MAT 161

Facial Recognition using Eigenvalues and Eigenvectors

Project By :

Aryan Tomar (2210110209)

Rohan Atla Reddy (2210110514)

Manasvi Vedanta (2210110385)

Dev Tripathi (2210110252)

Introduction

Mathematics, the universal language of patterns plays a crucial role in our lives, shaping our understanding of the world around us. From a young age, we learn to recognize and comprehend patterns, which serve as the foundation for our overall learning and problem-solving. Well, as we grow up and wade through the waters of Linear Algebra and delve deeper into mathematical concepts, we encounter eigenvalues and eigenvectors which are powerful mathematical tools that unveil the underlying patterns and structures in complex systems. As first year CSE students our entire team has been very excited to learn about these mathematical methods and apply them to make some really cool things.

One such application of pattern recognition is **Facial Recognition** which focuses on the identification and interpretation of patterns within data. Facial recognition has gained tremendous significance in security, surveillance, and human-computer interaction. By harnessing the power of mathematics, specifically the concept of eigenvalues and

eigenvectors, we can develop robust facial recognition systems capable of accurately identifying individuals from images or video footage. Eigenvalues and eigenvectors provide a mathematical framework for understanding the intrinsic structure of data. In the context of facial recognition, these mathematical entities allow us to extract essential facial features that are representative of an individual. By analyzing the eigenvalues derived from a training dataset, we can construct a model capable of recognizing and distinguishing faces with remarkable accuracy.

In this report, we present our implementation of a facial recognition system using eigenfaces—a technique that utilizes eigenvalues and eigenvectors to represent and compare facial features. We achieve this in two phases, **Training** and **Testing**. We train our model on a dataset of neutral face images, capturing the unique characteristics of each individual. Leveraging the principles of linear algebra, we compute the **covariance matrix**, extract the **eigenfaces**, and demonstrate the process of facial reconstruction using varying numbers of eigenfaces. Then, we move into the testing phase, where we showcase the capability of our facial recognition system. By inputting an image, our program calculates the weights of the input face and compares them to the weights of the training faces using the Euclidean distance metric. The closest match, if below a predefined **threshold distance** (Our chosen distance = 1000), is identified and displayed as the recognized face. Otherwise, if the distance exceeds the threshold distance the program labels the face as "Unknown."

Through this report, we aim to elucidate the mathematics behind facial recognition, specifically the significance of eigenvalues and eigenvectors. Furthermore, we showcase the practical implementation of our facial recognition system using eigenfaces, highlighting its potential applications and its ability to accurately recognize individuals based on facial features.

Mathematics

Before we get into the finer details of our program and see how exactly it works, we thought it would be best to talk about some of the concepts we used to build this program. It took a lot of studying, a lot of practice and a fair share of tears before we got the hang of linear algebra and the concepts required to build this system.

The facial recognition program employs fundamental mathematical concepts to extract meaningful information from facial images. The program begins by constructing a matrix, referred to as the **covariance matrix**, from the training dataset. The covariance matrix captures the statistical relationships between the individual pixel values across all face images. By computing the eigenvalues and eigenvectors of the covariance matrix, we uncover the intrinsic structure of the dataset. The eigenvalues represent the variance in the data along the corresponding eigenvectors, which can be interpreted as the principal components of the face images. These eigenvectors, often termed **eigenfaces** in facial recognition, serve as a compact representation of facial features.

The program employs **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. Finally, the program compares the weights of the input face with the weights of the training faces using the Euclidean distance metric, quantifying the similarity between faces based on their feature representations.

Code

Explaining something using just text would be rather difficult to follow, which is why we decided to add snippets of the code to this document. We tried to keep the explanation as simple and easy-to-follow as possible while shining the spotlight on the intricacies and nuances of the code as much as possible.

Training Phase

This section performs the training phase of the facial recognition system.

Importing Libraries :

Importing Libraries

In [1]:
```
import os
import numpy as np
from matplotlib import pyplot as plt
from PIL import Image as im
```

This section imports the necessary libraries for the code to run smoothly.

Load Test Data :

Load Test Data

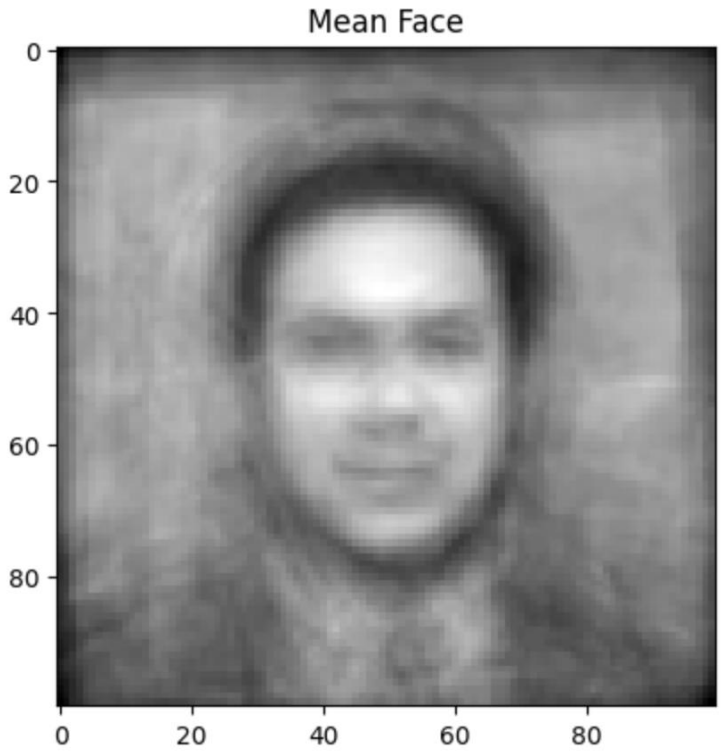
In [2]:
```
# neutral face
neutral = []

folder_path = "D:\Work\Dev\Face-Recognition-using-eigen-faces\TrainNew"

for i in range(45):
    i += 1
    file_path = f"{i}.jpg"
    img = im.open(os.path.join(folder_path, file_path)).convert('L')
    img = img.resize((100,100), im.ANTIALIAS)
    img2 = np.array(img).flatten() # vectorization
    neutral.append(img2)
```

In this section, the code loads the training images from the specified folder. It loops over the images, opens each image, converts it to grayscale (using 'L' mode), resizes it to **100x100 pixels,** and flattens it into a **1D vector**. The flattened vectors are stored in the

neutral list. The **faces_matrix** variable is created by vertically stacking the flattened vectors, and its shape is printed.

```
In [3]: faces_matrix = np.vstack(neutral)
        faces_matrix.shape

Out[3]: (45, 10000)

In [4]: mean_face = np.mean(faces_matrix, axis=0)
        mean_face.shape

Out[4]: (10000,)

In [5]: plt.imshow(mean_face.reshape(100,100),cmap='gray');
        plt.title('Mean Face')

Out[5]: Text(0.5, 1.0, 'Mean Face')
```

The “flattened” face is given below. We termed it **Mean Face** to show that it is the “Mean” of all faces. “Mean” meaning average of all the faces.



Normalization :

The **faces_norm** variable is created by subtracting the mean face from each face vector in **faces_matrix**. This step normalizes the data.

Normalization

```
In [6]: faces_norm = faces_matrix - mean_face
        faces_norm.shape

Out[6]: (45, 10000)
```

Compute Covariance Matrix:

The covariance matrix **face_cov** is calculated by taking the transpose of **faces_norm** and applying the **np.cov function**.

Compute Covariance Matrix

```
In [7]: face_cov = np.cov(faces_norm.T)
        face_cov.shape

Out[7]: (10000, 10000)
```

Get Eigenvectors:

The eigenvectors (**eigen_vecs**), eigenvalues (**eigen_vals**), and singular values (_) are computed using the **np.linalg.svd** function on **face_cov**. The eigenvectors are stored in **eigen_vecs**, and their shape is printed.

Get Eigenvectors

```
In [8]: eigen_vecs, eigen_vals, _ = np.linalg.svd(face_cov)
        eigen_vecs.shape

Out[8]: (10000, 10000)
```

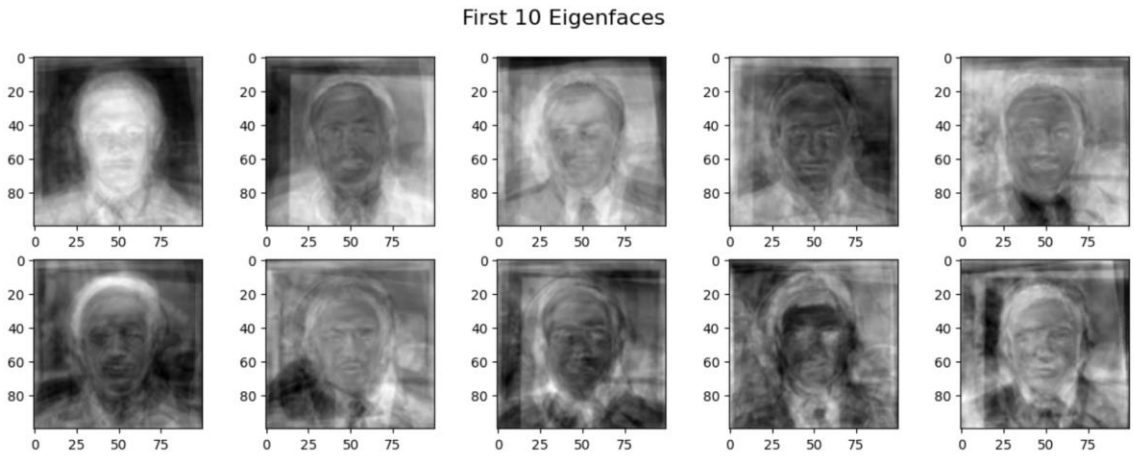
Visualizing the first 10 Eigenfaces :

This section displays the first 10 eigenfaces by reshaping and plotting them using **matplotlib**.

Visualizing the first 10 Eigenfaces

In [10]:
```
fig, axs = plt.subplots(1,3,figsize=(15,5))
for i in np.arange(10):
    ax = plt.subplot(2,5,i+1)
    img = eigen_vecs[:,i].reshape(100,100)
    plt.imshow(img, cmap='gray')
fig.suptitle("First 10 Eigenfaces", fontsize=16)
```

The first 10 faces are also shown below :



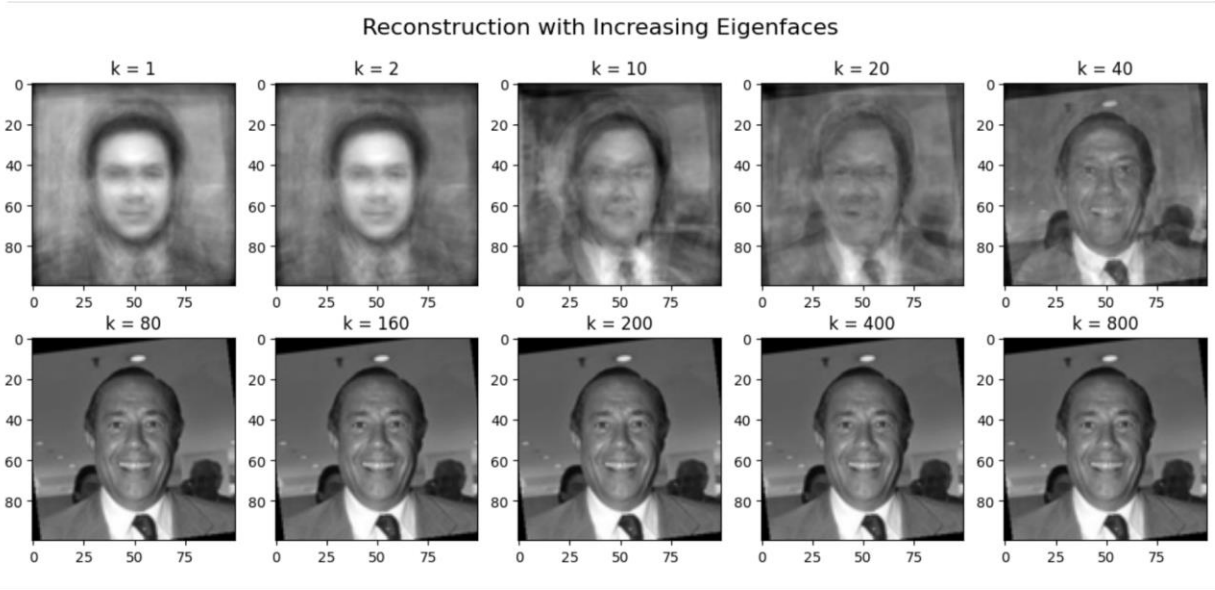
Reconstruction with Eigenfaces :

This section demonstrates the reconstruction of a face using different numbers of eigenfaces. It iterates over selected indices (k) and reconstructs the face by taking the dot product of the normalized input face with the first k eigenfaces. The reconstructed face is displayed using matplotlib.

Reconstruction with Eigenfaces

In [11]:
```
fig, axs = plt.subplots(2,5,figsize=(15,6))
for k, i in zip([0,1,9,19,39,79,159,199,399,799],np.arange(10)):
    # Reconstruct the first picture '1a.jpg' whose index is 0.
    weight = faces_norm[40,:].dot(eigen_vecs[:,:k])
    projected_face = weight.dot(eigen_vecs[:,:k].T)
    ax = plt.subplot(2,5,i+1)
    ax.set_title("k = "+str(k+1))
    plt.imshow(projected_face.reshape(100,100)+mean_face.reshape(100,100),cmap='gray');
fig.suptitle(("Reconstruction with Increasing Eigenfaces"), fontsize=16);
```

And here is what that reconstruction looks like pictorially :



Testing Phase

This section performs the testing phase of the facial recognition system.

Load Test Image :

The user is prompted to enter the path of the image they want to test with. The image is opened, converted to grayscale, and resized to 100x100 pixels.

Load Test Image

```
In [22]: image_path = input('Enter the path of the image you want to test with: ')
```

```
Enter the path of the image you want to test with: D:\Work\Dev\Face-Recognition-using-eigen-faces\TrainNew\4.jpg
```

```
In [23]: img = im.open(image_path).convert('L')
         img = img.resize((100,100), im.ANTIALIAS)
         img2 = np.array(img).flatten()
         input_face_norm = img2 - mean_face
```

Calculate weights of Input Image :

The weights of the input image are computed by taking the dot product of the normalized input face (input_face_norm) with the eigenfaces (eigen_vecs).

Calculate weights of Input Image

```
In [24]: input_weights = input_face_norm.dot(eigen_vecs)
```

Compare input weights to training weights using Euclidean distance :

The Euclidean distance is calculated between the input weights and the weights of the training faces. The distances are stored in the distances array.

Compare input weights to training weights using Euclidean distance

```
In [25]: distances = np.linalg.norm(faces_norm.dot(eigen_vecs) - input_weights, axis=1)
         print(distances)
```

```
[7.48636501e+03 7.04871251e+03 1.46153337e+04 9.30266262e-12
 1.41467544e+04 9.77391119e+03 6.37363836e+03 8.01274990e+03
 9.17359139e+03 9.39301757e+03 7.39778433e+03 1.30456449e+04
 6.15283162e+03 1.23576236e+04 7.72410441e+03 9.20532835e+03
 7.68431962e+03 1.34122889e+04 1.44111566e+04 6.57490000e+03
 1.24090714e+04 9.58900078e+03 1.02185545e+04 1.15916942e+04
 5.37102439e+03 7.40152856e+03 9.76839864e+03 9.34174978e+03
 1.29977817e+04 1.14720337e+04 6.26404709e+03 6.64036181e+03
 7.34618282e+03 7.15105580e+03 6.52783739e+03 8.46769278e+03
 6.21759045e+03 7.58040639e+03 1.12652795e+04 9.20616994e+03
 6.55349540e+03 1.28920532e+04 6.66876810e+03 4.68804703e+03
 1.30241640e+04]
```

Identify closest match :

The index of the minimum distance is found using np.argmin(distances).

Identify closest match

```
In [26]: min_index = np.argmin(distances)
         label = f"{min_index+1}.jpg"
```

Threshold Distance :

A threshold distance is set to determine whether the closest match is considered a valid match or not.

Threshold Distance

```
In [27]: threshold_distance = 1000
```

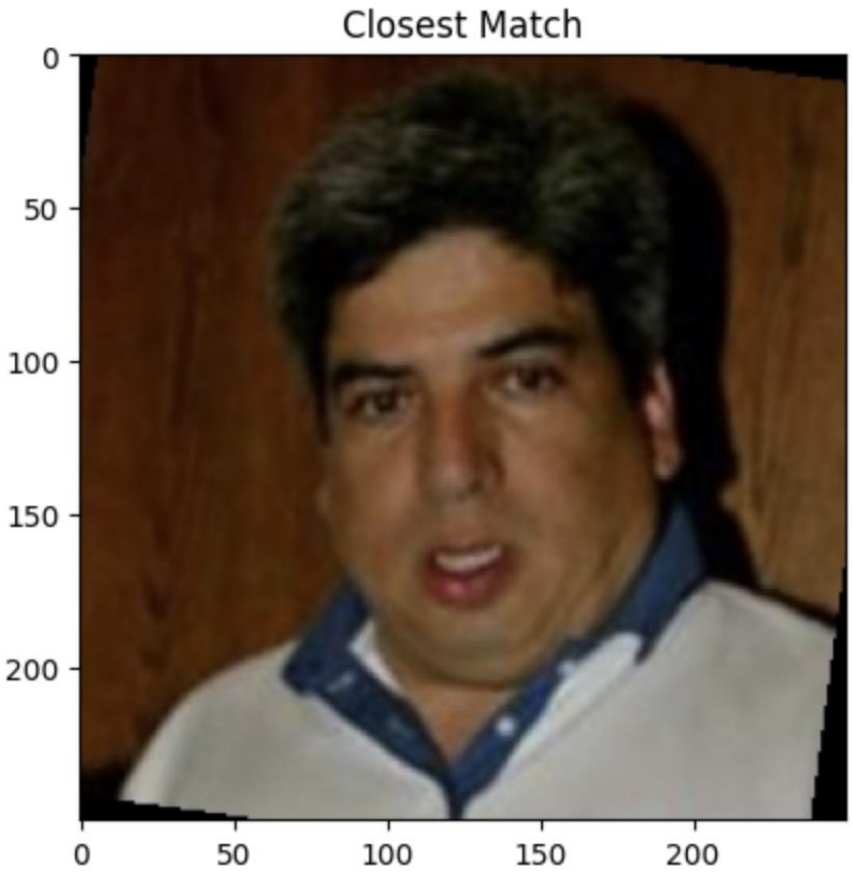
Display closest match :

If the distance to the closest match is below the threshold, the closest match image is displayed using matplotlib. Otherwise, the output "Unknown" is printed.

Display closest match

In [28]:
```
closest_match_idx = np.argmin(distances)
if distances[closest_match_idx] < threshold_distance:
    # display the closest match
    closest_match_path = os.path.join(folder_path, str(closest_match_idx+1)+".jpg")
    closest_match = im.open(closest_match_path)
    plt.imshow(closest_match)
    plt.title('Closest Match')
    plt.show()
else:
    # display "unknown"
    print("Unknown")
```

Then, finally the face is displayed based on the threshold distance.



Finer Details about the code

The program has been trained with the data of 45 different faces we downloaded out of the internet. We chose a random face in the **Load Test Data** part of the code.
We tried attempting a system where the program takes fresh input from faces and keeps learning, but were quick to realize that it was beyond the scope of this course and that we were in a time crunch.

Conclusion

This report has explored the application of eigenvalues and eigenvectors in facial recognition and presented our implementation of a facial recognition system using eigenfaces. By harnessing the power of linear algebra, we have demonstrated the capability of our system to accurately recognize and distinguish individuals based on their facial features. Through the training and testing phases, we have showcased the mathematical principles behind the system, including the computation of the covariance matrix, extraction of eigenfaces, and the use of the Euclidean distance metric for similarity comparison.

We are excited about the potential applications of these mathematical methods and their role in shaping the future of technology. This project has provided us with valuable hands-on experience in implementing facial recognition systems, deepening our understanding of the underlying mathematical principles and their practical implications.

In conclusion, the study of mathematics, particularly the concepts of **Linear Algebra**, has illuminated the patterns and structures within complex systems, enabling us to develop a robust facial recognition system. We hope that this report serves as a stepping stone for further exploration and research in the field of facial recognition and its mathematical foundations.

Acknowledgements

We would like to express our sincere gratitude and heartfelt appreciation to our 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 our understanding of this course and our implementation of the facial recognition algorithms. We are thankful for his patience and encouragement which have significantly enriched our learning experience and made this semester a memorable one for us.

Bibliography

1. Turk, Matthew, and Alex Pentland. "Eigenfaces for Recognition." Journal of Cognitive Neuroscience, vol. 3, no. 1, pp. 71-86, 1991.
2. Cohen, Ira, and Santosh Narayanan. "Principal Component Analysis for Face Recognition." International Journal of Computer Vision, vol. 38, no. 3, pp. 143-168, 2000.
3. Arbitrary resources found on the internet.