Applied Mathematics and Statistics

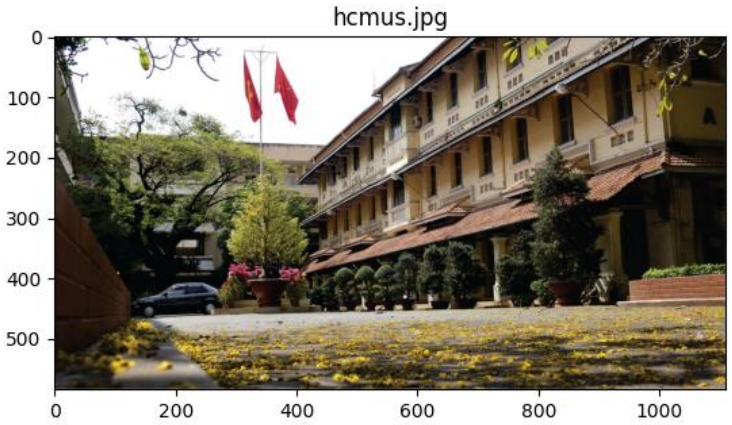
Color Compression

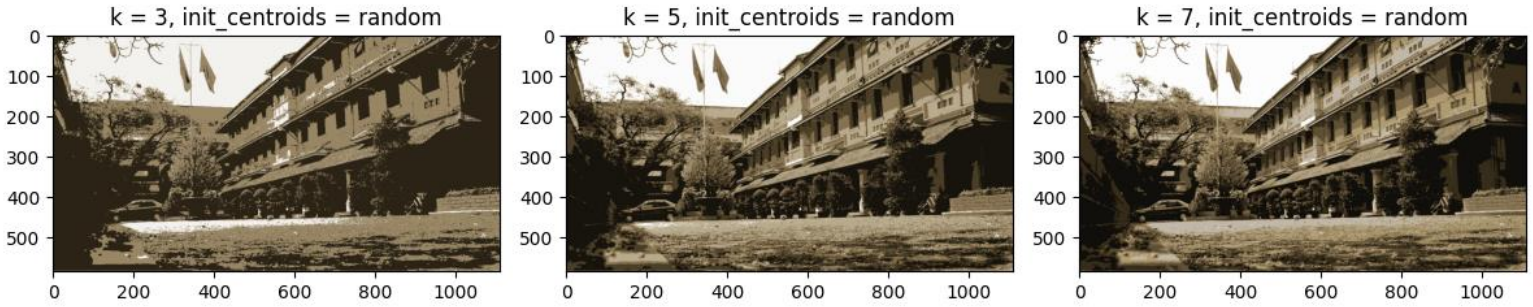
Project 01

21CLC05 - University of Science - VNUHCM

21127135 – Diệp Hữu Phúc

✦





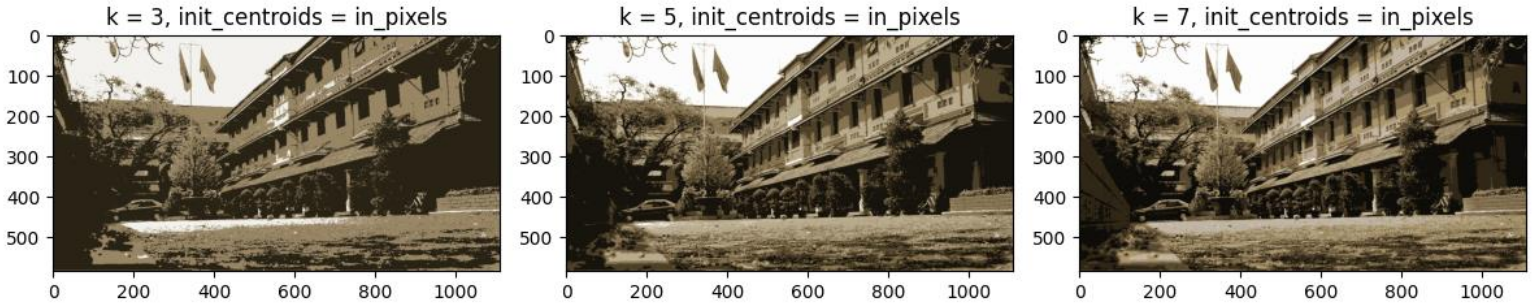


Table of contents

Table of Contents

0. Report

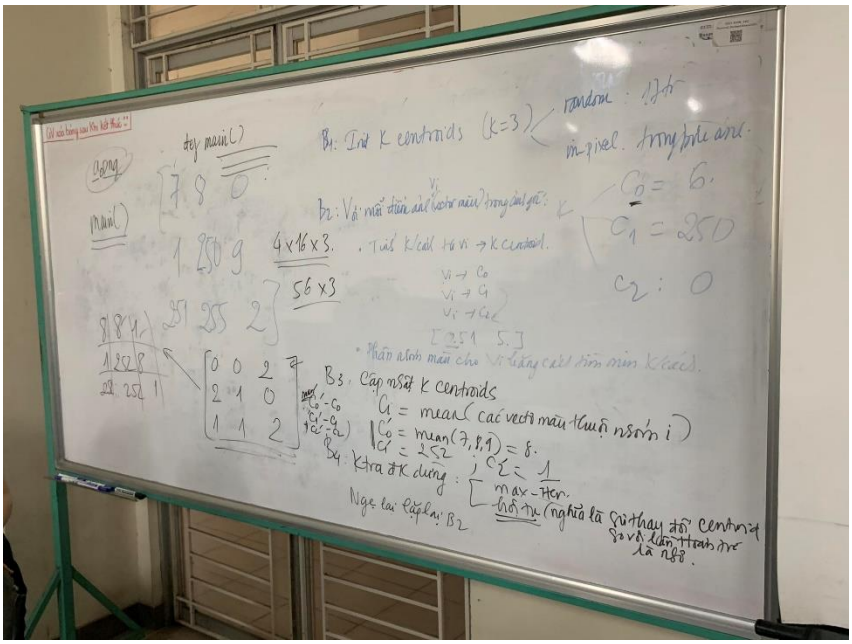
This will be a long read so please make use of bookmarks for ease of navigation.

Priority	No.	Task	Status
Required	1	Use only Numpy, PIL and matplotlib	Done
	2	Implement K-means from scratch	Done
	3	Handle inputs and outputs	Done
	4	Allow saving as **.png** and **.pdf**	Done
	5	IPython Notebook	Done
	6	Report and documentation • Idea and implementation • Results with k_clusters = {3 , 5, 7} • Comments • References	Done

Below is the link to my Github repository,

- https://github.com/kru01/ColorCompression_K-means

1. Idea



After extensively studying the pseudocode provided by Lecturer Phan Thi Phuong Uyen along with a few other supplements, namely,

- https://en.wikipedia.org/wiki/K-means_clustering
- https://youtu.be/4b5d3muPQmA

- https://towardsdatascience.com/create-your-own-k-means-clustering-algorithm-in-python-d7d4c9077670

My outline for the **K-means clustering algorithm** is as follows,

1. Initialize all the centroids with the desired number of clusters **k_clusters** and the given method **init_centroids**.
2. Loop until the iteration limit **max_iter** is reached. **With each loop, steps from 3 to 7 are sequentially performed**.
3. Calculate the Euclidean distance between every centroid and each pixel in the image.
4. Assign each pixel to share the same color as the centroid nearest to it, i.e., use the index of the cluster belonging to said centroid to label the pixel.
5. Keep a copy of the current centroids as **prev_centroids**.
6. Update each centroid by calculating the mean of all the pixels in the cluster associated with that centroid.
    a. It cannot be guaranteed that there will always be at least one pixel in each cluster. In other words, the possibility for empty clusters exists.
    b. If a cluster is empty, it implies there is no pixel having the same color as the centroid of that cluster, and the mean cannot be computed. Thus, the centroid managing that cluster should be discarded.
    c. If the number of centroids is less than the number of clusters, update k_clusters to the former.
7. Check for convergence by seeing whether the difference between **prev_centroids** and the updated centroids falls within a pre-determined threshold. If the condition is satisfied, break out of the loop.
8. Return the centroids and the labels for all the pixels in the image.

For the reasons stated in step 6, we will not always get the number of clusters *k_clusters* we actually want in the output. This phenomenon is much more prevalent when dealing with *init_centroids=random*.

2. Implementation

```
def kmeans(img_1d:np.ndarray, k_clusters:int, max_iter:int,
init_centroids:str='random'):
```

The function definition is kept identical to the template provided by Lecturer Phuong Uyen, ➢ **denotes parameters** and **o denotes outputs, i.e., returns**.

- **img_1d** – $np.ndarray$ with $shape = (height * width, num\_channels)$ – The original image reshaped beforehand into an 1d array.
- **k_clusters and max_iter** – Integers representing the number of clusters and the iteration limit respectively.
- **init_centroids** – String describing how to initialize centroids.
    - random – Each centroid has c channels. The value of each channel is randomized from the range $[0,255]$.

        - in_pixels – Each centroid is a pixel randomized from the original image.
    - **centroids** – $np.ndarray$ with $shape = (k\_clusters, num\_channels)$ – Array storing resulting centroids.
    - **labels** – $np.ndarray$ with $shape = (height * width, num\_channels)$ – Array storing label for every pixel in the image, i.e., the cluster's index to which the pixel belongs.

1. Initialize all the centroids

```
centroids = np.zeros((k_clusters, img_1d.shape[1]))

if init_centroids == 'random':
    while len(np.unique(centroids, axis=0)) != k_clusters:
        centroids = np.random.choice(256, (k_clusters, img_1d.shape[1]))
else: centroids = img_1d[np.random.choice(len(img_1d), k_clusters, False)]
```

random – We want the randomized centroids to be reasonably unique for more diverse coloring. Therefore, centroids will be continuously generated until we have at least **k_clusters** distinct colors. However, one caveat to my method is that the same colors but of slightly different shades will also be allowed.

in_pixels – Utilizing **numpy** trivializes this process. As described before, the centroids will be **k_clusters** pixels taken randomly from the original image while ensuring that we don't take the same pixel multiple times. Not allowing duplicates is as simple as setting $replace = False$.

2. Calculate Euclidean distances and labeling

Broadcasting

The general formula to compute the Euclidean distance for points given by Cartesian coordinates in n-dimensional Euclidean space is,

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}$$

In practice, a traditional implementation would involve at least one kind of loop to iterate over every single value in each vector. Fortunately for us, with the concept of **numpy broadcasting**, we can neatly package everything into just one line of code.

```
distances = np.sqrt(((img_1d - centroids[:, None])**2).sum(axis=2))
```

To attain a clear understanding of what is happening, we must first be comfortable with **broadcasting**. Here are the resources I used to achieve this,

- https://numpy.org/doc/stable/user/basics.broadcasting.html
- https://youtu.be/oG1t3qlzq14

During execution, **img_1d** should have a shape of $(x, num\_channels)$ while **centroids** possesses $(y, num\_channels)$. Clearly, these are incompatible with each other, and **Python** would yield a

ValueError. Here is where **broadcasting** comes into place, there are 2 main approaches we can take to solving this problem.

Method/Form	A	B
1	$img\_1d[:, None] - centroids[None, :]$	$img\_1d[:, None] - centroids$
2	$img\_1d[None, :] - centroids[:, None]$	$img\_1d - centroids[:, None]$

- $np.newaxis$ is just an alias for $None$, they can be used interchangeably.
- The broadcasting operation does not modify the array itself, it returns a new array with the new shape. In this case, both the original img_1d and $centroids$ will retained their shapes. If you want to modify an array in-place, the code would be $arr = arr[:, None]$.

Method 1 – Appending a new axis to **img_1d** – After the process, **img_1d** would have the shape $(x, 1, num\_channels)$. Then to make **centroids** compatible, we prepend a new axis to it resulting in $(1, y, num\_channels)$. All of that is plainly described in **Form A**. Furthermore, **numpy** will implicitly handle the prepending new axes for us if the shapes are already compatible up to that point. Hence, we have **Form B**. Applying the subtract operator, in this case, would give us a new array **arr** with shape $(x, y, num\_channels)$.

- $arr[i]$ is the result of $img\_1d[i] - centroids$.
- $arr[i][j]$ is the result of $img\_1d[i] - centroids[j]$.
- $arr[i][j][k]$ is the channel k of $arr[i][j]$.

Method 2 – Appending a new axis to **centroids** – After the process, **centroids** would have the shape $(y, 1, num\_channels)$. Next, we follow virtually the same logic as before to arrive at $(1, x, num\_channels)$ for **img_1d**, and **Form A** then shortening it to **Form B**. The resulting array **arr** would be of shape $(y, x, num\_channels)$.

- $arr[i]$ is the result of $img\_1d - centroids[i]$.
- $arr[i][j]$ is the result of $img\_1d[j] - centroids[i]$.
- $arr[i][j][k]$ is the channel k of $arr[i][j]$.

- One little tidbit, if we take **appending** literally then a shape of $(2,3)$ should evidently become $(2,3,1)$, it was something that also confused me when I first thought about it. However, if we imagine the **3-dimension** as a big package of values and itself is also a value. Then the shape $(2,3)$ really is just $(2,)$. For illustration, $[[4,5,6],[7,8,9]]$ really is just $[a, b]$ where $a = [4,5,6], b = [7,8,9]$. Picturing everything like this makes it abundantly clearer why we ultimately get $(2,1,3)$ instead of $(2,3,1)$. Now if you truly want $(2,3,1)$ then it would be $arr[:, :, None]$.

Choosing axes

Since we have already grasped what $((img\_1d - centroids[:, None]) ** 2)$ does, let us simply substitute it with an arbitrary array **arr**.

```
distances = np.sqrt(arr.sum(axis=2))
```

To finish the calculation of Euclidean distances, for each array resulting from $img\_1d[i] - centroids[j]$, we want to sum its values. Still, just calling $sum()$ would end up flattening the whole **arr** and summing everything, which is not what we wish. And so, a new conundrum arises, how do we know what **axis** to specify and what's with this seemingly erratic **axis indexing**.

Before proceeding with our $sum()$ problem, let us take note of some of the useful outcomes we have gotten from the **Broadcasting** section.

Method	Notes
1	arr has shape $(x, y, num\_channels)$. $arr[i][j]$ is the result of $img\_1d[i] - centroids[j]$. $arr[i][j][k]$ is the channel k of $arr[i][j]$.
2	arr has shape $(y, x, num\_channels)$. $arr[i][j]$ is the result of $img\_1d[j] - centroids[i]$. $arr[i][j][k]$ is the channel k of $arr[i][j]$.

Notice that in both methods, the 3rd entry in arr remains as $num\_channels$, and $arr[i][j][k]$ represents the same thing. Simply put, $arr[i][j]$ is the array containing all the channels derived from taking the difference between a pixel of the original image and a centroid. Said channels are exactly the values that we want to sum for each $arr[i][j]$.

Now to determine the **axis**, mark that in order to access the channels, we must pass through 2 *iterators* of arr, i.e., i and j. These can also be thought of as **axes**, so, since we have skipped 2 *axes*, our destination must be the 3rd one. In Python, indexing starts from 0, thus, our 3rd **axis** is actually numbered 2. And that's how the parameter $axis = 2$ is set.

After the sum is completely computed, $distances$ will be an array of shape (x, y) for **method 1** and (y, x) for **method 2**. The reason for that is quite straightforward, here is an example of an array with shape $(2,2,3)$ and the resulting array with shape $(2,2)$.

$$\begin{pmatrix}[[1\ 2\ 3]\\ [4\ 5\ 6]]\\ [[7\ 8\ 9]\\ [2\ 3\ 4]]]\end{pmatrix}.sum(axis = 2) = \begin{matrix}[[6\ 15]\\ [24\ 9]]\end{matrix}$$

As we can see, 6 is the result gotten from summing $[1,2,3]$, 15 from $[4,5,6]$, 24 from $[7,8,9]$ and lastly, 9 from $[2,3,4]$.

Next, employing the same concept, we will attempt to start the labeling process,

```
labels = np.argmin(distances, axis=0)
```

Regrettably however, picking **axes** becomes a tad more complicated because of the dissimilar shapes we get by using different **broadcasting** methods.

Method	Notes and solution

1	arr has shape $(x, y, num\_channels)$. $arr[i][j]$ is the result of $img\_1d[i] - centroids[j]$. $distances$ has shape (x, y). ➢ $labels = np.argmin(distances, axis = 1)$
2	arr has shape $(y, x, num\_channels)$. $arr[i][j]$ is the result of $img\_1d[j] - centroids[i]$. $distances$ has shape (y, x). ➢ $labels = np.argmin(distances, axis = 0)$

- $np.argmin$ returns the indices of the minimum values along an axis.

From my experience doing this project, I think a neat trick to deciding **axes** is just to observe the shape itself. This does require a thorough understanding of what each number in the shape denotes. In our case, we know that,

- x is the size of img_1d, i.e., the number of pixels in the image.
- y is the number of centroids, it is also $k\_clusters$ in most cases.
- $num\_channels$ is the number of channels of each pixel.

$num\_channels$ was already dealt with and we are now working with $distances$, there are only x and y to consider. Since we label a pixel by assigning it to share the same color as the centroid nearest to it, we must think in terms of $centroids$, in other words, along the **centroids axis**. And that is y, which is the 2nd entry – numbered 1 – in **method 1** or the 1st entry – numbered 0 – in **method 2**. Additionally, as $np.argmin$ returns the indices of the minimum values, we will use those to represent the indices of the clusters.

The final implementation

```
distances = np.linalg.norm(img_1d - centroids[:, None], axis=2)
labels = np.argmin(distances, axis=0)
```

Instead of expressing the Euclidean distance using the general formula mentioned in the **Broadcasting** section, it can also be written more compactly in terms of the **Euclidean norm** of the Euclidean vector difference, $d(p, q) = \|p - q\|$, which is also called the **2-norm**. This is the default value of the ord parameter in $np.linalg.norm$. Following the same logic as described in the above sections, we choose which array to broadcast then establish the axes accordingly.

- https://stackoverflow.com/questions/1401712/how-can-the-euclidean-distance-be-calculated-with-numpy

3. Calculate means and update centroids

Masks

```
points = img_1d[labels == i]
```

Each index in the $labels$ array holds a value signifying the index of a cluster, each index of $labels$ also corresponds to that same index in img_1d. Suppose we want to mark every spot in $labels$ containing the index i of a specific cluster, instead of manually iterating through the array and checking if $value == i$ every time, we can just do $mask = (labels == i)$. At first, this may seem very cryptic, but I think working through an example is enough to clear it up.

Assuming we have $labels = [1\ 2\ 3\ 1]$, here is the result for each i,

Array/i	**1**	**2**	**3**
mask	$[True\ False\ False\ True]$	$[False\ True\ False\ False]$	$[False\ False\ True\ False]$

As we can easily infer, all the places having $True$ are where the cluster's index i we want resides. Now that we have ourselves some **masks**, let us examine what they can do. The best way to accomplish this is with yet another visualization. This time, pretend this is our image array,

$$img\_1d = \begin{matrix}[[1\ 2\ 3] \\ [4\ 5\ 6] \\ [7\ 8\ 9] \\ [3\ 2\ 1]]\end{matrix} \ or\ [[1\ 2\ 3][4\ 5\ 6][7\ 8\ 9][3\ 2\ 1]]$$

Here are the outcomes for every mask we use.

Array/*mask*	$[True\ False\ False\ True]$	$[False\ True\ False\ False]$	$[False\ False\ True\ False]$
points	$\begin{matrix}[[1\ 2\ 3] \\ [3\ 2\ 1]]\end{matrix}$	$[[4\ 5\ 6]]$	$[[7\ 8\ 9]]$

So, when we pass a **mask** into img_1d by $img\_1d[mask]$, we are going to receive an array of **points**, or pixels, at the indices that are labeled as $True$.

Equipped with a better understanding of the concept of **masks** in Python, or in **numpy** specifically, circling back to the code at the start of this section, we can effortlessly discern that, with said line, we obtain an array $points$ consisting of **every pixel in the image that belongs to the cluster with index *i***.

- https://towardsdatascience.com/the-concept-of-masks-in-python-50fd65e64707

Mean

```
if len(points): centroids.append(np.mean(points, axis=0))
```

Given a data set $X = \{x_1, \dots, x_n\}$, the arithmetic mean is defined by the formula,

$$\bar{x} = \frac{1}{n}(\sum_{i=1}^{n} x_i) = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

meaning that to compute the mean, we are required to have a valid size n as the divisor. For that reason, $len(points)$ is checked to see if at least a pixel is in $points$, thus giving us a valid length.

Here we are faced with another **axes**-related decision, but at this point, we are well-prepared to handle it. Our $points$ is having a $(p, num\_channels)$ shape where p is the number of pixels in the cluster. We want to take the **mean** along this p axis so it is $axis = 0$.

- If we are to put $axis = 1$, it means we are taking the mean of all the channels for each pixel, which is obviously not we want.

The final implementation

```
prev_centroids = centroids.copy()
centroids = []

for i in range(k_clusters):
    points = img_1d[labels == i]
    if len(points): centroids.append(np.mean(points, axis=0))

k_clusters = len(centroids)
centroids = np.array(centroids)
```

As I have mentioned in the **Idea** section, my implementation does not guarantee that each cluster will contain at least one pixel, i.e., empty clusters can exist. An empty cluster indicates that its centroid has no pixel sharing its color. Such centroids are deemed useless and should be disposed of to not affect further calculations.

Due to the inability to remove nor append into a $np.array$, vanilla $list$ is used. We kept a copy of the current $centroids$ as $prev\_centroids$, which will come in handy in the upcoming section. Then, we empty out $centroids$ and append the means as new **centroids** as we go. After updating, if there really are **centroids** that were removed, we update $k\_clusters$ to reflect this change. Lastly, we convert the $list$ of **centroids** back to a $np.array$.

4. Check for convergence

```
try:
    if np.allclose(prev_centroids, centroids, atol=convergence_threshold):
        break
except ValueError:
    if np.allclose(prev_centroids, centroids[:, None],
                   atol=convergence_threshold):
        break
```

$np.allclose$ checks for convergence by examining whether the difference between $prev\_centroids$ and $centroids$ falls within a certain threshold. Here, I arbitrarily chose 0.2 as the $convergence\_threshold$. And that is really all there is to checking convergence.

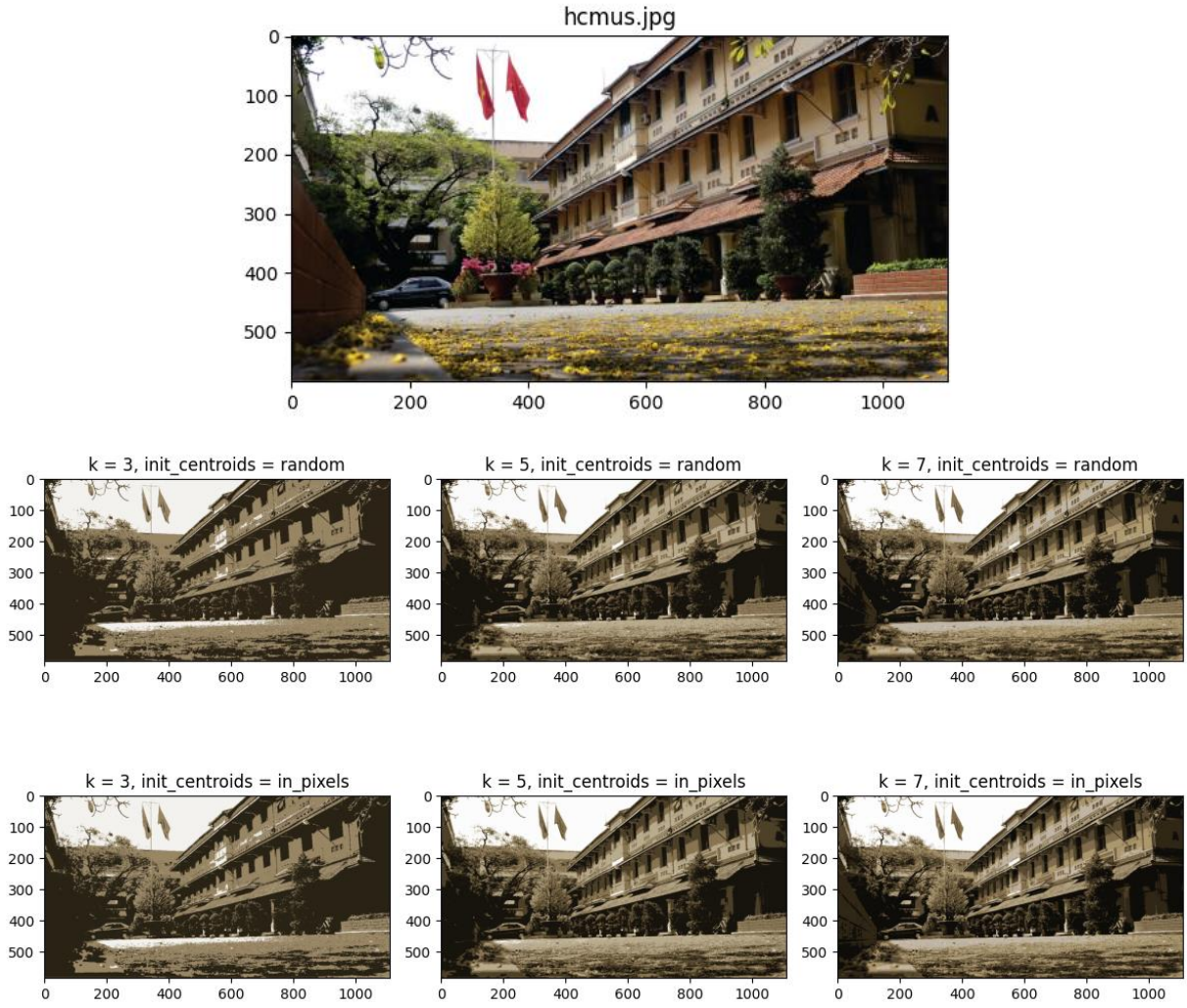
On a side note, I do recognize that my implementation looks highly makeshift with this $try \ldots except$ shenanigan, but as of writing this report, I still cannot think of a better solution. After our $centroids$ is revised, if there were centroids scrapped then $prev\_centroids$ with shape $(x, num\_channels)$ would definitely be incompatible with $centroids$ now shaped as

$(y, num\_channels)$ where $x < y$. However, if nothing changed then both would still retain the same shape. The problem arises as we cannot predict when this will happen, so the best thing we can do is try checking normally and if Python yells a $ValueError$ at us, we use $except$ to tell it to switch to the other lane which performs **broadcasting** before the check.

3. Experimentation and comments

$max\_iter = 1000$ and $convergence\_threshold = 0.2$. Each sample was run at least 5 times and the most common outcome is chosen to be included in this section. The runtime is an average of all the runs.

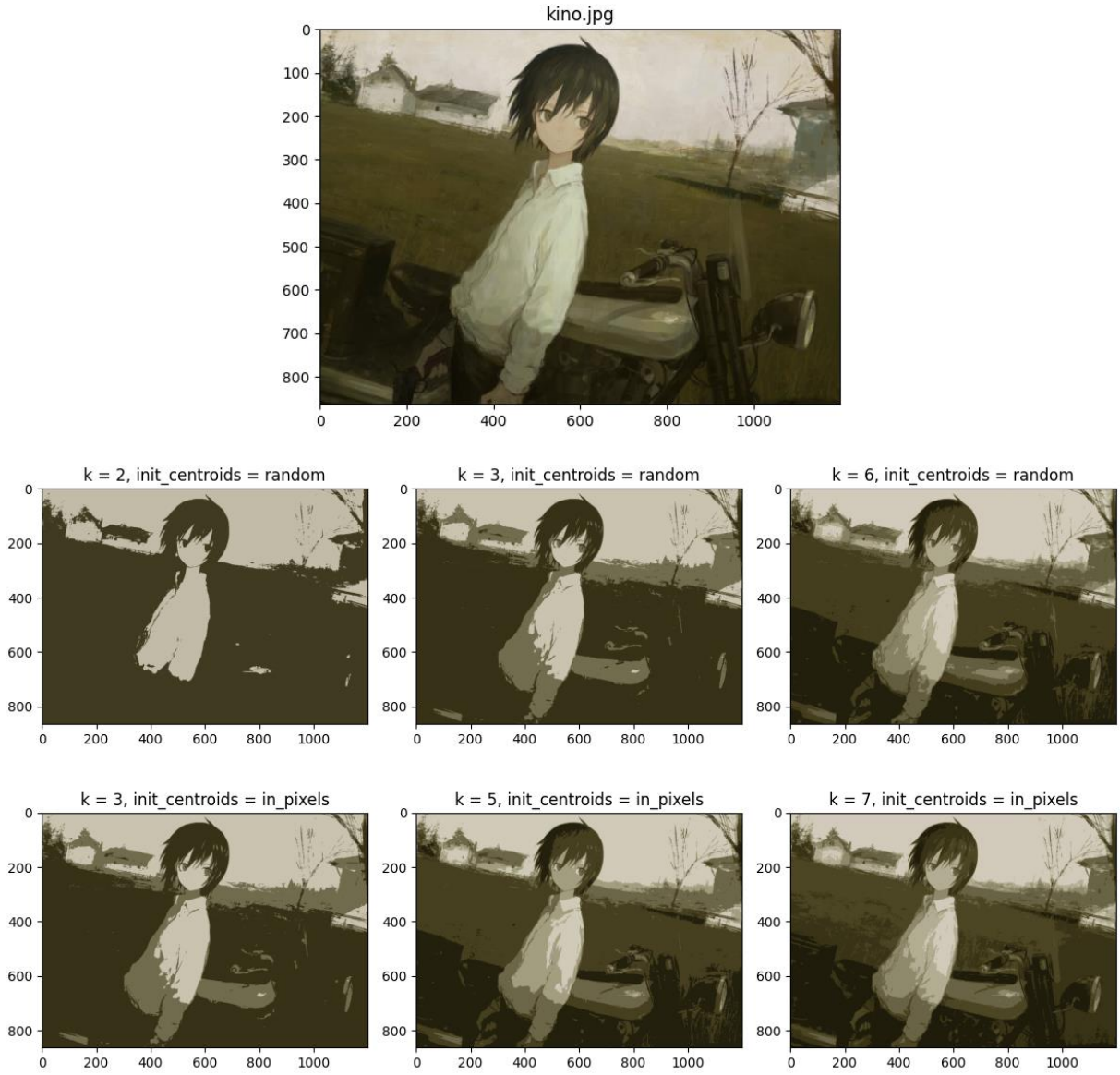
hcmus.jpg – 1110x584 – runtime: 17s



The algorithm is very suitable for $hcmus.jpg$, with only 17s to produce all 6 images and almost every object in each image is well discernible. In general, pictures that have a wide range of colors, and especially vibrant colors, tend to provide the best results both in terms of the recognizability of subjects and the lack of missing **clusters** post compression. Although, I must

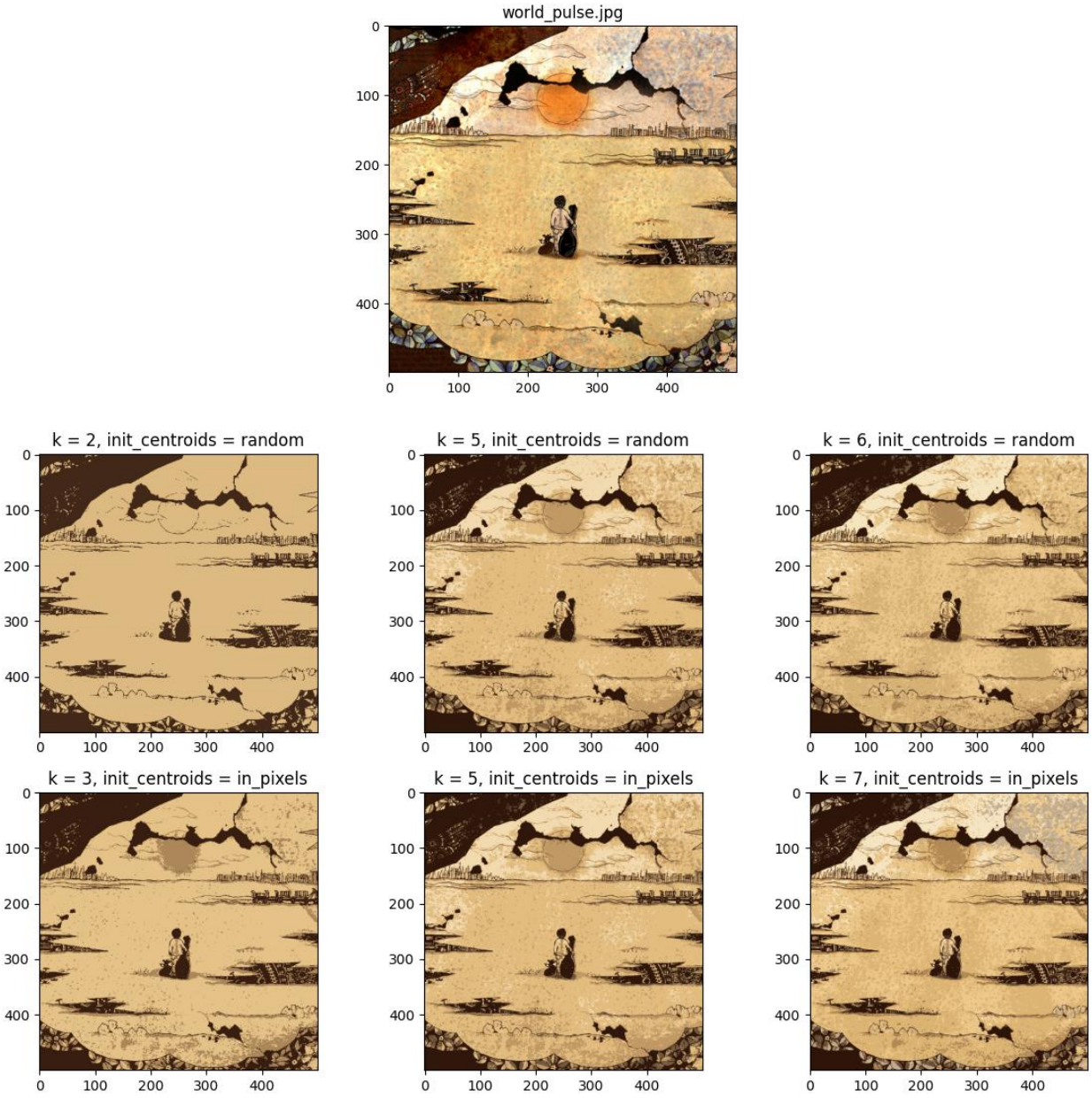
admit getting $init\_centroids = random$ to output 3,5,7 in one run with this sample did take quite some tries.

kino.jpg – 1200x864 – runtime: 47s



Out of the 3 samples, $kino.jpg$ has by far the worst results, a high runtime of 47s, awful coloring at lower $k_clusters$s and the consistent loss of **clusters**, mostly in **random init**. It seems K-means struggles quite a lot with pictures where the lines are not clearly defined, or the colors share similar characteristics. For instance, this sample has both, it mimics an oil painting where the colors blend together and the lines are either messy or not actually there. Every color has a kind of green, brown shade to them. All of these make it difficult for the algorithm to accurately label each pixel. Additionally, the steep rise in runtime mainly comes from the increase in the size of the picture.

world_pulse.jpg – 500x500 – runtime: 10s



This sample boasts the smallest size out of the 3 and evidently the lowest runtime. It only has one of 2 problems described in $kino.jpg$, i.e., everything possesses a yellow feel to them. However, it makes up for this by having extremely definitive lines. As such, $world\_pulse.jpg$ looks fantastic in every single one of its compressed forms.

4. Reference

- **numpy's API reference**

    - https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html
    - https://numpy.org/doc/stable/reference/generated/numpy.unique.html
    - https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html#numpy.linalg.norm
    - https://numpy.org/doc/stable/reference/generated/numpy.argmin.html
    - https://numpy.org/doc/stable/reference/generated/numpy.mean.html
    - https://numpy.org/doc/stable/reference/generated/numpy.allclose.html

- **Pillow's API reference**
    - https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html
    - https://pillow.readthedocs.io/en/stable/reference/Image.html#PIL.Image.fromarray
    - https://pillow.readthedocs.io/en/stable/reference/Image.html#PIL.Image.Image.save

- **matplotlib's API reference**
    - https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html#matplotlib.pyplot.subplots
    - https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.tight_layout.html

Idea

- **Pseudocode provided by Lecturer Phan Thi Phuong Uyen.**
- https://en.wikipedia.org/wiki/K-means_clustering
- https://youtu.be/4b5d3muPQmA
- https://towardsdatascience.com/create-your-own-k-means-clustering-algorithm-in-python-d7d4c9077670

Implementation

- https://numpy.org/doc/stable/user/basics.broadcasting.html
- https://youtu.be/oG1t3qlzq14
- https://stackoverflow.com/questions/1401712/how-can-the-euclidean-distance-be-calculated-with-numpy
- https://towardsdatascience.com/the-concept-of-masks-in-python-50fd65e64707
- https://stackoverflow.com/questions/10580676/comparing-two-numpy-arrays-for-equality-element-wise
- https://stackoverflow.com/questions/49643907/clipping-input-data-to-the-valid-range-for-imshow-with-rgb-data-0-1-for-floa
- https://stackoverflow.com/questions/14770735/how-do-i-change-the-figure-size-with-subplots