ECS 111 HW 1

Kevin Oghalai

Algorithm Implementation

The K-means algorithm was implemented by randomly selecting k centroids within the range of the data. Those centroids then have the data points nearest to them assigned to them. The centroid centers are then updated to be the means of the points assigned to them. The data point assignment and centroid location averaging are then repeated multiple times, until convergence has been achieved. Because there is some randomness created from the initial positions of the centroids, leading to inconsistent results, the algorithm was run a couple times and the clusters that looked the best were chosen to be the final results. Code for this is shown at the end of the report with comments, although the logic behind the algorithm is already explained here.

The DBScan algorithm was implemented by looking in a neighborhood of epsilon distance around a chosen point, finding its neighbors, then classifying it as a inner point(the number of neighbors is greater than a defined minimum) or noise. While noise is left untouched for the moment, inner points have their neighbors checked to see if they are also inner points. This continues until there are no more inner points being created. Inner points are directly labeled as belonging to a cluster. Any points within epsilon distance of the inner points are set to be boundary points, which are still part of a cluster, but cannot extend the reach of the cluster. Even if a point is initially classified as noise, it can later be assigned as a boundary point of a cluster. Any points that do not get assigned a cluster at the end of the process are truly classified as noise and are plotted as being separate from the rest of the clusters. Code for this is shown at the end of the report with comments, although the logic behind the algorithm is already explained here.

Visualization of Clustered Data:

Shown in Figure 1 is a K-means clustering of the training data given for the problem with $k = 6$. The upper left plot shows the unlabeled data with axes set to have equal scaling. The upper right shows the results of a K-means clustering with no normalization, while the bottom left and right show normalization by data range and z-scale respectively. In this case, normalization actually squishes the axis along which the data can be easily distinguished, leading to poor results when normalization is performed. The optimal clustering was chosen to be the top right graph for this dataset.

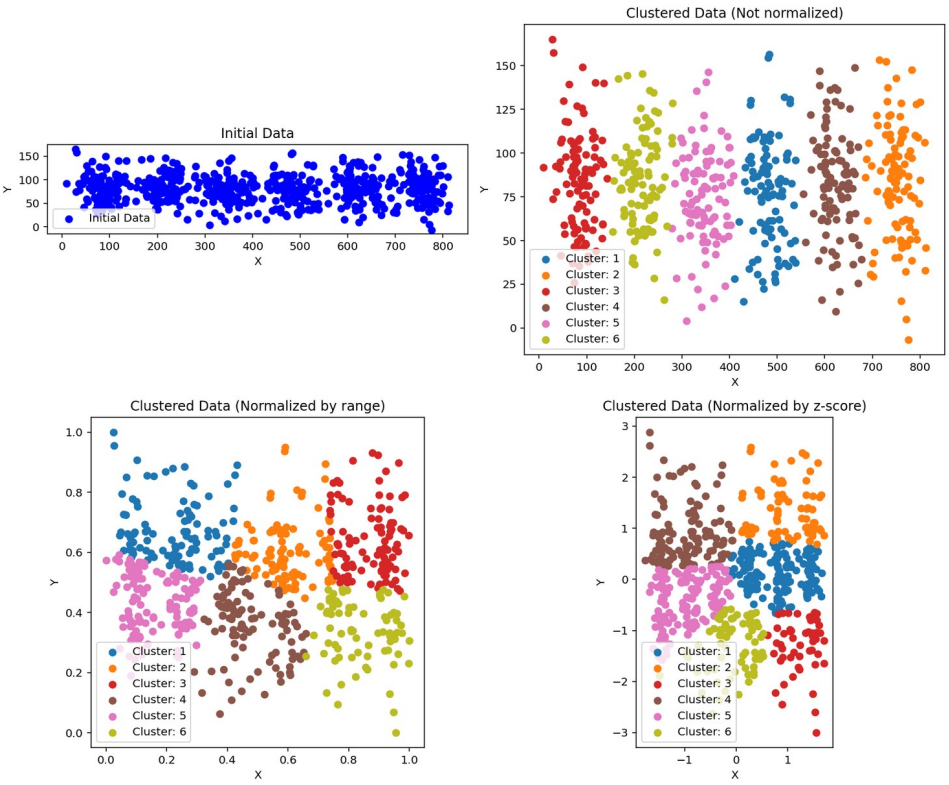

Figure 1: Plots of the training data clustered by the K-means algorithm.

The K-means clustering generally worked very well, and there are no major issues with it. Shown in Figure 3 is a clustering of the same dataset, but using the DBScan algorithm with $\epsilon = 21$, $minPoints = 5$. In this case, once again, the normalization did not end up helping. Only the non-normalized data is shown.

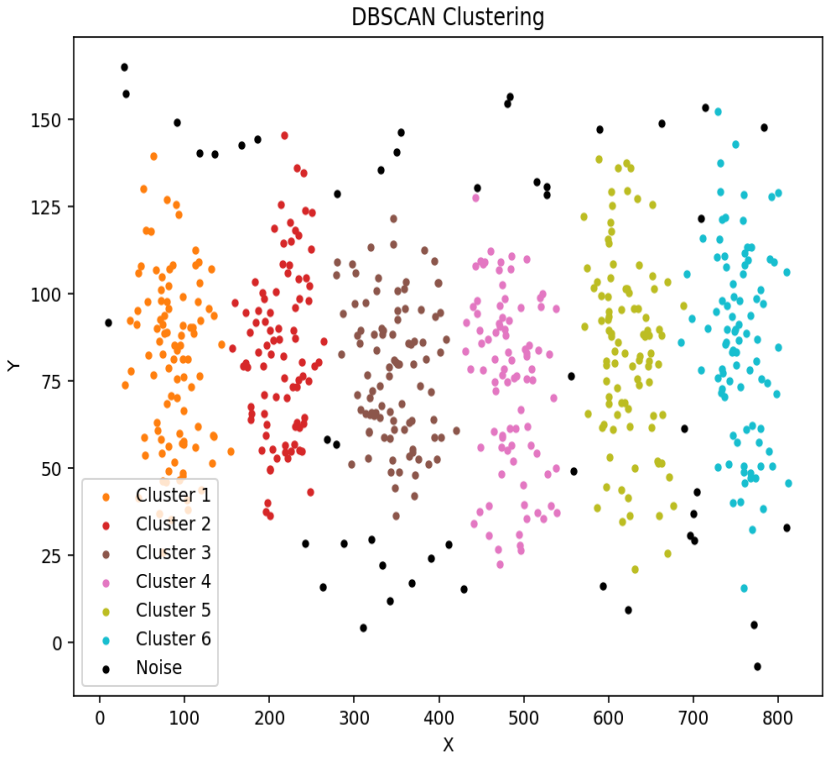

Figure 2: Plots of the training data clustered by the DBScan algorithm.

In this case, the program had issues with putting some of the points on the extremities into the clusters. If the conditions to be added to a cluster were relaxed, it frequently led to the clusters combining, which is also a poor result. Overall, the clusters still look very nice and outliers could be assigned based on their proximity to the centroids of the clusters that were created if necessary.

The two clustering algorithms were then applied to the test set, which is shown in Figure 3. Once again, the upper right shows the results of a K-means clustering with no normalization, while the bottom left and right show normalization by data range and z-scale respectively. Like before, normalization actually squishes the axis along which the data can be easily distinguished, leading to poor results when normalization is performed. The optimal clustering was chosen to be the top right graph for this dataset.

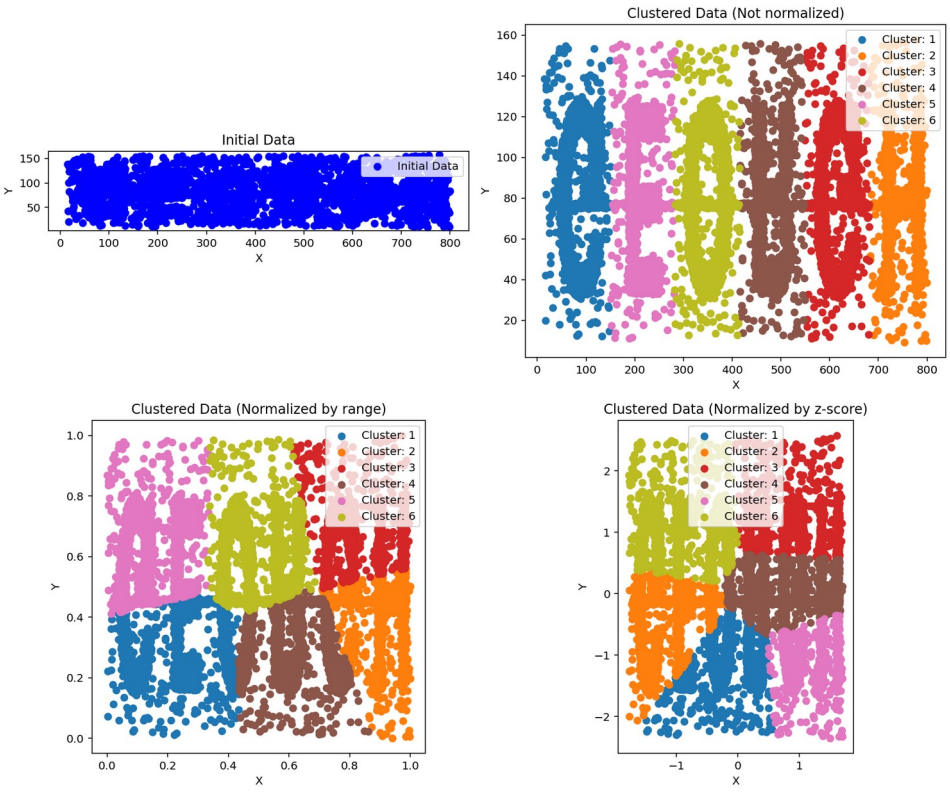

Figure 3: K-means implementation on the test set. The top left is the unclustered data, top right is clustered with no normalization, bottom left shows normalization by data range, and bottom left shows normalization by z-score.

Once again, the K-means algorithm with no normalization looks the best. The DBScan algorithm was implemented on the test set as well with $\epsilon = 8, minPoints = 20$.

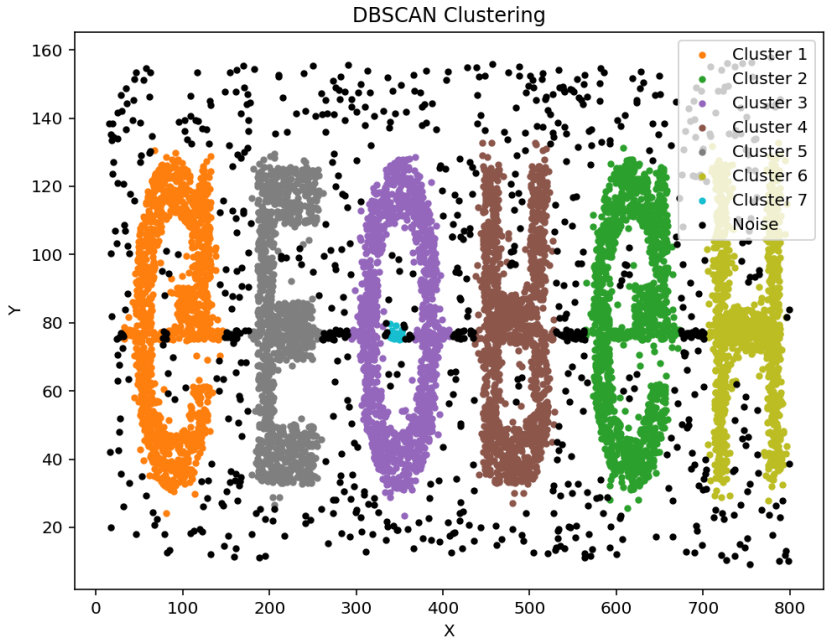

Figure 4:

This clustering method arguably performed better than the K-means method, since many of the data points that seem to be randomly distributed throughout the range end up being classified as outliers.

Unfortunately, there does seem to be an extra cluster created from noise that is very close together, but the general groupings seem to work pretty well.

Code

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import csv

data = np.loadtxt("clusterData_test_unlabeled.dat")

#Normalize data based on what method is preferred
def normalizeData(data, method='none'):
    if method == 'none' :
        normalizedData=data
    elif method == 'range' :
        maxVal = np.max(data,axis=0)
        minVal = np.min(data,axis=0)
        normalizedData = (data-minVal)/(maxVal-minVal)
    elif method == 'z-score' :
        normalizedData=(data-np.mean(data,axis=0))/np.std(data,axis=0)
    return normalizedData

#Implement K-Means clustering
def kMeans(normalizedData,k=1,maxIter=50):
    #Choose points randomly within the data range
    rng = np.random.default_rng()
    pointsNormalized = rng.random((k,2))
    maxVal = np.max(normalizedData,axis=0)
    minVal = np.min(normalizedData,axis=0)
    #Initialize centroids and distances to points
    centroids = pointsNormalized*(maxVal-minVal)+minVal
    distanceMat = np.zeros((np.size(normalizedData,axis=0),k))
    #Allocate points to centroids and update centroid locations to be the averages of
    #points. Repeat multiple times
    for i in range(maxIter):
        for j in range(k):
            distanceMat[:,j] = np.sqrt(np.sum(np.square(normalizedData-centroids[j,:]),axis=1))
        clusterInds = np.argmin(distanceMat,axis=1)
        for j in range(k):
            centroids[j,:] = np.mean(normalizedData[np.where(clusterInds==j),:],axis=1)
    #Calculate final results and return them
    distanceMat[:,j] = np.sqrt(np.sum(np.square(normalizedData-centroids[j,:]),axis=1))
    clusterInds = np.argmin(distanceMat,axis=1)
    return centroids, clusterInds

#Implement DBScan algorithm
def dbscan(normalizedData,eps,minpts):
    #Create function to find neighbors, since this will be done multiple times
    def findNeighbors(normalizedData,eps,element):
        point = normalizedData[element]
        distances = np.linalg.norm(normalizedData-point,axis=1)
        neighborIndices = np.where(distances<eps)[0]

        return neighborIndices
```

```
    #Create another function. Given an inner point, this will allocate all nearby inner
    #and boundary points to the same cluster
    def findFullCluster(normalizedData,eps,element,neighborIndices,labels,clusterNum,minpts):
        i=0
        #Search over all neighbors
        while len(neighborIndices)>i:
            nearbyPointInd = neighborIndices[i]
            #Based on whether a neighbor has already been classified as noise or
            #could be another inner point, search for neighbors and assign them to
            #the cluster. Append new neighbors to the list so that they can be
            #searched for being inner points as well
            if labels[nearbyPointInd] == -1:
                labels[nearbyPointInd] = clusterNum
            if labels[nearbyPointInd] == 0:
                labels[nearbyPointInd] = clusterNum
                newNeighborIndices = findNeighbors(normalizedData,eps,nearbyPointInd)
                #If they are inner points, check their neighbors as well
                if len(newNeighborIndices) > minpts:
                    neighborIndices = np.append(neighborIndices, newNeighborIndices)
            i += 1
        return labels

    clusterNum = 1

    labels = np.full(np.shape(data)[0],0)
    #Go through the array of data points and assign them to clusters or ignore if they have already
    #been tested
    for element in range(len(labels)):
        datapoint = normalizedData[element]
        dataPointLabel = labels[element]
        neighborIndices = findNeighbors(normalizedData,eps,element)
        if dataPointLabel != 0:
            continue #corresponds to data points that have already been grouped
        if len(neighborIndices) < minpts:
            labels[element] = -1 #corresponds to noise
        else:
            labels =
findFullCluster(normalizedData,eps,element,neighborIndices,labels,clusterNum,minpts)
            clusterNum += 1

    return labels

# DBSCAN clustering
normalizedData = normalizeData(data, method='none')
eps = 8
minpts = 20
clusterInds = dbscan(data, eps, minpts)
cmap = matplotlib.colormaps.get_cmap('tab10')  # Updated to use the new colormap API

#Plot data
fig, ax = plt.subplots(figsize=(8, 6))
unique_labels = set(clusterInds)
```

```
for label in unique_labels:
    if label == -1:
        color = 'black'  # Noise points
        label_name = 'Noise'
    else:
        color = cmap(label / max(unique_labels))
        label_name = f'Cluster {label}'
    cluster_points = data[clusterInds == label]
    ax.scatter(cluster_points[:, 0], cluster_points[:, 1], s=10, color=color, label=label_name)

ax.set_title("DBSCAN Clustering")
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.legend()
plt.savefig("dbscan_clustering.png")  # Save the figure as a PNG file
plt.close(fig)

# Save DBSCAN results to a CSV file
with open("dbscan_results.csv", "w", newline="") as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(["X", "Y", "Cluster"])
    for i in range(len(data)):
        writer.writerow([data[i, 0], data[i, 1], clusterInds[i]])

# K-means clustering
normalizedData = normalizeData(data, method='none')
k = 6
centroids, clusterInds = kMeans(normalizedData, k, maxIter=10)

# Save K-means results to a CSV file
with open("kmeans_results.csv", "w", newline="") as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(["X", "Y", "Cluster"])
    for i in range(len(data)):
        writer.writerow([data[i, 0], data[i, 1], clusterInds[i]])

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Plot initial data
axes[0, 0].scatter(data[:, 0], data[:, 1], color='blue', label='Initial Data')
axes[0, 0].set_title("Initial Data")
axes[0, 0].set_xlabel("X")
axes[0, 0].set_ylabel("Y")
axes[0, 0].legend()
axes[0, 0].set_aspect('equal')

cmap = matplotlib.colormaps.get_cmap('tab10')  # Updated to use the new colormap API
colors = [cmap(i / k) for i in range(k)]
for i in range(k):
    plotData = data[clusterInds == i]
    axes[0, 1].scatter(plotData[:, 0], plotData[:, 1], color=colors[i], label=f'Cluster: {i+1}')
```

```
axes[0, 1].set_title("Clustered Data (Not normalized)")
axes[0, 1].set_xlabel("X")
axes[0, 1].set_ylabel("Y")
axes[0, 1].legend()

# Normalized by range
normalizedData = normalizeData(data, method='range')
centroids, clusterInds = kMeans(normalizedData, k, maxIter=10)
for i in range(k):
    plotData = normalizedData[clusterInds == i]
    axes[1, 0].scatter(plotData[:, 0], plotData[:, 1], color=colors[i], label=f'Cluster: {i+1}')

axes[1, 0].set_title("Clustered Data (Normalized by range)")
axes[1, 0].set_xlabel("X")
axes[1, 0].set_ylabel("Y")
axes[1, 0].legend()
axes[1, 0].set_aspect('equal')

# Normalized by z-score
normalizedData = normalizeData(data, method='z-score')
centroids, clusterInds = kMeans(normalizedData, k, maxIter=10)
for i in range(k):
    plotData = normalizedData[clusterInds == i]
    axes[1, 1].scatter(plotData[:, 0], plotData[:, 1], color=colors[i], label=f'Cluster: {i+1}')

axes[1, 1].set_title("Clustered Data (Normalized by z-score)")
axes[1, 1].set_xlabel("X")
axes[1, 1].set_ylabel("Y")
axes[1, 1].legend()
axes[1, 1].set_aspect('equal')

plt.tight_layout()
plt.savefig("kmeans_clustering.png")  # Save the figure as a PNG file
plt.close(fig)
```