DSP Project

FACIAL RECOGNITION

TABLE OF CONTENTS

PROJECT ABSTRACT

Facial recognition is the process of identifying individuals in images or videos by comparing the appearance of faces in captured imagery to a database. It is also described as a Biometric Artificial Intelligence based application that can uniquely identify a person by analyzing patterns based on the person's facial textures and shape.

Face recognition has many applications ranging from security and surveillance to biometric identification to access secure devices. The goal is to implement a system or a model that can successfully detect and match faces from the database and perhaps venture into real-time face recognition. After extensive research and reading, there are several approaches that we have come across and we will attempt to incorporate the best ones into our project.

Facial Recognition/Detection can be achieved considering the following approaches:

- A method of approach for face recognition is using Haar Cascades, where a cascade function is trained with a set of input data. OpenCV is the most important tool that will be used during this approach. It is one of the prominent features of Viola-Jones object detection framework.

- PCA is a statistical approach used for reducing the number of variables in face recognition. In PCA, every image in the training set is represented as a linear combination of weighted eigenvectors called eigenfaces.

- Eigenfaces have a parallel to one of the most fundamental ideas in mathematics and signal processing – The Fourier Series. The aim is to represent a face as a linear combination of a set of basis images (in the Fourier Series the bases were simply sines and cosines).

METHODOLOGY

// requires modification

The algorithm used to identify objects in an image or video and based on the concept of features proposed by Paul Viola and Michael Jones . It is an approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images.

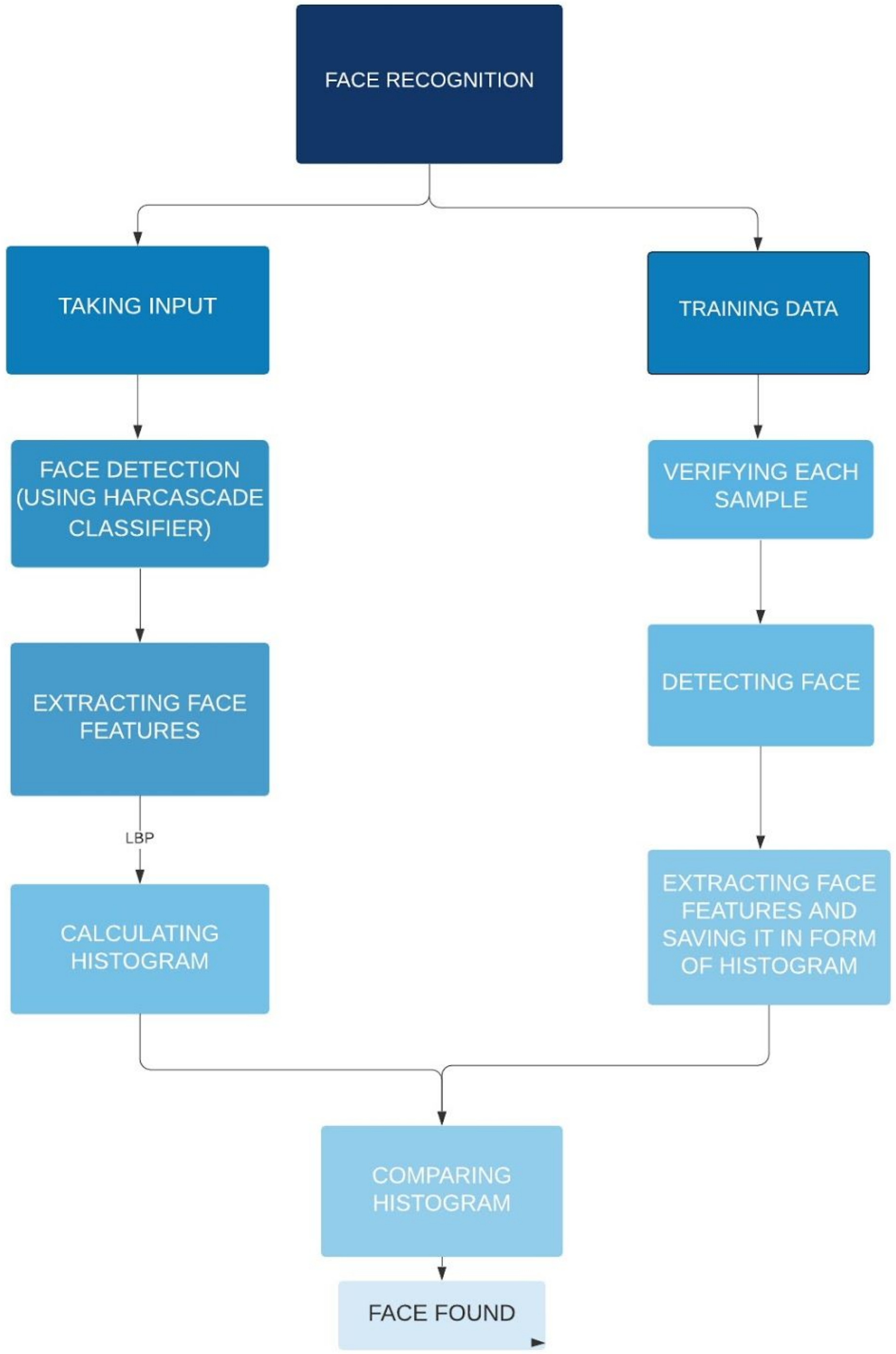
Haar-feature is a sequence of rescaled square-shaped functions very similar to Fourier Analysis, which was proposed by **Alfred Haar** in **1909**, which are like convolution kernels.

This method works on edge detection using pixel intensities, for a black and white image (where the black represents pixel intensity as 1 and white to be 0), an ideal Haar-Feature the difference of average pixel intensities will be 1. But for a real scenario where a gray scale image has integer pixel intensities ranging from 0 to 1 because there is no pixel which is completely balck or completely white therefore the difference will not be one.

The closer the value (difference between pixel intensities of black and white/ dark or bright) to one, the more likely we found a haar feature.

Haar feature face detection works on detecting the face features classified in the face model used for extracting the face from the input image.

FLOWCHART:


FACE RECOGNITION
TAKING INPUT
TRAINING DATA
FACE DETECTION (USING HARCASCADE CLASSIFIER)
VERIFYING EACH SAMPLE
EXTRACTING FACE FEATURES
DETECTING FACE
LBP
CALCULATING HISTOGRAM
EXTRACTING FACE FEATURES AND SAVING IT IN FORM OF HISTOGRAM
COMPARING HISTOGRAM
FACE FOUND

The Process for **Face Identification** comprises of the following steps:

- **Face Detection:** To differentiate and point out the face part if available in the image.
- **Data Collection:** To extract unique characteristics of the individuals face that it can use to differentiate him from another person, like eyes, mouth, nose, etc
- **Data Comparison:** Irrespective of the some conditions (light or expressions), it will compare those unique features to all the features of identities in the database.
- **Face Recognition:** If matched , it will identify the individual.

First Step: **Face Detection**

The process has main 4 stages, namely -

1. **Conversion** : Loading the image and converting it into gray-scale. Generally the images that we see are in the form of RGB channels(Red, Green, Blue). We need to convert this BGR channel to gray channel. The reason for this is that the gray channel is easy to process and is computationally less intensive as it contains only 1-channel of black-white.
2. **Feature Location** : using the face_classifier which is an object loaded with an appropriate face_model.xml, we are using an inbuilt function with it called the detectMultiScale. This function will help us to find the features/locations of the new image. The way it does is, it will use all the features from the face_classifier object to detect the features of the new image.

3. **Rectangulation** : From the above step, the function detectMultiScale returns 4 values — x-coordinate, y-coordinate, width(w) and height(h) of the detected feature of the face. Based on these 4 values we will draw a rectangle around the face.

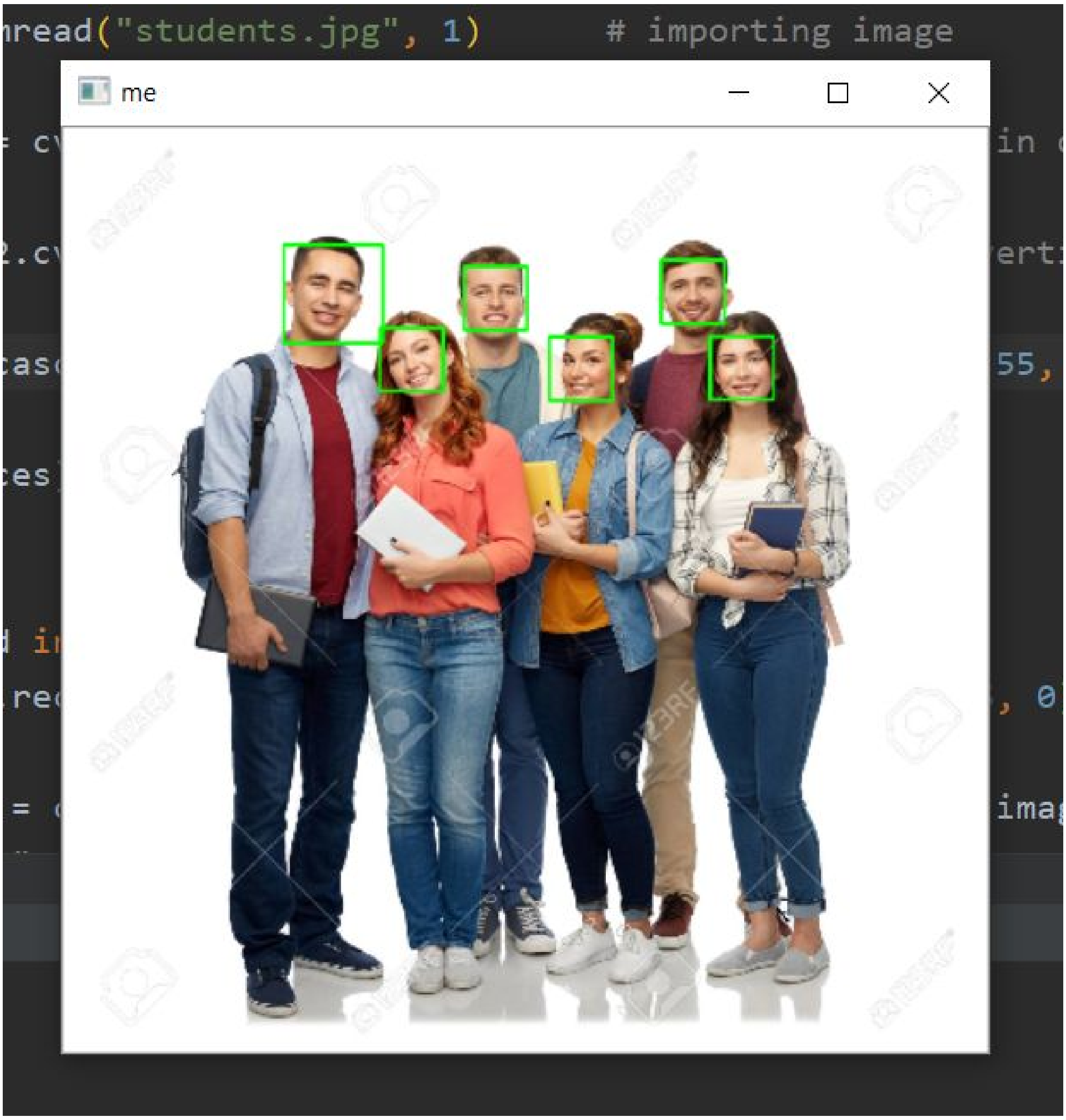
4. **Re-Sizing**: (optional) If the dimensions of the image that we use here is pretty large, we scale down the image dimensions for obtaining a better output.

Haar feature face detection works on detecting the face features classified in the face model used for extracting the face from the input image.

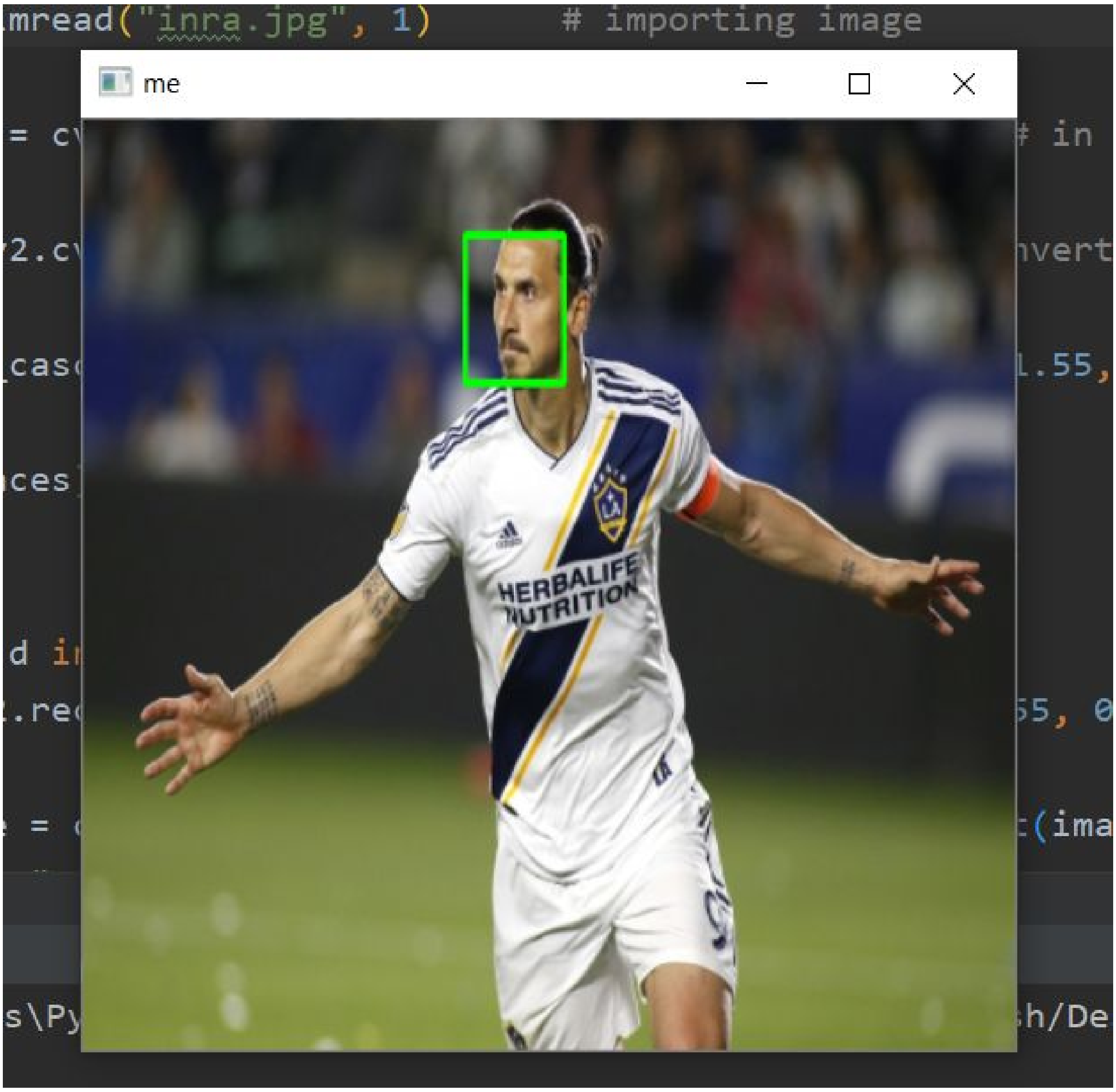
CODE

For Face Detection

```
import cv2 # OpenCv

image = cv2.imread("image.jpg", 1)       # importing image
face_cascade = cv2.CascadeClassifier("face_model.xml")
# in order to use the face cascade class# converting to
gray image
img_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# identifying faces
faces = face_cascade.detectMultiScale(img_gray, scaleFactor
= 1.05, minNeighbors = 5)

print(type(faces))
print(faces)
# adding rectangle to the face
for a, b, c, d in faces:
  img = cv2.rectangle(image, (a, b), (a + c, b + d), (0,
255, 0), 3)

resized_image = cv2.resize(img, (int(image.shape[1] / 3),
int(image.shape[1] / 3)))
# resizing the image for better visual output

cv2.imshow("me", resized_image)

cv2.waitKey(0)
cv2.destroyAllWindows()

# print(image.shape)
```



```
mread("students.jpg", 1)        # importing image
```



```
mread("inra.jpg", 1)      # importing image
= cv                                    # in
2.cv                                    nvert
casc                                    1.55,
ces
d in
.rec                                    55, 0
= c                                     t(ima
s\Py                                    h/De
```

Commands Used:

- imread
  - to load and display an image

- cascadeClassifier
  - saving face cascade(face data) into a variable(face_cascade) for further usage

- cvtColor
  - applies an adaptive threshold to an array.

- detectMultiScale
  - used for object (here face) detection
  - scaleFactor: the greater the scale factor greater is the precision reference, reference.

- minNeighbour
  - parameter specifying how many neighbors each candidate rectangle should have to retain it reference

- rectangle
  - for adding rectangle shape into faces detected using the end coordinates reference

- resize
  - resizing image reference

- imshow
  - for image display output

- COLOR_BGR2GRAY
    - converting an image to grayscale(balck and white image) which is required for haarCascade detection BGR image to Gray image. (color space conversions) Reference here and here

- waitKey(0)
    - waits for a key to be pressed reference

- destoryAllWindows
    - destroys all of the output HighGUI windows. reference

Noteworthy Observations / Learnings:

- ➔ Precision of the Face Detection can be increased for different images using corresponding haarcascade models.

- ➔ We perform face detection on the grayscale image but the rectangle is displayed on the original image using the coordinates.

- ➔ Every Image can be converted as an array of matrices using numpy/OpenCv (dimension of the matrix depending on the type of the image, RGB - 3d, Gray - 2d) which are very crucial for image processing.
- ➔ The inconsistency of the implementation increases with detailed images and cannot be overlooked, should be removed accordingly.

Second Step: **Data Collection**

Understanding the process of face recognition:

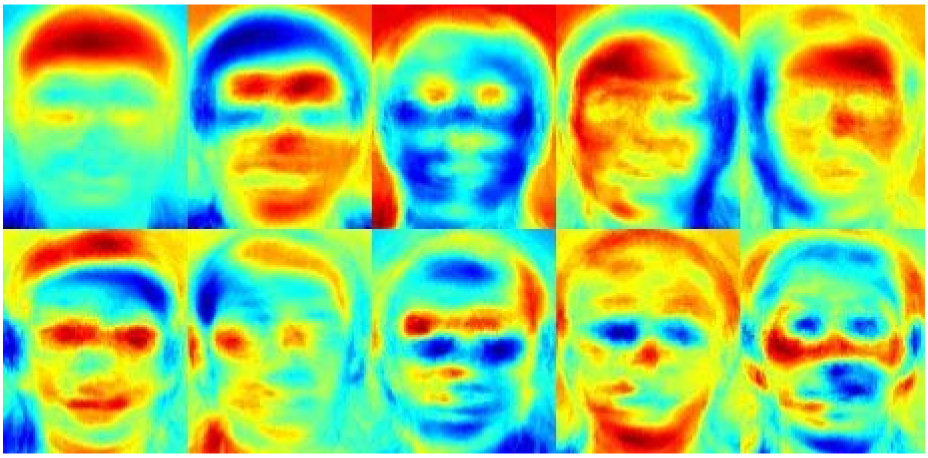
OpenCv has three built-in face recognizer methods which are extremely easy to implement.

1. **Eigenfaces face recognizer**
   This algorithm considers the fact that not all parts of a face are equally important or useful for face recognition.
   It recognizes a face by the distinct features like the eyes, nose, cheeks or forehead; and how they vary with respect to each other.

   In that sense we are focusing on the areas of maximum changes. For example from the eyes to the nose there is a significant change, and the same applies from the nose to the mouth. When we look at multiple faces, we compare them by looking at these areas, because by catching the maximum variation among faces, they help us differentiate one face from another.

   The EigenFace method looks at all the training images of all the people as a whole and tries to extract the components which are relevant and useful and discards the rest. These important features are called **principal components**(eigenfaces)
   Or equivalently,
   the eigenvectors of the covariance matrix of the set of face images, where an image with N pixels is considered a point (or vector) in N-dimensional space.

Below is an image showing the variance extracted from a list of faces.



So, EigenFaces recognizer trains itself by extracting principal components, but it also keeps a record of which ones belong to which person. Thus, whenever you introduce a new image to the algorithm, it repeats the same process as follows:

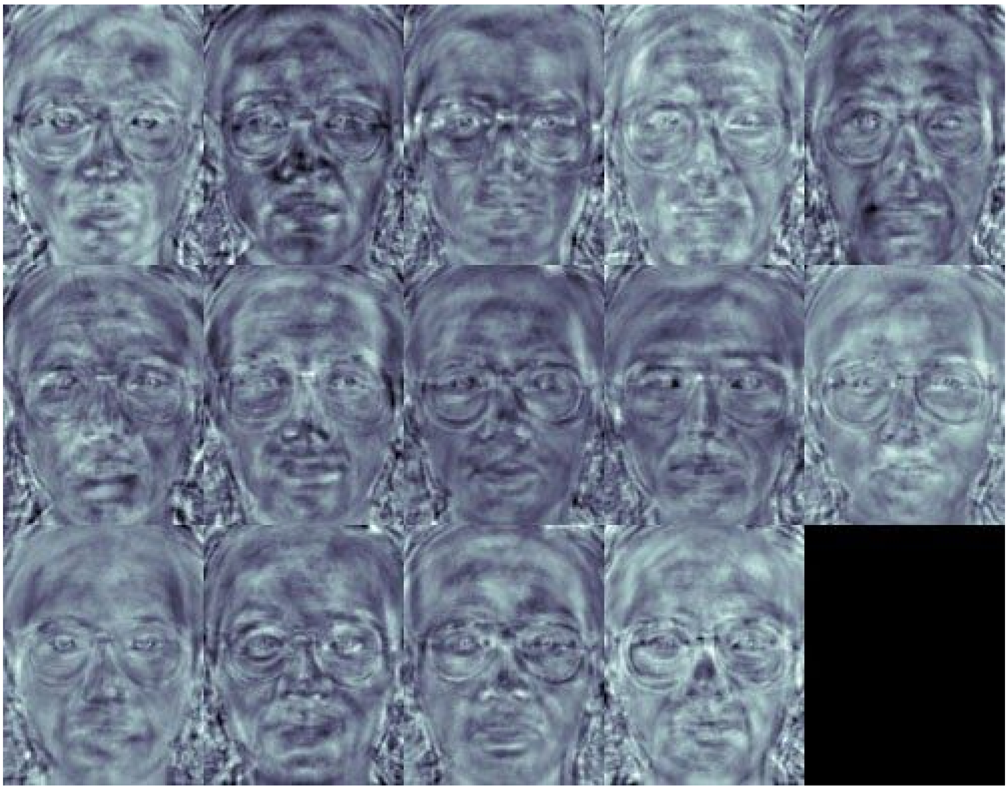
1. Extract the principal components from the new picture.
2. Compare those features with the list of elements stored during training.
3. Find the ones with the best match.
4. Return the 'person' label associated with that best match component

However a point worth noting in the EigenFaces algorithm is that this algorithm also considers illumination as an important feature. As a result, lights and shadows are picked up by EigenFaces, which unnecessarily classifies them as representing a "face".

2. **FisherFaces face recognizer**
   This algorithm is an improved version of the last one. As we just saw, EigenFaces looks at all the training faces of all the faces at once and finds principal components from all of them combined. By doing that, it doesn't focus on the features that discriminate one individual from another. Instead, it concentrates on the ones that represent all the faces of all the people in the training data, as a whole.

   Also, we know that eigenfaces considers illumination an important feature of a face but it actually isn't.

Below is an image of principal components using FisherFaces algorithm.



By using **Fisherfaces** we can prevent features of an individual from being dominant but it still considers **illumination an important feature**. But we know that illumination is not an important feature as it's not even a part of the face.

Precisely, FisherFaces face recognizer algorithm extracts principal components that differentiate one person from the others. In that sense, an individual's components do not dominate (become more useful) over the others.

In other words, the difference of Eigen and FisherFaces can be explained as follows:

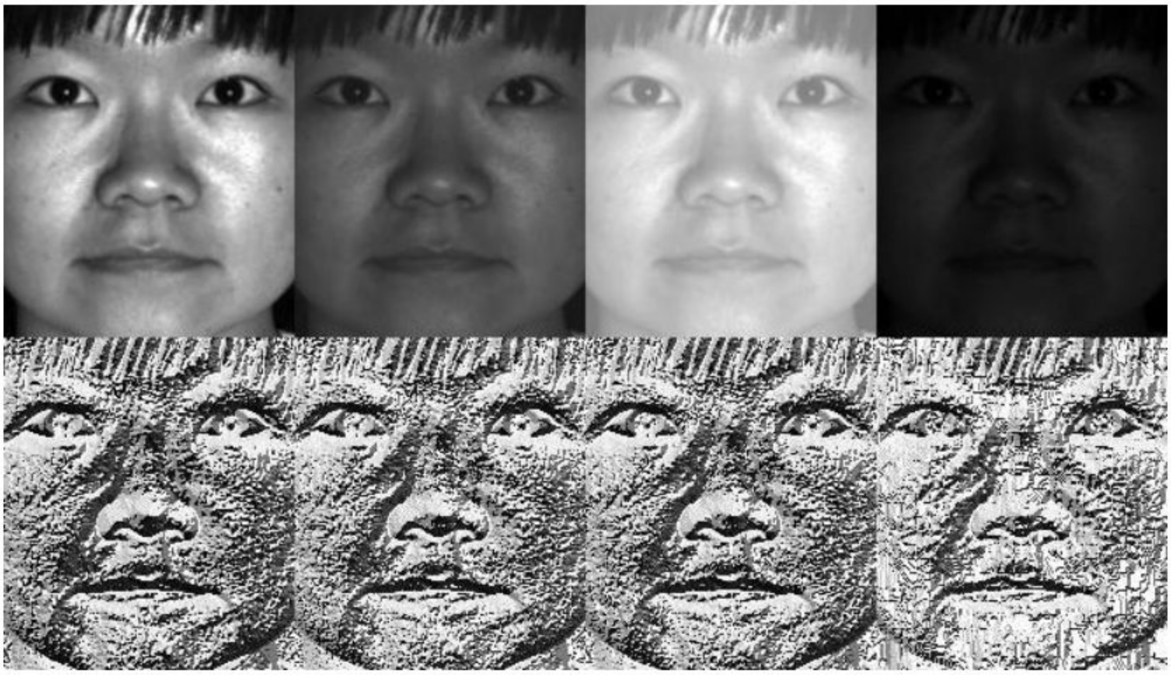
Eigenface tries to maximize variation. So, with pca they usually get a decent model of the face. Fisherface wants to maximize the mean distance of different classes while minimizing the variance within class. They get face models that are more useful in discrimination.

3. **Local Binary patterns histograms (LBPH) face recognizer**

We know that Eigenfaces and Fisherfaces are both affected by light and, in real life, we can't guarantee perfect light conditions. LBPH face recognizer is an improvement to overcome this drawback.

The idea with LBPH is not to look at the image as a whole, but instead, try to find its local structure by comparing each pixel to the neighboring pixels.

Image below shows the results after using LBPH on a face with different light conditions (illumination).



The Local Binary patterns Histogram algorithm uses a concept of a sliding window, based on the parameters.

- Parameters of LBPH are :

  1. **Radius:** the radius is used to build the circular local binary pattern and represents the radius around the central pixel. It is usually set to 1.

  2. **Neighbours:** the numbers of sample points to build the circular local binary pattern.

  3. GridX: the number of cells in the horizontal direction. The more cells, the finer the grid, the higher the dimensionality of the resulting feature vector. It Usually set to 8.

4. GridY: the number of cells in the vertical direction. The more cells, the finer the grid, the higher the dimensionality of the resulting feature vector. It is usually set to 8.

- Training the Algorithm:

  First, we need to train the algorithm. To do so, we need to use a dataset with the facial images of the people we want to recognize.

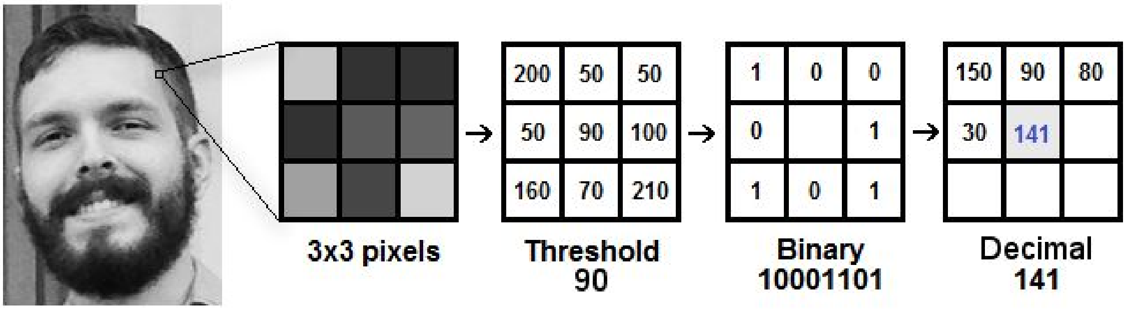
  We need to set an unique Id (it may be a number or name) for each image, so the algorithm will use this information to recognize an input image and give us an output.

  // add training images steps here

- Applying the LBP operation: (Local Binary Pattern)

  The first computational step of the process is to create an intermediate image that describes the original image in a better way, by highlighting the facial characteristics. To do so, the algorithm uses a concept of a sliding window, based on the parameters radius and neighbors.

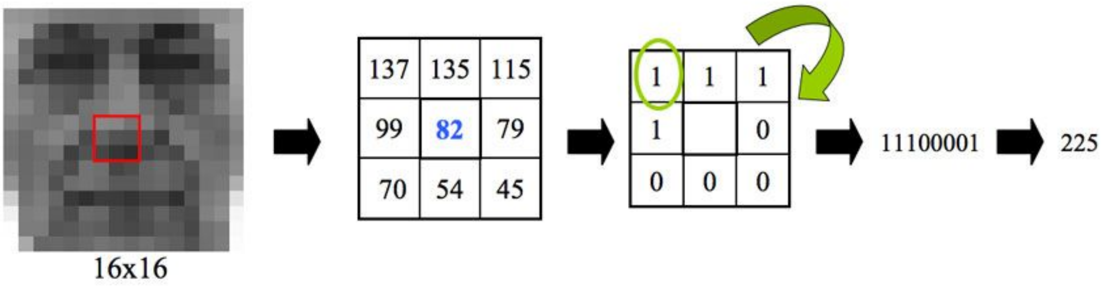
The image below shows the process in applying the LBPH algorithm.



- We have a grayscale taken from the input. (converted) // edit this
- We can get part of this as a **window of 3x3 pixels**. LBP looks at these 9 pixels (3×3 window) at a time, and with a particular interest in the pixel located in the center of the window.
- It can also be represented as a 3x3 matrix containing the **intensity of each pixel** (0~255).
- Then, we need to take the central value of the matrix to be used as the threshold.
- This value will be used to define the new values from the 8 neighbors.
- For each neighbor of the central value (threshold), we set a new binary value. We **set 1 for values equal or higher than the threshold and 0 for values lower than the threshold.**
- Therefore, the matrix will contain only binary values (ignoring the central value). We need to concatenate each binary value from each position from the matrix line by line into a new binary value (e.g. 10001101).

  Note: Some authors use other approaches to concatenate the binary values (e.g. clockwise direction), but the final result will be the same.

- Then we need to convert this binary value to a decimal value and set it to the central value of the matrix, which is actually a pixel from the original image.
- As a result of this procedure, we obtain a new image which represents better the characteristics of the original image.

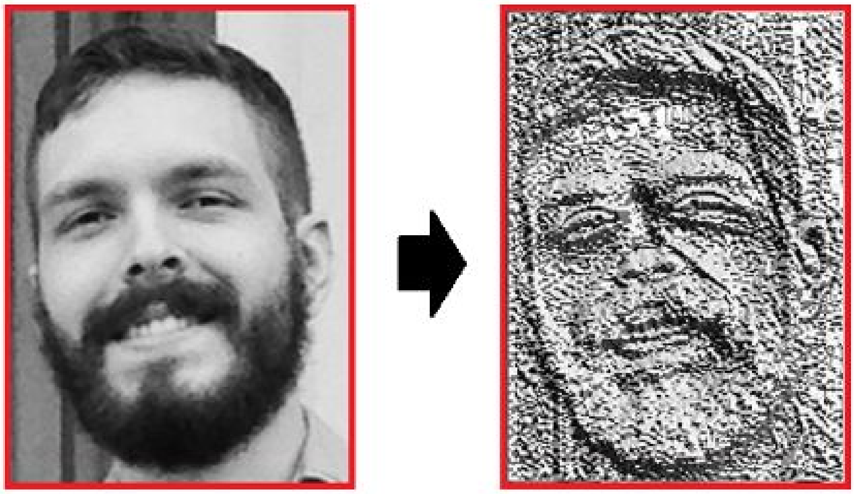


137	135	115
99	82	79
70	54	45

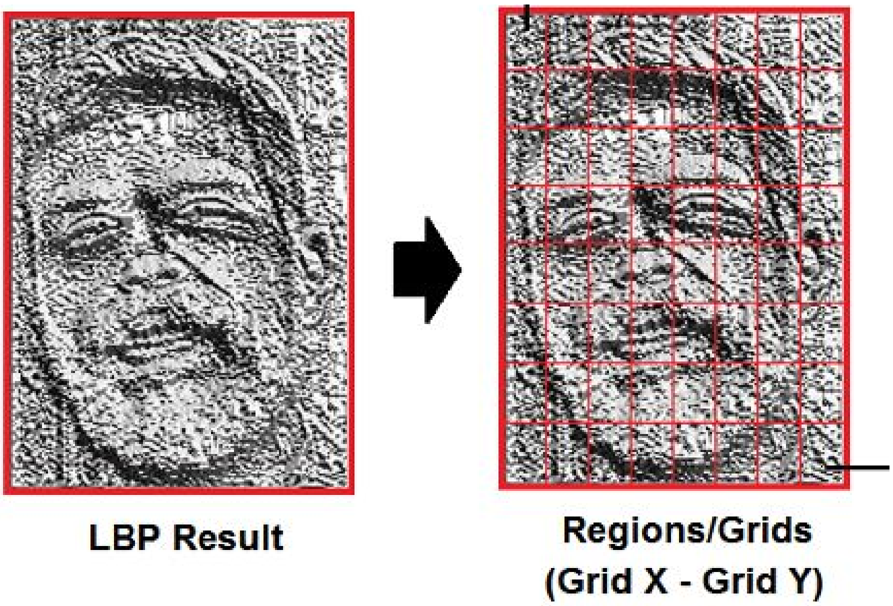
1	1	1
1		0
0	0	0

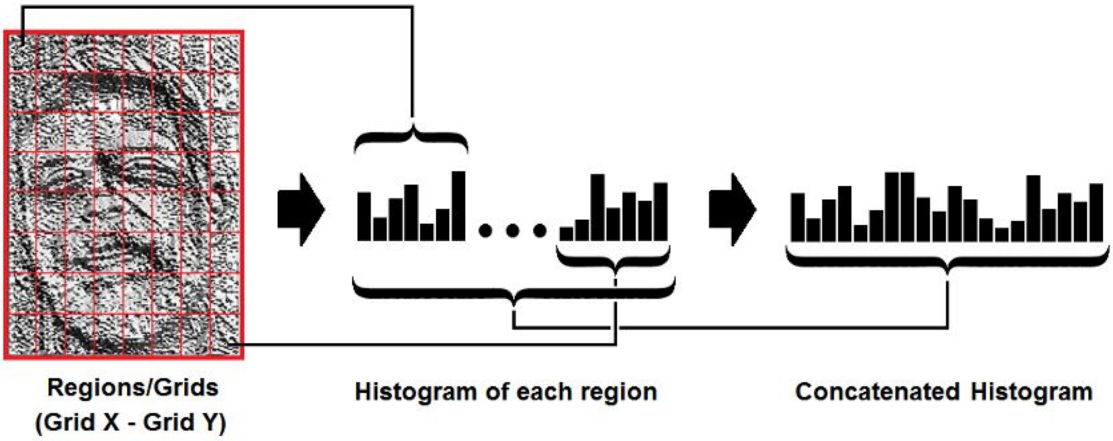
- Extracting the Histograms:

  Using the image generated in the last step, we can use the **Grid X** and **Grid Y** parameters to divide the image into multiple grids, as can be seen in the image below:



Original Image **LBP Result**





As we have an image in a grayscale each histogram (from each grid) will contain only 256 positions (0~255) representing the occurrences of each pixel intensity.

Then, we need to concatenate each histogram to create a new and bigger histogram. Suppose we have 8x8 grids, we will have 8x8x256=16,384 positions in the final histogram. The final histogram represents the characteristics of the original image.

We need to apply this process for each face image we have in our **training data set**. As a result, we will have one histogram for each face in the training data set, which will be stored for later recognition. The algorithm also keeps track of which histogram belongs to which person.

- Performing face recognition

  The process of recognition basically consists of the following steps:

  1. Feed a new image/video to the recognizer for face recognition.
  2. The recognizer creates a histogram using the algorithm for that new picture.
  3. It then compares that histogram with the histograms it already has in the trained data set.
  4. Finally, it finds the best match and returns the face id associated with that best match.

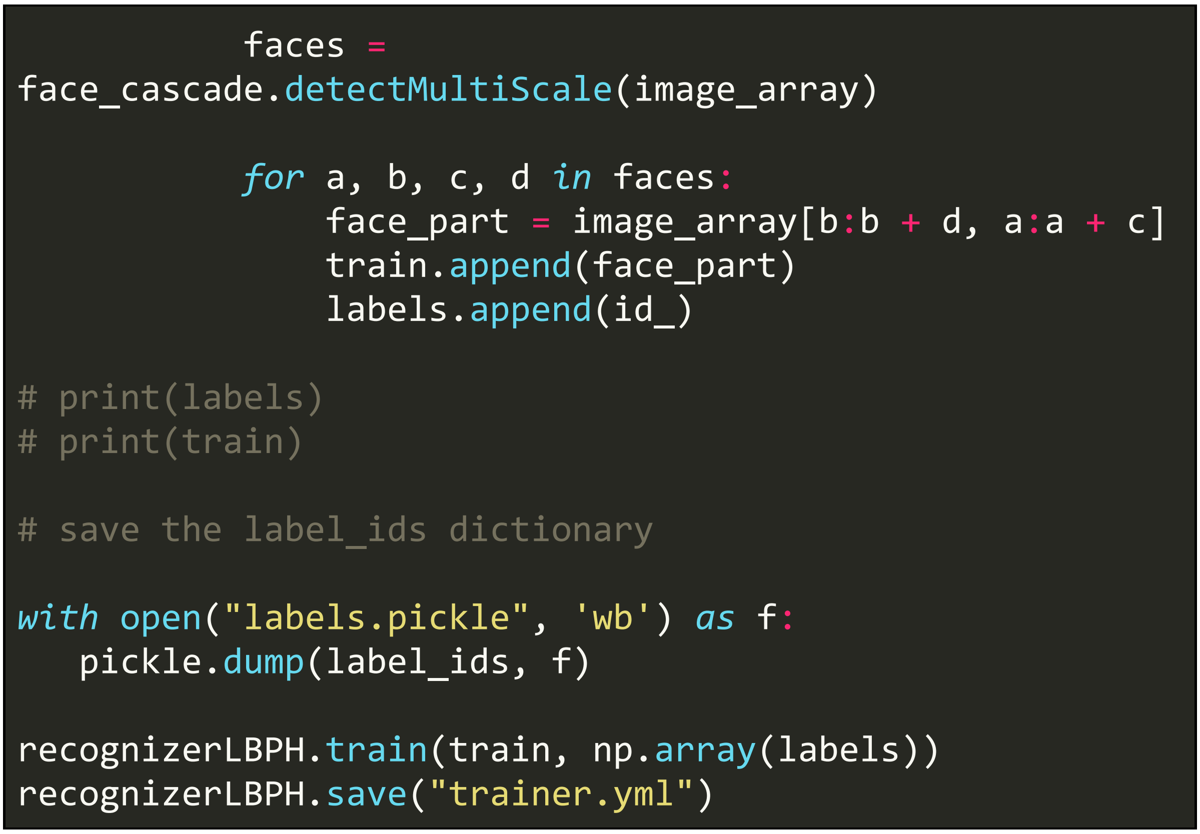
CODE For Recognition

The entire project code can be found here.
The python code for facial recognition includes the use of the following libraries:

• OpenCv	*pypi.org/project/opencv-python/*
• OS	*docs.python.org/3/library/os.html*
• Numpy	*numpy.org/*
• PIL	*pypi.org/project/Pillow/*
• Pickle	*docs.python.org/2/library/pickle.html*

First code for Face Data training faces-train.py

```
import os
import numpy as np
from PIL import Image
import cv2
import pickle

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
image_DIR = os.path.join(BASE_DIR, "training_data")

face_cascade = cv2.CascadeClassifier("face_model.xml")
# in order to use the face cascade class
recognizerLBPH = cv2.face.LBPHFaceRecognizer_create()

temp_id = 0
label_ids = {}
labels = []
train = []

for root, dirs, files, in os.walk(image_DIR):
   for file in files:
       if file.endswith("png") or file.endswith("jpg"):
           path = os.path.join(root, file)
           label = os.path.basename(root).replace(" ",
"-").lower()
           # print(path)

           if label in label_ids:
               pass
           else:
               label_ids[label] = temp_id
               temp_id += 1

           id_ = label_ids[label]
           # print(label_ids)

           # labels.append(label)
           # train.append(path)
           pil_image = Image.open(path).convert("L")
           image_array = np.array(pil_image, "uint8")
```



```
           faces =
face_cascade.detectMultiScale(image_array)

           for a, b, c, d in faces:
               face_part = image_array[b:b + d, a:a + c]
               train.append(face_part)
               labels.append(id_)

# print(labels)
# print(train)

# save the label_ids dictionary

with open("labels.pickle", 'wb') as f:
   pickle.dump(label_ids, f)

recognizerLBPH.train(train, np.array(labels))
recognizerLBPH.save("trainer.yml")
```

An explanation for the above code:

- ➔ Importing the necessary libraries for the code which include **os** (for reading the images from their folders/files) NumPy for working with numbers and arrays, **Pillow** an image library, **OpenCv**, **Pickle** for serialization.
- ➔ **Note**: We've set the folder name as the name of the person of which the folder consists of the photo.

training_data (main folder)

↳ **person1** (sub-folder)

↳ image1

↳ image2

↳ image3 .....

↳ **person2**

↳ **person3** .....

- Getting the base directory of the project in order to get further folders where the images are stored.

- We only work with the faces of the entire images as the region of interest for facial recognition is only the facial part. Stored into the list `train`.

- Getting the Image folder directory (named: training_data) of the project in order to get the further folders where the images are stored.

- Initializing the face cascade model which will be required for training the data. (here: `face_model.xml`)

- Initializing the LBP face recognition method for performing each photo of training, in order to create the histograms of the trained data.(`recognizerLBPH`)

- We will add the identification id to the dictionary created (here: `label_ids`) and perform the training for all the images provided using the for loop using the directory locations obtained using the **os**.

- Save the above stored of label (names & id) into the dictionary as a serialized file (named: labels) using the Pickle library

  ```
  with open("labels.pickle", 'wb') as f:
      pickle.dump(label_ids, f)
  ```

- Further, training using the function `recognizerLBPH.train()` all the facial parts using the train list and the labels obtained from the above process and saving them into the yml file named `trainer.yml` using function `recognizerLBPH.save()`

- ➔ At the end of the execution of this python program, we will have two files one **labels.pickle** for the list of identification ids and second the **trained yml** file for comparing the Histograms.

Second code for performing the face recognition to the faces we got from input (video/image) test.py

```
# OpenCV module
import cv2

import pickle

# list of names for trained images
recognizer = cv2.face.LBPHFaceRecognizer_create()
recognizer.read("trainer.yml")

labels = {"personsName": 1}
with open("labels.pickle", 'rb') as f:
   labels = pickle.load(f)
   labelsNew = {v: k for k, v in labels.items()}

cap = cv2.VideoCapture(0)

# face detection
# harCascade method for face detection
while True:
   # image = cv2.imread("inra.jpg", 1)  # importing image
   ret, frame = cap.read()
   face_cascade = cv2.CascadeClassifier(
       "haarcascades/haarcascade_frontalface_default.xml")
# in order to use the face cascade class

   gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)  #
converting to gray image

   facesFound = face_cascade.detectMultiScale(gray,
scaleFactor=1.5, minNeighbors=5)  # identifying faces
```

```
   print(type(facesFound))
   print(facesFound)

   for a, b, c, d in facesFound:

       face_part_gray = gray[b:b + d, a:a + c]
       face_part_color = frame[b:b + d, a:a + c]

       output_image = "my-image.png"
       cv2.imwrite(output_image, face_part_gray)

       # comparing the histograms
       id_, confidence =
recognizer.predict(face_part_gray)

       if 45 <= confidence >= 75:
           print(id_)
           print(labelsNew[id_])
           font = cv2.FONT_HERSHEY_COMPLEX
           name = labelsNew[id_]
           color = (255, 255, 255)
           cv2.putText(frame, name, (a, b), font, 1,
color, 2, cv2.LINE_AA)

       # width = a + c
       # height = b + d
       # (0,255,0) is green
       # 3 represents the thickness of the line
       # adding rectangle to the face
       img = cv2.rectangle(frame, (a, b), (a + c, b + d),
(0, 255, 0), 2)

   resized_image = cv2.resize(frame, (int(frame.shape[1] /
2), int(frame.shape[1] / 2)))
   cv2.imshow("me", frame)
   if cv2.waitKey(20) & 0xFF == ord('q'):
       break

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

An explanation for the above code:

- The above code is used for detecting faces from a live video using a camera.

- Importing the necessary libraries for the code which include **OpenCv**, **Pickle** for serialization.

- Initializing the LBP face recognition method for performing in order to create the histograms of the input data.(`recognizer`) and using the required functions of the same.

- Reading the trainer.yml that we generated from the training program and similarly getting the list of labels from the pickle file that we generated using the training program and saving them into the variables.

- Using OpenCv we get the video/image as an input.

- The next step includes detecting the facial part using the HarCascade face detection method from the image and storing it.

- We obtain a number (confidence) of the method of prediction using `recognizer.predict(face_part_gray)` this method compares all the histograms to the histogram of the input face obtained from the image/video

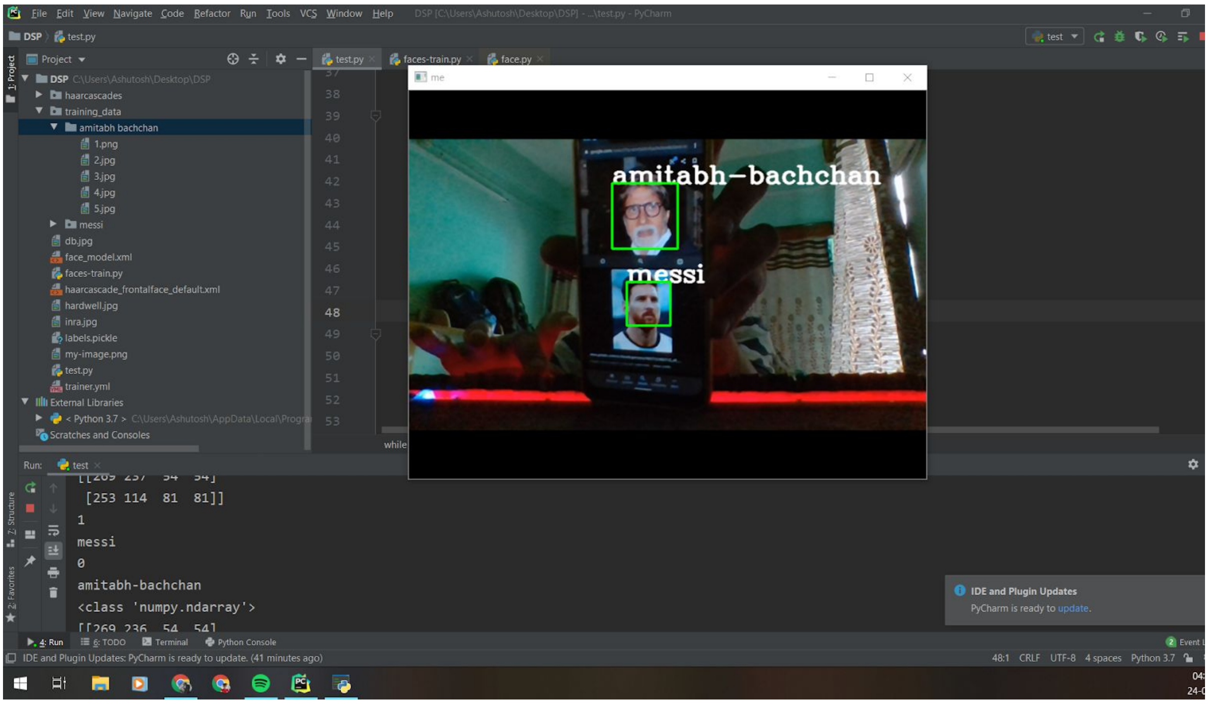
- Using the confidence, `if 45 <= confidence >= 75:` we print the id name along with the rectangle over the facial part to the face detected.

- Image can be resized to get a better visual output using the resize function from OpenCv `cv2.resize()`

- This method continues till we press the ‘q’ which is obtained by using the while loop. `cap.release()` `cv2.destroyAllWindows()` to destroy all windows.

➔ At the end of execution of this python program, we will have a window opened with video/image showing the output provided the frame has any recognized face and the label (name) will appear along with the rectangle at the face.

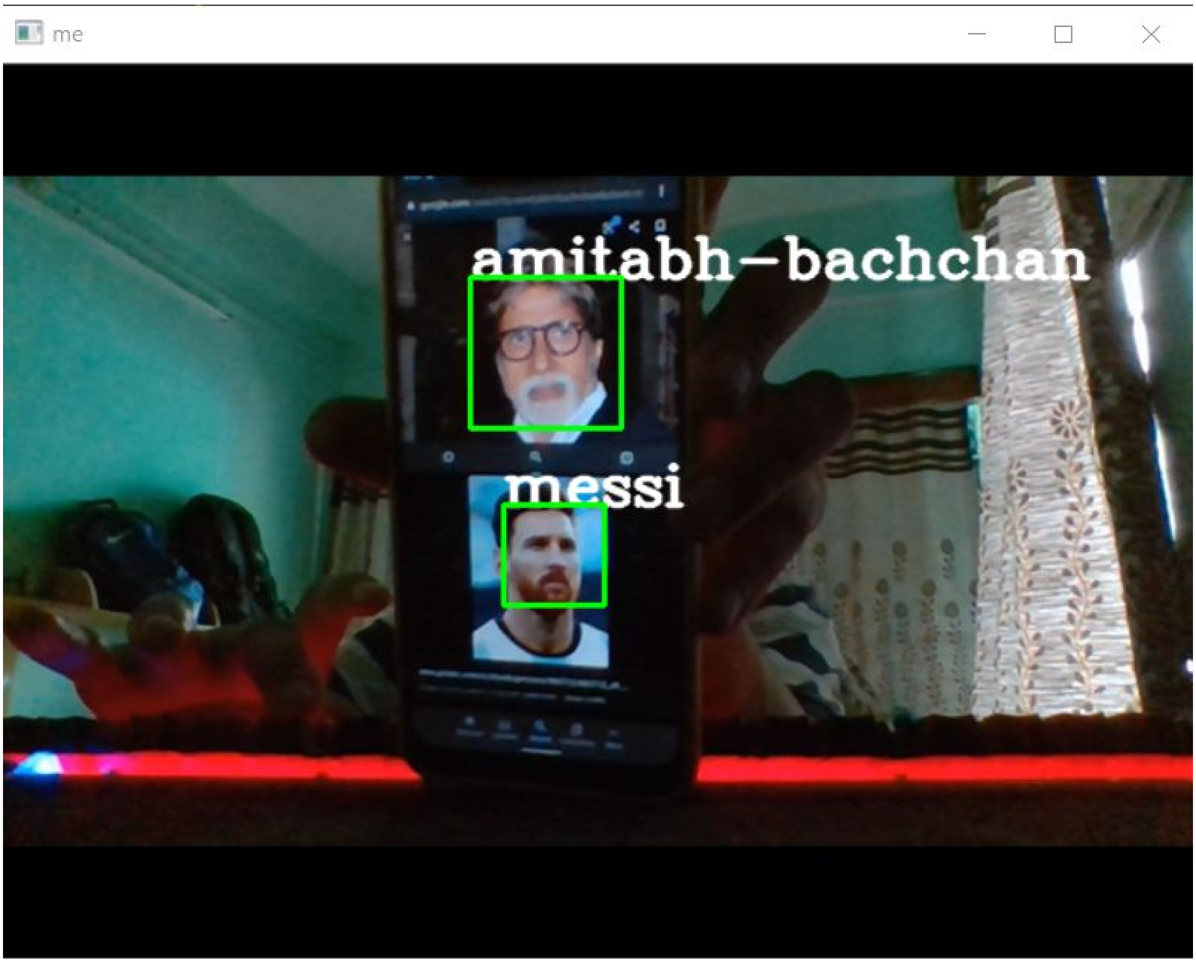
Output:



```
[[269 237  54  54]
 [253 114  81  81]]
1
messi
0
amitabh-bachchan
<class 'numpy.ndarray'>
[[269 236  54  54]
```

IDE and Plugin Updates
PyCharm is ready to update.

IDE and Plugin Updates: PyCharm is ready to update. (41 minutes ago)



(ScreenShot of the recognized faces from a video, using the trained data of faces)

FUTURE WORK

The goal was to implement a system or a model that can detect and match faces from the database. After having done that successfully, we would like to discuss the improvements / extensive features that we can attempt to add to this project, the scope of applications where we can implement a project at this level and the applications of such a project on large scale :

- **Cross Platform Functionalities**: Developing cross platform functionalities by producing mobile apps that work with the face recognition principles of this project

- **Development of a GUI:** Current face recognition process is all manual and one does require some technical skills to set this face recognition method. In future a user friendly User Interface (Software) can be created to make this process easy and convenient for users.

- **Product Development**: Addition and support of separate hardware such as a working camera and a database so that it can be applied to various locations or can be treated as a completed product.

- **Small Scale Application:** Scope of application on a basic level would include the usage of this real-time recognition system for security purposes on a small scale. For example, it can be used as an extra security measure for our homes.

- **Large Scale Application:** With the help of better equipment, we can apply this product to large scale security systems with CCTVs and increase the accuracy to better recognize faces in a crowd.

SETBACKS & CHALLENGES

Despite the vast advantages, there are four major factors that limit the effectiveness of facial recognition technology:

Poor Image Quality

- Since the image quality of scanning video is quite low compared with that of a digital camera, recognition won't be accurate.

Small Image Sizes/Distant Target

- The relative size of the detecting face compared with the enrolled image size affects the facial recognition.
- So if the target face is distant from the camera, the small sized image of the face goes undetected/not recognized.

Different Face Angles

- The relative angle of the target's face also affects the facial recognition.
- For an accurate result, the view of both enrolled and input images should be nearly the same.

Existing LBP(Local Binary Pattern)

- They produce long Histograms, slowing down the recognition speed especially on large-scale.
- Binary Data produced by them is sensitive to noise.

REFERENCES

HaarCascade Detection in OpenCV here
OpenCV haarcascades files here
Digital Images (8 bit color here and here)
Face Recognition using PAC here
Eigen Faces here here
Haar Filters & legacy Viola-Jones method here
YouTube Videos :
here, here and here - facedetection
Here and here - haar features
8 bit image grayscale here
8 bit image RGB here

// add every reference here please