록 점들의 [x\_coordinate], [y\_coordinate]의 type을 float로 변환하고, 점의 개수와 동일하게 cluster를 표현할 point\_type과 mark여부를 확인하는 processed 변수를 추가하였다 point\_type의 경우 기본 outlier (noise point)를 -1로 설정하고, 추후에 cluster가 결정될 때 값이 변한다.

```

for point in data:
    point_data = np.array([point[1], point[2]])
    if processed[int(point[0])] != True:
        mem_num = clustering(point[0], point_data, data, point_type, processed)
        if mem_num > 0:
            able_cluster.append([now_cluster, mem_num])

```

마크가 되어 있지 않은 점 하나 하나를 선택해가면서 clustering(point\_id, point, point\_list, point\_type, point\_processed) 함수를 이용해 cluster를 분석하는 부분으로 clustering(...) 함수가 해당 cluster에 해당하는 점의 개수를 반환한다. (outlier라면 0을 반환) 이를 이용해서 cluster의 점 개수가 0개 이상이라면 able\_cluster라고 하는 변수에 해당 cluster의 numbering과 포함된 점의 개수를 저장하게 된다.

```

write_cluster = []
if len(able_cluster) > args.clust_num:
    while len(write_cluster) < args.clust_num:
        max = 0
        temp = [-1, 0]
        for i in able_cluster:
            if i[1] > max:
                max = i[1]
                temp = i
        write_cluster.append(temp)
        able_cluster.remove(temp)
else:
    write_cluster = able_cluster

fileable = []
for i in write_cluster:
    fileable.append(i[0])

```

선택적으로 작동하게 되는 코드이다. 만약에 분류한 cluster의 개수가 지정된 cluster의 개수보다 많다면 포함된 point가 가장 많은 것부터 선택한다.

```

for i in range(0, args.clust_num):
    txt = args.input.split('.')[0] + '_cluster_' + str(i) + '.txt'
    file = open(txt, 'w')
    output_file.append(file)

for point_id in range(len(data)):
    point = data[point_id]
    if point_type[point_id] in fileable:
        output_file[fileable.index(point_type[point_id])].write(str(point_id) + '\n')

for f in output_file:
    f.close()

```

분류된 cluster를 파일로 저장하는 부분이다. 지정된 cluster의 개수만큼 파일을 생성하고, 각각의 point를 확인하여 해당 point에 맞는 파일에 id를 입력한다.

## clustering(point\_id, point, point\_list, point\_type, point\_processed)

point\_id로 들어온 point를 중심으로 cluster를 분류하는 함수.

```

for i in point_list:
    dist = distance(point, np.array([i[1], i[2]]))
    if args.esp >= dist:
        neighbor.append(i)

if len(neighbor) >= (args.minpts): #check core
    now_cluster += 1
    #print(point_id + ' is in ' + str(now_cluster))
    point_type[pid] = now_cluster
    point_processed[pid] = True
else: #if point is not core, end function
    return 0

```

입력으로 들어온 point에 대해 자신을 포함한 모든 point와의 거리를 측정하여 esp 이하의 거리를 가진 점들을 이웃으로 모아두고, 이웃의 개수가 minpts 이상이라면 core point로 감지를 하게 되어 하나의 cluster를 구성하고 해당 core point를 마크 한다. 만약에 입력으로 받은 해당 point가 core point가 아니라면 바로 함수를 종료하고 0을 반환하게 된다.

```

for i in neighbor:
    point_type[int(i[0])] = now_cluster #set neighbor cluster

for i in neighbor: #secend and after than point of cluster
    near_point = []
    if point_processed[int(i[0])] != True:
        point_processed[int(i[0])] = True
        for j in point_list:
            dist = distance(np.array([i[1], i[2]]), np.array([j[1], j[2]]))

            if args.esp >= dist:
                near_point.append(j)

        if len(near_point) >= (args.minpts):
            #print(i[0])
            for temp in near_point:
                if temp not in neighbor:
                    point_type[int(temp[0])] = now_cluster
                    neighbor.append(temp)
            #clustering(i[0], np.array([i[1], i[2]]), point_list, point_type, point_processed)

return len(neighbor)

```

앞에서 구해진 이웃들의 cluster를 설정하고, 이웃들에 대해서도 core point 여부를 확인한 후 이웃도 core point라면 이웃의 이웃들도 cluster를 설정하고, 이웃 목록에 추가를 해주는데, 이 때 중복이 되지 않게 추가를 해준다. 더 이상 이웃 목록에 남은 point가 없을 때까지 함수를 진행하게 되고, 최종적으로는 cluster에 포함된 point의 크기를 반환한다.

## distance(pointA, pointB)

두 point 사이의 거리를 반환해주는 함수. numpy의 내부 함수를 이용.

```

dist = np.sqrt(np.sum(np.square(pointA-pointB)))
return dist

```

다음은 테스트 결과이다.

```
> python3 clustering.py input3.txt 4 5 5
> python3 clustering.py input2.txt 5 2 7
> python3 clustering.py input1.txt 8 15 22
> ./PA3.exe input1
98.97691점%
> ./PA3.exe input2
94.86023점%
> ./PA3.exe input3
99.97736점%
```

테스트 결과 점수로 보면 어느 정도 명세서가 요구한 점수에 비슷했다고 생각한다. 하지만 cluster를 분류하는 과정에서 point 사이의 거리를 구할 때 서로 다른 모든 point의 조합에 대해서 모든 거리를 구하는 과정이 있기 때문에 runtime이 오래 걸리는 것 같다.

input1 기준으로 10분 30초 정도 걸렸고, input2, input3 기준으로는 34-5초 정도 걸렸다. input의 data 개수 차이를 생각하면 대충  $O(n^2)$  정도의 복잡도를 가지고 있는 것 같다.