INF-552
MACHINE LEARNING FOR DATA SCIENCE

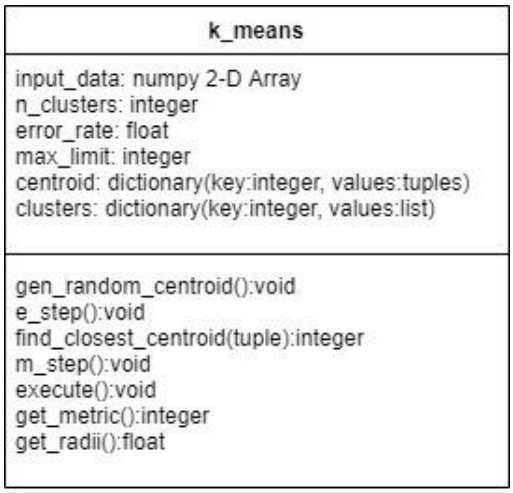
PROGRAMMING ASSIGNMENT 2:
K-MEANS & GMM

NAME	USC-ID	EMAIL
Amitabh Rajkumar Saini	7972003272	amitabhr@usc.edu
Shilpa Jain	4569044628	shilpaj@usc.edu
Sushumna Khandelwal	7458911214	sushumna@usc.edu

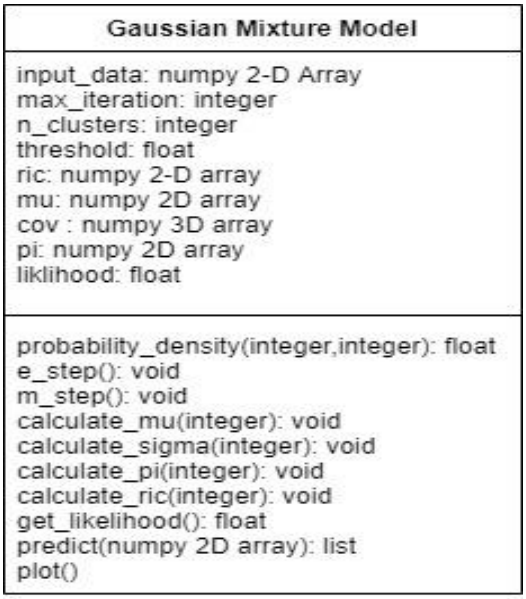
PART 1: IMPLEMENTATION

- **Data Structures Defined/Used**
  - Dataset represented as numpy 2-D Array
  - Parameters used in Kmeans is Centroid and Gaussian Mixture Model is Mean, Variance and Size
  - The approach Kmeans and Gaussian Mixture Model follows to solve the problem is called **Expectation-**Maximization. The E-step is assigning the data points to the closest cluster. The M-step is computing the parameters of each cluster.

  - **Kmeans**
    Kmeans algorithm is an iterative algorithm that tries to partition the dataset into *K*pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to **only one group**. Following is the full structure of a k_means class (Kmeans Algorithm)



  - **Gaussian Mixture Model**
    Gaussian mixture models can be used to cluster unlabeled data in much the same way as k-means. There are, however, a couple of advantages to using Gaussian mixture models over k-means.



- **Code: (Language - Python)**

```
'''

PA-2: K-Means and GMM
Authors:
Amitabh Rajkumar Saini, amitabhr@usc.edu
Shilpa Jain, shilpaj@usc.edu
Sushumna Khandelwal, sushumna@usc.edu

Dependencies:
1. numpy: pip install numpy
2. matplotlib: pip install matplotlib

Output:
Returns a k-means and gmm model, writes model parameters on console
and generates the plot of the same

'''

import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import math
import copy
import itertools


class k_means:
    '''
    k_means class defines functions and variables to run the k-means
algorithm
    '''

    def __init__(self, input_data, n_clusters, max_limit, error_rate):
        '''
        Constructs k-means object
        :param input_data: input data to make the model
        :param n_clusters: no. of clusters to be generated
        :param max_limit: maximum number of iterations for convergence
        :param error_rate: error which can be accomodated in model
        :return: return object for running k-means
        '''
        self.input_data = input_data
        self.n_clusters = n_clusters
        self.error_rate = error_rate
        self.max_limit = max_limit
        self.error_rate = error_rate
        self.centroid = dict()  # {1:(),2:()..n_clusters}
        self.clusters = dict()  #
{1:[(),(),(),],2:[(),(),(),]..n_clusters}

    def gen_random_centroid(self):
        '''
        Generates initial random centroids from the input data
        :return: returns nothing
        '''
        list_centroid =
```

```
self.input_data[np.random.choice(self.input_data.shape[0],
self.n_clusters, replace=False), :]
        for i in range(self.n_clusters):
            self.centroid[i] = list_centroid[i]
            self.clusters[i] = []

    def e_step(self):
        '''
        Runs e-step of the k-means, to find closest centroid and
assign the data to that cluster
        :return: returns nothing
        '''
        for data in self.input_data:
            id_ = self.find_closest_centroid(data)
            self.clusters[id_].append(data)

    def find_closest_centroid(self, point):
        '''
        Finds the closest centroid to the input data point
        :param point: data point
        :return: returns closest cluster_id
        '''
        closest_dist = float("inf")
        closest_centroid = None

        for each in self.centroid.keys():
            dist = self.calculate_distance(self.centroid[each], point)
            if dist < closest_dist:
                closest_dist = dist
                centroid_id = each

        return centroid_id

    def calculate_distance(self, centroid, cluster):
        '''
        Calculates euclidean distance between 2 points
        :param centroid: point 1
        :param cluster: point 2
        :return: return euclidean distance
        '''
        return np.linalg.norm(centroid - cluster)

    def m_step(self):
        '''
        Runs m-step of k-means algorith to recompute centroids from
clusters generated by e-step
        :return: returns nothing
        '''
        for key in self.clusters.keys():
            list_of_points = self.clusters[key]
            new_centroid = np.mean(list_of_points, axis=0)
            self.centroid[key] = new_centroid

    def execute(self):
        '''
        Runs the EM Algorith for gmm
        :return: returns nothing
        '''
        self.gen_random_centroid()
```

```
        current_error = float("inf")
        current_iteration = 1
        while current_error > self.error_rate and current_iteration <
self.max_limit:
            self.e_step()
            old_centroid = copy.copy(self.centroid)  # copy dict
            self.m_step()
            new_centroid = copy.copy(self.centroid)  # copy dict
            dif_list = []
            for each in self.centroid:
                dif_list.append(abs(new_centroid[each] -
old_centroid[each]))
            current_error = np.mean(dif_list)
            current_iteration += 1

    def get_metric(self):
        '''
        Calculates metric for a model
        :return: returns the metric value
        '''
        metric = 0
        for each in self.clusters:
            for point in self.clusters[each]:
                metric += self.calculate_distance(self.centroid[each],
point)
        return metric

    def get_radii(self, cluster_id):
        '''
        Find maximum radii of a given cluster
        :param cluster_id: cluster_id
        :return: returns radius of the cluster
        '''
        radii = float('-inf')
        for point in self.clusters[cluster_id]:
            radii = max(radii,
self.calculate_distance(self.centroid[cluster_id], point))
        return radii

    def plot(self):
        '''
        Plots the k-means cluster and write output to console
        :return: returns nothing
        '''
        colors = list("rgy")
        for each in self.clusters:
            temp = np.asarray(self.clusters[each])
            plt.scatter(temp[:, 0], temp[:, 1], color=colors.pop(), )
        print("------------K-Means------------")
        print("Centroid:")
        print(self.centroid)
        centers = np.asarray(list(self.centroid.values()))
        axes = plt.gca()
        plt.scatter(centers[:, 0], centers[:, 1], c='black')
        for i in range(self.n_clusters):
            axes.add_patch(plt.Circle(centers[i], self.get_radii(i),
fc='#CCCCCC', lw=3, alpha=0.2, zorder=2))
        plt.title("K-Means")
        plt.show()
```

```
class gmm:
    '''
    gmm class defines functions and variables to run the gmm algorithm
    '''

    def __init__(self, data, max_iteration, n_clusters,
threshold=0.5):
        '''
        Constructs gmm object
        :param data: input data to make the model
        :param max_iteration: maximum number of iterations for
convergence
        :param n_clusters: no. of clusters to be generated
        :param threshold: error which can be accomodated in model
        :return: return object for running gmm
        '''
        self.input_data = data
        self.max_iteration = max_iteration
        self.n_clusters = n_clusters
        self.threshold = threshold
        self.ric = np.full((self.input_data.shape[0], n_clusters), 1 /
self.n_clusters)
        for i in range(self.ric.shape[0]):
            x = random.uniform(0, 1)
            y = random.uniform(0, (1 - x))
            z = 1 - (x + y)
            self.ric[i] = np.asarray([x, y, z])
        # print(self.ric)
        self.mu = np.zeros((n_clusters, self.input_data.shape[1]))
        # print(self.mu)
        self.cov = np.zeros((n_clusters, self.input_data.shape[1],
self.input_data.shape[1]))
        # print(self.cov)
        self.pi = np.zeros(n_clusters)
        self.likelihood = 0
        # print(self.pi)

    def probability_density(self, i, gaussian_id):
        '''
        Computes the probablility density function for multivariate
normal distribution
        :param gaussian_id: gaussian_id
        :returns : return pdf(probability density function) value
        '''
        try:
            probability = 1 / pow((2 * math.pi), -self.n_clusters / 2)
* pow(abs(np.linalg.det(self.cov[gaussian_id])),

-1 / 2) * \
                           np.exp(-1 / 2 *
np.dot(np.dot((self.input_data[i] - self.mu[gaussian_id]).T,

np.linalg.inv(self.cov[gaussian_id])),
                                                  (self.input_data[i] -
self.mu[gaussian_id])))
        except:
            print(np.linalg.inv(self.cov[gaussian_id]))
```

```
        return probability

    def e_step(self):
        '''
        Runs e-step of the gmm, to calculate gaussian parameters
mean,co-variance and amplitude
        :return: returns nothing
        '''
        for i in range(self.n_clusters):
            self.calculate_mu(i)
            self.calculate_pi(i)
            self.calculate_sigma(i)

    def m_step(self):
        '''
        Runs m-step of the gmm, to calculate responsibility/membership
for each data point
        :return: returns nothing
        '''
        for i in range(self.n_clusters):
            self.calculate_ric(i)

    def calculate_mu(self, gaussian_id):
        """
        Computes the mean for a gaussian and updates the mean value
        :param gaussian_id: gaussian_id
        :returns : returns nothing
        """
        ric = self.ric[:, gaussian_id]
        avg = np.average(self.input_data, axis=0, weights=ric)
        self.mu[gaussian_id] = avg

    def calculate_sigma(self, gaussian_id):
        '''
        Computes the co-variance for a gaussian and updates the
covariance matrix
        :param gaussian_id: gaussian_id
        :returns : returns nothing
        '''
        summ = np.zeros((self.input_data.shape[1],
self.input_data.shape[1]))
        summ1 = np.zeros((self.input_data.shape[1],
self.input_data.shape[1]))
        for i in range(self.input_data.shape[0]):
            data_temp =
self.input_data[i].reshape(self.input_data.shape[1], 1)
            mu_temp = self.mu[gaussian_id].reshape(self.mu.shape[1],
1)
            diff_temp = data_temp - mu_temp
            summ += self.ric[i, gaussian_id] * np.dot(diff_temp,
diff_temp.T)

        self.cov[gaussian_id] = summ / np.sum(self.ric[:,
gaussian_id])

    def calculate_ric(self, gaussian_id):
        '''
        Computes the ric for all data points for a gaussian_id and
updates the ric value
```

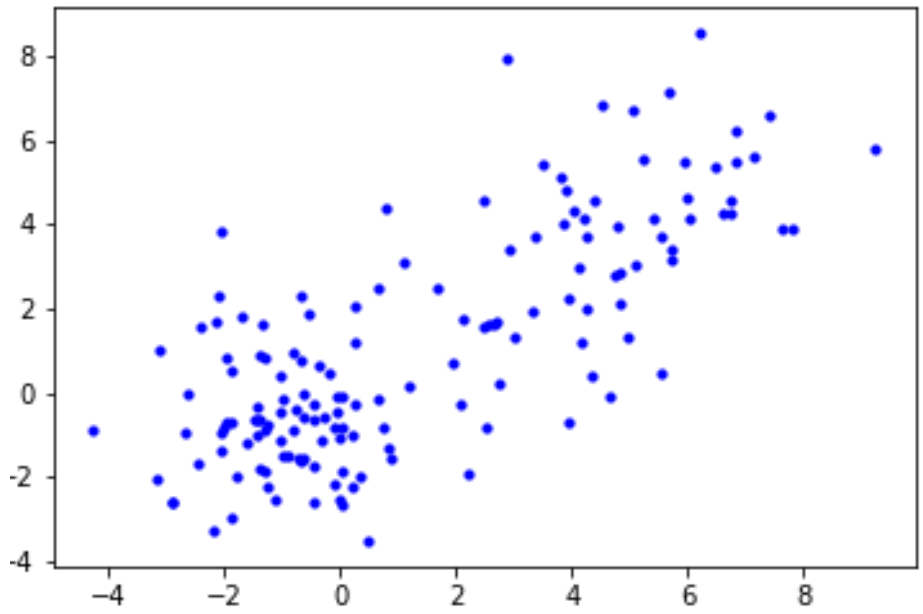
```
        :param gaussian_id: gaussian_id
        :returns : returns nothing
        '''
        for i in range(self.input_data.shape[0]):
            summ = 0
            for each in range(self.n_clusters):
                summ += self.pi[each] * self.probability_density(i,
each)

            self.ric[i][gaussian_id] = self.pi[gaussian_id] *
self.probability_density(i, gaussian_id) / summ
            # print(self.ric)

    def calculate_pi(self, gaussian_id):
        '''
        Computes the amplitude for a gaussian and updates the
amplitude
        :param gaussian_id: gaussian_id
        :returns : returns nothing
        '''
        self.pi[gaussian_id] = np.sum(self.ric[:, gaussian_id]) /
self.input_data.shape[0]

    def get_likelihood(self):
        '''
        Calculates the log likelihood
        :return : returns log likelihood
        '''
        new_likelihood = 0
        for i in range(self.input_data.shape[0]):
            temp = 0
            for k in range(self.n_clusters):
                temp += self.pi[k] * self.probability_density(i, k)
            new_likelihood += np.log(temp)

        return new_likelihood

    def execute(self):
        '''
        Runs the EM Algorith for gmm
        :return: returns nothing
        '''
        iterations = 0
        delta = float('inf')
        while delta > self.threshold and iterations <
self.max_iteration:
            self.e_step()
            self.m_step()
            lld = self.get_likelihood()
            delta = lld - self.likelihood
            self.likelihood = lld
            iterations += 1

    def predict(self, X):
        '''
        Returns predicted labels using Bayes Rule to
        :param X: data points
        :return: predicted cluster based on highest responsibility
gamma.
```

```
        '''
        labels = np.zeros(X.shape[0])

        for i in range(X.shape[0]):
            max_density = 0
            for c in range(self.n_clusters):
                density = self.pi[c] * self.probability_density(i, c)
                if max_density < density:
                    labels[i] = c
                    max_density = density

        return labels

    def plot(self):
        '''
        Plots the k-means cluster and write output to console
        :return: returns nothing
        '''
        print("---------GMM----------")
        print("Means")
        print(self.mu)
        print("Amplitudes")
        print(self.pi)
        print("Covariance")
        print(self.cov)

        centers = np.zeros((self.n_clusters, 2))
        predicted_values = self.predict(self.input_data)
        axes = plt.gca()
        for c in range(self.n_clusters):
            max_density = 0
            point = None
            for i in range(self.input_data.shape[0]):
                density = math.log(self.probability_density(i, c))
                if max_density < density:
                    max_density = density
                    point = self.input_data[i]
            centers[c, :] = point
        print("Centers")
        print(centers)
        color_iter = itertools.cycle(['navy', 'cornflowerblue',
'gold', 'darkorange'])
        colors = list("rgy")
        plt.scatter(self.input_data[:, 0], self.input_data[:, 1],
c=predicted_values, s=50, cmap='viridis', zorder=1)
        plt.scatter(centers[:, 0], centers[:, 1], c='black', s=300,
alpha=0.5, zorder=2)

        for i, (mean, covar, color) in enumerate(zip(
                self.mu, self.cov, color_iter)):
            v, w = np.linalg.eigh(covar)
            v = 2. * np.sqrt(2.) * np.sqrt(v)
            u = w[0] / np.linalg.norm(w[0])
            angle = np.arctan(u[1] / u[0])
            angle = 180. * angle / np.pi
            ell = patches.Ellipse(mean, v[0], v[1], 180. + angle,
color=color)
            ell.set_alpha(0.5)
            axes.add_artist(ell)
```

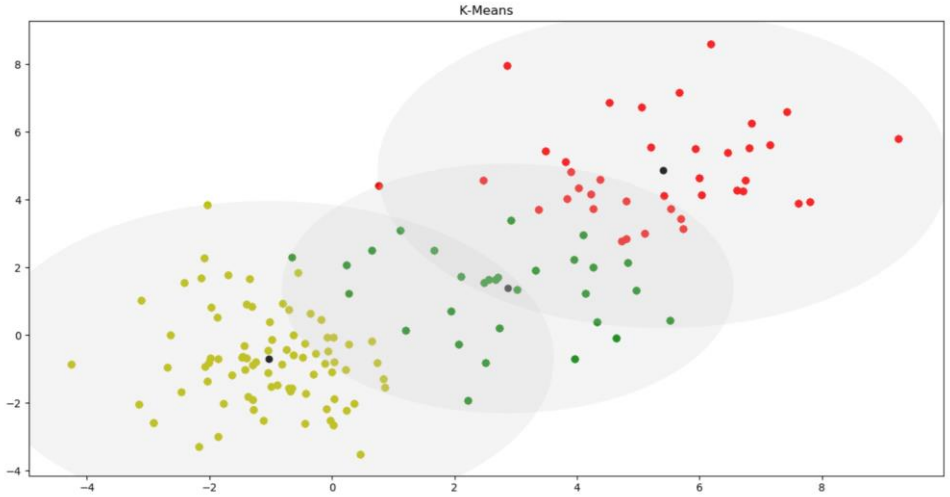
```
        plt.title("GMM")
        plt.show()


def main():
    '''
    Runner Program
    :return: returns nothing
    '''
    data = np.loadtxt('clusters.txt', delimiter=',')
    k_means_model = run_k_means(data, 10)
    k_means_model.plot()
    gmm_obj = gmm(data, 100, 3, 0.01)
    gmm_obj.execute()
    gmm_obj.plot()


def run_k_means(data, no_of_runs):
    '''
    Runs the k-means multiple times and gets the best model
    :param no of runs: #runs
    :return: returns best k-means model
    '''
    metrics = dict()
    for i in range(no_of_runs):
        k_obj = k_means(data, 3, 100, 0.01)
        k_obj.execute()
        metrics[k_obj] = k_obj.get_metric()
    key = min(metrics, key=metrics.get)
    print(metrics)
    return key


if __name__ == "__main__":
    main()
```

- **Run**

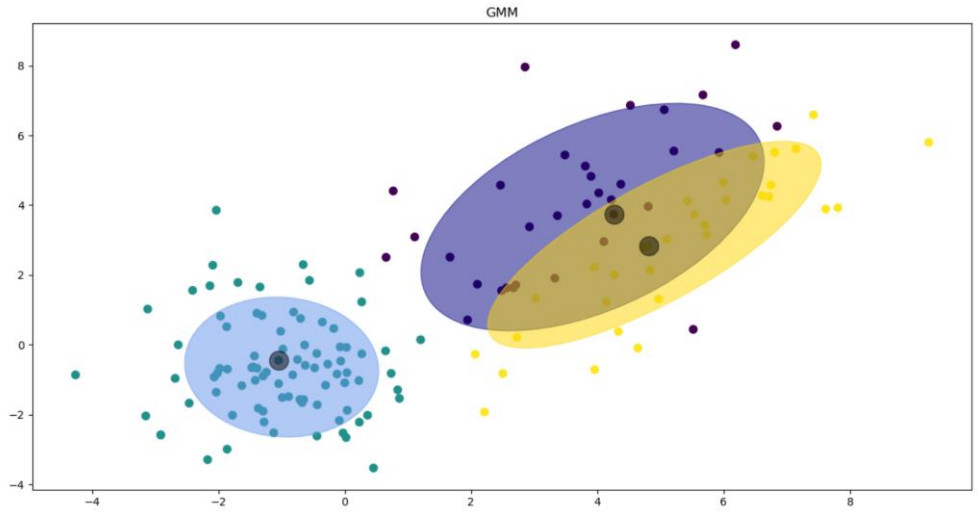
INITIAL DATASET:



KMEANS



GAUSSIAN MIXTURE MODEL:



- **Output**

```
------------K-Means------------
Centroid:
{0: array([-1.03469123, -0.69772229]), 1: array([2.87048536, 1.37938964]), 2: array([5.40705818, 4.87334962])}
---------GMM----------
Means
[[ 3.92448852  3.65922588]
 [-0.99723746 -0.63968754]
 [ 4.90240147  2.86791445]]
Amplitudes
[0.26051274 0.55747728 0.18200999]
Covariance
[[[ 3.68863697  2.14838316]
  [ 2.14838316  5.31177385]]

 [[ 1.17843763 -0.08665453]
  [-0.08665453  2.00748878]]

 [[ 3.48078259  3.02493953]
  [ 3.02493953  4.38613154]]]
Centers
[[ 4.25985513  3.73715647]
 [-1.04646474 -0.44770405]
 [ 4.8083377   2.83435798]]
```

- **Challenges Faced**
  1. Deciding the suitable data structures for the model.
  2. Visualizing GMM model using matplotlib.
  3. As all of us were coding different parts of algorithm so it initially become difficult for us to integrate each other's code.
  4. Evaluating the correctness of both the algorithms manually.

- **Code-level Optimizations**
  1. Used Numpy to perform linear algebraic operations efficiently.
  2. Improved k-means clusters by running the algorithm multiple times and then selecting the model using an evaluation function.
  3. Used log-likelihood to determine the convergence for GMM algorithm.

PART 2: SOFTWARE FAMILIARIZATION

Library Function:

Kmeans

class `sklearn.cluster.`**`KMeans`**(*n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001, precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=None, algorithm='auto'*)

GMM

class `sklearn.mixture.`**`GaussianMixture`**(*n_components=1, covariance_type='full', tol=0.001, reg_covar=1e-06, max_iter=100, n_init=1, init_params='kmeans', weights_init=None, means_init=None, precisions_init=None, random_state=None, warm_start=False, verbose=0, verbose_interval=1)*

Implementation:

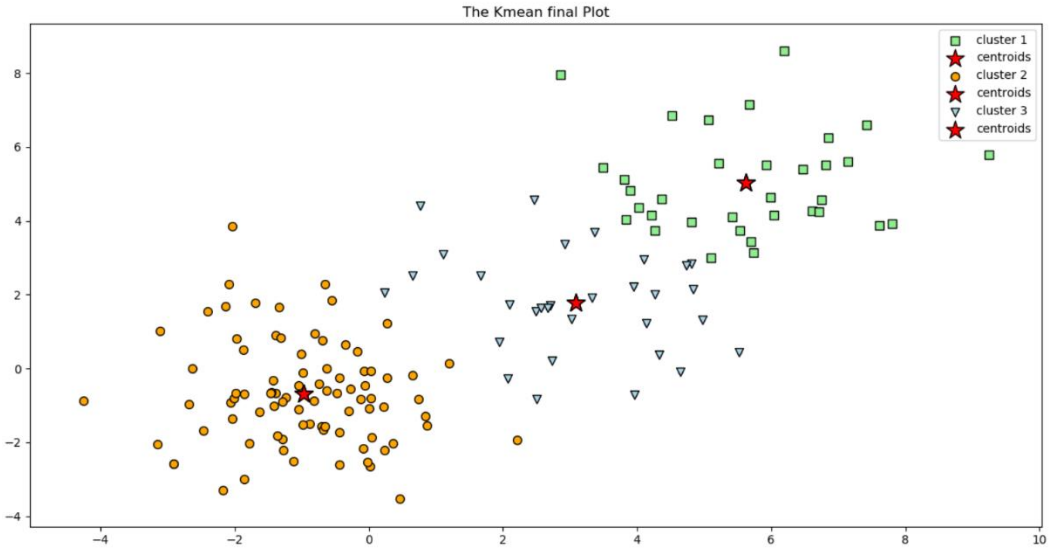
```
#!/usr/bin/env python
# coding: utf-8



import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from matplotlib.patches import Ellipse


file = open("clusters.txt")
data = []
data = file.readlines()
data = [i.replace("\n", "").split(",") for i in data]
data = [[float(i[0]), float(i[1])] for i in data]
data = np.array(data)
wcss = []

for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)
    kmeans.fit(data)
    # print(kmeans.inertia_)
    wcss.append(kmeans.inertia_)

plt.scatter(data[:, 0], data[:, 1], c='blue', marker='o', s=10)
plt.savefig('books_read.png')

row, col = data.shape

km = KMeans(
    n_clusters=3, init='random',
    n_init=10, max_iter=100,
    tol=1e-04, random_state=0
)
```

```
# n_init=number of random times to run kmean with different initial
centroid value
# tol=error_rate to stop
# init={kmeans++,random} kmeans ++ chooses initial state in smart way
# random chooses randomly


y_km = km.fit_predict(data)


def plot(data, n_clusters):
    marker = ['s', 'o', 'v']
    edgecolor = ['lightgreen', 'orange', 'lightblue']

    for i in range(n_clusters):
        plt.scatter(
            data[y_km == i, 0], data[y_km == i, 1],
            s=50, c=edgecolor[i],
            marker=marker[i], edgecolor='black',
            label='cluster ' + str(i + 1)
        )
        # plot the centroids
        plt.scatter(
            km.cluster_centers_[:, 0], km.cluster_centers_[:, 1],
            s=250, marker='*',
            c='red', edgecolor='black',
            label='centroids'
        )
        plt.title('The Kmean final Plot')

        plt.legend(scatterpoints=1)

    # plt.grid()
    plt.show()
    plt.savefig("kmean.png")

plot(data, 3)
print("========Kmean==========\n")
print(km.cluster_centers_)

gmm = GaussianMixture(n_components=3, max_iter=100,
covariance_type="full")

gmm.fit_predict(data)
print("Means")
print(gmm.means_)
print("\n")
print("Covariances")
print(gmm.covariances_)
print("\n")
print("Amplitudes")
print(gmm.weights_)
print("\n")


def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Draw an ellipse with a given position and covariance"""
    ax = ax or plt.gca()
```

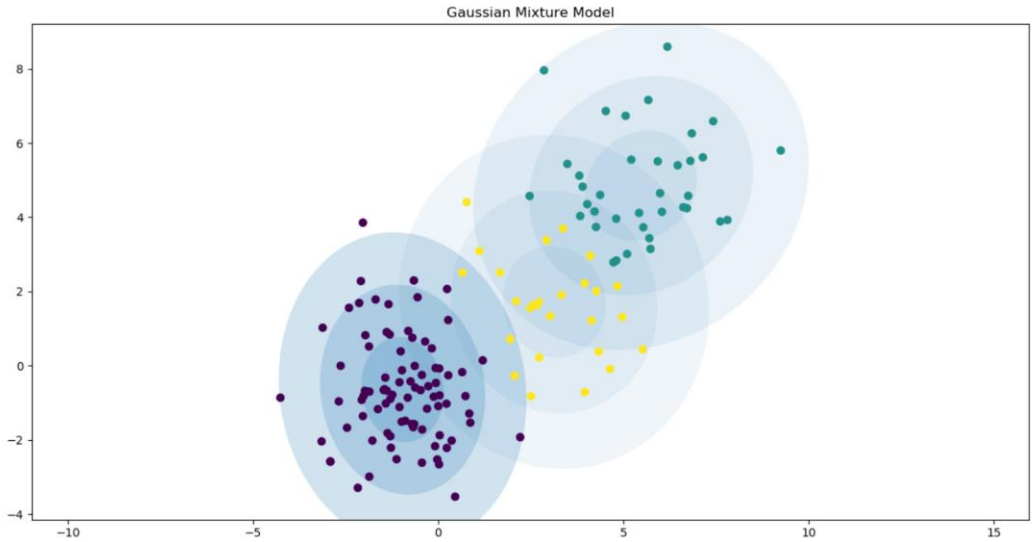
```
    # Convert covariance to principal axes
    if covariance.shape == (2, 2):
        U, s, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
        width, height = 2 * np.sqrt(s)
    else:
        angle = 0
        width, height = 2 * np.sqrt(covariance)

    # Draw the Ellipse
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, **kwargs))


def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis',
zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
    ax.axis('equal')

    w_factor = 0.2 / gmm.weights_.max()

    for pos, covar, w in zip(gmm.means_, gmm.covariances_,
gmm.weights_):
        draw_ellipse(pos, covar, alpha=w * w_factor)
    plt.title("Gaussian Mixture Model")
    plt.show()
    plt.savefig("gmm.png")


plot_gmm(gmm, data)
```

Output:

- KMEANS



- GAUSSIAN MIXTURE MODEL



- Program Output

```
========Kmean==========

Centroids
[[ 5.62016573  5.02622634]
 [-0.97476572 -0.68419304]
 [ 3.08318256  1.77621374]]


========Gaussian Mixture Model==========

Means
[[-0.97911637 -0.64069093]
 [ 5.45432037  4.83869236]
 [ 3.08530762  1.61520359]]


Covariances
[[1.20297373 2.02118177]
 [2.31416707 2.18322802]
 [1.98709636 2.31248971]]


Amplitudes
[0.56581904 0.24193577 0.19224518]
```

Comparison:

Our Implementation :

```
------------K-Means------------
Centroid:
{0: array([-1.03469123, -0.69772229]), 1: array([2.87048536, 1.37938964]), 2: array([5.40705818, 4.87334962])}
---------GMM----------
Means
[[ 3.92448852  3.65922588]
 [-0.99723746 -0.63968754]
 [ 4.90240147  2.86791445]]
Amplitudes
[0.26051274 0.55747728 0.18200999]
Covariance
[[[ 3.68863697  2.14838316]
  [ 2.14838316  5.31177385]]

 [[ 1.17843763 -0.08665453]
  [-0.08665453  2.00748878]]

 [[ 3.48078259  3.02493953]
  [ 3.02493953  4.38613154]]]
Centers
[[ 4.25985513  3.73715647]
 [-1.04646474 -0.44770405]
 [ 4.8083377   2.83435798]]
```

Library Implementation :

```
========Kmean==========

Centroids
[[ 5.62016573  5.02622634]
 [-0.97476572 -0.68419304]
 [ 3.08318256  1.77621374]]


========Gaussian Mixture Model==========

Means
[[-0.97911637 -0.64069093]
 [ 5.45432037  4.83869236]
 [ 3.08530762  1.61520359]]


Covariances
[[1.20297373 2.02118177]
 [2.31416707 2.18322802]
 [1.98709636 2.31248971]]


Amplitudes
[0.56581904 0.24193577 0.19224518]
```

Comments:

1. The library function provides a parameter to give random_state enabling the algorithm to reproduce the same model. In our case, it can be difficult to debug due to initial random setting.
2. The library implementation for GMM gives a functionality of using the last fit gaussian to be used again for training the gaussian helping in faster convergence. In our case, once we attain convergence, we stop and do not re-use.

PART 3: APPLICATIONS

KMEANS

Kmeans algorithm is very popular and used in a variety of applications such as market segmentation, document clustering, image segmentation and image compression, etc. The goal usually when we undergo a cluster analysis is either:

- Get a meaningful intuition of the structure of the data we're dealing with.
- Cluster-then-predict where different models will be built for different subgroups if we believe there is a wide variation in the behaviors of different subgroups. An example of that is clustering patients into different subgroups and build a model for each subgroup to predict the probability of the risk of having heart attack.
- Geyser eruptions segmentation.
- Image compression.

GAUSSIAN MIXTURE MODEL

- Gaussian mixture model has been widely applied in the fields of signal and information processing.
- We can use Gaussian mixture model (GMM) simulate arbitrary clustering graphics.
- Gaussian mixture models are used for database retrieval of texture and colour for image.
- Gaussian mixture speaker models are applied in Robust text-independent speaker identification.

PART 4: CONTRIBUTION

The project was planned and implemented by all group members.

1. We all discussed the design of the project.
2. The code was built in group together with discussion and peer reviews within the group.
3. Library function was studied by each member individually and collaborated to compare and analyse the difference between our results and library functions output of Kmeans & GMM algorithm.
4. Design of GMM and Kmeans class, structure of input data set, centroid, mean, covariance, membership function, tolerance variables and calculation of Probability Density function and covariance was all concluded after discussion