Assignment-6: Unsupervised Learning (k-Means Clustering)

Sajan Kumer Sarker
Id: 2111131642
Email: sajan.sarker@northsouth.edu

Rafsan Jani Chowdhury
Id: 2011424642
Email: rafsan.chowdhury@northsouth.edu

Rosely Mohammad
Id: 2014219642
Email: rosely.mohammad@northsouth.edu

November 26, 2024

1. Introduction

In this assignment report, we will discuss about the implementation process of K-means Clustering algorithm. We implemented this for n-dimensional feature vectors or data points . After implementing the k-means clustering algorithm we first apply this and plot the cluster of 'kmeans2d.npy' datasets and then we compress an image using our k-means clustering algorithm. Our main objective was to implement the K-means algorithm without using library functions and to ensure that it supports n-dimensional feature vectors or data points and produce similar visualizations, as shown in the provided notebook.

2. Methods

Necessary Imports:

```python
import numpy as np
import matplotlib.pyplot as plt
```

We have imported all necessary libraries and modules to be used in each notebook.

2.1 User Define K-Means Algorithm Functions:

Cluster Centroids Initialization:

```python
# randomly initialize cluster centroids
def initialize_centroids(X, k):
  return X[np.random.choice(X.shape[0], size=k, replace=False)]
```

This function randomly selects k number of unique data points from the given dataset 'X' and initializes them as the starting centroids for the clustering algorithm.

Cluster Assignment:

```
# Assign data point to the nearest cluster centroid
def assign_clusters(X, centroids):
  distances = np.sqrt(((X - centroids[:, np.newaxis])**2).sum(axis=2))
  return np.argmin(distances, axis=0)
```

This function will assign each data points to the closest cluster centroid. It first calculate the distance between each data point and all the centroids, then assigns each data point to the centroid with the smallest distance.

Cluster Centroids Update:

```
# move cluster centroids
def move_centroids(X, labels, k):
  centroids = np.zeros((k, X.shape[1]))

  for i in range(k):
    points = X[labels == i]

    if len(points) > 0:
      centroids[i] = points.mean(axis=0)
  return centroids
```

This function updates the cluster centroids based on the current clusters. For each cluster, it calculated the mean for all data points assigned to the cluster. These mean position becomes the new centroids.

Cost Function:

```
# cost function
def kmeans_cost(X, centroids, labels):
  cost = 0
  m = len(X)
  for i in range(len(X)):
    cost += np.linalg.norm(X[i] - centroids[labels[i]])**2

  cost = (1/m)*cost
  return cost
```

This cost function will compute the k-means cost, which is average squared distance between each data point and its assigned cluster centroid.

K-means Algorithm:

```
# k-means algorithm
def kmeans(X, k):
  max_iters=100
  tol=1e-4
  costs = []
  centroids = initialize_centroids(X, k)

  for iter in range(max_iters):
    labels = assign_clusters(X, centroids)
    cost = kmeans_cost(X, centroids, labels)
    costs.append(cost)

    new_centroids = move_centroids(X, labels, k)
    if np.all(np.abs(new_centroids - centroids) < tol):       # Check for convergence
      break

    centroids = new_centroids
  return centroids, labels, iter , costs    # c(m), mu(k)
```

This function implements the k-means clustering algorithm, which will partitions the dataset into K clusters. It first initialize the centroids, then assigns data points to the nearest centroid. and then update the centroids iteratively. At each iteration, the cost is computed and the algorithm checks for convergence based on centroid movement to a predefined tolerance. The algorithm will stop if the centroids converge or after reaching the maximum number of iterations.

Optimized K-Means with Multiple Runs:

```
# for n-dimention
def run_kmeans(X, k):
  n = 100
  best_cost = float('inf')
  best_centroids = None
  best_labels = None
  best_costs = None

  for i in range(n):
    centroids, labels, iter , costs = kmeans(X, k)
    final_cost = costs[-1]

    if final_cost < best_cost:    # only updated if a better initial point is found (as cost w
      best_cost = final_cost
      best_centroids = centroids
      best_labels = labels
      best_costs = costs
      best_iter = iter

  return best_centroids, best_labels, best_costs, best_iter
```

This function will run the k-means algorithm multiple times to find the best initial points that minimize the clustering cost. It initializes the best cost with infinity and iterates over a fixed number of runs. For each run, it initializes centroids randomly, executes the k-means algorithm, and evaluates the final cost. The function tracks the best solution, defined as the clustering with the lowest final cost, and updates the centroids, labels, cost history, and iteration count if a better solution is found.

Elbow Method: Cost vs. Number of Clusters:

```
# cost vs k graph for choosing the value of k
def cost_vs_k(X):
  total_cost = []

  for k in range(1,11):
    centroids, labels, _, _ = kmeans(X, k)
    cost = kmeans_cost(X, centroids, labels)
    total_cost.append(cost)

  plt.plot(total_cost)
  plt.title('Cost␣vs␣No.␣of␣Clusters(K)')
  plt.xlabel('No.␣of␣Clusters(K)')
  plt.ylabel('Cost')
  plt.xticks(range(len(total_cost)))
  plt.show()
```

This function will plot the cost vs number of clusters (K) graph. It iterates over a range of K values from 1 to 10, and calculate the cost for each value of K. Then it will plots the cost against the number of clusters, showing how the cost decrease as k increase.

2.2 K-Means on 2D Data:

Dataset Loading:

```
X = np.load("kmeans2d.npy")

print("First␣five␣elements␣of␣X␣are:\n", X[:5])
print('\nThe␣shape␣of␣X␣is:', X.shape)

cost_vs_k(X)
```

Imported the dataset and plot the Cost vs Number of Clusters graph to selecting the K's value using Elbow Method.

Running K-Means & Cluster Visualization:

```
#Running K-Means for k = 2
k = 2
best_centroids, best_labels, costs, iter = run_kmeans(X, k)  #Find Best Centroid from 100 Rand

plt.scatter(X[:, 0], X[:, 1], c=best_labels, cmap='viridis', marker='o')
plt.scatter(best_centroids[:, 0], best_centroids[:, 1], s=300, c='red', marker='X')
plt.title('Clusters␣Visualization')
plt.show()
```

Run K-Means for K=2 and visualize the clusters.

Running K-Means & Cluster Visualization:

```
#Running K-Means for k = 3
k = 3
best_centroids, best_labels, costs, iter = run_kmeans(X, k)  #Find Best Centroid from 100 Rand

plt.scatter(X[:, 0], X[:, 1], c=best_labels, cmap='viridis', marker='o')
plt.scatter(best_centroids[:, 0], best_centroids[:, 1], s=300, c='red', marker='X')
plt.title('Clusters␣Visualization')
plt.show()
```

Run K-Means for K=3 and visualize the clusters.

2.3 Image Compression using K-Means:

Loading Image:

```
original_img = plt.imread('/content/bird_small.png')
plt.imshow(original_img)
print("Shape␣of␣original_img␣is:", original_img.shape)
```

Import the image and display the image.

Image Pre-processing:

```
# Divide by 255 so that all values are in the range 0 - 1
original_img = original_img / 255

# Reshape the image into an m x 3 matrix where m = number of pixels
# (in this case m = 128 x 128 = 16384)
# Each row will contain the Red, Green and Blue pixel values
# This gives us our dataset matrix X_img that we will use K-Means on.

X_img = np.reshape(original_img, (original_img.shape[0] * original_img.shape[1], 3))
X_img.shape
```

We divided the image's pixel value by 255 to normalize it to a range between 0 to 1. Then, reshape the image into m x 3 matrix, where m is number of pixels of the image. Then for a 128 x 128 size image's size becomes 16384. Now each row of the matrix contain the RBG normalize values.

Running K-Means & Reconstruct the compressed Image:

```
K_img = 8 # 4 8 16
centroids, labels, iter, costs = run_kmeans(X_img, K_img)

print('Cluster_center', centroids.shape)
print('Clusters:', K_img)
print('Labels', labels.shape)
print("Iterations:␣", len(iter))
X_recovered = centroids[labels,:]
X_recovered = np.reshape(X_recovered, original_img.shape)
```

Run the K-means algorithm for different values of K (4, 8, 16, 32) and then, reconstruct the image using the centroids and labels obtained from the clustering process.

Display Original vs Compressed Image:

```
# Display original image
fig, ax = plt.subplots(1,2, figsize=(8,8))
plt.axis('off')

ax[0].imshow(original_img*255)
ax[0].set_title('Original')
ax[0].set_axis_off()

# Display compressed image
ax[1].imshow(X_recovered*255)
ax[1].set_title('Compressed␣with␣%d␣colours'%K_img)
ax[1].set_axis_off()
```

Display the original image and the compressed image.

3. Experiment results

3.1 K-Means Result on 2D Data:

Cost vs Number of Clusters(K):

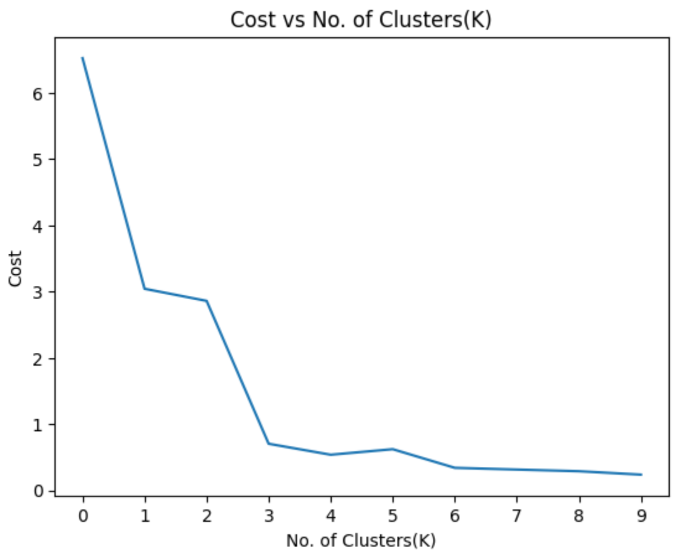

Fig. 1: Cost Vs Number of Clusters

This graph shows the relationship between the number of clusters and the costs. Initially as K increases, the cost decreases significantly because adding more clusters allows the centroids to better represent the data. However, after certain points, the rate of decrease slows down. The elbow method suggests the optimal value for k from the graph.

Clusters Visualization with Different K Values:

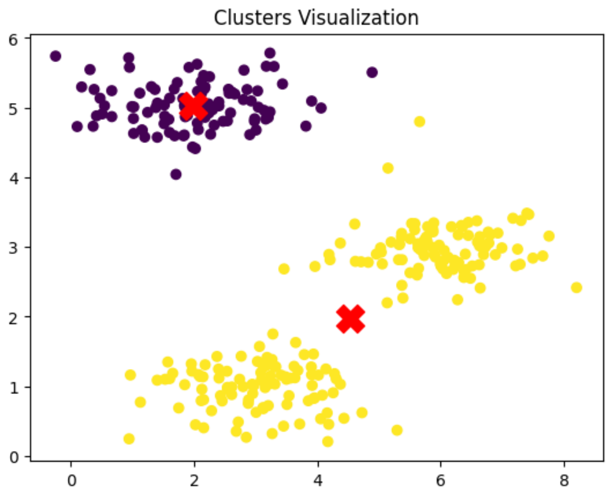

Fig. 2: Cluster Visualization with K=2

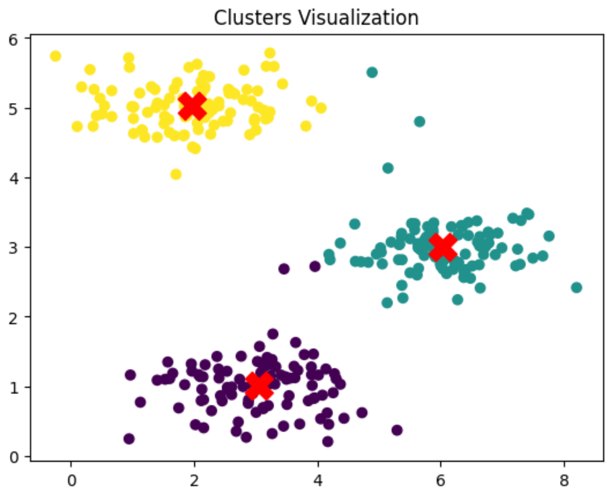

Fig. 3: Cluster Visualization with K=3

Here, Fig-2 shows the data clustered into two groups , where the clusters are less distinct, particularly with one larger cluster containing more speed-out data points. And Fig-3 shows the data clustered into three groups, with well-separated clusters and centroids. Here K=3 provides a more meaningful and well-defined cluster.

3.2 K-Means Result on Image Compression:

Compressed Images for Different Values of K:

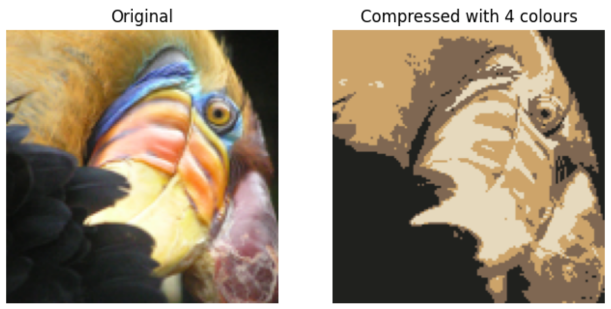

Fig. 4: Compressed with 4 Colors

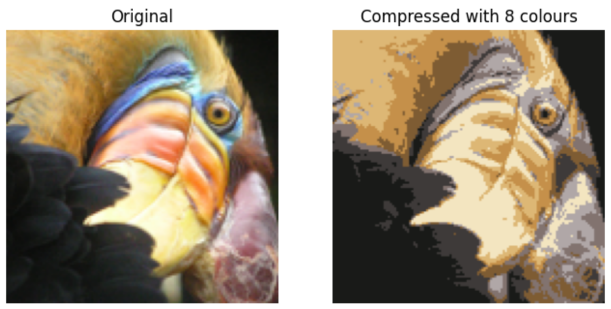

Fig. 5: Compressed with 8 Colors

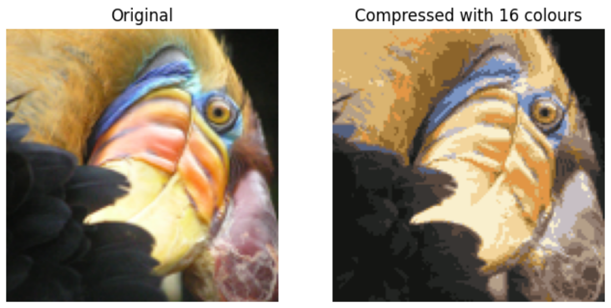

Fig. 6: Compressed with 16 Colors

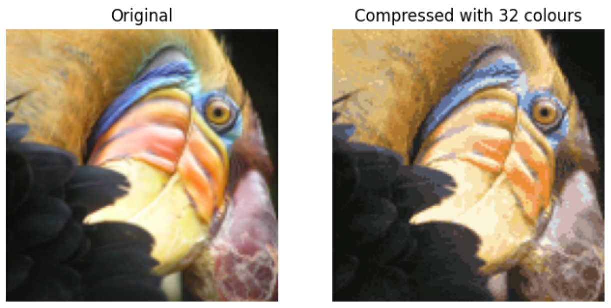

Fig. 7: Compressed with 32 Colors

Here, these images shows that what happens when original image is compressed for different number of clusters. These images loses most of its colors and details for different values of K. As we increase K, we essentially increase the number of colors that each pixel can be assigned, so there is a significant visual improvement in details when K is increased from 4 to 32, such as when k is 32, the compressed image looks almost same as the original image compare to the compressed image for k = 4.

4. Discussion

In this assignment we implemented the K-means clustering algorithm and then we apply this algorithm on a 2D dataset and then we compressed a image using this algorithm. For the 2D dataset, we clustered the data points for different values of K, where K=3 was identified as the optimal number of clusters for the dataset and then visualize that. And then for image compression, the implemented algorithm properly reduced the number of colors of the original image. So in conclusion we can say that, our implemented algorithm properly supports n-dimensional feature vectors or data points and produce similar visualizations, as shown in the provided notebook.