2014000082 구희강

Data Science

PROGRAMMING ASSIGNMENT #3 - DBSCAN

Code Description and Algorithm

<main>

```
if __name__ == "__main__":
    input_file = sys.argv[1]
    n = int(sys.argv[2])
    eps = int(sys.argv[3])
    minPts = int(sys.argv[4])
    data = readData(input_file)
    DBSCAN(data,eps,minPts)
    sorted_cluster = sortCluster(data)
    for i,(num,_) in enumerate(sorted_cluster):
        name = input_file.split('.')[0] + "_cluster_{}.txt".format(i)
        writeCluster(data,num,name)
```

This is the flow of entire code. It requires 4 arguments: input file name, n(number of clusters for the corresponding input data), eps(maximum radius of the neighborhood), minPts(the minimum number of points required to form a dense region). First, make ‘data’ from input file. Then call DBSCAN() function with ‘data’. Finally, truncate cluster by given ‘n’ and print out output files.

<readData()>

```
def readData(name):
    fin = open(name,'r')
    data = []
    while True:
        tmp = fin.readline()
        if not tmp:
            break
        tmp = list(tmp.strip('\n').split('\t'))
```

```
        for i in range(len(tmp)):
            tmp[i] = float(tmp[i])
        tmp.append(0)
        data.append(tmp)
    fin.close()
    return data
```

This function reads input text file and makes list type data whose element is also a list of point's id(starts from zero), x-coordinate, y-coordinate, and cluster info (default value is 0).

<distance()>

```
def distance(p1,p2):
    return math.sqrt((p1[1]-p2[1])**2 + (p1[2]-p2[2])**2)
```

This function calculates and return Euclidean distance of given two points.

<findNeighbors()>

```
def findNeighbors(data,point,eps):
    neighbors = []
    for x in data:
        if distance(point,x) < eps:
            neighbors.append(int(x[0])) # save id
    return neighbors
```

This function finds and returns neighbors of given point as a list data type. The list's element is only point's id, not entire value.

<DBSCAN>

```
def DBSCAN(data,eps,minPts):
    cluster_label = 0
    for i, point in enumerate(data):
        if point[-1] == 0:
            neighbors = findNeighbors(data,point,eps)
            if len(neighbors) < minPts:
                data[i][-1] = -1
            else:
                cluster_label += 1
                # cluster
                data[i][-1] = cluster_label # set cluster label
                j = 0
```

```
            while j < len(neighbors):
                tmp = neighbors[j] # list [id,x,y,cluster]
                if data[tmp][-1] == -1:
                    data[tmp][-1] = cluster_label
                elif data[tmp][-1] == 0:
                    data[tmp][-1] = cluster_label
                    tmp_neighbors = findNeighbors(data,data[tmp],eps)
                    if len(tmp_neighbors) >= minPts: # is core point
                        for x in tmp_neighbors:
                            if x not in neighbors:
                                neighbors.append(x)
                j += 1
```

This is the core part of DBSCAN algorithm. cluster_label is for numbering clusters. It increases when DBSCAN found new cluster. At readData() function, the data's all points' cluster info was initialized as 0(unlabeled). While iterating over the 'data', check the cluster info of point. If cluster info is 0(unlabeled), then DBSCAN starts to check whether this point could be core point or not. First, find neighbors of the point with given eps value. If the number of neighbors is less than minPts value, the point is not core point but regarded as outliers (-1). However, it can be part of a cluster later. If the number of neighbors is larger than minPts value, the point is core point. All points in the Cluster should be labeled as cluster_label. And now DBSCAN should check the neighbor points are core point or not. This is because if a point is found to be core point ifself, it's neighors are also part of that cluster. By this algorithm, the whole points in data are classified as 'cluster_label' number of clusters.

<sortCluster()>

```
def sortCluster(data,n):
    cluster_count = {}
    for i in range(len(data)):
        if int(data[i][-1]) == -1: # outliers
            continue
        if not int(data[i][-1]) in cluster_count:
            cluster_count[int(data[i][-1])] = 1
        else:
            cluster_count[int(data[i][-1])] += 1
    sorted_cluster = sorted(cluster_count.items(),
                            key=operator.itemgetter(1),reverse=True)
    sorted_cluster = sorted_cluster[:n]
    return sorted_cluster
```

This function counts the number of points in each clusters, then truncate the top n number of large clusters.

<writeCluster()>

```
def writeCluster(data,num,name):
    tmp = []
    # ‘input#_cluster_0.txt’
    fout = open(name,'w+')
    for point in data:
        if point[-1] == num:
            tmp.append(str(int(point[0]))) # id
    fout.write('\n'.join(tmp))
    fout.close()
```

This function writes each cluster as a text file.

How to run program

≫ **python clustering.py [input file name]** [n] [eps] [$minPts$]

Results

Test environment - Windows 10

Test requirement - Python 3