Drowsy Driving Detector - Application Documentation

Kieran Chai and Htet Wai Yan

1. Overview

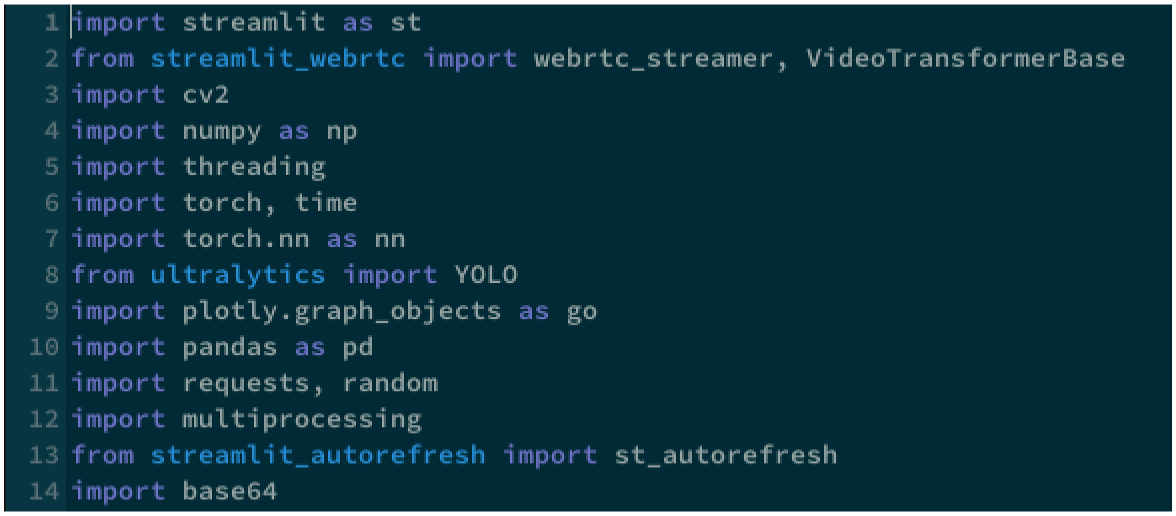
This document describes a Streamlit web application designed for real-time drowsiness detection using a webcam feed. The application performs the following key functions:

1. **Webcam Access:** Captures video input from the user's webcam.
2. **Face Detection:** Utilizes a YOLO (You Only Look Once) model to detect faces within the video frames.
3. **Drowsiness Prediction:** For each detected face, it uses a separate pre-trained model (loaded from .pth) to predict a drowsiness score (ranging from 0.0 to 1.0, where higher values indicate greater drowsiness).
4. **Real-time Feedback:** Displays the processed webcam feed, drawing bounding boxes around detected faces and annotating them with the calculated drowsiness score.
5. **Metrics Visualization:** Presents real-time metrics including:
    a. A **gauge chart** showing the current drowsiness level.
    b. A **line chart** displaying the historical trend of drowsiness scores, along with a rolling average to smooth out fluctuations.
6. **Alerting System:** Implements warnings and critical alerts:
    a. A visual warning for instantaneously high drowsiness levels.
    b. A critical alert (visual and audio) if the drowsiness score remains above a specified threshold for a defined duration (sustained drowsiness).
7. **User Interface:** Provides a clean, styled web interface built with Streamlit, including a loading animation while models initialize and a button to reset the detection history.

The primary goal is to provide drivers (or observers) with immediate feedback on potential drowsiness to enhance safety.

2. Imports and Setup

The application begins by importing necessary libraries.



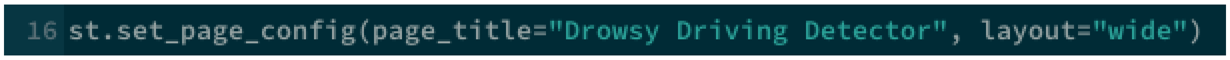
```
import streamlit as st
from streamlit_webrtc import webrtc_streamer, VideoTransformerBase
import cv2
import numpy as np
import threading
import torch, time
import torch.nn as nn
from ultralytics import YOLO
import plotly.graph_objects as go
import pandas as pd
import requests, random
import multiprocessing
from streamlit_autorefresh import st_autorefresh
import base64
```

- **streamlit (st):** The core library for building the web application UI.
- **streamlit_webrtc:** Enables real-time video streaming and processing within Streamlit. webrtc_streamer handles the connection, and VideoTransformerBase is the base class for frame processing.
- **cv2 (OpenCV):** Used for image processing tasks like reading frames, resizing, color conversion, and drawing annotations (rectangles, text).
- **numpy (np):** Essential for numerical operations, especially manipulating image arrays.
- **threading:** Used for creating locks (threading.Lock) to safely manage access to shared data (like the drowsiness history) between different threads (e.g., the main Streamlit thread and the video processing thread).
- **torch, time:** PyTorch is the deep learning framework used for loading and running the drowsiness prediction model. time is used for tracking time, particularly for the sustained drowsiness detection logic and the loading screen.
- **ultralytics (YOLO):** Provides the implementation for the YOLO object detection model, specifically used here for face detection.
- **plotly.graph_objects (go):** Used to create interactive charts (gauge and line chart) for visualizing drowsiness metrics.
- **pandas (pd):** Used for data manipulation, particularly for calculating the rolling average of the drowsiness history.
- **streamlit_autorefresh:** A Streamlit component that automatically triggers reruns of the script at a specified interval, used here to update the metrics display.

- **base64:** Used to encode the alert sound file for embedding directly into HTML for autoplay.

3. Page Configuration

The basic Streamlit page settings are configured first.

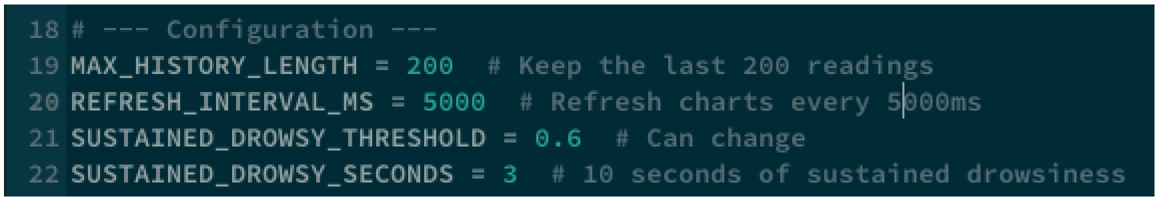


```
st.set_page_config(page_title="Drowsy Driving Detector", layout="wide")
```

- **st.set_page_config:** Sets global configurations for the Streamlit page.
  - page_title: Sets the title that appears in the browser tab.
  - layout="wide": Makes the app content use the full width of the browser window.

4. Application Configuration Constants

These constants define key parameters for the application's behavior.

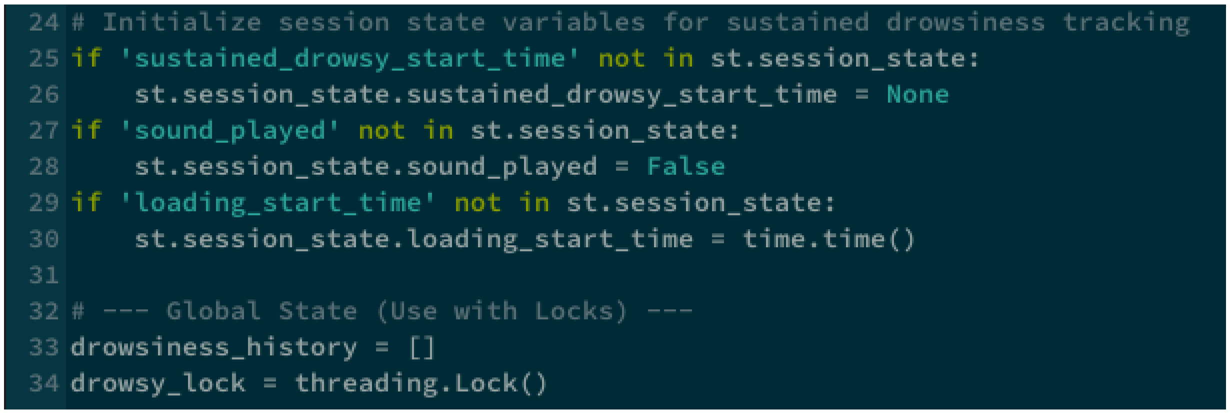


```
# --- Configuration ---
MAX_HISTORY_LENGTH = 200  # Keep the last 200 readings
REFRESH_INTERVAL_MS = 5000  # Refresh charts every 5000ms
SUSTAINED_DROWSY_THRESHOLD = 0.6  # Can change
SUSTAINED_DROWSY_SECONDS = 3  # 10 seconds of sustained drowsiness
```

- **MAX_HISTORY_LENGTH:** Limits the number of drowsiness readings stored in memory to prevent it from growing indefinitely. Older readings are discarded.
- **REFRESH_INTERVAL_MS:** The interval (in milliseconds) at which the metrics section of the app will automatically refresh using st_autorefresh.
- **SUSTAINED_DROWSY_THRESHOLD:** The drowsiness score threshold. If the score stays *above* this value continuously, the sustained drowsiness timer starts.
- **SUSTAINED_DROWSY_SECONDS:** The duration (in seconds) the drowsiness score must remain above SUSTAINED_DROWSY_THRESHOLD before the critical sustained drowsiness alert is triggered.

5. State Management

Streamlit reruns the entire script on user interaction or via auto-refresh. Therefore, state needs to be preserved across these reruns. This application uses both Streamlit's session_state and a module-level variable with a lock.



```python
# Initialize session state variables for sustained drowsiness tracking
if 'sustained_drowsy_start_time' not in st.session_state:
    st.session_state.sustained_drowsy_start_time = None
if 'sound_played' not in st.session_state:
    st.session_state.sound_played = False
if 'loading_start_time' not in st.session_state:
    st.session_state.loading_start_time = time.time()

# --- Global State (Use with Locks) ---
drowsiness_history = []
drowsy_lock = threading.Lock()
```

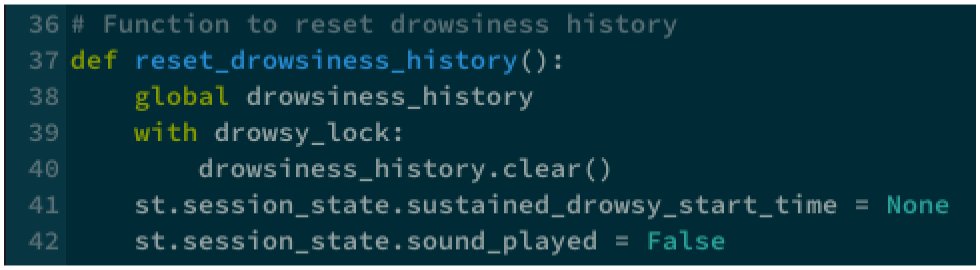
- **st.session_state:** A dictionary-like object provided by Streamlit to store variables that persist across script reruns within a single user session.
    - sustained_drowsy_start_time: Stores the timestamp when drowsiness first exceeded the SUSTAINED_DROWSY_THRESHOLD. None if not currently in a sustained state.
    - sound_played: A boolean flag to ensure the alert sound only plays once per sustained drowsiness event. Reset when the drowsiness level drops below the threshold.
- **drowsy_lock:** A threading.Lock object. This is essential because streamlit-webrtc runs the video processing (VideoTransformer.transform) in a separate thread. The lock ensures that appending data to the drowsiness_history list (which happens in the video thread) and reading/copying it (which happens in the main Streamlit thread during metric updates) do not occur simultaneously, preventing race conditions and data corruption.

6. Helper Functions

Several utility functions are defined for specific tasks.

6.1. reset_drowsiness_history()

This function clears the recorded drowsiness data and resets the sustained drowsiness state.

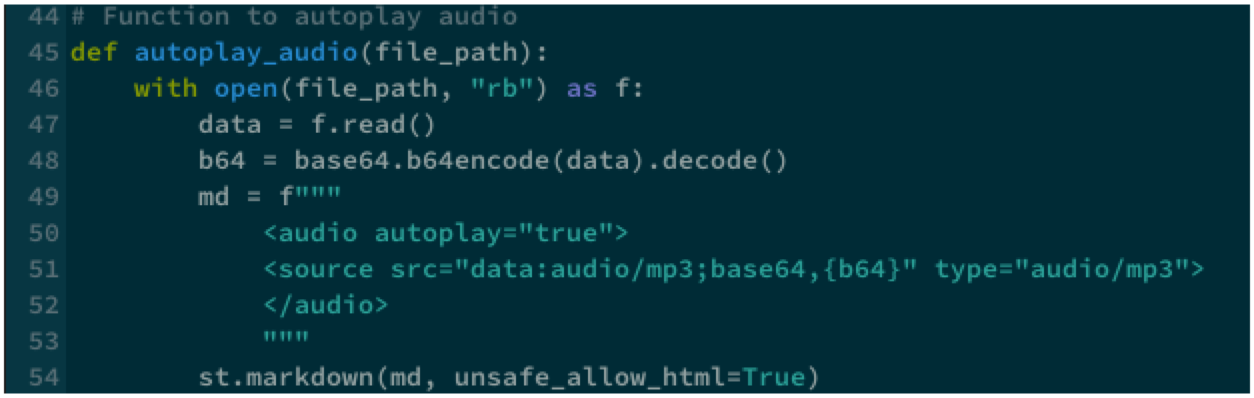


```python
# Function to reset drowsiness history
def reset_drowsiness_history():
    global drowsiness_history
    with drowsy_lock:
        drowsiness_history.clear()
    st.session_state.sustained_drowsy_start_time = None
    st.session_state.sound_played = False
```

- Acquires the drowsy_lock to prevent conflicts with the video processing thread.
- Clears the drowsiness_history list stored in st.session_state.
- Resets the sustained_drowsy_start_time and sound_played flags in st.session_state to their initial states.

6.2. autoplay_audio()

Embeds and automatically plays an audio file within the Streamlit app.

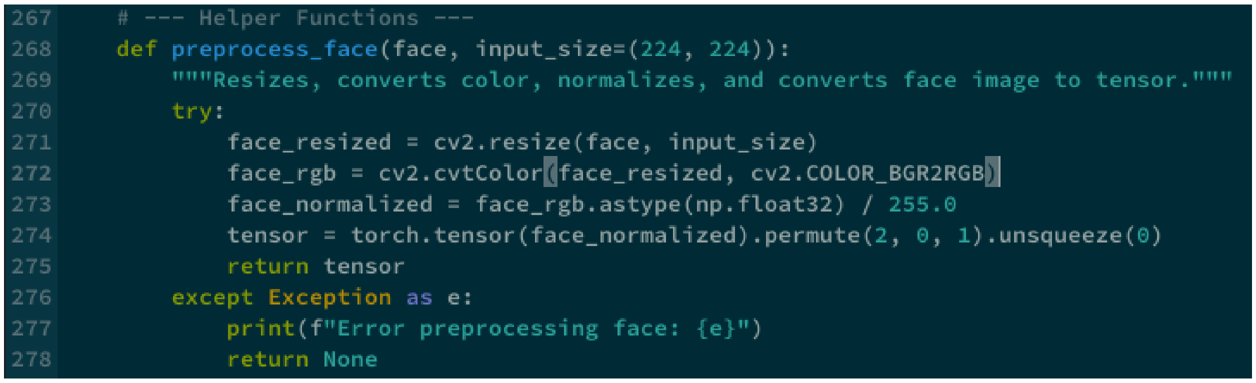


```python
# Function to autoplay audio
def autoplay_audio(file_path):
    with open(file_path, "rb") as f:
        data = f.read()
        b64 = base64.b64encode(data).decode()
        md = f"""
            <audio autoplay="true">
            <source src="data:audio/mp3;base64,{b64}" type="audio/mp3">
            </audio>
            """
        st.markdown(md, unsafe_allow_html=True)
```

- Takes the file_path of an audio file (e.g., "alert_sound.mp3").
- Opens the file in binary read mode ("rb").
- Reads the file content and encodes it into Base64 format.
- Creates an HTML string containing an <audio> tag with the autoplay attribute. The audio source (src) is set to a data URL containing the Base64-encoded audio data.
- Uses st.markdown with unsafe_allow_html=True to render the HTML, causing the browser to automatically play the embedded audio.

6.3. preprocess_face()

Prepares a detected face image patch for input into the drowsiness prediction model.

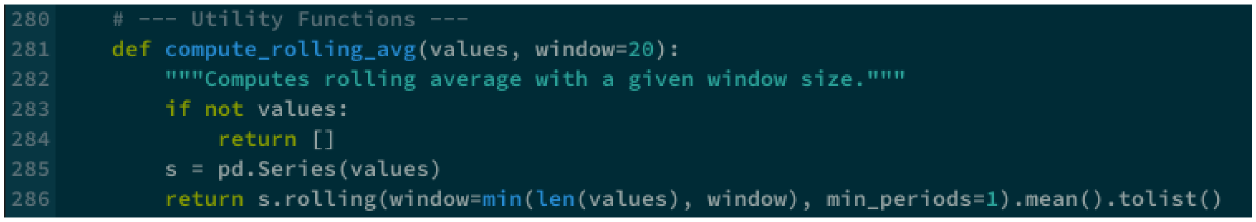


```python
# --- Helper Functions ---
def preprocess_face(face, input_size=(224, 224)):
    """Resizes, converts color, normalizes, and converts face image to tensor."""
    try:
        face_resized = cv2.resize(face, input_size)
        face_rgb = cv2.cvtColor(face_resized, cv2.COLOR_BGR2RGB)
        face_normalized = face_rgb.astype(np.float32) / 255.0
        tensor = torch.tensor(face_normalized).permute(2, 0, 1).unsqueeze(0)
        return tensor
    except Exception as e:
        print(f"Error preprocessing face: {e}")
        return None
```

- Takes a face image (face, expected to be a NumPy array) and the target input_size.
- Resizes the image using cv2.resize.
- Converts the image from BGR (OpenCV's default color order) to RGB using cv2.cvtColor.
- Normalizes the pixel values to the range [0.0, 1.0] by dividing by 255.0. This is a common step for neural network inputs.
- Converts the NumPy array to a PyTorch tensor using torch.tensor.
- Permutes the dimensions from Height x Width x Channels (HWC) to Channels x Height x Width (CHW), which is the standard format for PyTorch convolutional layers.
- Adds a batch dimension at the beginning (unsqueeze(0)) as models typically expect batches of images.
- Includes error handling and returns None if any step fails.

6.4. compute_rolling_avg()

Calculates the rolling average of a list of values.



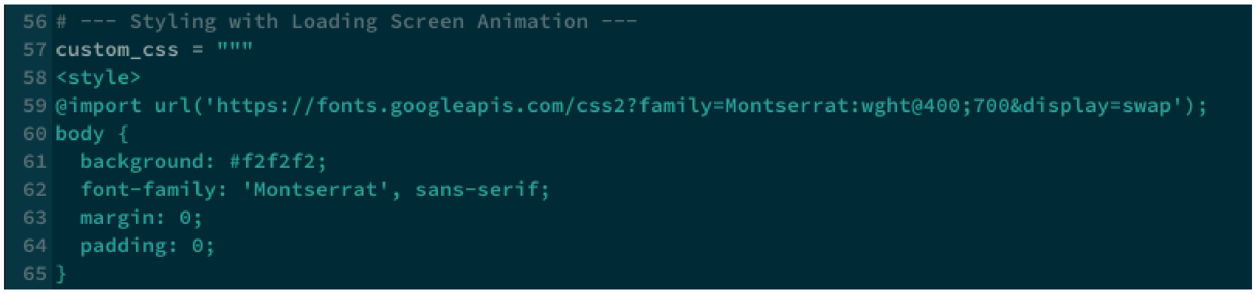
```python
# --- Utility Functions ---
def compute_rolling_avg(values, window=20):
    """Computes rolling average with a given window size."""
    if not values:
        return []
    s = pd.Series(values)
    return s.rolling(window=min(len(values), window), min_periods=1).mean().tolist()
```

- Takes a list of values and a window size.
- Uses pandas.Series to facilitate the rolling calculation.
- Applies the .rolling() method with the specified window size (capped at the length of the data) and min_periods=1 (to produce output even with fewer data points than the window size).
- Calculates the .mean() for each window.

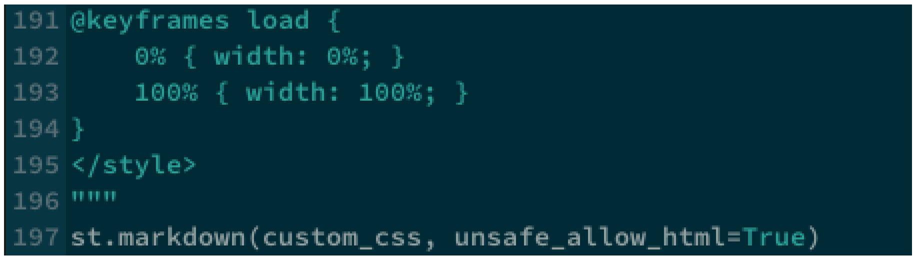
- Returns the result as a list. This helps in smoothing out short-term fluctuations in the drowsiness score history.

7. Styling and Loading Screen

Custom CSS is injected to style the application components and create an animated loading screen.



```
# --- Styling with Loading Screen Animation ---
custom_css = """
<style>
@import url('https://fonts.googleapis.com/css2?family=Montserrat:wght@400;700&display=swap');
body {
  background: #f2f2f2;
  font-family: 'Montserrat', sans-serif;
  margin: 0;
  padding: 0;
}
```

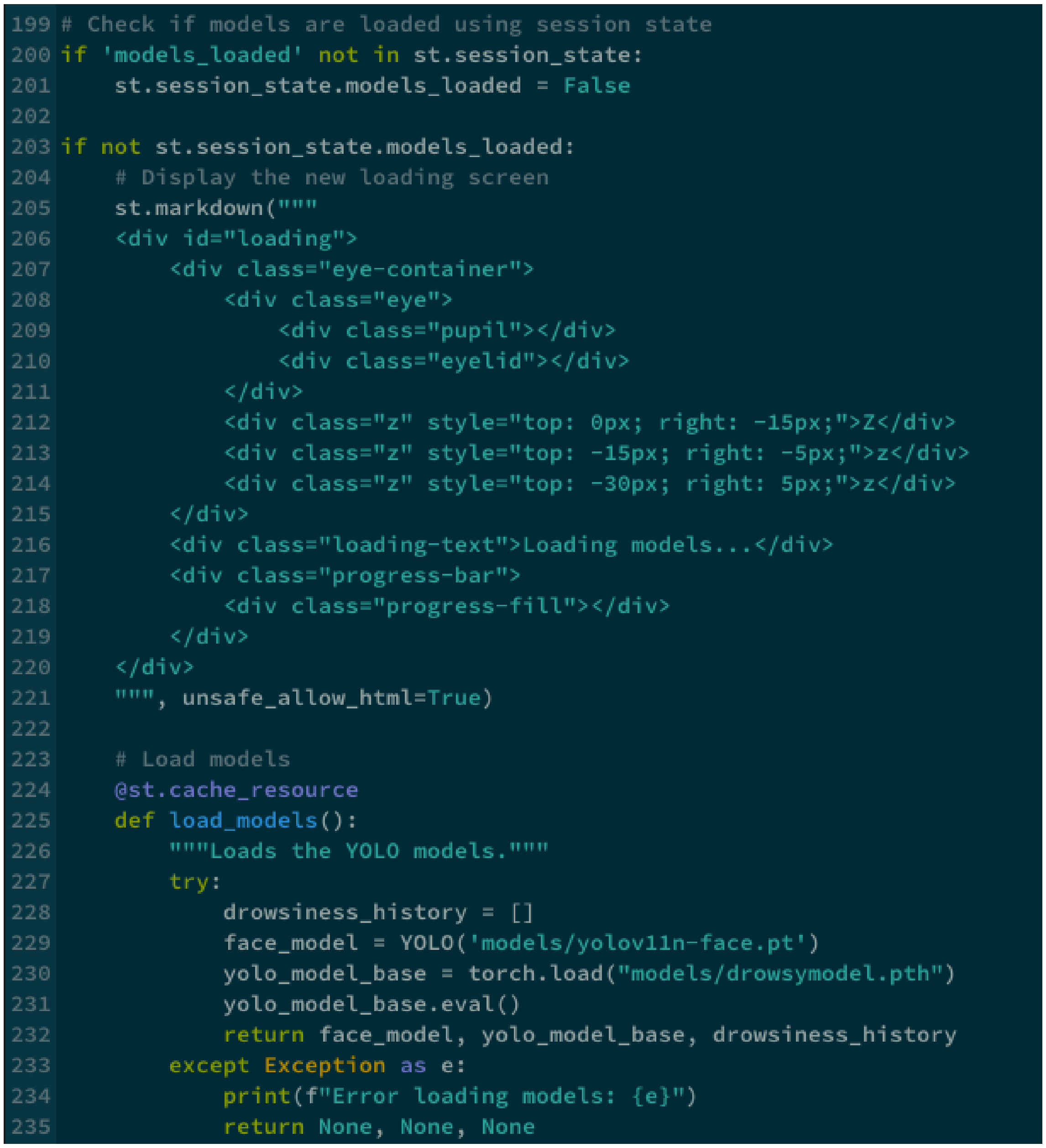


```
@keyframes load {
    0% { width: 0%; }
    100% { width: 100%; }
}
</style>
"""
st.markdown(custom_css, unsafe_allow_html=True)
```

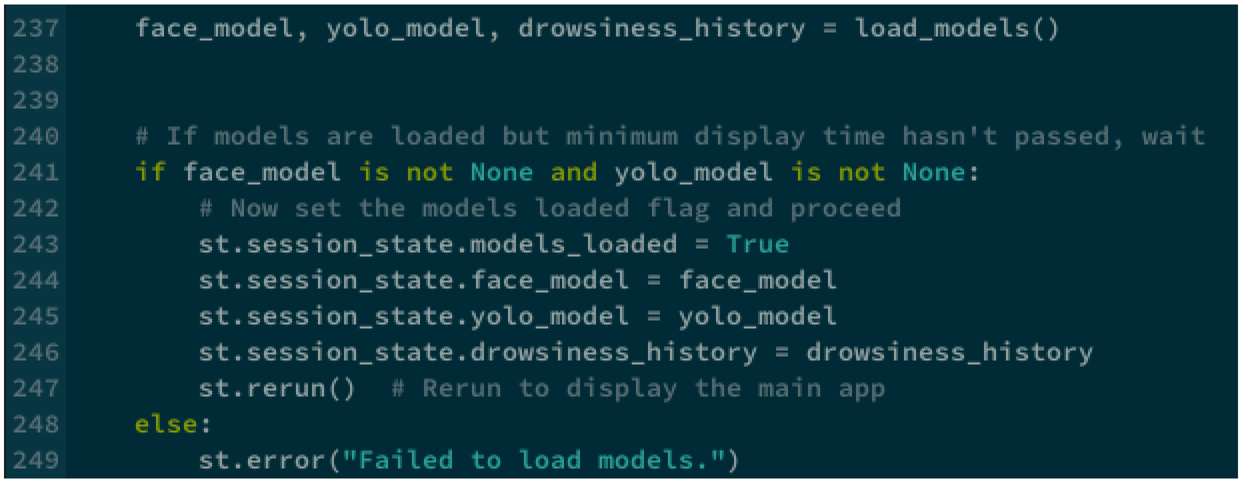
- The custom_css string contains standard CSS rules to define fonts, background colors, element layouts (header, container, cards), margins, padding, and box shadows for a polished look.
- Crucially, it includes CSS rules and @keyframes animations (blink, z-fade, load) to create the visual elements of the loading screen:
    - An "eye" with a blinking "eyelid".
    - Floating "Z" characters.
    - A loading progress bar animation.
- st.markdown(custom_css, unsafe_allow_html=True) injects this CSS into the Streamlit application's HTML structure.

8. Model Loading

This section handles the loading of the machine learning models, displaying the loading screen while this occurs. It uses session state to ensure models are loaded only once per session.



```python
# Check if models are loaded using session state
if 'models_loaded' not in st.session_state:
    st.session_state.models_loaded = False

if not st.session_state.models_loaded:
    # Display the new loading screen
    st.markdown("""
    <div id="loading">
        <div class="eye-container">
            <div class="eye">
                <div class="pupil"></div>
                <div class="eyelid"></div>
            </div>
            <div class="z" style="top: 0px; right: -15px;">Z</div>
            <div class="z" style="top: -15px; right: -5px;">z</div>
            <div class="z" style="top: -30px; right: 5px;">z</div>
        </div>
        <div class="loading-text">Loading models...</div>
        <div class="progress-bar">
            <div class="progress-fill"></div>
        </div>
    </div>
    """, unsafe_allow_html=True)

    # Load models
    @st.cache_resource
    def load_models():
        """Loads the YOLO models."""
        try:
            drowsiness_history = []
            face_model = YOLO('models/yolov11n-face.pt')
            yolo_model_base = torch.load("models/drowsymodel.pth")
            yolo_model_base.eval()
            return face_model, yolo_model_base, drowsiness_history
        except Exception as e:
            print(f"Error loading models: {e}")
            return None, None, None
```



```
    face_model, yolo_model, drowsiness_history = load_models()


    # If models are loaded but minimum display time hasn't passed, wait
    if face_model is not None and yolo_model is not None:
        # Now set the models loaded flag and proceed
        st.session_state.models_loaded = True
        st.session_state.face_model = face_model
        st.session_state.yolo_model = yolo_model
        st.session_state.drowsiness_history = drowsiness_history
        st.rerun()  # Rerun to display the main app
    else:
        st.error("Failed to load models.")
```

- **Loading Guard:** It first checks the st.session_state.models_loaded flag. If False, it proceeds to display the loading screen and load the models.
- **Loading Screen Display:** Uses st.markdown to render the pre-defined HTML for the loading animation.
- **@st.cache_resource:** This Streamlit decorator is crucial. It caches the result of load_models(). This means the potentially time-consuming process of loading models from disk and initializing them in memory happens only once per session. Subsequent calls to load_models() return the cached objects instantly.
- **Model Loading Implementation:**
    - Inside load_models(), it attempts to load the face detection model (YOLO(...)) and the drowsiness prediction model (torch.load(...)).
    - map_location=torch.device('cpu') is added to torch.load for better compatibility, allowing models trained on a GPU to be loaded on a CPU-only machine.
    - yolo_model_base.eval() sets the PyTorch model to evaluation mode, which is important for consistent inference results (e.g., it disables layers like dropout).
    - Robust try...except blocks catch potential errors during loading (e.g., FileNotFoundError, general exceptions) and display informative messages using st.error.
    - If successful, it returns the loaded models and an empty list intended to initialize the drowsiness history.
- **State Update and Rerun:** If load_models() returns valid models:
    - The loaded models and the initial empty history list are stored in st.session_state.

- The models_loaded flag in st.session_state is set to True.
- st.rerun() is called. This tells Streamlit to stop the current script run and immediately start a new run from the top. On the next run, the if not st.session_state.models_loaded: condition will be false, skipping the loading process and proceeding to render the main application UI.

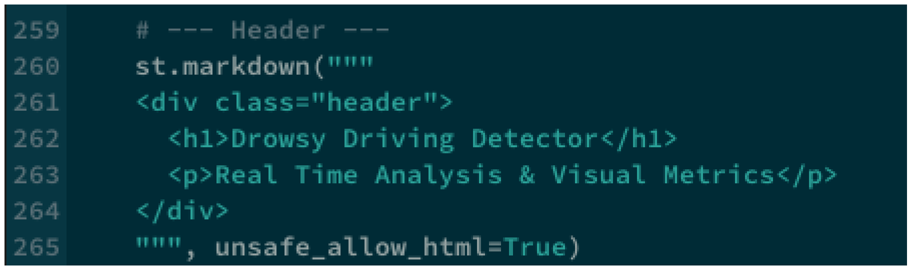
- **Error Handling:** If models fail to load, an error is displayed, and st.stop() prevents the rest of the script from executing, as the core functionality depends on the models.

9. Main Application UI and Logic

This part executes only after the models are successfully loaded.

9.1. Header

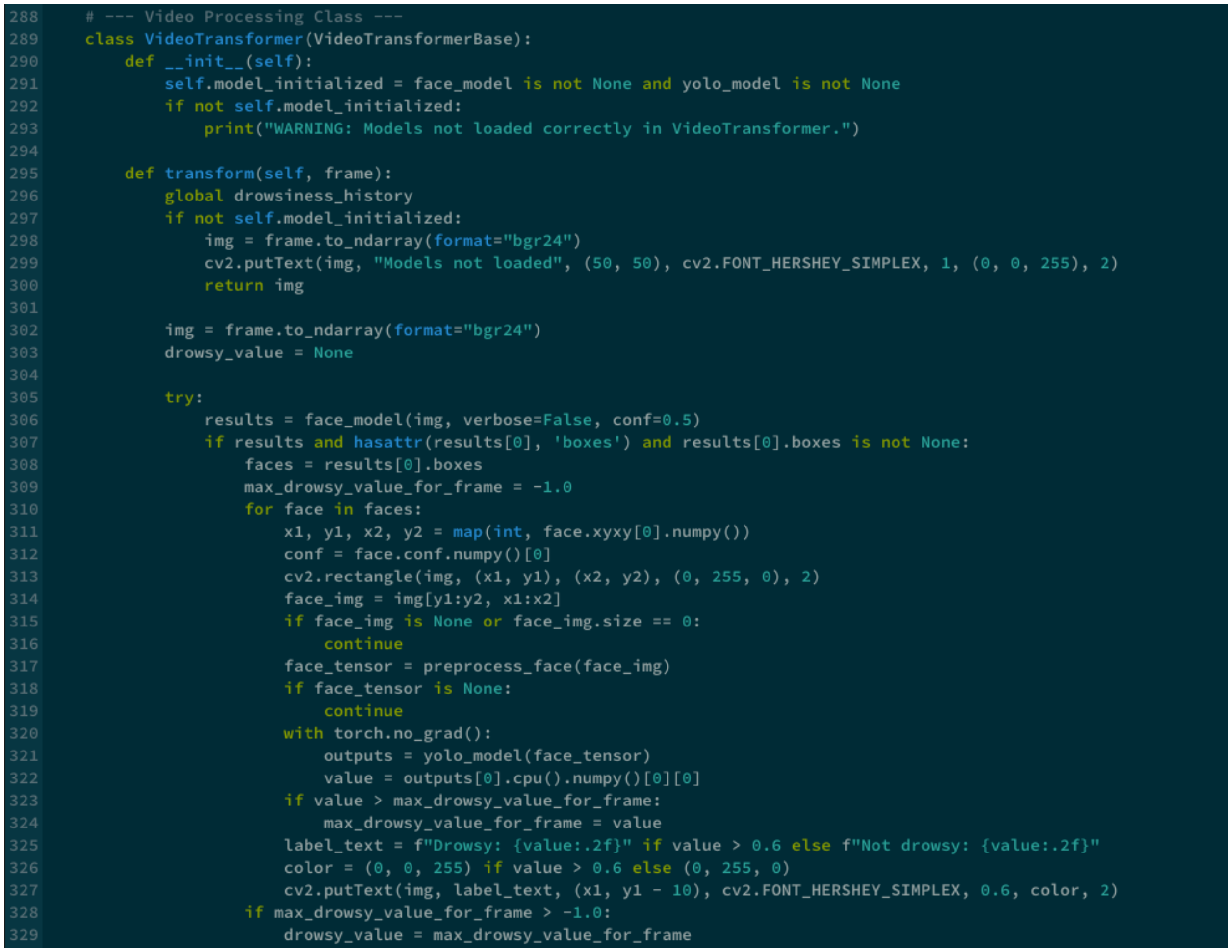
Displays the main title and subtitle of the application using styled HTML.



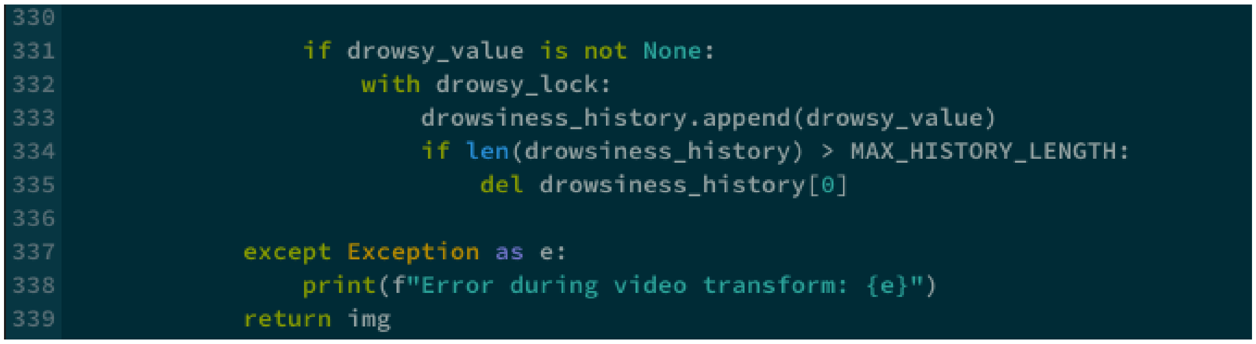
```
    # --- Header ---
    st.markdown("""
    <div class="header">
      <h1>Drowsy Driving Detector</h1>
      <p>Real Time Analysis & Visual Metrics</p>
    </div>
    """, unsafe_allow_html=True)
```

9.2. Video Processing Class (VideoTransformer)

This class handles the frame-by-frame processing logic for the webcam stream. It inherits from streamlit_webrtc.VideoTransformerBase.



```python
# --- Video Processing Class ---
class VideoTransformer(VideoTransformerBase):
    def __init__(self):
        self.model_initialized = face_model is not None and yolo_model is not None
        if not self.model_initialized:
            print("WARNING: Models not loaded correctly in VideoTransformer.")

    def transform(self, frame):
        global drowsiness_history
        if not self.model_initialized:
            img = frame.to_ndarray(format="bgr24")
            cv2.putText(img, "Models not loaded", (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
            return img

        img = frame.to_ndarray(format="bgr24")
        drowsy_value = None

        try:
            results = face_model(img, verbose=False, conf=0.5)
            if results and hasattr(results[0], 'boxes') and results[0].boxes is not None:
                faces = results[0].boxes
                max_drowsy_value_for_frame = -1.0
                for face in faces:
                    x1, y1, x2, y2 = map(int, face.xyxy[0].numpy())
                    conf = face.conf.numpy()[0]
                    cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0), 2)
                    face_img = img[y1:y2, x1:x2]
                    if face_img is None or face_img.size == 0:
                        continue
                    face_tensor = preprocess_face(face_img)
                    if face_tensor is None:
                        continue
                    with torch.no_grad():
                        outputs = yolo_model(face_tensor)
                        value = outputs[0].cpu().numpy()[0][0]
                    if value > max_drowsy_value_for_frame:
                        max_drowsy_value_for_frame = value
                    label_text = f"Drowsy: {value:.2f}" if value > 0.6 else f"Not drowsy: {value:.2f}"
                    color = (0, 0, 255) if value > 0.6 else (0, 255, 0)
                    cv2.putText(img, label_text, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.6, color, 2)
                if max_drowsy_value_for_frame > -1.0:
                    drowsy_value = max_drowsy_value_for_frame
```



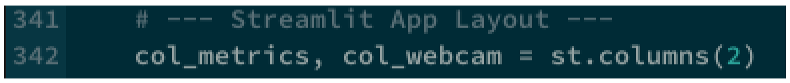
```python

            if drowsy_value is not None:
                with drowsy_lock:
                    drowsiness_history.append(drowsy_value)
                    if len(drowsiness_history) > MAX_HISTORY_LENGTH:
                        del drowsiness_history[0]

        except Exception as e:
            print(f"Error during video transform: {e}")
        return img
```

- **Initialization (__init__)**: Retrieves the pre-loaded models from st.session_state when a VideoTransformer object is created by webrtc_streamer. Checks if models were successfully retrieved.
- **Transformation (transform)**: This method is called by streamlit-webrtc for each incoming video frame.
    - **Get Frame & Check Models**: Converts the frame object to an OpenCV-compatible NumPy array (img). Checks if models are initialized; returns early with an error message on the frame if not.

    - **Face Detection**: Runs the face_model (YOLO) on the img.
    - **Process Faces**: If faces are detected, it iterates through each face:
        - Extracts bounding box coordinates (x1, y1, x2, y2).
        - Draws a rectangle on the img around the face.
        - Extracts the face region (face_img).
        - Skips invalid face regions.
    - **Preprocess & Predict**:
        - Calls preprocess_face to prepare the face_img for the drowsiness model.
        - Runs the yolo_model (drowsiness predictor) on the preprocessed tensor (face_tensor) using torch.no_grad() for efficiency during inference.
        - Extracts the resulting drowsiness value.
        - Updates max_drowsy_value_for_frame if the current face's score is higher. This handles multiple faces by focusing on the drowsiest one.
    - **Annotate**: Adds text (label_text) near the bounding box showing the drowsiness score, colored red if above the threshold, green otherwise.
    - **Update History**: If a valid drowsiness score (drowsy_value, which is the max score found) was obtained for the frame, it acquires the drowsy_lock, appends the score to the drowsiness_history list in st.session_state, and trims the list if it exceeds MAX_HISTORY_LENGTH. The lock is crucial here for thread safety.
    - **Return Frame**: Returns the annotated img array, which streamlit-webrtc then displays in the browser.
- **Error Handling**: A try...except block catches general errors during processing, prints them to the console.

9.3. UI Layout (Columns)

The main application area is divided into two columns for layout.

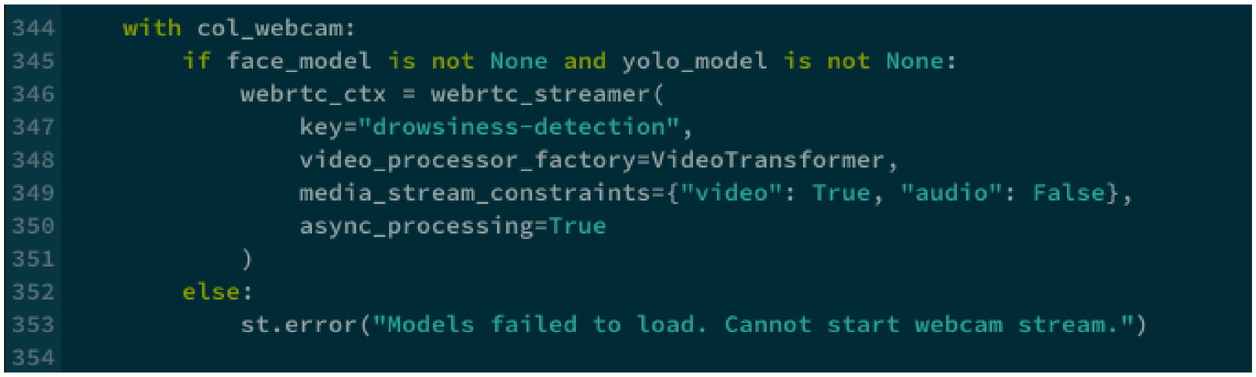


```
# --- Streamlit App Layout ---
col_metrics, col_webcam = st.columns(2)
```

- st.columns(2) creates two columns of equal width. col_metrics will contain the charts and alerts, while col_webcam will contain the video feed.

9.4. Webcam Display Section

This column displays the live video feed processed by the VideoTransformer.

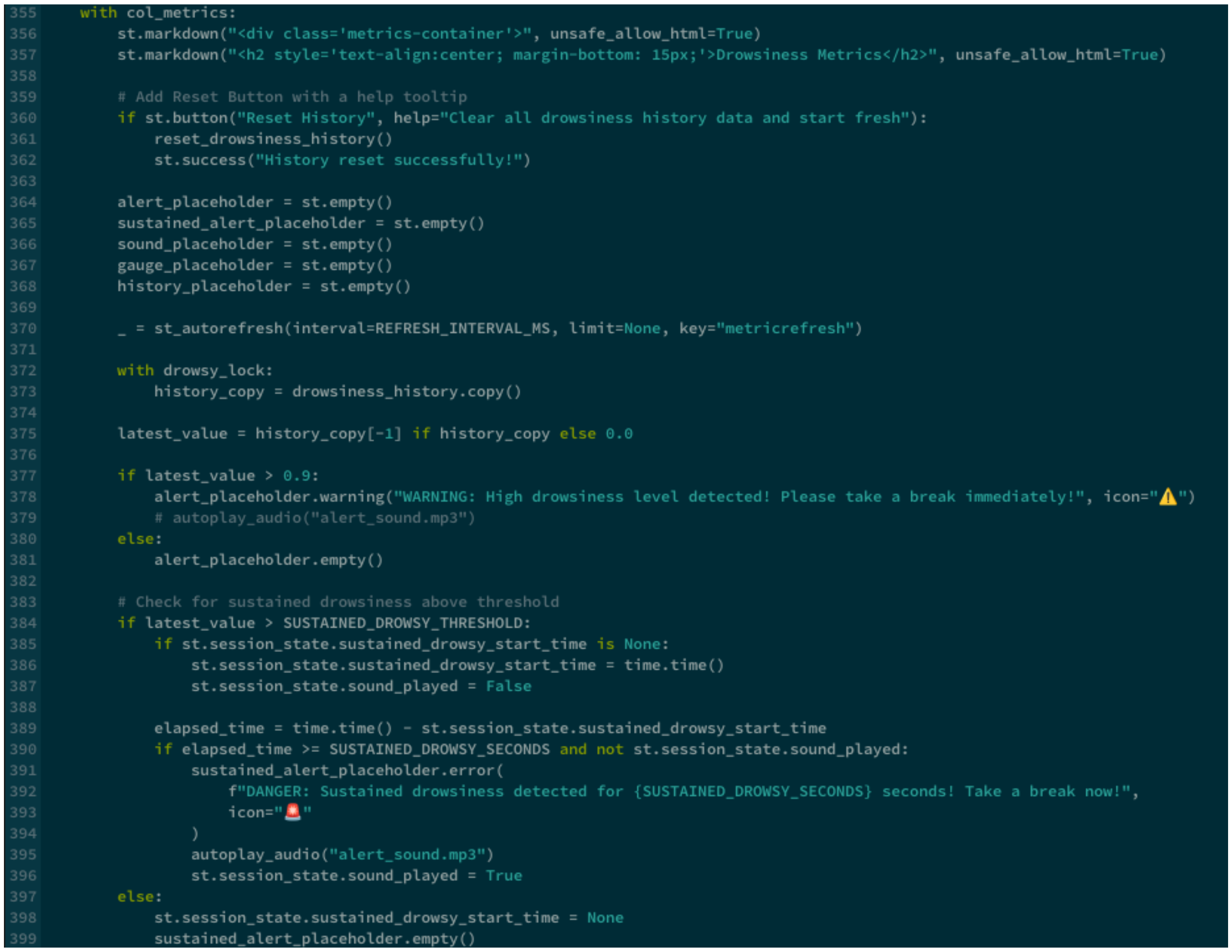


```
    with col_webcam:
        if face_model is not None and yolo_model is not None:
            webrtc_ctx = webrtc_streamer(
                key="drowsiness-detection",
                video_processor_factory=VideoTransformer,
                media_stream_constraints={"video": True, "audio": False},
                async_processing=True
            )
        else:
            st.error("Models failed to load. Cannot start webcam stream.")

```

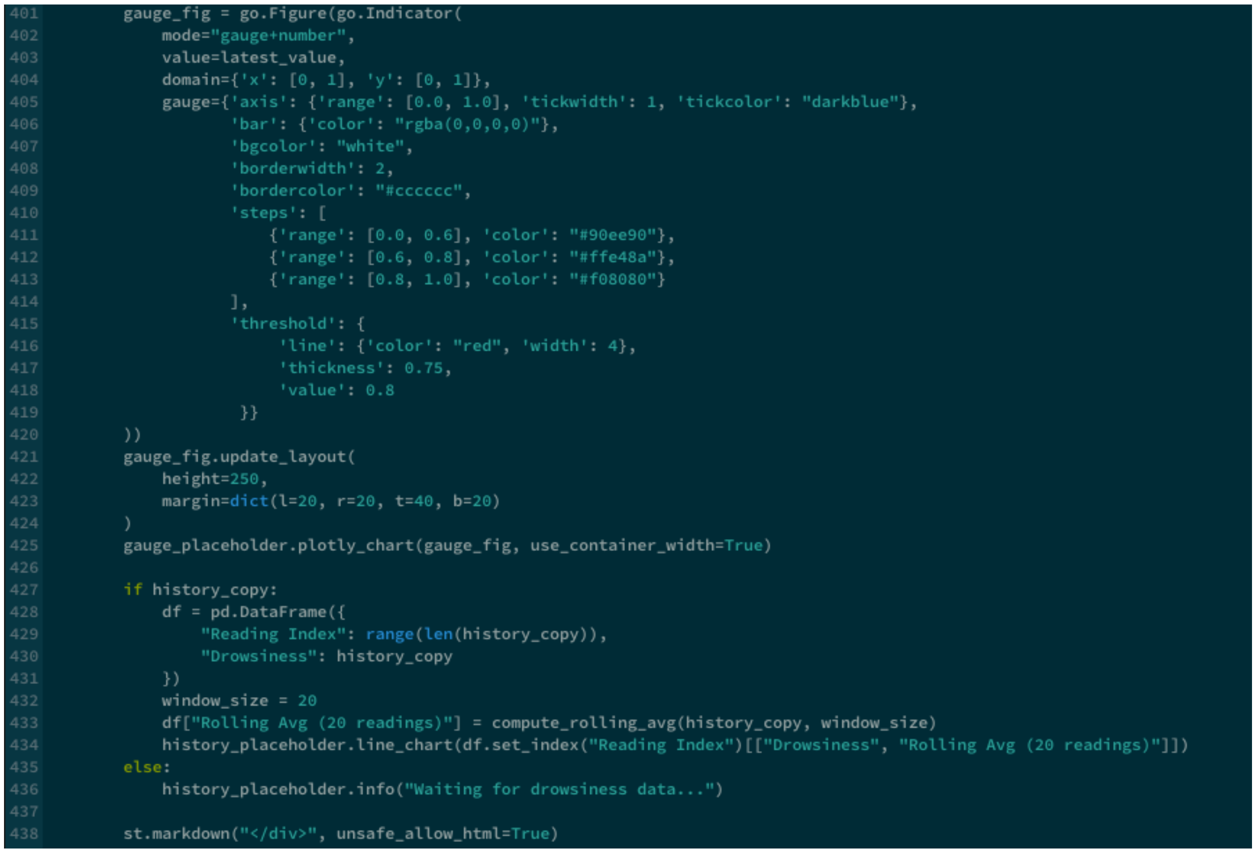
- Uses a with block to place content inside the col_webcam.
- Adds some styled HTML (webcam-card, webcam-header) for visual structure.
- Conditionally calls webrtc_streamer only if the models are loaded.
    - key: A unique identifier for this component instance.
    - video_processor_factory=VideoTransformer: This tells webrtc_streamer to create an instance of our VideoTransformer class and pass video frames to its transform method.
    - media_stream_constraints: Specifies that only video access is required from the user's device.
    - async_processing=True: Runs the transform method in a background thread, preventing the UI from freezing during processing.

9.5. Metrics Display Section

This column displays the real-time metrics, alerts, and controls.



```python
    with col_metrics:
        st.markdown("<div class='metrics-container'>", unsafe_allow_html=True)
        st.markdown("<h2 style='text-align:center; margin-bottom: 15px;'>Drowsiness Metrics</h2>", unsafe_allow_html=True)

        # Add Reset Button with a help tooltip
        if st.button("Reset History", help="Clear all drowsiness history data and start fresh"):
            reset_drowsiness_history()
            st.success("History reset successfully!")

        alert_placeholder = st.empty()
        sustained_alert_placeholder = st.empty()
        sound_placeholder = st.empty()
        gauge_placeholder = st.empty()
        history_placeholder = st.empty()

        _ = st_autorefresh(interval=REFRESH_INTERVAL_MS, limit=None, key="metricrefresh")

        with drowsy_lock:
            history_copy = drowsiness_history.copy()

        latest_value = history_copy[-1] if history_copy else 0.0

        if latest_value > 0.9:
            alert_placeholder.warning("WARNING: High drowsiness level detected! Please take a break immediately!", icon="⚠️")
            # autoplay_audio("alert_sound.mp3")
        else:
            alert_placeholder.empty()

        # Check for sustained drowsiness above threshold
        if latest_value > SUSTAINED_DROWSY_THRESHOLD:
            if st.session_state.sustained_drowsy_start_time is None:
                st.session_state.sustained_drowsy_start_time = time.time()
                st.session_state.sound_played = False

            elapsed_time = time.time() - st.session_state.sustained_drowsy_start_time
            if elapsed_time >= SUSTAINED_DROWSY_SECONDS and not st.session_state.sound_played:
                sustained_alert_placeholder.error(
                    f"DANGER: Sustained drowsiness detected for {SUSTAINED_DROWSY_SECONDS} seconds! Take a break now!",
                    icon="🚨"
                )
                autoplay_audio("alert_sound.mp3")
                st.session_state.sound_played = True
        else:
            st.session_state.sustained_drowsy_start_time = None
            sustained_alert_placeholder.empty()
```



```python
        gauge_fig = go.Figure(go.Indicator(
            mode="gauge+number",
            value=latest_value,
            domain={'x': [0, 1], 'y': [0, 1]},
            gauge={'axis': {'range': [0.0, 1.0], 'tickwidth': 1, 'tickcolor': "darkblue"},
                   'bar': {'color': "rgba(0,0,0,0)"},
                   'bgcolor': "white",
                   'borderwidth': 2,
                   'bordercolor': "#cccccc",
                   'steps': [
                       {'range': [0.0, 0.6], 'color': "#90ee90"},
                       {'range': [0.6, 0.8], 'color': "#ffe48a"},
                       {'range': [0.8, 1.0], 'color': "#f08080"}
                   ],
                   'threshold': {
                        'line': {'color': "red", 'width': 4},
                        'thickness': 0.75,
                        'value': 0.8
                    }}
        ))
        gauge_fig.update_layout(
            height=250,
            margin=dict(l=20, r=20, t=40, b=20)
        )
        gauge_placeholder.plotly_chart(gauge_fig, use_container_width=True)

        if history_copy:
            df = pd.DataFrame({
                "Reading Index": range(len(history_copy)),
                "Drowsiness": history_copy
            })
            window_size = 20
            df["Rolling Avg (20 readings)"] = compute_rolling_avg(history_copy, window_size)
            history_placeholder.line_chart(df.set_index("Reading Index")[["Drowsiness", "Rolling Avg (20 readings)"]])
        else:
            history_placeholder.info("Waiting for drowsiness data...")

        st.markdown("</div>", unsafe_allow_html=True)
```

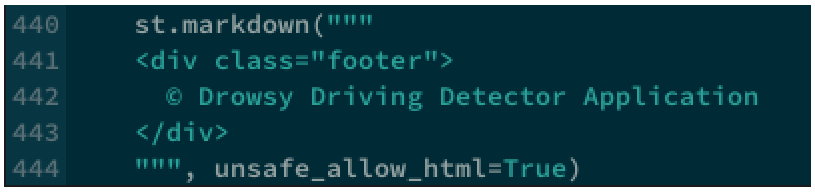
- **Container & Title**: Sets up the container div and title for the metrics section.
- **Reset Button**: Creates a button that, when clicked, calls reset_drowsiness_history, shows a success message, and triggers st.rerun() to update the UI immediately.
- **Placeholders (st.empty)**: Creates empty containers (alert_placeholder, gauge_placeholder, etc.). These act as slots where content (like alerts or charts) can be dynamically inserted or removed on each refresh without affecting the layout of other elements.
- **Auto-Refresh**: st_autorefresh is called to automatically rerun the script every REFRESH_INTERVAL_MS milliseconds. This drives the updates to the metrics display.
- **Data Retrieval**: Gets a thread-safe copy() of the drowsiness_history from session state using the drowsy_lock. Calculates the latest_value.
- **Alert Logic**:
    - Checks latest_value against an instantaneous high_drowsiness_threshold (e.g., 0.9) and displays a warning in alert_placeholder if exceeded.
    - Checks latest_value against SUSTAINED_DROWSY_THRESHOLD. If exceeded, it manages the sustained_drowsy_start_time timer and sound_played flag in session state. If the timer runs beyond

SUSTAINED_DROWSY_SECONDS, it displays an error in sustained_alert_placeholder and calls autoplay_audio. If the score drops, it resets the timer and clears the alert. An optional intermediate warning shows the timer progress.

- **Gauge Chart**: Creates a Plotly Indicator gauge chart showing the latest_value. The gauge has colored steps and a threshold line corresponding to SUSTAINED_DROWSY_THRESHOLD. The chart is displayed in gauge_placeholder.
- **History Chart**: If history_copy is not empty, it creates a Pandas DataFrame, calculates the rolling average using compute_rolling_avg, and generates a Plotly line chart with two traces (raw data and rolling average). The chart is displayed in history_placeholder. If no data exists, an informational message is shown instead.

9.6. Footer

Adds a simple footer at the bottom of the page.



```
    st.markdown("""
    <div class="footer">
      © Drowsy Driving Detector Application
    </div>
    """, unsafe_allow_html=True)
```

10. Conclusion

This Streamlit application provides a comprehensive tool for real-time drowsiness detection. It leverages machine learning models for face detection and drowsiness prediction, integrates seamlessly with a webcam feed using streamlit-webrtc, offers clear visual feedback through annotations and charts, and includes an essential alerting system for sustained drowsiness. The use of session state, caching, and thread locking ensures reasonably efficient and robust operation within the Streamlit framework.