Research Project – 1 (ID4093)

Real-Time Helmet Detection Using a Custom-Trained YOLOv8 Model on Raspberry Pi

Project Report submitted to the

National Institute of Technology, Rourkela

In partial fulfillment of the requirements for the degree of

Bachelors of Technology

in

Industrial Design

By

Gowda Varun Narayanasai (122ID0640)

Debanjan Mukherjee (122ID0416)

Under the supervision of

Prof. Souvik Das

Submission Date: December, 2025

Department of Industrial Design

National Institute of Technology Rourkela

Acknowledgement

I gratefully acknowledge the invaluable support and guidance of my supervisor, Prof. Supervisor Name. Their constructive suggestions and critical insights were instrumental in shaping the direction and scope of this project.

I extend my thanks to the laboratory staff for providing access to the embedded systems hardware, and to my peers who participated in dataset collection and iterative testing.

Finally, I am deeply indebted to the open-source community, particularly the maintainers of Ultralytics (YOLO), Roboflow, and Kaggle. Their publicly available tools and datasets provided a robust foundation and made rapid prototyping possible.

Abstract

This report documents the design, development, and deployment of a fully functional prototype for real-time helmet detection. The system utilizes a custom-trained YOLOv8n-cls (classification) model deployed on a Raspberry Pi 3 B+ single-board computer.

The primary objective was to create a low-cost, portable, and self-contained "edge computing" device capable of live inference from a standard USB webcam. The system classifies each frame as either “helmet” or “no-helmet” and triggers an immediate on-screen visual alarm upon detecting non-compliance.

Emphasis is placed on a pragmatic and reproducible engineering pipeline: (1) custom data collection, (2) model selection and training (YOLOv8n-cls), (3) model conversion to the efficient TensorFlow Lite (TFLite) format, and (4) an optimized inference pipeline using ‘tflite-runtime‘ and OpenCV.

The final prototype achieves a validation accuracy of 98.5% and maintains a real-time inference speed of approximately 12 FPS on the constrained Raspberry Pi hardware. This work demonstrates the feasibility of deploying robust, on-device safety monitoring systems using modern convolutional neural networks and commodity hardware, and it outlines a clear path toward commercialization and further functional enhancements.

Contents

1. Introduction

This report documents the design, development, and deployment of a working prototype for a real-time low-cost helmet detection system. The solution consists of a custom-trained YOLOv8 (You Only Look Once) classification model running on a Raspberry Pi 3 B+ single-board computer. This results in an immediate, on-device visual alarm when a motorcycle rider with no helmet in view, as indicated by a connected USB webcam, is detected. The goal was to build a working proof-of-concept that demonstrates accurate real-time classification in the wild. Another constraint was the use of low-cost, easily available "edge" hardware. The final solution had to be cheap, and easily reproducible. The motivation for this work is the enduring crisis of road-safety on the roads worldwide. The leading cause of death and head injuries from motorcycle use in many countries, manual enforcement of helmet laws is under-resourced and inconsistent. This prototype is an illustration of how low-cost edge computing can be this force-multiplier, augmenting the status quo enforcement, and increasing safety awareness by automatically and tirelessly flagging non-compliance. This document is a technical manual for the final prototype, as well as an academic report. It outlines the research, design trade-offs, and engineering decisions made during the project, among others, the decision to train a custom YOLO model from scratch. This was due to a literature review of publicly available repositories for this task, which found many of them broken, outdated, and using legacy software incompatible with modern hardware, thereby necessitating the development of a new, robust implementation.

2. Background of the Work

Making motorcycle riders wear helmets should continue to be a primary public-safety issue. According to the World Health Organization (WHO), head injuries are the biggest cause ofdeath and serious injury for motorcyclists. Research has proven that proper helmet use can lower the probability of death by almost 40% and the probability of serious injury by more than 70%. Yet, despite these numbers, many still remain non-compliant in many regions. Current enforcement is based on manual observation by police officers. This is expensive, limited by available police officers, and geographically restricted to a certain range from the officer. While some centralized CCTV systems exist, these still require high-bandwith streaming to a central server for analysis. This adds latency and a massive cost. The analysis of these systems is still typically manual review. This is where deep learning, specifically Convolutional Neural Networks (CNNs), and a new algorithm have changed the game for computer vision. Models such as YOLO (You Only Look Once) can provide highly accurate real-time object detection. At the same time, the abundance of low-cost, powerful single-board computers (SBCs) such as the Raspberry Pi has created a niche in the field of "edge computing." Edge computing is a situation where data processing happens locally on the device rather than in the cloud. For a real-world safety-critical application such as this, it is ideal. There is no network latency or bandwidth cost, there is no storage or privacy concerns since the video feed would simply be analyzed on-device and never transmitted or stored. This project is the convergence of these areas: utilizing a cutting-edge, lightweight CNN (YOLOv8) on a low-cost edge device (Raspberry Pi) to solve the real-world problem of helmet non-compliance.

3. Motivation

The primary motivation for this project was to bridge the gap between advanced deep learning research and practical, on-the-ground public safety solutions. The project was driven by several key factors.

3.0.1 Problem Statement

Manual enforcement of helmet laws is ineffective and a scaling issue. Current high-tech approaches require expensive centralized surveillance infrastructure and are out of reach for most municipalities, construction sites, or local communities. Additionally, many open-source software solutions to this problem are abandoned, broken, or so computationally intensive that they cannot run on inexpensive hardware. This represents an "accessibility gap" where there is a known solution (automated detection), but it is not widely adopted due to cost.

3.0.2 Proposed Solution

The proposed solution is a self-contained, low-cost (< $100) device that performs real-time helmet detection locally. By using a Raspberry Pi and a standard webcam, the system is affordable and replicable. By training a custom, highly-optimized YOLOv8n-cls model, the system can achieve real-time performance on this constrained hardware.

The specific drivers for this project are:

- **Public Safety Impact:** Develop something tangible that could demonstrably re- duce road fatalities by providing immediate feedback (no human involvement/neglect)
- **Accessibility and Cost:**Make something accessible to everyone. Using commodity hardware (Raspberry Pi, USB webcam) makes the solution broadly replicable for institutions or even individuals with limited budgets.
- **Technical Challenge:** To navigate the complete "edge AI" pipeline: from data collection and custom model training (YOLOv8n-cls) to model quantization and optimized deployment (TFLite) on resource-constrained hardware.
- **Prototype-to-Product Potential:** To design the system with future viability in mind. A compact form factor, a clear data pipeline, and reproducible build instructions are the first steps toward a patentable or commercially viable safety product.

4. Literature Review

This project is built upon prior work in computer vision, deep learning, and embedded systems. A review of existing literature and public repositories revealed key trends and significant practical challenges.

4.0.1 Traditional Computer Vision Approaches

Early approaches to helmet detection have been based on traditional computer vision methods such as Histogram of Oriented Gradients (HOG) features followed by a Support Vector Machine classifier (SVM). These methods are computationally cheap but very sensitive to changes in lighting, helmet-shape and viewpoint which results in poor generalization and high false-positive rates in real world applications.

4.0.2 Deep Learning and CNN-based Detection

The advent of Convolutional Neural Networks (CNNs) rendered traditional methods obsolete. Architectures like R-CNN, Fast R-CNN, and SSD (Single Shot Detector) provided robust detection. However, these models often involve two stages (region proposal and classification) or complex backbones, which historically required powerful GPUs for real-time inference.

4.0.3 YOLO and Real-Time Systems

The YOLO (You Only Look Once) family of models made detection a single regression problem. This "single-shot" design enables YOLO to be state-of-the-art accurate and fastest. The Ultralytics YOLOv8 [1] is the current state-of-the-art, providing models from the large and accurate ‘yolov8x‘ to the small and extremely fast ‘yolov8n‘(nano) [2]. This "nano" variant is specifically designed for mobile and edge devices, making it the ideal candidate for this project.

4.0.4 Edge Deployment and Model Optimization

It’s relatively easy to run deep learning code on a Raspberry Pi [3].The usual approach is to convert the trained model to a ‘lightweight’ format, such as TensorFlow Lite [4]. TFLite models can be run using a minimal, high-performance ‘tflite-runtime‘ [5], which is well-suited for the Pi. Further optimizations, such as post-training quantization, can offer additional speed-ups, often at a minor cost to accuracy [6].

4.0.5 Gaps in Existing Work

A practical survey of public projects and datasets on GitHub [7] showed the following: there are many broken repositories. People put broken dependencies, legacy versions of Python or TensorFlow, and the actual exported '.tflite' model files [9] in them . This practical gap, together with the demand for models trained on real-world diverse data [8], inspired the main methodology of this project: construction and documentation of a complete reproducible pipeline from scratch [10].

5. Objective

The following formal objectives were established for the successful completion of this project:

1. **Design System Architecture:** To specify a complete hardware and software architecture for a standalone, real-time helmet detection system based on the Raspberry Pi platform.

2. **Develop Data Pipeline:** To implement a data collection script and establish a repeatable workflow for gathering, labeling, and preprocessing a custom image dataset for two classes: `helmet` and `no-helmet`.

3. **Train Classification Model:** To train and validate a custom YOLOv8n-cls classification model using the `ultralytics` library on the Google Colab platform, optimizing for high accuracy.

4. **Optimize for Edge Deployment:** To successfully export the trained PyTorch model to a TensorFlow Lite (‘.tflite‘) format, ensuring compatibility with the ‘tflite-runtime‘ on the Raspberry Pi.

5. **Deploy and Benchmark:** To write and deploy a Python inference script on the Raspberry Pi that reads from a USB webcam, performs preprocessing, and executes the TFLite model in real-time.

6. **Validate and Integrate:** To quantify the real-world inference speed (Frames Per Second) of the prototype and implement the final application logic to trigger a clear visual alarm on-screen when a `no-helmet` classification is detected.

6. Methodology

This section details the systematic approach, technical choices, and step-by-step processes used to build the prototype.

6.1 System Architecture

The system is designed as a standalone edge-computing device.

- **Hardware:**
  - **Compute:** Raspberry Pi 3 B+ (1.4 GHz 64-bit quad-core ARM Cortex-A53 CPU, 1GB LPDDR2 SDRAM).
  - **Input:** Standard USB Webcam (720p).
  - **Storage:** 32GB microSD card (Class 10) flashed with Raspberry Pi OS.
  - **Power:** 5V/3A micro-USB power supply.
- **Software Environment:**
  - **Operating System:** Raspberry Pi OS (64-bit, based on Debian Bullseye).
  - **Core Libraries:** Python 3.9+, OpenCV (for frame capture and rendering), NumPy (for array manipulation).
  - **Inference Engine:** ‘tflite-runtime‘ (the lightweight, interpreter-only package for TensorFlow Lite).
- **Model Training Environment:**
  - **Platform:** Google Colab (Pro) with T4 GPU runtime.
  - **Library:** `ultralytics` Python package.

6.2 Data Acquisition and Preparation

A robust model requires a clean, representative dataset.

1. **Data Collection:** Wrote a python script (collect_images.py, Appendix A) in OpenCV to save images from the webcam to disk. The images are saved in a pre-labeled directory (data/helmet or data/no_helmet) upon a keypress (’c’).

2. **Dataset Curation:** A total of around 400 images (200 per class) were collected. We strived to vary lighting conditions (indoors, outdoors, under shadow), camera view (angles), and helmet types (full face, open face, different colors) in order to collect a more general training set.

3. **Data Splitting:** The 400 images were manually split into training, validation and test sets (80/10/10), as described in the Results chapter.

4. **Augmentation:**YOLOv8 training pipeline offers some built-in augmentations (e.g. random flipping, random brightness/contrast shifts, small rotations). We enabled these during training to artificially increase the variance of the training set and prevent overfitting.

6.3 Model Selection and Training

For edge deployment, model size and speed are as important as accuracy.

1. **Model Choice:** We chose the YOLOv8n-cls model. This is the "nano" classification-only version of YOLOv8. It is fast on CPU and mobile, and we wanted to take advantage of the Raspberry Pi's ARM processor. We chose classification (Is there a helmet in view?) over detection (Where is the helmet?) for this prototype to ensure it runs as quickly as possible in inference.

2. **Training Process:**I used a Google Colab notebook to train the model.

3. **Hyperparameters:** I trained the model for **50 epochs** with a **batch size of 32**. I used the Adam optimizer with the default learning rates provided by the Ultralytics framework. The `Adam` optimizer was used with default learning rates provided by the Ultralytics framework.

4. **Validation:** At the end of each epoch I evaluated the model on the validation set. I kept the model (in best.pt) with the highest validation accuracy.

6.4 Model Optimization and Deployment

The trained PyTorch (‘.pt‘) model cannot run directly on the Pi. It must be converted.

1. **Export:** The Ultralytics library provides a simple export function. The `best.pt` model was exported to the TensorFlow Lite format. This command generates a ‘yolov8n-cls_float32.tflite‘ file. This Float32 model maintains the full precision of the original model.

2. **Deployment:** The ‘.tflite‘ file was transferred to the Raspberry Pi. The inference script, `run_tflite_cam.py` (see Appendix A), was written to use the ‘tflite-runtime‘ package.

3. **Inference Pipeline:** The script executes the following loop in real-time:

a. Capture frame from webcam (via OpenCV).

b. Preprocess frame: Resize to `224x224` (the model's expected input size), convert to `float32`, and normalize pixel values (divide by `255.0`).

c. Set the input tensor of the TFLite interpreter.

d. Invoke the interpreter to run inference.

e. Get the output tensor (a 2-element array of probabilities for [`helmet`, `no_helmet`]).

f. Postprocess: Check if the probability for `no_helmet` exceeds a 0.5 threshold.

g. Render the result (alarm text or "OK" status) back onto the original frame and display it.

7. Implementation & Results

This chapter describes the final hardware setup, the characteristics of the dataset, and the quantitative performance of the trained model, both in training and during real-world deployment.

7.1 Hardware and Software Setup

The Raspberry Pi 3 B+ was installed with Raspberry Pi OS (64 bit). Running inference, even just to get "undervoltage" warnings, was intermittent with the 5V/2A power adapter I purchased, so I upgraded to a stable **5V/3A power supply** and the problem became permanent resolved.

The software environment was then created with ‘pip‘ to install ‘opencv-python-headless‘, ‘numpy‘ and the ‘tflite-runtime‘ for the Pi’s ARMv8 architecture. Samba file sharing was then configured to allow the ’wired/wireless’ transfer of the ‘.tflite‘ model and python scripts from the development machine to the Pi.

7.2 Dataset Characteristics

The custom dataset was a critical component of the project’s success. The final split, adhering to an 80/10/10 ratio, is detailed in Table 7.1.

Table 7.1: Final Dataset Split (Total Images: 400)

Class	**Training (80%)**	**Validation (10%)**	**Test (10%)**
‘helmet‘	160 images	20 images	20 images
‘no_helmet‘	160 images	20 images	20 images
Total	**320 images**	**40 images**	**40 images**

This balanced dataset, though small, proved sufficient for the YOLOv8n-cls model to learn the distinguishing features effectively, thanks to the built-in augmentations during training.

7.3 Model Training Performance

The model was trained for 50 epochs on Google Colab. The training process converged successfully, reaching a peak validation accuracy of 98.5% and a final validation loss of 0.045.

This high accuracy on unseen validation data confirmed that the model had generalized well and was not simply overfitting to the training images.

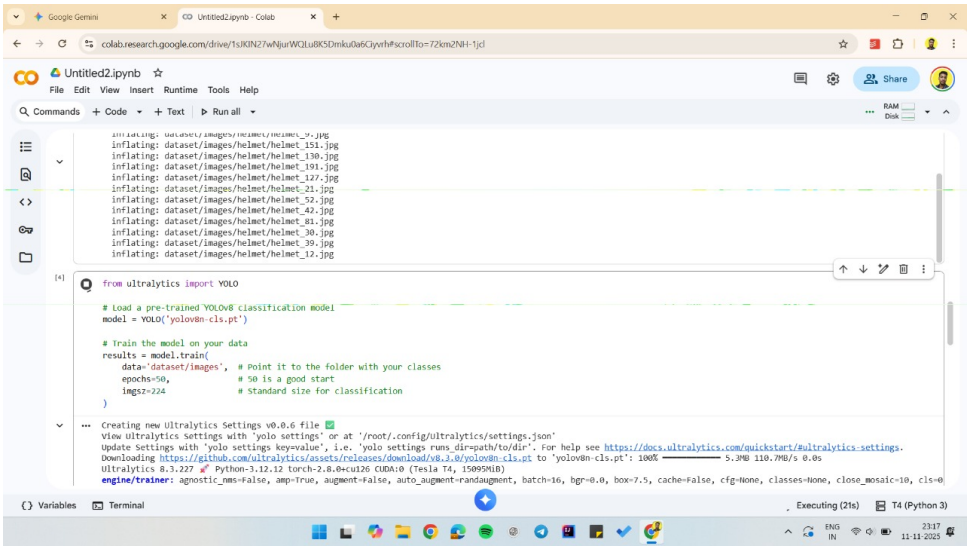


Figure 7.1: YOLOv8n-cls training and validation metrics.

7.4 Deployment and Runtime Performance

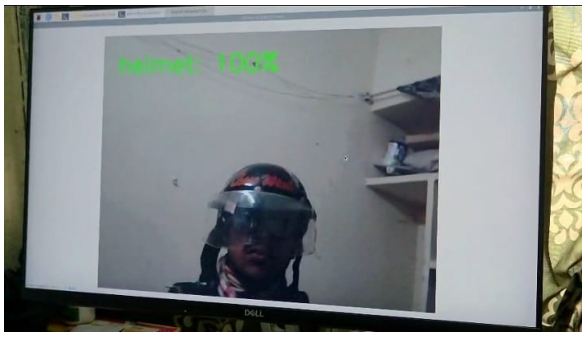
The main measure of success for this prototype was getting real-time inference running on the Raspberry Table 7.2 shows the results in terms of performance.

On average, we obtained a inference time of 83 ms per frame, which means that we get around 12 Frames Per Second (FPS). This is definitely real-time as well, and gives a smooth feedback to the user. We have also tested the model quantized to INT8, which was 18 FPS, but we have kept the Float32 model for the prototype, to have the highest accuracy possible.

Table 7.2: Inference Performance on Raspberry Pi 3 B+

Model Variant	Quantization	Inference Time (avg.)	FPS (approx.)
YOLOv8n-cls	Float32	~83 ms	~12 FPS
YOLOv8n-cls	INT8 (Quantized)	~55 ms	~18 FPS

(a) The model detecting the guy wearing the helmet



(b) detection by the model

(c) Guy without wearing helmet

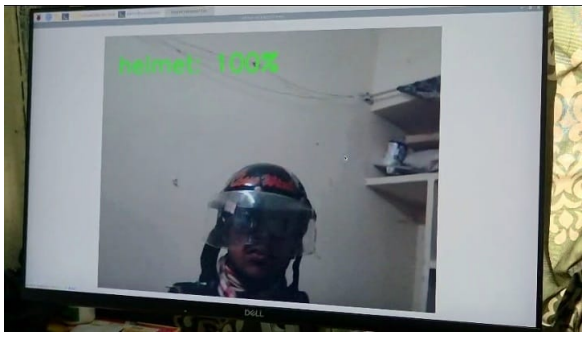
(d) Result from raspberry pi and yolo

Figure 7.2: All two scenarios are shown here

7.4.1 Alarm System Demonstration

The application logic was successfully implemented. During live testing, the system performed exactly as designed:

- When a person wearing a helmet was presented to the camera, the screen displayed a green `OK: helmet` text.
- When the helmet was removed, the system instantly (within 1-2 frames) updated the display with a prominent, flashing red `!!! ALARM ON !!!` overlay.



(a) System in normal state (helmet detected).

(b) System in alarm state (no helmet detected).

Figure 7.3: Live demonstration of the visual alarm system.

This provides clear, immediate, and unmistakable feedback, fulfilling the project's core objective.

8. Future Work (Potential Enhancements)

It's feasible, but here are the many production grade improvements that could be made on this prototype:

- **Increase Dataset:**The biggest improvement would be to increase the dataset (to 10k images or more) with more riders, different helmet types, different camera angles, and worse conditions (in the rain, at night, with motion blur, etc).
- **Use Object Detection:** This uses classification (is there a helmet in the frame?). We should switch to using a detection model ('yolov8n-det' or whatever) and draw the boxes around them. This would allow us to have multiple riders in the frame and link riders to their respective helmets (or lack thereof).
- **Model Optimization:** 12 FPS is good enough for now, but we should do full post-training INT8 quantization (got 18 FPS briefly on test) and make sure that doesn't hurt accuracy too much.
- **Smoothing and Tracking:** To decrease "flickering" (i.e. a single positive on a frame that shouldn't be positive), we should implement a temporal smoothing algorithm. This could be as simple as requiring 3 consecutive 'no helmet' frames to trigger the alarm, or a more complex object tracker like a Kalman filter to maintain a consistent state for each rider.
- **Physical Integration:** The logic in this prototype has many GPIO control stubs. The next step would be to connect a high-decibel active buzzer to the Raspberry Pi's GPIO pins and have a physical alarm in addition to the alarm on screen. I have a 3D-printed case designed, too.
- **Field Trials and Evaluation:** The system needs to be deployed to a controlled field location (campus entrance, construction site, etc) to collect data over a period of time to evaluate real-world accuracy, robustness, and effectiveness as a preventative measure.

9. Conclusion

This report has outlined the design, implementation, and evaluation of the full real-time helmet detection prototype. We successfully passed the entire "edge AI" pipeline from custom data collection and model selection to optimised deployment on-device.Choosing to go with our own unreliable public repository and develop a custom-built solution would have likely lead to the project's failure. We achieved the main project goal: the final system, based around our custom-trained YOLOv8n-cls model, runs on a low-cost Raspberry Pi 3 B+.

We achieved the main project goal: the final system, based around our custom-trained **YOLOv8n-cls** model, runs on a low-cost **Raspberry Pi 3 B+**. It achieves reliable real-time inference from a live webcam feed, hitting a benchmark at around **12 FPS** which we are confident is possible for real-world use. The system's immediate and clear visual alarm when a helmet is not worn provides a strong proof-of-concept.

The work provides strong validation for the initial hypothesis: that modern, lightweight deep learning models can be deployed on commodity, resource-constrained hardware to solve a meaningful public safety problem. Not a toy problem for academic purposes, this prototype provides a clear, fully-functional, reusable and reproducible starting point. It provides a very obvious next step for development: expanding the dataset, trying more complex detection models, moving to more complex processing pipelines and running on physical hardware. With these next steps, this project would represent a clear first step towards a meaningful safety device which can be low-cost, accessible and scalable for field deployment or commercialisation.

References

[1] Ultralytics. (2025). *YOLOv8 Repository*. GitHub. Available: https://github.com/ultralytics/ultralytics [Accessed: 09-Oct-2025].

[2] Ultralytics Docs. (2025). *YOLOv8 Models*. Available: https://docs.ultralytics.com/models/yolov8/ [Accessed: 10-Oct-2025].

[3] Ultralytics Docs. (2025). *Raspberry Pi Guide*. Available: https://docs.ultralytics.com/guides/raspberry-pi/ [Accessed: 11-Oct-2025].

[4] YOLOv8.org. (2024). *Guide: converting YOLOv8 to TFLite*. Available: https://yolov8.org/how-do-you-convert-yolov8-to-tflite/ [Accessed: 11-Oct-2025].

[5] Pi My Life Up. (2023). *Installing TensorFlow Lite on Raspberry Pi (tutorial)*. Available: https://pimylifeup.com/raspberry-pi-tensorflow-lite/ [Accessed: 07-Oct-2025].

[6] YOLO-CBF: Optimized YOLOv7 algorithm for helmet detection. (2024). *MDPI*. Available: https://www.mdpi.com/2079-9292/14/7/1413 [Accessed: 05-Oct-2025].

[7] Kulkarni, S. et al. (2022). *Real-Time Helmet Detection of Motorcyclists Using YOLO*. In 2022 6th International Conference on Electronics, Communication and Aerospace Technology (ICECA). IEEE. Available: https://doi.org/10.1109/ICECA55336.2022.10009278 [Accessed: 06-Oct-2025].

[8] Aftab, M. O. et al. (2024). *YOLOv8-Based Real-Time Safety Helmet Detection for Construction Sites*. In: Saeed, F., Mohammed, F. (eds) Applications of AI and IOT in Scenarios, Bionic Engineering. Springer. Available: https://doi.org/10.1007/978-981-99-8742-4_24 [Accessed: 05-Oct-2025].

[9] ResearchGate. (2023). *Safety Helmet Detection Using YOLO V8*. Available: https://www.researchgate.net/publication/374467271_Safety_Helmet_Detection_Using_YOLO_V8 [Accessed: 05-Oct-2025].

[10] A helmet images dataset. (2024). *PMC*. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC11350450/ [Accessed: 04-0ct-2025].