```python
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import os

#This function generate random mean and covariance
def gauss_params_gen(num_clusters, num_dims, factor):
    mu = np.random.randn(num_clusters,num_dims)*factor
    sigma = np.random.randn(num_clusters,num_dims,num_dims)
    for k in range(num_clusters):
        sigma[k] = np.dot(sigma[k],sigma[k].T)

    return (mu, sigma)

#Given mean and covariance generate data
def data_gen(mu, sigma, num_clusters, num_samples):
    labels = []
    X = []
    cluster_prob = np.array([np.random.rand() for k in
     range(num_clusters)])
    cluster_num_samples = (num_samples * cluster_prob /
     sum(cluster_prob)).astype(int)
    cluster_num_samples[-1] = num_samples-sum(cluster_num_samples[:-1])

    for k, ks in enumerate(cluster_num_samples):
        labels.append([k]*ks)
        X.append(np.random.multivariate_normal(mu[k], sigma[k], ks))

    # shuffle data
    randomize = np.arange(num_samples)
    np.random.shuffle(randomize)
    X =  np.vstack(X)[randomize]
    labels =  np.array(sum(labels,[]))[randomize]

    return X, labels


def data2D_plot(ax, x, labels, centers, cmap, title):
    data = {'x0': x[:,0], 'x1': x[:,1], 'label': labels}
    ax.scatter(data['x0'], data['x1'], c=data['label'], cmap=cmap, s=20,
     alpha=0.3)
    ax.scatter(centers[:, 0], centers[:, 1],
     c=np.arange(np.shape(centers)[0]), cmap=cmap, s=50, alpha=1)
    ax.scatter(centers[:, 0], centers[:, 1], c='black', cmap=cmap, s=20,
     alpha=1)
    ax.title.set_text(title)

def plot_init_means(x, mus, algs, fname):
    import matplotlib.cm as cm
    fig = plt.figure()
    plt.scatter(x[:,0], x[:,1], c='gray', cmap='viridis', s=20, alpha= 0.4,
     label='data')
    for mu, alg, clr in zip(mus, algs, cm.viridis(np.linspace(0, 1,
     len(mus)))):
        plt.scatter(mu[:,0], mu[:, 1], c=clr, s=50, label=alg)
        plt.scatter(mu[:, 0], mu[:, 1], c='black', s=10, alpha=1)
    legend = plt.legend(loc='upper right', fontsize='small')
    plt.title('Initial guesses for centroids')
    fig.savefig(fname)

def loss_plot(loss, title, xlabel, ylabel, fname):
    fig = plt.figure(figsize = (13, 6))
    plt.plot(np.array(loss))
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    fig.savefig(fname)


def gaussian_pdf (X, mu, sigma):
    # Gaussian probability density function
    return np.linalg.det(sigma) ** -.5 ** (2 * np.pi) ** (-X.shape[1]/2.) \
                * np.exp(-.5 * np.einsum('ij, ij -> i',\
                X - mu, np.dot(np.linalg.inv(sigma) , (X - mu).T).T ) )

def EM_initial_guess (num_dims, data, num_samples, num_clusters):
    # randomly choose the starting centroids/means
    # as num_clusters of the points from datasets
    mu = data[np.random.choice(num_samples, num_clusters, False), :]

    # initialize the covariance matrice for each gaussian
    sigma = [np.eye(num_dims)] * num_clusters

    # initialize the probabilities/weights for each gaussian
    # begin with equal weight for each gaussian
    alpha = [1./num_clusters] * num_clusters

    return mu, sigma, alpha

def EM_E_step (num_clusters, num_samples, data, mu, sigma, alpha):
    ## Vectorized implementation of e-step equation to calculate the
    ## membership for each of k -gaussians
    Q = np.zeros((num_samples, num_clusters))
    for k in range(num_clusters):
        Q[:, k] = alpha[k] * gaussian_pdf(data, mu[k], sigma[k])

    ## Normalize so that the responsibility matrix is row stochastic
    Q = (Q.T / np.sum(Q, axis = 1)).T

    return Q

def EM_M_step (num_clusters, num_dims, num_samples, Q, data):

    # M Step
    ## calculate the new mean and covariance for each gaussian by
    ## utilizing the new responsibilities
    mu      = np.zeros((num_clusters, num_dims))
    sigma   = np.zeros((num_clusters, num_dims, num_dims))
    alpha = np.zeros(num_clusters)
    ## The number of datapoints belonging to each gaussian
    num_samples_per_cluster = np.sum(Q, axis = 0)

    for k in range(num_clusters):
        ## means
        mu[k] = 1. / num_samples_per_cluster[k] * np.sum(Q[:, k] * data.T,
         axis = 1).T
        centered_data = np.matrix(data - mu[k])

        ## covariances
        sigma[k] = np.array(1. / num_samples_per_cluster[k] *
         np.dot(np.multiply(centered_data.T,  Q[:, k]), centered_data))

        ## and finally the probabilities
        alpha[k] = 1. / (num_clusters*num_samples) *
         num_samples_per_cluster[k]

    return mu, sigma, alpha

def EM_log_likelihood_calc (num_clusters, num_samples, data, mu, sigma,
 alpha):
    L = np.zeros((num_samples, num_clusters))
    for k in range(num_clusters):
        L[:, k] = alpha[k] * gaussian_pdf(data, mu[k], sigma[k])
    return np.sum(np.log(np.sum(L, axis = 1)))


def EM_calc (num_dims, num_samples, num_clusters, x):
    log_likelihoods = []
    labels          = []
    iter_cnt        = 0
    epsilon         = 0.0001
    max_iters       = 200
    update          = 2*epsilon

    # initial guess
    mu, sigma, alpha = EM_initial_guess(num_dims, x, num_samples,
     num_clusters)
    mus = [mu]
    sigmas = [sigma]
    alphas = [alpha]
    while (update > epsilon) and (iter_cnt < max_iters):
        iter_cnt += 1

        # E - Step
        Q = EM_E_step (num_clusters, num_samples, x, mu, sigma, alpha)

        # M - Step
        mu, sigma, alpha = EM_M_step (num_clusters, num_dims, num_samples,
         Q, x)

        mus.append(mu)
        sigmas.append(sigma)
        alphas.append(alpha)
        # Likelihood computation
        log_likelihoods.append(EM_log_likelihood_calc(num_clusters,
         num_samples, x, mu, sigma, alpha))

        # check convergence
        if iter_cnt >= 2 :
            update = np.abs(log_likelihoods[-1] - log_likelihoods[-2])

        # logging
        print("iteration {}, update {}".format(iter_cnt, update))

        # print current iteration
        labels.append(np.argmax(Q, axis = 1))

    return labels, log_likelihoods, {'mu': mus, 'sigma': sigmas, 'alpha':
     alphas}

def kmeans_initial_guess (data, num_samples, num_clusters):
    # randomly choose the starting centroids/means
    # as num_clusters of the points from datasets
    mu = data[np.random.choice(num_samples, num_clusters, False), :]
    return mu

def kmeans_get_labels(num_clusters, num_samples, num_dims, data, mu):
    # set all dataset points to the best cluster according to minimal
     distance
    #from centroid of each cluster
    dist = np.zeros((num_clusters, num_samples))
    for k in range(num_clusters):
        dist[k] = np.linalg.norm(data - mu[k], axis=1)

    labels = np.argmin(dist, axis=0)

    return labels

def kmeans_get_means(num_clusters, num_dims, data, labels):
    # Compute the new means given the reclustering of the data
    mu = np.zeros((num_clusters, num_dims))
    for k in range(num_clusters):
        idx_list = np.where(labels == k)[0]
        if (len(idx_list) == 0):
            r = np.random.randint(len(data))
            mu[k] = data[r,:]
        else:
            mu[k] = np.mean(data[idx_list], axis=0)
    return mu

def kmeans_calc_loss(num_clusters, num_samples, data, mu, labels):
    dist = np.zeros((num_samples, num_clusters))
    for j in range(num_samples):
        for k in range(num_clusters):
            if (labels[j] == k) :
                dist[j,k] = np.linalg.norm(data[j] - mu[k])
    return sum(sum(dist))
```

```python
def k_means_calc (num_dims, num_samples, num_clusters, x):
    loss            = []
    labels          = []
    iter_cnt        = 0
    epsilon         = 0.00001
    max_iters       = 100
    update          = 2*epsilon

    # initial guess
    mu = [kmeans_initial_guess(x, num_samples, num_clusters)]

    while (update > epsilon) and (iter_cnt < max_iters):
        iter_cnt += 1
        # Assign labels to each datapoint based on centroid
        labels.append(kmeans_get_labels(num_clusters, num_samples,
         num_dims, x, mu[-1]))

        # Assign centroid based on labels
        mu.append(kmeans_get_means(num_clusters, num_dims, x, labels[-1]))
        # check convergence
        if iter_cnt >= 2 :
            update = np.linalg.norm(mu[-1] - mu[-2], None)

        # Print distance to centroids vs iteration
        loss.append(kmeans_calc_loss(num_clusters, num_samples, x, mu[-1],
         labels[-1]))

        # logging
        print("iteration {}, update {}".format(iter_cnt, update))


    return labels, loss, mu

def k_qda_initial_guess (num_dims, data, num_samples, num_clusters):
    # randomly choose the starting centroids/means
    # as num_clusters of the points from datasets
    mu = data[np.random.choice(num_samples, num_clusters, False), :]

    # initialize the covariance matrice for each gaussian
    sigma = [np.eye(num_dims)] * num_clusters

    return mu, sigma

def k_qda_get_parms(num_clusters, num_dims, data, labels):
    ## calculate the new mean and covariance for each gaussian
    mu      = np.zeros((num_clusters, num_dims))
    sigma   = np.zeros((num_clusters, num_dims, num_dims))

    for k in range(num_clusters):
        c_k = labels==k
        if (len(data[c_k]) == 0):
            r = np.random.randint(len(data))
            mu[k] = data[r,:]
        else:
            mu[k] = np.mean(data[c_k], axis=0)

        if (len(data[c_k]) > 1):
            centered_data = np.matrix(data[c_k] - mu[k])
            sigma[k] = np.array(1. / len(data[c_k]) *
             np.dot(centered_data.T, centered_data))
        else:
            sigma[k] = np.eye(num_dims)

    return mu, sigma

def k_qda_get_labels(num_clusters, num_samples, mu, sigma, data):
    # set all dataset points to the best cluster according to best
    # probability given calculated means and covariances
    dist = np.zeros((num_clusters, num_samples))
    for k in range(num_clusters):
        data_center = (data - mu[k])
        dist[k] = np.einsum('ij, ij -> i', data_center,
         np.dot(np.linalg.inv(sigma[k]) , data_center.T).T )
        labels = np.argmin(dist, axis=0)
    return labels


def k_qda_calc(num_dims, num_samples, num_clusters, x):
    loss            = []
    labels          = []
    iter_cnt        = 0
    epsilon         = 0.00001
    max_iters       = 100
    update          = 2*epsilon

    # initial guess
    mu, sigma = k_qda_initial_guess(num_dims, x, num_samples, num_clusters)
    mus = [mu]
    sigmas = [sigma]
    while (update > epsilon) and (iter_cnt < max_iters):
        iter_cnt += 1

       # Assign labels to each datapoint based on probability
        labels.append(k_qda_get_labels(num_clusters, num_samples, mus[-1],
         sigmas[-1], x))

       # Assign centroid and covarince based on labels
        mu, sigma = k_qda_get_parms(num_clusters, num_dims, x, labels[-1])

        mus.append(mu)
        sigmas.append(sigma)

       # check convergence
        if iter_cnt >= 2 :
            update = np.linalg.norm(mus[-1] - mus[-2], None)
            update += np.linalg.norm(sigmas[-1] - sigmas[-2], None)

       # logging
        print("iteration {}, update {}".format(iter_cnt, update))
    return labels, {'mu': mus, 'sigma': sigmas}


def experiments(seed, factor, dir='plots', num_samples=500, num_clusters=3):

    if not os.path.exists(dir):
        os.makedirs(dir)

    np.random.seed(seed)
    num_dims     = 2

    # generate data samples
    (mu, sigma) = gauss_params_gen(num_clusters, num_dims, factor)
    x, true_labels   = data_gen(mu, sigma, num_clusters, num_samples)
    #### Expectation-Maximization
    EM_labels, log_likelihoods, EM_parms = EM_calc (num_dims, num_samples,
     num_clusters, x)
    #### K QDA
    kqda_labels, kqda_parms = k_qda_calc(num_dims, num_samples,
     num_clusters, x)
    #### K means
    kmeans_labels, loss, kmean_mus = k_means_calc (num_dims, num_samples,
     num_clusters, x)

    #Collect all results
    labels = [true_labels, EM_labels[-1], kqda_labels[-1],
     kmeans_labels[-1]]
    mus_fin = np.array([mu, EM_parms['mu'][-1], kqda_parms['mu'][-1],
     kmean_mus[-1]])
    algs = np.array(['True', 'EM', 'KQDA', 'Kmeans'])

    #### Plot
    fig = plt.figure()
    fig.subplots_adjust(hspace=0.4, wspace=0.4)
    for i, (lbl,alg, mu) in enumerate(zip(labels, algs, mus_fin)):
        ax = fig.add_subplot(2, 2, i+1)
        data2D_plot(ax, x, lbl, mu, 'viridis', alg)

    fname = os.path.join(dir,
     'Results_s{}_f{}_n{}_k{}.png'.format(seed,factor,num_samples,
     num_clusters))
    fig.savefig(fname)

    mus_init = np.array([mu, EM_parms['mu'][0], kqda_parms['mu'][0],
     kmean_mus[0]])
    init_mu_fname = os.path.join(dir,
     'init_mu_s{}_f{}_n{}_k{}.png'.format(seed,factor, num_samples,
     num_clusters))
    plot_init_means(x, mus_init, algs, init_mu_fname)

if __name__ == "__main__":
    experiments(seed=63, factor=10, num_samples=500, num_clusters=3)
```