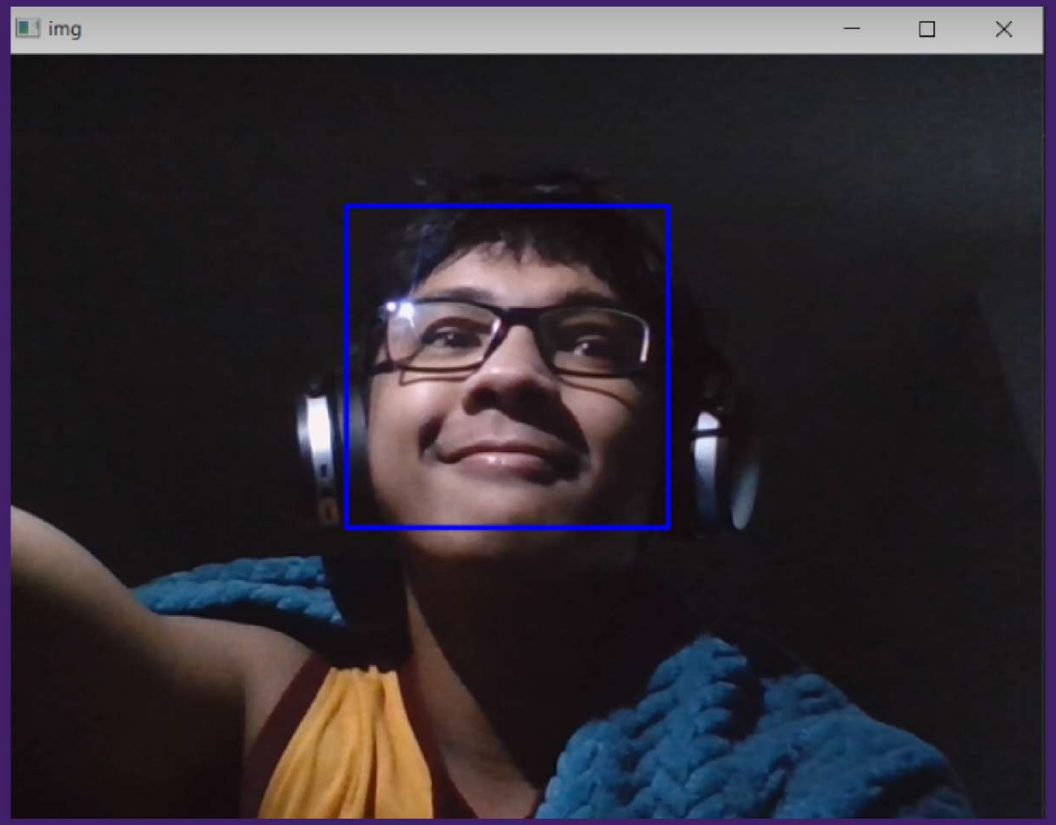
GROUP 4
PRESENTATION

COMPUTER VISION

Members:
Allen S. Roldan
Nethan Gabriel B. Bagasbas
Kirk Vallecera
Kyle Villena
Angelie Garcia



Face Detection using OpenCV

Face detection is the process of identifying one or more human faces in digital images or video streams. OpenCV is a computer vision library that can be used to perform face detection. It provides pre-trained classifiers for detecting faces, eyes, smiles, and other features in images, and it also includes tools for training custom classifiers.

STUDIO SHODWE

System Overview

System overview of Face Detection using OpenCV

The overview of the system is that it uses the OpenCV library to detect faces in images or videos. It does this by applying a trained classifier, specifically a Cascade Classifier, to the image.

The classifier is trained on thousands of positive and negative images of faces, and it uses machine learning algorithms to learn the features of a face.

Once the classifier is trained, it can be used to detect faces in new images by sliding the classifier over the image and checking for regions that match the features of a face. When a face is detected, the system will typically draw a rectangle around the face, highlighting the location of the face in the image. The process can be done real-time using a webcam.

STUDIO SHODWE

Goal of the System

The goal of the system is to detect human faces in real-time video stream, typically from a webcam, using the OpenCV library. This is achieved by training a classifier, such as a Haar or LBP classifier, on a dataset of labeled face images, and then applying the trained classifier to detect faces in new, unseen images. The system utilizes various image processing techniques, such as image pyramids and sliding windows, to increase the robustness and accuracy of the detection.

STUDIO SHODWE

System Scope and Limitations

System Scope

The system scope includes the detection and identification of faces in images and videos using OpenCV and its pre-trained classifiers.

Limitations

The limitations of the system include its inability to detect faces under certain lighting conditions, low resolution images, and certain angles or poses. Additionally, the system may have a lower accuracy for detecting faces of individuals with unique facial features or for detecting multiple faces in an image or video.

STUDIO SHODWE

Conceptual Framework or System Architecture

1The conceptual framework of the system is based on the use of OpenCV, which is an open-source computer vision library.

It uses Haar cascades, which are classifiers that are trained to detect features in images, specifically for face detection. These cascades are trained using thousands of positive and negative images.

The cascade is then applied to an input image, where it scans the image for faces. If a face is detected, it will be highlighted with a rectangle. The architecture of the system is relatively simple, it captures video frames from the camera, and applies the face detection cascade on each frame. The frames with detected faces are then displayed to the user.

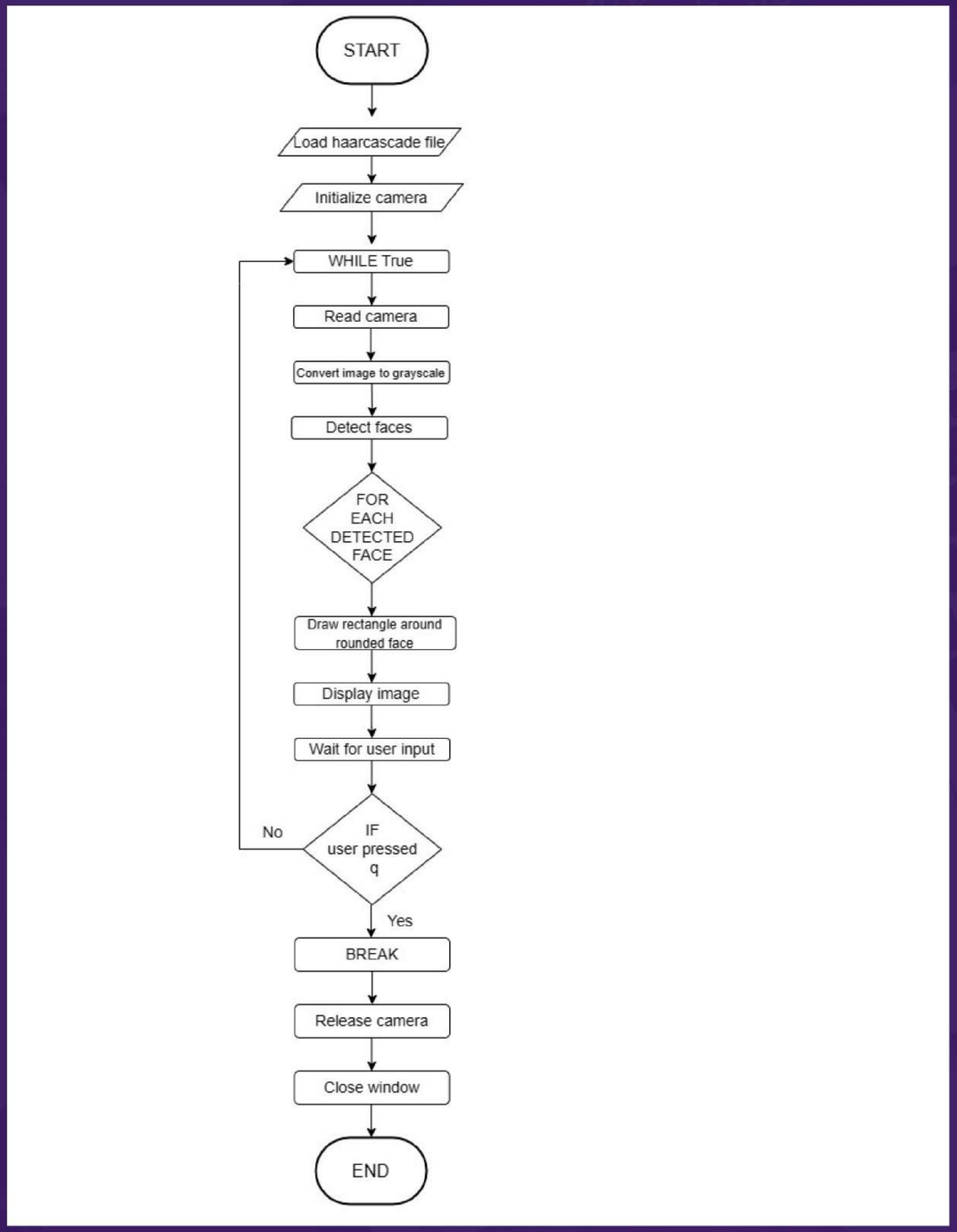
STUDIO SHODWE

Programming Language and Algorithm

The programming language used for face detection using OpenCV is **Python**.

PSEUDOCODE

1. import cv2 library
2. Load the face cascade classifier from the cv2 data using haarcascade_frontalface_default.xml
3. Initialize a video capture object using the default camera (index 0)
4. Start a while loop
   a. Read a frame from the video capture object
   b. Convert the frame to grayscale
   c. Detect faces in the grayscale frame using the face cascade classifier
   d. For each face detected: i. Draw a rectangle around the face in the original frame
   e. Display the frame with the detected faces
   f. Check if the user pressed the 'q' key, if so, exit the loop
5. Release the video capture object
6. Close all windows



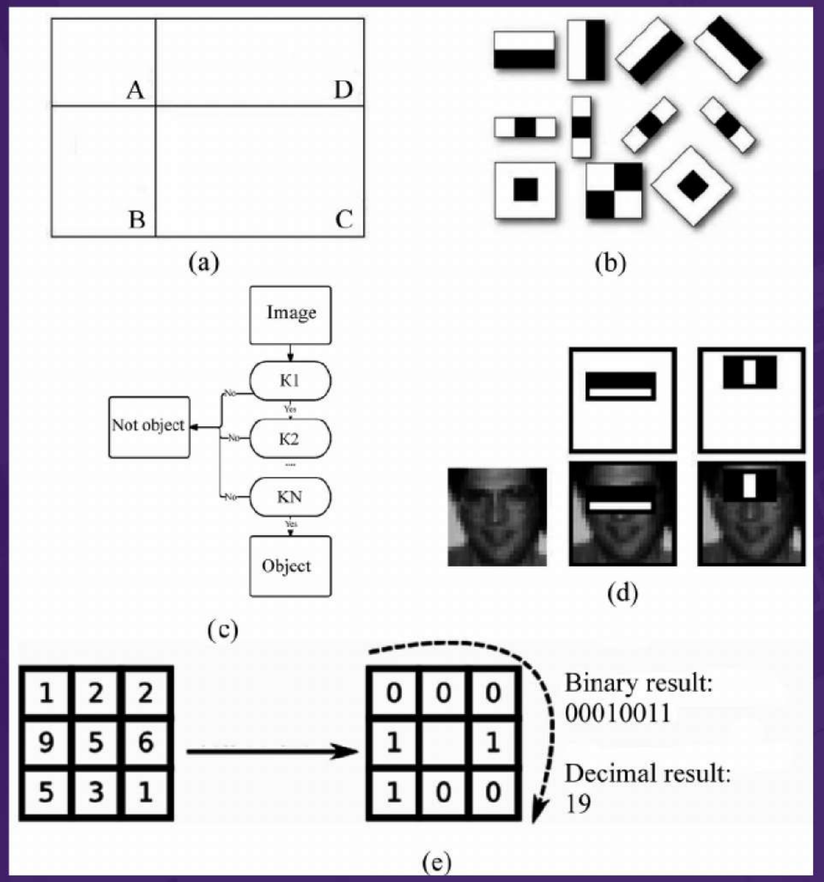
Flowchart

This is the improved version of the Face Detection using Python's OpenCV library.

For a clear view of the flowchart, kindly go enter this link: https://drive.google.com/file/d/1UlUMFJj11kSsfr2_A7qCatGs7-KjzI4G/view?usp=sharing

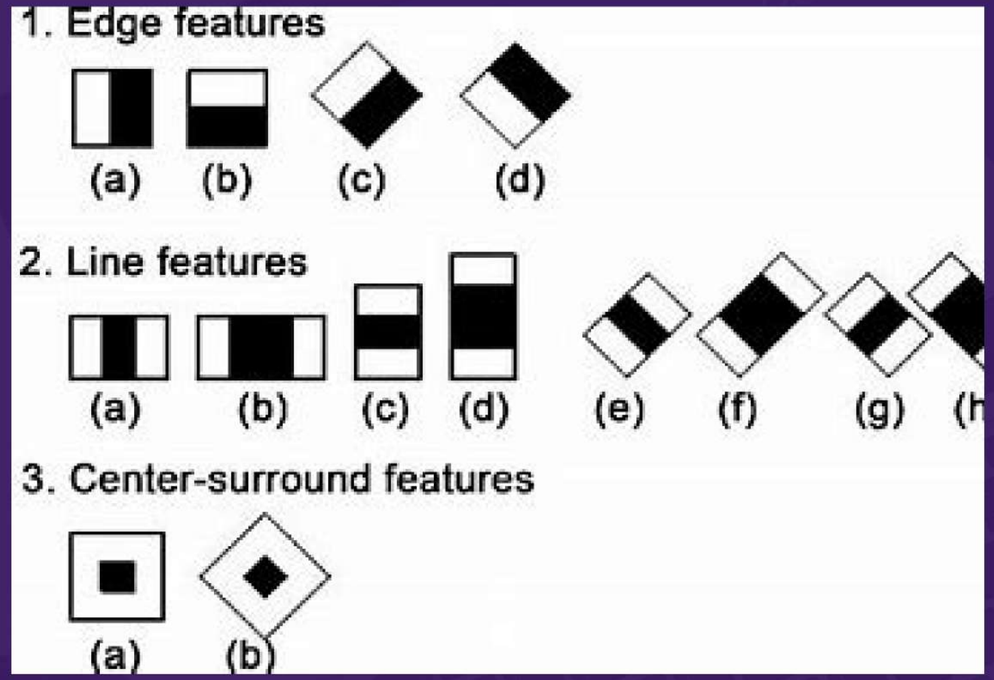
Math Formula/Equation

STUDIO SHODWE



In terms of the math equation, the Haar feature-based cascade classifier uses an integral image representation and the Viola-Jones algorithm to quickly and efficiently detect faces in images.

The Viola-Jones algorithm uses a set of simple features, such as edges and lines, calculated from the integral image, to create a classifier that can detect faces. However, it is not necessary to understand the math equations behind this algorithm to use it in this system.



https://www.researchgate.net/figure/Haar-like-features-9-of-the-pixels-in-the-black-rectangular-regions-are-subtracted_fig3_317012786

The Viola-Jones algorithm uses a concept called "Haar-like features" to detect objects in images. These features are calculated by subtracting the sum of pixel intensities in a white rectangular region from the sum of pixel intensities in a black rectangular region. These values are then used to form a "feature vector" that represents the region of the image being evaluated.

The algorithm uses a set of classifiers, each with a different set of Haar-like features, to evaluate the feature vector and determine if the region contains an object. The classifiers are trained using a set of positive and negative images, where positive images contain the object of interest and negative images do not.

Algorithm

To study the algorithm in detail, we start with the image features for the classification task.

Features and Integral Image

The Viola-Jones algorithm uses Haar-like features, that is, a scalar product between the image and some Haar-like templates. More precisely, let I and P denote an image and a pattern, both of the same size N × N (see Figure 1). The feature associated with pattern P of image I is defined by

$$\sum_{1\leq i\leq N}\sum_{1\leq j\leq N} I(i,j)1_{P(i,j) \text{ is white}} - \sum_{1\leq i\leq N}\sum_{1\leq j\leq N} I(i,j)1_{P(i,j) \text{ is black}}.$$

To compensate the effect of different lighting conditions, all the images should be mean and variance normalized beforehand. Those images with variance lower than one, having little information of interest in the first place, are left out of consideration.

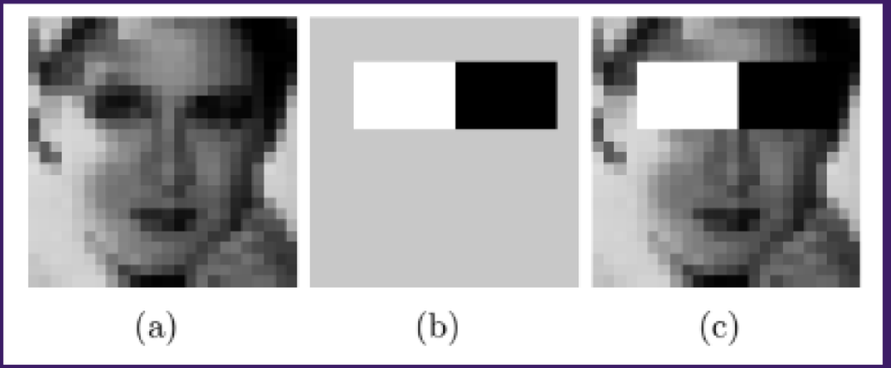


Figure 1: Haar-like features. Here as well as below, the background of a template like (b) is painted gray to highlight the pattern's support. Only those pixels marked in black or white are used when the corresponding feature is calculated.

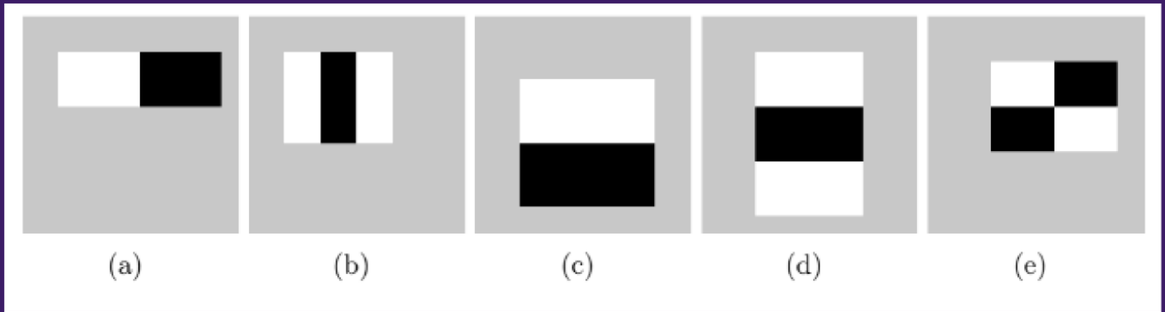


Figure 2: Five Haar-like patterns. The size and position of a pattern's support can vary provided its black and white rectangles have the same dimension, border each other and keep their relative positions. Thanks to this constraint, the number of features one can draw from an image is somewhat manageable: a 24 × 24 image, for instance, has 43200, 27600, 43200, 27600 and 20736 features of category (a), (b), (c), (d) and (e) respectively, hence 162336 features in all.

In practice, five patterns are considered (see Figure 2 and Algorithm 1). The derived features are assumed to hold all the information needed to characterize a face. Since faces are by and large regular by nature, the use of Haar-like patterns seems justified. There is, however, another crucial element which lets this set of features take precedence: the integral image which allows to calculate them at a very low computational cost. Instead of summing up all the pixels inside a rectangular window, this technique mirrors the use of cumulative distribution functions. The integral image II of I

$$\mathrm{II}(i,j) := \begin{cases} \sum_{1 \le s \le i} \sum_{1 \le t \le j} I(s,t), & 1 \le i \le N \text{ and } 1 \le j \le N \\ 0, & \text{otherwise} \end{cases},$$

is so defined that

$$\sum_{N_1 \le i \le N_2} \sum_{N_3 \le j \le N_4} I(i,j) = \mathrm{II}(N_2, N_4) - \mathrm{II}(N_2, N_3 - 1) - \mathrm{II}(N_1 - 1, N_4) + \mathrm{II}(N_1 - 1, N_3 - 1), \quad (1)$$

holds for all N1 ≤ N2 and N3 ≤ N4. As a result, computing an image's rectangular local sum requires at most four elementary operations given its integral image. Moreover, obtaining the integral image itself can be done in linear time: setting N1 = N2 and N3 = N4 in (1), we find

$$I(N_1, N_3) = \mathrm{II}(N_1, N_3) - \mathrm{II}(N_1, N_3 - 1) - \mathrm{II}(N_1 - 1, N_3) + \mathrm{II}(N_1 - 1, N_3 - 1).$$

System Features

STUDIO SHODWE

1. Ability to detect faces in real-time video feed from a camera

2. Can detect multiple faces in an image or frame

3. Uses Haar cascades, which are machine learning-based classifiers, to detect faces

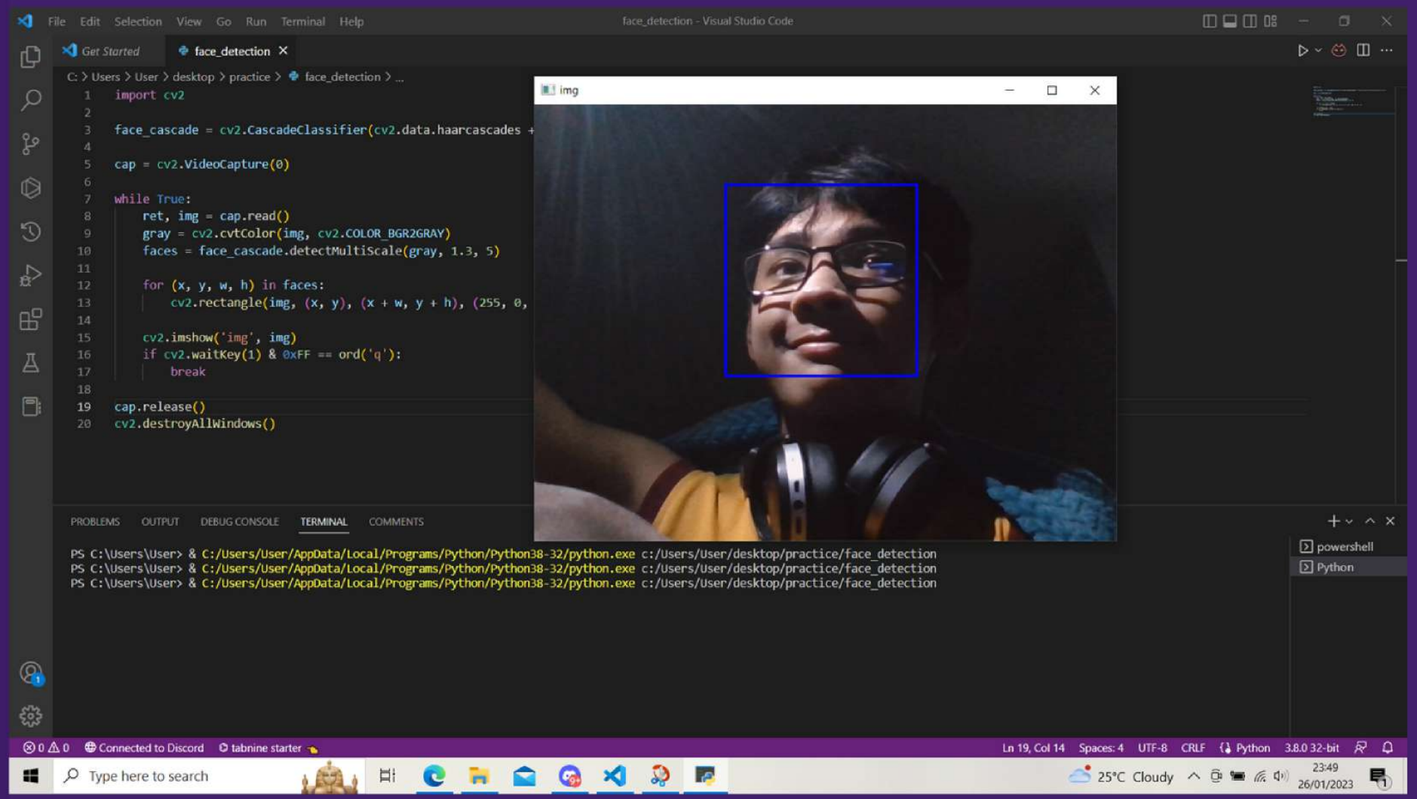
4. Can be used for a wide range of applications such as security systems, photo tagging, and augmented reality

5. Can be integrated with other computer vision algorithms for additional functionality, such as facial recognition or emotions detection

6. The system can be trained to detect other objects or features besides faces, such as eyes, mouths, or full bodies.

STUDIO SHODWE

User Interface Design (Sample Screenshot of the application)



Face detection using OpenCV does not typically involve a user interface design, as it is usually implemented as a background process in a larger application.

However, it is possible to create a simple user interface that allows the user to start and stop the face detection process, adjust settings, and view the output. Such a user interface would be created using a library or framework such as PyQt or Tkinter.

Source Code

```
import cv2

face_cascade =
cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_frontalface_default.xml")

cap = cv2.VideoCapture(0)

while True:
  ret, img = cap.read()
  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  faces = face_cascade.detectMultiScale(gray, 1.3, 5)

  for (x, y, w, h) in faces:
    cv2.rectangle(img, (x, y), (x + w, y + h), (255, 0, 0), 2)

  cv2.imshow('img', img)
  if cv2.waitKey(1) & 0xFF == ord('q'):
    break

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

STUDIO SHODWE

THANK YOU

2022 PRESENTATION