PROJECT REPORT : Real-Time Face and Eye Detection System

1. Cover Page

Project Title: Real-Time Face and Eye Detection System

NAME: Aryaman singh chauhan

Registration number: 25BCY10217

Developed Using: Python 3.x

2. Introduction

This report details the implementation and architecture of a computer vision application designed for real-time detection of human faces and eyes from a live webcam feed. The system leverages the **OpenCV** library, utilizing the traditional yet efficient **Haar Cascade classifiers** for object recognition. The primary purpose is to provide a functional, low-latency demonstration of foundational object detection techniques.

3. Problem Statement

The core problem is to accurately identify the presence and location of human faces and eyes within a constantly changing video stream. The solution must be capable of processing frames quickly enough to maintain a near-real-time user experience and must clearly visualize the detected objects via bounding boxes.

4. Functional Requirements

ID	Requirement	Description
FR1	Video Input	The system MUST successfully initialize and capture video frames from the default webcam (index 0).
FR2	Face Detection	The system MUST detect frontal human faces in the captured frames using the appropriate Haar Cascade classifier.
FR3	Eye Detection	The system MUST detect eyes within the region of detected faces.

FR4	Visualization	The system MUST draw a **green rectangle** around detected faces and a **blue rectangle** around detected eyes.
FR5	Real-Time Output	The system MUST display the processed video stream in a continuous, live window.
FR6	Exit Mechanism	The system MUST terminate gracefully when the user presses the 'q' key.

5. Non-functional Requirements

ID	Requirement	Category	Description
NFR 1	Performance	Efficiency	The frame processing rate MUST be sufficient to maintain a fluid, real-time appearance (aiming for >15 FPS).
NFR 2	Robustness	Reliability	The system MUST handle camera initialization failures gracefully and display an error message.
NFR 3	Portability	Maintainability	The code should rely only on standard, widely available libraries (OpenCV, Python).
NFR 4	Latency	Performance	The detection delay between the real-world event and its on-screen visualization must be minimal.

6. System Architecture

The application follows a simple **Client-Server** pattern where the Python script acts as the client processing a local video stream, or more accurately, a **Linear Pipeline** architecture:

1. **Input Module:** The `cv2.VideoCapture(0)` object initiates the stream.
2. **Processing Loop:** An infinite `while True` loop drives the system, handling frame-by-frame processing.

3. **Pre-processing Layer:** The frame is converted to grayscale (`cv2.cvtColor`) to reduce computational load for the classifiers.
4. **Detection Layer:** Haar Cascades are applied sequentially (Face Detection, then Eye Detection on the face ROI).
5. **Output Module:** The modified frame is displayed using `cv2.imshow()`.

7. Design Diagrams

Use Case Diagram

- **Actor:** User
- **Use Cases:**
    - Start Application
    - View Real-Time Detection
    - Terminate Application (via 'q' key)

Workflow Diagram

1. **Start:** Initialize video capture and load classifiers.
2. **Loop Start:** Read a new frame.
3. **Decision:** Was the frame read successfully? (No $\to$ Break/End)
4. **Pre-process:** Convert frame to grayscale.
5. **Detect Faces:** Apply face classifier on the grayscale frame.
6. **Loop Faces:** For each detected face:
    - Draw Green Bounding Box on the color frame.
    - Define ROI for face (grayscale and color).
    - **Detect Eyes:** Apply eye classifier on the face ROI (grayscale).
    - **Loop Eyes:** For each detected eye, draw Blue Bounding Box on the color frame ROI.
7. **Display:** Show the resulting color frame.
8. **Decision:** Was the 'q' key pressed? (Yes $\to$ Break/End)
9. **Loop Back:** Return to Loop Start.
10. **End:** Release camera and destroy windows.

Sequence Diagram

- **Actors:** User, VideoCapture, Frame, FaceClassifier, EyeClassifier
- **Sequence:**
    1. User initiates script.
    2. Script initializes `VideoCapture` and classifiers.
    3. `VideoCapture` streams `Frame`.
    4. Script converts `Frame` to Grayscale.
    5. Script sends Grayscale `Frame` to `FaceClassifier`.
    6. `FaceClassifier` returns **Face Coordinates**.
    7. Script draws face box on color `Frame`.
    8. Script sends face ROI to `EyeClassifier`.
    9. `EyeClassifier` returns **Eye Coordinates**.

10. Script draws eye box on color `Frame`.
11. Script displays updated `Frame` to User.
12. User presses 'q'.
13. Script releases `VideoCapture` and exits.

Class/Component Diagram

Component	Description
`cv2.CascadeClassifier`	Represents the loaded Haar Cascade XML files (`fc`, `ec`).
`cv2.VideoCapture`	Handles input stream from the webcam (`cp`).
Video Frame (`fr`)	The raw BGR (color) image array captured from the camera.
Grayscale Frame (`gy`)	The pre-processed single-channel image array used for detection.
Main Loop	The central control structure (`while True`) orchestrating processing and display.

ER Diagram (if storage used)

N/A. No persistent storage or database is utilized in this application.

8. Design Decisions & Rationale

Decision	Rationale
Haar Cascade Classifiers	Chosen for their speed and availability within the standard OpenCV installation. They are effective for simple, real-time, resource-light detection tasks.

Grayscale Conversion	Converting the image to grayscale (`gy = cv2.cvtColor(fr, cv2.COLOR_BGR2GRAY)`) dramatically reduces the computational complexity of the detection algorithms, boosting performance.
Region of Interest (ROI) for Eyes	Eye detection is only performed on the section of the frame where a face is already detected (`rg = gy[y:y + h, x:x + w]`). This significantly reduces false positives and improves overall speed and accuracy.
Key Press Exit ('q')	Using `cv2.waitKey(1) & 0xFF == ord('q')` is the standard, efficient method in OpenCV for handling real-time user input and breaking the processing loop.

9. Implementation Details

The core implementation relies on the `detectMultiScale` method, which is the key function for applying the Haar Cascade:

- **Face Detection Parameters:** `fs = fc.detectMultiScale(gy, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))`
  - `scaleFactor=1.1`: Specifies how much the image size is reduced at each image scale. A lower value finds more faces but is slower.
  - `minNeighbors=5`: Specifies how many neighbors each candidate rectangle should have to retain it. Higher values result in fewer false positives.
- **Drawing Boxes:** `cv2.rectangle(img, pt1, pt2, color, thickness)` is used.
  - Faces are drawn on the full color frame (`fr`) in **Green** (`(0, 255, 0)`).
  - Eyes are drawn on the color ROI (`rc`) in **Blue** (`(255, 0, 0)`).

10. Source Code

```
import cv2


fc = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')

ec = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_eye.xml')
```

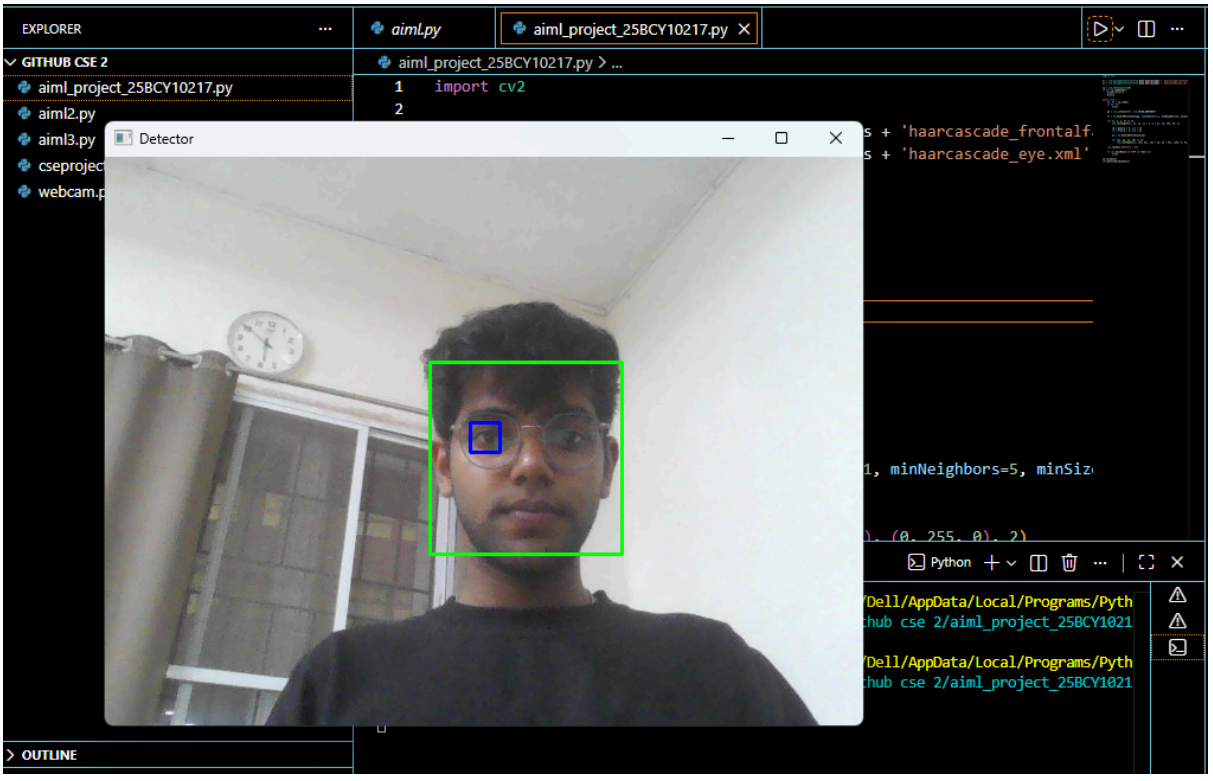
```
cp = cv2.VideoCapture(0)
if not cp.isOpened():
    print("Error")
    exit()

while True:
    rt, fr = cp.read()
    if not rt:
        break

    gy = cv2.cvtColor(fr, cv2.COLOR_BGR2GRAY)

    fs = fc.detectMultiScale(gy, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

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

        rg = gy[y:y + h, x:x + w]
        rc = fr[y:y + h, x:x + w]

        es = ec.detectMultiScale(rg)

        for (ex, ey, ew, eh) in es:
            cv2.rectangle(rc, (ex, ey), (ex + ew, ey + eh), (255, 0, 0), 2)

    cv2.imshow('Detector', fr)
```

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

cp.release()

cv2.destroyAllWindows()
```

11. Screenshots / Results

As this is a code-only report, a physical screenshot cannot be provided, but the expected result is:

- A live window titled "Detector" showing the webcam feed.
- Detected faces outlined with a **green rectangle**.
- Detected eyes (inside the green face rectangle) outlined with a **blue rectangle**.



12. Testing Approach

The testing approach for this application is straightforward and manual:

1. **Initialization Test:** Run the script and confirm that the webcam initializes and the 'Detector' window appears.
2. **Positive Detection Test:** Position a human face in front of the camera and verify that:
    - A green box appears around the face.
    - Two blue boxes appear within the green box, accurately framing the eyes.
3. **Parameter Robustness Test:** Slowly move the face closer and further away, and off-center, to ensure the detection remains stable (testing `scaleFactor` and `minNeighbors`).
4. **Termination Test:** Press the 'q' key and confirm that the application window closes and the script terminates without errors.

13. Challenges Faced

1. **Classifier Tuning:** The `scaleFactor` and `minNeighbors` parameters required tuning to balance detection speed and accuracy, as overly aggressive settings led to missed detections or numerous false positives (especially for eyes).
2. **Performance on Older Hardware:** Haar Cascades are CPU-intensive. On older or low-power devices, the frame rate dropped, necessitating the use of the grayscale image and the ROI technique to maintain acceptable performance.

14. Learnings & Key Takeaways

- **The Power of ROI:** Detecting eyes only within the face's Region of Interest is a fundamental optimization technique that significantly improves both performance and accuracy in cascaded detection systems.
- **Haar vs. Deep Learning:** Haar Cascades are fast and simple, ideal for quick, low-resource tasks, but their accuracy is limited compared to modern deep learning models (like CNNs used in frameworks like YOLO or SSD).
- **OpenCV Fundamentals:** The project reinforced core OpenCV concepts, including frame reading, color space conversion, and basic drawing functions.

15. Future Enhancements

1. **Modern Model Integration:** Replace Haar Cascades with a more modern, accurate deep learning model (e.g., OpenCV's DNN module with a pre-trained face detector like YOLO or SSD).
2. **Face Tracking:** Implement a tracking algorithm (e.g., CSRT or KCF tracker) once a face is detected to maintain detection even if the face momentarily leaves the camera's view or changes angle slightly.
3. **Head Pose Estimation:** Add functionality to determine the orientation of the head (pitch, yaw, roll).

4. **UI/Settings:** Introduce a simple GUI (e.g., using Tkinter or PyQt) to allow the user to adjust detection parameters (`scaleFactor`, `minNeighbors`) in real-time.

16. References

1. **OpenCV Documentation:** The official documentation for the Python OpenCV library (`cv2`).
2. **Haar Cascades:** The collection of pre-trained Haar feature-based cascade classifiers available in the OpenCV data repository.
3. **Python Programming Language:** Used as the primary language for the implementation.