**Backend Flask App**

import os

import cv2

import numpy as np

import tensorflow as tf

from flask import Flask, request, jsonify

from flask\_cors import CORS

from werkzeug.utils import secure\_filename

import mysql.connector

from model\_loader import load\_custom\_model

from retinaface import RetinaFace # Import RetinaFace for face detection

from twilio.rest import Client

# Twilio API credentials

account\_SID = "AC20da77581fc8e099047e434f24390993"

account\_token = "db205e7d8c74f34eb9ce700ea06d2590"

twilio\_phn = "+12674158112" # Twilio phone number

myphn = "+917092779445" # Your phone number

app = Flask(\_\_name\_\_)

CORS(app) # Enable CORS for all routes in the app

# MySQL configuration

mysql\_config = {

'host': 'localhost',

'user': 'root',

'password': '',

'database': 'logs\_final\_year'

}

# Load the custom model

model = load\_custom\_model()

UPLOAD\_FOLDER = r"E:\Saved Video" # Define upload folder for videos

ALLOWED\_EXTENSIONS = {'mp4', 'avi'} # Define allowed video file extensions

app.config['UPLOAD\_FOLDER'] = UPLOAD\_FOLDER

# Define the output folder paths for frames and faces

violent\_frames\_folder = r"E:\Violent Frames"

violent\_faces\_folder = r"E:\Violent Faces"

# Create the output folders if they don't exist

if not os.path.exists(violent\_frames\_folder):

os.makedirs(violent\_frames\_folder)

if not os.path.exists(violent\_faces\_folder):

os.makedirs(violent\_faces\_folder)

def allowed\_file(filename):

"""

Function to check if a file has an allowed extension.

Parameters:

filename (str): The name of the file to check.

Returns:

bool: True if the file has an allowed extension, False otherwise.

"""

return '.' in filename and \

filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

@app.route('/api/login', methods=['POST'])

def login\_user():

"""

Route for user login.

Returns:

JSON: User information if login successful, error message otherwise.

"""

try:

data = request.json

email = data.get('email')

password = data.get('password')

conn = mysql.connector.connect(\*\*mysql\_config)

cursor = conn.cursor(dictionary=True)

cursor.execute("SELECT \* FROM users WHERE email = %s", (email,))

user = cursor.fetchone()

if user:

stored\_password = user['password']

if password == stored\_password:

return jsonify({'user': user}), 200

else:

return jsonify({'error': 'Password does not match'}), 401

else:

return jsonify({'error': 'User not found'}), 404

except mysql.connector.Error as err:

return jsonify({'error': f"MySQL Error: {err}"}), 500

except Exception as e:

return jsonify({'error': str(e)}), 500

finally:

cursor.close()

conn.close()

@app.route('/api/register', methods=['POST'])

def register\_user():

"""

Route for user registration.

Returns:

JSON: Success message if registration is successful, error message otherwise.

"""

try:

data = request.json

name = data['name']

email = data['email']

phone = data['phone']

password = data['password']

conn = mysql.connector.connect(\*\*mysql\_config)

cursor = conn.cursor()

cursor.execute("INSERT INTO users (name, password, phone\_number, email) VALUES (%s, %s, %s, %s)", (name, password, phone, email))

conn.commit()

cursor.close()

conn.close()

return jsonify({'message': 'User registered successfully'}), 201

except Exception as e:

return jsonify({'error': str(e)}), 500

@app.route('/predict\_video', methods=['POST'])

def process\_video():

try:

# Check if the request contains a file

if 'file' not in request.files:

return jsonify({"error": "No file provided"}), 400

file = request.files['file']

# Check if the file has an allowed extension

if file and allowed\_file(file.filename):

# Save the uploaded file

filename = secure\_filename(file.filename)

file\_path = os.path.join(app.config['UPLOAD\_FOLDER'], filename)

file.save(file\_path)

# Predict violence in the uploaded video

violence\_predictions = predict\_violence(file\_path, model)

# Extract scores for both classes

violence\_scores = [pred[:2] for pred in violence\_predictions]

# Calculate the mean of both classes from the output predictions

average\_scores = np.mean(np.array(violence\_scores), axis=0)

average\_scores\_list = average\_scores.tolist()

# Determine violence detection based on predictions

violence\_detected = detect\_violence(average\_scores[0], average\_scores[1])

if violence\_detected:

# Send alert message via Twilio when violence is detected

client = Client(account\_SID, account\_token)

message = client.messages.create(

body="Violence is detected",

from\_=twilio\_phn,

to=myphn)

# Extract frames and faces when violence is detected

extract\_frames\_and\_faces(file\_path)

# Return the average scores along with violence detection result as JSON response

return jsonify({

"Average Human Assault score": average\_scores\_list[0],

"Average no-human-assault score": average\_scores\_list[1],

"Human Assault detected": violence\_detected

}), 200

else:

return jsonify({"error": "Invalid file format"}), 400

except Exception as e:

print("Error:", e)

return jsonify({"error": "Internal server error"}), 500

def predict\_violence(video\_path, model, n\_frames=10, output\_size=(224, 224), batch\_size=8):

"""

Predict violence in a video by analyzing its frames.

Args:

video\_path (str): Path to the video file.

model: Trained model for violence prediction.

n\_frames (int): Number of frames to process per batch.

output\_size (tuple): Size of the output frame images.

batch\_size (int): Size of the batch for prediction.

Returns:

list: Predictions for each frame in the video.

"""

# Read the video file

src = cv2.VideoCapture(video\_path)

video\_length = int(src.get(cv2.CAP\_PROP\_FRAME\_COUNT))

# Create batches of frames

frames\_batch = []

for \_ in range(0, video\_length, n\_frames):

batch\_frames = []

for \_ in range(n\_frames):

ret, frame = src.read()

if not ret:

break

frame = format\_frames(frame, output\_size)

batch\_frames.append(frame)

frames\_batch.append(batch\_frames)

src.release()

# Pad the last batch if necessary

if len(frames\_batch[-1]) < n\_frames:

padding = n\_frames - len(frames\_batch[-1])

frames\_batch[-1].extend([np.zeros\_like(frames\_batch[-1][0])] \* padding)

# Convert to numpy array

frames\_batch = np.array(frames\_batch)

# Predict violence for each batch

predictions = []

for batch in range(0, len(frames\_batch), batch\_size):

batch\_predictions = model.predict(frames\_batch[batch:batch+batch\_size])

predictions.extend(batch\_predictions)

return predictions

def format\_frames(frame, output\_size):

"""

Pad and resize an image from a video.

Args:

frame: Image that needs to be resized and padded.

output\_size: Pixel size of the output frame image.

Returns:

Formatted frame with padding of specified output size.

"""

frame = tf.image.convert\_image\_dtype(frame, tf.float32)

frame = tf.image.resize\_with\_pad(frame, \*output\_size)

return frame

def detect\_violence(average\_scores\_0, average\_scores\_1):

"""

Determine violence detection based on average scores.

Args:

average\_scores\_0 (float): Average score for human-assault class.

average\_scores\_1 (float): Average score for no-human-assault class.

Returns:

bool: True if violence is detected, False otherwise.

"""

if average\_scores\_0 > average\_scores\_1:

return True

else:

return False

def extract\_frames\_and\_faces(video\_path):

"""

Extract frames and faces from a video when violence is detected.

Args:

video\_path (str): Path to the video file.

"""

# Open the video file

vid = cv2.VideoCapture(video\_path)

frame\_count = 0

interval = 15 # Interval to save frames

while True:

ret, frame = vid.read()

if not ret:

break

# Save frames at the specified interval

if frame\_count % interval == 0:

# Save the frame

frame\_filename = os.path.join(violent\_frames\_folder, f"frame\_{frame\_count}.jpg")

cv2.imwrite(frame\_filename, frame)

# Extract faces from the frame

num\_faces = extract\_faces(frame, frame\_count)

frame\_count += 1

vid.release()

def extract\_faces(image, frame\_count):

"""

Extract faces from a frame using RetinaFace.

Args:

image: Frame image.

frame\_count: Index of the frame.

Returns:

int: Number of faces detected in the frame.

"""

obj = RetinaFace.detect\_faces(image)

if obj is not None:

for i, (\_, value) in enumerate(obj.items()):

facial\_area = value["facial\_area"]

x1, y1, x2, y2 = facial\_area

# Increase the size of the extracted facial area by expanding the bounding box

expansion\_factor = 0.2

width = x2 - x1

height = y2 - y1

x1 = max(0, int(x1 - expansion\_factor \* width))

y1 = max(0, int(y1 - expansion\_factor \* height))

x2 = min(image.shape[1], int(x2 + expansion\_factor \* width))

y2 = min(image.shape[0], int(y2 + expansion\_factor \* height))

face\_image = image[y1:y2, x1:x2]

# Ensure that the extracted face is not empty

if face\_image.size != 0:

# Save the face image to the Violent Faces folder

face\_filename = os.path.join(violent\_faces\_folder, f"frame\_{frame\_count}\_face\_{i}.jpg")

cv2.imwrite(face\_filename, face\_image)

return len(obj) if obj is not None else 0

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)