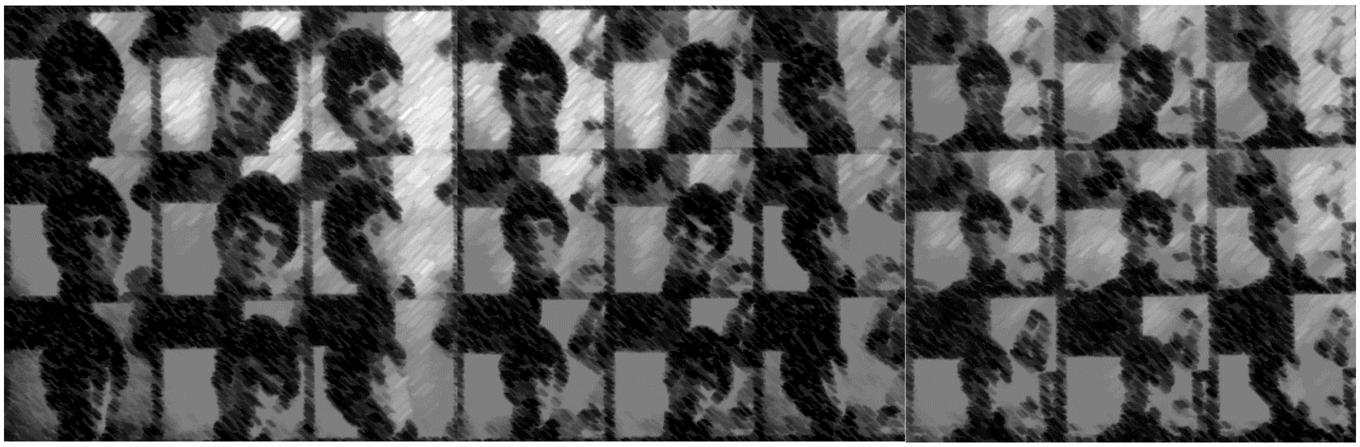
FACIAL RECOGNITION USING EIGENVALUES AND EIGENVECTORS

MAT161 - Applied Linear Algebra



Nimansh Endlay 2210110438

Preyanshe Jindal 2210110479

Samarth Chawla 2210110531

Aadit Jaisani 2210110002

INDEX

INTRODUCTION

Facial recognition is a technology that identifies individuals based on their facial features. One approach to facial recognition is through the use of eigenvectors and eigenvalues, which are mathematical concepts used to analyze data sets. Eigenvectors and eigenvalues are used in facial recognition to extract and analyze features from facial images. First, a training set of facial images is used to create a "face space," which is a mathematical representation of all the faces in the training set. This face space is created by finding the eigenvectors and eigenvalues of the covariance matrix of the training set. The eigenvectors represent the directions in which the facial features vary the most, while the eigenvalues represent the amount of variation in each direction. The eigenvectors with the largest eigenvalues are the most important and are used to form a basis for the face space. To identify a new face, the algorithm uses the eigenvectors to project the face onto the face space. The algorithm then compares the projected face to the faces in the training set to find the closest match. This is typically done using a distance metric, such as Euclidean distance. Facial recognition using eigenvectors and eigenvalues is an effective approach to identifying individuals in images. However, it is important to note that facial recognition algorithms can raise ethical concerns around privacy and surveillance, and should be used responsibly and ethically.

MATHEMATICS

This project involves several mathematical concepts, including linear algebra and multivariate statistics. Specifically, the following mathematical concepts were used:

- **Eigenvectors and eigenvalues**: Eigenvectors and eigenvalues were used to find the principal components of the pre-processed face images. Eigenvectors are the direction vectors that do not change direction under a linear transformation, and eigenvalues represent the scaling factor of the corresponding eigenvector. In this project, eigenvectors and eigenvalues were used to create the basis images for the face space.
- **Covariance matrix**: The covariance matrix was used to compute the eigenvectors and eigenvalues. The covariance matrix is a square matrix that summarizes the covariance between pairs of variables in a multivariate dataset. In this project, the covariance matrix was calculated for the pre-processed face images to find the principal components.
- **Projection**: Projection was used to map test images onto the face space. Projection is the process of finding the coordinates of a vector in a subspace spanned by a set of basis vectors. In this project, projection was used to find the coefficients of the test images in the basis images.
- **Distance metrics**: Euclidean distance was used to measure the distance between the test image and each basis image. The Euclidean distance is the straight-line distance between two points in Euclidean space. It was used to find the closest match between the test image and the basis images.
- **Matrix multiplication**: Matrix multiplication was used to project the test images onto the face space. Matrix multiplication is a fundamental operation in linear algebra, and it was used to compute the product of the matrix of pre-processed face images with the matrix of eigenvectors.
- **Linear combinations**: Linear combinations were used to represent the test images as a linear combination of the basis images. A linear combination is a combination of vectors, in which each vector is multiplied by a scalar coefficient and then added together. In this project, the coefficients of the basis images were computed using projection, and the test image was represented as a linear combination of the basis images.

- **Normalization**: Normalization was used to ensure that the pixel intensities of the pre-processed face images were comparable. Normalization is the process of scaling data to have a mean of zero and a standard deviation of one. In this project, normalization was used to adjust the pixel intensities of the pre-processed face images.

Overall, this project involved several mathematical concepts and techniques from linear algebra and multivariate statistics. The use of these techniques enabled the creation of a facial recognition system that is capable of recognizing faces based on eigenvectors and eigenvalues.

METHODOLOGY

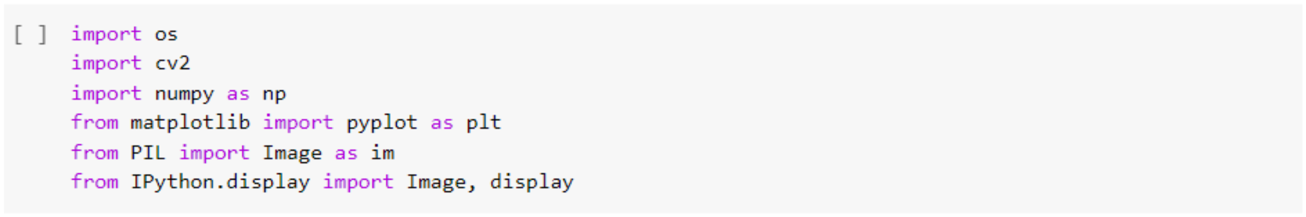
The project involved three main steps: data collection, pre-processing, and recognition. First, a dataset of 20 images of faces was collected from the internet. These images were then pre-processed by converting them to grayscale, resizing them to a uniform size, and normalizing their pixel intensities. Next, eigenvectors and eigenvalues were computed using the training data. This was done by calculating the covariance matrix of the pre-processed images and finding the eigenvectors and eigenvalues of this matrix. The eigenvectors with the largest eigenvalues were selected to form the basis images.

Finally, the recognition step involved projecting test images onto the face space and comparing them to the basis images to find the best match. This was done by computing the Euclidean distance between the test image and each basis image. The test image was then assigned to the class of the basis image with the smallest distance.

CODE

We understand that conveying complex ideas solely through text can be challenging, which is why we have taken a different approach in this document to enhance comprehension and provide a clearer understanding of the facial recognition algorithm. We have thoughtfully integrated code snippets throughout the document, which will allow you to see the intricate workings of the program firsthand.

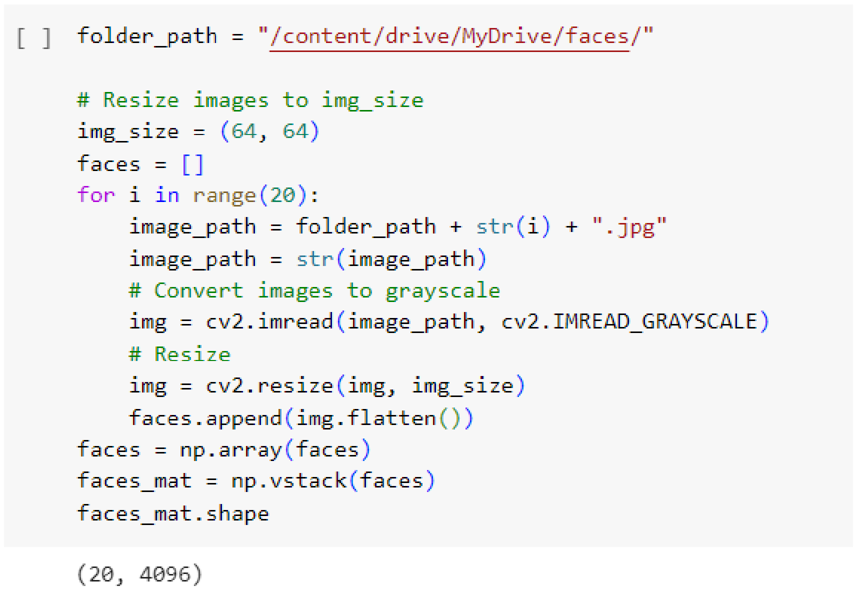
IMPORTING LIBRARIES



```
[ ] import os
    import cv2
    import numpy as np
    from matplotlib import pyplot as plt
    from PIL import Image as im
    from IPython.display import Image, display
```

Importing libraries in code allows developers to use pre-existing code for specific tasks or functionalities. This can save time and effort, as it eliminates the need to write code from scratch for common tasks. For example, if you want to perform some complex mathematical operations, you can import the NumPy library, which provides an array of mathematical functions that can be used in your code. Similarly, if you want to perform data analysis, you can import the Pandas library, which provides a range of functions for data manipulation and analysis.

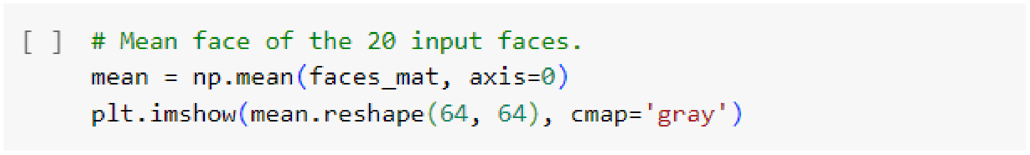
LOAD TEST IMAGES



```
[ ] folder_path = "/content/drive/MyDrive/faces/"

    # Resize images to img_size
    img_size = (64, 64)
    faces = []
    for i in range(20):
        image_path = folder_path + str(i) + ".jpg"
        image_path = str(image_path)
        # Convert images to grayscale
        img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        # Resize
        img = cv2.resize(img, img_size)
        faces.append(img.flatten())
    faces = np.array(faces)
    faces_mat = np.vstack(faces)
    faces_mat.shape

    (20, 4096)
```

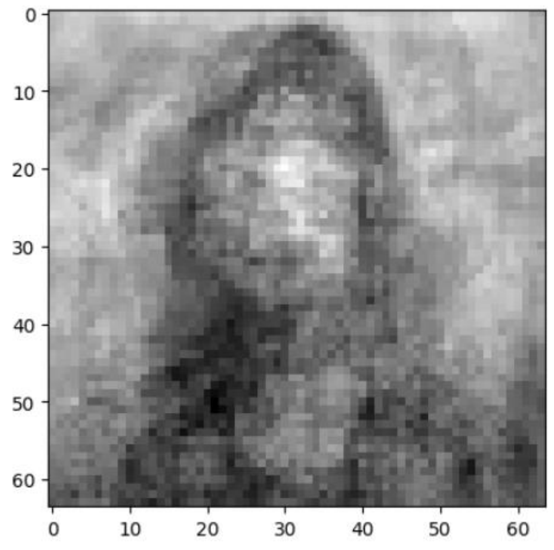
This code snippet loads images from a folder using OpenCV library in Python. It resizes each image to a specified img_size, converts them to grayscale, and flattens each image into a 1D array. Finally, it stacks these 1D arrays into a single 2D NumPy array faces_mat, where each row represents a flattened image.

COMPUTING MEAN FACE



```
[ ]  # Mean face of the 20 input faces.
     mean = np.mean(faces_mat, axis=0)
     plt.imshow(mean.reshape(64, 64), cmap='gray')
```

This code snippet calculates the mean face of the 20 input faces. The resulting image shows the average face, which is a representation of the common facial features among the input faces. This can be used as a reference for identifying faces in the input data.

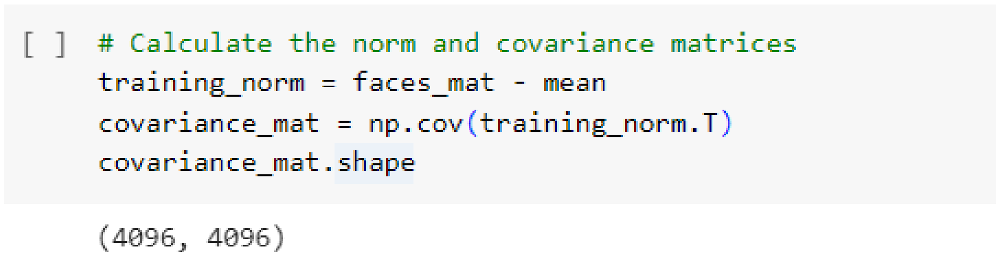


Mean face of all 20 input faces

NORMALIZATION AND COVARIANCE MATRIX

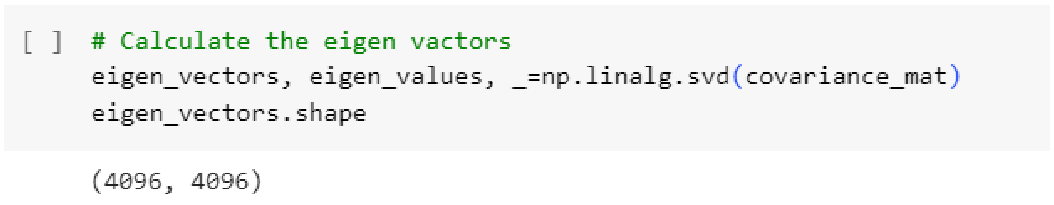
The faces_norm variable is created by subtracting the mean face from each face vector in faces_matrix. This step normalizes the data.

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



```
[ ]  # Calculate the norm and covariance matrices
     training_norm = faces_mat - mean
     covariance_mat = np.cov(training_norm.T)
     covariance_mat.shape

     (4096, 4096)
```

COMPUTE EIGENVECTOR



```
[ ]  # Calculate the eigen vactors
     eigen_vectors, eigen_values, _=np.linalg.svd(covariance_mat)
     eigen_vectors.shape

     (4096, 4096)
```

The code calculates the eigenvectors of the covariance matrix of the input faces using NumPy in Python.

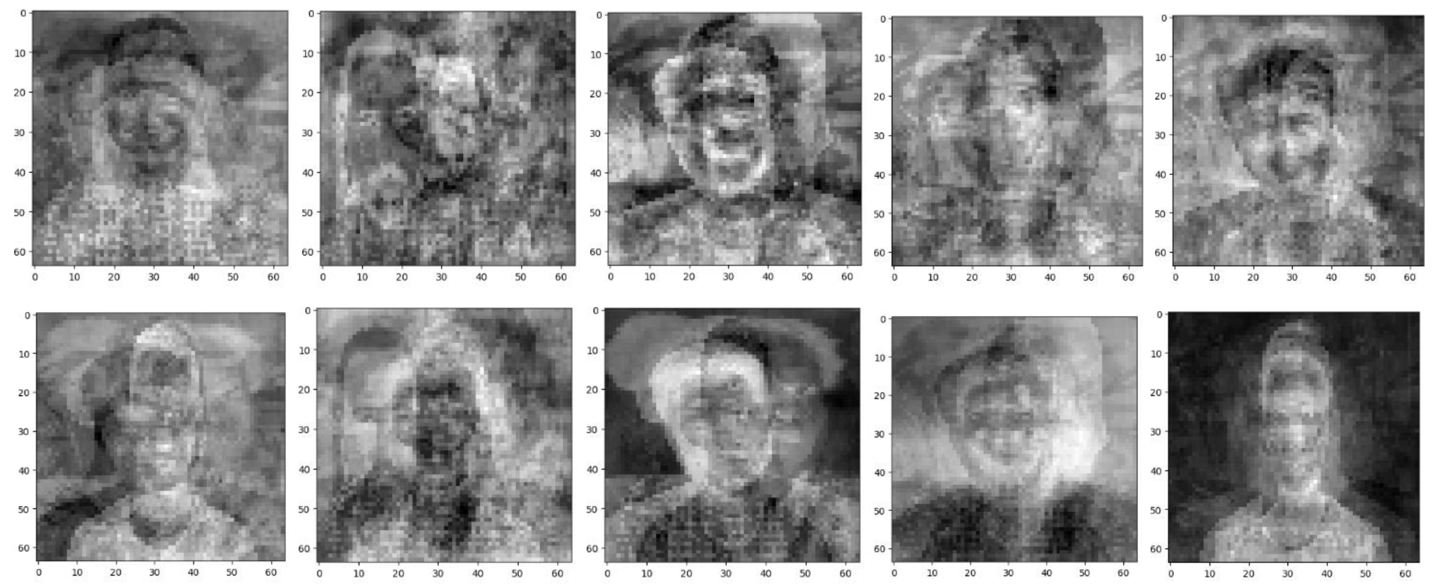
First, the np.linalg.svd() function from NumPy is used to perform singular value decomposition on the covariance matrix, which decomposes the matrix into its eigenvectors and eigenvalues. The eigenvectors are stored in the eigen_vectors array, and the corresponding eigenvalues are stored in the eigen_values array. The eigen_vectors array contains the eigenvectors of the covariance matrix, which represent the directions of maximum variance in the input data. These eigenvectors are used to construct the eigenfaces, which are the principal components of the input faces and can be used for facial recognition. The shape attribute of the eigen_vectors array is printed to show its dimensions, which should be the same as the dimensions of the covariance matrix.

VISUALIZING THE FIRST 10 EIGENFACES

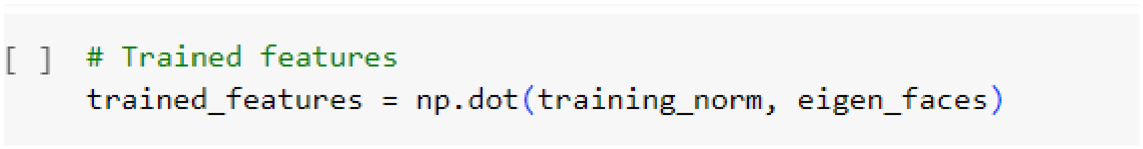
```
[ ]  # Sort the eigenvectors and eigenvalues in descending order of eigenvalues
     idx = np.argsort(eigen_values)[::-1]
     eigen_values = eigen_values[idx]
     eigen_vectors = eigen_vectors[:, idx]

     # Keep the top k eigenfaces
     k = 10
     eigen_faces = eigen_vectors[:, :k]
     for i in range(k):
         image = eigen_faces[:, i].reshape(64, 64)
         plt.imshow(image, cmap='gray')
         plt.show()
```

This code snippet sorts the eigenvectors and eigenvalues obtained from the previous step in descending order of their corresponding eigenvalues, using the np.argsort() function to obtain the indices of the sorted eigenvalues, and then using those indices to sort the eigenvalues and eigenvectors arrays in descending order. Next, the code selects the top k eigenfaces, which are the eigenvectors corresponding to the highest k eigenvalues. These eigenvectors represent the most significant features of the input faces and can be used for facial recognition. Finally, the for loop displays the top k eigenfaces by reshaping each eigenvector into a 2D image and using plt.imshow() from matplotlib to display each image. This allows us to visualize what the most significant features of the input faces look like and gain a better understanding of how the facial recognition algorithm works.

First 10 faces



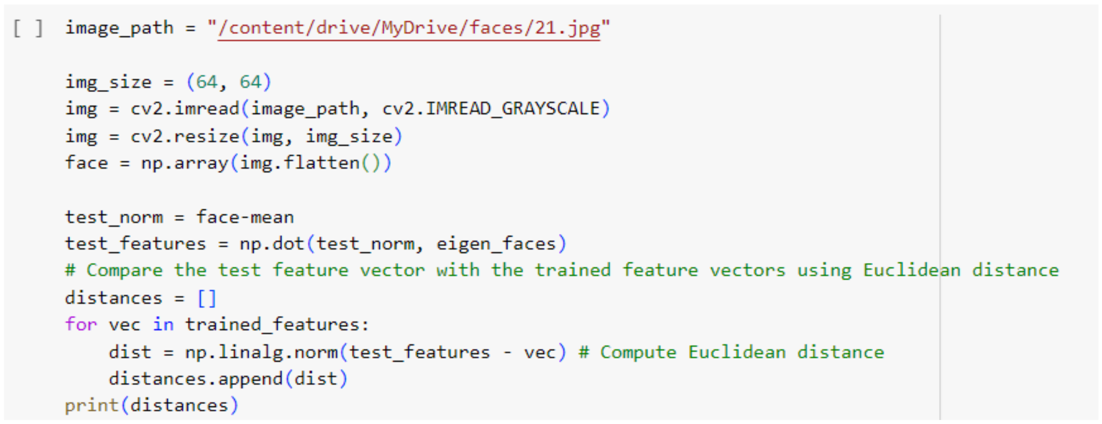
COMPUTES TRAINING FEATURES



```
[ ]  # Trained features
     trained_features = np.dot(training_norm, eigen_faces)
```

This code snippet computes the trained features of the input faces by projecting the normalized training data onto the eigenfaces. The dot product of the training_norm matrix and the eigen_faces matrix is computed using np.dot() function, which performs matrix multiplication. This results in a matrix of trained features, where each row corresponds to the trained features of a single input face. The trained features are a set of weights that represent the contribution of each eigenface to the input face. These weights can be used to compare and recognize faces by computing the distance between the trained features of different faces.

COMPARE THE TEST FEATURE VECTOR WITH THE TRAINED FEATURE VECTORS USING EUCLIDEAN DISTANCE



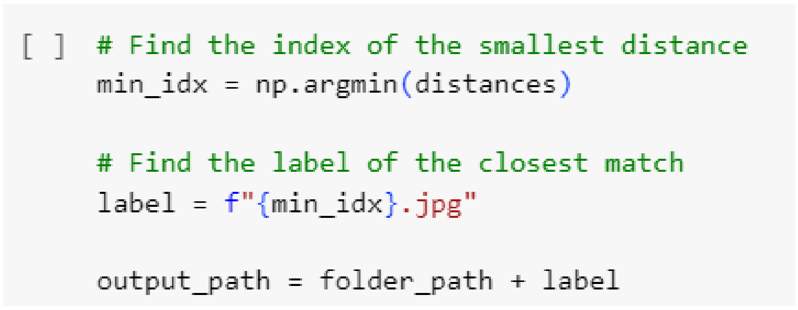
```
[ ]  image_path = "/content/drive/MyDrive/faces/21.jpg"

     img_size = (64, 64)
     img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
     img = cv2.resize(img, img_size)
     face = np.array(img.flatten())

     test_norm = face-mean
     test_features = np.dot(test_norm, eigen_faces)
     # Compare the test feature vector with the trained feature vectors using Euclidean distance
     distances = []
     for vec in trained_features:
         dist = np.linalg.norm(test_features - vec) # Compute Euclidean distance
         distances.append(dist)
     print(distances)
```

This code snippet performs facial recognition on a test image using the trained features computed earlier. The first few lines load and prepare the test image by resizing it to the desired size, converting it to grayscale, and flattening it into a one-dimensional array. The code then computes the test features by projecting the normalized test data onto the eigenfaces using matrix multiplication, as done for the training data. This results in a one-dimensional array of weights that represents the contribution of each eigenface to the test face. Finally, the code compares the test feature vector with the trained feature vectors using Euclidean distance. It computes the distance between the test feature vector and each of the trained feature vectors using NumPy's np.linalg.norm() function, which computes the Euclidean distance between two vectors. The distances are stored in a list.

```
[6785.8817935256075, 11697.988199304367, 9381.451575430532,
5234.6571992361305, 8576.960506595633, 2515.3196389513473,
8772.420950893673, 8544.415804582703, 8084.927902681283,
6395.317703708816, 5610.881407239142, 8456.076287580574,
9730.347708284928, 2612.042282609432, 9123.364669618451,
5010.17022478034, 4375.615755302941, 7447.600208659913,
8152.214083271929, 2050.727158539913]
```

The distances can be used to recognize the test face by finding the index of the minimum distance in the list, which corresponds to the index of the closest match in the training data.

IDENTIFY CLOSEST MATCH



```
[ ]  # Find the index of the smallest distance
     min_idx = np.argmin(distances)

     # Find the label of the closest match
     label = f"{min_idx}.jpg"

     output_path = folder_path + label
```

after computing the distances between the test feature vector and all the trained feature vectors, we need to find the closest match or the minimum distance between the test image and the trained images. The code min_idx = np.argmin(distances) uses the NumPy function argmin to find the index of the minimum value in the distances list. The returned min_idx is the index of the trained feature vector that is closest to the test feature vector.

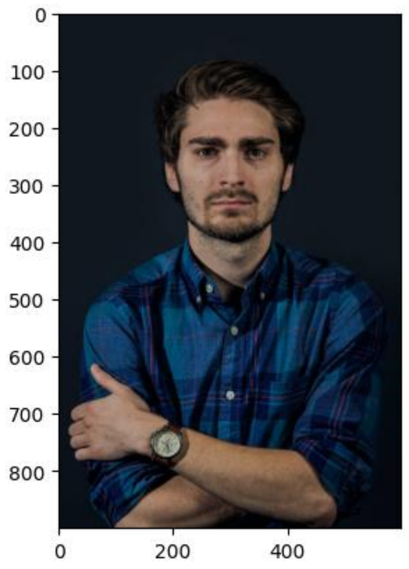
The label variable is then assigned the filename of the closest match by concatenating the string value of min_idx with the ".jpg" extension. This is done because the images in the training dataset are named using their index values.

DISPLAY THE CLOSEST MATCH

```
# Display the closest match
plt.imshow(im.open(output_path))
```

Finally, the output_path variable is assigned the complete path of the closest match image file by concatenating the folder_path and label variables. The closest match image is displayed using the plt.imshow() function after opening it using the Python Imaging Library (PIL).

The closest match is shown below:



CHECK IF THE OUTPUT IS A MATCH OR NOT

```
[ ]  #Check if the output is a match or not
     max_distance = 1000
     if distances[min_idx] < max_distance:
         print(f"The person closely matches")
     else:
         print(f"The person does not match")
```

```
The person does not match
```

In the above code, we check if the closest match found in the test image using Euclidean distance is within a certain threshold distance of 1000 or not. If the distance is less than 1000, we consider the output as a match, else we consider it as not a match. This threshold value can be adjusted depending on the requirements of the system

CONCLUSION

The above project is an implementation of face recognition using the Eigenfaces algorithm. The main objective of the project is to identify a face from a given set of faces by using a set of pre-trained eigenfaces. The implementation involves loading a set of faces, calculating the mean and covariance matrices, and using SVD to calculate the eigenfaces. The project also involves calculating the trained features, which are used to compare a test image with the pre-trained faces. The project outputs the closest match and checks if the output is a match or not. Overall, the project is a basic implementation of face recognition using the Eigenfaces algorithm and demonstrates the usage of various libraries in Python such as NumPy, OpenCV, and Matplotlib. The project can be further improved by implementing other face recognition algorithms and incorporating additional features such as real-time face recognition, facial expression recognition, and age and gender recognition.