Leonardo Gomez
CS596

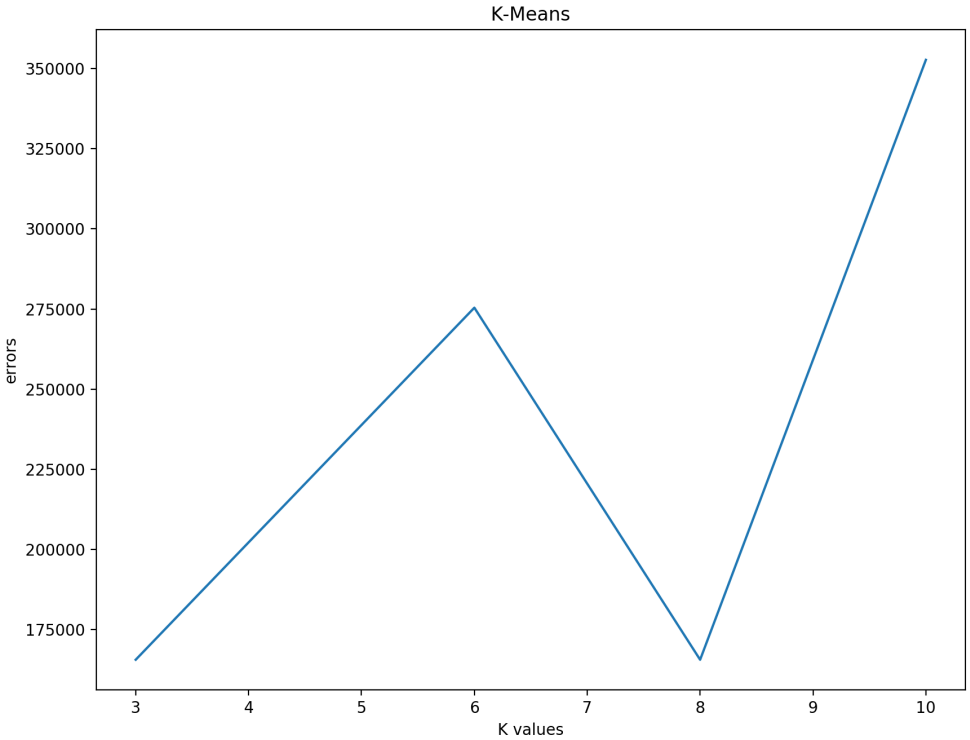
Homework 7: K-Means

The K-means clustering is a method of vector quantization. The Purpose of the algorithm aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. The problem with this algorithm is that it is very computational difficult. The algorithm has a loose relationship to the k-nearest neighbor classifier, a popular machine learning technique for classification that is often confused with k-means because of the k in the name.

Results:

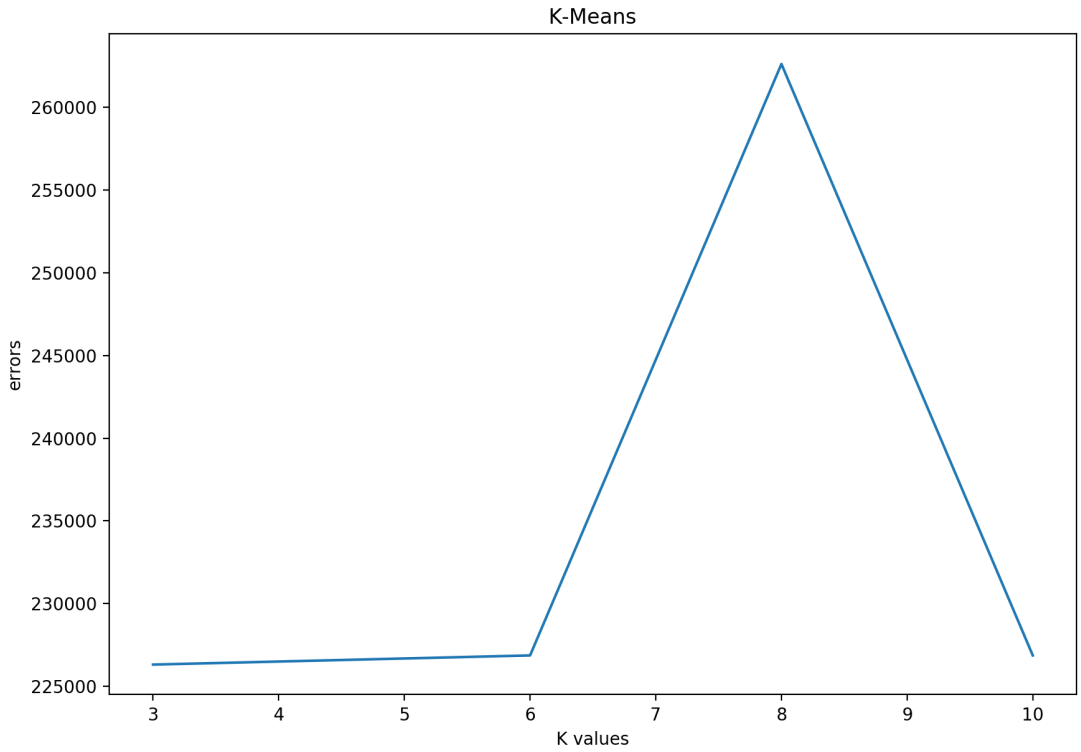
First Run:

Found due to 0 minimum changes
Found after 2 iterations
Found due to 0 minimum changes
Found after 8 iterations
Found due to 0 minimum changes
Found after 2 iterations
Found due to 0 minimum changes
Found after 14 iterations



K = 3)
SSE = 165619.5306719591)
K = 6)
SSE = 275417.0247294612)
K = 8)
SSE = 165619.5306719591)
K = 10)
SSE = 352724.87354760827)

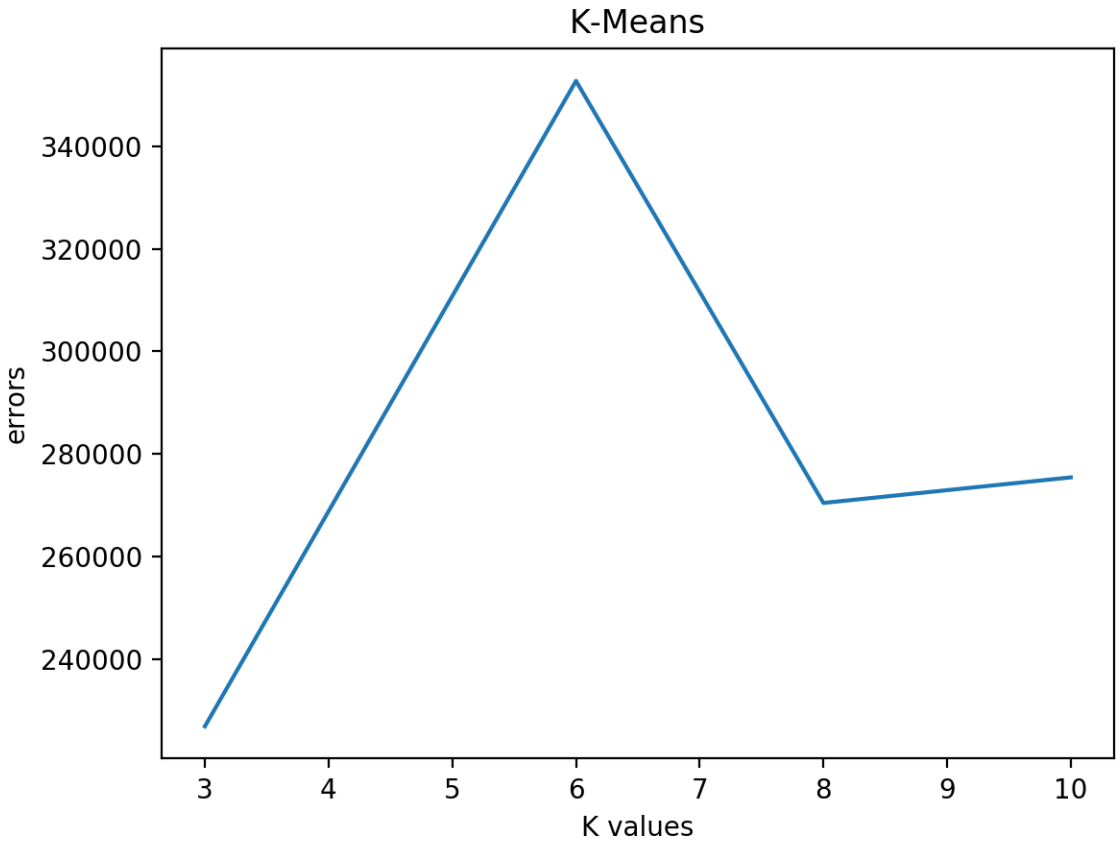
Second Run:
Found due to 0 minimum changes
Found after 11 iterations
Found due to 0 minimum changes
Found after 5 iterations
Found due to 0 minimum changes
Found after 6 iterations
Found due to 0 minimum changes
Found after 12 iterations



K = 3)
SSE = 226326.81527117064)
K = 6)

SSE = 226879.9606208076)
K = 8)
SSE = 262592.1337478175)
K = 10)
SSE = 226879.9606208076)

Third Run:
Found due to 0 minimum changes
Found after 11 iterations
Found due to 0 minimum changes
Found after 19 iterations
Found due to 0 minimum changes
Found after 10 iterations
Found due to 0 minimum changes
Found after 23 iterations



K = 3)
SSE = 226879.9606208076)
K = 6)
SSE = 352724.8735476082)
K = 8)

SSE = 270429.6436111089)
K = 10)
SSE = 275417.0247294612)

Conclusion:

For this examples, we see that three and eight cluster gave the least amount of error. However, we see great variation of results depending on the run. This could be attributed to the initial setting of the centroids. Since each one starts at a random place it could greatly affect the end results. Due to the nature of its randomness we can see that each run could output multiple different results. Nevertheless, three Clusters seems to give the most consistent answer. This could be attributed to the same clusters regardless of the point of where they start. The more cluster, the more one cluster can overtake the other.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from download_data import download_data

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)
    data_dimensions = len(data[0])
    xy_samples = data

    # make sure not to show plots if more than 2 dimensions
    if data_dimensions != 2:
        showPlots = False

    if showPlots:
        plt.scatter(xy_samples[:, 0], xy_samples[:, 1], c='g')
        plt.title("Data Points")
        plt.show()

    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)

    # initialize dictionary
    grouped_coordinates_indexes = initialize_dict(k_value)
```

```
    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
            centroid_points = np.random.randint(samples_size, size=(k_value,
data_dimensions))
            if showPlots:
                for i in range(k_value):
                    plt.scatter(centroid_points[i, 0], centroid_points[i, 1],
c=colors[i], marker='x')
                plt.scatter(xy_samples[:, 0], xy_samples[:, 1], c='g')
                plt.title("Initial centroids")
                plt.show()
        else:
            # find centroids based on the current memberships
            centroid_points = _get_centroids(data_dimensions, xy_samples,
grouped_coordinates_indexes)
            if showPlots:
                _display_plot("Calculate centroids", centroid_points,
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)

        if showPlots:
            _display_plot("Assign points to two centroids", centroid_points,
xy_samples,
                          new_grouped_coordinates_indexes)

        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
            if data_dimensions == 2 and not showPlots:
                _display_plot("Assign points to two centroids",
centroid_points, xy_samples,
                              new_grouped_coordinates_indexes)

            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, :]
        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[centr
oid_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 # start ####################
    result = np.linalg.norm(vector1 - vector2)
    ##################placeholder # end ####################
    return result


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):
        """ take care of issue of empty list
        """
        if len(array_values) == 0:
            return 0
        return int(array_values.mean())

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

    ###################placeholder # end ####################

    return centroid_points


def _display_plot(title, centroid_points, xy_coordinate_samples,
grouped_coordinate_sample_indexes):
    for i in range(len(centroid_points)):
        plt.scatter(centroid_points[i, 0], centroid_points[i, 1],
c=colors[i], marker='x')

plt.scatter(xy_coordinate_samples[grouped_coordinate_sample_indexes[i], 0],

xy_coordinate_samples[grouped_coordinate_sample_indexes[i], 1], c=colors[i])
```

```
    plt.title(title)
    plt.show()


def get_sum_of_squares(dimensions, center, samples):
    """  Get the sum of squared error, given centroid and all of its grouped
points  """
    ###################placeholder # start ####################
    if (len(samples) == 0):
        return 0
    sse = np.sqrt(np.sum((samples - np.transpose(center)) ** 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]))
```