XIAMEN UNIVERSITY MALAYSIA

Course Code	:	G0191
Course Name	:	Face Recognition With Machine Learning
Lecturer	:	Wang Han
Academic Session	:	02/2025
Assessment Title	:	PCA in face recognition
Submission Due Date	:	12 March 2025

Prepared by :

Student ID	Student Name
AIT2104019	Yee Qing Wei
AIT2302009	Chin Lian Ann
EEE2404150	Yap Han Yong
EEE2404286	Leong Yao Wei
SWE2404271	Leong Yao Wu
EEE2404388	Wong Yue Wei

Date Received :

Feedback from Lecturer:

Mark:

Own Work Declaration

I/We acknowledge that my/our work will be examined for plagiarism or any other form of academic misconduct, and that the digital copy of the work may be retained for future comparisons.

I/We confirm that all sources cited in this work have been correctly acknowledged and that I/we fully understand the serious consequences of any intentional or unintentional academic misconduct.

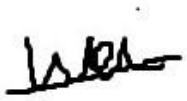
In addition, I/we affirm that this submission does not contain any materials generated by AI tools, including direct copying and pasting of text or paraphrasing. This work is my/our original creation, and it has not been based on any work of other students (past or present), nor has it been previously submitted to any other course or institution.

Signature:



（Wong Yue Wei） (Yee Qing Wei) (Yap Han Yong)





(Leong Yao Wu) (Chin Lian Ann) (Leong Yao Wei)

Date: 7 March 2025

Contents

1. Introduction

Face recognition system is used to verify a person's identity based on their facial feature. A successful recognition system should be able to find human faces inside an image, then extract and analyse the features of the face (such as the location of eyes, the shape of face and the length of mouth). Finally, it should compare these input features with the existing features in the database and find if there is anything that matches.

Face recognition system is widely used in institutions that requires some sort of identification to allow people to enter. It is also used by digital devices and online services in their login systems. It has advantages such as convenient to use and low processing time. Therefore, we plan to create our face recognition system. This system should have high accuracy so we can incorporate it directly into our future projects.

We collect our image data by taking photos of ourselves and our friends. We ensure that our images have different variations by photographing our faces from different angles. After that, we manually crop our images so only our heads are shown in the image. We also resize our image to be 100 x 100 pixels so each image can have equal representation to our recognition system. We have also provided the link to our code here.

2. Initial Training

2.1 Preprocessing

Before making an actual recognition system that scans faces in real time, we need to train different face classification models and see which suits us. First, we will have to preprocess our face data. This reduces the variance of face data and causes the data values to be within a much smaller range. This makes image pixels representing the same portion have a closer value. Therefore, the classifier can better determine their similarities. Furthermore, we reduce the dimensionality of each image. As each image has fewer features, the classifier does not need to perform complex tasks. The classifier can recognise the name of the image more accurately and in a shorter amount of time. We apply Principal Component Analysis (PCA) method in our project. We chose PCA because it can maximize the variance of the new dataset (GeeksforGeeks, 2025a).

```
path = "D:\Old_Recognition_Cropped"
os.chdir(path)
```

```
faces_list = os.listdir()
```

The filename for each photo is the name of the person in the photo. Our recognition system only needs to find the image that has the most similar image data with the input image. It does not need to train with the name of each image.

```
# Store all pixels values of the images
for i in range (1, len(faces_list)):
    # Initializing 'image_temp' array with the image
    image_temp = np.array([image.imread(path + f'/{faces_list[i]}')])

    # Resizing the image to (100 x 100) pixels size
    image_temp = np.array([resize(image_temp[0], (100, 100, 3))])
    image_temp.shape

    # Converting the image to grayscale
    gray = lambda rgb : np.dot(rgb[... , :3] , [0.299 , 0.587, 0.114])
    image_temp = np.array([gray(image_temp[0])])

    # Append 'image_temp' to 'face_images'
    face_images = np.concatenate((face_images, image_temp), axis=0)

    # Flattening the pixels
    data_temp = np.array([face_images[i].flatten()])

    # Append 'data_temp' to 'face_data'
    face_data = np.concatenate((face_data, data_temp), axis=0)
```

After getting enough photos, we will begin to preprocess our images. We first ensure that our photos are 100 x 100 pixels in size by resizing them. We will then convert these images into grayscales by performing dot products of RGB values of images with specific numbers. Afterwards, we flatten our image data into one dimension. Finally, we duplicate our image dataset into two. PCA is then performed with eigendecomposition and singular value decomposition (SVD) factorization methods respectively. Our code will perform PCA methods manually, as it is easier to explain their process.

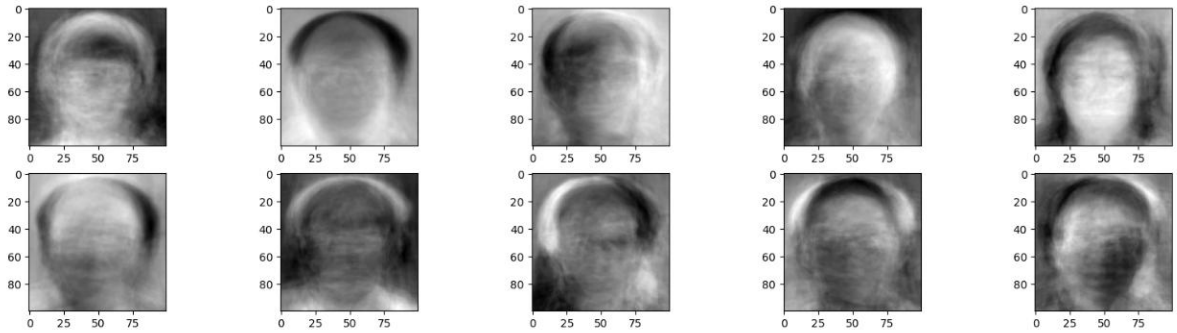
2.2 Eigendecomposition

```
:
# Mean centering of facedata for eigendecomposition
face_data_eig = face_data_eig / 255 - np.mean(face_data_eig / 255, axis=0)

# Computing the covariance matrix
covariance_matrix_eig = face_data_eig.T @ face_data_eig / len(face_data_eig)
print(f"The covariance matrix of eigen-decomposition:\n{covariance_matrix_eig}")
```

Eigendecomposition decomposes a data matrix into eigenvectors and eigenvalues. However, we first must standardize our data (through mean-centering) and then compute their covariance matrix. We decompose the covariance matrix instead of the original data matrix because covariance matrices are symmetric. This means that our eigenvalues will be real values and eigenvectors will be perpendicular to each other (Pramoditha, 2023). As we can see in the code, covariance matrices are obtained by performing matrix multiplication of the transpose of data matrices and the data matrices, before dividing it with the number of matrices.

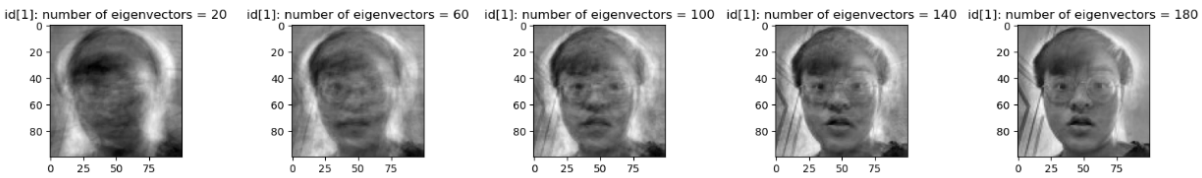
```
%%time

# Computing the eigenvalues and eigenvectors of the covariance matrix
eval, evec  = np.linalg.eig(covariance_matrix_eig)

# Sorting eigenvalues and its corresponding eigenvector in decending order
sorted_evec = np.array([x for _,x in sorted(zip(eval, evec.T), reverse=True)]).T
sorted_eval = np.array(sorted(eval, reverse=True))
```

We simply use NumPy functions to perform eigendecomposition. After that, we sort our eigenvectors in descending order. This is because we only need to select the few eigenvectors that have the most impact on face recognition.



This segment shows what our eigenvectors look like. It records the shape of faces and the rough location of eyes, noses, mouths and ears.

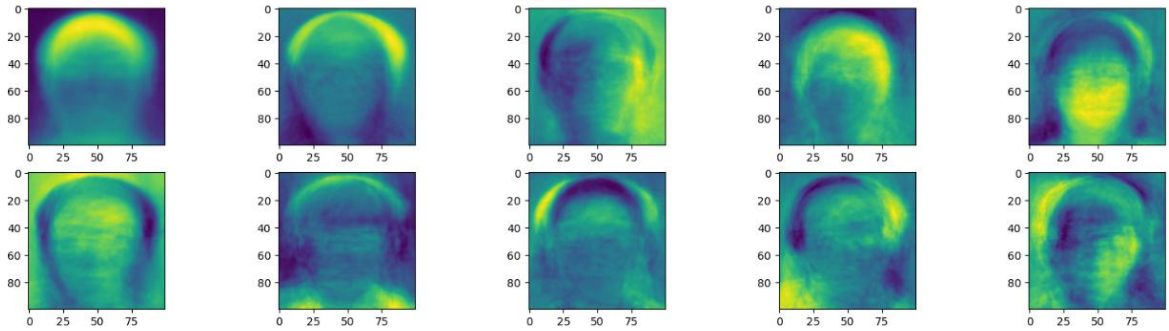
```
fig = plt.figure(figsize=(20, 5))
rows = 2
column = 4

# Number of eigenvectors
n_evec = [n for n in range (20, 201, 40)]

# Index of image
id = 1

# Start plotting the approximated images
for i in range(len(n_evec)):
    fig.add_subplot(rows, columns, i+1)

    img1_coeffs = np.linalg.pinv(real_evec[:,np.array(range(n_evec[i]))])@(face_data_eig[id].T)
    imgr_1 = real_evec[:,np.array(range(n_evec[i]))]@img1_coeffs

    plt.title(f"id[{id}]: number of eigenvectors = {n_evec[i]}")
    plt.imshow(imgr_1.reshape(100,100), cmap='gray')

    plt.show
```



We can reconstruct our image using only a portion of eigenvectors. The total number of eigenvectors is as same as the number of images we train. The question is, how many eigenvectors are enough? We calculate the pseudo-inverse of our most impactful eigenvectors and multiply it with the face data matrices to create image coefficients. This image coefficient is used for classification later. Next, we multiply the most impactful eigenvectors with these coefficients to form image reconstruction. This code then prints out the image reconstructed with top 20, 60, 100, 140 and 180 eigenvectors. If the reconstructed image has too few eigenvectors, it will be blurry. However, we only need half amount of total eigenvectors to reconstruct our face data well. In this demonstration, we produce image coefficients from the top 25 and 100 eigenvectors.

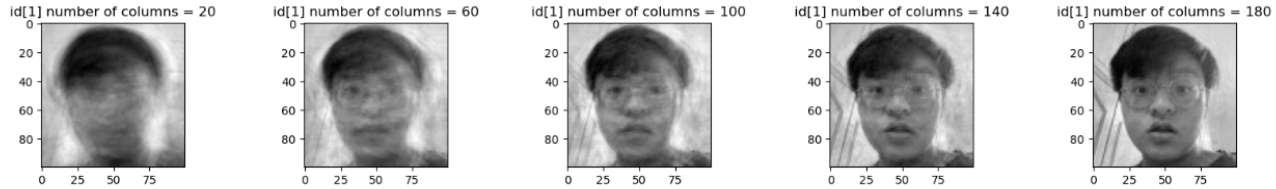
2.3 Singular Value Decomposition

```
# Perform Singular Value Decomposition
U, S, VT = np.linalg.svd(face_data_svd)
```



Performing the SVD task is much easier as we do not need to do any extra process. We throw in the preprocessed dataset into the NumPy module and it will decompose the dataset into a right singular vector matrix (U), a diagonal singular matrix (S) and the left singular vector matrix (V^T). The left singular vector matrix acts like our eigenvector in the eigendecomposition method. When we print out the image, we can also see the face outline and the approximate location of eyes, mouths, etc (Pramoditha, 2025).

```
fig = plt.figure(figsize=(20, 5))
rows = 2
columns = 5
k_columns = [k for k in range (20, 201, 40)]
id = 1
for i in range (0, len(k_columns)):
    fig.add_subplot(rows, columns, i+1)
    face_data_svd_approx = U[:, :k_columns[i]] @ np.diag(S)[:k_columns[i], :k_columns[i]] @ VT[:k_columns[i], :]
    plt.imshow(face_data_svd_approx[id].reshape(100, 100), cmap='gray')
    plt.title(f"id[{id}] number of columns = {k_columns[i]}")
    plt.show
```



We reduce the number of columns in U, S and V^T matrices instead of the number of eigenvectors in image reconstruction with the SVD method. The total number of columns is also as same as the number of images trained. It is rather simple to reconstruct images using the SVD method. We simply just do matrix multiplication on these three matrices together. However, we extract the diagonal of the singular

matrix before doing matrix multiplication. This is just to ensure that our singular matrix is indeed diagonal.

As we can see in the code segment, having a low amount of SVD columns in the image reconstruction does make the image blurry, just like its equivalent which is formed from eigenvalues and eigenvectors. We also do not need a high number of columns to reconstruct images. An image that is reconstructed using 50% of the columns is already a great image that reconstructs most of the human face features.

```
image_path = "D:/Face_recognition_cropped/Test/test7.jpg"

input_image = np.array([image.imread(image_path)])

gray = lambda rgb : np.dot(rgb[... , :3] , [0.299 , 0.587, 0.114])
input_image = np.array([gray(input_image)])

input_image = np.array([input_image.flatten()])
```

```
img1_coeffs = np.linalg.pinv(real_evec[:,np.array(range(k))])@(input_image[0])
```

```
svd_test_image = input_image.flatten()
```

2.4 Classifiers

We will try to import a new image to see how well it performs. We first have to do preprocessing. Then, we will produce the eigencoefficient of the input image by multiplying the input with the pseudo-inverse of eigenvectors. We will then flatten the input image data for SVD classifiers.

We have coded 3 different types of classifiers to determine the name of the faces. They are Euclidean distance, cosine similarity and K-Nearest Neighbours. Our training dataset and input data are the same in all 3 classifiers. However, we use the eigendecomposition coefficient of the whole dataset if our input is a coefficient. Furthermore, we use the image dataset reconstructed by SVD if our input is flattened image data.

```
# Finding Euclidean Distance
safeguard_distance = 40 # must be lower than 40

img1_coeffs = np.linalg.pinv(real_evec[:,np.array(range(k))])@(input_image[0])

euclidean_eig_distances = np.linalg.norm(coefficient_matrix - img1_coeffs, axis=1)

# Find the index of the closest image
euc_eig_image_index = np.argmin(euclidean_eig_distances)

# Retrieve the closest image
euclidean_eig_closest_image = face_data_eig[euc_eig_image_index]
```

Euclidean distance is the straightest and shortest distance of two points or vectors in an Euclidean space (GeeksforGeeks, 2025b). Therefore, we only need to find the different matrices of each image in the training dataset and the input image. Then, we perform the Frobenius norm on matrices (square root of the sum of the absolute squares of matrix elements) to find the Euclidean distance. This input will classify as the name in the training image that gives the smallest Euclidean distance.

```
from scipy.spatial.distance import cosine

# Compute cosine similarity (1 - cosine distance)
eig_cosine_distances = [cosine(img1_coeffs, coefficient_matrix[i, :])
             for i in range(coefficient_matrix.shape[0])]

cosine_eig_closest_image_index = np.argmin(eig_cosine_distances)  # Find the maximum similarity

cosine_eig_closest_image = face_data_eig[cosine_eig_closest_image_index]
```

Cosine similarity measures the cosine of the angle between two vectors (GeeksforGeeks, 2025c). We calculate the cosine distance directly using the cosine module by SciPy. Then, we can classify the image that has the smallest cosine distance to be the one matching the input image.

```
knn_eig_distances, knn_eig_indices = knn.kneighbors(knn_eig_input_coeffs)
```

```
closest_eig_knn_labels = []

for i in knn_eig_indices.flatten():
    closest_eig_knn_labels.append(y_train[i])
```

```
closest_eig_knn_labels
```

```
['Alvin', 'William', 'YaoWu']
```

KNN in our project finds the k neighbouring training images with the smallest cosine similarities. KNN will then find the name that appears the most in the neighbour list.

2.5 Observations

We used different new faces to project the accuracy of these classifiers. The results showed that KNN has the best accuracy in identifying the faces. Although it is not 100% accurate, it is able to predict most of my inputs' names correctly. Cosine similarity seems to be good at predicting only the faces of certain users. Meanwhile, Euclidean distance will always project one specific image to be the one with the smallest distance, even if we change our input image. If we change our training dataset, it will continue to project another specific image. Therefore, we create our real-time face recognition system with KNN and cosine similarity as classifiers.

In addition, image approximations created by SVM and Eigendecomposition methods give similar results no matter which classifier we use. However, the decomposition of SVM runs faster than eigendecomposition. This naturally makes the SVM method the better choice. However, our priority is to look at how accurate our face recognition system is. We save our eigenvectors, eigencoefficients and SVD image approximations. We also save our KNN models so we can directly import them into our real-time recognition system. We will not have to waste time decomposing the training dataset or train the KNN model every time we start our system.

Furthermore, the number of eigenvectors / SVD columns used to create image approximation does not seem to affect the accuracy of our classifier. Therefore, we recommend only applying a small number of them. This can reduce the time required for classifier training and the code's memory space. We use the top 25 number of eigenvectors / SVD columns to train our recognition system.

3. Real-Time Face Recognition System

3.1 Code Explanation

```
# Define the directory where images are stored  ERROR: Failed building wheel for dlib
image_directory = "D:\Old_Recognition_Cropped" # original faces
eigen_directory = "D:\\Eigen_Approximations\\" # Eigendecomposition coefficient matrix
svd_directory = "D:\\SVD_Approximations\\" #SVD Image Approximation
main_directory = "D:\\"

os.chdir(image_directory)
faces_list = os.listdir()

#Load Eigen
os.chdir(eigen_directory)
n_list = os.listdir()
```

```
os.chdir(main_directory)
with open("knn_model1.pkl", "rb") as fp:
    knn = pickle.load(fp) #Eigen

with open("knn_model2.pkl", "rb") as fp:
    knn2 = pickle.load(fp) #SVD
```

We first have to load in the aforementioned eigendecomposition and SVD datasets into our face recognition system. We also imported the names of each face and the KNN models.

```
# Initialize webcam
video_capture = cv2.VideoCapture(0)

cascPath = os.path.dirname(cv2.__file__) + "/data/haarcascade_frontalface_default.xml"
faceCascade = cv2.CascadeClassifier(cascPath)
```

We will use OpenCV module to open our camera. We also set the length and width of our camera UI. We will also create a Haar-feature-based cascade classifier. It is a cascade function (a function that runs repeatedly in a sequential order) where face objects are extracted based on the brightness of features in lighter and darker regions.

```
while True:
    ret, frame = video_capture.read()
    grey = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    frame2= cv2.resize(frame, dsize: (100,100))
    gray = lambda rgb: np.dot(rgb[..., :3], b: [0.299, 0.587, 0.114])
    frame2 = np.array([gray(frame2)])
    frame2 = frame2.flatten()

    # Detect faces in the image
    face_locations = faceCascade.detectMultiScale(
        grey,
        scaleFactor=1.05,
        minNeighbors=5,
        minSize=(100, 100),
        flags=cv2.CASCADE_SCALE_IMAGE
    )
```

Our computer camera will first capture a frame. Then, the frame is preprocessed and flattened into a grayscale 100x100 one-dimensional array. The cascade classifier will then try to detect the face objects in the frame. We set the minimum size of the image to be 100 x 100 pixels, and each object should have at least 5 neighbouring objects surrounding it. The classifier returns a list of face locations. Each location consists of the x and y coordinates of a starting point and the length and width of the whole face object.

```
    for (x, y, w, h) in face_locations:
        name = "Unknown"

        # Eigen Coefficient Calculation
        input_coefficients = evec @ frame2

        # Eigen Cosine Similairty
        matches = [1 - cosine(input_coefficients, approximations[i])
                                for i in range(approximations.shape[0])]

        # Eigen KNN
        input_coefficients = [input_coefficients]
        name = knn.predict(input_coefficients)[0]
```

```
#Cosine Similairity Checking
if max(matches) > 0.6:
    closest_image_index = np.argmax(matches)
    tokens = word_tokenize(faces_list[closest_image_index])
    name = tokens[0]
```

```
#SVD Cosine Similarity

matches = [1 - cosine(frame2, approximations[i])
                    for i in range(approximations.shape[0])]

#SVD KNN
frame2 = [frame2]
name = knn2.predict(frame2)[0]
```

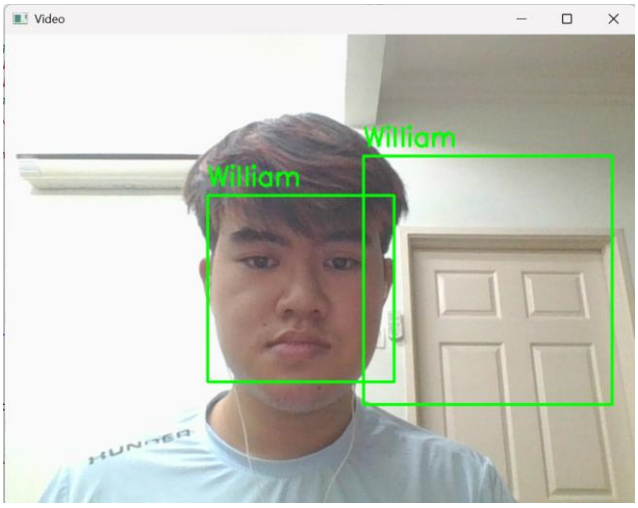
We calculate the cosine similarity between each face object and every single eigen coefficient or SVD image approximation and obtain the ones that give the shortest cosine distance. We also check whether this cosine distance is short enough for us to actually use the face associated with it. We consider the cosine distance should be less than 0.4. Alternatively, we can obtain the closest image predicted by our imported KNN models. We only use one classifier per run so it will not potentially corrupt our result.

```
image_area = (x+w) * (y+h)

if image_area > largest_area:
    largest_area = image_area
    largest_x = x
    largest_y = y
    largest_w = w
    largest_h = h
cv2.rectangle(frame, (largest_x, largest_y), (largest_x + largest_w, largest_y + largest_h), (0, 255, 0), 2)
cv2.putText(frame, name, org: (largest_x, largest_y - 10), cv2.FONT_HERSHEY_SIMPLEX, fontScale: 0.9,
            color: (0, 255, 0), thickness: 2)
```

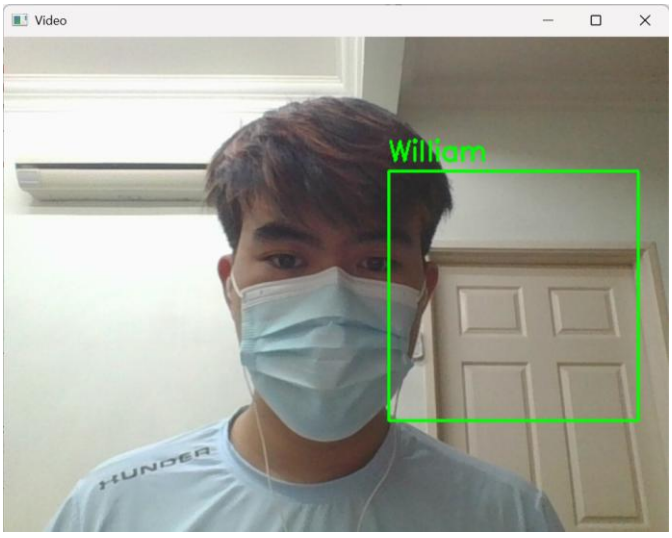
Now that we have associated each face object with an image label, we need to determine which face object we should use in the frame. We use the object that has the largest surface area as it is generally the entire face of the person we want to detect. After determining the face object that has the largest surface area, the program draws a rectangle surrounding the outlines of the face object and states the name of the face object on top of it. Finally, the program showed the final result and take the frame of the next photo.

3.2 Observations (n_components = 25)

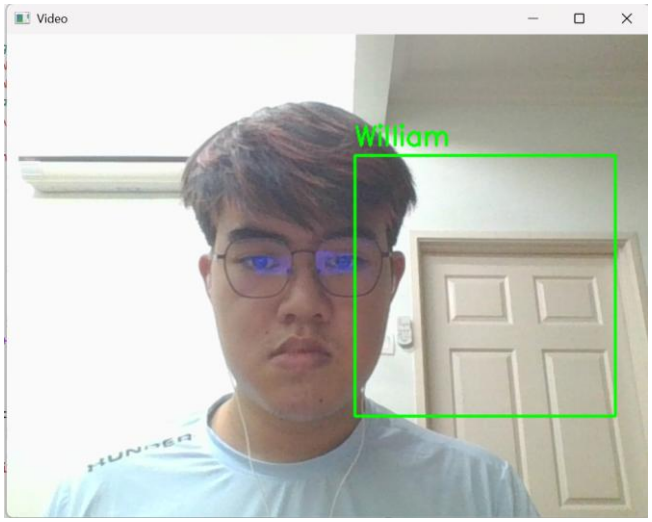
1)Face only (Subject: Wong Yue Wei)



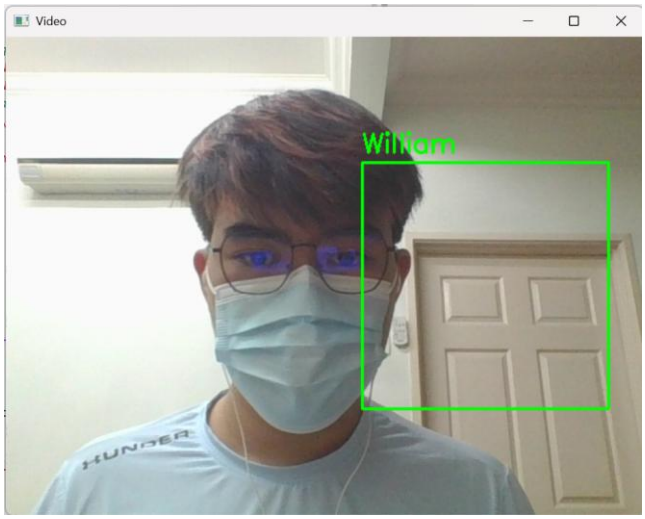
2) With face mask (Subject: Wong Yue Wei)



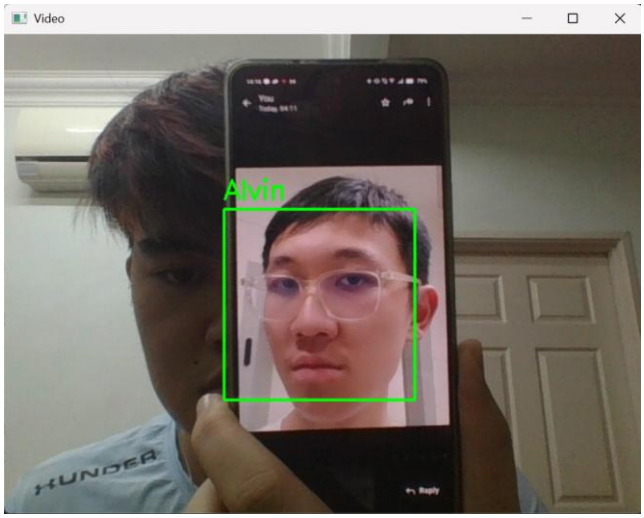
3) With glasses (Subject: Wong Yue Wei)



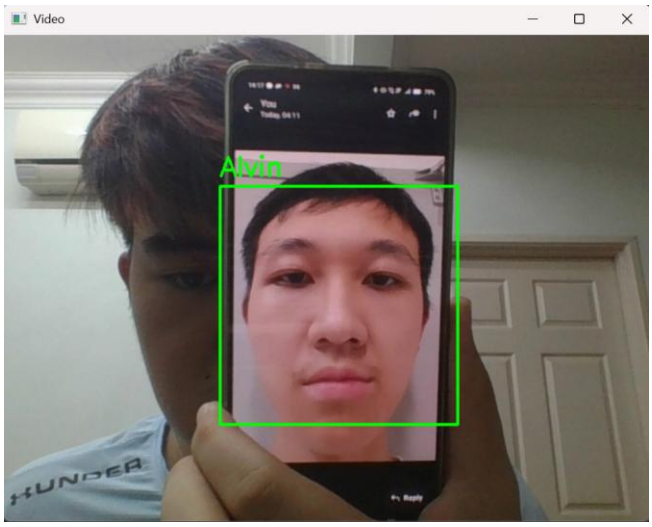
4) With mask and glasses (Subject: Wong Yue Wei)



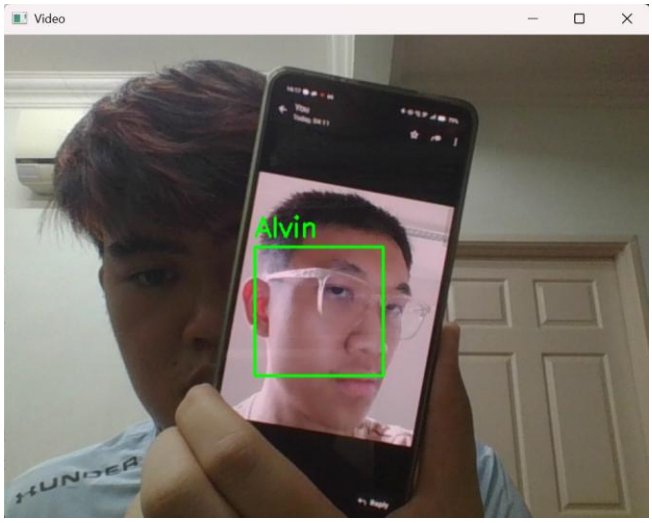
5) Image from dataset (Subject: William, 1st image)



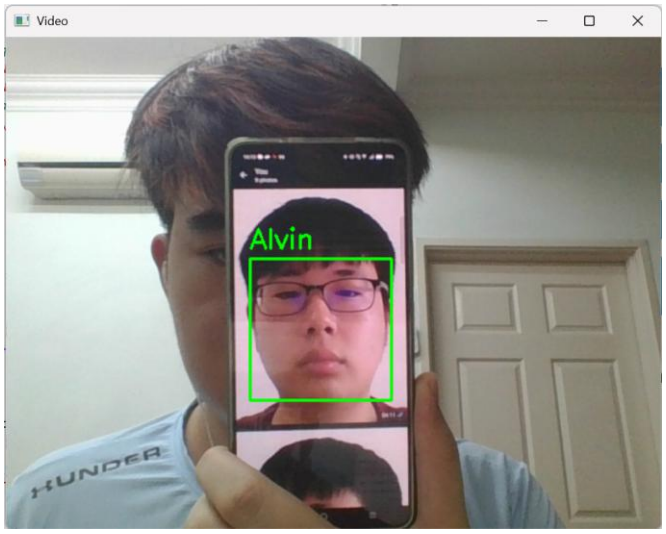
6) Image from dataset (Subject: William, 2nd image)



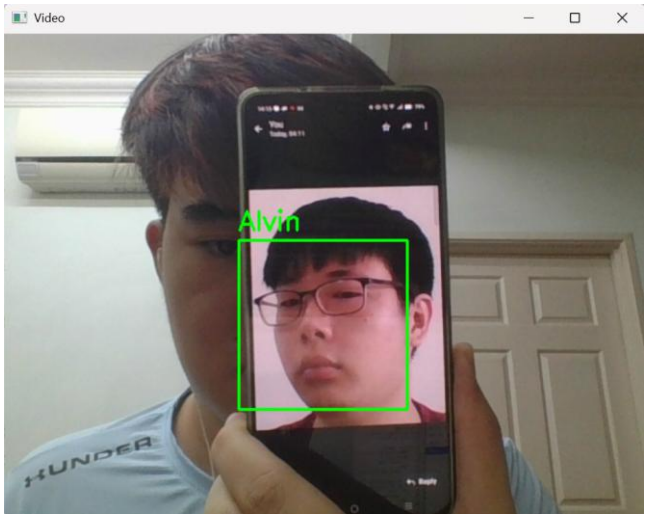
7) Image from dataset (Subject: William, 3rd image)



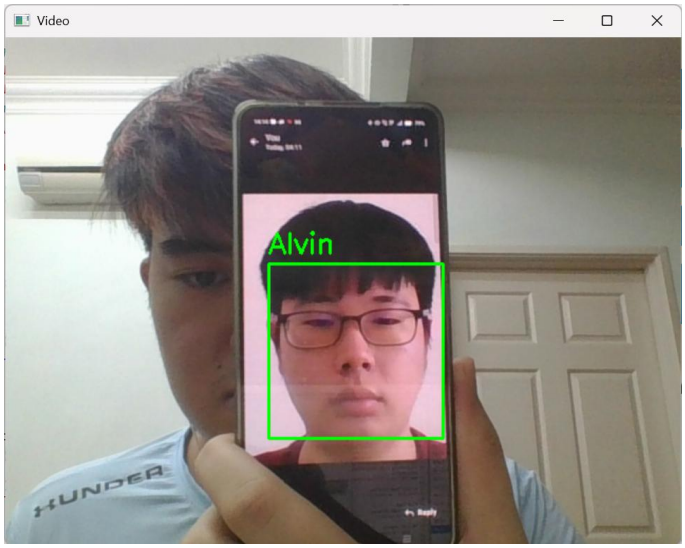
8) Image from dataset (Subject: Han Yong, 1st image)



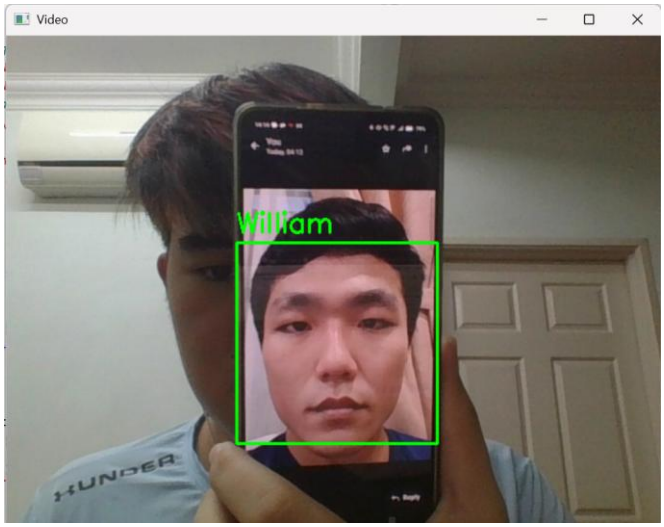
9) Image from dataset (Subject: Han Yong, 2nd image)



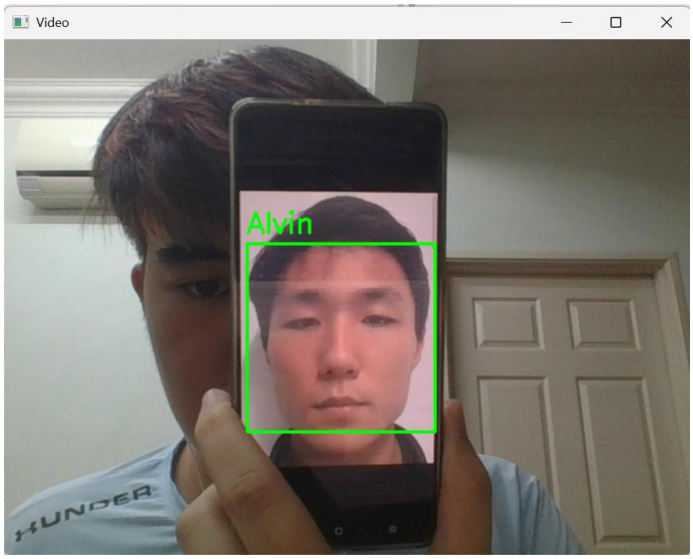
10) Image from dataset (Subject: Han Yong, 3rd image)



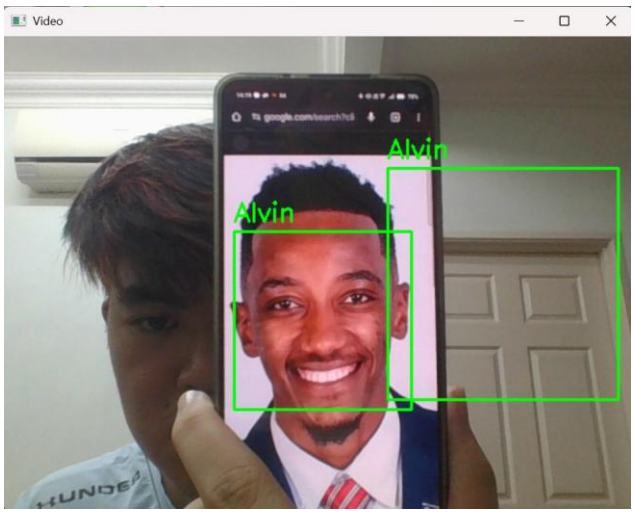
11) Image from dataset (Subject: Yao Wei)



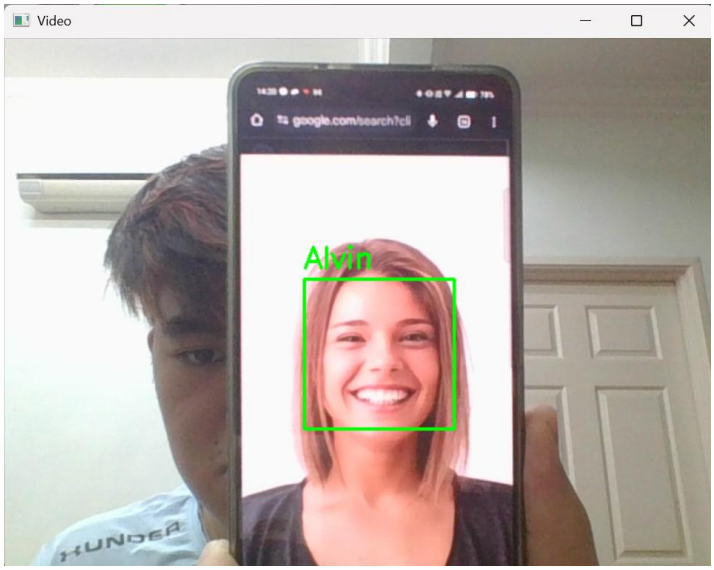
12) Image from dataset (Subject: Yao Wu)



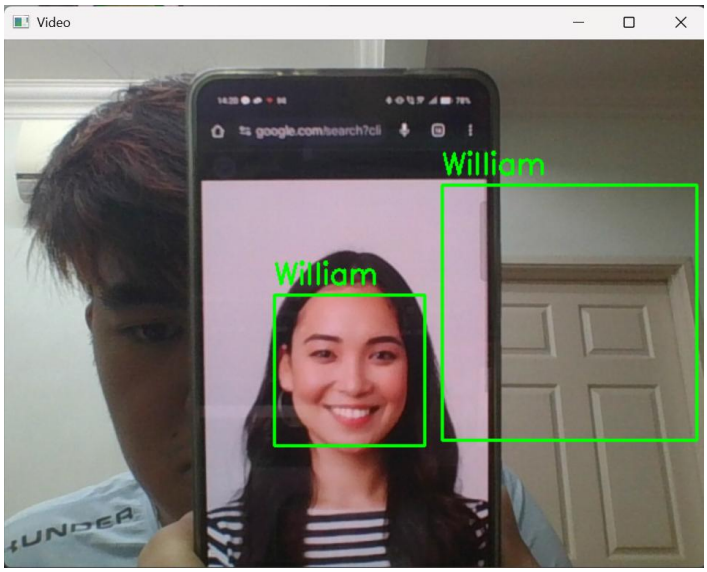
13) Image not in dataset



14) Image not in dataset



15) Image not in dataset



Total Accuracy = 0/15 = 0%

Some flaws have been shown when we are implementing the UI of our face recognition system. Even though both cosine similarity and KNN classifier work during training, they are unable to recognise the faces in dynamic frames correctly. Sometimes, they may misinterpret an empty space as the location of a face object too. Even though we have set up a threshold to clear out the unknown image, their confidence score is as high as the faces known in the dataset. Therefore, we have to make some improvements to our training models and recognition system.

4 Improved Prototype

4.1 Imports and Configuration

Cell (1): Code Overview

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import LabelEncoder

# Configuration
dataset_path = r"C:\Users\OMGWh\Pictures\Face recognition cropped"
prototxt_path = r"C:\Users\OMGWh\Downloads\deploy.prototxt.txt"
caffemodel_path =
r"C:\Users\OMGWh\Downloads\res10_300x300_ssd_iter_140000.caffemodel"
eye_cascade_path = cv2.data.haarcascades + 'haarcascade_eye.xml'
target_size = (100, 100)
n_components = 25
base_confidence_threshold = 0.7
min_face_size = 50
threshold_multiplier = 2
```

We slightly modify our dataset before starting our improved prototype. Now, we do not force the size of our dataset to be 100 x 100 pixels. Our images are still cropped but we do not manually resize them. We will use the cascade classifier to extract faces from images and resize them.

In cell (1), the code begins by importing essential libraries. ***Os*** handles file handling. ***Cv2 (OpenCV)*** for face detection, image processing, and DNN operations. ***NumPy*** performs numerical operations (Matrix operations for PCA calculation and data handling). ***Matplotlib*** is used to plot visualization of training images, eigenvalues graph and eigenfaces. ***Scikit-learn's KNeighborsClassifier*** and ***LabelEncoder*** are imported for machine learning tasks.

The configuration parameters that define paths to the dataset with face images are provided. The ***pre-trained Caffe models*** (prototxt and caffemodel files) are used for deep learning-based face detection with high-accuracy. Besides that, ***Haar cascade XML*** file is used for face alignment by detecting the locations of eyes and measuring their distances.

The key settings include the target face size (100×100 pixels) and the number of principal components (25) for PCA. When the number of principal components is 25 for PCA, it provides a balanced trade-off between computational efficiency and retained facial features that gives the highest accuracy. We also set the thresholds for face detection confidence (0.7) and adaptive recognition.

These configurations are the standard for the process to proceed, ensuring consistency across data loading, preprocessing, and real-time inference.

4.2 PCA Implementation with Eigenvalue Storage

Cell (2): Code Overview

```
class ManualPCA:
    def __init__(self, n_components):
        self.n_components = n_components
        self.mean = None
        self.components = None
        self.eigenvalues = None

    def fit(self, X):
        self.mean = np.mean(X, axis=0)
        X_centered = X - self.mean
        cov_matrix = np.cov(X_centered, rowvar=False)
        eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)
        idx = np.argsort(eigenvalues)[::-1]
        self.eigenvalues = eigenvalues[idx]
        self.components = eigenvectors[:, idx][:, :self.n_components]

    def transform(self, X):
        return np.dot(X - self.mean, self.components)
```

A custom ***ManualPCA*** class is implemented to manually perform Principal Component Analysis. ***ManualPCA*** class is used, so users have full control over eigenvalue storage for analysis and calculation for the decomposition. The class computes the mean of the input data, centers the data by subtracting this mean, and calculates its covariance matrix. Eigendecomposition is then applied to the covariance matrix to extract eigenvalues and eigenvectors, which are sorted in descending order of variance.

Next, the top “n_components” eigenvectors (principal components) are stored, along with their corresponding eigenvalues. The transform method projects input data onto these components while reducing the dimensionality. Storing the eigenvalues and eigenvectors enables plot visualization of the variance distribution of eigenvalues in a graph, and also eigenfaces.

Mathematic formula used:

1. **Centering Data**:
   Subtract mean from each sample:

$$X_{centered} = X - \mu \qquad (\mu = \frac{1}{N}\sum_{i=1}^{N} X_i)$$

2. **Covariance Matrix**:
   Compute covariance to identify feature relationships:

$$\mathrm{Cov} = \frac{\mathrm{X}_{\mathrm{centered}}^{\mathrm{T}} \cdot \mathrm{X}_{\mathrm{centered}}}{\mathrm{N} - 1}$$

3. **Eigendecomposition**:
   Solve for eigenvalues (λ) and eigenvectors (v):

$$Cov \cdot v = \lambda \cdot v$$

   Eigenvectors (also known as principal components) are then sorted by descending eigenvalues.

4.3 Image Processing Pipeline

Cell (3): Code Overview

```
def augment_face(face):
    augmented = [face]
    augmented.append(cv2.flip(face, 1))
    for angle in [-15, 15]:
        M = cv2.getRotationMatrix2D((face.shape[1]//2, face.shape[0]//2), angle, 1)
        rotated = cv2.warpAffine(face, M, (face.shape[1], face.shape[0]))
        augmented.append(rotated)
    augmented.append(cv2.GaussianBlur(face, (3,3), 0))
    bright = cv2.convertScaleAbs(face, alpha=1.3, beta=40)
    augmented.append(bright)
    return augmented

def preprocess_face(face):
    face_eq = cv2.equalizeHist(face)
    clahe = cv2.createCLAHE(clipLimit=2.5, tileGridSize=(8,8))
    return clahe.apply(face_eq)

def align_face(gray_face, eye_cascade):
    if gray_face.shape[0] < min_face_size or gray_face.shape[1] < min_face_size:
        return gray_face

    try:
        eyes = eye_cascade.detectMultiScale(gray_face, scaleFactor=1.05,
minNeighbors=8, minSize=(35,35))
        if len(eyes) >= 2:
            eyes = sorted(eyes, key=lambda x: x[0])
            left_eye = (eyes[0][0] + eyes[0][2]//2, eyes[0][1] + eyes[0][3]//2)
            right_eye = (eyes[1][0] + eyes[1][2]//2, eyes[1][1] + eyes[1][3]//2)

            dx = right_eye[0] - left_eye[0]
            dy = right_eye[1] - left_eye[1]
            angle = np.degrees(np.arctan2(dy, dx))

            center = (gray_face.shape[1]//2, gray_face.shape[0]//2)
            M = cv2.getRotationMatrix2D(center, angle, 1)
            return cv2.warpAffine(gray_face, M, gray_face.shape[::-1],
flags=cv2.INTER_CUBIC)

        return gray_face
    except:
        return gray_face
```

3 different key functions have been proposed:

- augment_face(): Generates augmented and mirrored versions of a face image through horizontal flipping for training. We applied rotation (±15°) to produce images with tilted head for training. Lastly, Gaussian blur and brightness adjustments, are used to process varying lighting conditions with images for training. While our dataset is small, this function can be used to expand the training dataset, thus improving model robustness to variations in pose and lighting.
- preprocess_face(): Enhances contrast using histogram equalization and CLAHE (Contrast Limited Adaptive Histogram Equalization), which limits **noise amplification** (A phenomenon of an image processing technique. When we attempt to enhance certain features or details in an image, it inadvertently increases the visibility of existing noise in an image, making unwanted random variations in the image to be more prominent instead of being ignored) while improving local contrast.
- align_face(): Detects eyes using a Haar cascade, computes the angle between them and rotates the face to align the eyes horizontally. This ensures consistent facial orientation, which is crucial for PCA's sensitivity to differentiate patterns.

Mathematical Operations

- **Rotation Matrix**:

$$M_{rotation} = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \end{bmatrix}$$

Where θ is the angle between eyes, and t_x and t_y are translations to keep the face centered.

These steps are used to standardize face images before putting them into the PCA reduction and KNN classifier.

4.4 Data Loading & Visualization

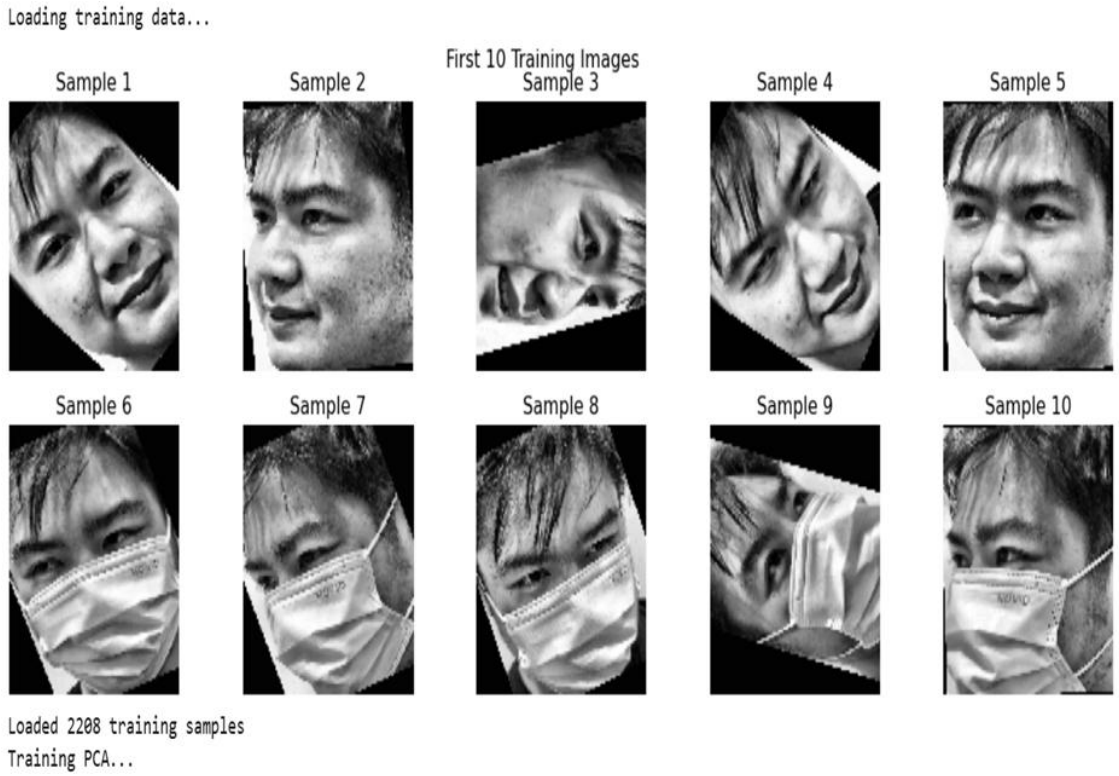
Code Overview

```
def load_training_data(dataset_path):
    net = cv2.dnn.readNetFromCaffe(prototxt_path, caffemodel_path)
    eye_cascade = cv2.CascadeClassifier(eye_cascade_path)
    X, labels = [], []

    # Collect first 10 images for visualization
    first_10_images = []

    for person_name in os.listdir(dataset_path):
        person_dir = os.path.join(dataset_path, person_name)
        if not os.path.isdir(person_dir):
            continue

        for img_name in os.listdir(person_dir):
            img_path = os.path.join(person_dir, img_name)
            img = cv2.imread(img_path)
            if img is None:
                continue

            h, w = img.shape[:2]
            blob = cv2.dnn.blobFromImage(cv2.resize(img, (300, 300)), 1.0,
                                (300, 300), (104.0, 177.0, 123.0))
            net.setInput(blob)
            detections = net.forward()

            for i in range(detections.shape[2]):
                confidence = detections[0, 0, i, 2]
                if confidence > 0.5:
                    box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
                    startX, startY, endX, endY = box.astype("int")
                    face_roi = img[startY:endY, startX:endX]

                    if face_roi.size == 0:
                        continue

                    gray = cv2.cvtColor(face_roi, cv2.COLOR_BGR2GRAY)
                    aligned = align_face(gray, eye_cascade)
                    processed = preprocess_face(aligned)
                    resized = cv2.resize(processed, target_size)

                    # Store first 10 images
                    if len(first_10_images) < 10:
                        first_10_images.append(resized)

                    for augmented in augment_face(processed):
                        resized = cv2.resize(augmented, target_size)
```

```
                X.append(resized.flatten())
                labels.append(person_name)

    # Plot first 10 images
    plt.figure(figsize=(15, 5))
    for i in range(10):
        plt.subplot(2, 5, i+1)
        plt.imshow(first_10_images[i], cmap='gray')
        plt.axis('off')
        plt.title(f"Sample {i+1}")
    plt.suptitle("First 10 Training Images", y=0.95)
    plt.show()

    return np.array(X), np.array(labels)
```

The ***load_training_data*** function iterates through the dataset directory, using the Caffe model to detect and crop faces from each image. Next, the images are converted to grayscale. Detected faces are aligned, preprocessed, resized to 300 x 300 pixels to retain more facial details, and augmented (it generates 6 randomly chosen variants per face (flip, rotate, blur, bright). The first 10 preprocessed faces are plotted using Matplotlib to verify alignment and preprocessing quality. Caffe DNN Detector is used because it has higher accuracy than Haar cascades for complex angles and lighting impact.

First 10 preprocessed faces



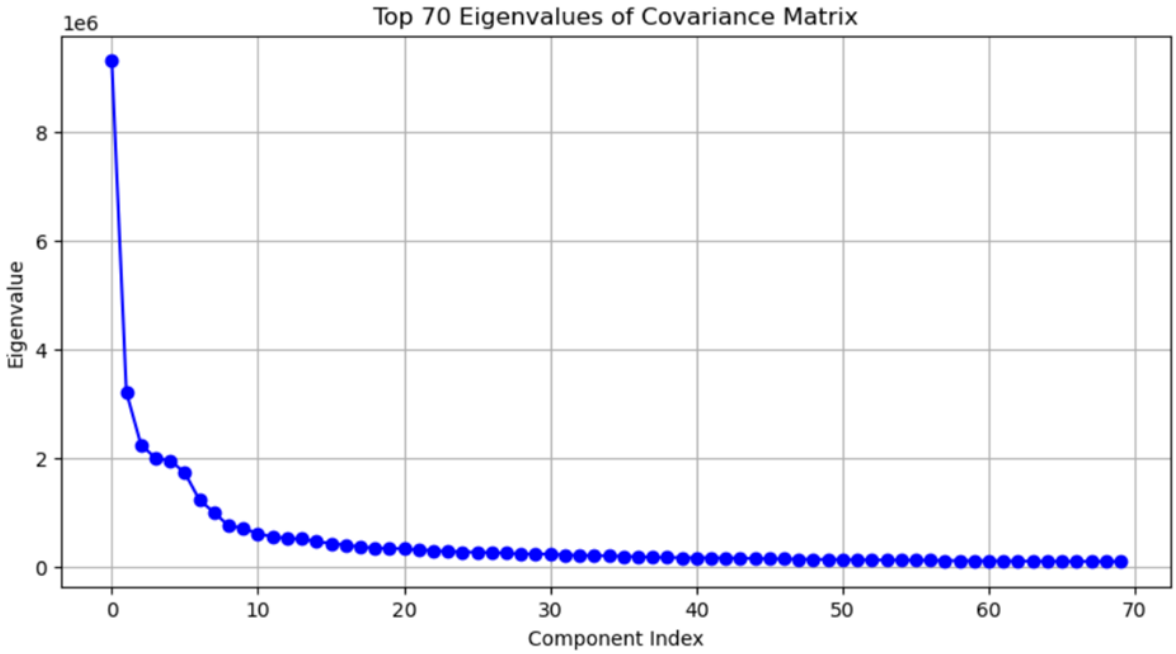
4.5 Model Training & Eigen Analysis

Code Overview

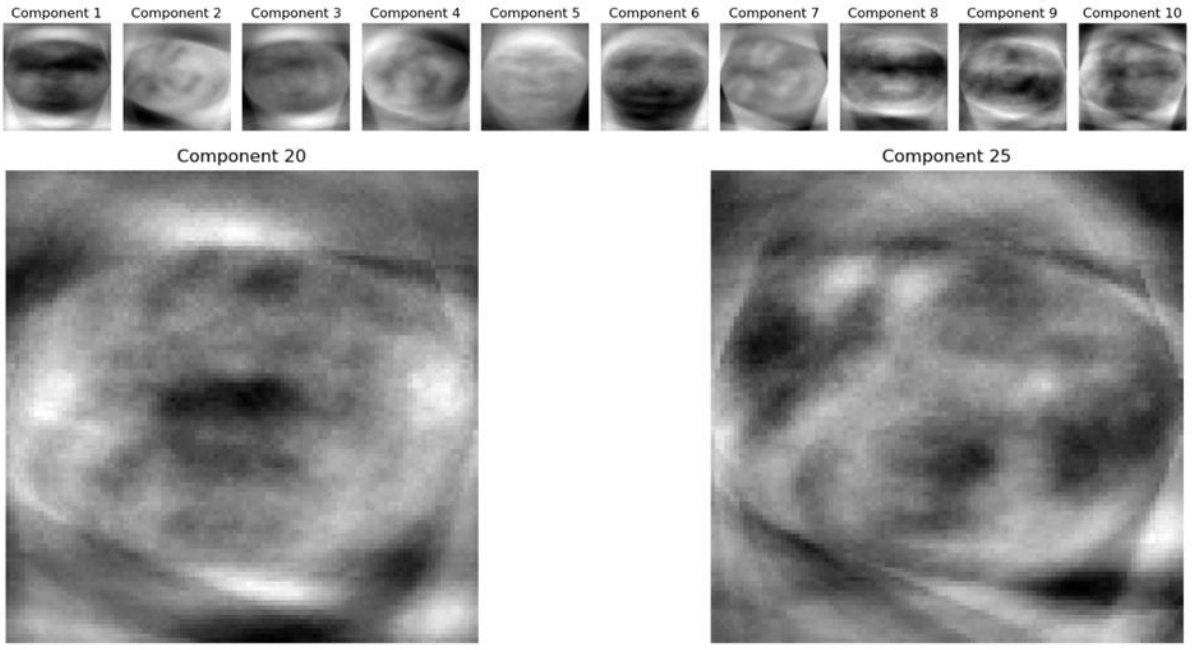
```
# Load data
print("Loading training data...")
X_train, y_train = load_training_data(dataset_path)

if len(X_train) == 0:
    raise ValueError("No training data found! Check your dataset path.")

print(f"Loaded {len(X_train)} training samples")

# Encode labels
le = LabelEncoder()
y_encoded = le.fit_transform(y_train)

# Train PCA
print("Training PCA...")
pca = ManualPCA(n_components)
pca.fit(X_train)
X_pca = pca.transform(X_train)

# Eigenvalue visualization
plt.figure(figsize=(10, 5))
plt.plot(pca.eigenvalues[:70], 'b-o')
plt.title('Top 70 Eigenvalues of Covariance Matrix')
plt.xlabel('Component Index')
plt.ylabel('Eigenvalue')
plt.grid(True)
plt.show()

# Eigenface visualization function
def plot_eigenfaces(components, target_size, indices):
    plt.figure(figsize=(15, 5))
    for i, idx in enumerate(indices):
        plt.subplot(1, len(indices), i+1)
        eigenface = components[:, idx].reshape(target_size)  # Fixed indexing
        plt.imshow(eigenface, cmap='gray')
        plt.title(f'Component {idx+1}')
        plt.axis('off')
    plt.tight_layout()
    plt.show()

# First 10 eigenfaces
plot_eigenfaces(pca.components, target_size, range(10))

# Specific components
plot_eigenfaces(pca.components, target_size, [19, 24])  # Components 20, 25
```

In this part, the data will be loaded and preprocessed. The dataset will be fed into the ***ManualPCA*** reduction training. After training data is centered, PCA is applied to the data with dimensionality of 25 components. Eigenvalues are plotted in a graph to show the variance captured by the top 70 components. Eigenfaces are reshaped from stored eigenvectors as 100 x100 px grayscale images, then plotted to be visualized. Our code displays the first 10 eigenface components and then the 20th and 25th components. These components reveal certain dominant facial features such as eyes, nose etc. Labels are also encoded as integers using ***LabelEncoder***, which converts string labels (example: "Person 1") to integers. Labels are actually the names of each face in the dataset.

Graph of top 70 Eigenvalues of Covariance Matrix



Eigenfaces visualization



4.6 Recognition System Setup

Cell (6): Code Overview

```
# Normalize features
X_pca_normalized = X_pca / np.linalg.norm(X_pca, axis=1, keepdims=True)

print("Training classifier...")
knn = KNeighborsClassifier(n_neighbors=3, weights='distance', metric='cosine')
knn.fit(X_pca_normalized, y_encoded)

# Dynamic threshold calculation
def calculate_threshold(X_train_pca, knn):
    distances, _ = knn.kneighbors(X_train_pca, n_neighbors=2)
    distances = distances[:, 1]  # Exclude self
    return np.mean(distances) + threshold_multiplier * np.std(distances)

threshold = calculate_threshold(X_pca_normalized, knn)
print(f"Dynamic recognition threshold: {threshold:.2f}")
```

In cell (6), the PCA features will be normalized. PCA-transformed features are normalized to unit length for cosine distance calculations. A dynamic recognition threshold is also calculated using the KNN distances from the training data.

Mean distance to nearest neighbor (μ)and standard deviation of distance (σ) are all obtained in training data with the KNN model. A ***KNN classifier*** (k=3, cosine metric) is trained on these features to compare new faces against the training set. KNN is input with k=3 to obtain balanced noise reduction and locality. This adaptive threshold accounts for natural variability in the dataset, thus reducing false positives for unknown faces. The KNN model and threshold are now ready for real-time inference.

Mathematical Operations

1. **Feature Normalization**:
   $$X_{normal} = \frac{X_{PCA}}{\|X_{PCA}\|}$$
2. **Dynamic Threshold**:
   $$Threshold = \mu_{distances} + 2 \cdot \sigma_{distances}$$

4.7. Real-Time Implementation

Code Overview

```
cap = cv2.VideoCapture(0)
net = cv2.dnn.readNetFromCaffe(prototxt_path, caffemodel_path)
eye_cascade = cv2.CascadeClassifier(eye_cascade_path)

while True:
    ret, frame = cap.read()
    if not ret:
        break

    h, w = frame.shape[:2]
    blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)), 1.0,
                                 (300, 300), (104.0, 177.0, 123.0))
    net.setInput(blob)
    detections = net.forward()

    for i in range(detections.shape[2]):
        confidence = detections[0, 0, i, 2]
        if confidence > 0.5:
            box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
            (startX, startY, endX, endY) = box.astype("int")
            face_roi = frame[startY:endY, startX:endX]

            if face_roi.size == 0:
                continue

            try:
                gray = cv2.cvtColor(face_roi, cv2.COLOR_BGR2GRAY)
                aligned = align_face(gray, eye_cascade)
                processed = preprocess_face(aligned)
                resized = cv2.resize(processed, target_size).flatten().reshape(1, -1)

                face_pca = pca.transform(resized)
                face_pca_normalized = face_pca / np.linalg.norm(face_pca)

                distances, _ = knn.kneighbors(face_pca_normalized)
                avg_distance = np.mean(distances)
                confidence = (1 - avg_distance) * 100

                if avg_distance > threshold:
                    label = f"Unknown ({confidence:.1f}%)"
                else:
                    pred = knn.predict(face_pca_normalized)
                    name = le.inverse_transform(pred)[0]
                    label = f"{name} ({confidence:.1f}%)"
```

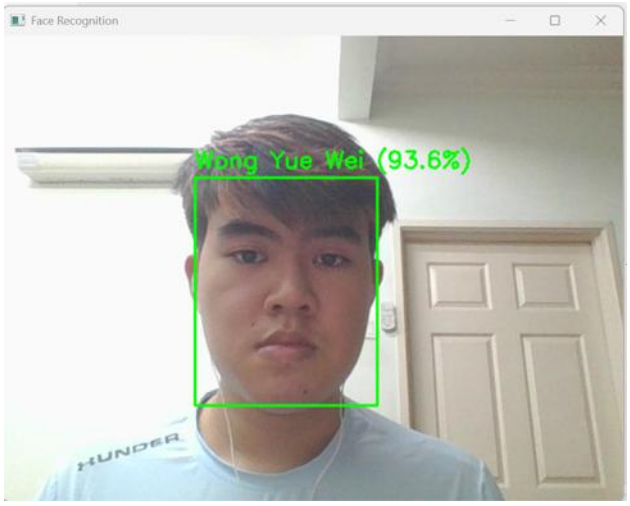
```
            cv2.rectangle(frame, (startX, startY), (endX, endY), (0,255,0), 2)
            cv2.putText(frame, label, (startX, startY-10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,255,0), 2)

        except Exception as e:
            pass

    cv2.imshow("Face Recognition", frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

The webcam feed is captured using "cv2.VideoCapture". Each frame is processed by the Caffe face detector. After that, detected faces undergo alignment, preprocessing, and PCA projection that are identical to the training phase. The normalized PCA features are fed into the KNN model to compute the cosine distances to their nearest neighbors. A confidence score is derived with a pre-built **Confidence Calculation** function**.** It shows how confident our recognition system is with its prediction.

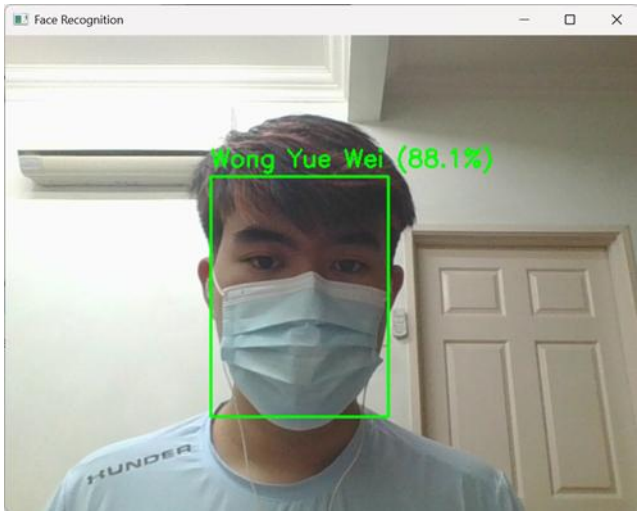
If the distance exceeds the dynamic threshold, the face is labelled "Unknown"; otherwise, the predicted label is displayed. When the face is detected, green bounding boxes and labels are rendered in real-time, and confidence scores are displayed beside the label. To exit the loop, the user must press 'Q'.

4.8 Observations (n_component = 25)

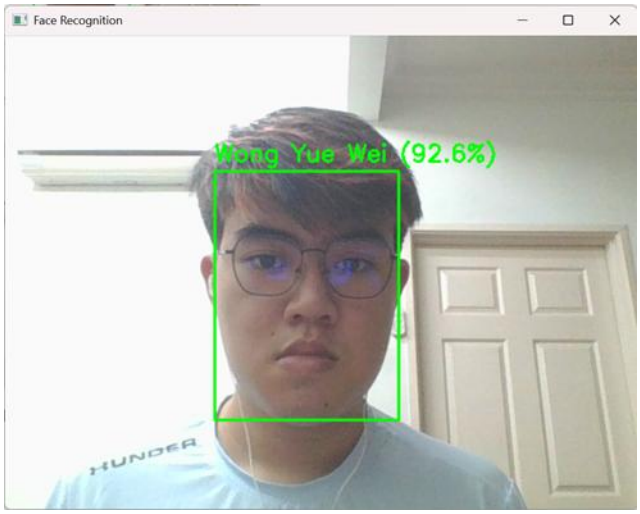
1)Face only (Subject: Wong Yue Wei)



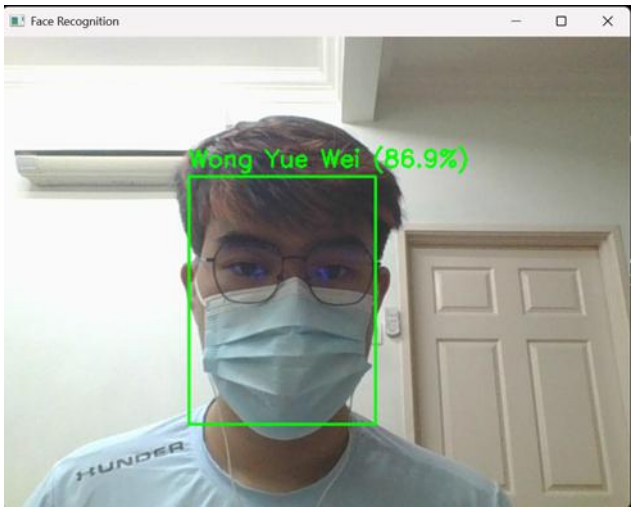
2) With face mask (Subject: Wong Yue Wei)



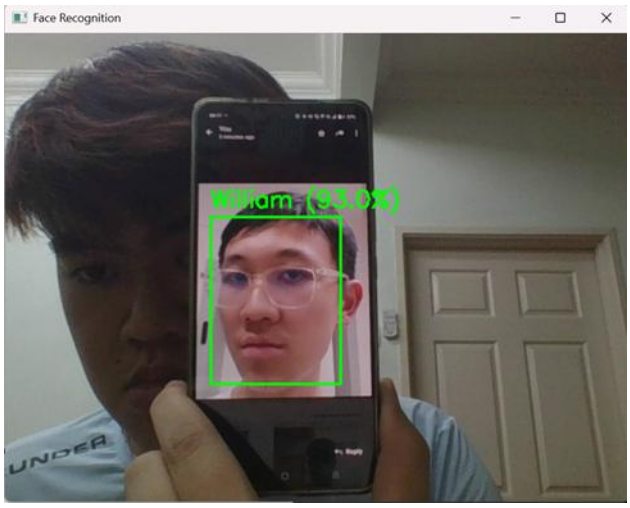
3) With glasses (Subject: Wong Yue Wei)



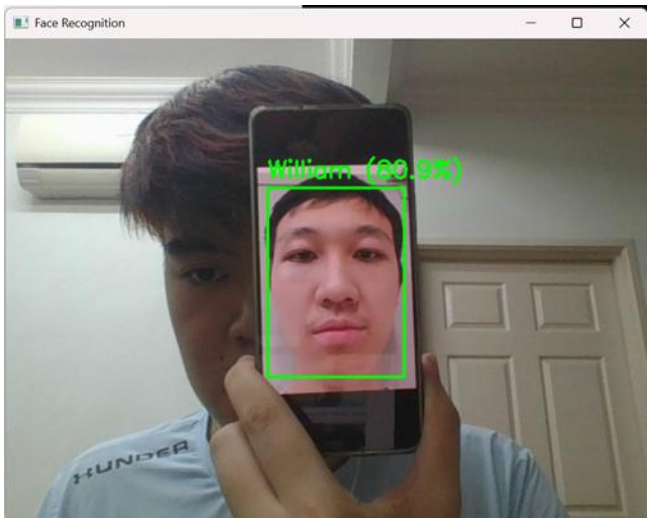
4) With mask and glasses (Subject: Wong Yue Wei)



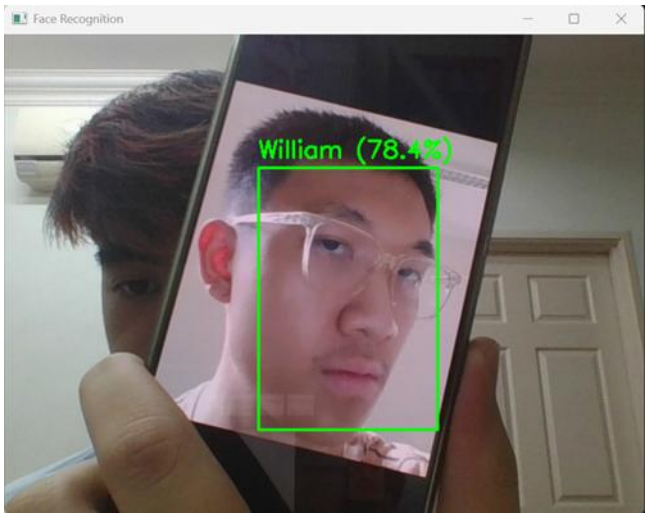
5) Image from dataset (Subject: William, 1st image)



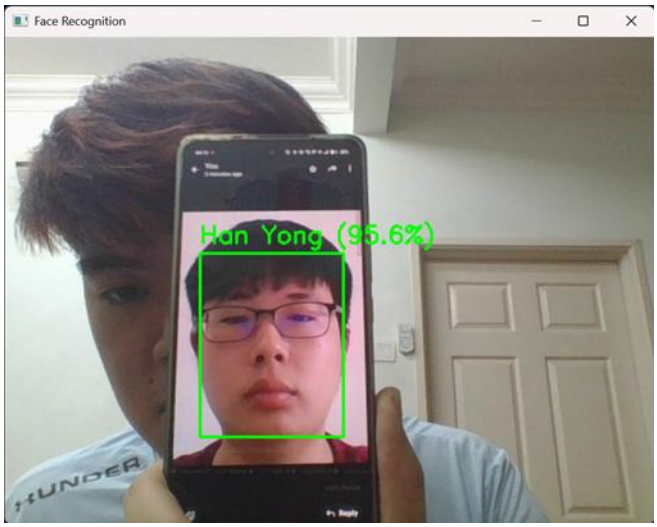
6) Image from dataset (Subject: William, 2nd image)



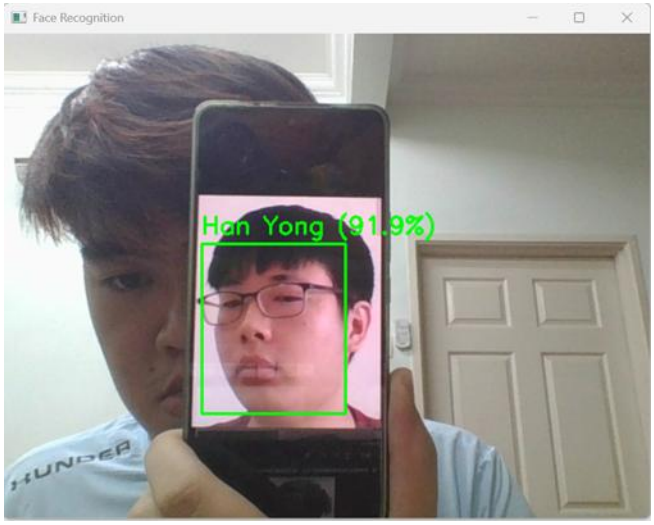
7) Image from dataset (Subject: William, 3rd image)



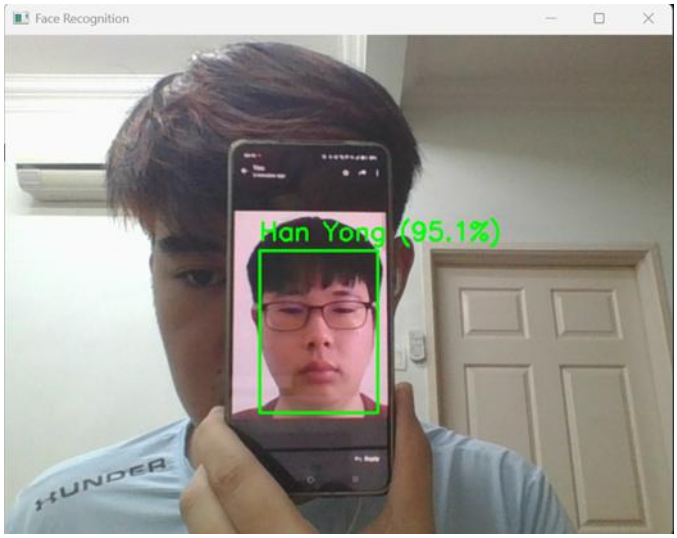
8) Image from dataset (Subject: Han Yong, 1st image)



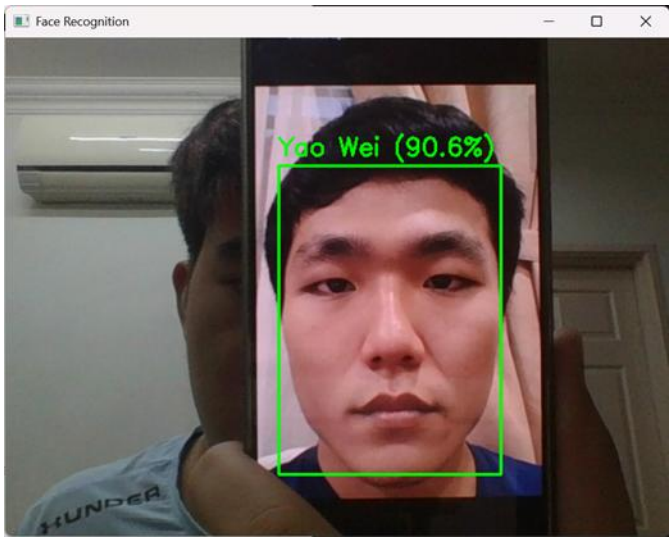
9) Image from dataset (Subject: Han Yong, 2nd image)



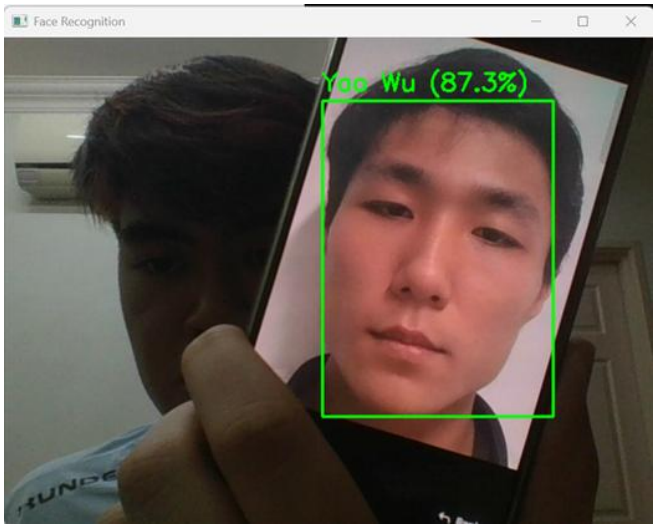
10) Image from dataset (Subject: Han Yong, 3rd image)



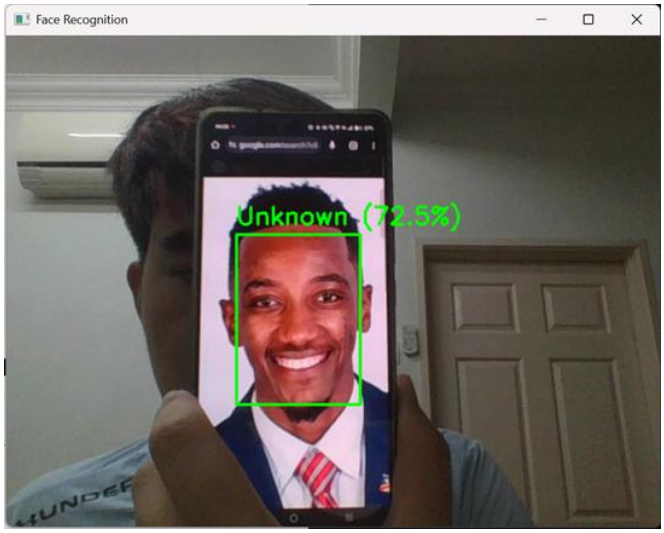
11) Image from dataset (Subject: Yao Wei)



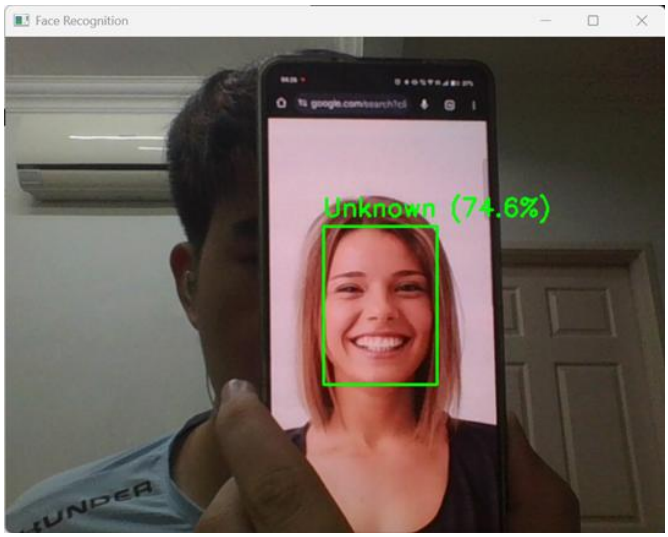
12) Image from dataset (Subject: Yao Wu)



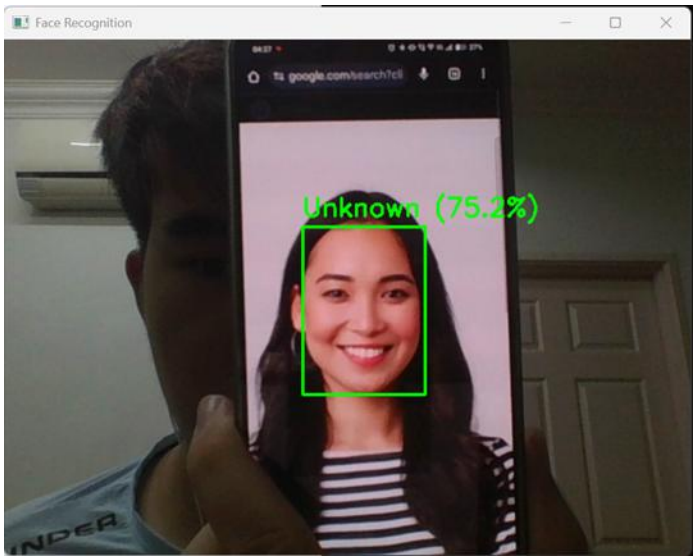
13) Image not in dataset



14) Image not in dataset



15) Image not in dataset



Total Accuracy = 15/15 = 100%

After we modified our original code and added features like Caffe model to do face detection, Haar Cascade model for eye alignment and detection, and artificial augmentation (mirror face, tilt face image), we now have a higher number of samples to aid the PCA reduction and KNN classifier training. All these features greatly improve the accuracy of face recognition.

5 Results

As we can see in the observations, the accuracy of our recognition system after we have done some modifications (accuracy of 100%) is significantly higher than our original recognition system (accuracy of 0%). We have tried to only implement one of these modifications to our original system. However, while the accuracy has been increased, it does not reach as same par as our recognition system. In other words, our recognition system needs every single modification to maintain great accuracy in detecting faces.

One of the improvements is how we preprocess our dataset. In the beginning, we have the wrong perception that rotating the dataset may decrease our accuracy, due to the fact that there are always pitch black areas in them. However, it seems like the eigendecomposition method seems to do a great job of extracting eigenvectors from these images.

The act of aligning and augmenting images created different variations of the exact same image. They can be flipped or rotated to a different degree. Many noises are also hindered from being noticed by the classifier while preprocessing the data. Thus, the KNN classifier can be trained with a higher number of faces with numerous variations. KNN is able to recognise data pre-processed by our 2nd recognition system better.

Furthermore, there is a slight change in how we obtain the eigenvectors and eigenvalues. In the original code, we used as many manual calculations as we could to prove that we understand how eigendecomposition and SVD methods work. We cannot deny the importance of our original code in this report. However, we have to acknowledge that it is not as good as our improved recognition system. Changing these calculations to pre-defined functions is simply better as each function is basically an entire Python file. These widely used functions have many parameters for us to choose from and could have a few best possible scenarios to decompose different types of data. They also include their error calling conditions and could tell us if anything goes wrong clearly.

6 Conclusion

In conclusion, the accuracy of our current improved prototype is quite high. It is able to correctly predict the faces in 15 out of 15 test cases. Even though our prototype occasionally predicts our moving face incorrectly in one frame, it always follows up with correct predictions in the next frames. It is clear that this face recognition system is ready to be sold as a commercial product, though some tweaks are definitely needed to improve the product.

Although the accuracy of our current prototype is quite great, different environments that have different levels of light brightness, the brightness of the environment will still heavily affect our system's accuracy. Not only that, limited data on face images, also causes the training for PCA and KNN classiier to make mistakes in face recognition. To solve the situation we face, we will need to collect more face data and purchase a better webcam on our computer. We also realize that the time taken for eigendecomposition and running our recognition system is too long. We should prepare better computer resources to train our model faster in our next experiment.

Contributions	
Group Member	*Contributions*
Yee Qing Wei	- Create codes to test the accuracy of recognition system with different PCA methods and classifiers - In charge of most of the reports
Chin Lian Ann	- Created presentation slides for this project
Wong Yue Wei	- Added modifications into the original prototype to improve its accuracy - Writing sections in report about these modifications and their results
Yap Han Yong	- Created the real-time face detection UI that can use a webcam to take frames
Leong Yao Wu	- Found and showed the result for some algorithms together with the face detection UI - Checked the code validity
Leong Yao Wei	- Found suitable algorithm with the UI - Tested the code if it can be executed

References

GeeksforGeeks. (2025a, February 3). *Principal Component Analysis(PCA)*. GeeksforGeeks. https://www.geeksforgeeks.org/principal-component-analysis-pca/

Pramoditha, R. (2023, February 22). Eigendecomposition of a Covariance Matrix with NumPy. *Medium*. https://medium.com/data-science-365/eigendecomposition-of-a-covariance-matrix-with-numpy-c953334c965d

Pramoditha, R. (2025, January 22). *Singular value decomposition vs eigendecomposition for dimensionality reduction*. Towards Data Science. https://towardsdatascience.com/singular-value-decomposition-vs-eigendecomposition-for-dimensionality-reduction-fc0d9ac24a8e/

GeeksforGeeks. (2025b, March 1). *Euclidean distance*. GeeksforGeeks. https://www.geeksforgeeks.org/euclidean-distance/

GeeksforGeeks. (2025c, March 1). *Cosine similarity*. GeeksforGeeks. https://www.geeksforgeeks.org/cosine-similarity/

APPENDIX 1

MARKING RUBRICS

Component Title	PCA in face recognition				**Percentage (%)**		
Criteria	**Score and Descriptors**					**Weight (%)**	**Marks**
	Excellent (5)	**Good (4)**	**Average (3)**	**Need Improvement (2)**	**Poor (1)**		
Introduction	Excellent introduction with all the necessary information, well-defined objective and clear description of topic under study.	Good introduction with most of the necessary information, defined objective and clear description of the topic under study.	Moderate introduction with some information on the topic	Weak introduction with limited information about the topic.	Irrelevant introduction.	**25**	**25**
Content	Very rich content which covers more than expected.	Rich content which covers almost all aspects of the topic.	Content sufficiently illustrates the topic.	Content is weakly relevant to the topic.	Content is not relevant to the topic.	**25**	**25**
Difficulty Level	The level of difficulty far exceeds what is expected from this course.	The level of difficulty slightly exceeds what is expected from this course.	The level of difficulty is comparable to what is expected from this course.	The level of difficulty is below what is expected from this course.	The level of difficulty is far below what is expected from this course.	**25**	**25**
Discussion and Conclusion	Interesting discussions are presented, which are logically correct and involve a lot of new ideas. Conclusions give rise to new theories or findings.	Discussions are logically presented, which involve some new ideas. Conclusions are correct.	Sufficient logically correct discussions are presented, with correct conclusions.	Some discussions are presented, some are not logically correct. Some conclusions are presented.	Very few discussions and no conclusion; or all discussions are irrelevant or logically incorrect.	**25**	**25**
					TOTAL	**100**	**100**

Note to students: Please print out and attach this appendix together with the submission of coursework