DROWSINESS DETECTION SYSTEM

Enhancing Safety and Preventing Accidents

TEAM - 14

R.B.Deepika

D.Manish

M.Likhith

A Drowsiness Detection System is an intelligent software application or device that aims to monitor and detect the level of drowsiness or fatigue in a person, particularly while they are engaged in activities that require attentiveness and focus, such as driving, operating heavy machinery, or performing critical tasks.

The main purpose of a Drowsiness Detection System is to enhance safety and prevent accidents in situations where a person's reduced alertness due to drowsiness could lead to hazardous consequences. It is commonly used in the following scenario:

Driving Safety: Driver fatigue and drowsiness are major factors contributing to road accidents. By detecting early signs of drowsiness in drivers, the system can issue warnings or alerts to the driver, allowing them to take necessary breaks or corrective actions to avoid accidents.

How Drowsiness Detection Systems Work:

Drowsiness detection systems employ various technologies, such as computer vision, machine learning approaches, to monitor and analyze a person’s eyes ,indicative of drowsiness.

Computer vision techniques may involve analyzing eye movements to detect signs of fatigue, such as drooping eyelids or prolonged eye closure. Machine learning models , are trained to classify open and closed eyes to determine the drowsiness level based on the eye status.

Phase – 1 : A Comprehensive CNN Model Training and Evaluation

Importing Libraries: The necessary libraries and modules are imported to perform various tasks like data handling, image processing, and building the CNN model using TensorFlow and Keras.

```
import numpy as np
import pandas as pd
import os
import tensorflow as tf

from glob import glob
from tqdm import tqdm

import matplotlib.pyplot as plt
from PIL import Image
```

tqdm is used to visualize the progress of iterations, to know how much time is left for the loop to complete.
glob is used to retrieve files and folder names that match a specified pattern.
PIL provides various image processing capabilities, such as opening, manipulating, and saving image files.
With this import statement, you can use the **Image** class and its methods to work with images in your Python code. For example, you can open an image file, resize it, convert it to grayscale, apply filters, and perform various image processing operations

Loading and Preprocessing Images: The code loads images from the “eyes" and resizes them to a uniform size of 64x64 pixels. The images are then normalized using Min-Max Scaling, which ensures pixel values are in the range [0, 1].

```
X = []
Y = []

for i in tqdm(glob('eyes/train/Open_Eyes/*')):
    temp = np.array(Image.open(i).resize((64,64)))
    X.append(temp)
    Y.append(1)

for i in tqdm(glob('eyes/train/Open_Eyes/*')):
    temp = np.array(Image.open(i).resize((64,64)))
    X.append(temp)
    Y.append(0)

# Normalizing the images using Min-Max Scaling algorithm
X = (np.array(X) - np.min(X)) / (np.max(X) - np.min(X))
X = X/255.0
Y = (np.array(Y) - np.min(Y)) / (np.max(Y) - np.min(Y))

# Expanding the dimensions of array (numpy)
X = np.expand_dims(X,-1)
```

Splitting the Dataset: The dataset is split into training and testing sets using the train_test_split function from sklearn.model_selection. This division is crucial to evaluate the model's performance on unseen data.

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=42)
```

CNN Model Architecture: A Convolutional Neural Network (CNN) model is built using the Sequential API of Keras. The model includes several layers for feature extraction and classification:

- **Convolutional Layers**: Two sets of convolutional layers with filter sizes of 5x5 and 3x3, respectively, are employed. These layers extract features from the input images.
- **Batch Normalization**: Batch normalization layers help in stabilizing and accelerating the training process by normalizing the activations of the previous layer.
- **MaxPooling Layers**: These layers downsample the spatial dimensions of the feature maps, reducing computation while retaining essential information.
- **Dropout Layers**: Dropout layers randomly drop a fraction of neurons during training to prevent overfitting.
- **Flatten Layer**: The flatten layer converts the 2D feature maps into a 1D vector, preparing it for the fully connected layers.
- **Dense (Fully Connected) Layers**: Two fully connected layers are used for classification. These layers help the model learn complex relationships between extracted features.
- **Output Layer**: The output layer has a sigmoid activation function to produce a binary classification output (0 for closed eyes and 1 for open eyes).

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Conv2D, BatchNormalization, MaxPooling2D,Dropout, Flatten

model = tf.keras.models.Sequential([
      Input(shape=(64, 64, 1)),

      Conv2D(filters = 32, kernel_size = 5, strides = 1, activation = 'relu'),
      Conv2D(filters = 32, kernel_size = 5, strides = 1, activation = 'relu', use_bias=False),
      BatchNormalization(),
      MaxPooling2D(strides = 2),
      Dropout(0.3),

      Conv2D(filters = 64, kernel_size = 3, strides = 1, activation = 'relu'),
      Conv2D(filters = 64, kernel_size = 3, strides = 1, activation = 'relu', use_bias=False),
      BatchNormalization(),
      MaxPooling2D(strides = 2),
      Dropout(0.3),

      Flatten(),
      Dense(units  = 256, activation = 'relu', use_bias=False),
      BatchNormalization(),

      Dense(units = 128, use_bias=False, activation = 'relu'),

      Dense(units = 84, use_bias=False, activation = 'relu'),
      BatchNormalization(),
      Dropout(0.3),

      Dense(units = 1, activation = 'sigmoid')
  ])
```

Compiling the Model: The model is compiled with the binary cross-entropy loss function and the Adam optimizer. Additionally, the accuracy metric is used to monitor the model's performance during training.

Training the Model: The model is trained on the training dataset with a validation split of 20%. The training process occurs over 30 epochs with a batch size of 32. The '*ModelCheckpoint*' callback is used to save the best model based on validation loss.

Model Evaluation: After training, the model is evaluated on the testing dataset to measure its accuracy and performance.

```
model.compile(loss='binary_crossentropy',optimizer='adam', metrics=['accuracy'])

callback = tf.keras.callbacks.ModelCheckpoint(
    filepath='bestModel.h5',
    save_weights_only=False,
    monitor='val_loss',
    mode='min',
    save_best_only=True,
    verbose =1)
model.fit(x_train, y_train, validation_split=0.2, epochs=30, batch_size=32, callbacks=callback)

model.evaluate(x_test, y_test)
```

Loading and Detecting with the Best Model: The best model obtained during training is loaded using ‘load_model’ . The code then tests the model's performance on a few images from the testing dataset, displaying the images and the predicted eye status (open or closed).

```
# Loading the Model Performance

from keras.models import load_model
best_model = load_model('bestModel.h5')
best_model.evaluate(x_test, y_test)

# Changing the dimensions and Plotting the trained images

for i in x_test[0:5]:
   result = best_model.predict(np.expand_dims(i,0))
   plt.imshow(i)
   plt.show()

   if result > 0.5:
      print('Open')
   else:
      print("Closed")

# Accuracy

accuracy = best_model.evaluate(x_test, y_test)[1]
print("Accuracy:", accuracy)
```

Phase – 2 : Real-Time Drowsiness Detection System with Twilio Alert using CNN

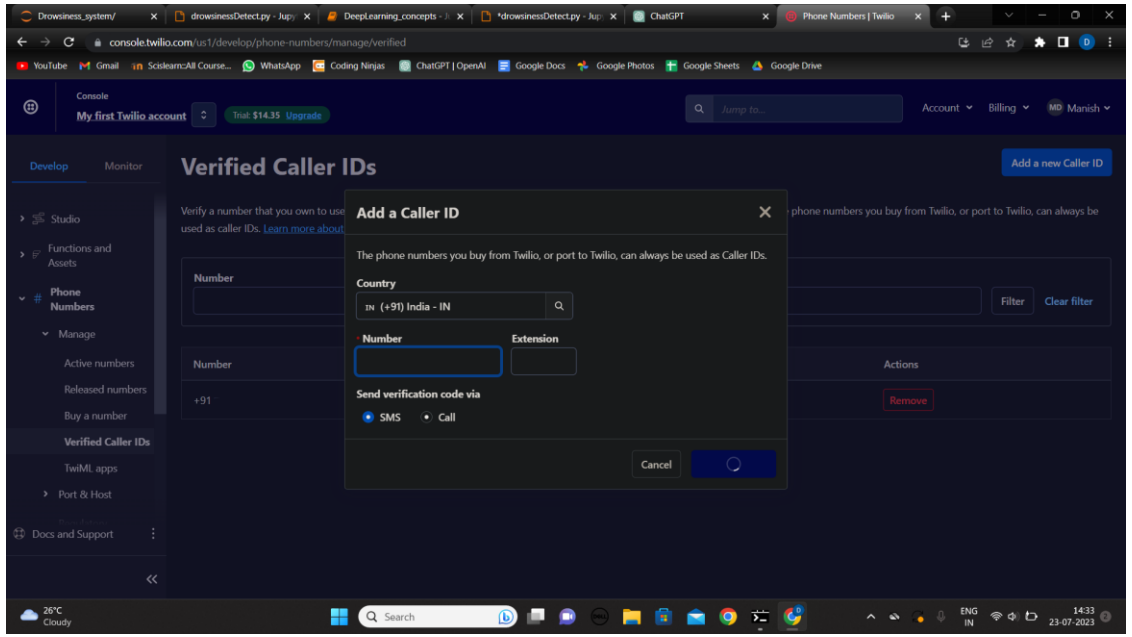
- **Importing Libraries:** The required libraries such as NumPy, pandas, OpenCV (cv2), Keras, Twilio, and others are imported to enable the functionality needed for the drowsiness detection system.
- **Loading the Model:** The pre-trained Keras model for eye classification (open/closed) is loaded using 'load_model()' from the Keras library.
- **Loading Haar Cascade Classifiers:** Pre-trained Haar Cascade classifiers for face and eye detection are loaded using the 'cv2.CascadeClassifier' from OpenCV.

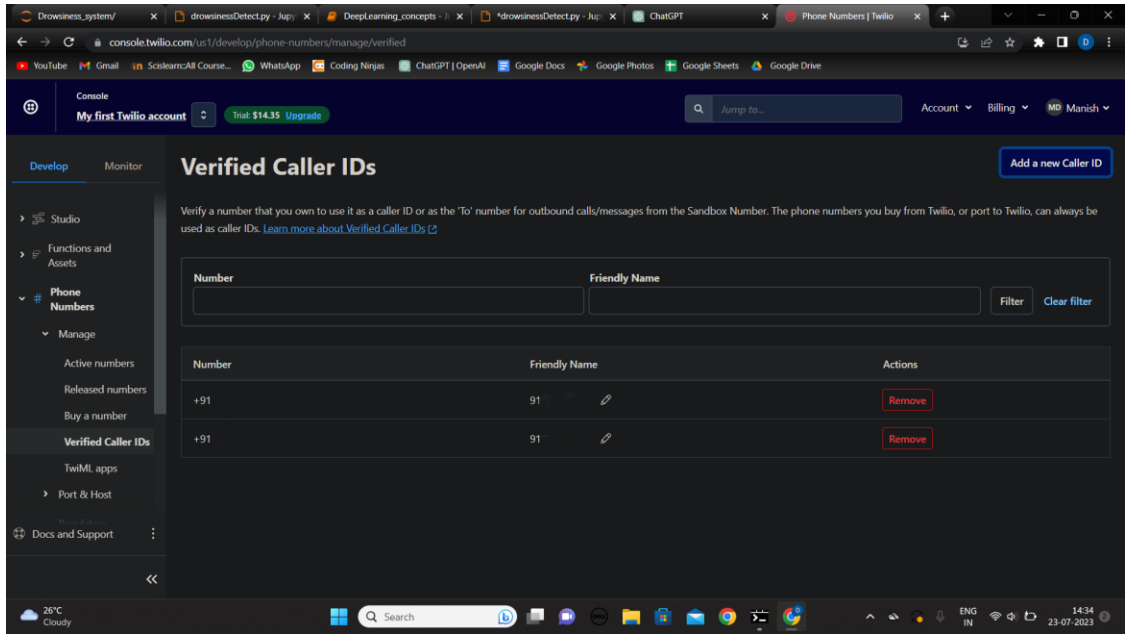
```
# Importing Libraries
import cv2
import numpy as np
from keras.models import load_model
from twilio.rest import Client

# Load the pre-trained Keras model for eye classification (open/closed)
model = load_model('bestModel.h5')  # Update with your model's path

# Load the pre-trained Haar Cascade classifiers for face and eye detection
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_eye.xml')
```

- **Accessing the Webcam:** The webcam is accessed using cv2.VideoCapture() to capture real-time video frames.
- **Initializing Variables:** Variables like closed_eyes_counter , drowsiness_threshold and alert are initialized.
- To make a call to a phone number we need a third-party service that provides a phone call API , i.e., Twilio. Twilio allows you to make and receive phone calls, send SMS messages, and more programmatically through their API. Go to the Twilio website **(https://www.twilio.com/)** and sign up for a Twilio account.

```
# Access the webcam
cap = cv2.VideoCapture(0)

# Initialize variables for drowsiness detection
closed_eyes_counter = 0
drowsiness_threshold = 30  # Number of consecutive closed eye frames before alerting
alert = False

# Twilio account settings
twilio_account_sid = 'YOUR_TWILIO_ACCOUNT_SID'
twilio_auth_token = 'YOUR_TWILIO_AUTH_TOKEN'
twilio_phone_number = 'YOUR_TWILIO_PHONE_NUMBER'  # The Twilio phone number you obtained earlier
phone_number_to_call = 'CUSTOMER_PHONE_NUMBER'   # Replace with the customer's phone number you want to call

twilio_client = Client(twilio_account_sid, twilio_auth_token)
```

You need to verify the customers Phone number in your twilio account , for sending the alert to that phone number.



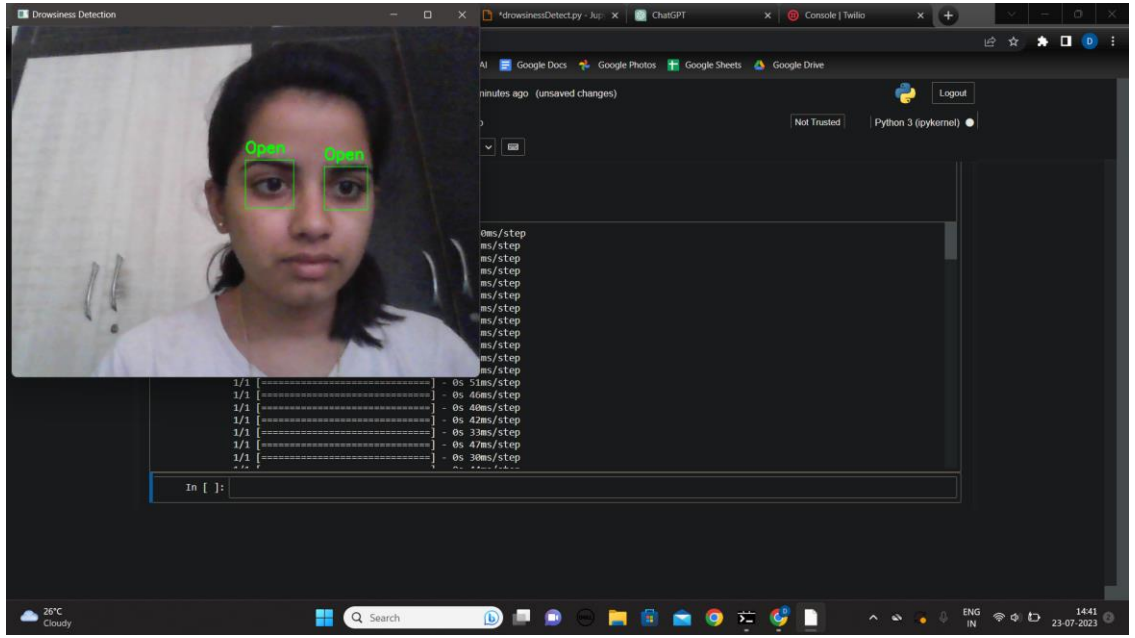


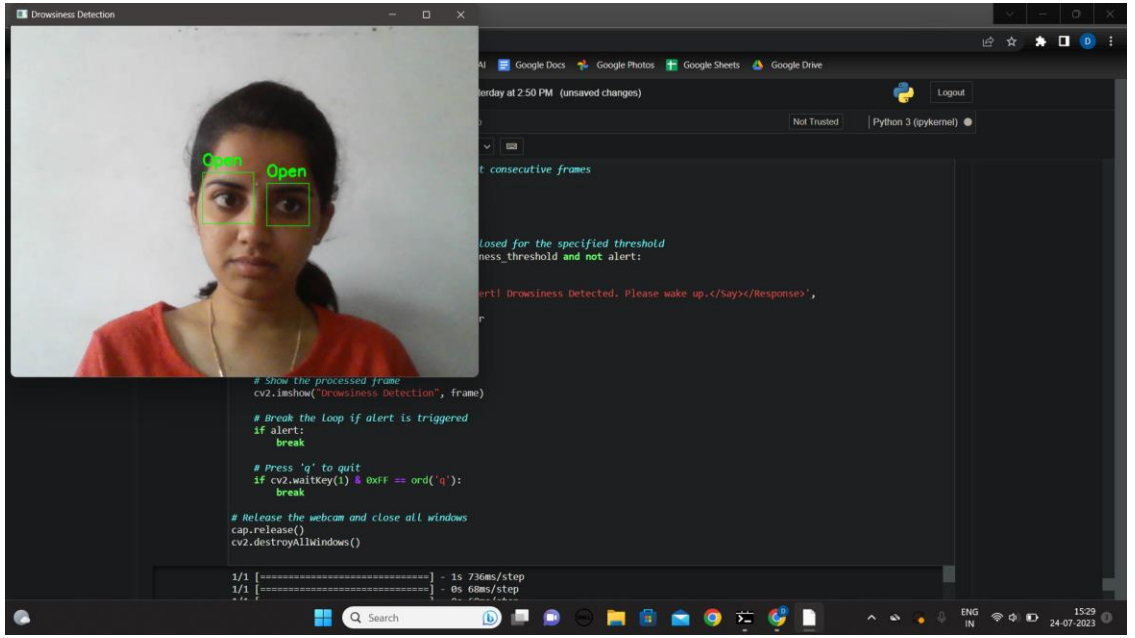
- **Main Loop**: The main loop runs continuously to process video frames from the webcam.
- **Detecting Faces and Eyes**: The Haar Cascade classifiers are used to detect faces and eyes in the grayscale frame.
- **Preprocessing Eye Images**: The detected eye regions are preprocessed to make them compatible with the input format expected by the loaded model.
- **Eye Classification**: The model predicts whether the detected eye is open or closed based on the preprocessed eye image.
- **Displaying Results**: The classification result is displayed on the frame with a rectangle around the eye and the corresponding label (Open or Closed).
- **Drowsiness Detection:** The code checks for consecutive closed eye frames and counts them using the closed_eyes_counter. If the number of closed eye frames exceeds the drowsiness_threshold , an alert is triggered using Twilio.
- **Real-Time Display:** The processed frame is shown in real-time using cv2.imshow() .
- **Alert and Quit:** If drowsiness is detected and the alert is sent, the loop is broken, and the program terminates after releasing the webcam.

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

    # Convert the frame to grayscale for faster processing
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Detect faces in the frame
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.3, minNeighbors=5, minSize=(30, 30))

    for (x, y, w, h) in faces:
        # Get the region of interest (ROI) within the face and detect eyes
        roi_gray = gray[y:y + h, x:x + w]
        eyes = eye_cascade.detectMultiScale(roi_gray)

        for (ex, ey, ew, eh) in eyes:
            # Preprocess the eye image for classification
            eye_roi = roi_gray[ey:ey + eh, ex:ex + ew]
            eye_roi = cv2.resize(eye_roi, (64, 64))  # Resize to (64, 64)
            eye_roi = eye_roi.astype("float") / 255.0
            eye_roi = np.expand_dims(eye_roi, axis=-1)  # Add a single channel (gray scale)
            eye_roi = np.expand_dims(eye_roi, axis=0)  # Add a batch dimension

            # Use the model to predict if the eye is open or closed
            prediction = model.predict(eye_roi)

            # Assuming the prediction shape is (batch_size, 1), update the label logic
            if prediction[0, 0] > 0.5:  # Assuming 0.5 is the threshold for binary classification
                label = "Open"
            else:
                label = "Closed"
```

```
# Draw a rectangle around the eye and display the label
        color = (0, 255, 0) if label == "Open" else (0, 0, 255)
        cv2.rectangle(frame, (x + ex, y + ey), (x + ex + ew, y + ey + eh), color, 1)
        cv2.putText(frame, label, (x + ex, y + ey - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.7, color, 2)

        # Check for closed eyes and count consecutive frames
        if label == "Closed":
            closed_eyes_counter += 1
        else:
            closed_eyes_counter = 0

        # Alert the driver if eyes are closed for the specified threshold
        if closed_eyes_counter >= drowsiness_threshold and not alert:
            # Make a call using Twilio
            twilio_client.calls.create(
                twiml='<Response><Say>Alert! Drowsiness Detected. Please wake up.</Say></Response>',
                to=phone_number_to_call,
                from_=twilio_phone_number
            )

            alert = True

    # Show the processed frame
    cv2.imshow("Drowsiness Detection", frame)

    # Break the loop if alert is triggered
    if alert:
        break

    # Press 'q' to quit
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the webcam and close all windows
cap.release()
cv2.destroyAllWindows()
```

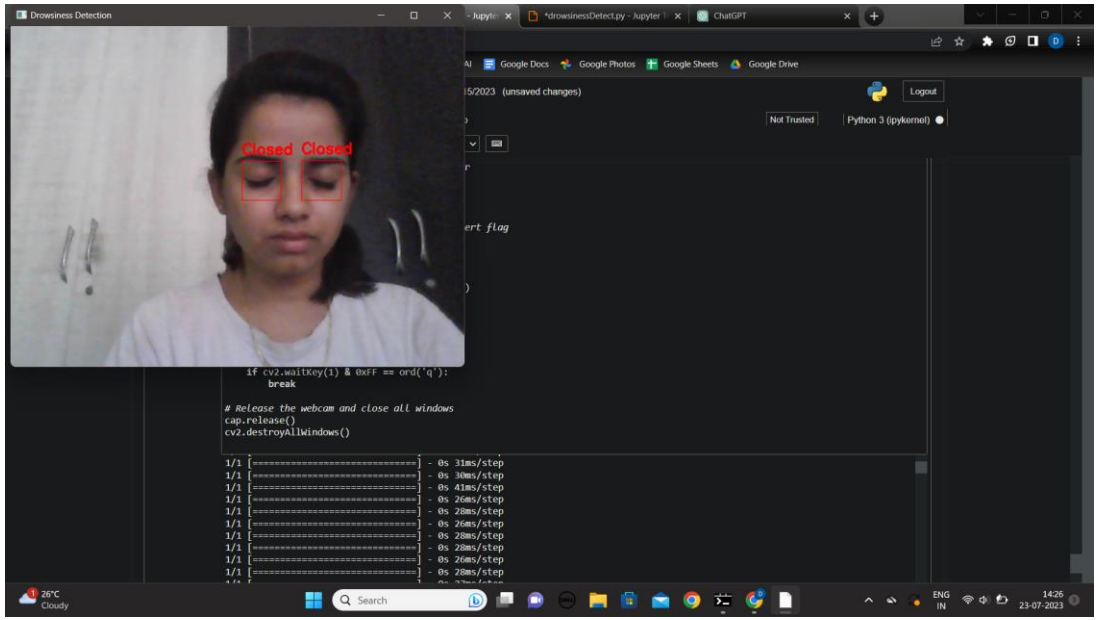
Samples of how our Drowsiness Detection system works :



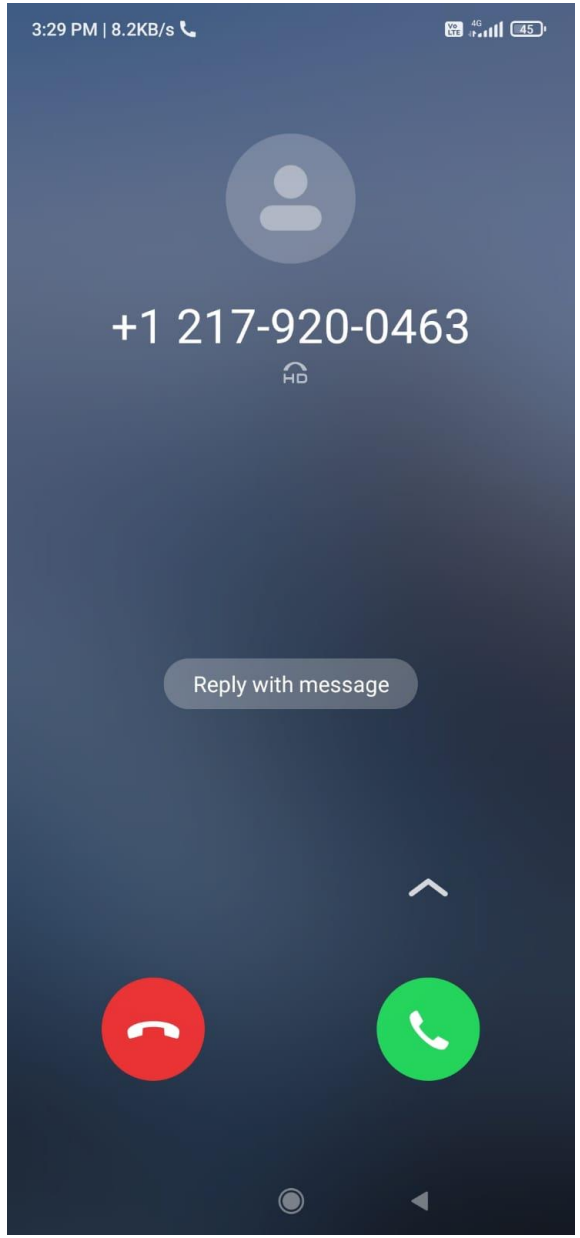


If the eyes are closed or semi-closed continuously upto 30 frames , the it triggers an alert and calls to the respective mobile number.

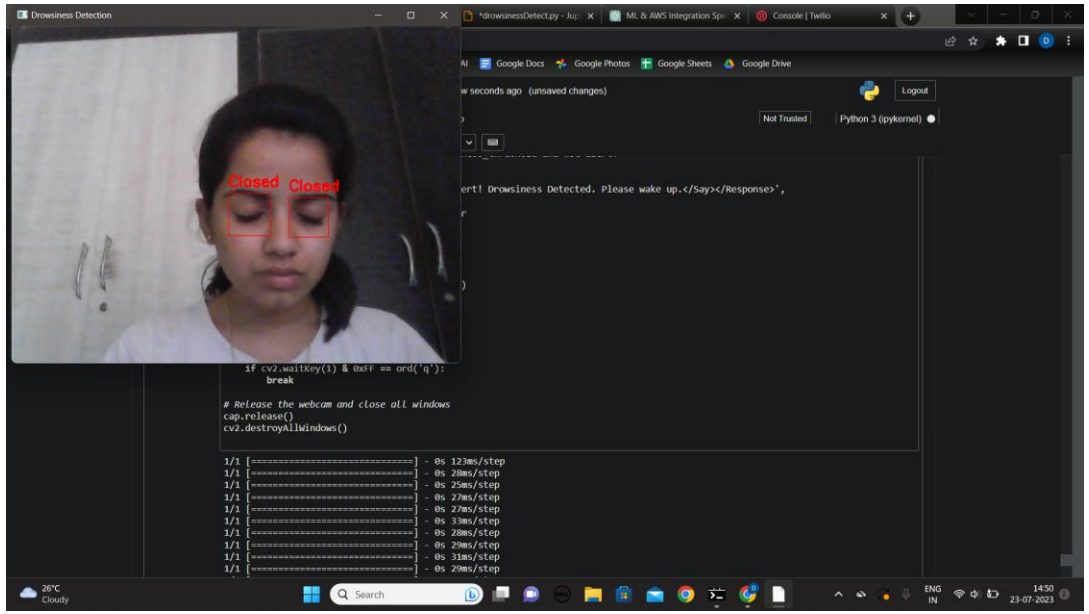
Note : The phone number must be verified in the twilio account.



Eyes are semi-closed



Eyes are closed



THANK YOU