```python
import os
import shutil
import matplotlib.pyplot as plt
import numpy as np
import functools
import scipy.stats as sc
import sklearn.metrics as skl

MIN_X = -10
MAX_X = 10
MIN_Y = MIN_X
MAX_Y = MAX_X
LIMIT_ITER = 100


def generate_specific_means():
    n = 100
    c = 2

    means = np.array([[-3, 1.5], [-3, -1.5]])
    sigma = np.array([[6, 0], [0, 0.8]])
    x = np.random.rand(100, 2)

    x = np.matmul(x, sigma)
    x1 = x + means[0]
    x2 = x + means[1]

    data = np.vstack([x1, x2])
    data_real = data

    label_default = np.zeros((n*c, 1))
    data = np.append(data, label_default, axis=1)

    label_real = np.repeat(np.array(range(c)), repeats=n).reshape(-1, 1)
    data_real = np.append(data_real, label_real, axis=1)

    return data, data_real


def generate_data(n: int, c: int):
    """
    Generate two matrices of random samples. The dimensions of the matrices are (nc, d+1)
    :param n: The number of samples
    :param m: The number of generated cluster
    :return: data: data with empty label
             data_real: data with correct label
    """
    means = 12*(np.random.rand(c, 2) - 0.5)
    means = means.repeat(n, axis=0)

    data = means + np.random.randn(n*c, 2)
    data_real = data

    label_default = np.zeros((n*c, 1))
    data = np.append(data, label_default, axis=1)

    label_real = np.repeat(np.array(range(c)), repeats=n).reshape(-1, 1)
    data_real = np.append(data_real, label_real, axis=1)

    return data, data_real


def generate_means(k):
    """
    Generate a matrix of labeled random means. The dimension of the matrix is (k, d+1)
    :param k: The number of means
    :return: A matrix of labeled random means
    """
    means = 12*(np.random.rand(k, 2) - 0.5)
    label = np.array([np.array(range(k))]).T
    return np.append(means, label, axis=1)


def generate_sigmas(k):
    """
    Generate k identical matrices of sigma
```

```
    :param k: The number of gaussians
    :return: A list of matrices
    """
    return np.repeat(np.eye(2)[None, ...], 2, axis=0)


def generate_alphas(k):
    return np.ones(k)


def show_plot(data, mu=None, i=0, gaussian_list=None, alpha_list=None, save=False, show=True):

    plt.scatter(x=data[:, 0], y=data[:, 1], c=data[:, 2], s=5)
    if mu is not None:
        plt.scatter(x=mu[:, 0], y=mu[:, 1], c=mu[:, 2],
                    marker="*", edgecolors="black", s=100)
    if gaussian_list is not None:
        N = 200
        X = np.linspace(MIN_X, MAX_X, N)
        Y = np.linspace(MIN_Y, MAX_Y, N)
        Z = draw_gaussian(X, Y, gaussian_list, alpha_list)
        plt.contour(X, Y, Z)

    plt.xlim([MIN_X, MAX_X])
    plt.ylim([MIN_Y, MAX_Y])

    if save:
        if not os.path.exists('./output'):
            os.mkdir('./output')
        plt.savefig(f'./output/{i}')
        plt.close()
    else:
        if show == True:
            plt.show()

    if gaussian_list is not None:
        return Z


def duplicate_mu(data, mu):
    n_data = data.shape[0]
    n_mu = mu.shape[0]

    # Duplicate the values of mu along the vertical axis
    mu_v = np.apply_along_axis(functools.partial(
        np.repeat, repeats=n_data, axis=0), axis=0, arr=mu[:, :2])

    # Split the array into multiple sub-arrays vertically
    return np.vsplit(mu_v, n_mu)


def assignment(data, mu):

    mu_duplicated = duplicate_mu(data, mu)
    x = data[:, :2]
    result = np.zeros((1, data.shape[0]))

    for i in range(mu.shape[0]):
        diff = x-mu_duplicated[i]
        result = np.vstack([result, np.apply_along_axis(
            lambda a: a[0]**2+a[1]**2, 1, diff)])
    result = result[1:]
    arg_min = np.argmin(result, 0)

    data[:, 2] = arg_min
    cost = np.sum(np.min(result, 0))

    return data, cost


def helper_sum(data, k):
    result = np.zeros((k, 2))
    result[int(data[2])] = [data[0], data[1]]
    return result


def get_linked_data(data, k):

```

```python
    sum_data_linked = np.zeros((1, k))
    for row in data:
        sum_data_linked[0, int(row[2])] += 1

    return sum_data_linked


def centroid_update(data, k):
    linked_data = get_linked_data(data, k)
    new_mu = np.apply_along_axis(functools.partial(helper_sum, k=k), 1, data)

    new_mu = np.sum(new_mu, axis=0)
    new_mu = np.divide(new_mu, linked_data.repeat(2, 0).T, out=np.zeros_like(
        new_mu), where=linked_data.repeat(2, 0).T != 0)
    label = np.array([np.array(range(k))]).T
    new_mu = np.append(new_mu, label, axis=1)

    return new_mu


def cluster(data, mu, save):

    total_cost = np.array([])

    for i in range(16):

        data, cost = assignment(data, mu)
        total_cost = np.append(total_cost, cost)

        new_mu = centroid_update(data, k)
        if np.array_equal(new_mu, mu):
            break
        mu = new_mu
        i += 1
        show_plot(data, mu, i, save=save, show=False)

    return data, mu, total_cost


def draw_gaussian(X, Y, gaussian_list:
list[sc._multivariate.multivariate_normal_frozen], alpha_list):

    X, Y = np.meshgrid(X, Y)
    pos = np.dstack((X, Y))
    Z_tot = np.zeros(X.shape)

    for i in range(len(gaussian_list)):
        Z_tot += alpha_list[i]*gaussian_list[i].pdf(pos)
    return Z_tot


def generate_list_gaussians(means, sigma):

    gaussian_list = []
    sigma = np.array([[1, 0], [0, 1]])

    for mu in means:
        gaussian_list.append(sc.multivariate_normal(mu, sigma))

    return gaussian_list


def expectation_step(gaussian: list[sc._multivariate.multivariate_normal_frozen],
data, k, alpha_list):
    x = data[:, :2]

    # Sum for the denominator
    divisor = 0
    for l in range(k):
        divisor += gaussian[l].pdf(x)*alpha_list[l]

    w_list = []
    for j in range(k):
        w = gaussian[j].pdf(x)*alpha_list[j] / divisor
        w = w.reshape((w.shape[0], 1))
        w_list.append(w)

    return w_list

```

```python

def minimalization_step(w_list, data):
    x = data[:, :2]
    gaussian_list = []
    alpha_list = []

    for w in w_list:
        n = w.shape[0]

        alpha = 1/n * np.sum(w)
        mu = sum(w*x)/np.sum(w)
        sigma = np.matmul((x-mu).T, w*(x-mu))/np.sum(w)

        gaussian_list.append(sc.multivariate_normal(
            mu, sigma, allow_singular=True))
        # TODO numpy.linalg.LinAlgError: When `allow_singular is False`, the input matrix must be symmetric positive definite

        alpha_list.append(alpha)

    return gaussian_list, alpha_list


def gmm(gaussian_list, data, k, alpha_list):

    # Show the data with the initialized gaussians
    Z = show_plot(data, mu, -1, save=False,
                  gaussian_list=gaussian_list, alpha_list=alpha_list)

    # Run the Gaussian Mixture Method
    for i in range(LIMIT_ITER):
        w_list = expectation_step(gaussian_list, data, k, alpha_list)
        gaussian_list, alpha_list = minimalization_step(w_list, data)

        new_Z = show_plot(data, mu, i, save=True,
                          gaussian_list=gaussian_list, alpha_list=alpha_list)
        print(f'Iteration {i}', end="\r")
        if np.allclose(Z, new_Z, rtol=1e-4, atol=1e-4):
            break

        Z = new_Z

    print("")
    w_as_array = np.array(w_list).squeeze()
    return w_as_array


def get_label(w_array):
    labels = np.argmax(w_array, axis=0)
    return labels.reshape(labels.shape[0])


if __name__ == "__main__":

    k = 2  # The number of clusters we'd like to find
    n = 100  # The size of the data
    c = k  # The number of generated cluster (we should define c=k)

    if os.path.exists('./output'):
        shutil.rmtree('./output')

    data, data_real = generate_data(n, c)

    # If you want to generate specific data, uncomment this
    # data, data_real = generate_specific_means()

    # Show only the data
    show_plot(data_real, i=-2, save=False)

    # Generate the data for the gmm algorithm
    mu = generate_means(k)
    sigma = generate_sigmas(k)
    alpha_list = generate_alphas(k)
    gaussian_list = generate_list_gaussians(mu[:, :2], sigma)

    # Run the gmm algorithm
    w_array = gmm(gaussian_list, data, k, alpha_list)
    labels = get_label(w_array)

```

```
    data_parsed = data
    data_parsed[:, 2] = labels
    show_plot(data_parsed, i=-2, save=False)

    score_gmm = skl.adjusted_rand_score(data_real[:, 2], labels)

    data, mu, total_cost = cluster(data, mu, save=False)

    score_kmeans = skl.adjusted_rand_score(data_real[:, 2], data[:, 2])

    print("gmm score: ", score_gmm)
    print("kmeans score: ", score_kmeans)

    print("eof")
```