Machine Learning
COMP 5630/ COMP 6630/ COMP 6630 - D01

Instructor: Dr. Shubhra ("Santu") Karmaker
TA 1: Dongji Feng
TA 2: Souvika Sarkar
Department of Computer Science and Software Engineering
Auburn University

Fall, 2022 October

26, 2022

Assignment #7
K-Means Clustering

Submission Instructions

This assignment is due Tuesday, November 8, 2022, at 11:59 pm. Please submit your solutions via Canvas (https://auburn.instructure.com/). You should submit your assignment as a PDF file. Please do not include blurry scanned/photographed equations as they are difficult for us to grade.

Late Submission Policy

The late submission policy for assignments will be as follows unless otherwise specified:

1. 75% credit within 0-48 hours after the submission deadline.
2. 50% credit within 48-96 hours after the submission deadline.
3. 0% credit beyond 96 hours after the submission deadline.

Tasks

1 K-Means Implementation [50 pts]

In this problem, you will implement Lloyd's method for the k-means clustering problem and answer several questions about the k-means objective, Lloyd's method, and k-means++. Recall that given a set $S = x_1,\ldots,x_n \rightarrow Rd$ of n points in d-dimensional space, the goal of the k-means clustering is to find a set of centers $c_1,\ldots,c_k \in R^d$ that minimize the k-means objective:

$$\sum_{j=1}^{n} \min_{i \in \{1,\ldots,k\}} ||x_j - c_i||_2$$

which measures the sum of squared distances from each point x_j to its nearest center.

Consider the following simple brute-force algorithm for minimizing the k-means objective: enumerate all the possible partitionings of the n points into k clusters. For each possible partitioning, compute the optimal centers $c_1,\ldots,c_k$ by taking the mean of the points in each cluster and computing the corresponding k-means objective value. Output the best clustering found. This algorithm is guaranteed to output the optimal set of centers, but unfortunately, its running time is exponential in the number of data points.

a) **[10 pts]:** For the case $k = 2$, argue that the running time of the brute-force algorithm above is exponential in the number of data points n.

A brute-force algorithm for k=2 means is exponential in the number of data points n. This is because we find all possible partitioning's with k centroids. We compare every possible centroids initialization on to every possible point, with n being the number of points's. With this we know that $2^n/2$ is the total number of partitions we can have with 2 centroids and n data points.

In class, we discussed that finding the optimal centers for the k-means objective is NPhard, which means that there is likely no algorithm that can efficiently compute the optimal centers. Instead, we often use Lloyd's method, which is a heuristic algorithm for minimizing the k-means objective that is efficient in practice and often outputs reasonably good clusterings. Lloyd's method maintains a set of centers $c_1,\ldots,c_k$ and a partitioning of the data S into k clusters, $C_1,\ldots,C_k$. The algorithm alternates between two steps: (i) improving the partitioning $C_1,\ldots,C_k$ by reassigning each point to the cluster with the nearest center, and (ii) improving the centers $c_1,\ldots,c_k$ by setting c_i to be the mean of those points in the set C_i for $i = 1,\ldots,k$. Typically, these two steps are repeated until the clustering converges (i.e, the partitioning $C_1,\ldots,C_k$ remains unchanged after an update). Pseudocode is given below:

(a) Initialize the centers $c_1,\ldots,c_k$ and the partition $C_1,\ldots,C_k$ arbitrarily.

(b) Do the following until the partitioning C1,..., Ck does not change:

i. For each cluster-index i, let $C_i = \{x \in S : x$ is closer to c_i than any other center$\}$, breaking ties arbitrarily but consistently. ii. For each cluster index i, let $c_i = \frac{1}{|Ci|} \sum_{x \in C_i} x$.

In the remainder of this problem, you will implement and experiment with of Lloyd's kmeans clustering algorithm for image segmentation. Specifically, You will work on the "k-means.py" python file along with the image dataset and template package provided to you. Using PIL, this program will load a selected image, represent each pixel using its RGB values and analyze pixel-by-pixel RGB values to find the centroid

values of the image. *K* represents the number of centroids that are initialized randomly within the min and max RGB values of the image. Once the centroid values have been optimized using k-means, the program will produce and display the segmented image with the found RGB centroid values.

(a) **[10 pts]:** Complete the function *assignPixels*(*centroids*). The input *centroids* is a list of current centroids. The function assigns each pixel to the current centroids for the algorithm. Specifically, this method finds the closest centroid to the given pixel, then assigns that centroid to the pixel. The function should return a dictionary *clusters* where the keys are the unique centroids, and values are the pixels assigned to each centroid. Note that, this process can reduce the number of clusters if there are duplicate centroids.

(b) **[10 pts]:** Complete the function *adjustCentroids*(*clusters*) that is used to recenter the centroids according to the pixels assigned to each. A mean average is applied to each cluster's RGB values, which are then set as the new centroids. Output is the list of new centroids.

(c) **[10 pts]:** Complete the function *initializeKmeans*(*someK*) which creates a list of *K* centroids and initializes them with the RGB values of randomly selected *K* different pixels. That means, first sample *K* sample pixels, i.e. (x,y) locations, read their RGB values and used those RGB values for initializing the *K* centroids. Finally, return the list of *K* centroids.

(d) **[10 pts]:** Complete the function *iterateKmeans*(*centroids*) that iterates the kmeans clustering steps for maximum 20 iterations. However, you can stop early if *converged*(*centroids,old centroids*) returns *True*. Converged is a function provided to you that will help determine if the centroids have converged or not.

2 Analyze the Effect of k on Lloyd's method [20 pts]

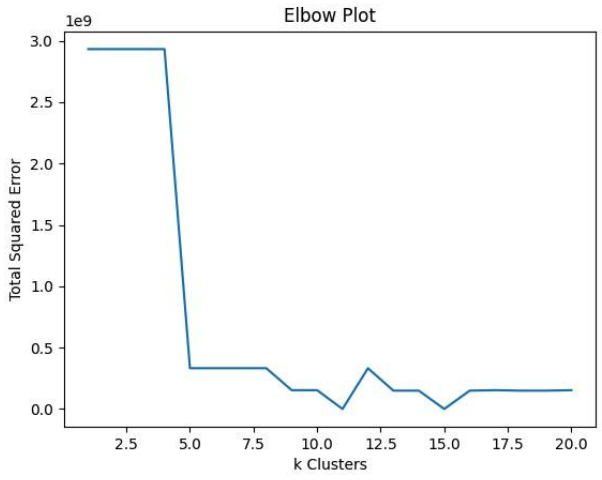
In this part, you will investigate the effect of the number of clusters *k* on Lloyd's method and a simple heuristic for choosing a good value for *k*. Your dataset will still be the 12 different 2D images provided in the "img" folder.

(a) **[10 pts]:** Write a short script to plot the k-means objective value obtained for each value of k in 1,...,20. The x-axis of your plot should be the value of *k* used, and the y-axis should be the k-means objective. Include both the plot and script in your written report.

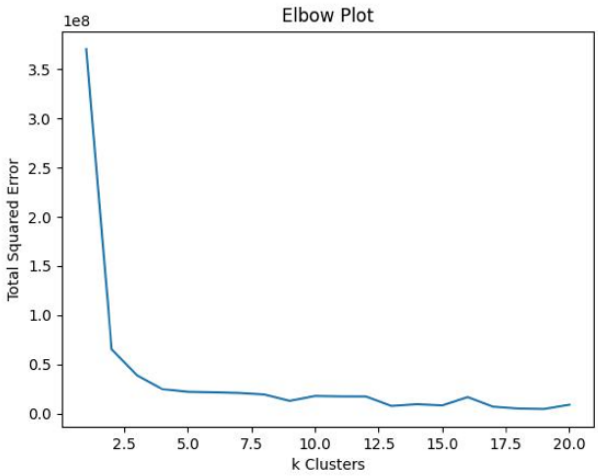
All the included code will be pasted at the end of the report with proper documentation. "elbowPlot()" and "objectiveFunction()" are the two main scripts to be referred to below. Additionally the code will be provided in a separate folder, as well as a link to the repository.

Its important to mention that the specific plots included below are there due to their mention in section 3. Further, plot 11 had a lot of RGB value variety for initial randomization. Meaning the elbow plot was one of the clearer curves. Less sporadic then image 5's elbow plot as the number of different RGB values in the image is much less.

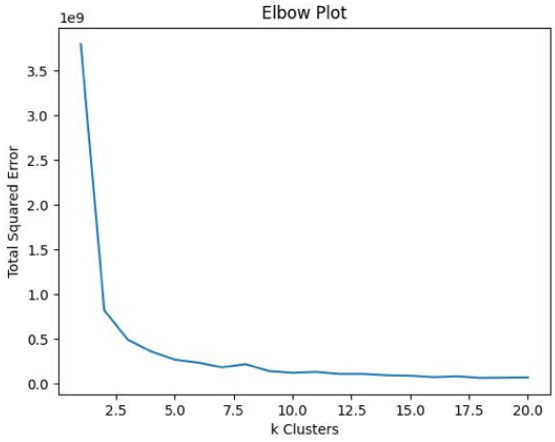
ELBOW PLOT for IMAGE 5



ELBOW PLOT for IMAGE 8



ELBOW PLOT for IMAGE 11



(b) **[10 pts]:** One heuristic for choosing the value of *k* is to pick the value at the "elbow" or bend of the plot produced in part (a), since increasing *k* beyond this point results in a relatively little gain. Based on your plot, what value of *k* would you choose? Does this agree with your intuition when looking at the data? Why or why not?

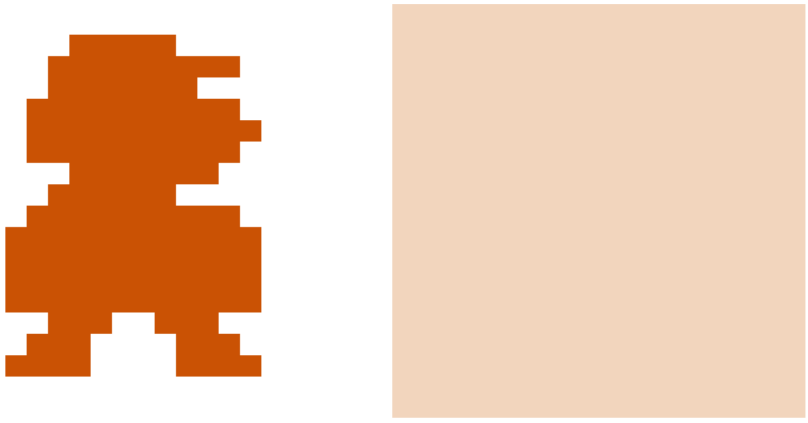
Looking at the plots provided above, you can see the diminished gains per increasing k clearly. You can also see the that the "intuitive" k can depend greatly on the image selected. The reason for this is discussed a little above, but it boils down to the number of centroids that best reduce the total squared error and their placement. With image 5, for example, the number of centroids that can be randomly initialized to one of the data points is low, because the amount of unique datapoints is low. Meaning, that the number of pixels with unique RGB values is less in image 5 then the other images. This means that smaller k values can suffice to group data properly, but placement of these centroids has a much larger impact on the elbow plot.

The "intuitive" choice will be right around the point where the squared error begins to flatten. Looking at image 8's elbow plot is perhaps the most revealing, as there is a distinct change in slope, or gain, from 2 clusters to 3 or 4. My intuitive choice would be 4, as 5+ clusters has no perceivable reduction in squared error. When looking at only the data, with k = 4 we are able to see all the features of the photo for proper identificiation, for example we see that Christian Bale is wearing a suit and tie. k = 5, 6, 7 has those same identifiable features with an additional change given to those same features. If we look at k = 3 not only is it difficult to tell he is wearing a suit, but you cannot make out the features on one side of his face. This leads me to believe my intuition being correct. K = 4 looks to be the best selection.

3 Analyze the Effect of Initialization on Lloyd's method [30 pts]

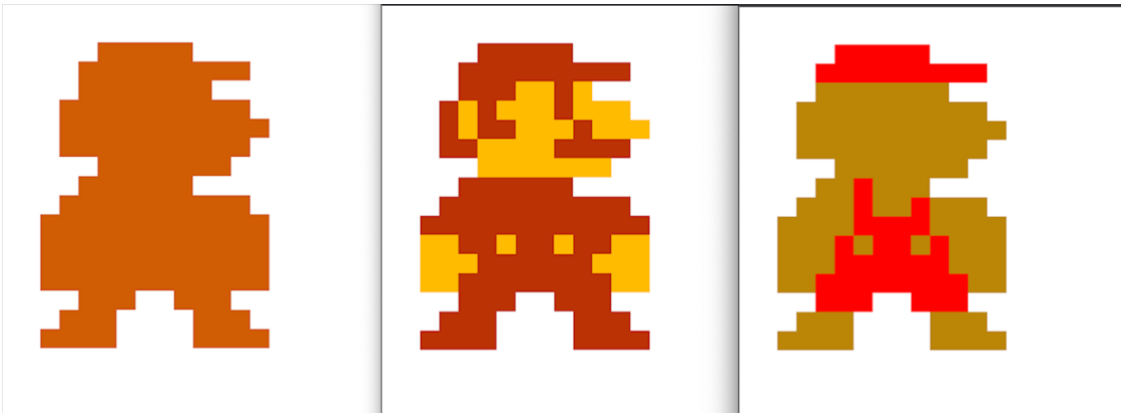
Now, you will investigate the effect of centroid initialization on Lloyd's method. In particular, you will compare random initialization with Lloyd's method. Note that the provided k-means script has the randomized initialization already implemented. In this problem, you just need to run the code and answer questions.

a) **[5 pts]:** For test image 5, set *k* = 2 and run Lloyd's method 10 times (each time with random initialization) and save the 10 output plots. What do you see? Can you explain the output?



All 10 of the generated images looked like one of the two images displayed above. Because the variety of RGB values in the original photo are relatively low, the random initialization of centroids has a high reliance on placement. For the left image above, the initialized centroids clustered the data better than the right image above. In the right image above, one of the centroids was placed in such a way that all the pixels were clustered with it.

b) **[5 pts]:** For test image 5, set k = 10 and run Lloyd's method 10 times (each time with random initialization) and save the 10 output plots. What do you see? Can you explain the output?



All 10 of the generated images looked like one of the three images displayed above. As expected with k being higher we see a chance for the squared error to be reduced further then if k = 2. However, as mentioned above, because there is only about 3 different RGB values each pixel could be, (3 values centroids could be initialized too), placement of each centroid in relation to each pixel value still dictate output quite a bit.

c) **[10 pts]:** For test image 8, set k = 2 and run Lloyd's method 10 times (each time with random initialization) and save the 10 output plots. Pick an image that shows the highest contrast. Repeat this process for k = {3,4,5}. Again, pick the highest contrast image for each k. Add the highest contrast images to your report. You should have 4 images.

d) **[10 pts]:** Based on all these experiments, propose some heuristics for initialization of centroids (apart from random initialization) which can help achieve meaningful segmentation results.

One easy measure would be to ensure that during initialization no duplicate centroid is made by probabilistically choosing subsequent centroids selected from the remaining data points that are a squared distance away from any already existing centroid. This will ensure that centroids are spread out and covering more of the data space. This method is also known as k-means++. Initialization heuristics like this one can help more meaningful segmentation of data.

Disclaimers: This assignment re-uses some materials from the publicly available website: CMU Introduction to Machine Learning Course, 10-315, Spring 2019. I personally thank Prof. Maria-Florina Balcan for sharing her teaching materials publicly. This assignment is exclusively used for instructional purposes.

SCRIPTS AND CODE:

```
#!/usr/bin/python
from PIL import Image, ImageStat
import numpy as np
import random
import matplotlib.pyplot as plt
from tqdm import tqdm
# =========
# converged
# =========
#
# Will determine if the centroids have converged or not.
# Essentially, if the current centroids and the old centroids
# are virtually the same, then there is convergence.
#
# Absolute convergence may not be reached, due to oscillating
# centroids. So a given range has been implemented to observe
# if the comparisons are within a certain ballpark
#


def converged(centroids, old_centroids):
    if len(old_centroids) == 0:
        return False

    if len(centroids) <= 5:
        a = 1
```

```
    elif len(centroids) <= 10:
        a = 2
    else:
        a = 4

    for i in range(0, len(centroids)):
        cent = centroids[i]
        old_cent = old_centroids[i]

        if ((int(old_cent[0]) - a) <= cent[0] <= (int(old_cent[0]) + a)) and ((int(old_cent[1]) - a) <=
cent[1] <= (int(old_cent[1]) + a)) and ((int(old_cent[2]) - a) <= cent[2] <= (int(old_cent[2]) +
a)):
            continue
        else:
            return False

    return True

def getMin(pixel, centroids):
    """Find and return the centroid that has the smalled distance to each pixel

    Args:       pixel and list of centroids

    Return:     centroid with the minimum distance to the pixel
    """
    minDist = 9999
    minIndex = 0
```

```
    for i in range(0, len(centroids)):
        d = np.sqrt(int((centroids[i][0] - pixel[0]))**2 + int((centroids[i][1] - pixel[1]))**2 +
int((centroids[i][2] - pixel[2]))**2)
        if d < minDist:
            minDist = d
            minIndex = i

    return centroids[minIndex]

def assignPixels(centroids:list) -> dict:
    """Groups pixels and assigns to closes centroid

    Args:       list containing centroid coords

    Returns:    dict with keys being centroid
        and vals being pixels closest to that centroid
    """
    clusters = dict.fromkeys(centroids)

    # initialize values in dict as list()
    for key in clusters:
        clusters[key] = list()

    for h in range(img_height):
        for w in range(img_width):
            closest_centroid = getMin(px[w, h], centroids)
```

```
            clusters[closest_centroid] += tuple(px[w, h]),

    return clusters

def adjustCentroids(clusters: dict) -> list:
    """Recenter the centroid using the mean average of each clusters pixels

    Args:       dict of centroids with assigned pixels {centroid: [pixel list]}

    Returns:    list of new centroid coords
    """
    new_centroids = []

    for old_centroid, pixels in clusters.items():
        new_centroids += tuple(np.mean(pixels, axis=0)),

    return new_centroids

def initializeKmeans(k: int) -> list:
    """Create a list of k number of centroids by randomly sampling pixels

    Args:       k int representing number of clusters

    Returns:    list of centroids
    """
    centroids = []
```

```
    for i in range(k):
        random_pixel = px[random.randint(0, img_width-1), random.randint(0, img_height-1)]
        centroids += (random_pixel),

    return centroids

def iterateKmeans(centroids: list) -> list:
    """Iterate the k-means clustering steps for <= 20 steps or for convergence

    Args:       initial list of centroics

    Returns:    list of centroids (final result)
    """
    old_centroids = centroids

    for i in range(20):
        clusters = assignPixels(old_centroids)
        centroids = adjustCentroids(clusters)

        if converged(centroids, old_centroids):
            break
        old_centroids = centroids

    return centroids

# ==========
# drawWindow
```

```
# ==========
#
# Once the k-means clustering is finished, this method
# generates the segmented image and opens it.
#
def drawWindow(iteration, result):
    img = Image.new('RGB', (img_width, img_height), "white")
    p = img.load()

    for x in range(img.size[0]):
        for y in range(img.size[1]):
            RGB_value = getMin(px[x, y], result)
            p[x, y] = tuple([int(i) for i in list(np.round_(np.array(RGB_value)))])

img.save("./img_results/img"+str(num_input)+"_withk"+str(k_input)+"_icoutn"+str(iteration)+".
png")
    img.show()

def objectiveFunction(centroids: list) -> float:
    """Optimization criterion is to minimize total sqaured error
            between pixels and centroids

        Args: list of centroids, then used to make clusters

        Returns: Summation of the squared errors between pixels and centroid
    """
    clusters = assignPixels(centroids)
```

```
    SE_centroid = []
    for centroid, pixels in clusters.items():
        # Kmeans objective function
        SE_centroid += np.sum( np.sum ( np.subtract(pixels, centroid) **2, axis=0 ) ),


    Total_SE = np.sum(SE_centroid)
    return Total_SE

def elbowPlot():
    x = []
    y = []

    for k in tqdm(range(1, 21)):
        x += k,
        k_centroids=initializeKmeans(k)
        k_result = iterateKmeans(k_centroids)
        y += objectiveFunction(k_result),

    plt.xlabel("k Clusters")
    plt.ylabel("Total Squared Error")
    plt.title("Elbow Plot")
    plt.plot(x, y)
    fig = plt.gcf()
    fig.savefig("./plots/img"+str(num_input)+"_plot.jpg", dpi=100)
    plt.show()
```

```
def experiment():
    for i in tqdm(range(1, iterations+1)):
        k_centroids=initializeKmeans(k_input)
        k_result = iterateKmeans(k_centroids)
        drawWindow(i, k_result)




num_input = str(input("Enter image number: "))
k_input = int(input("Enter K value: "))
iterations = int(input("Enter the number of epochs: "))

img = "img/test" + num_input.zfill(2) + ".jpg"
im = Image.open(img)
img_width, img_height = im.size
px = im.load()
elbowPlot()
# experiment()
```