Spring 2023
CS 464 INTRODUCTION TO MACHINE LEARNING

Yagiz Yaman

2023-05-08

Homework 2

Library reticulate is going to be used to call python from the rmd file.

```
library(reticulate)
```

```
## Warning: package 'reticulate' was built under R version 4.2.3
```

```
use_python("C:/Users/yagiz/Desktop/4-2/CS-464/Homeworks/HW2/myVenv/Scripts/python.exe")
```

```
import numpy as np
from PIL import Image
import glob
import matplotlib.pyplot as plt
```

Library Used

The libraries **PIL, glob** are used to open images, **matplotlib** is used to create graphs and the library **numpy** is used to perform vectorized operations.

QUESTION 1

In this question, you are expected to analyze cat images using PCA. In this assignment, you will be working on the cat images inside both the validation and the training sets of the Animal Faces dataset. You are requested to use the combined versions of these images afhq cat.zip2 instead of combining them by yourself. The provided dataset is composed of 5653 images of dogs.

For this question, the use of any library for PCA calculations is not allowed. You are requested to implement the PCA algorithm by yourself. It is suggested to use the numpy.linalg.eig function to find the eigenvalues and the eigenvectors in your calculations.

Before the analysis, resize the images to 64x64 pixels by using the bilinear interpolation method3 implemented in the PIL library4.

```
image_directory = "C:/Users/yagiz/Desktop/4-2/CS-464/Homeworks/HW2/afhq_cat"
image_paths = glob.glob(image_directory + "/*.jpg")

resized_images = []

for path in image_paths:
    img = Image.open(path)
    img = img.resize((64, 64), Image.BILINEAR)
```

```
    # Resize the image to 64x64 pixels using bilinear interpolation
    resized_images.append(img)
```

Then, flatten all images of size 64x64x3 to obtain a 4096x3 matrix for each image. Remember that the PIL library reads the image files in uint8 format. Since unsigned integer values cannot be negative, the calculations in the following parts may fail. In such a case, the problem can be solved by converting the data type to int or float32. Note that all images are channel RGB. Create a 3-D array, X, of size 5653x4096x3 by stacking flattened matrices of the images provided in the dataset. Slice X as X_i = X[:, :, i], where i corresponds to the three indexes (0: Red, 1: Green, and 2: Blue), to obtain color channel matrix (5653x4096) of all images for each channel.

```
# flatten the resized images.
flattened_images = []

for img in resized_images:
    flattened_img = np.array(img).reshape(-1, 3)
    flattened_images.append(flattened_img)

X = np.stack(flattened_images)
# Stack the flattened images to create a 3-D array of size 5653x4096x3
```

Let's analyze **X**,

```
X.shape
```

```
## (5653, 4096, 3)
```

So,

- **X** is a 3-D array with dimensions 5653x4096x3.
- **X** has 5653 picture.
- Each picture in **X** has a total of 4096 pixels.
- Each pixel in the picture is represented by three values, which are the intensities of the red, green, and blue color channels.

Question 1.1 [30 pts]

Apply PCA on X_i's to obtain first 10 principal components for each X_i. Report the proportion of variance explained (PVE) for each of the principal components and their sum for each X_i. Discuss your results and find the minimum number of principal components that are required to obtain at least 70% PVE for all channels.

Answer 1.1

```
X_0 = X[:, :, 0]  # Extract the red channel matrix X_0 (5653x4096)
X_1 = X[:, :, 1]  # Extract the green channel matrix (5653x4096)
X_2 = X[:, :, 2]  # Extract the blue channel matrix (5653x4096)
```

Apply PCA on each color channel matrix

We are going to try to find a lower dimensional representation of X_0, X_1 and X_2. Let's consider X_0. So, we want to find a lower dimensional representation $\tilde{\tilde{x}}_i$ of X_n that can be expressed using fewer basis vectors, say M.

There are two assumptions of PCA,

- The data is centered. That means the dataset has mean 0.
- We also assume that b_1 to b_D are an orthonormal bases of R^D.

We can write $\tilde{\tilde{x_i}}$ in the following way,

$$\tilde{\tilde{x_i}} = \sum_{i=1}^{M} x_i \beta_{i,n} b_i + \sum_{i=M+1}^{D} x_i \beta_{i,n} b_i \in R^D$$

We'll ignore the second term and then we call the subspace that is spanned by the basis vectors b_1 to b_m the **principal subspace**.

Assuming we have data

$$X = \{X_1, \ldots, X_N\}, \quad X_i \in \mathbb{R}^D$$

, we want to find parameters $\beta_{i,n}$ such that the *average squared reconstruction error (J)* is minimized.

$$J = \frac{1}{N} \sum_{i=1}^{M} \|X_n - \tilde{\tilde{X}}_n\|^2$$

This is our **loss function**. Our approach is to compute the partial derivatives of **J** with respect to the parameters. The parameters are the $\beta_{i,n}$ and b_is. We will set the partial derivatives of **J** with respect to these parameters to 0 and solve for the optimal parameters.

After multiple steps of derivations, we are going to get the following equation,

$$J = \sum_{j=M+1}^{D} b_j^T (\frac{1}{N} \sum_{i=1}^{M} x_N x_N^T) b_j$$

We call

$$\frac{1}{N} \sum_{i=1}^{M} x_N x_N^T$$

this part **data covariance matrix S** because we assumed we have centered data. So,

$$J = \sum_{j=M+1}^{D} b_j^T S b_j$$

The covariance matrix is given as,

$$S = \frac{1}{N} X^T X$$

After some manipulations, we end up with the following eigenvalue - eigenvector equation,

$$S b_i = \lambda_i b_i$$

So, we calculate the eigenvector and eigenvalues of the covariance matrix.

```
def apply_pca(X_i):

    n = X_i.shape[0]
    # As we stated one of the assumptions of PCA is centered data.
    #So, we are going to center our data.
    X_i_centered = X_i - np.mean(X_i, axis=0)

    # Now we are writing the covariance matrix
    cov_matrix = (1 / n) * X_i_centered.T @ X_i_centered

    # As stated now the eigenvectors and eigenvalues of the
    #covariance matrix are going to be calculated.
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # Sort eigenvalues and eigenvectors
    sorted_indices = np.argsort(eigenvalues)[::-1]
    sorted_eigenvalues = eigenvalues[sorted_indices]
    sorted_eigenvectors = eigenvectors[:, sorted_indices]

    # Select the first 10 principal components
    principal_components = sorted_eigenvectors[:, :10]

    # Calculate PVE
    pve = sorted_eigenvalues / np.sum(sorted_eigenvalues)

    # Calculate the cumulative PVE
    cumulative_pve = np.cumsum(pve)

    return principal_components, pve, cumulative_pve, np.array(eigenvectors)
```

Apply PCA on each channel matrix

```
red_principals, red_pve, red_cumulative_pve, eigen_red = apply_pca(X_0)
green_principals, green_pve, green_cumulative_pve, eigen_green = apply_pca(X_1)
blue_principals, blue_pve, blue_cumulative_pve, eigen_blue = apply_pca(X_2)
```

```
for i in range(10):
    print(f"Red Channel, PC{i+1}: PVE = {red_pve[i]}, Cumulative PVE = {red_cumulative_pve[i]}")
    print(f"Green Channel, PC{i+1}: PVE = {green_pve[i]}, Cumulative PVE = {green_cumulative_pve[i]}")
    print(f"Blue Channel, PC{i+1}: PVE = {blue_pve[i]}, Cumulative PVE = {blue_cumulative_pve[i]}")
```

```
## Red Channel, PC1: PVE = 0.23506969936279912, Cumulative PVE = 0.23506969936279912
## Green Channel, PC1: PVE = 0.20873714854025407, Cumulative PVE = 0.20873714854025407
## Blue Channel, PC1: PVE = 0.22859035906545522, Cumulative PVE = 0.22859035906545522
## Red Channel, PC2: PVE = 0.1565111520673864, Cumulative PVE = 0.3915808514301855
## Green Channel, PC2: PVE = 0.15884565962402114, Cumulative PVE = 0.36758280816427524
## Blue Channel, PC2: PVE = 0.15649257925344165, Cumulative PVE = 0.38508293831889684
## Red Channel, PC3: PVE = 0.09005253857044325, Cumulative PVE = 0.48163339000062877
## Green Channel, PC3: PVE = 0.09258856862586831, Cumulative PVE = 0.4601713767901435
## Blue Channel, PC3: PVE = 0.08790595575693264, Cumulative PVE = 0.47298889407582945
## Red Channel, PC4: PVE = 0.06829954682854641, Cumulative PVE = 0.5499329368291752
## Green Channel, PC4: PVE = 0.0681111174610961, Cumulative PVE = 0.5282824942512396
## Blue Channel, PC4: PVE = 0.06203548174652196, Cumulative PVE = 0.5350243758223514
## Red Channel, PC5: PVE = 0.037527339511068326, Cumulative PVE = 0.5874602763402436
## Green Channel, PC5: PVE = 0.03798505275724374, Cumulative PVE = 0.5662675470084834
```

```
## Blue Channel, PC5: PVE = 0.037401342032656444, Cumulative PVE = 0.5724257178550078
## Red Channel, PC6: PVE = 0.023947539134905344, Cumulative PVE = 0.6114078154751489
## Green Channel, PC6: PVE = 0.024467317448625063, Cumulative PVE = 0.5907348644571084
## Blue Channel, PC6: PVE = 0.02416587386242639, Cumulative PVE = 0.5965915917174343
## Red Channel, PC7: PVE = 0.022764658780062626, Cumulative PVE = 0.6341724742552115
## Green Channel, PC7: PVE = 0.02427916341309129, Cumulative PVE = 0.6150140278701998
## Blue Channel, PC7: PVE = 0.024047333972009168, Cumulative PVE = 0.6206389256894435
## Red Channel, PC8: PVE = 0.021128209465633437, Cumulative PVE = 0.6553006837208449
## Green Channel, PC8: PVE = 0.021490528264504007, Cumulative PVE = 0.6365045561347038
## Blue Channel, PC8: PVE = 0.020596134585636253, Cumulative PVE = 0.6412350602750797
## Red Channel, PC9: PVE = 0.017935920584406462, Cumulative PVE = 0.6732366043052513
## Green Channel, PC9: PVE = 0.018870002898966134, Cumulative PVE = 0.6553745590336699
## Blue Channel, PC9: PVE = 0.018458994366195094, Cumulative PVE = 0.6596940546412748
## Red Channel, PC10: PVE = 0.01349360904861361, Cumulative PVE = 0.686730213353865
## Green Channel, PC10: PVE = 0.01421133520365073, Cumulative PVE = 0.6695858942373206
## Blue Channel, PC10: PVE = 0.014285720074286417, Cumulative PVE = 0.6739797747155613
```

The PVE values for the first principal component (PC1) are relatively high for all channels, ranging from approximately 0.21 to 0.24. This suggests that PC1 captures a significant portion of the variance in the data. When we move through higher-numbered PC's, the PVE values decrease.

```
def find_min_components(cumulative_pve):
    min_components = np.argmax(cumulative_pve >= 0.7) + 1
    return min_components

min_red_components = find_min_components(red_cumulative_pve)
min_green_components = find_min_components(green_cumulative_pve)
min_blue_components = find_min_components(blue_cumulative_pve)

print(f"Minimum number of Red Channel components for 70% PVE: {min_red_components}")
```

```
## Minimum number of Red Channel components for 70% PVE: 12
```

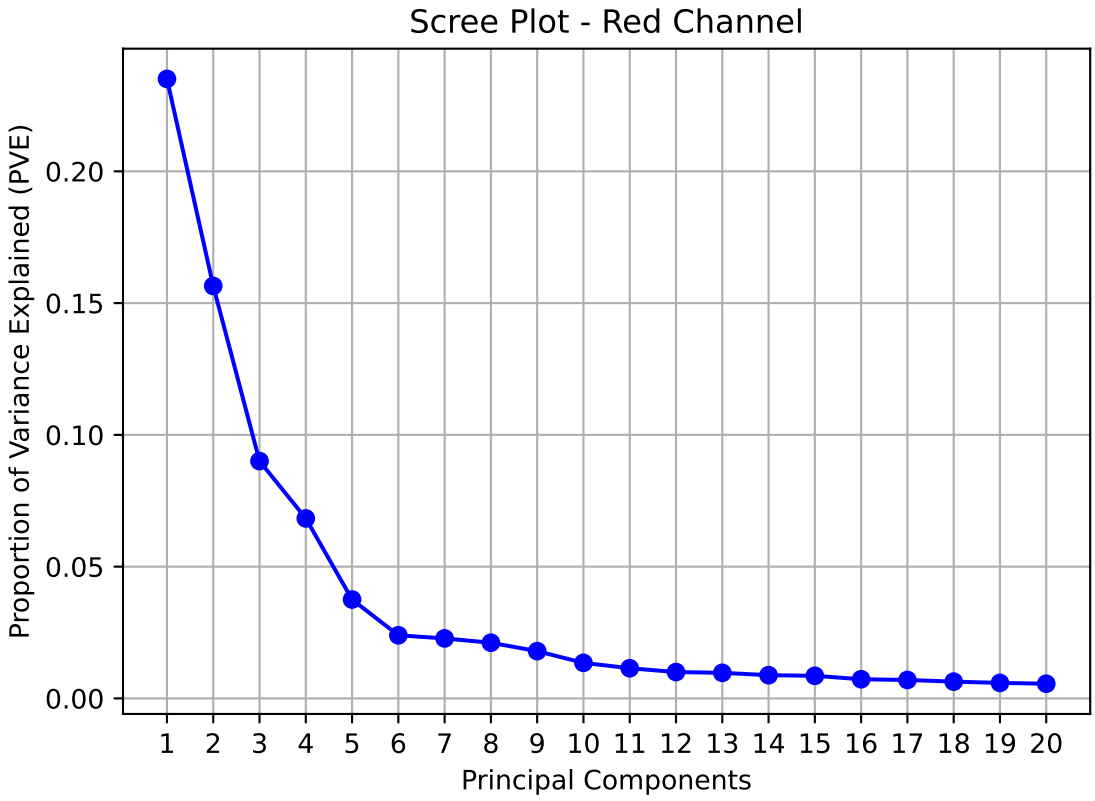
```
print(f"Minimum number of Green Channel components for 70% PVE: {min_green_components}")
```

```
## Minimum number of Green Channel components for 70% PVE: 13
```

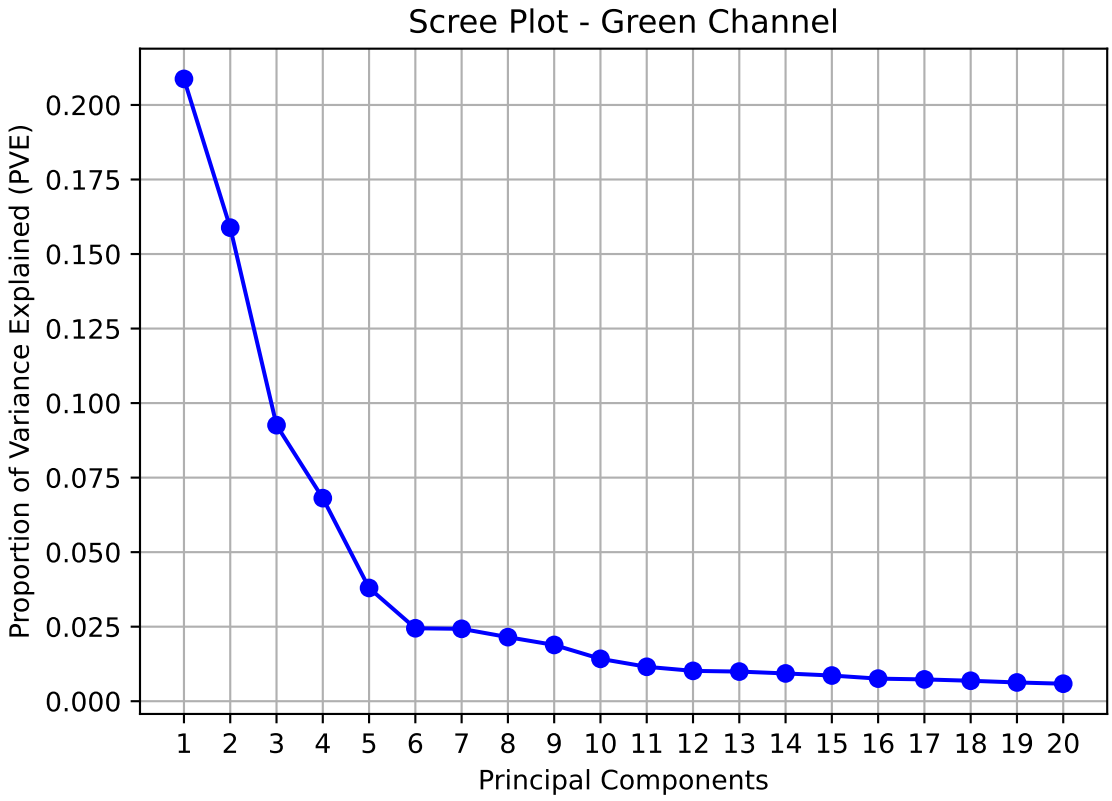
```
print(f"Minimum number of Blue Channel components for 70% PVE: {min_blue_components}")
```

```
## Minimum number of Blue Channel components for 70% PVE: 13
```

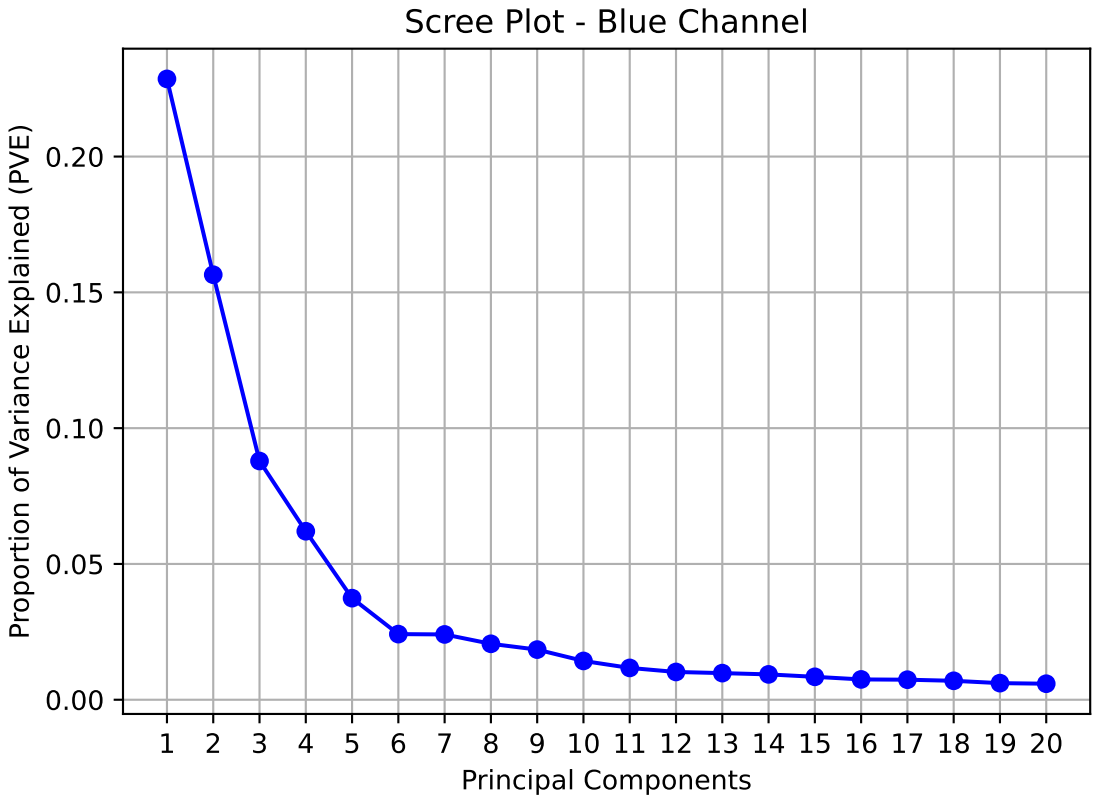
```
def draw_scree_plot(pve, channel_name, num_components):
    x = np.arange(1, num_components + 1)
    y = pve[:num_components]

    plt.plot(x, y, marker='o', linestyle='-', color='b')
    plt.xlabel('Principal Components')
    plt.ylabel('Proportion of Variance Explained (PVE)')
    plt.title(f'Scree Plot - {channel_name} Channel')
    plt.xticks(np.arange(1, num_components + 1))
    plt.grid(True)
    plt.show()

# Draw scree plots for each channel
draw_scree_plot(red_pve, 'Red', 20)
```




```
plt.clf()
draw_scree_plot(green_pve, 'Green', 20)
```




```
plt.clf()
draw_scree_plot(blue_pve, 'Blue', 20)
```



Question 1.2 [30 pts]

Using the first 10 principal components found for each color channel, reshape each principal component to a 64x64 matrix. Later, normalize the values of each of them between 0 and 1 using the min-max scaling method5 . After scaling them, stack corresponding color channels (R, G, and B) of each principal component to obtain 10 RGB images of size 64x64x3, which are the visuals of eigenvectors. Display all and discuss your results.

Answer 1.2

Reshape the principal components

```
red_images = red_principals[:, :10].reshape(-1, 64, 64)
green_images = green_principals[:, :10].reshape(-1, 64, 64)
blue_images = blue_principals[:, :10].reshape(-1, 64, 64)
```

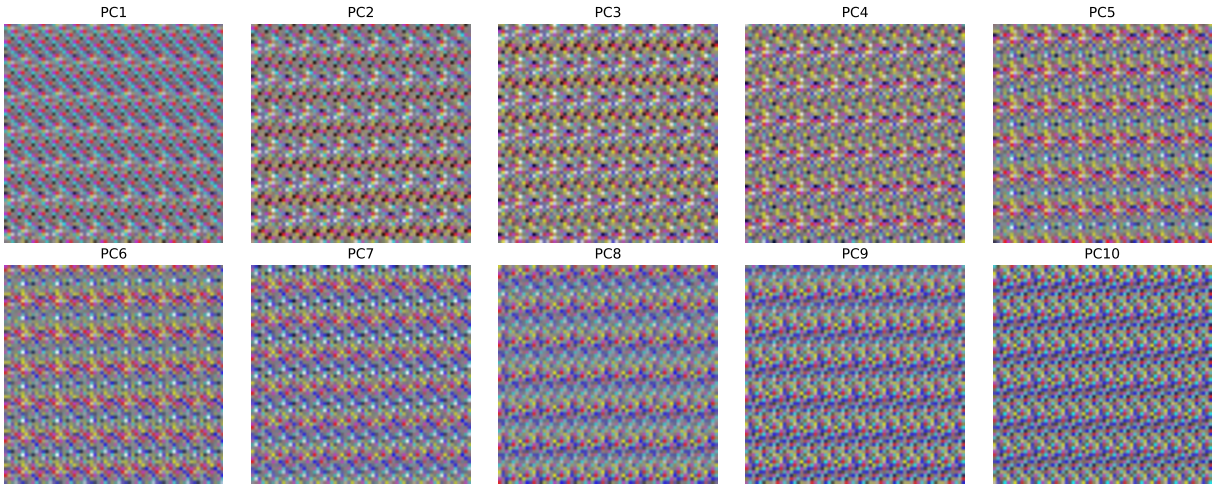
Normalize the values of each component between 0 and 1 using min-max scaling

```
red_images = (red_images - np.min(red_images)) / (np.max(red_images) - np.min(red_images))
green_images = (green_images - np.min(green_images)) / (np.max(green_images) - np.min(green_images))
blue_images = (blue_images - np.min(blue_images)) / (np.max(blue_images) - np.min(blue_images))
```

Show the 10 RGB images

```
# Stack corresponding color channels
visuals = np.stack([red_images, green_images, blue_images], axis=3)

fig, axes = plt.subplots(2, 5, figsize=(15, 6))
```

```
for i, ax in enumerate(axes.flat):
    ax.imshow(visuals[i])
    ax.axis('off')
    ax.set_title(f'PC{i+1}')
plt.tight_layout()
plt.show()
```



Question 1.3 [40 pts]

Describe how you can reconstruct a cat image using the principal components you obtained in Question 1.1. Use the first k principal components to analyze and reconstruct the second image in the dataset where;

$$k \in \{1, 50, 250, 500, 1000, 4096\}$$

In order to reconstruct an image, you should first calculate the dot product with principle components and the image. Later, you project the data you obtained back onto the original space using the first k eigenvectors. Discuss your results in the report.

Hint: Do not forget to add up the mean values at the end of the reconstruction process if you subtracted them from the data in Question 1.1 to calculate the principle components.

Answer 1.3

Load and resize the image

```
image_path = "C:/Users/yagiz/Desktop/4-2/CS-464/Homeworks/HW2/afhq_cat/flickr_cat_000003.jpg"
img = Image.open(image_path)
img = img.resize((64, 64), Image.BILINEAR)
```

Flatten the image

```
flattened_img = np.array(img).reshape(-1, 3)
```

Separate the RGB channels and center the pixel values

```
flattened_img_red = flattened_img[:, 0].astype(np.float32)
flattened_img_green = flattened_img[:, 1].astype(np.float32)
flattened_img_blue = flattened_img[:, 2].astype(np.float32)
```

```
flattened_img_red_centered = flattened_img_red - flattened_img_red.mean()
flattened_img_green_centered = flattened_img_green - flattened_img_green.mean()
flattened_img_blue_centered = flattened_img_blue - flattened_img_blue.mean()
```

Create a function to reconstruct the image.

```
def reconstruct_image(flattened_img, flattened_img_centered, eigen_vectors, k):
    reconstructed = np.dot(eigen_vectors[:k], flattened_img_centered)
    reconstructed_image = np.dot(eigen_vectors[:k].T, reconstructed) + flattened_img.mean()
    reconstructed_image = np.reshape(reconstructed_image, (64, 64))
    add_image = Image.fromarray(np.uint8(reconstructed_image))
    return add_image
```

Create the layout to show the reconstructed images together.

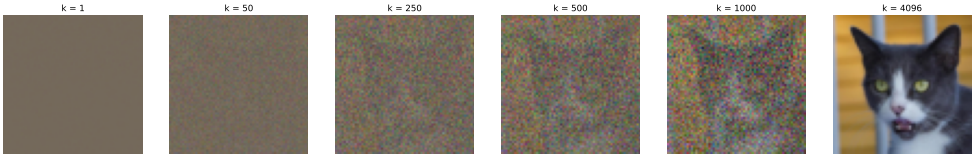
```
k_values = [1, 50, 250, 500, 1000, 4096]
num_plots = len(k_values)
fig, axes = plt.subplots(1, num_plots, figsize=(4 * num_plots, 4))
```

Reconstruct the image for each RGB channel and then merge the RGB channels back into a single image.

```
for i, k in enumerate(k_values):
    part_red = reconstruct_image(flattened_img_red, flattened_img_red_centered, eigen_red, k)
    part_green = reconstruct_image(flattened_img_green, flattened_img_green_centered, eigen_green, k)
    part_blue = reconstruct_image(flattened_img_blue, flattened_img_blue_centered, eigen_blue, k)

    reconstructed_image = np.stack((np.array(part_red), np.array(part_green), np.array(part_blue)), axis
    reconstructed_image = Image.fromarray(np.uint8(reconstructed_image))

    axes[i].imshow(reconstructed_image)
    axes[i].set_title(f"k = {k}")
    axes[i].axis('off')
```

```
## <matplotlib.image.AxesImage object at 0x000001EABD4CDF90>
## Text(0.5, 1.0, 'k = 1')
## (-0.5, 63.5, 63.5, -0.5)
## <matplotlib.image.AxesImage object at 0x000001EABD4CCB50>
## Text(0.5, 1.0, 'k = 50')
## (-0.5, 63.5, 63.5, -0.5)
## <matplotlib.image.AxesImage object at 0x000001EABD4CFE80>
## Text(0.5, 1.0, 'k = 250')
## (-0.5, 63.5, 63.5, -0.5)
## <matplotlib.image.AxesImage object at 0x000001EABD4CFB50>
## Text(0.5, 1.0, 'k = 500')
## (-0.5, 63.5, 63.5, -0.5)
## <matplotlib.image.AxesImage object at 0x000001EABCDB8C40>
## Text(0.5, 1.0, 'k = 1000')
## (-0.5, 63.5, 63.5, -0.5)
## <matplotlib.image.AxesImage object at 0x000001EABCDF16C0>
## Text(0.5, 1.0, 'k = 4096')
## (-0.5, 63.5, 63.5, -0.5)
```

```
plt.show()
```



The result is expected as when we use higher number of principal components, we expect the reconstructed image to be closer to the original image as it retains more information.