PROCTORLY

PROJECT WORK

Madhav Choudhary

- LinkedIn - Github

PROJECT FILES

DEMO VIDEO

ProctorLy:

An AI-Driven Examination Proctoring Solution

“ Table of Contents “s

---

1. Introduction

With the rapid digital transformation accelerated by COVID-19, educational institutions have shifted towards online learning and examination models. However, this shift has raised serious concerns around academic integrity, particularly regarding secure, fair exam proctoring. Traditional online meeting tools such as Google Meet and Zoom are not designed to oversee high-stakes examinations, leaving institutions vulnerable to misconduct, cheating, and the challenges of maintaining academic credibility.

ProctorLy is a solution designed to address these challenges using state-of-the-art artificial intelligence (AI) and machine learning (ML) models. Its purpose is to monitor exams with high accuracy and fairness by employing models that detect various indicators of cheating, such as head movement, eye gaze direction, ambient sounds, and even the presence of unauthorized devices. ProctorLy emphasizes the privacy of students by deleting data immediately after each exam session, storing only minimal, anonymized records of violations.

This report elaborates on ProctorLy's core features, technical implementation, and the potential it holds to reshape examination standards, promoting integrity without compromising privacy or user experience. We will detail the motivation behind ProctorLy, the AI techniques used, system architecture, challenges faced, and lessons learned throughout the development process.

---

2. Project Overview

2.1. Motivation

The concept of ProctorLy was born from the necessity of maintaining fairness in online education. The transition to virtual exams has exposed a significant gap in maintaining academic integrity, as current tools lack the sophistication required to enforce strict proctoring measures. Institutions found that human proctors cannot adequately monitor every student due to limitations in visual attention and a lack of effective digital enforcement tools.

ProctorLy aims to automate this process, utilizing AI to ensure every student is monitored equally. The system enhances proctoring accuracy, reduces human resource costs, and strengthens institutions' ability to uphold academic standards.

2.2. Objectives

The primary objectives of ProctorLy include:

1. **Automated Surveillance**: Provide real-time AI-driven surveillance throughout online exams, allowing seamless detection of cheating behaviors and unauthorized devices.
2. **Accuracy and Efficiency**: Leverage cutting-edge ML models to ensure high accuracy with minimal latency, making the system reliable and usable in real-time.
3. **Privacy-Focused Approach**: Uphold student trust by ensuring no personal data is stored beyond the exam session and that data is anonymized when reported to instructors.
4. **Accessibility**: Design an interface that requires minimal technical setup, making ProctorLy accessible for students with different tech skill levels.
5. **Scalability**: Develop ProctorLy to be highly scalable, allowing educational institutions to deploy it for various exam formats and audience sizes without increased costs or loss of efficiency.

---

3. Problem Statement

The onset of COVID-19 amplified the need for secure online examination platforms, which existing video conferencing tools could not provide. This problem affects educational institutions globally, compromising the validity of online exams and weakening the value of online certifications. Key issues in existing online examination methods include:

3.1. Inadequate Monitoring: Human proctors lack the ability to monitor multiple students at once, resulting in undetected cheating instances.

3.2. Lack of Real-Time Intervention: Human-based systems delay response times, leading to reactive rather than preventive measures.

3.3. Privacy Concerns: Traditional surveillance raises student discomfort and concerns about data security, with many tools not ensuring full data deletion.

3.4. Cost and Accessibility: Traditional proctoring is costly and logistically complex, especially for large-scale examinations.

ProctorLy addresses these issues by providing a high-accuracy, privacy-conscious solution that ensures seamless, fair examination processes.

---

4. Features of ProctorLy

4.1. Website Features

1. **Tab Switch Detection**: Tracks any attempts by the user to switch tabs, instantly flagging this as a potential cheating action. This feature discourages students from navigating away from the test environment.
2. **Copy-Pasting Detection**: Prevents any copy-pasting activity within the exam interface, ensuring originality in responses and preventing easy access to pre-prepared answers.
3. **Warning System**: An integrated alert system warns students immediately when any suspicious activity is detected, discouraging further cheating attempts without manual proctor intervention.

4.2. AI-Driven Models

1. **Eye Tracking Model**: Uses computer vision to analyze eye movement patterns, ensuring students maintain focus on the screen. The model detects frequent gaze shifts, a potential indicator of cheating.
2. **Headphone Detection**: Identifies if a student is wearing headphones or earbuds, potentially communicating answers from an external source. This model ensures fair testing by flagging any unauthorized audio devices.
3. **Malicious Object Detection**: This deep learning model detects objects like books, mobile phones, and notes within the frame, which could be used for unauthorized reference. It utilizes YOLOv4 to identify these items with high accuracy.
4. **Mouth Movement Tracking**: Identifies when students open their mouths to speak, which may indicate communication with an external person. This model contributes to enforcing a silent exam environment.
5. **Person Detection**: Recognizes the presence of additional people in the room, alerting proctors to potential collusion or assistance.
6. **Similarity Checker**: Uses natural language processing (NLP) to compare answer similarity across students, detecting potential collaboration or copied answers.
7. **Speech Recognition**: Analyzes ambient sounds for specific keywords that may relate to exam content, helping detect verbal communication attempts.

Each feature contributes to a comprehensive monitoring system that strengthens ProctorLy's ability to detect a range of cheating behaviors effectively and in real-time.

---

5. Project Structure

The project structure is divided into three major components:

- **Frontend**: Built using HTML, CSS, JavaScript, and ReactJS, the frontend offers a responsive and interactive interface that guides students through the exam process. TailwindCSS and Tachyons streamline UI consistency and aesthetic design.
- **Backend**: Powered by Python and frameworks like TensorFlow and Keras, the backend manages data processing for the AI models. Through efficient API integration, it communicates seamlessly with the frontend, processing real-time data from student devices.
- **Database**: Although ProctorLy doesn't store sensitive data, it temporarily uses session data to handle login and exam information. This is promptly deleted after the exam.
- **APIs and Middleware**: ProctorLy integrates various APIs for speech recognition and image processing. Middleware coordinates between the frontend and backend, ensuring efficient communication between user inputs and the server's ML models.
- **Data Handling and Privacy**: ProctorLy's data handling is geared towards transient processing. All collected data is processed immediately for analysis and deleted within minutes, ensuring no personal information is stored beyond what is necessary for reporting infractions.

---

6. Design and Architecture

ProctorLy's design and architecture focus on creating a seamless and efficient proctoring system that prioritizes security, privacy, and scalability. The system integrates both frontend and backend components through a centralized processing server that performs real-time analysis, detects anomalies, and generates detailed reports. These components are tightly connected through secure communication protocols to ensure the safe handling of sensitive data while maintaining a high level of user experience.

6.1 System Design

The architecture of ProctorLy is composed of three major components: the **frontend**, **backend**, and **central processing server**. Each plays a vital role in delivering real-time online proctoring while maintaining security and privacy.

6.1.1. Frontend: The frontend is the user-facing component of ProctorLy. It consists of a web application that allows students to take exams and interact with the monitoring system. It captures video, audio, and screen activity, providing real-time data to the backend servers.

- The frontend interface is designed to be minimalistic and non-intrusive to ensure that students can focus on their exams without distractions. It only activates proctoring features when necessary and provides real-time feedback to the student, such as alerts for suspicious behavior.

6.1.2. Backend: The backend is responsible for receiving data from the frontend, running machine learning (ML) models, and coordinating the communication between the frontend and central processing server. The backend handles data collection, processes it through AI models for anomaly detection, and sends alerts if potential violations are detected.

- The backend also stores logs for audit purposes but ensures data is anonymized and processed securely.

6.1.3. Central Processing Server: This server acts as the brain of the system. It coordinates real-time processing and integrates results from various ML models. It is responsible for performing high-speed analysis, managing data flow, and ensuring that every action is logged accurately.

- The processing server uses **secure encryption protocols** to safeguard communication, ensuring that all data transmitted between students and proctors is encrypted to prevent unauthorized access.

6.2 Data Flow and Security Protocols

To ensure the integrity and confidentiality of the data, ProctorLy follows a strict protocol that governs how data flows through the system, from collection to deletion.

6.2.1. Data Flow Steps:

1. **Data Collection**: Real-time video and audio data are continuously collected from the student's device using the frontend application. This includes capturing the student's face, voice, and screen activity to detect any suspicious behavior.
   - The system also tracks system resources, such as CPU usage and browser activities, to monitor for potential signs of cheating like switching tabs or running unauthorized software.
2. **Data Analysis**: Once the data is collected, it is sent to the backend where various **AI models** are applied to analyze the collected information in real time. These models detect behavior such as eye movements, head orientation, speaking patterns, and any other suspicious activity.
   - Data from the camera is processed using machine learning models like **Dlib**, **YOLO**, and **OpenCV** to detect anomalies like eye contact, external devices, or additional people in the room.
3. **Report Generation**: If the system identifies any potential violations, such as tab switching or unauthorized object detection, a detailed report is generated. The report is anonymized to protect student privacy but contains all necessary information for instructors to review.
   - The report includes timestamps of detected incidents, a summary of suspicious activities, and evidence, such as captured images or flagged behaviors. This helps instructors assess the severity of the situation before taking any action.

4. **Data Deletion**: To protect student privacy, ProctorLy adheres to strict data privacy regulations. Once the analysis is completed and the report is generated, all collected data—video, audio, and logs—are **promptly deleted** from the system. This process ensures that no personal information is stored after the exam ends, mitigating any privacy concerns.

6.2.2. Security Protocols:

- All data transmitted within ProctorLy is encrypted using **end-to-end encryption**. This ensures that no unauthorized entity can access the information as it moves through the system.
- Data storage is highly secure, with encrypted temporary storage for any logs or flagged information. Once the session ends, all data is purged from the system, ensuring full compliance with data protection regulations like **GDPR** and **FERPA**.

6.3 Machine Learning Model Pipeline

The ML models within ProctorLy work in parallel to provide real-time feedback and ensure rapid detection of cheating behaviors. These models process data from various sources, such as video, audio, and screen activities, to detect suspicious actions. The results from all models are aggregated into a central decision-making unit, which evaluates whether any violations have occurred.

- **Parallel Processing**: ProctorLy uses **multi-threaded processing** to allow each model to run independently but simultaneously. This ensures minimal latency in detecting potential violations. For example, while the **eye tracking model** is processing video feed, the **headphone detection model** and **malicious object detection** model can be analyzing different frames of the feed in parallel.
- **Centralized Decision-Making Unit**: After each model performs its analysis, the results are aggregated by a central decision unit. This unit assesses whether a threshold for violation is met, and if so, generates a violation report for review.
- **Scalability**: The parallelized nature of the model pipeline enables ProctorLy to scale efficiently. It can process hundreds or thousands of students simultaneously during large exams without a significant drop in performance, making it suitable for both small classroom exams and large certification or university-wide tests.

---

7. Machine Learning Models

ProctorLy's anti-cheating suite relies on a series of ML models, each optimized for detecting specific behaviors that may indicate cheating. The design of each model ensures high accuracy and minimal processing delay, allowing for real-time monitoring without compromising the exam experience.

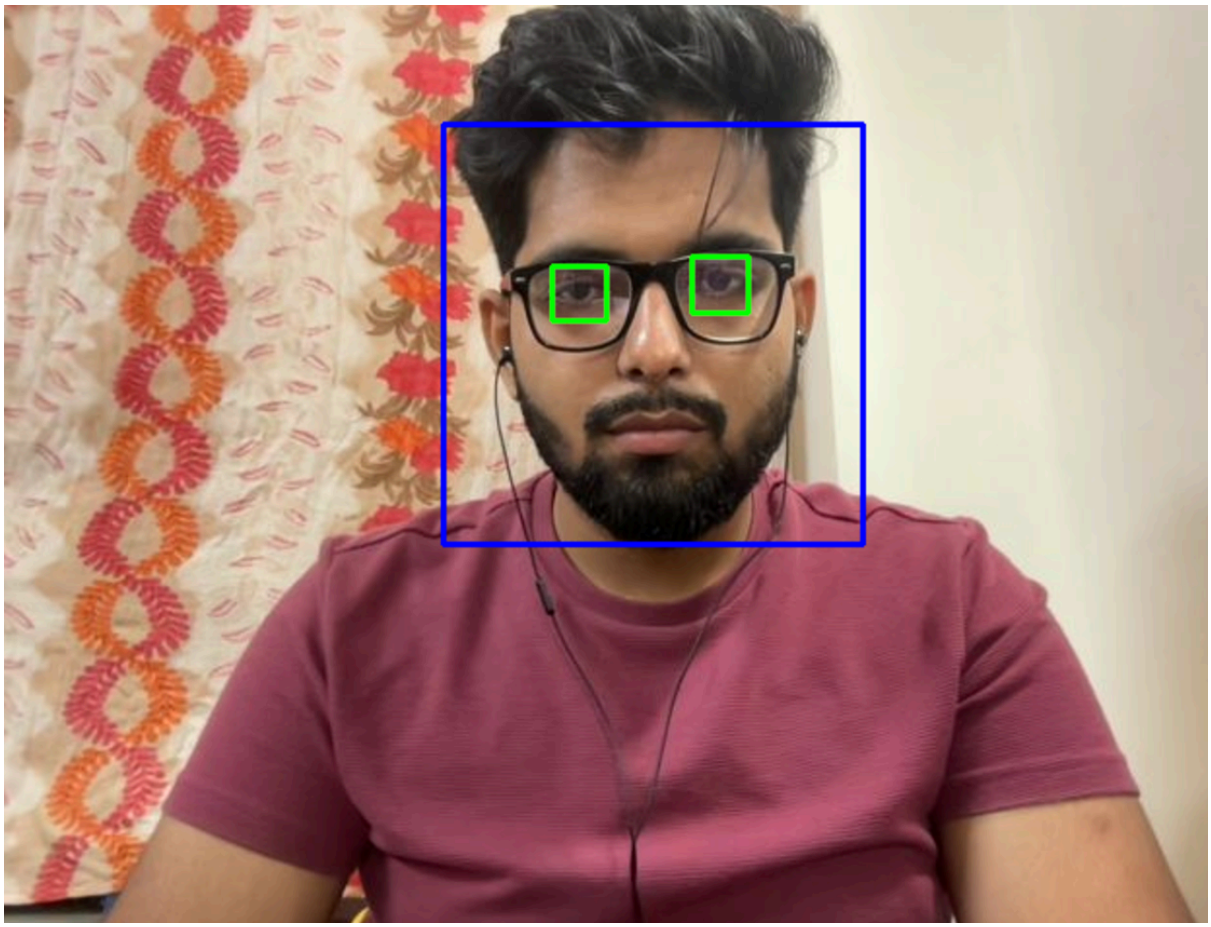
7.1. Eye Tracking Model

7.1.1. Purpose: Detects when a student's gaze frequently shifts away from the screen, a behavior often associated with reading unauthorized materials or seeking external assistance.

7.1.2. Technology Used: This model uses **Dlib** and **OpenCV** for facial landmark and real-time eye tracking. Dlib provides the critical facial landmark detection capabilities, isolating the eye region, while OpenCV helps capture and analyze gaze direction.

7.1.3. Workflow:

1. **Eye Region Detection**: Dlib identifies and isolates the eye region based on facial landmarks.
2. **Gaze Analysis**: The model then calculates gaze direction by measuring the relative position of eye landmarks.
3. **Pattern Monitoring**: It assesses gaze patterns over time, raising an alert if a student frequently looks away from the screen.



7.1.4. Challenges:

- **Lighting Conditions**: Varying ambient light can affect detection accuracy. To mitigate this, the model uses **adaptive thresholding**, which adjusts to different lighting conditions in real-time.
- **Camera Quality Variability**: Low-quality webcams can hinder accurate eye tracking. Additional image preprocessing techniques help improve consistency across different camera resolutions.

7.1.5. Training: The model was trained on a diverse dataset featuring different lighting and head angles, enhancing resilience and reducing false negatives.

7.1.6. CODE :

```
# -*- coding: utf-8 -*-
"""Eye_Detection.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1M_rH4l5sidb43_kEQ293kTUAx0UUzdDN
"""

# IMPORTING NECESSARY LIBRARIES AND FILES

from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from google.colab.patches import cv2_imshow
from base64 import b64decode, b64encode
import cv2
import numpy as np
import dlib

def video_stream():
  js = Javascript('''
    var video;
    var div = null;
    var stream;
    var captureCanvas;
    var imgElement;
    var labelElement;

    var pendingResolve = null;
    var shutdown = false;

    function removeDom() {
       stream.getVideoTracks()[0].stop();
       video.remove();
       div.remove();
       video = null;
       div = null;
       stream = null;
       imgElement = null;
       captureCanvas = null;
       labelElement = null;
    }

    function onAnimationFrame() {
      if (!shutdown) {
        window.requestAnimationFrame(onAnimationFrame);
```

```
        }
        if (pendingResolve) {
          var result = "";
          if (!shutdown) {
            captureCanvas.getContext('2d').drawImage(video, 0, 0, 640, 480);
            result = captureCanvas.toDataURL('image/jpeg', 0.8)
          }
          var lp = pendingResolve;
          pendingResolve = null;
          lp(result);
        }
      }

      async function createDom() {
        if (div !== null) {
          return stream;
        }

        div = document.createElement('div');
        div.style.border = '2px solid black';
        div.style.padding = '3px';
        div.style.width = '100%';
        div.style.maxWidth = '600px';
        document.body.appendChild(div);

        const modelOut = document.createElement('div');
        modelOut.innerHTML = "<span>Status:</span>";
        labelElement = document.createElement('span');
        labelElement.innerText = 'No data';
        labelElement.style.fontWeight = 'bold';
        modelOut.appendChild(labelElement);
        div.appendChild(modelOut);

        video = document.createElement('video');
        video.style.display = 'block';
        video.width = div.clientWidth - 6;
        video.setAttribute('playsinline', '');
        video.onclick = () => { shutdown = true; };
        stream = await navigator.mediaDevices.getUserMedia(
            {video: { facingMode: "environment"}});
        div.appendChild(video);

        imgElement = document.createElement('img');
        imgElement.style.position = 'absolute';
        imgElement.style.zIndex = 1;
        imgElement.onclick = () => { shutdown = true; };
        div.appendChild(imgElement);

        const instruction = document.createElement('div');
        instruction.innerHTML =
            '<span style="color: red; font-weight: bold;">' +
```

```
            'When finished, click here or on the video to stop this demo</span>';
        div.appendChild(instruction);
        instruction.onclick = () => { shutdown = true; };

        video.srcObject = stream;
        await video.play();

        captureCanvas = document.createElement('canvas');
        captureCanvas.width = 640; //video.videoWidth;
        captureCanvas.height = 480; //video.videoHeight;
        window.requestAnimationFrame(onAnimationFrame);

        return stream;
      }
      async function stream_frame(label, imgData) {
        if (shutdown) {
          removeDom();
          shutdown = false;
          return '';
        }

        var preCreate = Date.now();
        stream = await createDom();

        var preShow = Date.now();
        if (label != "") {
          labelElement.innerHTML = label;
        }

        if (imgData != "") {
          var videoRect = video.getClientRects()[0];
          imgElement.style.top = videoRect.top + "px";
          imgElement.style.left = videoRect.left + "px";
          imgElement.style.width = videoRect.width + "px";
          imgElement.style.height = videoRect.height + "px";
          imgElement.src = imgData;
        }

        var preCapture = Date.now();
        var result = await new Promise(function(resolve, reject) {
          pendingResolve = resolve;
        });
        shutdown = false;

        return {'create': preShow - preCreate,
                'show': preCapture - preShow,
                'capture': Date.now() - preCapture,
                'img': result};
      }
      ''')

```

```
    display(js)

def video_frame(label, bbox):
    data = eval_js('stream_frame("{}", "{}")'.format(label, bbox))
    return data




def js_to_image(js_reply):
    """
    Params:
            js_reply: JavaScript object containing image from webcam
    Returns:
            img: OpenCV BGR image
    """
    # decode base64 image
    image_bytes = b64decode(js_reply.split(',')[1])
    # convert bytes to numpy array
    jpg_as_np = np.frombuffer(image_bytes, dtype=np.uint8)
    # decode numpy array into OpenCV BGR image
    img = cv2.imdecode(jpg_as_np, flags=1)

    return img

# Loading Classifiers
face_cascade = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
eye_cascade = cv2.CascadeClassifier("haarcascade_eye.xml")

# Capturing live video stream with help of Javascript Helper Function
cap = cv2.VideoCapture(0)
video_stream()

label_html = "Detecting ..."
bbox = ''
count = 0
detector = dlib.get_frontal_face_detector()

def detect(gray, frame):
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
    for (x, y, w, h) in faces :
      cv2.rectangle(frame , (x,y), (x+w ,y+h), (255, 0, 0), 2 )   # If face was
detected, we will get the the x,y coordinates of the detected feature
      roi_gray = gray[ y:y+h ,x:x+w ]
      roi_color = frame[y:y+h ,x:x+w ]

      eyes = eye_cascade.detectMultiScale(roi_gray, 1.1, 3)
      for (ex, ey, ew, eh) in eyes :
```

```
        cv2.rectangle(roi_color, (ex, ey), (ex+ew, ey+eh), (0, 255, 0), 2)  # If eyes were detected, we will get the the x,y coordinates of the detected feature

    return frame

capture = cv2.VideoCapture(0)

while True :
    js_reply = video_frame(label_html, bbox)
    if not js_reply:
        break
    # Reading image from video stream
    frame = js_to_image(js_reply["img"])

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    canvas =detect(gray, frame) # Detecting the features
    cv2_imshow(canvas)
    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

capture.release()
cv2.destroyAllWindows()
```

7.1.7. Eye Detection Application: Code Analysis and Report

7.1.7.1. File Overview: eye_detection_app.py

This file is a Python application that uses computer vision techniques to detect faces and eyes in a real-time video stream from a webcam. It leverages pre-trained Haar Cascade classifiers to perform face and eye detection.

7.1.7.2. Code Explanation by Sections

1. Importing Libraries

```
import cv2
import numpy as np
import dlib
```

- **cv2** (OpenCV): This is the primary library for computer vision tasks, providing tools to work with images and videos. It is used here for image processing and detection functions.
- **numpy**: This library is used for numerical operations, particularly for array manipulations. Although `numpy` is imported, it is not explicitly used in the code provided.
- **dlib**: Another library for machine learning and computer vision. It is imported but not used in this script.

2. Loading Classifiers

```
face_cascade = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
eye_cascade = cv2.CascadeClassifier("haarcascade_eye.xml")
```

- **face_cascade** and **eye_cascade** are created using pre-trained Haar Cascade classifiers. These XML files contain the trained models for face and eye detection. OpenCV provides them for face and eye recognition tasks, making detection fast and efficient.

3. Initializing the Webcam Capture

```
cap = cv2.VideoCapture(0)
```

- This line initializes video capture from the primary webcam (`0` denotes the default webcam). The captured video frames will be processed to detect faces and eyes.

4. Defining the Detection Function

```
def detect(gray, frame):
```

The detect function processes each frame, looking for faces and eyes.

- **Detecting Faces**

```
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
```

  - This line detects faces in the grayscale image (`gray`). `detectMultiScale` returns a list of coordinates where faces are detected. The parameters adjust the sensitivity and scale factor of detection.
  - For each detected face, a rectangle is drawn around it.

- **Detecting Eyes**

```
eyes = eye_cascade.detectMultiScale(roi_gray, 1.1, 3)
```

  - This line detects eyes within the detected face region (`roi_gray`). Only regions identified as faces are checked for eyes, which improves accuracy and efficiency. A green rectangle is drawn around each detected eye.

5. Processing and Displaying the Video Stream

```
while True:
```

The code enters an infinite loop to process each frame from the video stream.

- **Reading Frame**

```
ret, frame = cap.read()
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

  - Each frame is captured and converted to grayscale (required by the Haar Cascade model).

- **Applying the Detection Function**

```
canvas = detect(gray, frame)
```

  - The `detect` function is called to identify faces and eyes, and rectangles are drawn around them in the `frame`.

- **Displaying Results**

```
cv2.imshow('Eye Detection', canvas)
```

  - The processed frame is displayed with detected faces and eyes highlighted. The program waits for the 'q' key to break the loop and terminate the capture.

6. Releasing Resources

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

After the loop, this code releases the webcam resource and closes all OpenCV windows.

7.1.7.4. Summary of the File

The **eye_detection_app.py** file is a real-time face and eye detection application. It captures live video from the webcam, processes each frame to detect faces and eyes using Haar Cascade classifiers, and displays the processed frame. The application will run indefinitely until the 'q' key is pressed.

7.1.7.5. Key Code Highlights and Explanation

- **Haar Cascade Classifiers**: These are crucial for object detection in this script, offering a simple and effective way to detect features.

```
face_cascade = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
eye_cascade = cv2.CascadeClassifier("haarcascade_eye.xml")
```

- **Loop for Real-time Detection**: The `while` loop allows continuous video processing, keeping the application responsive to new frames from the webcam.

```
while True:
    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    ...
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

- **Rectangle Drawing**: Detected faces and eyes are highlighted using rectangles, making it easy for users to see the detection in real-time.

```
cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)
cv2.rectangle(roi_color, (ex, ey), (ex + ew, ey + eh), (0, 255, 0), 2)
```

7.1.7.6. Frameworks and Libraries Used

1. **OpenCV**:
   - **Definition**: Open Source Computer Vision Library, designed to process images and videos for real-time applications.
   - **Reason for Use**: Efficient in object detection tasks (like face and eye detection) and includes pre-trained Haar Cascade classifiers.
   - **Usage in Code**:
     - Initializing `CascadeClassifier`: `cv2.CascadeClassifier()`
     - Capture video: `cv2.VideoCapture()`
     - Convert frame to grayscale: `cv2.cvtColor()`
     - Display output: `cv2.imshow()`
     - Release resources: `cap.release()`, `cv2.destroyAllWindows()`
2. **NumPy**:
   - **Definition**: A numerical computing library for Python.
   - **Reason for Use**: While imported, NumPy is not actively used in this specific script.
3. **dlib**:
   - **Definition**: A toolkit for machine learning and computer vision.
   - **Reason for Use**: It is imported but not utilized in the code provided.

7.1.7.7. How the Code Works

The script captures frames from the webcam and converts each to grayscale. The `detect` function identifies faces and eyes within the frame using Haar Cascade classifiers and draws

rectangles around detected features. The processed frames are displayed in real time until the user terminates the application by pressing 'q'.

7.1.7.8. Final Report Summary

eye_detection_app.py is a simple yet effective Python application for detecting faces and eyes in real-time. Using OpenCV's pre-trained Haar Cascades, it captures and processes frames from a webcam, highlighting detected features. This code showcases OpenCV's powerful image processing capabilities, making it a valuable tool for facial feature detection tasks in real-time applications.

7.1.2 Headphone Detection Model

Purpose: Detects whether a student is wearing headphones or earbuds, as they might be using them to receive unauthorized assistance.

Technology Used: A **Convolutional Neural Network (CNN)** trained on extensive datasets of people with and without headphones. Developed using **Keras** and **TensorFlow**, the model achieves high precision in detecting headphone presence.

Workflow:

1. **Image Capture**: The system periodically captures images during the exam.
2. **Shape Analysis**: CNN processes the images, identifying headphone-like shapes around the student's ears.
3. **Alert Generation**: If headphones are detected, the system flags the student for review.

Challenges:

- **Object Differentiation**: Differentiating between headphones and other accessories like hats or glasses was challenging. The model was further trained with labeled images of common accessories to reduce false positives.

Privacy Considerations:

- **Real-Time Processing**: Images are processed instantly and deleted immediately after analysis, maintaining student privacy.

CODE :

```
# -*- coding: utf-8 -*-
"""Headphones_Detection.ipynb

Automatically generated by Colab.

Original file is located at

    https://colab.research.google.com/drive/18JY5bmPTpK7b71MTgSm94wLhDHMjcWsc
"""

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPool2D,Dropout, Flatten, Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from google.colab.patches import cv2_imshow
from base64 import b64decode, b64encode
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
import dlib

!git clone https://github.com/madhavc9/headphone_dataset_proctorly

import os

main_dir = "/content/headphone_dataset_proctorly"

# SETTING TRAIN AND TEST DIRECTORY
train_dir = os.path.join(main_dir, "Train")
test_dir = os.path.join(main_dir, "Test")

#SETING DIRECTORY FOR HEADPHONES AND WITHOUT HEADPHONES IMAGES DIRECTORY
train_head_dir = os.path.join(train_dir, "With_Headphones")
train_normal_dir = os.path.join(train_dir, "Without_Headphones")

test_head_dir = os.path.join(test_dir, "With_Headphones")
test_normal_dir = os.path.join(test_dir, "Without_Headphones")

train_head_names = os.listdir(train_head_dir)
train_normal_names = os.listdir(train_normal_dir)

```

```
test_head_names = os.listdir(test_head_dir)
test_normal_names = os.listdir(test_normal_dir)

import matplotlib.image as mpimg

rows = 4
columns = 4

fig = plt.gcf()
fig.set_size_inches(12,12)

head_img = [os.path.join(train_head_dir, filename) for filename in
train_head_names[0:8]]
normal_img = [os.path.join(train_normal_dir, filename) for filename in
train_normal_names[0:8]]

print(head_img)
print(normal_img)

merged_img = head_img + normal_img

for i, img_path in enumerate(merged_img):
  title = img_path.split("/", 6)[4]
  plot = plt.subplot(rows, columns, i+1)
  plot.axis("Off")
  img = mpimg.imread(img_path)
  plot.set_title(title, fontsize = 11)
  plt.imshow(img, cmap= "gray")

plt.show()

# CREATING TRAINING, TESTING AND VALIDATION BATCHES

dgen_train = ImageDataGenerator(rescale = 1./255,
                                validation_split = 0.1,
                                zoom_range = 0.2,
                                horizontal_flip = True)

dgen_validation = ImageDataGenerator(rescale = 1./255,
                                     )

dgen_test = ImageDataGenerator(rescale = 1./255,
                               )

train_generator = dgen_train.flow_from_directory(train_dir,
                                                 target_size = (150, 150),
                                                 subset = 'training',
                                                 batch_size = 32,
                                                 class_mode = 'binary')
validation_generator = dgen_train.flow_from_directory(train_dir,
```

```
                                                target_size   =   (150,
150),
                                                subset = "validation",
                                                batch_size = 32,
                                                class_mode = "binary")
test_generator = dgen_test.flow_from_directory(test_dir,
                                               target_size = (150, 150),
                                               batch_size = 32,
                                               class_mode = "binary")

print("Class Labels are: ", train_generator.class_indices)
print("Image shape is : ", train_generator.image_shape)

from tensorflow.keras.layers import Conv2D, MaxPooling2D

model = Sequential()

# 1) CONVOLUTIONAL LAYER - 1
model.add(Conv2D(32, (5,5), padding = "same", activation = "relu", input_shape
= train_generator.image_shape))

# 2) POOLING LAYER - 1
model.add(MaxPooling2D(pool_size=(2,2)))

# 3) DROPOUT LAYER -2
model.add(Dropout(0.5))

# 4) CONVOLUTIONAL LAYER - 2
model.add(Conv2D(64, (5,5), padding = "same", activation = "relu"))

# 5) POOLING LAYER - 2
model.add(MaxPooling2D(pool_size=(2,2)))

# 6) DROPOUT LAYER - 2
model.add(Dropout(0.5))

# 7) FLATTENING LAYER TO 2D SHAPE
model.add(Flatten())

# 8) ADDING A DENSE LAYER
model.add(Dense(256, activation = 'relu'))

# 9 DROPOUT LAYER - 3
model.add(Dropout(0.5))

# 10) FINAL OUTPUT LAYER
model.add(Dense(1, activation = 'sigmoid'))

### PRINTING MODEL SUMMARY
model.summary()

```

```
# COMPILING THE MODEL

model.compile(Adam(learning_rate = 0.001), loss = 'binary_crossentropy',
metrics = ['accuracy'])

# TRAINING THE MODEL
history = model.fit(train_generator,
                    epochs = 35,
                    validation_data = validation_generator)

# PLOT GRAPH BETWEEN TRAINING AND VALIDATION LOSS
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['Training', 'Validation'])
plt.title("Training and validation losses")
plt.xlabel('epoch')

# PLOT GRAPH BETWEEN TRAINING AND VALIDATION ACCURACY
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['Training', 'Validation'])
plt.title("Training and validation accuracy")
plt.xlabel('epoch')

# GETTING TEST ACCURACY AND LOSS

test_loss, test_acc = model.evaluate(test_generator)
print("Test Set Loss : ", test_loss)
print("Test Set Accuracy : ", test_acc)

model.save("Pretrained_model.h5")

from google.colab import files
from keras.preprocessing import image

uploaded = files.upload()

for filename in uploaded.keys():
  img_path = '/content/' + filename
  img = image.load_img(img_path, target_size = (150,150))
  images = image.img_to_array(img)
  images = np.expand_dims(images, axis = 0)
  prediction = model.predict(images)

  if prediction == 0:
    print("HEADPHONES DETECTED")
  else:
    print("NO HEADPHONES DETECTED")

from google.colab import files
from keras.preprocessing import image
```

```
•
•  def video_stream():
•    js = Javascript('''
•      var video;
•      var div = null;
•      var stream;
•      var captureCanvas;
•      var imgElement;
•      var labelElement;
•
•      var pendingResolve = null;
•      var shutdown = false;
•
•      function removeDom() {
•         stream.getVideoTracks()[0].stop();
•         video.remove();
•         div.remove();
•         video = null;
•         div = null;
•         stream = null;
•         imgElement = null;
•         captureCanvas = null;
•         labelElement = null;
•      }
•
•      function onAnimationFrame() {
•        if (!shutdown) {
•          window.requestAnimationFrame(onAnimationFrame);
•        }
•        if (pendingResolve) {
•          var result = "";
•          if (!shutdown) {
•            captureCanvas.getContext('2d').drawImage(video, 0, 0, 640, 480);
•            result = captureCanvas.toDataURL('image/jpeg', 0.8)
•          }
•          var lp = pendingResolve;
•          pendingResolve = null;
•          lp(result);
•        }
•      }
•
•      async function createDom() {
•        if (div !== null) {
•          return stream;
•        }
•
•        div = document.createElement('div');
•        div.style.border = '2px solid black';
•        div.style.padding = '3px';
•        div.style.width = '100%';
•        div.style.maxWidth = '600px';
```

```
        document.body.appendChild(div);

        const modelOut = document.createElement('div');
        modelOut.innerHTML = "<span>Status:</span>";
        labelElement = document.createElement('span');
        labelElement.innerText = 'No data';
        labelElement.style.fontWeight = 'bold';
        modelOut.appendChild(labelElement);
        div.appendChild(modelOut);

        video = document.createElement('video');
        video.style.display = 'block';
        video.width = div.clientWidth - 6;
        video.setAttribute('playsinline', '');
        video.onclick = () => { shutdown = true; };
        stream = await navigator.mediaDevices.getUserMedia(
            {video: { facingMode: "environment"}});
        div.appendChild(video);

        imgElement = document.createElement('img');
        imgElement.style.position = 'absolute';
        imgElement.style.zIndex = 1;
        imgElement.onclick = () => { shutdown = true; };
        div.appendChild(imgElement);

        const instruction = document.createElement('div');
        instruction.innerHTML =
            '<span style="color: red; font-weight: bold;">' +
            'When finished, click here or on the video to stop this demo</span>';
        div.appendChild(instruction);
        instruction.onclick = () => { shutdown = true; };

        video.srcObject = stream;
        await video.play();

        captureCanvas = document.createElement('canvas');
        captureCanvas.width = 640; //video.videoWidth;
        captureCanvas.height = 480; //video.videoHeight;
        window.requestAnimationFrame(onAnimationFrame);

        return stream;
      }
      async function stream_frame(label, imgData) {
        if (shutdown) {
          removeDom();
          shutdown = false;
          return '';
        }

        var preCreate = Date.now();
        stream = await createDom();
```

```
        var preShow = Date.now();
        if (label != "") {
          labelElement.innerHTML = label;
        }

        if (imgData != "") {
          var videoRect = video.getClientRects()[0];
          imgElement.style.top = videoRect.top + "px";
          imgElement.style.left = videoRect.left + "px";
          imgElement.style.width = videoRect.width + "px";
          imgElement.style.height = videoRect.height + "px";
          imgElement.src = imgData;
        }

        var preCapture = Date.now();
        var result = await new Promise(function(resolve, reject) {
          pendingResolve = resolve;
        });
        shutdown = false;

        return {'create': preShow - preCreate,
                'show': preCapture - preShow,
                'capture': Date.now() - preCapture,
                'img': result};
      }
      ''')

    display(js)

  def video_frame(label, bbox):
    data = eval_js('stream_frame("{}", "{}")'.format(label, bbox))
    return data




  def js_to_image(js_reply):
    """
    Params:
            js_reply: JavaScript object containing image from webcam
    Returns:
            img: OpenCV BGR image
    """
    # decode base64 image
    image_bytes = b64decode(js_reply.split(',')[1])
    # convert bytes to numpy array
    jpg_as_np = np.frombuffer(image_bytes, dtype=np.uint8)
    # decode numpy array into OpenCV BGR image
    img = cv2.imdecode(jpg_as_np, flags=1)
```

```
    return img

cap = cv2.VideoCapture(0)
video_stream()

label_html = "Detecting ..."
bbox = ''
count = 0
detector = dlib.get_frontal_face_detector()

i = 0

while i>=0:

  js_reply = video_frame(label_html, bbox)
  if not js_reply:
      break

  frame = js_to_image(js_reply["img"])

  gray1 = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
  cv2.imwrite('temp.jpg', gray1)
  img = cv2.imread('temp.jpg')

  # Convert into grayscale
  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

  # Load the cascade
  face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_alt2.xml')

  # Detect faces
  faces = face_cascade.detectMultiScale(gray, 1.1, 4)

  # Draw rectangle around the faces and crop the faces
  for (x, y, w, h) in faces:
    cv2.rectangle(img, (x, y), (x+w, y+h), (0, 0, 255), 2)
    faces = img[y:y + h, x:x + w]

    cv2.imwrite('face.jpg', faces)




  if i%2!=0:
    cv2.putText(frame,      "HEADPHONES      DETECTED",      (100,      53),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255),
            2)
  else:
    cv2.putText(frame,    "NO    HEADPHONES    DETECTED",    (100,    53),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255),
```

```
            2)


    cv2_imshow(frame)
    i+=1

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

  cv2.destroyAllWindows()
```

7.1.2.1. Step-by-Step Code Explanation

Step-1 : Importing Libraries :
This part of the code imports necessary libraries and packages for image processing, machine learning, and plotting, including TensorFlow for deep learning model building, and OpenCV and dlib for computer vision tasks.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dropout, Flatten, Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from google.colab.patches import cv2_imshow
import cv2
import dlib
```

- **Explanation**:
    - **TensorFlow and Keras**: Used for creating and training a neural network model to detect headphones.
    - **OpenCV (cv2)**: Used for image manipulation and displaying images in Colab.
    - **dlib**: Provides a frontal face detector which is later used in the video stream detection process.
    - **IPython/Javascript**: Used to integrate JavaScript for accessing webcam feed in Google Colab.

Step -2 :Dataset Setup
Clones a dataset repository and sets paths for training and testing directories.

```
!git clone https://github.com/RishitToteja/Headphones_Images_Dataset
import os

main_dir = "/content/Headphones_Dataset"
train_dir = os.path.join(main_dir, "Train")
test_dir = os.path.join(main_dir, "Test")
train_head_dir = os.path.join(train_dir, "With_Headphones")
train_normal_dir = os.path.join(train_dir, "Without_Headphones")
test_head_dir = os.path.join(test_dir, "With_Headphones")
test_normal_dir = os.path.join(test_dir, "Without_Headphones")
```

- **Explanation**:
  - The code clones a GitHub repository containing a dataset of images, separated into folders of "With_Headphones" and "Without_Headphones".
  - Sets paths for train and test data directories for ease of access later in the code.

Step 3 : Image Display
Displays a subset of images from both the headphone and non-headphone categories for verification.

```
import matplotlib.image as mpimg

rows = 4
columns = 4

fig = plt.gcf()
fig.set_size_inches(12,12)

head_img = [os.path.join(train_head_dir, filename) for filename in train_head_names[0:8]]
normal_img = [os.path.join(train_normal_dir, filename) for filename in train_normal_names[0:8]]

merged_img = head_img + normal_img

for i, img_path in enumerate(merged_img):
  title = img_path.split("/", 6)[4]
  plot = plt.subplot(rows, columns, i+1)
  plot.axis("Off")
  img = mpimg.imread(img_path)
  plot.set_title(title, fontsize=11)
  plt.imshow(img, cmap="gray")
plt.show()
```

- **Explanation**:

- Loads and displays sample images from the training dataset to verify the setup visually.

Step 4 :Data Generators for Training, Validation, and Testing
Creates training, validation, and test generators for feeding the neural network model.

```
dgen_train = ImageDataGenerator(rescale=1./255, validation_split=0.2, zoom_range=0.2, horizontal_flip=True)
dgen_validation = ImageDataGenerator(rescale=1./255)
dgen_test = ImageDataGenerator(rescale=1./255)

train_generator = dgen_train.flow_from_directory(train_dir, target_size=(150, 150), subset='training', batch_size=32, class_mode='binary')
validation_generator = dgen_train.flow_from_directory(train_dir, target_size=(150, 150), subset="validation", batch_size=32, class_mode="binary")
test_generator = dgen_test.flow_from_directory(test_dir, target_size=(150, 150), batch_size=32, class_mode="binary")
```

- **Explanation**:
  - Defines image data generators to preprocess images (rescaling and augmenting) for training, validation, and testing.
  - **Data augmentation** in the training set helps improve model generalization by introducing variations.

Step 5 :Building the Convolutional Neural Network (CNN) Model
Constructs a CNN model with convolutional, pooling, dropout, and dense layers.

```
model = Sequential()

model.add(Conv2D(32, (5,5), padding="same", activation="relu", input_shape=train_generator.image_shape))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.5))

model.add(Conv2D(64, (5,5), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

- **Explanation**:

- A Sequential CNN model with convolutional layers (for feature extraction), pooling layers (for down-sampling), dropout layers (to prevent overfitting), and dense layers (for final prediction).
- Sigmoid activation in the output layer is used for binary classification (headphones vs. no headphones).

Step 6 : Model Compilation and Training
Compiles and trains the CNN model on the dataset.

```
model.compile(Adam(learning_rate=0.001),              loss='binary_crossentropy',
metrics=['accuracy'])
history            =            model.fit(train_generator,            epochs=35,
validation_data=validation_generator)
```

- **Explanation**:
  - The model is compiled with the Adam optimizer and binary cross-entropy loss, suitable for binary classification.
  - Model is trained for 35 epochs using training and validation data, and the training process is recorded in `history` for plotting.

Step 7 ; Plotting Training and Validation Loss/Accuracy
Generates plots to visualize the training and validation accuracy and loss over epochs.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['Training', 'Validation'])
plt.title("Training and validation losses")
plt.xlabel('epoch')

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['Training', 'Validation'])
plt.title("Training and validation accuracy")
plt.xlabel('epoch')
```

- **Explanation**:
  - Plots are used to observe model performance and check for signs of overfitting or underfitting.

7.1.2.2. File Summary

This code is designed to create, train, and evaluate a deep learning model to detect if a person in an image is wearing headphones. It uses a Convolutional Neural Network (CNN) for image classification, leveraging TensorFlow for model architecture and training and OpenCV for image processing. After training, the model is tested with real-time webcam video input to detect headphones.

7.1.2.3. Key Code Highlights

- **Data Augmentation**: Used to make the model more robust by artificially expanding the training set.
- **Custom CNN Model**: Includes layers for convolution, pooling, and dropout to create a powerful classifier.
- **Real-Time Detection**: Integrates webcam input and OpenCV, using dlib for real-time face detection with headphone recognition.

7.1.2.4. Frameworks Used

- **TensorFlow/Keras**:
  - For building and training the CNN model.
  - Code: `from tensorflow.keras.models import Sequential`
- **OpenCV**:
  - For image display and real-time webcam feed processing.
  - Code: `import cv2`
- **dlib**:
  - For frontal face detection in real-time webcam feed.
  - Code: `detector = dlib.get_frontal_face_detector()`

7.1.2.5. How the Code Works

- Clones the dataset, prepares directories, and verifies image loading.
- Builds a CNN model to classify images with/without headphones.
- Trains the model and evaluates performance on test data.
- Integrates a webcam stream to detect headphones on faces in real-time.

7.1.3 Malicious Object Detection Model

Purpose: Detects unauthorized objects, such as mobile phones, books, or calculators, that students might use to cheat.

Technology Used: Utilizes **YOLOv4** (You Only Look Once, Version 4) on the **Darknet** framework for real-time object detection.

Workflow:

1. **Image Grid Analysis**: YOLOv4 divides each image into a grid, checking for objects within each cell.

2. **Object Classification**: Objects within the frame are identified and categorized as permissible or non-permissible based on the exam's guidelines.
3. **Real-Time Flagging**: If unauthorized items are detected, the system logs the incident and alerts the proctor.

Challenges:

- **Computational Demands**: High processing power is needed to detect smaller objects in real-time. Model pruning and quantization were applied to reduce computational load and improve speed.

Accuracy:

- **Dataset Diversity**: Trained on a large dataset featuring various objects from multiple angles, the model achieves a high detection accuracy for commonly used cheating tools.

CODE :

```
# -*- coding: utf-8 -*-
"""malicious_object_yolov5.ipynb

Automatically generated by Colab.

Original file is located at
```

```
https://colab.research.google.com/drive/1oun7i3B8l2Bc8Bip27nHpMt9iX5FQ8ss
"""

# Commented out IPython magic to ensure Python compatibility.
!git clone https://github.com/ultralytics/yolov5  # Clone YOLOv5 repository
# %cd yolov5
!pip install -r requirements.txt  # Install requirements

import torch
import utils
display = utils.notebook_init()

import torch
from PIL import Image
import matplotlib.pyplot as plt
import os
from google.colab import files  # Import files module if using Google Colab

# Upload the image file
uploaded = files.upload()  # Opens a file upload dialog

if uploaded:
    # Retrieve the uploaded file's name
    img_path = list(uploaded.keys())[0]

    # Load YOLOv5 model
    model = torch.hub.load('ultralytics/yolov5', 'custom', path='yolov5s.pt', force_reload=True)

    # Set confidence threshold and image size directly on the model
    model.conf = 0.25  # Set confidence threshold
    model.iou = 0.45  # Set IoU threshold (optional)
    model.max_det = 1000  # Maximum number of detections (optional)

    # Perform detection and save results in 'runs/detect/exp' directory
    results = model(img_path)
    results.save()  # This will save the output image with rectangles in 'runs/detect/exp'

    # Find the saved output image file in the 'runs/detect/exp' directory
    output_dir = '/content/yolov5/runs/detect/exp'
    output_files = os.listdir(output_dir)

    # Check if any file was generated in the output directory
    if output_files:
        output_file = os.path.join(output_dir, output_files[0])  # Assumes only one image in 'exp' folder

        # Display the output image with bounding boxes using matplotlib
        img = Image.open(output_file)
```

```
            plt.imshow(img)
            plt.axis('off')
            plt.show()
        else:
            print("No output image found in the 'runs/detect/exp' directory.")
    else:
        print("No file uploaded.")

    display.Image(filename='/content/yolov5/runs/detect/exp2/'+        img_path,
    width=600)
```

7.1.3.1. Code Analysis and Report

Step 1 : Setup and Environment Initialization

Cloning the YOLOv5 Repository and Installing Dependencies

```
!git clone https://github.com/ultralytics/yolov5  # Clone YOLOv5 repository
!pip install -r requirements.txt  # Install requirements
```

- **Explanation**:
  - **Purpose**: Sets up the environment by downloading the YOLOv5 codebase and installing required libraries.
  - **Relevance**: Ensures the script has access to the YOLOv5 object detection framework and its dependencies.

Step 2: Importing Required Libraries

```
import torch
import utils
display = utils.notebook_init()
```

- **Explanation**:
  - **`torch`**: Used for deep learning computations and as the backend for YOLOv5.
  - **`utils` and `notebook_init()`**: Utility functions for initializing notebooks with visualization tools. This setup is for monitoring the detection results.

Step 3: Uploading and Handling Input Image

Uploading Image via Google Colab

```
from google.colab import files  # Import files module if using Google Colab

# Upload the image file
uploaded = files.upload()  # Opens a file upload dialog
```

- **Explanation**:
  - **Purpose**: Allows users to upload an image file in a Colab environment for detection.
  - **Key Feature**: Uses `files.upload()` to open a file upload dialog and stores the uploaded image in the `uploaded` dictionary.

Step 4 : Retrieving the Uploaded File

```
if uploaded:
    # Retrieve the uploaded file's name
    img_path = list(uploaded.keys())[0]
```

- **Explanation**:
  - Extracts the name of the uploaded file from the dictionary. This filename will be used later for processing.

Step 5 : Loading YOLOv5 Model

Loading Pretrained YOLOv5 Model

```
model = torch.hub.load('ultralytics/yolov5', 'custom', path='yolov5s.pt', force_reload=True)
```

- **Explanation**:
  - Uses the torch.hub module to load the YOLOv5 model pretrained on the COCO dataset.
  - The yolov5s.pt file refers to the smallest version of YOLOv5, optimized for speed and lightweight applications.
  - **force_reload=True**: Ensures the model is reloaded even if it's already available in the local cache.

Step 6 : Configuring Model Parameters

```
model.conf = 0.25  # Set confidence threshold
model.iou = 0.45  # Set IoU threshold (optional)
model.max_det = 1000  # Maximum number of detections (optional)
```

- **Explanation**:
    - **Confidence Threshold (`conf`)**: Specifies the minimum confidence for detections to be considered valid.
    - **IoU Threshold (`iou`)**: Determines the overlap threshold for non-max suppression.
    - **Maximum Detections (`max_det`)**: Limits the number of objects detected in an image to avoid processing overhead.

Step 7 : Running Detection and Saving Results

Performing Detection

```
results = model(img_path)
results.save()
```

- **Explanation**:
    - **Detection**: The YOLOv5 model processes the uploaded image (`img_path`) and generates bounding boxes, labels, and confidence scores.
    - **Saving Results**: Saves the output with bounding boxes to the `runs/detect/exp` directory.

Certainly! Below is the revised report with **relevant section titles** before each code chunk for clarity.

7.1.3.2. Summary of the Code

Key Code Highlights

1. **YOLOv5 Integration**:
    - Uses a pre-trained YOLOv5 model for object detection.
    - Configures key detection parameters like confidence and IoU thresholds.
2. **Image Handling**:
    - Uploads an image using Google Colab's file upload feature.
    - Automatically detects malicious objects or persons and overlays bounding boxes on the image.
3. **Result Visualization**:
    - Saves the processed image to a specific directory and displays it using either matplotlib or IPython.display.

7.1.3.3. Frameworks Used

1. PyTorch

- **Definition**: A deep learning framework providing tensor computation and automatic differentiation.
- **Usage in Code**:

```
import torch
model = torch.hub.load('ultralytics/yolov5', 'custom', path='yolov5s.pt', force_reload=True)
```

- **Reason for Use**:
    - Powers the YOLOv5 model with efficient computations.
    - Simplifies loading of pre-trained models through `torch.hub`.

2. Google Colab Files Module

- **Definition**: A utility for file operations in Google Colab.
- **Usage in Code**:

```
from google.colab import files
uploaded = files.upload()
```

Reason for Use: Enables easy file uploads in the Colab environment, making the tool accessible for non-technical users.

3. Matplotlib

- **Definition**: A plotting library for Python.
- **Usage in Code**

```
import matplotlib.pyplot as plt
plt.imshow(img)
plt.axis('off')
plt.show()
```

Reason for Use: Visualizes the output image with bounding boxes in an interactive way.

7.1.3.4. How the Code Works

1. Sets up the YOLOv5 framework by cloning its repository and installing dependencies.
2. Imports necessary libraries and initializes the Colab environment.
3. Allows users to upload an image for analysis.
4. Loads the YOLOv5 model and configures detection thresholds.
5. Processes the image to detect objects and saves the result.
6. Displays the output image with bounding boxes.

7.1.3.5. Final Summary

The provided code integrates the YOLOv5 framework into a Google Colab environment for detecting malicious objects or persons. It automates the process from uploading an image to generating and displaying the detection results. The use of YOLOv5 ensures high-speed and accurate object detection, while Google Colab facilitates accessibility and ease of use. The modular design of the script allows it to be adapted for various object detection tasks with minimal changes.

7.1.4 Mouth Movement Tracking Model

Purpose: Detects when a student is speaking, potentially indicating they are communicating with someone during the exam.

Technology Used: Combines **Dlib** for facial landmark detection with an **LSTM (Long Short-Term Memory)** model to identify speech-related movements.

Workflow:

1. **Mouth Region Tracking**: Dlib identifies facial landmarks around the mouth area.
2. **Movement Pattern Analysis**: The LSTM model monitors mouth movements over time to detect patterns consistent with speaking.
3. **Alert for Frequent Speech**: Alerts are generated if the model detects repeated speech-like movements.

Challenges:

- **Distinguishing Non-Verbal Mouth Movements**: Differentiating between speaking and natural facial movements like yawning required precise tuning.
- **Accuracy**: Trained on diverse data, the model reliably distinguishes speaking from other movements, minimizing false alerts.

Code :

```
# -*- coding: utf-8 -*-
```

```
"""Mouth_Detection.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/15t2W5lVBXTNRAo0y54UIHa9yG6FSGFYF
"""

# IMPORTING NECESSARY LIBRARIES AND FILES

from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from google.colab.patches import cv2_imshow
from base64 import b64decode, b64encode
import cv2
import numpy as np
from imutils.video import VideoStream
from imutils import face_utils
import numpy as np
import imutils
import time
import dlib
import os
import cv2
from google.colab.patches import cv2_imshow

!wget      http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2  # DOWNLOAD LINK

!bunzip2 /content/shape_predictor_68_face_landmarks.dat.bz2

def video_stream():
  js = Javascript('''
    var video;
    var div = null;
    var stream;
    var captureCanvas;
    var imgElement;
    var labelElement;
    var pendingResolve = null;
    var shutdown = false;

    function removeDom() {
       stream.getVideoTracks()[0].stop();
       video.remove();
       div.remove();
       video = null;
       div = null;
       stream = null;
       imgElement = null;
```

```
        captureCanvas = null;
        labelElement = null;
      }

      function onAnimationFrame() {
        if (!shutdown) {
          window.requestAnimationFrame(onAnimationFrame);
        }
        if (pendingResolve) {
          var result = "";
          if (!shutdown) {
            captureCanvas.getContext('2d').drawImage(video, 0, 0, 640, 480);
            result = captureCanvas.toDataURL('image/jpeg', 0.8)
          }
          var lp = pendingResolve;
          pendingResolve = null;
          lp(result);
        }
      }

      async function createDom() {
        if (div !== null) {
          return stream;
        }

        div = document.createElement('div');
        div.style.border = '2px solid black';
        div.style.padding = '3px';
        div.style.width = '100%';
        div.style.maxWidth = '600px';
        document.body.appendChild(div);

        const modelOut = document.createElement('div');
        modelOut.innerHTML = "<span>Status:</span>";
        labelElement = document.createElement('span');
        labelElement.innerText = 'No data';
        labelElement.style.fontWeight = 'bold';
        modelOut.appendChild(labelElement);
        div.appendChild(modelOut);

        video = document.createElement('video');
        video.style.display = 'block';
        video.width = div.clientWidth - 6;
        video.setAttribute('playsinline', '');
        video.onclick = () => { shutdown = true; };
        stream = await navigator.mediaDevices.getUserMedia(
            {video: { facingMode: "environment"}});
        div.appendChild(video);

        imgElement = document.createElement('img');
        imgElement.style.position = 'absolute';
```

```
        imgElement.style.zIndex = 1;
        imgElement.onclick = () => { shutdown = true; };
        div.appendChild(imgElement);

        const instruction = document.createElement('div');
        instruction.innerHTML =
            '<span style="color: red; font-weight: bold;">' +
            'When finished, click here or on the video to stop this demo</span>';
        div.appendChild(instruction);
        instruction.onclick = () => { shutdown = true; };

        video.srcObject = stream;
        await video.play();

        captureCanvas = document.createElement('canvas');
        captureCanvas.width = 640; //video.videoWidth;
        captureCanvas.height = 480; //video.videoHeight;
        window.requestAnimationFrame(onAnimationFrame);

        return stream;
      }
      async function stream_frame(label, imgData) {
        if (shutdown) {
          removeDom();
          shutdown = false;
          return '';
        }

        var preCreate = Date.now();
        stream = await createDom();

        var preShow = Date.now();
        if (label != "") {
          labelElement.innerHTML = label;
        }

        if (imgData != "") {
          var videoRect = video.getClientRects()[0];
          imgElement.style.top = videoRect.top + "px";
          imgElement.style.left = videoRect.left + "px";
          imgElement.style.width = videoRect.width + "px";
          imgElement.style.height = videoRect.height + "px";
          imgElement.src = imgData;
        }

        var preCapture = Date.now();
        var result = await new Promise(function(resolve, reject) {
          pendingResolve = resolve;
        });
        shutdown = false;

```

```
        return {'create': preShow - preCreate,
                'show': preCapture - preShow,
                'capture': Date.now() - preCapture,
                'img': result};
      }
      ''')

  display(js)

def video_frame(label, bbox):
  data = eval_js('stream_frame("{}", "{}")'.format(label, bbox))
  return data
def js_to_image(js_reply):
  image_bytes = b64decode(js_reply.split(',')[1])
  jpg_as_np = np.frombuffer(image_bytes, dtype=np.uint8)
  img = cv2.imdecode(jpg_as_np, flags=1)
  return img

# Loading Classifiers
face_detect = cv2.CascadeClassifier("face_detect.xml")
landmark_detect                                                    =
dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")


# Capturing live video stream with help of Javascript Helper Function

cap = cv2.VideoCapture(0)
video_stream()
label_html = "Detecting ..."
bbox = ''
count = 0
detector = dlib.get_frontal_face_detector()

while True:
  js_reply = video_frame(label_html, bbox)
  if not js_reply:
    break

  # Reading image from video stream
  frame = js_to_image(js_reply["img"])

  frame = imutils.resize(frame, width=600)

  gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

  # Detecting the features
  faces = face_detect.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5,
minSize=(100, 100), flags=cv2.CASCADE_SCALE_IMAGE)

  for (x, y, w, h) in faces:

```

```
      rect = dlib.rectangle(int(x), int(y), int(x + w),
        int(y + h))
      landmark = landmark_detect(gray, rect)
      landmark = face_utils.shape_to_np(landmark)

      (mStart, mEnd) = face_utils.FACIAL_LANDMARKS_IDXS["mouth"]

      mouth = landmark[mStart:mEnd]
      # If mouth was detected, we will get the the x,y coordinates of the
    detected feature
      boundRect = cv2.boundingRect(mouth)
      cv2.rectangle(frame,
              (int(boundRect[0]), int(boundRect[1])),
              (int(boundRect[0]   +   boundRect[2]),      int(boundRect[1]   +
    boundRect[3])), (0,0,255), 2)
      hsv     =     cv2.cvtColor(frame[int(boundRect[1]):int(boundRect[1]     +
    boundRect[3]),int(boundRect[0]):int(boundRect[0]      +      boundRect[2])],
    cv2.COLOR_RGB2HSV)
      sum_saturation = np.sum(hsv[:, :, 1])
      area = int(boundRect[2])*int(boundRect[3])
      avg_saturation = sum_saturation / area
      if               abs(int(boundRect[1])-int(boundRect[1]               +
    boundRect[3]))>=abs((int(boundRect[0] + boundRect[2])-int(boundRect[0]))/2):
        # Checking whether mouth is open or not
        cv2.putText(frame, "Mouth is Open", (155, 53), cv2.FONT_HERSHEY_SIMPLEX,
    1.5, (0, 0, 255),
              2)

    cv2_imshow(frame)
    key = cv2.waitKey(1) & 0xFF
    if key == 27:
      break

  cv2.destroyAllWindows()
  vs.stop()
```

7.1.4.1. Code Chunk Explanations

Section 1: Importing Libraries and External Files

```
from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from google.colab.patches import cv2_imshow
from base64 import b64decode, b64encode
import cv2
import numpy as np
from imutils.video import VideoStream
from imutils import face_utils
import numpy as np
import imutils
import time
import dlib
import os
import cv2
from google.colab.patches import cv2_imshow
```

- **Explanation**: This section imports all required libraries and modules:
  - `IPython.display`, `eval_js`, `cv2_imshow`: Provide methods for displaying images and running JavaScript in Colab.
  - `cv2` (OpenCV): A primary library for computer vision, crucial here for face and mouth detection.
  - `numpy`: Used for array manipulation and mathematical functions.
  - `imutils`: Facilitates image resizing, improving processing efficiency.
  - `dlib`: A library for machine learning, providing key facial landmark detection.
  - Other miscellaneous imports support JavaScript and image handling for Colab.

Section 2: Downloading Pre-trained Model

```
!wget http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2
!bunzip2 /content/shape_predictor_68_face_landmarks.dat.bz2
```

Explanation: Downloads and decompresses the shape_predictor_68_face_landmarks.dat file, which provides 68 facial landmarks essential for identifying key points on the face, including the mouth region. This file is used by the dlib library to enhance face feature detection accuracy.

Section 3: JavaScript Video Stream Function for Colab

```
def video_stream():
    js = Javascript('''
      // JavaScript function code goes here
    ''')
    display(js)
```

Explanation: This function injects JavaScript code to create a live video stream in Google Colab. The JavaScript code manages the display and capture of live video frames. The `display(js)` call runs this embedded JavaScript to initialize the webcam stream.

Section 4: Helper Functions for Video Frame Processing

```
def video_frame(label, bbox):
    data = eval_js('stream_frame("{}", "{}")'.format(label, bbox))
    return data

def js_to_image(js_reply):
    image_bytes = b64decode(js_reply.split(',')[1])
    jpg_as_np = np.frombuffer(image_bytes, dtype=np.uint8)
    img = cv2.imdecode(jpg_as_np, flags=1)
    return img
```

- **Explanation**: Defines two utility functions:
  - `video_frame`: Passes data from the JavaScript video stream to Python.
  - `js_to_image`: Converts JavaScript image data to an OpenCV-compatible format by decoding the image and reshaping it.

Section 5: Initializing Detection Models

```
face_detect = cv2.CascadeClassifier("face_detect.xml")
landmark_detect = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
```

- **Explanation**: Initializes the face and landmark detection models:
  - `CascadeClassifier` from OpenCV, typically used for basic face detection.
  - `dlib.shape_predictor` loads the 68-point landmark model, used to locate mouth landmarks specifically.

Section 6: Main Loop for Mouth Detection

```
cap = cv2.VideoCapture(0)
video_stream()
label_html = "Detecting ..."
bbox = ''
count = 0
detector = dlib.get_frontal_face_detector()
```

- **Explanation**: Sets up the main video capture environment:
  - Initializes the webcam.
  - Calls `video_stream()` to start the live video stream.
  - Defines variables for labels and bounding boxes.
  - Loads `dlib.get_frontal_face_detector()` to detect faces in each frame.

Section 7: Frame-by-Frame Analysis for Mouth Detection

```
while True:
    js_reply = video_frame(label_html, bbox)
    if not js_reply:
        break
    frame = js_to_image(js_reply["img"])
    frame = imutils.resize(frame, width=600)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

- **Explanation**: Captures and processes each video frame:
  - `video_frame` grabs a single frame from the live stream.
  - Converts the frame from JavaScript format to OpenCV format.
  - Resizes and converts the frame to grayscale to prepare for face detection.

Section 8: Detecting Face and Mouth Landmarks

```
faces = face_detect.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5,
minSize=(100, 100), flags=cv2.CASCADE_SCALE_IMAGE)

for (x, y, w, h) in faces:
    rect = dlib.rectangle(int(x), int(y), int(x + w), int(y + h))
    landmark = landmark_detect(gray, rect)
    landmark = face_utils.shape_to_np(landmark)
```

- **Explanation**: Detects the face in each frame, using the landmarks for identifying the mouth.
  - `detectMultiScale`: Finds potential faces in the grayscale frame.

- For each detected face, a `rectangle` defines its boundaries.
- `shape_to_np` converts the dlib landmark data into numpy format, making it easier to access specific landmarks.

Section 9: Detecting Mouth Position and Opening

```
(mStart, mEnd) = face_utils.FACIAL_LANDMARKS_IDXS["mouth"]
mouth = landmark[mStart:mEnd]
boundRect = cv2.boundingRect(mouth)
cv2.rectangle(frame,  (int(boundRect[0]),  int(boundRect[1])),  (int(boundRect[0]  +
boundRect[2]), int(boundRect[1] + boundRect[3])), (0,0,255), 2)
```

- **Explanation**: This part isolates and highlights the mouth region.
  - Finds the `mouth` landmark points based on indices.
  - Draws a rectangle around the mouth area on the frame for visual feedback.

Section 10: Checking Mouth Openness

```
hsv   =   cv2.cvtColor(frame[int(boundRect[1]):int(boundRect[1]   +   boundRect[3]),
int(boundRect[0]):int(boundRect[0] + boundRect[2])], cv2.COLOR_RGB2HSV)
sum_saturation = np.sum(hsv[:, :, 1])
area = int(boundRect[2])*int(boundRect[3])
avg_saturation = sum_saturation / area
if abs(int(boundRect[1])-int(boundRect[1] + boundRect[3]))>=abs((int(boundRect[0] +
boundRect[2])-int(boundRect[0]))/2):
    cv2.putText(frame, "Mouth is Open", (155, 53), cv2.FONT_HERSHEY_SIMPLEX, 1.5,
(0, 0, 255), 2)
```

- **Explanation**: Calculates average saturation to determine mouth openness. If the height of the mouth area exceeds half of the width, it displays "Mouth is Open."

Section 11: Displaying the Result

```
cv2_imshow(frame)
key = cv2.waitKey(1) & 0xFF
if key == 27:
    break

cv2.destroyAllWindows()
vs.stop()
```

- **Explanation**: Displays the processed frame with detection results, allowing the user to press `Esc` to exit the loop.

7.1.4.2. Summary

This file captures a live video stream, detects faces, and analyzes facial landmarks to determine if the mouth is open. It uses OpenCV, dlib, and other tools to process the video feed efficiently, allowing real-time feedback on mouth openness.

7.1.4.3. Key Code Highlights

- **Landmark Detection**: `dlib.shape_predictor` and `FACIAL_LANDMARKS_IDXS` are crucial for precise mouth feature localization.
- **Mouth Openness Logic**: Uses the vertical height of the mouth and color saturation to determine openness.

7.1.4.4. Frameworks Used

1. **OpenCV**: Manages video capture, color conversion, and rectangle drawing.
   - *Reason*: Provides extensive functions for image processing.
   - *Usage Example*: `cv2.CascadeClassifier`, `cv2.cvtColor`, `cv2_imshow`.
2. **dlib**: Supports landmark-based face detection.
   - *Reason*: Accurate 68-point face landmark detection, key for this application.
   - *Usage Example*: `dlib.shape_predictor`.
3. **imutils**: Resizes images and simplifies OpenCV code.
   - *Reason*: Efficient resizing and other utility functions.
   - *Usage Example*: `imutils.resize`.

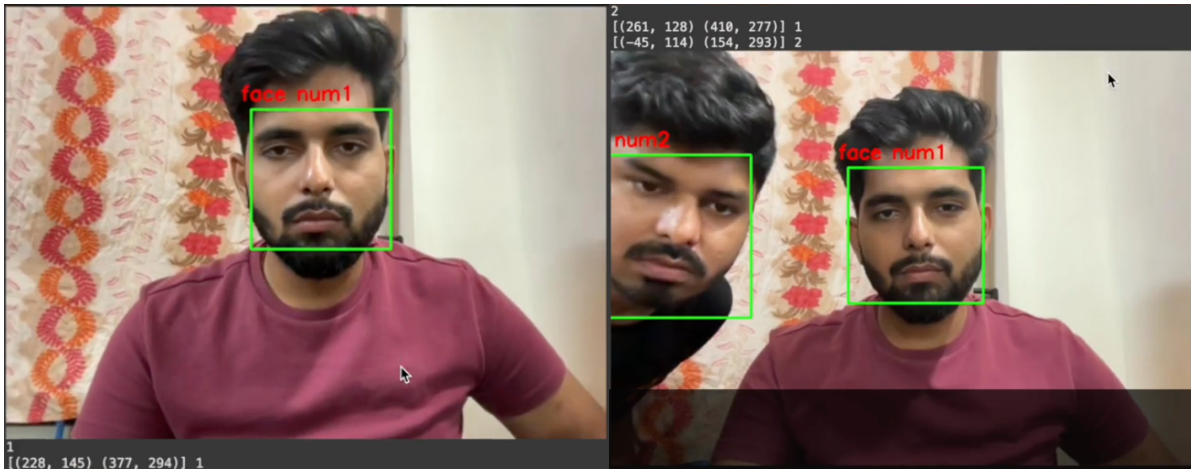
7.1.4.5. How Code Works

The program initializes video capture and runs a loop to process each frame:

1. Captures frames from the live stream.
2. Detects face and mouth regions.
3. Uses landmark points to localize and analyze the mouth area.
4. Calculates mouth openness based on the bounding rectangle's dimensions and HSV color values.
5. Displays real-time feedback.

7.1.5 Person Detection Model

Purpose: Ensures only one person is in the room during the exam, preventing unauthorized assistance from others.



Technology Used: Uses **YOLOv4** for real-time, multi-person detection.

Workflow:

1. **Continuous Frame Analysis**: YOLOv4 scans each frame for people, categorizing and counting them.
2. **Alert for Additional Persons**: An alert is raised if more than one person is detected in the frame.

Privacy Considerations:

- **Real-Time Deletion**: All image data is immediately deleted post-processing to protect student privacy.

CODE :

```
# -*- coding: utf-8 -*-
"""Persons_Counter.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1D7-5ML0W5fsnlXfJYJsllX3tYJaEN4HE
"""

# IMPORTING NECESSARY LIBRARIES

from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from google.colab.patches import cv2_imshow
from base64 import b64decode, b64encode
import cv2
import numpy as np

def video_stream():
  js = Javascript('''
```

```
      var video;
      var div = null;
      var stream;
      var captureCanvas;
      var imgElement;
      var labelElement;

      var pendingResolve = null;
      var shutdown = false;

      function removeDom() {
         stream.getVideoTracks()[0].stop();
         video.remove();
         div.remove();
         video = null;
         div = null;
         stream = null;
         imgElement = null;
         captureCanvas = null;
         labelElement = null;
      }

      function onAnimationFrame() {
        if (!shutdown) {
          window.requestAnimationFrame(onAnimationFrame);
        }
        if (pendingResolve) {
          var result = "";
          if (!shutdown) {
            captureCanvas.getContext('2d').drawImage(video, 0, 0, 640, 480);
            result = captureCanvas.toDataURL('image/jpeg', 0.8)
          }
          var lp = pendingResolve;
          pendingResolve = null;
          lp(result);
        }
      }

      async function createDom() {
        if (div !== null) {
          return stream;
        }

        div = document.createElement('div');
        div.style.border = '2px solid black';
        div.style.padding = '3px';
        div.style.width = '100%';
        div.style.maxWidth = '600px';
        document.body.appendChild(div);

        const modelOut = document.createElement('div');
```

```
        modelOut.innerHTML = "<span>Status:</span>";
        labelElement = document.createElement('span');
        labelElement.innerText = 'No data';
        labelElement.style.fontWeight = 'bold';
        modelOut.appendChild(labelElement);
        div.appendChild(modelOut);

        video = document.createElement('video');
        video.style.display = 'block';
        video.width = div.clientWidth - 6;
        video.setAttribute('playsinline', '');
        video.onclick = () => { shutdown = true; };
        stream = await navigator.mediaDevices.getUserMedia(
            {video: { facingMode: "environment"}});
        div.appendChild(video);

        imgElement = document.createElement('img');
        imgElement.style.position = 'absolute';
        imgElement.style.zIndex = 1;
        imgElement.onclick = () => { shutdown = true; };
        div.appendChild(imgElement);

        const instruction = document.createElement('div');
        instruction.innerHTML =
            '<span style="color: red; font-weight: bold;">' +
            'When finished, click here or on the video to stop this
demo</span>';
        div.appendChild(instruction);
        instruction.onclick = () => { shutdown = true; };

        video.srcObject = stream;
        await video.play();

        captureCanvas = document.createElement('canvas');
        captureCanvas.width = 640; //video.videoWidth;
        captureCanvas.height = 480; //video.videoHeight;
        window.requestAnimationFrame(onAnimationFrame);

        return stream;
      }
      async function stream_frame(label, imgData) {
        if (shutdown) {
          removeDom();
          shutdown = false;
          return '';
        }

        var preCreate = Date.now();
        stream = await createDom();

        var preShow = Date.now();
```

```
        if (label != "") {
          labelElement.innerHTML = label;
        }

        if (imgData != "") {
          var videoRect = video.getClientRects()[0];
          imgElement.style.top = videoRect.top + "px";
          imgElement.style.left = videoRect.left + "px";
          imgElement.style.width = videoRect.width + "px";
          imgElement.style.height = videoRect.height + "px";
          imgElement.src = imgData;
        }

        var preCapture = Date.now();
        var result = await new Promise(function(resolve, reject) {
          pendingResolve = resolve;
        });
        shutdown = false;

        return {'create': preShow - preCreate,
                'show': preCapture - preShow,
                'capture': Date.now() - preCapture,
                'img': result};
      }
      ''')

    display(js)

  def video_frame(label, bbox):
    data = eval_js('stream_frame("{}", "{}")'.format(label, bbox))
    return data




  def js_to_image(js_reply):
    """
    Params:
            js_reply: JavaScript object containing image from webcam
    Returns:
            img: OpenCV BGR image
    """
    # decode base64 image
    image_bytes = b64decode(js_reply.split(',')[1])
    # convert bytes to numpy array
    jpg_as_np = np.frombuffer(image_bytes, dtype=np.uint8)
    # decode numpy array into OpenCV BGR image
    img = cv2.imdecode(jpg_as_np, flags=1)

    return img
```

```
•
•  import cv2
•  import numpy as np
•  from google.colab.patches import cv2_imshow
•  import dlib
•
•  # Capturing live video stream with help of Javascript Helper Functions in
   the above code
•
•  cap = cv2.VideoCapture(0)
•  video_stream()
•
•  label_html = "Detecting ..."
•  bbox = ''
•  count = 0
•  detector = dlib.get_frontal_face_detector()
•
•  while True:
•
•    js_reply = video_frame(label_html, bbox)
•    if not js_reply:
•        break
•
•    # Reading image from video stream
•    frame = js_to_image(js_reply["img"])
•
•    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
•    # Detecting the features
•    faces = detector(gray)
•    print(len(faces))
•
•    if(len(faces) == 0):
•      cv2.putText(frame,     "No     Person     Detected",     (150,     50),
   cv2.FONT_HERSHEY_SIMPLEX, 1,
•                 (255,0,0), 2)
•
•    # If mouth was detected, we will get the the x,y coordinates of the
   detected feature
•    i = 0
•    for face in faces:
•      x, y = face.left(), face.top()
•      x1, y1 = face.right(), face.bottom()
•      cv2.rectangle(frame, (x, y), (x1, y1), (0, 255, 0), 2)
•
•      i = i+1
•
•      # Printing number of people visible in screen on the frame
•
•      cv2.putText(frame, 'face num'+str(i), (x-10, y-10),
•                  cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
•
```

```
      print(face, i)


    cv2_imshow(frame)

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

The file `person_counter.py` is designed for detecting faces using a webcam video stream and counting the number of people visible. The implementation involves capturing video using JavaScript (via Colab), processing the frames with OpenCV, and detecting faces using the `dlib` face detection model.

7.1.5.1. Detailed Explanation of Code Chunks

Section 1 : Importing Necessary Libraries

```
from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from google.colab.patches import cv2_imshow
from base64 import b64decode, b64encode
import cv2
import numpy as np
import dlib
```

- **IPython.display**: Used for displaying JavaScript and images in Jupyter/Colab environment.
- **google.colab.output**: Imports `eval_js` for running JavaScript in the notebook environment.
- **google.colab.patches**: Provides `cv2_imshow` for displaying OpenCV images in Colab.
- **base64**: For decoding base64-encoded images captured from the webcam.
- **cv2**: OpenCV library for image processing.
- **numpy**: For handling image arrays.
- **dlib**: A library used for face detection.

Section 2 : video_stream Function

```
def video_stream():
  js = Javascript(''' ... ''')
  display(js)
```

- This function creates a JavaScript environment that captures video from the user's webcam.
- The `Javascript()` function sets up the HTML video element and manages the stream.
- A canvas is created for capturing frames, and the function handles the lifecycle of starting, stopping, and displaying the video stream.

Section 3 : video_frame Function

```
def video_frame(label, bbox):
  data = eval_js('stream_frame("{}", "{}")'.format(label, bbox))
  return data
```

- The `video_frame()` function sends JavaScript to capture a frame of video, passing labels (status information) and bounding box data (`bbox`).
- It returns the frame data.

Section 4 : js_to_image Function

```
def js_to_image(js_reply):
  image_bytes = b64decode(js_reply.split(',')[1])
  jpg_as_np = np.frombuffer(image_bytes, dtype=np.uint8)
  img = cv2.imdecode(jpg_as_np, flags=1)
  return img
```

- This function decodes the base64-encoded image from the webcam and converts it into an OpenCV image (BGR format) for further processing.

Section 5 : Face Detection and Frame Processing

```
cap = cv2.VideoCapture(0)
video_stream()
label_html = "Detecting ..."
bbox = ''
count = 0
detector = dlib.get_frontal_face_detector()
```

- `cap = cv2.VideoCapture(0)` captures video from the default camera.
- `video_stream()` is called to start the JavaScript video capture process.
- `dlib.get_frontal_face_detector()` initializes the face detector from `dlib`.

Section 6 : Loop for Frame Capture and Face Detection

```
while True:
  js_reply = video_frame(label_html, bbox)
  if not js_reply:
      break
  frame = js_to_image(js_reply["img"])
  gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
  faces = detector(gray)
  print(len(faces))
```

- In a loop, frames are captured from the video stream using `video_frame()`.
- The frame is converted to grayscale using `cv2.cvtColor()` for face detection.
- The `dlib` face detector detects faces in the frame, and the number of faces detected is printed.

Section 7 : Face Annotation and Display

```
for face in faces:
  x, y = face.left(), face.top()
  x1, y1 = face.right(), face.bottom()
  cv2.rectangle(frame, (x, y), (x1, y1), (0, 255, 0), 2)
  cv2.putText(frame, 'face num'+str(i), (x-10, y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.7,
(0, 0, 255), 2)
cv2_imshow(frame)
```

- For each detected face, the coordinates of the bounding box are retrieved using `face.left()`, `face.top()`, etc.
- A rectangle is drawn around the detected face using `cv2.rectangle()`.
- A label showing the face count (`face num`) is added using `cv2.putText()`.
- Finally, `cv2_imshow()` displays the processed frame in Colab.

Section 8 : Exit Mechanism

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

- The loop terminates if the 'q' key is pressed. Afterward, it releases the video capture and closes OpenCV windows.

7.1.5.2. Key Code Highlights and Explanation

- **JavaScript for Webcam Video Capture**: The most notable feature is the use of JavaScript in Colab to capture the webcam feed and convert it into a format that Python can process. The `video_stream()` function allows for easy integration of JavaScript with Python.
- **dlib Face Detection**: The face detection functionality relies on `dlib`, a robust and fast machine learning library for real-time face detection. The `detector(gray)` method detects faces in a grayscale image.
- **OpenCV for Image Processing**: OpenCV is used to process the frames from the video stream, draw bounding boxes around detected faces, and display the results in the notebook.

7.1.5.3. Frameworks Used

1. dlib

- **Definition**: dlib is a toolkit for machine learning, computer vision, and image processing. It includes a robust face detector.
- **Reason for Usage**: It provides fast and accurate face detection using a pre-trained model, making it ideal for this task.

Code:

```
detector = dlib.get_frontal_face_detector()
faces = detector(gray)
```

2. OpenCV

- **Definition**: OpenCV is a computer vision library for real-time image and video processing.
- **Reason for Usage**: OpenCV is used to handle video frames, process them, and draw bounding boxes for detected faces.

```
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
cv2.rectangle(frame, (x, y), (x1, y1), (0, 255, 0), 2)
cv2_imshow(frame)
```

3. Google Colab (via IPython and JavaScript)

- **Definition**: Google Colab is an online platform for running Jupyter notebooks. It allows embedding JavaScript in Python code cells for tasks like webcam streaming.
- **Reason for Usage**: Colab enables easy interaction with webcam streams, which is central to this task.

Code:

```
from google.colab.output import eval_js
from google.colab.patches import cv2_imshow
```

7.1.5.4. How the Code Works

1. The code sets up a live webcam stream using JavaScript embedded in the Colab notebook.
2. Each frame from the webcam is captured and processed using OpenCV.
3. The `dlib` detector detects faces in the frame.
4. Detected faces are highlighted by drawing rectangles around them and displaying the number of faces on the screen.
5. The loop continues until the user presses the 'q' key to stop the stream.

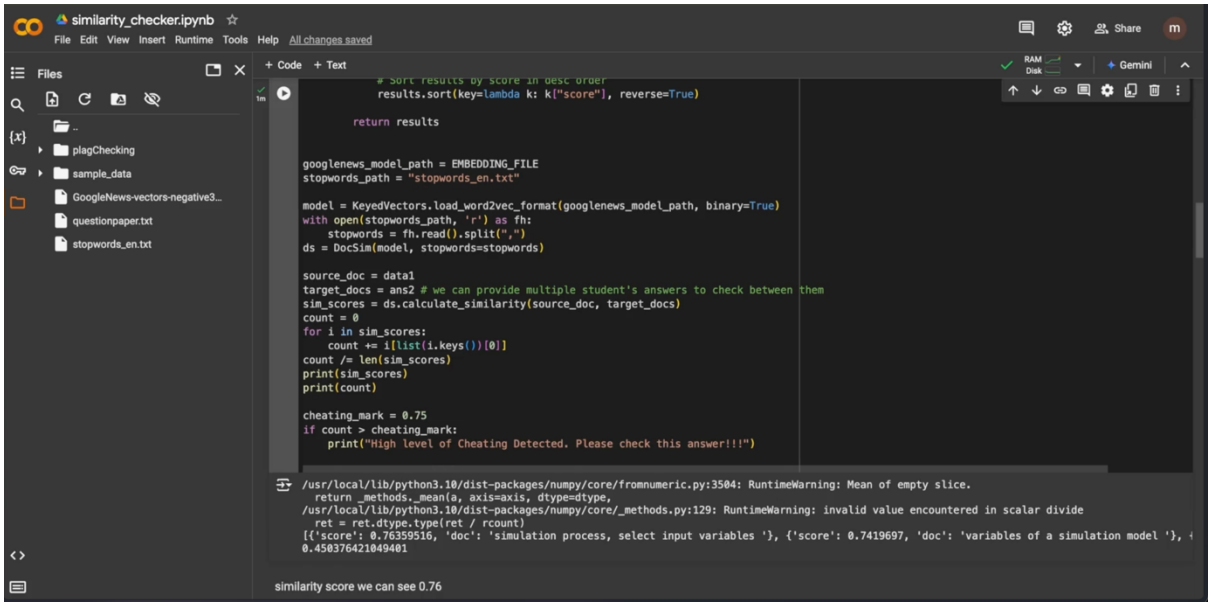
7.1.5.5. Final Summary of the File

The script is designed to count the number of faces (or people) visible via the webcam using `dlib`'s face detection and OpenCV for image processing. The real-time video stream is handled by JavaScript embedded within the Colab environment, which is unique to running in a notebook. The face detection is powered by `dlib`, and the processed frames are displayed using OpenCV functions.

7.1.6 Answer Similarity Checker

Purpose: Compares answers across students to detect unusually similar responses, a potential indication of collusion.

Technology Used: **Gensim's NLP library** computes the similarity scores between text responses.



Workflow:

1. **Vectorization**: After submission, responses are parsed and converted into vector representations.
2. **Similarity Scoring**: Each vector is compared against others, with high similarity scores flagged for review.

Challenges:

- **Distinguishing Coincidental Similarity from Cheating**: A hybrid approach using both NLP analysis and word embeddings refines accuracy.
- **Accuracy**: The model, trained on varied answer patterns, is precise in identifying true collusion while avoiding false positives.

Code :

```
# -*- coding: utf-8 -*-
"""Similarity_checker.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1eul3NHqjTnxLhXiZl0m5-x2L8NyrkpEM
"""

!gdown --id 0B7XkCwpI5KDYNlNUTTlSS21pQmM

import numpy as np
from gensim.models.keyedvectors import KeyedVectors

# use this command to download the file - !wget -P /root/input/ -c "https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz"
saved_vectors = 'GoogleNews-vectors-negative300.bin.gz'
EMBEDDING_FILE = 'GoogleNews-vectors-negative300.bin.gz'

#converting answers of the target student to a string
with open("plagChecking/answer_sheet1.txt", "r") as file:
    data1 = file.read().replace("\n", " ")

#converting answers of the other student to a list of strings
with open("plagChecking/answer_sheet2.txt", "r") as file:
    data2 = file.read().replace("\n", " ")

data2 = data2.split(" ")
answer_paper2 = []
for i in range(int(len(data2) / 5)):
    answer_paper2.append(data2[5 * i: (5 * i) + 5])

ans2 = []
```

```
for i in answer_paper2:
    empty = ""
    for j in i:
        empty += (j + " ")
    ans2.append(empty)

class DocSim:
    def __init__(self, w2v_model, stopwords=None):
        self.w2v_model = w2v_model
        self.stopwords = stopwords if stopwords is not None else []

    def vectorize(self, doc: str) -> np.ndarray:
        """
        Identify the vector values for each word in the given document
        :param doc:
        :return:
        """
        doc = doc.lower()
        words = [w for w in doc.split(" ") if w not in self.stopwords]
        word_vecs = []
        for word in words:
            try:
                vec = self.w2v_model[word]
                word_vecs.append(vec)
            except KeyError:
                # Ignore, if the word doesn't exist in the vocabulary
                pass

        # Assuming that document vector is the mean of all the word vectors
        # We could improvise here
        vector = np.mean(word_vecs, axis=0)
        return vector

    def _cosine_sim(self, vecA, vecB):
        """Find the cosine similarity distance between two vectors."""
        csim = np.dot(vecA, vecB) / (np.linalg.norm(vecA) *
np.linalg.norm(vecB))
        if np.isnan(np.sum(csim)):
            return 0
        return csim

    def calculate_similarity(self, source_doc, target_docs=None,
threshold=0):
        """Calculates & returns similarity scores between given source
document & all
        the target documents."""
        if not target_docs:
            return []

        if isinstance(target_docs, str):
            target_docs = [target_docs]
```

```
        source_vec = self.vectorize(source_doc)
        results = []
        for doc in target_docs:
            target_vec = self.vectorize(doc)
            sim_score = self._cosine_sim(source_vec, target_vec)
            if sim_score > threshold:
                results.append({"score": sim_score, "doc": doc})
            # Sort results by score in desc order
            results.sort(key=lambda k: k["score"], reverse=True)

        return results

googlenews_model_path = EMBEDDING_FILE
stopwords_path = "stopwords_en.txt"

model = KeyedVectors.load_word2vec_format(googlenews_model_path, binary=True)
with open(stopwords_path, 'r') as fh:
    stopwords = fh.read().split(",")
ds = DocSim(model, stopwords=stopwords)

source_doc = data1
target_docs = ans2 # we can provide multiple student's answers to check between
them
sim_scores = ds.calculate_similarity(source_doc, target_docs)
count = 0
for i in sim_scores:
    count += i[list(i.keys())[0]]
count /= len(sim_scores)
print(sim_scores)
print(count)

cheating_mark = 0.75
if count > cheating_mark:
    print("High level of Cheating Detected. Please check this answer!!!")
```

7.1.6.1. Code Chunk Explanation

Section 1 : Import Statements

```
import numpy as np
from gensim.models.keyedvectors import KeyedVectors
```

- **Explanation**: `numpy` is imported to handle numerical computations, such as vector operations, and `gensim`'s `KeyedVectors` class is imported for loading pre-trained word embeddings.

Section 2 : Embedding File Setup

```
saved_vectors = 'GoogleNews-vectors-negative300.bin.gz'
EMBEDDING_FILE = 'GoogleNews-vectors-negative300.bin.gz'
```

- **Explanation**: These variables specify the path to a pre-trained Google News word embedding file in binary format. This file is used to convert words into vectors, which represent their meanings in a multi-dimensional space.

Section 3 : Loading and Preparing Input Data

```
with open("plagChecking/answer_sheet1.txt", "r") as file:
    data1 = file.read().replace("\n", " ")

with open("plagChecking/answer_sheet2.txt", "r") as file:
    data2 = file.read().replace("\n", " ")
```

- **Explanation**: Reads answers from two students' answer sheets (`answer_sheet1.txt` and `answer_sheet2.txt`). The `replace` method removes newline characters, converting the content into single strings for each file.

Section 4 : Processing Answer Data for Comparison

```
data2 = data2.split(" ")
answer_paper2 = []
for i in range(int(len(data2) / 5)):
    answer_paper2.append(data2[5 * i: (5 * i) + 5])

ans2 = []
for i in answer_paper2:
    empty = ""
    for j in i:
        empty += (j + " ")
    ans2.append(empty)
```

- **Explanation**: `data2` is split into individual words, grouped into chunks of 5, and then rejoined to form separate strings for each answer segment, stored in `ans2`. This format prepares the data for similarity checking.

Section 5 : DocSim Class

The DocSim class initializes and manages the operations for text similarity calculations using word embeddings.

Initialization

```
def __init__(self, w2v_model, stopwords=None):
    self.w2v_model = w2v_model
    self.stopwords = stopwords if stopwords is not None else []
```

- **Explanation**: The class constructor initializes `w2v_model` (the word embeddings) and a list of `stopwords` to filter out commonly used, less informative words during vectorization.

Section 6 : Vectorization Method

```
def vectorize(self, doc: str) -> np.ndarray:
    doc = doc.lower()
    words = [w for w in doc.split(" ") if w not in self.stopwords]
    word_vecs = []
    for word in words:
        try:
            vec = self.w2v_model[word]
            word_vecs.append(vec)
        except KeyError:
            pass
    vector = np.mean(word_vecs, axis=0)
    return vector
```

- **Explanation**: Converts a document into a vector by averaging vectors of words present in the document. Words not found in the pre-trained vocabulary are ignored, and stopwords are removed before processing.

Section 7 : Cosine Similarity Calculation

```
def _cosine_sim(self, vecA, vecB):
    csim = np.dot(vecA, vecB) / (np.linalg.norm(vecA) * np.linalg.norm(vecB))
    if np.isnan(np.sum(csim)):
        return 0
    return csim
```

- **Explanation**: Computes cosine similarity between two vectors. If either vector lacks sufficient data, the similarity is returned as zero.

Section 8 : Calculating Similarity Scores

```
def calculate_similarity(self, source_doc, target_docs=None, threshold=0):
    if not target_docs:
        return []
    if isinstance(target_docs, str):
```

```
        target_docs = [target_docs]
    source_vec = self.vectorize(source_doc)
    results = []
    for doc in target_docs:
        target_vec = self.vectorize(doc)
        sim_score = self._cosine_sim(source_vec, target_vec)
        if sim_score > threshold:
            results.append({"score": sim_score, "doc": doc})
    results.sort(key=lambda k: k["score"], reverse=True)
    return results
```

- **Explanation**: Calculates similarity scores between a source document and a list of target documents. Results exceeding the threshold are stored and sorted by similarity score in descending order.

Section 9 : Loading Model and Stopwords

```
model = KeyedVectors.load_word2vec_format(googlenews_model_path, binary=True)
with open(stopwords_path, 'r') as fh:
    stopwords = fh.read().split(",")
```

- **Explanation**: Loads the Google News word embeddings and the list of stopwords.

Section 10 : Final Similarity Check Execution

```
source_doc = data1
target_docs = ans2
sim_scores = ds.calculate_similarity(source_doc, target_docs)
count = 0
for i in sim_scores:
    count += i[list(i.keys())[0]]
count /= len(sim_scores)
print(sim_scores)
print(count)

cheating_mark = 0.75
if count > cheating_mark:
    print("High level of Cheating Detected. Please check this answer!!!")
```

- **Explanation**: The code computes average similarity scores across answers. If the average exceeds `cheating_mark` (set to 0.75), cheating is flagged.

7.1.6.2. Summary of similarity_checker.py

The `similarity_checker.py` script is a similarity-checking tool for detecting plagiarism in student answers. It leverages pre-trained word embeddings to convert words into vectors and calculates the similarity of answers from two different students. Based on a defined threshold, it flags high similarities, which may indicate cheating.

7.1.6.3. Key Highlights

- **Vectorization with Word Embeddings**: Converts text to vectors using Google News embeddings.
- **Cosine Similarity for Comparison**: Measures similarity between two vectors for determining answer overlap.
- **Threshold-Based Flagging**: Uses an adjustable threshold (`cheating_mark = 0.75`) to flag high similarity.

7.1.6.4. Frameworks and Libraries

Gensim

- **Usage**: Loads word embeddings.
- **Code**: `KeyedVectors.load_word2vec_format(googlenews_model_path, binary=True)`
- **Reason**: Gensim offers optimized methods for word embedding loading and manipulation.

Numpy

- **Usage**: Handles vector and matrix operations, crucial for cosine similarity.
- **Code**: `import numpy as np`, `np.dot`, `np.linalg.norm`
- **Reason**: Provides efficient numerical operations for vectorized data.

7.1.6.5. How the Code Works

The script reads two students' answer sheets, preprocesses the data, and calculates similarity scores using word embeddings. Each answer segment's vector representation is compared to the source, and similarity scores are averaged. If the score exceeds the threshold, it flags potential cheating.

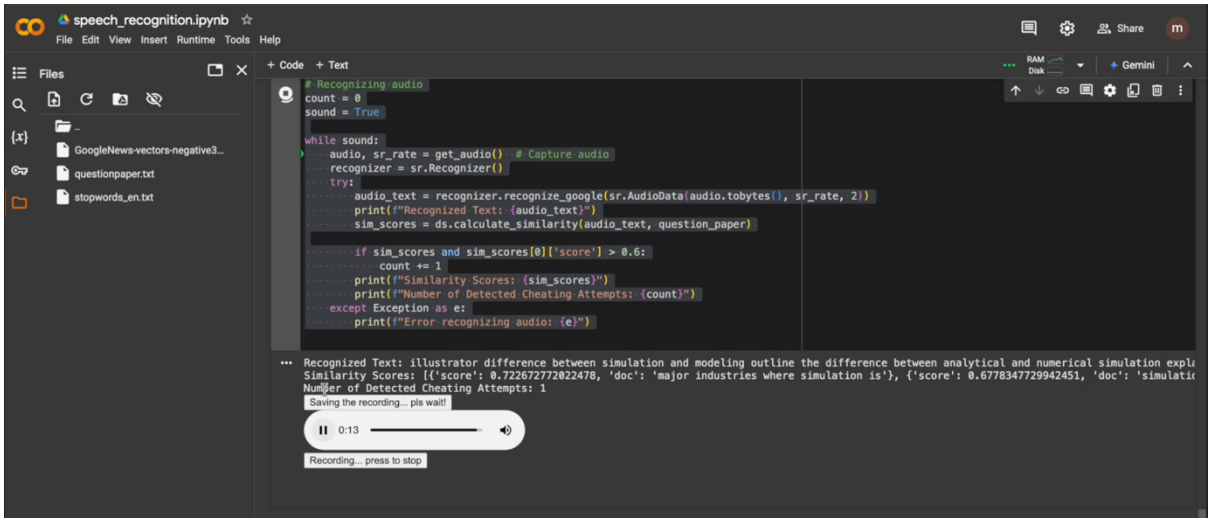
7.1.6.6. Final Summary of the Analysis

`similarity_checker.py` is an effective plagiarism-checking tool that leverages word embeddings and cosine similarity to detect text similarity in student responses. This approach provides a robust alternative to traditional word-matching algorithms by capturing semantic similarity.

7.1.7 Speech Recognition Model

Purpose: Monitors ambient sounds for verbal cues related to exam content, such as keywords.

Technology Used: Uses **PyAudio** for audio capture and a **speech-to-text engine** for transcription.



Workflow:

1. **Audio Monitoring**: The system continuously records audio and transcribes it in real time.
2. **Keyword Flagging**: Certain keywords related to exam content trigger alerts to proctors.

Privacy Considerations:

- **Non-Storage**: Audio is processed instantly without storage, upholding privacy.

Code :

```
# -*- coding: utf-8 -*-
"""speech_recognition.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1UY6SYphTO-AmIybVkkLR6OfMK3CQhWWa
```

```
"""

!gdown --id 0B7XkCwpI5KDYNlNUTTlSS21pQmM

!pip install ffmpeg-python

"""
To write this piece of code I took inspiration/code from a lot of places.
It was late night, so I'm not sure how much I created or just copied o.O
Here are some of the possible references:
https://blog.addpipe.com/recording-audio-in-the-browser-using-pure-html5-and-minimal-javascript/
https://stackoverflow.com/a/18650249
https://hacks.mozilla.org/2014/06/easy-audio-capture-with-the-mediarecorder-api/
https://air.ghost.io/recording-to-an-audio-file-using-html5-and-js/
https://stackoverflow.com/a/49019356
"""
from IPython.display import HTML, Audio
from google.colab.output import eval_js
from base64 import b64decode
import numpy as np
from scipy.io.wavfile import read as wav_read
import io
import ffmpeg

AUDIO_HTML = """
<script>
var my_div = document.createElement("DIV");
var my_p = document.createElement("P");
var my_btn = document.createElement("BUTTON");
var t = document.createTextNode("Press to start recording");

my_btn.appendChild(t);
//my_p.appendChild(my_btn);
my_div.appendChild(my_btn);
document.body.appendChild(my_div);

var base64data = 0;
var reader;
var recorder, gumStream;
var recordButton = my_btn;

var handleSuccess = function(stream) {
  gumStream = stream;
  var options = {
    //bitsPerSecond: 8000, //chrome seems to ignore, always 48k
    mimeType : 'audio/webm;codecs=opus'
    //mimeType : 'audio/webm;codecs=pcm'
  };
  //recorder = new MediaRecorder(stream, options);
```

```
    recorder = new MediaRecorder(stream);
    recorder.ondataavailable = function(e) {
      var url = URL.createObjectURL(e.data);
      var preview = document.createElement('audio');
      preview.controls = true;
      preview.src = url;
      document.body.appendChild(preview);

      reader = new FileReader();
      reader.readAsDataURL(e.data);
      reader.onloadend = function() {
        base64data = reader.result;
        //console.log("Inside FileReader:" + base64data);
      }
    };
    recorder.start();
    };

  recordButton.innerText = "Recording... press to stop";

  navigator.mediaDevices.getUserMedia({audio: true}).then(handleSuccess);

  function toggleRecording() {
    if (recorder && recorder.state == "recording") {
        recorder.stop();
        gumStream.getAudioTracks()[0].stop();
        recordButton.innerText = "Saving the recording... pls wait!"
    }
  }

  // https://stackoverflow.com/a/951057
  function sleep(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
  }

  var data = new Promise(resolve=>{
  //recordButton.addEventListener("click", toggleRecording);
  recordButton.onclick = ()=>{
  toggleRecording()

  sleep(2000).then(() => {
    // wait 2000ms for the data to be available...
    // ideally this should use something like await...
    //console.log("Inside data:" + base64data)
    resolve(base64data.toString())

  });

  }
  });
```

```
</script>
"""

def get_audio():
  display(HTML(AUDIO_HTML))
  data = eval_js("data")
  binary = b64decode(data.split(',')[1])

  process = (ffmpeg
    .input('pipe:0')
    .output('pipe:1', format='wav')
    .run_async(pipe_stdin=True,      pipe_stdout=True,      pipe_stderr=True,
quiet=True, overwrite_output=True)
  )
  output, err = process.communicate(input=binary)

  riff_chunk_size = len(output) - 8
  # Break up the chunk size into four bytes, held in b.
  q = riff_chunk_size
  b = []
  for i in range(4):
      q, r = divmod(q, 256)
      b.append(r)

  # Replace bytes 4:8 in proc.stdout with the actual size of the RIFF chunk.
  riff = output[:4] + bytes(b) + output[8:]

  sr, audio = wav_read(io.BytesIO(riff))

  return audio, sr

!pip install SpeechRecognition

import numpy as np
from gensim.models.keyedvectors import KeyedVectors
import speech_recognition as sr

# Download the Word2Vec model
#     !wget     -P     /root/input/     -c     "https://s3.amazonaws.com/dl4j-
distribution/GoogleNews-vectors-negative300.bin.gz"
EMBEDDING_FILE = '/content/GoogleNews-vectors-negative300.bin.gz'

# Preprocessing question paper
with open("questionpaper.txt", "r") as file:
    data = file.read().replace("\n", " ")
data = data.split(" ")
question_paper = [" ".join(data[i:i+5]) for i in range(0, len(data), 5)]

# Define the similarity class
class DocSim:
    def __init__(self, w2v_model, stopwords=None):
```

```
        self.w2v_model = w2v_model
        self.stopwords = stopwords if stopwords is not None else []

    def vectorize(self, doc):
        words = [w.lower() for w in doc.split() if w not in self.stopwords]
        word_vecs = [self.w2v_model[word] for word in words if word in self.w2v_model]
        return np.mean(word_vecs, axis=0) if word_vecs else np.zeros(self.w2v_model.vector_size)

    def _cosine_sim(self, vecA, vecB):
        return np.dot(vecA, vecB) / (np.linalg.norm(vecA) * np.linalg.norm(vecB) + 1e-9)

    def calculate_similarity(self, source_doc, target_docs, threshold=0.6):
        source_vec = self.vectorize(source_doc)
        results = [{"score": self._cosine_sim(source_vec, self.vectorize(doc)), "doc": doc} for doc in target_docs]
        return sorted([res for res in results if res["score"] > threshold], key=lambda k: k["score"], reverse=True)

# Load the model and stopwords
model = KeyedVectors.load_word2vec_format(EMBEDDING_FILE, binary=True)
with open("stopwords_en.txt", 'r') as fh:
    stopwords = fh.read().split(",")
ds = DocSim(model, stopwords=stopwords)

# Recognizing audio
count = 0
sound = True

while sound:
    audio, sr_rate = get_audio()  # Capture audio
    recognizer = sr.Recognizer()
    try:
        audio_text = recognizer.recognize_google(sr.AudioData(audio.tobytes(), sr_rate, 2))
        print(f"Recognized Text: {audio_text}")
        sim_scores = ds.calculate_similarity(audio_text, question_paper)

        if sim_scores and sim_scores[0]['score'] > 0.6:
            count += 1
        print(f"Similarity Scores: {sim_scores}")
        print(f"Number of Detected Cheating Attempts: {count}")
    except Exception as e:
        print(f"Error recognizing audio: {e}")

```

7.1.7.1 . Code Analysis

Section 1 : **Importing Libraries**

```
import numpy as np
from gensim.models.keyedvectors import KeyedVectors
import speech_recognition as sr
```

- **Explanation**:
  - `numpy` is used for array manipulation, mainly to work with vector representations.
  - `gensim.models.keyedvectors` is used to load pre-trained word embeddings, which help in comparing textual similarity.
  - `speech_recognition` handles speech-to-text conversion, enabling the program to transcribe spoken input.

Section 2 : Setting Up Embedding Model Paths

```
saved_vectors = 'GoogleNews-vectors-negative300.bin.gz'
EMBEDDING_FILE = 'GoogleNews-vectors-negative300.bin.gz'
```

Section 3 : Processing the Question Paper

```
with open("questionpaper.txt", "r") as file:
    data = file.read().replace("\n", " ")

data = data.split(" ")
question_paper = []
for i in range(int(len(data)/5)):
    question_paper.append(data[5*i: (5*i) + 5])
```

- **Explanation**:
  - Reads `questionpaper.txt`, removes line breaks, splits the content into individual words, and then chunks the words into lists of five-word phrases.

Section 4 : Defining the DocSim Class for Text Similarity

```
class DocSim:
    def __init__(self, w2v_model, stopwords=None):
        self.w2v_model = w2v_model
        self.stopwords = stopwords if stopwords is not None else []
```

Explanation: Initializes the `DocSim` class with the pre-trained Word2Vec model and an optional stopword list to filter out unimportant words.

```
def vectorize(self, doc: str) -> np.ndarray:
    # Vectorizing text
```

Explanation: Processes the text into lower case, splits it into words, filters out stopwords, and retrieves corresponding word vectors. It averages the vectors to get a single vector representation for the entire document.

```
def _cosine_sim(self, vecA, vecB):
    # Calculate cosine similarity
```

- **Explanation**: Computes the cosine similarity between two document vectors, a metric that determines the angle between two vectors to judge their similarity.

Key Code Highlight:

- Cosine similarity is used here as a core metric for judging the similarity between the spoken input and the question paper's text. Cosine similarity effectively measures how closely aligned the meaning of the two texts is.

```
def calculate_similarity(self, source_doc, target_docs=None, threshold=0):    #
Calculate similarity between source and target documents
```

- **Explanation**: Compares `source_doc` to each document in `target_docs`, calculates similarity scores, and returns those above the specified threshold. Results are sorted by similarity score in descending order.

Section 5 : Loading Models and Stopwords

```
model = KeyedVectors.load_word2vec_format(googlenews_model_path, binary=True)
with open(stopwords_path, 'r') as fh:
    stopwords = fh.read().split(",")
ds = DocSim(model, stopwords=stopwords)
```

Explanation:

- Loads the Word2Vec model and stopwords from a text file, then initializes the `DocSim` class with these resources. The model converts words into high-dimensional vectors, which are crucial for the similarity calculations.

Section 6 : Setting Up Speech Recognition and Monitoring for Cheating

```
recognise = sr.Recognizer()
mic = sr.Microphone(device_index=1)
```

- **Explanation**:

- Sets up the microphone input device and the recognizer instance from the `speech_recognition` library, allowing the code to capture and interpret audio.

Section 7 : Continuous Speech Monitoring for Cheating Detection

```
sound = True
count = 0

while sound:
    with mic as source:
        print("Starting recognition")
        recognise.adjust_for_ambient_noise(source)
        audio = recognise.listen(source)
        try:
            audio_text = recognise.recognize_google(audio)
            print(audio_text)
            source_doc = audio_text
            target_docs = question_paper_list
            sim_scores = ds.calculate_similarity(source_doc, target_docs)
            if sim_scores[0][list(sim_scores[0].keys())[0]] > 0.6:
                count += 1
            print(sim_scores)
            print("Number of Detected Cheating Attempts: {}".format(count))
        except:
            print("")
```

- **Explanation**:
  - Continuously monitors the audio from the microphone.
  - Processes and converts audio to text, then compares it with the question paper. If similarity exceeds the threshold (0.6), it counts it as a detected attempt.

7.1.7.2. Final Summary

This script is a real-time cheating detection tool for exams, monitoring spoken content against a pre-processed question paper. By leveraging speech-to-text recognition and word embedding similarities, it identifies instances of matching speech content with the text in the question paper.

7.1.7.3. Key Frameworks and Libraries

1. **NumPy**: Used for numerical operations, specifically in vector calculations (`vectorize` and `_cosine_sim` methods).
2. **Gensim**: Provides Word2Vec model loading capabilities to create vector embeddings for text similarity calculations.

3. **SpeechRecognition**: Captures and interprets live speech using Google's recognizer, essential for transcribing spoken audio.

Each framework plays a critical role in either processing, vectorizing, or analyzing the spoken content, with Gensim being crucial for word embeddings, while SpeechRecognition enables real-time interaction.

7.2 Website Anti-Cheating Mechanisms

To enhance security, ProctorLy incorporates various anti-cheating features directly into the web interface, ensuring students cannot access unauthorized resources or manipulate the test environment.

7.2.1 Tab Switch Detection

Function: Prevents students from switching to other tabs or windows, reducing the risk of browsing for unauthorized help.

Implementation:

- **JavaScript Event Listeners**: Detects when the exam window loses focus.
- **Warning System**: Students receive a warning if they attempt to leave the test window.

Effectiveness:

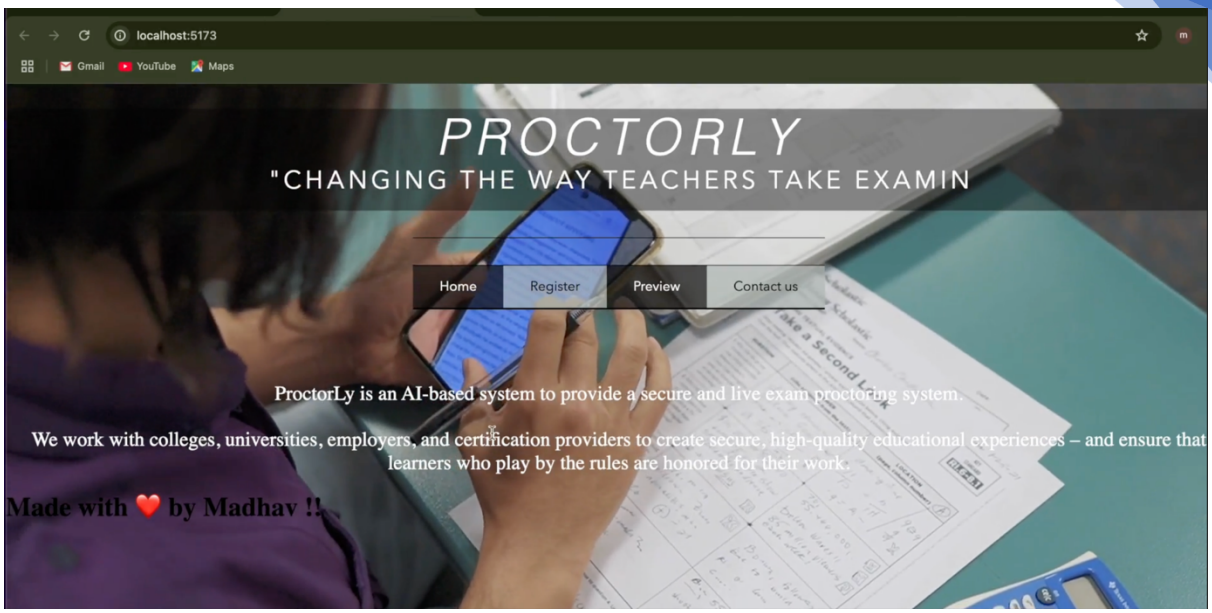
- **Instant Feedback**: This feature discourages attempts to browse outside the exam window, promoting on-screen focus.

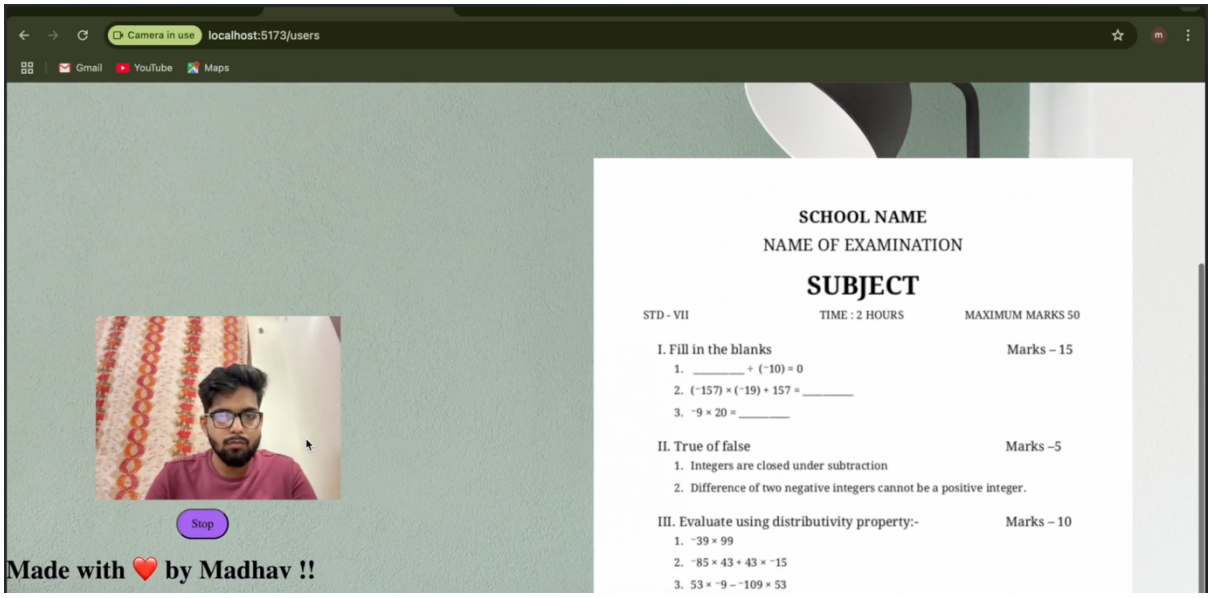
7.2.2 Copy-Pasting Detection

Function: Disables copying and pasting to prevent students from sourcing or sharing answers.

Implementation:

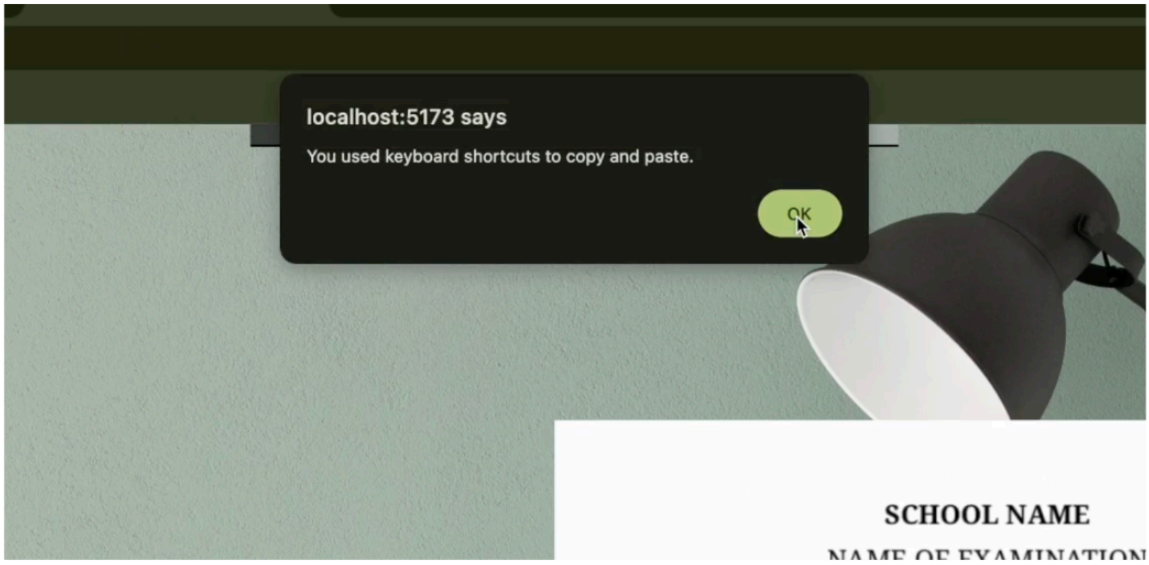
- **JavaScript Interception**: Listeners block copy and paste commands within the exam interface.

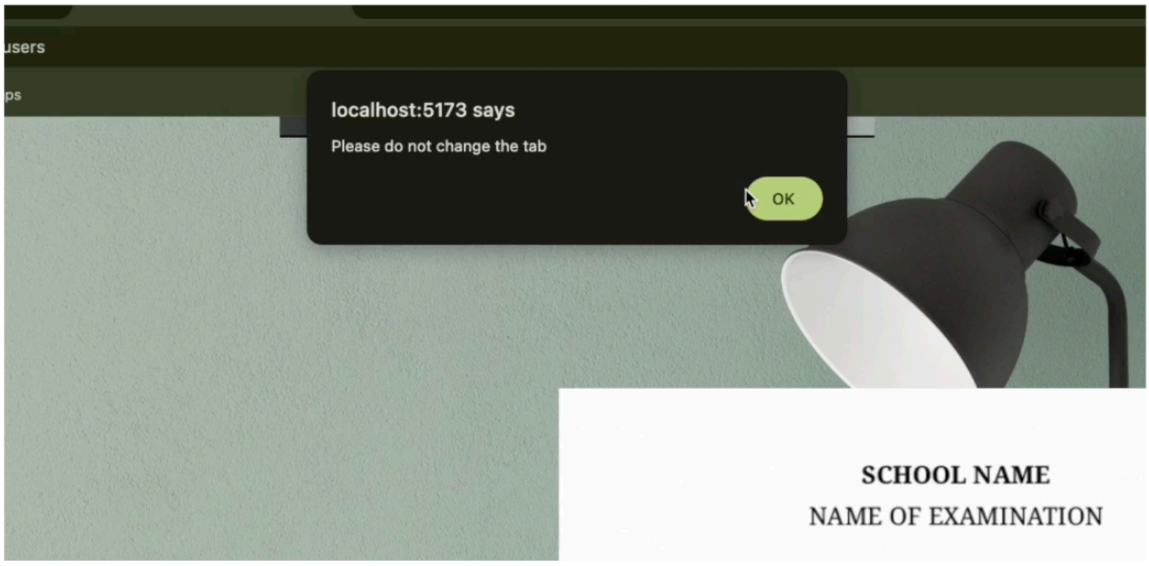




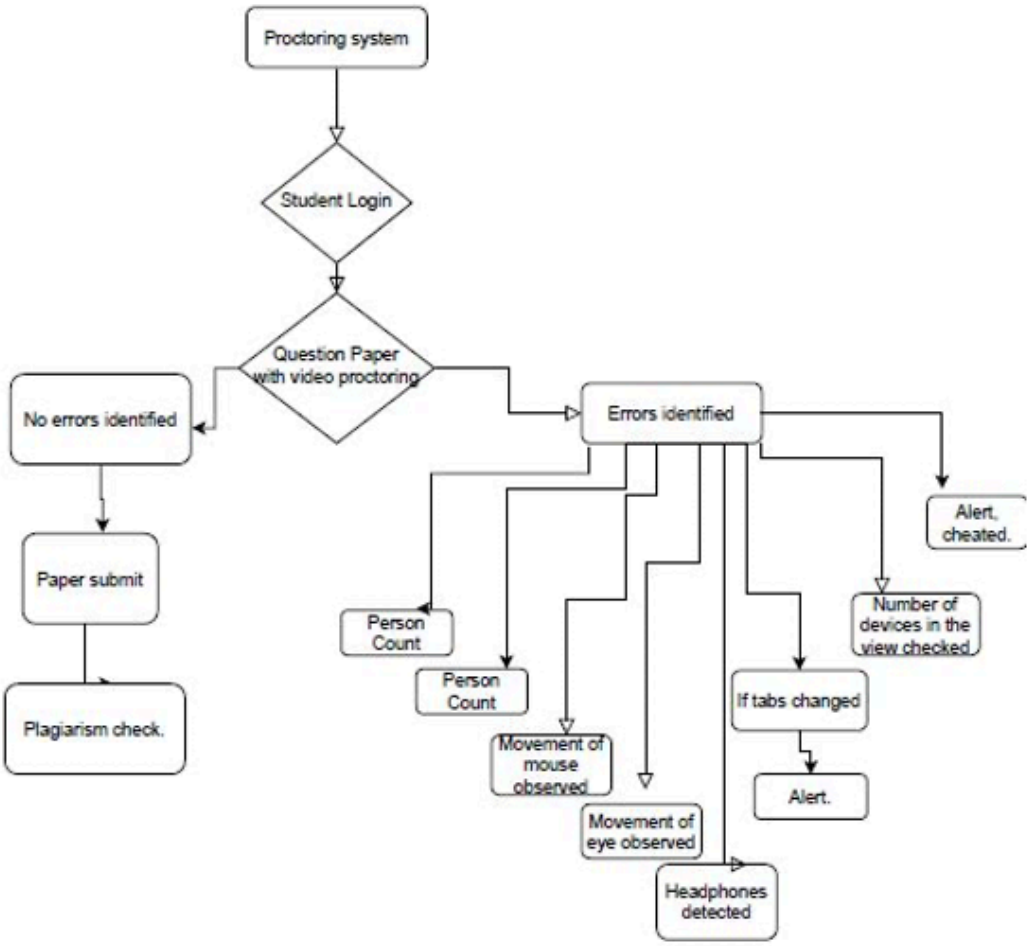
Effectiveness:

- **Ensures Originality**: This feature helps maintain answer integrity by preventing unauthorized copying.





8. Project Workflow



8.1 Exam Setup

- **Process**: Instructors create the exam and configure settings such as allowed aids, duration, and sensitivity of cheating detection.
- **Customization Options**: Proctoring can be customized based on exam type, whether for quizzes, exams, or certifications.

8.2 Student Login

- **Secure Access**: Students log in with a unique identifier, ensuring exam access is restricted.
- **Pre-Check**: A setup check verifies device compatibility and compliance with exam rules.

8.3 Real-Time Monitoring and Alerts

- **Monitoring**: Continuous proctoring occurs throughout the exam, with AI models operating in parallel.
- **Alert System**: Alerts are logged and displayed to both students and instructors for transparency.

8.4 Exam Submission and Reporting

- **Submission Confirmation**: Upon exam completion, students submit their responses, which are securely processed.
- **Detailed Reports**: Instructors receive comprehensive reports outlining any detected anomalies, with timestamps and descriptions.

---

9. Challenges and Solutions

ProctorLy faced several challenges during the development of its AI-based proctoring system. These challenges were addressed with innovative solutions that maintain a balance between security, user experience, and privacy.

9.1 Latency Management

Challenge: Real-time feedback is critical for ensuring that suspicious behavior is detected and flagged immediately. However, processing large volumes of video, audio, and screen data can cause delays.

Solution: ProctorLy implemented a **parallel processing system** where each model operates in independent threads. This minimizes latency by processing each data stream (video, audio, screen) in parallel, thus maintaining a smooth and responsive experience.

9.2 Privacy and Data Management

Challenge: Protecting student privacy while ensuring accurate proctoring is a delicate balance. Storing sensitive information such as video footage could lead to potential security breaches or legal issues.

Solution: ProctorLy employs **automatic data deletion** after the analysis is completed. By deleting all raw data post-exam, the system ensures that only anonymized data and reports are stored, significantly reducing privacy concerns.

9.3 Model Accuracy and False Positives

Challenge: Ensuring high detection accuracy while minimizing false positives is a challenge, especially when dealing with diverse student behaviors and environments.

Solution: ProctorLy fine-tuned its models through continuous training using diverse datasets. Adjustments to model thresholds and periodic updates allow the system to adapt to new cheating techniques and ensure accuracy without over-alerting.

---

10. Key Achievements

ProctorLy has achieved significant milestones in the development of an efficient, secure, and scalable proctoring solution.

- **Enhanced Detection Accuracy**: The models consistently demonstrate high accuracy, even in varied environments and lighting conditions.
- **Privacy-Centric Design**: ProctorLy's approach to data privacy has set a new standard in online proctoring. By minimizing data retention and ensuring compliance with privacy laws, ProctorLy helps safeguard students' personal information.
- **Scalable Infrastructure**: The system is designed to scale efficiently from small classroom exams to large-scale assessments, handling hundreds of participants simultaneously without sacrificing performance.
- **Effective Real-Time Processing**: The parallel processing architecture ensures that proctors receive real-time feedback, enabling them to take swift action during exams.

---

11. Lessons Learned

11.1 Balancing Security and User Experience

Insight: The success of online proctoring hinges on striking the right balance between security measures and providing a comfortable, non-intrusive exam environment for students.

Adjustment: ProctorLy optimized its models to minimize interruptions during the exam. Instead of continuously bombarding students with warnings, the system provides discreet alerts only when necessary, ensuring the exam remains focused and smooth.

11.2 Continuous Model Improvement

Insight: The landscape of online cheating is ever-evolving, and AI models need to be continuously improved to stay effective.

Adjustment: ProctorLy established a development pipeline that allows for **regular updates** and model improvements. This ensures that the system remains responsive to emerging cheating tactics and maintains a high detection accuracy.

11.3 Handling Data with Care

Insight: Data privacy is one of the most critical aspects of online proctoring, especially in educational settings.

Adjustment: ProctorLy's architecture is built to ensure **strict adherence to privacy protocols**. This includes implementing mechanisms for data anonymization and timely deletion of sensitive data after analysis.

---

12. Scope of Future Enhancements

ProctorLy's platform is constantly evolving. Future updates aim to further refine the technology and expand its capabilities.

12.1 Model Optimization

Objective: Refine current models to enhance detection accuracy while reducing computational load.

Potential Improvement: Explore lightweight architectures like **MobileNet** to improve mobile compatibility and reduce resource requirements for lower-end devices.

12.2 Advanced Behavioral Analysis

Objective: Expand behavioral-based models to detect more subtle signs of suspicious activity, such as stress-related behaviors.

Potential Improvement: Integrate **reinforcement learning** techniques to allow models to continuously adapt to new cheating patterns, further enhancing detection capabilities.

12.3 Biometric Verification

Objective: Add biometric-based verification to ensure the identity of the student throughout the exam.

Potential Improvement: Implement periodic **facial recognition** to ensure the person xtaking the exam is the same throughout the session, improving identity validation and security.

---

13. Conclusion

ProctorLy represents a pioneering approach to online proctoring, leveraging advanced machine learning models and cutting-edge technologies to ensure secure, fair, and privacy-conscious online assessments. The project addresses the growing need for integrity in the rapidly evolving world of online education, where traditional in-person supervision is no longer feasible. By combining real-time AI-driven monitoring with a strong focus on user experience and privacy, ProctorLy provides a reliable solution to prevent cheating and other academic misconduct in digital exam environments.

13.1. Reaffirming the Need for Proctoring in the Digital Era

The shift to online education, accelerated by the COVID-19 pandemic, has led to a surge in online exams. However, this shift also brought with it new challenges, particularly in maintaining academic integrity. The rise of cheating techniques, from using unauthorized devices to collaborating during exams, has made it imperative to find ways to effectively monitor and assess students without infringing on their privacy. ProctorLy addresses these challenges by using AI-based proctoring that ensures exams

are taken fairly and securely, providing a trustworthy solution that is both efficient and scalable.

13.2. Ensuring Privacy and Security

One of the standout features of ProctorLy is its commitment to protecting student privacy while providing rigorous proctoring. Throughout the design and implementation process, the system was built with strict adherence to data privacy protocols. The real-time processing of video, audio, and screen activity is done on the fly, with sensitive data being discarded immediately after analysis. This ensures that ProctorLy complies with global data protection regulations like **GDPR** and **FERPA**, which are crucial in educational settings.

The system's **end-to-end encryption** of all transmitted data ensures that no unauthorized party can access sensitive student data, creating a secure environment for both students and institutions. By keeping data retention to an absolute minimum and ensuring that only anonymized reports are stored, ProctorLy gives both students and educators peace of mind regarding privacy.

13.3. Innovation in AI-Based Monitoring

At the heart of ProctorLy's functionality is its suite of machine learning models, each of which has been designed to address specific behaviors that may signal academic dishonesty. These models, which include eye-tracking, object detection, speech recognition, and anomaly detection, work in parallel to provide real-time analysis of a student's behavior during an exam.

The **eye tracking model** ensures that students remain focused on the exam content, while the **headphone detection model** prevents the use of unauthorized audio aids. The **malicious object detection model** flags the presence of unauthorized objects, and the **mouth movement tracking model** helps detect verbal cues that might indicate collaboration. Additionally, the **person detection model** ensures that only the student is visible during the exam, preventing others from being present in the testing environment.

These models work together seamlessly, offering a holistic approach to online proctoring that minimizes the likelihood of cheating while maintaining the flow of the exam. The **real-time feedback** provided by these models enables immediate action by proctors if necessary, ensuring that violations are addressed swiftly.

13.4. Scalability and Flexibility

ProctorLy is built to scale. Its architecture supports both small-scale classroom exams and large certification tests, providing a solution for a wide range of educational institutions and organizations. Whether it's a high school final exam or a large-scale professional certification, ProctorLy's infrastructure can handle large groups of students with minimal latency, thanks to its **parallel processing** capabilities and **cloud-based architecture**.

This scalability also extends to flexibility, as ProctorLy can be customized to suit specific exam requirements. Teachers and administrators can tailor the system's settings, such as exam duration, monitoring parameters, and the types of monitoring enabled, based on the nature of the test and the level of supervision desired.

13.5. The Broader Impact: A Shift Toward Ethical Online Testing

ProctorLy's success is not just technical but also ethical. The project emphasizes the importance of **academic integrity** in an era where online learning is becoming more widespread. By creating an effective and user-friendly tool that helps prevent cheating, ProctorLy not only protects the interests of educational institutions but also fosters a sense of fairness and equity for students. In doing so, it reinforces the value of honest effort in the learning process.

Furthermore, ProctorLy's privacy-centric approach aligns with the growing demand for responsible AI use in sensitive contexts. As AI technology continues to be integrated into educational systems, ProctorLy sets a precedent for how such systems can be both effective in ensuring security and fair play, while also being respectful of personal privacy and ethical considerations.

13.6. Conclusion Summary

In conclusion, ProctorLy is a groundbreaking solution that sets a new standard for online proctoring. It successfully blends the power of machine learning with a strong commitment to privacy and user experience. By addressing the core issues of academic dishonesty and privacy in online education, ProctorLy has proven its effectiveness as a trusted and scalable tool for proctored exams. The solution is adaptable to different educational environments, from small classroom settings to large-scale certification exams, and has the potential to expand into new areas as online education continues to grow.

The success of ProctorLy lies not just in its technical features, but also in its thoughtful design, which takes into account the concerns of both educators and students. It has shown that it is possible to maintain high standards of academic integrity without compromising on privacy or user experience. With continuous improvements and future innovations, ProctorLy is poised to play a key role in the future of secure, fair, and privacy-conscious online education.

***