MAT161 - Applied Linear Algebra

Facial Recognition using Eigenfaces

Lalit Maurya 2310110164

Introduction

The need for facial recognition has become increasingly important in various domains, from security to personal devices. At its core, facial recognition heavily relies on mathematics, particularly eigenvectors and eigenvalues. These allow us to extract the essential facial features of an individual and compare it. With the analysis of the eigenvectors generated from a dataset, we can construct a model capable of recognizing and distinguishing faces.

In this report, I present my implementation of a facial recognition system using eigenfaces. My implementation consists of two parts **TRAINING** and **TESTING**.

- **TRAINING**
  I am using the LFWcrop greyscale dataset consisting of 13,233 images of various individuals. In the training phase, we leverage linear algebra to perform Principal Component Analysis (PCA) on our training data. This allows us to reduce the the data to a smaller set while still maintaining significant patterns and trends.
- **TESTING**
  In the testing phase, weights corresponding to an input image is calculated, after which the euclidean distances are calculated. A threshold distance is defined, above which the program displays “Unknown”

Mathematics

Let us take a look at the mathematics behind the eigenfaces approach. In particular, we will look at the algorithm behind Principal Component Analysis (PCA) in the context of images.

Terms used:

- **N** - Order of the matrix. A grayscale image is a matrix consisting of entries that correspond with brightness values of each pixel. For an image of square resolution, this matrix is a square matrix of the same order as the resolution of our image.
- **M** - Total number of images in our dataset.

To begin with, we take each matrix (synonymous with image) of size $(N \times N)$ and flatten it to a matrix of size $(1 \times N^2)$. After this is done for all matrices in our dataset, we stack them to get a **faces_matrix** of size $(M \times N^2)$.

We then calculate the **mean_face** by taking the mean of every column in our **faces_matrix**. This matrix has a size of $(1 \times N^2)$.

Then we proceed to subtract this **mean_face** with each row of our **faces_matrix** to normalize our data. This gives us **normalized_faces** of size $(M \times N^2)$.

We then calculate the **covariance_matrix** of the **normalized_faces** matrix. This will give us a matrix of size $(N^2 \times N^2)$. The covariance matrix represents the relationships between features of images.

PCA aims to find a new set of orthogonal axes (principal components) such that the variance along these axes is maximized. These principal components are the eigenvectors of the covariance matrix, and their corresponding eigenvalues represent the amount of variance explained by each principal component. Thus we calculate the **eigenvectors** and **eigenvalues** of our **covariance_matrix**. These **eigenvectors** are termed **eigenfaces**.

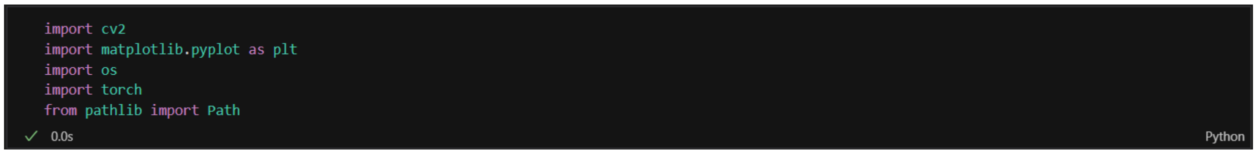
The top **eigenfaces** represent the most important characteristics of our faces, concentrating on changes in these characteristics will result in the most efficient way to tell apart individual faces. Thus we have used PCA to reduce the set of data while still preserving the most important characteristics and trends.

These **eigenfaces** are then used to reconstruct images, and calculate euclidean distances to best match a given face with one in our dataset.

Code

Let us now take a look at the practical implementation of eigenfaces. I have added snippets of code in this report. The whole project can be found at this Github Repository.

- Importing Libraries



```
import cv2
import matplotlib.pyplot as plt
import os
import torch
from pathlib import Path
```

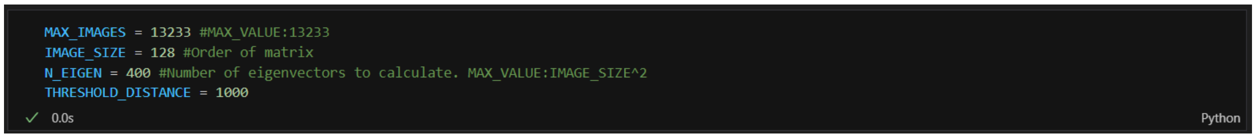
✓ 0.0s Python

- Parameters

These allow us to change the model we create based on our requirements.

- MAX_IMAGES defines the number of images in our dataset
- IMAGE_SIZE defines the resolution each image gets resized to.
- N_EIGEN defines the number of top eigenvectors to calculate

- THRESHOLD_DISTANCE defines the euclidean distance above which any displayed image is said to be “NO MATCH”

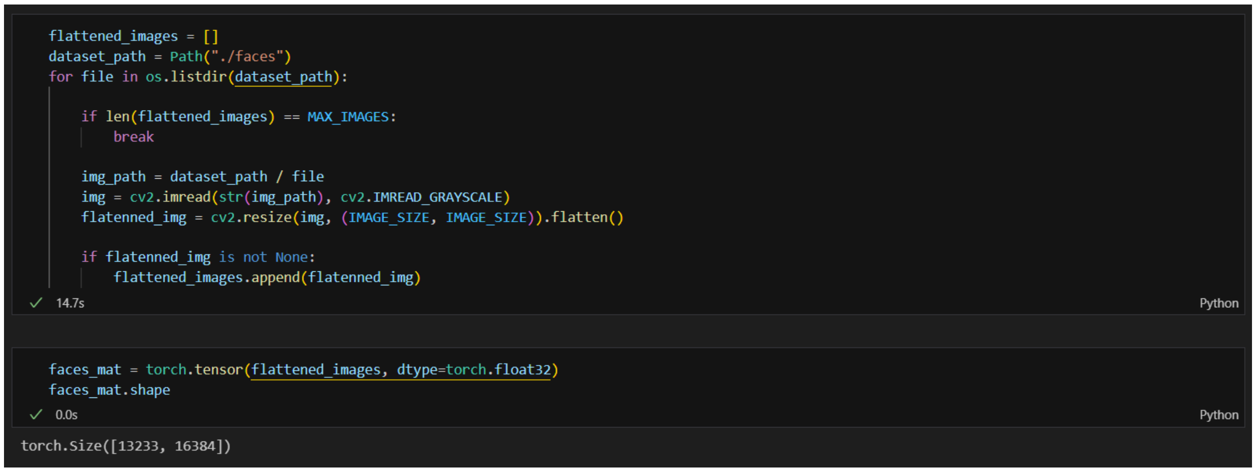


```
MAX_IMAGES = 13233 #MAX_VALUE:13233
IMAGE_SIZE = 128 #Order of matrix
N_EIGEN = 400 #Number of eigenvectors to calculate. MAX_VALUE:IMAGE_SIZE^2
THRESHOLD_DISTANCE = 1000
✓ 0.0s                                                                    Python
```

- TRAINING PHASE

- Creating faces_mat

Here we load up every image in our dataset and flatten it. Each image is of size (128 x 128) and there are 13,233 such images. Therefore our faces_mat has a size of (13,233 x 16,384). We also convert our matrix into a tensor to utilize the pytorch library for efficient matrix calculations.

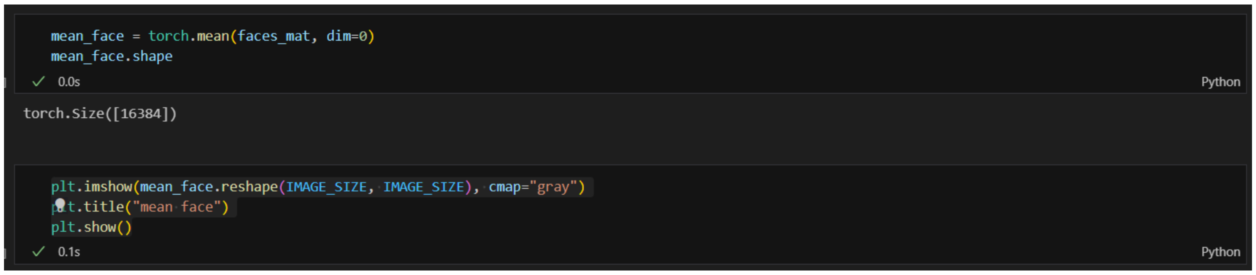


```
flattened_images = []
dataset_path = Path("./faces")
for file in os.listdir(dataset_path):

    if len(flattened_images) == MAX_IMAGES:
        break

    img_path = dataset_path / file
    img = cv2.imread(str(img_path), cv2.IMREAD_GRAYSCALE)
    flatenned_img = cv2.resize(img, (IMAGE_SIZE, IMAGE_SIZE)).flatten()

    if flatenned_img is not None:
        flattened_images.append(flatenned_img)
✓ 14.7s                                                                   Python
```

```
faces_mat = torch.tensor(flattened_images, dtype=torch.float32)
faces_mat.shape
✓ 0.0s                                                                    Python
torch.Size([13233, 16384])
```

- Calculating mean_face and displaying it

The following snippet of code calculates the mean face by taking the mean of every column in faces_mat. This is then reshaped into a (16,384 x 16,384) image and displayed using matplotlib.

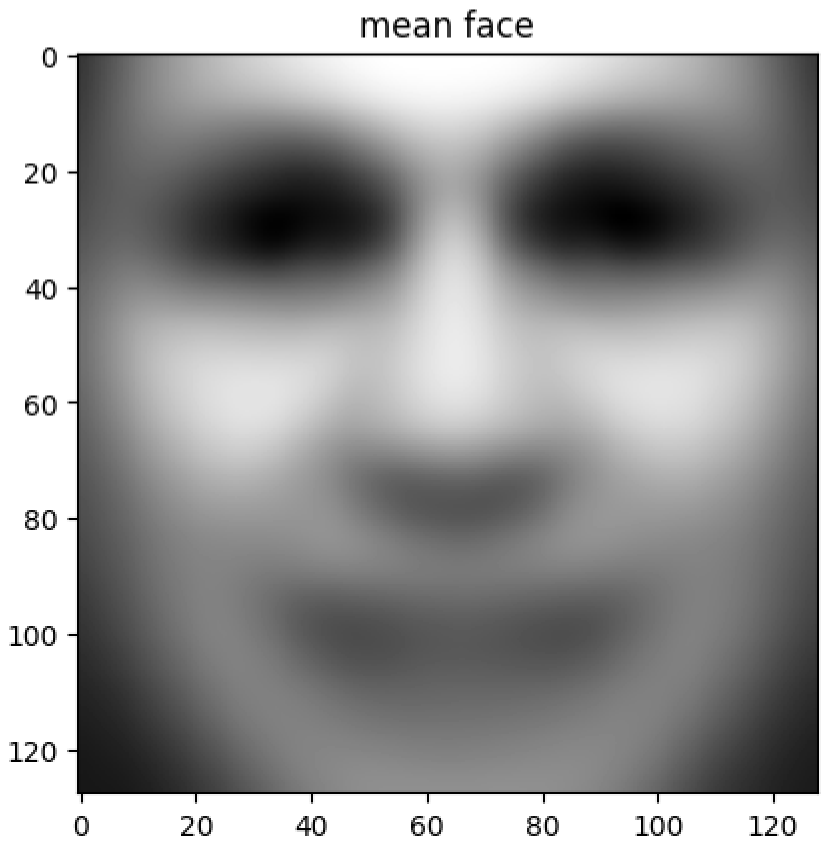


```
mean_face = torch.mean(faces_mat, dim=0)
mean_face.shape
```
✓ 0.0s Python

```
torch.Size([16384])
```

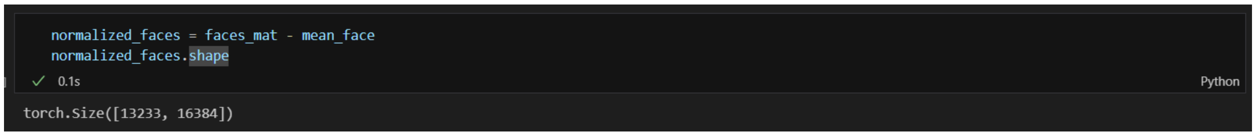
```
plt.imshow(mean_face.reshape(IMAGE_SIZE, IMAGE_SIZE), cmap="gray")
plt.title("mean face")
plt.show()
```
✓ 0.1s Python

The following is the average face of all 13,233 images in our dataset



- Normalizing our faces

As discussed earlier, we normalize our faces_mat by subtracting the mean face from each row in face_mat. This gives us normalized_faces with size (13,233 x 16,384)

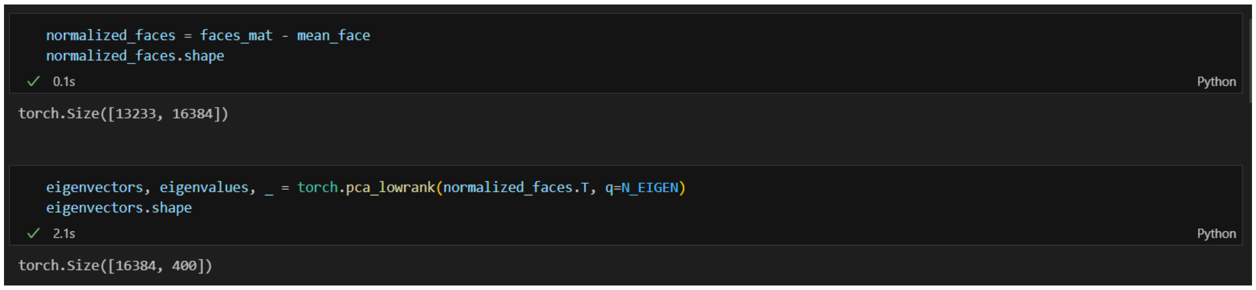


```
normalized_faces = faces_mat - mean_face
normalized_faces.shape
```
✓ 0.1s Python

```
torch.Size([13233, 16384])
```

- Calculating the covariance matrix and eigenvalues, eigenvectors.

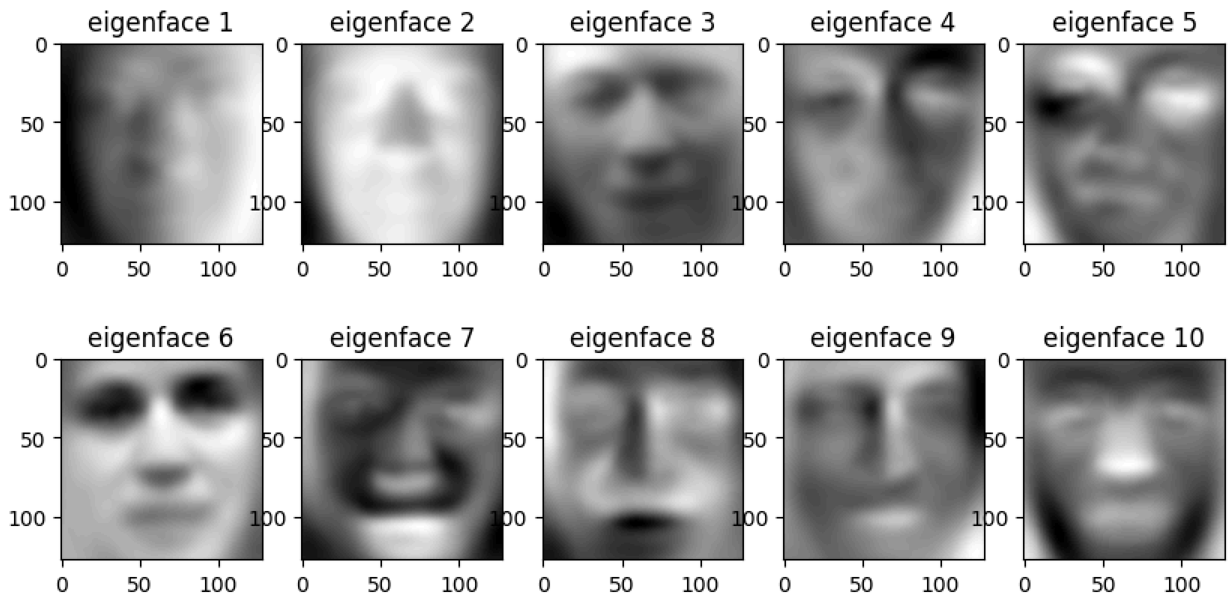
Since our dataset is so large, I used the inbuilt torch.pca_lowrank function that calculates the top q eigenvectors. This is more optimized than calculating every eigenvector using torch.eig. The end result is the same in both cases, one is more efficient than the other.



```
normalized_faces = faces_mat - mean_face
normalized_faces.shape
✓ 0.1s                                                    Python
torch.Size([13233, 16384])
```

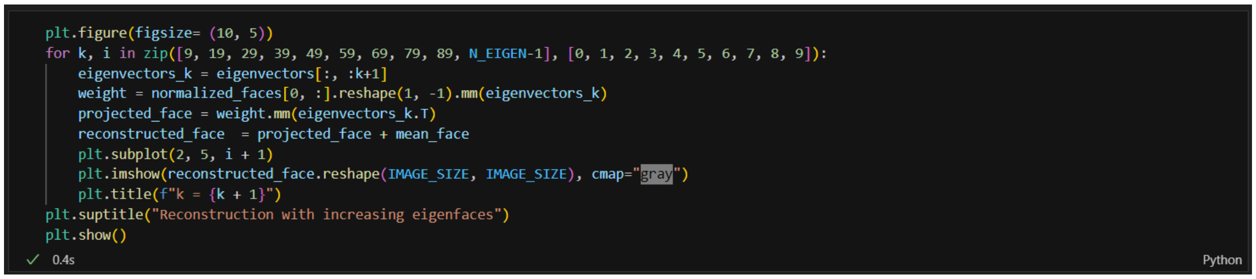
```
eigenvectors, eigenvalues, _ = torch.pca_lowrank(normalized_faces.T, q=N_EIGEN)
eigenvectors.shape
✓ 2.1s                                                    Python
torch.Size([16384, 400])
```

Here we get 400 eigenvectors as defined in the parameters. These vectors are termed eigenfaces. Below are the top 10 eigenfaces. The rank of an eigenface is determined by the magnitude of its corresponding eigenvalue.

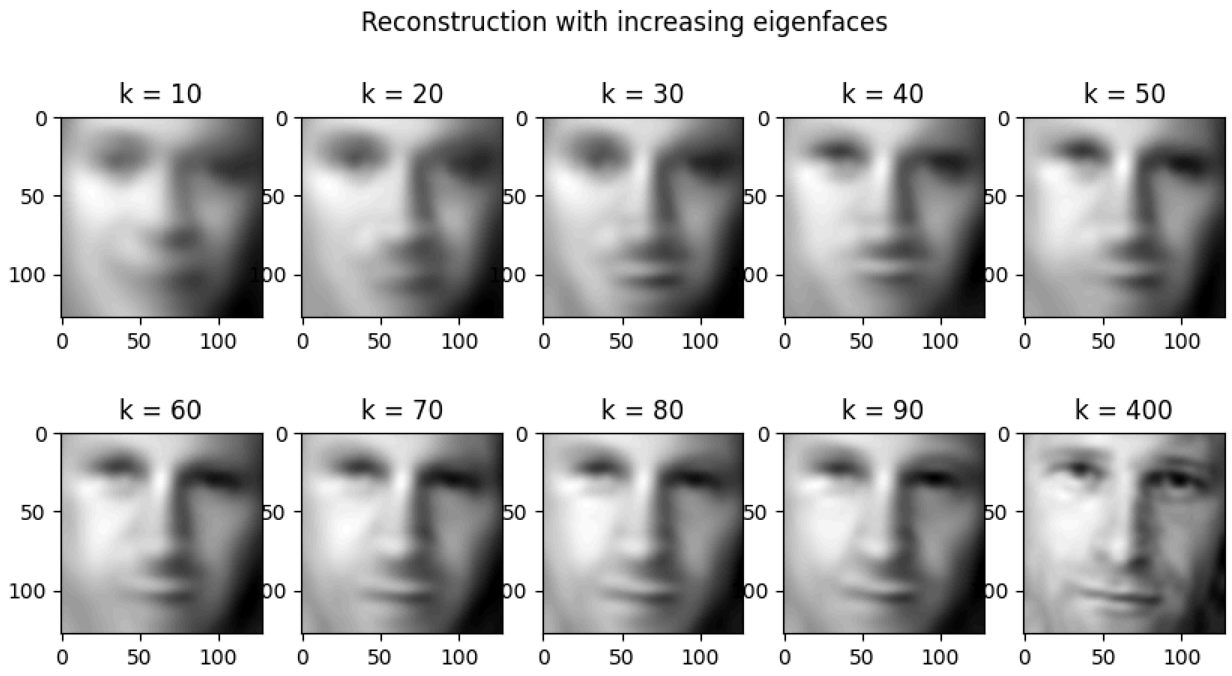


- Reconstruction of an Image with increasing eigenfaces

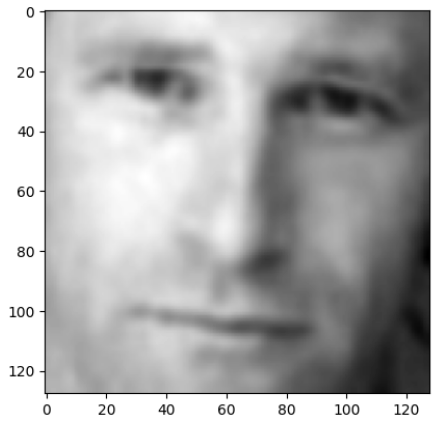
We now try to reconstruct the face by taking the dot product of the normalized input face with the first k eigenfaces. We increase the value of k gradually. The reconstructed image for each value of k is then displayed.



```python
plt.figure(figsize= (10, 5))
for k, i in zip([9, 19, 29, 39, 49, 59, 69, 79, 89, N_EIGEN-1], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]):
    eigenvectors_k = eigenvectors[:, :k+1]
    weight = normalized_faces[0, :].reshape(1, -1).mm(eigenvectors_k)
    projected_face = weight.mm(eigenvectors_k.T)
    reconstructed_face  = projected_face + mean_face
    plt.subplot(2, 5, i + 1)
    plt.imshow(reconstructed_face.reshape(IMAGE_SIZE, IMAGE_SIZE), cmap="gray")
    plt.title(f"k = {k + 1}")
plt.suptitle("Reconstruction with increasing eigenfaces")
plt.show()
```



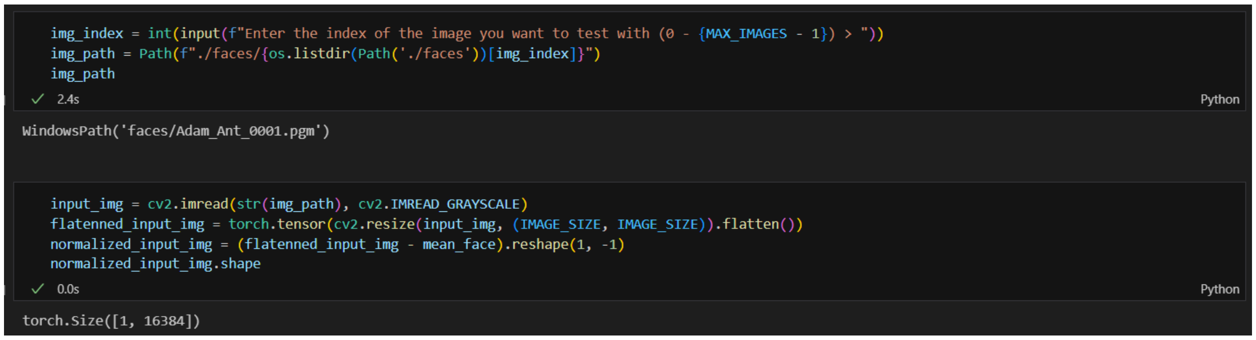
For comparison, the original image looks as follows.



- TESTING PHASE

- User Input

The user is prompted for an input for the image they want to run recognition on. The image is loaded up, flattened and normalized.



```
img_index = int(input(f"Enter the index of the image you want to test with (0 - {MAX_IMAGES - 1}) > "))
img_path = Path(f"./faces/{os.listdir(Path('./faces'))[img_index]}")
img_path
```
✓ 2.4s Python

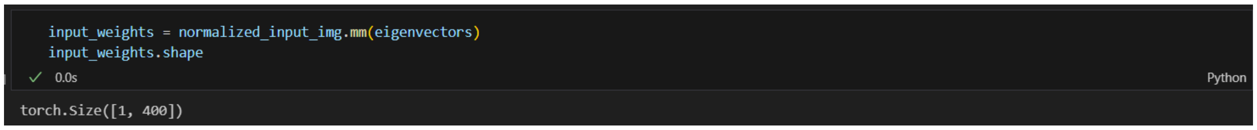
```
WindowsPath('faces/Adam_Ant_0001.pgm')
```

```
input_img = cv2.imread(str(img_path), cv2.IMREAD_GRAYSCALE)
flatenned_input_img = torch.tensor(cv2.resize(input_img, (IMAGE_SIZE, IMAGE_SIZE)).flatten())
normalized_input_img = (flatenned_input_img - mean_face).reshape(1, -1)
normalized_input_img.shape
```
✓ 0.0s Python

```
torch.Size([1, 16384])
```

- Calculating weights of the input image

We calculate the weights by taking a dot product with the eigenvectors

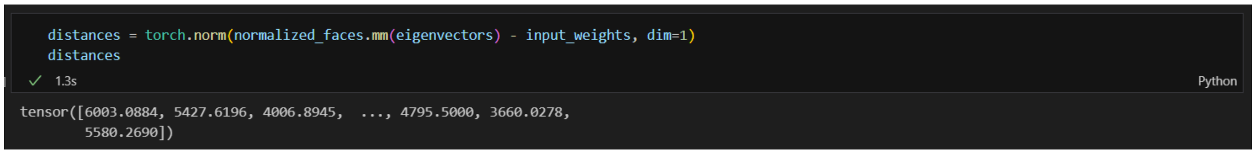


```
input_weights = normalized_input_img.mm(eigenvectors)
input_weights.shape
```
✓ 0.0s Python

```
torch.Size([1, 400])
```

- Calculating the distances

The euclidean distances is calculated between input weights and the weights of training faces.



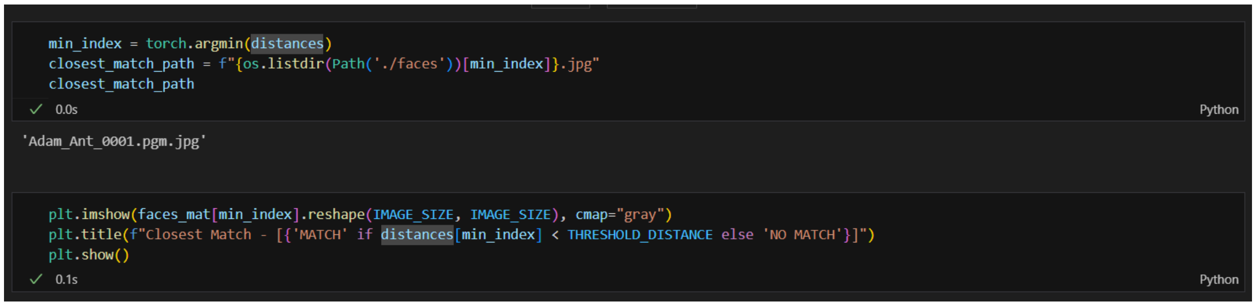
```
distances = torch.norm(normalized_faces.mm(eigenvectors) - input_weights, dim=1)
distances
```
✓ 1.3s Python

```
tensor([6003.0884, 5427.6196, 4006.8945,  ..., 4795.5000, 3660.0278,
        5580.2690])
```

- Identify closest match and display image

Here we find the closest distance and display the corresponding image, we also make sure to tag the output with whether it is a "MATCH" or "NO MATCH". This is decided based on the

THRESHOLD_DISTANCE defined in the parameters



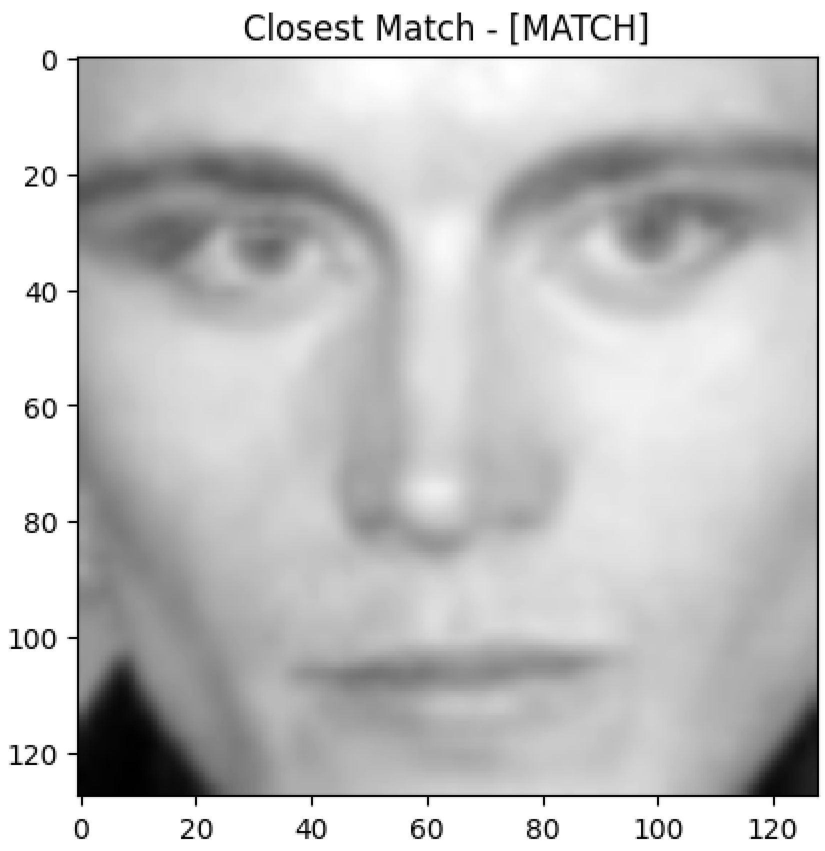
```
min_index = torch.argmin(distances)
closest_match_path = f"{os.listdir(Path('./faces'))[min_index]}.jpg"
closest_match_path
```

✓ 0.0s Python

```
'Adam_Ant_0001.pgm.jpg'
```

```
plt.imshow(faces_mat[min_index].reshape(IMAGE_SIZE, IMAGE_SIZE), cmap="gray")
plt.title(f"Closest Match - [{'MATCH' if distances[min_index] < THRESHOLD_DISTANCE else 'NO MATCH'}]")
plt.show()
```

✓ 0.1s Python



In this case, we have found a match. In the event of no matches being found. The program would still display the closest image, but tag it with “NO MATCH”.

Conclusion

This report has demonstrated a simple implementation of facial recognition using eigenfaces. With the principles of linear algebra we can perform Principle Component Analysis to effectively reduce the set of our data while preserving the important characteristics and trends.

By delving into the training and testing of my model, I have explained the concepts of Principle Component Analysis (PCA) and the power of eigenvectors and eigenvalues, as well as the use of euclidean distances in determining the best match of an input image.

In conclusion, the study of linear algebra allows us to find patterns within our complex dataset. These patterns can then be used to implement robust facial recognition systems.

Acknowledgement

I would like to express my sincere gratitude and heartfelt appreciation to my professor, ***Dr. Santosh Singh***, for his valuable guidance and support throughout this course. His expertise in the field of Mathematics has been instrumental in shaping my understanding of this course and the implementation of the facial recognition algorithm. I am thankful for his patience and encouragement which has significantly enriched my learning experience and made this semester a memorable one.

Bibliography

1. Turk, Matthew, and Alex Pentland. "Eigenfaces for Recognition." Journal of Cognitive Neuroscience, vol. 3, no. 1, pp. 71-86, 1991.
2. LFWcrop Dataset.
3. Miscellaneous resources found on the Internet.