EP4130 – PROJECT REPORT

Aryan Sharan Reddy – BT21BTECH11002

Manikanta Uppulapu – BT21BTECH11005

Image Compression with K-Means Clustering

Abstract

"Image Compression with K-Means Clustering" explores a method to reduce the storage space required for digital images while preserving visual quality. The technique leverages the K-Means Clustering algorithm to group similar pixel colours together and represent them with fewer colours. This process significantly reduces the number of bits needed to store the image, resulting in efficient compression. By assigning each pixel to its closest centroid, the original image can be reconstructed with minimal loss of information. Experimental results demonstrate substantial compression ratios, making this approach valuable for optimizing storage and transmission of images in various applications.

Introduction

"Image Compression with K-Means Clustering" is a technique employed to minimize the storage requirements of digital images while maintaining their visual fidelity. By leveraging the K-Means Clustering algorithm, similar pixel colors are grouped together, reducing the overall number of distinct colors used to represent the image. This process facilitates efficient compression, as fewer bits are needed to encode the image data. In this introduction, we delve into the principles behind this approach, exploring how K-Means Clustering aids in identifying representative color clusters and subsequently reconstructing the compressed image. Through this method, we aim to achieve significant reductions in file size without compromising on image quality, making image compression with K-Means Clustering a valuable tool in various domains such as image storage, transmission, and processing.

K-Means Clustering

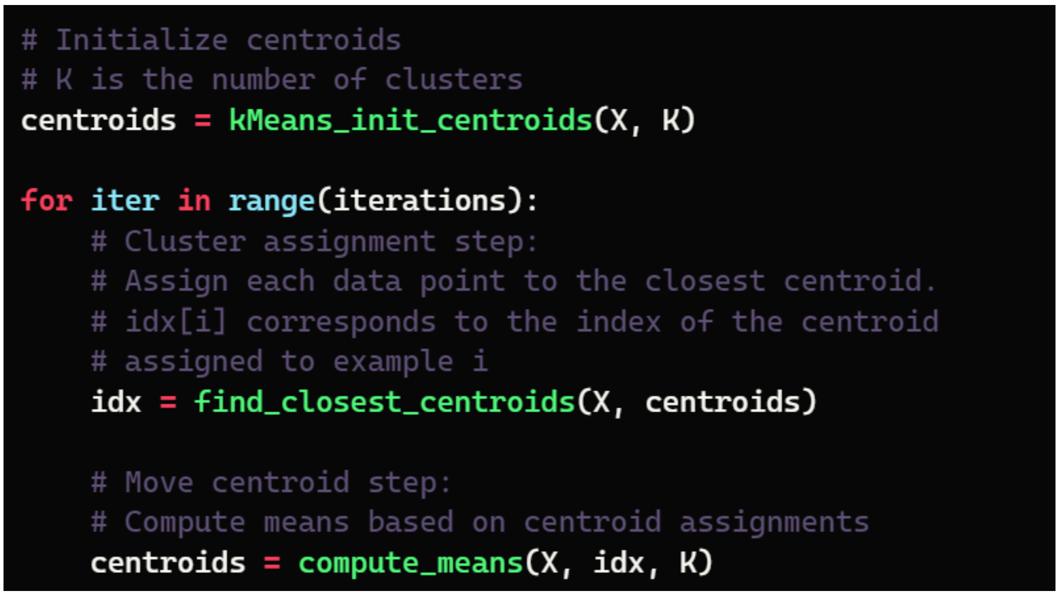
- K-means Clustering is an Unsupervised Learning algorithm, which groups unlabelled data into different clusters.
- Clustering of data points is based on a similarity measure such as euclidean distance between points.
- It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum.
- The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.

Algorithm

The K-means algorithm is a method to automatically cluster similar data points together. Concretely, you are given a training set {x^(1), ..., x^(m)}, and you want to group the data into a few cohesive "clusters".

K-means is an iterative procedure that starts by guessing the initial centroids, and then refines this guess by repeatedly assigning examples to their closest centroids, and then recomputing the centroids based on the assignments.

In pseudocode, the K-means algorithm is as follows:



```
# Initialize centroids
# K is the number of clusters
centroids = kMeans_init_centroids(X, K)

for iter in range(iterations):
    # Cluster assignment step:
    # Assign each data point to the closest centroid.
    # idx[i] corresponds to the index of the centroid
    # assigned to example i
    idx = find_closest_centroids(X, centroids)

    # Move centroid step:
    # Compute means based on centroid assignments
    centroids = compute_means(X, idx, K)
```

The inner-loop of the algorithm repeatedly carries out two steps:

1. Assigning each training example x^(i) to its closest centroid, and
2. Recomputing the mean of each centroid using the points assigned to it.

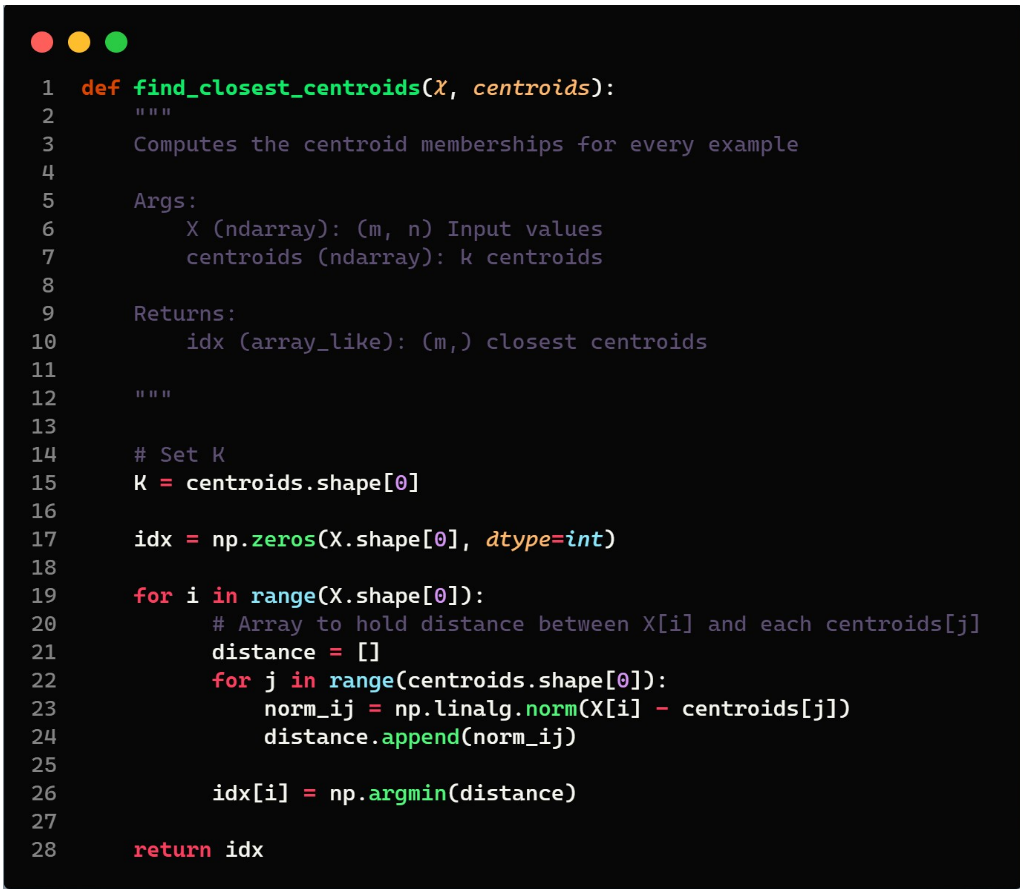
The K-means algorithm will always converge to some final set of means for the centroids. However, the converged solution may not always be ideal and depends on the initial setting of the centroids.

- Therefore, in practice, the K-means algorithm is usually run a few times with different random initializations.
- One way to choose between these different solutions from different random initializations is to choose the one with the lowest cost function value (distortion).

Finding Closest Centroids

1. This function takes the data matrix X and the locations of all centroids inside centroids.
2. It should output a one-dimensional array idx (which has the same number of elements as X) that holds the index of the closest centroid (a value in {1,...,K}, where K is the total number of centroids) to every training example.
3. Specifically, for every example x^(i) we set $c^{(i)} := j$ that minimizes where
    - c^(i) is the index of the centroid that is closest to x^(i) , and
    - μ_j is the position (value) of the j'th centroid.

Code:



```python
def find_closest_centroids(X, centroids):
    """
    Computes the centroid memberships for every example

    Args:
        X (ndarray): (m, n) Input values
        centroids (ndarray): k centroids

    Returns:
        idx (array_like): (m,) closest centroids

    """

    # Set K
    K = centroids.shape[0]

    idx = np.zeros(X.shape[0], dtype=int)

    for i in range(X.shape[0]):
          # Array to hold distance between X[i] and each centroids[j]
          distance = []
          for j in range(centroids.shape[0]):
              norm_ij = np.linalg.norm(X[i] - centroids[j])
              distance.append(norm_ij)

          idx[i] = np.argmin(distance)

    return idx
```

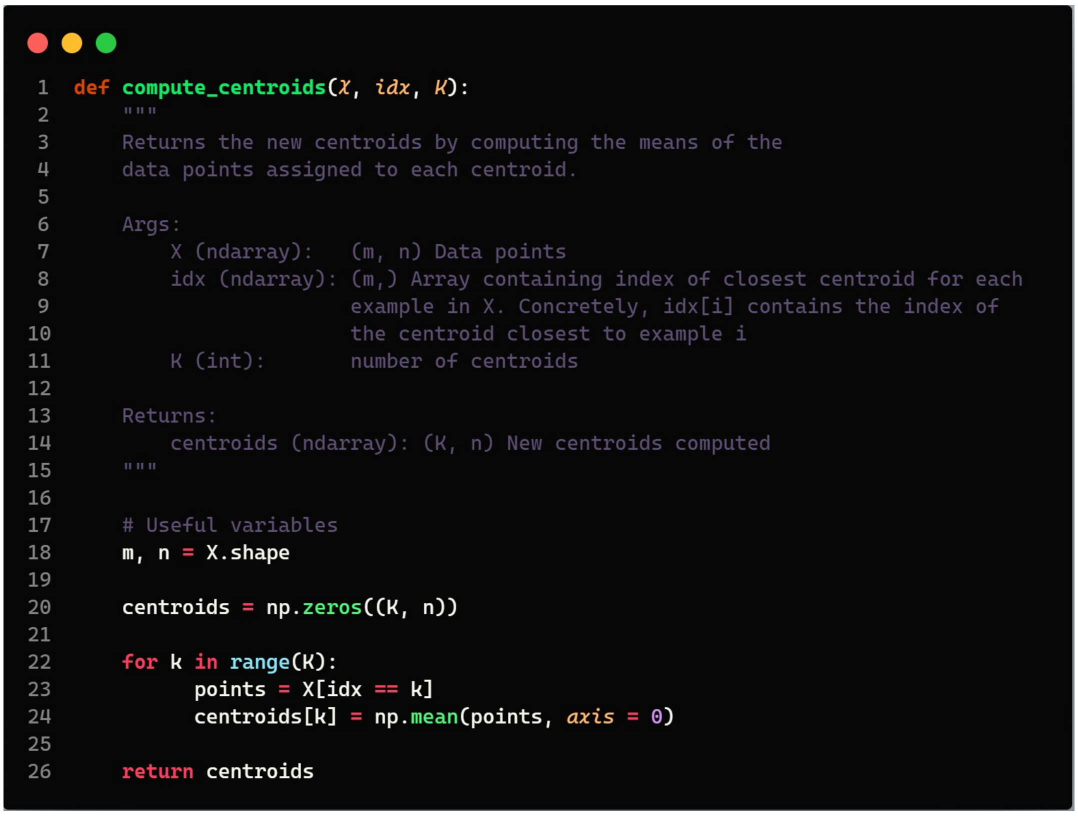
Computing Centroid Means

- For every centroid μ_k we set:

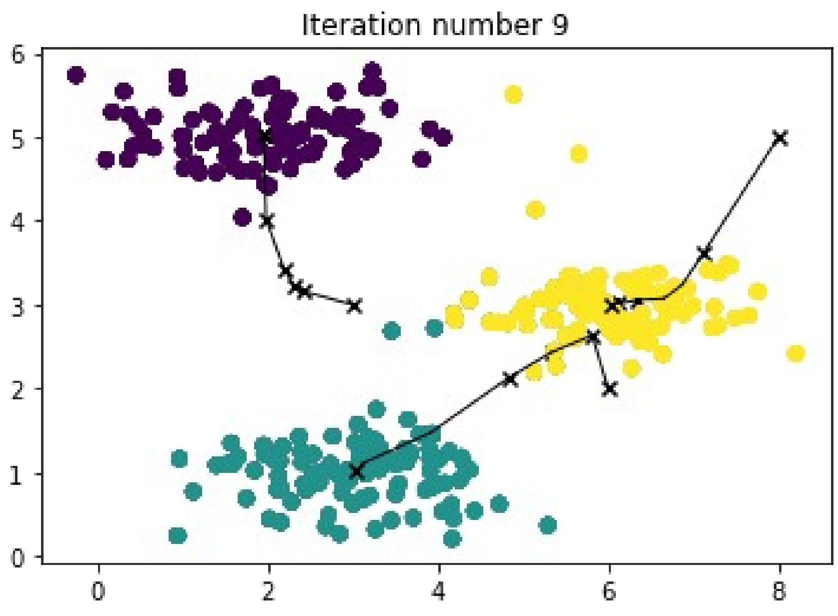
$$\mu_k = \frac{1}{|C_k|} \sum_{i \in C_k} x^{(i)} \text{ where}$$

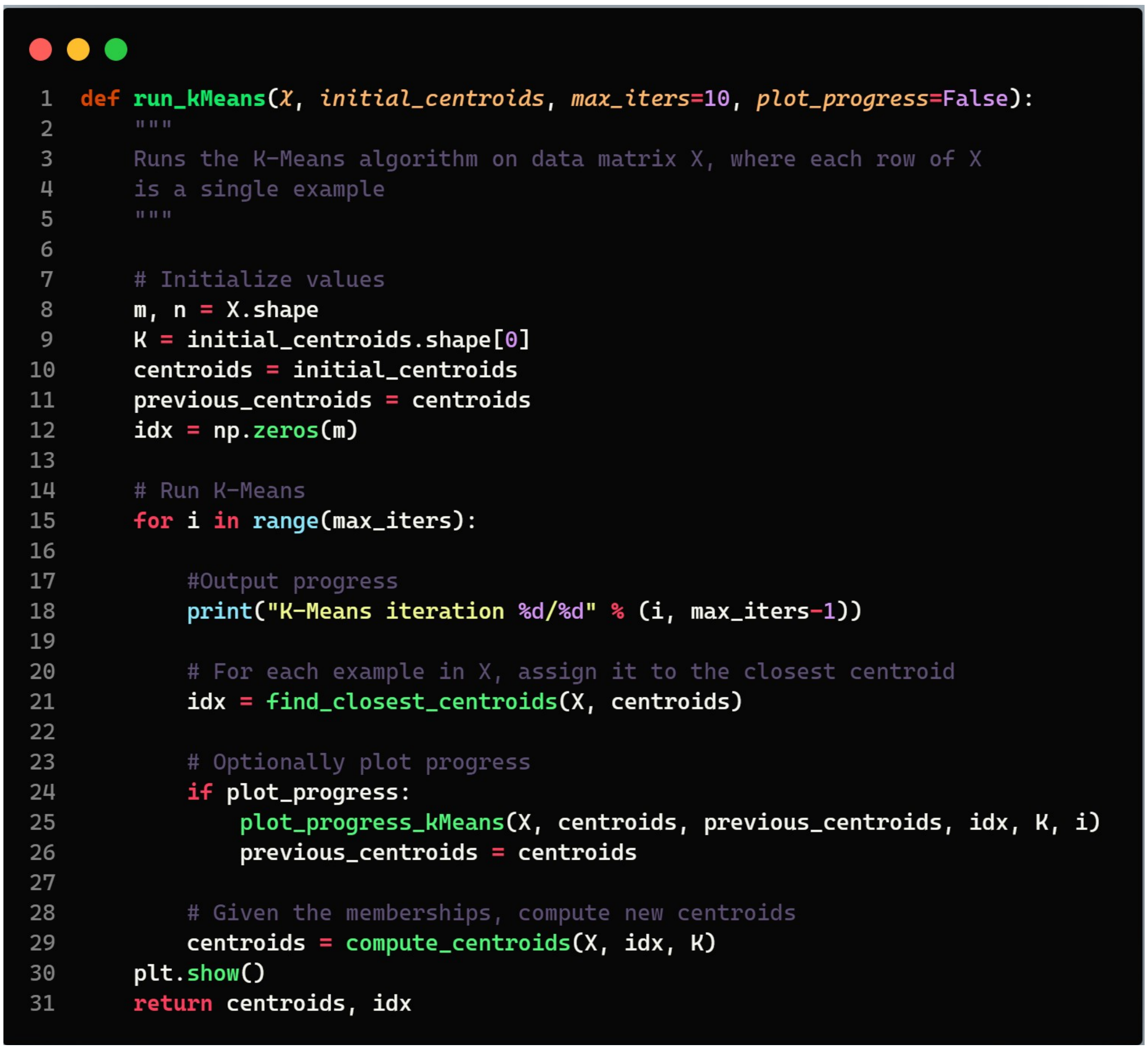
  - C_k is the set of examples that are assigned to centroid k.
  - |C_k| is the number of examples in the set C_k.

Code:

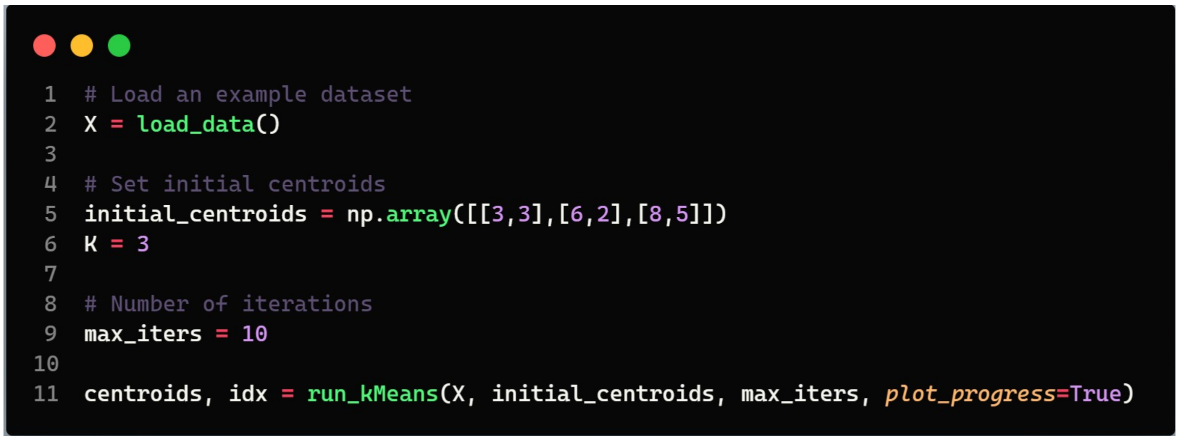


```python
def compute_centroids(X, idx, K):
    """
    Returns the new centroids by computing the means of the
    data points assigned to each centroid.

    Args:
        X (ndarray):   (m, n) Data points
        idx (ndarray): (m,) Array containing index of closest centroid for each
                       example in X. Concretely, idx[i] contains the index of
                       the centroid closest to example i
        K (int):       number of centroids

    Returns:
        centroids (ndarray): (K, n) New centroids computed
    """

    # Useful variables
    m, n = X.shape

    centroids = np.zeros((K, n))

    for k in range(K):
        points = X[idx == k]
        centroids[k] = np.mean(points, axis = 0)

    return centroids
```





```python
def run_kMeans(X, initial_centroids, max_iters=10, plot_progress=False):
    """
    Runs the K-Means algorithm on data matrix X, where each row of X
    is a single example
    """

    # Initialize values
    m, n = X.shape
    K = initial_centroids.shape[0]
    centroids = initial_centroids
    previous_centroids = centroids
    idx = np.zeros(m)

    # Run K-Means
    for i in range(max_iters):

        #Output progress
        print("K-Means iteration %d/%d" % (i, max_iters-1))

        # For each example in X, assign it to the closest centroid
        idx = find_closest_centroids(X, centroids)

        # Optionally plot progress
        if plot_progress:
            plot_progress_kMeans(X, centroids, previous_centroids, idx, K, i)
            previous_centroids = centroids

        # Given the memberships, compute new centroids
        centroids = compute_centroids(X, idx, K)
    plt.show()
    return centroids, idx
```



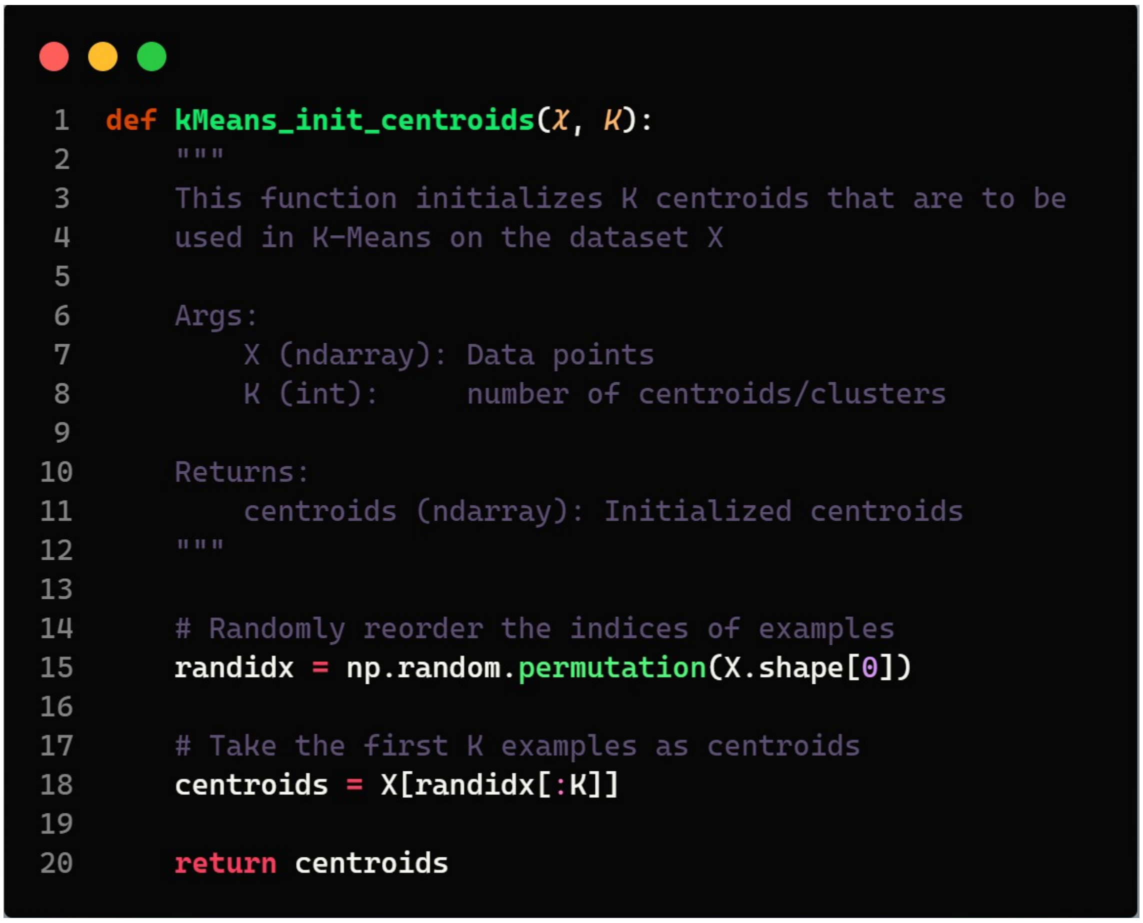
```python
# Load an example dataset
X = load_data()

# Set initial centroids
initial_centroids = np.array([[3,3],[6,2],[8,5]])
K = 3

# Number of iterations
max_iters = 10

centroids, idx = run_kMeans(X, initial_centroids, max_iters, plot_progress=True)
```

Random Initialization

In practice, a good strategy for initializing the centroids is to select random examples from the training set.

Random Initialization is used for two main purposes:

1. **Avoiding Bias:** If we initialize centroids using a biased method, such as selecting the first K data points, it might lead to biased clusters, especially if the data is ordered or structured in a certain way. Random initialization helps in avoiding this bias by ensuring that centroids are initialized from different regions of the dataset.
2. **Increasing Robustness:** Random initialization increases the robustness of the algorithm. By starting from different initial centroid positions in each run, K-means clustering can explore different possible cluster configurations. This helps in mitigating the impact of local optima, where the algorithm gets stuck in suboptimal solutions due to the initial centroid positions.



```python
def kMeans_init_centroids(X, K):
    """
    This function initializes K centroids that are to be
    used in K-Means on the dataset X

    Args:
        X (ndarray): Data points
        K (int):     number of centroids/clusters

    Returns:
        centroids (ndarray): Initialized centroids
    """

    # Randomly reorder the indices of examples
    randidx = np.random.permutation(X.shape[0])

    # Take the first K examples as centroids
    centroids = X[randidx[:K]]

    return centroids
```

- The code first randomly shuffles the indices of the examples (using np.random.permutation()).
- Then, it selects the first K examples based on the random permutation of the indices.

    - This allows the examples to be selected at random without the risk of selecting the same example twice.

Image Compression

In a straightforward 24-bit color representation of an image, each pixel is represented as three 8-bit unsigned integers (ranging from 0 to 255) that specify the red, green, and blue intensity values. This encoding is often referred to as the RGB encoding. The image contains thousands of colours, and the goal is to reduce the number of colours to 16 colours.

By making this reduction, it becomes possible to represent (compress) the photo in an efficient way.

Specifically, only the RGB values of the 16 selected colours need to be stored, and for each pixel in the image, only the index of the colour at that location needs to be stored (where only 4 bits are necessary to represent 16 possibilities).

Concretely, every pixel in the original image will be treated as a data example, and the K-means algorithm will be used to find the 16 colours that best group (cluster) the pixels in the 3-dimensional RGB space.

Once the cluster centroids on the image have been computed, the 16 colours will then be used to replace the pixels in the original image.

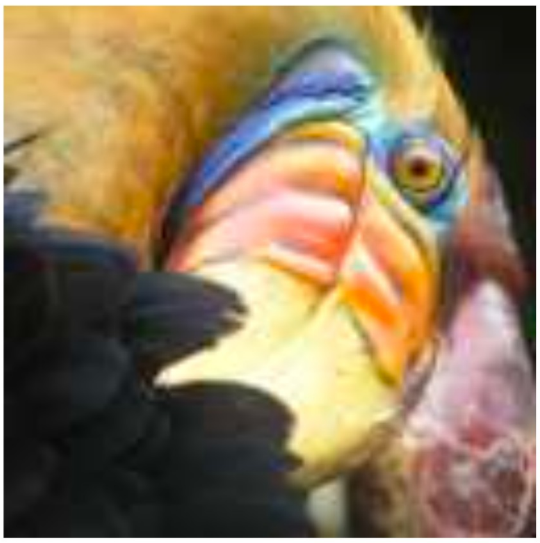
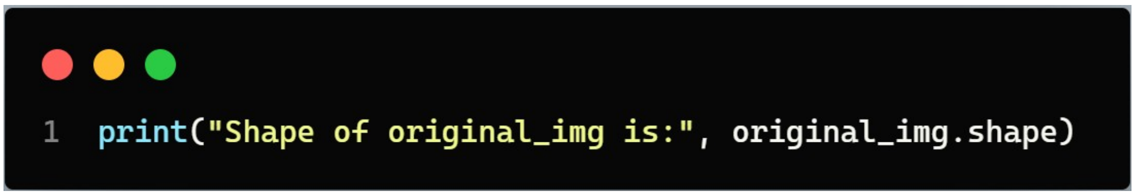
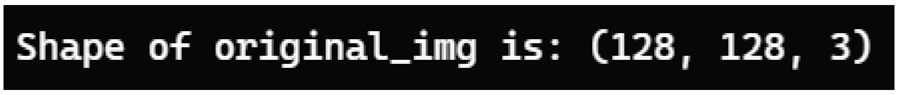


Figure 2: The original 128x128 image.

Code



```python
print("Shape of original_img is:", original_img.shape)
```



```
Shape of original_img is: (128, 128, 3)
```

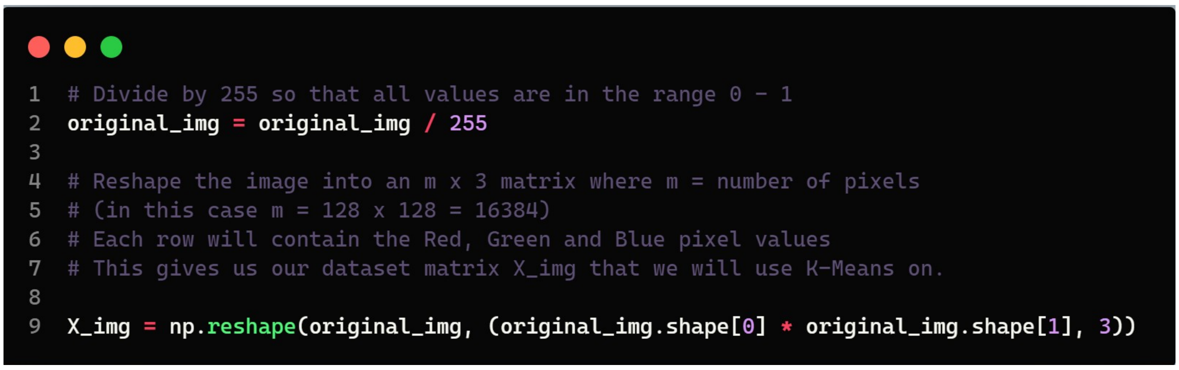
As observed, this creates a three-dimensional matrix original_img where:

- the first two indices identify a pixel position, and
    - the third index represents red, green, or blue.

For instance, original_img[50, 33, 2] gives the blue intensity of the pixel at row 50 and column 33.

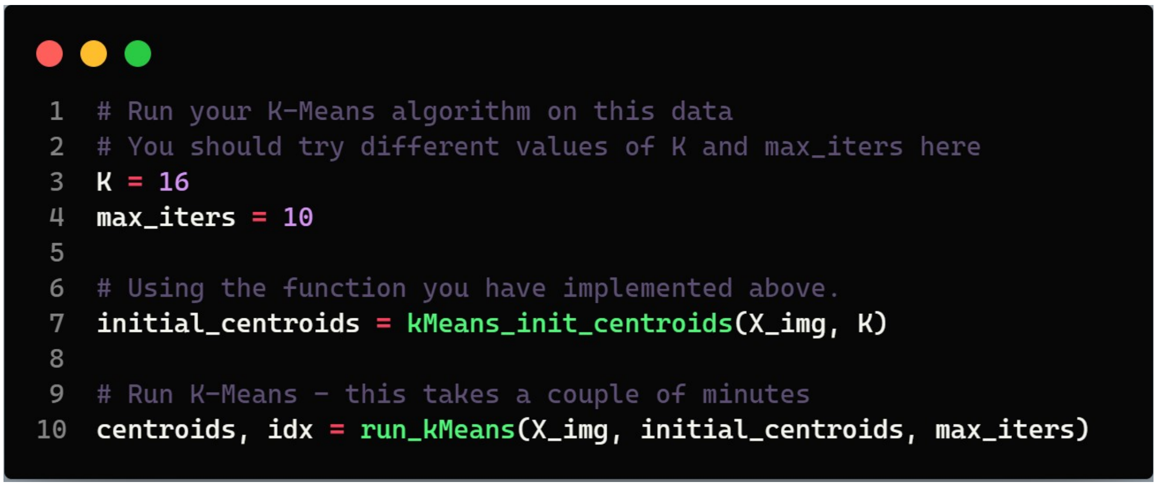
Processing data:

To call the run k Means, it's necessary to first transform the matrix original_img into a two-dimensional matrix. The following code reshapes the matrix original_img to create an m*3 matrix of pixel colours (where m=16384 = 128*128).

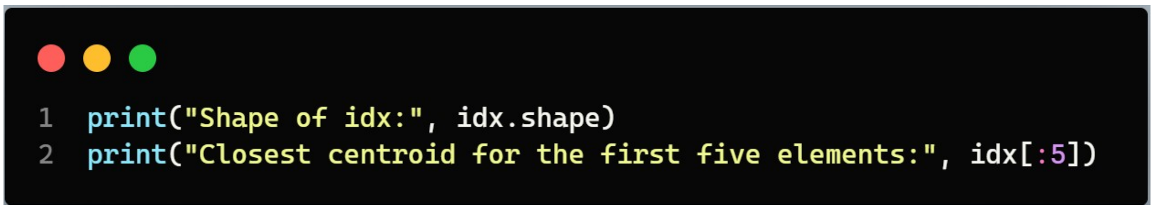


```python
# Divide by 255 so that all values are in the range 0 - 1
original_img = original_img / 255

# 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))
```

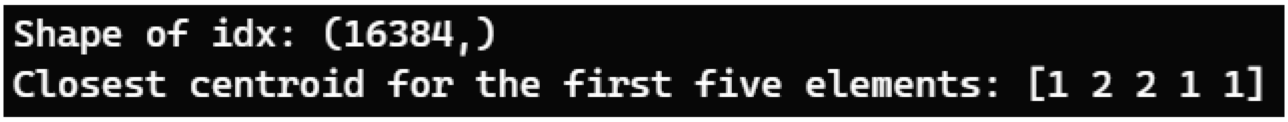
K-Means on Image Pixels:



```python
# Run your K-Means algorithm on this data
# You should try different values of K and max_iters here
K = 16
max_iters = 10

# Using the function you have implemented above.
initial_centroids = kMeans_init_centroids(X_img, K)

# Run K-Means - this takes a couple of minutes
centroids, idx = run_kMeans(X_img, initial_centroids, max_iters)
```



```python
print("Shape of idx:", idx.shape)
print("Closest centroid for the first five elements:", idx[:5])
```

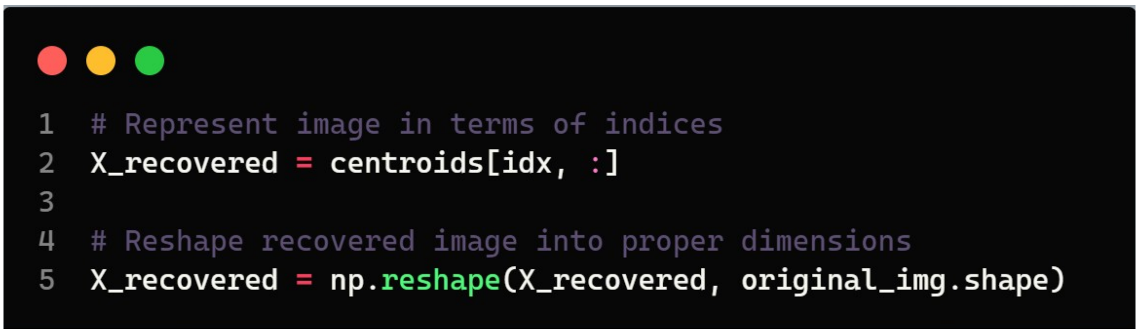


```
Shape of idx: (16384,)
Closest centroid for the first five elements: [1 2 2 1 1]
```

After finding the top $K=16$ colours to represent the image, each pixel position can be assigned to its closest centroid using the find_closest_centroids function. This enables the representation of the original image using the centroid assignments of each pixel.

It's noteworthy that there has been a significant reduction in the number of bits required to describe the image.

- The original image required 24 bits for each one of the 128 * 128 pixel locations, resulting in a total size of 128 * 128 * 24 = 393,216 bits.
- The new representation requires some overhead storage in the form of a dictionary of 16 colours, each of which requires 24 bits. However, the image itself then only requires 4 bits per pixel location.
- The final number of bits used is therefore 16 * 24 + 128 * 128 * 4 = 65,920 bits, which corresponds to compressing the original image by about a factor of 6.

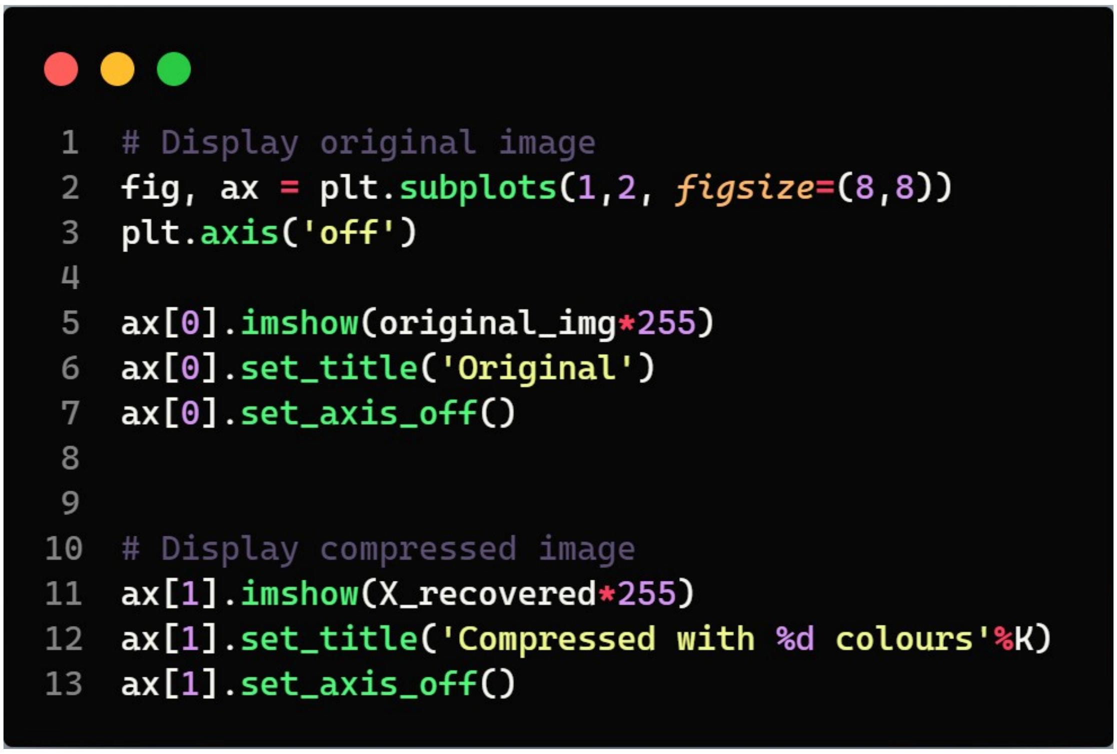


```python
# Represent image in terms of indices
X_recovered = centroids[idx, :]

# Reshape recovered image into proper dimensions
X_recovered = np.reshape(X_recovered, original_img.shape)
```

Results and Conclusion

Finally, we can view the effects of the compression by reconstructing the image based only on the centroid assignments.

- Specifically, you can replace each pixel location with the mean of the centroid assigned to it.
- Even though the resulting image retains most of the characteristics of the original, we shall also see some compression artefacts.



```python
# 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)
ax[1].set_axis_off()
```