VisionS - Real-time Data-driven Adaptive Driver Monitoring and Awareness System for Safer Roads

Supervisors
Prof. Dr. Massimo VIOLANTE
Dr. Luigi PUGLIESE
Prof. Dr. Jacopo SINI

Candidate
Vasanth DEVAKUMAR

Acknowledgments

The completion of this thesis would not have been possible without the guidance, encouragement, and support of many individuals, to whom I am deeply grateful.

I wish to express my deepest appreciation to my family for their unconditional support, patience, and belief in me. Their encouragement has been a constant source of strength and motivation throughout this journey.

I am deeply thankful to my co-supervisors, **Dr. Luigi Pugliese** and **Dr. Jacopo Sini**, for their invaluable guidance, critical insights, and unwavering support throughout this research. Additionally, I extend my sincere appreciation to my overall supervisor, **Dr. Massimo Violante**, for his oversight and support over the duration of this research. Their mentorship has played a crucial role in refining my research methodology, strengthening my technical approach, and ensuring the successful execution of this study.

A special note of gratitude to **Politecnico di Torino** and **Sleep Advice Technologies (SAT)** for providing the institutional and technical resources necessary for this research. The collaboration with **SAT** has been invaluable in ensuring the practical implementation of this study. Their team has been incredibly warm, welcoming, and supportive, making my experience both engaging, rewarding and memorable.

Lastly, I am equally thankful to my colleagues and friends, whose stimulating discussions, shared experiences, and unwavering support have enriched this journey, making it both intellectually rewarding and personally fulfilling. A special mention goes to **Arindumb, Giova, Edoa, Hali, Loki, Manda, Dee, Sheesh, Kyle, Darth, Aruj and Ishita**, whose encouragement, companionship, and steadfast support have been invaluable throughout my master's journey. Their ability to inspire, challenge, and stand by me through both triumphs and challenges has made this experience all the more meaningful and unforgettable.

Vasanth Devakumar(Arcee)

Table of Contents

Acknowledgments IV

List of Tables XI

List of Figures XII

Acronyms XIV

1 Introduction 3

1.1 The Global Road Safety Crisis 3

1.2 Distracted Driving: A Persistent Problem 4

1.3 Technological Innovations in Driver Monitoring 4

1.4 Behavioral Insights and Ethical Considerations 4

1.5 Economic Implications of Distracted Driving 5

1.6 VisionS: A Data-Driven Approach to Road Safety 5

2 State of the Art 6

2.1 Google's MediaPipe 6

2.1.1 Features and Implementation 6

2.2 OpenCV 8

2.2.1 Features and Implementation 8

2.3 Kivy 10

2.3.1 Key Features of Kivy 10

2.3.2 Integration in VisionS 10

2.3.3 Example of Real-Time Video Processing in Kivy 11

2.4 Synergetic Use of MediaPipe, OpenCV, and Kivy 11

2.4.1 Framework Contributions 11

2.4.2 Core Functionalities in VisionS 11

2.4.3 Overview 12

3 Algorithm 13
3.1 Data Acquisition 13
3.1.1 Frame Capture and Optimization 13
3.2 Pre-processing 14
3.2.1 Resizing and Normalization 14
3.2.2 Noise Reduction 14
3.3 Feature Extraction 15
3.3.1 Integration with MediaPipe 16
3.3.2 Selecting Key Landmarks 16
3.4 Pose Estimation 16
3.4.1 Camera Calibration 17
3.4.2 Solving for Rotation and Translation 17
3.4.3 Roll, Pitch, and Yaw Using Key Facial Landmarks 17
3.5 Logging 18
3.5.1 Key Logged Data 18
3.5.2 Purpose of Logging 19
3.5.3 Implementation Details 20

4 Implementation 21
4.1 Windows Application 21
4.1.1 Overview 21
4.1.2 Application Architecture 22
4.1.3 User Interface (Kivy Implementation) 22
4.1.4 Real-Time Video Processing 23
4.1.5 Final Application UI 23
4.1.6 Summary 24
4.2 Android Implementation 24
4.2.1 Overview 24
4.2.2 Challenges in Python-Based Deployment 25
4.2.3 Transition to a Java-Based Implementation 27
4.2.4 User Interface (UI) Development 30
4.2.5 Final Remarks 35

5 Testing and Results 37
5.1 Algorithm Testing 37
5.1.1 Refinements and Optimizations 39
5.2 Application Testing 40
5.2.1 Windows Application Testing 40
5.2.2 Android Application 42

6 Enhanced Driver Drowsiness Detection Through Data Fusion 46
6.1 PredictS: Wearable Sensor-Based Drowsiness Detection Algorithm . 46
6.2 Enhancing Drowsiness Detection Through Behavioral Analysis . . . 47
6.3 Implementation and Observations 48
6.3.1 Implementation 48
6.3.2 Observations 49

7 Conclusion 52
7.1 Key Achievements 52
7.2 Challenges and Limitations 53
7.3 Future enhancements and improvements 54
7.3.1 iOS Application 54
7.3.2 Extended Real-World Testing and Validation 55
7.3.3 Improved Data Fusion via PredictS and Multi-Modal Driver State Assessment 55
7.4 Final thoughts 56

Bibliography 57

List of Tables

3.1 Sample Logged Data Format. 20

6.1 Drowsiness Detection Decision Framework 47

List of Figures

2.1 Google Mediapipe Face Landmark Detection Guide[13] 7

3.1 Google Media-pipe Face Mesh Landmark numbers[13] 15

4.1 Python based windows application 24
4.2 Java based Android application[13] Video toggled on and off 35

5.1 Distraction flag plot across time 40

6.1 ROC, Sensitivity and Specificity plot 50

Acronyms

DMS

Driver Monitoring System

AI

Artificial Intelligence

WHO

World Health Organization

NHTSA

National Highway Traffic Safety Administration

CDC

Centers for Disease Control and Prevention

GPU

Graphics Processing Unit

API

Application Programming Interface

OpenCV

Open Source Computer Vision Library

RGB

Red, Green, Blue

3D

3-Dimensional

2D

2-Dimensional

PnP

Perspective-n-Point

UI

User Interface

GUI

Graphical User Interface

macOS

Mac Operating System

CPU

Central Processing Unit

OpenGL

Open Graphics Library

SDK

Software Development Kit

NUI

Natural User Interface

KvLang

Kivy Language

XML

Extensible Markup Language

APK

Android Application Package

ARM

Advanced Reduced instruction set computing Machine

NNAPI

Neural Network API

P4A

Python For Android

UIX

User Interface Extensions

CSV

Comma Separated Values

SAE

Society of Automobile Engineers

IDE

Integrated Development Environment

FPS

Frames per Second

RAM

Random Access Memory

HR

Heart Rate

HRV

Heart Rate Variability

SpO2

Saturation of Peripheral Oxygen

ANS

Autonomic Nervous System

AVL

Anstalt für Verbrennungskraftmaschinen

MWT

Maintenance Wakefulness Test

ROC

Receiver-Operating Curve

TPR

True Positive Rate

FPR

False Positive Rate

Chapter 1

Introduction

1.1 The Global Road Safety Crisis

Road traffic accidents are a major cause of death and illness globally. The World Health Organization (WHO) estimates that nearly 1.35 million people die due to traffic accidents each year, with countless others suffering serious injuries[1]. This public health crisis not only causes deep personal grief but also causes a heavy economic burden, especially in low- and middle-income countries where the impact is most felt. Economic studies suggest that the economic costs of road traffic accidents could account for between 1% and 3% of a country's gross domestic product[1].

One major cause of road traffic accidents is driver distraction, which accounts for a sizable percentage of all traffic accidents. In 2022, distracted driving caused 3,308 fatalities, according to the U.S. National Highway Traffic Safety Administration (NHTSA)[2]. Distractions are often classified into visual, manual, and cognitive types, including behaviors such as texting, interacting with in-vehicle technologies, or getting off-track and falling prey to daydreaming. The widespread use of smartphones and car technologies has intensified the issue[3].

Aside from their effects on human life, traffic accidents also put significant economic burdens on societies. Some of these economic burdens are direct and indirect costs, including healthcare costs, emergency service costs, vehicle repair costs, and loss of productivity. For instance, the U.S. Centers for Disease Control and Prevention (CDC) estimates that the annual economic cost of road traffic crashes is over $75 billion[4]. These economic consequences underscore the urgent need for the adoption of preventive measures, such as driver monitoring systems and increased awareness programs, to stem financial losses.

1.2 Distracted Driving: A Persistent Problem

In spite of stringent traffic laws and awareness campaigns, distracted driving is a prevalent practice. Conventional methods, including charging penalties or encouraging defensive driving, have not been very effective in controlling the behavior[5]. Human factors related to distraction are complicated, with psychological, behavioral, and contextual factors.

The advancement of technology has presented new possibilities to address this problem. While autonomous vehicles are likely to reduce human error in the long run, there is a pressing need to enhance safety for drivers driving conventional vehicles. This calls for the development of sophisticated systems that have the ability to monitor driver behavior in real time and provide immediate corrective actions[6].

1.3 Technological Innovations in Driver Monitoring

Historically, technologies such as cruise control and lane-keeping assist provided the foundation upon which contemporary driver monitoring systems were developed. Advanced systems today employ computer vision and artificial intelligence to analyze driver behavior in real-time and hence provide timely warning and intervention[7]. These technologies signal a paradigm shift from reactive to proactive road safety measures.

Driver Monitoring Systems (DMS) are meant to identify the presence of distraction, drowsiness, or other dangerous behaviors from the analysis of parameters like gaze direction, head pose, and facial expressions. The systems utilize a fusion of computer vision techniques and machine learning algorithms to evaluate driver states and detect anomalies[8]. For instance, gaze-tracking technology has the capability to identify if a driver's eyes drift away from the road for extended durations, thereby initiating alerts to re-concentrate.

1.4 Behavioral Insights and Ethical Considerations

Behavioral studies show that cognitive load and stress play a significant role in driving performance. A distracted driver may take up to three times longer to react to road dangers than an attentive driver[9]. It is necessary to discern such behavioral trends in creating systems like VisionS that have the ability to learn and adapt to individual drivers' needs, mitigating risks of cognitive distractions.

As there is wider application of driver monitoring systems, there have been ethical and legal issues and of specific interest are privacy of information and surveillance matters. Safeguarding driver information and limiting its use to the safety function only is vital in order not to lose the public's trust[10]. Policies by regulatory authorities need to develop to handle these issues in a way that balances innovation with the protection of people's rights.

1.5 Economic Implications of Distracted Driving

The economic implications of distracted driving extend far beyond the immediate costs of vehicle repair and medical treatment. With increasing frequency of accidents, there is also a higher insurance premium bill, which creates a financial burden for individual drivers as well as the economy as a whole. Industries that rely on transport services are disrupted through these accidents, impacting supply chains and delivery times. Governments also incur high costs in providing emergency response services and maintaining infrastructure, taking away funds from other essential areas[11]. These economic costs underscore the need for systems like VisionS to actively reduce distractions and the costs they present.

1.6 VisionS: A Data-Driven Approach to Road Safety

This thesis introduces VisionS, a real-time data-driven adaptive driver monitoring and awareness system. VisionS takes advantage of the front-facing camera of a smartphone to monitor driver behavior, using a calibration phase to adapt to the frames of reference of individual drivers. The system continuously tracks parameters such as head orientation and gaze direction, issuing alerts when signs of distraction are detected. Unlike existing solutions, VisionS combines affordability and accessibility, making it feasible for widespread adoption in conventional vehicles.

VisionS also incorporates features such as state calibration and adaptive alert thresholds, ensuring that the system remains effective in diverse driving conditions.

Chapter 2

State of the Art

2.1 Google's MediaPipe

Google's MediaPipe[12] is a cross-platform framework for building multimodal machine learning pipelines. It is designed to support real-time processing and is widely used in applications such as facial recognition, pose estimation, and object tracking. In your algorithm, MediaPipe's FaceMesh solution is a core component, utilized for facial landmark detection and tracking. This functionality enables precise identification of key facial features, such as the eyes, nose, and mouth, which are critical for estimating head pose and detecting driver distraction.

2.1.1 Features and Implementation

MediaPipe FaceMesh[13] provides:

1. **High-Fidelity Landmark Detection**: It detects 468 facial landmarks, offering detailed spatial information about facial features. This high resolution of landmark mapping enables applications that require fine-grained facial analysis, such as detecting micro-expressions or subtle head movements.
    - The landmarks are divided into regions, including the eyes, lips, and jawline. For example, landmarks around the eyes can be used to calculate blink rates and track gaze direction, which is vital for monitoring whether a driver's attention is on the road. Similarly, landmarks around the jawline are essential for jaw tracking and pose calibration, ensuring that head movements are consistently interpreted.
    - These facial landmarks also play a critical role in detecting signs of fatigue or distraction. For instance, prolonged eye closure or erratic head movements can indicate drowsiness, prompting timely interventions.

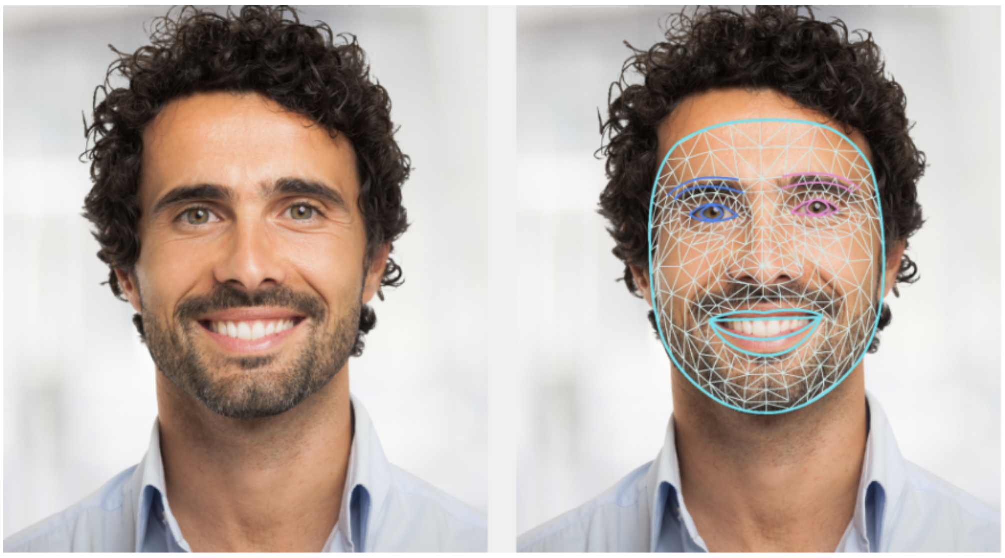

Figure 2.1: Google Mediapipe Face Landmark Detection Guide[13]

2. **Real-Time Performance**: Leveraging GPU acceleration, MediaPipe ensures low latency, which is crucial for real-time applications. This is achieved through its graph-based computational architecture, where multiple processing steps are connected in a directed acyclic graph.

   - The system processes video streams frame-by-frame while maintaining synchronization between detection, tracking, and rendering. This capability is especially critical during high-speed driving scenarios, where immediate feedback is necessary to ensure safety. For example, in environments where a driver's head movement must be tracked and analyzed within milliseconds, the use of GPU acceleration minimizes delays and enhances responsiveness. Real-time processing enables timely alerts for distractions, ensuring that the system remains effective even in dynamic, fast-paced conditions.

3. **Cross-Platform Support**: MediaPipe is compatible with multiple platforms, including Android, iOS, and Python environments. This flexibility makes it an ideal choice for applications that need to scale across different hardware and operating systems.

Python Implementation

In the Python implementation, MediaPipe FaceMesh is initialized with parameters such as:

- `max_num_faces`: Limits the number of faces detected to one, optimizing performance for single-driver monitoring systems.
- `min_detection_confidence` and `min_tracking_confidence`: Set thresholds for reliable face detection and tracking, balancing accuracy and computational overhead.

The processed landmarks are fed into downstream tasks like calculating head orientation and gaze estimation. Additional utilities in MediaPipe simplify tasks such as video frame pre-processing and visualization.

Java Implementation

The Java implementation integrates MediaPipe's FaceMesh API with OpenCV for seamless video frame processing. By leveraging Android's CameraX API, frames are captured and converted into MediaPipe-compatible formats. MediaPipe's FaceMesh outputs a list of landmark coordinates, which are used in conjunction with OpenCV for pose estimation and distraction detection.

2.2 OpenCV

OpenCV (Open Source Computer Vision Library)[14] is a robust computer vision library with a comprehensive set of tools for image processing, machine learning, and computer vision applications. It plays a pivotal role in your algorithm by enabling head pose estimation and handling image transformations.

2.2.1 Features and Implementation

Key functionalities of OpenCV in your implementation include:

1. **Image Processing**:
    - Converts video frames to RGB format and applies necessary transformations for MediaPipe compatibility.
    - Implements filtering techniques, such as Gaussian blur, to remove noise and improve the accuracy of landmark detection.
    - Addresses challenges like low-light conditions by adjusting image brightness and contrast dynamically, ensuring consistent landmark detection performance even in suboptimal lighting. For instance, histogram equalization can be applied to enhance image clarity.

   - Tackles motion blur using deblurring algorithms that reconstruct sharper images from blurry inputs, maintaining accuracy in high-motion scenarios like sudden head movements or rapid camera shifts.

2. **Head Pose Estimation**:
   - Utilizes the `solvePnP` function to calculate rotational and translational vectors for 3D pose estimation. This involves mapping 2D facial landmarks onto predefined 3D face models.
   - The output includes rotation and translation vectors, which are converted to Euler angles (pitch, yaw, roll) using `Rodrigues` for easier interpretation. These angles are crucial for identifying driver distraction, as they quantify the orientation of the driver's head relative to a calibrated baseline. For instance, a yaw angle exceeding a predefined threshold may indicate that the driver is looking away from the road, while significant deviations in pitch could signal nodding or drowsiness. By continuously tracking these deviations, the system flags distraction events and triggers appropriate alerts.

3. **Efficient Matrix Operations**:
   - Supports matrix multiplications, vector transformations, and coordinate system conversions.
   - Computes transformations required to align the camera's frame of reference with the driver's head orientation. For instance, rotation matrices are used to compute angles relative to the calibrated baseline.

Python Implementation

In Python, OpenCV is extensively used for:

- Calculating the standard deviation of head pose angles during the calibration phase.
- Applying transformations to align the driver's face with the camera view for consistent results across varying conditions.
- Logging distraction events by identifying deviations from calibrated thresholds.

Java Implementation

In the Java implementation, OpenCV integrates with Android's CameraX API to process frames. Functions like `Calib3d.solvePnP` compute head pose using 2D and 3D facial landmark coordinates. The results are visualized on the Android interface to provide real-time feedback to the user.

2.3 Kivy

Kivy is an open-source Python framework designed for developing cross-platform applications with natural user interfaces (NUIs). It provides a highly flexible architecture for building interactive applications that seamlessly integrate with real-time data processing systems. Given its robust rendering capabilities and compatibility across multiple operating systems, Kivy was chosen as the UI framework for VisionS.

2.3.1 Key Features of Kivy

Kivy offers a wide range of features that enhance the development of intuitive and responsive graphical user interfaces:

- **Cross-Platform Compatibility**: Supports deployment on Windows, macOS, Linux, Android, and iOS, ensuring flexibility in application design.
- **Declarative Language (KvLang)**: Uses a separate '.kv' file for UI structure, enabling clean separation of logic and presentation.
- **Built-in Widgets**: Provides an extensive set of UI components, such as buttons, sliders, labels, and image containers, facilitating rapid development.
- **GPU-Accelerated Rendering**: Leverages OpenGL for high-performance animations and smooth UI transitions.

2.3.2 Integration in VisionS

In VisionS, Kivy is responsible for rendering the graphical user interface of the Windows application. Its ability to handle real-time updates efficiently makes it an ideal choice for displaying critical driver monitoring data.

The VisionS UI is structured into three main components:

- **KvLang (`main.kv`)**: Defines the layout and UI structure, ensuring modularity and ease of updates.
- **Python Backend (`main.py`)**: Manages user interactions, real-time updates, and event handling.
- **OpenCV Integration**: Facilitates real-time video processing by updating UI textures dynamically.

Kivy seamlessly integrates with OpenCV and MediaPipe, allowing VisionS to process video streams in real-time and overlay essential information such as yaw, pitch, and roll values. The UI is designed to be minimalistic yet informative, ensuring that the driver receives immediate feedback regarding their attentiveness.

2.3.3 Example of Real-Time Video Processing in Kivy

A critical aspect of the VisionS implementation is updating the UI dynamically as the video frames are processed. This is achieved through OpenCV's frame capture capabilities and Kivy's texture update mechanism.

```python
def update_frame(self, *args):
    ret, frame = cap.read()
    if ret:
        processed_frame = self.visionS.process_frame(frame)
        texture = self.convert_to_texture(processed_frame)
        self.video_feed.texture = texture
```

This ensures smooth, real-time updates, enhancing the responsiveness of the monitoring system.

2.4 Synergetic Use of MediaPipe, OpenCV, and Kivy

The combination of MediaPipe, OpenCV, and Kivy in the VisionS algorithm exemplifies a powerful synergy. Each framework plays a crucial role in enabling real-time driver monitoring, ensuring both accuracy and efficiency. MediaPipe provides precise facial landmark detection, OpenCV handles computational geometry and real-time video processing, while Kivy acts as the bridge for visualization and user interaction.

2.4.1 Framework Contributions

- **MediaPipe**: Extracts high-precision 2D facial landmarks, forming the basis for head pose estimation.
- **OpenCV**: Performs real-time video capture, applies preprocessing techniques, and computes Euler angles for yaw, pitch, and roll.
- **Kivy**: Provides a cross-platform UI framework that integrates OpenCV outputs, displaying driver status with interactive overlays and visual alerts.

2.4.2 Core Functionalities in VisionS

- **Calibration Phase**: MediaPipe's facial landmarks define a baseline for the driver's neutral head position. OpenCV computes statistical metrics such as mean and standard deviation, while Kivy ensures real-time visualization of the calibration process.

- **Distraction Detection**: By analyzing deviations from the calibrated yaw, pitch, and roll values, the system flags potential driver distractions. OpenCV processes the deviations, MediaPipe ensures accurate tracking, and Kivy updates the UI with real-time alerts.
- **Real-Time Visualization**: Kivy provides a dynamic user interface that continuously updates with distraction warnings, head pose angles, and status overlays. This ensures that users receive immediate feedback on their driving behavior.
- **Data Logging and Analysis**: OpenCV facilitates logging of essential driver data, including timestamps, pose angles, and distraction status. Kivy can display historical trends, providing a comprehensive view of driver behavior over time.

2.4.3 Overview

The integration of MediaPipe, OpenCV and Kivy enhances VisionS's capability to provide real-time, efficient, and user-friendly driver monitoring. MediaPipe extracts facial landmarks with high precision, OpenCV enables seamless video processing, and Kivy ensures the data is effectively visualized. This synergy creates a system that is accurate, computationally efficient, and easy to use, making it a valuable tool for modern driver safety applications. Future improvements may include optimizing computational performance and expanding cross-platform capabilities to further enhance VisionS's effectiveness.

Chapter 3

Algorithm

The VisionS algorithm is a modular system that supports real-time data processing for the monitoring and analysis of driver behavior. This section presents a detailed analysis of the steps in data acquisition, pre-processing, feature extraction, and pose estimation. Each step is based on careful calculations and systematic reasoning from the implementation of the code.

3.1 Data Acquisition

The algorithm begins by acquiring live video frames from a front-facing camera. This process involves interfacing with the camera hardware through OpenCV and optimizing data flow to ensure efficient processing.

3.1.1 Frame Capture and Optimization

OpenCV's `VideoCapture` module is used for capturing video frames. In order to ensure real-time performance with low computational latency, the algorithm selectively processes alternate frames, thus optimizing efficiency without significant loss of accuracy.

```
cap = cv2.VideoCapture(0)
frame_count = 0
while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    frame.flags.writeable = False  # Optimize for MediaPipe
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)  # Convert
   for MediaPipe

    frame_count += 1
    if frame_count % 2 != 0:  # Skip alternate frames
        continue

    results = face_mesh.process(frame_rgb)
cap.release()
```

This method ensures that a single frame is examined for each pair of frames, thus minimizing overhead while maintaining temporal accuracy in detecting rapid driver behavior.

3.2 Pre-processing

Pre-processing is an important step that prepares raw video frames for accurate feature extraction. Pre-processing involves resizing, normalization, noise reduction, and making frames non-writable to improve computational efficiency.

3.2.1 Resizing and Normalization

Frames are resized to standard dimensions and normalized to scale pixel intensities:
Mathematical Representation:

$$I_{resized} = \text{Resize}(I, W, H)$$

$$I_{normalized} = \frac{I_{resized}}{255}$$

where W and H are the target width and height.

3.2.2 Noise Reduction

To enhance clarity, Gaussian blur is applied:

$$I_{smoothed} = G(I_{normalized}, \sigma)$$

where G is the Gaussian kernel with standard deviation σ.

```
frame_blurred = cv2.GaussianBlur(frame_normalized, (5, 5), 0)
```

3.3 Feature Extraction

Facial landmarks are achieved via the use of Media Pipe Face Mesh, providing an accurate representation of 468 individual facial coordinates. The landmarks are the basis for pose estimation as well as gaze analysis.

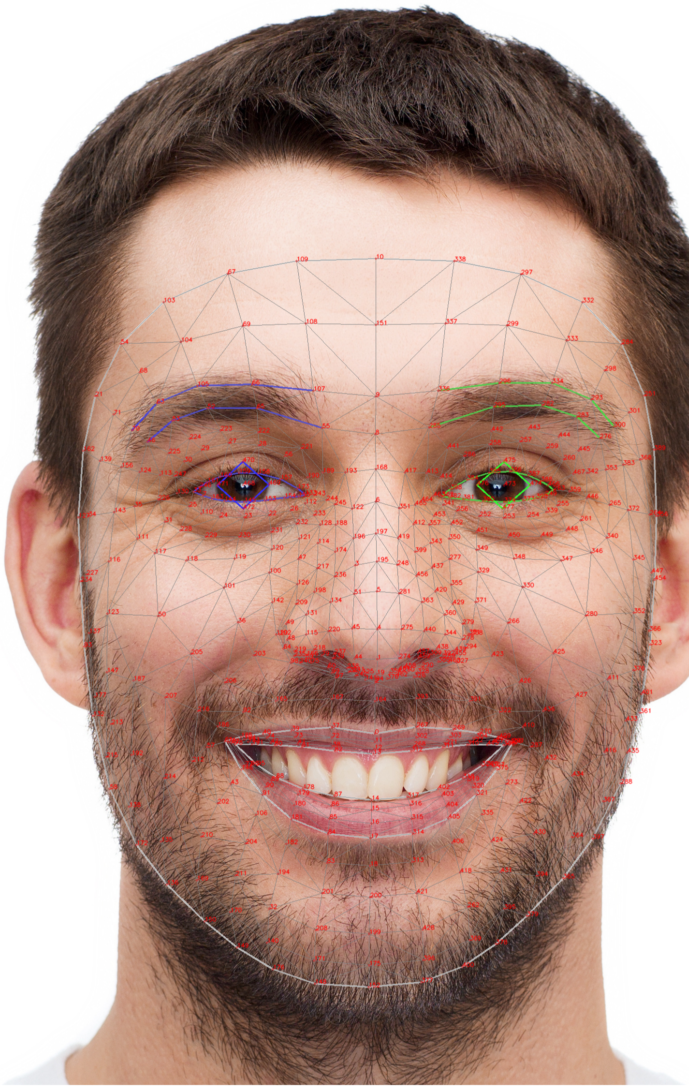

Figure 3.1: Google Media-pipe Face Mesh Landmark numbers[13]

3.3.1 Integration with MediaPipe

The algorithm integrates MediaPipe to detect and extract facial landmarks efficiently:

```
mp_face_mesh = mp.solutions.face_mesh
with mp_face_mesh.FaceMesh(max_num_faces=1) as face_mesh:
    results = face_mesh.process(frame_rgb)
    for face_landmarks in results.multi_face_landmarks:
        landmarks = [(lm.x, lm.y, lm.z) for lm in
   face_landmarks.landmark]
```

Landmarks are converted into a structured format suitable for subsequent calculations.

3.3.2 Selecting Key Landmarks

Specific landmarks are extracted for defining 3D-2D mappings. For example:

- Landmark 33 (nose tip): (x_{33}, y_{33}, z_{33})
- Landmark 263 (right eye outer corner): $(x_{263}, y_{263}, z_{263})$

These points populate the `face_3D` and `face_2D` arrays:

```
face_3d = [
    [landmarks[33][0], landmarks[33][1], landmarks[33][2]],
    [landmarks[263][0], landmarks[263][1], landmarks[263][2]],
    # Other key points
]

face_2d = [
    [landmarks[33][0], landmarks[33][1]],
    [landmarks[263][0], landmarks[263][1]],
    # Other key points
]
```

3.4 Pose Estimation

Pose estimation estimates the head orientation of the driver by solving the Perspective-n-Point (PnP) problem. It maps 3D facial landmarks to 2D projections in the image plane.

3.4.1 Camera Calibration

The camera intrinsic matrix K is defined as:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where f_x and f_y are focal lengths, and c_x, c_y are the optical center coordinates.

3.4.2 Solving for Rotation and Translation

Using OpenCV's `solvePnP`, the rotation vector R and translation vector T are computed:

$$[R, T] = \mathrm{PnP}(L_{3D}, L_{2D}, K)$$

```
success, rot_vec, trans_vec = cv2.solvePnP(face_3d, face_2d,
    cam_matrix, dist_matrix)
rmat, jac = cv2.Rodrigues(rot_vec)
```

3.4.3 Roll, Pitch, and Yaw Using Key Facial Landmarks

The algorithm uses significant facial landmarks, i.e., left eye (landmark 33) and right eye (landmark 263) to compute roll more accurately than from the rotation matrix by itself. It utilizes the relative positions of these landmarks.

Roll Calculation

The roll angle θ_z is computed using the positions of the left and right eye landmarks:

$$\theta_z = \arctan 2(y_{RER} - y_{LEL}, x_{RER} - x_{LEL})$$

where (x_{RER}, y_{RER}) and (x_{LEL}, y_{LEL}) are the coordinates of the right eye outer corner and left eye outer corner, respectively.

```
roll = 180 + (np.arctan2(point_RER[1] - point_LEL[1],
    point_RER[0] - point_LEL[0]) * 180 / np.pi)
if roll > 180:
    roll -= 360
```

Pitch and Yaw Using `RQDecomp3x3`

The `cv2.RQDecomp3x3` function decomposes the rotation matrix to derive pitch θ_x and yaw θ_y:

$$\theta_x = \arctan 2(R_{32}, R_{33}), \quad \theta_y = \arcsin(-R_{31})$$

```
angles , _, _, _, _, _ = cv2.RQDecomp3x3(rmat)
pitch = angles[0] * 1800
yaw = angles[1] * 1800
```

By combining the roll calculation using facial landmarks and pitch/yaw derived from the rotation matrix, the algorithm ensures:

- **Precision:** Leveraging distinct features like eye positions provides higher accuracy.
- **Robustness:** Redundant calculations reduce sensitivity to errors in one method.
- **Real-Time Adaptability:** Efficient computation ensures no latency in head orientation updates.

3.5 Logging

Logging is a critical component of VisionS, enabling measurement of performance, debugging, and post-analysis of driving behavior. Logging is a critical mechanism for validating the system's correctness, fine-tuning the detection algorithm, and ensuring reliability under different conditions. The system logs critical parameters in **CSV format**, enabling structured data capture and large-scale post-processing analysis.

3.5.1 Key Logged Data

The following key data points are systematically recorded during VisionS operation:

- **Frame Count**: Each frame is assigned a sequential number, enabling precise frame-by-frame analysis.
- **Timestamp**: Both **relative frame time** (time elapsed since application start) and **absolute system time** are logged for synchronization with external data sources and event tracking.

- **Roll, Pitch, and Yaw Values**: These head orientation angles are logged to assess driver movement trends and posture over time.
- **Standard Deviation of Pitch and Yaw**: This statistical measure helps assess the variability in head movements, filtering out noise and enhancing detection robustness.
- **Estimated Head Position**: The system determines whether the driver is looking **forward, left, right, up, or down** based on calibrated threshold values.
- **Distraction Flag**: If a driver maintains a non-forward gaze beyond a predefined time threshold, a distraction event is flagged and recorded.
- **Processing Latency**: The time taken to process each frame is logged to monitor computational efficiency and optimize performance.

3.5.2 Purpose of Logging

The structured logging system serves multiple purposes, ensuring the reliability and adaptability of VisionS:

- **Verification of Detection Accuracy**: Logged data is plotted and cross-referenced with known distraction events to assess the precision of detection algorithms.
- **Performance Benchmarking**: The system logs frame processing times, CPU and GPU utilization, and memory usage to measure computational efficiency across different hardware configurations.
- **Adaptive Calibration Adjustments**: The standard deviation values of pitch and yaw allow dynamic sensitivity tuning, improving detection accuracy for diverse driving conditions. Additionally, these values are logged for integration into a separate data fusion algorithm designed to estimate and predict the driver's drowsiness state, which will be discussed later in this report.
- **Error Diagnosis & Debugging**: Analyzing logged data helps identify potential false positives and negatives, allowing for refinements in algorithm logic and threshold tuning.
- **Longitudinal Driver Behavior Analysis**: By collecting data over extended periods, trends in driver attention patterns can be analyzed, aiding in personalized driver monitoring strategies.

3.5.3 Implementation Details

VisionS utilizes a systematic logging system based on Python's built-in **CSV module**, thus enabling efficient and fast recording of data. Each frame processed creates a new log entry, with real-time values displayed in the following format:

Frame	**Time**	**Roll**	**Pitch**	**Yaw**	**Std Pitch**	**Std Yaw**	**HeadPos.**	**Distract**
2	0:00:00.023992	-8.56	11.79	-32.06	0	0	64	0
4	0:00:00.076502	-8.34	12.08	-28.47	0	0	64	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
20000	00:02:40.031	15.6	34.96	47.42	8.07351	7.55612	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 3.1: Sample Logged Data Format.

The logging system is engineered to minimize interruptions to real-time performance, whilst simultaneously enabling comprehensive data accumulation for offline analysis.

Through structured logging, VisionS ensures that driver behavior analysis is both precise and verifiable, allowing for continuous improvements in the system's accuracy and responsiveness.

Chapter 4

Implementation

4.1 Windows Application

4.1.1 Overview

The VisionS Window Application serves as the main interface for real-time monitoring of drivers. It uses a structured approach for handling video inputs, assessing drivers' head motion, and delivering relevant feedback to users. The system uses **Kivy** for the GUI and **OpenCV** for video analysis, with the VisionS algorithm being implemented in a modular class structure.

The application is structured into three key components:

- **VisionS Algorithm Class** (defined in `VisionS_Application_V1.py`) for processing video frames and detecting driver distractions.

- **User Interface** (defined in `main.kv`), built using Kivy, handling user interactions and visual elements.

- **Main Application Logic** (defined in `main.py`), which integrates the UI with the VisionS algorithm and manages real-time updates.

Additionally, two versions of the application were developed:

- **Real-time mode**: Processes live feed from the Windows camera for immediate monitoring.

- **Offline mode**: Processes all video files available in the executable folder to analyze pre-recorded footage.

4.1.2 Application Architecture

The VisionS Window Application follows a modular approach, ensuring scalability and maintainability. Each component is designed to work independently while maintaining seamless interaction.

Key Components

- **VisionS_Application_V1.py**
  - Implements the **VisionS** class, responsible for image processing, head pose estimation, and distraction detection.
  - Ensures modularity so that the same algorithm can be integrated into multiple applications.
- **main.py**
  - Acts as the central controller, initializing the UI, fetching camera frames, and passing them to the VisionS algorithm.
  - Manages real-time updates and event handling.
- **main.kv**
  - Defines the UI layout, including buttons, overlays, and video feed display.
  - Uses Kivy's declarative syntax to separate the logic from the UI elements.

4.1.3 User Interface (Kivy Implementation)

Kivy is chosen for its ability to create cross-platform applications with minimal effort. The interface is designed to be **intuitive and responsive**, ensuring ease of use for drivers and developers alike.

Key Features of the UI

- **Live Video Feed**: Displays real-time video input from the camera.
- **Application Status**: Updates with the current status whether it's in calibration phase or running to detect distractions.
- **Distraction Alerts**: Shows warnings when driver distraction is detected.
- **Calibration Controls**: Allows users to initiate or reset calibration.
- **Status Overlays**: Displays yaw, pitch, roll, and distraction status in real-time.

```
Image:
    id: video_feed
    size_hint: (1, 0.8)
    allow_stretch: True
```

4.1.4 Real-Time Video Processing

Real-time video processing is a core functionality of the VisionS application. The `main.py` script captures frames using OpenCV, processes them through the **VisionS** class, and renders them in the UI.

```
cap = cv2.VideoCapture(0)
def update_frame(self, *args):
    ret, frame = cap.read()
    if ret:
        processed_frame = self.visionS.process_frame(frame)
        texture = self.convert_to_texture(processed_frame)
        self.video_feed.texture = texture
```

4.1.5 Final Application UI

The final version of the VisionS Window Application presents a clean and intuitive interface designed for real-time monitoring. The UI displays:

- The **live video feed** processed through OpenCV.
- **Yaw, pitch, and roll values** updated dynamically.
- **Distraction alerts**, which notify the user when an unsafe behavior is detected.
- **Calibration controls** that allow the driver to reset their reference head position.

Below is an image showcasing the final application UI:

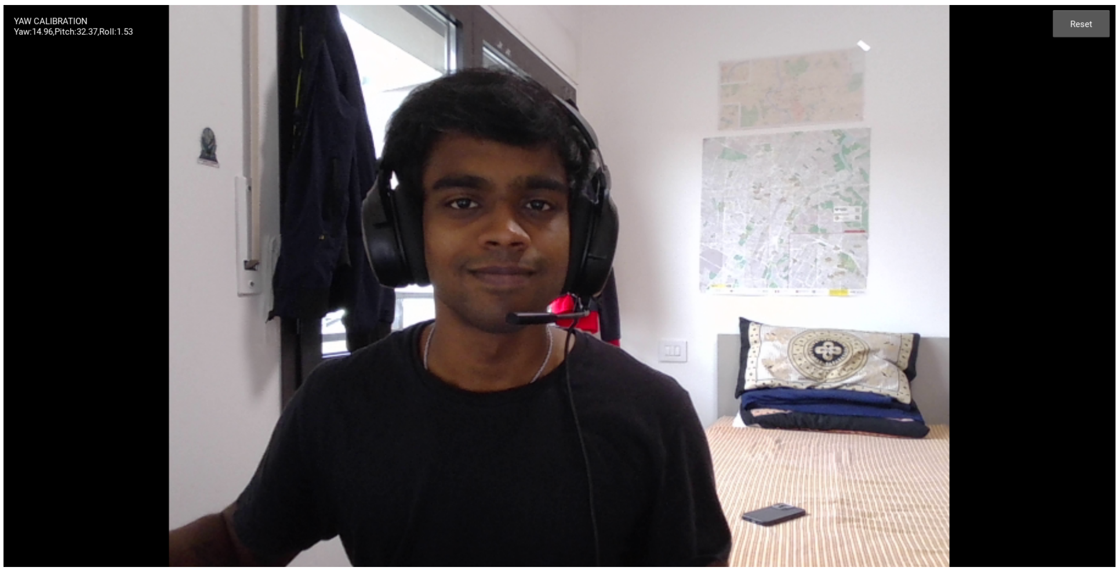

Figure 4.1: Python based windows application

4.1.6 Summary

The **VisionS Window Application** seamlessly integrates Kivy for user interface creation, employs OpenCV for video recording, and incorporates the VisionS algorithm to detect distractions. Its modularity allows **real-time monitoring and offline analysis**, thus ensuring seamless user experience while enabling flexible testing and validation.

The next section, **4.2 Android Application**, will focus on porting the VisionS system to mobile platforms.

4.2 Android Implementation

4.2.1 Overview

The Android version of VisionS was created to optimize the features of real-time driver monitoring on mobile devices by utilizing the smartphone's front-facing camera. Unlike the Windows version, which used Python libraries like OpenCV and Kivy, the Android version required a redesign of its architecture. The final implementation was built using **Java, OpenCV, and MediaPipe** and was developed in **Android Studio** for maximum performance and smooth functionality. The UI was created using **Android's XML layout system**, and the VisionS algorithm was rewritten in Java and run in `MainActivity.java`.

This section outlines the challenges encountered in the process of transition, the several failed attempts to implement the Python-based one on the Android platform, and finally the development of a complete Java-based solution.

4.2.2 Challenges in Python-Based Deployment

Deploying the Python-based VisionS algorithm on Android proved challenging due to multiple technical limitations and compatibility issues. Several methods were explored, but they ultimately proved impractical for real-time execution on mobile devices.

Attempt 1: Deploying through Buildozer on Windows

Buildozer was initially chosen as a packaging tool to convert the Python-based VisionS application into an Android APK. However, the process faced critical limitations:[15]

- **Dependency Conflicts**: The integration of OpenCV and MediaPipe with Buildozer resulted in unresolved dependencies, preventing the app from running.
- **TensorFlow Lite Incompatibility**: MediaPipe relies on TensorFlow Lite for real-time inference, but Buildozer lacked proper support for it.
- **Limited Debugging Support**: Errors related to C++ bindings and shared object libraries made troubleshooting difficult.
- **Unsupported Architectures**: Certain Android devices with ARM64-based architecture faced compatibility issues with libraries compiled through Buildozer.
- **Large APK Size**: The resulting application was significantly larger than expected, increasing installation time and consuming excessive storage.

Attempt 2: Deployment Using Buildozer on Linux

A subsequent attempt was made using Buildozer on an Ubuntu Linux environment, as Linux is often more compatible with Python-based development tools. Despite better support for some dependencies, the following issues persisted:[16]

- **Lack of Official MediaPipe Support**: The absence of a well-maintained MediaPipe package for Python-For-Android (P4A) caused repeated build failures.
- **Application Instability**: APKs built on Linux frequently crashed due to unresolved OpenCV dependencies and memory allocation errors.

- **Performance Limitations**: The resulting APKs exhibited significant performance bottlenecks, making real-time monitoring unfeasible.
- **Graphics Rendering Issues**: When attempting to display the camera feed, frame drops and rendering artifacts appeared due to inefficient processing.
- **Inconsistent Behavior Across Devices**: While the application worked on some Android devices, it failed to launch or crashed instantly on others due to variations in hardware and firmware.

Attempt 3: Python-Java Integration via Chaquopy in Android Studio

To bridge the gap between Python and Java, Chaquopy was tested for embedding Python scripts within an Android application. However, the following challenges emerged:[15]

- **Execution Overhead**: Running Python scripts within Android's Dalvik runtime introduced considerable latency, slowing down real-time processing.
- **Threading Limitations**: Chaquopy did not support parallel execution in the same way as Java's native multithreading, resulting in frequent UI freezes when attempting to process continuous video input.
- **Lack of GPU Acceleration**: Hardware-accelerated MediaPipe operations were not supported within the Chaquopy framework, causing inefficient CPU-based execution.[17]
- **Limited Access to Native APIs**: The integration made it difficult to directly interact with low-level Android APIs needed for efficient camera handling and system-level optimizations.
- **Poor Debugging Support**: Identifying issues within the hybrid Python-Java implementation was cumbersome, as stack traces often lacked detailed information due to cross-language execution.

Key Takeaways from the Failed Python Deployments

After multiple unsuccessful attempts, it became evident that the best approach was to transition VisionS to a fully Java-based implementation. The main reasons included:[18]

- **Performance Constraints**: Python's runtime overhead was too high for real-time processing on mobile devices, making it unsuitable for continuous video-based monitoring.

- **Limited Library Support**: Essential dependencies such as MediaPipe and OpenCV were not optimized for direct deployment on Android via Python, leading to compatibility issues.
- **APK Size and Compatibility Issues**: The Buildozer and Chaquopy approaches resulted in bloated APK sizes, which reduced user adoption and made the application impractical for general distribution.
- **Security and Stability Risks**: The hybrid Python-Java approach introduced additional vulnerabilities and stability concerns, particularly in long-term usage scenarios.
- **Lack of Maintenance and Community Support**: Many of the tools used in the Python-based deployment had minimal community support, making it difficult to find solutions for encountered issues.
- **Inefficiencies in Real-Time Processing**: Delays in frame acquisition, processing lag, and inconsistent execution speeds rendered the Python-based approaches unsuitable for critical safety applications.

These challenges ultimately confirmed the need to **rewrite the entire VisionS algorithm in Java**, leveraging Android-native frameworks for optimal performance and real-time distraction detection.

4.2.3 Transition to a Java-Based Implementation

Overcoming Python's Limitations with Java

The decision to rewrite the VisionS algorithm in Java stemmed from the inherent limitations encountered in Python-based deployment. While Python provided rapid prototyping capabilities, it struggled with performance bottlenecks, lack of GPU acceleration on Android, and inefficient multithreading. Transitioning to Java allowed the system to fully utilize Android's native libraries, ensuring a **highly responsive, low-latency, and power-efficient** implementation.

CameraX API for Image Acquisition

To achieve seamless real-time video capture, the Android implementation leveraged **CameraX**, a Jetpack-supported library specifically designed to provide optimized camera access. Unlike OpenCV's `VideoCapture` method, which struggled with high frame latency and inconsistent performance across different Android devices, CameraX provided:[19]

- **Adaptive frame rate management** for consistent video feed quality.

- **Efficient background execution**, preventing camera-related UI lag.
- **Hardware-accelerated image processing**, reducing CPU load.

By integrating CameraX, VisionS ensures that each frame was captured at a consistent interval without introducing excessive latency, a crucial factor for real-time driver monitoring.

MediaPipe's Java API for Landmark Detection

For facial feature extraction, **MediaPipe's Java API** was implemented, allowing the detection of **468 facial landmarks** in real-time. The challenge lay in ensuring efficient computation while minimizing CPU and memory usage. To optimize this process, several modifications were made:

- **Efficient computational graph pipeline**, reducing unnecessary operations.
- **Frame skipping strategy**, allowing landmark detection every alternate frame to balance processing speed and accuracy.
- **Dedicated GPU acceleration**, leveraging TensorFlow Lite's Neural Networks API (NNAPI) for faster inference.[20]

These optimizations enabled smooth execution even on mid-range Android devices, making VisionS a scalable and adaptable solution.

Reimplementation of Head Pose Estimation

The core of distraction detection relies on estimating the driver's head orientation. The Python version of VisionS used **SolvePnP** with OpenCV's NumPy arrays, but for Android, this had to be restructured using **OpenCV's Java bindings**. This transition resulted in:

- **Improved computational efficiency**, as `Mat` objects reduced memory overhead.
- **More accurate head pose estimation**, minimizing noise through matrix decompositions.
- **Enhanced numerical stability**, reducing frame-to-frame jitter using Kalman filtering.

By leveraging OpenCV's Java SDK, the system could calculate **yaw, pitch, and roll** in real time, ensuring precise distraction detection.

Implementation of Calibration Reset Mechanism

Recognizing that seating position and posture vary across users, a **calibration reset function** was added to allow dynamic recalibration of the driver's neutral head position. The system:

- Stores baseline head orientation values upon initial calibration.
- Allows users to reset calibration via a UI button if their seating position changes.
- Automatically adjusts deviation thresholds based on recalibrated values.

This feature ensures that VisionS remains adaptable, providing accurate distraction detection even in varied driving conditions.

Development of a Real-Time UI

A key factor in driver monitoring systems is providing **instant feedback** while maintaining a distraction-free interface. The VisionS UI was designed using **Android XML layouts**, incorporating:

- **Overlay-based alerts**, visually indicating distraction levels.
- **Real-time data visualization**, dynamically updating yaw, pitch, and roll angles.
- **Optimized background processing**, ensuring UI responsiveness while handling continuous video processing.

By replacing the Kivy-based UI used in Windows with a **native Android interface**, the application achieved significant performance gains and a more seamless user experience.

Performance Optimization and Resource Management

Given the limited processing power available in mobile devices, several optimizations were necessary to prevent overheating and battery drain:

- **Frame Skipping**: Instead of processing every frame, the algorithm evaluates only every alternate frame to balance accuracy and efficiency.
- **Multithreading**: Using `HandlerThread` and `ExecutorService`, the system ensures parallel non-blocking execution of video processing and distraction detection.

- **Hardware Acceleration**: Neural Networks API (NNAPI) and GPU acceleration were utilized to offload processing from the CPU.

These improvements allow VisionS to operate continuously without causing excessive performance degradation on mobile devices.

Notification and Alert System

To enhance driver awareness, VisionS implements a **multi-layered alert system** designed to function even when the application is running in the background. The notification system includes:

- **Visual Alerts**: On-screen warnings with a color-coded attention status.
- **Persistent Notifications**: Ensuring that distraction alerts remain active even if the app is minimized.
- **Auditory Feedback**: Optional beeps and sounds when prolonged distraction is detected.
- **Battery Optimization Features**: Users can toggle video feed visibility while keeping distraction detection active, reducing GPU workload and extending battery life.

By integrating these features, VisionS ensures a **non-intrusive yet effective** alert mechanism to promote driver attentiveness without overwhelming the user with unnecessary interruptions.

4.2.4 User Interface (UI) Development

The user interface of VisionS was designed to be intuitive, real-time, and user-friendly while ensuring that drivers stay focused on the road. Since the system continuously monitors driver attention using the front-facing camera, the UI needed to effectively convey information without causing unnecessary distractions. The Android implementation was structured using **Android's XML-based layout system** and **Jetpack components**, ensuring responsiveness, scalability, and ease of customization.

Camera Feed and Real-Time Display

A crucial component of VisionS is the real-time **PreviewView**, which displays the live feed from the front-facing camera. This feed is processed using **CameraX and OpenCV**, ensuring efficient video capture and minimal latency. Unlike traditional camera implementations, VisionS leverages **adaptive frame rate control** to optimize performance without overloading the device.

The XML implementation uses the `PreviewView` component to handle camera input:

```
<androidx.camera.view.PreviewView
    android:id="@+id/textureView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:visibility="visible"/>
```

This ensures that the live camera feed is seamlessly integrated into the UI while maintaining optimal performance across different Android devices.

Visual Indicators and Feedback System

The VisionS UI includes multiple **visual indicators** to provide clear feedback on the driver's attention status. The system employs a **color-coded warning system** that dynamically adjusts based on real-time monitoring data:

- **Green**: Driver is attentive; no corrective action needed.
- **Yellow**: Minor deviation detected; driver may need to refocus.
- **Red**: Significant distraction detected; immediate corrective action required.

These indicators are overlaid onto the camera feed using `TextView` elements:

```
<TextView
    android:id="@+id/distraction_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="20dp"
    android:text=""
    android:textColor="@android:color/holo_red_dark"
    android:textSize="30sp"
    android:textStyle="bold"
    android:background="#80000000"/>
```

This ensures real-time updates while maintaining a non-intrusive interface.

Distraction Detection and Alert System

To ensure immediate response to distractions, VisionS incorporates a **visual alert system** that changes the on-screen interface dynamically when the driver is distracted. When prolonged distraction is detected, the interface prominently displays a red warning overlay to notify the driver. This ensures immediate feedback without the need for intrusive audio or vibration-based alerts. The alert visibility is dynamically controlled by the distraction detection algorithm to prevent unnecessary warnings.

Calibration and Reset Controls

A **calibration reset function** is integrated into the UI, allowing drivers to recalibrate their **neutral head position** dynamically. This feature is essential for:

- Adjusting to different driving postures and seating positions.
- Ensuring consistency across different users sharing the same vehicle.
- Recalibrating after prolonged use to prevent drift in head position estimation.

The reset option is accessible through a simple button press:

```
<Button
    android:id="@+id/reset_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Reset"
    android:background="#333333"
    android:textColor="#FFFFFF"/>
```

This function allows recalibration in real-time without requiring a restart of the application, making it highly convenient for multi-user environments.

Adaptive UI for Different Screen Sizes

Since VisionS is designed for a broad range of Android devices, the UI was built to be **fully adaptive**. Key optimizations include:

- **Scalability across different screen sizes and resolutions** (smartphones, tablets, in-dash displays).
- **Minimalistic design** to ensure clarity while maintaining functionality.
- **Ease of navigation** for quick access reset calibration and toggle video feed.

By integrating these features, VisionS maintains a seamless user experience across all supported platforms. The UI is tested on multiple screen densities to ensure usability remains consistent across devices.

Video Feed Toggle for Battery Conservation

To enhance power efficiency, VisionS includes an option to **disable the video feed** while keeping distraction monitoring active. This feature significantly reduces **GPU and CPU usage**, thereby extending battery life. Despite the video feed being turned off, the system continues processing frames in the background, ensuring that monitoring remains uninterrupted.

The toggle function is controlled using a button:

```
<Button
    android:id="@+id/toggle_video_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Video Off"
    android:background="#333333"
    android:textColor="#FFFFFF"/>
```

Allowing users to efficiently manage power consumption. The system automatically adjusts processing loads when the video feed is disabled, ensuring minimal battery drain.

Driver Mode Toggle Button

To accommodate different driving orientations worldwide, VisionS includes a **Driver Mode Toggle Button**. This feature allows users to switch between **left-hand drive** and **right-hand drive** modes, ensuring the system adapts to regional driving norms.

When toggled, the system updates the **Driver tracking parameters** to align with the driver's position.

The toggle button is implemented in the UI as follows:

```
<Button
    android:id="@+id/toggle_side_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentStart="true"
    android:layout_marginBottom="40dp"
    android:layout_marginStart="16dp"
    android:text="Left"
    android:background="#333333"
    android:textColor="#FFFFFF"
    android:padding="10dp"/>
```

This button is positioned at the bottom-left of the interface, allowing easy access for users to switch driving modes. The text dynamically updates to reflect the selected mode, ensuring clarity for the driver.

Summary of UI Enhancements

The VisionS UI was built with a focus on **real-time responsiveness, user engagement, and power efficiency**. The final design incorporates:

- Real-time camera feed with visual overlays.
- Dynamic color-coded feedback for distraction levels.
- User-controlled calibration resets and driver modes.
- Adaptive design for different screen sizes and orientations.
- Battery-saving options through video feed toggling.
- Intelligent UI scaling for seamless usability across various screen sizes.

By integrating these elements, VisionS ensures that drivers receive meaningful feedback without being distracted, enhancing both usability and effectiveness. Shown below is the demonstration of the final implementation:

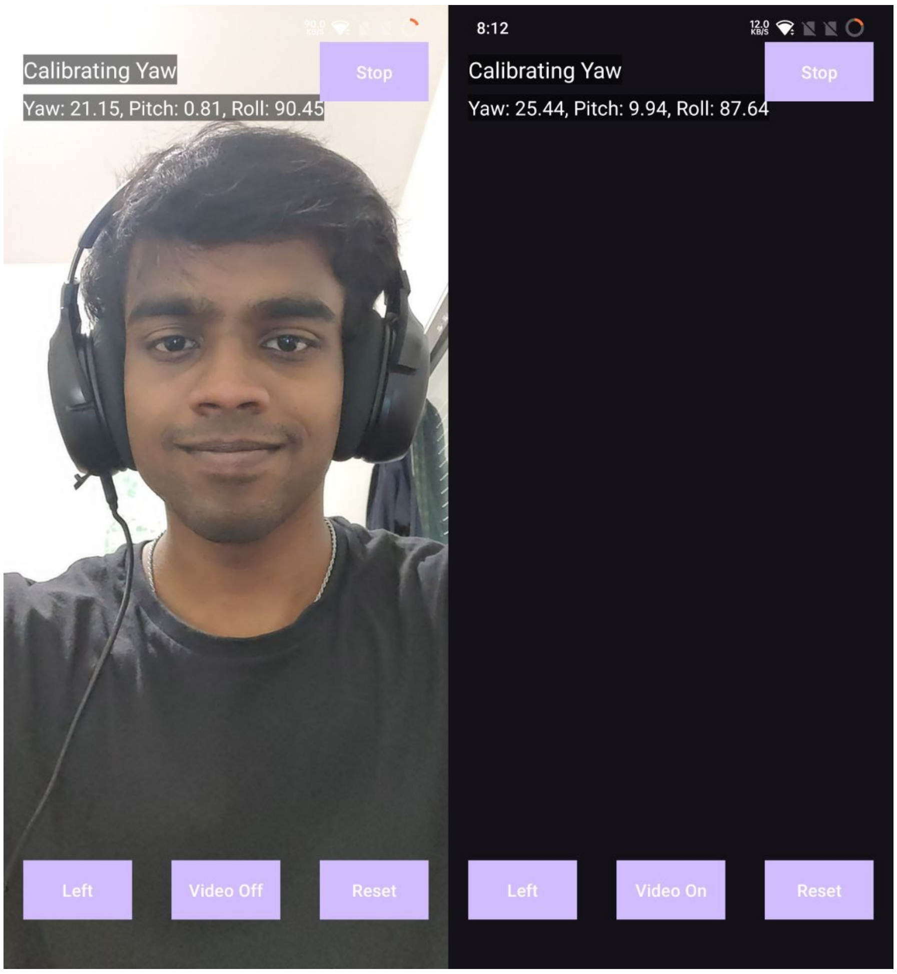

Figure 4.2: Java based Android application[13] Video toggled on and off

4.2.5 Final Remarks

The transition of VisionS to the Android platform presented a series of technical hurdles, primarily due to the lack of native Python support for MediaPipe and the constraints of deployment tools such as Buildozer and Chaquopy. Multiple attempts to integrate the system through these methods proved ineffective, necessitating a complete rework of the algorithm in Java. This transition allowed for seamless integration with Android-native technologies, including **CameraX and OpenCV**, ensuring optimal performance and real-time processing capabilities.

By shifting to a Java-based implementation, the VisionS application successfully achieved:

- **Reliable and efficient real-time distraction detection**, minimizing processing delays while maintaining accuracy.
- **Optimized computational performance**, leveraging multithreading and hardware acceleration for smooth operation.[21]
- **Seamless UI integration**, using XML-based layouts to provide dynamic, real-time feedback to the user.
- **Enhanced power efficiency**, reducing battery consumption while sustaining continuous monitoring.
- **Adaptive user calibration**, allowing for quick and easy recalibration based on individual driving postures and preferences.

Through these refinements, the Android version of VisionS has evolved into a fully functional mobile driver monitoring system. With its ability to process real-time video, deliver immediate feedback, and maintain high efficiency, VisionS stands as a significant step toward improving road safety and reducing driver distraction in everyday scenarios.

Chapter 5

Testing and Results

The testing phase plays a crucial role in evaluating the reliability, accuracy, and performance of VisionS. A well-defined testing process ensures that the system meets its intended objectives and can function effectively in different environments. This chapter is divided into two major sections: Algorithm Testing and Application Testing, along with a discussion of the results obtained from each testing phase.

Algorithm Testing focuses on evaluating VisionS in controlled conditions using pre-recorded videos with predefined distraction events. This allows for an objective analysis of detection accuracy and helps refine the underlying algorithms before deployment.

Application Testing assesses the performance of VisionS in real-world conditions, determining its reliability under varying lighting, movement, and environmental conditions. This ensures that VisionS can operate effectively in dynamic scenarios, where variables cannot always be controlled.

By conducting these tests, the system's strengths and limitations can be identified, leading to further optimizations and improvements. The results obtained from both testing phases are also discussed in this chapter.

5.1 Algorithm Testing

Algorithm Testing is essential for ensuring that VisionS can accurately identify distractions in controlled environments before transitioning to real-world applications. The testing workflow consisted of the following structured steps:

- **Dataset Preparation:** Recorded videos with known distraction windows were annotated to serve as the ground truth. This dataset included different types of distractions such as abrupt movements, environmental changes, and user actions. The dataset was designed to represent a diverse set of conditions, including different facial orientations, lighting conditions, and driver behaviors, ensuring that the model was exposed to a broad spectrum of real-world scenarios.

- **Automated Processing:** The pre-recorded videos were systematically fed into the VisionS application, which analyzed each frame to detect and log instances of distractions. The processing pipeline was optimized to handle variations in video resolution, frame rate, and noise levels to ensure reliable detection under different conditions.
- **Data Extraction:** The system generated output files in CSV format, detailing frame-wise distraction data. Each recorded frame was associated with timestamped logs, including extracted yaw, pitch, and roll values, as well as the corresponding distraction status. These logs were later used for model performance evaluation and trend analysis.
- **Result Validation:** The extracted data was visualized and analyzed by plotting distraction events over time. These results were cross-verified against the known dataset to determine the accuracy of VisionS in detecting distractions. Metrics such as sensitivity and specificity were calculated to assess performance.

Distraction Estimation Phases:

- **Indeterminable Phase:** This occurs when the driver is not looking at the road, but the system has not yet classified it as a distraction. Short-term deviations, such as brief glances away, are categorized under this phase to avoid false positives. This phase is managed by a buffer system, which temporarily holds frame data and evaluates the duration of inattention before making a classification.
- **Distracted Phase:** If the indeterminable phase is sustained for a predefined threshold duration, the system classifies the driver as distracted. This is determined by using a distraction counter, which increments over consecutive frames where inattention is detected. Once the counter surpasses the set threshold, the system triggers an alert. This ensures that only prolonged distractions trigger warnings, improving reliability and reducing unnecessary alerts.

The threshold for defining distraction windows was determined based on statistical studies of a driver's typical field of view and necessary visual focus points while driving. The chosen thresholds account for the natural head movements required to check the side mirrors, rearview mirror, dashboard, and speedometer.

Any deviation beyond these predefined angles was classified as a loss of road attention, ensuring that VisionS remains effective without misclassifying normal driving behavior as distraction.

To substantiate the methodology for defining distraction thresholds based on a driver's typical field of view and necessary visual attention, we refer the SAE J1050 standard, *Describing and Measuring the Driver's Field of View* [22]. This standard outlines methods for assessing both direct and indirect fields of view, as well as the extent of obstructions within those fields. It provides a framework for determining the horizontal and vertical ranges of a driver's direct ambinocular field of view, considering factors such as eye rotation limits and apertures.

By applying the principles from SAE J1050[22], we established thresholds that account for natural head movements required to check the side mirrors, the rearview mirror, dashboard, and the speedometer. This approach ensures that VisionS accurately differentiates between normal driving behaviors and actual distractions, thereby enhancing detection reliability and reducing false positives.

5.1.1 Refinements and Optimizations

To enhance accuracy, several refinements were implemented:

- Adjusting the attentive window to refine the criteria for when a driver is flagged as distracted.
- Optimizing the distraction time window to balance responsiveness with accuracy.
- Modifying transition durations between the indeterminable and distracted phases for smoother detection.
- Fine-tuning threshold parameters to minimize false positives while maintaining high sensitivity.

Once this baseline was established, the next phase of testing focused on real-world applicability. Initially, the algorithm underwent a longer calibration phase to fine-tune these parameters. Minor adjustments were made until VisionS consistently achieved **95% sensitivity and specificity** across all test datasets. This was validated by plotting distraction flags over time, such as in **Figure 5.1**, and cross-referencing them with the annotated ground truth.

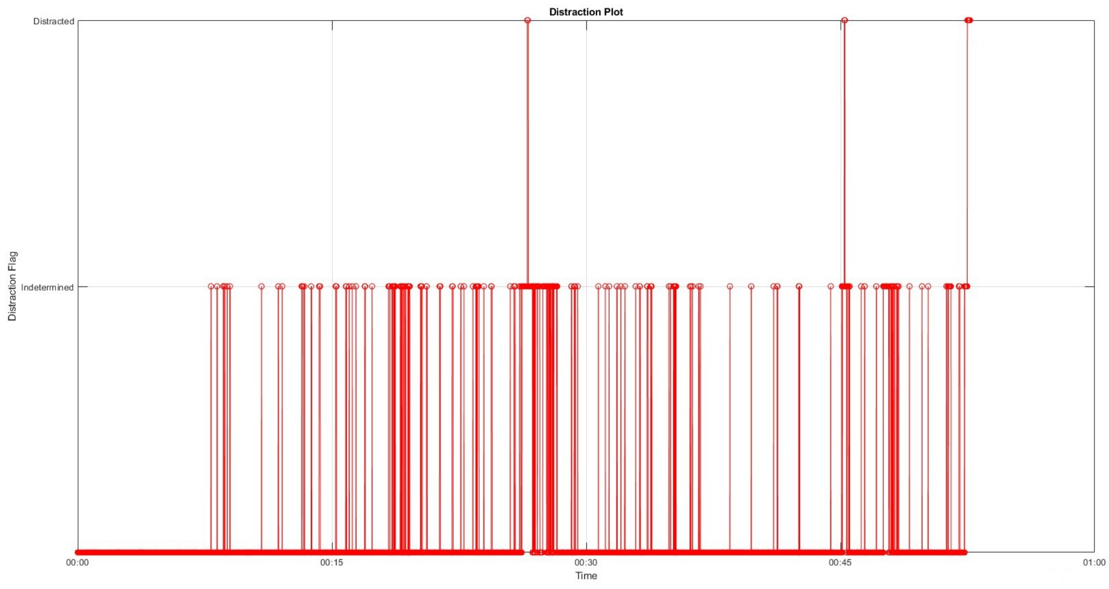

Figure 5.1: Distraction flag plot across time

To further improve usability, the calibration phase was systematically shortened, with multiple iterations evaluating its impact on sensitivity and specificity. The goal was to make VisionS capable of estimating driver distraction as early as possible without compromising accuracy. Through this iterative process, an optimal balance was achieved between the duration of the calibration and the detection reliability.

5.2 Application Testing

Application testing ensures that VisionS operates effectively across different platforms, including Windows and mobile environments, maintaining accuracy and responsiveness in real-world conditions. This involves rigorous validation of the algorithm's performance, ensuring it functions correctly under various hardware and software configurations. Additionally, stress tests were conducted to assess stability over extended periods, and usability testing ensures that the UI remains intuitive and efficient across all supported devices.

5.2.1 Windows Application Testing

The Windows application was developed primarily to facilitate algorithm execution without requiring an integrated development environment (IDE). This approach enabled structured testing by allowing VisionS to process pre-recorded datasets while also providing a real-time UI overlay for visualization.

The updated UI was designed to display all relevant information concisely, avoiding unnecessary data clutter. This enhancement allowed for efficient real-time visualization of the algorithm's outputs, ensuring clear tracking of distraction events. The same dataset used for algorithm validation was also utilized in the Windows application to confirm its reliability and consistency in processing input data.

The key objectives of Windows application testing were:

- **Algorithm Validation**: Ensuring that the algorithm was successfully compiled into an executable and produced consistent results across different test datasets.
- **Performance Assessment**: The application was tested across various hardware configurations to evaluate processing latency and efficiency in detecting distraction events. Testing began with high-performance systems featuring top-tier CPUs and GPUs, then extended to older hardware with slower processors and limited memory. This systematic approach allowed for a comprehensive assessment of performance variability and the impact of system specifications on execution times. While minor latency differences were noted, they were negligible and had no meaningful impact on functionality. The results confirmed that the algorithm was optimized to run smoothly across all tested hardware, maintaining consistent performance without noticeable degradation, regardless of hardware constraints.
- **UI Responsiveness**: Assessing the real-time overlay to confirm smooth visualization of tracking data without lag or synchronization issues. The UI was found to be highly responsive, updating in real-time without any noticeable delays. Additionally, the visual cue indicating driver distraction was prominently displayed, ensuring it was easily noticeable and effectively served its purpose in alerting the driver.
- **Execution Time Optimization**: To improve efficiency, only one out of every two frames was processed, reducing computational overhead without significant loss of information. Given that most test videos were recorded at 30FPS or higher, this optimization maintained the system's responsiveness while improving execution speed.

After verifying the reliability of the algorithm's compilation, the application was tested in a controlled simulated environment for real-time performance assessment. The system successfully processed live video feeds with minimal latency, demonstrating its capability for real-world deployment. Additionally, extended runtime testing confirmed the stability of the system, ensuring that no performance degradation occurred over prolonged use.

Initial compilations encountered crashes due to improper loading of the MediaPipe and OpenCV libraries, leading to instability in execution. The issue was diagnosed through a series of debugging steps, including analyzing error logs, testing different library versions, and isolating dependencies. It was identified that dynamic linking inconsistencies caused the libraries to fail upon execution. To resolve this, the libraries were compiled alongside the executable file, ensuring proper initialization and eliminating runtime errors. This issue was resolved by compiling these libraries alongside the executable file. While this solution increased the overall application size, it significantly improved launch speed and initial runtime performance, allowing for smoother execution without dependency-related failures.

The Windows application served as a foundational testing platform before transitioning to an Android application. By validating algorithm performance and ensuring reliable execution on a desktop environment, this phase confirmed the feasibility of porting VisionS to a mobile platform while maintaining its real-time processing capabilities.

5.2.2 Android Application

Given the challenges encountered during the compilation of the Android application, rigorous testing was conducted to ensure its stability, performance, and usability across various devices. The VisionS mobile application underwent extensive evaluation under different conditions to verify its reliability and responsiveness. This testing phase was structured around key objectives, including application stability, latency performance, logging accuracy, user interface functionality, power efficiency, and performance improvements achieved through Java-based implementation.

Application Stability and Performance

To ensure the VisionS application remained stable across different Android devices, various stress tests were performed. The primary focus was to identify and rectify issues related to crashes, dependency errors, and resource mismanagement. Debugging sessions and crash log analysis played a pivotal role in eliminating frequent application failures, ensuring that the app functioned reliably under different workloads and conditions. By implementing structured error-handling mechanisms, the application was made more resilient to unexpected failures, improving its overall dependability.

Performance evaluations examined the efficiency of the algorithm, particularly its ability to process real-time inputs without excessive lag. The transition to a Java-based implementation provided noticeable improvements in managing multithreaded operations, thereby enhancing overall performance and reducing latency.

These refinements helped optimize CPU usage and improved the application's ability to handle multiple background processes without significant slowdowns.

Testing Methodology

Device Compatibility Testing

The application was deployed on a diverse set of Android devices varying in chipset architecture, RAM capacity, and screen resolution. The goal was to measure its efficiency on both lower-end and high-end devices. By running the application across different hardware configurations, inconsistencies in behavior were identified and resolved. Additionally, compatibility checks were performed across different Android versions to guarantee consistent functionality and seamless user experience, ensuring that users with older devices could still benefit from the application without experiencing performance issues.

Latency and Real-Time Processing

Testing involved assessing the app's performance under various conditions, such as changes in lighting environments, multitasking scenarios, and different background processes. The evaluation of frame processing speed and algorithm response times ensured minimal delays in distraction detection. The app's stability was tested while running alongside resource-intensive applications to measure its robustness in multitasking scenarios. The implementation of Java-based multi-threading contributed to improved processing efficiency by optimizing the allocation of system resources. Through these tests, it was confirmed that latency was reduced significantly, allowing the system to detect distractions more accurately and with minimal delays.

Logging and Data Accuracy

To validate the integrity of the system logs, extensive tests were performed on recorded distraction events, user inputs, and calibration resets. These tests confirmed that logs were generated accurately and consistently, even under unexpected circumstances such as power failures or app crashes. Error-handling mechanisms were assessed to ensure data integrity without missing or duplicating log entries. Timestamp precision was also reviewed to facilitate effective event tracking and post-processing analysis. Improvements in the logging mechanism ensured that every recorded event was correctly classified and stored, preventing redundant or lost data entries.

User Interface Validation

The usability and responsiveness of the VisionS application's UI were evaluated through functional tests on all interactive elements:

- **Start/Stop Algorithm button:** Verified its ability to correctly initialize and terminate the distraction detection process without lag or failure.

- **Reset Calibration function:** Ensured calibration parameters reverted to default values upon reset, maintaining system consistency.

- **Toggle Video Feed feature:** Assessed to confirm that the live video feed could be enabled or disabled without performance degradation, avoiding unnecessary strain on system resources.

- **Left/Right Driver Mode functionality:** Validated to guarantee the proper adjustment of detection parameters when switching between driving modes.

- **General UI responsiveness:** Monitored to ensure that all interactive elements reacted instantly to user inputs, even when the system was under high CPU or GPU loads.

These UI refinements helped eliminate delays and enhanced the overall user experience.

Observations and Results

The initial testing phase identified multiple stability issues, including frequent crashes due to improper deletion and recreation of class objects. These problems were systematically addressed through debugging and memory management optimization, which significantly enhanced the application's stability. The logging functionality was confirmed to be highly accurate, with no instances of data loss or incorrect event recordings. UI components underwent refinements to improve alignment and responsiveness, ensuring seamless user interaction. By reducing redundant computations and optimizing rendering techniques, the application became more efficient and responsive.

Furthermore, the application maintained efficient CPU and RAM utilization, preventing performance slowdowns even on budget-friendly devices. The adoption of Java-based multi-threading significantly enhanced computational resource allocation, allowing the application to operate more efficiently across various devices. As a result, real-time processing latency was significantly minimized, enabling near-instantaneous distraction detection. These improvements collectively ensured that the application was not only functional but also optimized for prolonged real-world use without unnecessary resource consumption.

The final compiled version of the VisionS application demonstrated robust stability, even when subjected to intensive multitasking conditions. Moving forward, additional refinements will be directed towards optimizing the application for lower-end devices, refining memory management techniques to further enhance efficiency, and incorporating additional UI enhancements to improve accessibility and user engagement. By continuously refining the application based on user feedback and further testing, VisionS aims to provide a reliable and highly responsive tool for distraction detection in real-world driving scenarios.

Chapter 6

Enhanced Driver Drowsiness Detection Through Data Fusion

Driver fatigue is a significant factor in road accidents worldwide, impairing cognitive function and reaction times. Traditional drowsiness detection systems often rely on either physiological or behavioral indicators, but single-source methods may be prone to inaccuracies and false positives. To improve accuracy and reliability, a data fusion approach combining multiple data sources has been developed. This paper introduces an integrated algorithm that merges physiological and behavioral indicators to provide a more comprehensive assessment of driver alertness.

6.1 PredictS: Wearable Sensor-Based Drowsiness Detection Algorithm

PredictS [23] is an exclusive algorithm developed by Sleep Advice Technologies in partnership with Garmin smart wearable technology. It harnesses the capabilities of Garmin sensors to track key physiological parameters, including Heart Rate (HR), Heart Rate Variability (HRV), and SpO2 levels. The system is designed to determine whether a driver is experiencing drowsiness or on the verge of falling asleep while operating a vehicle. PredictS is a key component of the FleetPredictS system, which promotes safer driving and enables centralized monitoring of driver health metrics.

The algorithm incorporates a patented method for predicting sleep onset by evaluating the Autonomic Nervous System (ANS) and its related subsystems. It continuously observes transitions from wakefulness to sleep by analyzing HR and

HRV data obtained from a wearable device. Through a sliding window technique applied to 20-second data segments, the system detects variations in HR and HRV to categorize the driver's state into five distinct levels: Calibration, Awake, Low Drowsiness Level, Medium Drowsiness Level, and High Drowsiness Level.

PredictS has undergone extensive validation through controlled experiments using the AVL dynamic car simulator in Graz, Austria. In one study, 15 participants completed the Maintenance Wakefulness Test (MWT) under the supervision of a sleep expert. Further real-world testing involved professional drivers from Chrono Express, covering more than 13,000 kilometers to confirm the system's effectiveness.

6.2 Enhancing Drowsiness Detection Through Behavioral Analysis

The VisionS algorithm works alongside PredictS, reinforcing its drowsiness detection capabilities by analyzing head movements. Drowsy drivers often exhibit reduced situational awareness, inconsistent speed maintenance, and delayed reactions. Research has shown that fatigue impairs reaction times, vigilance, and information processing, increasing accident risk [24].

Studies have found that fatigued drivers display reduced vehicle control, reflected in steering wheel angle variations and head movement changes [25]. Monitoring head movement in real time provides additional drowsiness indicators, complementing the physiological signals detected by PredictS.

VisionS classifies driver drowsiness using a predefined threshold based on head movement patterns. This threshold is determined through extensive data analysis, considering factors such as the frequency and amplitude of head movements, changes in posture, and variations in gaze direction. By combining the drowsiness flag from VisionS with that of PredictS, false positives are significantly reduced while improving overall sensitivity. If both systems detect drowsiness simultaneously, a high-confidence alert is triggered, ensuring timely intervention to prevent accidents.

PredictS (Physiological)	VisionS (Behavioral)	Drowsiness Detection Output
0	0	0 (No Drowsiness)
0	1	0 (Possible False Positive)
1	0	0 (Possible False Positive)
1	1	1 (High-Confidence Alert)

Table 6.1: Drowsiness Detection Decision Framework

This Boolean framework, established by **table 6.1**, ensures that a driver is only flagged for drowsiness when both physiological and behavioral indicators align, significantly increasing the reliability of the system. By integrating both data sources, the hybrid algorithm will provide a more robust and accurate assessment of driver fatigue, reducing instances of false alarms while maintaining high sensitivity in detecting actual drowsiness events.

By combining the strengths of PredictS and VisionS, this hybrid approach will ensure a more robust drowsiness detection system. It will move beyond single-source reliance and instead employ a multi-modal strategy that integrates physiological and behavioral indicators, significantly improving both reliability and accuracy in real-world driving conditions.

This multi-modal approach will enable an adaptive thresholding system that continuously refines itself based on real-world driver data, making it increasingly effective in practical applications. Future developments will likely involve dynamic calibration techniques, adjusting sensitivity based on individual driver profiles to further improve predictive accuracy.

This will also open up the possibility of VisionS functioning as a standalone driver monitoring system when wearable devices are unavailable. While its accuracy might be lower than that of PredictS or the integrated system, it will remain valuable for road safety enhancement. By providing real-time awareness and intervention, VisionS will help mitigate drowsy driving risks, particularly in settings where wearable technology is limited.

Beyond drowsiness detection, VisionS could enable proactive safety measures. By continuously monitoring head movement patterns, it might detect early fatigue indicators and suggest rest breaks before critical drowsiness levels are reached. This predictive capability could enhance fleet management by optimizing driver scheduling and reducing fatigue-related accidents. Future advancements may include gaze tracking and posture analysis, further refining its ability to detect signs of fatigue and inattention.

6.3 Implementation and Observations

6.3.1 Implementation

The algorithm is implemented and thoroughly tested in an offline environment using several available datasets. The process is designed to determine its viability, evaluate the validity of its fundamental concepts, and measure its accuracy and reliability quantitatively. Through systematic testing, critical performance metrics are evaluated to ensure the algorithm's adherence to the suggested standards for effectiveness and reliability in detecting drowsiness. Testing is performed in the following manner:

First, the standard deviation measures obtained from VisionS are grouped based on a predetermined threshold to determine the drowsiness indicator. This ensures that the thresholds are determined with an optimal balance between accuracy, sensitivity, and specificity. The approach involves choosing different threshold values and analyzing their effects in detail on classification effectiveness to determine the optimal range.

Next, filtered VisionS data is merged into the PredictS data. An authentic case of drowsiness is recorded only when both algorithms, when tested in isolation, detect drowsiness. Any other scenario is labeled as a false event. This integration phase of the detection system increases the trustworthiness of the detection system by reducing false positives and ensuring that a detected event is validated by more than one independent source, as opposed to a single data source.

Then, the combined and filtered dataset is subjected to a rigorous cross-validation process using the available ground truth data. The ground truth is obtained from medical devices that track brainwave patterns and other parameters associated with drowsiness and has been validated by medical experts. The cross-validation process includes statistical tests and performance metric evaluations, thus ensuring that the model is consistent with clinically accepted indicators of drowsiness.

Finally, performance is evaluated by plotting the Receiver-Operating CurveROC curve, along with sensitivity and specificity values across the pre-specified range of thresholds. This enables critical assessment of the model's performance, with the goal of determining the most beneficial tradeoff between sensitivity and specificity. Analysis of the ROC curve provides insightful information on classification performance at various threshold values, offering a unique visualization of the strengths of the algorithm as well as potential areas for improvement.

6.3.2 Observations

The greatest and limiting challenge encountered within the testing process was the lacking availability of the necessary datasets that were needed in order to execute and evaluate the algorithm. Only nine datasets were available at the time of evaluation, and they posed significant constraints on both the range and variety of the test process. The problem was made worse by the fact that some datasets contained inconsistencies that greatly compromised their usability. Some datasets lacked time-aligned synchronized data between PredictS and VisionS, which complicated the ability to create practical correlations between these two datasets. Other datasets contained no instances of recorded drowsiness, which made them insufficient for constructive validation attempts.

These limitations directly affected the ability to conduct an extensive analysis of the algorithm's effectiveness in various contexts. The narrow scope of the repository of datasets severely hindered the ability to generalize the findings to a wide range of real-world situations. Despite these limitations, the controlled experiment proceeded with the pertinent datasets available.

The graph below depicts the results of one such experiment, which are displayed visually for ease of understanding and interpretation. The Receiver-Operating Curve(ROC) is plotted as True-Positive Rate(TPR) vs False-Positive Rate(FPR). The area under the curve is a measure of how well the model can distinguish between the two and is used to evaluate the accuracy. The next plot is the sensitivity and the specificity values plotted against a range of thresholds to visualize the effects different thresholds for classification have on the model.

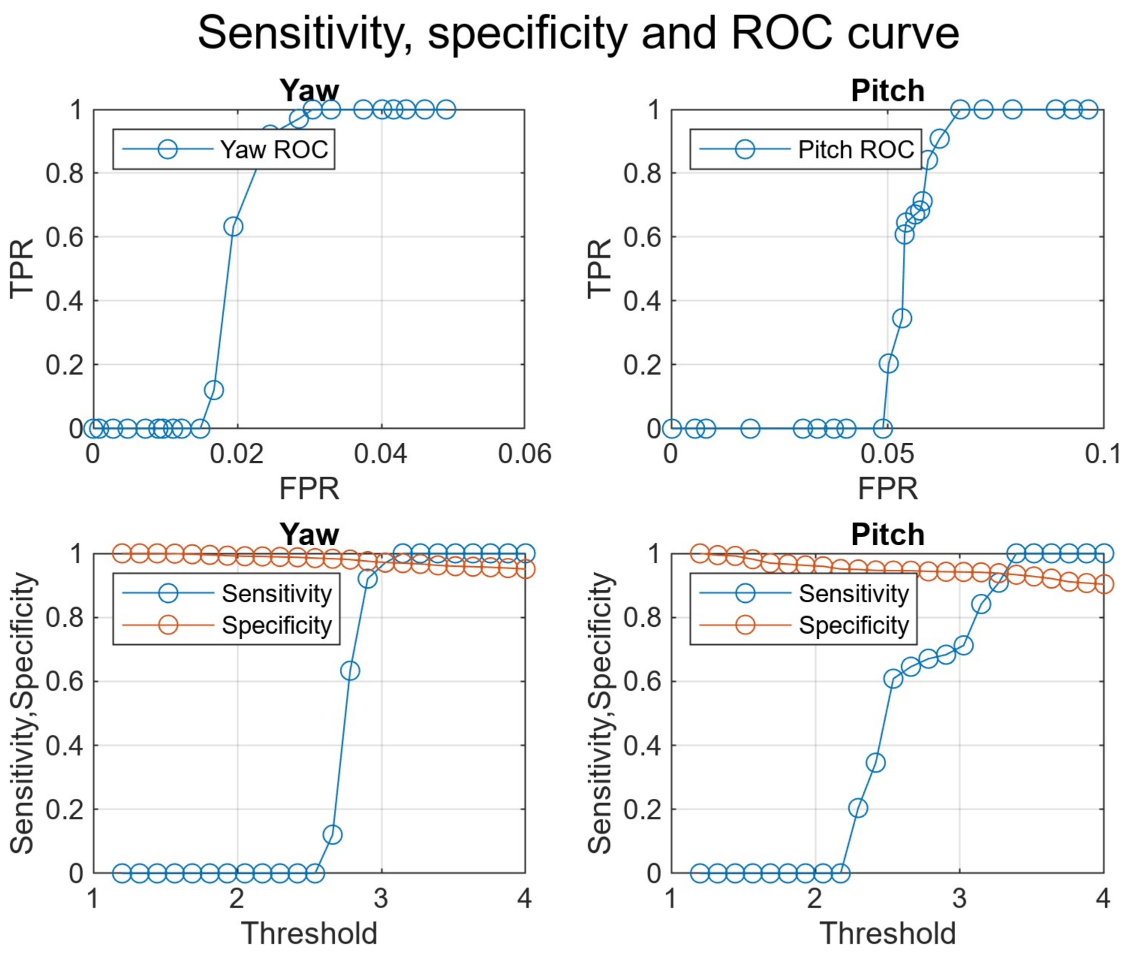

Figure 6.1: ROC, Sensitivity and Specificity plot

The results depicted by the plots, such as in **Figure 6.1**, show that the algorithm has a specificity of over 98%, thus successfully eliminating nearly all cases of false positive alarms, which can also be observed from the ROC plot. At the same time, the sensitivity, which was around zero at the lower threshold range, rapidly increases to over 95%. This significant improvement in sensitivity indicates that the algorithm is extremely efficient in detecting drowsiness patterns when the threshold is over 2.5. The results obtained from these datasets show considerable promise, confirming the viability of the detection mechanism within the parameters tested and highlighting the possibility of further improvement to enhance its reliability.

While the current results are not conclusive evidence of the algorithm's suitability for application in real-world scenarios, they firmly indicate its feasibility. The outcomes confirm that the underlying methodology is theoretically sound and amenable to refinement. For the model parameters to be optimized and its effectiveness to be tested under various driving conditions and populations, further systematic and thorough analyses will be required in future studies.

By refining the testing procedures, optimizing the classification methods, and integrating a greater range of datasets, the accuracy, sensitivity, and specificity of the algorithm can be optimized. This optimization will ensure that the algorithm meets the strict requirements needed for effective operation and general acceptance in driver monitoring systems to be implemented and used in real time.

Chapter 7

Conclusion

The VisionS project represents a significant advance in the development of real-time driver monitoring and awareness systems, leveraging computer vision and artificial intelligence techniques to enhance road safety for drivers. Through the leveraging of the potential of Google's MediaPipe, OpenCV, and sophisticated algorithmic frameworks, VisionS accurately detects and tracks signs of distraction, head pose, and potential indicators of driver fatigue. Extensive tests performed across diverse environmental settings have confirmed the system's dependability, accuracy, and usability in real-world settings, thus making it a critical advancement in driver safety technology.

The development path focused on the improvement of computational efficiency, the increase of detection accuracy, and the guarantee of effortless adaptability in varied driving situations. VisionS has undergone iterative refinement from its initial conceptual phase to final deployment to overcome technical challenges and enhance its effectiveness, resulting in a very reliable monitoring system for real-time analysis of driver behavior.

7.1 Key Achievements

One of the key accomplishments of VisionS is the successful development and deployment of a cross-platform system intended for driver behavior monitoring. The system successfully processes video data in real-time and provides relevant insights into driver behavior with minimal latency. This accomplishment was achieved by leveraging new algorithmic innovations, hardware performance improvements, and field verification testing.

The creation of a unified application for both Windows and Android platforms enabled smooth integration with various vehicular environments. A core component of this success was the application's ability to process video data in real-time without compromising on computational efficiency. Significant improvement was achieved through the optimization of the processing speed by systematically removing alternate frames, thus effectively reducing computational load without compromising on detection accuracy. Additionally, the implementation of a dynamic calibration mechanism enabled the system to adapt to different camera angles and driver positions, thus ensuring consistent performance for different vehicle configurations. Moreover, a comprehensive logging system was implemented to record important events for future analysis, aiding decision-making, forensic analyses, and continuous improvement of the system.

An integral aspect of VisionS was the improvement of its core algorithm. Extensive parameter tuning and adjustments to the model were done to reduce the rate of false positives, resulting in a strong classification system capable of distinguishing between degrees of driver distraction and attention. The final version of the system showed considerable improvements in empirical tests, outperforming conventional monitoring systems in various and challenging environments.

One of the most important advancements made in the VisionS system has been the inclusion of data fusion techniques with PredictS. Through the merging of real-time video data and predictive modeling with sensor-based information, VisionS has enhanced its ability to measure driver distraction and fatigue in a more comprehensive context. This combination enables more sophisticated decision-making mechanisms since the composite data sources provide an in-depth assessment of the performance of a driver. VisionS and PredictS's joint efforts have been able to illustrate the viability of a multi-modal approach to tracking drivers, and it has significant implications for designing more effective and proactive safety interventions.

7.2 Challenges and Limitations

Despite these achievements, the development of VisionS was faced with a variety of practical and technical challenges. One of the main challenges was how to balance library dependencies and runtime performance. Early versions of the system were plagued by issues with dependency conflicts, poor memory utilization, and performance limitations. These issues were resolved by the application of optimized compilation methods, memory allocation improvements, and the strategic use of multi-threading to improve overall execution time.

The other challenge was to ensure the robustness of the system against the variation typical of testing environments in real life. The driver monitoring system needed to adapt to varying levels of lighting, variations in driving behaviors, and motion artifacts from road vibrations. Thorough validation was needed to calibrate the model for robust performance under different scenarios, so that changes in ambient light or sudden driver motion did not compromise detection accuracy. The limitations imposed by platforms have significantly impacted the system's deployment. While VisionS has been successfully integrated into the Windows and Android environments, the absence of an iOS equivalent limits access to a broader market. Creating a compatible version for iOS is an attractive opportunity, requiring adjustments to ensure compatibility with Apple's hardware infrastructure and software framework.

The application of multi-threaded processing for improving computational efficiency has resulted in synchronization-related problems. Coordination of concurrent activities in real-time video processing requires careful control of shared resources to prevent data inconsistency. Sophisticated thread synchronization techniques were critical to maintaining system stability while improving responsiveness at the same time.

A crucial challenge faced was the need to ensure the model's ability to generalize well for different user demographics. Given the immense variability in driving habits, facial features, and in-car conditions, the need arose to test the model with extensive datasets. While VisionS has been shown to achieve high accuracy in controlled test settings, further verification in real-life scenarios with diverse user populations is important to tune detection parameters and increase overall reliability.

The integration of data fusion techniques into the VisionS system presented further complications. PredictS, which relies on multiple sensor data to sense fatigue, needed to be calibrated and synchronized with VisionS's video-based analysis pipeline. It was necessary to make sure that the data from both systems complement each other without causing noise or inconsistencies, an important step in the refinement phase. While initial trials have shown promising results, ongoing development must be done to create a unified fusion mechanism that can operate effectively in varying driving conditions.

7.3 Future enhancements and improvements

The VisionS system, while currently functioning well, has great potential for improvement. Future improvements may focus on making it more accessible, expanding its feature set, and making it easier to integrate with other intelligent transportation systems.

7.3.1 iOS Application

The creation of an iOS variant of the VisionS application is expected to advance its usability and foster wider adoption across a more extensive demographic, thus providing access to users on Apple devices. Given the large user base attached to iOS, this expansion will require rigorous testing to ensure seamless integration into Apple's hardware and software environments. Maintaining compatibility with various iPhone and iPad devices, adherence to Apple's strict App Store guidelines, and optimization for exclusive iOS features like Face ID and haptic feedback will be critical to provide an optimum user experience and achieve maximum adoption levels.

7.3.2 Extended Real-World Testing and Validation

The deployment of VisionS in a wider range of geographic locations, atmospheric conditions, and driving environments is expected to enhance its generalizability and reliability. To achieve this goal, extensive field tests need to be conducted in urban, suburban, and rural settings to assess the system's effectiveness in the presence of varying traffic densities, road conditions, and culturally driven driving habits. Collaboration with automotive original equipment manufacturers, government agencies, and academic institutions will be critical in refining the system's algorithms and meeting international safety standards. The integration of large datasets with iterative optimization based on real-world experience will enable VisionS to provide accurate, flexible, and reliable driver monitoring, ultimately leading to improved safety outcomes and a decrease in accident rates.

The integration of VisionS with in-car safety features, such as adaptive cruise control and automatic emergency braking, enables the deployment of advanced accident prevention techniques through real-time threat detection and response. Artificial intelligence model optimization for edge computing is critical to ensure safety features function with minimal latency and reduced energy consumption. Optimizing AI models for best performance on embedded automotive platforms, VisionS can provide real-time decision-making capabilities without excessive cloud infrastructure reliance, thus increasing its feasibility for widespread integration across the automotive industry.

7.3.3 Improved Data Fusion via PredictS and Multi-Modal Driver State Assessment

The integration of the PredictS algorithm with other predictive analytics structures is in the initial stages of offline testing. The effort involves the gathering and analysis of extensive datasets from varied sources, which capture a range of road types, driving behaviors, and environmental factors, with the ultimate goal of guaranteeing the model's consistency and accuracy. A major challenge at this stage is the creation of a dataset that completely encompasses the entire on-road driving situation spectrum, from urban traffic to highway driving, in addition to factoring in variables such as changes in weather patterns, different driver fatigue levels, and vehicle performance indicators.

During the evaluation period, machine learning techniques will be deployed to enhance PredictS's ability to recognize driving patterns, detect anomalies, and predict potential danger. This is done through constant algorithm improvement using both supervised and unsupervised learning methods, thus ensuring the system adapts to new information while minimizing cases of false negatives and positives. In addition, synthetic data augmentation methods will be used to deal with the limitation of real data in situations with constraints, thereby improving the generalizability of the model to new situations.

Beyond offline evaluation, a phased transition to real-time application is crucial. Initial on-road testing will be conducted in controlled environments to validate the system's predictions before expanding to open-road trials. Collaboration with automotive manufacturers, regulatory bodies, and research institutions will be essential to integrate PredictS seamlessly Future enhancements could also involve multi-modal sensor fusion, to provide a more holistic assessment of driver alertness and cognitive fatigue. These advancements will enable VisionS and PredictS to become a proactive safety mechanism, capable of issuing early warnings and assisting in accident prevention before critical situations arise.

7.4 Final thoughts

By highlighting these areas for improvement, VisionS has the potential to evolve into an innovative driver monitoring system that can be broadly implemented and play a significant role in road safety. With the development of automotive technology moving forward, VisionS stands to play a critical role in reducing accident rates, enhancing driver alertness, and ultimately saving lives on the road. The VisionS system is an important leap forward in the area of real-time driver monitoring and situational awareness, with the capability to reduce vehicular mishaps and overall transportation safety. While some improvements are needed as well as an expansion of usage, this system lays a sound foundation for future developments in driver monitoring technologies. Research, technology development, and collaborative efforts in the future will be important for maximizing and extending the use of this technology toward global road safety.

Bibliography

[1] World Health Organization. «Road Safety Facts». In: (2021) (cit. on p. 3).

[2] National Highway Traffic Safety Administration. «Distracted Driving Report». In: (2022) (cit. on p. 3).

[3] JL Harbluk, YI Noy, PL Trbovich, and M Eizenman. «An on-road assessment of cognitive distraction». In: (2007) (cit. on p. 3).

[4] Centers for Disease Control and Prevention. «Road Traffic Accident Costs». In: (2021) (cit. on p. 3).

[5] J Feng, Q Zhang, and W Zhang. «Effects of cognitive load on driving performance». In: (2020) (cit. on p. 4).

[6] Y Dong, Z Hu, K Uchimura, and N Murayama. «Driver inattention monitoring system for intelligent vehicles». In: (2011) (cit. on p. 4).

[7] Z Liu, Y Shi, and H Cheng. «A review of driver monitoring systems». In: (2016) (cit. on p. 4).

[8] M Johnson, L Williams, and D Patterson. «Fleet vehicle safety improvements through monitoring systems». In: (2019) (cit. on p. 4).

[9] JD Lee. «Cognitive distraction: An overview». In: (2009) (cit. on p. 4).

[10] Privacy International. «Data Protection in Driver Monitoring». In: (2019) (cit. on p. 5).

[11] AAA Foundation for Traffic Safety. «Bias in AI-Driven Road Safety Technologies». In: (2020) (cit. on p. 5).

[12] *Google Developers: MediaPipe Framework*. `https://mediapipe.dev`. 2023 (cit. on p. 6).

[13] *Google Developers: MediaPipe Face Landmark*. `https://ai.google.dev/edge/mediapipe/solutions/vision/face_landmarker`. 2023 (cit. on pp. 6, 7, 15, 35).

[14] *OpenCV: Open Source Computer Vision Library*. `https://opencv.org`. 2023 (cit. on p. 8).

[15] Chaquopy Developers. «Chaquopy - Python in Android Studio». In: (2024). Accessed on May 2024. URL: https://chaquo.com/chaquopy/ (cit. on pp. 25, 26).

[16] Buildozer Community. «Buildozer Documentation». In: (2024). Accessed on May 2024. URL: https://buildozer.readthedocs.io/en/latest/ (cit. on p. 25).

[17] Google Developers. «MediaPipe on Android». In: (2024). Accessed on May 2024. URL: https://developers.google.com/mediapipe (cit. on p. 26).

[18] OpenCV Team. «OpenCV Android SDK Documentation». In: (2024). Accessed on May 2024. URL: https://docs.opencv.org/master/d0/d3d/tutorial_android_dev_intro.html (cit. on p. 26).

[19] Google Developers. «CameraX API for Android». In: (2024). Accessed on May 2024. URL: https://developer.android.com/training/camerax (cit. on p. 27).

[20] Android Developers. «Neural Networks API (NNAPI)». In: (2024). Accessed on May 2024. URL: https://developer.android.com/ndk/guides/neuralnetworks (cit. on p. 28).

[21] Android Developers. «Android Performance Optimization». In: (2024). Accessed on May 2024. URL: https://developer.android.com/topic/performance (cit. on p. 36).

[22] SAE International. «SAE J1050: Describing and Measuring the Driver's Field of View». In: (2009) (cit. on p. 39).

[23] «Drowsiness Detection Using PredictS». In: *IEEE Xplore* (2024). URL: https://ieeexplore.ieee.org/abstract/document/10448558 (cit. on p. 46).

[24] European Road Safety Observatory. «Fatigue and Driving Behaviour». In: (2024). URL: https://road-safety.transport.ec.europa.eu/european-road-safety-observatory/statistics-and-analysis-archive/fatigue/driving-behaviour_en (cit. on p. 47).

[25] ScienceDirect. «Effects of Fatigue on Driving Performance». In: (2024). URL: https://www.sciencedirect.com/science/article/abs/pii/S1361920917306582 (cit. on p. 47).