```python
import numpy as np
import matplotlib.pyplot as plt
from download_data import download_data
from sklearn.cluster import KMeans
import random

MAX_ITERATIONS = 50;
MIN_SAMPLE_CHANGE = 0


colors=['red', 'blue', 'black', 'brown', 'c', 'm', 'y', 'k', 'w', 'orange']


def run_kmeans_clustering(k_value=2, showPlots=False, data=None):
    """  Calculate K means, given K  """
    # pick "samples" random x,y coordinates from 0 to "samples"

    # number of columns of data
    samples_size = len(data) #329
    data_dimensions = len(data[0]) #9
    xy_samples = data #329*9 all data points

    def indexes_to_list_array(indexes):
        """  convert from indexes to an array of the x-y coordinates at each index   """
        items = []
        for index in indexes:
            items.append(xy_samples[index, :])
        return np.array(items)
    #returned a np.array type of indexed points' coordinates in all samples

    # initialize dictionary
    grouped_coordinates_indexes = initialize_dict(k_value) #a dictionary with k types

    current_iteration = 0

    # main iterations
    while(current_iteration < MAX_ITERATIONS):

        if current_iteration == 0:
            #  pick 2 random sets of x,y coordinates to be centroids to start off
            #10*9 np.array this is random generated centroid points, performance is really bad!!!
            #centroid_points = np.random.randint(samples_size, size=(k_value, data_dimensions))

            #below is randomly picked centroid points, better performance
            centroid_points = []
            random_samples = random.sample(range(samples_size),k_value)
            for index in random_samples:
                centroid_points.append(xy_samples[index, :])
            centroid_points = np.array(centroid_points)
        else:
            #find centroids based on the current memberships
            #                                     9            329*9            dict
            centroid_points = _get_centroids(data_dimensions, xy_samples, grouped_coordinates_indexes)


        #  with new centroids, calculate distances and assign points to new groups
        new_grouped_coordinates_indexes = assign_points_to_groups(k_value, xy_samples, centroid_points)


        max_changed = check_minimum_changes_met(k_value, current_iteration, grouped_coordinates_indexes, new_grouped_coordinates_indexes)

        if max_changed >= 0:
            # already shown, but show if "showPlots" if false
            print("Found due to {} minimum changes".format(max_changed))
            break;

        # reassign new index group to existing
        grouped_coordinates_indexes = new_grouped_coordinates_indexes.copy()

        current_iteration += 1

    print("Found after {} iterations".format(current_iteration + 1))

    result_arrays = []

    for i in range(k_value):
        # return list of list of <centroid, grouped points>
        result_arrays.append([centroid_points[i], indexes_to_list_array(grouped_coordinates_indexes[i])])

    return result_arrays


def initialize_dict(k_value):
    dict={}
    for i in range(k_value):
```

```
            dict[i] = []
    return dict

def assign_points_to_groups(k_value, xy_samples, centroid_points):
    """
    assign each sample to the centroid it is closest to
    returns:
        dictionary of centroid (index) mapped to grouping of samples (indexes) that are closest
    """
    grouped_sample_indexes = initialize_dict(k_value)

    for index in range(len(xy_samples)):
        current_xy_samples = xy_samples[index, :]
        current_xy_samples = np.array(current_xy_samples)
        distances_to_centroid = []
        # calculate distance of x/y coordinate from center
        for centroid_index in range(len(centroid_points)):
            distances_to_centroid.append(calc_euclidean_dist_vector(centroid_points[centroid_index, :], current_xy_samples))

        minimum_index = distances_to_centroid.index(min(distances_to_centroid))
        grouped_sample_indexes[minimum_index].append(index)
    return grouped_sample_indexes


def calc_euclidean_dist_vector(vector1, vector2):
    #################placeholder # stanrt ###################
    result = np.linalg.norm(vector1-vector2)
    #################placeholder # end ###################
    return result

#                                 10          x
def check_minimum_changes_met(k_value, current_iteration, old_grouped_samples, new_grouped_samples):
    if current_iteration > 0:
        # check to see if points within groups changed or not
        unchanged_coordinates = []
        for i in range(k_value):
            original_group_length = len(old_grouped_samples[i])
            unchanged_group_coordinates = set(new_grouped_samples[i]).intersection(old_grouped_samples[i])
            unchanged_coordinates.append(abs(original_group_length - len(unchanged_group_coordinates)))
        # find array that has the most number of samples that have changed

        max_changed_index = unchanged_coordinates.index(max(unchanged_coordinates))
        max_changed = unchanged_coordinates[max_changed_index]
        if max_changed <= MIN_SAMPLE_CHANGE:
            return max_changed
    return -1


def _get_centroids(dimensions, xy_coordinates, groups):
    xy_centroids = []
    for i in range(len(groups)):
        xy_centroids.append(xy_coordinates[groups[i], :])
    centroid_points = get_centroids(dimensions, xy_centroids)
    return centroid_points


def get_centroids(dimensions, xy_groups):
    """ Takes tuple of coordinates
        returns:
        2,2 array of centroid points
    """

    def get_point_mean(array_values,dimensions):
        """ take care of issue of empty list
        """
        if len(array_values) == 0:
            return np.zeros(dimensions)
        return array_values.mean(axis = 0)


    length = len(xy_groups) #10
    centroid_points = np.zeros((length, dimensions)) #10*9
    #  for each group, get the average point
    #################placeholder # start ###################
    for i in range(length):
        centroid_points[i, :] = get_point_mean(xy_groups[i],dimensions)
    #################placeholder # end ###################
    return centroid_points

def get_sum_of_squares(dimensions, center, samples):
    """  Get the sum of squared error, given centroid and all of its grouped points  """
    #################placeholder # start ###################
    length = len(samples)
    sse = 0
    for i in range(length):
        sse = sse + np.linalg.norm(center - samples[i])**2
```

```
    #################placeholder # end ###################
    return sse


if __name__ == "__main__":
#  load data
    data = download_data("cities_life_ratings.csv").values

    dimensions = len(data[0])

#  evaluating Ks

    k_values = [3, 6, 8, 10] # if go higher than 10, need to add to "colors" list
    k_errors = []
    for k in k_values:
        result_arrays = run_kmeans_clustering(k, showPlots=False, data=data)
        # step 6: calculate the sum of squared errors (SSE)
        sse_total = 0
        for i in range(k):
            center = result_arrays[i][0]
            samples = result_arrays[i][1]
            sse_total += get_sum_of_squares(dimensions, center, samples)
        k_errors += [sse_total]

    plt.plot(k_values, k_errors)
    plt.title("K-Means")
    plt.xlabel("K values")
    plt.ylabel("errors")
    plt.show()

    for i in range(len(k_values)):
        print("K = {})".format(k_values[i]))
        print("SSE = {})".format(k_errors[i]))
```