```
''' Import Libraries'''
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

class Classifier:
    ''' This is a class prototype for any classifier. It contains two empty methods:
predict, fit'''
    def __init__(self, data):
        self.model_params = {'data': None}
        self.model_params['data'] = data

    def predict(self, x):
        '''This method takes in x (numpy array) and returns a prediction y'''
        raise NotImplementedError

    def fit(self, dataframe):
        '''This method is used for fitting a model to data: x, y'''
        raise NotImplementedError



class KMeans_custom(Classifier):
    '''No init function, as we inherit it from the base class'''
    def __init__(self, data):
        super().__init__(data)

    def fit(self, k=2, tol = 0.01):
        '''k is the number of clusters, tol is our tolerance level'''
        '''Randomly choose k vectors from our data'''
        '''Your code here'''
        data = self.model_params['data']

        centroids = []

        min_x, max_x = np.min(data[:, 0]), np.max(data[:, 0])
        min_y, max_y = np.max(data[:, 1]), np.max(data[:, 1])

        # print(min_x, min_y)
        # print(max_x, max_y)

        for i in range(k):
            random_centroid = np.array([np.random.uniform(min_x, max_x),
np.random.uniform(min_y, max_y)])
            centroids.append(random_centroid)

        i = 0
        run = True
        list_of_centroids = []
```

```
        list_of_centroids.append(centroids)

        while(run):
            clusters = self.cluster_assign(data, k, centroids)
            new_centroids_val = self.new_cluster(data, k, clusters)

            sse_old = self.sse_binary(data, centroids, clusters)
            sse_new = self.sse_binary(data, new_centroids_val, clusters)

            if i > 1:
                if np.absolute(sse_new - sse_old)/sse_old <= tol:
                    run = False
                else:
                    centroids = new_centroids_val
                    list_of_centroids.append(centroids)


            i = i + 1
        self.centroids = centroids
        return list_of_centroids

    def sse_binary(self, data, centroids, clusters):
        errors = []
        for i in range(data.shape[0]):
            if(clusters[i] == 0):
                errors.append(np.linalg.norm(data[i] - centroids[0]))
            else:
                errors.append(np.linalg.norm(data[i] - centroids[1]))

        sse = sum(errors)
        return sse

    def sse(self, data, k, centroids, square=True):
        data_dists = []
        for centroid in centroids:
            data_dists.append(data - centroid)

        min_dists = []
        for data_dist in data_dists:
            min_dist = np.sum(np.square(data_dist), axis = 1)
            if square:
                min_dist = np.sqrt(min_dist)
            min_dists.append(min_dist)
        # print(min_dists)

        return min_dists

    def cluster_assign(self, data, k, centroids):
        min_dists = self.sse(data, k, centroids)
```

```
        stacked_min_dists = min_dists[0]
        for min_dist in min_dists[1:]:
            stacked_min_dists = np.column_stack((stacked_min_dists, min_dist))

        clusters = np.argmin(stacked_min_dists, axis=1)

        return clusters

    def new_cluster(self, data, k, clusters):
        new_centroids = []
        for i in range(k):
            pt_cluster = []
            for x in range(len(data)):
                if clusters[x] == i:
                    pt_cluster.append(data[x])
            # mean_c = np.mean(pt_cluster, axis=0)
            # new_centroids.append(mean_c)
            stacked_pt_cluster = np.stack(pt_cluster)
            #0 element for x and 1 element for y
            mean_x = np.mean(stacked_pt_cluster[:, 0])
            mean_y = np.mean(stacked_pt_cluster[:, 1])
            new_centroids.append(np.array([mean_x, mean_y]))

        return new_centroids

    def predict(self, x):
        '''Input: a vector (x) to classify
           Output: an integer (classification) corresponding to the closest cluster
           Idea: you measure the distance (calc_distance) of the input to
           each cluster centroid and return the closest cluster index'''
        '''Your code here'''
        euclidian = []
        for centroid in self.centroids:
            euclidian.append(self.calc_distance(x, centroid))

        predicted_cluster = np.argmin(np.array(euclidian))

        return predicted_cluster

    def calc_distance(self, point1, point2):
        '''Your code here'''
        '''Input: two vectors (point1 and point2)
           Output: a single value corresponding to the euclidan distance betwee the
two vectors'''
        '''Your code here'''


        return np.linalg.norm(point1 - point2)
```