HW2 Report

> Author: Jiayi Liu (9749111299)

This is an individual work. The project structure is as follows

```
# Entry Point
- main.py
# K-means Implementation
- kmeans.py
# GMM Implementation
- gmm.py
# Visualization Helper
- visualize.py
```

Implementation Detail

To better understand the distribution of the clusters, I implemented the plotting function with the help of `matplotlob`. All of the visualization parts were commented in the code for avoiding affecting the main functions. You can just open the comments to see the plotting figures.

0. Read file

- Input data points from `clusters.txt` line by line and store the coordinates in a list.
- Calculate the scope of data points' coordinates for the convience of specifying the range of randomization for centroids.
- All the information was returned as a dictionary.

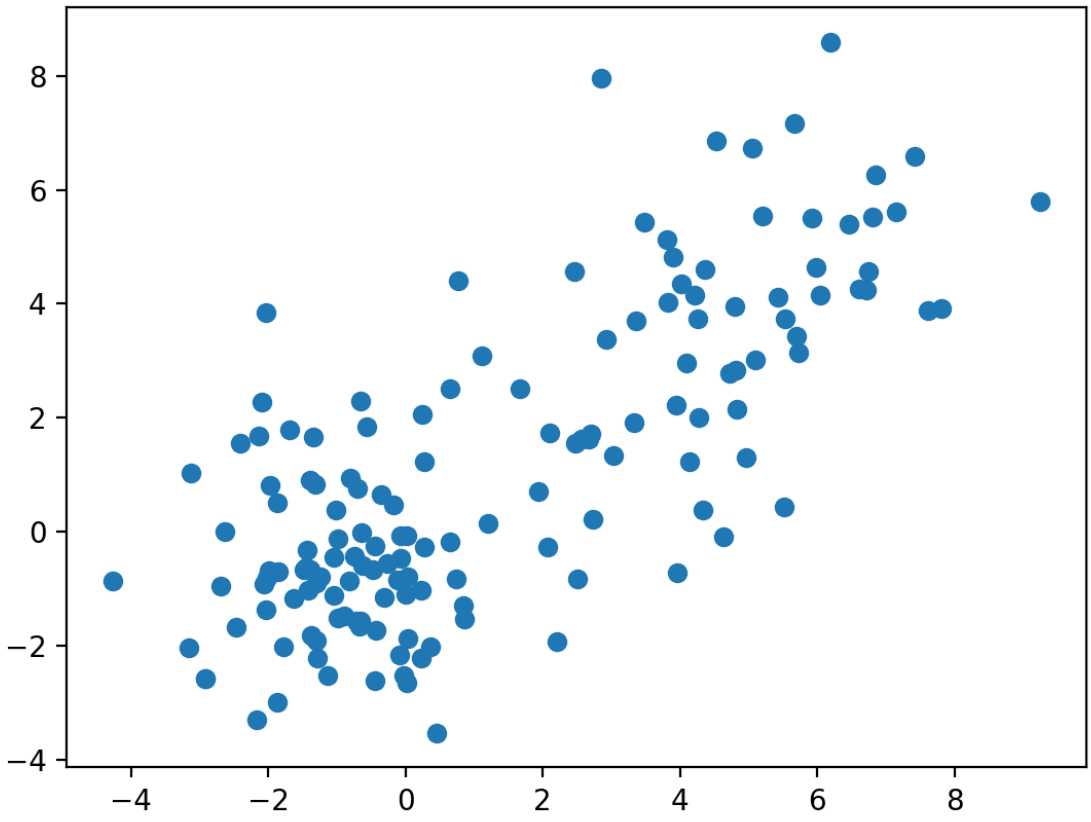
```
def read_file(file_path):
    points = []
    xs = []
    ys = []
    with open(file_path) as f:
        lines = f.readlines()
        for line in lines:
            x, y = line.split(',')
            xs.append(float(x))
            ys.append(float(y))
            points.append((float(x), float(y)))

    # Visualization
    # plot_original(xs, ys)
```

```
    # Scope Calculate
    x_max = max(xs)
    x_min = min(xs)
    y_max = max(ys)
    y_min = min(ys)

    return {"points": points, "x_scope":(x_min, x_max), "y_scope": (y_min, y_max)}
```

To know the distribution of data points, I create a scattered plot.



1. K-Means Clustering

1.1 Initialization

All the logic of implementing a k-means clustering is encapsulated in the class `KMeans` in the `kmeans.py` file. The information above was passed into the construction function.

```
class KMeans:
    def __init__(self, info):
        self.K = 3
        self.points = info["points"]                                    #
data points
        self.centroids = self.init_centroid(info["x_scope"], info["y_scope"]) # K
centroids
        self.assignments = []                                           #
Just for visualization
```

- `self.K`: As mentioned in the description, k is specified as 3.
- `self.points`: A list of coordinates of data points.
- `self.centroids`: A list of coordinates of K centroids. They were initialized according to the scope calculated. The initialization function is as follows.
- `self.assignments`: A list of indices of data points assigned to their closest centroid. In order to check the correctness of the algorithm, I implemented the visualization function. This variable is just for visualizing the cluster distribution.

```
def init_centroid(self, x_scope, y_scope):
        centroids = []
        for i in range(self.K):
            x = random.uniform(x_scope[0], x_scope[1])
            y = random.uniform(y_scope[0], y_scope[1])
            centroids.append((x, y))
        return centroids
```

1.2 Train Model

Model training is a recursive process of assigning data points to their closest centroid and recalculating the centroids. `clustering` is the recursive function itself. After convergence, the coordinates of *k* centroids will be printed out, and centroids' location and the clustering distribution will be visualized.

```
def train(self):
        self.clustering()
        print("------Final Centroids-----")
        for centroid in self.centroids:
            print(centroid)

        # Visualization
        # plot_centroids(self.points, self.centroids)
        # plot_clusters(self.points, self.assignments)
```

For recursion, I set up a `need_stop` flag to indicate if the algorithm is convergent and recursion needs stopping.

```
def clustering(self, need_stop=False):
        # Termination check
        if need_stop:
            return

        # Init assignment list: points mapping to the closest centroid
        assignments = []
        for i in range(self.K):
            assignments.append([])

        # Assign data points to the closest centroid
        for idx, point in enumerate(self.points):
            closest = self.find_closest(point)
            assignments[closest].append(idx)

        # Just for clustering visualization
        self.assignments = assignments

        # Recalculate the centroids
        new_centroids = []
        new_centroids = self.calculate_centroid(assignments)

        # Check if reach the stable status
        # If not, update the centroids
        # Otherwise, update the stop flag
        stop_flag = self.check_stable(new_centroids)

        if not stop_flag:
            self.centroids = new_centroids

        # Recursion
        self.clustering(stop_flag)

        return
```

The `clustering` function basically consists of 4 steps:

- Check the termination flag first
- If it is not convergent, start to assign each data point to its closest centroid
- After assignment process, recalculate the centroids.
- Check if the algorithm is getting convergence

Centroid Recalculation

For each temporary cluster, new centroid was calculated according to the mean of assigned data points. Here we have a corner case that if no data point was assigned to this centroid, we reset this centroid to one of the data point randomly (here I just use the first data point as the replacement).

```
def calculate_centroid(self, assignments):
        centroids = []
        for i in range(self.K):
            centroid = self.mean(assignments[i])
            centroids.append(centroid)
        return centroids

def mean(self, indices):
        if len(indices) == 0:
            return self.points[0]
        sum_x = 0
        sum_y = 0
        for idx in indices:
            sum_x += self.points[idx][0]
            sum_y += self.points[idx][1]
        return (sum_x / len(indices), sum_y / len(indices))
```

Convergence Check

To know if the algorithm has converged, I use `check_stable` function to check if the recalculated centroids are same as the previous ones.

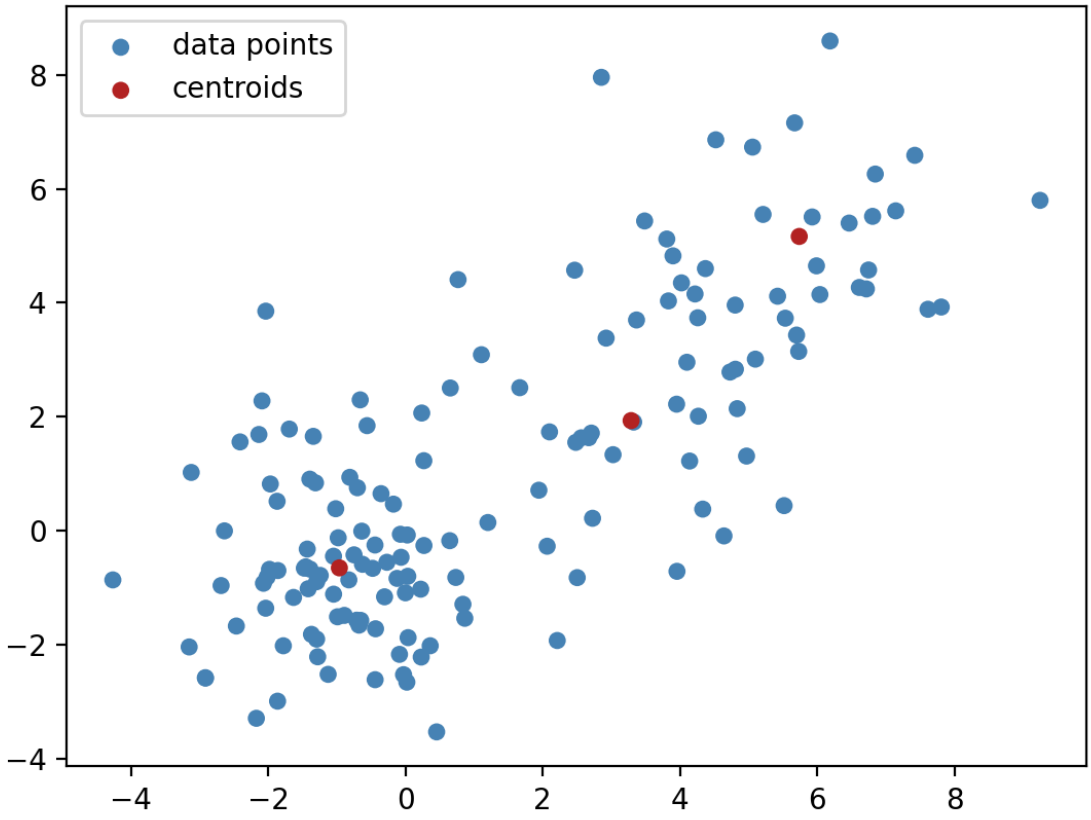
```
def check_stable(self, new_centroids):
        for idx, centroid in enumerate(new_centroids):
            if centroid[0] != self.centroids[idx][0] or centroid[1] !=
self.centroids[idx][1]:
                return False
        return True
```

1.3 Clustering Result

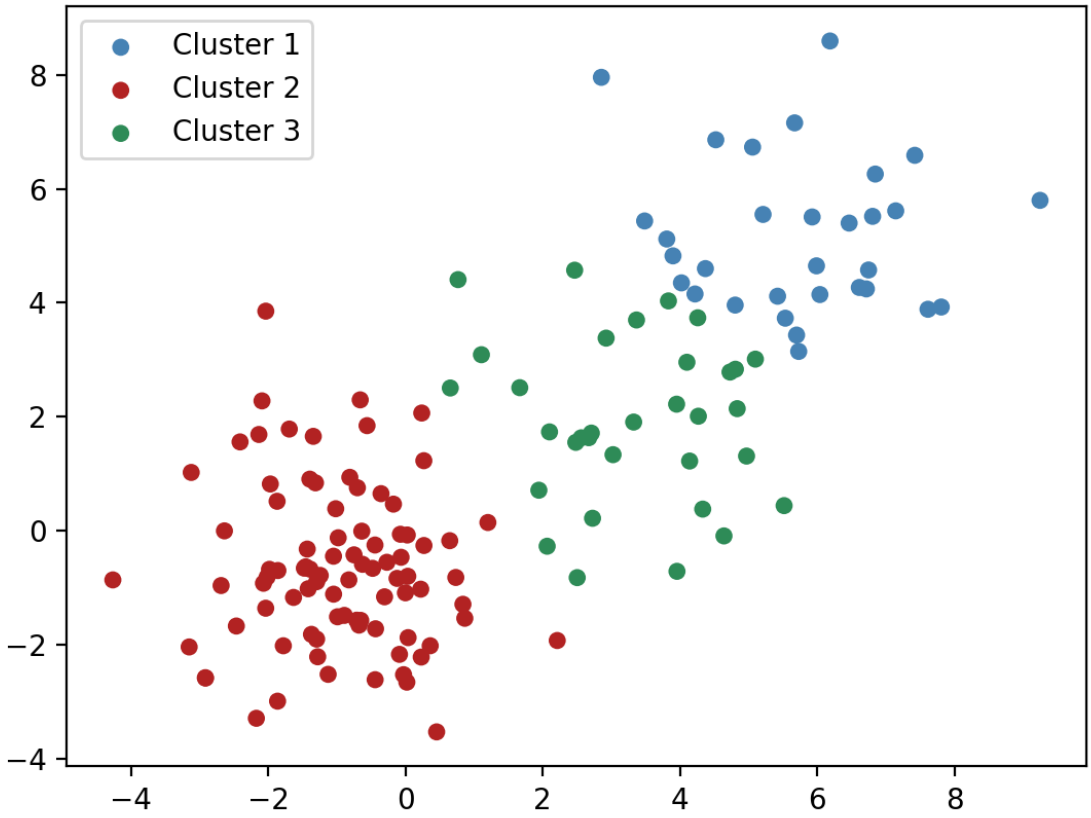
After running `main.py`, we can see the centroids' coordinates printed out.

```
------Final Centroids-----
(3.2888485605151514, 1.9326883657575762)
(-0.9606529070232559, -0.6522184128604652)
(5.738495346032257, 5.164838081193549)
```

To make it more clear, I plotted the location of centroids in the figure.



In addition, I plotted the clusters' distribution in the figure.



2. GMM Clustering

2.1 Initialization

All the logic of implementing a GMM clustering is encapsulated in the class `GMM` in the `gmm.py` file. The input was passed into the construction function. For the convinience of matrix operations, I used NumPy library to represent variables here.

```
class GMM:
    def __init__(self, info):
        self.K = 3
        self.threshold = 1e-7                                    # threshold for
convergence
        self.points = np.array(info["points"])                   # numpy array of
data points (N x D)
        self.N, self.dim = self.points.shape                     # markdown the
shape of data array
        self.weights = self.random_guess()                       # numpy array of
weights (N x K)
        self.means = np.zeros((self.K, self.dim))                # numpy array of
means (K x D)
        self.covariances = np.zeros((self.K, self.dim, self.dim)) # numpy array of
covariance matrix (K x D x D)
        self.amplitudes = np.zeros((self.K,))                    # numpy array of
amplitudes (K, )
```

2.2 Train Model

GMM training is a recursive process. `clustering` is the recursive function itself. After convergence, data points were assigned to the cluster with the highest weight for the convience of visualization.

```
def clustering(self, stop_flag=False):
        # Check Termination
        if stop_flag:
            return

        # M Step
        means, covariances, amplitudes = self.m_step()

        # E Step
        weights = self.e_step(means, covariances, amplitudes)

        # Check convergence
        need_stop = self.check_convergence(weights)

        # Update variables
        self.weights = weights
        self.means = means
        self.covariances = covariances
        self.amplitudes = amplitudes

        # Recursion
        self.clustering(need_stop)
```

`stop_flag` is the indicator of whether to continue recursion process. In each recursion process, we check the termination condition first, then conduct the M-step and E-step, respectively. After updating the information of each Gaussian distribution, we go to check if it has reached the convergence status.

M-Step

In M-step, we calculate the weighted mean, weighted covariance matrix, and amplitude for each Gaussian cluster by aggregating the weight information of data points.

```
def m_step(self):
        # Weighted mean
        means = self.weighted_mean()

        # Amplitudes
        amplitudes = np.sum(self.weights, axis=0) / self.N

        # Weighted covariance
        covariances = self.weighted_covariance(means)

        return means, covariances, amplitudes
```

E-Step

In E-Step, we recalculate the new weights according to the Gaussian distribution we just obtained.

```
def e_step(self, means, covariances, amplitudes):
        weights = np.zeros_like(self.weights)
        for c in range(self.K):
            mean = means[c]
            cov = covariances[c]
            amplitude = amplitudes[c]
            for i, point in enumerate(self.points):
                weights[i, c] = Gaussian_2D(mean, cov, amplitude, point)
        normalize_factor = np.sum(weights, axis=1)
        return (weights.T / normalize_factor).T
```

The 2D Gaussian function is implemented as follows.

```
def Gaussian_2D(mean, cov, amplitude, point):
    A = amplitude / np.sqrt( 2 * np.pi * np.linalg.det(cov))
    return A * np.exp(-0.5 * np.dot(point - mean, np.linalg.inv(cov)).dot((point-
mean).T))
```

Convergence Check

I check the convergence status by estimating if the difference of new weights and old weights is larger than the threshold I set. If all the weights get to be stable, then return true as the `stop_flag` to stop the recursion process.

```
def check_convergence(self, weights):
        return np.all(np.abs((weights - self.weights)) - self.threshold <= 0)
```

1.3 Clustering Result

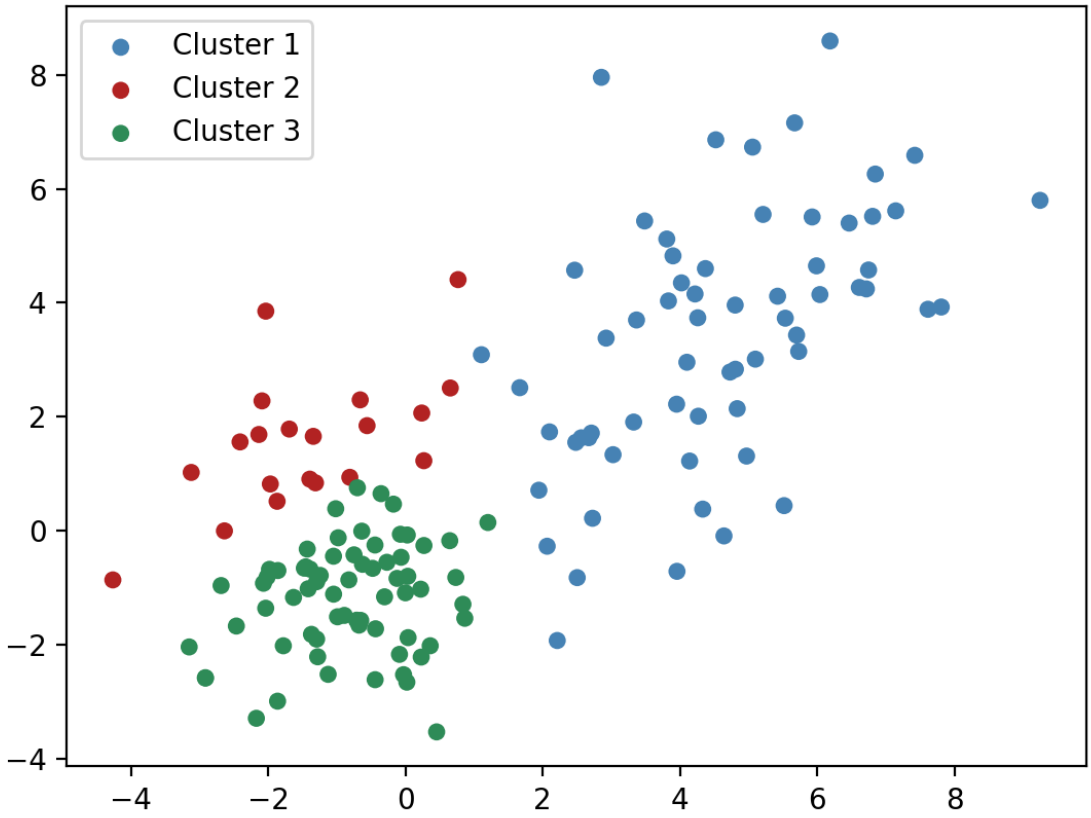
After the completion of recursion process, the information of gaussian distribution will be printed out as follows.

```
----------- mean -----------
[[ 4.45404428  3.35430883]
 [-1.34482001  1.38022794]
 [-0.8451978  -1.12264961]]
-------- covariance --------
[[[3.44045258 2.29943654]
  [2.29943654 5.1720779 ]]

 [[1.60466288 0.74371594]
  [0.74371594 1.74883038]]

 [[1.05766765 0.13954123]
  [0.13954123 1.02259689]]]
--------- amplitude --------
[0.4281226  0.12969684 0.44218057]
```

To better visualize the result, I plotted the figure of clustering.



3 Discussion

The clustering results from k-means and GMM algorithm are totally different. We can infer from the figures that k-means algorthm tends to gather spherical clusters, while GMM algorthm tends to gather elliptic clusters. From the perspectve of computation complexity, GMM is more time-consuming because of matrix operations and more iterations to converge.