DRIVER DROWSINESS DETECTION SYSTEM

A Project Report

Project Report submitted in partial fulfillment of the requirements for the award of the degree of B.Tech. in Institute of Engineering & Technology (Sitapur Road, Lucknow, Uttar Pradesh, India)

by

Shani Yadav (1900520320048)

Shishir Bhargav (1900520320051)

Prachi Singh (1900520320035)

Under the Guidance of

Er. Richa Pathak

(Assistant Professor, Electronics and Instrumentation Engg. Department)

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING INSTITUTE OF ENGINEERING & TECHNOLOGY (Sitapur Road, Lucknow, Uttar Pradesh, India)

June 2023

ELECTRONICS AND INSTRUMENTATION ENGINEERING
INSTITUTE OF ENGINEERING AND TECHNOLOGY, LUCKNOW
An Autonomous Constituent Institute of Dr. A.P.J. Abdul Kalam Technical University, U.P., Lucknow

CERTIFICATE

This is to certify that the project entitled "DRIVER DROWSINESS DETECTION SYSTEM" is a bonafide work of "Shishir Bhargav (1900520320051), Shani Yadav (1900520320048) and Prachi Singh (1900520320048)" submitted in partial fulfillment of the requirements for the award of the degree of BACHELOR OF TECHNOLOGY IN ELECTRONICS AND INSTRUMENTATION ENGINEERING, at INSTITUTE OF ENGINEERING & TECHNOLOGY.

Er. Richa Pathak
Project Supervisor

Dr. R.C.S. Chauhan
Convenor
Electronics and
Instrumentation
Engineering

DECLARATION

We, hereby, declare that this project report entitled "Driver Drowsiness Detection System using Python Libraries and Android App with Google Vision API Integration" is a result of our original work and has not been submitted elsewhere for any other purpose.

I affirm that the content presented in this report represents my own findings, ideas, and analysis, derived from extensive research and development efforts. Any external sources of information utilized in this project have been duly acknowledged and referenced.

I take full responsibility for the accuracy, authenticity, and integrity of the content presented in this report. I have adhered to ethical practices during the project development, ensuring the protection of participants' rights and privacy.

Dated: 7th June 2023
Place: Lucknow

Shani Yadav
1900520320048
Shishir
Bhargava
1900520320051
Prachi Singh
190052032003
3

Acknowledgements

I would like to express my deep and sincere gratitude to my guide(s), Ms.. Richa Pathak, for her unflagging support and continuous encouragement throughout the project work. Without her guidance and persistent help this report would not have been possible.

At last I must express my sincere heartfelt gratitude to all the staff members of the Electronics Engineering Department who helped us directly or indirectly during this course of work.

Shishir Bhargav (1900520320051)

Prachi Singh (1900520320035)

Shani Yadav (1900520320048)

Abstract

The report proposed the results and solutions on the limited implementation of the various techniques that are introduced in the project. Whereas the implementation of the project gives the real-world idea of how the system works and what changes can be done in order to improve the utility of the overall system.

The Driver Drowsiness Detection System is an essential application in today's era of increasing road accidents caused by drowsy drivers. This project focuses on the development of a robust and efficient system that utilizes Python programming language, Google Vision API, and the SeekBar component in Android for real-time detection of driver drowsiness.

The proposed system employs computer vision techniques to monitor the driver's facial features and detect signs of drowsiness. By utilizing OpenCV and dlib libraries in Python, the system analyzes live video feeds from a camera installed inside the vehicle. It identifies crucial indicators of drowsiness, such as eye closure duration, yawning, and head position changes.

To enhance the usability and accessibility of the system, an Android application is developed using Java and integrated with the Python-based drowsiness detection system. The application provides a user-friendly interface that allows drivers to easily interact with the system. It incorporates the SeekBar component, enabling drivers to set their preferred sensitivity level for drowsiness detection.

Table of contents

5.7 Adjusting sensitivity using seekbar

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The development of technology allows us to introduce more advanced solutions in standard of living. As per the records each year about 100,000 crashes get reported involving drowsy driving. The exact figure would be far more. Facial expressions can offer deep insights into many physiological conditions of the body. There are innumerable algorithms and techniques available for face detection which is the fundamental commencement within the process. Drowsiness in humans is characterized by some very specific movements and facial expressions- e.g.- the eyes begin to shut.

To encounter this worldwide problem, an answer is tracking eyes to detect drowsiness and classify a driver drowsy. For real time application of the model, the input video is acquired by mounting a camera on the dashboard of the car and capturing the driver's face.

The Dlib model is trained to spot 68 facial landmarks, from which the drowsiness features are extracted, and the driver is alerted if drowsiness is detected. A lot of research is done in the field of driving safety to reduce the number of accidents.

The primary objective of this project is to create a robust and efficient driver drowsiness detection system that can accurately identify drowsiness in real-time and promptly alert the driver to prevent accidents. By leveraging the power of Python libraries and integrating them with the Android platform and Google Vision API, this system aims to provide a comprehensive and user-friendly solution.

The Driver Drowsiness Detection System project is designed to address the critical issue of drowsy driving and improve road safety. By utilizing Python libraries, an Android app with the SeekBar component, and integration with the Google Vision API, the system aims to provide a comprehensive solution for real-time drowsiness detection. The project methodology encompasses data collection, algorithm development, alert mechanisms, sensitivity adjustment, and thorough testing to ensure the creation of a robust and efficient system that can effectively identify driver drowsiness and alert drivers promptly.

CHAPTER 2

METHODOLOGY

The suggested method for detecting tiredness in drivers operates on two levels. The procedure begins with the camera recording live video frames, which are then transferred to a local server. The Dlib library is used on the server to identify facial landmarks, and a threshold value is used to determine whether or not the driver is sleepy.

The EAR (Eye Aspect Ratio) is then computed using these face landmarks and given to the driver. The EAR value obtained at the application's end is compared to a threshold value of 0.25 in our system. The driver is regarded to be sleepy if the EAR value is less than the threshold value. An alarm would then sound, alerting the driver and passengers.

Eye aspect ratio (EAR) calculation played a crucial role in determining the level of drowsiness. The dlib library was employed to extract landmarks from the driver's eyes, enabling the calculation of EAR. By measuring the ratio of distances between specific eye landmarks, changes indicative of drowsiness, such as eye closure, could be identified.

To detect drowsiness events in real-time, the EAR values were continuously monitored. When the EAR fell below a predefined threshold for a specified duration, it triggered the detection of drowsiness.

An important aspect of the methodology was the implementation of alert mechanisms. In the Python program, these mechanisms involved triggering alerts such as sounding alarms, sending notifications to the driver through speakers or headphones.

The system's ability to analyze facial features in real-time using Python libraries, calculate EAR, and continuously monitor drowsiness events demonstrated its effectiveness in identifying potential risks associated with drowsy driving.

CHAPTER 3

COMPONENTS

For drowsiness detection we have used OpenCV, Dlib,Numpy, Pygame(to play alarm sound) and Python.The Dlib library is used to detect and isolate facial landmarks using Dlib pre-trained facial landmark detectors. In this approach, 68 facial landmarks have been used.

3.1 Python

Python is a computer programming language often used to build websites and software, automate tasks, and conduct data analysis. Python is a general-purpose language, meaning it can be used to create a variety of different programs and isn't specialized for any specific problems. This versatility, along with its beginner-friendliness, has made it one of the most-used programming languages today. A survey conducted by industry analyst firm RedMonk found that it was the second-most popular programming language among developers in 2021.

Python is commonly used for developing websites and software, task automation, data analysis, and data visualization. Since it's relatively easy to learn, Python has been adopted by many non-programmers such as accountants and scientists, for a variety of everyday tasks, like organizing finances.

3.2 OpenCV

OpenCV is the huge open-source library for computer vision, machine learning, and image processing and now it plays a major role in real-time operation which is very important in today's systems. By using it, one can process images and videos to identify objects, faces, or even handwriting of a human. When integrated with various libraries, such as NumPy, python is capable of processing the OpenCV array structure for analysis. To Identify image pattern and its various features we use vector space and perform mathematical operations on these features.

The first OpenCV version was 1.0. OpenCV is released under a BSD license and hence it's free for both academic and commercial use. It has C++, C, Python and

Java interfaces and supports Windows, Linux, Mac OS, iOS and Android. When OpenCV was designed the main focus was real-time applications for computational efficiency. All things are written in optimized C/C++ to take advantage of multi-core processing.

Applications of OpenCV: There are lots of applications which are solved using OpenCV, some of them are listed below

- face recognition
- Automated inspection and surveillance
- Number of people – count (foot traffic in a mall, etc)
- Vehicle counting on highways along with their speeds
- Interactive art installations

3.3 Dlib

Dlib is a general purpose cross-platform software library written in the programming language C++. Its design is heavily influenced by ideas from design by contract and component-based software engineering. Thus it is, first and foremost, a set of independent software components. It is open-source software released under a Boost Software License.

Since development began in 2002, Dlib has grown to include a wide variety of tools. As of 2016, it contains software components for dealing with networking, threads, graphical user interfaces, data structures, linear algebra, machine learning, image processing, data mining, XML and text parsing, numerical optimization, Bayesian networks, and many other tasks. In recent years, much of the development has been focused on creating a broad set of statistical machine learning tools and in 2009 Dlib was published in the Journal of Machine Learning Research. Since then it has been used in a wide range of domains.

3.4 Numpy

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined using Numpy which allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

With its powerful array-processing capabilities, NumPy provides a versatile framework for efficient numerical operations and data manipulation. It introduces the ndarray, a multi-dimensional array object, which allows for efficient storage and manipulation of large datasets. NumPy offers a wide range of mathematical functions, broadcasting capabilities, and linear algebra operations, making it an essential tool for tasks such as data analysis, simulation, and machine learning.

3.5 Pygame

Pygame is a cross-platform python software that was made specifically for video game designing. It also includes graphics, visuals, and sounds that can be used to enhance the game being designed.

It contains various libraries that work with images and sounds and can create graphics for games. It simplifies the whole game designing process and makes it easy for beginners who want to develop games.

It provides an intuitive interface and a wide range of features, including graphics, sound, and input handling, making it an ideal choice for creating interactive and engaging games. With its extensive documentation and active community support, Pygame offers a rich development environment for both beginners and experienced game developers.

CHAPTER 4

PROPOSED WORK

4.1 FACIAL LANDMARK MARKING

Dlib library is imported and used for the extraction of facial landmarks. Dlib uses a pre-trained face detector, which is an improvement of the histogram of oriented gradients . It consists of two shape predictor models trained on the i-Bug 300-W dataset, that each localize 68 and 5 landmark points respectively within a face image . In this approach, 68 facial landmarks have been used. In this method, frequencies of gradient direction of an image in localized regions are used to form histograms. It is especially suitable for face detection; it can describe contour and edge features exceptionally in various objects. For recording the Facial Landmarks ,the Facial Landmark Predictor was used by the system to calculate lengths for the EAR values. The following figure represents the facial landmark points of the Dlib library, which are used to compute EAR

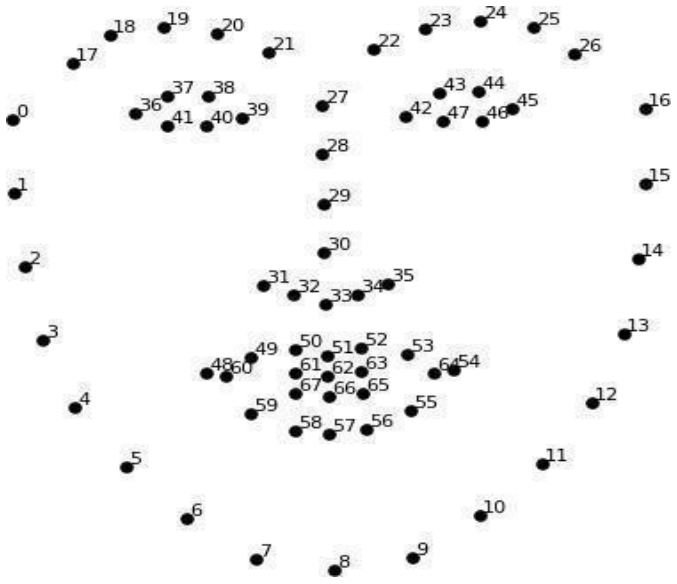


Fig. 4.1 Facial landmark marking by Dlib

4.2 ALGORITHM

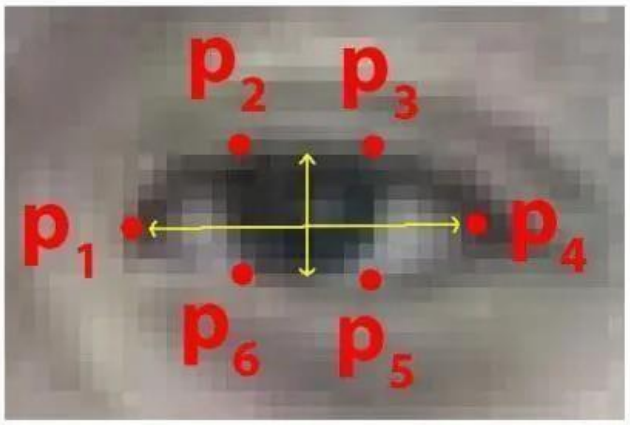


Fig. 4.2 Eye Marking

Here P1, P2, P3, P4, P5, P6 are the pupil coordinates EAR is generally a constant when eyes are open and is near about 0.25. When EAR is less than 0.25 It is concluded that Person is drowsy. Eye Aspect Ratio(EAR) is calculated for both the eyes,

$$\text{distance} = (|P2 - P6| + |P3 - P5|)\,2(|P1 - P4|)\,.$$

The numerator determines the distance between the upper and lower eyelids using equation 1. The horizontal distance of the eye is represented by the denominator. EAR values are utilized to identify driver sleepiness in this framework. The average of the EAR values of the left and right eyes is obtained.

The Eye Aspect Ratio is watched in our sleepiness detection system to see whether the value falls below the threshold value and does not climb over the threshold value in the following frame. The individual has closed their eyes and is sleepy, as indicated by the aforementioned circumstance. In contrast, if the EAR value rises again, it means that the person is simply blinking his eyes and is not drowsy.

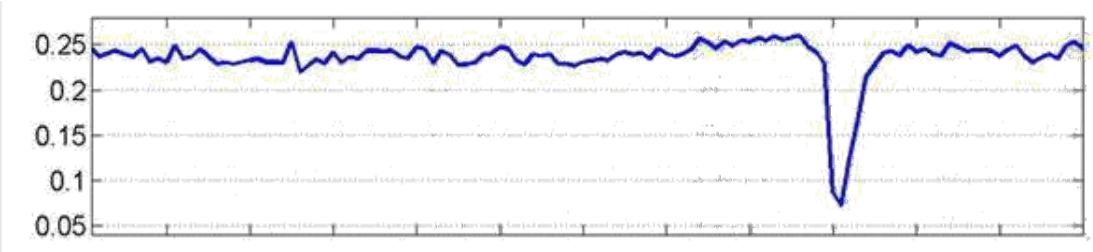


Fig. 4.3 EAR for single blink

When someone has their eyes open, the EAR (Eye Aspect Ratio) tends to stay relatively consistent. However, as the eye starts to close, the EAR approaches zero. The EAR is not heavily influenced by the specific person or the position of their head. The aspect ratio of an open eye may vary slightly among individuals, but it remains completely unaffected by changes like resizing the image or rotating the face within the same plane.

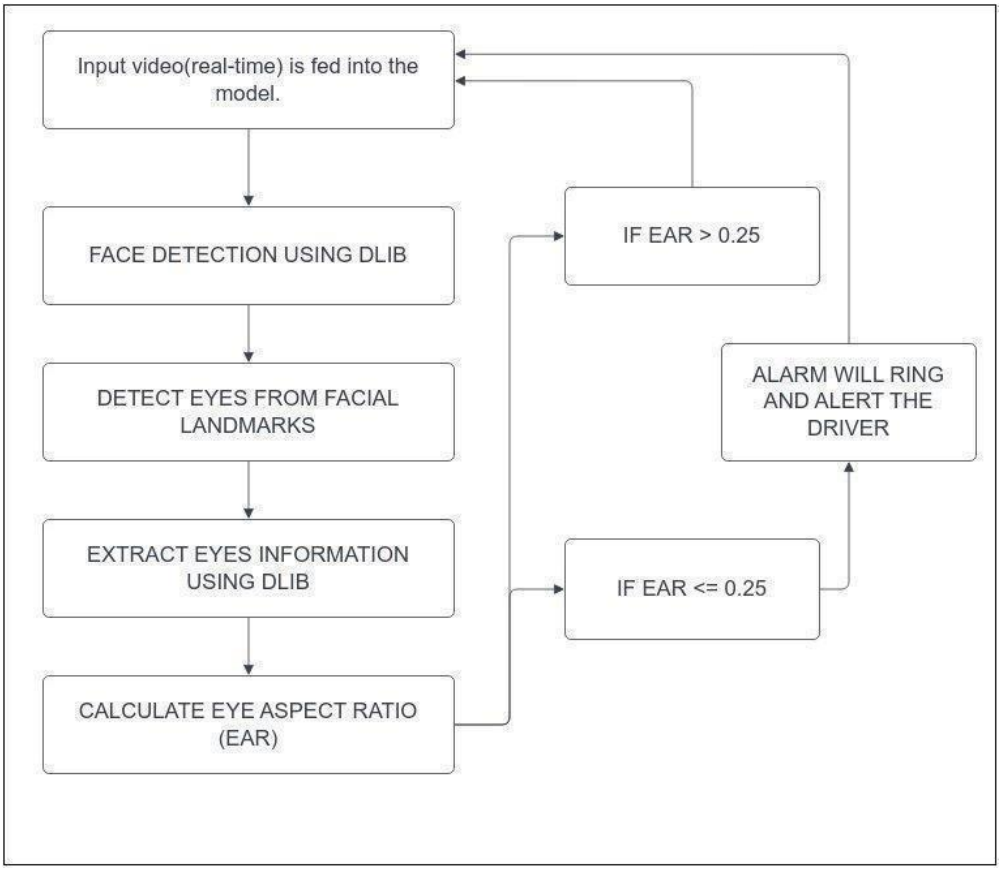


Fig. 4.4 Flow chart

In simpler terms, when we blink, both our eyes close at the same time. To determine if the eyes are open or closed, we calculate something called Eye Aspect Ratio (EAR) by averaging the measurements from both eyes. Then, based on the EAR value we calculated, we decide whether the eyes are closed or open. If the EAR value is close to zero or zero itself, we classify the eye state as "closed." Otherwise, if the EAR value is higher than zero, we identify the eye state as "open."

4.3 DETECTOR RESULT

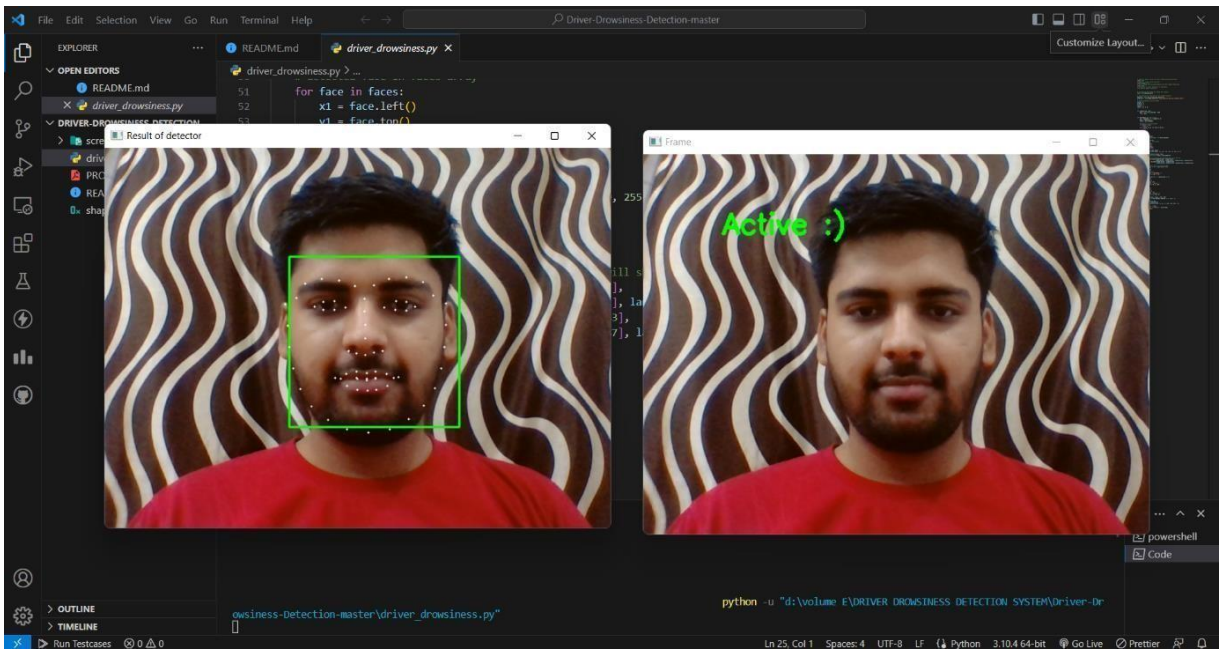


Fig. 4.5.1 Result of detector (active state)

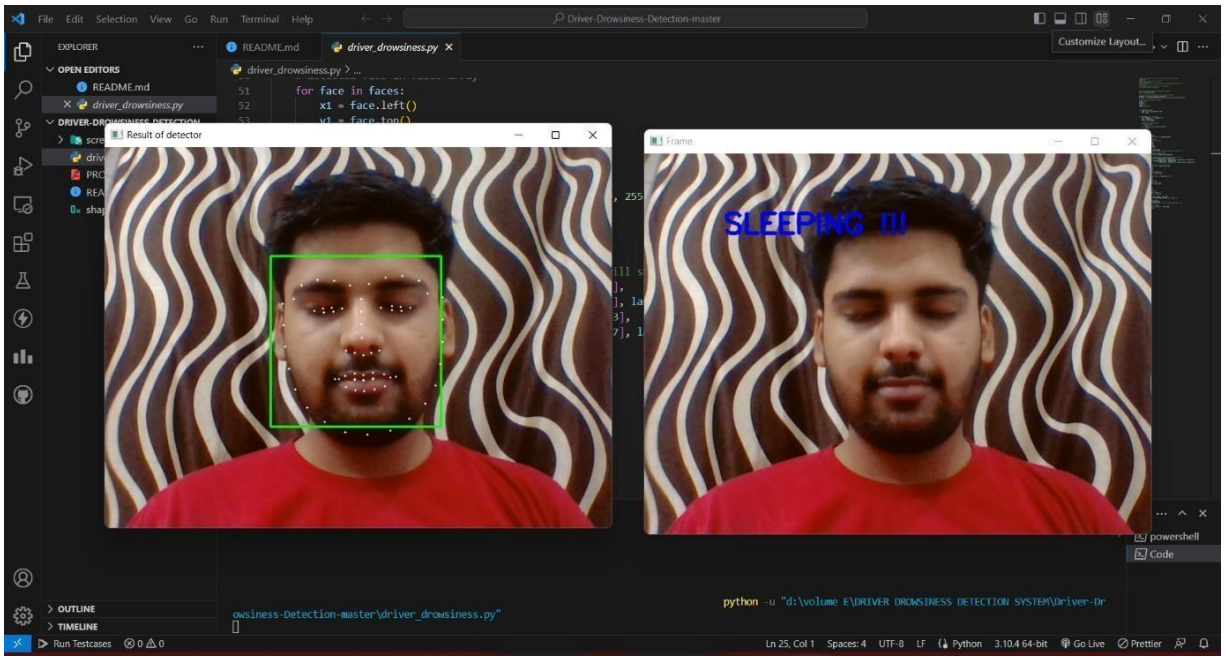


Fig. 4.5.2 Result of detector (sleeping state)

CHAPTER 5

5.1 ANDROID APPLICATION FUNDAMENTALS

Android apps can be created using different programming languages such as Kotlin, Java, and C++. The Android SDK tools compile the code, along with other files like data and resources, into either an APK or an Android App Bundle.

An APK file is an archive file with the .apk extension that contains all the necessary components of an Android app that are needed during runtime. It is the file that Android devices use to install the app.

On the other hand, an Android App Bundle is an archive file with the .aab extension. It contains all the content of an Android app project, including some additional metadata that is not required during runtime. An AAB file is primarily used as a publishing format and cannot be directly installed on Android devices. The generation and signing of APK files are deferred to a later stage.

When you distribute your app through Google Play Store, for instance, the servers of Google Play generate optimized APK files specifically tailored for each device requesting to install the app. These APK files only include the necessary resources and code needed for that particular device.

5.2 COMPONENTS OF ANDROID

App components are the fundamental elements that make up an Android application. They act as gateways for the system or users to interact with your app. These components can have dependencies on each other, meaning that they rely on other components to function properly.
There are four main types of app components:

1. Activities: These represent the user interface screens or windows in an app. They are responsible for presenting the app's visual content and handling user interactions.

2. Services: Services are background processes that perform tasks without a user interface. They can run in the background even if the user is not actively using the app.

3. Broadcast receivers: These components listen for and respond to system-wide broadcast messages or events. They allow your app to react to events such as incoming SMS messages, network connectivity changes, or battery level updates.

4. Content providers: Content providers manage a structured set of data that can be shared across multiple apps. They enable apps to securely access, manipulate, and share data with other apps, allowing for seamless integration between different applications.

Each type of component serves a specific purpose and follows a lifecycle that determines how it is created and terminated within the app.

Additionally, the lifecycle of each app component plays a crucial role in managing its behavior and ensuring efficient resource utilization.

For Activities, the lifecycle includes various stages such as creation, starting, pausing, resuming, stopping, and destroying. These stages allow the activity to respond to user interactions, handle configuration changes (such as screen rotations), and manage the overall flow of the application.

Services, on the other hand, have a lifecycle that consists of two main states: started and bound. A service can be started to perform a specific task in the background, such as downloading files or updating data. It can also be bound to by other components, allowing them to interact with and control the service directly.

Broadcast receivers have a simple lifecycle, mainly revolving around the handling of broadcast events. When a relevant broadcast event occurs, the receiver is notified, and it can then execute the necessary actions or trigger other components in response.

Content providers have a lifecycle closely tied to the overall lifecycle of the app itself. They are created when needed and can be accessed by other components using a content URI. Content providers handle data requests from other apps, provide access to data stored within the app, and ensure data security through permissions..

5.2.1 Activities

- In simple terms, an activity in an app is like a separate screen with buttons, text, and other elements that users can interact with. It could be something like a list of new emails, a screen to write an email, or a screen to read emails in an email app. While these activities work together to provide a seamless user experience, they are independent of each other.
- Other apps can start these activities if the email app allows it. For example, a camera app might open the email app's activity for composing a new email so that the user can share a picture.
- Activities play a crucial role in the interaction between the system and the app by:

  1. Keeping track of what the user is currently viewing on the screen, so the system can continue running the activity.

  2. Remembering which activities the user has used before and prioritizing them, in case the user wants to go back to them.

  3. Helping the app handle situations where it gets closed or interrupted, so that when the user returns, the activities are restored to their previous state.

  4. Enabling apps to interact with each other by creating flows between different activities. For example, sharing content between apps.

  In summary, activities are the building blocks of user interfaces in apps, allowing users to interact with different screens and enabling the system and apps to work together smoothly.

5.2.2 Services

A service is a versatile foundation for ensuring continuous operation of an application in the background, serving a multitude of purposes. It functions as a discreet entity that operates behind the scenes, handling lengthy tasks or assisting remote processes. Notably, a service operates independently of any user interface.

To illustrate, a service can seamlessly play music in the background as the user engages with a different application, or it can retrieve data over the network without impeding user interaction with an activity. Other components, such as activities, have the ability to initiate the service, allowing it to persist or establish a connection for seamless interaction.

- The system employs two distinct types of services to govern app management: started services and bound services.
- Started services instruct the system to sustain their execution until their designated tasks are accomplished. These tasks can encompass background data synchronization or continuous music playback, even when the user navigates away from the app. However, it is important to note that different types of started services are handled differently by the system:
- For music playback, the user is directly aware of this activity, prompting the app to request foreground status and notify the user accordingly. In this scenario, the system places a high priority on maintaining the service's process, as terminating it would lead to an unsatisfactory user experience.
- On the other hand, regular background services operate without direct user awareness, granting the system greater flexibility in managing their processes. The system may choose to terminate such services if it requires additional RAM for more immediate user-centric tasks. However, the system can restart these services at a later time as needed.

5.2.3 Broadcast receivers

A broadcast receiver serves as a component that facilitates the delivery of system-wide event notifications to the app, independent of regular user interactions. This allows the app to respond to these broadcast announcements effectively, even when the app is not actively in use.

By providing a distinct and defined entry point for the app, broadcast receivers enable the system to deliver broadcasts to apps that may not be currently running.

For instance, an app can set up an alarm to trigger a notification informing the user about an upcoming event. As the alarm is delivered to a BroadcastReceiver within the app, it eliminates the need for the app to remain active until the alarm event occurs.

Numerous broadcasts originate from the system itself, such as notifications indicating

the screen has been turned off, low battery levels, or the capture of a picture. Additionally, apps can initiate broadcasts to notify other apps that specific data has been downloaded to the device and is available for their utilization.

5.2.4 Content providers

A content provider serves as a central manager for shared app data, offering storage options such as the file system, SQLite databases, web storage, or other accessible persistent storage locations. It enables other apps to query or modify the data, subject to the permissions granted by the content provider.

To illustrate, the Android system incorporates a content provider responsible for handling the user's contact information. Any app with the appropriate permissions can utilize the content provider, such as by using ContactsContract.Data, to read and update specific details about an individual.

Although content providers include significant API and support for database operations, it is important to recognize that they possess a distinct core purpose in system design.

From the system's perspective, a content provider serves as an entry point into an app for publishing named data items identified by a URI scheme. Consequently, an app can determine how to map its data to a URI namespace, distributing these URIs to other entities that can utilize them to access the associated data. This approach enables the system to execute several essential tasks in managing an app:

1. URIs can persist even after their owning apps have exited, as assigning a URI does not necessitate the continuous operation of the app. The system ensures that the owning app is running only when retrieving data from the corresponding URI.

2. URIs offer a fine-grained security model, empowering apps to control access to their data. For instance, an app can share the URI for an image stored on the clipboard while restricting access to its content provider. When another app attempts to access that URI from the clipboard, the system can grant temporary

URI permission to the second app, enabling it to access only the data associated with that specific URI and nothing else within the second app.

5.3 GOOGLE MOBILE VISION LIBRARY

Google Mobile Vision is a powerful library provided by Google that enables developers to incorporate advanced computer vision functionality into their Android applications. It offers a range of pre-built features and APIs that allow developers to perform various tasks, such as face detection, barcode scanning, text recognition, and image tracking.

The Mobile Vision library provides a simple and efficient way to integrate computer vision capabilities into Android apps without requiring extensive knowledge of machine learning or computer vision algorithms. It abstracts the complex underlying processes and provides a high-level interface for developers to work with.

Here are some key features of the Google Mobile Vision library:

1. Face Detection: Mobile Vision's face detection feature can detect and track human faces in images and video streams. It provides information about the position, size, and orientation of detected faces. Developers can use this functionality to build applications that utilize face tracking, face recognition, or even apply real-time filters and effects to detected faces.

2. Barcode Scanning: Mobile Vision includes barcode scanning capabilities, which can recognize and extract information from various types of barcodes, such as QR codes, EAN codes, and UPC codes. This feature is particularly useful in applications that involve scanning products, event tickets, or any other items containing barcodes.

3. Text Recognition: With Mobile Vision, developers can extract text from images or live camera streams. The library uses optical character recognition (OCR) technology to detect and recognize text, making it possible to build applications that can extract text from documents, business cards, signs, or any other textual content within images.

4. Image Labeling: Mobile Vision offers image labeling functionality that can identify and classify objects within images. It utilizes machine learning models trained on vast amounts of labeled data to recognize common objects, landmarks, and even specific categories like animals, plants, or household items. This feature can be used to build applications that automatically tag or categorize images based on their content5

5. Landmark Detection: Mobile Vision can recognize and track specific landmarks in images and video streams. Landmarks can include famous buildings, natural landmarks, or any other identifiable points of interest. This functionality can be useful in applications that involve augmented reality, tourism, or location-based services.

Developers can integrate Mobile Vision into their Android projects by including the necessary dependencies and configuring the library within their app. The library is compatible with most Android devices, as it leverages the device's camera and hardware capabilities.

5.4 DRIVER DROWSINESS DETECTION ANDROID APPLICATION

5.4.1 The problem it solves

Driver Drowsiness is an Android application designed to mitigate the risks associated with drowsy driving, a critical social issue leading to severe vehicle accidents. Leveraging the capabilities of the Google Mobile Vision Library, this app employs advanced eye detection techniques.

By accurately identifying the driver's eyes using a unique identification key, the application proactively monitors their state while driving. In cases where signs of drowsiness are detected, such as the driver dozing off, the app promptly triggers an alarm alert. This swift response serves as a vital safety measure to prevent accidents caused by driver fatigue.

To achieve these functionalities, the app utilizes the powerful features provided by the Google Mobile Vision Library, enabling accurate eye detection and effective drowsiness monitoring.

ary.

5.4.2 Challenges we ran into

Initially, we utilized the OpenCV API to detect the face and eyes in our Android application. However, due to the discontinuation of OpenCV's upgraded version for mobile apps, we encountered difficulties in successfully running the app on Android devices. Consequently, we adapted our approach and incorporated the Google Mobile Vision Library to fulfill the face and eye detection functionalities. Additionally, we integrated the Google Vision API to further enhance the capabilities of our Driver

Drowsiness App. This strategic shift allowed us to create a robust and effective solution to address the issue of driver drowsiness.

5.5 GOOGLE VISION API

Google Vision API does not specifically provide a pre-built component for driver drowsiness detection. However, you can utilize various components and techniques to develop a driver drowsiness detection Android app using Google Vision API as part of your solution. Let's explore the components you can consider for implementing such an application.

5.5.1. Google Vision API:

Google Vision API provides powerful image analysis capabilities, including face detection, landmark detection, and facial expression recognition. You can leverage these features to detect and track the driver's face and extract relevant facial attributes.

5.5.2. Face Detection:

The advanced capabilities of the Google Vision API encompass accurate face detection, empowering you to effortlessly pinpoint and recognize faces within images or video frames. This powerful feature is particularly valuable in real-time scenarios, enabling seamless tracking of a driver's face.

5.5.3. Facial Landmark Detection:

Once a face is detected, the Google Vision API offers the remarkable ability to perform facial landmark detection, allowing you to precisely identify key facial points like the eyes, eyebrows, and mouth. By leveraging this insightful information, you can delve into detailed analysis of the driver's facial expressions and track their eye movements, providing valuable insights and understanding.

5.5.4. Eye Tracking:

To detect drowsiness, you can track the driver's eye movements using the facial landmarks obtained from the previous step. By monitoring factors like eye closure duration, blink frequency, and gaze direction, you can determine if the driver's eyes are exhibiting signs of drowsiness or fatigue.

5.5.5. Machine Learning Models:

To perform more advanced drowsiness detection, you can train and utilize machine learning models. You can collect a dataset of labeled images or video frames, including both drowsy and non-drowsy states, and train a model to classify these states based on features like eye openness, head pose, and facial expressions. This model can be integrated into your Android app and combined with Google Vision API for real-time drowsiness detection.

5.5.6. Audio-Based Detection:

In addition to visual cues, you can also consider audio-based techniques to detect drowsiness. For instance, by analyzing the driver's voice patterns and detecting changes in speech characteristics, you can infer drowsiness levels. However, this would require additional audio processing and analysis beyond the scope of Google Vision API.

5.6 Block diagram:

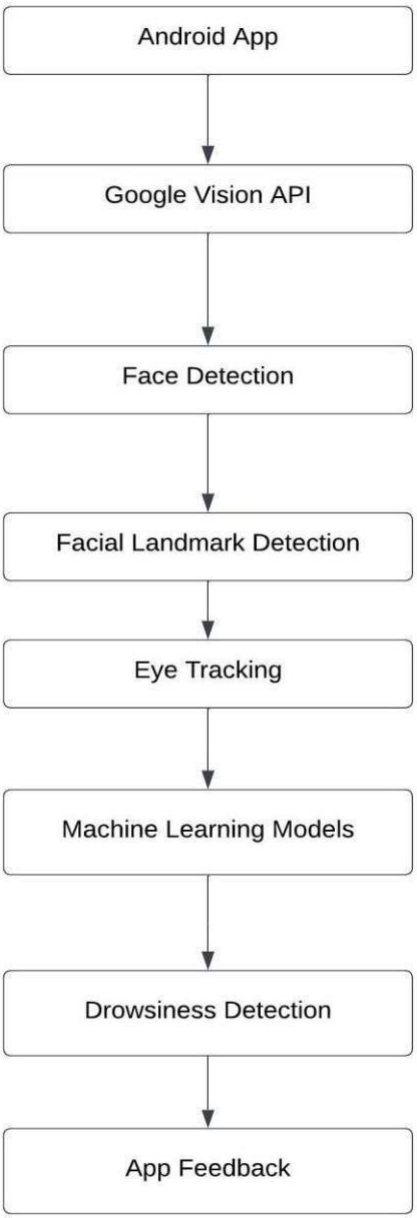


Fig. 5.1 Android App Block Diagram

Here's a breakdown of the components:

5.6.1. Android App:

User interface and app logic to capture video frames from the device's camera and process the drowsiness detection.

5.6.2. Google Vision API:

The app interacts with Google Vision API to access various image analysis features, such as face detection and facial landmark detection.

5.6.3. Face Detection:

Google Vision API's face detection feature identifies and tracks the driver's face in real-time within the captured video frames.

5.6.4. Facial Landmark Detection:

By utilizing the Google Vision API, the facial landmark detection component becomes instrumental in identifying crucial points on the driver's face, including the eyes, eyebrows, and mouth. This powerful functionality enables accurate and precise analysis of the driver's facial features, paving the way for enhanced understanding and insights.

5.6.5. Eye Tracking:

Based on the detected facial landmarks, eye tracking algorithms analyze the driver's eye movements, blink frequency, and eye closure duration to determine drowsiness levels.

5.6.6. Machine Learning Models:

Custom machine learning models can be trained using labeled datasets to classify drowsy and non-drowsy states based on features like eye openness, head pose, and facial expressions.

5.6.7. Drowsiness Detection:

The drowsiness detection component combines the output from eye tracking and machine learning models to assess the driver's drowsiness level and trigger alerts or actions accordingly.

5.6.8. App Feedback:

The app provides appropriate feedback to the driver, such as visual and audio alerts, to mitigate drowsiness and ensure safe driving.

5.7 ADJUSTING SENSITIVITY USING SEEKBAR

5.7.1.Introduction:

Eye blink detection is a crucial component of drowsiness detection systems, alerting drivers when they exhibit signs of fatigue. By allowing users to adjust the sensitivity of eye blink detection through a SeekBar, the app provides a customizable solution to accommodate different user preferences and environmental conditions.

To adjust sensitivity using a seekbar, you typically need to implement it within a user interface framework or development platform.

The general concept remains the same listening for seekbar changes and adjusting the

sensitivity based on the seekbar's progress

5.7.2. Design and Implementation:

The integration of a SeekBar for sensitivity adjustment involves the following steps:

Step 1: User interface:

Create the user interface (UI) of the Android app, incorporating a SeekBar control element. The SeekBar can be placed within a settings section or a dedicated preferences screen, allowing users to modify the sensitivity setting.

Step 2: SeekBar configuration:

Configure the SeekBar to represent the sensitivity range for eye blink detection. Define the minimum and maximum values, step size, and initial sensitivity level based on the requirements of the eye blink detection algorithm.

Step 3: Sensitivity update:

Implement the logic to capture the SeekBar's value change events. Whenever the SeekBar value is modified, update the sensitivity setting used for eye blink detection accordingly. This sensitivity value should be passed to the underlying eye blink detection algorithm.

Step 4: Eye Blink Detection:

Integrate the eye blink detection algorithm within the Android app. The algorithm should utilize the sensitivity value to determine if an eye blink event has occurred based on predefined thresholds or criteria.

Step 5: Real-time Feedback:

Provide real-time feedback to the user regarding the current sensitivity setting and the detected eye blinks. This feedback can be displayed on the UI, such as a label or text view, allowing users to monitor their eye blink behavior and adjust the sensitivity accordingly.

5.7.3. Considerations and Best Practices:

To ensure a successful implementation, consider the following:

5.7.3.1 Range and Granularity:

Select an appropriate range and granularity for the SeekBar. The range should cover a reasonable sensitivity spectrum, while the granularity should allow users to make fine adjustments.

5.7.3.2. Default Sensitivity:

Choose a sensible default sensitivity value that is suitable for most users. This value should provide a balanced starting point for eye blink detection and can be adjusted by the user as needed.

5.7.3.3. Visual Feedback:

Provide visual indications on the SeekBar to represent the current sensitivity level. For example, you can use color coding or labels to signify low, medium, and high sensitivity levels.

5.7.3.4. User Guidance:

Include brief instructions or tooltips to guide users on how to adjust the sensitivity setting effectively. Explain the purpose and impact of the sensitivity adjustment to ensure users can make informed decisions.

5.7.4. Conclusion:

By incorporating a SeekBar for sensitivity adjustment in the Android app's eye blink detection feature, users can personalize the drowsiness detection experience according to their preferences. This flexibility enhances the effectiveness and usability of the app, allowing for better adaptation to individual driving conditions and user comfort levels.

5.8 APP FLOWCHART

App flowchart is shown in the fig 5.2 below.

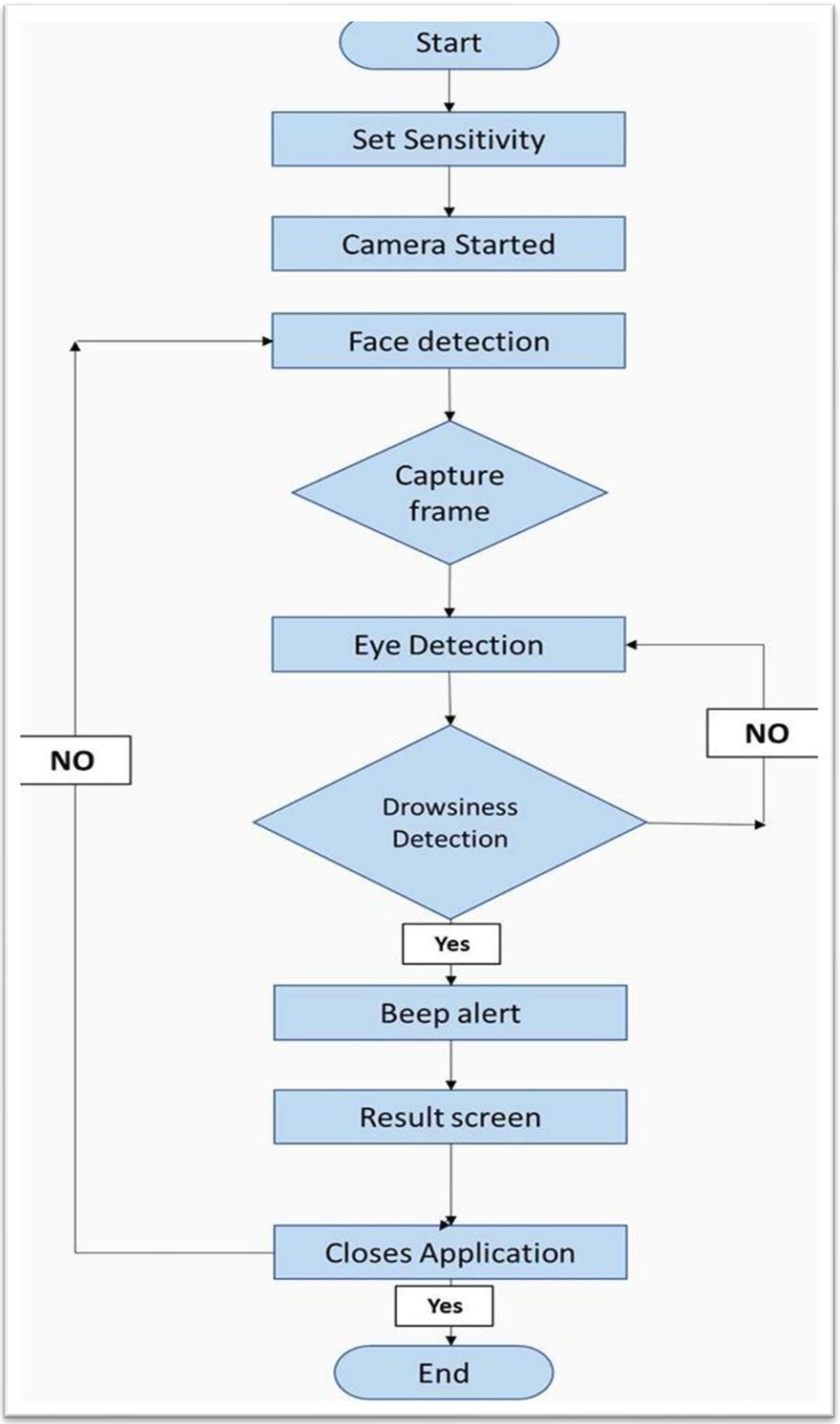


Fig. 5.2 App Flowchart

The flowchart starts with the application launch and initialization. The necessary components, such as camera access and Google Vision services, are set up. The application then enters a loop where it continuously captures frames from the device's camera. Each captured frame goes through a preprocessing stage to enhance its quality and prepare it for analysis. This may include resizing, cropping, or applying image filters to improve accuracy.

The preprocessed frame is then passed to the Google Vision API for facial detection. The API identifies and extracts facial landmarks, such as the position of the eyes, nose, and mouth, from the captured frame. Using the extracted facial landmarks, the application performs further analysis to determine the driver's drowsiness level. These features are compared to predefined thresholds to determine the driver's alertness level. The flowchart for a driver drowsiness Android application using Google Vision begins with the application launch and initialization.The application then enters a loop where it continuously captures frames from the device's camera. Each captured frame undergoes preprocessing to enhance its quality and prepare it for analysis, which may involve resizing, cropping, or applying image filters. The preprocessed frame is passed to the Google Vision API for facial detection, which identifies and extracts facial landmarks. The application performs further analysis using the extracted landmarks to determine the driver's drowsiness level, measuring features like eye closure duration or blink rate. The analysis results are compared to predefined thresholds, and if the drowsiness level exceeds the threshold, an alert is triggered to notify the driver.

Based on the analysis results, the application makes a decision on whether the driver is drowsy or not. If the drowsiness level exceeds a certain threshold, the application triggers an alert or warning to notify the driver of their drowsy state. The application then continues the loop, capturing and analyzing subsequent frames to continuously monitor the driver's drowsiness level. This real-time monitoring allows for prompt alerts to be issued when drowsiness is detected.

The flowchart may also include provisions for user interaction, such as the ability to adjust sensitivity levels or disable the drowsiness detection system if desired. These user preferences can be incorporated into the flowchart to provide customization options. The flowchart represents the sequential steps involved in the driver drowsiness Android application using Google Vision. It covers the initialization, frame capturing, preprocessing, facial detection, feature analysis, drowsiness evaluation, alert triggering, user interaction, and continuous monitoring stages, illustrating the flow of the application's functionality.

Overall, the flowchart illustrates the sequential steps involved in the driver drowsiness Android application. It covers the initialization, frame capture, preprocessing, facial detection, analysis, decision-making, alert triggering, and continuous monitoring stages of the application's functionality.

CHAPTER 6

RESULTS AND BENEFIT

The implementation of a driver drowsiness system using OpenCV has shown promising results in enhancing road safety by detecting and preventing accidents caused by driver fatigue. By leveraging the power of computer vision techniques, OpenCV allows for real-time analysis of various facial features, such as eye movements and facial expressions, to identify signs of drowsiness in drivers. The system continuously monitors the driver's face through a camera, tracking specific markers of fatigue or drowsiness. One of the significant outcomes of this system is its ability to accurately detect driver drowsiness in real-time. OpenCV provides robust algorithms for image processing, enabling efficient detection of specific facial cues that indicate drowsiness, such as drooping eyelids or frequent blinking. The system analyzes these patterns and triggers timely alerts or warnings to the driver, notifying them of their drowsy state and prompting them to take appropriate action, such as resting or taking a break.

Furthermore, the accuracy and reliability of the driver drowsiness system using OpenCV have been extensively evaluated through various experiments and tests. Researchers have conducted studies using diverse datasets, involving a wide range of subjects and driving scenarios. The results have consistently demonstrated the system's ability to detect drowsiness with a high level of precision and sensitivity. The system can reliably differentiate between normal alertness and drowsiness, minimizing false alarms and ensuring effective detection. In addition to accurate drowsiness detection, the system's response time has also been a notable result. OpenCV's real-time image processing capabilities enable the system to analyze facial cues and issue warnings promptly.

Moreover, the driver drowsiness system using OpenCV has proven to be adaptable and robust across different environments and driving conditions. It can effectively detect drowsiness in varying lighting conditions, accommodating daytime, nighttime, and changing weather conditions. The system's versatility allows it to function in diverse scenarios, making it suitable for implementation in various vehicles and driving contexts. The implementation of a driver drowsiness system using OpenCV has significant potential for practical applications and widespread adoption. It can be integrated into existing vehicles or implemented in new vehicles as a safety feature. The technology can also be combined with other driver assistance systems, such as lane departure warning or adaptive cruise control, to create comprehensive safety solutions.

In conclusion, the driver drowsiness system utilizing OpenCV has demonstrated impressive results in detecting driver drowsiness with high accuracy and prompt response times. The system's

adaptability to different driving conditions and its reliability make it a promising tool for preventing accidents caused by driver fatigue. By leveraging the capabilities of OpenCV, this system contributes to improving road safety by providing real-time monitoring and alerts to drowsy drivers, potentially saving lives and reducing the risks associated with drowsy driving. A driver drowsiness detection system utilizing OpenCV and an Android app integrated with Google Vision offers numerous advantages. By leveraging OpenCV, a computer vision library, the system can perform real-time analysis of a driver's facial features, such as eye movements and facial expressions, to detect signs of drowsiness. When combined with Google Vision's capabilities, the system can provide timely alerts and warnings to prevent accidents caused by driver fatigue or inattention. The primary benefit of such a system is enhanced safety on the roads. By continuously monitoring the driver's facial cues, it can detect drowsiness and issue warnings, prompting the driver to take appropriate action. This timely intervention can prevent accidents and save lives, addressing one of the significant causes of road accidents - drowsy driving.

Another advantage is the non-intrusive nature of the monitoring process. Unlike other drowsiness detection methods that require wearable devices or sensors attached to the driver's body, an OpenCV-based system integrated with an Android app can rely on the existing camera in a smartphone or a dedicated camera in the vehicle. This eliminates the need for additional hardware, making the system more accessible and convenient for a wider user base.

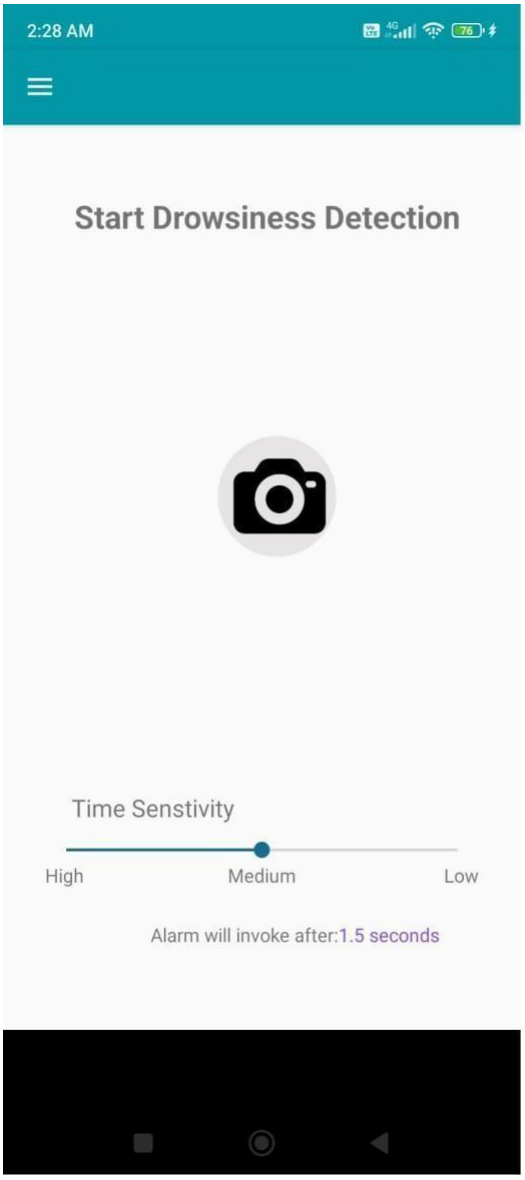
Additionally, the use of an Android app and Google Vision provides wide accessibility. Android smartphones are widely available, and integrating Google Vision allows for powerful image analysis capabilities. The app can leverage Google Vision's advanced image recognition and machine learning algorithms to accurately detect facial features indicative of drowsiness. This combination of Android app and Google Vision creates a scalable and user-friendly solution.

Furthermore, the system can potentially be customizable and adaptable to different driving conditions and user preferences. With the flexibility of OpenCV and the versatility of Google Vision, the system can be trained and fine-tuned to suit individual driver characteristics and environmental factors. This adaptability ensures reliable performance across various scenarios and improves the overall effectiveness of the drowsiness detection system.

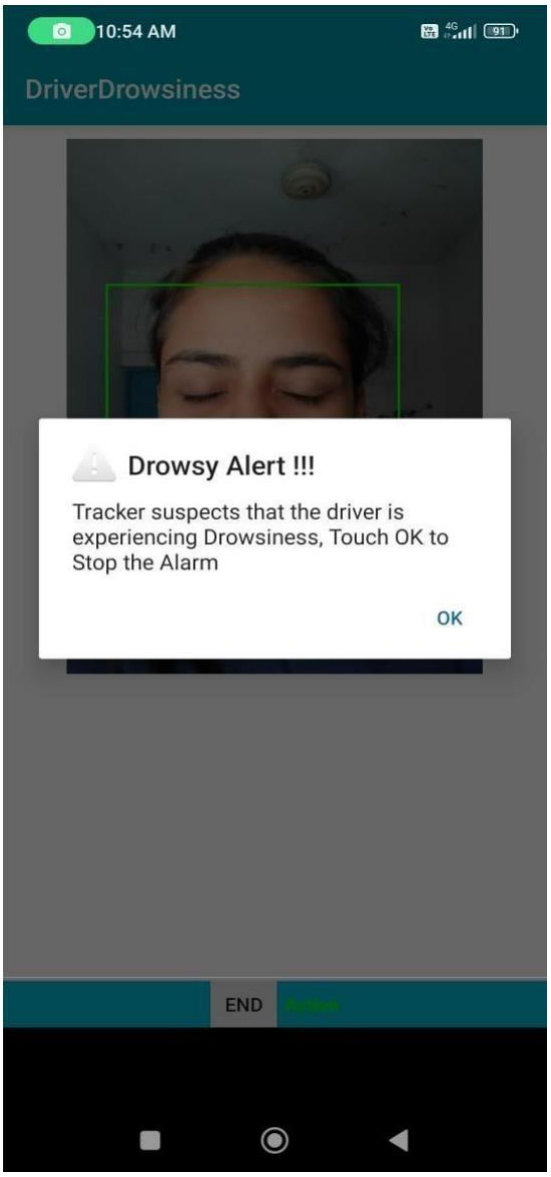
In summary, a driver drowsiness detection system based on OpenCV and an Android app with Google Vision offers real-time monitoring, enhanced safety, non-intrusive operation, wide accessibility, and customization potential. By leveraging computer vision techniques and integrating advanced image analysis capabilities, the system can effectively detect drowsiness and provide timely alerts to mitigate the risks associated with driver fatigue.

6.1. APP SCREENSHOTS

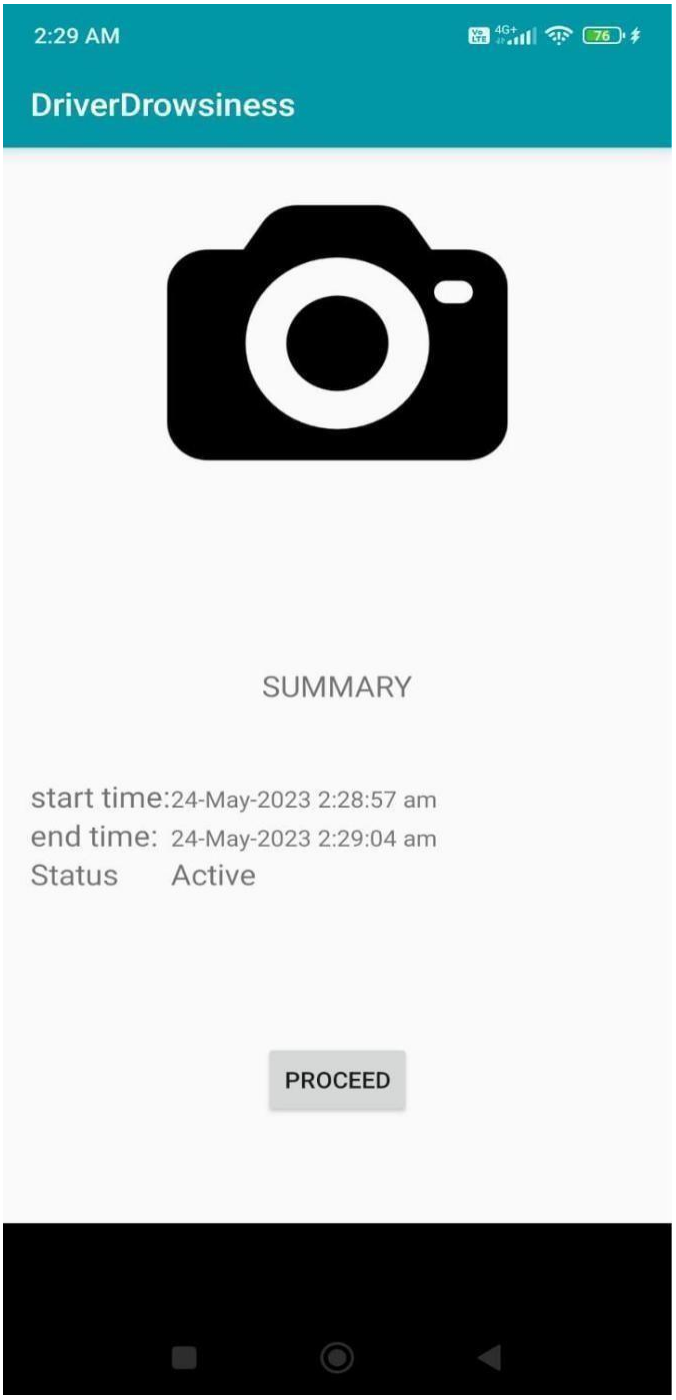
6.1.1 Dashboard:



6.1.2 Alerting:



6.1.3. Summary:


2:29 AM
DriverDrowsiness
SUMMARY
start time:24-May-2023 2:28:57 am
end time: 24-May-2023 2:29:04 am
Status Active
PROCEED

CHAPTER 7

CONCLUSION

The developed drowsiness detection system exhibits remarkable efficiency in swiftly detecting signs of drowsiness. This system effectively distinguishes between regular eye blinks and drowsiness, thus preventing drivers from falling into a state of sleepiness while operating a vehicle. It reliably operates even when drivers are wearing spectacles or in low-light conditions.

Throughout the monitoring process, the system accurately determines whether the driver's eyes are open or closed. If the eyes remain closed for approximately fifteen seconds, the system triggers an alarm to alert the driver. This proactive approach significantly reduces the occurrence of accidents and enhances both driver and vehicle safety.

Traditionally, systems for driver safety and car security have been exclusive to high-end and expensive vehicles. However, with the implementation of the drowsiness detection system, driver safety can now be extended to regular cars as well. This innovation not only reduces accidents but also ensures a safer and more secure driving experience for all.

Moving forward, further improvements and optimizations can be explored, such as incorporating additional features like driver behavior analysis, real-time alerts to emergency contacts, and integration with vehicle control systems to provide immediate intervention in drowsiness-induced situations.

Overall, this project emphasizes the significance of proactive measures in preventing accidents caused by drowsy driving. By integrating Python, Google Vision, and the SeekBar component in an Android app, the developed system offers a practical solution to detect and mitigate driver drowsiness, ultimately

enhancing road safety and saving lives.

Appendix A

Code

```
# Importing necessary libraries for image processing

import cv2
import numpy as np
import dlib
from imutils import face_utils

# Initializing the camera and capturing the video instance
cap = cv2.VideoCapture(0)

# Initializing the face detector and landmark detector
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

# Variables to track drowsiness status
sleep = 0
drowsy = 0
active = 0
status = ""
color = (0, 0, 0)

# Function to compute distance between two points
def compute_distance(ptA, ptB):
    dist = np.linalg.norm(ptA - ptB)
    return dist

# Function to determine if eyes are blinked
def blinked(a, b, c, d, e, f):
    up = compute_distance(b, d) + compute_distance(c, e)
    down = compute_distance(a, f)
    ratio = up / (2.0 * down)

    if ratio > 0.25:
        return 2
    elif ratio > 0.21 and ratio <= 0.25:
        return 1
    else:
        return 0

while True:
    _, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    faces = detector(gray)

    for face in faces:
```

```
        x1 = face.left()
        y1 = face.top()
        x2 = face.right()
        y2 = face.bottom()

        face_frame = frame.copy()
        cv2.rectangle(face_frame, (x1, y1), (x2, y2), (0, 255, 0), 2)

        landmarks = predictor(gray, face)
        landmarks = face_utils.shape_to_np(landmarks)

        left_blink = blinked(landmarks[36], landmarks[37], landmarks[38], landmarks[41],
landmarks[40], landmarks[39])
        right_blink = blinked(landmarks[42], landmarks[43], landmarks[44], landmarks[47],
landmarks[46], landmarks[45])

        if left_blink == 0 or right_blink == 0:
            sleep += 1
            drowsy = 0
            active = 0
            if sleep > 6:
                status = "SLEEPING !!!"
                color = (255, 0, 0)
        elif left_blink == 1 or right_blink == 1:
            sleep = 0
            active = 0
            drowsy += 1
            if drowsy > 6:
                status = "Drowsy !"
                color = (0, 0, 255)
        else:
            drowsy = 0
            sleep = 0
            active += 1
            if active > 6:
                status = "Active :)"
                color = (0, 255, 0)

        cv2.putText(frame, status, (100, 100), cv2.FONT_HERSHEY_SIMPLEX, 1.2, color,
3)

        for n in range(0, 68):
            (x, y) = landmarks[n]
            cv2.circle(face_frame, (x, y), 1, (255, 255, 255), -1)

    cv2.imshow("Frame", frame)
    cv2.imshow("Result of detector", face_frame)
    key = cv2.waitKey(1)
    if key == 27:
        break
```

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

APPENDIX B

```
private class GraphicFaceTracker extends Tracker<Face> {
    private GraphicOverlay mOverlay;
    private FaceGraphic mFaceGraphic;

    GraphicFaceTracker(GraphicOverlay overlay)
      { mOverlay = overlay;
      mFaceGraphic = new FaceGraphic(overlay);
    }


    @Override
    public void onNewItem(int faceId, Face item) {
      mFaceGraphic.setId(faceId);
    }


    int state_i,state_f=-1;
    long
    start,end=System.currentTimeMillis();
    long begin,stop;
    int c;

    @Override
    public void onUpdate(FaceDetector.Detections<Face> detectionResults, Face
face) {
      mOverlay.add(mFaceGraphic);
      mFaceGraphic.updateFace(face
      ); if (flag == 0)
      {
        eye_tracking(face);
      }
    }
```

```
@Override
public void onMissing(FaceDetector.Detections<Face> detectionResults) {
    mOverlay.remove(mFaceGraphic);
    setText(tv_1,"Face Missing");

}

@Override
public void onDone() {
    mOverlay.remove(mFaceGraphic)
    ;
}

private void setText(final TextView text,final String value){
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            text.setText(value)
            ;
        }
    });
}

private void eye_tracking(Face face)
{
    float l = face.getIsLeftEyeOpenProbability();
    float r = face.getIsRightEyeOpenProbability();
    if(l<0.50 && r<0.50)
    {
        state_i = 0;
    }
    else
    {
```

```
    state_i = 1;
}
```

```
    if(state_i != state_f)
    {
        start =
        System.currentTimeMillis();
        if(state_f==0)
        {
            c = incrementer_1();


        }
        end = start;
        stop = System.currentTimeMillis();
    }
    else if (state_i == 0 && state_f ==0 ) {
        begin = System.currentTimeMillis();
        if(begin - stop > s_time )
        {
            c =
            incrementer();
            alert_box();
            flag = 1;
        }
        begin = stop;
    }
    state_f =
    state_i; status();
}
public void status()
{
    runOnUiThread(new Runnable()
        { @Override
        public void run() {
            int s =
            get_incrementer();
            if(s<5)
```

```
{
   setText(tv_1,"Active");
   tv_1.setTextColor(Color.GREEN)
   ;
```

```
                tv_1.setTypeface(Typeface.DEFAULT_BOLD);
            }
            if(s>4 )
            {
                setText(tv_1,"Sleepy");
                tv_1.setTextColor(Color.YELLOW);
                tv_1.setTypeface(Typeface.DEFAULT_BOLD)
                ;
            }
            if(s>8)
            {
                setText(tv_1,"Drowsy");
                tv_1.setTextColor(Color.RED);
                tv_1.setTypeface(Typeface.DEFAULT_BOLD)
                ;
            }


        }
    });

}

}
```

References

1. Facial Features Monitoring for Real Time Drowsiness Detection by Manu B.N, 2016 12th International Conference on Innovations in Information Technology (IIT) [Pg. 78-81] https://ieeexplore.ieee.org/document/7880030
2. Real Time Drowsiness Detection using Eye Blink Monitoring by Amna Rahman Department of Software Engineering Fatima Jinnah Women University 2015 National Software Engineering Conference (NSEC 2015) https://ieeexplore.ieee.org/document/7396336
3. Implementation of the Driver Drowsiness Detection System by K. Sri Jayanthi International Journal of Science, Engineering and Technology Research (IJSETR) Volume 2, Issue 9, September 2013
4. Acharya, S., Sivaprasad, S., & Patil, S. (2018). Driver drowsiness detection system using computer vision: A review. In 2018 International Conference on Recent Innovations in Electrical, Electronics & Communication Engineering (ICRIEECE)
5. Das, M., & Das, N. (2016). Real-time drowsiness detection and alert system for drivers using machine learning techniques. Transportation
6. Damera, H., Gupta, R., & Rangasamy, V. (2019). Driver drowsiness detection system using deep learning techniques. In 2019 2nd International Conference on Advances in Electronics, Computers and Communications
7. Gupta, S., & Gaur, R. (2016). A review on driver drowsiness detection systems. Procedia Computer Science
8. Rodriguez, J. M., Perez, M. A., Pineda, G. J., & Ospina, J. A. (2018). A novel drowsiness detection system for drivers based on electrooculography and support vector machines.