Eigenfaces - Facial Recognition

Project Report for MAT161

Pustak Pathak | 2310110564

May 10, 2024

Introduction

Performing tasks that are seemingly simple, to humans often pose challenges for computers and machines. One such task that stands out is **facial recognition**. While humans excel at recognizing and distinguishing thousands of faces with accuracy and speed, teaching machines to do the same has proven to be quite a task. Facial recognition goes beyond spotting a face in an image; it involves matching that face to a known individual from a database. This intricate process requires machines to extract features from the data and compare them effectively against stored information.

One popular method used for recognition is using **eigenfaces**, which revolves around identifying the **eigenvectors** of the scatter matrix derived from adjusted image data. In this approach, facial images undergo preprocessing to transform them into sets of vectors. The average vector across all these image vectors is then determined and subtracted from each vector resulting in adjusted data based on the values. Subsequently, the **scatter matrix** of this modified data is. Its eigenvectors are obtained through eigendecomposition. These special vectors, referred to as eigenfaces encapsulate the components or defining characteristics, within the facial image dataset.

For this project, I have made a custom dataset of **1600 distinct images** and tested the accuracy of the technique on that.

By representing any image as a linear combination of these eigenfaces it can be projected onto what's known as the eigenface space.

The reduced version of this can be compared to the saved projections of people, for identification reasons.

Through our course in **Applied Linear Algebra**, I feel I have picked up a thorough enough understanding of the subject and its usage to put this method into practice and test its efficacy in a real use case. That will be the objective of this report.

The Mathematics

Before getting into the application of the project, it would be wise to shed some light on what is actually happening behind the scenes. We start with the images themselves.

The Images

The project and the algorithm itself revolve around the images. However, computers can not see photos. So we need to reduce images into data they can understand, and we can practice our knowledge of linear algebra on.

Suppose there are m number of n x n grayscale face images *(Strictly face images. Positioning of each face must be consistent between images.)*.

- Each of the images can be represented as an n x n matrix

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & . & . & . & x_{1n} \\ x_{21} & x_{22} & . & . & . & x_{2n} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ x_{n1} & x_{n2} & . & . & . & x_{nn} \end{bmatrix}$$

or, equivalently each of them can be represented as an n2-vector where each of the columns are stacked below the previous columns as such :

$$\mathbf{x} = \begin{bmatrix} x_{11} \\ . \\ . \\ . \\ x_{n1} \\ x_{12} \\ . \\ . \\ . \\ x_{nn} \end{bmatrix}$$

• Now, all the m images can be represented as an n2 x m matrix A where the superscript (i) will be used to denote the i-th image. Hence, i ∈ 1, . . . , m.

$$\mathbf{A} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & . & . & . & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & . & . & . & x_2^{(m)} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ x_{n^2}^{(1)} & x_{n^2}^{(2)} & . & . & . & x_{n^2}^{(m)} \end{bmatrix}$$

Facial Recognition

It is now entirely possible to perform facial recognition 1 by comparing the distance between the given face image x hat and each image x(i), from the dataset.

$$d_i = \| \hat{\boldsymbol{x}} - \mathbf{x}^{(i)} \|$$

Whenever the value of d is smaller than a chosen threshold α, the given face image can be identified as the person that is associated with the i-th face image. However, the vectors that represent the face images often have large dimensionality (n^2-dimensional vectors).

Even if the face images are relatively small in size such as **256 by 256 pixels**, the vectors would have a dimension of **65,536** or equivalently, a point in **65,536-dimensional space**. Performing computations on these huge dimensional vectors would be time-consuming and computationally expensive. Hence, the need to perform facial recognition using other methods is needed.

Therefore, we have to employ **Singular Value Decomposition (SVD)** to obtain the eigenvalues and eigenvectors. SVD decomposes the **covariance matrix** into the product of three matrices, wherein the middle matrix consists of the eigenvectors. The program then utilizes these eigenfaces to reconstruct a face by selecting a subset of eigenvectors and weighting them with coefficients calculated from the input face. This **reconstruction process** is based on the linearity property of eigenvectors, allowing for efficient representation and reconstruction.

Code

Since it is increasingly difficult to explain the process with just an explanation, the code and the process of execution that follows are something I hope will be of aid to this project report.

1) Gathering the Images

For this project, I decided to obtain a custom dataset of images, instead of using something already existing. I used a GAN (Generative Adversarial Network) application called **StyleGAN2**, developed by NVIDIA, to obtain **1600 images** of faces that are unique and have never been used before. We start with the data collection-

main2.py

```python
from PIL import Image
import shutil
import os
from thispersondoesnotexist import get_online_person #imports


for i in range(1600):
    picture = get_online_person()  # bytes representation of the image

    # Save to a file

    print("Saving picture...", i, end="\r")

    from thispersondoesnotexist import save_picture


    save_picture(picture, str(i)+".jpeg") # saves image to file

    # Open the image
    image_path = str(i) + ".jpeg"
    image = Image.open(image_path)

    # Rescale the image to 100x100
    rescaled_image = image.resize((100, 100))

    # Convert the image to grayscale
    grayscale_image = rescaled_image.convert("L")

    # Save the grayscale image
    grayscale_image.save(image_path)

    # Move the image to the faces subfolder
    shutil.move(image_path, "faces/" + image_path)
```

After the data collection, we get 1600 fresh unique images. A few of these are displayed using the following code below-

faces.ipynb

```
import matplotlib.pyplot as plt
import numpy as np
import os
from PIL import Image
plt.rcParams['figure.figsize'] = [10, 10]
plt.rcParams.update({'font.size': 18})

m = 100
n = 100

allPersons = np.zeros((n*6, m*6))
count = 0

for j in range(6):
    for k in range(6):
        img_path = f"../faces/{count}.jpeg"
        img = Image.open(img_path).convert('L')
        img_array = np.array(img)
        allPersons[j*n : (j+1)*n, k*m : (k+1)*m] = img_array
        count += 1

img = plt.imshow(allPersons)
img.set_cmap('gray')
plt.axis('off')
plt.show()
```

The code to show the first 36 images.

The first 36 generated images

Then, we move on to the facial recognition.

Preparing the environment

First, we need to prepare the environment and some auxiliary checks before we start doing our calculations. First, we set up our imports.

Eigenface.ipynb

```
import os
import glob
from sklearn import preprocessing
import cv2
import numpy as np
import matplotlib.pyplot as plt
import math
```

Then, we set our path to the images and define a function to plot said images for evaluation.

```
Eigenface.ipynb
dataset_path = os.path.join('..', 'faces')  # path to the dataset

# function to plot the images
def plot_portraits(images, titles, h, w, n_row, n_col):
    plt.figure(figsize=(2.2 * n_col, 2.2 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.20)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i])
        plt.xticks(())
        plt.yticks(())
```

Function to display 20 images. Used throughout the project.

Then, we verify the total number of images.

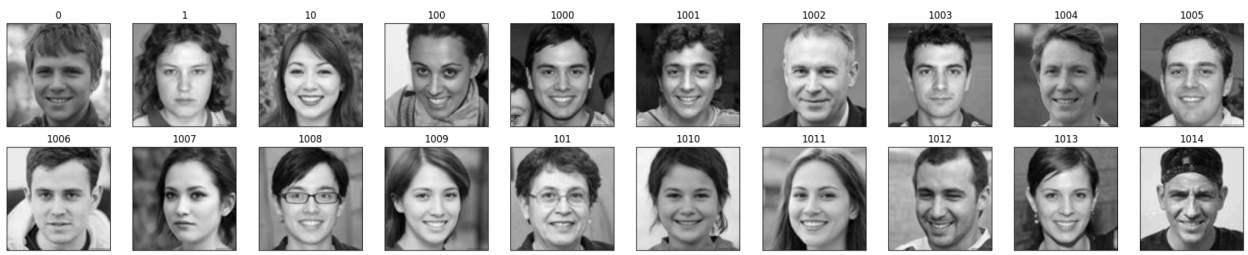
```
Eigenface.ipynb
# get the total number of images
total_images = 0
for image_file in glob.glob(os.path.join(dataset_path, '*.jpeg')):
    total_images += 1

print(f"Total images: {total_images}")
```

```
Total images: 1687
```

Total images used.

Then, we read all of the images and add them to an array, after which we verify it by plotting 20 of them.

```
Eigenface.ipynb
shape = (100, 100)  # size of the images
all_images = np.zeros((total_images, shape[0], shape[1]), dtype='float64')  # initialize the numpy array
names = []
i = 0
for image_file in glob.glob(os.path.join(dataset_path, '*.jpeg')):
    names.append(os.path.basename(image_file)[:-5])  # get the filename without extension
    read_image = cv2.imread(image_file, cv2.IMREAD_GRAYSCALE)  # read the image in grayscale
    resized_image = cv2.resize(read_image, (shape[1], shape[0]))  # cv2.resize resizes an image into (# column x # height)
    if i < total_images:
        all_images[i] = np.array(resized_image)
    i += 1

plot_portraits(all_images, names, 100, 100, 2, 10)  # plot 20 of the images
```

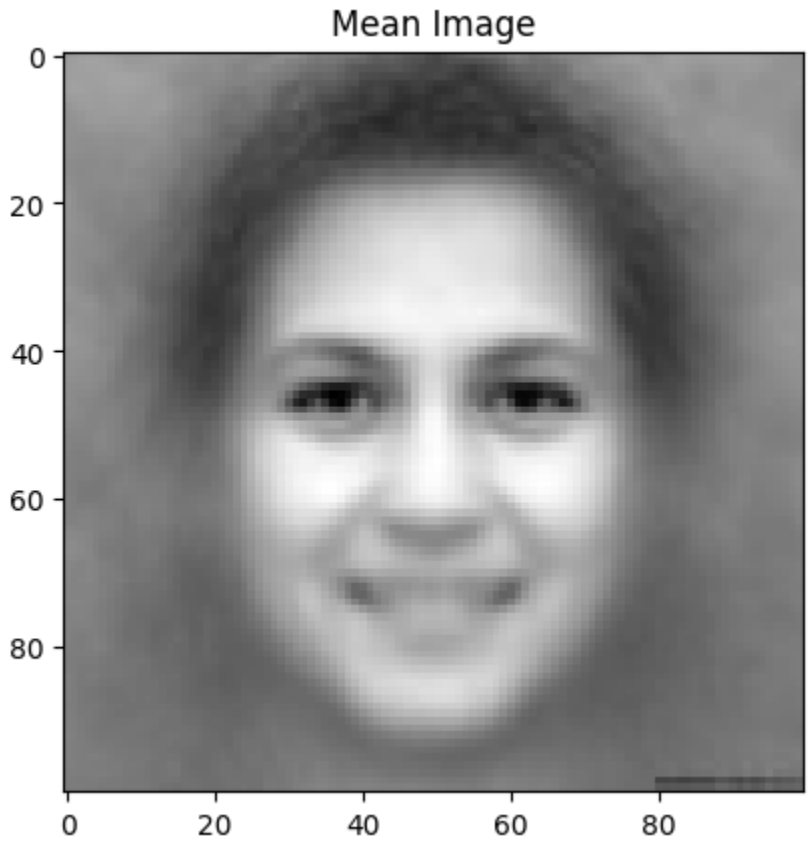


20 unique test images

Calculating the Mean Face

```
Eigenface.ipynb
A = np.resize(all_images, (total_images, shape[0]*shape[1])) #convert the images into vectors. Each row has an image vector. i.e.
samples x image_vector matrix
mean_vector = np.sum(A, axis=0, dtype='float64')/total_images #calculate the mean vector
mean_matrix = np.tile(mean_vector, (total_images, 1)) #make a 400 copy of the same vector. 400 x image_vector_size matrix.
A_tilde = A - mean_matrix #mean-subtracted image vectors
plt.imshow(np.resize(mean_vector, (shape[0],shape[1])), cmap='gray') #show the mean image vector
plt.title('Mean Image')
plt.show()
```

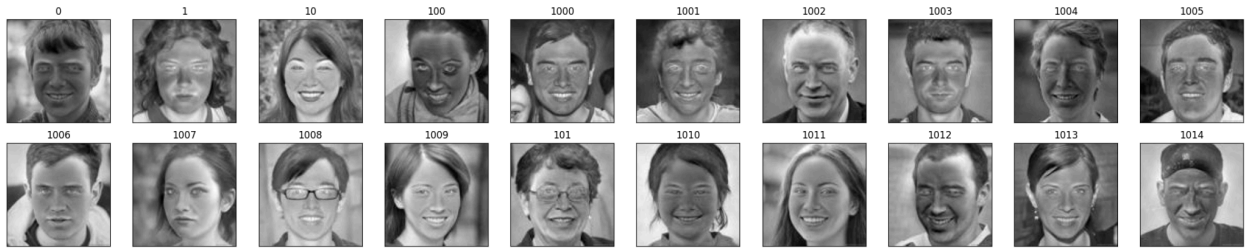
Code to calculate the mean face across 1680 images



Resultant mean face (also referred to as average face)

After this, we reprint the 20 images we printed earlier but with the mean face subtracted from the original,

```
Eigenface.ipynb
plot_portraits(A_tilde, names, 100,100, 2, 10) # matrix A_tilde that contains vectors of each mean subtracted img
```



Images with mean face subtracted

Calculating the eigenvalues and preprocessing eigenfaces

```
Eigenface.ipynb
L = (A_tilde.dot(A_tilde.T))/total_images #since each row is an image vector and we want to find the covariance matrix, we multiply
A_tilde with its transpose.
print("L shape : ", L.shape)
eigenvalues, eigenvectors = np.linalg.eig(L) #find the eigenvalues and the eigenvectors of L
idx = eigenvalues.argsort()[::-1] #get the indices of the eigenvalues by its value. Descending order.
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx] #sorted eigenvalues and eigenvectors in descending order
eigenvectors_C = A_tilde.T @ eigenvectors #linear combination of each column of A_tilde
eigenvectors_C.shape #each column is an eigenvector of C where C = (A_tilde.T)(A_tilde). Eigenvectors of C are the eigenvectors of
L
#normalize the eigenvectors
eigenfaces = preprocessing.normalize(eigenvectors_C.T) #normalize only accepts matrix with n_samples, n_feature. Hence the
transpose.
eigenfaces.shape
```

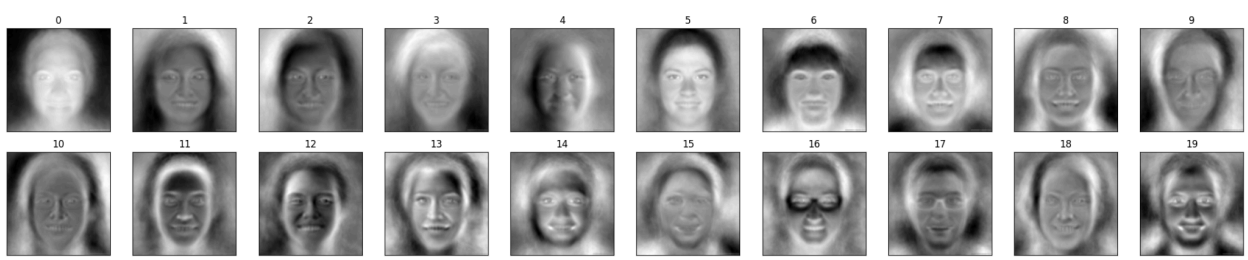
```
L shape :  (1687, 1687)

(1687, 10000)
```

Preprocessing data

Displaying a few eigenfaces

Now, we visualise the first 20 eigenfaces.

```
Eigenface.ipynb
#to visualize some of the eigenfaces
eigenface_labels = [x for x in range(eigenfaces.shape[0])] #list containing values from 1 to number of eigenfaces
plot_portraits(eigenfaces, eigenface_labels , 100,100, 2, 10)
```

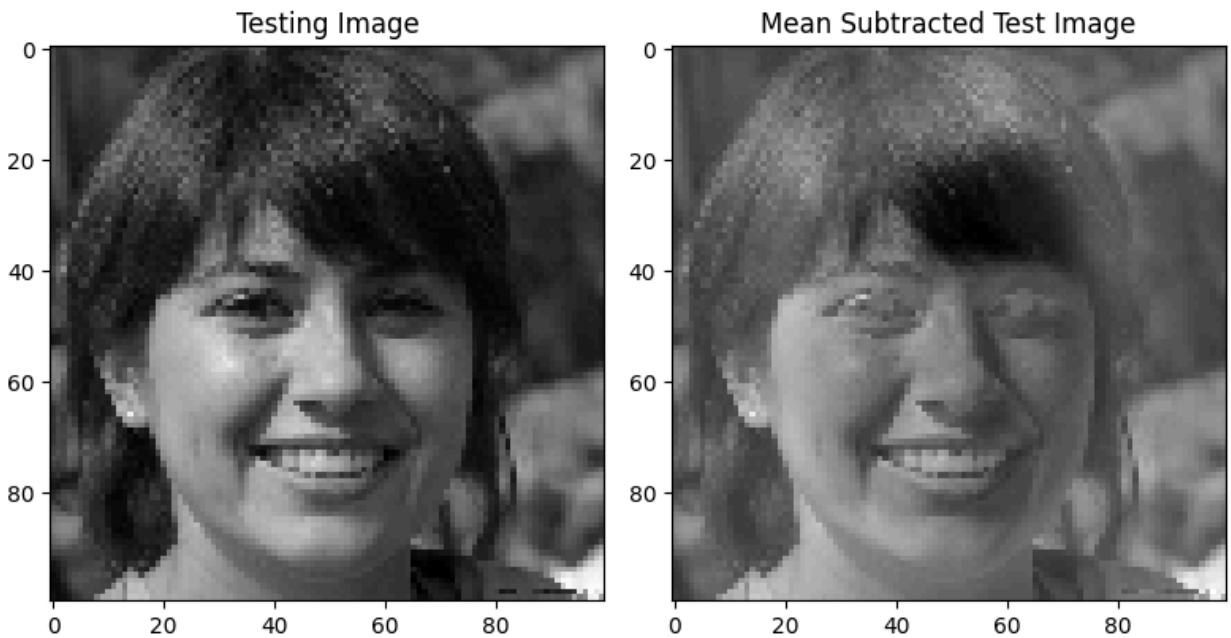


First 20 eigenfaces

Calculating mean subtraction on a test image.

First, I take a test image and calculate its mean-subtracted values.

```
Eigenface.ipynb
fig, axs = plt.subplots(1, 2, figsize=(8, 4))

test_img = cv2.imread('../1.jpeg', cv2.IMREAD_GRAYSCALE) #testing image
test_img = cv2.resize(test_img, (shape[1],shape[0])) #resize the testing image. cv2 resize by width and height.
axs[0].imshow(np.reshape(test_img, (100,100)), cmap='gray')
axs[0].set_title("Testing Image")
mean_subracted_testimg = np.reshape(test_img, (test_img.shape[0]*test_img.shape[1])) - mean_vector #subtract the mean
axs[1].imshow(np.reshape(mean_subracted_testimg, (100,100)), cmap='gray')
axs[1].set_title("Mean Subtracted Test Image")

plt.tight_layout()
plt.show()
```

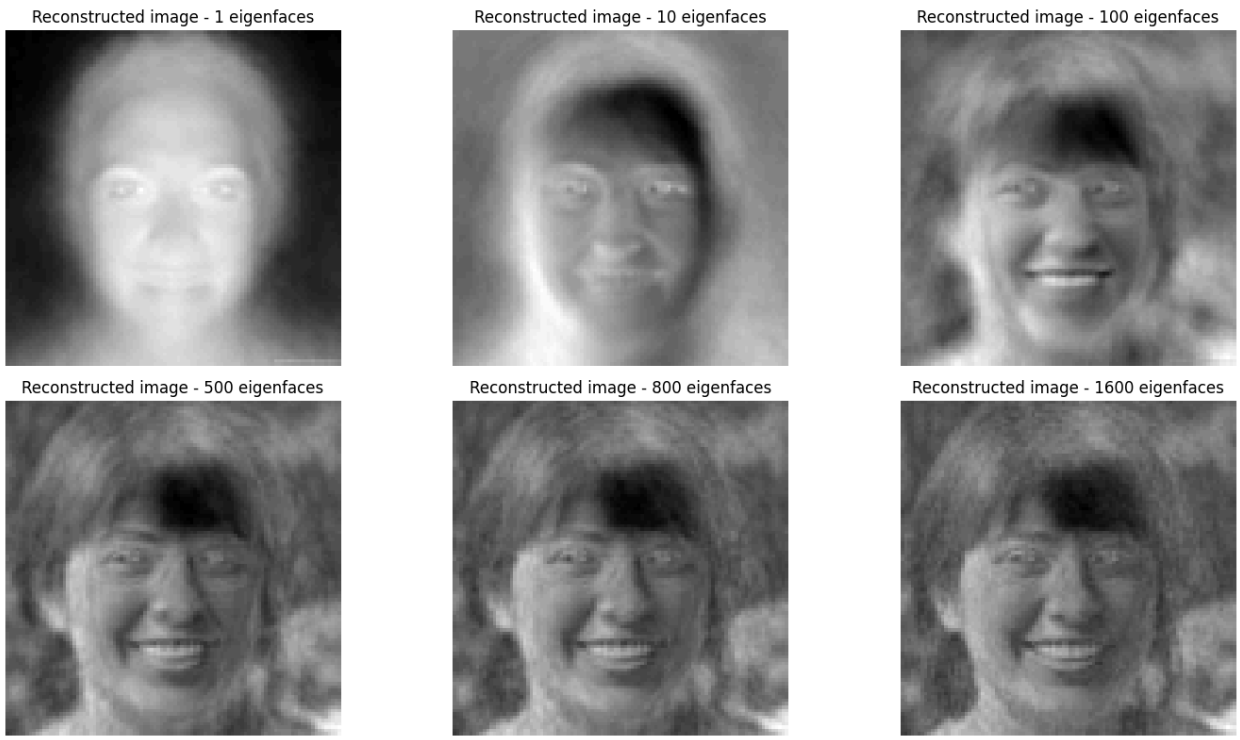


Code for loading test image, and test image

Performing reconstruction and facial recognition on test image.

Now, we take the same mean-subtracted test image and perform **reconstruction** by projecting it onto our eigenvectors formed from our 1600 training images.

```
Eigenface.ipynb
num_eigenfaces = [1, 10, 100, 500, 800, 1600]

fig, axs = plt.subplots(2, 3, figsize=(15, 8))
axs = axs.flatten()

for i, q in enumerate(num_eigenfaces):
    omega = eigenfaces[:q].dot(mean_subracted_testimg)  # the vector that represents the image with respect to the eigenfaces.
    reconstructed = eigenfaces[:q].T.dot(omega)  # image reconstructed using q eigenfaces.

    axs[i].imshow(np.reshape(reconstructed, (shape[0], shape[1])), cmap='gray')
    axs[i].set_title(f"Reconstructed image - {q} eigenfaces")
    axs[i].axis('off')

plt.tight_layout()
plt.show()
```



Reconstruction of image

There is a visible difference, and we see promising results after 500.

After this, we set a certain **a-value threshold** for facial recognition, which I have set to **3000** for this case. And with that, we check for the presence of a face.

```
Eigenface.ipynb
alpha_1 = 3000 #chosen threshold for face detection

projected_new_img_vector = eigenfaces[:q].T @ omega #n^2 vector of the new face image represented as the linear combination of the
chosen eigenfaces
diff = mean_subracted_testimg - projected_new_img_vector
beta = math.sqrt(diff.dot(diff)) #distance between the original face image vector and the projected vector.

if beta < alpha_1:
    print("Face detected in the image! ", beta)
else:
    print("No face detected in the image! ", beta)
```

```
Face detected in the image!  1009.8175210726279
```

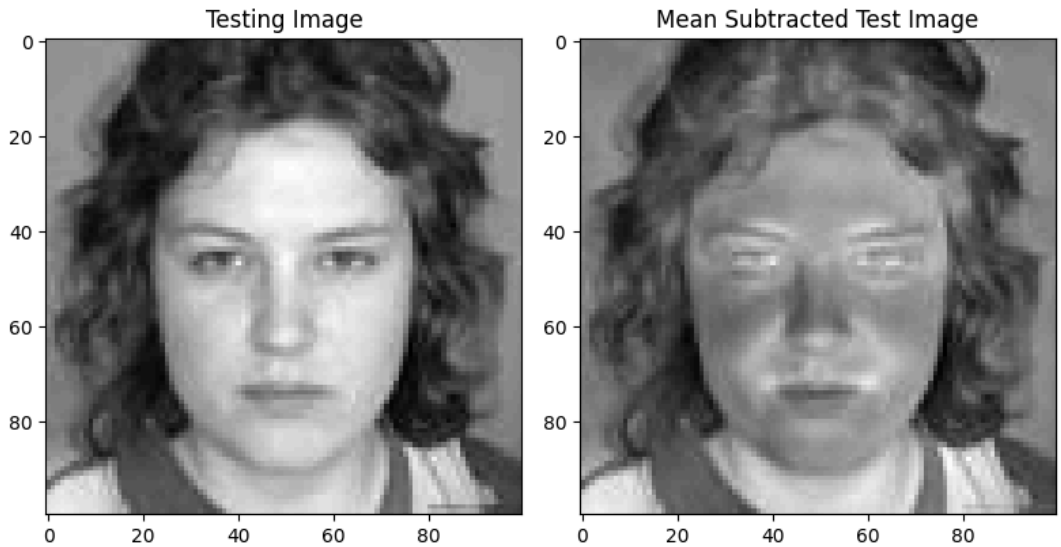
(The lower the value the better)

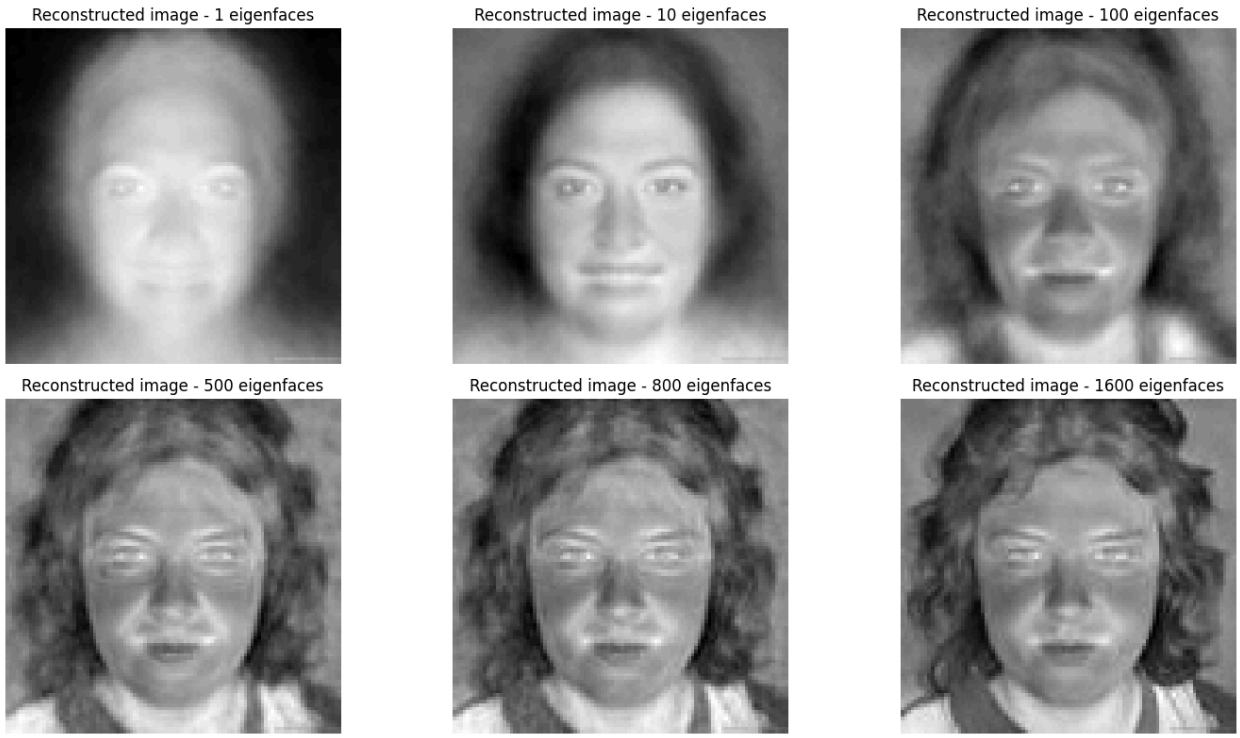
Code for facial recognition

After this, we perform a few more calculations to check the efficacy of this algorithm.

Performing facial recognition on an already existing image

Here we see the changes in results when doing the same process on an image already in the dataset. We observe better reconstruction and **great a-value**.

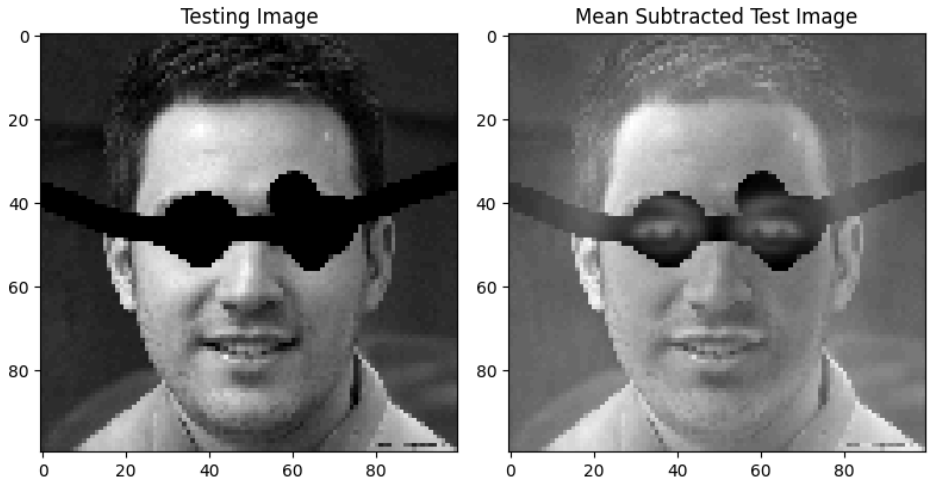


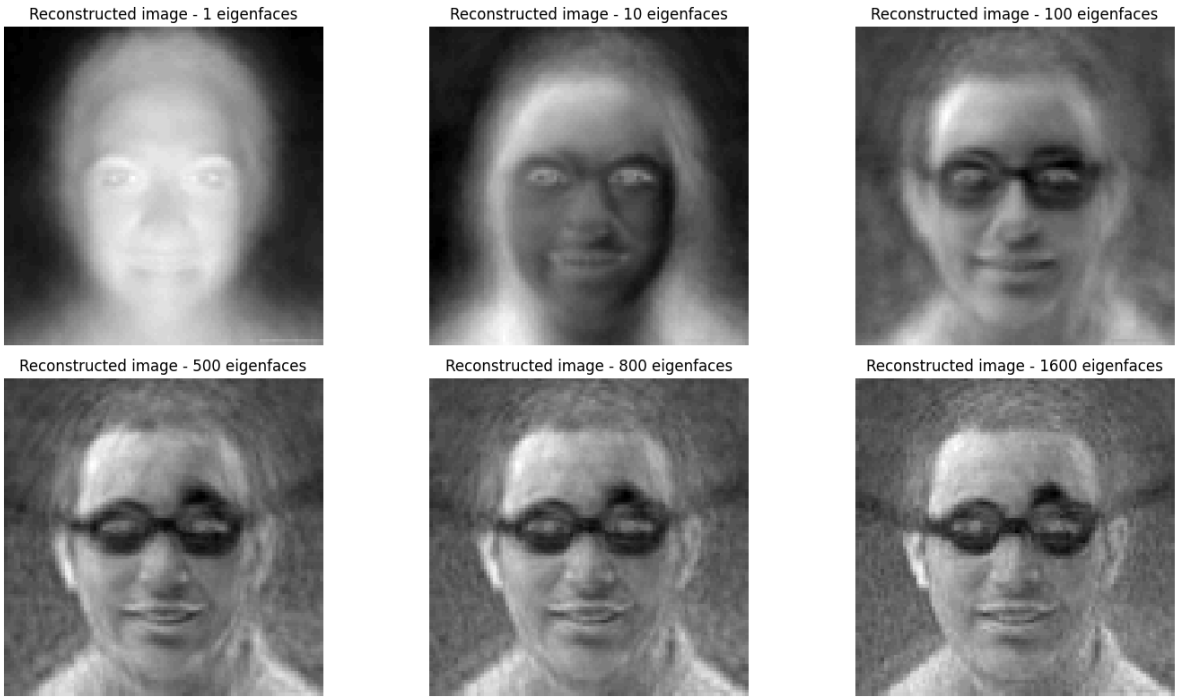



```
Face detected in the image!  100.50838847111994
```

Performing facial recognition on defaced image

To further test the code, I deform an image and test the results. I block out the eyes and run the code. The results are encouraging, and the face is still **recognizable**.

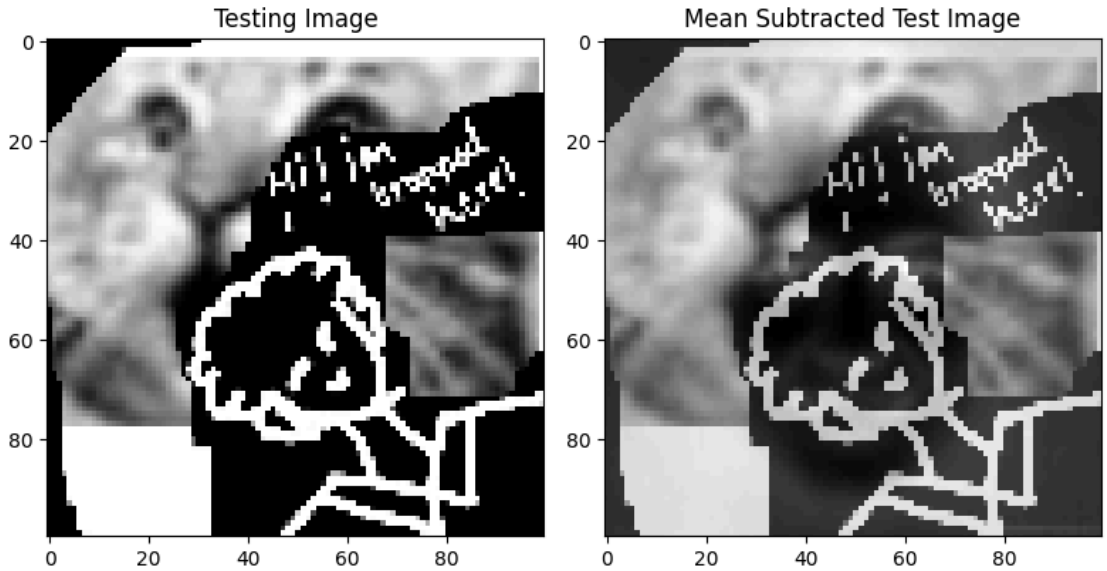


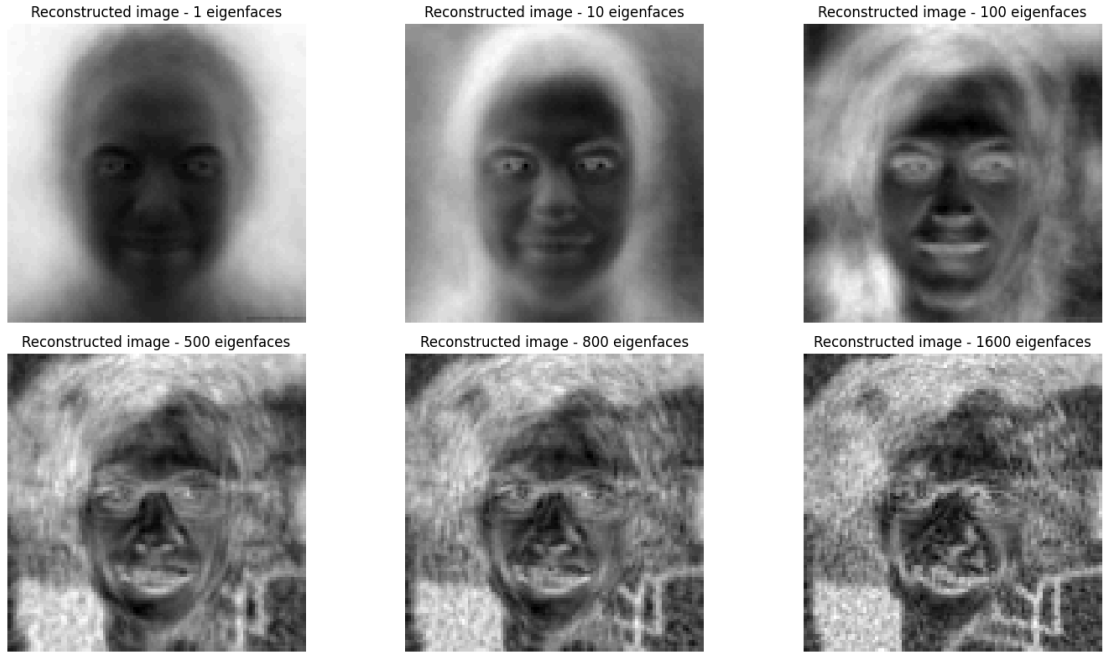



```
Face detected in the image!  1251.5044307048333
```

Running the code on something that is not a face

We test again on an image without a face. We do **not** get a **false positive** and get an a-value much **above** the threshold.






```
No face detected in the image!  5218.705902099763
```

Conclusion

This project delved deep into the concepts of linear algebra and its sheer **versatility** was made abundantly clear. The entire process was very enlightening and I feel closer to the subject afterward. The inner workings of such a strikingly simple technology we use on the daily were very interesting to **observe** and imitate. I feel significantly more comfortable talking about and practicing the **concepts** of eigenvalues and eigenvectors, which used to be daunting to me earlier.

Acknowledgments

I would like to that **Prof. Santosh Singh** for assigning me this project and allowing me to work on it. If not for their guidance and teachings throughout the semester, this project would not have been possible. I would also like to thank my father, who helped me gain a clear vision of how to approach this project very early on, which has proven to be nothing by pivotal in how I have completed the project.

Bibliography

Works cited/taken reference from in this project are:

- *Turk, Matthew, and Alex Pentland. "Eigenfaces for Recognition." Journal of Cognitive Neuroscience, vol. 3, no. 1, pp. 71-86, 1991.*
- *Brunton, Steven L., and Jose Nathan Kutz. Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control. Cambridge University Press, 2022.*
- *Karras, Tero, et al. "Analyzing and Improving the Image Quality of StyleGAN." arXiv:1912.04958 [cs.CV], 3 Dec. 2019, https://arxiv.org/abs/1912.04958.*