HO CHI MINH CITY – UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

APPLIED MATHEMATICS AND STATISTICS PROJECT

COLOR COMPRESSION USING K-MEANS CLUSTERING ALGORITHM

Instructors:

Prof. Vũ Quốc Hoàng

TA. Phan Thị Phương Uyên

TA. Nguyễn Văn Quang Huy

TA. Nguyễn Ngọc Toàn

Student:

22127218 – Văn Bá Đức Kiên

Color Compression using K-Means

22CLC10 - 22127218 - Văn Bá Đức Kiên

Contents

1) Project overview

In this project in the Applied Mathematics and Statistics, I learn and implement the K-means clustering algorithm in Python in order to reduce the number of colors used in a picture. This report will explain my ideas and understanding of the K-means clustering algorithm and how I will use it to implement the desired product.

2) Implementation idea

2.1) Initial thoughts

There are many ways to represent colors of an image, but for the scope of this project, I will use RGB colors only. A pixel is represented by three main colors red, green and blue (RGB) and three integers between 0 and 255 indicating the intensity of those three main colors. Because of this, we can consider a pixel in the image as a vector in the three dimensional vector space $\mathbb{R}^3$.

To reduce the number of colors in an image, the idea is to group the pixels that have similar color together, then elect a representative color for that group. The number of representative colors for their own group is the number of colors of that image after being compressed. To execute this idea, K-means clustering algorithm is one great choice.

2.2) K-means clustering

Based on the ideas that I have researched from [1], I made some adjustments to have an easier time implementing this algorithm in Python using NumPy:

Let X denote the array of n pixels in our image, because each pixel can be represented by a vector of $\mathbb{R}^3$ we have:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^{n\times 3} \tag{1}$$

Let k is the number of clusters that we want to divide the pixels into, then we also have a matrix that represents all the centroids (centers) of each clusters:

$$M = \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_k \end{pmatrix} \in \mathbb{R}^{k\times 3} \tag{2}$$

Let Y denote the label vector, each y_i in Y is the label for the pixel x_i in X.

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \in \mathbb{R}^{n} \tag{3}$$

Because we will replace all the pixels in the same group with their centroids, the error of a pixel x_i in a cluster k will be $(x_i - m_k)$. Because we want to error to be as small as possible, we will find a way to make $\|x_i - m_k\|^2$ achive the smallest value possible for every pixel.

To achive this goal, I will follow the K-means clustering algorithm [1]. Based on the algorithm, we have 4 steps to do:

1. Set M to random values.
2. Assign every pixels to their own nearest centroid which means updating the matrix Y such that:

$$y_i = \underset{k}{\operatorname{argmin}} \|x_i - m_k\| \tag{4}$$

3. Update new centroids for every clusters, which means updating matrix M such that:

$$S_i = \{x_j \mid x_j \in X \land y_j = i\} \tag{5}$$

$$m_i = \frac{1}{|S_i|} \sum_{s_k \in S_i} s_k \tag{6}$$

4. Repeat from step 2 until the value of the centroids converged (matrix M does not change (or very close if we do not round the value after taking average to an integer) after an iteration) or the maximum number of iterations is reached.

3) Detailed implementation description

3.1) Environments

- Operating system: Windows 11 - 23H2.
- Python v3.12.3
  - **NumPy** v1.26.4 (for matrix creation, manipulation and calculation)
  - **pillow** v10.3.0 (for reading and saving images)
  - **matplotlib** v3.9.0 (for displaying images in Jupyter Notebook)

3.2) Implementations

3.2.1) Reading image

The act of reading image is implemented in the **read_img()** function:

- **Parameters: img_path** (string): path of the image.
- **Returns**: 2D Image as a **numpy.ndarray** with **shape = (height, width, 3)**

Code explanation:

- Simply use the **PIL.Image.open()** [2] with the image path as the parameter and PIL will open the image for you as a PIL.Image.Image object.
- Because different images can use different ways to represent colors but we are working with RGB colors only, so we use the **convert()** function of the PIL.Image.Image class [3] with 'RGB' as the parameter on the image object from the previous step to convert the image's mode to RGB.

- Since we will do the calculations using NumPy and show the image in Jupyter Notebook using matplotlib, I decided that converting the image object to a numpy.ndarray before returning the image so that other functions can use the return value right away instead of having to convert it to numpy.ndarray many times later. The converting is performed by the **numpy.asarray()** function [4] with the image object from the previous step.

All those step can be accomplished using one line of code in Python:

```
return numpy.asarray(Image.open(img_path).convert('RGB'))
```

3.2.2) Showing image in Jupyter Notebook

The act of showing image in Jupyter Notebook is implemented in the **show_img()** function:

- **Parameters: img_2d** (numpy.ndarray with shape = (height, width, 3)): the 2D image.
- **Returns**: none

Code explanation:

- Simply use the **matplotlib.pyplot.imshow()** [5] function with the appropriate numpy.ndarray as the parameter to show the image in Jupyter Notebook.

3.2.3) Saving image

The act of saving image in Jupyter Notebook is implemented in the **save_img()** function:

- **Parameters:**
  - **img_2d** (numpy.ndarray with shape = (height, width, 3)): the 2D image.
  - **img_path** (string): path of the image (including image filename and extension).
- **Returns**: none

Code explanation:

- Simple convert the image in the form of numpy.ndarray to a PIL.Image.Image object using the **PIL.Image.fromarray()** function [4] with the numpy.ndarray and named argument mode = 'RGB'.
- Then use the **save()** function of the PIL.Image.Image class [6] to save the image to the desired location.

3.2.4) Converting 2D image to 1D image

The act of converting 2D image to 1D image is implemented in the **convert_img_to_1d()** function:

- **Parameters: img_2d** (numpy.ndarray with shape = (height, width, 3)): the 2D image.
- **Returns**: the 1D image as **numpy.ndarray** with **shape = (height x width, 3)**

Code explanation:

- Simply use the **numpy.reshape()** [7] function with the 2D image and the tuple (-1, 3) as the parameter. The tuple (-1, 3) tell the function that the second dimension length is 3 and the length of the first dimension can be automatically inferred by the function based on NumPy documentation's parameter description. [7]

3.2.5) Finding centroids and labeling pixels using K-means

Now come the interesting part. The act of finding centroids and labeling pixels using the K-means clustering algorithm is implemented in the **kmeans()** function:

- **Parameters:**
  - **img_1d** (numpy.ndarray with shape = (height x width, 3)): the 1D image.
  - **k_clusters** (int): the number of clusters.
  - **max_iter** (int): the maximum number of iterations allowed.
  - **init_centroids** (string): the value of the string is either "random" or "in_pixels".
- **Returns**:
  - **centroids** (**numpy.ndarray** with **shape = (k_clusters, 3)**): stores the color centroids for each cluster
  - **labels** (**numpy.ndarray** with **shape = (height x width,)**): stores the cluster label for each pixel in the image

Code explanation: The following code follows the logic from the in Section 2.2.

- The first step is to **initialize the centroids**: We initialize the centroids based on the matrix defined in Equation 2, which is an ndarray with shape = (k_clusters, 3). Let "centroids" is the name of this ndarray.
  - If init_centroids is "random": use the **numpy.random.randint()** function [8] with 256 as the first argument (the function will random values from 0 to 255) and the tuple (k_-clusters, 3) as the second argument as the shape of the ndarray generated by the function.

    ```
    if (init_centroids == 'random'):
        centroids = np.random.randint(256, size=(k_clusters, 3))
    ```

  - If init_centroids is "in_pixels": use the **numpy.unique()** [9] function with the img_1d and axis=0 as arguments which will return sorted unique elements of the img_1d array according to the verticle axis to get unique vectors from the img_1d. After that we generates random indices by using **numpy.random.choice()** function [10] with (size of the set from the previous step, k_clusters, replace='false') as the argument to generate random distinguished indices. Then we assign the elements from set of unique pixels using the generated indices to the centroids list. This will make sure that there is no two identical centroids in the ndarray.

    ```
    elif (init_centroids == 'in_pixels'):
        row_set = np.array(np.unique(img_1d, axis=0))
        centroids = row_set[np.random.choice(row_set.shape[0], k_clusters,
    replace=False)]
    ```

- The next step is to **asign labels** for each pixels based on the logic described in Section 2.2. This involves the process of calculating the norm (Euclid distance) between a pixel and every other centroids then assigning the index of the nearest centroid as the label of that pixel. This can be easily implemented using Python loops but Python is notoriously known for its slow loop performance. Because of this, calculating everything in just NumPy will increase the performance significantly since numpy.ndarray is implemented in C array and many of NumPy's function are coded in C which is way faster than Python. This is how I implemented the asign labels process:

- **Calculating the differences** using NumPy broadcasting[1] [11]. My goal is to create an numpy.ndarray name diff that diff[i][j] $\in \mathbb{R}^3$ is the differences between centroids[i] and img_1d[j]. Then the diff numpy.ndarray will have shape = (k_clusters, n, 3). img_1d has shape = (n, 3). Based on the broadcasting rule[2] , we can deduce that the shape of centroids must be (k_clusters, 1, 3) to be able to use broadcasting. Because of this, using the reshape() function [7] with (k_clusters, 1, 3) as the argument on the centroids ndarray, we can now use broadcasting to create the diff numpy.ndarray.

```
diff = np.array(img_1d - centroids.reshape((k_clusters, 1, 3)))
```

- **Calculating the distances** from using the differences numpy.ndarray from the previous step. Because diff[i][j] $\in \mathbb{R}^3$ so the distance of diff[i][j] is its norm. To calculate the norm of diff[i][j], you need to square all the elements in diff[i][j], then take their sum and then take the square root of that sum. Thanks to NumPy ndarray feature, we can just use those operations on the numpy.ndarray as if we are using those operations on a number variable and it will automatically apply that to every element in the array. The numpy.sum() function has an extra argument axis=2 because we want the sum of the elements in diff[i][j] only. The resulting ndarray name dist will have shape = (k_clusters, n) and dist[i][j] is the distance between centroids[i] and img_1d[j].

```
dist = np.array(np.sqrt(np.sum(diff**2, axis=2)))
```

- **Assigning the labels** from the previous dist ndarray using the numpy.argmin() function [12]. We just need to apply the numpy.argmin() function to the dist array along the k_clusters axis to get the indices of the corresponding nearest centroid of each pixel.

```
labels = np.array(np.argmin(dist, axis=0))
```

- The next step after assigning the labels is to get the new centroid of each cluster. The idea for this part is mentioned in Equation 5 in Section 2.2.

  - First, we get the set of the cluster indices from the labels and counts how many pixels got assigned to that cluster using the numpy.unique() function [9] with the labels as argument and set the named argument return_counts=True to get the counts of each element.

  ```
  cluster_ids, counts = np.unique(labels, return_counts=True)
  ```

  - Cast the results from previous step to a Python dictionary using zip.

  ```
  labels_cnt = dict(zip(cluster_ids, counts))
  ```

  - Finally, get the values of the new centroids by getting the sum of the pixels in each cluster then divide that sum by the number of pixels in that cluster. There maybe some old centroids that did not get any pixels into their group. We have two ways to handle this, the first one is to just leave the "lonely" centroids there and hope that they will get some pixels in their cluster in later iteration, the second way is to generate a new random value for those centroids. I tested both options and found out that the random approach seems to make the centroids converge more quickly so I decided to choose the random approach. To implement this, I created a new_centroid numpy.ndarray that had its values randomized then for every cluster that has at least one element, update that cluster's new centroid, the ones that are not updated will have randomized values from the step before.

```python
    # f4 = float32
    new_centroids = np.random.randint(256, size=(k_clusters, 3)).astype('f4')
        for i in labels_cnt:
            new_centroids[i] = np.sum(img_1d[labels == i], axis=0) / labels_cnt[i]
```

- The next step is simply comparing the new centroids with the old ones. I allow the centroids values to be real number to make the centroids selection more accurate (For example, if the centroids values are integer only, a difference of 0.5 between the new and old centroids is neglected but because we are running through many iterations, that small number may lead to some more changes in the centroids values and labeling of pixels). I set the limit to stop the algorithm to 1/max_iter so that even if the differences between the old and new centroids is 1/max_iter, it takes max_iter iterations to change the color of that centroid (because we will cast the centroids values to unsigned integer 8bit so that PIL can save the image). Just in case the user set the max_iter too high, I set 0.01 to be the lower bound for the limit.

```python
limit = max(1e-2, 1/max_iter)
if (np.all(np.abs(new_centroids - centroids) < limit)):
    break
```

The algorithm will do the assigning labels, updating centroids and performing convergence test until the centroids converged or when max iteration is reached. At that time, the function will return the centroids and labels. I use the numpy.round() and numpy.ndarray.astype() functions to round every real values in the centroids to its nearest integer.

3.2.6) Generating image based on resulting centroids and labels

The act of generating 2D image based on resulting centroids and labels is implemented in the **generate_2d_img()** function:

- **Parameters:**
  - **img_2d_shape** (tuple (height, width, 3)): Shape of image.
  - **centroids** (numpy.ndarray with shape=(k_clusters, 3)): Store color centroids.
  - **labels** (np.ndarray with shape=(height x width)): Store label for pixels (cluster's index on which the pixel belongs).
- **Returns**: 2D Image as a **numpy.ndarray** with **shape = (height, width, 3)**

Code explanation:

- First, we will reconstruct the 1D array from the centroids and labels using list comprehension. For each pixel[i], the pixel is reconstruted by centroids[labels[i]], this means set pixel[i] to the value of the centroid of index labels[i] since labels[i] is the index of the centroid of the cluster that pixel[i] is in. Then use numpy.ndarray.reshape() to convert the 1D array into the 2D array. Finally, use numpy.ndarray.astype() with 'uint8' (8 bits unsigned integer) so that PIL can save the image and matplotlib can show the image in Jupyter Notebook.

[1]Broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations, the smaller array is "broadcast" across the larger array so that they have compatible shapes.

[2]When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimension and works its way left. Two dimensions are compatible when they are equal, or one of them is 1.

```
  return np.array(
    [centroids[labels[i]] for i in range(len(labels))]
  ).reshape(img_2d_shape).astype('uint8')
```

4) Result evaluation

Thanks Glenn Ostle, the winner of the 2023 edition of The Nature Photography Contest for the picture Sea Lion in Los Islotes[3].

Figure 1: Sea Lion in Los Islotes, by Glenn Ostle.

[3]https://thenaturephotocontest.com/photo-contest-winners-2023

4.1) Clustering evaluation

For all compressed images in this section, I set the max number of iterations to 100 when running the K-means clustering algorithm.

From the pictures below, we can see that the 7, 5 and 3 colors compressed images still maintain the content of the original image, all the dominant color schemes are present. We can clearly see that the main three colors are black, dark cyan (mixed between blue and green) and grey. However, the 5 and 7 colors compressed image provides more details and is more vibrant compare to the 3 colors counterpart as they can now display more shades of grey and cyan.

I also have the centroids of these images sorted (to compare them easier) and print them out. As we can see, the value of the centroids between two images that have the same colors limit but different centroids initialization.

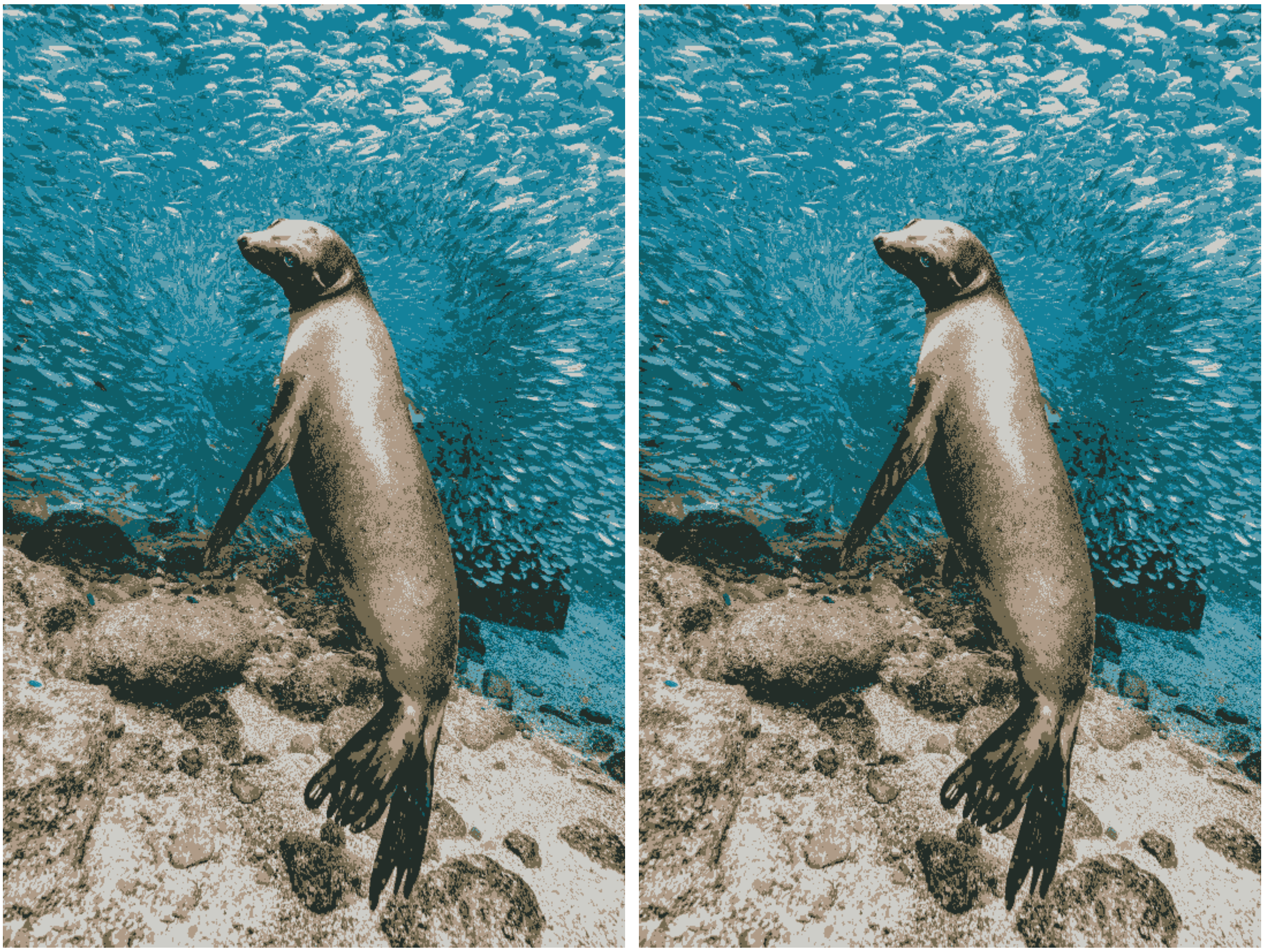

Figure 2: Compressed images using 7 colors with "random" and "in pixels" centroids initialization consecutively.

```
7 colors with random centroids init:
[[ 17.418758  48.353798  43.444195]
 [ 24.461733  97.39144   84.015785]
 [ 39.305855 104.58567  105.487076]
 [ 96.783104 132.39235  134.63005 ]
 [116.458916 157.07967  155.01537 ]
 [176.10838  163.06201  178.48909 ]
 [209.34653  208.96506  200.59639 ]]
```

```
7 colors with in-pixel centroids init:
[[ 17.416134  48.37236   43.447628]
 [ 24.493135  97.392265  84.066055]
 [ 39.337082 104.635284 105.49705 ]
 [ 96.82667  132.40602  134.682   ]
 [116.52293  157.1258   155.01648 ]
 [176.14468  163.08919  178.52672 ]
 [209.37463  208.9908   200.61964 ]]
```

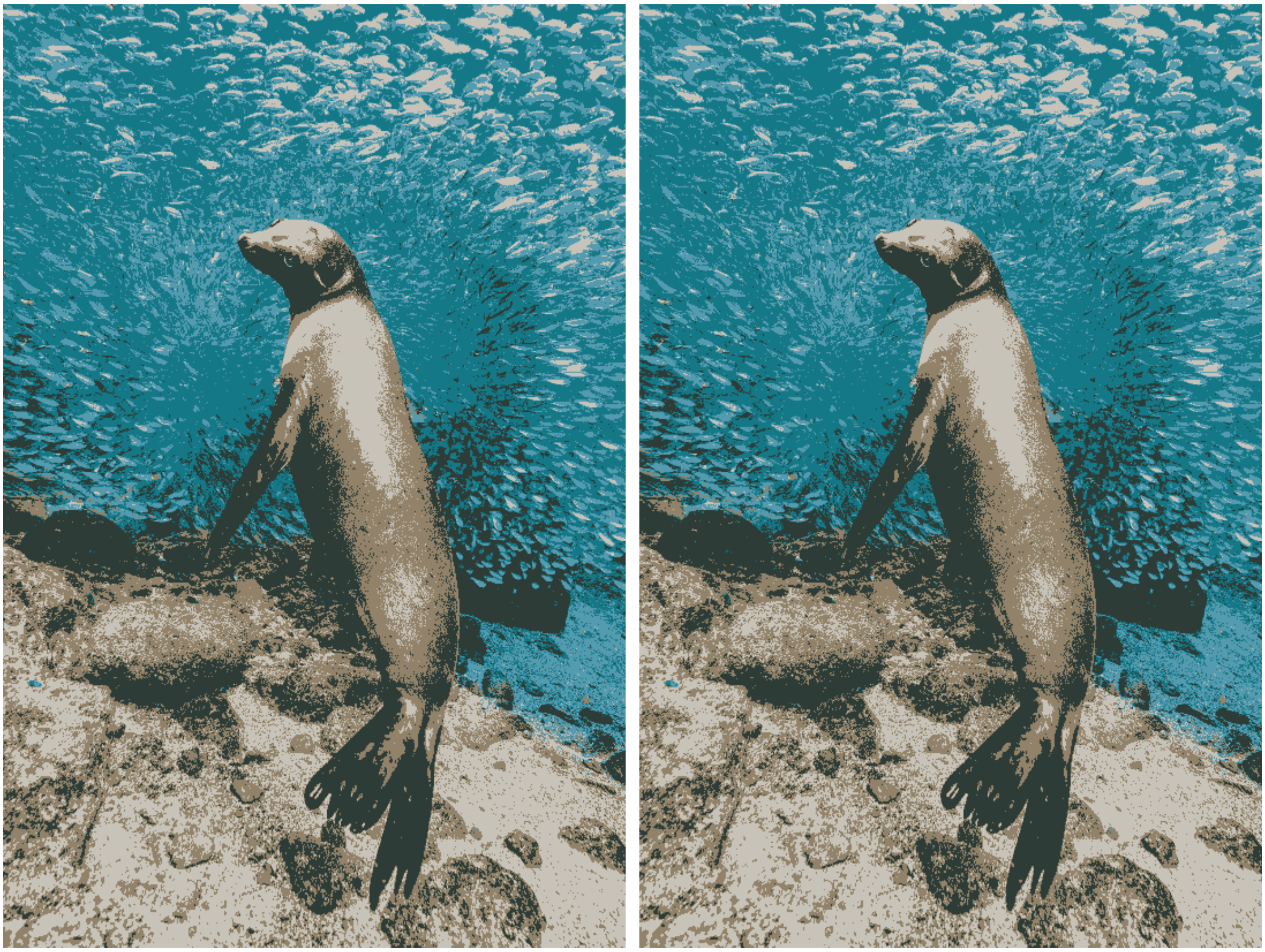

Figure 3: Compressed images using 5 colors with "random" and "in pixels" centroids initialization consecutively.

```
5 colors with random centroids init:
[[ 19.836847  61.33801   55.343292]
 [ 45.35936  120.51242  109.00889 ]
 [ 87.50553  131.45445  138.62708 ]
 [149.13022  159.38922  175.13623 ]
 [202.90372  199.38802  188.49161 ]]
```

```
5 colors with in-pixel centroids init:
[[ 19.856285  61.346443  55.367355]
 [ 45.31119  120.53973  108.96693 ]
 [ 87.551315 131.41698  138.65804 ]
 [149.09656  159.40764  175.15541 ]
 [202.89476  199.36958  188.46729 ]]
```

Figure 4: Compressed images using 3 colors with "random" and "in pixels" centroids initialization consecutively.

```
3 colors with random centroids init:
[[ 36.44004  74.16158  65.80936]
 [ 59.15815 131.86673 149.90112]
 [178.58443 177.5324  166.19737]]
```

```
3 colors with in-pixel centroids init:
[[ 36.406727  74.13963   65.774475]
 [ 59.19278  131.83441  149.86252 ]
 [178.55423  177.52689  166.20197 ]]
```

4.2) Image size evaluation

The image size of an image after being compressed using this method will depend on the image type (extension) because each image type such as jpg or png have different methods of compressing the images. I will only cover the image size evaluation of png type images only.

Based on the Portable Network Graphics (PNG) Specification [13], if the number of distinct pixel values is 256 or less and the alpha channel is absent then the alternative indexed-colour representation, achieved through an indexing transformation, may be more efficient for encoding.

In the indexed-colour representation, each pixel is replaced by an index into a palette. The palette is a list of entries each containing three 8-bit samples (red, green, blue). If an alpha channel is present, there is also a parallel table of 8-bit alpha samples, called the alpha table but in our case, we did not use the alpha channel.

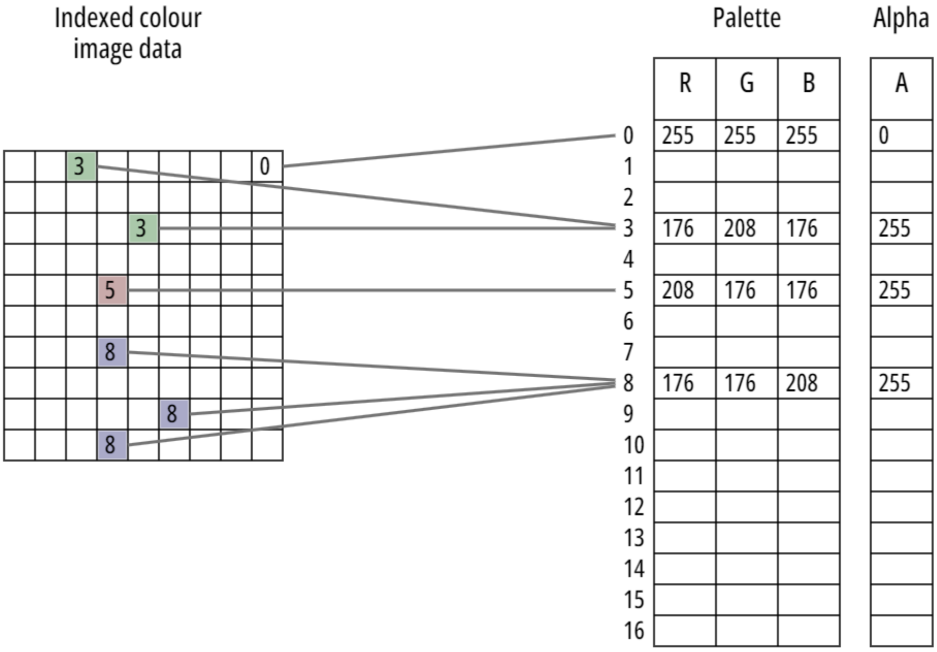

Figure 5: Indexed-color image.

Because of this reason, reducing the number of colors in a image may reduce the file size of a png image (still depends on the encoder). In case of PIL in the Pillow package, I have observed a large decrease in image size when using fewer colors.

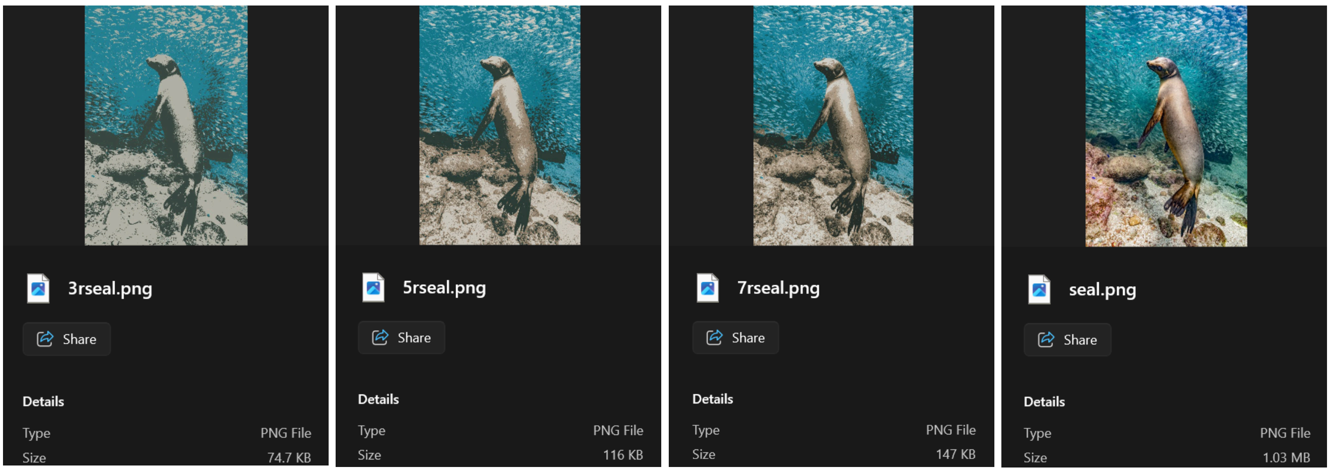

Figure 6: in order: 3, 5, 7 colors compressed images with uncompressed image.

5) References

All of these references are available on 2024/06/15, the contents and availability of the references may not be the same after this day.

[1] Chapter 20.1: K-means clustering, in the textbook Information Theory, Inference and Learning Algorithms by David MacKay".

[5] Showing image in Jupyter Notebook using the matplotlib.pyplot.imshow() function in the Matplotlib documentation.

[13] 4.4.2 Indexing from the Portable Network Graphics (PNG) Specification (Third Edition).

5.1) References from the Pillow documentation

[2] Pillow PIL.Image.open() function.

[3] Pillow convert() function of the PIL.Image.Image class.

[4] Converting PIL.Image.Image object to numpy.ndarray and vice versa.

[6] Saving image using the save() function of the PIL.Image.Image class.

5.2) References from the NumPy documentation

[7] Reshaping an ndarray with the numpy.reshape() function.

[8] Generating random integer values for numpy.ndarray using the numpy.random. randint() function.

[9] Getting the sorted unique elements of an array and their counts using the numpy.unique() function.

[10] Generating a random sample from a given 1D array using the numpy.random.choice() function.

[11] Guide to NumPy broadcasting.

[12] Getting indices of the minimum values along an axis using numpy.argmin() function.