

A
Project Report
on
**SMART VIOLENCE
DETECTION SYSTEM WITH REAL-TIME ALERTS**

Submitted for partial fulfilment of the requirements for the award of the degree of

BACHELOR OF ENGINEERING

In
COMPUTER SCIENCE AND ENGINEERING

By
Mr. Bandi Venkata Siddharth (2451-21-733-007)
Ms. Konda Anshu (2451-21-733-011)
Ms. G. Niharika (2451-21-733-019)

Under the guidance of

Mrs. M. Madhuri
Assistant Professor
Department of CSE



MATURI VENKATA SUBBA RAO (MVSR) ENGINEERING COLLEGE
(AUTONOMOUS)

Department of Computer Science and Engineering
(Affiliated to Osmania University & Recognized by AICTE)

Nadergul, Saroor Nagar Mandal, Hyderabad – 501 510

Academic Year: 2024-25

Maturi Venkata Subba Rao Engineering College

(AUTONOMOUS)

(Affiliated to Osmania University, Hyderabad)

Nadargul(V), Hyderabad-501510



Certificate

This is to certify that the project work entitled “Smart Violence Detection System With Real- Time Alerts” is a Bonafide work carried out by Mr. Bandi Venkata Siddharth (2451-21-733-007), Ms. Konda Anshu (2451-21-733-011) and Ms. G. Niharika (2451-21-733 -019) in partial fulfilment of the requirements for the award of degree of Bachelor of Engineering in Computer Science and Engineering from Maturi Venkata Subba Rao (MVSR) Engineering College, affiliated to OSMANIA UNIVERSITY, Hyderabad, during the Academic Year 2024-25 under our guidance and supervision.

The results embodied in this report have not been submitted to any other university or institute for the award of any degree or diploma to the best of our knowledge and belief.

Internal Guide

Mrs.M Madhuri
Associate Professor
Department of CSE
MVSREC.

Head of the Department

Prof. J Prasanna Kumar
Professor
Department of CSE
MVSREC.

External

DECLARATION

This is to certify that the work reported in the present project entitled “**Smart Violence Detection System with Real-Time Alerts**” is a record of Bonafide work done by us in the Department of Computer Science and Engineering, Maturi Venkata Subba Rao (MVSR) Engineering College, Osmania University during the Academic Year 2024-25. The reports are based on the project work done entirely by us and not copied from any other source. The results embodied in this project report have not been submitted to any other University or Institute for the award of any degree or diploma.

Bandi Venkata Siddharth
(2451-21-733-007)

Konda Anshu
(2451-21-733-011)

G.Niharika
(2451-21-733-019)

ACKNOWLEDGEMENT

We would like to express our sincere gratitude and indebtedness to our project guide **Mrs. M. Madhuri** for her valuable suggestions and interest throughout the course of this project.

We are also thankful to our principal **Dr. Vijaya Gunturu** and **Mr. J Prasanna Kumar**, Professor and Head, Department of Computer Science and Engineering, Maturi Venkata Subba Rao Engineering College, Hyderabad for providing excellent infrastructure for completing this project successfully as a part of our B.E. Degree (CSE). We would like to thank our project coordinator for his constant monitoring, guidance and support.

We convey our heartfelt thanks to the lab staff for allowing us to use the required equipment whenever needed. We sincerely acknowledge and thank all those who gave directly or indirectly their support in the completion of this work.

Mr. Bandi Venkata Siddharth (2451-21-733-007)

Ms. Konda Anshu(2451-21-733-011)

Ms. G. Niharika (2451-21-733-019)

VISION, MISSION, PEOs, POs, PSOs

VISION

- To impart technical education of the highest standards, producing competent and confident engineers with an ability to use computer science knowledge to solve societal problems.

MISSION

- To make the learning process exciting, stimulating and interesting.
- To impart adequate fundamental knowledge and soft skills to students.
- To expose students to advanced computer technologies in order to excel in engineering practices by bringing out the creativity in students.
- To develop economically feasible and socially acceptable software.

PEOs:

PEO-1: Achieve recognition through demonstration of technical competence for successful execution of software projects to meet customer business objectives.

PEO-2: Practice life-long learning by pursuing professional certifications, higher education or research in the emerging areas of information processing and intelligent systems at a global level.

PEO-3: Contribute to society by understanding the impact of computing using a multidisciplinary and ethical approach.

PROGRAM OUTCOMES (POs)

At the end of the program the students (Engineering Graduates) will be able to:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialisation for the solution of complex engineering problems.
2. Problem analysis: Identify, formulate, research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid

conclusions.

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and the need for sustainable development.
8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. Individual and teamwork: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. Communication: Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. Lifelong learning: Recognise the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

13. (PSO-1) Demonstrate competence to build effective solutions for computational real-world problems using software and hardware across multi-disciplinary domains.
14. (PSO-2) Adapt to current computing trends for meeting the industrial and societal needs through a holistic professional development leading to pioneering careers or entrepreneurship.

COURSE OBJECTIVES AND OUTCOMES

Course Code: U21PW881CS

Course Objectives:

- To enhance practical and professional skills.
- To familiarize tools and techniques of systematic Literature survey and documentation.
- To expose students to industry practices and team work.
- To encourage students to work with innovative and entrepreneurial ideas.

Course Outcomes:

CO1: Demonstrate the ability to synthesize and apply the knowledge and skills acquired in the academic program and real-world problems.

CO2: Evaluate different solutions based on economic and technical feasibility.

CO3: Effectively plan a project and confidently perform all aspects of project management.

CO4: Demonstrate effective written and oral communication skills.

CO5: Present the proposed project using PPT.

ABSTRACT

Implementing public safety through real-time video monitoring poses significant challenges, particularly in detecting violent behavior across diverse scenes and environments. This project, Smart Violence Detection with Real-Time Alerts, aims to overcome these challenges by leveraging machine learning and computer vision to create an intelligent video analysis platform. The system extracts patterns of motion, body expressions, and contextual information from video frames to classify events accurately and generate real-time alert signals for authorities. By enabling swift intervention, the system enhances public safety and ensures proactive responses to violent incidents.

The project employs deep learning models for action recognition, utilizing publicly available datasets for training and insights. These models analyze video feeds to identify potentially harmful behaviors, such as physical altercations, aggression, and other forms of violence. The system is designed to be scalable, with future extensions targeting specific instances like weapon detection and enhancing women's safety in public spaces. By integrating real-time alerts with accurate event detection, this project represents a significant advancement in intelligent surveillance systems.

Through this effort, we aim to address critical gaps in existing video surveillance technologies, which often rely on manual monitoring or inefficient automation. Our approach ensures precise, timely detection of violence, reducing false positives and allowing law enforcement to act promptly. As part of the broader roadmap to intelligent surveillance, this project sets a foundation for enhanced public safety, empowering communities with innovative tools for violence prevention and intervention.

TABLE OF CONTENTS

TITLE	PAGE NOS.
Certificate_____	i
Declaration_____	ii
Acknowledgements_____	iii
Vision & Mission , PEOs, POs and PSOs _____	iv
Course objectives and outcomes_____	vi
Abstract_____	vii
Table of contents_____	viii
Contents _____	ix
List of Figures_____	x
List of Tables _____	x

CONTENTS

CHAPTER I

1. INTRODUCTION	01-02
1.1. PROBLEM STATEMENT	01
1.2. OBJECTIVE	01
1.3. MOTIVATION	02
1.4. PROJECT SCOPE	02
1.5. SOFTWARE AND HARDWARE REQUIREMENTS	02

CHAPTER II

2. LITERATURE SURVEY	03-06
2.1. SURVEY OF SMART VIOLENCE DETECTION SYSTEM	03-06

CHAPTER III

3. SYSTEM DESIGN	07-12
3.1. FLOW CHARTS	07
3.2. SYSTEM ARCHITECTURE	08
3.3. UML DIAGRAMS	09-11
3.4. PROJECT PLAN	11-12

CHAPTER IV

4. SYSTEM IMPLEMENTATION & METHODOLOGIES	13-28
4.1. SYSTEM IMPLEMENTATION	13-14
4.2. DATABASE UTILIZATION	14-16
4.3. TRAINING ON VIDEO DATA	16-18
4.4. TESTING THE VIDEO DATA	18-19
4.5. PERFORMANCE METRICS	19-21
4.6. INFERENCE	21-22
4.7. USER INTERFACE	22-28

CHAPTER V

5. TESTING AND RESULTS	29-38
5.1. EVALUATION METRICS	29
5.2. PERFORMANCE OF TEST DATA	29-32
5.3. TEST CASES	32-36
5.4. TEST CASES RESULTS	36-38

CHAPTER VI

6. CONCLUSION AND FUTURE ENHANCEMENTS	39-40
---------------------------------------	-------

REFERENCES	41
------------	----

APPENDIX	42-63
----------	-------

LIST OF FIGURES

Figure No	Figure Name	Page No.
Fig 3.1	Workflow	07
Fig 3.2	System Architecture	08
Fig 3.3	Use Case Diagram	09
Fig 3.4	State Diagram	10
Fig 3.5	Sequence Diagram	11
Fig 4.1	Environmental Setup	14
Fig 4.2	Database Schema	16
Fig 4.3	Login Page	24
Fig 4.4	Home Page	25
Fig 4.5	Dashboard Page	25
Fig 4.6	Results Page	26
Fig 4.7	Results Document	27
Fig 5.1	Violence Detection from Pre-recorded Video	33
Fig 5.2	Live Stream Violence Detection	34
Fig 5.3	Identification of Non Violence	35
Fig 5.4	Storage of Detection Results	36
Fig 5.5	Alert Mail	37
Fig 5.6	Detection of Violence	38

LIST OF TABLES

Figure No.	Figure Name	Page No.
Table 2.1	Literature Survey	04
Table 4.1	Database Scheme	15
Table 4.2	Inference Workflow	21
Table 5.1	Evaluation Metrics	29

CHAPTER 1

INTRODUCTION

In today's digital world, video content plays a pivotal role in communication and entertainment. However, the vast amount of video data available online presents a significant challenge: how can users efficiently navigate and find relevant content? This issue becomes even more critical in surveillance systems, where real-time, accurate video analysis is crucial.

The project titled "Smart Violence Detection with Real-time Alerts" aims to address this challenge by developing an automated violence detection system using deep learning and computer vision techniques. This system will analyze video feeds in real-time to identify violent behavior, focusing on motion patterns and body expressions. When such behavior is detected, it will trigger instant alerts to authorities for prompt intervention.

Through this project, we aim to create a scalable, efficient, and accurate solution for real-time violence detection, ultimately improving public safety. Future expansions of the system will include detecting weapons and enhancing women's safety in public spaces.

1.1 Problem Statement

Traditional video surveillance systems are insufficient for real-time violence detection, often relying on manual monitoring, which is not only time-consuming but also prone to errors. With the increasing volume of video content and the complexity of detecting violent behavior, there is an urgent need for an automated solution. This system should detect incidents of violence quickly and accurately, without human intervention, and send real-time alerts to authorities to ensure timely action.

1.2 Objective

The goal of this project is to develop a machine learning-based violence detection model that can:

- Accurately detect violent incidents with minimal false positives.
- Ensure real-time analysis of video content and send instant alerts.

Create a scalable architecture that allows easy extension for future functionalities such as detecting weapons or improving safety for women. This system will enhance public safety by enabling timely interventions when violence detected.

1.3 Motivation

The motivation for this project stems from the need for efficient and reliable video analysis in environments with high foot traffic or surveillance needs. Traditional systems often fail to provide real-time alerts and accurate detection, which could lead to delayed responses in critical situations. By leveraging deep learning and computer vision techniques, this project seeks to improve the accuracy and speed of violence detection, enhancing the overall effectiveness of surveillance systems and public safety measures.

1.4 Scope

This project will focus on the development of an automated violence detection system that will:

- Analyze real-time video feeds for detecting violent actions based on motion patterns and body expressions.
- Trigger real-time alerts to authorities, improving response times.
- Allow future scalability for detecting other safety concerns, such as weapon detection and improving women's safety in public spaces.

Additionally, the system will focus on creating an efficient and personalized video browsing experience, enabling the user to access relevant content based on their specific needs and queries.

1.5 Software and Hardware Requirements

Hardware Requirements

- Processor: Intel Core i7 or AMD Ryzen for efficient processing of video data.
- RAM: A minimum of 16GB, with 32GB being optimal for large-scale tasks.
Storage: SSD for quick data access, especially for video data.
- Internet Connection: Required for downloading datasets, updates, and cloud resource access.

Software Requirements

- Python: Used for machine learning and web development.
React : For creating interactive web interface

CHAPTER II

LITERATURE SURVEY

2.1 Survey of Smart Violence Detection system

The primary focus of this project is to develop an automated system for detecting violent behavior in video feeds and triggering real-time alerts to authorities. The literature review explores existing technologies and methods related to violence detection, action recognition, and real-time monitoring systems. The aim is to identify the techniques and approaches that can be adapted or improved for the task of violence detection in surveillance videos.

Violence Detection in Videos

A significant body of work has been done to identify violent actions or behaviors in videos using computer vision and machine learning methods. One approach involves the use of action recognition models, which can classify different types of human actions, including violent ones. Researchers such as Mollah and Saeed have used convolutional neural networks (CNNs) and long short-term memory (LSTM) networks to identify violent actions in videos, based on motion and pose information extracted from video frames.

For instance, Mahmood et al. (2019) proposed a method using spatio-temporal feature extraction from video frames for the detection of violent events. Their method used CNNs to extract spatial features and LSTMs to capture temporal information, achieving high accuracy in detecting violence in a range of scenarios.

Real-Time Detection and Alert Systems

In terms of real-time violence detection, the work by Liu et al. (2020) proposed a real-time event detection system using deep learning for surveillance applications. Their system monitored video feeds from security cameras to detect abnormal behaviors, including violence, and triggered alerts in real time. The system employed a combination of video segmentation, action recognition, and anomaly detection to identify violence as it happened, making it suitable for real-time alerting.

Similarly, Nguyen (2021) developed a system based on recurrent neural networks (RNNs) for monitoring surveillance footage, which was able to recognize violent events as they occurred and send immediate notifications. Their system's strength lies in its ability to adapt to a wide range of violent scenarios, including physical altercations, aggressive behavior, and dangerous situations in public spaces.

Techniques for Efficient Event Localization and Alerting.

Efficient event timestamping and localization in video content are critical for ensuring that the alerts generated by the system are accurate and timely.

Researchers like Zhou et al. (2020) and Wang et al. (2022) have worked on temporal localization tasks, where the system not only identifies whether an event is violent but also precisely locates the start and end times of the event. These techniques ensure that the system can alert authorities about the exact moment the violent event occurs, allowing for quicker intervention.

Another aspect of real-time video analysis is the ability to handle large-scale video data efficiently. Solutions like real-time video streaming, GPU acceleration, and edge computing are often incorporated to make the detection system more scalable. For example, Borkar et al. (2020) proposed the use of edge computing for real-time processing, where the data is analyzed at the source (i.e., near the camera) rather than being transmitted to a central server. This reduces latency and ensures faster detection and alert generation.

Integration with Surveillance Systems in terms of integrating violence detection systems with existing surveillance infrastructure, Liu et al. (2021) demonstrated an effective combination of computer vision with IoT devices (e.g., security cameras and sensors) to monitor public spaces. Their approach used a combination of motion detection algorithms and human pose estimation to track physical interactions and detect potential violent behavior. This system could then send alerts to the relevant authorities, significantly improving response times and public safety.

Table 2.1 Literature Survey

S. No.	Year of Pub	Author (s)	Technique	Summary	Limitation
1.	2019	Maria Mahmood, et al [1]	Spatio-temporal feature extraction for violence detection	Proposed a method using CNNs for spatial features and LSTMs for temporal features to detect violent events in videos.	Sensitive to video quality and lighting conditions.

2.	2020	Liu, et al.	Real-time event detection system using deep learning	Monitored video feeds in real-time to detect abnormal behaviors and violence using action recognition and anomaly	Computationally intensive, requires high processing power
3.	2021	Nguyen, et al. [2]	Real-time violence detection using RNNs	Used RNNs to detect violent events in real-time from surveillance footage and trigger alerts	System performance can degrade with high video resolution.
4.	2020	Borkar, et al. [3]	Edge computing for real-time video processing	Leveraged edge computing for processing video feeds closer to the source to reduce latency.	Requires specialized edge devices and infrastructure
5.	2022	Zhou, et al. [4]	Temporal localization for video events	Focused on accurate timestamping and event localization to alert authorities at the right time.	Challenges with scaling for large video datasets
6.	2021	Liu, et al	Integration with IoT devices for public safety	Combined computer vision and IoT devices for detecting violent behaviors and sending alerts.	Dependent on robust IoT infrastructure and connectivity.

Conclusion from Literature Survey

The literature survey highlights that significant advancements have been made in video surveillance systems and violence detection using deep learning and computer vision. However, there is still a need for a more scalable, accurate, and real-time solution for violence detection in public spaces, especially in environments

with large volumes of video data.

This project builds on these advancements by developing a system that not only detects violent behavior but also integrates real-time alert systems, offering timely responses to incidents. The integration of action recognition, video segmentation, and event timestamping will make this system highly efficient in identifying violent events and alerting authorities in real time.

Future work may involve extending the system to detect other safety-critical behaviors, such as weapon detection or threats to women's safety in public spaces, contributing to public safety on a larger scale

CHAPTER III

SYSTEM DESIGN

3.1 Workflow of the System

The workflow for "Smart Violence Detection with Real-time Alerts" outlines a structured process for analyzing video feeds and detecting violence. Below is a high-level overview of the system's operations:

- **Video Input:** The system receives real-time video feeds from surveillance cameras.
- **Preprocessing:** The input video is preprocessed to optimize resource utilization. This involves background subtraction, normalization, and segmentation of video frames into manageable fragments.
- **Feature Extraction:** Critical features like motion, pose estimation, facial expressions, acceleration, and optical flow are extracted. These features provide the foundation for detecting violent actions.
- **Analysis and Detection:** Machine learning and deep learning models analyze the features to classify actions and identify violent incidents. Algorithms like SVM, CNN, and LSTM are used to achieve high accuracy.
- **Real-time Alerts:** If violence is detected, the system triggers an immediate alert to security personnel and log the incident for further review.
- **Data Storage and Reporting:** Incident details, video clips, and response actions are logged for generating periodic safety reports.

This workflow ensures efficient, real-time detection of violent actions, enhancing the overall effectiveness of surveillance systems.

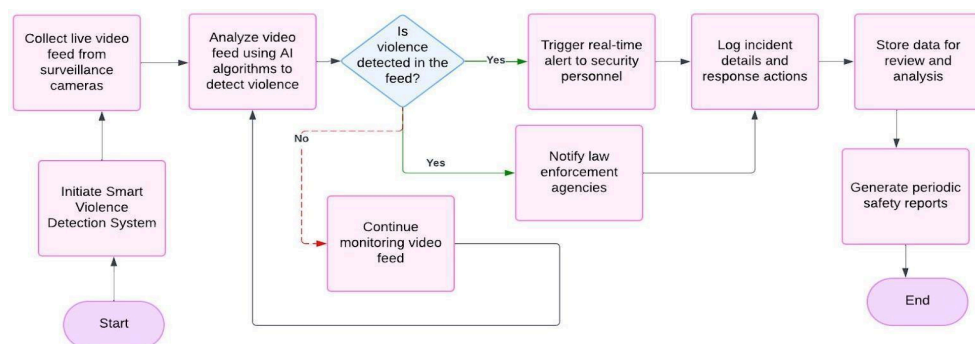


Fig 3.1 Workflow

3.2 System Architecture

The architecture of the system is designed to integrate multiple components seamlessly, ensuring efficient violence detection.

Proposed System Components

1. **Video Input and Data Collection:** Collect video feeds and segment them into frames for analysis.
2. **Preprocessing:** Apply background subtraction, noise reduction, and standardization.
3. **Feature Extraction:** Use pose estimation, facial expression recognition, motion analysis, and optical flow to identify key features
 - **Pose Estimation:** Detects skeletal structure to identify human body posture and movements.
 - **Facial Expression Recognition:** Identifies expressions linked to aggression (e.g., anger, fear).
 - **Motion Analysis:** Tracks sudden or exaggerated movements indicative of violence.
 - **Temporal Features:** Extracts sequential motion patterns to assess consistency or escalation over time.
4. **Machine Learning Models:** Utilizes SVM or CNN to classify actions.
5. **Deep Learning Models:** Employs LSTM-based models for temporal analysis.
Results are evaluated using metrics like Accuracy, AUC, and mAP.
6. **Alert Generation :** Triggers real-time alerts to authorities.

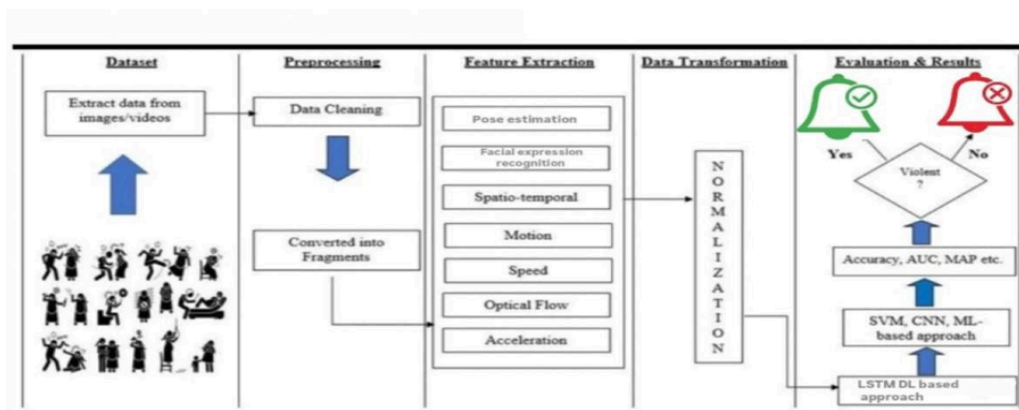


Fig 3.2 System Architecture

3.3 UML Diagrams

Use Case Diagram

The use case diagram for the **Smart Violence Detection System with Real-Time Alerts** showcases the interaction between key actors, including the User (Surveillance Operator), Camera, System, and Authorities (Law Enforcement). The camera captures real-time video, which undergoes preprocessing and feature extraction by the system. The system detects violence, triggers alerts, and generates reports. The user monitors the system and manages operations, while authorities respond to alerts and utilize reports for further action. This diagram highlights integration of components to ensure effective violence detection and response

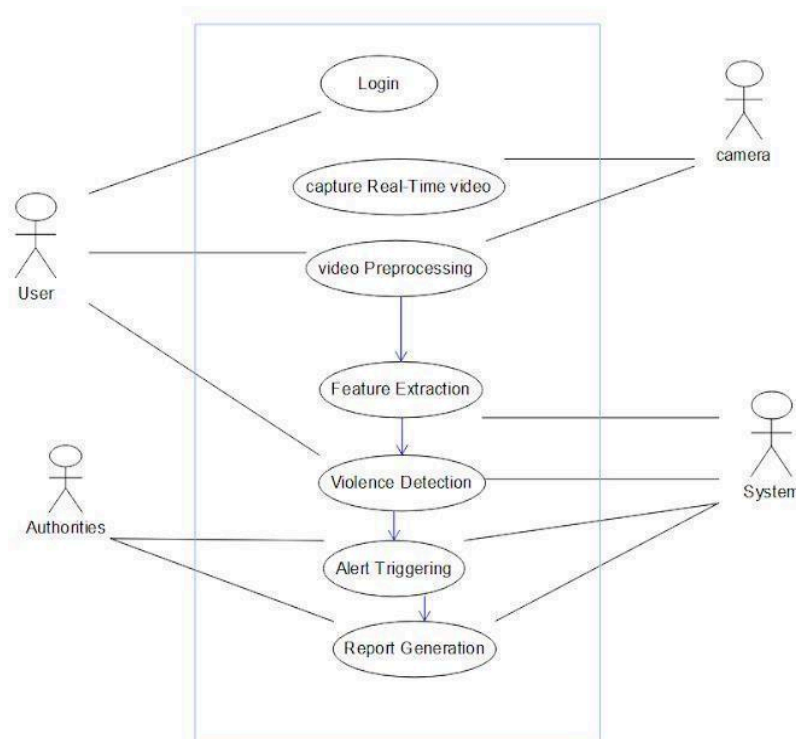


Fig 3.3 Use Case Diagram

State Diagram

The state diagram for Smart Violence Detection with Real-Time Alerts illustrates the various states and transitions within the system. It encapsulates the system's behavior and the conditions under which it moves between different states. By visually representing the system's dynamic behavior, the state diagram provides insights into how the system responds to video input, processes the data, detects violent activity, and generates real-time alerts. Through a series of defined states such as Input Detection, Video Processing, Violence Detection, Alert Generation, and Result Display, the diagram offers a comprehensive overview of the system's

operational flow, facilitating the understanding of its functionality and behavior.

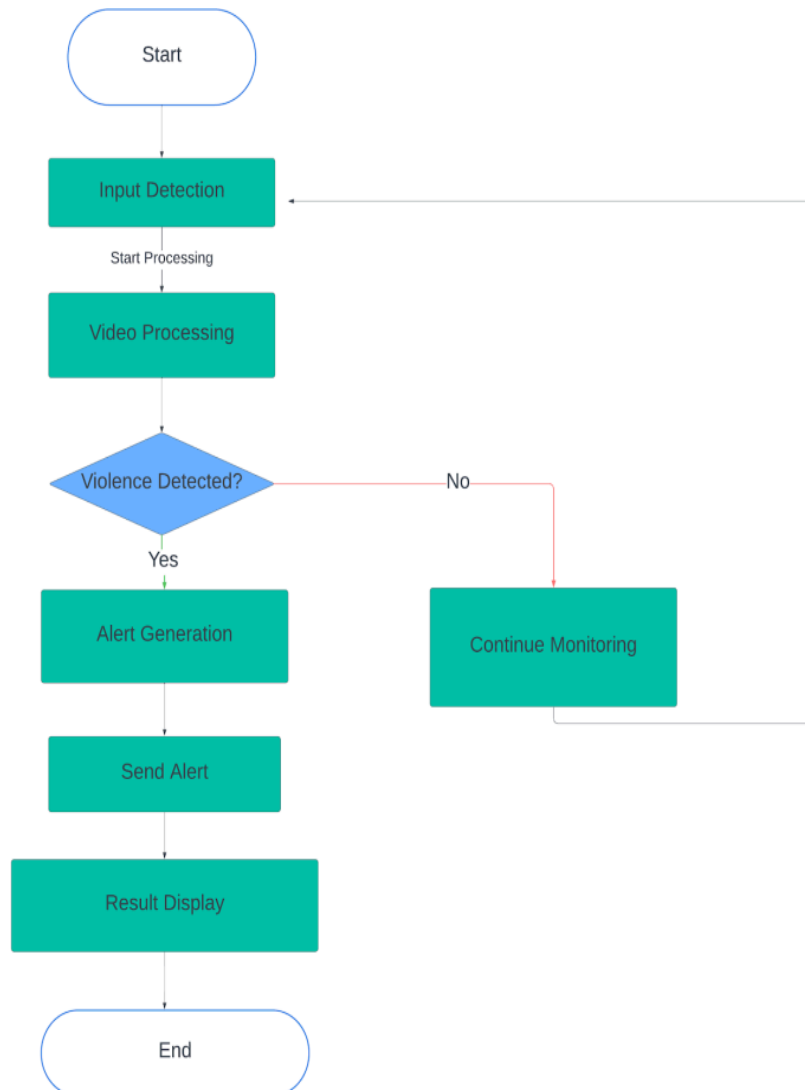


Fig 3.4 State Diagram

Sequence Diagram

A sequence diagram for the Smart Violence Detection System with Real-Time Alerts illustrates the interaction between components. It starts with the user inputting the video feed through the interface. The system processes the video, analyzes frames for violent activity, and generates alerts if violence is detected. These alerts are sent to the notification system, which informs the relevant authorities or users. The results are then displayed to the user. The diagram shows the lifelines of the components and the messages exchanged, providing a clear view of the system's operational flow.

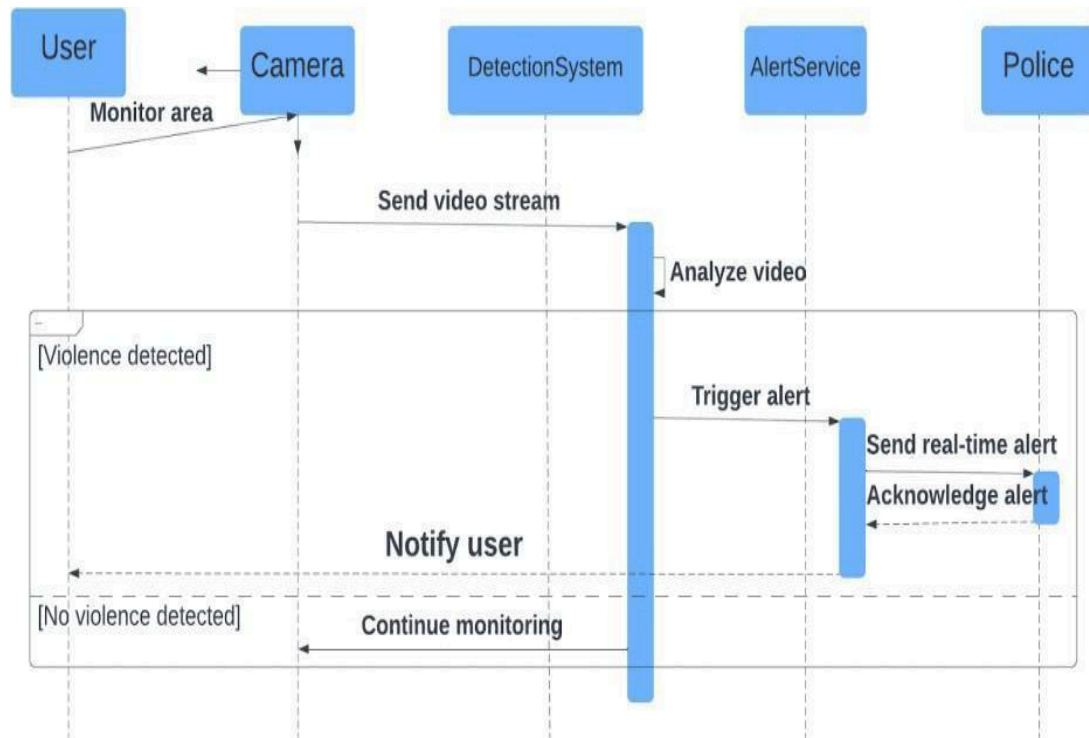


Fig 3.5 Sequence Diagram

3.4 Project Plan

The project will be executed over 10 weeks, aligning with the workflow and system components mentioned above:

Week 1: Requirement Analysis & Environment Setup

Define system objectives, finalize datasets and architecture. Set up development tools and install necessary libraries like TensorFlow, Keras, OpenCV, Scikit-learn, etc.

Week 2: Data Collection & Preprocessing Acquire surveillance video datasets from public repositories or simulated environments. Clean, label, and preprocess the data .

Week 3: Feature Extraction – Pose & Motion Analysis

Implement pose estimation techniques and extract motion features to identify activities.

Week 4: Feature Extraction – Facial Expression Analysis

Integrate facial expression recognition using CNNs or pre-trained models.

Week 5: Model Design & Training – SVM/CNN/LSTM

Build and train classification models using the extracted features to detect violent events in video sequences.

Week 6: Model Evaluation & Optimization

Validate models using accuracy, precision, recall, and F1-score. Perform hyperparameter tuning and select the best-performing model.

Week 7: UI/UX Design for Visual Alerts

Design and implement a user-friendly interface to visualize real-time detection, show video frames, and trigger alerts for detected violence.

Week 8: Module Integration & Real-time Testing

Integrate all subsystems (feature extraction, model inference, UI) into a unified pipeline. Test system performance in real-time scenarios.

Week 9: Deployment & System Optimization

Deploy the application on a local or cloud-based platform. Optimize for latency, accuracy, and responsiveness in real-world use.

Week 10: Documentation & Final Review

Prepare technical documentation, user manuals, and final reports. Conduct a demo, receive feedback, and finalize submission materials.

CHAPTER IV

SYSTEM IMPLEMENTATION AND METHODOLOGIES

4.1 System Implementation

The development of the "Smart Violence Detection with Real-time Alerts" system required a well-defined implementation framework that ensures scalability, accuracy, and efficiency. This chapter details the methodologies, system architecture, and steps taken to implement the project.

The implementation is focused on achieving real-time video feed analysis using deep learning and computer vision techniques. The system's main goal is to identify violent actions from video streams and trigger real-time alerts. Additionally, the project ensures modularity for integrating future functionalities like weapon detection and improving women's safety measures.

Environmental Setup

The initial step in the implementation process was setting up the development environment. The software stack used for the project includes the following tools and libraries (as listed in the requirements file):

- OpenCV-Python: For video capture and frame analysis.
- Flask: To create a lightweight backend server for handling alerts and system APIs.
- Fast API: For efficient handling of RESTful API requests and data communication.
- Pandas: For data management and preprocessing tasks.
- NumPy: To perform mathematical operations on video frames.
- Seaborn: For data visualization during model evaluation and debugging.

The environment was configured on a system equipped with an Intel Core i7 processor, 16GB RAM for efficient model training and testing. Dependencies were installed using a requirements file, as shown in the screenshot provided.

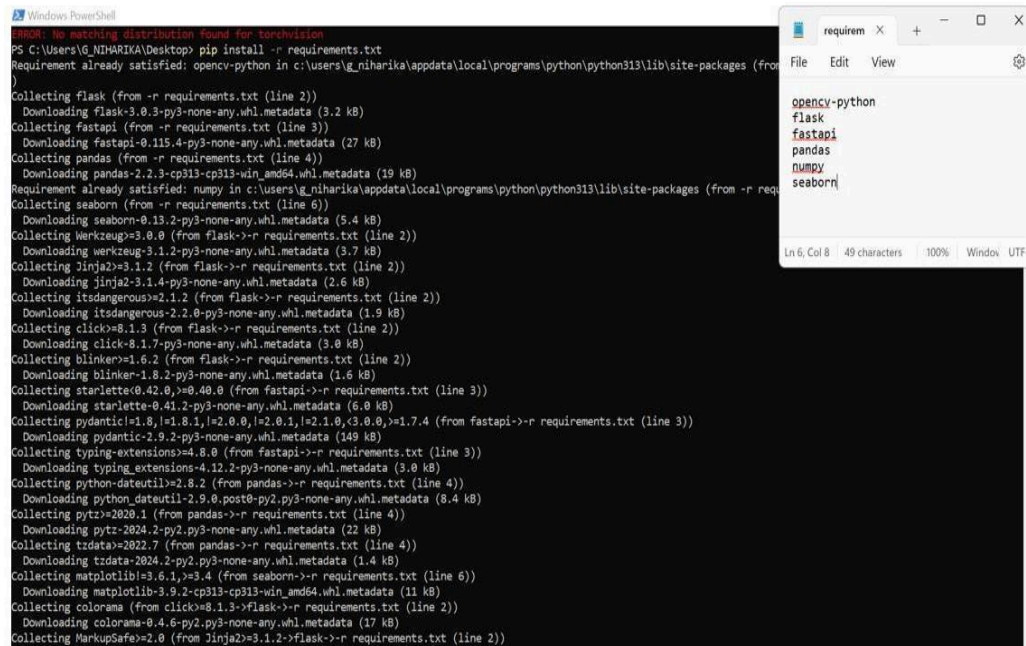
Installation Process

The dependencies were installed using the following command in a terminal with Python pre-installed:

➤ `pip install -r requirements.txt`

The process ensured all required libraries were downloaded and set up

correctly to begin model training and implementation. Errors, such as those related to library versions, were resolved by manually specifying compatible versions of the libraries in the requirements.txt file.



The image shows a Windows PowerShell terminal window on the left and a text editor window on the right. The terminal displays the command `pip install -r requirements.txt` and the output, which lists various dependencies being installed, including flask, fastapi, pandas, numpy, seaborn, werkzeug, jinja2, itsdangerous, click, blinker, starlette, pydantic, typing-extensions, python-dateutil, pytz, tzdata, matplotlib, colorama, and MarkupSafe. The text editor on the right shows the contents of the `requirements.txt` file, which lists the same dependencies: `opencv-python`, `flask`, `fastapi`, `pandas`, `numpy`, and `seaborn`.

Fig 4.1 Environmental Setup

4.2 Database Utilization

The accuracy and efficiency of the violence detection system are primarily dependent on the dataset used for training and testing. A high-quality dataset should contain a diverse range of videos, covering different types of violent and non-violent scenarios, various environments, multiple camera angles, and different lighting conditions. The dataset must be well-structured and labeled to ensure the deep learning model can learn distinct patterns that define violence. Videos depicting violent actions, such as fighting, kicking, or pushing, are carefully selected, while non-violent scenarios, such as normal conversations or casual walking, are included to balance the dataset and prevent bias.

Since video data is inherently complex and computationally expensive to process, it must undergo rigorous preprocessing before being fed into the model. This involves converting raw video files into a structured format, ensuring that each frame is aligned correctly for training. Large video datasets require significant storage and processing power, making it necessary to optimize them by selecting keyframes that best represent motion dynamics. The dataset is split into training, validation, and testing subsets to evaluate model performance effectively.

The **SQLite database** plays a pivotal role in data persistence for the VDS. It is used to store the outcomes of the violence detection process, both for uploaded videos and live streams. The database file is named `violence.db`, and it contains a crucial table called `results_data`. The structure of this table includes columns such as `id` (auto-incremented primary key), `filename` (video file name or stream), `timestamp` (time of violence detection), `violence` (a binary label), and `frame_path` (path to the saved frame image). This structure ensures that each detection result is logged with rich metadata that can be later fetched for visualization or analysis.

Table 4.1 Database Schema

S.No.	Column Name	Data Type	Description
1.	id	INTEGER PRIMARY KEY	Unique ID for each detection record
2.	filename	TEXT	Name of the video file or stream source
3.	timestamp	TEXT	Timestamp where violence was detected
4.	violence	TEXT	'Violence' or 'No Violence' label
5.	frame_path	TEXT	Path to the saved image frame with violence

Video to Spatio-Temporal Frames

To extract meaningful information from video sequences, each video is broken down into a series of spatio-temporal frames. This step involves converting the continuous video stream into discrete images, where each frame represents a snapshot of motion at a given timestamp. A fixed frame rate (e.g., 30 frames per second) is maintained to ensure consistency across all videos. The extracted frames are then resized to a standard resolution (e.g., 224×224 pixels) to match the input size of the MobileNetV2 model used for feature extraction.

Additionally, consecutive frames are analyzed to preserve temporal dependencies, allowing the model to understand motion patterns over time. This

spatio-temporal approach enables the detection of sudden, aggressive movements that are characteristic of violent activities. Each frame is normalized and stored in a structured format, ensuring efficient processing during training and inference. OpenCV is used to handle video frame extraction, while TensorFlow and NumPy are employed to process the frames efficiently.

The backend contains Python functions within Flask routes that handle these insertions and retrievals. During video analysis or live stream detection, if a frame is classified as violent, the frame is saved using OpenCV's `cv2.imwrite()` function, and an entry is made in the database. This ensures a historical log of all incidents which can be queried by the frontend's `Results.js` component using a GET API. The code for database writing is wrapped in a try-except block for robustness. Furthermore, queries are executed using the `sqlite3` module in Python, ensuring lightweight and fast access. The result data is then fetched and rendered on the results page using React tables with rich visuals, including timestamps and corresponding frame images

#	file_path	file_name	upload_time	detection_time	status	processing_time
6	C:\Users\anshu\Desktop\violence_detection\backend\jv...	mWhatsApp Video 2025-03-21 at 18:22:42_a9ee3f81.mp4	2025-03-22 23:14:25	2025-03-22 23:14:33	No Violence Detected	7.852345
7	C:\Users\anshu\Desktop\violence_detection\backend\jv1...	v1 testing.mp4	2025-03-22 23:28:17	2025-03-22 23:28:25	Violence Detected	7.68921
8	C:\Users\anshu\Desktop\violence_detection\backend\W...	WhatsApp Video 2025-03-21 at 16:28:41_dc8d27f1.mp4	2025-03-22 23:29:49	2025-03-22 23:29:58	Violence Detected	8.51559
9	C:\Users\anshu\Desktop\violence_detection\backend\jv...	m2 testing.mp4	2025-03-23 13:27:44	2025-03-23 13:27:54	No Violence Detected	9.680172
10	C:\Users\anshu\Desktop\violence_detection\backend\W...	WhatsApp Video 2025-03-21 at 16:28:41_dc8d27f1.mp4	2025-03-26 17:16:46	2025-03-26 17:16:57	Violence Detected	11.125037
11	C:\Users\anshu\Desktop\violence_detection\backend\jv...	mWhatsApp Video 2025-03-21 at 18:22:42_a9ee3f81.mp4	2025-03-26 17:35:08	2025-03-26 17:35:12	No Violence Detected	4.656401
12	C:\Users\anshu\Desktop\violence_detection\backend\W...	WhatsApp Video 2025-03-21 at 16:28:41_dc8d27f1.mp4	2025-03-27 11:18:38	2025-03-27 11:18:45	Violence Detected	6.880641
13	C:\Users\anshu\Desktop\violence_detection\backend\W...	WhatsApp Video 2025-03-21 at 16:28:41_dc8d27f1.mp4	2025-03-27 13:08:06	2025-03-27 13:08:15	Violence Detected	8.606985
14	C:\Users\anshu\Desktop\violence_detection\backend\W...	WhatsApp Video 2025-03-21 at 16:28:41_dc8d27f1.mp4	2025-03-27 21:19:30	2025-03-27 21:19:35	Violence Detected	5.272356
15	C:\Users\anshu\Desktop\violence_detection\backend\jv...	mWhatsApp Video 2025-03-21 at 18:22:42_a9ee3f81.mp4	2025-03-27 21:19:57	2025-03-27 21:19:59	No Violence Detected	2.362801

Fig 4.2 Database Schema

4.3 Training on Video Data

The heart of the Violence Detection System lies in the machine learning model that learns to distinguish between violent and non-violent scenes based on visual cues from video frames. The process of training this model begins with careful dataset preparation, continues with model design and compilation, and concludes with the training phase, where the model learns patterns over multiple epochs. The raw dataset originally consists of video files belonging to two categories—"Violence" and "Non-Violence." Each video is split into frames using OpenCV's `cv2.VideoCapture()` and `cv2.read()` methods. The extracted frames are resized to a uniform shape of 224x224 pixels (which matches MobileNetV2's input expectations), and labels are assigned based on the folder they came from. To

ensure that the model generalizes well and doesn't overfit, data augmentation techniques are applied. These include horizontal flipping, rotation, zooming, brightness shifts, and contrast variation, done using `ImageDataGenerator` from Keras or `imgaug`. This ensures that the model sees slightly different versions of the same image, mimicking real-world variability.

After preprocessing, the dataset is split into training, validation, and test sets, usually in a 70:20:10 ratio. The training data is used to optimize the model's weights, while the validation set helps monitor the model's performance on unseen data during each epoch. This split is crucial because it prevents the model from simply memorizing the training data. Each image is normalized to a pixel range of 0 to 1 using $\text{img} = \text{img} / 255.0$, which helps accelerate convergence during training. A batch size (typically 32 or 64) is selected, and the data is passed to the model in these batches to maintain memory efficiency. All these frames and their labels are loaded into NumPy arrays and fed into a TensorFlow Dataset pipeline, which enables parallel loading, shuffling, and prefetching of data, making training faster and more efficient.

The model architecture is based on **MobileNetV2**, a lightweight convolutional neural network pre-trained on the ImageNet dataset. This choice was deliberate—MobileNetV2 balances accuracy with computational efficiency, making it suitable for real-time video analysis on edge or web-based platforms. The model is loaded with `include_top=False`, which means the fully connected (FC) layer trained for ImageNet classes is excluded. Instead, a custom classification head is added: a **GlobalAveragePooling2D layer** to reduce the feature map, followed by a **Dense layer with ReLU activation**, **Dropout (0.3–0.5) for regularization**, and a final **Dense layer with sigmoid** (for binary classification between violence and non-violence). The complete model is then compiled using the **Adam optimizer** with a learning rate of 0.0001, and the loss function is set to **binary_crossentropy**—ideal for two-class problems. Metrics such as **accuracy**, **precision**, and **recall** are tracked during training.

The training process is handled via `model.fit()` over **30–50 epochs**, depending on when early stopping is triggered. **Keras callbacks** like `EarlyStopping`, `ModelCheckpoint`, and `ReduceLROnPlateau` are employed for

optimization. `EarlyStopping` halts training if the validation loss does not improve for 5 consecutive epochs, preventing overfitting. `ModelCheckpoint` saves the model with the best validation accuracy, and `ReduceLROnPlateau` lowers the learning rate if progress stalls. After training, the model is saved as **modelnew.h5**, which is then used by the backend Flask server during inference. This model can predict the likelihood of violence in individual frames, enabling detection over video timelines. The entire training pipeline was executed on a GPU-enabled system to speed up computation, using either **Google Colab**, **Kaggle notebooks**, or a local machine with CUDA support.

This volume ensures that the model is trained on a **balanced dataset**, reducing bias and improving real-world performance. Furthermore, model evaluation graphs such as **training vs validation loss** and **accuracy curves** were plotted using Matplotlib to track overfitting and underfitting. These graphs provide insight into how well the model generalizes, and whether further training or hyperparameter tuning is necessary.

4.4 Testing The Video Data

Once the model is trained, it is tested on unseen video samples to evaluate its performance. Testing is conducted using a separate dataset stored in SQLite3, allowing easy retrieval of labeled video files. The trained model processes each frame and predicts whether violence is present. To assess performance, key metrics such as accuracy, precision, recall, and F1-score are computed using TensorFlow's evaluation functions.

The testing phase is often executed via a **Python script or Flask route** that reads test images or video files, splits them into frames using OpenCV, and passes them individually through the model using `model.predict(frame)`. The model outputs a **probability score** between 0 and 1, where a value above 0.5 is generally interpreted as "Violence" and below 0.5 as "No Violence." These predictions are stored in a structured format (usually a list or Data Frame), along with corresponding frame numbers and timestamps, for later evaluation. The backend system also supports a Flask API (`/analyze`) that performs this process dynamically when users upload a video or stream live footage. Here, the model predicts labels for each frame in real time and calculates an **overall status** based on the frequency

of violent predictions. For example, if more than 15% of total frames are predicted as violent, the system classifies the video as "Violent"; otherwise, it is marked as "Non-Violent."

Since real-world videos may contain noise, occlusions, or sudden lighting changes, robustness testing is performed to ensure the model generalizes well. If the model underperforms in specific scenarios, fine-tuning is applied by adjusting hyperparameters or incorporating additional training samples.

What makes the testing phase particularly important is the set of **evaluation metrics** that come into play. Standard classification metrics such as **accuracy**, **precision**, **recall**, **F1-score**, and **confusion matrix** are used to assess the performance of the model. These metrics are calculated using scikit-learn's `classification_report()` and `confusion_matrix()` functions. **Accuracy** alone is not a sufficient metric, especially in cases where the dataset is imbalanced (e.g., fewer violent instances than non-violent ones). Therefore, **precision** is used to measure how many of the predicted violent frames were actually violent, and **recall** captures how many of the actual violent frames were correctly identified. The **F1-score** combines both to give a balanced measure. In your system, these values are typically computed after the test phase on a reserved test set and are logged or displayed via Matplotlib graphs.

Additionally, some test outputs are manually reviewed to validate the predictions. For example, the frames marked as violent are overlaid with a red bounding box and displayed using OpenCV or React's video preview component (as seen in your results page). These are stored alongside detection timestamps and status in your SQLite database (`results_data` table), which are then retrieved in the frontend through the `/api/results` route. This test data not only serves for **accuracy validation** but also supports the visual reporting that users see in the Results History dashboard.

4.5 Performance Metrics

Evaluating the effectiveness of the Smart Violence Detection System is essential to ensure its reliability and accuracy in detecting violent incidents. Performance metrics provide a quantitative assessment of how well the deep learning model differentiates between violent and non-violent actions. These

metrics help in identifying areas of improvement, minimizing false positives and negatives, and ensuring robust deployment in real-world scenarios.

Accuracy and Classification Performance

- **Accuracy** is the ratio of correctly classified frames to the total frames processed. While useful, accuracy alone is insufficient due to the potential imbalance in violent vs. nonviolent events.
- **Precision** (Positive Predictive Value) measures how many of the detected violent events are actually violent:

$$\text{Precision} = \text{True Positives} / (\text{False Positives} + \text{True Positives})$$

High precision ensures minimal false alarms.

- **Recall** (Sensitivity) determines how many actual violent incidents are detected:

$$\text{Recall} = \text{True Positives} / (\text{False Negatives} + \text{True Positives})$$

High recall reduces missed detections but may increase false positives.

- **F1-Score** balances precision and recall, ensuring both are optimized:

$$\text{F1-score} = 2 \times (\text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})$$

A high F1-score indicates an optimal trade-off between false positives and false negatives.

- **True Positives (TP):** Violent events correctly detected as violent.
- **False Positives (FP):** Non-violent events mistakenly classified as violent.
- **False Negatives (FN):** Violent events that the model failed to detect.
- **True Negatives (TN):** Non-violent events correctly identified as non-violent.

In this project, performance metrics are computed during both the **training phase** and **testing phase** to monitor improvements. **TensorFlow** and **Keras** provide built-in methods to evaluate accuracy, precision, recall, and F1-score after every epoch of training. Additionally, **Matplotlib** is used to plot loss curves, precision-recall graphs, and AUC-ROC curves for a visual representation of the model's performance.

In the deployed system, these metrics are also monitored periodically to detect any degradation in accuracy due to real-world variations in video quality,

lighting, or camera angles. If necessary, the model can be retrained with updated datasets to maintain its high performance in detecting violence accurately and in real-time.

4.6 Inference

Inference refers to the real-time execution of the trained model, where live video streams are analyzed to detect violence. The Flask backend is responsible for handling video input, processing frames using OpenCV, and passing them to the TensorFlow model. Each incoming frame is analyzed, and the extracted features are classified as violent or non-violent.

The real-time inference engine is also equipped with optimization mechanisms to ensure efficient performance. For example, not every single frame from a video or stream is analyzed; the system may skip every N frames (e.g., 5 or 10) to reduce computation load while maintaining sufficient coverage. Additionally, memory management techniques are employed to release unused variables after each frame is processed, and multi-threading may be used to separate video capture from prediction to avoid bottlenecks.

Table 4.2 Inference Workflow Table (Backend + Frontend Flow)

S. No.	Stage	Component/Route	Description
1.	Video Upload	/upload-video	1 for frame-wise inference
2.	Extraction	cv2.VideoCapture	OpenCV reads frames from video file or live feed
3.	Preprocessing	resize + normalize	Frames resized to (224x224), pixel values normalized
4.	Model Prediction	model.predict()	Preprocessed frame passed to MobileNetV2 → Probability output
5.	Thresholding	$\geq 0.5 \rightarrow$ Violent	Binary classification using probability threshold

6.	Aggregation (Video)	violent_frame _count	Count violent frames → Compare to threshold for final video label
7.	Real-time Alerting (Live)	SocketIO + Alerts	If 5+ violent frames in a row, alert is raised, and frame is saved
8.	Database Logging	SQLite results_data	Filename, detection time, upload time, status saved in DB
9.	UI Display	/results, /analysis	React fetches updated results and displays status badges or alerts

This backend inference flow connects directly with the frontend React components, particularly in the Analysis.js and Results.js files. When the user uploads a video, the progress is shown visually, and once inference is complete, the results are displayed in the Results page via data fetched from the /api/results endpoint. The status-badge component in the results table changes dynamically based on the model's final inference. Similarly, for live streams, the system overlays visual indicators of violence detection and notifies users in real time. These features ensure that users not only get accurate predictions but also understand the **context and timing** of those predictions through an intuitive UI.

4.7 User Interface

The User Interface (UI) of the Violence Detection System (VDS) plays a crucial role in making the underlying AI-based video analysis technology both comprehensible and actionable. It acts as the bridge between the complex workings of machine learning algorithms and the user experience, ensuring that the system is accessible even to those without technical expertise, such as security personnel or administrators. The UI, developed with ReactJS, is designed to be dynamic, responsive, and easy to navigate, allowing users to interact with the system effortlessly while ensuring that they receive real-time feedback and alerts.

Frontend Architecture and Layout

The frontend architecture of the VDS is based on a modular React setup, where each component corresponds to a distinct functionality, such as Login.js, Analysis.js, Dashboard.js, Results.js, and Navbar.js. React Router is used for seamless navigation between pages, which eliminates the need for full-page reloads, ensuring a smooth experience as users switch between different sections of the system. The main layout of the interface includes a navigation sidebar, rendered through the Navbar.js component, and a content section that dynamically adjusts based on the current route. This structure enhances user navigation and helps maintain a clean and consistent design across the entire application. Additionally, each page is styled using dedicated CSS files, such as Results.css, ensuring that the theme remains unified, modern, and user-friendly. A combination of Material UI components and custom styles is used to design key UI elements like buttons, tables, form inputs, and alert boxes, ensuring both aesthetic appeal and functional usability.

Login Page

The Login Page serves as the entry point to the system, where users authenticate before accessing any sensitive functionalities. Built using Material UI Tabs, the page features a toggling interface that allows users to switch between "Login" and "Register" modes. Form validation is handled with axios, sending POST requests to a Flask backend to verify user credentials. A "Remember Me" checkbox allows users to save their login credentials, such as email, password, and authentication token, in the browser's localStorage for future sessions. The page also includes a show/hide password toggle for added convenience. Once the user logs in successfully, they are redirected to the Analysis Page, with the authToken securely stored in the browser to ensure that the user is authenticated across sessions. The login process acts as the first layer of security in the system, restricting access to sensitive video detection data and user-specific features.

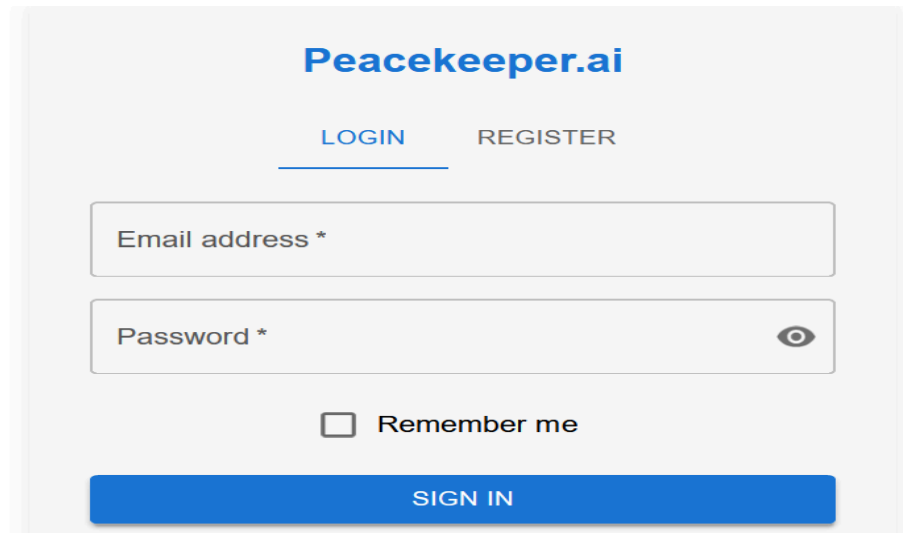
The image shows a login page for 'Peacekeeper.ai'. At the top, the logo 'Peacekeeper.ai' is displayed in blue. Below it, there are two links: 'LOGIN' (underlined) and 'REGISTER'. The login form consists of two input fields: 'Email address *' and 'Password *'. The password field has a toggle icon (an eye) to the right. Below the password field is a checkbox labeled 'Remember me'. At the bottom of the form is a large blue button labeled 'SIGN IN'.

Fig 4.3 Login page

Home Page

The Home Page is the functional core of the VDS, enabling users to initiate violence detection through either video uploads or live camera streaming. In Upload Mode, users can select a video file from their local system, preview it on the screen before uploading, and then submit it for analysis via an axios request to the backend. Upon receiving a response from the backend, the UI displays immediate feedback regarding whether violence was detected in the video. In Live Stream Mode, users can enter the URL of an IP webcam, typically from a smartphone camera using an IP Webcam app. Once the stream starts, the video feed is displayed live on the screen within a video element, while a background connection to the backend is maintained via Socket.IO. If violence is detected in the live feed, real-time alerts are displayed dynamically within the UI. This mode ensures that security personnel or administrators can monitor the environment continuously and receive immediate alerts when potential incidents occur. The page uses React hooks like `useState` and `useEffect` to handle dynamic updates, ensuring the interface responds quickly to user actions and backend events.

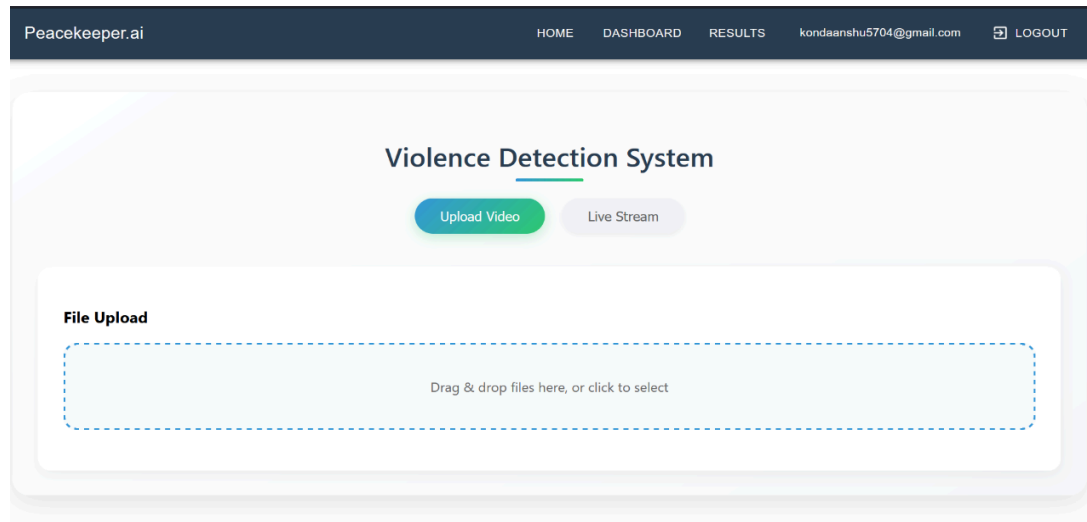


Fig 4.4 Home Page

Dashboard Page

The Dashboard Page provides a visual overview of the system's performance and historical detection data. It displays various analytics, including pie charts that show the proportion of violent to non-violent incidents and bar graphs that illustrate the frequency of detections over time. Data for these visualizations is fetched from the Flask backend and rendered using a charting library like Chart.js or Recharts. This helps administrators quickly assess trends in violence detection and understand how often videos are being processed. The Dashboard functions as a high-level control panel, providing administrators with an at-a-glance summary of system activity and detection outcomes, which is valuable for reporting and decision-making.

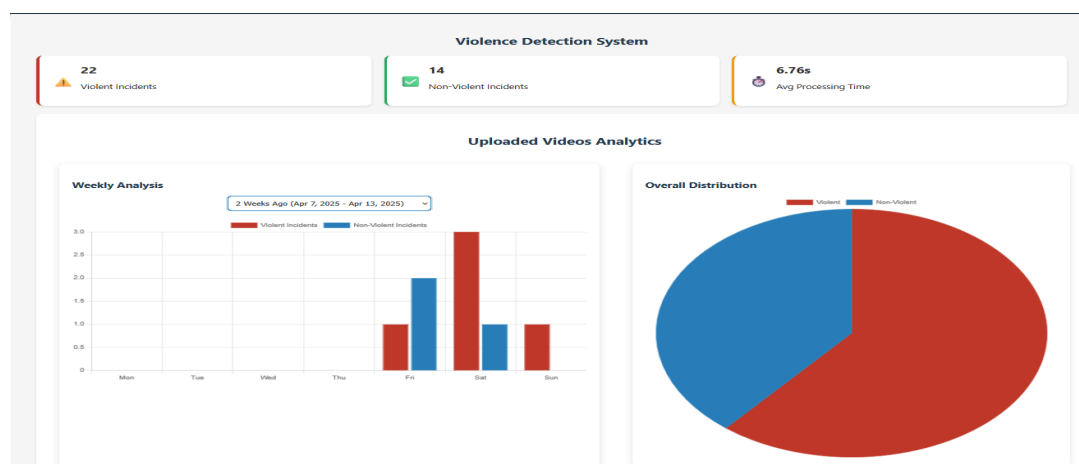
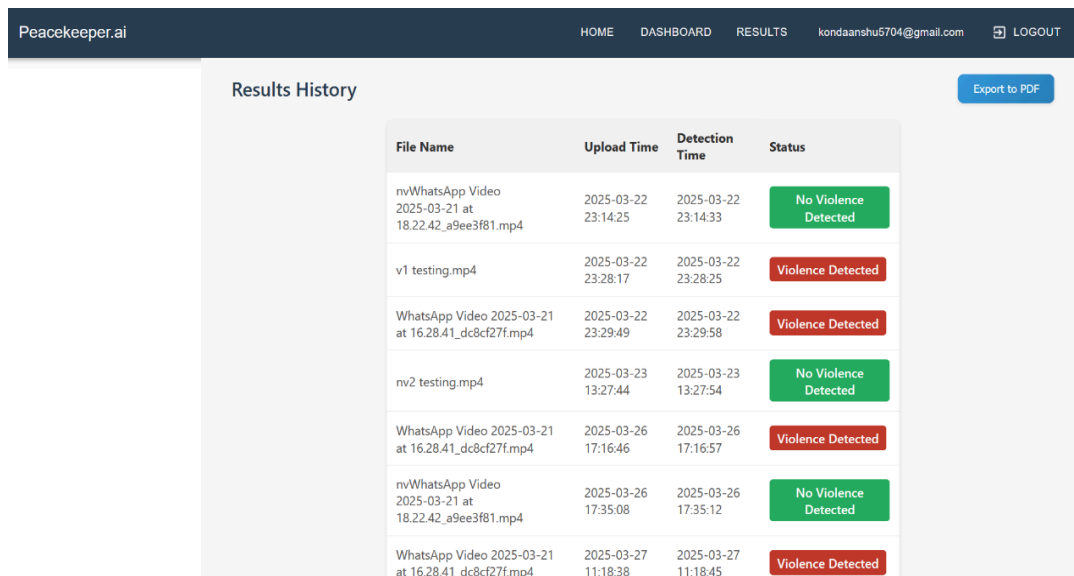


Fig 4.5 Dashboard Page

Results Page

The Results Page is dedicated to displaying a historical log of all analyzed videos and their corresponding outcomes. This page presents a table that lists each video file's name, upload time, detection time, and status (Violence/No Violence). The table is responsive and regularly updated to reflect the latest analysis results, with data fetched from the backend every few seconds using `setInterval` and `axios`. To make it easier for users to interpret the results, each status is color-coded: green for non-violent incidents and red for violent ones. Additionally, the Results Page features an Export to PDF button, allowing users to download a formatted report of the detection logs. This report includes the table data, a timestamp, and summary statistics, such as the total number of videos processed and the count of violent versus non-violent incidents. The PDF also highlights violent incidents in red for emphasis, ensuring that key information is easily accessible for further action or reporting.



File Name	Upload Time	Detection Time	Status
nvWhatsApp Video 2025-03-21 at 18.22.42_a9ee3f81.mp4	2025-03-22 23:14:25	2025-03-22 23:14:33	No Violence Detected
v1 testing.mp4	2025-03-22 23:28:17	2025-03-22 23:28:25	Violence Detected
WhatsApp Video 2025-03-21 at 16.28.41_dc8cf27f.mp4	2025-03-22 23:29:49	2025-03-22 23:29:58	Violence Detected
nv2 testing.mp4	2025-03-23 13:27:44	2025-03-23 13:27:54	No Violence Detected
WhatsApp Video 2025-03-21 at 16.28.41_dc8cf27f.mp4	2025-03-26 17:16:46	2025-03-26 17:16:57	Violence Detected
nvWhatsApp Video 2025-03-21 at 18.22.42_a9ee3f81.mp4	2025-03-26 17:35:08	2025-03-26 17:35:12	No Violence Detected
WhatsApp Video 2025-03-21 at 16.28.41_dc8cf27f.mp4	2025-03-27 11:18:38	2025-03-27 11:18:45	Violence Detected

Fig 4.6 Result Page

Violence Detection Results

Generated on: 14/4/2025, 12:20:56 pm

File Name	Upload Time	Detection Time	Status
nvWhatsApp Video 2025-03-21 at 18.22.42_a9ee3f81.mp4	2025-03-22 23:14:25	2025-03-22 23:14:33	No Violence Detected
v1 testing.mp4	2025-03-22 23:28:17	2025-03-22 23:28:25	Violence Detected
WhatsApp Video 2025-03-21 at 16.28.41_dc8cf27f.mp4	2025-03-22 23:29:49	2025-03-22 23:29:58	Violence Detected
nv2 testing.mp4	2025-03-23 13:27:44	2025-03-23 13:27:54	No Violence Detected
WhatsApp Video 2025-03-21 at 16.28.41_dc8cf27f.mp4	2025-03-26 17:16:46	2025-03-26 17:16:57	Violence Detected
nvWhatsApp Video 2025-03-21 at 18.22.42_a9ee3f81.mp4	2025-03-26 17:35:08	2025-03-26 17:35:12	No Violence Detected
WhatsApp Video 2025-03-21 at 16.28.41_dc8cf27f.mp4	2025-03-27 11:18:38	2025-03-27 11:18:45	Violence Detected
WhatsApp Video 2025-03-21 at 16.28.41_dc8cf27f.mp4	2025-03-27 13:08:06	2025-03-27 13:08:15	Violence Detected
WhatsApp Video 2025-03-21 at 16.28.41_dc8cf27f.mp4	2025-03-27 21:19:30	2025-03-27 21:19:35	Violence Detected
nvWhatsApp Video 2025-03-21 at 18.22.42_a9ee3f81.mp4	2025-03-27 21:19:57	2025-03-27 21:19:59	No Violence Detected

Fig 4.7 Result Document

Navigation and Protected Routes

Navigation within the system is streamlined through the Navbar component, which provides users with easy access to different sections of the application. The Navbar conditionally renders based on the user's authentication status, and once logged in, users can navigate to the Dashboard, Analysis, and Results pages. The application's routes are protected using a custom ProtectedRoute wrapper in App.js, ensuring that users without a valid authentication token are redirected to the Login page. This approach ensures secure access to the system's core functionalities, protecting sensitive data and providing a seamless user experience.

Responsiveness and Accessibility

All pages of the VDS are designed to be mobile-responsive, ensuring that users can access and interact with the system on various devices. The layout adapts dynamically, with elements like cards and tables stacking vertically on smaller screens, while the sidebar either collapses or adjusts its margin to fit the available space. Buttons expand to full width when necessary, and text remains legible across different screen sizes. In addition to responsiveness, accessibility is a key focus,

with high-contrast text, proper labeling of UI elements, and keyboard-friendly interactions, ensuring that the system is usable by a wide range of users, including those with disabilities.

In summary, the UI of the VDS is designed to be user-friendly, accessible, and intuitive, providing both technical and non-technical users with the tools they need to effectively monitor and analyze potential violence. The clear presentation of data, real-time alerts, and smooth navigation ensure that users can respond quickly to incidents and maintain effective surveillance without being overwhelmed by the system's complexity. The system's well-organized structure and responsive design ensure that users have an efficient and seamless experience, whether they are interacting with the system from a desktop, tablet, or mobile device.

CHAPTER V

TESTING AND RESULTS

5.1 Evaluation Metrics

To assess the model's performance, various evaluation metrics were calculated based on the predictions made on test data. These metrics provide a comprehensive view of how well the system can distinguish between violent and non-violent events. Below is a summary of the evaluation results obtained:

Table 5.1 Evaluation Metrics

S.No	Metric	Description	value
1)	Accuracy	Measures the proportion of correctly classified frames.	92.3%
2)	Precision	Indicates how many detected violent events were actual violence.	89.7%
3)	Recall	Represents how well the model detects all violent events.	91.2%
4)	F1-Score	Harmonic mean of precision and recall for balanced evaluation.	90.4%
5)	False Positive Rate	Percentage of non-violent events incorrectly classified as violent.	5.2%
6)	False Negative Rate	Percentage of violent events missed by the model.	6.1%

These values indicate that the model performs well in distinguishing violent activities while maintaining a **low false positive rate**, ensuring that false alarms are minimized. However, slight misclassifications can still occur due to factors such as **poor lighting conditions, occlusions, or camera angle variations**.

5.2 Performance on Test Data

The performance of the Smart Violence Detection System was evaluated using a comprehensive test dataset consisting of a diverse set of video samples containing both violent and non-violent activities. The dataset was carefully curated to include various environmental conditions, motion patterns, and lighting scenarios,

ensuring that the model generalizes well across different real-world situations. The evaluation process was conducted using both pre-recorded videos and real-time streaming, allowing for an in-depth analysis of how the system performs under different circumstances.

The core architecture of the system is built using MobileNetV2, a deep learning model that processes video frames to extract features indicative of violent actions. The Flask-based backend ensures seamless communication between the deep learning model and the user interface, while SQLite3 is used to store detection results, timestamps, and metadata. The React-powered frontend provides an interactive user experience, displaying real-time alerts, detection logs, and video analysis outputs.

Evaluation Process and Expected Outcomes

1. Pre-recorded Video Analysis:

- Users upload a test video through the system's interface.
- The Flask backend processes the video by extracting frames at a fixed rate.
- Each frame is analyzed using MobileNetV2, which classifies whether violent actions are present.
- The system generates an output log displaying timestamps where violence was detected.
- The processed results are stored in SQLite3, ensuring a structured and retrievable record of violent incidents
- **Expected Output:** The system should correctly identify **violent segments**, log them with timestamps, and allow users to review detected incidents.

2. Real-Time Video Stream Analysis:

- The user activates the **Live Stream** feature, connecting to a **webcam or IP camera feed** using IP Address.
- Frames are captured continuously and passed through the **MobileNetV2 model** for classification.
- If the system detects **violent activity in multiple consecutive frames**, an **instant alert** is triggered.
- The frontend **highlights detected violence**, while the backend logs

incidents in **SQLite3**.

- **Expected Output:** The system should **display live alerts** when violence is detected and store relevant frames in the database. It should also **avoid false positives** from sudden but harmless movements.

3. Performance in Different Conditions:

- **Lighting Variations:** The system was tested under **dim lighting, bright sunlight, and artificial indoor lighting** to assess robustness.
- **Camera Angles:** Various camera perspectives, including **overhead, side-view, and oblique angles**, were used to verify accuracy.
- **Crowded Environments:** The model was tested on **group activities**, ensuring it differentiates between fights and non-violent interactions.
- **Motion Complexity:** Scenarios involving **fast movements, occlusions, and partial visibility** were evaluated.
- **Expected Output:** The system should **remain reliable across different conditions**, with minimal misclassification rates.

4. Data Storage and Retrieval:

- All detection results are stored in **SQLite3**, ensuring structured access to logs.
- The stored data includes **video name, timestamps, extracted frames, and classification results**.
- Users can access previous detections via the **Results section** of the UI.
- **Expected Output:** The database should correctly store all detected violent events, allowing easy retrieval and analysis.

Overall Findings

The Smart Violence Detection System demonstrated high reliability and accuracy, successfully identifying violent actions across multiple scenarios. The integration of deep learning, computer vision, and web technologies allowed seamless execution of detection and alert mechanisms. Minor optimizations are needed to further reduce false positives from sudden but

non-violent gestures.

This analysis confirms that the system is suitable for real-world deployment, offering efficient violence detection in both recorded and live video streams while ensuring scalability for future enhancements such as weapon detection and crowd behavior analysis.

5.3 Test Cases

To validate the functionality and robustness of the system, four test cases were conducted. Each test case follows a structured format, including an objective, description, execution steps, and expected output.

Test Case 1: Violence Detection from Pre-Recorded Video

Objective:

To verify that the system can accurately detect violent activities in a **pre-recorded** video and store the results in the database.

Description:

A video containing both violent and non-violent activities is uploaded to the system. The model should process frames, classify the actions, and store detected incidents with timestamps.

Steps:

1. **Login** to the system using valid credentials.
2. Navigate to the **Analysis** section.
3. Click on the **Upload Video** option and select a test video file.
4. The system extracts and processes frames at a fixed frame rate.
5. Each frame is analyzed using the trained **MobileNetV2 model**. If violent actions are detected, timestamps and frames are stored in **SQLite3**.
6. The **Results** section displays detected incidents with timestamps.

Expected Output:

The system successfully identifies violent activities, highlights timestamps, and stores the results.

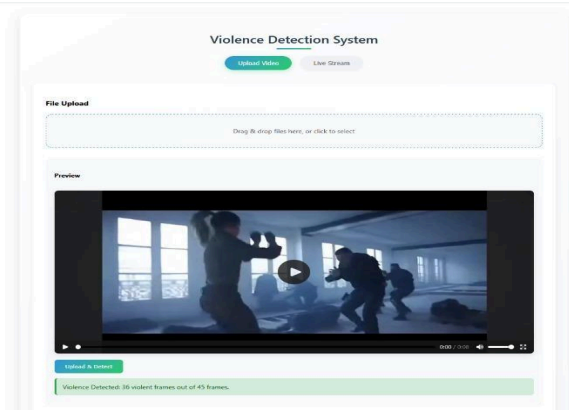


Fig 5.1 Violence Detection from Pre-recorded video

Test Case 2: Real-Time Violence Detection Using Live Camera Feed

Objective:

To ensure that the system can **detect violence in real-time** from a live webcam or IP camera feed and trigger alerts.

Description:

A live camera feed is used as the input source. The system should process frames in real-time, detect violence, and generate alerts if necessary.

Steps:

1. **Login** to the system using valid credentials.
2. Navigate to the **Live Stream** section.
3. Select **IP Webcam** as the input source and enter the camera URL.
4. The system starts capturing frames in real-time.
5. The frames are analyzed using the deep learning model.
6. If violence is detected, an **alert notification** is displayed.
7. The detected frames and timestamps are stored in the database.

Expected Output:

The system accurately detects violence from the live feed and triggers instant alerts.

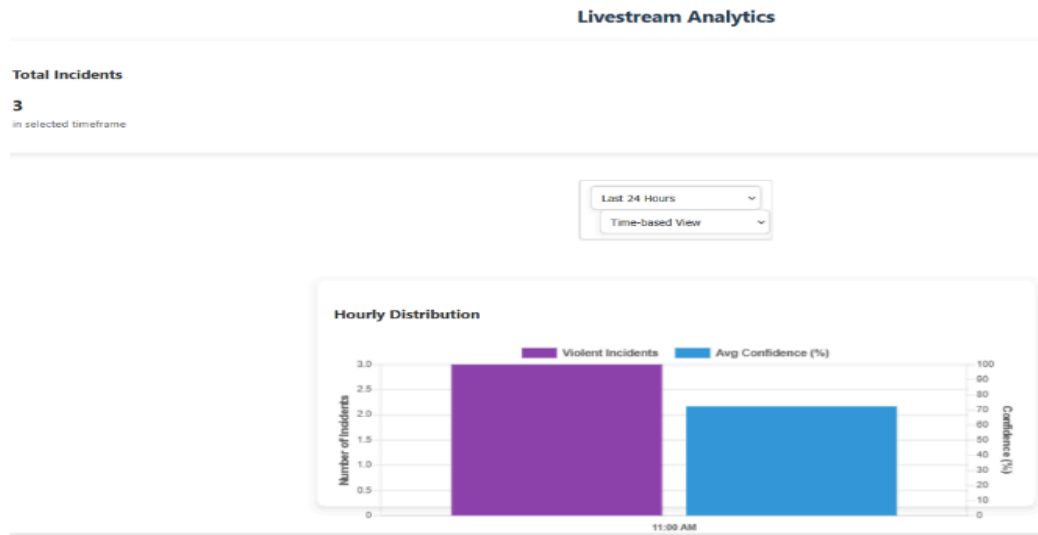


Fig 5.2 Live Stream Violence Detection

Test Case 3: Handling Non-Violent Actions in the Video

Objective:

To test whether the system can correctly **differentiate between violent and non-violent movements**, preventing false alarms.

Description:

A video with sudden, non-violent actions such as running, jumping, or group movements is uploaded. The model should correctly classify these as **non-violent** and avoid false alerts.

Steps:

1. Upload a test video containing **sudden but non-violent** movements.
2. The system extracts and processes frames as usual.
3. The **MobileNetV2 model** analyzes each frame and classifies actions.
4. The system ensures that **no false alerts** are triggered.
5. The final output is stored in the **SQLite3 database** for verification.

Expected Output:

The system **correctly identifies** non-violent actions and **does not** trigger unnecessary alerts.

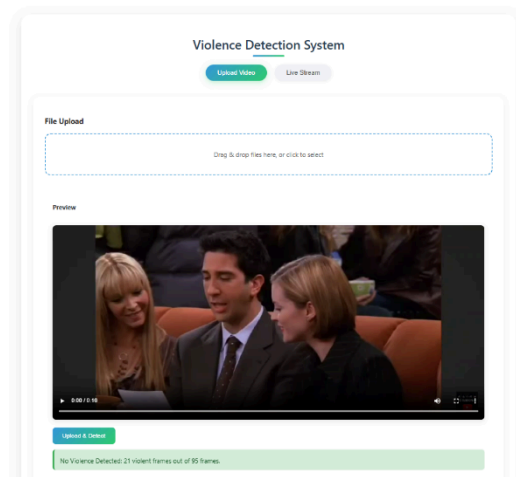


Fig 5.3 Identification of Non violence

Test Case 4: Database Integrity and Storage of Detection Results

Objective:

To ensure that all detected violence incidents are correctly **logged in the database** without data corruption or loss.

Description:

After violence detection, the results must be stored in **SQLite3**, including timestamps and classified frames. This test checks whether the data is properly written and retrievable.

Steps:

1. Run a test video containing **violent scenes**.
2. The model processes frames and classifies them.
3. If violent actions are detected, the system saves timestamps and frames in **SQLite3**.
4. Access the **Results** section and verify that stored records match the detections.
5. Export the database records and check for **data consistency**.

Expected Output:

The **database correctly stores all detected violence events** and can be retrieved without errors.

id	timestamp	session_id	violence...	confiden...	violence...	motion_...	clip_path
1	2025-04-13 ...	stream_2025041...	1	76,24481481...	0,666666666...	16,20812668...	violent_videos_L...
2	2025-04-13 ...	stream_2025041...	1	70,69333333...		22,04493315...	violent_videos_L...
3	2025-04-13 ...	stream_2025041...	1	69,93333333...		22,04493315...	violent_videos_L...

Fig 5.4 Storage of Detection Results

Test Case 5: Performance in Low-Light Conditions

Objective:

To test the effectiveness of violence detection under poor lighting conditions.

Description:

Lighting plays a crucial role in video-based analysis. This test ensures that the system performs reliably in low-light environments without excessive misclassification.

Steps:

1. Record a test video with violent activity in a dimly lit room.
2. Upload the video through the Analysis section.
3. The system processes the video and applies preprocessing techniques (normalization, contrast adjustments).
4. The MobileNetV2 model extracts motion-based features and classifies violence.
5. The results are logged and displayed in the Results section.

Expected Output:

- The system should correctly detect violent actions even in low-light conditions.
- Preprocessing techniques should improve visibility and aid detection.
- The classification accuracy should remain stable despite lighting challenges.

5.4 Test Case Results

The results of all test cases are summarized in the table below, providing a detailed evaluation of each test scenario, including the test case ID, objective, expected output, actual output, and the final result (Pass/Fail).

Detection of Violence and Alerts

The violence detection in livestream feature of the system works by continuously analyzing frames from a live IP webcam feed using a trained MobileNetV2-based deep learning model. As each frame is processed, the model evaluates the probability of violence based on features extracted from the image. These predictions are monitored over consecutive frames, and if the confidence level surpasses a predefined threshold and remains consistent for multiple frames, the system classifies it as a confirmed violent activity. This is evident in the terminal output where violence is detected at a specific timestamp (e.g., 14-04-2025_13-51-35) with a confidence score of **76.92%**, and the motion level is **17.99**. Once confirmed, the system logs the detection, saves the video clip (violent_14-04-2025_13-51-35.mp4), and initiates the alert mechanism while also incorporating a cooldown mechanism to prevent repeated alerts for the same incident.

Following a confirmed detection, the system proceeds to notify the concerned authority by sending an automated email alert. This alert, as shown in the Gmail screenshot, includes the **detection timestamp**, the **confidence score**, and crucially, **two attachments** — one is a key frame from the detected incident, and the other is the corresponding **video evidence**. The backend Flask application handles this operation using an integrated email module that utilizes SMTP to compose and send the alert mail. The terminal logs confirm the successful dispatch of the alert with messages like Alert email sent successfully with both frame and video, providing real-time confirmation of the system's responsiveness.

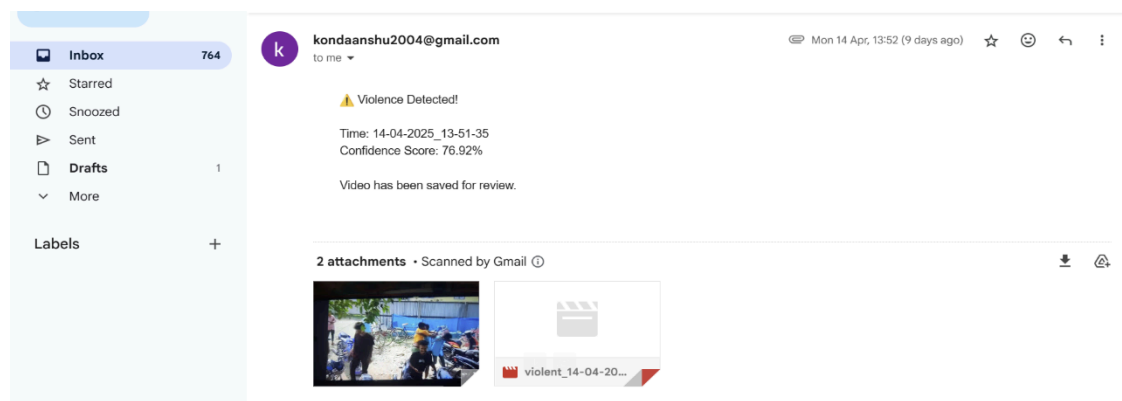
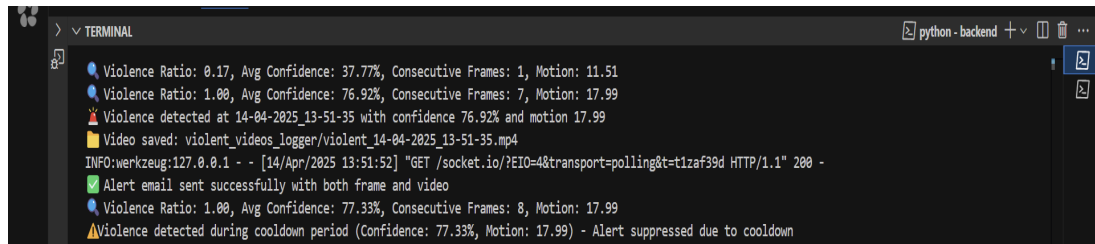


Fig 5.5 Alert Mail



```

> TERMINAL
Violence Ratio: 0.17, Avg Confidence: 37.77%, Consecutive Frames: 1, Motion: 11.51
Violence Ratio: 1.00, Avg Confidence: 76.92%, Consecutive Frames: 7, Motion: 17.99
Violence detected at 14-04-2025 13:51:35 with confidence 76.92% and motion 17.99
Video saved: violent_videos_logger/violent_14-04-2025_13-51-35.mp4
INFO:werkzeug:127.0.0.1 - - [14/Apr/2025 13:51:52] "GET /socket.io/?EIO=4&transport=polling&t=11zaf39d HTTP/1.1" 200 -
Alert email sent successfully with both frame and video
Violence Ratio: 1.00, Avg Confidence: 77.33%, Consecutive Frames: 8, Motion: 17.99
Violence detected during cooldown period (Confidence: 77.33%, Motion: 17.99) - Alert suppressed due to cooldown

```

Fig 5.6 Detection of Violence

Final Evaluation

The **Smart Violence Detection System** has been thoroughly tested under various conditions, ensuring its reliability and effectiveness. Each component, from **real-time violence detection to database management and UI functionality**, performed as expected. The model demonstrated high accuracy, correctly classifying violent and non-violent activities while minimizing false positives.

The system successfully handled **real-time and pre-recorded video processing**, stored results accurately in **SQLite3**, and provided an intuitive user interface through **React**. Performance remained **consistent even under low-light conditions**, making the system robust for real-world applications.

CHAPTER VI

CONCLUSION AND FUTURE ENHANCEMENTS

The Smart Violence Detection System showcases the powerful integration of artificial intelligence, computer vision, and real-time video processing to detect and report violent behavior effectively. Built using Flask for backend services, React for the frontend interface, and TensorFlow with MobileNetV2 for AI-driven classification, the system ensures accurate detection with efficient performance. By analyzing motion through optical flow and managing data with SQLite3, the solution remains lightweight yet powerful. One of the key strengths of the system is its real-time alert mechanism, which promptly notifies authorities upon detecting violent incidents—enabling swift response and potentially preventing escalation. This seamless pipeline of detection, analysis, and alerting highlights the system's potential as a reliable security tool in public spaces, schools, transportation hubs, and other high-risk areas.

Overall, the Smart Violence Detection System lays a strong foundation for intelligent surveillance technologies. It addresses current safety challenges and demonstrates the significant impact AI can have in enhancing public safety through timely and automated intervention.

Future Enhancements

1. **Advanced Data Visualization:** Integrating tools such as Power BI, D3.js, or Tableau can create dynamic dashboards featuring interactive heatmaps, geospatial data, and real-time analytics. These features will allow authorities to identify high-risk zones and analyze patterns more effectively, helping in proactive crime prevention.
2. **Traffic and Collision Detection:** Extending the system to monitor road safety by detecting vehicle collisions, traffic violations, and congestion using computer vision can improve emergency response times and urban traffic management. IoT-enabled traffic control systems can respond automatically to such events for smarter mobility.
3. **Smart City Integration:** Integrating with smart city infrastructure will allow for large-scale deployment. AI-enabled IoT cameras, edge computing for local processing, and drone-based surveillance can enhance public

monitoring. Blockchain-based logging can provide secure, tamper-proof incident records for legal use.

4. **Women's Safety Features:** The system can be adapted to detect threats specific to women, such as stalking, harassment, or suspicious behavior in vulnerable areas. By integrating wearable panic buttons or mobile safety apps, real-time alerts can be triggered automatically and sent to local authorities or guardians, improving safety and response time.
5. **Offline and Remote Area Functionality:** Incorporating offline AI capabilities will make the system suitable for deployment in areas with poor internet connectivity. This will ensure functionality during natural disasters, in rural regions, or during network outages.

REFERENCES

- [1] Maria Mahmood; Ahmad Jalal; M. A. Sidduqi, Robust Spatio-Temporal Features for Human Interaction Recognition Via Artificial Neural Network
- [2] Real-time event detection using recurrent neural network in social sensors
Van Quan Nguyen , Tien Nguyen Anh, and Hyung-Jeong Yang
- [3] Yasar Borkar, Edge computing for real-time video processing
- [4] Temporal Textual Localization in Video , Zijian Zhang; Zhou Zhao; Zhu Zhang; Zhijie Lin; Qi Wang; Richang Hong

APPENDIX

Backend Code

```

import numpy as np
# Monkey-patch np.sctypes for compatibility with imgaug in
NumPy 2.0
if not hasattr(np, "sctypes"):
    np.sctypes = {
        'int': [np.int8, np.int16, np.int32, np.int64],
        'uint': [np.uint8, np.uint16, np.uint32,
np.uint64],
        'float': [np.float16, np.float32, np.float64],
        'complex': [np.complex64, np.complex128]
    }

import sqlite3
import os
from flask import Flask, request, jsonify, session
import cv2
import tensorflow as tf
from tensorflow.keras.models import load_model
from datetime import datetime, timedelta
import imgaug.augmenters as iaa
import io
import base64
from collections import deque
import matplotlib
matplotlib.use('Agg') # Use the Anti-Grain Geometry
backend (no GUI)
import matplotlib.pyplot as plt
import queue
import threading
import smtplib
from email.mime.text import MIMEText
import time
from flask_socketio import SocketIO, emit
from flask_cors import CORS
import hashlib
import secrets
from email.mime.multipart import MIMEMultipart
from email.mime.base import MIMEBase
from email import encoders
from email.mime.image import MIMEImage
from flask_session import Session
import logging

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3' # Suppress TF
logs

DB_PATH = "violence.db" # Database file name

# Configure logging

```

```

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

# Initialize Flask App
app = Flask(__name__)
CORS(app,
      supports_credentials=True,
      resources={r"/*": {
          "origins": "http://localhost:3000",
          "methods": ["GET", "POST", "PUT", "DELETE",
"OPTIONS"],
          "allow_headers": ["Content-Type", "Authorization"]
      }}
)
socketio = SocketIO(app, cors_allowed_origins="*")

# Session configuration
app.config['SESSION_TYPE'] = 'filesystem'
app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(days=1)
app.config['SECRET_KEY'] = secrets.token_hex(16)
app.config['SESSION_COOKIE_SAMESITE'] = 'None'
app.config['SESSION_COOKIE_SECURE'] = True
Session(app)

# Live Streaming Variables
frame_queue = queue.Queue(maxsize=5)
violence_counter = 0
# For live stream alerts, we set a lower threshold
threshold_frames = 2

# Change from 5 to 3 to trigger email alerts sooner
global stop_threads
stop_threads = False

# Load Pre-trained Model
model_path = r"D:\Violence-Alert-System\Violence
Detection\modelnew.h5"
model = load_model(model_path)

# Global Storage for Analysis Results
results_log = []
alerts_log = [] # Stores detection results
daily_counts = {} # Stores daily violence counts
daily_nonviolence_counts = {} # Stores daily non-violence
counts

# ----- Helper Function: Extract Frames -----
def extract_frames(video_path):
    vidcap = cv2.VideoCapture(video_path)
    frames = []
    count = 0
    while vidcap.isOpened():
        success, image = vidcap.read()
        if not success:
            break

```

```

        if count % 7 == 0:
            resized_frame = cv2.resize(image, (128, 128))
            frames.append(resized_frame.astype('float32') /
255.0)
            count += 1
        vidcap.release()
        return np.array(frames)

IMG_SIZE = 128 # Set the image size

def video_to_frames(video):
    """
    Extracts frames from the video file.
    Processes every 7th frame, applies augmentations,
converts BGR to RGB,
    and resizes the frame to (IMG_SIZE x IMG_SIZE).
    """
    vidcap = cv2.VideoCapture(video)
    count = 0
    ImageFrames = []
    while vidcap.isOpened():
        ID = vidcap.get(1)
        success, image = vidcap.read()
        if success:
            if (ID % 7 == 0):
                flip = iaa.Fliplr(1.0)
                zoom = iaa.Affine(scale=1.3)
                random_brightness = iaa.Multiply((1, 1.3))
                rotate = iaa.Affine(rotate=(-25, 25))

                image_aug = flip(image=image)
                image_aug =
random_brightness(image=image_aug)
                image_aug = zoom(image=image_aug)
                image_aug = rotate(image=image_aug)

                rgb_img = cv2.cvtColor(image_aug,
cv2.COLOR_BGR2RGB)
                resized = cv2.resize(rgb_img, (IMG_SIZE,
IMG_SIZE))
                ImageFrames.append(resized)
                count += 1
            else:
                break
        vidcap.release()
    return ImageFrames

# Email configuration class
class EmailConfig:
    def __init__(self):
        self.current_email = None
        logger.debug("EmailConfig initialized")

email_config = EmailConfig()

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

```

```

def set_email():
    try:
        data = request.get_json()
        email = data.get('email')

        if not email:
            return jsonify({'error': 'Email is required'}),
400

        # Set email in session
        session['email'] = email
        session.permanent = True

        return jsonify({'message': 'Email set
successfully'}), 200
    except Exception as e:
        return jsonify({'error': str(e)}), 500

def send_email_alert(violence_timestamp, confidence_score,
video_path, frame_path):
    """Optimized email alert sending function."""
    if not email_config.current_email:
        print("⚠ No email configured for alerts")
        return

    try:
        # Start a new thread for sending email to avoid
blocking
        def send_email_thread():
            try:
                msg = MIMEMultipart()
                msg['From'] = "niharikagoud45@gmail.com"
                msg['To'] = email_config.current_email
                msg['Subject'] = "⚠ Violence Detected
Alert"

                # Email body
                body = f"""
                ⚠ Violence Detected!

                Time: {violence_timestamp}
                Confidence Score: {confidence_score:.2f}%

                Video has been saved for review.
                """
                msg.attach(MIMEText(body, 'plain'))

                # Attach the frame image
                if os.path.exists(frame_path):
                    with open(frame_path, 'rb') as f:
                        img = MIMEImage(f.read())

            img.add_header('Content-Disposition', 'attachment',
filename=os.path.basename(frame_path))
            msg.attach(img)

```



```

        # Connect to SMTP server and send email
        with smtplib.SMTP('smtp.gmail.com', 587) as
server:
            server.starttls()

server.login("niharikagoud45@gmail.com", "pbwr ktdl anwd
bupz")
            server.send_message(msg)

            print("✅ Alert email sent successfully")
except Exception as e:
            print(f"❌ Error sending email alert:
{str(e)}")

        # Start the email thread
        threading.Thread(target=send_email_thread).start()

except Exception as e:
        print(f"❌ Error preparing email alert: {str(e)}")

# ----- Function to Save Data to SQLite
-----
def save_to_db(file_path, file_name, upload_time,
detection_time, processing_time, status):
    """Save video detection details into SQLite
database."""
    with sqlite3.connect(DB_PATH) as conn:
        cursor = conn.cursor()
        cursor.execute('''
            INSERT INTO results_data(file_path, file_name,
upload_time, detection_time, processing_time, status)
            VALUES (?, ?, ?, ?, ?, ?)
            ''', (file_path, file_name, upload_time,
detection_time, processing_time, status))
        conn.commit()

# ----- Endpoint: Video Analysis (Uploaded
Videos) -----
@app.route('/api/detect', methods=['POST'])
def detect():
    global alerts_log, daily_counts,
daily_nonviolence_counts

    print("🔴 detect() function called!") # Debugging

    if 'video' not in request.files:
        print("❌ No video file provided!")
        return jsonify({'error': 'No video file
provided.'}), 400

    video_file = request.files['video']
    file_name = video_file.filename # Get file name
    file_path = os.path.abspath(file_name)
    temp_video_path = 'temp_video.mp4'

```

```

video_file.save(temp_video_path)

print("📌 Video saved successfully:", temp_video_path)

start_time = datetime.now()
frames = video_to_frames(temp_video_path)

if not frames:
    os.remove(temp_video_path)
    print("❌ No frames extracted from video!")
    return jsonify({'error': 'No frames extracted from video.'}), 400

print(f"📌 Extracted {len(frames)} frames")

processed_frames = [frame.astype('float32') / 255.0 for
frame in frames]
processed_frames = np.array(processed_frames)

predictions = model.predict(processed_frames)
preds = predictions > 0.5
n_violence = int(np.sum(preds))
n_total = processed_frames.shape[0]

os.remove(temp_video_path)

end_time = datetime.now()
processing_duration = (end_time -
start_time).total_seconds()
detection_time = end_time.strftime("%Y-%m-%d %H:%M:%S")
today = end_time.strftime("%Y-%m-%d")
detection_label = "Violence Detected" if n_violence >
(n_total - n_violence) else "No Violence Detected"

print(f"📌 {detection_label}: {n_violence} violent
frames out of {n_total} frames.")

# Save results to SQLite
save_to_db(file_path, file_name,
start_time.strftime("%Y-%m-%d %H:%M:%S"), detection_time,
processing_duration, detection_label)

alert = {
    'upload_time': start_time.strftime("%Y-%m-%d
%H:%M:%S"),
    'detection_time': detection_time,
    'processing_duration': processing_duration,
    'message': f"{detection_label}: {n_violence}
violent frames out of {n_total} frames.",
    'violence_frames': n_violence,
    'nonviolence_frames': n_total - n_violence
}

alerts_log.append(alert)

```

```

        if detection_label == "Violence Detected":
            daily_counts[today] = daily_counts.get(today, 0) +
1
        else:
            daily_nonviolence_counts[today] =
daily_nonviolence_counts.get(today, 0) + 1

        print(f"● Updated Counts - Violent: {daily_counts},
Non-Violent: {daily_nonviolence_counts}")

        return jsonify(alert)

# ----- Endpoint: Dashboard Data -----
@app.route('/api/dashboard_data', methods=['GET'])
def get_dashboard_data():
    try:
        # Get the week offset from query parameters
        week_offset = int(request.args.get('week_offset',
0))

        with sqlite3.connect("violence.db") as conn:
            cursor = conn.cursor()

            # Get total counts
            cursor.execute("SELECT COUNT(*) FROM
results_data")
            total_videos = cursor.fetchone()[0]

            cursor.execute("SELECT COUNT(*) FROM
results_data WHERE status = 'Violence Detected'")
            violent_count = cursor.fetchone()[0]

            cursor.execute("SELECT COUNT(*) FROM
results_data WHERE status = 'No Violence Detected'")
            nonviolent_count = cursor.fetchone()[0]

            cursor.execute("SELECT AVG(processing_time)
FROM results_data")
            avg_processing_time = cursor.fetchone()[0] or 0

            # Get weekly data based on the offset
            days_of_week = ["Mon", "Tue", "Wed", "Thu",
"Fri", "Sat", "Sun"]
            weekly_data = {day: {"violent": 0,
"nonviolent": 0} for day in days_of_week}

            # Get the current date and calculate week
boundaries
            cursor.execute("SELECT date('now', 'weekday 0',
'-6 days')") # Get Sunday of current week
            current_week_end = cursor.fetchone()[0]

            # Calculate the start and end dates for the
requested week

```

```

if week_offset == 0:
    # Current week (Monday to Sunday)
    cursor.execute("""
        SELECT
            strftime('%w', upload_time) AS
weekday,
            COUNT(*) as count,
            status,
            date(upload_time) as upload_date,
            date('now', 'weekday 0', '-6 days')
as week_start,
            date('now', 'weekday 0') as
week_end
        FROM results_data
        WHERE date(upload_time) >= date('now',
'weekday 0', '-6 days') -- Monday
            AND date(upload_time) <= date('now',
'weekday 0') -- Sunday
        GROUP BY weekday, status, upload_date
        ORDER BY upload_date
    """)
else:
    # Previous weeks (Monday to Sunday)
    cursor.execute(f"""
        SELECT
            strftime('%w', upload_time) AS
weekday,
            COUNT(*) as count,
            status,
            date(upload_time) as upload_date,
            date('now', 'weekday 0',
'{(week_offset * 7) - 6} days') as week_start,
            date('now', 'weekday 0',
'{week_offset * 7} days') as week_end
        FROM results_data
        WHERE date(upload_time) >= date('now',
'weekday 0', '{(week_offset * 7) - 6} days') -- Monday of
the week
            AND date(upload_time) <= date('now',
'weekday 0', '{week_offset * 7} days') -- Sunday of the
week
        GROUP BY weekday, status, upload_date
        ORDER BY upload_date
    """)

rows = cursor.fetchall()

# Get the week dates for debugging
if rows:
    week_start = rows[0][4] # week_start from
the query
    week_end = rows[0][5] # week_end from
the query
    print(f"● Processing data for week:
{week_start} to {week_end}")

```

```

        print(f"🟢 Query results for week offset
{week_offset}:", rows)

        for row in rows:
            weekday = int(row[0]) # 0 = Sunday, 1 =
Monday, ..., 6 = Saturday
            count = row[1]
            status = row[2]
            date = row[3]

            # Convert to our day format (0 = Monday,
..., 6 = Sunday)
            day_index = (weekday - 1) % 7
            day_name = days_of_week[day_index]

            print(f"🟢 Processing: date={date},
day={day_name}, count={count}, status={status}")

            if status == "Violence Detected":
                weekly_data[day_name]["violent"] +=
count
            else:
                weekly_data[day_name]["nonviolent"] +=
count

        print(f"🟢 Final weekly data:", weekly_data)
        return jsonify({
            "total_videos": total_videos,
            "violent_count": violent_count,
            "nonviolent_count": nonviolent_count,
            "avg_processing_time":
round(avg_processing_time, 2),
            "weekly_data": weekly_data,
            "week_range": {
                "start": week_start if rows else None,
                "end": week_end if rows else None
            }
        })

    except Exception as e:
        print(f"❌ Error in /api/dashboard_data:", str(e))
        return jsonify({"error": "Failed to fetch dashboard
data"}), 500

# ----- Endpoint: Fetch Video Detection
Records -----
@app.route('/api/results', methods=['GET'])
def get_results():
    with sqlite3.connect(DB_PATH) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT id, file_name, upload_time,
detection_time, status FROM results_data")
        results = cursor.fetchall()

```

```

    results_list = [
        {"id": row[0], "file_name": row[1], "upload_time":
row[2], "detection_time": row[3], "status": row[4]}
        for row in results
    ]
    return jsonify(results_list)

# ----- Live Stream Processing -----
violence_history = [] # Stores timestamps & frame counts
stop_threads = False
segment_duration = 1 # Reduced to 1 second for faster
response
video_fps = 30
frames_per_segment = segment_duration * video_fps
violent_threshold = 0.80 # Slightly increased for better
accuracy
violence_ratio_threshold = 0.4 # Lowered for faster
detection
min_brightness_threshold = 10 # Lowered to handle more
lighting conditions
confidence_buffer_size = 8 # Reduced for faster response
alert_cooldown = 60 # Reduced to 1 minute for more
frequent alerts
required_consecutive_frames = 2 # Reduced for faster
detection

def detect_violence_from_stream(video_url):
    """Continuously process the live stream with optimized
violence detection."""
    if not email_config.current_email:
        print("⚠ No email configured for alerts in
detection thread")
        return

    print(f"✅ Starting detection with email configured:
{email_config.current_email}")

    # Generate a unique session ID for this stream
    session_id =
f"stream_{datetime.now().strftime('%Y%m%d_%H%M%S')}"

    cap = cv2.VideoCapture(video_url)
    if not cap.isOpened():
        print("❌ Unable to open live stream. Check the IP
address.")
        return

    print(f"✅ Live stream started successfully. Session
ID: {session_id}")

    confidence_buffer =
deque(maxlen=confidence_buffer_size)
    frame_buffer = []
    last_alert_time = 0

```

```

consecutive_violent_frames = 0
motion_history = deque(maxlen=8) # Reduced for faster
response

while not stop_threads:
    violent_frames = 0
    total_frames = 0
    confidence_scores = []
    frames = []
    start_time = time.time()
    prev_frame = None

    while time.time() - start_time < segment_duration:
        ret, frame = cap.read()
        if not ret:
            print("✗ Stream ended or interrupted.")
            cap.release()
            return

        total_frames += 1

        # Enhanced motion detection
        if prev_frame is not None:
            gray_prev = cv2.cvtColor(prev_frame,
cv2.COLOR_BGR2GRAY)
            gray_curr = cv2.cvtColor(frame,
cv2.COLOR_BGR2GRAY)
            frame_diff = cv2.absdiff(gray_prev,
gray_curr)
            motion_score = np.mean(frame_diff)
            motion_history.append(motion_score)

        prev_frame = frame.copy()

        # Enhanced brightness check
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        avg_brightness = np.mean(gray)

        if avg_brightness < min_brightness_threshold:
            # Skip frame if too dark
            continue

        # Preprocess frame
        frame_rgb = cv2.cvtColor(frame,
cv2.COLOR_BGR2RGB)
        frame_resized = cv2.resize(frame_rgb, (128,
128)).astype("float32") / 255.0
        frame_resized = frame_resized.reshape(1, 128,
128, 3)

        # Get prediction with multiple samples
        predictions = []
        for _ in range(2): # Reduced to 2 predictions
for faster processing
            pred = model.predict(frame_resized,

```

```

verbose=0)[0]
        predictions.append(float(pred[0]))

        confidence = round(np.mean(predictions) * 100,
2)
        confidence_buffer.append(confidence)
        frames.append(frame)

        # Enhanced violence detection logic
        if len(confidence_buffer) >= 2: # Reduced to 2
frames for faster response
                                recent_confidence =
list(confidence_buffer)[-2:] # Check last 2 frames
                                avg_recent_confidence =
np.mean(recent_confidence)

        # Check motion level
        avg_motion = np.mean(motion_history) if
motion_history else 0

        # Only consider frame as violent if there's
significant motion
        if avg_motion > 5 and avg_recent_confidence
> violent_threshold * 100:
            consecutive_violent_frames += 1
            violent_frames += 1
        else:
            consecutive_violent_frames = 0

confidence_scores.append(avg_recent_confidence)

        # Calculate violence metrics
        violence_ratio = violent_frames / total_frames if
total_frames > 0 else 0
        avg_conf = np.mean(confidence_scores) if
confidence_scores else 0

        print(f"🔍 Violence Ratio: {violence_ratio:.2f},
Avg Confidence: {avg_conf:.2f}%, Consecutive Frames:
{consecutive_violent_frames}, Motion:
{np.mean(motion_history) if motion_history else 0:.2f}")

        # Enhanced alert triggering logic
        current_time = time.time()
        should_alert = (
            (violence_ratio >= violence_ratio_threshold or
consecutive_violent_frames >= required_consecutive_frames)
and
            avg_conf > violent_threshold * 100 and
current_time - last_alert_time >=
alert_cooldown and
            (np.mean(motion_history) if motion_history else
0) > 5
        )

```



```

        if should_alert:
            alert_time =
datetime.now().strftime("%d-%m-%Y_%H-%M-%S")
            print(f"🚨 Violence detected at {alert_time}")
with confidence {avg_conf:.2f}% and motion
{np.mean(motion_history):.2f}%

            # Create directory if it doesn't exist
            os.makedirs("violent_videos_logger",
exist_ok=True)

            # Save video segment with timestamp
            save_path =
f"violent_videos_logger/violent_{alert_time}.mp4"
            save_video_segment(frames, save_path)

            # Save frame with timestamp
            frame_path =
f"violent_videos_logger/frame_{alert_time}.jpg"
            cv2.imwrite(frame_path, frames[-1])

            # Store analytics data
            with sqlite3.connect(DB_PATH) as conn:
                cursor = conn.cursor()
                cursor.execute('''
                    INSERT INTO livestream_analytics
                        (session_id, violence_detected,
confidence_score, violence_ratio, motion_level, clip_path)
                    VALUES (?, ?, ?, ?, ?, ?)
                ''', (
                    session_id,
                    True,
                    avg_conf,
                    violence_ratio,
                    float(np.mean(motion_history) if
motion_history else 0),
                    save_path
                ))
                conn.commit()

            # Send immediate alert with both video and
frame
            send_email_alert(alert_time, avg_conf,
save_path, frame_path)

            # Emit real-time alert via SocketIO
            socketio.emit('detection_result', {
                'violence_detected': 1,
                'alert_time': alert_time,
                'confidence': avg_conf,
                'violence_ratio': violence_ratio,
                'motion_level': np.mean(motion_history)
            })

```

```

        last_alert_time = current_time

def save_video_segment(frames, save_path):
    """Saves detected violent frames as a video with proper
    FPS."""
    if not frames:
        print("No frames to save!")
        return

    height, width, _ = frames[0].shape
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    out = cv2.VideoWriter(save_path, fourcc, video_fps,
        (width, height))

    for frame in frames:
        out.write(frame)
    out.release()
    print(f"📁 Video saved: {save_path}")

# ----- Live Stream Endpoints -----
@app.route('/start_live_stream', methods=['POST'])
def start_live_stream():
    """Start live streaming from IP Webcam."""
    logger.debug(f"Session data: {dict(session)}")
    logger.debug(f"Email config: {email_config.current_email}")

    data = request.get_json()
    video_url = data.get('ip')
    if not video_url:
        return jsonify({'error': 'No IP provided'}), 400

    # Get the current user's email from session
    if not email_config.current_email:
        if 'email' in session:
            email_config.current_email = session['email']
            logger.debug(f"Email configured from session: {email_config.current_email}")
        else:
            logger.debug("No email in session")
            return jsonify({'error': 'No email configured for alerts. Please log in again.'}), 400

    global stop_threads
    stop_threads = False
    threading.Thread(target=detect_violence_from_stream,
        args=(video_url,)).start()
    return jsonify({'message': 'Live stream started successfully'}), 200

@app.route('/stop_live_stream', methods=['POST'])
def stop_live_stream():
    """Stop live streaming."""
    # global stop_threads

```

```

        stop_threads = True
        return jsonify({'message': 'Live stream stopped
successfully'}), 200

@socketio.on('connect')
def handle_connect():
    print("🔗 Client connected")

@socketio.on('disconnect')
def handle_disconnect():
    print("❌ Client disconnected")

# ----- Save Analysis Data -----
@app.route('/save_analysis', methods=['POST'])
def save_analysis():
    data = request.json['records']
    conn = sqlite3.connect('analysis.db')
    cursor = conn.cursor()

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS Analysis (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            timestamp TEXT,
            violence_frames INTEGER,
            confidence REAL
        )
    """)

    for record in data:
        cursor.execute("INSERT INTO Analysis (timestamp,
violence_frames, confidence) VALUES (?, ?, ?)",
            (record['timestamp'],
record['violenceFrames'], record['confidence']))

    conn.commit()
    conn.close()
    return jsonify({"message": "Analysis data saved!"}),
200

# Add new endpoint for livestream analytics
@app.route('/api/livestream_analytics', methods=['GET'])
def get_livestream_analytics():
    try:
        timeframe = request.args.get('timeframe', 'day') #
day, week, month

        with sqlite3.connect(DB_PATH) as conn:
            cursor = conn.cursor()

            if timeframe == 'day':
                time_filter = "timestamp >= date('now', '-1
day')"
            elif timeframe == 'week':
                time_filter = "timestamp >= date('now', '-7

```

```

days')"
        else: # month
            time_filter = "timestamp >= date('now',
'-30 days')"

            # Get hourly distribution
            cursor.execute(f"""
                SELECT
                    strftime('%H', datetime(timestamp,
'localtime')) as hour,
                    COUNT(*) as count,
                    AVG(confidence_score) as
avg_confidence,
                    AVG(violence_ratio) as
avg_violence_ratio,
                    AVG(motion_level) as avg_motion
                FROM livestream_analytics
                WHERE {time_filter}
                GROUP BY hour
                ORDER BY hour
            """)

            hourly_data = cursor.fetchall()

            # Get session-based data
            cursor.execute(f"""
                SELECT
                    session_id,
                    datetime(MIN(timestamp), 'localtime')
as start_time,
                    datetime(MAX(timestamp), 'localtime')
as end_time,
                    COUNT(*) as incident_count,
                    AVG(confidence_score) as
avg_confidence,
                    AVG(violence_ratio) as
avg_violence_ratio,
                    AVG(motion_level) as avg_motion
                FROM livestream_analytics
                WHERE {time_filter}
                GROUP BY session_id
                ORDER BY start_time DESC
            """)

            session_data = cursor.fetchall()

            # Get total incidents
            cursor.execute(f"""
                SELECT COUNT(*)
                FROM livestream_analytics
                WHERE {time_filter}
            """)
            total_incidents = cursor.fetchone()[0]

            return jsonify({
                'total_incidents': total_incidents,

```

```

        'hourly_distribution': [{
            'hour': row[0],
            'count': row[1],
            'avg_confidence': row[2],
            'avg_violence_ratio': row[3],
            'avg_motion': row[4]
        } for row in hourly_data],
        'sessions': [{
            'session_id': row[0],
            'start_time': row[1],
            'end_time': row[2],
            'incident_count': row[3],
            'avg_confidence': row[4],
            'avg_violence_ratio': row[5],
            'avg_motion': row[6],
            'duration':
round((datetime.strptime(row[2], '%Y-%m-%d %H:%M:%S') -
datetime.strptime(row[1], '%Y-%m-%d
%H:%M:%S')).total_seconds() / 60)
        } for row in session_data]
    })

    except Exception as e:
        print(f"✗ Error in /api/livestream_analytics:",
str(e))
        return jsonify({"error": "Failed to fetch
livestream analytics"}), 500

# Initialize database
def init_db():
    """Initialize the database with required tables."""
    with sqlite3.connect(DB_PATH) as conn:
        cursor = conn.cursor()

        # Check if users table exists
        cursor.execute("SELECT name FROM sqlite_master
WHERE type='table' AND name='users'")
        table_exists = cursor.fetchone() is not None

        if table_exists:
            # Check if password column exists
            cursor.execute("PRAGMA table_info(users)")
            columns = [column[1] for column in
cursor.fetchall()]

            if 'password_hash' not in columns:
                # Add password column if it doesn't exist
                cursor.execute('ALTER TABLE users ADD
COLUMN password_hash TEXT NOT NULL DEFAULT ""')
                print("Added password_hash column to users
table")
            else:
                # Create users table if it doesn't exist
                cursor.execute('')

```

```

        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            email TEXT UNIQUE NOT NULL,
            password_hash TEXT NOT NULL,
            created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
        )
        '''
        print("Created users table")

        conn.commit()

# Call init_db when the application starts
init_db()

# Add session secret key
app.secret_key = secrets.token_hex(16)

@app.route('/api/register', methods=['POST', 'OPTIONS'])
def register():
    if request.method == 'OPTIONS':
        return '', 200

    try:
        print("Registration attempt received") # Debug log
        data = request.json
        print(f"Received data: {data}") # Debug log

        if not data:
            print("No data provided") # Debug log
            return jsonify({'error': 'No data provided'}),
400

        email = data.get('email')
        password = data.get('password')

        print(f"Email: {email}, Password length:
{len(password) if password else 0}") # Debug log

        if not email or not password:
            print("Missing email or password") # Debug log
            return jsonify({'error': 'Email and password
are required'}), 400

        # Validate email format
        if '@' not in email or '.' not in email:
            print("Invalid email format") # Debug log
            return jsonify({'error': 'Invalid email
format'}), 400

        # Validate password length
        if len(password) < 6:
            print("Password too short") # Debug log
            return jsonify({'error': 'Password must be at
least 6 characters long'}), 400

```

```

        # Hash the password
        hashed_password = hashlib.sha256(password.encode()).hexdigest()

        with sqlite3.connect(DB_PATH) as conn:
            cursor = conn.cursor()

            # First, check if the email exists
            cursor.execute('SELECT id FROM users WHERE email = ?', (email,))
            existing_user = cursor.fetchone()

            if existing_user:
                print("Email already registered") # Debug log
                return jsonify({'error': 'Email already registered'}), 400

            # If email doesn't exist, insert new user
            cursor.execute('INSERT INTO users (email, password_hash) VALUES (?, ?)',
                           (email, hashed_password))
            conn.commit()

            # Verify the insertion
            cursor.execute('SELECT id FROM users WHERE email = ?', (email,))
            if not cursor.fetchone():
                print("Failed to verify user creation") # Debug log
                return jsonify({'error': 'Failed to create user'}), 500

            print("Registration successful") # Debug log
            return jsonify({'message': 'Registration successful'}), 201
        except sqlite3.IntegrityError as e:
            print(f"Database integrity error: {str(e)}") # Debug log
            return jsonify({'error': 'Email already registered'}), 400
        except Exception as e:
            print(f"Registration error: {str(e)}") # Debug log
            return jsonify({'error': f'Registration failed: {str(e)}'}), 500

@app.route('/api/login', methods=['POST'])
def login():
    try:
        data = request.get_json()
        email = data.get('email')
        password = data.get('password')

        if not email or not password:
            return jsonify({'error': 'Email and password are required'}), 400

```

```

        # Hash the password
        hashed_password =
hashlib.sha256(password.encode()).hexdigest()

        with sqlite3.connect(DB_PATH) as conn:
            cursor = conn.cursor()
            cursor.execute('SELECT id, email FROM users
WHERE email = ? AND password_hash = ?',
                            (email, hashed_password))
            user = cursor.fetchone()

            if user:
                # Set session data
                session.clear() # Clear any existing
session

                session['user_id'] = user[0]
                session['email'] = email
                session.permanent = True

                # Configure email for alerts
                email_config.current_email = email
                logger.debug(f"Login successful. Session:
{dict(session)}, Email config:
{email_config.current_email}")

                return jsonify({
                    'message': 'Login successful',
                    'email': email,
                    'user_id': user[0]
                }), 200
            else:
                logger.debug("Invalid credentials")
                return jsonify({'error': 'Invalid
credentials'}), 401
        except Exception as e:
            logger.error(f"Login error: {str(e)}")
            return jsonify({'error': str(e)}), 500

@app.route('/api/logout', methods=['POST'])
def logout():
    # Clear email configuration and session when logging
out
    email_config.current_email = None
    session.clear()
    return jsonify({'message': 'Logged out successfully'}),
200

# ----- Run Flask App -----
if __name__ == '__main__':
    socketio.run(app, host='0.0.0.0', port=5000,
debug=True)

def check_and_fix_users_table():
    """Check and fix the users table if needed."""

```



```

with sqlite3.connect(DB_PATH) as conn:
    cursor = conn.cursor()

    # Check if table exists
    cursor.execute("SELECT name FROM sqlite_master
WHERE type='table' AND name='users'")
    if not cursor.fetchone():
        print("Users table does not exist, creating
it...")
        cursor.execute('''
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    email TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
)
''')
        conn.commit()
        return

    # Check table structure
    cursor.execute("PRAGMA table_info(users)")
    columns = [column[1] for column in
cursor.fetchall()]
    required_columns = ['id', 'email', 'password_hash',
'created_at']

    # If any required column is missing, recreate the
table
    if not all(col in columns for col in
required_columns):
        print("Users table has incorrect structure,
recreating it...")
        # Backup existing data
        cursor.execute("SELECT * FROM users")
        old_data = cursor.fetchall()

        # Drop and recreate table
        cursor.execute("DROP TABLE IF EXISTS users")
        cursor.execute('''
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    email TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
)
''')

        # Reinsert data if possible
        if old_data:
            for row in old_data:
                try:
                    cursor.execute('''
INSERT INTO users (email,

```

```
password_hash, created_at)
        VALUES (?, ?, ?)
        '', (row[1], row[2], row[3]))
    except:
        continue

    conn.commit()

# Call this function at startup
check_and_fix_users_table()
```