HO CHI CITY UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

---------------o0o---------------

APPLIED MATHEMATICS AND STATISTICS REPORT

Project 1: Color Compression

Student name	:	Đặng Hà Huy
Class	:	21CLC05
Student ID	:	21127296
Instructor	:	Phan Thị Phương Uyên

Ho Chi Minh City, July 2023

TABLE OF CONTENT

MAIN ARTICLE

I/ Introduction

- **Student Name:** Đặng Hà Huy
- **Student ID:** 21127296
- **Class:** 21CLC05

II/ Implementation ideas and code description

1/ Implementation idea

A step by step on how the code work and implementation idea

- Step 1: Let the user choose how many k clusters, max iteration, centroids initialiaztion type, etc as the initial argument for the Kmeans function

- Step 2: First determine the input image shape to get the length and dimension

- Step 3: Check the init_centroids. If it is set to '**random**' then creates k_clusters random centroids with values ranging from 0 to 255. If it is set to '**in_pixels**' then it randomly selects k_clusters pixels from img_1d to use as centroids

- Step 4: Create an array that will store the cluster assignments for each pixel

- Step 5: The algorithm enters a loop that will run for **max_iter** iterations to iterates over each pixel in the image

- Step 6: Calculates the distance between the current pixel and each centroid

- Step 7: Update the centroids

- Step 8: Rejoining the pixels to create an image and save

2/ Code description

flatten_img(): flattens the image into a 1-dimensional array of pixels, where each pixel is represented by its RGB values. The resulting array has a shape of (width * height, 3).

```
# Flatten image
def flatten_img(img):
  height, width = img.size
  return np.reshape(img, (width * height, 3)).astype(int)
```

compress_img(): takes an image represented as a NumPy array, along with the centroids and labels obtained from a compression algorithm. It reconstructs the compressed image by mapping each label to its corresponding centroid value, reshapes it back to the original dimensions, and returns the compressed image as a NumPy array.

```
# Compress image base on the provided centroids and label
def compress_img(centroids, labels, img):
  height, width = img.size
  compressed = centroids[labels.astype(int)]
  compressed = compressed.reshape((width, height, 3)).astype(int)
  return compressed
```

show_img(): displays the image with the provided information (image name, initial centroids and the number of k cluster) as the plot title

```
# Show image function
def show_img(img, img_name, img_centroids, num_cluster):
  plt.title(f'{img_name} {img_centroids} with k cluster = {num_cluster}')
  plt.imshow(img)
  plt.axis('off') # Remove the axis
  plt.show
```

kmeans(): applies the k-means clustering algorithm with the following argument: a 1-dimensional image array (**img_1d**), the number of clusters (**k_clusters**) and a maximum number of iterations (**max_iter**).

- It supports two methods for initializing centroids: **random** (which is set t default) randomly select the values ranging from 0 to 255 and **in_pixels** which is selecting random pixels from the image.

```
def kmeans(img_1d, k_clusters, max_iter, init_centroids='random'):
  # Choose k random centroids
  length, dim = img_1d.shape
  if init_centroids == 'random':
    centroids = np.random.choice(256, size = (k_clusters, dim),
replace=False)
  elif init_centroids == 'in_pixels':
    centroids = img_1d[np.random.choice(length, size = k_clusters,
replace=False)]
```

- The function then iteratively calculate the distance from the centroids to each pixels and assigns pixels to clusters

```
labels = np.zeros(shape=(length))
  while max_iter:
    for i in range(length):
      min_dist = float('inf')

      # Distance between each pixel and centroids
      for j in range(k_clusters):
        mean_dist = np.sqrt(np.sum((img_1d[i] - centroids[j])**2, axis=0))
        if mean_dist < min_dist:
          min_dist = mean_dist
          labels[i] = j
```

- It then updates the centroids and repeats this process until the maximum number of iterations is reached. It returns the resulting centroids and labels for each pixel.

```
    # Update centroids
    for i in range(k_clusters):
      pixels = img_1d[labels == i]
      if len(pixels):
        centroids[i] = np.mean(pixels, axis = 0)
    max_iter -= 1

  return centroids, labels
```

main(): The main function that let user choose the the path to the image (**img_path**), the number of max iteration (**max_iter**), the number of k clusters (**k_cluster**) and the initialize centroids type (the user choose between 'random' with 0 and 'in_pixels' with 1). Be sure to use '/' for the directory instead of '\'. For example do not use **'C:\Users\Desktop'** but use **'C:/Users/Desktop'**

```
def main():
  # Input parameters
  img_path = input("Enter image's path (use C:/ instead of C:\): ")
  max_iter = int(input("Enter max iteration: "))
  k_cluster = int(input("Number of k cluster: "))
  centroidsT = int(input("Initial centroid: \n0) random \n1) in_pixels
\nYour choice: "))

  # File name add-on
  if centroidsT == 0:
    centroids_type = 'random'
  elif centroidsT == 1:
    centroids_type = 'in_pixels'
  else:
    print('Invalid centroid initialization method')
    return
```

- Open the image using Image.open() function with the image path that the user input as an argument and convert it to RGB. Next flatten the image into a 1-dimesional array for processing

```
  # Open the image
  init_img = Image.open(img_path).convert('RGB')

  # Flatten image for processing
  flat_img = flatten_img(init_img)
```

- Start processing the image with k-means algorithm that take all the user previously input variables as arguments. The image is then compress back with the centroids and lables that the k-means function return

```
  # Kmeans processing
  centroids, labels = kmeans(flat_img, k_cluster, max_iter,
init_centroidsT[centroidsT])
  final_img = compress_img(centroids, labels, init_img)
  final_img = Image.fromarray(final_img.astype('uint8'), 'RGB')
```

- Processing the image path that the user has input before into the directory and image name. For example with the img_path = 'C:/Users/Desktop/forest.jpg', the output_name is 'forest' with the extension remove because the extension is going to be added back when save. The directory_path is 'C:/Users/Desktop' and it is used to save the image.

```
# Process image name
  ## Remove the front directory and the .<extension>
  output_img = (img_path.split('/')[-1]).split('.')[0]

  # Get the directory path for saving purposes
  directory_path = img_path.replace('/' + img_path.split('/')[-1], '')
```

- And finally the save image part. We use the save() function that take the directory where the user want to save the image as a string and the extension of the image with 3 choices that are PNG, JPEG and PDF. After saving the image we show the image with the show_img() function previously mentioned above

```
 # Output file type choice
  save_choice = int(input('Enter output file type: \n1) PNG \n2) JPG
\n3) PDF \nYour choice: '))

  # Output file check and save image
  if save_choice == 1:
    final_img.save(f"{directory_path}/{output_img}_{centroids_type}_{k_c
luster}_compressed.png","PNG")
    print(f"Image saved at
{directory_path}/{output_img}_{centroids_type}_{k_cluster}_compressed.pn
g")
```

```
  elif save_choice == 2:
    final_img.save(f"{directory_path}/{output_img}_{centroids_type}_{k_c
luster}_compressed.jpg","JPEG")
    print(f"Image saved at
{directory_path}/{output_img}_{centroids_type}_{k_cluster}_compressed.jp
g")
```

```
elif save_choice == 3:
    final_img.save(f"{directory_path}/{output_img}_{centroids_type}_{k_c
luster}_compressed.pdf","PDF")
    print(f"Image saved at
{directory_path}/{output_img}_{centroids_type}_{k_cluster}_compressed.pd
f")
  else:
    print("Invalid file save type!")
    return

  # Show image
  show_img(final_img, output_img, init_centroidsT[centroidsT],
k_cluster)
```

Test program: Created solely for testing the k-means algorithm with k = {3, 5, 7} and max iteration = 10. There are two functions **random_test()** and **in_pixels_test()** that tested for all 3 cases of k in two different centroid types.

```
def random_test(img):
  output_img = []
  output_img.append(img)
  for k_cluster in [3, 5, 7]:
    flat_img = flatten_img(img)
    centroids, labels = kmeans(flat_img, k_cluster, 10, 'random')
    final_img = compress_img(centroids, labels, init_img)

    final_img = Image.fromarray(final_img.astype('uint8'), 'RGB')
    output_img.append(final_img)

  # Creating subplot
  plot_iter = 1
  fig, axis = plt.subplots(1, 3, figsize=(12, 8))

  for i,k in [(0,3), (1,5), (2,7)]:
    axis[i].set_title(f'Random with k cluster = {k}')
    axis[i].imshow(output_img[plot_iter])
    axis[i].axis('off')
    plot_iter += 1
  plt.tight_layout();
```

```
def in_pixels_test(img):
  output_img = []
  output_img.append(img)
  for k_cluster in [3, 5, 7]:
    flat_img = flatten_img(img)
    centroids, labels = kmeans(flat_img, k_cluster, 10, 'in_pixels')
    final_img = compress_img(centroids, labels, init_img)

    final_img = Image.fromarray(final_img.astype('uint8'), 'RGB')
    output_img.append(final_img)

  # Creating subplot
  plot_iter = 1
  fig, axis = plt.subplots(1, 3, figsize=(12, 8))

  for i,k in [(0,3), (1,5), (2,7)]:
    axis[i].set_title(f'In_pixels with k cluster = {k}')
    axis[i].imshow(output_img[plot_iter])
    axis[i].axis('off')
    plot_iter += 1
  plt.tight_layout();
```

III/ Demo and comment

1/ Test program with k = {3, 5, 7} and 'random' centroids initialization

Original Image

Random with k cluster = 3

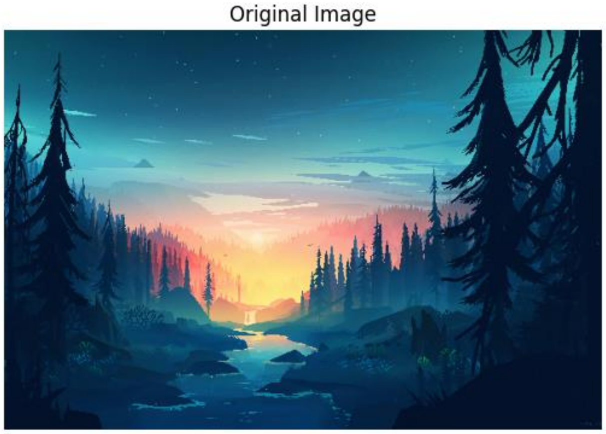
Random with k cluster = 5

Random with k cluster = 7

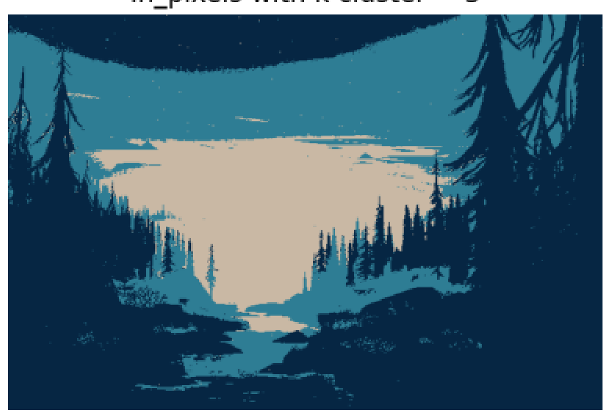
	Total runtime (10 iterations)	Average runtime/iteration
K_cluster = 3	53.2 seconds	5.3 seconds
K_cluster = 5	94.3 seconds	9.4 seconds
K_cluster = 7	142.8 seconds	14.2 seconds

2/ Test program with k = {3, 5, 7} and 'in_pixels' centroids initialization

Original Image



In_pixels with k cluster = 3



	Total runtime (10 iterations)	Average runtime/iteration
K_cluster = 3	82.6 seconds	8.2 seconds
K_cluster = 5	128.2 seconds	12.8 seconds
K_cluster = 7	174.12 seconds	17.4 seconds

3/ Comment

Runtime: Average runtime and total runtime for smaller k cluster value is significantly faster than bigger k cluster value

File size: Affect runtime, due to the number of data points to compare. Larger file size led to a much longer runtime

Image quality: Larger k cluser value will retain more details from the images than smaller k cluster value, but smaller k will still have some defining contents of the images

REFERENCES

Image compression using K-means clustering – GeeksForGeeks

Image Segmentation using K-means clustering – GeeksforGeeks

K-Means Clustering Algorithm with Python Tutorial - YouTube

Python Machine Learning Kmeans – W3School

Applied-Mathematics-and-Statistics by NgocTien0110 - Github

Image-Compression by PhuongBui712 - GitHub