Data Mining and Web algorithm

Lab Assignment 7:

[09-14 May, 2022]

Patil Amit Gurusidhappa
19104004
B11

1. Implement k-mean clustering using the Euclidean/Manhattan Distance metric to cluster redundant/repeated points into the same cluster. You are expected to do the K-means implementation by yourself, so please do not use any external library that has K-means implementation in your code.Use the following data set for this assignment: data.txt (uploaded on google classroom)
These data sets each describe the location of 26 points, each on one line of the file. The the first character is a label (from the lower case letters a,b,c,d,e,. . .). Then separated by white space are two numbers, the x and the y coordinate.

```
import pandas as pd
from pandas import DataFrame
import numpy as np
import matplotlib.pyplot as plt
```

```
df=pd.read_csv('E:/Work/JIIT/sem_6/JIIT-SEM-6/DataMining&WebAlgorithms/Lab
Test2_Practice/q1.csv');
df.head()
```

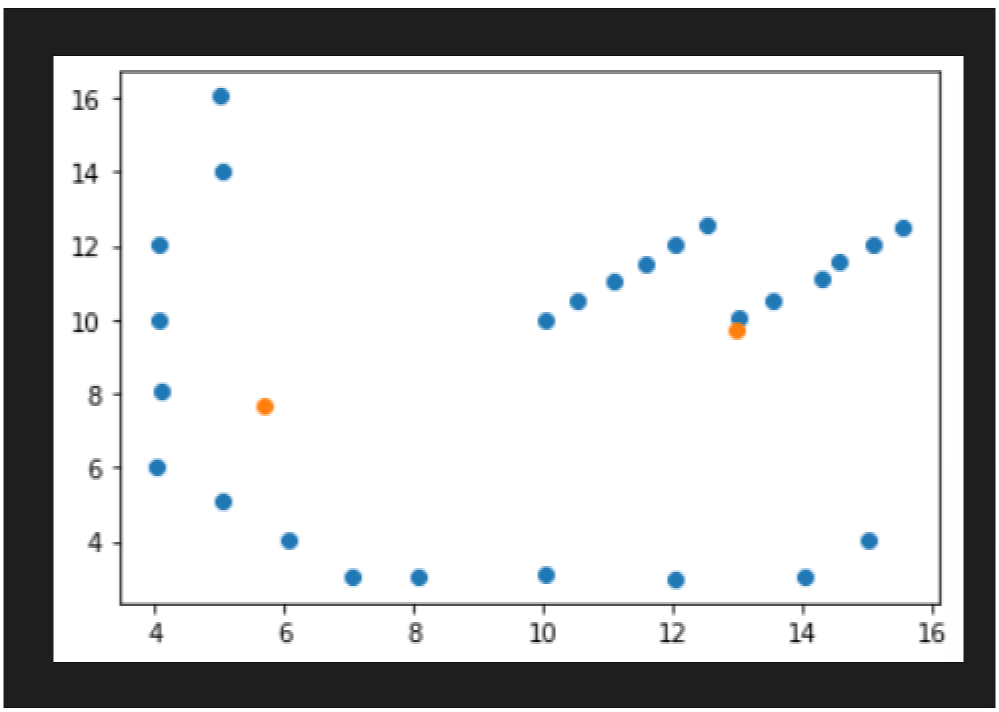
	point	x	y
0	a	4.09	8.06
1	b	4.08	10.02
2	c	4.07	12.01
3	d	12.51	12.54
4	e	12.03	12.04

```
data=pd.DataFrame(df[["x","y"]]).to_numpy()
data
```

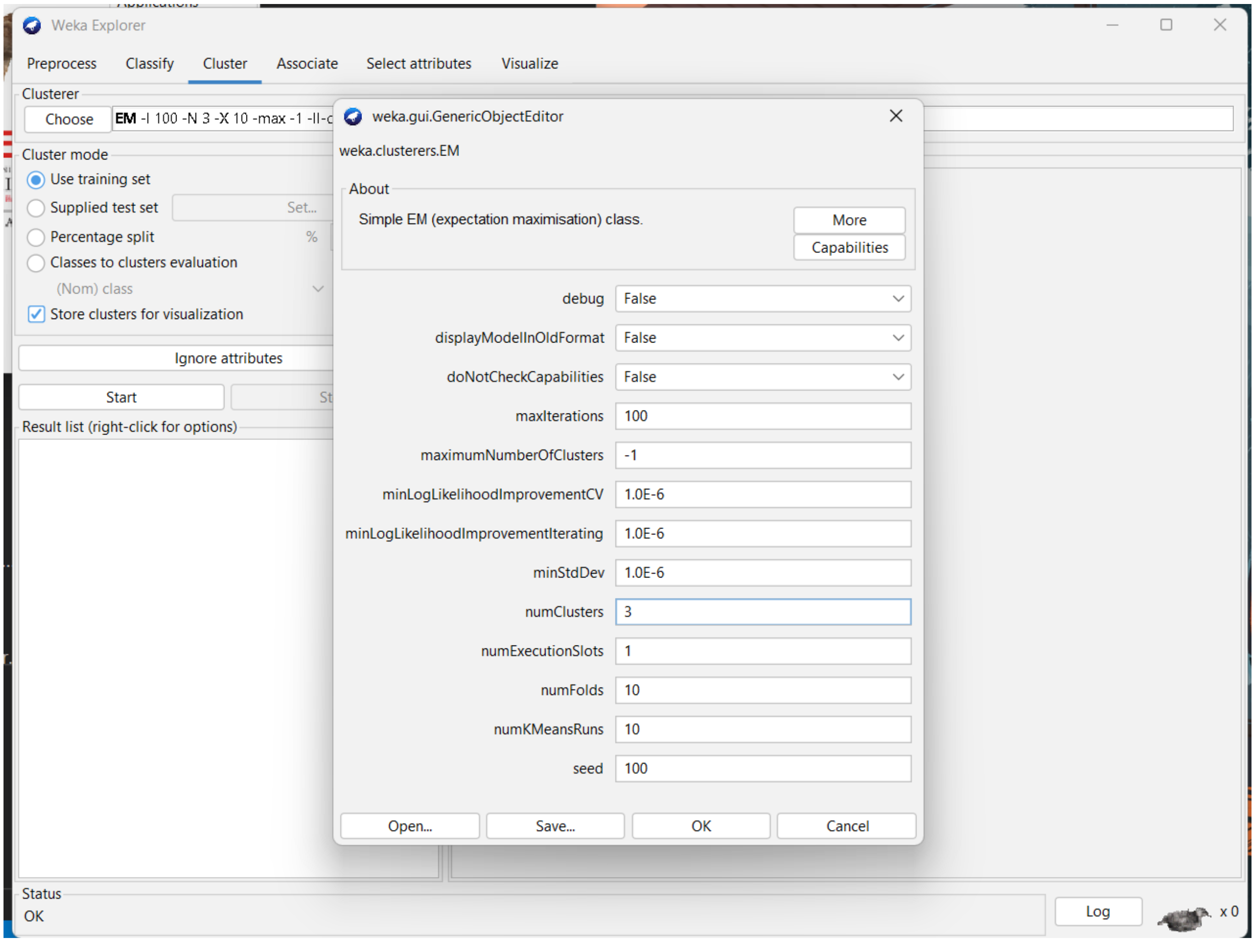
```
array([[ 4.09 ,  8.06 ],
       [ 4.08 , 10.02 ],
       [ 4.07 , 12.01 ],
       [12.51 , 12.54 ],
       [12.03 , 12.04 ],
       [11.57 , 11.52 ],
       [11.09 , 11.03 ],
       [10.53 , 10.51 ],
       [10.01 , 10.01 ],
       [15.52 , 12.5  ],
       [15.1  , 12.06 ],
       [14.57 , 11.55 ],
```

```
def update_assignments(data, centroids):
    c = []
    for i in data:
        c.append(np.argmin(np.sum((i.reshape((1, 2)) - centroids) ** 2,
axis=1)))
    return c

#  find mean of the points belonging to same clusters and update the
centroid
def update_centroids(data, num_clusters, assignments):
    cen = []
    for c in range(len(num_clusters)):
        cen.append(np.mean([data[x] for x in range(len(data)) if
assignments[x] == c], axis=0))
    return cen
```

```
# data = np.loadtxt('blobs.dat').T # (50, 2), 50 data points, 2 dimensions
each
print(data.shape)

# reshaped as 1 row and 2 columns
#  for k=3
centroids = (np.random.normal(size=(3, 2)) * 0.0001) + np.mean(data,
axis=0).reshape((1, 2))

for i in range(100):
    a = update_assignments(data, centroids)
    centroids = update_centroids(data, centroids, a)
    centroids = np.array(centroids)

plt.scatter(data[:, 0], data[:, 1])
plt.scatter(centroids[:, 0], centroids[:, 1])
plt.show()
```



2. Run K-Mean algorithm on WEKA to analyse the results on any data set. You may use given links to choose the desired dataset.
http://storm.cis.fordham.edu/~gweiss/data-mining/datasets.html
https://archive.ics.uci.edu/ml/machine-learning-databases


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Clusterer
Choose
EM -I 100 -N 3 -X 10 -max -1 -ll-c
Cluster mode
Use training set
Supplied test set
Set...
Percentage split
%
Classes to clusters evaluation
(Nom) class
Store clusters for visualization
Ignore attributes
Start
Result list (right-click for options)
Status
OK
Log
x 0
weka.gui.GenericObjectEditor
weka.clusterers.EM
About
Simple EM (expectation maximisation) class.
More
Capabilities
debug
False
displayModelInOldFormat
False
doNotCheckCapabilities
False
maxIterations
100
maximumNumberOfClusters
-1
minLogLikelihoodImprovementCV
1.0E-6
minLogLikelihoodImprovementIterating
1.0E-6
minStdDev
1.0E-6
numClusters
3
numExecutionSlots
1
numFolds
10
numKMeansRuns
10
seed
100
Open...
Save...
OK
Cancel

Clusterer output

```
  std. dev.     0.2627   0.1061   0.2456



Time taken to build model (full training data) : 0.06 seconds

=== Model and evaluation on training set ===

Clustered Instances

0       64 ( 43%)
1       50 ( 33%)
2       36 ( 24%)


Log likelihood: -2.055


Class attribute: class
Classes to Clusters:

  0  1  2  <-- assigned to cluster
  0 50  0 | Iris-setosa
 50  0  0 | Iris-versicolor
 14  0 36 | Iris-virginica

Cluster 0 <-- Iris-versicolor
Cluster 1 <-- Iris-setosa
Cluster 2 <-- Iris-virginica

Incorrectly clustered instances :       14.0      9.3333 %
```

3. Cluster the following data set of ten objects into two clusters i.e. k = 2.

Implement K-Medoid algorithm, so the configuration does not change and algorithm terminates with no Medoid changes. A Medoid can be defined as the object of a cluster whose average dissimilarity to all the objects in the cluster is minimal. i.e. it is a most centrally located point in the cluster.

```
import pandas as pd
from pandas import DataFrame
import numpy as np
```

```
df=pd.read_csv('E:/Work/JIIT/sem_6/JIIT-SEM-6/DataMining&WebAlgorithms/Lab Test2_Practice/q1.csv');
df.head()
data=pd.DataFrame(df[["x","y"]]).to_numpy()
```

```
data
```

```
def euclidean_distance(a,b):
    dist = np.sqrt(np.sum(np.square(a-b)))
    return dist
```

```

class PAM():
    """A simple clustering method that forms k clusters by first assigning
    samples to the closest medoids, and then swapping medoids with
non-medoid
    samples if the total distance (cost) between the cluster members and
their medoid
    is smaller than prevoisly.
    Parameters:
    -----------
    k: int
        The number of clusters the algorithm will form.
    """
    def __init__(self, k=2):
        self.k = k

    def _init_random_medoids(self, X):
        """ Initialize the medoids as random samples """
        n_samples, n_features = np.shape(X)
        medoids = np.zeros((self.k, n_features))
        for i in range(self.k):
            medoid = X[np.random.choice(range(n_samples))]
            medoids[i] = medoid
        return medoids

    def _closest_medoid(self, sample, medoids):
        """ Return the index of the closest medoid to the sample """
        closest_i = None
        closest_distance = float("inf")
        for i, medoid in enumerate(medoids):
            distance = euclidean_distance(sample, medoid)
            if distance < closest_distance:
                closest_i = i
                closest_distance = distance
```

```
        return closest_i

    def _create_clusters(self, X, medoids):
        """ Assign the samples to the closest medoids to create clusters
"""
        clusters = [[] for _ in range(self.k)]
        for sample_i, sample in enumerate(X):
            medoid_i = self._closest_medoid(sample, medoids)
            clusters[medoid_i].append(sample_i)
        return clusters

    def _calculate_cost(self, X, clusters, medoids):
        """ Calculate the cost (total distance between samples and their
medoids) """
        cost = 0
        # For each cluster
        for i, cluster in enumerate(clusters):
            medoid = medoids[i]
            for sample_i in cluster:
                # Add distance between sample and medoid as cost
                cost += euclidean_distance(X[sample_i], medoid)
        return cost

    def _get_non_medoids(self, X, medoids):
        """ Returns a list of all samples that are not currently medoids
"""
        non_medoids = []
        for sample in X:
            if not sample in medoids:
                non_medoids.append(sample)
        return non_medoids

    def _get_cluster_labels(self, clusters, X):
        """ Classify samples as the index of their clusters """
        # One prediction for each sample
        y_pred = np.zeros(np.shape(X)[0])
        for cluster_i in range(len(clusters)):
            cluster = clusters[cluster_i]
            for sample_i in cluster:
                y_pred[sample_i] = cluster_i
```

```
        return y_pred

    def predict(self, X):
        """ Do Partitioning Around Medoids and return the cluster labels
"""
        # Initialize medoids randomly
        medoids = self._init_random_medoids(X)
        # Assign samples to closest medoids
        clusters = self._create_clusters(X, medoids)

        # Calculate the initial cost (total distance between samples and
        # corresponding medoids)
        cost = self._calculate_cost(X, clusters, medoids)

        # Iterate until we no longer have a cheaper cost
        while True:
            best_medoids = medoids
            lowest_cost = cost
            for medoid in medoids:
                # Get all non-medoid samples
                non_medoids = self._get_non_medoids(X, medoids)
                # Calculate the cost when swapping medoid and samples
                for sample in non_medoids:
                    # Swap sample with the medoid
                    new_medoids = medoids.copy()
                    new_medoids[medoids == medoid] = sample
                    # Assign samples to new medoids
                    new_clusters = self._create_clusters(X, new_medoids)
                    # Calculate the cost with the new set of medoids
                    new_cost = self._calculate_cost(
                        X, new_clusters, new_medoids)
                    # If the swap gives us a lower cost we save the
medoids and cost
                    if new_cost < lowest_cost:
                        lowest_cost = new_cost
                        best_medoids = new_medoids
            # If there was a swap that resultet in a lower cost we save
the
            # resulting medoids from the best swap and the new cost
            if lowest_cost < cost:
```

```
                cost = lowest_cost
                medoids = best_medoids
            # Else finished
            else:
                break

        final_clusters = self._create_clusters(X, medoids)
        # Return the samples cluster indices as labels
        return self._get_cluster_labels(final_clusters, X)
```

```
pam=PAM(k=1)
predicted_val=pam.predict(data);
predicted_val
```

```
array([0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0.,
0.])
```

4. Consider the dataset of 6 objects below with distance matrix:

Apply Hierarchical clustering with Single, Complete and average linkage distance measures of agglomerative approach. Show the changes in matrix for each successive iteration till all forms a single cluster.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics.pairwise import pairwise_distances
import sys
```

```
#Our Dataset
data =
np.array([0.40,0.53,0.22,0.38,0.35,0.32,0.26,0.19,0.08,0.41,0.45,0.30]).re
shape(6,2)
print(data)
```

```
[[0.4  0.53]
 [0.22 0.38]
 [0.35 0.32]
 [0.26 0.19]
 [0.08 0.41]
 [0.45 0.3 ]]
```

```
def hierarchical_clustering(data,linkage,no_of_clusters):
    #first step is to calculate the initial distance matrix
    #it consists distances from all the point to all the point
    color = ['r','g','b','y','c','m','k','w']
    initial_distances = pairwise_distances(data,metric='euclidean')
    #making all the diagonal elements infinity
    np.fill_diagonal(initial_distances,sys.maxsize)
    clusters = find_clusters(initial_distances,linkage)

    #plotting the clusters
    iteration_number = initial_distances.shape[0] - no_of_clusters
    clusters_to_plot = clusters[iteration_number]
    arr = np.unique(clusters_to_plot)

    indices_to_plot = []
    fig = plt.figure()
    fig.suptitle('Scatter Plot for clusters')
    ax = fig.add_subplot(1,1,1)
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    for x in np.nditer(arr):
        indices_to_plot.append(np.where(clusters_to_plot==x))
    p=0

    print(clusters_to_plot)
    for i in range(0,len(indices_to_plot)):
        for j in np.nditer(indices_to_plot[i]):
               ax.scatter(data[j,0],data[j,1], c= color[p])
        p = p + 1
```

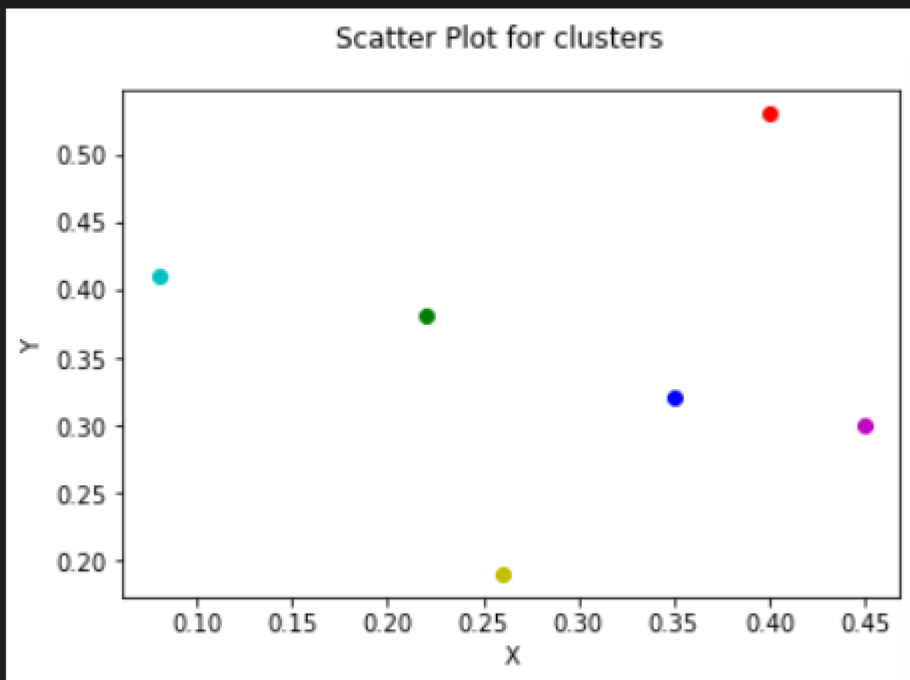
```
    plt.show()
```

```
def find_clusters(input,linkage):
    clusters = {}
    row_index = -1
    col_index = -1
    array = []

    for n in range(input.shape[0]):
        array.append(n)

    clusters[0] = array.copy()

    #finding minimum value from the distance matrix
    #note that this loop will always return minimum value from bottom
triangle of matrix
    for k in range(1, input.shape[0]):
        min_val = sys.maxsize

        for i in range(0, input.shape[0]):
            for j in range(0, input.shape[1]):
                if(input[i][j]<=min_val):
                    min_val = input[i][j]
                    row_index = i
                    col_index = j

        #once we find the minimum value, we need to update the distance
matrix
        #updating the matrix by calculating the new distances from the
cluster to all points

        #for Single Linkage
        if(linkage == "single" or linkage =="Single"):
            for i in range(0,input.shape[0]):
                if(i != col_index):
                    #we calculate the distance of every data point from
newly formed cluster and update the matrix.
```

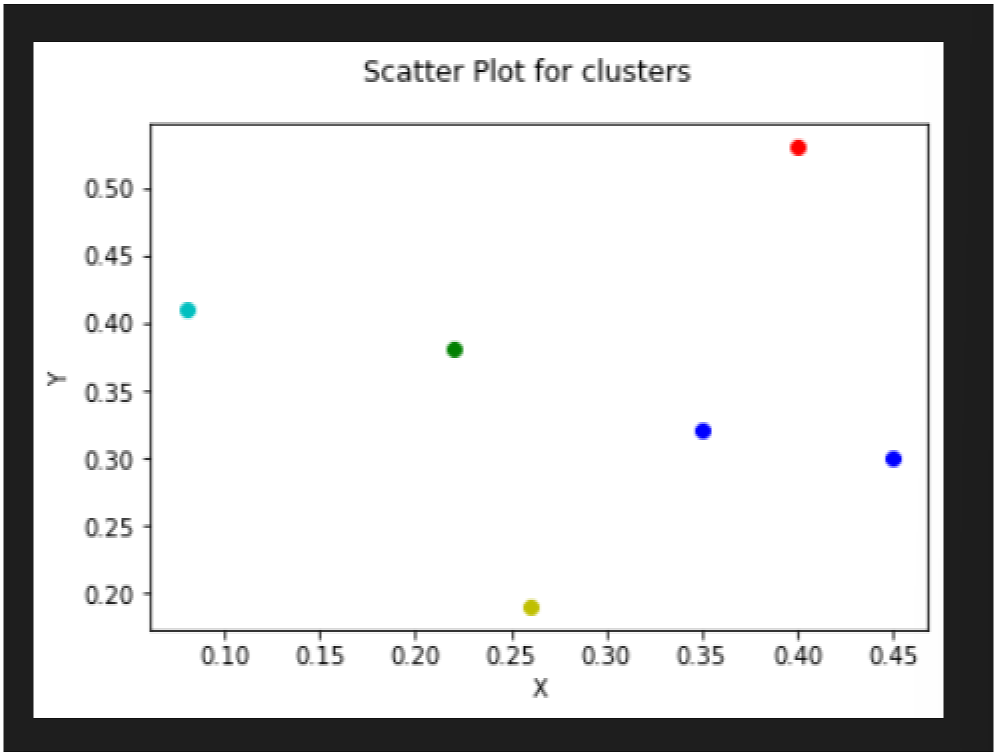
```
                    temp = min(input[col_index][i],input[row_index][i])
                    #we update the matrix symmetrically as our distance
matrix should always be symmetric
                    input[col_index][i] = temp
                    input[i][col_index] = temp
        #for Complete Linkage
        elif(linkage=="Complete" or linkage == "complete"):
             for i in range(0,input.shape[0]):
                if(i != col_index and i!=row_index):
                    temp = max(input[col_index][i],input[row_index][i])
                    input[col_index][i] = temp
                    input[i][col_index] = temp
        #for Average Linkage
        elif(linkage=="Average" or linkage == "average"):
             for i in range(0,input.shape[0]):
                if(i != col_index and i!=row_index):
                    temp = (input[col_index][i]+input[row_index][i])/2
                    input[col_index][i] = temp
                    input[i][col_index] = temp

        #set the rows and columns for the cluster with higher index i.e.
the row index to infinity
        #Set input[row_index][for_all_i] = infinity
        #set input[for_all_i][row_index] = infinity
        for i in range (0,input.shape[0]):
            input[row_index][i] = sys.maxsize
            input[i][row_index] = sys.maxsize

        #Manipulating the dictionary to keep track of cluster formation in
each step
        #if k=0,then all datapoints are clusters

        minimum = min(row_index,col_index)
        maximum = max(row_index,col_index)
        for n in range(len(array)):
            if(array[n]==maximum):
                array[n] = minimum
        clusters[k] = array.copy()

    return clusters
```

```
hierarchical_clustering(data,"single",6)
```

```
[0, 1, 2, 3, 4, 5]
```




```
hierarchical_clustering(data,"single",5)
#you can see that the color of data[2] and data[5] became same, thus they
are in same cluster now
```


Scatter Plot for clusters
0.50
0.45
0.40
0.35
0.30
0.25
0.20
Y
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
X