Artificial Intelligence & Expert Systems

(CT-361)

COMPLEX COMPUTING PROBLEM

Muhammad Akbar CT-22053

Sumaiyya Farhat CT-22054

Musab Umair CT-22095

CSIT – B

Submitted to: Mr. Muhammad Abdullah Siddiqui

Department of Computer Science and Information Technology

DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY
BACHELORS OF SCIENCE IN COMPUTER SCIENCE

Complex Computing Problem Assessment Rubrics

Course Code: CT-361		**Course Title: Artificial Intelligence & Expert System**	
Criteria and Scales			
Excellent (3)	**Good (2)**	**Average (1)**	**Poor (0)**
Criterion 1: Understanding the Problem: How well the problem statement is understood by the student			
Understand the problem clearly and identify the underlying issues and functionalities.	Adequately understands the problem and identifies the underlying issues and functionalities.	Inadequately defines the problem and identifies the underlying issues and functionalities.	Fails to define the problem adequately and does not identify the underlying issues and functionalities.
Criterion 2: Research: The amount of research that is used in solving the problem			
Contains all the information needed for solving the problem	Good research leads to a successful solution	Mediocre research which may or may not lead to an adequate solution	No apparent research
Criterion 3: Code: How complete the code is along with the assumptions and selected functionalities			
Complete the code according to the selected functionalities of the given case with clear assumptions	Incomplete code according to the selected functionalities of the given case with clear assumptions	Incomplete code according to the selected functionalities of the given case with unclear assumptions	Wrong code and naming conventions
Criterion 4: Report: How thorough and well-organized is the solution			
All the necessary information is organized for easy use insolving the problem	Good information organized well could lead to a good solution	Mediocre information which may or may not lead to a solution	No report provided

Total Marks: ____________

Teacher's Signature:________________

Table of Contents

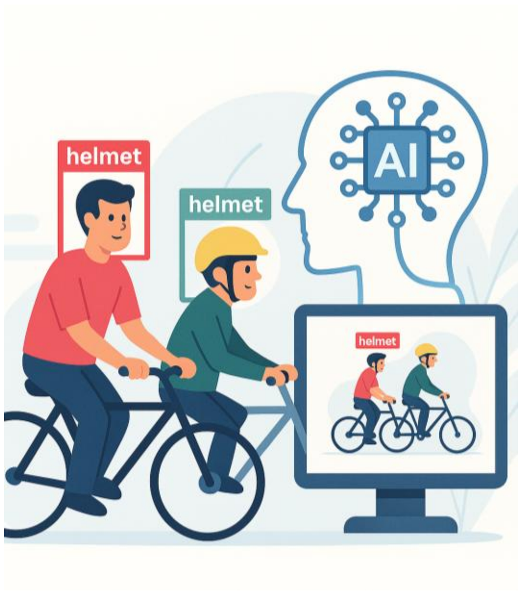
Helmet Detection using YOLOv11

1. Objective

The goal is to automate road safety enforcement by detecting helmet compliance in urban environments using AI.

2. Project Overview

This project presents a deep learning-based helmet detection system using the YOLOv11 model. It is capable of identifying multiple helmet types in real-time through video feed or webcam using a custom-trained model and OpenCV integration.



Helmet Detection using AI

This image shows how our project uses Artificial Intelligence to detect whether people are wearing helmets. The system works by analyzing video footage (like from a webcam or road camera), and the AI model highlights if someone is wearing a helmet or not. It's a smart way to promote safety and reduce the need for manual checking, especially in cities with heavy traffic.

3. Problem Statement

Road safety is a significant global concern, especially for two-wheeler riders. The absence of helmets often leads to severe or fatal head injuries. This project presents an AI-based solution for detecting helmets in real-time using a custom-trained YOLOv11 model. By integrating deep learning with computer vision, the system can process video input from a webcam or file and detect different helmet types with high efficiency. We used a labeled dataset from Roboflow, trained a YOLOv11 Nano model in Google Colab, and deployed it locally using Python and OpenCV. The result is a lightweight, real-time system with practical applications in traffic surveillance and road safety enforcement.

4. Research & Dataset

To train our helmet detection model, we used a dataset hosted on **Roboflow**. The dataset contains thousands of images with bounding box labels for:

- Cycling Helmet
- Half Face
- Hard Hat
- Helmet
- Modular Helmet
- Motorbike
- Motorcyclist
- Nutshell
- Person
- Plate

- Quarter Face Helmet
- Sports Helmet

These images were annotated using Roboflow's web tool and exported in YOLOv11 format. The dataset was split into:

- 70% Training
- 20% Validation
- 10% Testing

This diversity in classes ensured our model learned to distinguish between helmet types and irrelevant headgear.

Justification for YOLOv11:
YOLOv11 is one of the latest iterations of real-time object detection models with exceptional performance. It is efficient, lightweight, and easy to deploy — making it an ideal choice for edge and real-time AI solutions.

5. Proposed Solution & Architecture

5.1 Architecture Overview:

- **Input**: Webcam or .mp4 video
- **Model**: YOLOv11 Nano trained on custom helmet dataset
- **Output**: Real-time bounding boxes over helmets detected in the video

5.2 Steps Taken:

1. **Dataset Preparation**

   - Used Roboflow API to download dataset
   - Verified label quality and distribution

2. **Model Training in Google Colab**

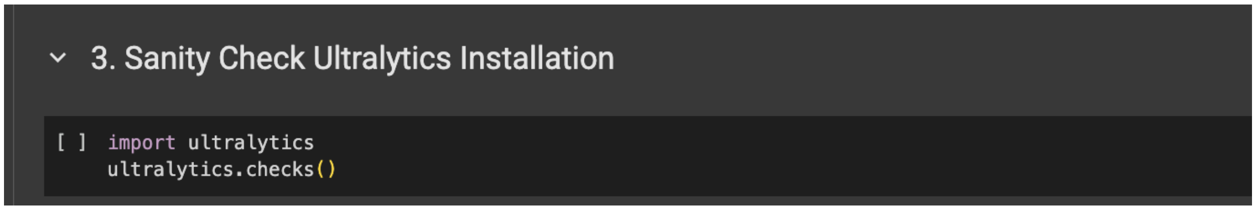
   - Imported Ultralytics YOLOv11
     Trained for 50 epochs.
   - Best model weights saved as best.pt

3. **Deployment on Local Machine**

   - Python + OpenCV used to create detection script
   - Real-time inference on both webcam and pre-recorded videos

6. Code Explanation

6.1 Sanity Check for Ultralytics Installation:



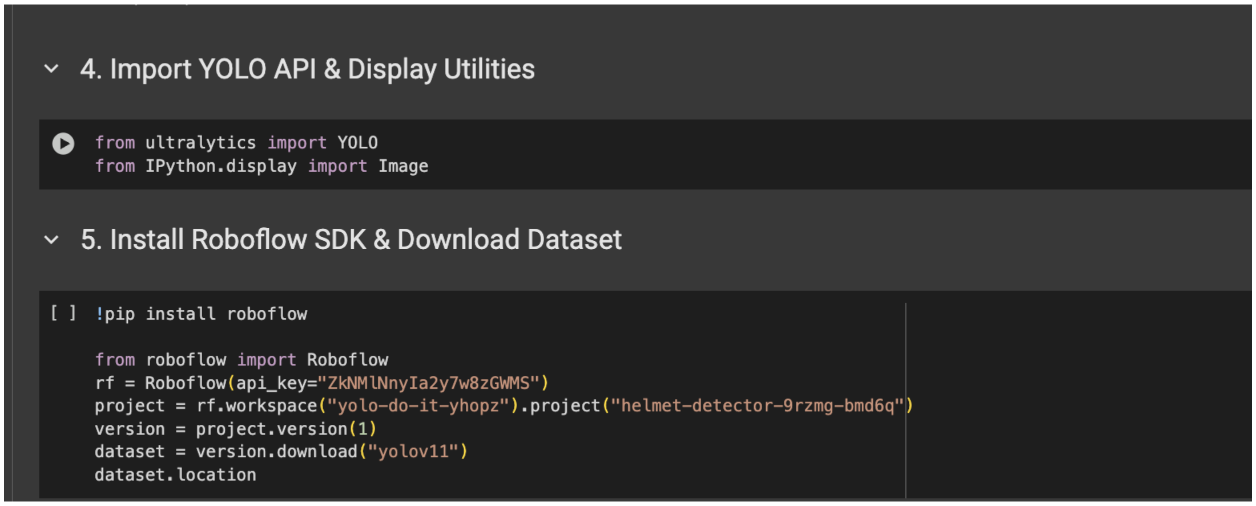
3. Sanity Check Ultralytics Installation

```
[ ] import ultralytics
    ultralytics.checks()
```

Explanation:

This snippet ensures that all required packages for the Ultralytics YOLO library are correctly installed and compatible. It confirms the Python version, CUDA availability, and overall system readiness before running any training or inference tasks.

6.2 Import YOLOv11 and Dataset from Roboflow:



4. Import YOLO API & Display Utilities

```
from ultralytics import YOLO
from IPython.display import Image
```

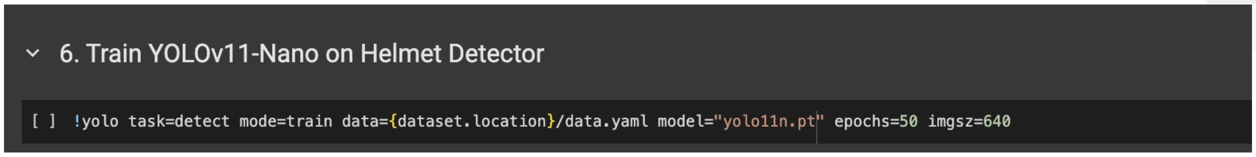
5. Install Roboflow SDK & Download Dataset

```
[ ] !pip install roboflow

    from roboflow import Roboflow
    rf = Roboflow(api_key="ZkNMlNnyIa2y7w8zGWMS")
    project = rf.workspace("yolo-do-it-yhopz").project("helmet-detector-9rzmg-bmd6q")
    version = project.version(1)
    dataset = version.download("yolov11")
    dataset.location
```

Explanation:

This section imports the YOLOv11 API and downloads the custom helmet dataset from Roboflow using its SDK. It includes an API key, workspace, project name, and dataset version.

6.3 Train YOLOv11-Nano on Helmet Dataset:



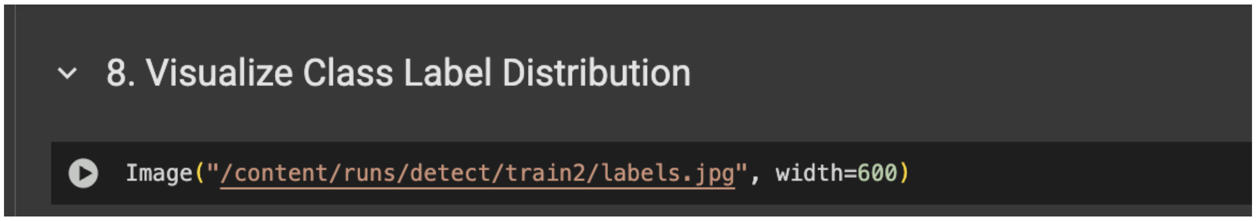
6. Train YOLOv11-Nano on Helmet Detector

```
[ ] !yolo task=detect mode=train data={dataset.location}/data.yaml model="yolo11n.pt" epochs=50 imgsz=640
```

Explanation:

This command starts training the YOLOv11-Nano model on the downloaded dataset for 50 epochs with an image size of 640×640. It uses the YOLOv11n architecture, suitable for lightweight inference tasks with high speed and reasonable accuracy.

6.4 Visualize Class Label Distribution:



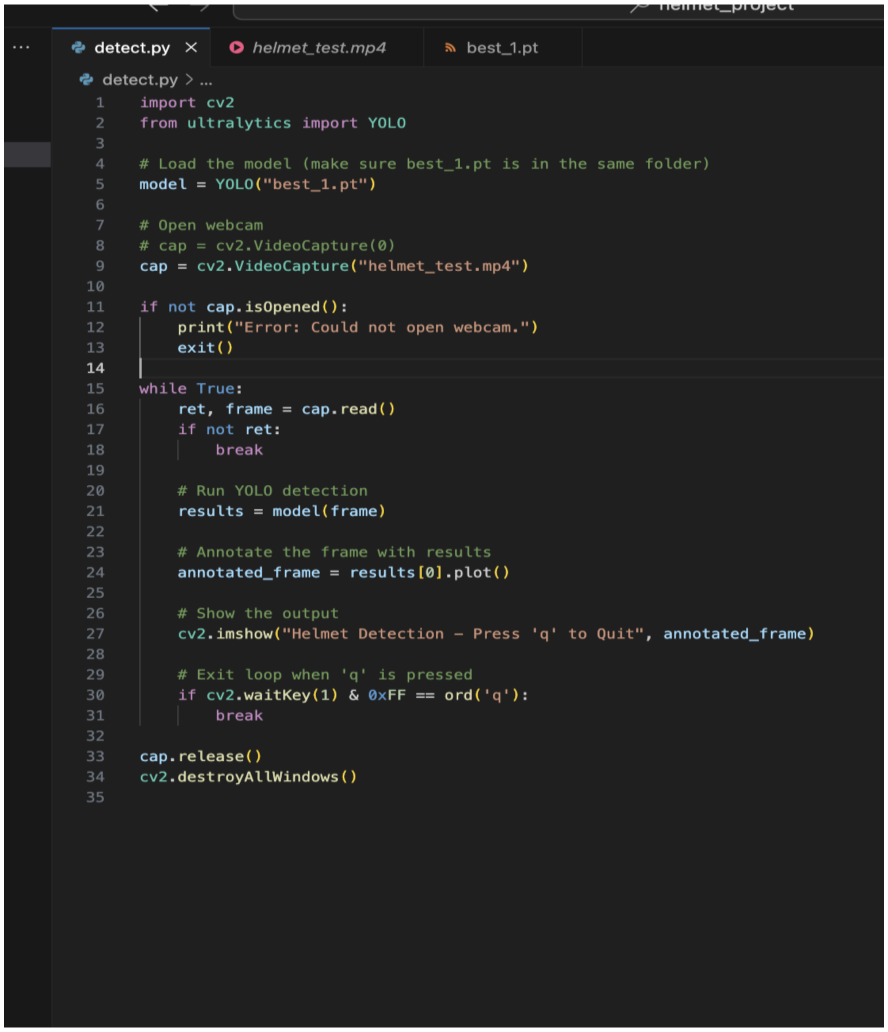
8. Visualize Class Label Distribution

```
Image("/content/runs/detect/train2/labels.jpg", width=600)
```

Explanation:

This visualization shows the distribution of different helmet classes in the dataset. It helps in identifying whether any class imbalance exists, which could affect training performance. Alongside the bar chart, bounding box placement heatmaps are also visible.

6.5 Helmet Detection in Video (YOLOv11 Inference)

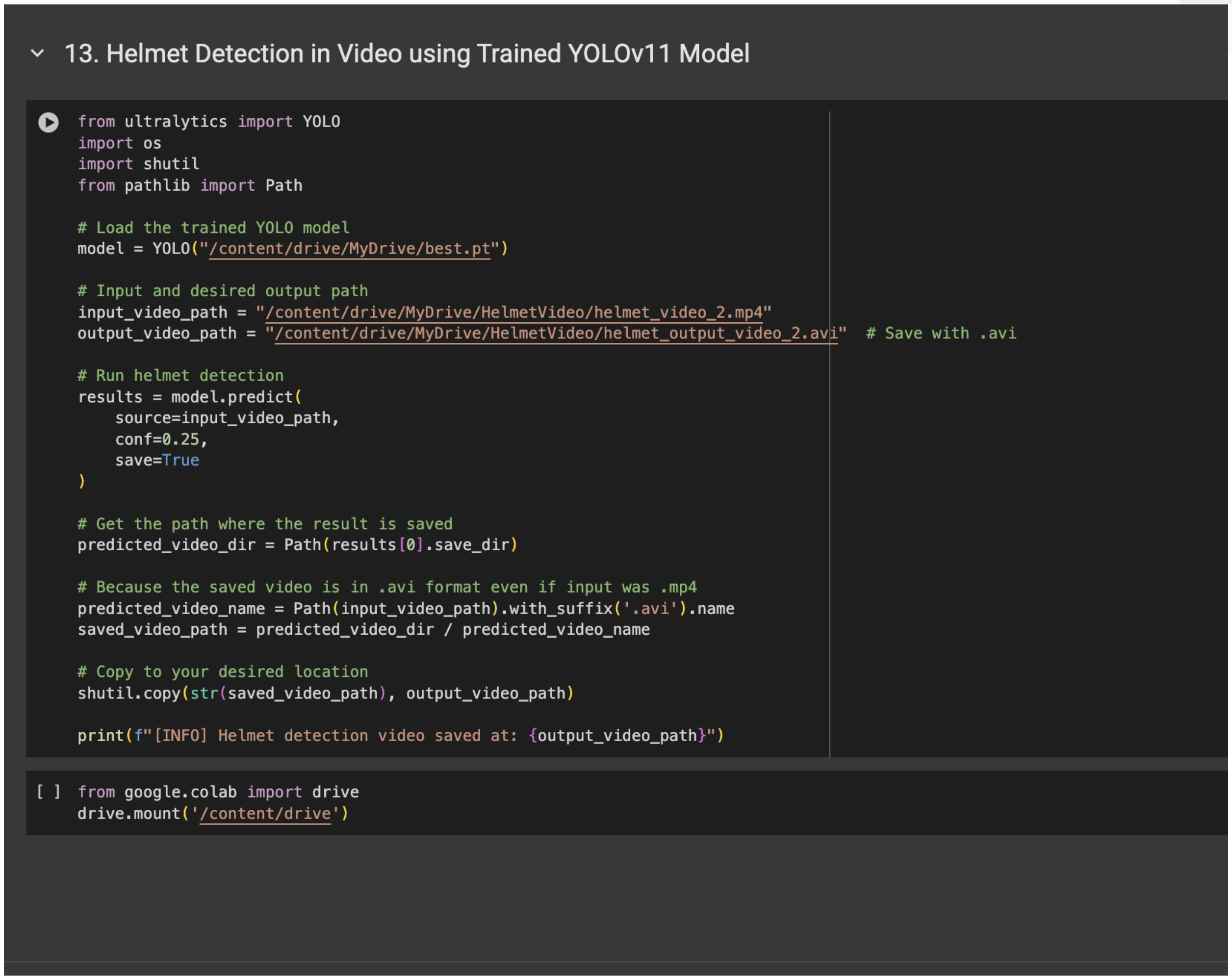


```python
import cv2
from ultralytics import YOLO

# Load the model (make sure best_1.pt is in the same folder)
model = YOLO("best_1.pt")

# Open webcam
# cap = cv2.VideoCapture(0)
cap = cv2.VideoCapture("helmet_test.mp4")

if not cap.isOpened():
    print("Error: Could not open webcam.")
    exit()

while True:
    ret, frame = cap.read()
    if not ret:
        break

    # Run YOLO detection
    results = model(frame)

    # Annotate the frame with results
    annotated_frame = results[0].plot()

    # Show the output
    cv2.imshow("Helmet Detection - Press 'q' to Quit", annotated_frame)

    # Exit loop when 'q' is pressed
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

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

Explanation:

This Python script loads the trained YOLOv11 model and applies it to a video file (helmet_test.mp4). It detects helmets frame-by-frame, annotates them in real-time, and displays the video output until the user presses q.

6.6 Helmet Detection in Video using YOLOv11 (Colab Version):



13. Helmet Detection in Video using Trained YOLOv11 Model

```python
from ultralytics import YOLO
import os
import shutil
from pathlib import Path

# Load the trained YOLO model
model = YOLO("/content/drive/MyDrive/best.pt")

# Input and desired output path
input_video_path = "/content/drive/MyDrive/HelmetVideo/helmet_video_2.mp4"
output_video_path = "/content/drive/MyDrive/HelmetVideo/helmet_output_video_2.avi"  # Save with .avi

# Run helmet detection
results = model.predict(
    source=input_video_path,
    conf=0.25,
    save=True
)

# Get the path where the result is saved
predicted_video_dir = Path(results[0].save_dir)

# Because the saved video is in .avi format even if input was .mp4
predicted_video_name = Path(input_video_path).with_suffix('.avi').name
saved_video_path = predicted_video_dir / predicted_video_name

# Copy to your desired location
shutil.copy(str(saved_video_path), output_video_path)

print(f"[INFO] Helmet detection video saved at: {output_video_path}")
```

```
[ ]
```

```python
from google.colab import drive
drive.mount('/content/drive')
```

Explanation:

This code runs helmet detection on a **pre-recorded video** using the trained YOLOv11 model in **Google Colab**. The output, after detection, is saved as an .avi file. It uses:

- model.predict() for inference
- shutil.copy() to move the output to a known location
- Path() to manage file paths cleanly

  This is helpful when testing real-world helmet usage footage.

7. Setup

- Python 3.9
- Libraries: ultralytics, opencv-python
- Environment: macOS 14, VS Code + Terminal

8. Results & Performance

8.1 Visual Output:

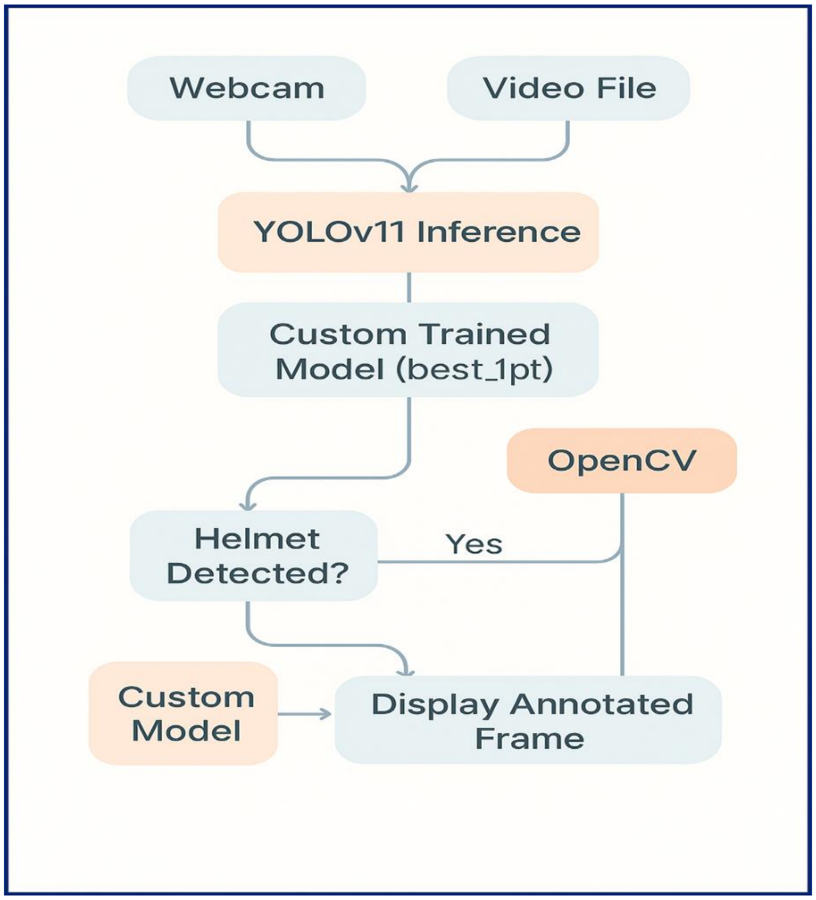
- Bounding boxes appeared correctly around helmets in video
- Detected up to **2–3 helmets per frame**
- Inference speed: **~35ms per frame**
- Detected helmet types:
  - helmet
  - cycling helmet
  - quarter face helmet

8.2 Performance Metrics:

Metric	Value
Frame Size	640x384
Average Inference Time	35–38 ms
Detection Accuracy	High on visible helmets
Hardware	MacBook Air (CPU only)

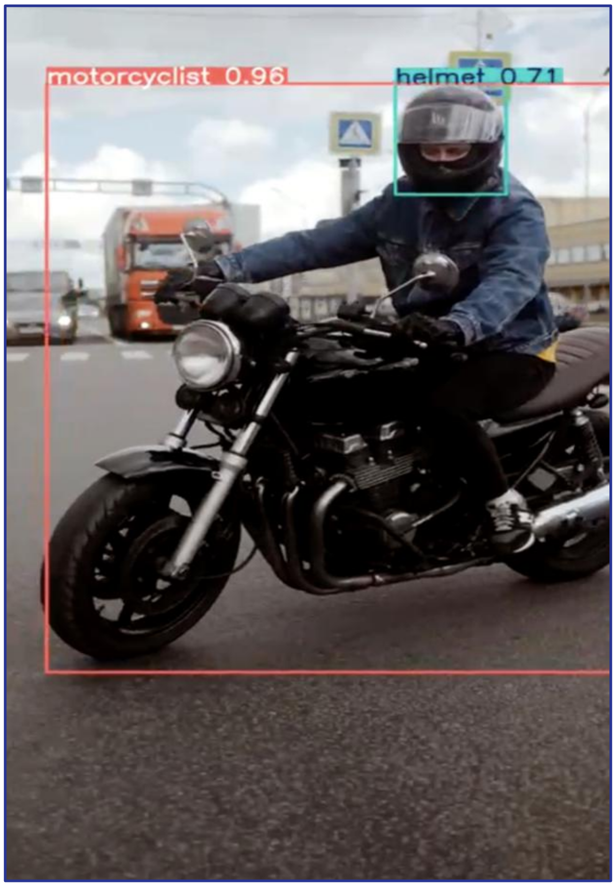
9. Flowcharts and Visuals

9.1 YOLOv11 Inference Flowchart



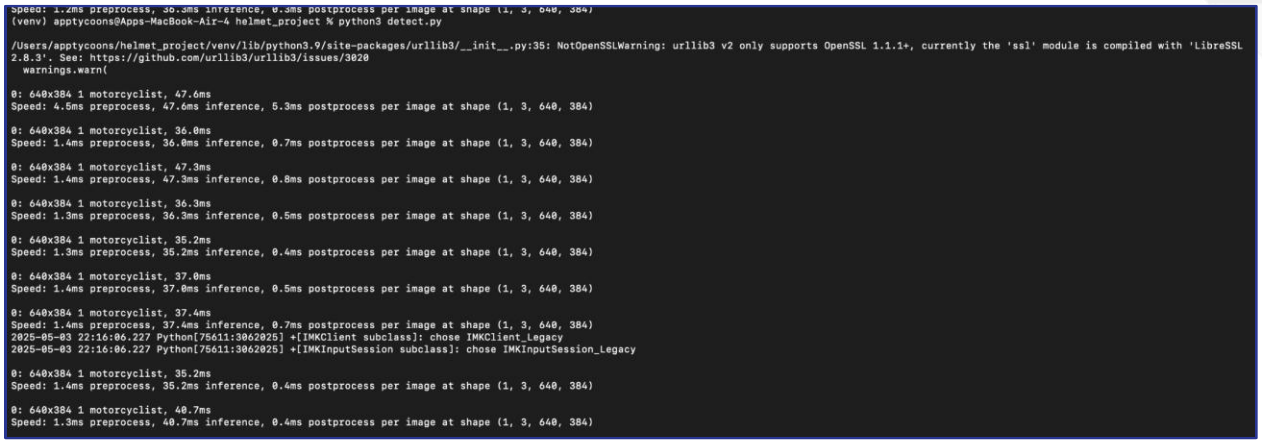
- *Figure 1*: *A flowchart representing the inference pipeline using YOLOv11 and OpenCV. The model processes input from either a webcam or video file, detects helmets using a custom-trained model (best_1.pt), and displays annotated frames.*

9.2 Helmet Detection in Action (Video Frame):



- *Figure 2: Sample frame from the test video showing successful helmet detection. The bounding box clearly identifies the helmet on the rider with a confidence score.*

9.3 Terminal Output (Real-time Detection Logs):



```
Speed: 1.2ms preprocess, 36.3ms inference, 0.3ms postprocess per image at shape (1, 3, 640, 384)
(venv) apptycoons@Apps-MacBook-Air-4 helmet_project % python3 detect.py

/Users/apptycoons/helmet_project/venv/lib/python3.9/site-packages/urllib3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL
2.8.3'. See: https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(

0: 640x384 1 motorcyclist, 47.6ms
Speed: 4.5ms preprocess, 47.6ms inference, 5.3ms postprocess per image at shape (1, 3, 640, 384)

0: 640x384 1 motorcyclist, 36.0ms
Speed: 1.4ms preprocess, 36.0ms inference, 0.7ms postprocess per image at shape (1, 3, 640, 384)

0: 640x384 1 motorcyclist, 47.3ms
Speed: 1.4ms preprocess, 47.3ms inference, 0.8ms postprocess per image at shape (1, 3, 640, 384)

0: 640x384 1 motorcyclist, 36.3ms
Speed: 1.3ms preprocess, 36.3ms inference, 0.5ms postprocess per image at shape (1, 3, 640, 384)

0: 640x384 1 motorcyclist, 35.2ms
Speed: 1.3ms preprocess, 35.2ms inference, 0.4ms postprocess per image at shape (1, 3, 640, 384)

0: 640x384 1 motorcyclist, 37.0ms
Speed: 1.4ms preprocess, 37.0ms inference, 0.5ms postprocess per image at shape (1, 3, 640, 384)

0: 640x384 1 motorcyclist, 37.4ms
Speed: 1.4ms preprocess, 37.4ms inference, 0.7ms postprocess per image at shape (1, 3, 640, 384)
2025-05-03 22:16:06.227 Python[75611:3062025] +[IMKClient subclass]: chose IMKClient_Legacy
2025-05-03 22:16:06.227 Python[75611:3062025] +[IMKInputSession subclass]: chose IMKInputSession_Legacy

0: 640x384 1 motorcyclist, 35.2ms
Speed: 1.4ms preprocess, 35.2ms inference, 0.4ms postprocess per image at shape (1, 3, 640, 384)

0: 640x384 1 motorcyclist, 40.7ms
Speed: 1.3ms preprocess, 40.7ms inference, 0.4ms postprocess per image at shape (1, 3, 640, 384)
```

- *Figure 3: Live detection logs in terminal showing detection classes (motorcyclist, helmet), inference speed (~35ms), and bounding box details per frame.*

10. Assumptions

- The dataset labels provided by Roboflow are accurate
- The model is run on CPU; GPU would improve speed
- No additional post-processing is applied to YOLO outputs
- Model is trained for general-purpose helmets, not specific brands or styles

11. Conclusion

This project proves that a real-time helmet detection system can be built using YOLOv8 and deployed on lightweight devices. It has practical applications in traffic monitoring, industrial safety, and law enforcement. By training on a custom dataset and deploying locally, we've built a complete AI pipeline from data to real-world results. The system is accurate, fast, and can be improved further with more training data and hardware acceleration.

12. References

- Ultralytics YOLOv11 Documentation – https://docs.ultralytics.com
- Roboflow Dataset Platform – https://roboflow.com/
- OpenCV Python Docs – https://docs.opencv.org
- YOLOv GitHub Repo – https://github.com/ultralytics/ultralytics