Murshidabad College of Engineering & Technology

Project Report on

Traffic Monitoring System using Python-OpenCV & YOLOv8

Submitted by

Debajyoti Talukder

Sukanya Saha | Surojit Barman | Kazi Nasrin Akhtar

In partial fulfillment of the degree

Bachelor of Technology

In

Computer Science & Engineering

Nov, 2023

Traffic Monitoring System using Python-OpenCV and YOLOv8

Murshidabad College of Engineering and Technology

Under guidance of Prof. Anindya Bakshi

(HOD, CSE Dept.)

Group Member

Debajyoti Talukder

Sukanya Saha | Surojit Barman | Kazi Nasrin Akhtar

Traffic Monitoring System using Python-OpenCV and YOLOv8

https://github.com/DebajyotiTalukder2001/Traffic-Monitoring-System

---

MURSHIDABAD COLLEGE OF ENGINEERING & TECHNOLOGY

Under guidance of **Prof. Anindya Bakshi**

(HOD, CSE Dept.)

Group Member

Debajyoti Talukder

Sukanya Saha | Surojit Barman |

Kazi Nasrin Akhtar

Acknowledgement

I would like to express my profound gratitude and thanks to **Prof. *Anindya Bakshi*, HOD, Department of Computer Science and Engineering (CSE), Murshidabad College of Engineering and Technology**, for the intellectual support, inspiring guidance, and his invaluable encouragement, suggestions, and cooperation that helped a lot to complete our project successfully.

Debajyoti Talukder (10600120012)
Sukanya Saha (10600120016)
Surojit Barman (10600121059)
Kazi Nasrin Akhtar (10600120017)
Dept. Computer Science & Engineering
Semester: 7th
Year: 4th (2023-24)

ESTD : 1998

ESTD : 1998

Murshidabad College of Engineering & Technology

Banjetia, Berhampore, PO. CossimBazar-Raj, Dist. Murshidabad, West Bengal, Pin-742102

https://www.mcetbhb.net/

Date .....

Certification

This is to certify that the project entitled "**Traffic Monitoring System using Python-OpenCV & YOLOv8**" submitted by **Debajyoti Talukder, Sukanya Saha, Surojit Barman, and Kazi Nasrin Akhtar, who** are the students of **Murshidabad College of Engineering & Technology, Berhampore, W.B.**, in partial fulfilment of the degree of **Bachelor of Technology in Computer Science and Engineering**, was carried out under my guidance and supervision.

They have completed the total parameters and requirements of the entire project.

The work undertaken in this dissertation is the result of their own efforts. The result embodied in this dissertation has not been submitted for any other degree and does not form a part of any other course undergone by them.

Signature of **Project Guide**

Prof. Anindya Bakshi

(HOD, CSE Dept.)

Signature of **HOD**

Prof. Anindya Bakshi

(CSE Dept.)

Abstract

Traffic monitoring is a critical component of intelligent transportation systems. It helps to improve traffic flow, reduce congestion, and enhance safety. Traditional traffic monitoring systems are often expensive and complex to deploy and maintain.

This project proposes a traffic monitoring system using Python-OpenCV and YOLOv8. YOLOv8 is a state-of-the-art object detection algorithm that is fast, accurate, and efficient. Python-OpenCV is a popular computer vision library for Python.

The proposed system will be able to:

- Detect, track, and count vehicles of different types (e.g., cars, trucks, buses, motorcycles).
- Detect vehicle speed.
- Detect vehicle speed limit violations.

The system will be implemented in Python and will be easy to deploy and maintain. It can be used to monitor traffic in a variety of settings, such as highways, intersections, and urban areas.

The system has the potential to be a valuable tool for traffic engineers and law enforcement agencies to improve traffic safety and efficiency.

Table of Contents

CHAPTER 1

Introduction

1.1. Project Objective

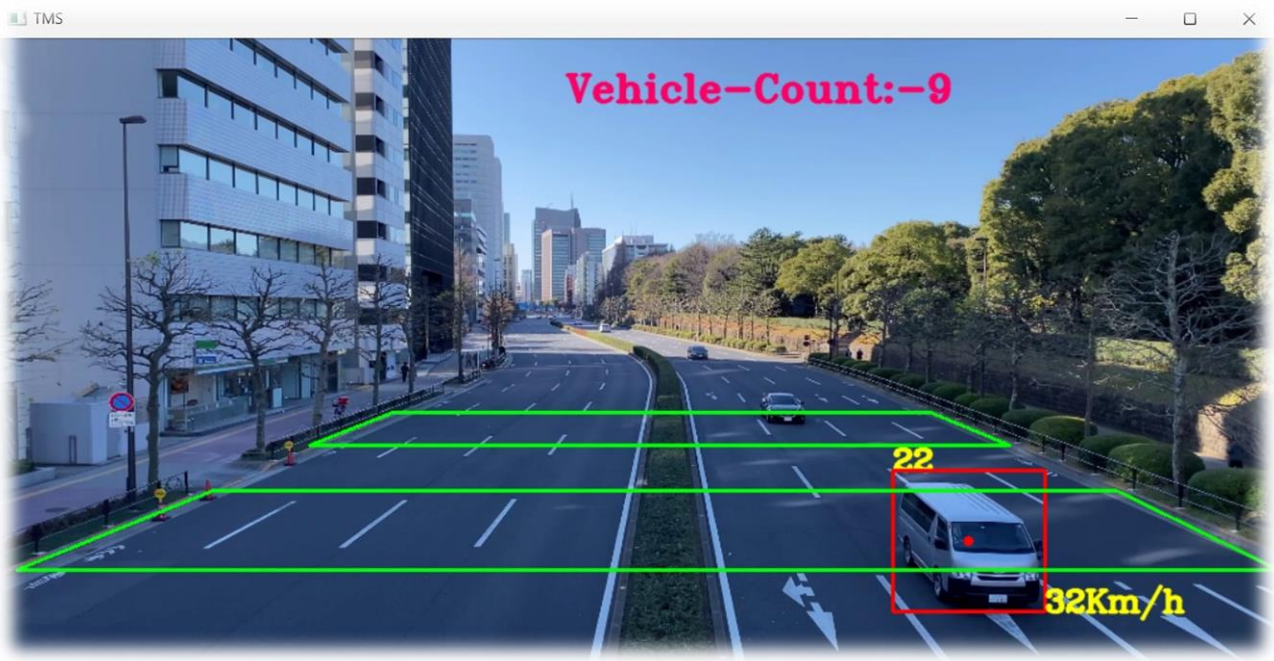
Title: "Traffic Monitoring System using Python-OpenCV and YOLOv8"

Main objective and goal of this project is to create a **Traffic Monitoring System using Python-OpenCV and YOLOv8,** which will be able to:

1. Detect, track, and count vehicles.

2. Detect vehicle speed.

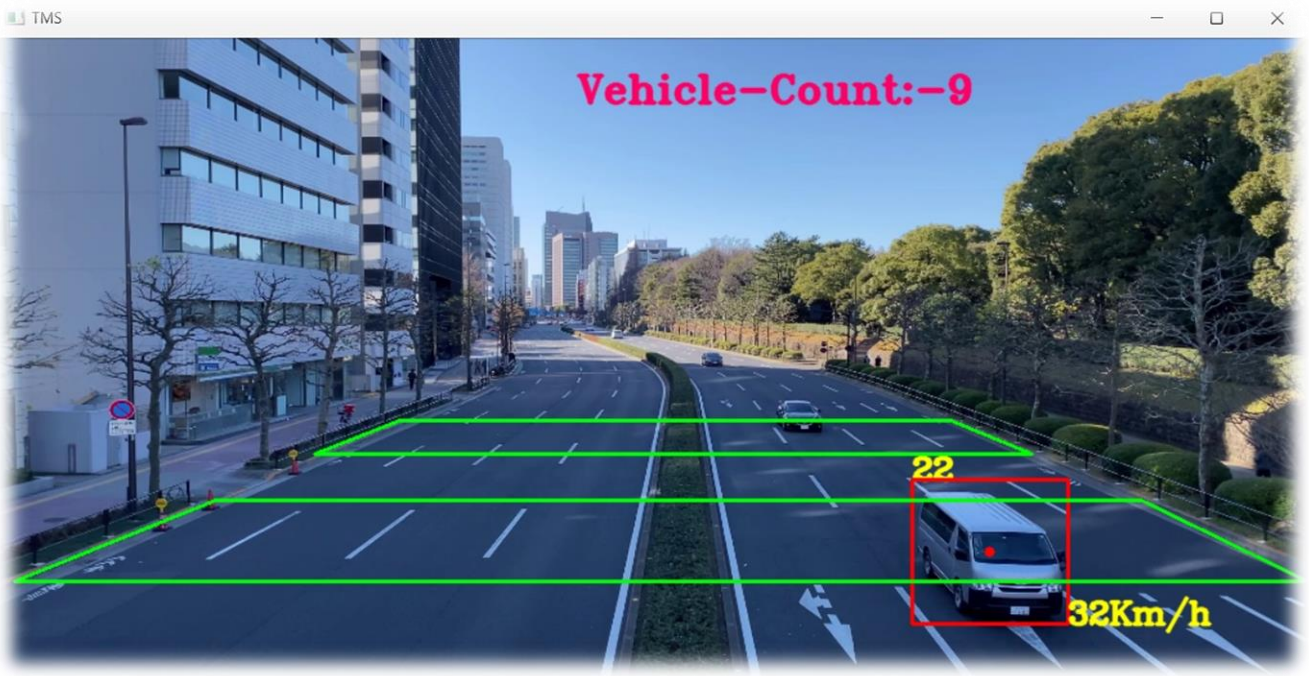
3. Detect vehicle speed limit violations.



1.2. Basic Information & Theory

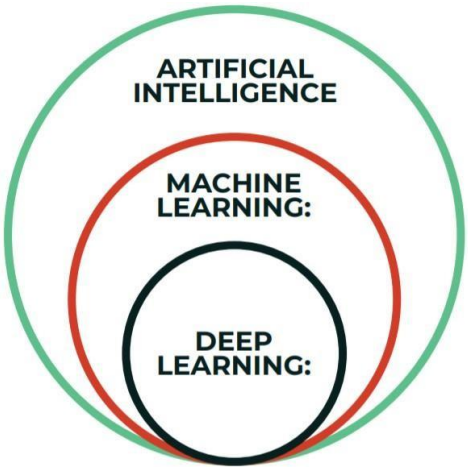
It is a **Traffic Monitoring System using Python-OpenCV and YOLOv8** which will be able to:

1. Detect, track, and count vehicles.
2. Detect vehicle speed.
3. Detect vehicle speed limit violations.



Artificial Intelligence

Science that empowers computers to mimic human intelligence such as decision making, text processing, and visual perception. AI is a broader field (i.e., the big umbrella that contains several subfields such as machine learning, robotics, and computer vision).



Machine Learning

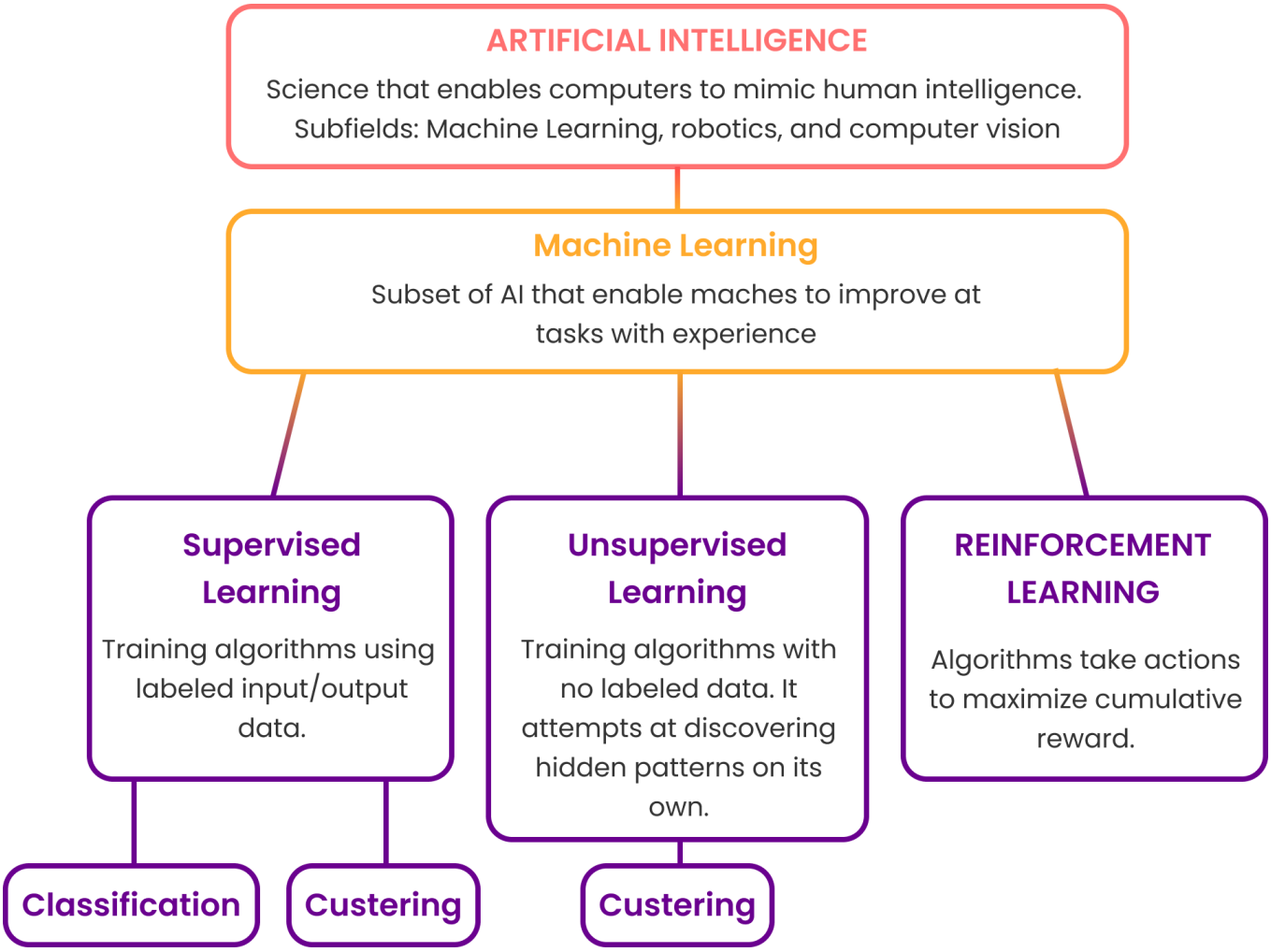
Machine Learning is a subfield of Artificial Intelligence that enables machines to improve at a given task with experience. It is important to note that all machine learning techniques are classified as Artificial Intelligence ones. However, not all Artificial Intelligence could count as Machine Learning since basic Rule-based engines could be classified as AI but they do not learn from experience therefore they do not belong to the machine learning category. Machine learning algorithms are often categorized as supervised, unsupervised and reinforcement.

- **Supervised machine learning algorithms** can apply what has been learned in the past to new data using labelled examples to predict future events. Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values. The system is able to provide targets for any new input after sufficient training. The learning algorithm can also compare its output with the correct, intended output and find errors in order to
modify the model accordingly.

- In contrast, **unsupervised machine learning algorithms** are used when the information used to train is neither classified nor labelled. Unsupervised learning studies how systems can infer a function to describe a hidden structure from unlabelled data. The system doesn't figure out the right output, but it explores the data and can draw inferences from datasets to describe hidden structures from unlabelled data.

- **Reinforcement machine learning algorithms** is a learning method that interacts with its environment by producing actions and discovers errors or rewards. Trial and error

search and delayed reward are the most relevant characteristics of reinforcement learning. This method allows machines and software agents to automatically determine the ideal behaviour within a specific context in order to maximize its performance. Simple reward feedback is required for the agent to learn which action is best; this is known as the reinforcement signal.



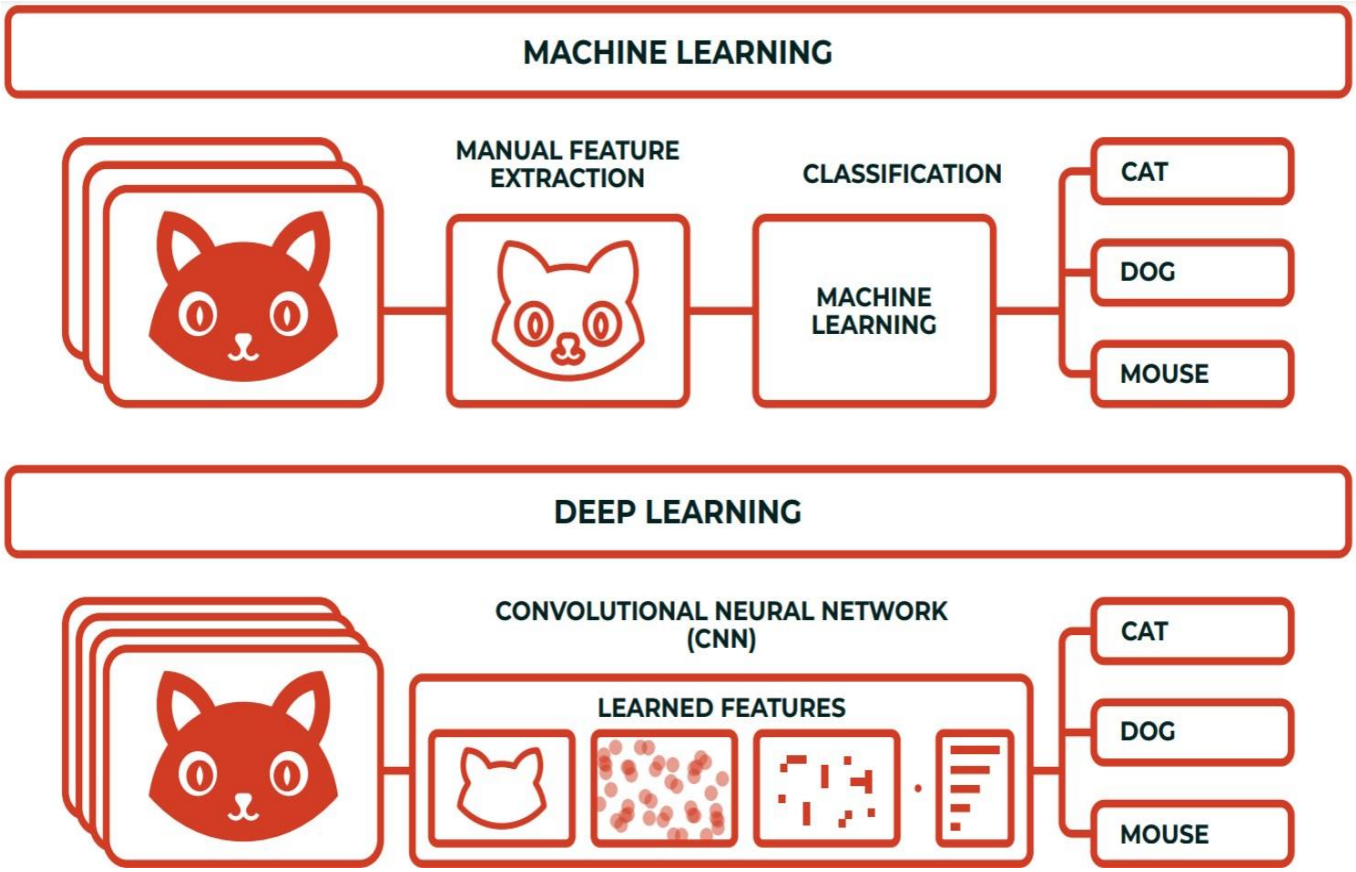
Deep Learning

Deep Learning is a specialized field of Machine Learning that relies on training of Deep Artificial Neural Networks (ANNs) using a large dataset such as images or texts. ANNs are information processing models inspired by the human brain. The human brain consists of billions of neurons that communicate to each other using electrical and chemical signals and enable humans to see, feel, and make decisions. ANNs works by mathematically mimicking the human brain and connecting multiple “artificial” neurons

in a multilayered fashion. The more hidden layers added to the network, the deeper the network gets. What differentiates deep learning from machine learning techniques are in their ability to extract feature automatically as illustrated in the following example:

- **Machine learning process:** (1) selecting the model to train, (2) manually performing feature extraction.

- **Deep Learning Process:** (1) select architecture of the network, (2) features are automatically extracted by feeding in the training data (such as images) along with the target class (label).



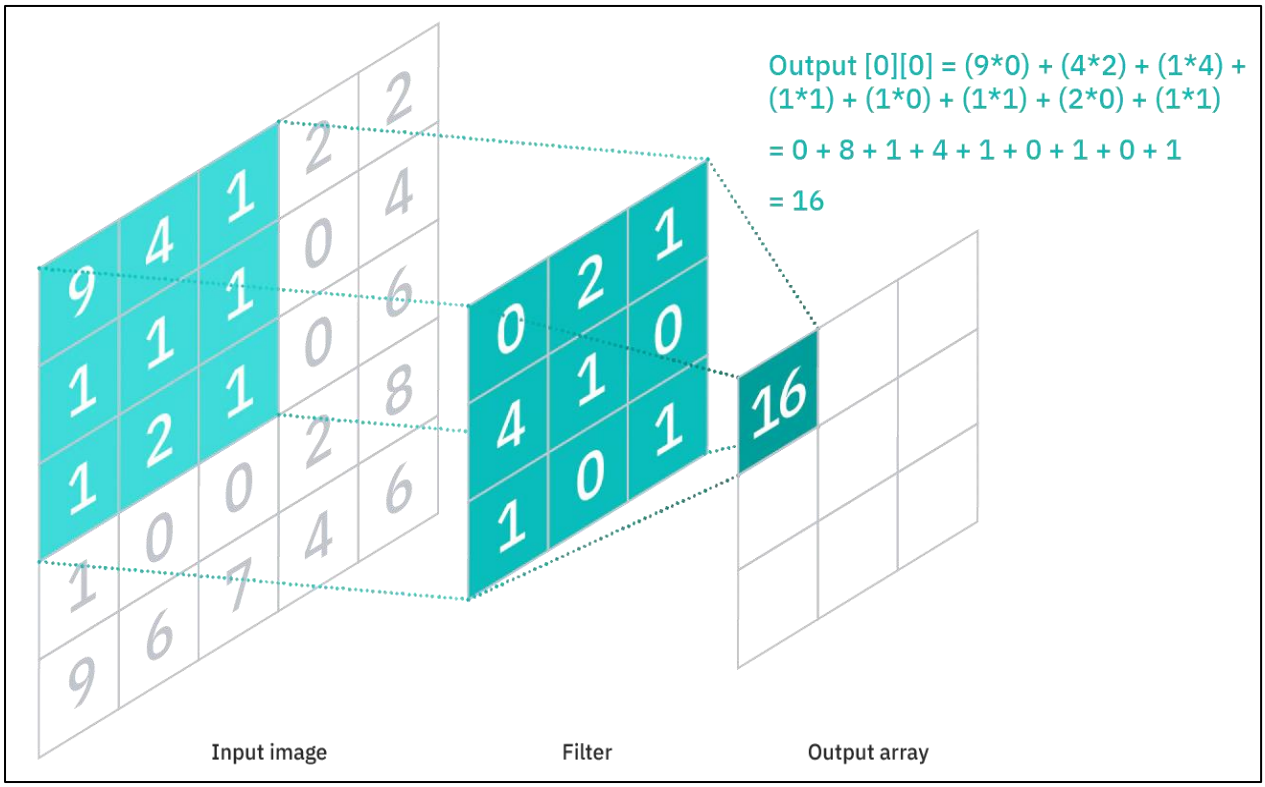
Convolutional Neural Network (CNN)

- Convolutional Neural Networks (CNN) is the most successful Deep Learning method used to process multiple arrays, e.g., 1D for signals, 2D for images and 3D for videos.

- CNN consists of a list of Neural Network layers that transform the input data into an output (class/prediction).

- Development of specialized CNN chips (by NVIDIA, Intel, Samsung etc.) for real-time applications in smartphones, cameras, robots, self-driving cars, etc.

The convolutional layer is the core building block of a CNN, and it is where the majority of computation occurs. It requires a few components, which are input data, a filter, and a feature map. Let's assume that the input will be a color image, which is made up of a matrix of pixels in 3D. This means that the input will have three dimensions—a height, width, and depth—which correspond to RGB in an image. We also have a feature detector, also known as a kernel or a filter, which will move across the receptive fields of the image, checking if the feature is present. This process is known as a convolution. The feature detector is a two-dimensional (2-D) array of weights, which represents part of the image. While they can vary in size, the filter size is typically a 3x3 matrix; this also determines the size of the receptive field. The filter is then applied to an area of the image, and a dot product is calculated between the input pixels and the filter. This dot product is then fed into an output array. Afterwards, the filter shifts by a stride, repeating the process until the kernel has swept across the entire image. The final output from the series of dot products from the input and the filter is known as a feature map, activation map, or a convolved feature.



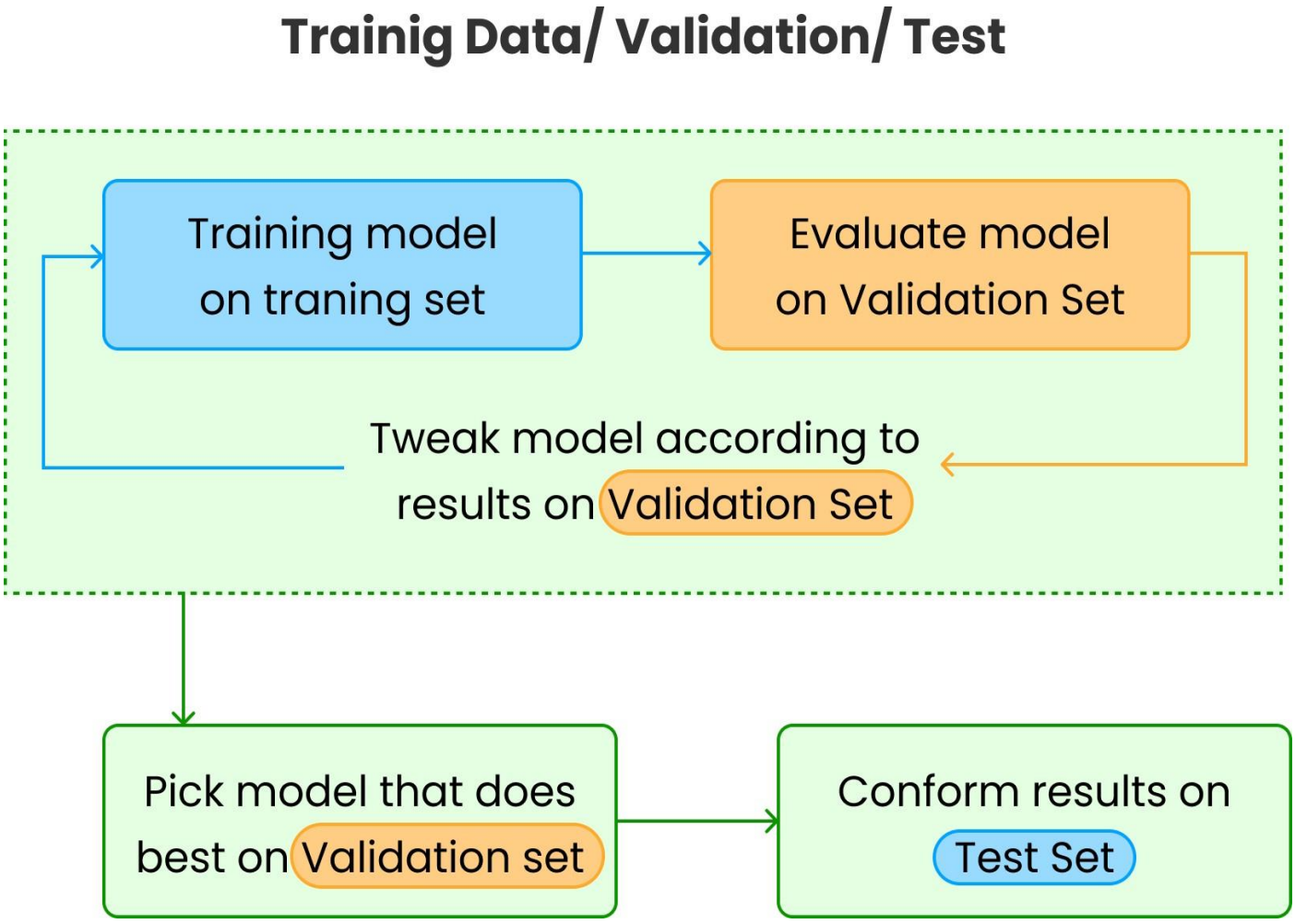
Training, Validation and Testing

- The Training Set

It is the set of data that is used to train and make the model learn the hidden features/patterns in the data.

In each epoch, the same training data is fed to the neural network repeatedly, and the model continues to learn the features of the data.

The Training set should have a diversified set of inputs so that the model is trained in all scenarios and can predict any unseen data sample that may appear in the future.



- The Validation Set

The validation set is a set of data, separate from the training set, that is used to validate our model performance during training.
This validation process gives information that helps us tune the model's hyperparameters and configurations accordingly. It is like a critic telling us whether the training is moving in the right direction or not. The model is trained on the training set and simultaneously the model evaluation is performed on the validation set after every epoch.

The main idea of splitting the dataset into a validation set is to prevent our model from overfitting. I.e., the model becomes really good at classifying the samples in the training set but cannot generalize and make accurate classification on the data it has not seen before.

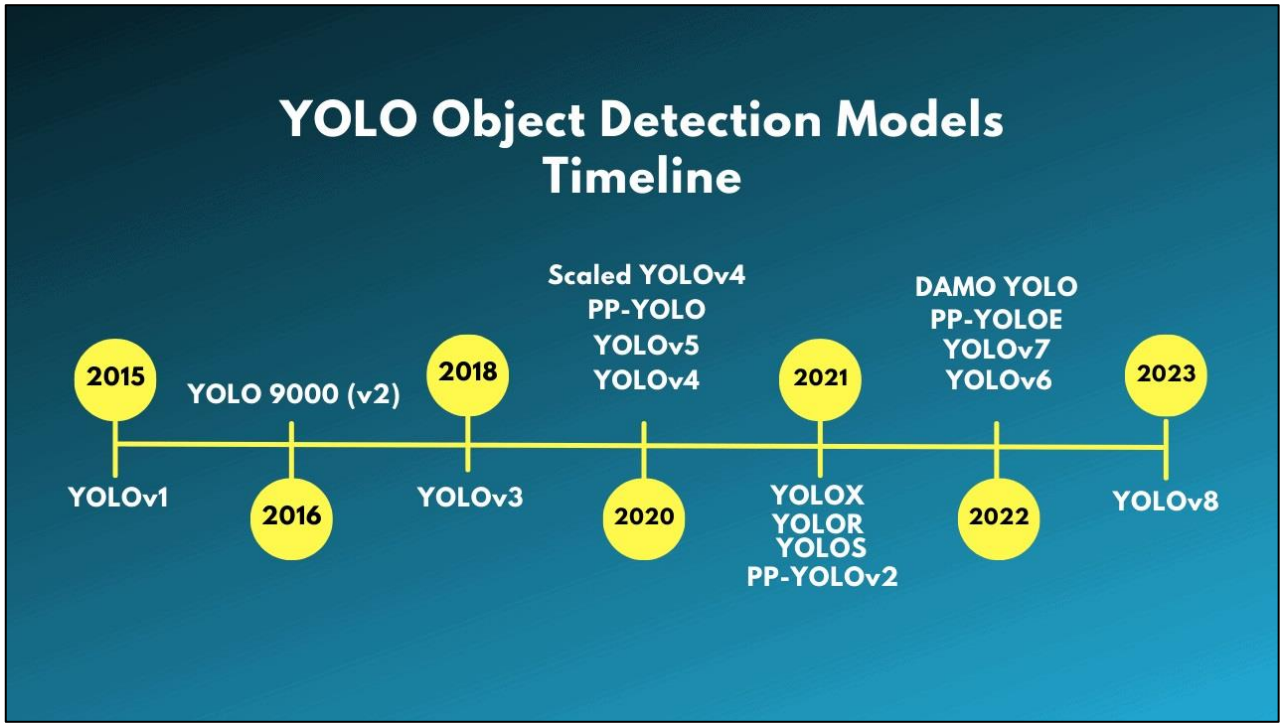
- The Testing Set

The test set is a separate set of data used to test the model after completing the training. It provides an unbiased final model performance metric in terms of accuracy, precision, etc. To put it simply, it answers the question of *"How well does the model perform?"*

YOLOv8 Architecture:

Technically speaking, YOLOv8 is a group of convolutional neural network models, created and trained using the PyTorch framework. YOLOv8 is the latest version of YOLO by **Ultralytics**. As a cutting-edge, state-of-the-art (SOTA) model, YOLOv8 builds on the success of previous versions, introducing new features and improvements for enhanced performance, flexibility, and efficiency. YOLOv8 supports a full range of vision AI tasks, including detection, segmentation, pose estimation, tracking, and classification. This versatility allows users to leverage YOLOv8's capabilities across diverse applications and domains.

YOLOv8 is the latest family of YOLO based Object Detection models from **Ultralytics** providing state-of-the-art performance. Leveraging the previous YOLO versions, the YOLOv8 model is faster and more accurate while providing a unified framework for training models for performing Object Detection, Instance Segmentation, and Image Classification.



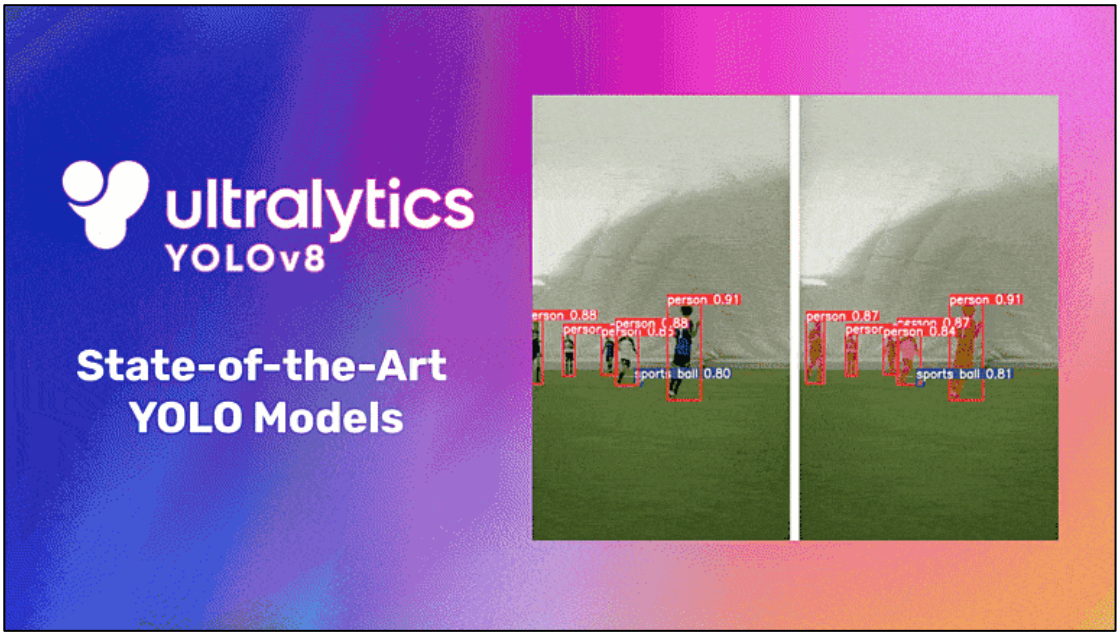
Key Features:

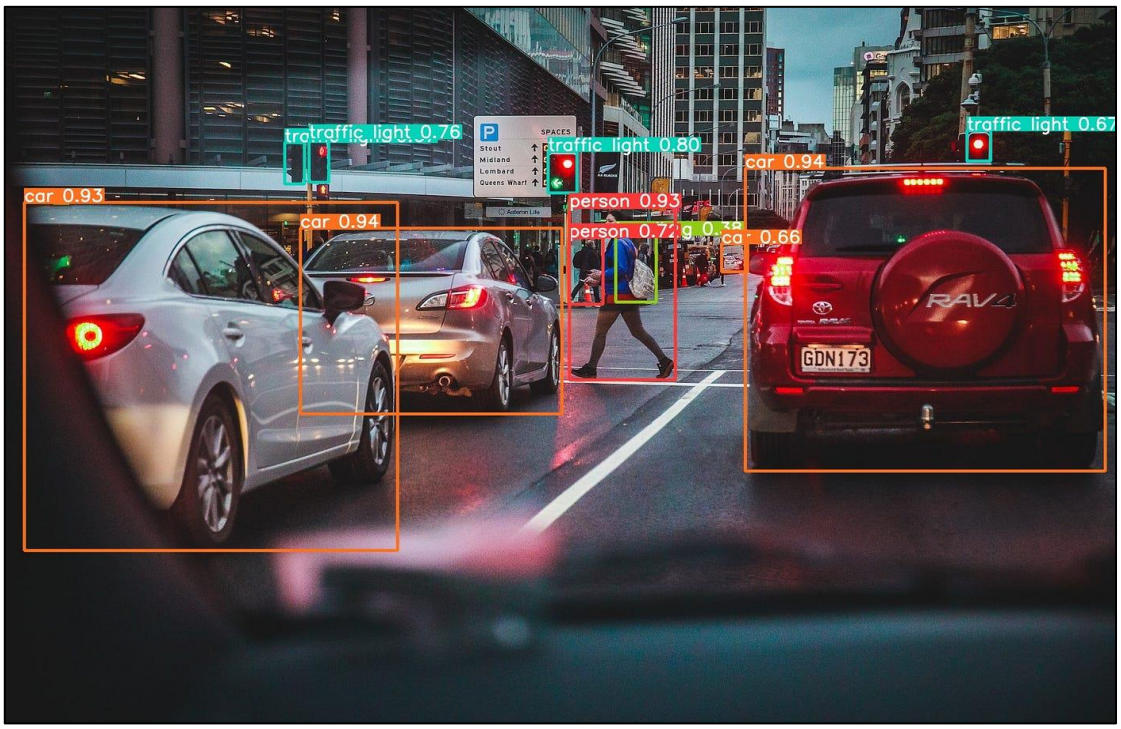
Advanced Backbone and Neck Architectures: YOLOv8 employs state-of-the-art backbone and neck architectures, resulting in improved feature extraction and object detection performance.

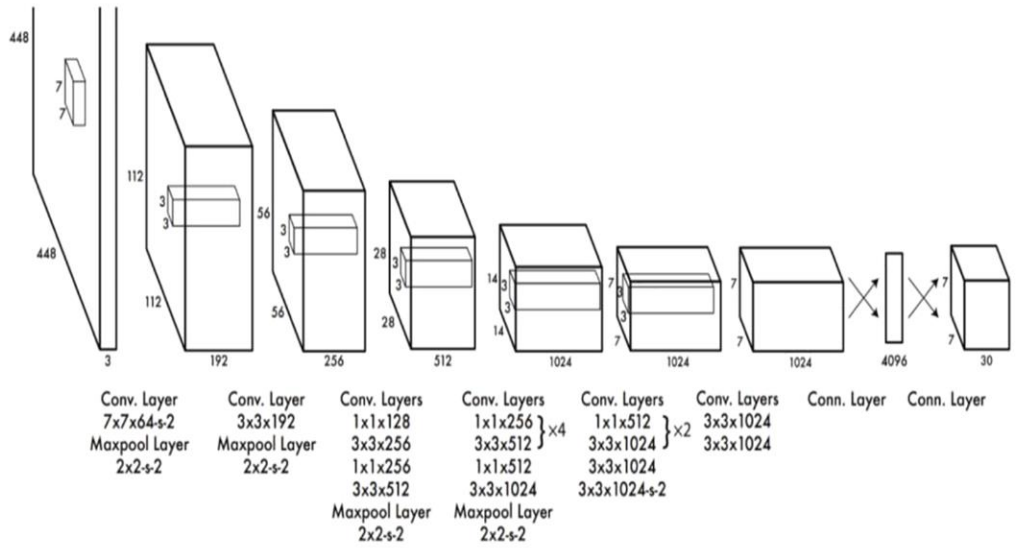
Anchor-free Split Ultralytics Head: YOLOv8 adopts an anchor-free split Ultralytics head, which contributes to better accuracy and a more efficient detection process compared to anchor-based approaches.

Optimized Accuracy-Speed Tradeoff: With a focus on maintaining an optimal balance between accuracy and speed, YOLOv8 is suitable for real-time object detection tasks in diverse application areas.

Variety of Pre-trained Models: YOLOv8 offers a range of pre-trained models to cater to various tasks and performance requirements, making it easier to find the right model for your specific use case.


ultralytics
YOLOv8
State-of-the-Art
YOLO Models
person 0.91
sports ball 0.80
person 0.87
person 0.91
sports ball 0.81


traffic light 0.76
traffic light 0.80
traffic light 0.67
car 0.93
car 0.94
person 0.93
car 0.94
car 0.66
GDN173



The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

Models Available in YOLOv8:

There are five models in each category of YOLOv8 models for detection, segmentation, and classification. YOLOv8 Nano is the fastest and smallest, while YOLOv8 Extra Large (YOLOv8x) is the most accurate yet the slowest among them.

YOLOv8 comes bundled with the following **pre-trained** models:

- **Object Detection** checkpoints trained on the COCO detection dataset with an image resolution of 640.
- **Instance segmentation** checkpoints trained on the COCO segmentation dataset with an image resolution of 640.
- **Image classification** models pretrained on the ImageNet dataset with an image resolution of 224.

- YOLO is a single-shot detector that uses a fully convolutional neural network (CNN) to process an image. You Only Look Once (YOLO) proposes using an end-to-end neural network that makes predictions of bounding boxes and class probabilities all at once. It differs from the approach taken by previous object detection algorithms, which repurposed classifiers to perform detection.

- Following a fundamentally different approach to object detection, YOLO achieved state-of-the-art results, beating other real-time object detection algorithms by a large margin. While algorithms like Faster RCNN work by detecting possible regions of interest using the Region Proposal Network and then performing recognition on those regions separately, YOLO performs all of its predictions with the help of a single fully connected layer.

How does YOLO work? YOLO Architecture

The YOLO algorithm takes an image as input and then uses a simple deep convolutional neural network to detect objects in the image. The architecture of the CNN model that forms the backbone of YOLO is shown below.

- YOLOv8 utilizes a convolutional neural network that can be divided into two main parts: the backbone and the head.

- The head of YOLOv8 consists of multiple convolutional layers followed by a series of fully connected layers. These layers are responsible for predicting bounding boxes, objectness scores, and class probabilities for the objects detected in an image.

YOLOv8 introduces a **new backbone network**, Darknet-53, which is significantly faster and more accurate than the previous backbone used in YOLOv7. DarkNet-53 is a convolutional neural network that is 53 layers deep and can classify images into 1000 object categories.

Additionally, YOLOv8 is more efficient than previous versions because it uses a larger feature map and a more efficient convolutional network. This allows the model to detect objects in a more accurate and faster way. With a larger feature map, the model can capture more complex relationships between different features and can better recognize patterns and objects in the data. A larger feature map also helps to reduce the amount of time it takes to train the model and can help to reduce overfitting.

Overall, YOLOv8 is a powerful and versatile object detection algorithm that can be used in various real-world scenarios to detect and classify objects with high accuracy and speed.

Performance Comparison of YOLOv8 vs YOLOv5

Model Size	Detection#	Segmentation#	Classification*
Nano	+33.21%	+32.97%	+3.10%
Small	+20.05%	+18.62%	+1.12%
Medium	+10.57%	+10.89%	+0.66%
Large	+7.96%	+6.73%	0.00%
Xtra Large	+6.31%	+5.33%	-0.76%

#Image Size = 640 *Image Size = 224

CHAPTER 2

Project Initialization

2.1. Project overview:

Title: "Traffic Monitoring System using Python-OpenCV and YOLOv8"

Main objective and goal of this project is to create a **Traffic Monitoring System using Python-OpenCV and YOLOv8** which will be able to:

1. Detect, track, and count vehicles.

2. Detect vehicle speed.

3. Detect vehicle speed limit violations.

Technically speaking, **YOLOv8** is a group of convolutional neural network models created and trained using the **PyTorch** framework. YOLO (You Only Look Once) is one of the most popular object detection algorithms in the fields of Deep Learning, Machine Learning, and Computer Vision. YOLOv8 is the latest version of the YOLO (You Only Look Once) AI models developed by **Ultralytics**.

YOLOv8 is the latest family of YOLO based Object Detection models from Ultralytics providing state-of-the-art performance. Leveraging the previous YOLO versions, the YOLOv8 model is faster and more accurate while providing a unified framework for training models for performing Object Detection, Instance Segmentation, and Image Classification.

There are five models in each category of **YOLOv8 models** for detection, segmentation, and classification. YOLOv8 Nano is the fastest and smallest, while YOLOv8 Extra Large (YOLOv8x) is the most accurate yet the slowest among them.

YOLOv8 comes bundled with the following pre-trained models:

- **Object Detection** checkpoints trained on the COCO detection dataset with an image resolution of 640.

- **Instance segmentation** checkpoints trained on the COCO segmentation dataset with an image resolution of 640.
- **Image classification** models pretrained on the ImageNet dataset with an image resolution of 224.

We have used the YOLOv8's pretrained model (e.g., **yolov8s.pt**). All YOLOv8 models for object detection are already pre-trained on the **COCO dataset**, which is a huge collection of images of 80 different types.

2.1.1. Existing system:

The existing traffic monitoring systems typically use radar, lidar, or cameras to detect vehicles. These systems are expensive to install and maintain, and they can only detect vehicles in a limited area.

2.1.2. Proposed system:

The proposed system is a more cost-effective and efficient way to monitor traffic. It uses **Python-OpenCV and YOLOv8** to detect, count and track vehicles in the video footage. The system can also detect vehicle speed and detects if a vehicle is violating the speed limit.

2.1.3. Feasibility Study:

The feasibility of the project is analyzed in this phase and business proposal is put forth with a very general plan for the project and some cost estimates. During system analysis the feasibility study of the proposed system is to be carried out. This is to ensure that the proposed system is not a burden to the company. For feasibility analysis, some understanding of the major requirements for the system is essential.
Three key considerations involved in feasibility study are:

- Economic Feasibility
- Technical Feasibility
- Social Feasibility

Economic Feasibility:

This study is carried out to check the economic impact that the system will have on the organization. The amount of fund that the company can pour into the research and development of the system is limited. The expenditures must be justified.

Thus, the developed system as well within the budget and this was achieved most of the technologies used are freely available.

Technical Feasibility:

This study is carried out to check the technical feasibility, that is, the technical requirements of the system. Any system developed must not have a high demand on the available technical resources. This will lead to high demands on the available technical resources. This will lead to high demands being placed on the client. The developed system must have a modest requirement, as only minimal or null changes are required for implementing this system.

Social Feasibility:

The aspect of study is to check the level of acceptance of the system by the user. This includes the process of training the user to use the system efficiently. The user must not feel threatened by the system, instead must accept it as a necessity. The level of acceptance by the users solely depends on the methods that are employed to educate the user about the system and to make him familiar with it. His level of confidence must be raised so that he is also able to make some constructive criticism, which is welcomed, as he is the final user of the system.

2.1.4. Module overview:

The system consists of the following modules:

- **Vehicle detection module:** This module uses YOLOv8 to detect vehicles in the video footage.
- **Vehicle tracking module:** This module tracks the vehicles that have been detected by the vehicle detection module.
- **Vehicle speed detection module:** This module estimates the speed of the vehicles that are being tracked.
- **Vehicle counting module:** This module counts the number of vehicles that pass through a specific area in the video footage.

- **Vehicle speed violation detection module:** This module detects if a vehicle is violating the speed limit.

2.1.5. Module functionalities:

The following are the functionalities of each module:

- **Vehicle detection module:** This module uses **YOLOv8** to detect vehicles in the video footage. YOLOv8 is a deep learning algorithm that can detect objects in images and videos. The algorithm is trained on a dataset of images that contain vehicles. When the algorithm is applied to a video footage, it can detect the vehicles in the video.
- **Vehicle tracking module:** This module tracks the vehicles that have been detected by the vehicle detection module. The module uses a **centroid tracking algorithm** to track the vehicles. The centroid tracking algorithm tracks the centre of mass of the vehicles in the video footage.
- **Vehicle speed detection module:** This module estimates the speed of the vehicles that are being tracked. To estimate the speed of the vehicles, it uses the distance between the two regions of interest (ROI) chosen and the elapsed time taken by the vehicles to cover that distance.
- **Vehicle counting module:** This module counts the number of vehicles that pass through a specific area in the video footage. The module uses a region of interest (ROI) to define the specific area. The module then counts the number of vehicles that enter and exit the ROI.
- **Vehicle speed violation detection module:** This module detects if a vehicle is violating the speed limit. The module uses the speed of the vehicles that are being tracked to determine if they are violating the speed limit.

2.2. System Requirements:

1. Functional Requirements

The TMS shall be able to:

- Detect vehicles of different types, including cars, motorcycles, buses, and trucks.
- Track vehicles across multiple frames.

- Count vehicles in real time.
- Detect vehicle speed in any direction.
- Detect vehicle speed limit violations by comparing vehicle speed to the speed limit.

2. Non-Functional Requirements

The TMS shall be:

- **Accurate:** The TMS shall be able to detect, track, and count vehicles with high accuracy.
- **Real-time:** The TMS shall be able to process video streams and generate results in real time.
- **Scalable:** The TMS shall be able to scale to handle multiple video streams and large numbers of vehicles.
- **User-friendly:** The TMS shall be easy to use and configure.

3. Hardware Requirements

- Processor: intel i5 processor
- RAM: 4GB or more
- Hard Disk: 500GB or more

4. Acceptance Criteria

The TMS will be accepted when it meets the following criteria:

- The TMS can detect, track, and count vehicles with an accuracy of at least 90%.
- The TMS can detect vehicle speed with an accuracy of at least 90%.
- The TMS can detect vehicle speed limit violations with an accuracy of at least 80%.
- The TMS can process video streams and generate results in real time.
- The TMS is easy to use and configure.

Chapter 3: Design & Diagram

Video Footage

Data Cleaning

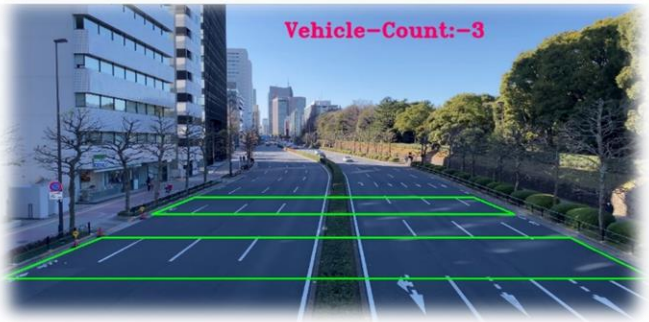
Input Video-Footage (.mp4)

Pretrained Model

Traffic Monitoring System using Python-OpenCV & YOLOv8

Import the YOLOv8's pre-trained model i.e., yolov8s.pt

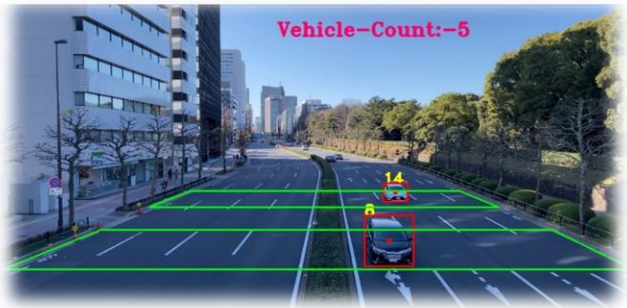
YOLOv8



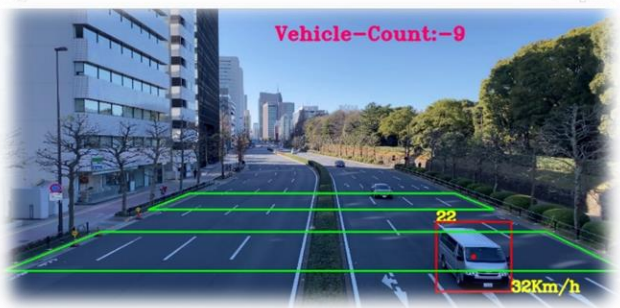
Region of Interest (ROI) (Drawn on the Input Video Footage using OpenCV)

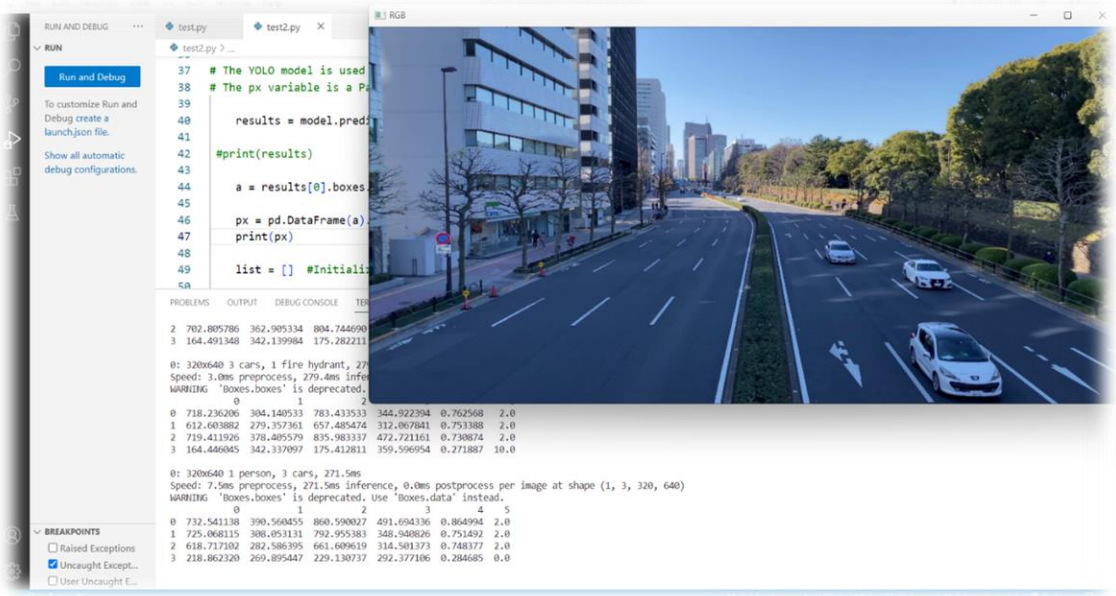
Implement Centroid-Tracking Algorithm

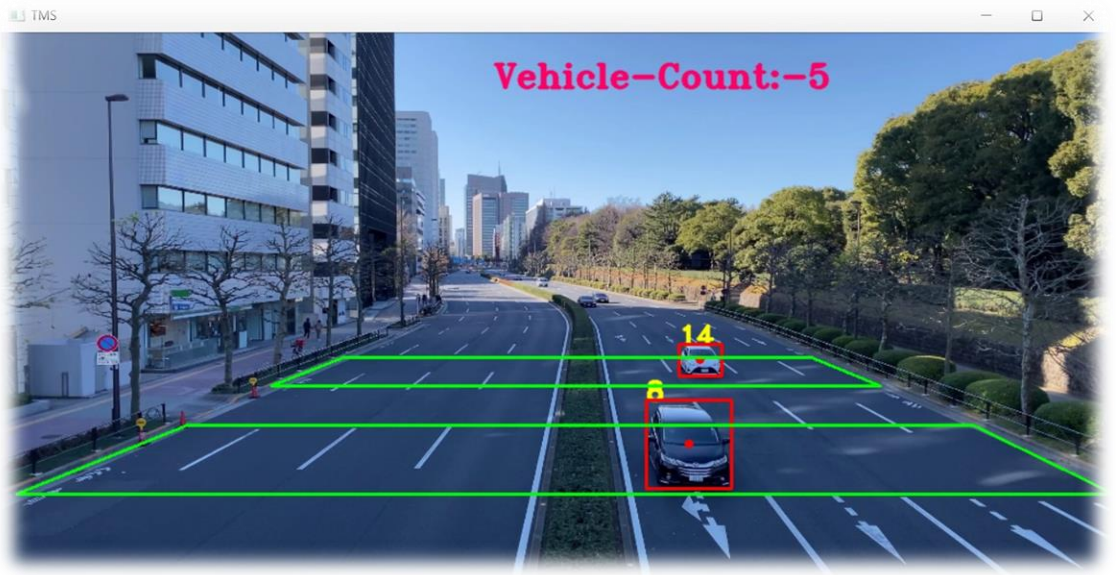
Vehicle Detection, Tracking, and Counting



Vehicle Speed Detection







Centroid-Tracking Algorithm

1. **Bounding Box Co-ordinates obtained fromYOLOv8 Model Predictions**

2. **Tracker Module (Implements Centroid Tracking Algorithm) Calculates Centroid (cx, cy) of each Vehicles**

rect = bounding box co-ordinates of each vehicle obtained from YOLOv8 pretrained model predictions
x, y, w, h = rect
cx = (x + x + w) // 2
cy = (y + y + h) // 2

3. **Calculates distance between every pair of possible centroids of moving vehicles**

dist = math.hypot(cx - pt[0], cy - pt[1])

// Uses Eucledian Diatance formula

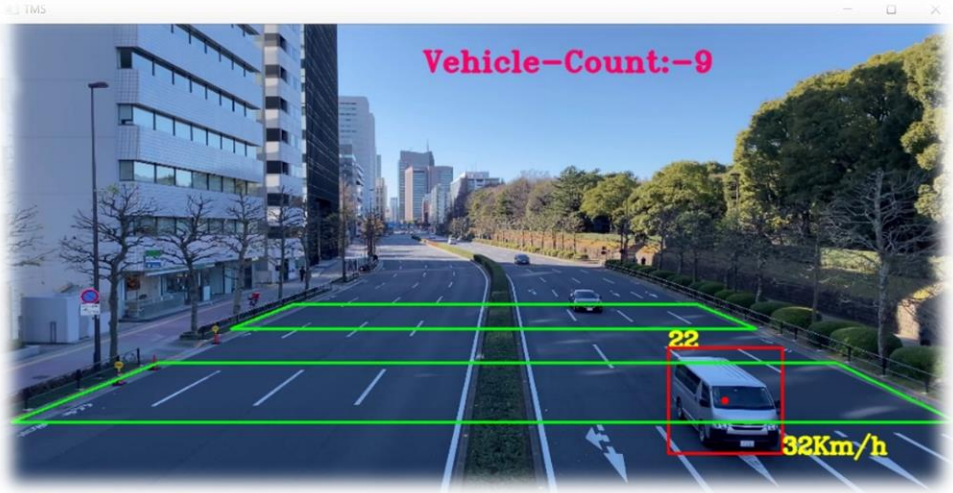
// If this distance is less than a threshold value, the object or vehicle is considered to be the same otherwise the vehicle will be considered as different

// same objects will be assigned same ID and different objects will be assigned different IDs

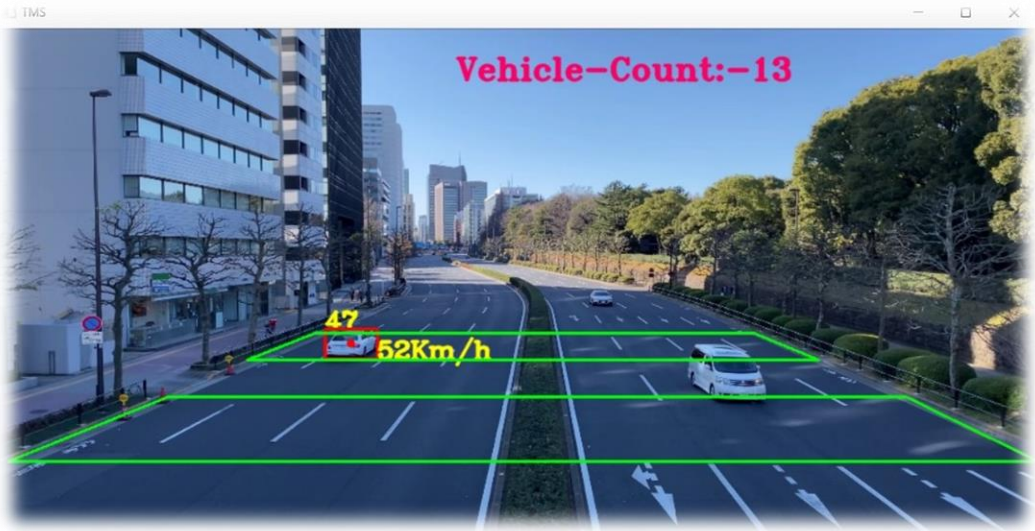
4. **Assigns ID to every bounding box it detects**

5. **Returns bounding box co-ordinates and object ID assigned**

We have used the YOLOv8's pretrained model (e.g., yolov8s.pt). All YOLOv8 models for object detection are already pre-trained on the COCO dataset, which is a huge collection of images of 80 different types.

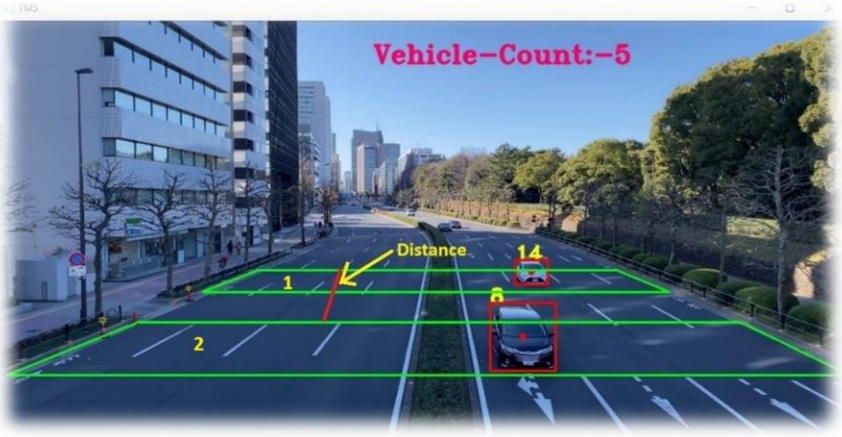


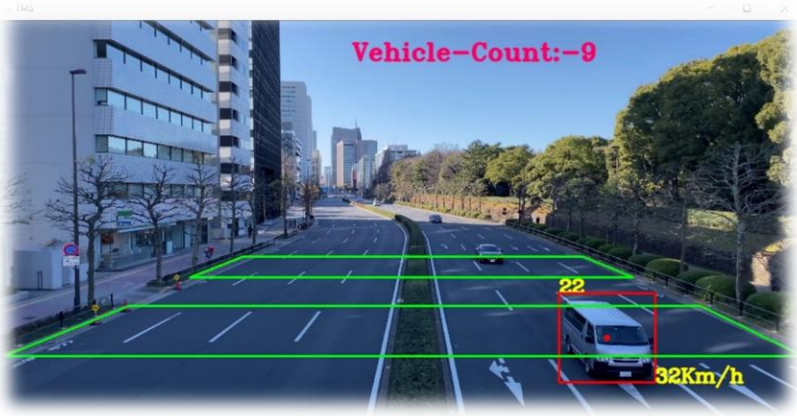
Vehicle Speed Detection

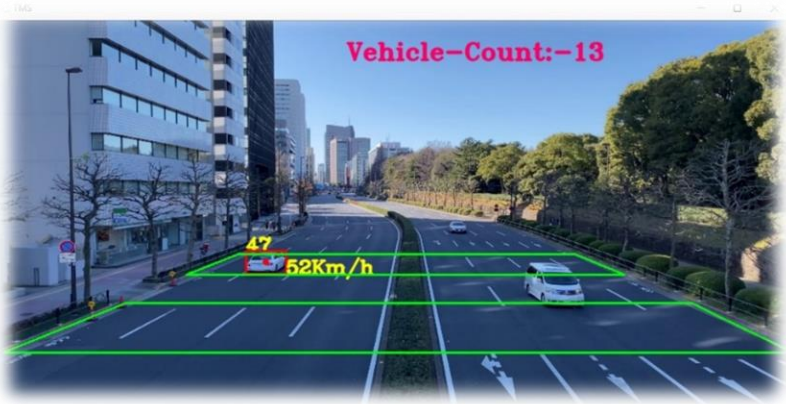


Vehicle Speed Detection (Vehicles moving in another direction)

The speed of the vehicles that are being tracked can be estimated using the distance between the starting line of first Region of Interest (ROI) and starting line of second Region of Interest (ROI) and the time taken by the vehicle to cover the distance.





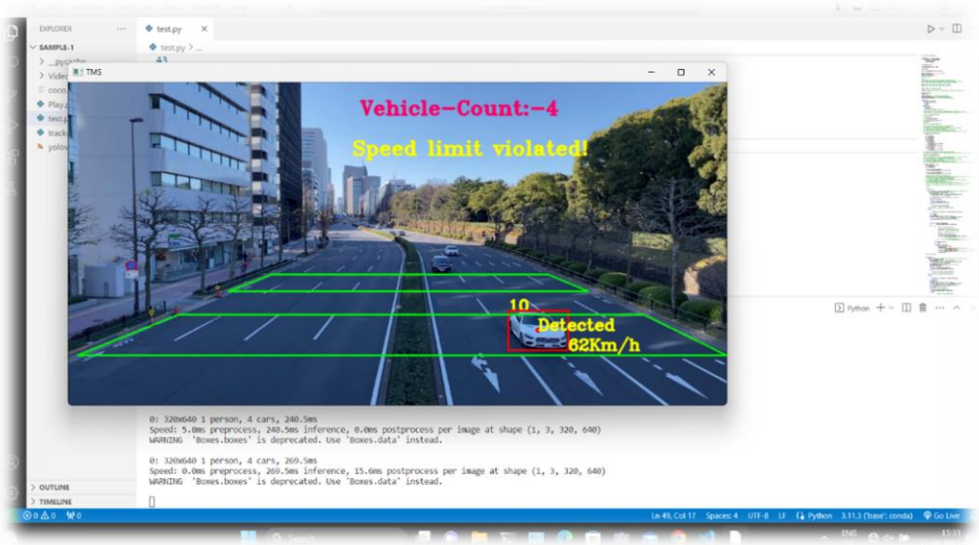


1. **Import Tracker Module (Returns bounding box co-ordinates and assigned IDs of the vehicles).**
2. **Import required Python Libraries (NumPy, Pandas, OpenCV).**
3. **Import YOLOv8's Pretrained Model, i.e., yolov8s.pt**
4. **The centroid tracking algorithm implemented using the Tracker Module returns bounding box co-ordinates and object ID assigned.**
5. **counts the number of vehicles that passes through the second region of interest (ROI) [This is the main ROI for vehicle counting].**

6. **Initializes dictionaries to store IDs and corresponding elapsed time taken by the vehicles to cover the distance between the two Regions of Interest (ROI) chosen.**

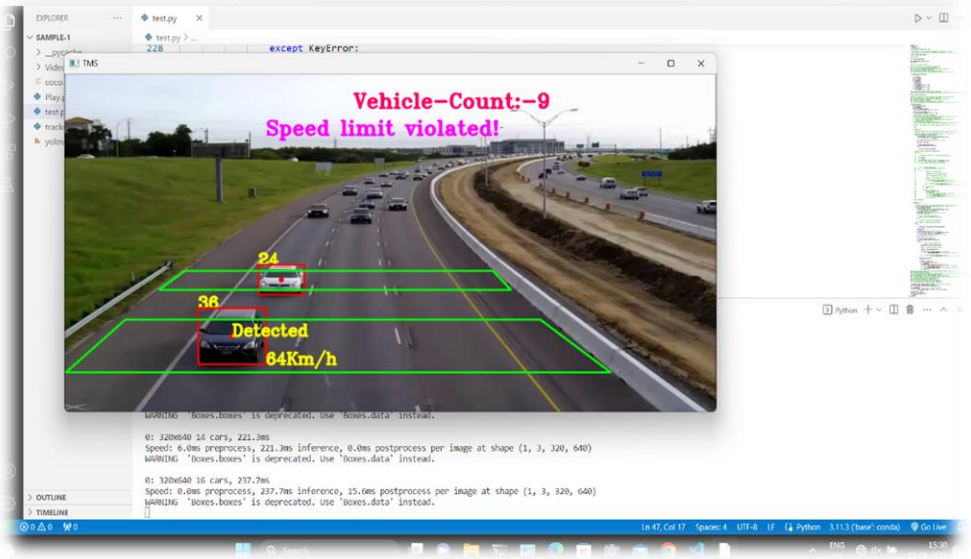
$$Speed = \frac{distance}{Elapsed\ Time}$$

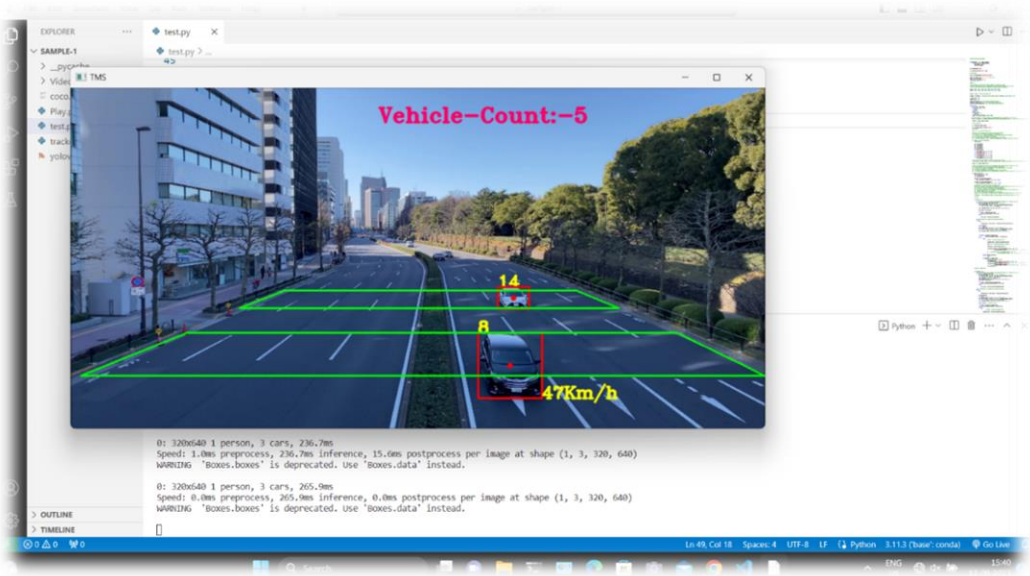
7. **Checks if the vehicle is forward moving vehicles (moving from first Region of Interest / ROI to second Region of Interest / ROI). Display the vehicle speed, Id, and bounding box for each detected vehicle. For the backward moving vehicles (moving from second Region of Interest / ROI to first Region of Interest / ROI), it displays the bounding box, ID, and vehicle speed when the vehicle passes the first Region of Interest / ROI.**

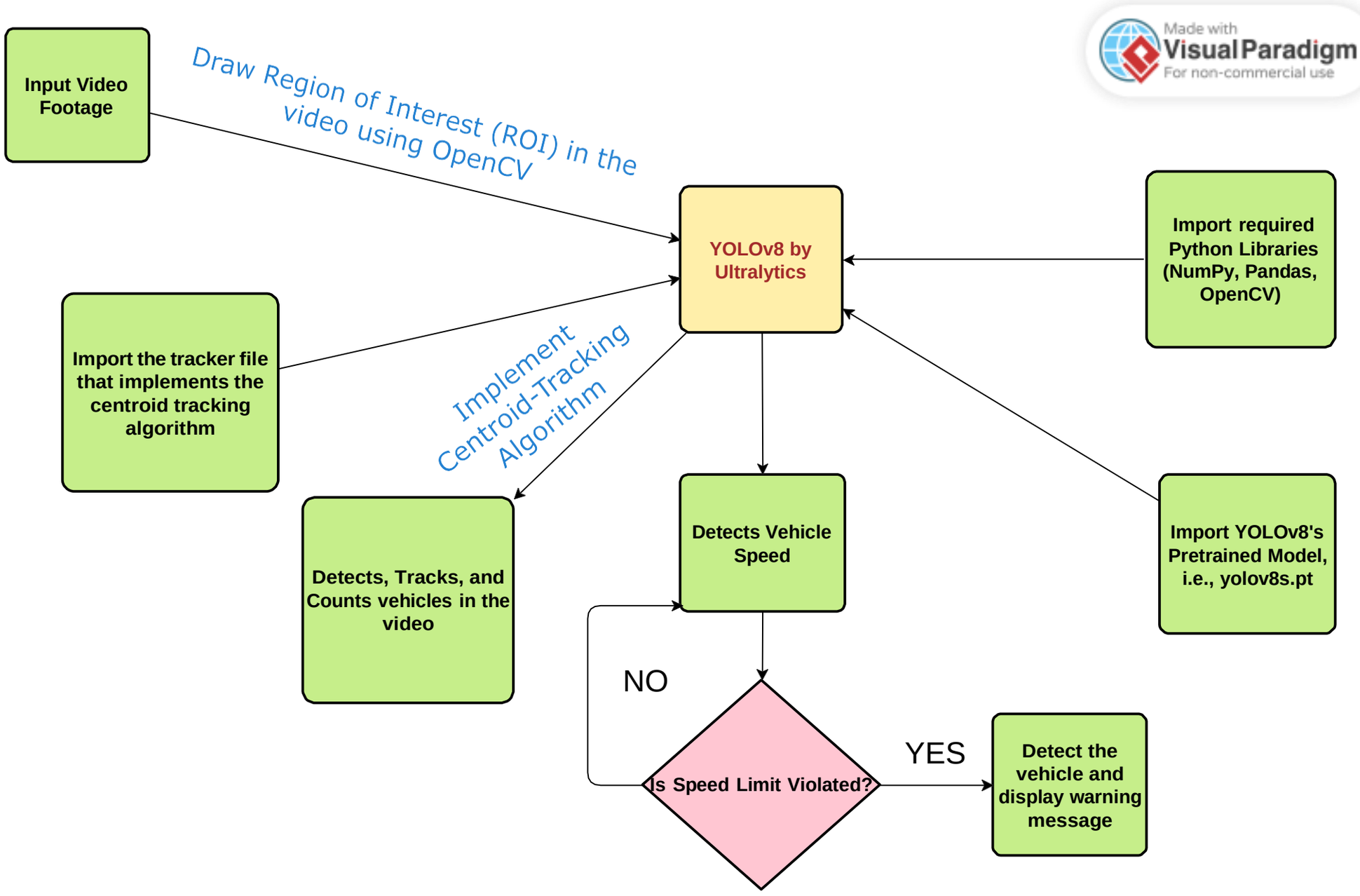


Let, speed_limit = 60 #Set Speed Limit

A warning message will be displayed if the system detects any vehicle speed limit violations

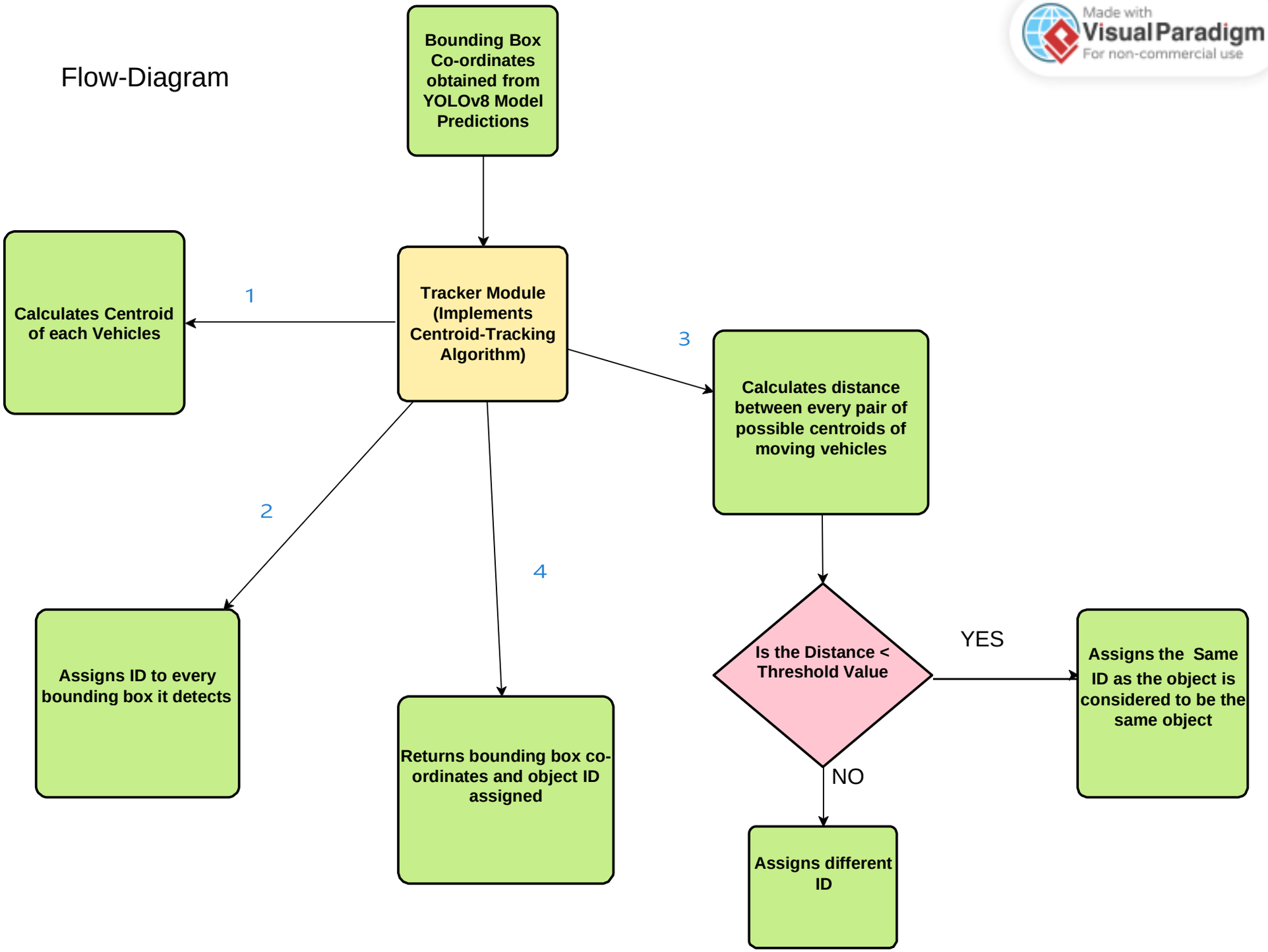


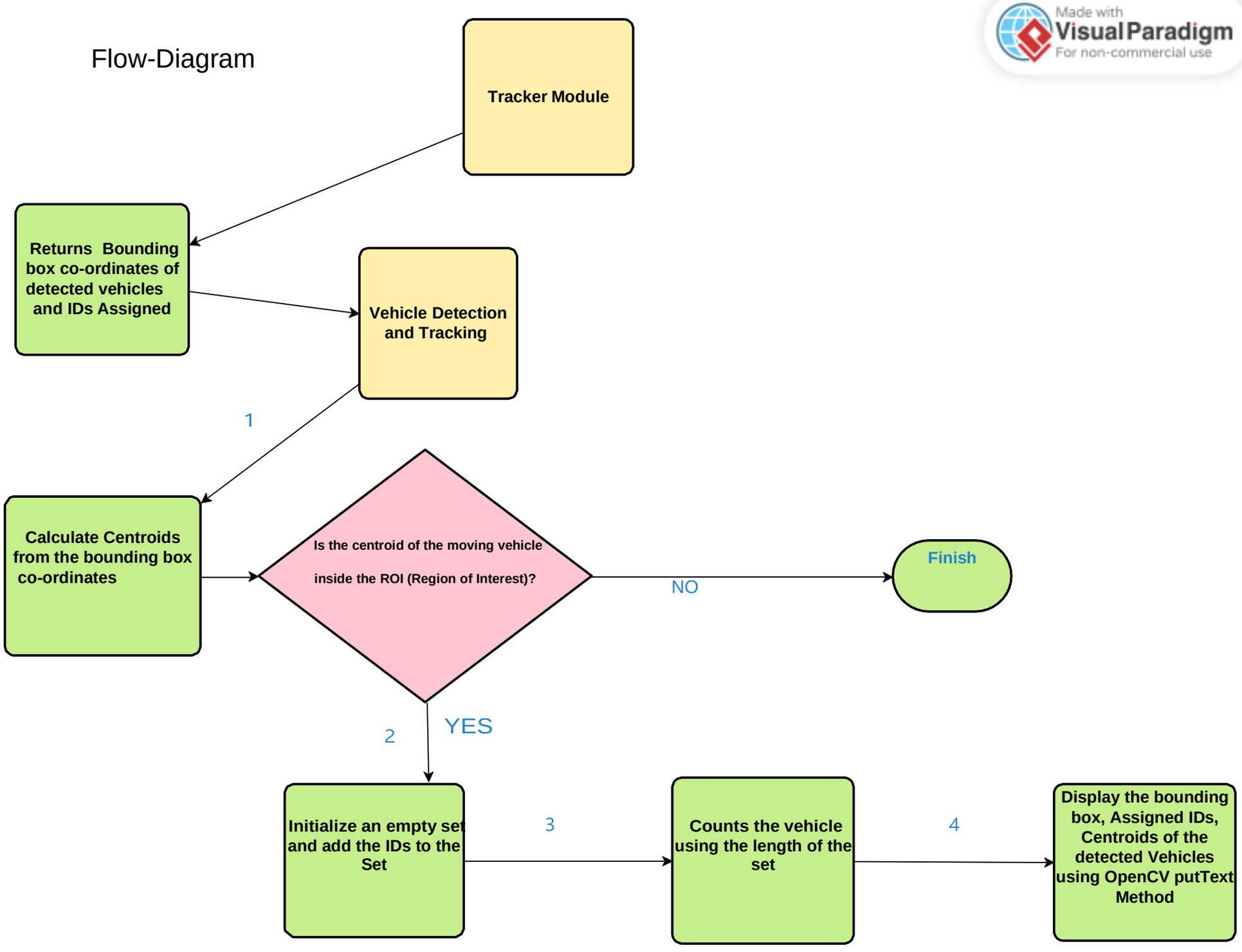




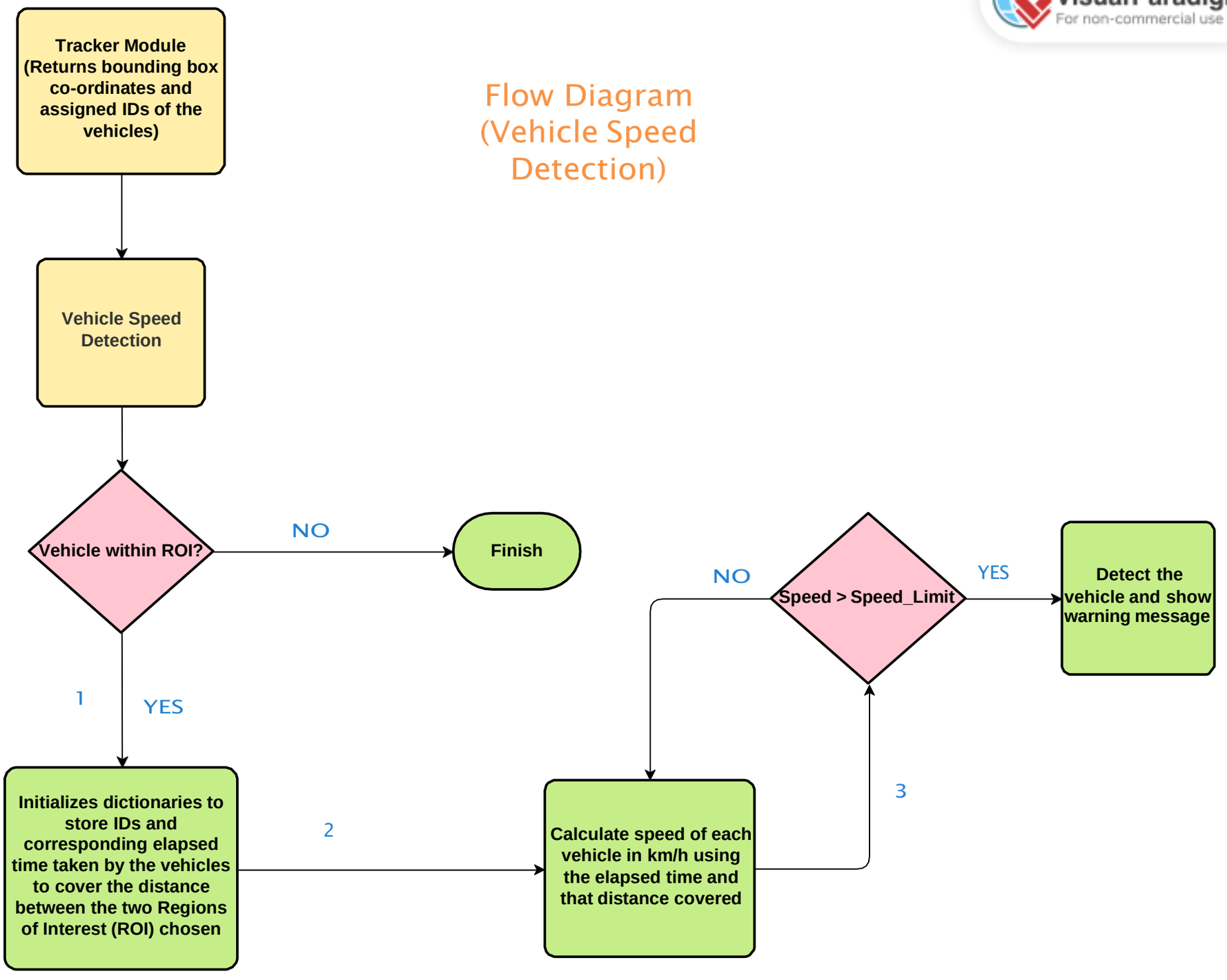
Flow-Diagram

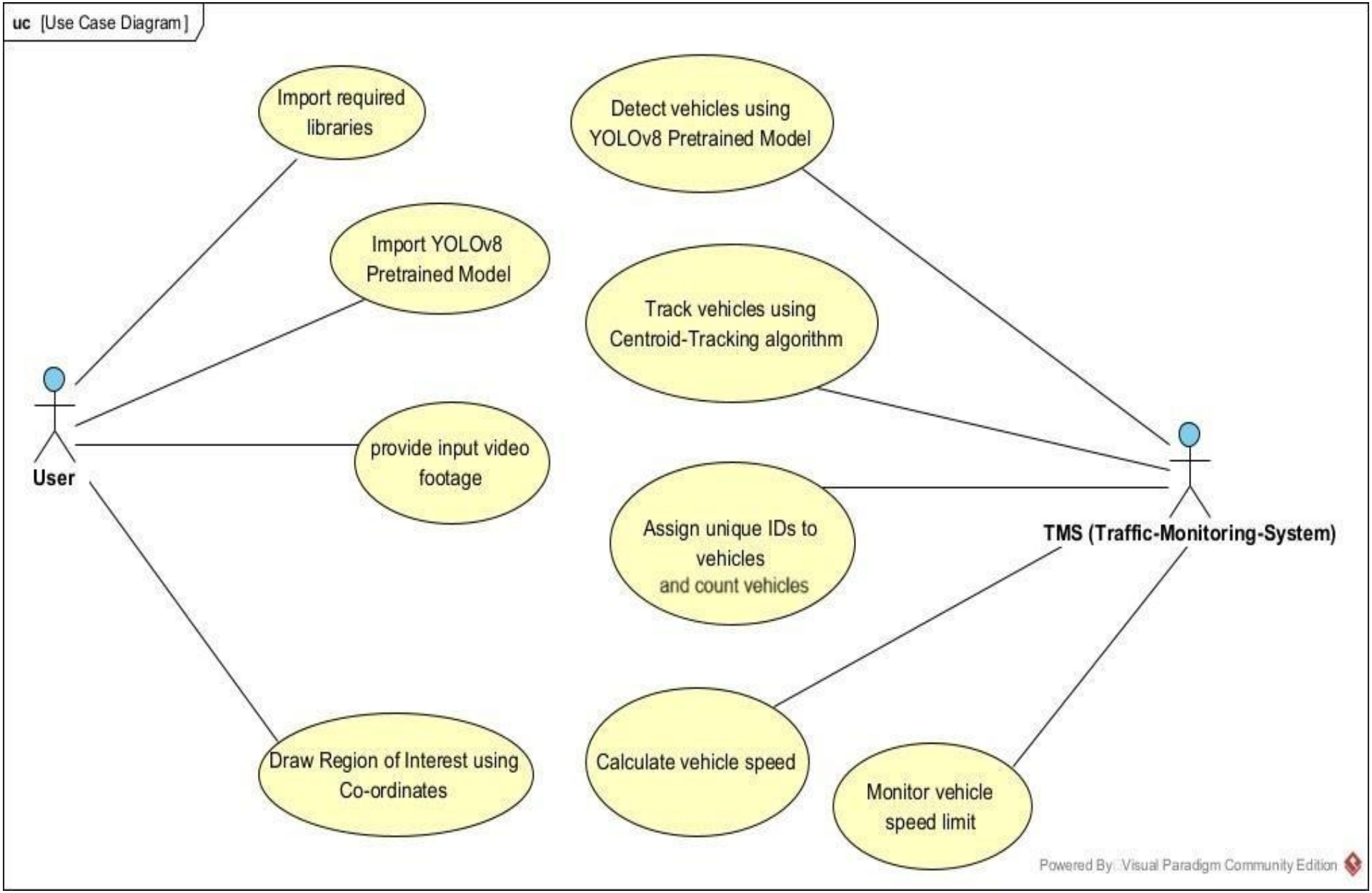
Flow-Diagram

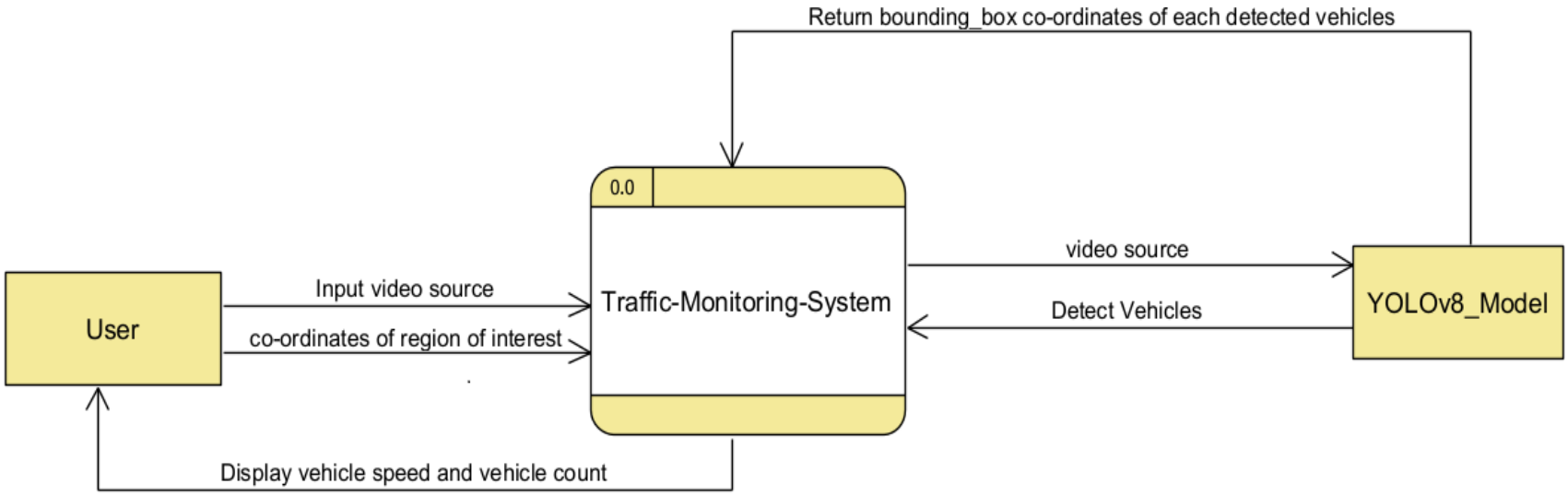




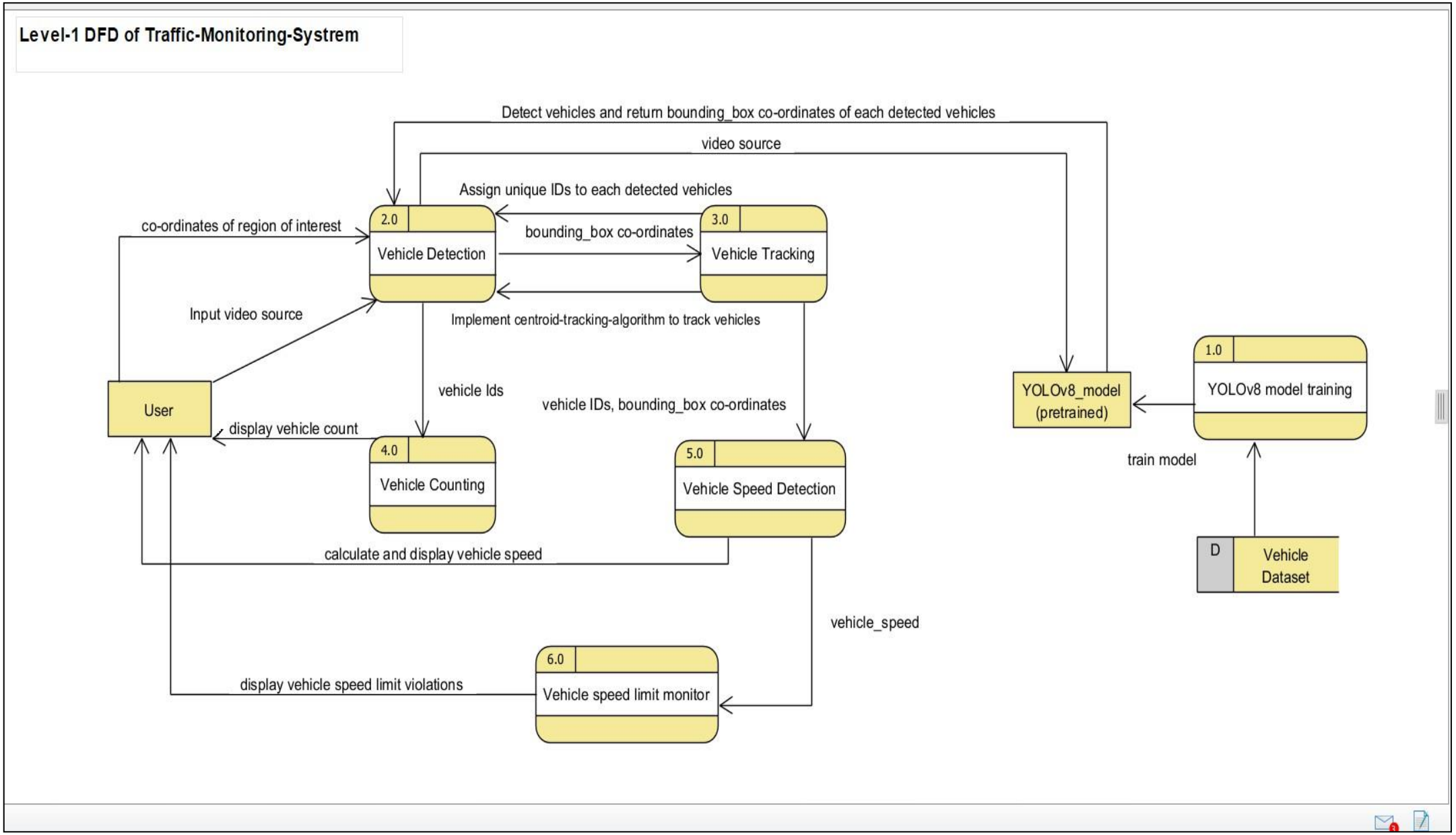
Flow Diagram (Vehicle Speed Detection)

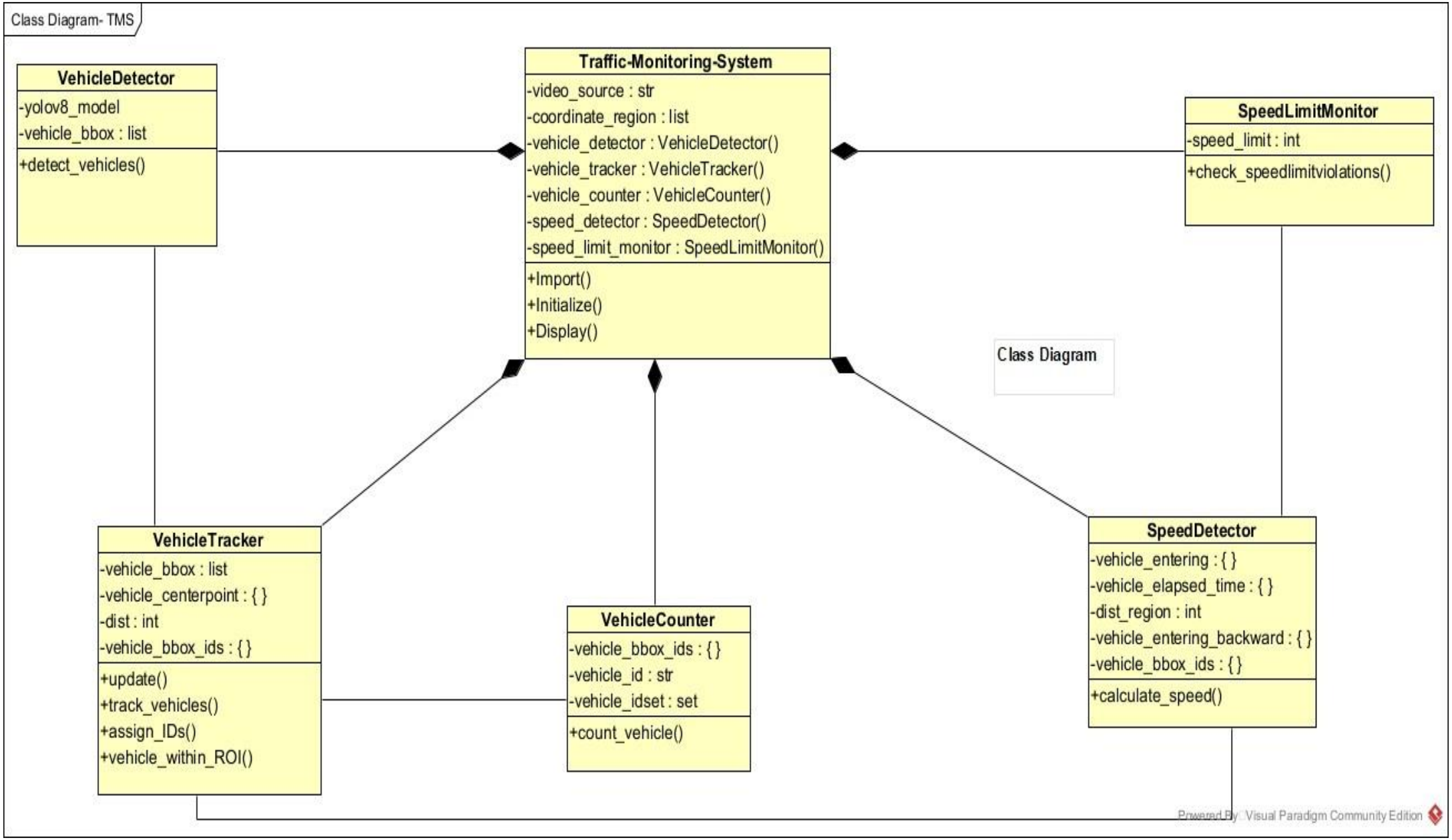



uc [Use Case Diagram]
Import required libraries
Import YOLOv8 Pretrained Model
provide input video footage
Draw Region of Interest using Co-ordinates
Detect vehicles using YOLOv8 Pretrained Model
Track vehicles using Centroid-Tracking algorithm
Assign unique IDs to vehicles and count vehicles
Calculate vehicle speed
Monitor vehicle speed limit
User
TMS (Traffic-Monitoring-System)
Powered By Visual Paradigm Community Edition


Return bounding_box co-ordinates of each detected vehicles
0.0
User
Input video source
co-ordinates of region of interest
Traffic-Monitoring-System
video source
Detect Vehicles
YOLOv8_Model
Display vehicle speed and vehicle count

Level-0 DFD


Level-1 DFD of Traffic-Monitoring-Systrem
Detect vehicles and return bounding_box co-ordinates of each detected vehicles
video source
Assign unique IDs to each detected vehicles
co-ordinates of region of interest
2.0
Vehicle Detection
bounding_box co-ordinates
3.0
Vehicle Tracking
Input video source
Implement centroid-tracking-algorithm to track vehicles
1.0
YOLOv8 model training
YOLOv8_model
(pretrained)
vehicle Ids
vehicle IDs, bounding_box co-ordinates
User
display vehicle count
4.0
Vehicle Counting
5.0
Vehicle Speed Detection
train model
D
Vehicle
Dataset
calculate and display vehicle speed
vehicle_speed
6.0
Vehicle speed limit monitor
display vehicle speed limit violations


Class Diagram- TMS
VehicleDetector
-yolov8_model
-vehicle_bbox : list
+detect_vehicles()
Traffic-Monitoring-System
-video_source : str
-coordinate_region : list
-vehicle_detector : VehicleDetector()
-vehicle_tracker : VehicleTracker()
-vehicle_counter : VehicleCounter()
-speed_detector : SpeedDetector()
-speed_limit_monitor : SpeedLimitMonitor()
+Import()
+Initialize()
+Display()
SpeedLimitMonitor
-speed_limit : int
+check_speedlimitviolations()
Class Diagram
VehicleTracker
-vehicle_bbox : list
-vehicle_centerpoint : { }
-dist : int
-vehicle_bbox_ids : { }
+update()
+track_vehicles()
+assign_IDs()
+vehicle_within_ROI()
VehicleCounter
-vehicle_bbox_ids : { }
-vehicle_id : str
-vehicle_idset : set
+count_vehicle()
SpeedDetector
-vehicle_entering : { }
-vehicle_elapsed_time : { }
-dist_region : int
-vehicle_entering_backward : { }
-vehicle_bbox_ids : { }
+calculate_speed()
Powered By Visual Paradigm Community Edition

CHAPTER 4

Methodology

Technologies:

• **Programming Language:** Python (Version-3.11)

• **Platform:** Anaconda (Python Distribution) and Visual Studio Code IDE.

• **Libraries:** Pandas (1.5.3), NumPy (1.24.3), Python-OpenCV (4.8.0.74) & Ultralytics (8.0.147)

Step-1. Collect the video footage: The video footage can be collected from a variety of sources, such as a traffic camera, a dashcam, or a security camera. The video footage should be in a format that can be read by OpenCV.

Video Footage

Step-2. Data Cleaning: This is useful if you only want to monitor a specific area of the road. Data cleaning is the process of removing errors and inconsistencies from data. It is an important step in any data analysis project, as it can significantly improve the accuracy and reliability of the results.

In the context of traffic monitoring system, data cleaning can be used to remove the following types of errors and inconsistencies from video footage:

- **Cropping:** This involves removing unwanted parts of the video, such as the edges or the sky. This can be useful for focusing on specific areas of interest, such as intersections or traffic lanes.
- **Noise removal:** This involves removing unwanted noise from the video, such as camera shake or shadows. This can improve the clarity of the video and make it easier to identify objects.
- **Brightness and contrast adjustment:** This involves adjusting the brightness and contrast of the video to improve the visibility of objects.
- **Denoising:** This involves removing noise from the video, such as compression artifacts or Gaussian noise. This can improve the quality of the video and make it easier to identify objects.
- **Segmentation:** This involves dividing the video into different segments, such as frames or objects. This can be useful for processing the video more efficiently or for identifying specific objects.

Step-3. Import the necessary libraries/modules: The following libraries/modules are necessary for this project:

- **Python-OpenCV: OpenCV** is a Python open-source library, which is used for computer vision in Artificial intelligence, Machine Learning, face recognition, etc.
- **https://opencv.org/**
- **pip install opencv-python**
- **Pandas: Pandas** is a library for python programming language. It is used for create, remove, edit and data manipulation in dataframe.
- **https://pandas.pydata.org/**
- **pip install pandas**
- **NumPy: NumPy** is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

- **https://numpy.org/**
- **pip install numpy**
- **Ultralytics: Ultralytics YOLOv8** is a cutting-edge, state-of-the-art (SOTA) model that builds upon the success of previous YOLO versions and introduces new features and improvements to further boost performance and flexibility. YOLOv8 is designed to be fast, accurate, and easy to use, making it an excellent choice for a wide range of object detection and tracking, instance segmentation, image classification and pose estimation tasks.
- **https://docs.ultralytics.com/**
- **pip install ultralytics**
- **Tracker:** This is a custom module based on the Centroid tracking algorithm to track the vehicles.

Step-4. Import the pretrained YOLOv8 model: The pretrained YOLOv8 model (e.g., **yolov8s.pt**) can be downloaded from the YOLO website. All YOLOv8 models for object detection are already pre-trained on the **COCO dataset,** which is a huge collection of images of 80 different types. The custom dataset, if required, can also be created by collecting a set of images that contain vehicles. The images should be labelled with the bounding boxes of the vehicles.

```
import cv2
import pandas as pd
import numpy as np
from ultralytics import YOLO
import time
from math import dist
from tracker import*
model = YOLO('yolov8s.pt')  # Load Pretrained Model
```

Step-5. Draw the region of interest (ROI): The ROI can be drawn on the video using the Python OpenCV "polylines" method. The ROI should be large enough to capture all of the vehicles that you want to monito

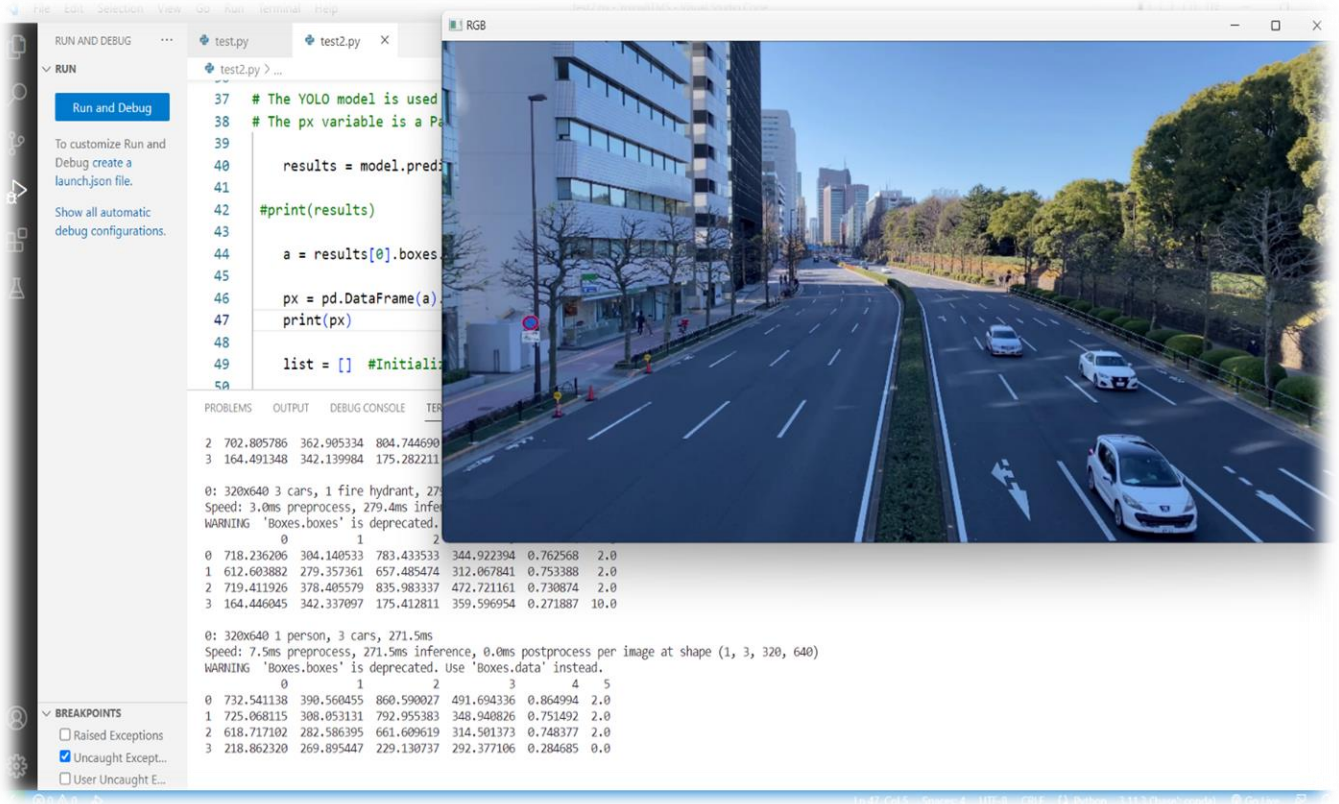
ROI (Region of Interest) [Green Rectangular Box]:

```
# get the mouse coordinates


def RGB(event, x, y, flags, param):
    if event == cv2.EVENT_MOUSEMOVE:
        colorsBGR = [x, y]
        print(colorsBGR)


cv2.namedWindow('TMS')

cv2.setMouseCallback('TMS', RGB)

# read Video

cap = cv2.VideoCapture('Videos/vid1.mp4')
```

```
# Co-ordinates of the desired region (Region of Interest or ROI).
# above co-ordinates can vary according to the input video-footage or test cases.
# So, we have to put proper co-ordinates using the mouse co-ordinate.

area = [(314, 297), (742, 297), (805, 323), (248, 323)]
area2 = [(171, 359), (890, 359), (1019, 422), (15, 422)]
```

```
cv2.polylines(frame, [np.array(area, np.int32)],
                        True, (0, 255, 0), 2)  # Area-1
cv2.polylines(frame, [np.array(area2, np.int32)],
                        True, (0, 255, 0), 2)  # Area-2 | Main Area
```

Step-6. Use the YOLOv8 model to detect vehicles in the ROI: The YOLOv8 model can be used to detect vehicles in the ROI. The model will output a list of bounding boxes for the detected vehicles.



```
while True:
    ret, frame = cap.read()
    # Check if frame is read
    if not ret:
        break
    count += 1
    # Skip Frames
    if count % 3 != 0:
        continue
    frame = cv2.resize(frame, (1020, 500))

   # The YOLO model is used to predict the bounding boxes of the objects in the frame.
   # The px variable is a Pandas DataFrame that stores the bounding boxes in a format that is easy to iterate over.

    results = model.predict(frame)

    # print(results)

    a = results[0].boxes.boxes

    px = pd.DataFrame(a).astype("float")
    # print(px)

    list = []  # Initialize empty List
```

Step-7. Use the centroid tracking algorithm to track the vehicles that have been detected: The centroid tracking algorithm can be used to track the vehicles that have been detected by YOLOv8. The algorithm will track the centre of mass of the vehicles in the video footage.

- **Track the vehicles and assign unique IDs to them:**

- This method first creates an empty List to store the unique IDs of the vehicles. Then, it loops through the frames of the video and detects vehicles in each frame. For each detected vehicle, it assigns a unique ID and updates the tracking information for the vehicle. Finally, it returns the List of unique IDs for the vehicles.

Centroid tracking algorithm:

Popular Tracking Algorithms

Multiple tracking algorithms depend on complexity and computation.

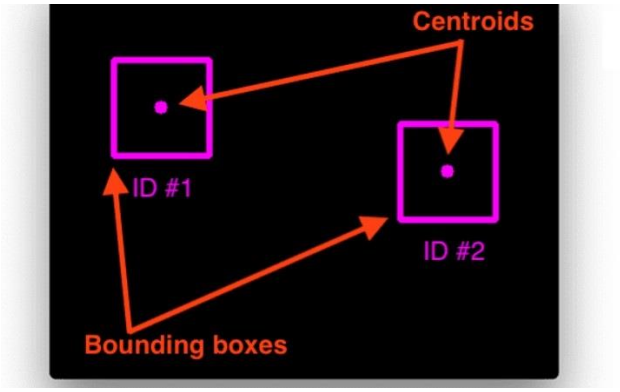
- **DEEP SORT**: DEEP SORT is one of the most widely used tracking algorithms, and it is a very effective object tracking algorithm since it uses deep learning techniques. it works with YOLO object detection and uses Kalman filters for its tracker.

- **Centroid Tracking algorithm:** The centroid Tracking algorithm is a combination of multiple-step processes. It uses a simple calculation to track the point using euclidean distance.

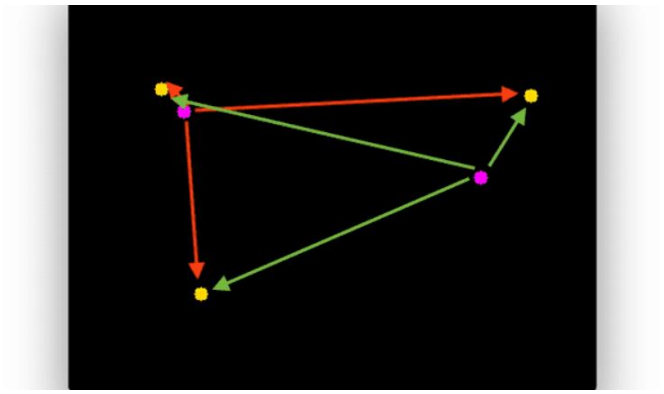
  In this project, we will be using Centroid Tracking Algorithm to build our tracker.

- Steps Involved in Centroid Tracking Algorithm:

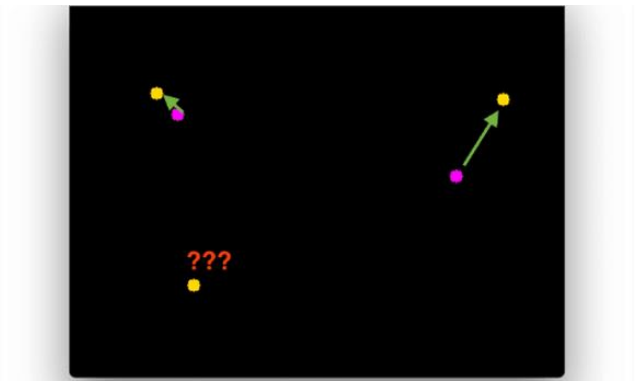
1. Calculate the Centroid of detected objects using the bounding box coordinates



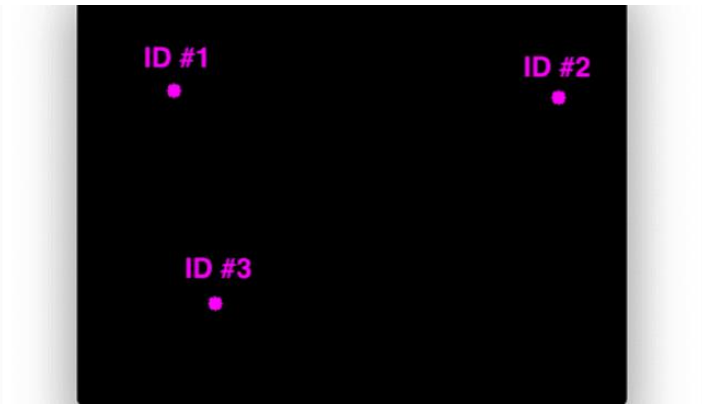
2. For every ongoing frame, it does the same; it computes the centroid by using the coordinates of the bounding box and assigns an id to every bounding box it detects. Finally, it computes the Euclidean distance between every pair of centroids possible.



3. We assume that the same object will be moved the minimum distance compared to other centroids, which means the two pairs of centroids having minimum distance in subsequent frames are considered to be the same object.



4. Now it's time to assign the IDs to the moved centroid that will indicate the same object.



The Tracker Module using Centroid Tracking Algorithm. (Tracker.py)

- **update** → It accepts an array of bounding box coordinates.
- tracker outputs a list containing [x,y,w,h,object_id].
- x,y,w,h are the bounding box coordinates, and object_id is the id assigned to that particular bounding box.
- After creating the tracker, we need to implement an object detector, and we will use the tracker in that.

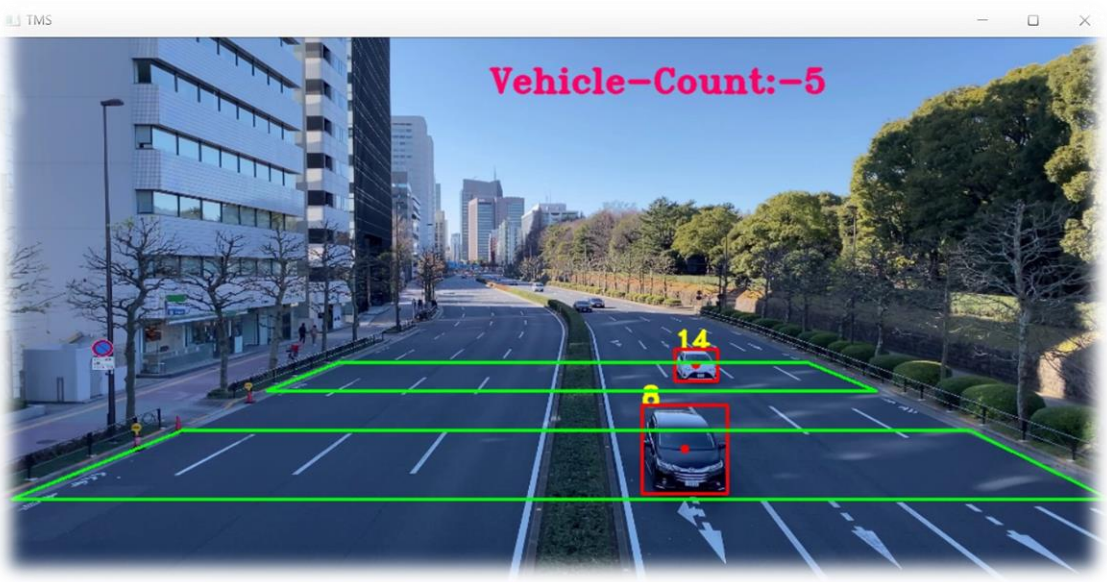
```
tracker = Tracker() # Initialize the Tracker object. (Defined in the tracker file, tracker.py)
```

```
for index, row in px.iterrows():
    # print(row)

    x1 = int(row[0])
    y1 = int(row[1])
    x2 = int(row[2])
    y2 = int(row[3])
    d = int(row[5])
    c = class_list[d]
    if 'car' in c:
        list.append([x1, y1, x2, y2])
    if 'motorcycle' in c:
        list.append([x1, y1, x2, y2])
    elif 'truck' in c:
        list.append([x1, y1, x2, y2])
    elif 'bus' in c:
        list.append([x1, y1, x2, y2])

# This line calls the update() method of the Tracker object.
# The update() method takes a list of bounding boxes as input and returns a list of bounding boxes with the IDs

bbox_id = tracker.update(list)
```

```python
import math

# ////////////////////////////Tracker Module//////////////////////////////////////////////////////////////////////////////\

#Centroid Tracking algorithm
#Eucledian Distance Tracker


class Tracker:
    def __init__(self):
        # Store the center positions of the objects
        #The self.center_points variable is a dictionary that stores the center positions of the objects that have been detected.
        #The key of the dictionary is the ID of the object, and the value is the tuple of the x and y coordinates of the object's center point.
        self.center_points = {}
        # Keep the count of the IDs
        #The self.id_count variable is a counter that keeps track of the ID of the next object that is detected.
        # each time a new object id detected, the count will increase by one
        self.id_count = 0

    def update(self, objects_rect):
        # Objects boxes and ids
        #The objects_bbs_ids variable is a list that will store the bounding boxes and IDs of the objects that have been detected in the current frame.
        objects_bbs_ids = []

        # Get center point of new object
        #The for loop iterates over the list of objects, objects_rect.
        #For each object, the code calculates the center point of the object by taking the average of the x and y coordinates of the top-left and bottom-right corners of the object's bounding box.
        #The center point is then assigned to the variable cx and cy.

        #The objects_bbs_ids variable is initialized to an empty list.
        #• The for loop iterates over the list of objects, objects_rect.
        #• The rect variable is a tuple that contains the x, y, w, and h coordinates of the current object's bounding box.
        #• The x, y, w, and h variables are assigned the values from the rect tuple.
        #• The cx variable is assigned the average of the x and w coordinates.
        #• The cy variable is assigned the average of the y and h coordinates.
        #• The (cx, cy) tuple is appended to the objects_bbs_ids list.

        for rect in objects_rect:
            x, y, w, h = rect
            cx = (x + x + w) // 2
            cy = (y + y + h) // 2

            # Find out if that object was detected already
            #The same_object_detected variable is a boolean variable that is used to keep track of whether or not the current object has been detected before.
            #The for loop iterates over the dictionary self.center_points.
            #For each object in the dictionary, the code calculates the distance between the center point of the current object and the center point of the object in the dictionary.
            #If the distance is less than a threshold value (50), then the code concludes that the same object has been detected and the same_object_detected variable is set to True.
            #The object's bounding box and ID are then appended to the objects_bbs_ids list.
            #The Threshold value (50) can vary according to the input video-footage or test cases.

            same_object_detected = False
            for id, pt in self.center_points.items():
                dist = math.hypot(cx - pt[0], cy - pt[1])

                if dist < 55:
                    self.center_points[id] = (cx, cy)
                    print(self.center_points)
                    objects_bbs_ids.append([x, y, w, h, id])
                    same_object_detected = True
                    break

            # New object is detected we assign the ID to that object
            ##The same_object_detected variable is a boolean variable that is used to keep track of whether or not the current object has been detected before.
            #The if statement checks if the same_object_detected variable is False.
            #If it is, then the code assigns a new ID to the object and appends the object's bounding box and ID to the objects_bbs_ids list.

            if same_object_detected is False:
                self.center_points[self.id_count] = (cx, cy)
                objects_bbs_ids.append([x, y, w, h, self.id_count])
                self.id_count += 1

        # Clean the dictionary by center points to remove IDS not used anymore
        #The new_center_points dictionary is a new dictionary that will be used to store the center points of the objects that have been detected in the current frame.
        #The for loop iterates over the list of objects, objects_bbs_ids.
        #For each object, the code retrieves the object's ID and center point from the self.center_points dictionary.
        #The center point is then added to the new_center_points dictionary.

        new_center_points = {}
        for obj_bb_id in objects_bbs_ids:
            _, _, _, _, object_id = obj_bb_id
            center = self.center_points[object_id]
            new_center_points[object_id] = center

        # Update dictionary with IDs not used removed
        #The self.center_points dictionary is updated with the new_center_points dictionary.
        #The new_center_points dictionary contains the center points of the objects that have been detected in the current frame.
        #The return statement returns the list of objects, objects_bbs_ids.

        self.center_points = new_center_points.copy()
        return objects_bbs_ids

# ////////////////////////////Tracker Module//////////////////////////////////////////////////////////////////////////////
```

Step-8. Count the number of vehicles that pass through the ROI: The number of vehicles that pass through the ROI (Region of Interest) can be counted by Initializing an empty set. We have selected the second region of Interest as the main area which will be used to count the vehicles passing through it. The ID of each vehicle is added to the empty set and then the vehicle count is calculated.

```
area_c = set()  # Initialize empty Set
area_c.add(id)  # Vehicle-counter

cnt = len(area_c)
    cv2.putText(frame, ('Vehicle-Count:-')+str(cnt), (452, 50),
                cv2.FONT_HERSHEY_TRIPLEX, 1, (102, 0, 255), 2,
cv2.LINE_AA)
```



Vehicle Detection, Tracking and Counting

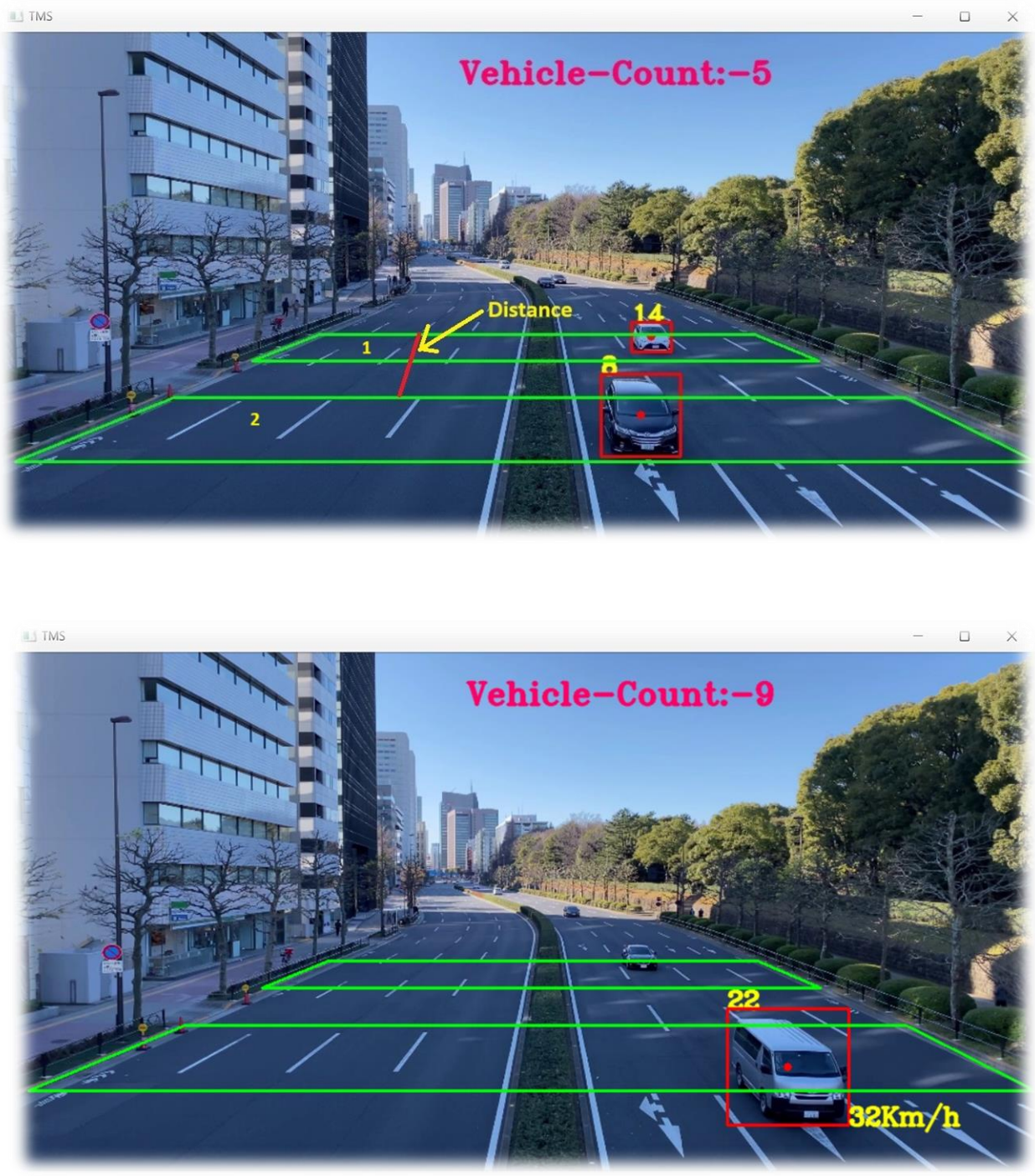
Step-9. Estimate the speed of the vehicles that are being tracked: The speed of the vehicles that are being tracked can be estimated using the distance between the starting line of first Region of Interest (ROI) and starting line of second Region of Interest (ROI) and the time taken by the vehicle to cover the distance. We have Initialized different python dictionaries to store IDs of each vehicle and their elapsed time to cover the distance and then we calculated the speed of vehicles coming from any direction.

```
vehicles_entering = {}  # Initialize empty dictionary
vehicles_elapsed_time = {}  # Initialize empty dictionary
vehicles_entering_backward = {}  # Initialize empty dictionary
```

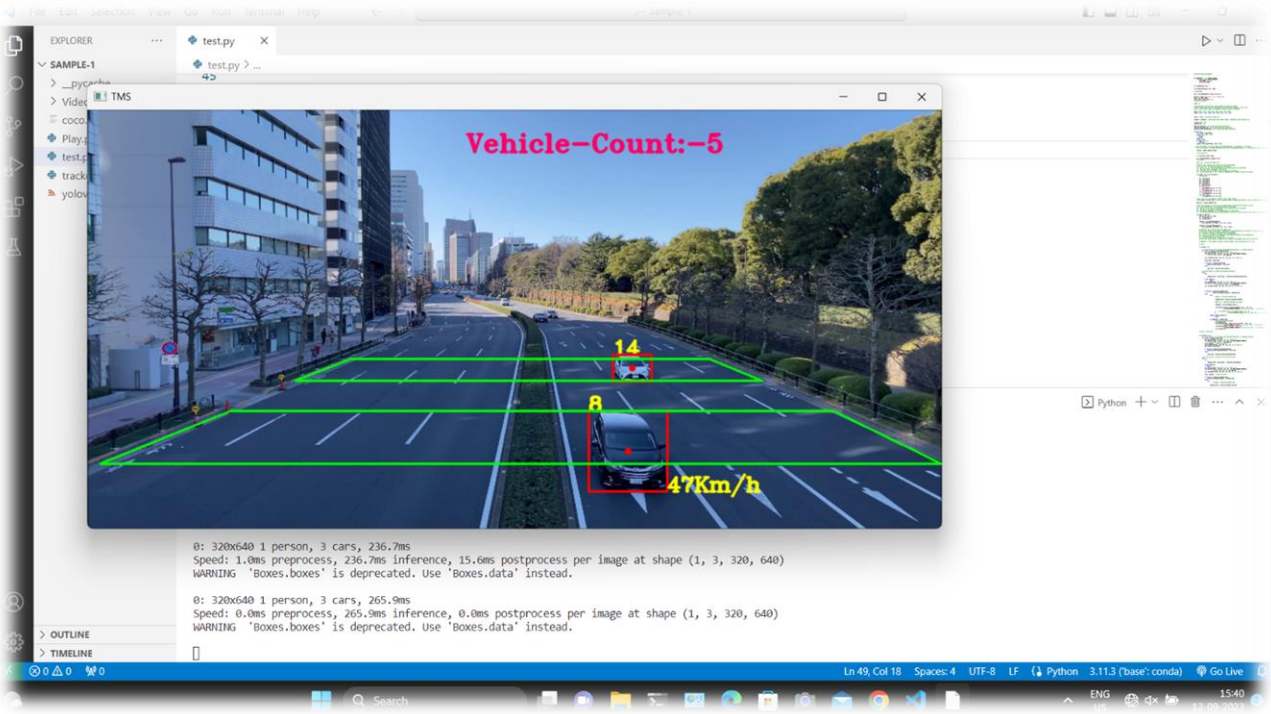
```
for bbox in bbox_id:
    x3, y3, x4, y4, id = bbox
    cx = int(x3+x4)//2
    cy = int(y3+y4)//2

    results = cv2.pointPolygonTest(
        np.array(area, np.int32), ((cx, cy)), False)

    results2 = cv2.pointPolygonTest(
        np.array(area2, np.int32), ((cx, cy)), False)


    # This line checks if the center coordinates of the bounding box are inside the specified area (ROI).
    # If they are, then the following steps are performed:
    # 1. A circle is drawn at the center of the bounding box.
    # 2. The ID of the object is displayed at the top-left corner of the bounding box.
    # 3. The bounding box is drawn in red.
    # 4. The ID of the object is added to the set area_c.
    # when, the vehicle passes through the Two regions, the elapsed time will be calculated.


    # Area-1

    if results >= 0:

        # if the vehicle is not coming from backward direction i.e. forward vehicles
        if id not in vehicles_entering_backward:
            cv2.circle(frame, (cx, cy), 4, (0, 0, 255), -1)
            cv2.putText(frame, str(id), (x3, y3), cv2.FONT_HERSHEY_COMPLEX,
                0.8, (0, 255, 255), 2, cv2.LINE_AA)

            cv2.rectangle(frame, (x3, y3), (x4, y4), (0, 0, 255), 2)

            Init_time = time.time()

            if id not in vehicles_entering:
                vehicles_entering[id] = Init_time
            else:

                Init_time = vehicles_entering[id]

        # if the vehicle is coming from forward direction
        else:
            try:

                elapsed_time = time.time() - vehicles_entering_backward[id]

            except KeyError:
                    pass
            cv2.circle(frame, (cx, cy), 4, (0, 0, 255), -1)
            cv2.putText(frame, str(id), (x3, y3), cv2.FONT_HERSHEY_COMPLEX,
                        0.8, (0, 255, 255), 2, cv2.LINE_AA)
            cv2.rectangle(frame, (x3, y3), (x4, y4), (0, 0, 255), 2)
```

```
            if id not in vehicles_elapsed_time:
                    vehicles_elapsed_time[id] = elapsed_time
            else:
                try:
                    # Speed -> distance/elapsed time

                    elapsed_time = vehicles_elapsed_time[id]

                    dist = 45  # Distance between two region

                    speed_KH = (dist/elapsed_time)*3.6

                    cv2.putText(frame, str(int(speed_KH))+'Km/h', (x4, y4),
                                cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2, cv2.LINE_AA)

                    #    cv2.putText(frame, str(elapsed_time), (x4, y4),
                    #                cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2, cv2.LINE_AA)

                except ZeroDivisionError:
                    pass

                if speed_KH >= speed_limit:
                    # Display a warning message
                    cv2.waitKey(500)
                    cv2.putText(frame, "Speed limit violated!", (440, 112),
                                cv2.FONT_HERSHEY_TRIPLEX, 1, (0, 255, 255), 2, cv2.LINE_AA)
                    cv2.putText(frame, 'Detected', (cx, cy),
                                cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2, cv2.LINE_AA)
                    cv2.waitKey(500)

    # Area-2 | Main Area


    if results2 >= 0:
        # if the vehicle is not coming from forward direction i.e. backward vehicles
        if id not in vehicles_entering:
            cv2.circle(frame, (cx, cy), 4, (0, 0, 255), -1)
            cv2.putText(frame, str(id), (x3, y3), cv2.FONT_HERSHEY_COMPLEX,
                0.8, (0, 255, 255), 2, cv2.LINE_AA)
            area_c.add(id)  # Vehicle-counter
            cv2.rectangle(frame, (x3, y3), (x4, y4), (0, 0, 255), 2)
            Init_time = time.time()

            if id not in vehicles_entering_backward:
                vehicles_entering_backward[id] = Init_time
            else:

                Init_time = vehicles_entering_backward[id]

        # if the vehicle is coming from forward direction
        else:
            try:

                elapsed_time = time.time() - vehicles_entering[id]
```

```
            else:
                try:
                    # Speed -> distance/elapsed time

                    elapsed_time = vehicles_elapsed_time[id]

                    dist = 45  # Distance between two region

                    speed_KH = (dist/elapsed_time)*3.6

                    cv2.putText(frame, str(int(speed_KH))+'Km/h', (x4, y4),
                                cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2, cv2.LINE_AA)

                    #    cv2.putText(frame, str(elapsed_time), (x4, y4),
                    #                cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2, cv2.LINE_AA)

                except ZeroDivisionError:
                    pass

                if speed_KH >= speed_limit:
                    # Display a warning message
                    cv2.waitKey(500)
                    cv2.putText(frame, "Speed limit violated!", (440, 112),
                                cv2.FONT_HERSHEY_TRIPLEX, 1, (0, 255, 255), 2, cv2.LINE_AA)
                    cv2.putText(frame, 'Detected', (cx, cy),
                                cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2, cv2.LINE_AA)
                    cv2.waitKey(500)
```



Vehicle Speed Detection

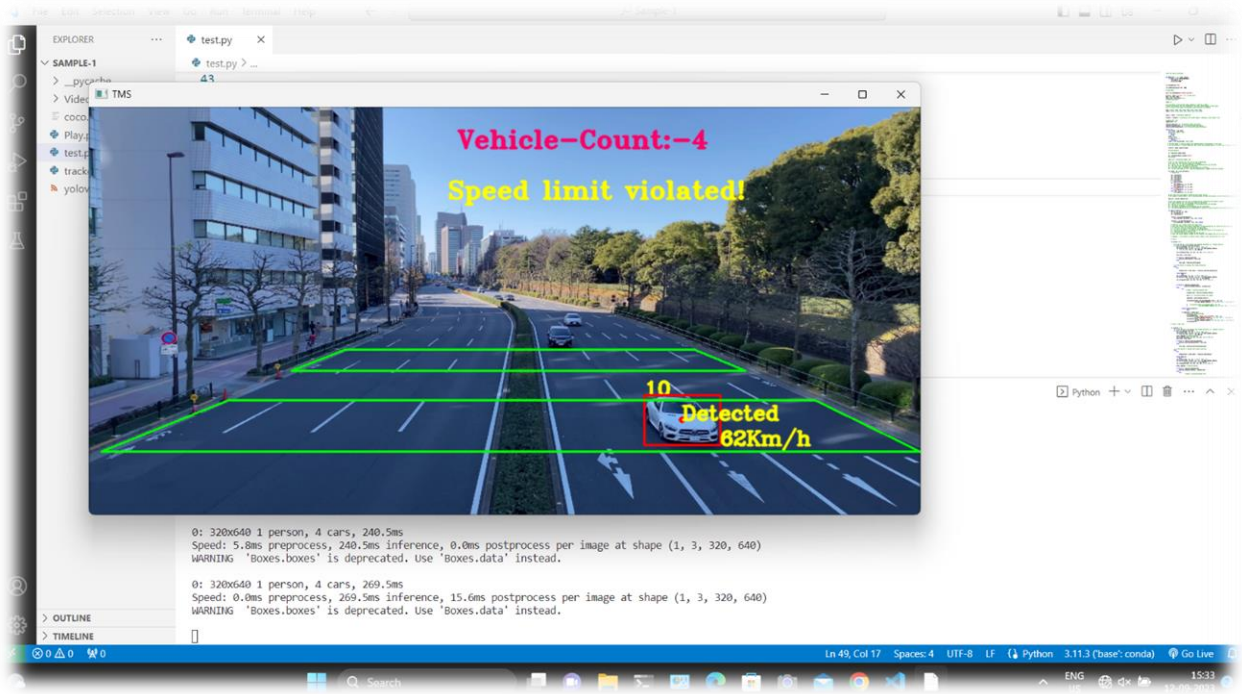


Vehicle Speed Detection

10. Detect if any of the vehicles are violating the speed limit: The algorithm can then be used to detect if any of the vehicles are violating the speed limit.

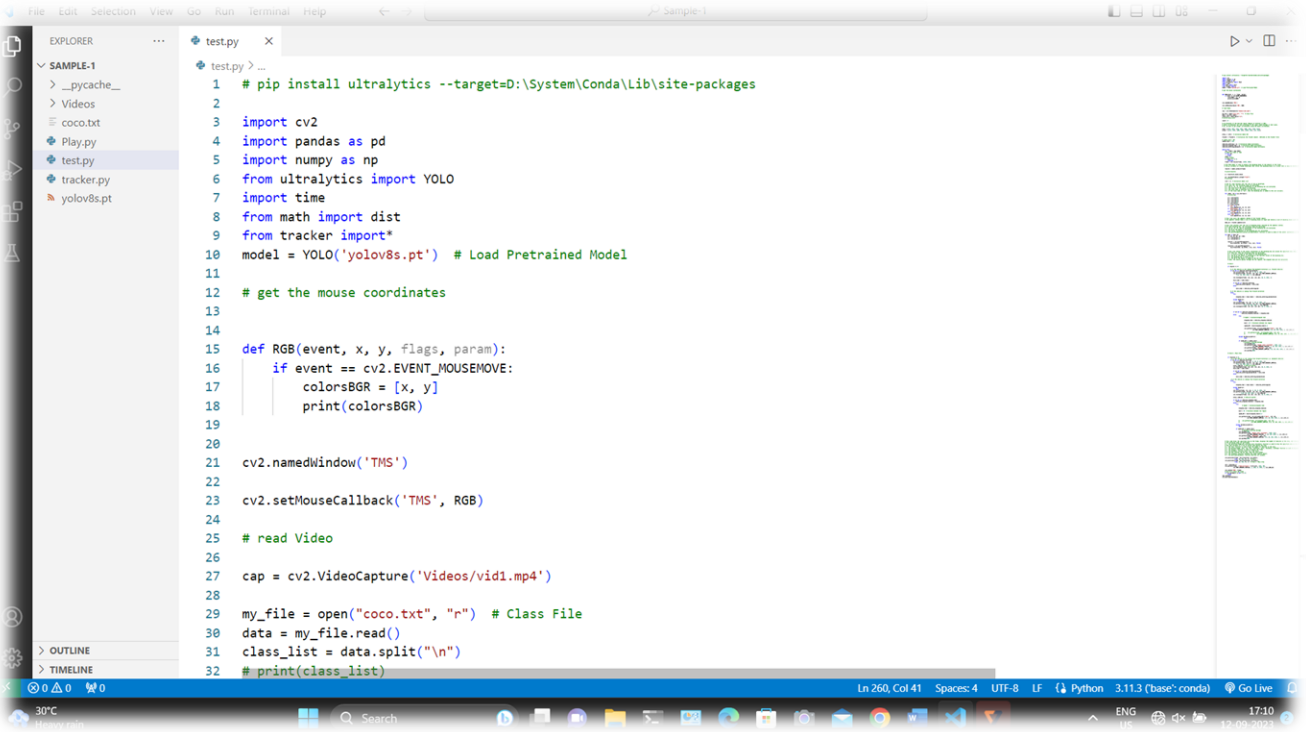
Show a warning message if a vehicle is violating the speed limit: A warning message can be displayed to the user if a vehicle is violating the speed limit

```
# speed_limit = 60
speed_limit = 120
 if speed_KH >= speed_limit:
                                # Display a warning message
                            cv2.waitKey(500)
                            cv2.putText(frame, "Speed limit violated!",
(440, 112),
                                        cv2.FONT_HERSHEY_TRIPLEX, 1, (0,
255, 255), 2, cv2.LINE_AA)
                            cv2.putText(frame, 'Detected', (cx, cy),
                                        cv2.FONT_HERSHEY_COMPLEX, 0.8, (0,
255, 255), 2, cv2.LINE_AA)
                            cv2.waitKey(500)
```



CHAPTER 5

Complete Source Code of The Project



#Play the Sample Video ----- play.py:

```
import cv2
import pandas as pd
import numpy as np

#get the mouse coordinates
def RGB(event, x, y, flags, param):
    if event == cv2.EVENT_MOUSEMOVE:
        colorsBGR = [x, y]
        print(colorsBGR)


cv2.namedWindow('TMS')
```

```
cv2.setMouseCallback('TMS', RGB)

#read Video

cap = cv2.VideoCapture('Videos/vid1.mp4')


while True:
    ret, frame = cap.read()
    #Check if frame is read
    if not ret:
        break
    frame = cv2.resize(frame, (1020, 500))
    cv2.imshow("TMS", frame)
    if cv2.waitKey(1) & 0xFF == 27:
        break
cap.release()
cv2.destroyAllWindows()
```

#test.py:

```
# pip install ultralytics --target=D:\System\Conda\Lib\site-packages

import cv2
import pandas as pd
import numpy as np
from ultralytics import YOLO
import time
from math import dist
from tracker import*
model = YOLO('yolov8s.pt')  # Load Pretrained Model

# get the mouse coordinates


def RGB(event, x, y, flags, param):
```

```
    if event == cv2.EVENT_MOUSEMOVE:
        colorsBGR = [x, y]
        print(colorsBGR)


cv2.namedWindow('TMS')

cv2.setMouseCallback('TMS', RGB)

# read Video

cap = cv2.VideoCapture('Videos/vid1.mp4')

my_file = open("coco.txt", "r")  # Class File
data = my_file.read()
class_list = data.split("\n")
# print(class_list)

count = 0

# Co-ordinates of the desired region (Region of Interest or ROI).
# above co-ordinates can vary according to the input video-footage or test cases.
# So, we have to put proper co-ordinates using the mouse co-ordinate.

area = [(314, 297), (742, 297), (805, 323), (248, 323)]
area2 = [(171, 359), (890, 359), (1019, 422), (15, 422)]


area_c = set()  # Initialize empty Set

tracker = Tracker()  # Initialize the Tracker object. (Defined in the tracker file)

# speed_limit = 60
speed_limit = 120

vehicles_entering = {}  # Initialize empty dictionary
vehicles_elapsed_time = {}  # Initialize empty dictionary
vehicles_entering_backward = {}  # Initialize empty dictionary

while True:
```

```
    ret, frame = cap.read()
    # Check if frame is read
    if not ret:
        break
    count += 1
    # Skip Frames
    if count % 3 != 0:
        continue
    frame = cv2.resize(frame, (1020, 500))

   # The YOLO model is used to predict the bounding boxes of the objects in the frame.
   # The px variable is a Pandas DataFrame that stores the bounding boxes in a format that is easy to iterate over.

    results = model.predict(frame)

    # print(results)

    a = results[0].boxes.boxes

    px = pd.DataFrame(a).astype("float")
    # print(px)

    list = []  # Initialize empty List

    # The for loop iterates over the rows of the px DataFrame.
    # For each row, the following steps are performed:
    # 1. The x1, y1, x2, and y2 coordinates of the bounding box are extracted.
    # 2. The class ID of the object is extracted.
    # 3. The class name is obtained from the class_list variable.
    # 4. If the class name is "car", then the bounding box is added to the list variable.

    for index, row in px.iterrows():
        # print(row)
        x1 = int(row[0])
        y1 = int(row[1])
        x2 = int(row[2])
        y2 = int(row[3])
        d = int(row[5])
```

```
        c = class_list[d]
        if 'car' in c:
            list.append([x1, y1, x2, y2])
        if 'motorcycle' in c:
            list.append([x1, y1, x2, y2])
        elif 'truck' in c:
            list.append([x1, y1, x2, y2])
        elif 'bus' in c:
            list.append([x1, y1, x2, y2])

    # This line calls the update() method of the Tracker object.
    # The update() method takes a list of bounding boxes as input and returns a list of bounding boxes with the IDs of the tracked objects.

    bbox_id = tracker.update(list)

    # This line iterates over the list of bounding boxes returned by the update() method.
    # For each bounding box, the following steps are performed:
    # 1. The x3, y3, x4, and y4 coordinates of the bounding box are extracted.
    # 2. The ID of the object is extracted.
    # 3. The center coordinates of the bounding box are calculated.
    # 4. The pointPolygonTest(contour,pt,measureDist) function is used to check if the center coordinates of the bounding box are inside the specified area.

    for bbox in bbox_id:
        x3, y3, x4, y4, id = bbox
        cx = int(x3+x4)//2
        cy = int(y3+y4)//2

        results = cv2.pointPolygonTest(
            np.array(area, np.int32), ((cx, cy)), False)

        results2 = cv2.pointPolygonTest(
            np.array(area2, np.int32), ((cx, cy)), False)


        # This line checks if the center coordinates of the bounding box are inside the specified area (ROI).
        # If they are, then the following steps are performed:
```

```
        # 1. A circle is drawn at the center of the bounding box.
        # 2. The ID of the object is displayed at the top-left corner of the bounding box.
        # 3. The bounding box is drawn in red.
        # 4. The ID of the object is added to the set area_c.
         # when, the vehicle passes through the Two regions, the elapsed time will be calculated.


        # Area-1

        if results >= 0:

            # if the vehicle is not coming from backward direction i.e. forward vehicles
            if id not in vehicles_entering_backward:
                cv2.circle(frame, (cx, cy), 4, (0, 0, 255), -1)
                cv2.putText(frame, str(id), (x3, y3), cv2.FONT_HERSHEY_COMPLEX,
                    0.8, (0, 255, 255), 2, cv2.LINE_AA)

                cv2.rectangle(frame, (x3, y3), (x4, y4), (0, 0, 255), 2)

                Init_time = time.time()

                if id not in vehicles_entering:
                    vehicles_entering[id] = Init_time
                else:

                    Init_time = vehicles_entering[id]

            # if the vehicle is coming from forward direction
            else:
                try:
                    elapsed_time = time.time() - vehicles_entering_backward[id]

                except KeyError:
                        pass
                cv2.circle(frame, (cx, cy), 4, (0, 0, 255), -1)
                cv2.putText(frame, str(id), (x3, y3), cv2.FONT_HERSHEY_COMPLEX,
                            0.8, (0, 255, 255), 2, cv2.LINE_AA)
                cv2.rectangle(frame, (x3, y3), (x4, y4), (0, 0, 255), 2)
```

```
            if id not in vehicles_elapsed_time:
                    vehicles_elapsed_time[id] = elapsed_time
            else:
                  try:
                        # Speed -> distance/elapsed time

                        elapsed_time = vehicles_elapsed_time[id]

                        dist = 45  # Distance between two region

                        speed_KH = (dist/elapsed_time)*3.6

                        cv2.putText(frame, str(int(speed_KH))+'Km/h', (x4, y4),
                                    cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2,
cv2.LINE_AA)

                        #    cv2.putText(frame, str(elapsed_time), (x4, y4),
                        #                cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255),
2, cv2.LINE_AA)

                  except ZeroDivisionError:
                        pass

                  if speed_KH >= speed_limit:
                        # Display a warning message
                        cv2.waitKey(500)
                        cv2.putText(frame, "Speed limit violated!", (440, 112),
                                    cv2.FONT_HERSHEY_TRIPLEX, 1, (0, 255, 255), 2,
cv2.LINE_AA)
                        cv2.putText(frame, 'Detected', (cx, cy),
                                    cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2,
cv2.LINE_AA)
                        cv2.waitKey(500)

      # Area-2 | Main Area


      if results2 >= 0:
          # if the vehicle is not coming from forward direction i.e. backward vehicles
          if id not in vehicles_entering:
```

```
        cv2.circle(frame, (cx, cy), 4, (0, 0, 255), -1)
        cv2.putText(frame, str(id), (x3, y3), cv2.FONT_HERSHEY_COMPLEX,
            0.8, (0, 255, 255), 2, cv2.LINE_AA)
        area_c.add(id)  # Vehicle-counter
        cv2.rectangle(frame, (x3, y3), (x4, y4), (0, 0, 255), 2)
        Init_time = time.time()

        if id not in vehicles_entering_backward:
            vehicles_entering_backward[id] = Init_time
        else:

            Init_time = vehicles_entering_backward[id]

    # if the vehicle is coming from forward direction
    else:
        try:

            elapsed_time = time.time() - vehicles_entering[id]

        except KeyError:
            pass
        cv2.circle(frame, (cx, cy), 4, (0, 0, 255), -1)
        cv2.putText(frame, str(id), (x3, y3), cv2.FONT_HERSHEY_COMPLEX,
                    0.8, (0, 255, 255), 2, cv2.LINE_AA)
        cv2.rectangle(frame, (x3, y3), (x4, y4), (0, 0, 255), 2)

        area_c.add(id)  # Vehicle-counter

        if id not in vehicles_elapsed_time:
            vehicles_elapsed_time[id] = elapsed_time
        else:
            try:
                    # Speed -> distance/elapsed time

                elapsed_time = vehicles_elapsed_time[id]

                dist = 45  # Distance between two region

                speed_KH = (dist/elapsed_time)*3.6
```

```
                cv2.putText(frame, str(int(speed_KH))+'Km/h', (x4, y4),
                            cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2,
cv2.LINE_AA)

                #    cv2.putText(frame, str(elapsed_time), (x4, y4),
                 #                cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2,
cv2.LINE_AA)

            except ZeroDivisionError:
                pass

            if speed_KH >= speed_limit:
                # Display a warning message
                cv2.waitKey(500)
                cv2.putText(frame, "Speed limit violated!", (440, 112),
                            cv2.FONT_HERSHEY_TRIPLEX, 1, (0, 255, 255), 2,
cv2.LINE_AA)
                cv2.putText(frame, 'Detected', (cx, cy),
                            cv2.FONT_HERSHEY_COMPLEX, 0.8, (0, 255, 255), 2,
cv2.LINE_AA)
                cv2.waitKey(500)

    # This code draws the specified area on the frame, displays the number of vehicles in
the area, and releases the video capture object and destroys all windows.
    # The following steps are performed:
     # 1. The polylines(image,pts,isClosed,color,thickness) function is used to draw the
specified area on the frame.
    # The area variable is a list of points that define the area.
    # 2. The len() function is used to get the number of vehicles in the area.
    # 3. The putText(image, text, org, font, fontScale, color, thickness, lineType) function
is used to display the number of vehicles in the area on the frame.
    # 4. The imshow() function is used to display the frame.
    # 5. The waitKey() function waits for a key press.
    # If the user presses the Esc key, then the loop breaks.
    # 6. The release() function releases the video capture object.
    # 7. The destroyAllWindows() function destroys all windows.

    cv2.polylines(frame, [np.array(area, np.int32)],
```

```
                True, (0, 255, 0), 2)  # Area-1
    cv2.polylines(frame, [np.array(area2, np.int32)],
                True, (0, 255, 0), 2)  # Area-2 | Main Area

    cnt = len(area_c)
    cv2.putText(frame, ('Vehicle-Count:-')+str(cnt), (452, 50),
                cv2.FONT_HERSHEY_TRIPLEX, 1, (102, 0, 255), 2, cv2.LINE_AA)

    cv2.imshow("TMS", frame)
    # put 0 to freeze the frame
    if cv2.waitKey(1) & 0xFF == 27:
        break
cap.release()
cv2.destroyAllWindows()
```

#tracker.py:

```
#////////////////////////////Tracker
Module//////////////////////////////////////////////////////////////////////////////\

#Centroid Tracking algorithm
#Eucledian Distance Tracker

import math


class Tracker:
    def __init__(self):
        # Store the center positions of the objects
        #The self.center_points variable is a dictionary that stores the center positions of
the objects that have been detected.
        #The key of the dictionary is the ID of the object, and the value is the tuple of the
x and y coordinates of the object's center point.
        self.center_points = {}
        # Keep the count of the IDs
```

```
        #The self.id_count variable is a counter that keeps track of the ID of the next object that is detected.
        # each time a new object id detected, the count will increase by one
        self.id_count = 0

    def update(self, objects_rect):
        # Objects boxes and ids
        #The objects_bbs_ids variable is a list that will store the bounding boxes and IDs of the objects that have been detected in the current frame.
        objects_bbs_ids = []

        # Get center point of new object
        #The for loop iterates over the list of objects, objects_rect.
        #For each object, the code calculates the center point of the object by taking the average of the x and y coordinates of the top-left and bottom-right corners of the object's bounding box.
        #The center point is then assigned to the variable cx and cy.

        #The objects_bbs_ids variable is initialized to an empty list.
        #• The for loop iterates over the list of objects, objects_rect.
        #• The rect variable is a tuple that contains the x, y, w, and h coordinates of the current object's bounding box.
        #• The x, y, w, and h variables are assigned the values from the rect tuple.
        #• The cx variable is assigned the average of the x and w coordinates.
        #• The cy variable is assigned the average of the y and h coordinates.
        #• The (cx, cy) tuple is appended to the objects_bbs_ids list.

        for rect in objects_rect:
            x, y, w, h = rect
            cx = (x + x + w) // 2
            cy = (y + y + h) // 2

            # Find out if that object was detected already
            #The same_object_detected variable is a boolean variable that is used to keep track of whether or not the current object has been detected before.
            #The for loop iterates over the dictionary self.center_points.
            #For each object in the dictionary, the code calculates the distance between the center point of the current object and the center point of the object in the dictionary.
```

#If the distance is less than a threshold value (50), then the code concludes that the same object has been detected and the same_object_detected variable is set to True.
#The object's bounding box and ID are then appended to the objects_bbs_ids list.
#The Threshold value (50) can vary according to the input video-footage or test cases.

```
            same_object_detected = False
            for id, pt in self.center_points.items():
                dist = math.hypot(cx - pt[0], cy - pt[1])

                if dist < 50:
                    self.center_points[id] = (cx, cy)
#                   print(self.center_points)
                    objects_bbs_ids.append([x, y, w, h, id])
                    same_object_detected = True
                    break
            # New object is detected we assign the ID to that object
```

#The same_object_detected variable is a boolean variable that is used to keep track of whether or not the current object has been detected before.
#The if statement checks if the same_object_detected variable is False.
#If it is, then the code assigns a new ID to the object and appends the object's bounding box and ID to the objects_bbs_ids list.

```
            if same_object_detected is False:
                self.center_points[self.id_count] = (cx, cy)
                objects_bbs_ids.append([x, y, w, h, self.id_count])
                self.id_count += 1

        # Clean the dictionary by center points to remove IDS not used anymore
```

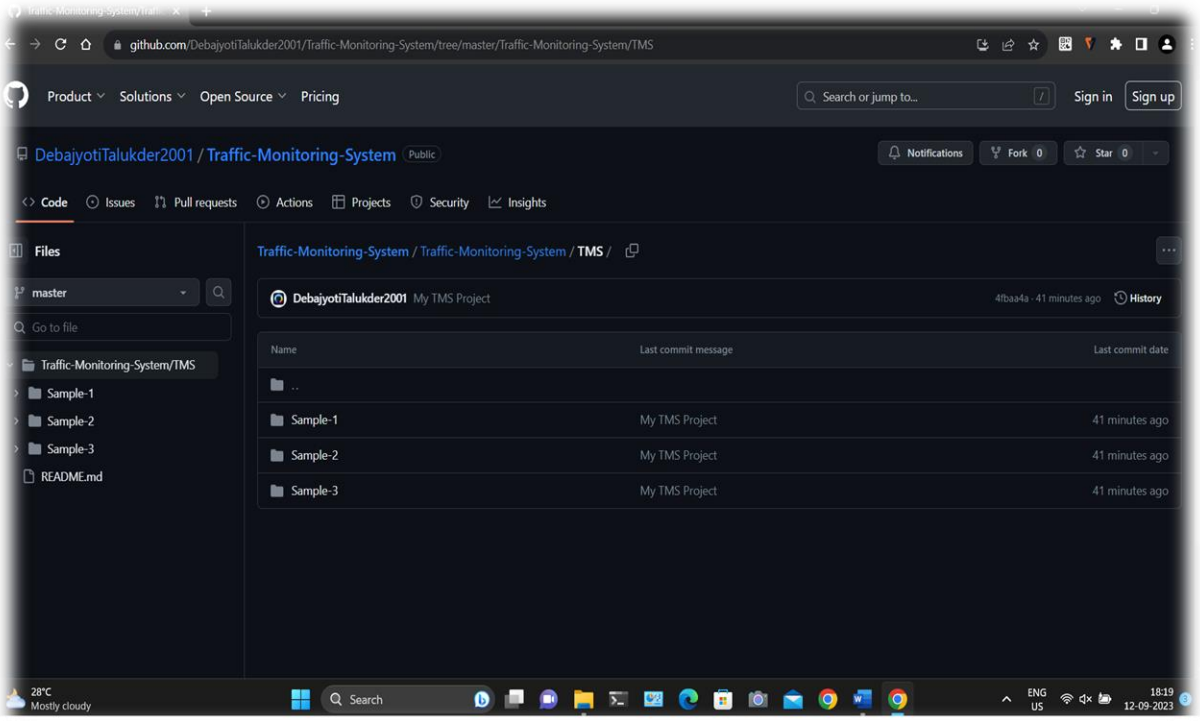
#The new_center_points dictionary is a new dictionary that will be used to store the center points of the objects that have been detected in the current frame.
#The for loop iterates over the list of objects, objects_bbs_ids.
#For each object, the code retrieves the object's ID and center point from the self.center_points dictionary.
#The center point is then added to the new_center_points dictionary.

```
        new_center_points = {}
        for obj_bb_id in objects_bbs_ids:
```

```
            _, _, _, _, object_id = obj_bb_id
            center = self.center_points[object_id]
            new_center_points[object_id] = center

        # Update dictionary with IDs not used removed
            #The self.center_points dictionary is updated with the new_center_points dictionary.
            #The new_center_points dictionary contains the center points of the objects that have been detected in the current frame.
            #The return statement returns the list of objects, objects_bbs_ids.

            self.center_points = new_center_points.copy()
            return objects_bbs_ids

# ////////////////////////////Tracker////////////////////////////////#
```

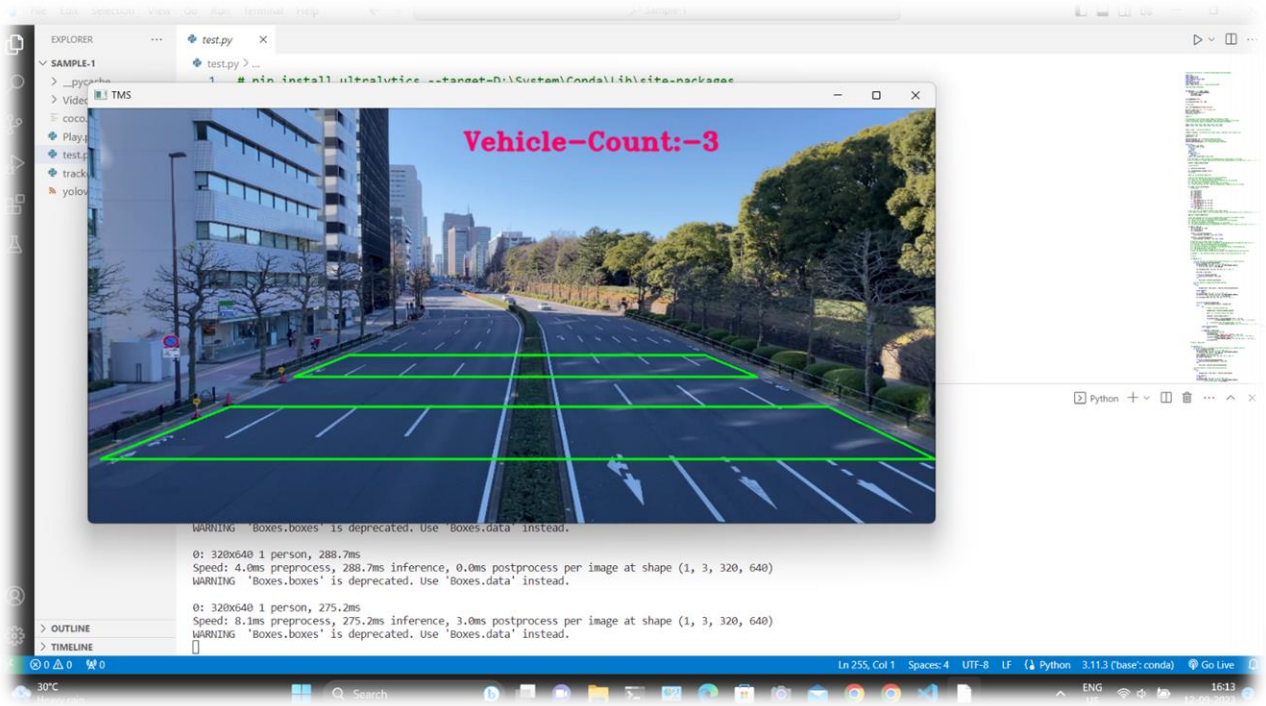
GitHub Repository of the Project:

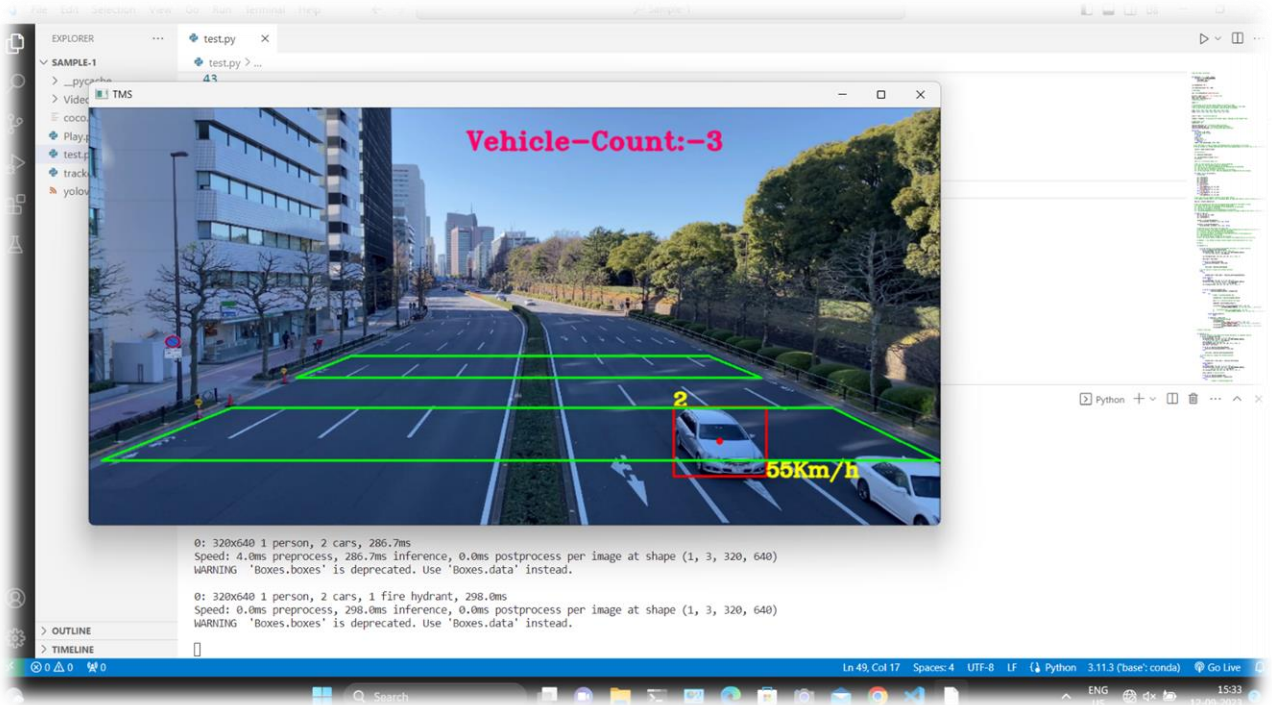
https://github.com/DebajyotiTalukder2001/Traffic-Monitoring-System

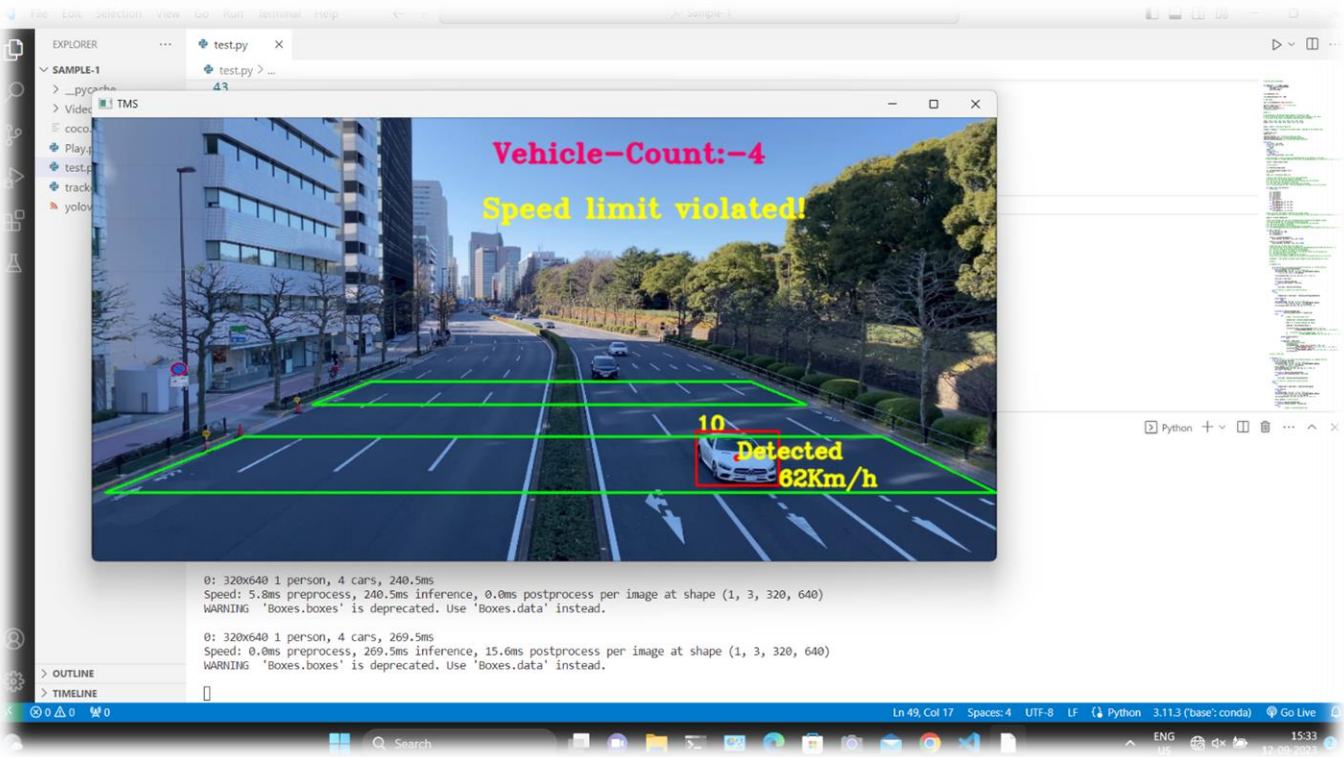


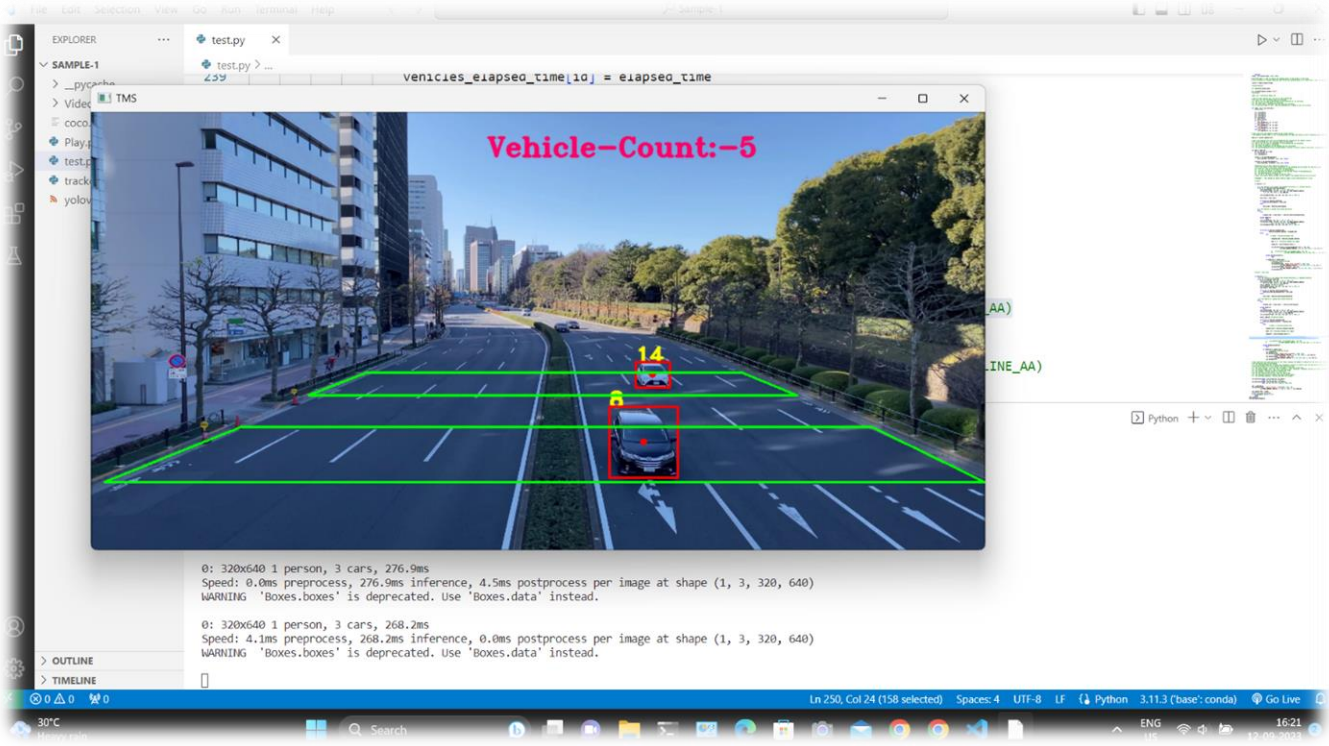
CHAPTER 6

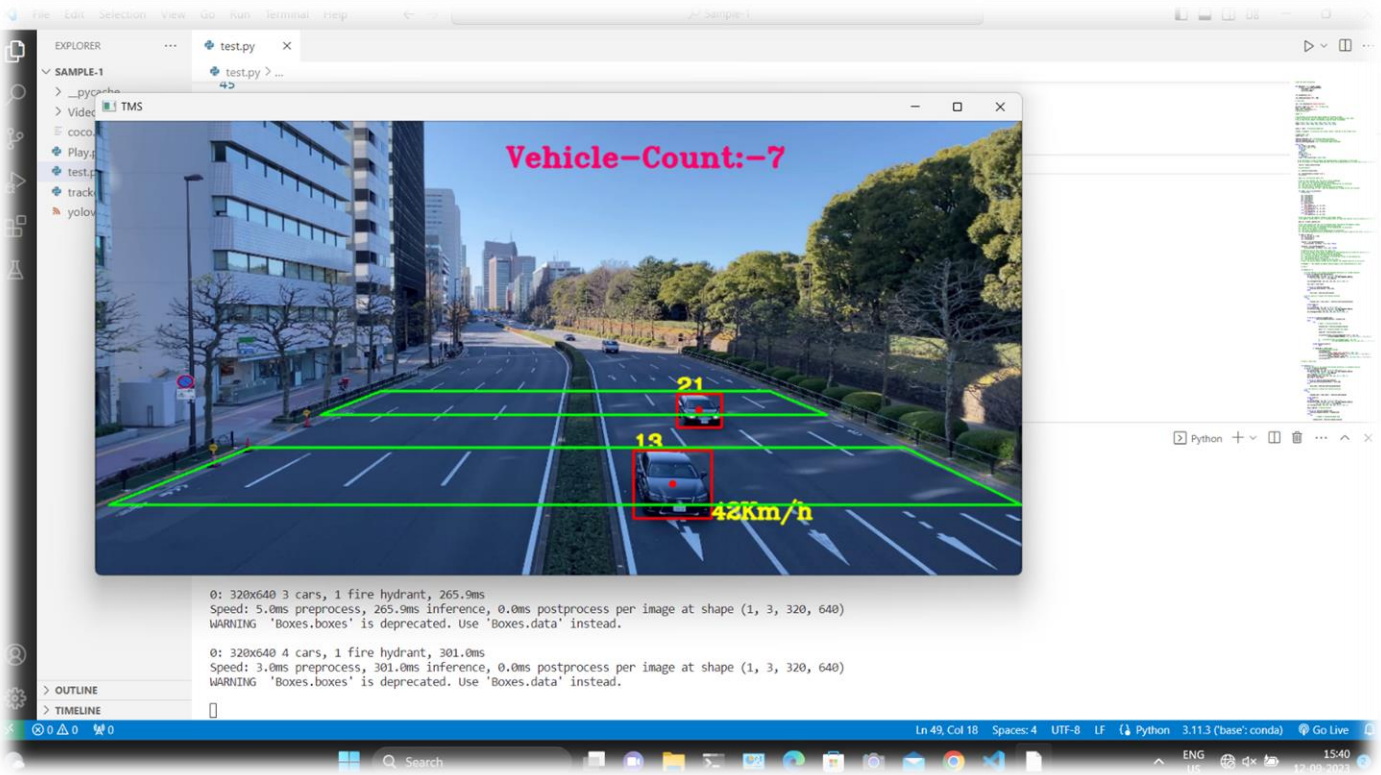
Sample Screenshots of The Project (Output)

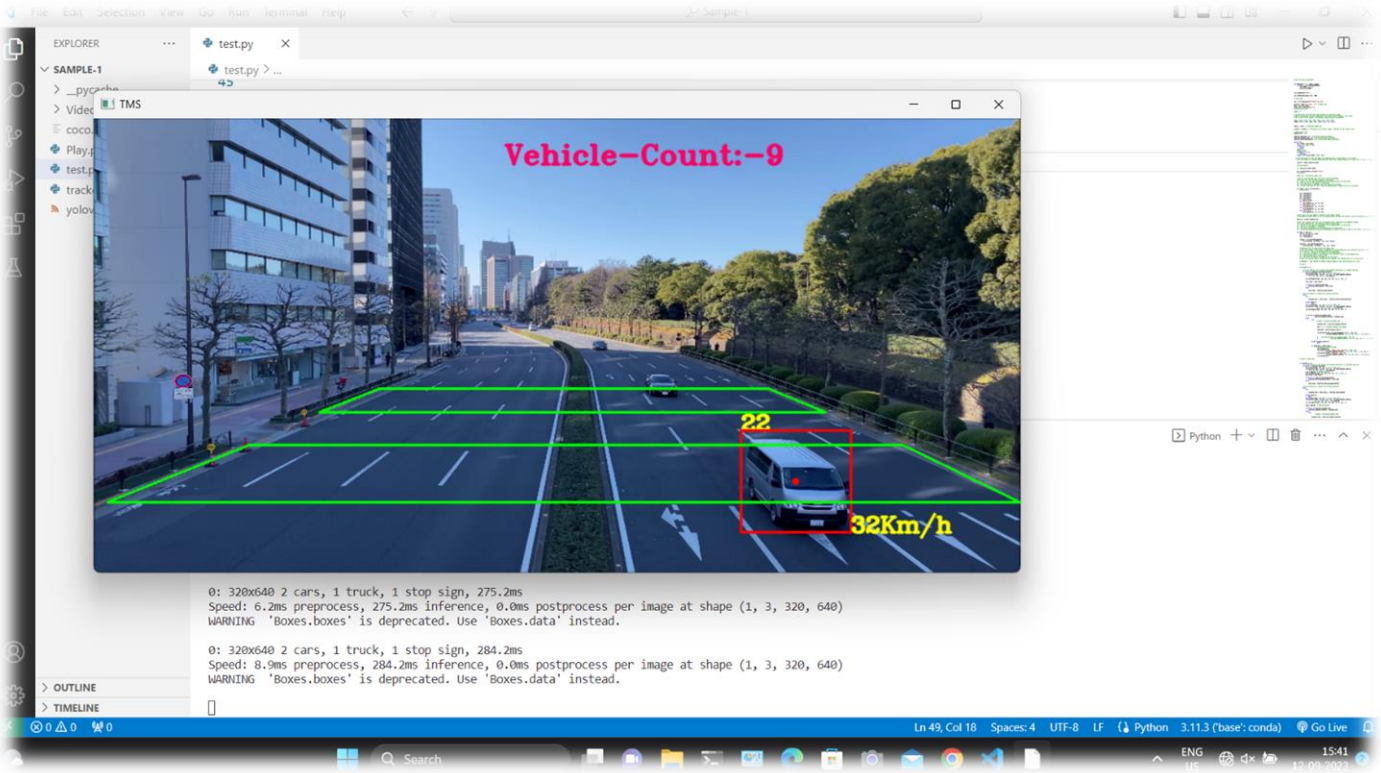


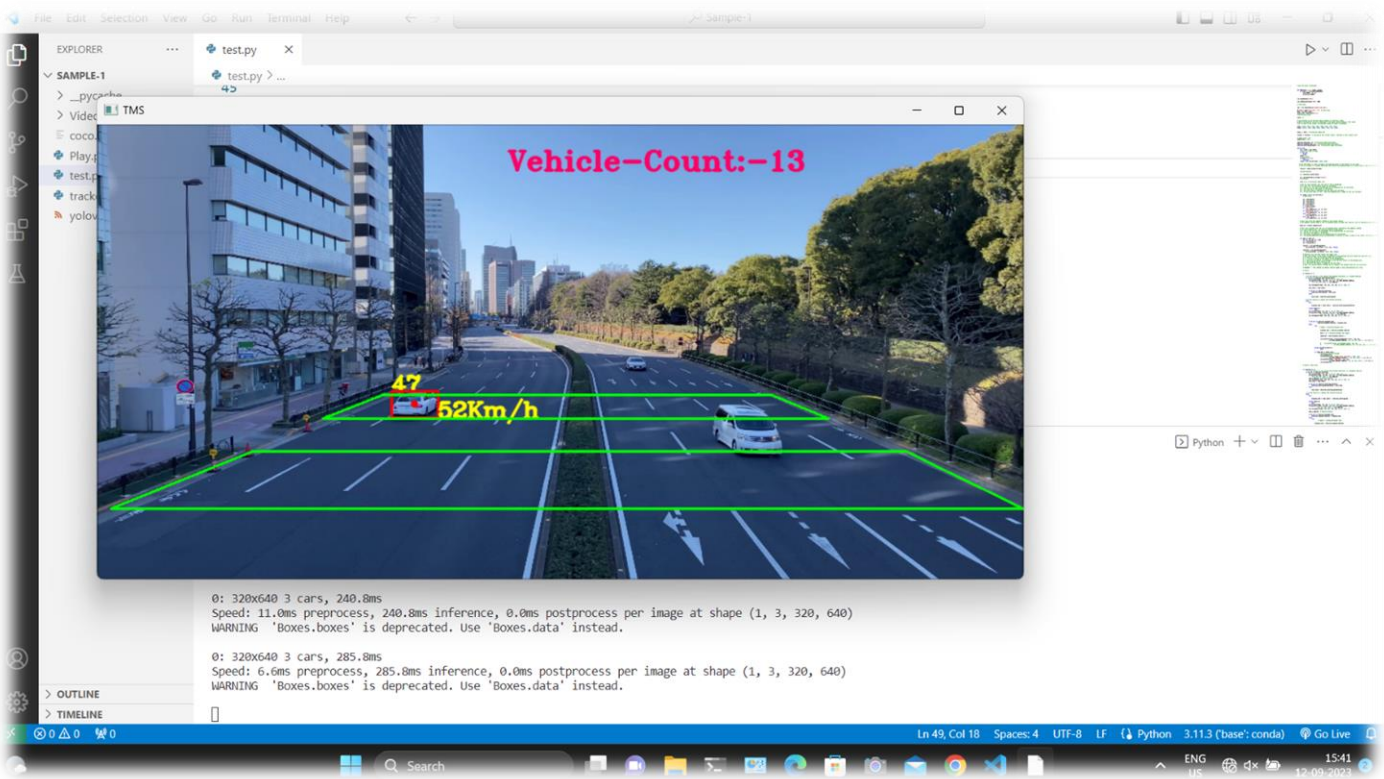



TMS
Vehicle-Count:-4
Speed limit violated!
10
Detected
62Km/h
0: 320x640 1 person, 4 cars, 240.5ms
Speed: 5.8ms preprocess, 240.5ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 1 person, 4 cars, 269.5ms
Speed: 0.0ms preprocess, 269.5ms inference, 15.6ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

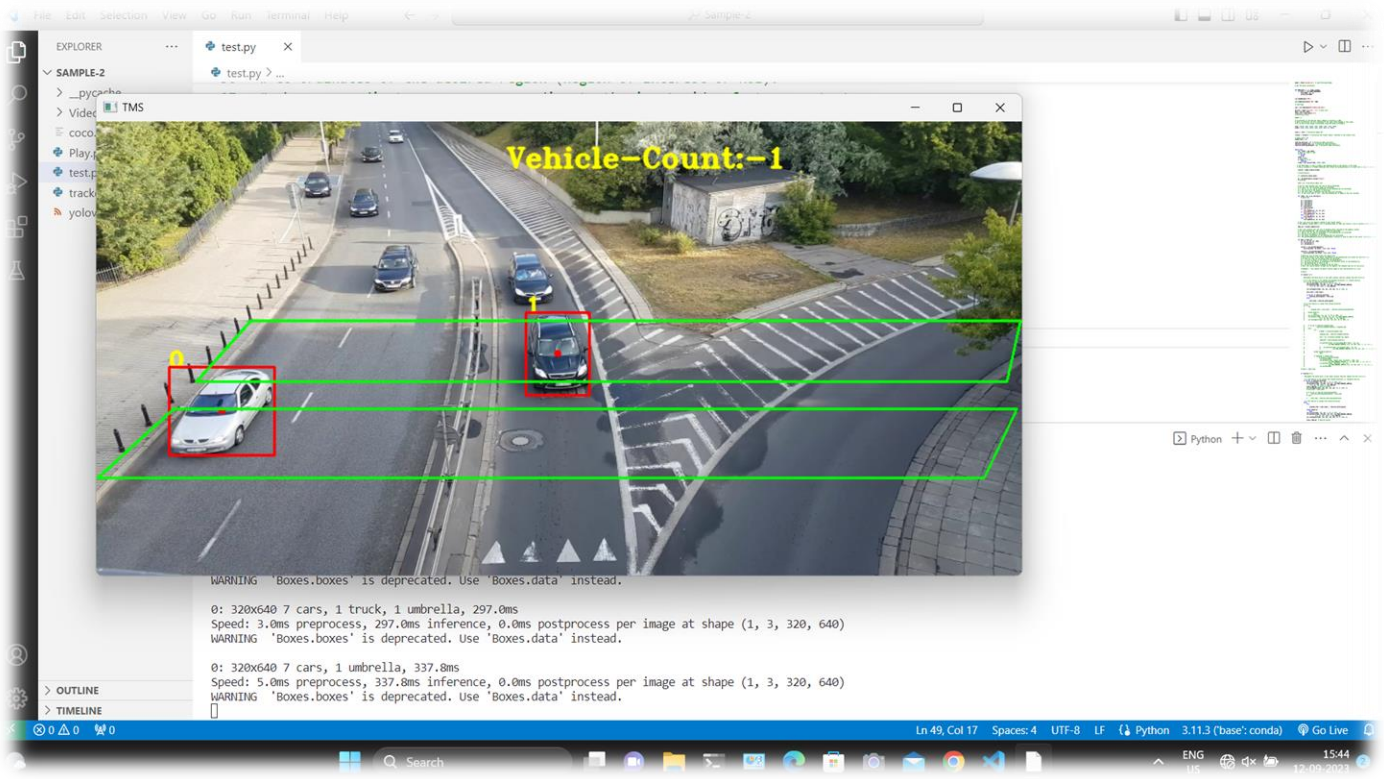

TMS
Vehicle-Count:-5
14
8
0: 320x640 1 person, 3 cars, 276.9ms
Speed: 0.0ms preprocess, 276.9ms inference, 4.5ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 1 person, 3 cars, 268.2ms
Speed: 4.1ms preprocess, 268.2ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

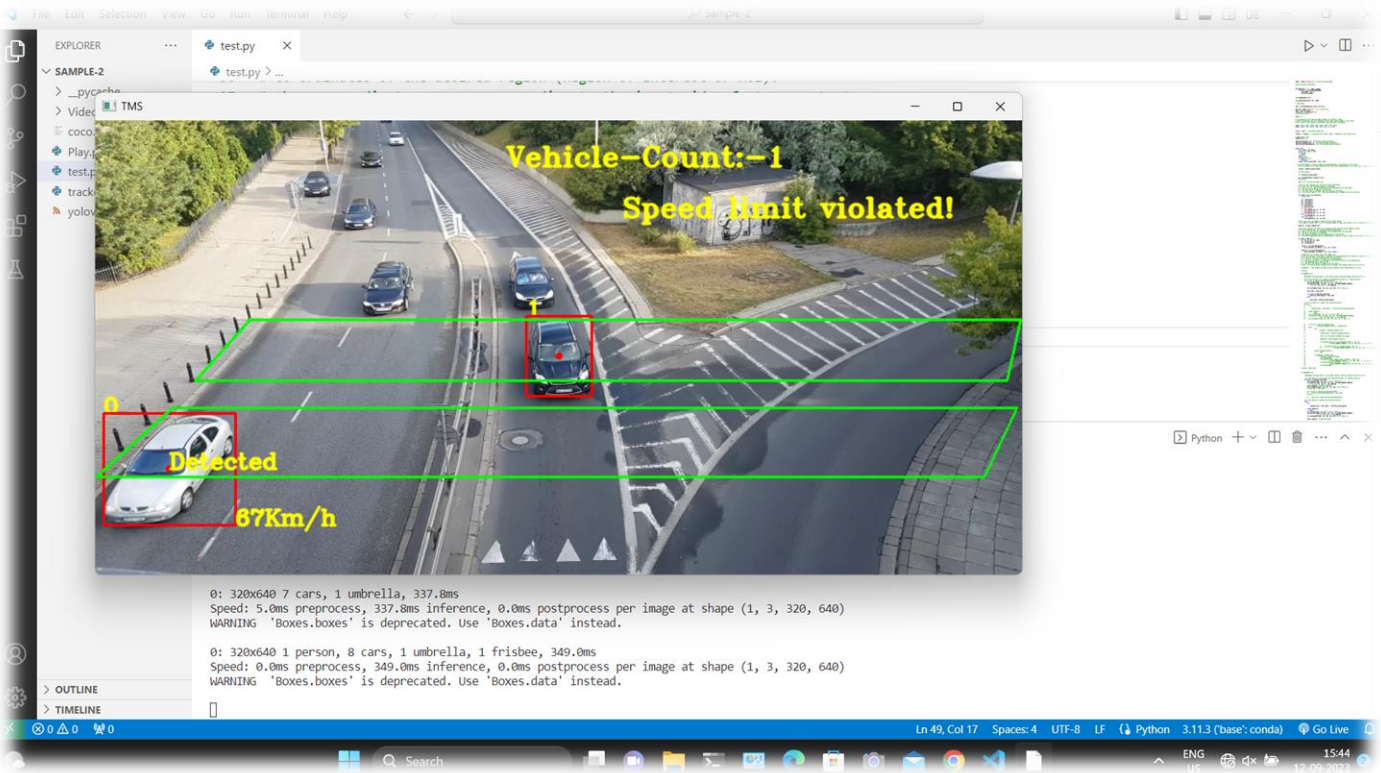

TMS
Vehicle-Count:-7
21
13
42Km/h
0: 320x640 3 cars, 1 fire hydrant, 265.9ms
Speed: 5.0ms preprocess, 265.9ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 4 cars, 1 fire hydrant, 301.0ms
Speed: 3.0ms preprocess, 301.0ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

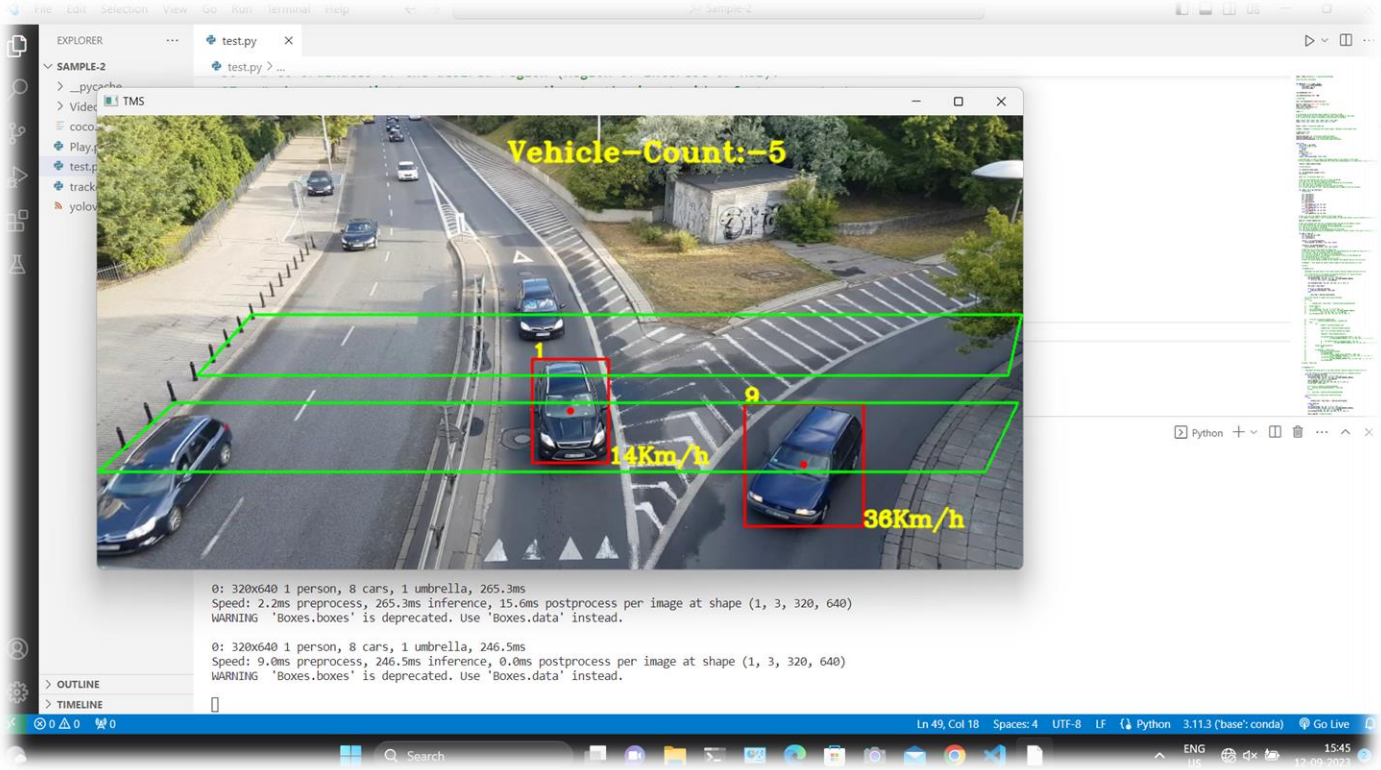

TMS
Vehicle-Count:-9
22
32Km/h
0: 320x640 2 cars, 1 truck, 1 stop sign, 275.2ms
Speed: 6.2ms preprocess, 275.2ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 2 cars, 1 truck, 1 stop sign, 284.2ms
Speed: 8.9ms preprocess, 284.2ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

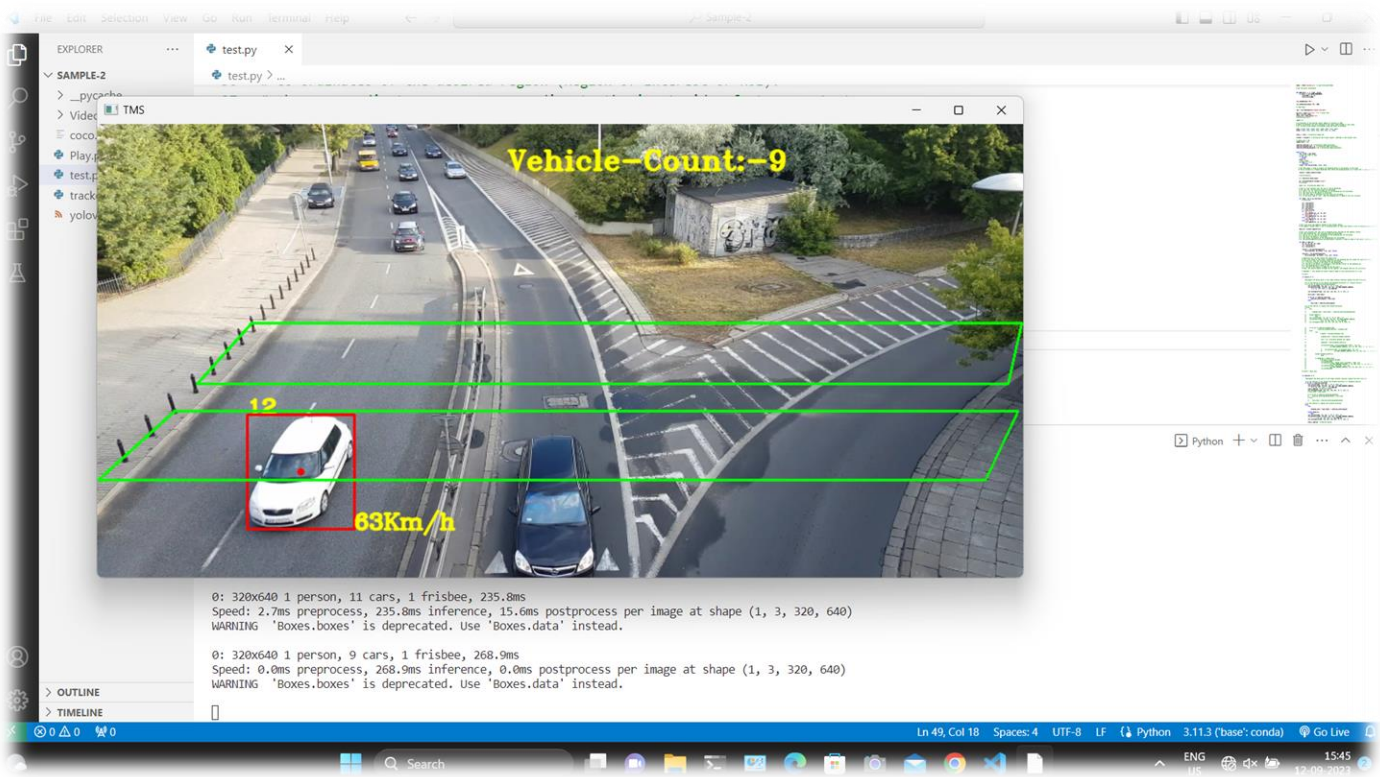

TMS
Vehicle-Count:-13
47
52Km/h
0: 320x640 3 cars, 240.8ms
Speed: 11.0ms preprocess, 240.8ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 3 cars, 285.8ms
Speed: 6.6ms preprocess, 285.8ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

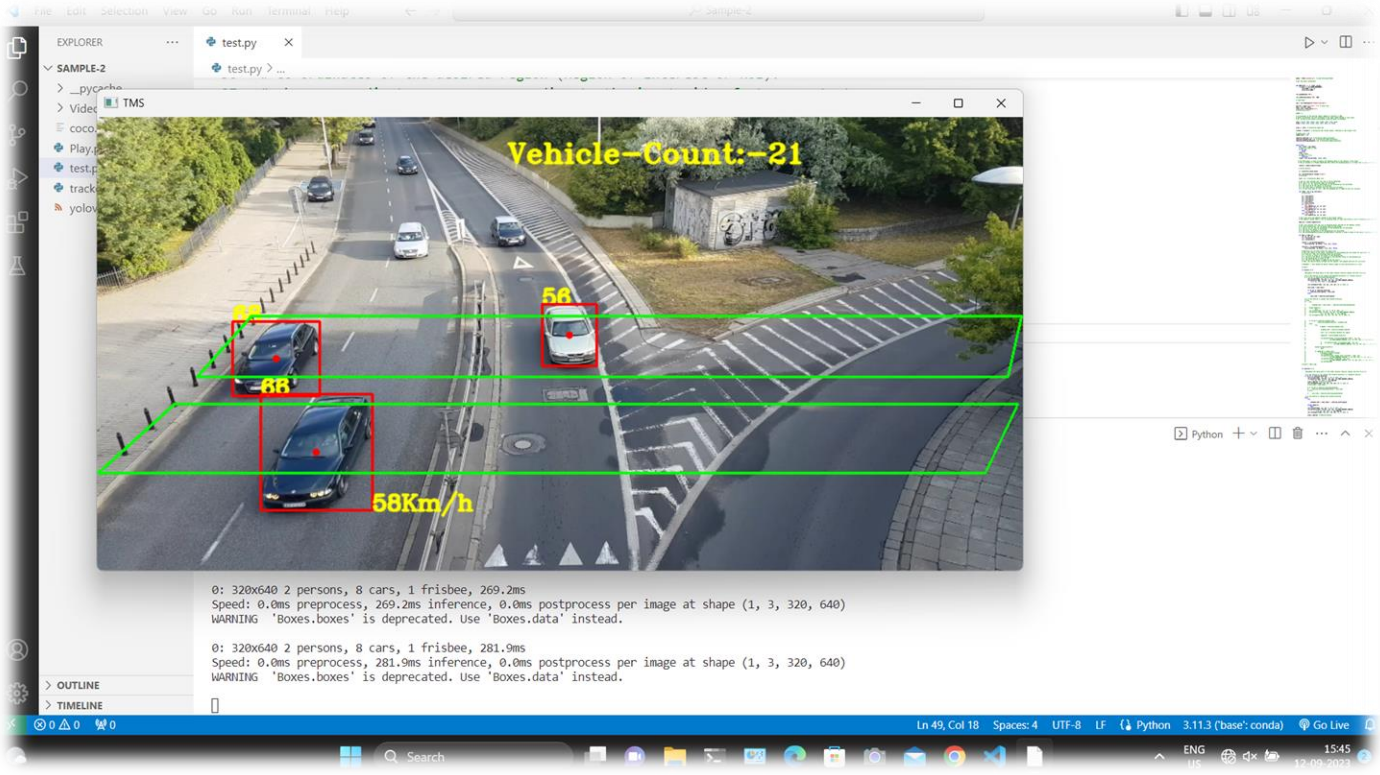
Sample-2:

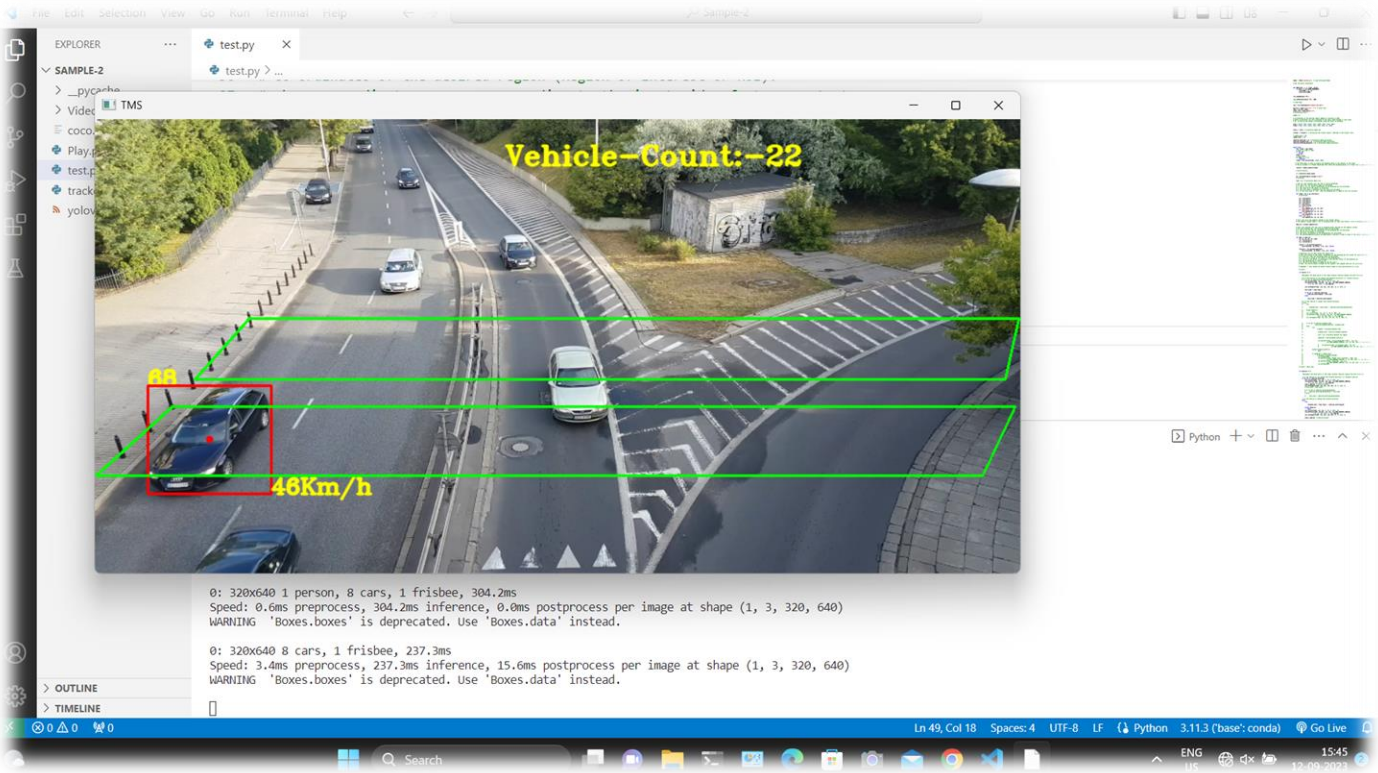

TMS
Vehicle-Count:-1
0
1
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 7 cars, 1 truck, 1 umbrella, 297.0ms
Speed: 3.0ms preprocess, 297.0ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 7 cars, 1 umbrella, 337.8ms
Speed: 5.0ms preprocess, 337.8ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

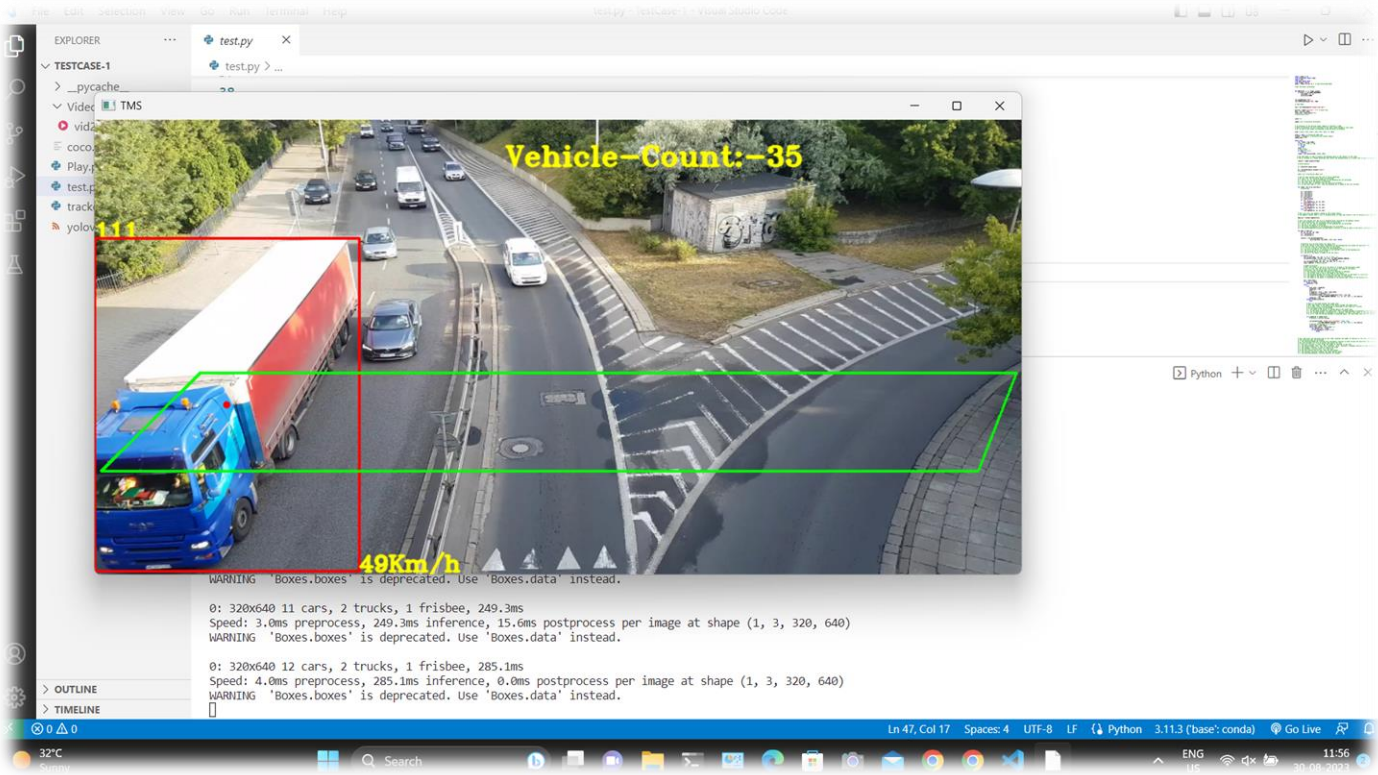

TMS
Vehicle-Count:-1
Speed limit violated!
1
0
Detected
67Km/h
0: 320x640 7 cars, 1 umbrella, 337.8ms
Speed: 5.0ms preprocess, 337.8ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 1 person, 8 cars, 1 umbrella, 1 frisbee, 349.0ms
Speed: 0.0ms preprocess, 349.0ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.


TMS
Vehicle-Count:-5
1
14Km/h
9
36Km/h
0: 320x640 1 person, 8 cars, 1 umbrella, 265.3ms
Speed: 2.2ms preprocess, 265.3ms inference, 15.6ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 1 person, 8 cars, 1 umbrella, 246.5ms
Speed: 9.0ms preprocess, 246.5ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

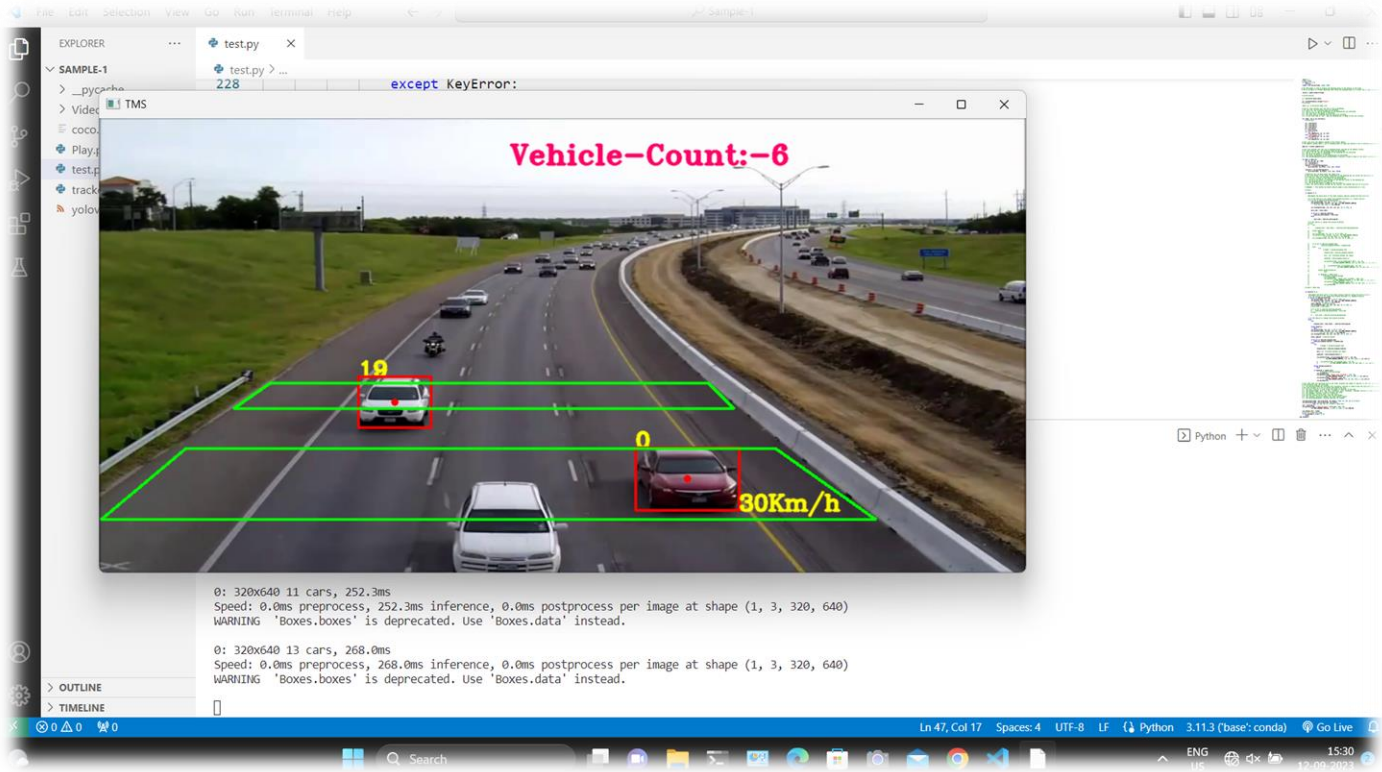

TMS
Vehicle-Count:-9
12
63Km/h
0: 320x640 1 person, 11 cars, 1 frisbee, 235.8ms
Speed: 2.7ms preprocess, 235.8ms inference, 15.6ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 1 person, 9 cars, 1 frisbee, 268.9ms
Speed: 0.0ms preprocess, 268.9ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

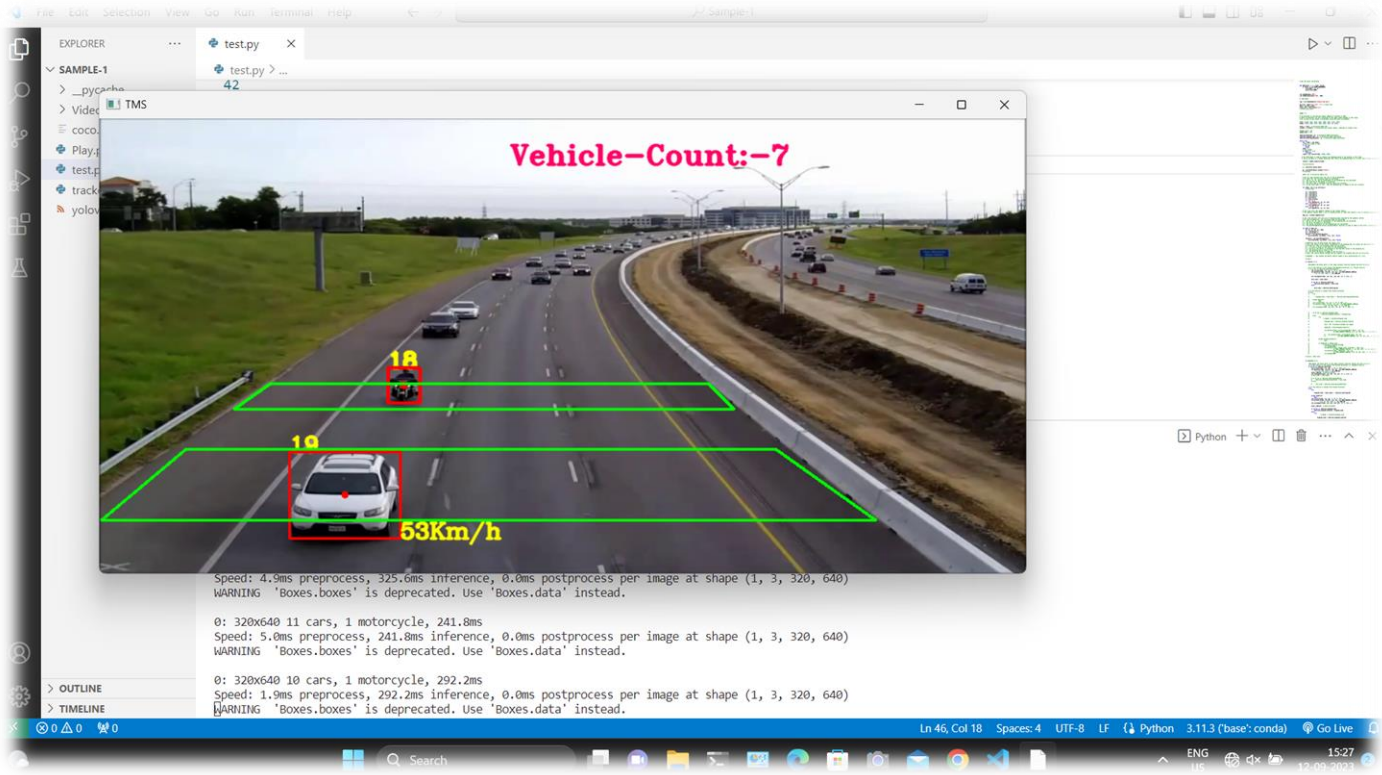

TMS
Vehicle-Count:-21
58
58Km/h
0: 320x640 2 persons, 8 cars, 1 frisbee, 269.2ms
Speed: 0.0ms preprocess, 269.2ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 2 persons, 8 cars, 1 frisbee, 281.9ms
Speed: 0.0ms preprocess, 281.9ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

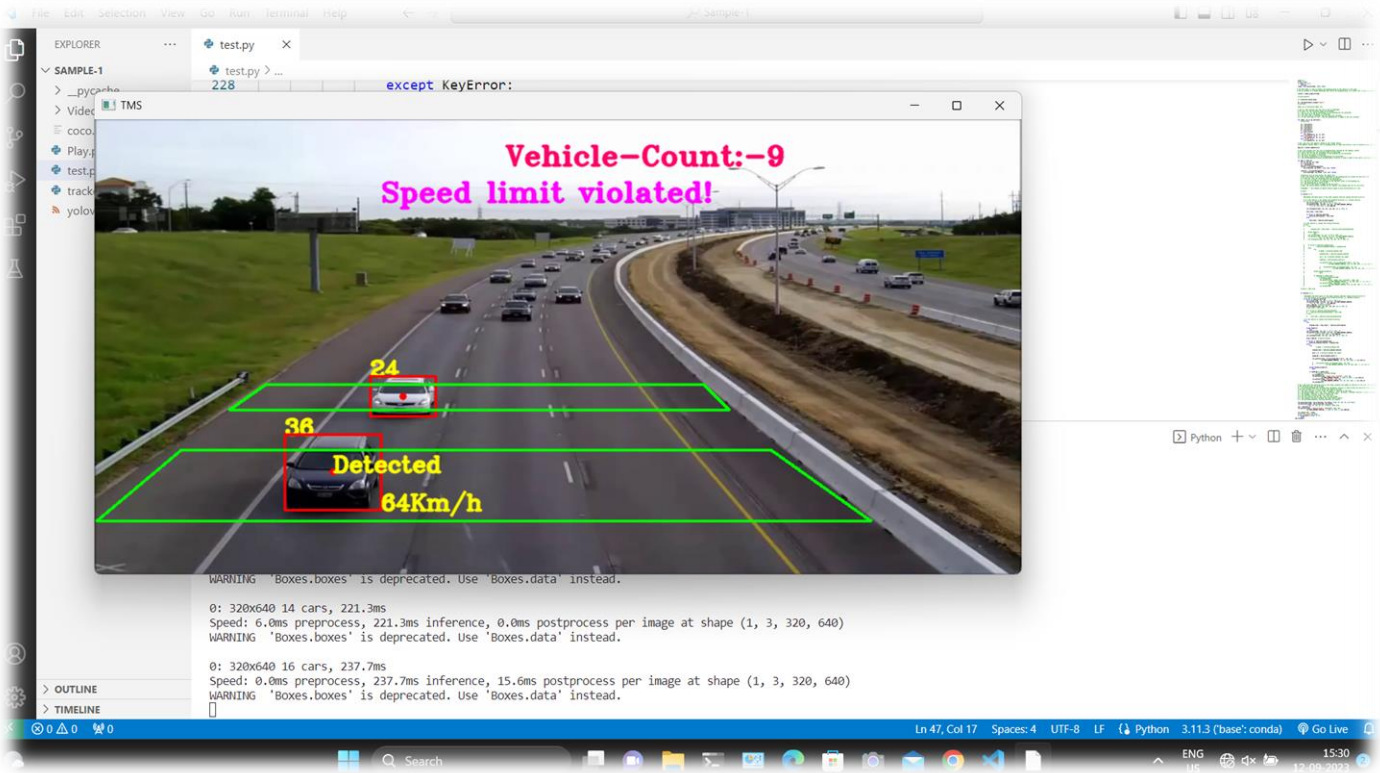

TMS
Vehicle-Count:-22
88
46Km/h
0: 320x640 1 person, 8 cars, 1 frisbee, 304.2ms
Speed: 0.6ms preprocess, 304.2ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 8 cars, 1 frisbee, 237.3ms
Speed: 3.4ms preprocess, 237.3ms inference, 15.6ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

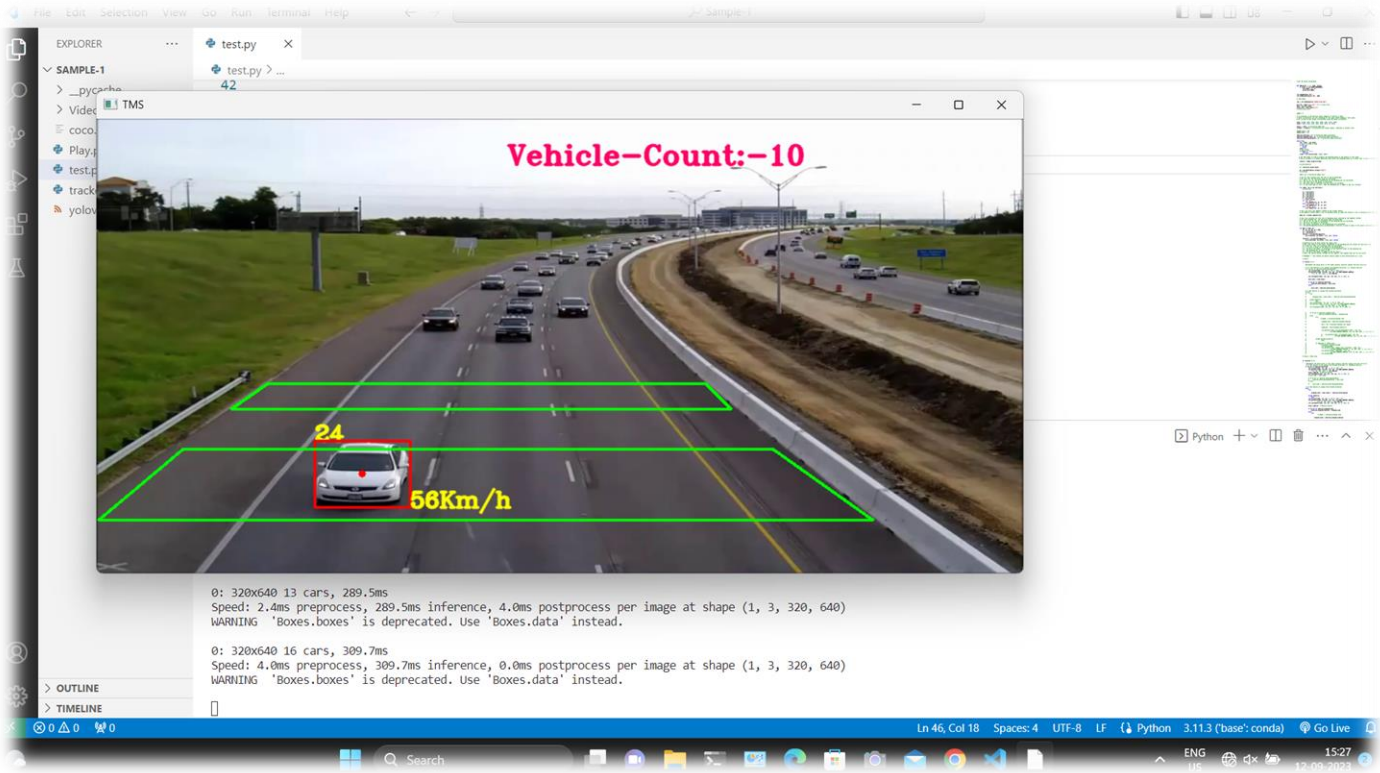

TMS
Vehicle-Count:-35
111
49Km/h
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 11 cars, 2 trucks, 1 frisbee, 249.3ms
Speed: 3.0ms preprocess, 249.3ms inference, 15.6ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 12 cars, 2 trucks, 1 frisbee, 285.1ms
Speed: 4.0ms preprocess, 285.1ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

Sample-3:


TMS
Vehicle-Count:-6
19
0
30Km/h
0: 320x640 11 cars, 252.3ms
Speed: 0.0ms preprocess, 252.3ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 13 cars, 268.0ms
Speed: 0.0ms preprocess, 268.0ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.


TMS
Vehicle-Count:-7
18
19
53Km/h
Speed: 4.9ms preprocess, 325.6ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 11 cars, 1 motorcycle, 241.8ms
Speed: 5.0ms preprocess, 241.8ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 10 cars, 1 motorcycle, 292.2ms
Speed: 1.9ms preprocess, 292.2ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.


TMS
Vehicle-Count:-9
Speed limit violated!
24
36
Detected
64Km/h
except KeyError:
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 14 cars, 221.3ms
Speed: 6.0ms preprocess, 221.3ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 16 cars, 237.7ms
Speed: 0.0ms preprocess, 237.7ms inference, 15.6ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.


TMS
Vehicle-Count:-10
24
56Km/h
0: 320x640 13 cars, 289.5ms
Speed: 2.4ms preprocess, 289.5ms inference, 4.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 16 cars, 309.7ms
Speed: 4.0ms preprocess, 309.7ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

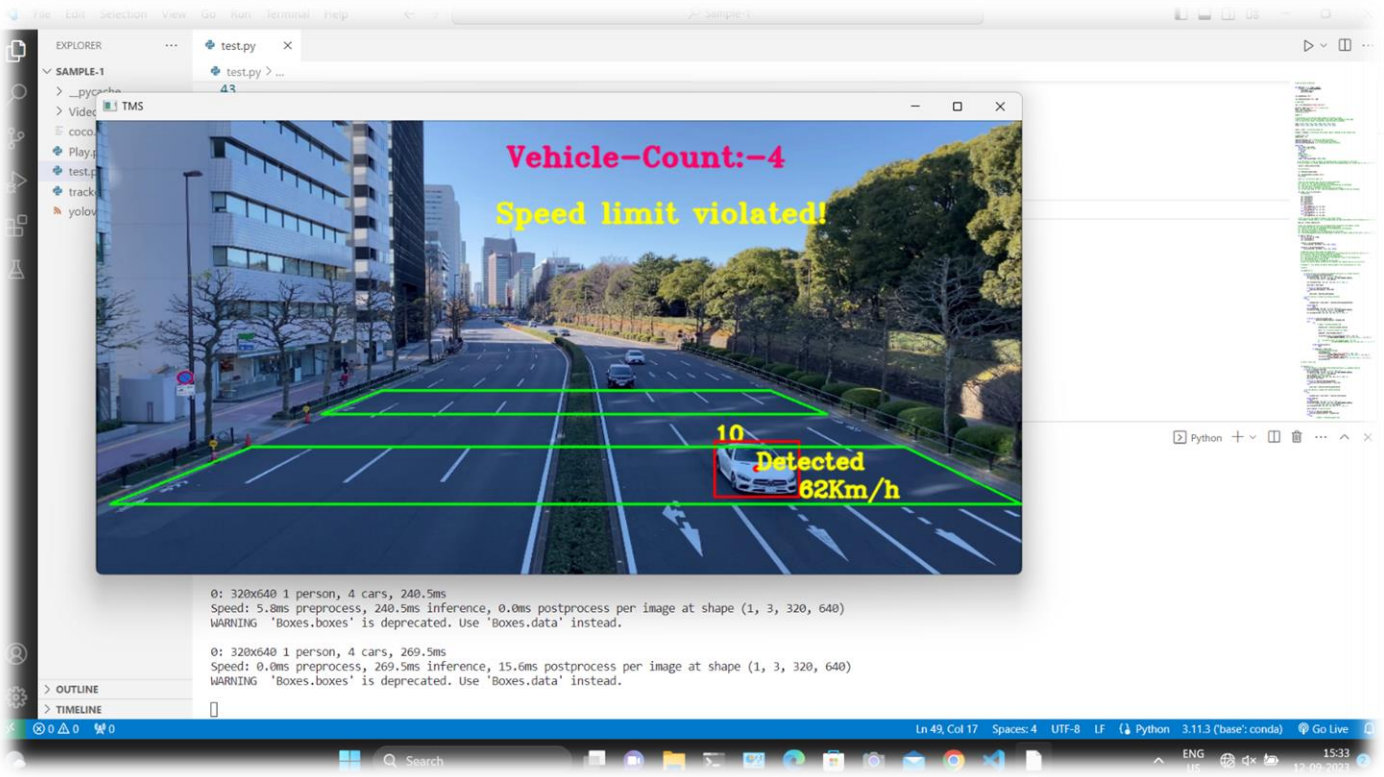
CHAPTER 7

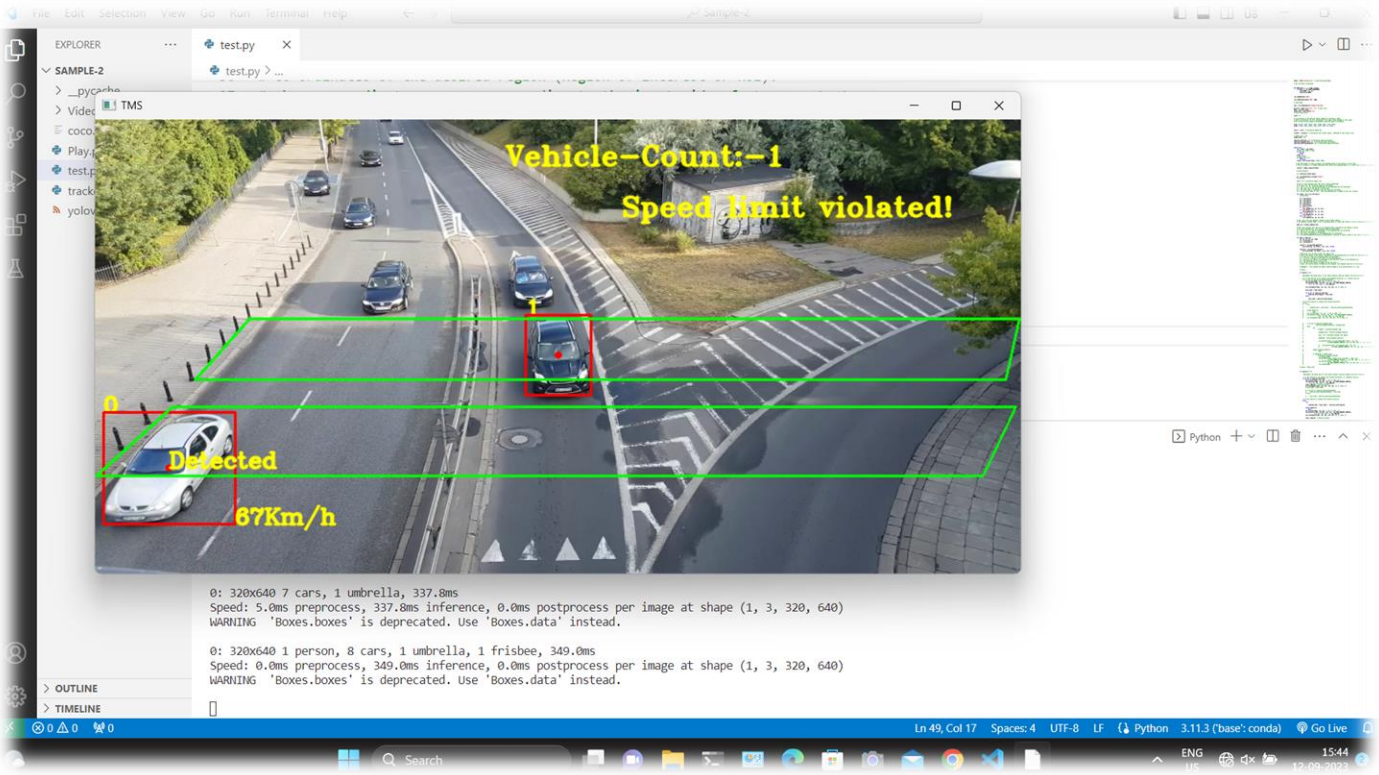
Experimental Results

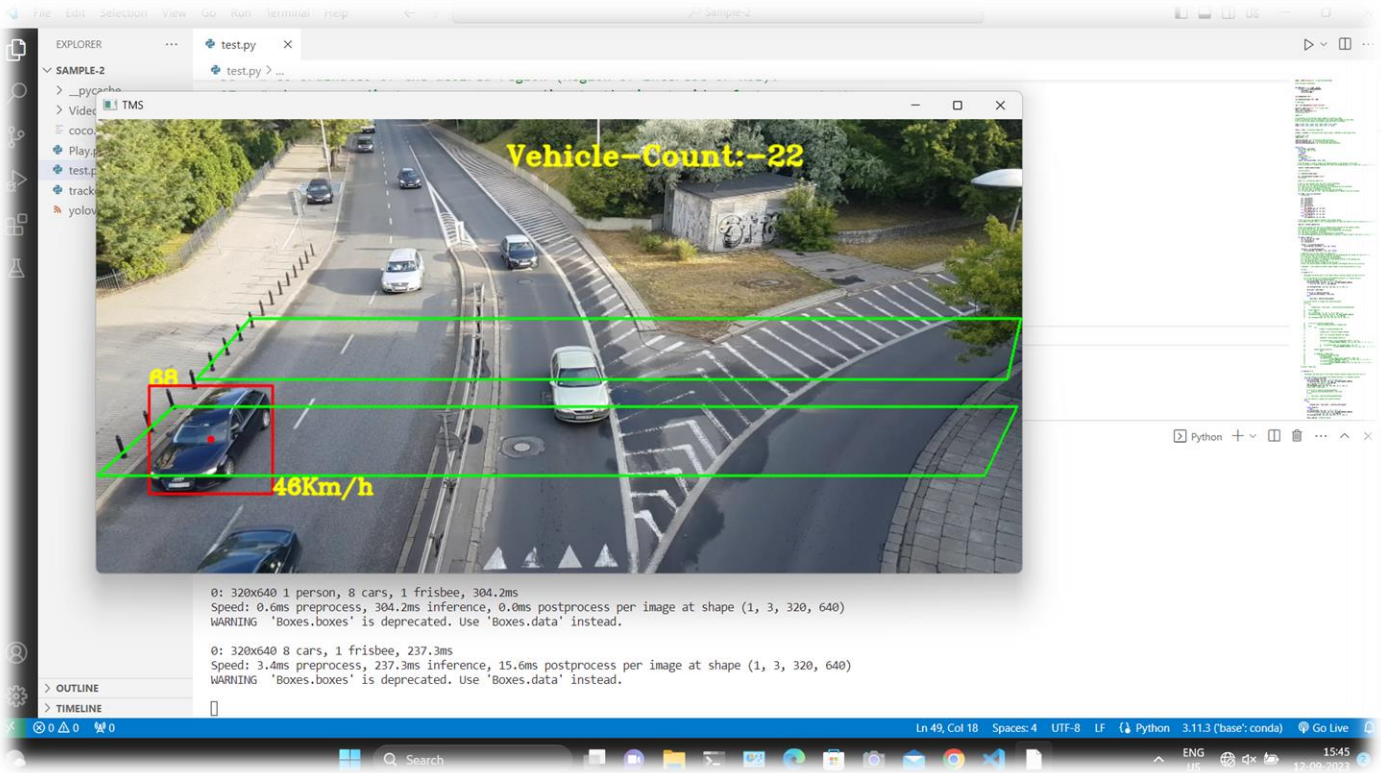
The approach proposed in this project is tested on four different videos. The average detection accuracy achieved by proposed approach is quite satisfactory. Maximum tracking accuracy achieved by the proposed technique is up to **95%**. Our algorithm was able to track, detect, and count most of the vehicles and also detect vehicle speed accurately. It was also able to detect if any vehicle violated the speed limit.

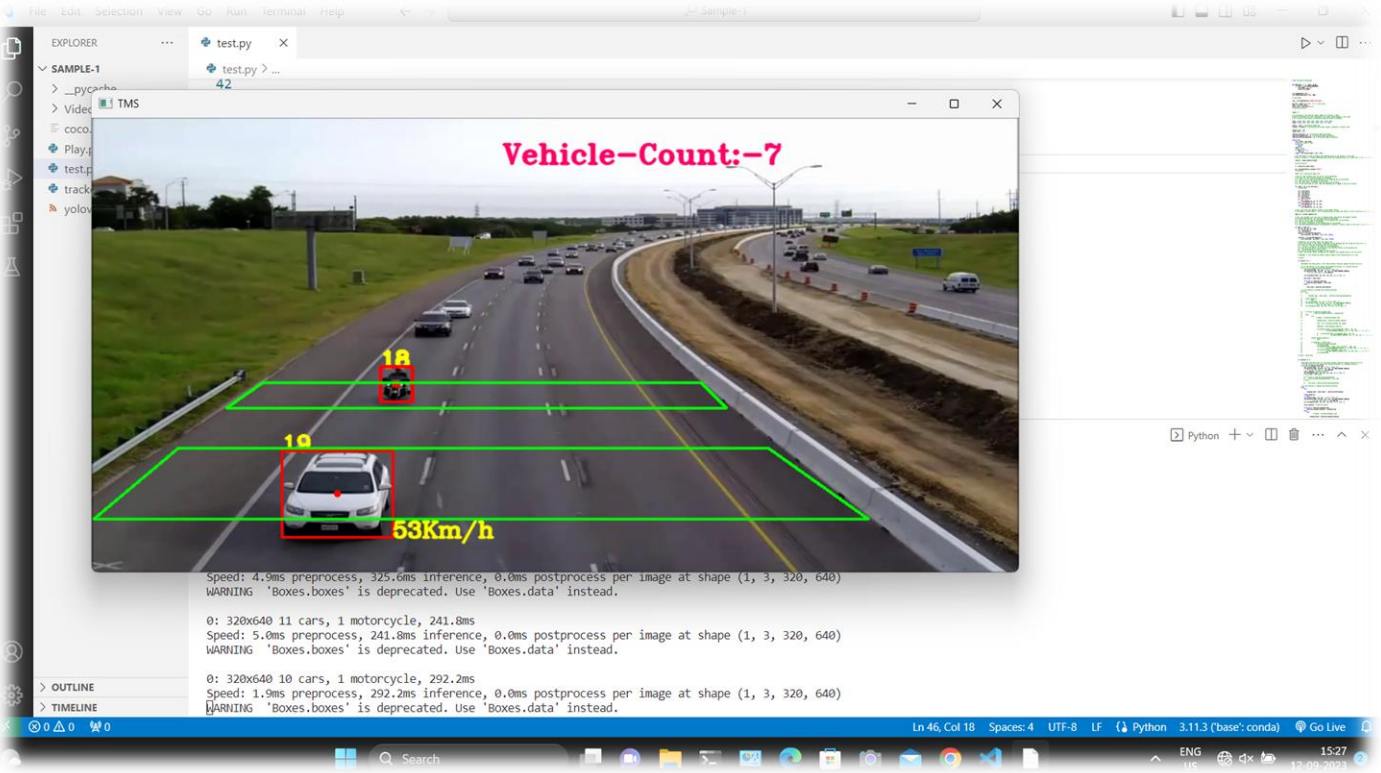
We have used the YOLOv8's pretrained model (e.g., **yolov8s.pt**). All YOLOv8 models for object detection are already pre-trained on the **COCO dataset,** which is a huge collection of images of 80 different types.

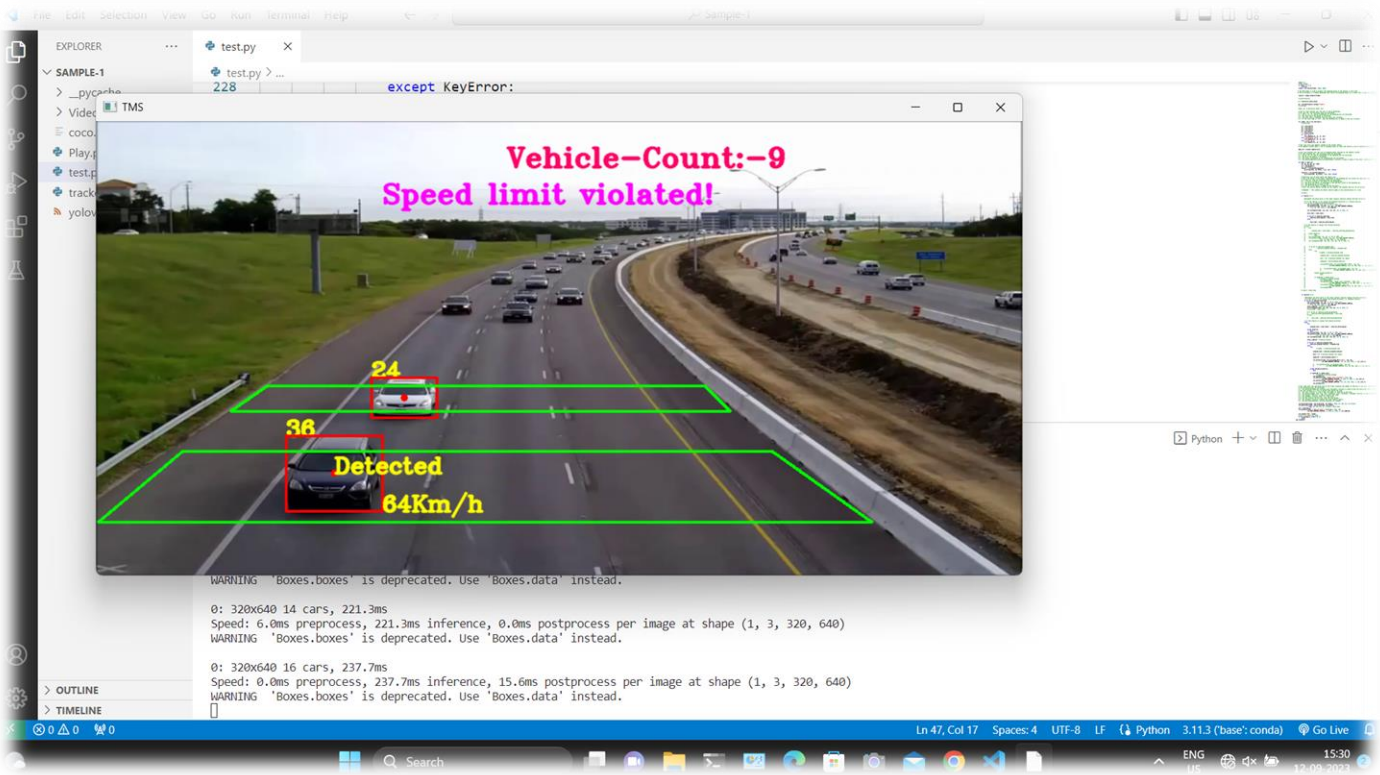
Output:

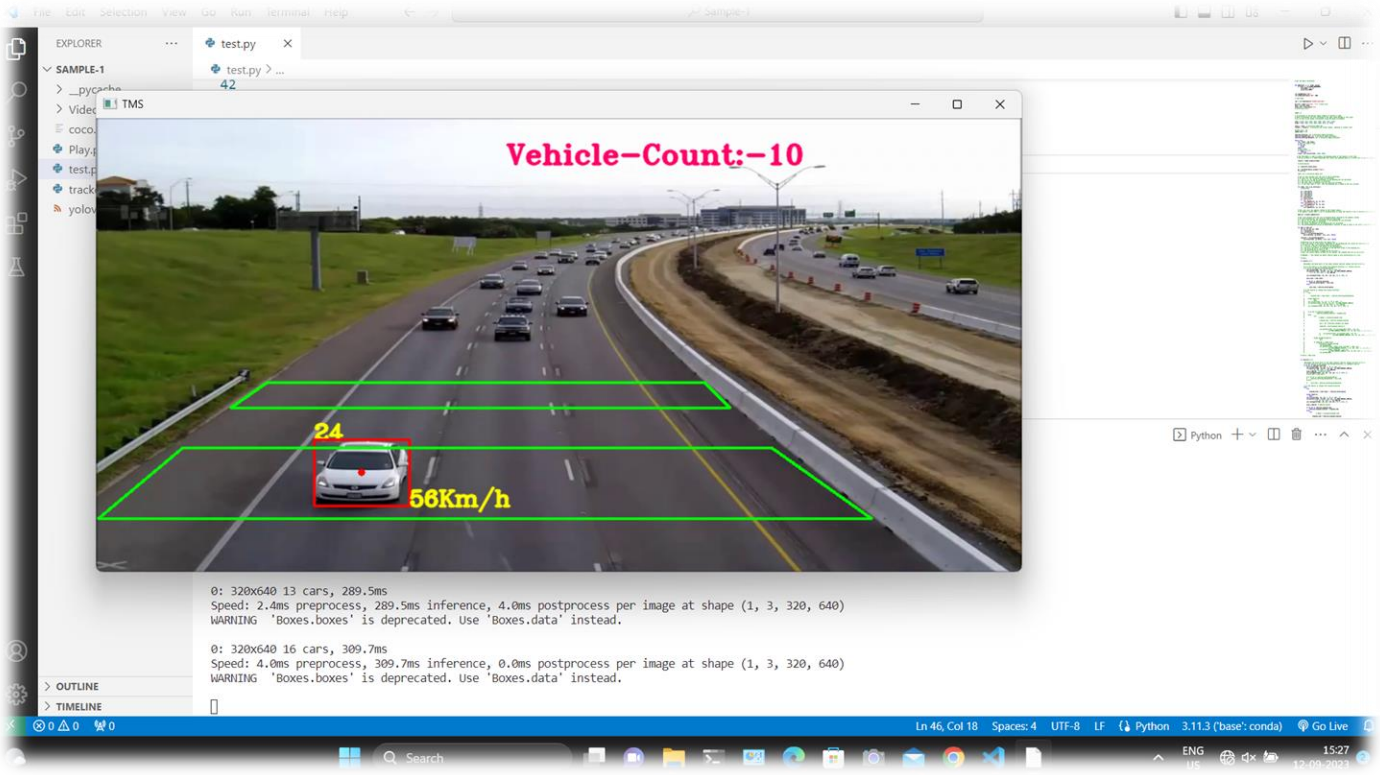

TMS
Vehicle-Count:-4
Speed limit violated!
10
Detected
62Km/h
0: 320x640 1 person, 4 cars, 240.5ms
Speed: 5.8ms preprocess, 240.5ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 1 person, 4 cars, 269.5ms
Speed: 0.0ms preprocess, 269.5ms inference, 15.6ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.


TMS
Vehicle-Count:-1
Speed limit violated!
1
0
Detected
67Km/h
0: 320x640 7 cars, 1 umbrella, 337.8ms
Speed: 5.0ms preprocess, 337.8ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 1 person, 8 cars, 1 umbrella, 1 frisbee, 349.0ms
Speed: 0.0ms preprocess, 349.0ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.


TMS
Vehicle-Count:-22
88
46Km/h
0: 320x640 1 person, 8 cars, 1 frisbee, 304.2ms
Speed: 0.6ms preprocess, 304.2ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 8 cars, 1 frisbee, 237.3ms
Speed: 3.4ms preprocess, 237.3ms inference, 15.6ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.


TMS
Vehicle-Count:-7
18
19
53Km/h
Speed: 4.9ms preprocess, 325.6ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 11 cars, 1 motorcycle, 241.8ms
Speed: 5.0ms preprocess, 241.8ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 10 cars, 1 motorcycle, 292.2ms
Speed: 1.9ms preprocess, 292.2ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.


TMS
Vehicle-Count:-9
Speed limit violated!
24
36
Detected
64Km/h
except KeyError:
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 14 cars, 221.3ms
Speed: 6.0ms preprocess, 221.3ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 16 cars, 237.7ms
Speed: 0.0ms preprocess, 237.7ms inference, 15.6ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.


TMS
Vehicle-Count:-10
24
56Km/h
0: 320x640 13 cars, 289.5ms
Speed: 2.4ms preprocess, 289.5ms inference, 4.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.
0: 320x640 16 cars, 309.7ms
Speed: 4.0ms preprocess, 309.7ms inference, 0.0ms postprocess per image at shape (1, 3, 320, 640)
WARNING 'Boxes.boxes' is deprecated. Use 'Boxes.data' instead.

Summary

This project proposes an approach to **detect and track the moving vehicles and estimation of their speeds**. The innovation of the approach lies in the selection of the **Region of Interest** for the vehicle detection. The **traffic monitoring system** developed in this project can be used for a variety of applications, such as traffic management, public safety, parking management, and smart cities.

The system uses **YOLOv8**, a state-of-the-art object detection algorithm, which is known for its speed and accuracy. YOLOv8 is a one-stage object detection algorithm, which means that it can detect and classify objects in a single pass of the image. This makes it faster than other object detection algorithms, such as Faster R-CNN. YOLOv8 uses a modified version of the CSPDarknet53 architecture as its backbone, which is known for its speed and accuracy. YOLOv8 also uses a number of other features to improve its performance, such as spatial attention, feature fusion, and context aggregation.

Ultralytics is a company that develops deep learning models for computer vision tasks. They are the creators of YOLOv8 and other popular deep learning models. Ultralytics also provides a number of tools and resources for training and deploying deep learning models. **Deep learning** is a type of machine learning that uses artificial neural networks to learn from data. Neural networks are inspired by the human brain and are able to learn complex patterns from data. Deep learning has been shown to be very effective for a variety of tasks, including object detection, image classification, and natural language processing.

YOLOv8 has been shown to be effective in a variety of object detection tasks, including traffic monitoring, pedestrian detection, and face detection. It is a powerful tool that can be used to solve a variety of real-world problems. The system was evaluated on the YOLOv8's pretrained model (e.g., **yolov8s.pt**) and was able to detect vehicles with an accuracy of 95%.

In addition to **YOLOv8**, we also used the **centroid tracking algorithm** to track the vehicles. The centroid tracking algorithm works by tracking the centroids of the objects detected by YOLOv8. The centroids are the points at the center of the objects. The centroid tracking algorithm then tracks the movement of the objects by tracking the movement of their centroids.

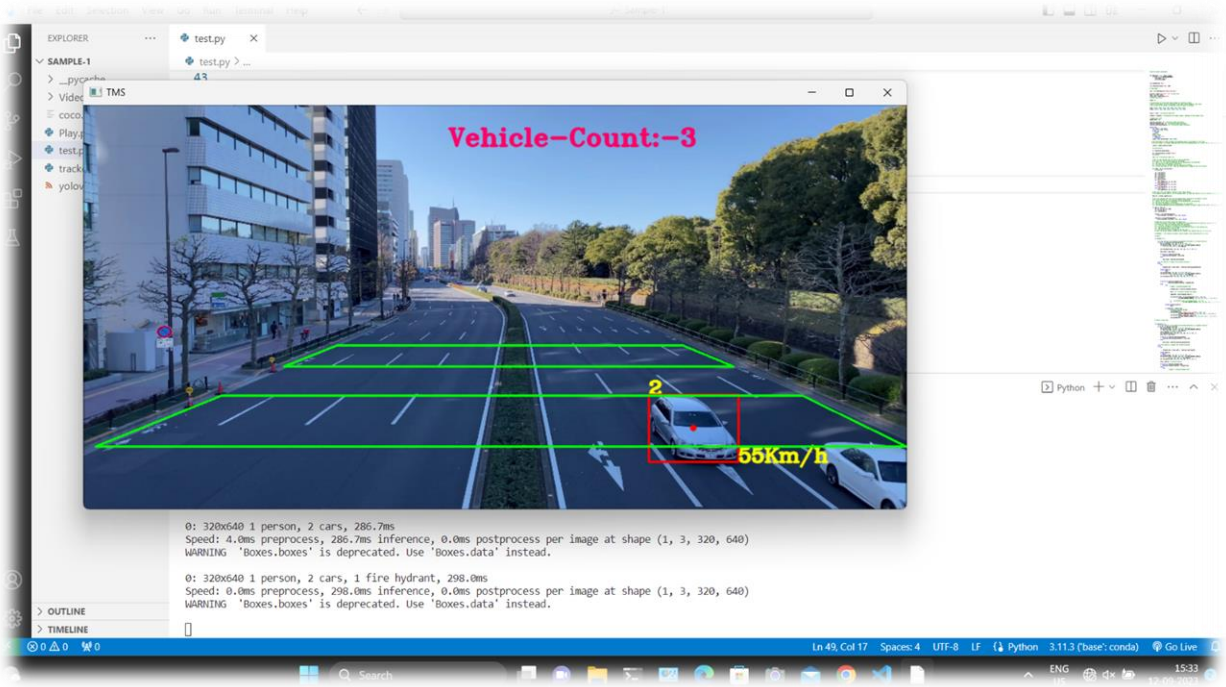
We have used the YOLOv8's pretrained model (e.g., **yolov8s.pt**). All YOLOv8 models for object detection are already pre-trained on the **COCO dataset,** which is a huge

collection of images of 80 different types. A pretrained model is a model that has been trained on a large dataset of images. This means that the model already has some knowledge of objects and their appearance. Using a pretrained model can help to speed up the training process for the traffic monitoring system.

The training method for YOLOv8 is a supervised learning method. This means that the model is trained on a dataset of labeled images. The labels for the images specify the objects that are present in the image and their locations. The model is then trained to predict the labels for new images.

The system has several limitations, such as the difficulty of detecting vehicles in difficult conditions, such as occlusion or poor lighting. However, the system is a promising approach to traffic monitoring and can be improved in future work.

The development of this system was a valuable learning experience. We learned about the use of deep learning for object detection, the challenges of traffic monitoring, and the potential applications of this technology.



Traffic-Monitoring-System

Future Scope

The future scope of the project is to improve the accuracy of the system, extend its capabilities, deploy it in a real-world setting, and integrate it with other systems.

To improve the accuracy of the system, a larger and more diverse dataset of traffic images can be used. A more powerful deep learning model can also be used. Overall, the future scope of the project is to develop a robust and reliable traffic monitoring system that can be used to improve traffic safety and efficiency.

Bibliography

- **https://www.w3schools.com/python/**
- **https://www.geeksforgeeks.org/python-programming-language/**
- **https://docs.ultralytics.com/**
- **https://www.geeksforgeeks.org/opencv-python-tutorial/**
- **https://www.w3schools.com/python/numpy**
- **https://www.w3schools.com/python/pandas**
- **https://www.analyticsvidhya.com/blog/2022/05/a-tutorial-on-centroid-tracker-counter-system/**