[Assignment report #3]

Clustering by DBSCAN Algorithm

2012023871
경영학과
강민철

1) Summary of Algorithm

Clustering algorithm classifies input data which doesn't have classification label. Initially, all objects in a given data set D are marked as "unvisited." DBSCAN randomly selects an unvisited object p, marks p as "visited," and checks whether the ε-neighborhood of p contains at least MinPts objects. If not, p is marked as a noise point. To find the next cluster, DBSCAN randomly selects an unvisited object from the remaining ones. The clustering process continues until all objects are visited.

2) Detailed Description of Codes

[Library Import]

```
import sys
import math

# Global variable for checking and noise
UNCHECKED = None
NOISE = -1
```

- 'sys' enables a .py file to take arguments on the bash level.
- 'math' are used to calculate square root and power for Euclidean distance.
- UNCHECKED, NOISE are global variable for checking and noise

[Main function]

```
if __name__ == "__main__":
    input_name = sys.argv[1]
    n = int(sys.argv[2])
    eps = int(sys.argv[3])
    min_points = int(sys.argv[4])


    # Read input file
    train = FileReader(input_name)

    # Run dbscan algorithm and draw cluster information
    labels = dbscan(train, eps, min_points)
    # In case the number of cluster is larger than n
    # Modify the extra cluster(with least number of elements) to final one
    for i in xrange(len(labels)):
        if labels[i] > (n - 1):
            labels[i] = (n - 1)
    print "Excessive cluster truncated"
    name = input_name.replace('.txt', '')

    # For each cluster in labels make output file
    for cl_id in xrange(n):
        file_name = name + '_cluster_' + str(cl_id) + '.txt'
        out_file = open(file_name, 'w')
        Result = write_file(labels, out_file, cl_id)
        out_file.close()

    print "Finished"
```

First few lines are for setting the sys.argv for taking arguments from user. Then it reads dataset with FileReader function. Next step is to run DBSCAN algorithm on input data.

As the DBSCAN algorithm does not take the number of cluster beforehand, there might be a chance that the resulting number of cluster exceeds the given number of cluster or 'n'. To handle that case, below code just reallocate the clusters which exceeds the number of cluster as the last cluster.

Finally, the last block of main function, creates the corresponding number of output file and write the index of each cluster.

[FileReader function]

```
def FileReader(file_path):

    #Read each row
    #in accordance with the given format
    df = []
    file_iter = open(file_path, 'rU')
    for row in file_iter:
        row = row.strip().rstrip('\t')
        tup = row.split('\t')
        df.append(tup)
    return df
```

This function opens the train and test data file by iterating each row of the data. Watch that the delimeter is '₩t'.

[dbscan function]

```
def dbscan(data, eps, min_points):
    # Designate first cluster id as 0
    cluster_id = 0
    n_points = len(data)
    print "DBSCAN Clustering initializing..."
    # Make cluster labels with same length
    cluster_labels = [UNCHECKED] * n_points
    for point in xrange(n_points):
        # Start DBSCAN if the point is unchecked
        if cluster_labels[point] == UNCHECKED:
            if diffusion(data, cluster_labels, point, cluster_id, eps, min_points):
                print "Cluster No.%s classified" % (cluster_id)
                cluster_id = cluster_id + 1

    return cluster_labels
```

First it starts with initializing cluster_id as 0 because the output file starts at cluster 0. Then it calculates the total length of input data, and proceeds to make cluster labels with the same length. By looping input data, it runs the diffusion function which recursively expands the family of each given point. This function repeats this process till the end unless the point is checked or noise.

[diffusion function]

```
def diffusion(data, cluster_labels, point, cluster_id, eps, min_points):
    # Make family with given point
    family = make_family(data, point, eps)
    # Family number smaller than minPts
    if len(family) < min_points:
        # Classify as noise
        cluster_labels[point] = NOISE
        return False
    else:
        # Else, classify as current cluster_id
        cluster_labels[point] = cluster_id
        # Classify child as current cluster_id, either
        for id in family:
            cluster_labels[id] = cluster_id

        # Expand family with child of family

        while len(family) > 0:
            # Choose one point in family
            current_point = family[0]
            # Recursively generate extended family
            # With child point
            results = make_family(data, current_point, eps)

            # Append extended family to original family
            # if the result is bigger than minPts and
            # each member of it is UNCHECKED or NOISE
            if len(results) >= min_points:
                for i in xrange(len(results)):
                    result_point = results[i]
                    if cluster_labels[result_point] == UNCHECKED or cluster_labels[result_point] == NOISE:
                        family.append(result_point)
                        cluster_labels[result_point] = cluster_id
            # Check other family member
            family = family[1:]

        return True
```

Diffusion function runs make_family function to its given points. If the number of family is smaller than minPts, it classifies the point as NOISE. Else, it classifies the point as current cluster_id which is given as argument from the previous function. Then it expands the family by applying make_family function to each child of original family. By looping the family member, it recursively generates extended family if the result is bigger than minPts and each member of extended family is unchecked and noise. It also appends the member of extended family which meets the condition above. Therefore, the member of original family changes dynamically, and this process is repeated until there are no more elements in original family.

[make_family function]

```
def make_family(data, point, eps):
    n_points = len(data)
    family = []
    # Check every other points
    for i in xrange(n_points):
        # Whether they are neighbor or not
        if is_neighbor(data[point][1:], data[i][1:], eps):
            # Add to family if it is neighbor
            family.append(i)
    return family
```

This function initiates blank variable named family. Then it checks every points of input data whether they are neighbor or not. If the other point is neighbor of current point or within the distance of epsilon, the function appends them to family and return that result to diffusion function.

[is_neighbor function]

```
def is_neighbor(p, q, eps):
    # Return true when the distance
    # to the target point is within epsilon
    return euclidean_dist(p, q) < eps
```

This function determines whether the target point is within the distance of epsilon from the given point and then return True if it is.

[Euclidean_dist function]

```
def euclidean_dist(p, q):
    # As we read input file as string, should transform data type to float
    x_1 = float(p[0])
    y_1 = float(p[1])
    x_2 = float(q[0])
    y_2 = float(q[1])

    # Return Euclidean distance of given two points
    return math.sqrt(math.pow(x_1 - x_2, 2) + math.pow(y_1 - y_2, 2))
```

As we read input files as string data type, we should transform each x and y value of point to float type. Then it calculates the Euclidean distance of two given points and return the result.

[write_file function]

```
def write_file(labels, f, cluster_id):
    # Loop through each element
    # and separate with its cluster id

    for i in xrange(len(labels)):
        if labels[i] == cluster_id:
            row = ""
            row += "%s\n" % (i)
            f.write(row)

    return f
```

For each given cluster_id, this function loops through each elements of cluster labels data and writes the matching index of cluster to the output file.

3) Instructions for compiling the source codes

[Step 1]: copy the repository and change directory to /project_DBSCAN

```
git clone http://hconnect.hanyang.ac.kr/2017_ITE4005_10065/2017_ITE4005_2012023871.git
```

- Copy my gitlab repository by using *git clone* or download the *zip file* from the website. [http://hconnect.hanyang.ac.kr/2017_ITE4005_10065/2017_ITE4005_2012023871]

```
cd DS_git/2017_ITE4005_2012023871/Programming_Assignment_3/project_DBSCAN/
```

- go the the project_DBSCAN directory by 'cd' command.

[Step 2] : Run the code by python (2.7 ONLY)

```
python clustering.py input1.txt 8 15 22
python clustering.py input2.txt 5 2 7
python clustering.py input3.txt 4 5 5
```

run $python clustering.py input1.txt 8 15 22 on terminal or command line

[Caution] python should be 2.7 # python3 will cause error

Then you will see the output files have been made in the same directory.