CS145 Howework 4

Important Note: HW4 is due on **11:59 PM PT, Nov 20 (Friday, Week 7)**. Please submit through GradeScope.

Print Out Your Name and UID

Name: Danning Yu, UID: 305087992

Before You Start

You need to first create HW4 conda environment by the given `cs145hw4.yml` file, which provides the name and necessary packages for this tasks. If you have `conda` properly installed, you may create, activate or deactivate by the following commands:

```
conda env create -f cs145hw4.yml
conda activate hw4
conda deactivate
```

OR

```
conda env create --name NAMEOFYOURCHOICE -f cs145hw4.yml
conda activate NAMEOFYOURCHOICE
conda deactivate
```

To view the list of your environments, use the following command:

```
conda env list
```

More useful information about managing environments can be found here.

You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

In this notebook, you must not delete any code cells in this notebook. If you change any code outside the blocks (such as some important hyperparameters) that you are allowed to edit (between STRART/END YOUR CODE HERE), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

In [1]:
```
import numpy as np
import pandas as pd
import sys
import random
import math
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
```

```
%load_ext autoreload
%autoreload 2
```

If you can successfully run the code above, there will be no problem for environment setting.

1. Clustering Evaluation

This workbook will walk you through an example for calculating different clustering metrics.

Note: This is a "question-answer" style problem. You do not need to code anything and you are required to calculate by hand (with a scientific calculator).

Questions

Suppose we want to cluster the following 20 conferences into four areas, with ground truth label and algorithm output label shown in third and fourth column. Please evaluate the quality of the clustering algorithm according to four different metrics respectively.

Questions (please include intermediate steps)

1. Calculate purity.
2. Calculate precision.
3. Calculate recall.
4. Calculate F1-score.
5. Calculate normalized mutual information.

Your answer here:

Note: you can use several code cells to help you compute the results and answer the questions. Again you don't need to do any coding.

In [2]:
```
# Code to help calculate some information
# answers to this question can be found below the code
from collections import defaultdict
import numpy as np

truth = [3, 3, 1, 1, 1, 4, 3, 3, 4, 2, 4, 2, 1, 2, 3, 2, 1, 2, 4, 4]
predict = [2, 2, 3, 3, 3, 4, 2, 2, 3, 1, 4, 1, 3, 1, 2, 1, 2, 1, 4, 4]
assert(len(truth) == len(predict))
tp, tn, fp, fn = 0, 0, 0, 0
N = len(truth)
for i in range(N):
    for j in range(i + 1, N):
        truth_same = truth[i] == truth[j]
        predict_same = predict[i] == predict[j]
        if predict_same and truth_same:
            tp += 1
        if not predict_same and not truth_same:
            tn += 1
        if predict_same and not truth_same:
            fp += 1
        if not predict_same and truth_same:
            fn += 1

print(f"# of data points: {len(truth)}")
print(f"# of pairs: {tp+tn+fp+fn}")
```

```
print(f"tp={tp}, tn={tn}, fp={fp}, fn={fn}")

clusters_pred = defaultdict(list)
clusters_truth = defaultdict(list)
for i in range(N):
    clusters_pred[predict[i]].append(truth[i])
    clusters_truth[truth[i]].append(predict[i])
print(clusters_pred)
print(clusters_truth)

counts = {1: defaultdict(int),
          2: defaultdict(int),
          3: defaultdict(int),
          4: defaultdict(int)}
for k, v in clusters_pred.items():
    for val in v:
        counts[k][val] += 1
nmi_table = np.zeros(16).reshape(4, 4).astype(int)
for k, v in counts.items():
    for k2, v2 in v.items():
        nmi_table[k2 - 1][k - 1] = v2
print("NMI table:")
for row in nmi_table:
    print(row, end="")
    print(f" {np.sum(row)}")
col_sums = np.sum(nmi_table, axis = 0)
for val in col_sums:
    print(f" {val}", end="")
print(f"  {N}")
```

```
# of data points: 20
# of pairs: 190
tp=32, tn=141, fp=9, fn=8
defaultdict(<class 'list'>, {2: [3, 3, 3, 3, 3, 1], 3: [1, 1, 1, 4, 1], 4: [4,
4, 4, 4], 1: [2, 2, 2, 2, 2]})
defaultdict(<class 'list'>, {3: [2, 2, 2, 2, 2], 1: [3, 3, 3, 3, 2], 4: [4, 3,
4, 4, 4], 2: [1, 1, 1, 1, 1]})
NMI table:
[0 1 4 0] 5
[5 0 0 0] 5
[0 5 0 0] 5
[0 0 1 4] 5
 5 6 5 4  20
```

Question 1

Purity = $\frac{1}{N}\sum_k \max_j |c_k \cap \omega_j|$

Purity = $\frac{5+5+4+4}{20} = 0.9$

Question 2

Precision = $\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}$

Precision = $\frac{32}{32+9} = 0.780$

Question 3

Recall = $\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$

Recall = $\frac{32}{32+8} = 0.800$

Question 4

F1 score = $\frac{2(\text{precision})(\text{recall})}{\text{precision} + \text{recall}}$

F1 score = $\frac{2(0.780)(0.8)}{0.780 + 0.8} = 0.790$

Question 5

$\text{NMI}(C, \Omega) = \frac{I(C,\Omega)}{\sqrt{H(C)H(\Omega)}}$

$I(C, \Omega) = \sum_k \sum_j \frac{|c_k \cap \omega_j|}{N} \log \frac{N|c_k \cap \omega_j|}{|c_k||\omega_j|}$

$I(C, \Omega) = \frac{5}{20} \log \frac{20(5)}{5(5)} + \frac{1}{20} \log \frac{20(1)}{5(6)}$

$+ \frac{5}{20} \log \frac{20(5)}{5(6)} + \frac{4}{20} \log \frac{20(4)}{5(5)}$

$+ \frac{1}{20} \log \frac{20(1)}{5(5)} + \frac{4}{20} \log \frac{20(4)}{5(4)}$

$I(C, \Omega) = 1.624$

$H(Y) = -\sum_j P(y_j) \log P(y_j)$

$H(C) = -(4)\frac{5}{20} \log \frac{5}{20} = 2$

$H(\Omega) = -(\frac{4}{20} \log \frac{4}{20} + \frac{6}{20} \log \frac{6}{20} + (2)\frac{5}{20} \log \frac{5}{20}) = 1.985$

$\text{NMI}(C, \Omega) = \frac{1.624}{\sqrt{(2)(1.985)}}$

$\text{NMI}(C, \Omega) = 0.815$

2. K-means

In this section, we are going to apply K-means algorithm against two datasets (dataset1.txt, dataset2.txt) with different distributions, respectively.

For each dataset, it contains 3 columns, with the format: x1 \t x2 \t cluster_label. You need to use the first two columns for clustering, and the last column for evaluation.

In [3]:
```
from hw4code.KMeans import KMeans
k = KMeans()
# As a sanity check, we print out a sample of each dataset
dataname1 = "data/dataset1.txt"
dataname2 = "data/dataset2.txt"
k.check_dataloader(dataname1)
k.check_dataloader(dataname2)
```

```
For dataset1: number of datapoints is 150
          x         y  ground_truth_cluster
0 -0.163880 -0.219869                     1
1 -0.886274 -0.356186                     1
2 -0.978910 -0.893314                     1
3 -0.658867 -0.371122                     1
4 -0.072518  0.399157                     1

For dataset2: number of datapoints is 200
          x         y  ground_truth_cluster
0  1.068587  0.136921                     1
1  0.705440  0.393068                     1
2  0.840811 -0.054906                     1
```

```
3 -0.923447  0.598501                    1
4  0.784353  0.724743                    1
```

2.1 Coding K-means

Complete the `reassignClusters` and `getCentroid` function in `KMeans.py` .

Print out each output cluster's size and centroid （x,y) for dataset1 and dataset2 respectively.

In [4]:
```
k = KMeans()
#========================#
# STRART YOUR CODE HERE  #
#========================#
k.main(dataname1)
k.main(dataname2)
#========================#
#   END YOUR CODE HERE   #
#========================#
```

```
For dataset1
Iteration :3
Cluster 0 size :50
Centroid [x=2.5737264423871213, y=-0.027462568841232993]
Cluster 1 size :50
Centroid [x=-0.4633368646347212, y=-0.46611409698195794]
Cluster 2 size :50
Centroid [x=0.9888766205736857, y=2.010478965197201]

For dataset2
Iteration :4
Cluster 0 size :102
Centroid [x=1.2708406269481844, y=-0.08583389704900128]
Cluster 1 size :98
Centroid [x=-0.2018593506236788, y=0.5726963240559535]
```

2.2 Purity and NMI Evaluation

Complete the `compute_purity` function in `KMeans.py` .

In order to compute NMI, you need to firstly compute NMI matrix and then do the calculation. That is to complete the `getNMIMatrix` and `calcNMI` functions in `KMeans.py` .

Print out the purity and NMI for each dataset respectively.

In [5]:
```
k = KMeans()
#========================#
# STRART YOUR CODE HERE  #
#========================#
k.main(dataname1, isevaluate=True)
k.main(dataname2, isevaluate=True)
#========================#
#   END YOUR CODE HERE   #
#========================#
```

```
For dataset1
Iteration :3
Purity is 1.000000
NMI is 1.000000
Cluster 0 size :50
Centroid [x=2.5737264423871213, y=-0.027462568841232993]
Cluster 1 size :50
```

```
Centroid [x=-0.4633368646347212, y=-0.46611409698195794]
Cluster 2 size :50
Centroid [x=0.9888766205736857, y=2.010478965197201]

For dataset2
Iteration :4
Purity is 0.760000
NMI is 0.205096
Cluster 0 size :102
Centroid [x=1.2708406269481844, y=-0.08583389704900128]
Cluster 1 size :98
Centroid [x=-0.2018593506236788, y=0.5726963240559535]
```

2.3 Visualization

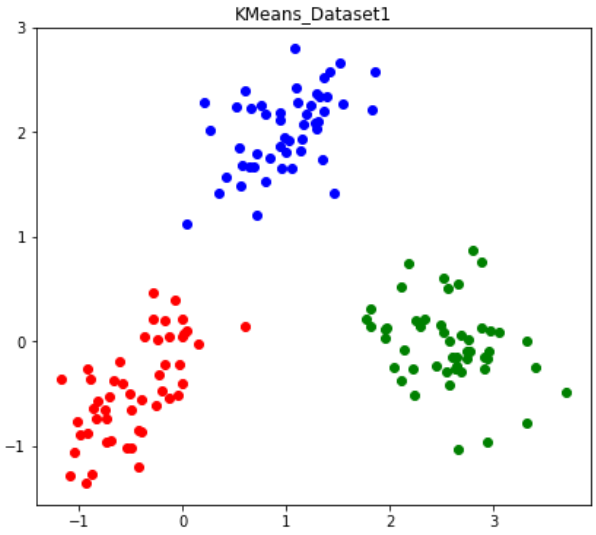
The clustering results for KMeans are saved as `KMeans_dataset1.csv` and `KMeans_dataset2.csv` respectively under your root folder. Plot the clustering results for the two datasets, with different colors representing different clusters.

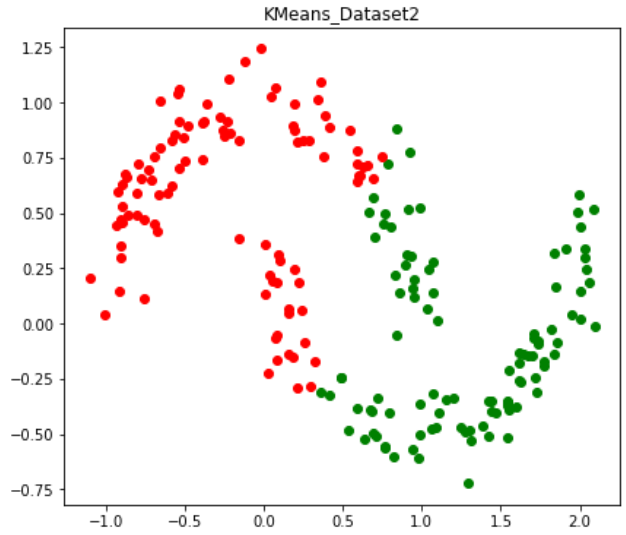
In [7]:
```
CSV_FILE_PATH1 = 'Kmeans_dataset1.csv'
CSV_FILE_PATH2 = 'Kmeans_dataset2.csv'

df1 = pd.read_csv(CSV_FILE_PATH1,header=None,names=['x','y','pred'])
df2 = pd.read_csv(CSV_FILE_PATH2,header=None,names=['x','y','pred'])
fig, [ax0,ax1] = plt.subplots(1, 2, figsize=(15, 6))
ax0.title.set_text("KMeans_Dataset1")
ax1.title.set_text("KMeans_Dataset2")

#========================#
# STRART YOUR CODE HERE  #
#========================#
num_clusters = len(set(df1['pred']))
COLORS = ["green", "red", "blue", "black", "purple"]
for i in range(num_clusters):
    df1_filter = df1.loc[df1['pred'] == i]
    ax0.scatter(df1_filter['x'], df1_filter['y'], color=COLORS[i])

num_clusters = len(set(df2['pred']))
for i in range(num_clusters):
    df2_filter = df2.loc[df2['pred'] == i]
    ax1.scatter(df2_filter['x'], df2_filter['y'], color=COLORS[i])
#========================#
#   END YOUR CODE HERE   #
#========================#
plt.show()
```





Question

Give the pros and cons of K-means algorithm. (At least one for pro and two for cons to get full marks)

Your answer here

Pros:
(1) It is very efficient and runs in O(tkn) time, where n is # of data points, k is # of clusters, and t is # of iterations.

Cons:
(1) The number of clusters needs to be specified in advance
(2) Only works on convex clusters.
(3) Sensitive to outliers

3 DBSCAN

In this section, we are going to use DBSCAN for clustering the same two datasets.

3.1 Coding DBSCAN

Complete the `dbscan` function in `DBSCAN.py` . Print out the purity, NMI and cluter size for each dataset respectively.

In [8]:
```
from hw4code.DBSCAN import DBSCAN
d = DBSCAN()
#=========================#
# STRART YOUR CODE HERE  #
#=========================#
dataname1 = "data/dataset1.txt"
dataname2 = "data/dataset2.txt"
d.main(dataname1)
d.main(dataname2)
#=========================#
#   END YOUR CODE HERE   #
#=========================#
```

```
For dataset1
```

```
Esp :0.3560832705047313
Number of clusters formed :4
Noise points :11
Purity is 0.940000
NMI is 0.959065
Cluster 0 size :49
Cluster 1 size :41
Cluster 2 size :47
Cluster 3 size :4

For dataset2
Esp :0.18652096476712493
Number of clusters formed :3
Noise points :3
Purity is 0.985000
NMI is 0.817349
Cluster 0 size :99
Cluster 1 size :51
Cluster 2 size :47
```

3.2 Visualization

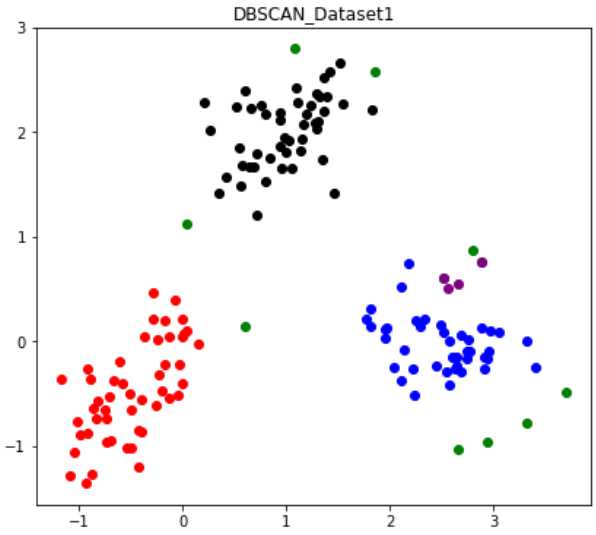
The clustering results for DBSCAN are saved as `DBSCAN_dataset1.csv` and `DBSCAN_dataset2.csv` respectively under your root folder. Plot the clustering results for the two datasets, with different colors representing different clusters.

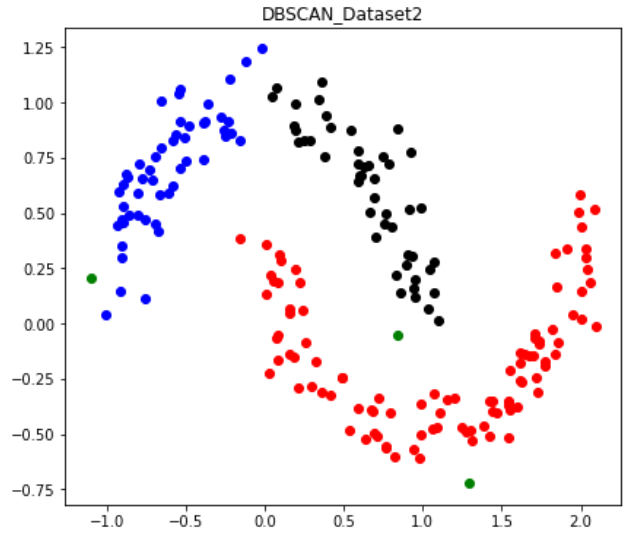
In [9]:
```
CSV_FILE_PATH1 = 'DBSCAN_dataset1.csv'
CSV_FILE_PATH2 = 'DBSCAN_dataset2.csv'

df1 = pd.read_csv(CSV_FILE_PATH1,header=None,names=['x','y','pred'])
df2 = pd.read_csv(CSV_FILE_PATH2,header=None,names=['x','y','pred'])
fig, [ax0,ax1] = plt.subplots(1, 2, figsize=(15, 6))
ax0.title.set_text("DBSCAN_Dataset1")
ax1.title.set_text("DBSCAN_Dataset2")

#========================#
# STRART YOUR CODE HERE  #
#========================#
num_clusters = len(set(df1['pred']))
COLORS = ["green", "red", "blue", "black", "purple"]
for i in range(num_clusters):
    df1_filter = df1.loc[df1['pred'] == i]
    ax0.scatter(df1_filter['x'], df1_filter['y'], color=COLORS[i])

num_clusters = len(set(df2['pred']))
for i in range(num_clusters):
    df2_filter = df2.loc[df2['pred'] == i]
    ax1.scatter(df2_filter['x'], df2_filter['y'], color=COLORS[i])
#========================#
#   END YOUR CODE HERE   #
#========================#
plt.show()
```





Question

Give the pros and cons of DBSCAN algorithm. (At least two for pro and one for cons to get full marks)

Your answer here

Pros:

(1) Can identify outliers and is robust to them

(2) Can create non-convex clusters

(3) No need to pre-specify the number of clusters

Cons:

(1) Sensitive to minPts and eps parameters - need to tune them, if they change the clustering results can be very different

4 GMM

In this section, we are going to use GMM for clustering the same two datasets.

4.1 Coding GMM

Complete the `Estep` and 'Mstep' function in `GMM.py` . Print out the purity, NMI, final mean, covariance and cluter size for each dataset respectively.

In [40]:
```
from hw4code.GMM import GMM
g = GMM()
#========================#
# STRART YOUR CODE HERE  #
#========================#
dataname1 = "data/dataset1.txt"
dataname2 = "data/dataset2.txt"
g.main(dataname1)
g.main(dataname2)
#========================#
#   END YOUR CODE HERE   #
#========================#
```

```
For dataset1
Number of Iterations = 22
```

```
After Calculations
Final mean =
-0.46247285694404044
-0.4638749980764899

0.9898929396029765
2.011802723814242

2.57342634413319
-0.027108746076609493


Final covariance =
For Cluster : 1
0.14918910487220216
0.1173463005433889

0.1173463005433889
0.21554861253107502


For Cluster : 2
0.16028233507625483
0.07486967581052754

0.07486967581052754
0.13939774162738802


For Cluster : 3
0.18039223672749394
-0.04672614559811056

-0.04672614559811056
0.15206459963738586


Purity is 1.000000
NMI is 1.000000
Cluster 0 size :50
Cluster 1 size :50
Cluster 2 size :50

For dataset2
Number of Iterations = 95

After Calculations
Final mean =
0.7464905663922624
0.45649665848541027

0.2828785188939092
-0.059705607271887506


Final covariance =
For Cluster : 1
0.7692790765358335
-0.28782809642382123

-0.28782809642382123
0.1901249384356512
```

```
For Cluster : 2
0.6828574757628689
-0.30058915994390517

-0.30058915994390517
0.17583559485120062


Purity is 0.690000
NMI is 0.107406
Cluster 0 size :106
Cluster 1 size :94
```

4.2 Visualization

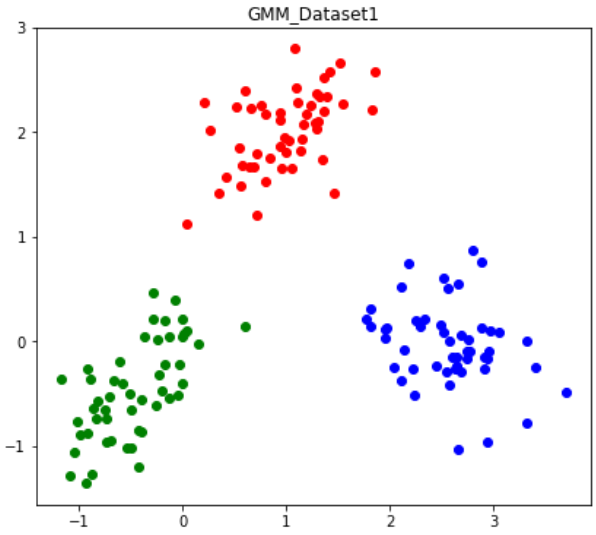
The clustering results for GMM are saved as `GMM_dataset1.csv` and `GMM_dataset2.csv` respectively under your root folder. Plot the clustering results for the two datasets, with different colors representing different clusters.

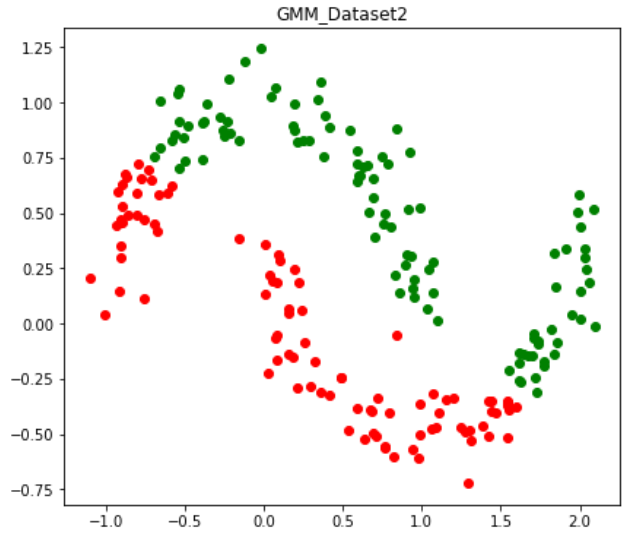
In [41]:
```
CSV_FILE_PATH1 = 'GMM_dataset1.csv'
CSV_FILE_PATH2 = 'GMM_dataset2.csv'

df1 = pd.read_csv(CSV_FILE_PATH1,header=None,names=['x','y','pred'])
df2 = pd.read_csv(CSV_FILE_PATH2,header=None,names=['x','y','pred'])
fig, [ax0,ax1] = plt.subplots(1, 2, figsize=(15, 6))
ax0.title.set_text("GMM_Dataset1")
ax1.title.set_text("GMM_Dataset2")

#========================#
# STRART YOUR CODE HERE  #
#========================#
num_clusters = len(set(df1['pred']))
COLORS = ["green", "red", "blue", "black", "purple"]
for i in range(num_clusters):
    df1_filter = df1.loc[df1['pred'] == i]
    ax0.scatter(df1_filter['x'], df1_filter['y'], color=COLORS[i])

num_clusters = len(set(df2['pred']))
for i in range(num_clusters):
    df2_filter = df2.loc[df2['pred'] == i]
    ax1.scatter(df2_filter['x'], df2_filter['y'], color=COLORS[i])
#========================#
#   END YOUR CODE HERE   #
#========================#
plt.show()
```





Questions

1. Give the pros and cons of GMM algorithm. (At least two for pro and two for cons to get full marks)
2. Compare the visualization results from three algorithms, analyze for each dataset why these algorithms would produce such result.

Your answer here:

Question 1

Pros:

(1) GMMs can handle clusters of different densities and/or sizes.

(2) The shape of the clusters can be determined by the mean and standard deviation (or covariance matrix).

(3) It may be a good fit for how the underlying data is generated (if the underlying data has a Gaussian model).

Cons:

(1) It only converges to a local optimum.

(2) Need to specify k at the start.

(3) Cannot handle non-convex clusters.

(4) Is more computationally expensive than K means. I also noticed it takes more iterations to reach a local optimum.

Question 2

Dataset 1:

For dataset 1, Kmeans and GMM did well because there are 3 clusters with convex shapes, similar densities, and clear boundaries between them. Both achieved purity and NMI scores of 1, which is the max possible. Their plots clearly show 3 separate clusters (with different colors/labels). Points that were slightly far away from a cluster were simply assigned to the nearest cluster. DBSCAN also does quite well on the dataset, achieving a purity of 0.94 and NMI of 0.959. As we can see in its plot, it doesn't achieve perfect purity or NMI because it creates a small 4th cluster and classifies some other points as outliers. This is because DBSCAN is a

density-based clustering method, and the points in the 4th cluster and the outlier points were too far from the 3 main clusters to be put in the same cluster.

Dataset 2:
For dataset 2, Kmeans did poorly, with a purity of 0.76 and a NMI of only 0.205. From looking at the graph and the raw data, we can immediately see that there are 2 non-convex clusters: a "U" shaped cluster that opens upwards and another that opens downwards. Kmeans was not able to detect this cluster shape at all, instead splitting the clusters directly down the middle to create 2 convex clusters. This makes sense, as Kmeans does not handle non-convex clusters well. GMM did even worse, with a purity of 0.69 and NMI of 0.107. Again, we see the same problem of the resulting clustering completely not reflecting the actual "U" shapes of the clusters; rather, it picked a downwards sloping diagonal line as the decision boundary. Again, this is because GMM is not designed to handle non-convex clusters. GMM split it differently from KMeans because its assignment of points is probabilistic rather than a hard assignment.

In contrast to GMM and Kmeans, DBSCAN did well, with a purity of 0.985 and NMI of 0.817. Looking at its graph, we see it detected 3 clusters: the upwards facing "U" as one cluster and then splitting the downwards facing "U" into 2 clusters. There are also some outlier points. Near the boundary of the 2 clusters for the downwards facing "U," the points are quite sparse, so that explains why 2 clusters were created: the points in this region were not dense enough to connect the 2 clusters together as 1. Finally, the outlier points (marked in green) are far away from the other points, which makes sense.

5 Bonus Question

Prove that KMeans algorithm would guarantee covergence. (**Hint: prove for each step the loss would descrease.**)

The loss in KMeans is defined as the sum of squared errors (SSE) between each point and its associated cluster center:
$J = \sum_{j=1}^{k} \sum_{i}^{n} w_{ij} ||\mathbf{x}_i - \mathbf{c}_j||^2$
In this equation, we have k clusters, $\mathbf{c}_j$ is the center of the jth cluster, and w_{ij} is either 0 or 1. If $w_{ij} = 1$, then $\mathbf{x}_i$ is assigned to cluster j.

The KMeans algorithm has 2 steps:

1. With fixed centers $\mathbf{c}_j$, find the assignment of data points w_{ij} that minimizes the SSE loss between each point and its associated cluster.
2. With fixed assignments w_{ij}, recalculate the new center as the mean of all the points currently assigned to that cluster.

Thus, we need to show that both steps decrease J (or cause it to stay the same, signalling convergence). For step 1, for each $\mathbf{x}_i$, we assign it to the cluster that's closest to it, so thus, the SSE for each point and its assigned cluster must be equal to what it was previously or decrease. Thus, expressed mathematically for all points, if $\mathbf{w}^{(t+1)}$ is the new cluster assignments, we have our desired relationship for step 1:

$J(\mathbf{w}^{(t+1)}, \mathbf{c}^{(t)}) \leq J(\mathbf{w}^{(t)}, \mathbf{c}^{(t)})$

For step 2, we must prove that the mean of a cluster is the optimal point that minimizes the SSE for that cluster. To do this, pretend we have a cluster of size m with data points $\mathbf{x}_i$ of dimension d, where $i = 1, 2, 3, \ldots, m$, and we designate $\mathbf{c}^*$ as the mean of the cluster and $\mathbf{c}$ as any point:

$\sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}||^2 = \sum_{i=1}^{m} ||(\mathbf{x}_i - \mathbf{c}^*) + (\mathbf{c}^* - \mathbf{c})||^2$
$\sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}||^2 = \sum_{i=1}^{m} \Big(||(\mathbf{x}_i - \mathbf{c}^*)||^2 + ||(\mathbf{c}^* - \mathbf{c})||^2 + 2(\mathbf{x}_i - \mathbf{c}^*)^T(\mathbf{c}^* - \mathbf{c}) \Big)$

Now we note that $\mathbf{c}^*$ and $\mathbf{c}$ are constants, so $||\mathbf{c}^* - \mathbf{c}||$ is a constant and thus its summation can be dropped. The same is true for $\mathbf{c}^{*T}\mathbf{c}^*$ and $\mathbf{c}^{*T}\mathbf{c}$.

$\sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}||^2 = \sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}^*||^2 + m||\mathbf{c}^* - \mathbf{c}||^2 + 2 \left(\sum_{i=1}^{m} (\mathbf{x}_i^T \mathbf{c}^* - \mathbf{x}_i^T \mathbf{c}) - m\mathbf{c}^{*T}\mathbf{c}^* + m\mathbf{c}^{*}\right.$

Now note that $\sum_{i=1}^{m} \mathbf{x}_i = (m)\frac{1}{m}\sum_{i=1}^{m} \mathbf{x}_i = m\mathbf{c}^*$, which causes all the terms in the $2(\ldots)$ term to cancel out:

$\sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}||^2 = \sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}^*||^2 + m||\mathbf{c}^* - \mathbf{c}||^2 + 2(m\mathbf{c}^{*T}\mathbf{c}^* - m\mathbf{c}^{*T}\mathbf{c} - m\mathbf{c}^{*T}\mathbf{c}^* + m\mathbf{c}^{*T}\mathbf{c}$

$\sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}||^2 = \sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}^*||^2 + m||\mathbf{c}^* - \mathbf{c}||^2$

Since $m > 0$ and $||\mathbf{c}^* - \mathbf{c}||^2 \geq 0$, this means that $m||\mathbf{c}^* - \mathbf{c}||^2 \geq 0$ and thus:

$\sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}||^2 \geq \sum_{i=1}^{m} ||\mathbf{x}_i - \mathbf{c}^*||^2$

Thus, applying this result to the cluster reassigment step, where $\mathbf{c}^{(t)} = \mathbf{c}$ and $\mathbf{c}^{(t+1)} = \mathbf{c}^*$, we get:

$J(\mathbf{w}^{(t+1)}, \mathbf{c}^{(t+1)}) \leq J(\mathbf{w}^{(t+1)}, \mathbf{c}^{(t)})$

Because both steps 1 and 2 always result in a loss that's less than or equal and there are a finite number of clusterings (k^n to be precise), this means that eventually the KMeans algorithm will converge to some local minimum that results from a particular clustering.

End of Homework 4 :)

After you've finished the homework, please print out the entire `ipynb` notebook and four `py` files into one PDF file. Make sure you include the output of code cells and answers for questions. Prepare submit it to GradeScope. Also this time remember assign the pages to the questions on GradeScope

```python
class DataPoints:
    # --------------------------------------------------------------------
    def __init__(self, x, y, label):
        self.x = x
        self.y = y
        self.label = label
        self.isAssignedToCluster = False
    # --------------------------------------------------------------------
    def __key(self):
        return (self.label, self.x, self.y)
    # --------------------------------------------------------------------
    def __eq__(self, other):
        return self.__key() == other.__key()
    # --------------------------------------------------------------------
    def __hash__(self):
        return hash(self.__key())
    # Computes mean of for each cluster
    @staticmethod
    def getMean(clusters, mean):
        for k in range(len(clusters)):
            temp = clusters[k]
            for point in temp:
                mean[k][0] += point.x
                mean[k][1] += point.y
            mean[k][0] /= float(len(temp))
            mean[k][1] /= float(len(temp))
    # Computes std for each cluster
    @staticmethod
    def getStdDeviation(clusters, mean, stddev):
        for k in range(len(clusters)):
            cluster = clusters[k]
            for point in cluster:
                stddev[k][0] += pow(point.x - mean[k][0], 2)
                stddev[k][1] += pow(point.y - mean[k][1], 2)
            stddev[k][0] /= len(cluster)
            stddev[k][1] /= len(cluster)
    # Computes covariance matrix for each cluster
    @staticmethod
    def getCovariance(clusters, mean, stddev, cov):
        for k in range(len(clusters)):
            cov[k][0][0] = stddev[k][0]
            cov[k][1][1] = stddev[k][1]
            cluster = clusters[k]
            for point in cluster:
                cov[k][0][1] += (point.x - mean[k][0]) * (point.y - mean[k][1])
            cov[k][0][1] /= len(cluster)
            cov[k][1][0] = cov[k][0][1]
    # Get groudtruth number of labels in a dataset
    @staticmethod
    def getNoOFLabels(dataSet):
        labels = set()
        for point in dataSet:
            labels.add(point.label)
        return len(labels)
    # write clusting results into .csv file
    @staticmethod
    def writeToFile(noise, clusters, fileName):
        # write clusters to file for plotting
        f = open(fileName, 'w')
        for pt in noise:
            f.write(str(pt.x) + "," + str(pt.y) + ",0" + "\n")
        for w in range(len(clusters)):
            print("Cluster " + str(w) + " size :" + str(len(clusters[w])))
            for point in clusters[w]:
```

```
65              f.write(str(point.x) + "," + str(point.y) + "," + str((w + 1)) + "\n")
66          f.close()
```

```
from hw4code.DataPoints import DataPoints
import random
import sys
import math
import pandas as pd

# ================================================================
def sqrt(n):
    return math.sqrt(n)

# ================================================================
def getEuclideanDist(x1, y1, x2, y2):
    dist = sqrt((x2 - x1), 2) + pow((y2 - y1), 2))
    return dist
# ================================================================
def compute_purity(clusters,total_points):
    # Calculate purity

    # Create list to store the maximum union number for each output cluster.
    maxLabelCluster = []
    num_clusters = len(clusters)
    # ========================#
    # STRART YOUR CODE HERE  #
    # ========================#
    import numpy as np
    for clus in clusters:
        # clus is a set of DataPoint objects
        labels = np.asarray([point.label for point in clus])
        maxLabelCluster.append(np.max(np.bincount(labels)))
    # ========================#
    #   END YOUR CODE HERE   #
    # ========================#
    purity = 0.0
    for j in range(num_clusters):
        purity += maxLabelCluster[j]
    purity /= total_points
    print("Purity is %.6f" % purity)

# ================================================================
def compute_NMI(clusters,noOfLabels):
    # Get the NMI matrix first
    nmiMatrix = getNMIMatrix(clusters, noOfLabels)
    # Get the NMI matrix first
    nmi = calcNMI(nmiMatrix)
    print("NMI is %.6f" % nmi)


# ================================================================
def getNMIMatrix(clusters, noOfLabels):
    # Matrix shape of [num_true_clusters + 1,num_output_clusters + 1] (example under week6's slide page 9)
    nmiMatrix = [[0 for x in range(len(clusters) + 1)] for y in range(noOfLabels + 1)]
    clusterNo = 0
    for cluster in clusters:
        # Create dictionary {true_class_No: Number of shared elements}
        labelCounts = {}
        # ========================#
        # STRART YOUR CODE HERE  #
        # ========================#
        for point in cluster:
            if point.label not in labelCounts:
                labelCounts[point.label] = 1
            else:
                labelCounts[point.label] += 1
        # ========================#
```

```
        #   END YOUR CODE HERE   #
        # ========================#
        labelTotal = 0
        labelCounts_sorted = sorted(labelCounts.items(), key=lambda item: item[1], reverse=True)
        for label, val in labelCounts_sorted:
            nmiMatrix[label - 1][clusterNo] = labelCounts[label]
            labelTotal += labelCounts.get(label)
        # Populate last row (row of summation)
        nmiMatrix[noOfLabels][clusterNo] = labelTotal
        clusterNo += 1
        labelCounts.clear()

    # Populate last col (col of summation)
    lastRowCol = 0
    for i in range(noOfLabels):
        totalRow = 0
        for j in range(len(clusters)):
            totalRow += nmiMatrix[i][j]
        lastRowCol += totalRow
        nmiMatrix[i][len(clusters)] = totalRow

    # Total number of datapoints
    nmiMatrix[noOfLabels][len(clusters)] = lastRowCol

    return nmiMatrix

# ======================================================================
def calcNMI(nmiMatrix):
    # Num of true clusters + 1
    row = len(nmiMatrix)
    # Num of output clusters + 1
    col = len(nmiMatrix[0])
    # Total number of datapoints
    N = nmiMatrix[row - 1][col - 1]
    I = 0.0
    HOmega = 0.0
    HC = 0.0

    for i in range(row - 1):
        for j in range(col - 1):
            # Compute the log part of each pair of clusters within I's formula.
            logPart_I = 1.0
            # ========================#
            # STRART YOUR CODE HERE  #
            # ========================#
            logPart_I = float(N) * nmiMatrix[i][j] / (nmiMatrix[-1][j] * nmiMatrix[i][-1])
            # ========================#
            #   END YOUR CODE HERE   #
            # ========================#

            if logPart_I == 0.0:
                continue
            I += (nmiMatrix[i][j] / float(N)) * math.log(float(logPart_I))

        # Compute HOmega
        # ========================#
        # STRART YOUR CODE HERE  #
        # ========================#
        HOmega += -1 * (nmiMatrix[i][-1] / N) * math.log((nmiMatrix[i][-1] / N))
        # ========================#
        #   END YOUR CODE HERE   #
        # ========================#

    #Compute HC
    # ========================#
    # STRART YOUR CODE HERE  #
```

```
    # ========================#
    for j in range(col - 1):
        HC += -1 * (nmiMatrix[-1][j] / N) * math.log((nmiMatrix[-1][j] / N))
    # ========================#
    #   END YOUR CODE HERE   #
    # ========================#

    return I / math.sqrt(HC * HOmega)




# ============================================================================
class Centroid:
    # ------------------------------------------------------------
    def __init__(self, x, y):
        self.x = x
        self.y = y
    # ------------------------------------------------------------
    def __eq__(self, other):
        if not type(other) is type(self):
            return False
        if other is self:
            return True
        if other is None:
            return False
        if self.x != other.x:
            return False
        if self.y != other.y:
            return False
        return True
    # ------------------------------------------------------------
    def __ne__(self, other):
        result = self.__eq__(other)
        if result is NotImplemented:
            return result
        return not result
    # ------------------------------------------------------------
    def toString(self):
        return "Centroid [x=" + str(self.x) + ", y=" + str(self.y) + "]"
    # ------------------------------------------------------------
    def __str__(self):
        return self.toString()
    # ------------------------------------------------------------
    def __repr__(self):
        return self.toString()









# ============================================================================
class KMeans:
    # ------------------------------------------------------------
    def __init__(self):
        self.K = 0
    # ------------------------------------------------------------
    def main(self, dataname,isevaluate=False):
        seed = 71
        self.dataname = dataname[5:-4]
        print("\nFor " + self.dataname)
        self.dataSet = self.readDataSet(dataname)
```

```
196         self.K = DataPoints.getNoOFLabels(self.dataSet)
197         random.Random(seed).shuffle(self.dataSet)
198         self.kmeans(isevaluate)
199 
200     # ---------------------------------------------------------------
201     def check_dataloader(self,dataname):
202 
203         df = pd.read_table(dataname,sep = "\t", header=None, names=['x','y','ground_truth_cluster'])
204         print("\nFor " + dataname[5:-4] + ": number of datapoints is %d" % df.shape[0])
205         print(df.head(5))
206 
207 
208     # ---------------------------------------------------------------
209     def kmeans(self,isevaluate=False):
210         clusters = []
211         k = 0
212         while k < self.K:
213             cluster = set()
214             clusters.append(cluster)
215             k += 1
216 
217         # Initially randomly assign points to clusters
218         i = 0
219         for point in self.dataSet:
220             clusters[i % k].add(point)
221             i += 1
222 
223         # calculate centroid for clusters
224         centroids = []
225         for j in range(self.K):
226             centroids.append(self.getCentroid(clusters[j]))
227 
228         self.reassignClusters(self.dataSet, centroids, clusters)
229 
230         # continue till converge
231         iteration = 0
232         while True:
233             iteration += 1
234             # calculate centroid for clusters
235             centroidsNew = []
236             for j in range(self.K):
237                 centroidsNew.append(self.getCentroid(clusters[j]))
238 
239             isConverge = False
240             for j in range(self.K):
241                 if centroidsNew[j] != centroids[j]:
242                     isConverge = False
243                 else:
244                     isConverge = True
245             if isConverge:
246                 break
247 
248             for j in range(self.K):
249                 clusters[j] = set()
250 
251             self.reassignClusters(self.dataSet, centroidsNew, clusters)
252             for j in range(self.K):
253                 centroids[j] = centroidsNew[j]
254         print("Iteration :" + str(iteration))
255 
256         if isevaluate:
257             # Calculate purity and NMI
258             compute_purity(clusters, len(self.dataSet))
259             compute_NMI(clusters, self.K)
260 
261 
```

```
        # write clusters to file for plotting
        f = open("Kmeans_"+ self.dataname + ".csv", "w")
        for w in range(self.K):
            print("Cluster " + str(w) + " size :" + str(len(clusters[w])))
            print(centroids[w].toString())
            for point in clusters[w]:
                f.write(str(point.x) + "," + str(point.y) + "," + str(w) + "\n")
        f.close()

    # ------------------------------------------------------------------
    def reassignClusters(self, dataSet, c, clusters):
        # reassign points based on cluster and continue till stable clusters found
        dist = [0.0 for x in range(self.K)]
        for point in dataSet:
            for i in range(self.K):
                dist[i] = getEuclideanDist(point.x, point.y, c[i].x, c[i].y)

            minIndex = self.getMin(dist)
            # assign point to the closest cluster
            # ========================#
            # STRART YOUR CODE HERE  #
            # ========================#
            for clus in clusters:
                if point in clus:
                    clus.remove(point)

            clusters[minIndex].add(point)
            # ========================#
            #   END YOUR CODE HERE   #
            # ========================#
    # ------------------------------------------------------------------
    def getMin(self, dist):
        min = sys.maxsize
        minIndex = -1
        for i in range(len(dist)):
            if dist[i] < min:
                min = dist[i]
                minIndex = i
        return minIndex

    # ------------------------------------------------------------------
    def getCentroid(self, cluster):
        # mean of x and mean of y
        cx = 0
        cy = 0
        # ========================#
        # STRART YOUR CODE HERE  #
        # ========================#
        for c in cluster:
            cx += c.x
            cy += c.y

#       print(len(cluster))
        cx = cx / len(cluster)
        cy = cy / len(cluster)
        # ========================#
        #   END YOUR CODE HERE   #
        # ========================#
        return Centroid(cx, cy)
    # ------------------------------------------------------------------
    @staticmethod
    def readDataSet(filePath):
        dataSet = []
        with open(filePath) as f:
            lines = f.readlines()
```

```
328         lines = [x.strip() for x in lines]
329         for line in lines:
330             points = line.split('\t')
331             x = float(points[0])
332             y = float(points[1])
333             label = int(points[2])
334             point = DataPoints(x, y, label)
                dataSet.append(point)
            return dataSet
```

```
from hw4code.KMeans import KMeans,compute_purity,compute_NMI,getEuclideanDist
from hw4code.DataPoints import DataPoints
import random


class DBSCAN:
    # ---------------------------------------------------------------------
    def __init__(self):
        self.e = 0.0
        self.minPts = 3
        self.noOfLabels = 0
    # ---------------------------------------------------------------------
    def main(self, dataname):
        seed = 71

        self.dataname = dataname[5:-4]
        print("\nFor " + self.dataname)
        self.dataSet = KMeans.readDataSet(dataname)
        random.Random(seed).shuffle(self.dataSet)
        self.noOfLabels = DataPoints.getNoOFLabels(self.dataSet)
        self.e = self.getEpsilon(self.dataSet)
        print("Esp :" + str(self.e))
        self.dbscan(self.dataSet)


    # ---------------------------------------------------------------------
    def getEpsilon(self, dataSet):
        distances = []
        sumOfDist = 0.0
        for i in range(len(dataSet)):
            point = dataSet[i]
            for j in range(len(dataSet)):
                if i == j:
                    continue
                pt = dataSet[j]
                dist = getEuclideanDist(point.x, point.y, pt.x, pt.y)
                distances.append(dist)

            distances.sort()
            sumOfDist += distances[7]
            distances = []
        return sumOfDist/len(dataSet)
    # ---------------------------------------------------------------------
    def dbscan(self, dataSet):
        clusters = []
        visited = set()
        noise = set()

        # Iterate over data points
        for i in range(len(dataSet)):
            point = dataSet[i]
            if point in visited:
                continue
            visited.add(point)
            N = []
            minPtsNeighbours = 0

            # check which point satisfies minPts condition
            for j in range(len(dataSet)):
                if i==j:
                    continue
                pt = dataSet[j]
                dist = getEuclideanDist(point.x, point.y, pt.x, pt.y)
                if dist <= self.e:
```

```
                minPtsNeighbours += 1
                N.append(pt)

            if minPtsNeighbours >= self.minPts:
                cluster = set()
                cluster.add(point)
                point.isAssignedToCluster = True

                j = 0
                while j < len(N):
                    point1 = N[j]
                    minPtsNeighbours1 = 0
                    N1 = []
                    if not point1 in visited:
                        visited.add(point1)
                        for l in range(len(dataSet)):
                            pt = dataSet[l]
                            dist = getEuclideanDist(point1.x, point1.y, pt.x, pt.y)
                            if dist <= self.e:
                                minPtsNeighbours1 += 1
                                N1.append(pt)
                        if minPtsNeighbours1 >= self.minPts:
                            self.removeDuplicates(N, N1)

                    # Add point1 is not yet member of any other cluster then add it to cluster
                    # Hint: use self.isAssignedToCluster function to check if a point is assigned to any clusters
                    # ========================#
                    # STRART YOUR CODE HERE  #
                    # ========================#
                    if not point1.isAssignedToCluster:
                        if point in noise:
                            noise.remove(point1)
                        cluster.add(point1)
                    # ========================#
                    #   END YOUR CODE HERE   #
                    # ========================#
                    j += 1

                # add cluster to the list of clusters
                clusters.append(cluster)

            else:
                noise.add(point)


        # List clusters
        print("Number of clusters formed :" + str(len(clusters)))
        print("Noise points :" + str(len(noise)))

        # Calculate purity
        compute_purity(clusters,len(self.dataSet))
        compute_NMI(clusters,self.noOfLabels)
        DataPoints.writeToFile(noise, clusters, "DBSCAN_"+ self.dataname + ".csv")
    # ------------------------------------------------------------------
    def removeDuplicates(self, n, n1):
        for point in n1:
            isDup = False
            for point1 in n:
                if point1 == point:
                    isDup = True
                    break
            if not isDup:
                n.append(point)
```

```
from hw4code.DataPoints import DataPoints
from hw4code.KMeans import KMeans, compute_purity,compute_NMI
import math
from scipy.stats import multivariate_normal

# =============================================================================
class GMM:
    # ---------------------------------------------------------------------
    def __init__(self):
        self.dataSet = []
        self.K = 0
        self.mean = [[0.0 for x in range(2)] for y in range(3)]
        self.stdDev = [[0.0 for x in range(2)] for y in range(3)]
        self.coVariance = [[[0.0 for x in range(2)] for y in range(2)] for z in range(3)]
        self.W = None
        self.w = None
    # ---------------------------------------------------------------------
    def main(self, dataname):

        self.dataname = dataname[5:-4]
        print("\nFor " + self.dataname)
        self.dataSet = KMeans.readDataSet(dataname)
        self.K = DataPoints.getNoOFLabels(self.dataSet)
        # weight for pair of data and cluster
        self.W = [[0.0 for y in range(self.K)] for x in range(len(self.dataSet))]
        # weight for pair of data and cluster
        self.w = [0.0 for x in range(self.K)]
        self.GMM()

    # ---------------------------------------------------------------------
    def GMM(self):
        clusters = []
        # [num_clusters,2]
        self.mean = [[0.0 for y in range(2)] for x in range(self.K)]
        # [num_clusters,2]
        self.stdDev = [[0.0 for y in range(2)] for x in range(self.K)]
        # [num_clusters,2]
        self.coVariance = [[[0.0 for z in range(2)] for y in range(2)] for x in range(self.K)]
        k = 0
        while k < self.K:
            cluster = set()
            clusters.append(cluster)
            k += 1

        # Initially randomly assign points to clusters
        i = 0
        for point in self.dataSet:
            clusters[i % self.K].add(point)
            i += 1

        # Initially assign equal prior weight for each cluster
        for m in range(self.K):
            self.w[m] = 1.0 / self.K

        # Get Initial mean, std, covariance matrix
        DataPoints.getMean(clusters, self.mean)
        DataPoints.getStdDeviation(clusters, self.mean, self.stdDev)
        DataPoints.getCovariance(clusters, self.mean, self.stdDev, self.coVariance)

        length = 0
        while True:
            mle_old = self.Likelihood()
            self.Estep()
            self.Mstep()
```

```
65          length += 1
66          mle_new = self.Likelihood()
67
68          # convergence condition
69          if abs(mle_new - mle_old) / abs(mle_old) < 0.000001:
70              break
71
72      print("Number of Iterations = " + str(length))
73      print("\nAfter Calculations")
74      print("Final mean = ")
75      self.printArray(self.mean)
76      print("\nFinal covariance = ")
77      self.print3D(self.coVariance)
78
79      # Assign points to cluster depending on max prob.
80      for j in range(self.K):
81          clusters[j] = set()
82
83      i = 0
84      for point in self.dataSet:
85          index = -1
86          prob = 0.0
87          for j in range(self.K):
88              if self.W[i][j] > prob:
89                  index = j
90                  prob = self.W[i][j]
91          temp = clusters[index]
92          temp.add(point)
93          i += 1
94
95      # Calculate purity and NMI
96      compute_purity(clusters,len(self.dataSet))
97      compute_NMI(clusters,self.K)
98
99      # write clusters to file for plotting
100     f = open("GMM_" + self.dataname + ".csv", "w")
101     for w in range(self.K):
102         print("Cluster " + str(w) + " size :" + str(len(clusters[w])))
103         for point in clusters[w]:
104             f.write(str(point.x) + "," + str(point.y) + "," + str(w) + "\n")
105     f.close()
106 # ---------------------------------------------------------------------
107 def Estep(self):
108     # Update self.W
109     for i in range(len(self.dataSet)):
110         denominator = 0.0
111         for j in range(self.K):
112             gaussian = multivariate_normal(self.mean[j], self.coVariance[j])
113             # Compute numerator for self.W[i][j] below
114             numerator = 0.0
115             # ========================#
116             # STRART YOUR CODE HERE  #
117             # ========================#
118             numerator = self.w[j] * gaussian.pdf([self.dataSet[i].x, self.dataSet[i].y])
119             # ========================#
120             #   END YOUR CODE HERE   #
121             # ========================#
122             self.W[i][j] = numerator
123             denominator += numerator
124
125         # normalize W[i][j] into probabilities
126         # ========================#
127         # STRART YOUR CODE HERE  #
128         # ========================#
129
130
```

```
131         for j in range(self.K):
132             self.W[i][j] = self.W[i][j] / denominator
133         # ========================#
134         #   END YOUR CODE HERE   #
135         # ========================#
136     # ---------------------------------------------------------------
137     def Mstep(self):
138         for j in range(self.K):
139             denominator = 0.0
140             numerator_x = 0.0
141             numerator_y = 0.0
142             cov_xy = 0.0
143             updatedMean_x = 0.0
144             updatedMean_y = 0.0
145 
146             # update self.w[j] and self.mean
147             for i in range(len(self.dataSet)):
148                 denominator += self.W[i][j]
149                 updatedMean_x += self.W[i][j] * self.dataSet[i].x
150                 updatedMean_y += self.W[i][j] * self.dataSet[i].y
151 
152             self.w[j] = denominator / len(self.dataSet)
153 
154             # update self.mean
155             # ========================#
156             # STRART YOUR CODE HERE  #
157             # ========================#
158             self.mean[j][0] = updatedMean_x / denominator
159             self.mean[j][1] = updatedMean_y / denominator
160             # ========================#
161             #   END YOUR CODE HERE   #
162             # ========================#
163 
164             # update covariance matrix
165             for i in range(len(self.dataSet)):
166                 numerator_x += self.W[i][j] * pow((self.dataSet[i].x - self.mean[j][0]), 2)
167                 numerator_y += self.W[i][j] * pow((self.dataSet[i].y - self.mean[j][1]), 2)
168                 # Compute conv_xy +=?
169                 # ========================#
170                 # STRART YOUR CODE HERE  #
171                 # ========================#
172                 cov_xy += self.W[i][j] * (self.dataSet[i].x - self.mean[j][0]) * (self.dataSet[i].y - self.mean[j][1])
173                 # ========================#
174                 #   END YOUR CODE HERE   #
175                 # ========================#
176 
177             self.stdDev[j][0] = numerator_x / denominator
178             self.stdDev[j][1] = numerator_y / denominator
179 
180 
181             self.coVariance[j][0][0] = self.stdDev[j][0]
182             self.coVariance[j][1][1] = self.stdDev[j][1]
183             self.coVariance[j][0][1] = self.coVariance[j][1][0] = cov_xy / denominator
184     # ---------------------------------------------------------------
185     def Likelihood(self):
186         likelihood = 0.0
187         for i in range(len(self.dataSet)):
188             numerator = 0.0
189             for j in range(self.K):
190                 gaussian = multivariate_normal(self.mean[j], self.coVariance[j])
191                 numerator += self.w[j] * gaussian.pdf([self.dataSet[i].x, self.dataSet[i].y])
192             likelihood += math.log(numerator)
193         return likelihood
194     # ---------------------------------------------------------------
195     def printArray(self, mat):
196         for i in range(len(mat)):
```

```
197             for j in range(len(mat[i])):
198                 print(str(mat[i][j]) + " "),
199             print("")
200     # --------------------------------------------------------------------
201     def print3D(self, mat):
202         for i in range(len(mat)):
203             print("For Cluster : " + str((i + 1)))
204             for j in range(len(mat[i])):
205                 for k in range(len(mat[i][j])):
206                     print(str(mat[i][j][k]) + " "),
207                 print("")
208             print("")
209
210
211 # ============================================================================
212 if __name__ == "__main__":
213     g = GMM()
      dataname = "dataset1.txt"
      g.main(dataname)
```