VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

TOÁN ỨNG DỤNG VÀ THỐNG KÊ

PROJECT 1
COLOR COMPRESSION

Sinh viên thực hiện: Nguyễn Đình Quang Khánh – 20127530

Tp. Hồ Chí Minh, Tháng 06/2022

Contents

1 Project ideas and K-means clustering algorithm

1.1 Project ideas

In computer, a picture can be stored in an array of pixels. In practical, there are many types of pictures. When the computer read a color picture, it will read a picture as an pixel array which is specified by the size of the picture and number of channels in one pixel. And therefore, we have countless number of matrices. Nowadays, the number of channels of common pictures are 3 represents for RGB color with the values in range [0; 255].

K-means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

In this project, we want to compress the color of the picture therefore we have to group some pixels together to generate some clusters. And then we will find the representative pixel for each cluster, which will usually be the center of that cluster. But the special point here that we don't know which criteria to group pixels together from. Therefore we can choose k randomly to compress the color of a picture into k clusters only.

1.2 Description of K-means clustering algorithm

In here, I will summarize of **K-means clustering** algorithm

Input: Data X and a number of clusters $k\_clusters$ you want to group.

Output: All centers M and $labels$ vectors for each pixels.

K-means clustering algorithm has 5 steps to follow:

- Step 1: Pick $k\_clusters$ randomly outside or inside the pixel array for the initial centers.
- Step 2: Assign each pixels into the clusters with the nearest distance to their $centroids$
- Step 3: If the assignment of each pixels into one cluster in step 2 doesn't change from the previous iteration then we can stop the algorithm.
- Step 4: Update the $centroids$ for each clusters by getting the mean of all data-points have been assigned to each clusters in step 2.
- Step 5: Go back to step 2.

2 Functions description

```python
def kmeans(img_1d, k_clusters, max_iter, init_centroids='random'):
  # Get 3 dimensions (height, width, num_channels) information of image
  height = img_1d.shape[0]
  width = img_1d.shape[1]
  num_channels = img_1d.shape[2]
  # Convert an array with 3 dimensions into 2 dimensions
  img_1d = img_1d.reshape(height * width, num_channels)
  # Create k_clusters centroids with init_centroids = 'random' or 'in_pixels'
  centroids = [kmeans_init_centers(img_1d, k_clusters, init_centroids)]
  labels = []
  while max_iter:
    labels.append(kmeans_assign_labels(img_1d, centroids[-1]))
    new_centroids = kmeans_update_centers(img_1d, labels[-1], k_clusters,
                                                                num_channels)
    if has_converged(centroids[-1], new_centroids): break
    centroids.append(new_centroids)
    max_iter -= 1
  return (centroids, labels)
```

- This function will return the centroids and the labels vectors of each data-points.
- In the first step, we can easily see that the pixel array of a picture read in will have the dimensions of 3, this will be a little challenge to deal with it. So we will make it easier to manipulate by reshape the 3 dimensions into 2 dimensions.

  Then we will initialize the centers of *k_clusters* clusters depend on the parameter *init_centroids* pass in.

  If *init_centroids* = 'random' –> centroid has 'c' channels, with 'c' is initial random in [0,255].

  If *init_centroids* = 'in_pixels' –> centroid is a random pixels of original image.
- In step 2, we will assign each pixels into the clusters with the nearest distance to their *centroids* in line 12.
- In step 3, we will check if the assignment of each pixels into one cluster in step 2 doesn't change from the previous iteration then we can stop the algorithm in line 15.
- In step 4, we will append the *new_centroids* into the *centroids* array for later check in line 16.
- We will stop the algorithm in *max_iter* times or when the converge condition has been reach.
- At the end, the function will return a tuple of 2 values *centroids* and *labels*.

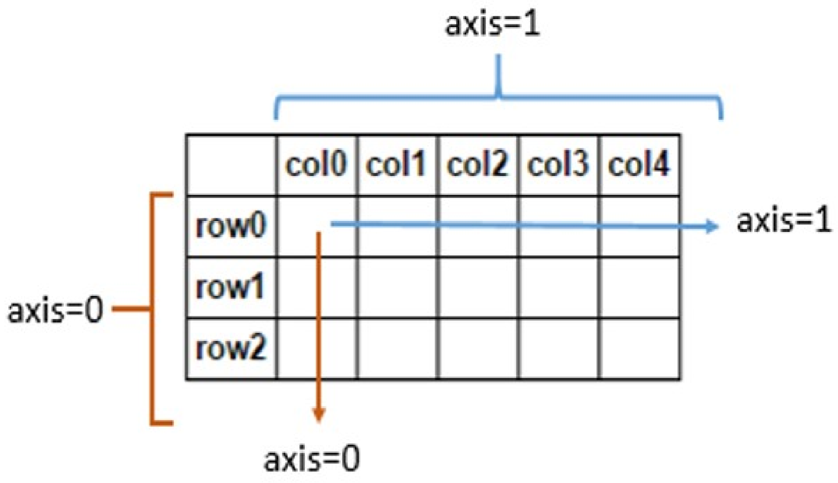
```python
def kmeans_init_centers(img_1d, k_clusters, init_centroids):
  # if centroid is a random pixels of original image we will pick k_clusters pixels in img_1d
  if init_centroids == 'in_pixels':
    return img_1d[np.random.choice(img_1d.shape[0], k_clusters, replace=False)]
    # if centroid has 'c' channels, with 'c' is initial random in [0,255]
    # then we will pick k_clusters tuple of img_1d.shape[1] values in range(256)
  if init_centroids == 'random':
    return np.random.choice(256, size = (k_clusters, img_1d.shape[1]),
                                                    replace=False)
```

- This function will return *k_clusters* pixels randomly inside or outside our image array as the center points for each clusters.
- In first condition, if *init_centroids* = 'in_pixels' so we will use *np.random.choice()* from numpy module for pick a random array with *k_clusters* pixels from the *img_1d* array to make *centroids* array inside our image array. Therefore, the *centroid* is a random pixels of original image.
- In second condition, if *init_centroids* = 'random' then we will use *np.random.choice()* function from numpy module for an array picked randomly with size = (*k_clusters*, *num_channels*) and we want all centers are distinguish so we add *replace = False*. At the end, the centroid will have 'c' channels, with 'c' is initial random in [0,255].

```python
def kmeans_assign_labels(img_1d, centroid):
  distances = []
  for i in range(centroid.shape[0]):
  # Distance between each elements in img_1d vs centroid[i] in Euclidean distance
    distance = np.sqrt(np.sum((img_1d - centroid[i]) ** 2, axis=1))
    distances.append(distance)
  # convert list to np.ndarray
  distances = np.array(distances)
  # Return the indices of clusters where each pixels with minimum distances to all centroids belong to
  return np.argmin(distances.T, axis = 1)
```

- This function will return the indices of clusters where each pixels with minimum distances to all *centroids* belong to or we can call it as *labels* in previous section.

- In this function, I will calculate the distances of all pixels to all centroids and pick the nearest centroid for each pixels. I will calculate the distance in Euclidean distance by using this formula $np.sqrt(np.sum((img\_1d - centroid[i]) * *2,\ axis = 1))$, and parameter $axis = 1$ is for calculate sum in horizontal, as the figure demonstration below.



- Then we will get a *distances* array and we will convert to np.ndarray for later use.
- Now, our *distances* array have $size = (num\_channels, img\_1d.shape[0])$. Then we will transpose it to get array with $size = (img\_1d.shape[0], num\_channels)$.
- At the end, we will use $np.argmin()$ function from numpy module and pass in the transpose form of *distances* np.ndarray with $axis = 1$ to get the indices of clusters where each pixels with minimum distances to all *centroids* belong to.

```
def kmeans_update_centers(img_1d, label, k_clusters, num_channels):
  centroids = np.zeros((k_clusters, num_channels))
  for k in range(k_clusters):
    # collect all points assigned to the k-th cluster
    clusterk = img_1d[label == k, :]
    # if clusterk is empty we don't need to update
    if len(clusterk) == 0:
      continue
    # Take average of all datapoints belong to cluster k
    centroids[k, :] = np.mean(clusterk, axis = 0)
  return centroids
```

- This function will return the new *centroids* updated from all clusters get from the latest labels.
- First, we will collect all the *data-points* assigned to the k-th cluster.
- Secondly, if the *k-th cluster* doesn't have any data-points so we don't need to update the center of that *k-th cluster*.
- Thirdly, We will take average of all *data-points* belong to *k-th cluster*. And we have to iterate these steps for all k clusters.
- Finally, after updated all k clusters we will return the new *centroids*.

```
def has_converged(centers, new_centers):
  # if we have absolute(centers[i] - new_centers[i]) <= atol(=1)
  # => the difference of two RGBs are less than 1
  # we can approve these both RGBs are the same.
  return np.allclose(centers, new_centers, atol = 1)
```

- This function will check the converge condition for the new *centroids* and the old *centroids*, if these two arrays have each relative elements is close to each other and the *errornumber* is less than 1 by using $atol = 1$ then we can consider that both RGBs colors are the same.

```
def get_new_image(img_1d, k_clusters, centroid, label):
  # Get 3 dimensions (height, width, num_channels) information of image
  height = image_array.shape[0]
  width = image_array.shape[1]
  num_channels = image_array.shape[2]
  # Because the original image_array have 3 dimensions so we have to reshape it to
   2 dimensions
  img_1d = img_1d.reshape(height * width, num_channels)
  # Create new image array by adding the cluster index (=centroid[k]) to each
  pixels
  new_img = np.zeros((img_1d.shape[0], img_1d.shape[1]))
  # Get the new image array with k_clusters centroids assigned to each pixels.
  for k in range(k_clusters):
    new_img[label == k, :] += centroid[k]
  new_img = new_img.reshape(height, width, num_channels)
  return new_img
```

- This function will return the new image array with *k_clusters centroids* assigned to each pixels that we will export to .png and .pdf files at the end.

3 Results

I prepared 2 pictures for testing time.

Figure 1: *avatar.jpg*

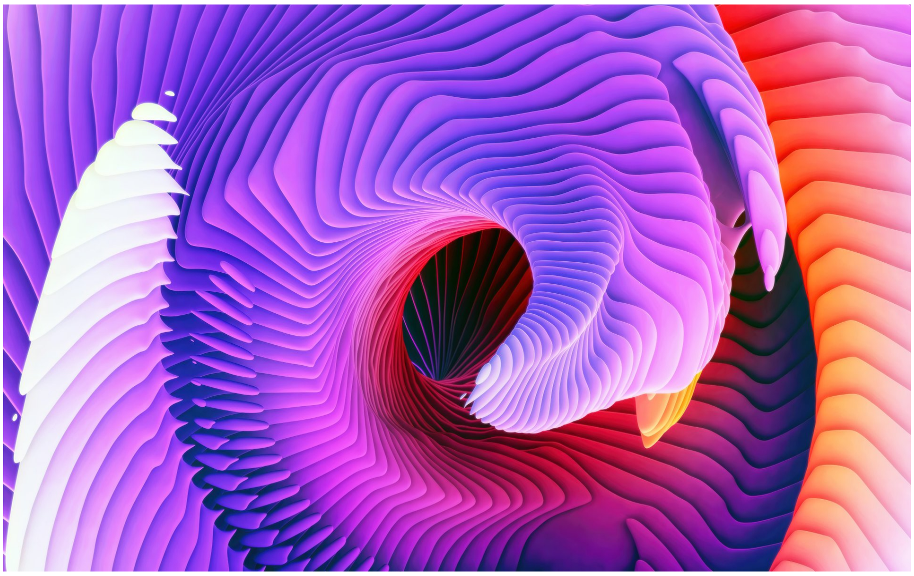


Figure 2: *img.jpg*

3.1 Testing 1: K = 3

3.1.1 Sample 1

With sample 1, we will get the centroids and labels results as below.

My new centroids result:

```
[[ 51.19524933 152.9280978  253.43506519]
 [ 52.54914301  56.8495358   92.77106767]
 [232.01616437 220.94881897 220.59589693]]
```

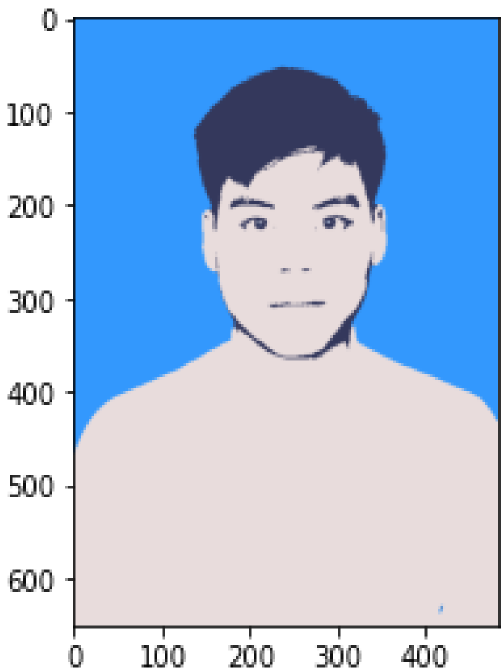


Figure 3: *my_avatar_3.jpg*

Sklearn centroids result:

```
[[ 51.19613927 152.92673114 253.43151257]
 [231.93665774 220.82114514 220.46234457]
 [ 51.30013981  56.10282776  92.4355297 ]]
```

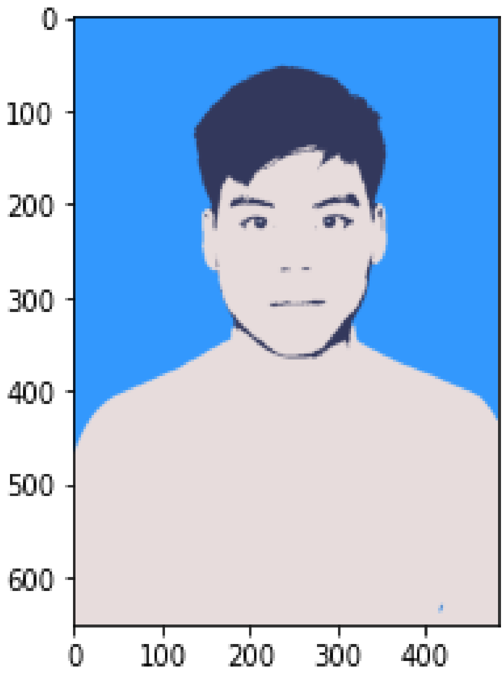


Figure 4: *Sklearn_ avatar_ 3.jpg*

⇒ We can see that elements in order [0, 1, 2] of my new centroids are approximately with elements in order [0, 2, 1] in Sklearn centroids. And the picture show between both ways are also the same.

3.1.2 Sample 2

With sample 2, we will get the centroids and labels results as below.

My new centroids result:

```
[[152.49458003  70.77234422 230.48494644]
 [108.40246167  32.02635536  87.00908169]
 [240.40357334 168.593392   213.77850318]]
```

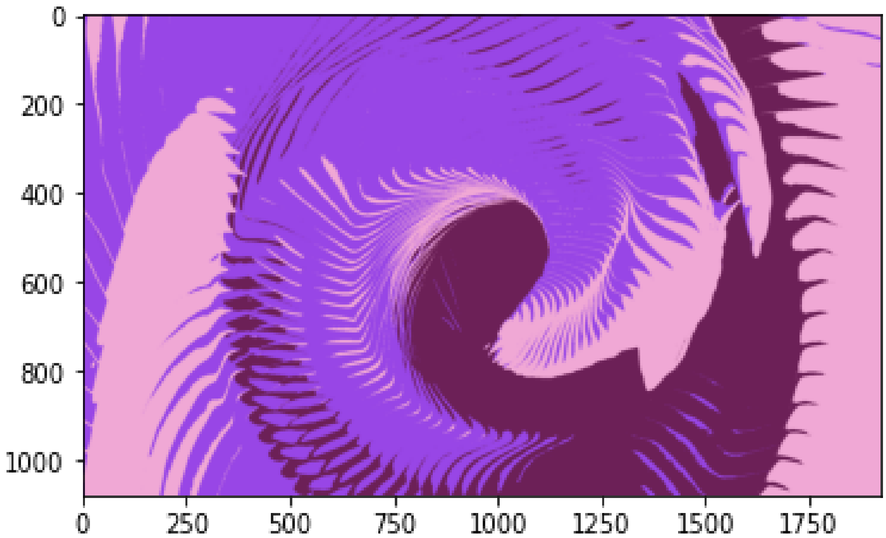


Figure 5: *my_ img_ 3.jpg*

Sklearn centroids result:

```
[[150.98283514  70.00983381 229.72601646]
```

```
[239.94627034 167.68524163 214.65643656]
[109.36530543  32.08481787  85.15615837]]
```

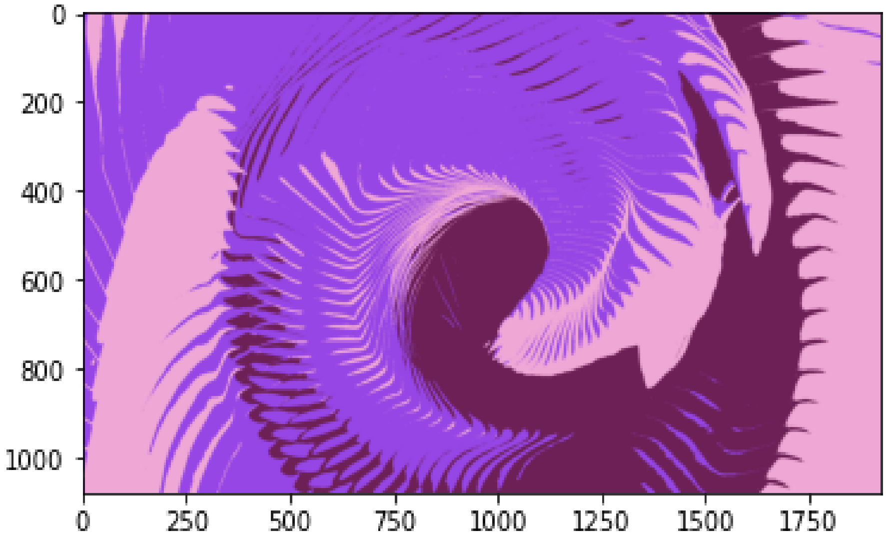


Figure 6: *Sklearn_ img_ 3.jpg*

⇒ We can see that elements in order [0, 1, 2] of my new centroids are approximately with elements in order [0, 2, 1] in Sklearn centroids. And the picture show between both ways are also the same.

3.2 Testing 2: K = 5

3.2.1 Sample 1

With sample 1, we will get the centroids and labels results as below.

My new centroids result:

```
[[237.17988422 235.71493613 238.71842856]
[183.0349162  137.36658686 130.67936951]
[223.59273408 181.44805243 168.96662921]
[ 51.13458859 152.93720386 253.49105014]
[ 28.92059499  44.65924751  88.76326151]]
```

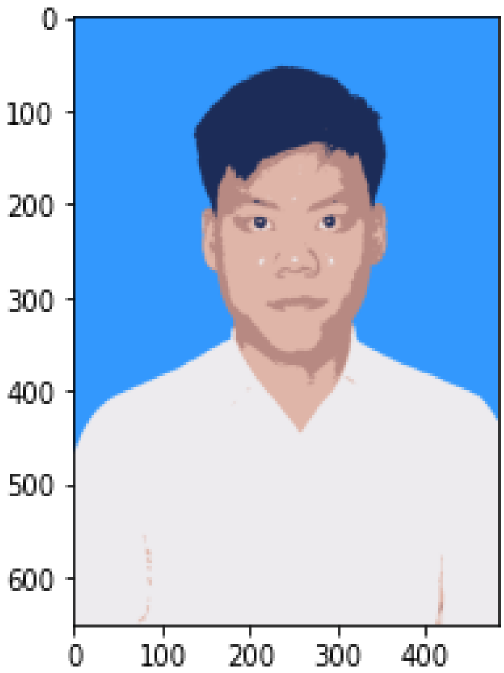


Figure 7: *my_ avatar_ 5.jpg*

Sklearn centroids result:

```
[[237.17835512 235.70043832 238.69961747]
[ 51.13306751 152.93809229 253.49341127]
[180.81530172 135.26429598 129.06688218]
[222.76587715 180.41203321 167.96858714]
[ 28.59175495  44.52006368  88.74908053]]
```

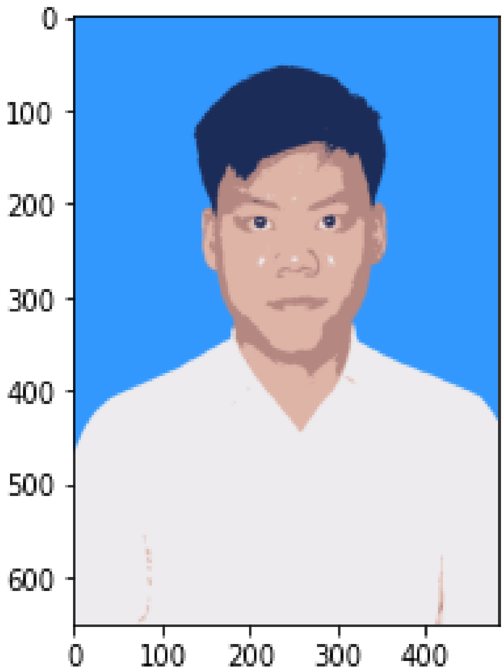


Figure 8: *Sklearn_ avatar_ 5.jpg*

⇒ We can see that elements in order [0, 1, 2, 3, 4] of my new centroids are approximately with elements in order [0, 2, 3, 1, 4] in Sklearn centroids. And the picture show between both ways are also the same.

3.2.2 Sample 2

With sample 2, we will get the centroids and labels results as below.

My new centroids result:

```
[[225.94005141  76.23554964  92.81608875]
[184.41939849  87.45554131 242.36262026]
[ 61.20119247  25.94475322  70.36237693]
[241.88296006 189.97461332 231.20937672]
[105.90379453  49.6280514  204.01769168]]
```

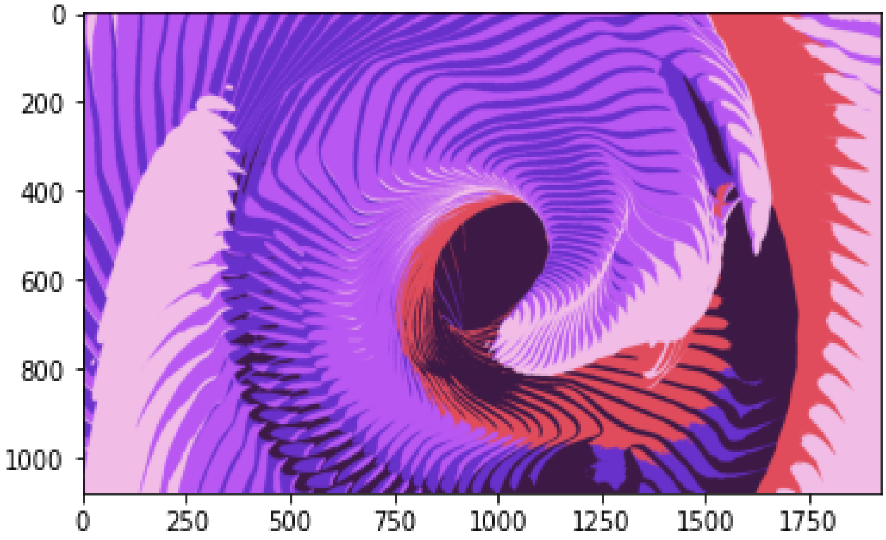


Figure 9: *my_img_5.jpg*

Sklearn centroids result:

```
[[105.56029721  49.23909135 202.91437094]
[241.96136162 190.19693936 230.95834668]
[225.63839039  75.76976228  92.37659333]
[184.31749334  87.48831962 242.33615897]
[ 60.22890173  25.90469444  69.19434866]]
```

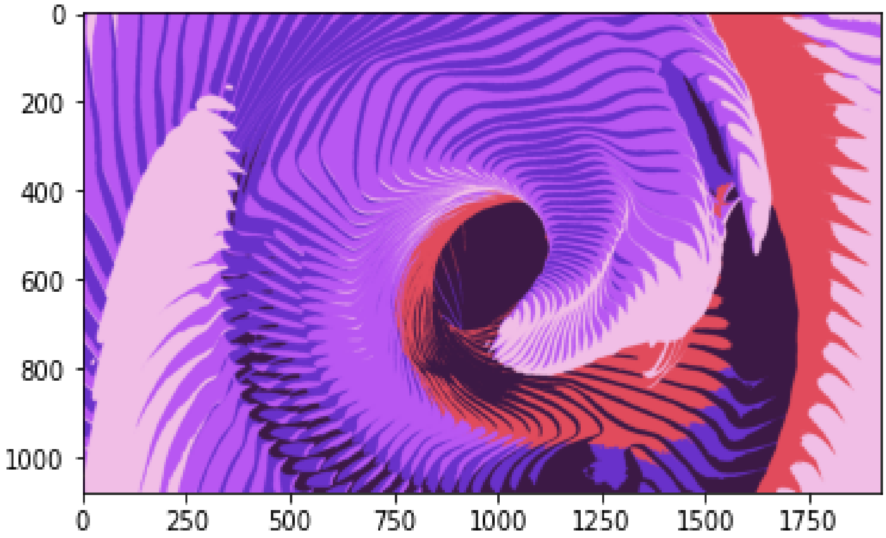


Figure 10: *Sklearn_img_5.jpg*

⇒ We can see that elements in order [0, 1, 2, 3, 4] of my new centroids are approximately with elements in order [2, 3, 4, 1, 0] in Sklearn centroids. And the picture show between both ways are also the same.

3.3 Testing 3: K = 7

3.3.1 Sample 1

With sample 1, we will get the centroids and labels results as below.

My new centroids result:

```
[[200.66189834 155.36184558 144.38218913]
[154.21294206 109.71162528 110.63694507]
[225.47337502 223.78750553 229.20712526]
[ 51.10398854 152.92473668 253.4950666 ]
[228.10697128 186.355668   173.86778084]
[ 26.45727991  43.66089165  88.68510158]
[238.73843986 237.30446404 239.98408037]]
```

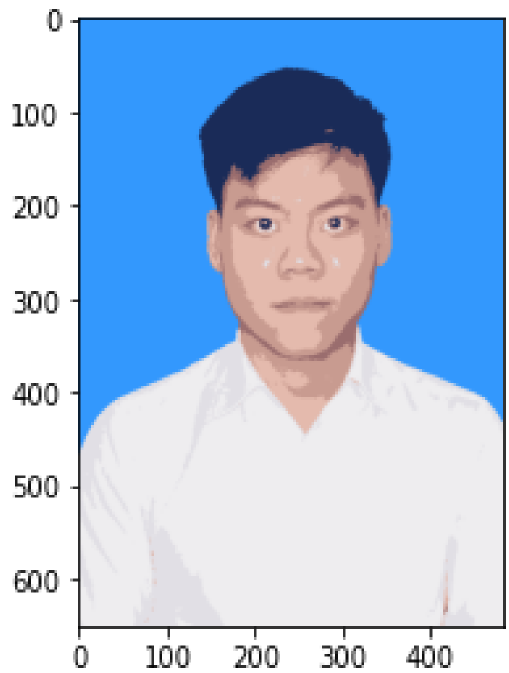


Figure 11: *my_ avatar_ 7.jpg*

Sklearn centroids result:

```
[[237.22112779 235.78870616 238.79750074]
[ 51.13558876 152.93647132 253.48837191]
[ 69.55573248  61.79578025  89.03423567]
[208.0948382  163.46853656 150.79033703]
[172.61675158 126.37305828 122.22530135]
[ 16.03271923  39.22903464  88.75806927]
[230.42063091 190.31753943 179.18567823]]
```

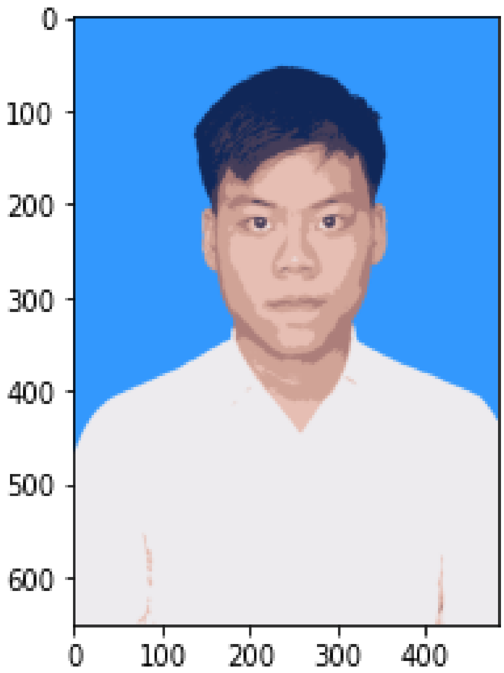


Figure 12: *Sklearn_ avatar_ 7.jpg*

⇒ We can see that elements of my new centroids are elements in Sklearn centroids are a little bit different between due to small iteration and number of channel. And the picture show between both ways are also still the same.

3.3.2 Sample 2

With sample 2, we will get the centroids and labels results as below.

My new centroids result:

```
[[250.6213815  131.09667809 112.78639734]
 [184.28725409  30.54411052  85.21425341]
 [215.18507578 111.2322328  244.68671812]
 [ 44.50792719  26.43193553  58.06813534]
 [242.807568   208.01291449 238.10918674]
 [ 85.11012753  41.21778787 181.9972537 ]
 [151.49337859  70.21748747 235.78896857]]
```

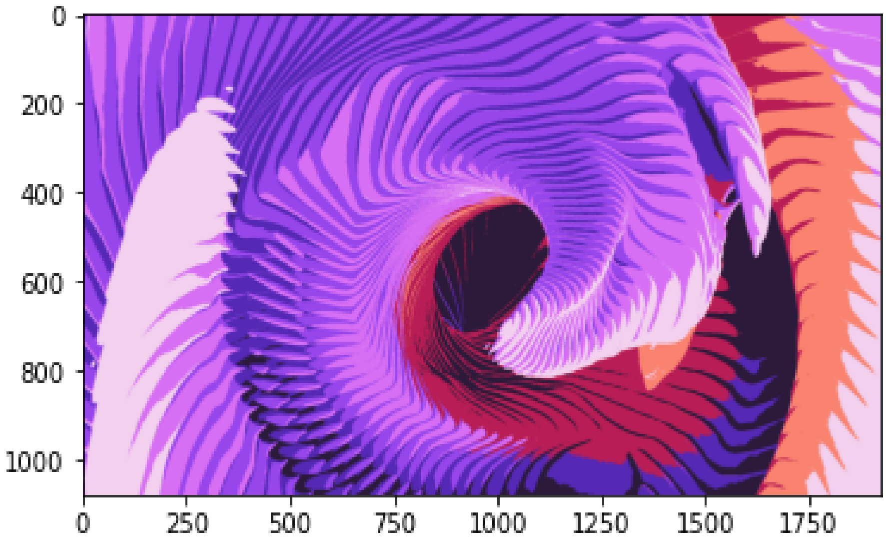


Figure 13: *my_img_7.jpg*

Sklearn centroids result:

```
[[214.80672576 110.90651762 244.7791835 ]
[188.20019256  32.19658732  84.24964964]
[ 84.84550239  40.77940424 180.35044748]
[242.70645544 208.33447862 238.88828772]
[250.6891642  134.3522698  116.17787354]
[ 45.37386589  26.24445252  57.33711496]
[150.78475293  69.92127439 235.55118058]]
```

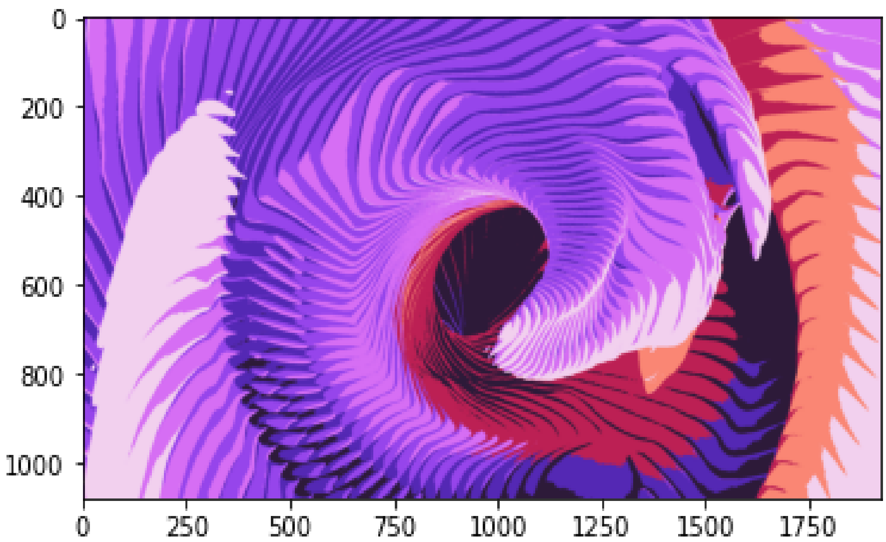


Figure 14: *Sklearn_img_7.jpg*

⇒ We can see that elements of my new centroids are elements in Sklearn centroids are a little bit different between due to small iteration and number of channel. And the picture show between both ways are also still the same.

4 Comment

Overall, this k-mean clustering program give quite good answer compare to the answer provided by $KMeans()$ of $scikit-learn$ module. And k-mean clustering algorithm are easy to implemented due to its clearly run-flows. Because at the very last iteration, *centroids* will be updated with a very small error number or maybe not be changed because it will be closer to the center of its cluster time by time. And with a small number of iteration we cannot get the accuracy result in a large data given. And with the randomly chosen each restart we will get different results each time so we have to redo it time by time and choose the most accuracy result as possible.

5 References

[1] machinelearningcoban.com

[2] nguyenvanhieu.vn

[3] stackoverflow.com

[4] en.wikipedia.org

[5] courses.ctda.hcmus.edu.vn