GRAVTON

Multi modal model for Nvidia Based Communication System

[Description goes here]

Prepared By **Vaishnavi Swarnoj**

Reviewed By **Bhanu Prakash**

Aim: To Build a Multi modal model for NVIDIA Based Communication System

Abstract:

The development of semi-autonomous electric two-wheelers represents a significant advancement in the integration of electric mobility with intelligent driving technologies. This paper presents a comprehensive approach to designing and implementing a semi-autonomous electric two-wheeler, focusing on essential features such as traffic light detection, vehicle proximity sensing, adaptive speed control based on battery state of charge (SoC), and strategic battery station navigation. Semi-autonomous electric two-wheelers offer a promising solution by combining the environmental benefits of electric mobility with the safety and efficiency of autonomous driving technologies. This research aims to explore the design and development of such a vehicle, focusing on key functionalities that enhance rider safety and vehicle efficiency without full autonomy. Future work will focus on refining these systems and conducting extensive field testing to validate their effectiveness in real-world scenarios.

Literature Review:

Multi-modal models:

A multi-modal model is an AI system designed to simultaneously process multiple forms of sensory input, similar to how humans experience the world. Unlike traditional uni-modal AI systems, trained to perform a specific task using a single sample of data, multi modal models are trained to integrate and analyze data from various sources, including text, images, audio, and video. This approach allows for a more comprehensive and nuanced understanding of the data, as it incorporates the context and supporting information essential for making accurate predictions. The learning in a multi modal model involves combining disjointed data collected from different sensors

and data inputs into a single model, resulting in more dynamic predictions than in uni modal systems. Using multiple sensors to observe the same data provides a more complete picture, leading to more intelligent insights.

To achieve this, multi modal models rely on deep learning techniques that involve complex neural networks. The model's encoder layer transforms raw sensory input into a high-level abstract representation that can be analyzed and compared across modalities. The input/output mixer layer combines information from the various sensory inputs to generate a comprehensive representation of the data, while the decoder layer generates an output that represents the predicted result based on the input. Multi modal models change how AI systems operate by mimicking how the human brain integrates multiple forms of sensory input to understand the world. With the ability to process and analyze multiple data sources simultaneously, the multi modal model offers a more advanced and intelligent approach to problem-solving, making it a promising area for future development.

Multimodal vs. Unimodal AI models

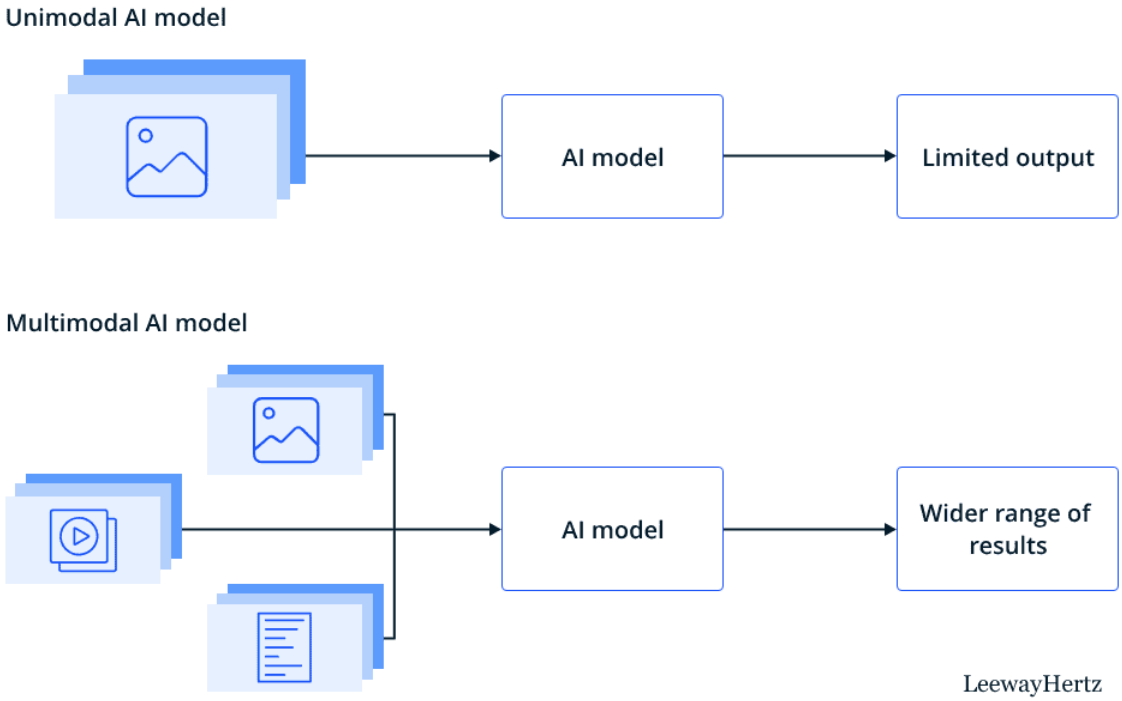


Fig 1 - Uni-modal vs. Multi modal models

The multi modal and uni modal models represent two different approaches to developing artificial intelligence systems. While the uni-modal model focuses on training systems to perform a single task using a single data source, the multi modal model seeks to integrate multiple data sources to analyze a given problem comprehensively. Here is a detailed comparison of the two approaches:

- **Scope of data:** Uni-modal AI systems are designed to process a single data type, such as images, text, or audio. In contrast, multi modal AI systems are designed to integrate multiple data sources, including images, text, audio, and video.
- **Complexity:** Uni modal AI systems are generally less complex than multi modal AI systems since they only need to process one type of data. On the other hand, multi modal AI systems require a more complex architecture to integrate and

analyze multiple data sources simultaneously.

- **Context:** Since uni modal AI systems focus on processing a single type of data, they lack the context and supporting information that can be crucial in making accurate predictions. Multi modal AI systems integrate data from multiple sources and can provide more context and supporting information, leading to more accurate predictions.
- **Performance:** While uni modal AI systems can perform well on tasks related to their specific domain, they may struggle when dealing with tasks requiring a broader context understanding. Multi modal AI systems integrate multiple data sources and can offer more comprehensive and nuanced analysis, leading to more accurate predictions.
- **Data requirements:** Uni modal AI systems require large amounts of data to be trained effectively since they rely on a single type of data. In contrast, multi modal AI systems can be trained with smaller amounts of data, as they integrate data from multiple sources, resulting in a more robust and adaptable system.
- **Technical complexity:** Multi modal AI systems require a more complex architecture to integrate and analyze multiple sources of data simultaneously. This added complexity requires more technical expertise and resources to develop and maintain than uni modal AI systems.

How does the multi modal model work?

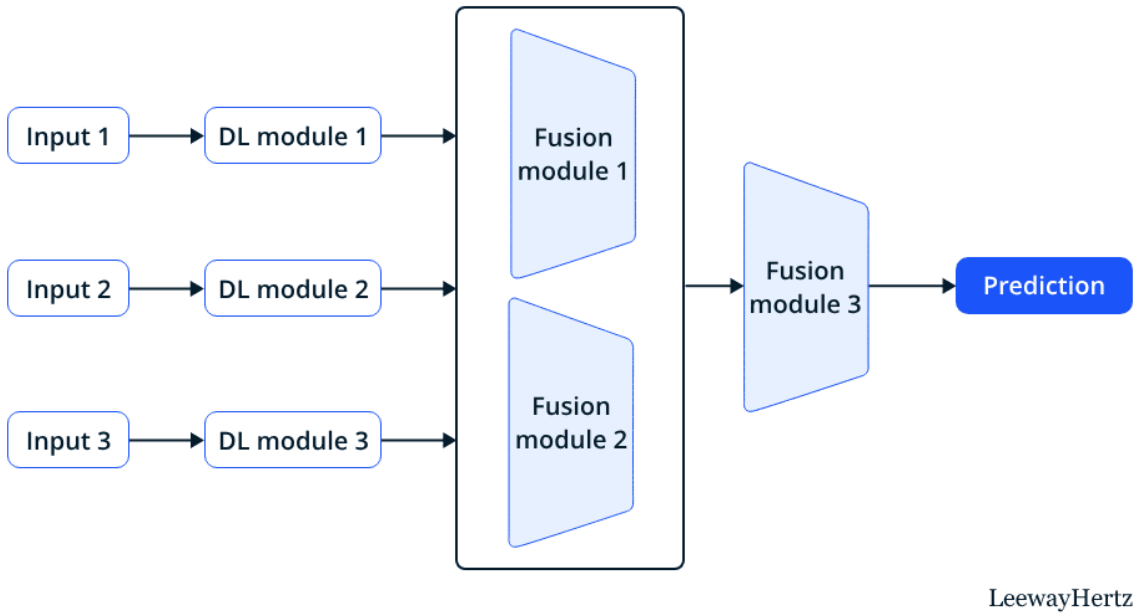


Fig 2 - Overview of multi modal models

Multi modal AI combines multiple data sources from different modalities, such as text, images, audio, and video. The process starts with individual uni modal neural networks trained on their respective input data, using Convolutional neural networks for images and recurrent neural networks for text. The output of these networks is a set of features that capture the essential characteristics of the input. To accomplish this, they are composed of three main components, starting with the uni modal encoders. These encoders are responsible for processing the input data from each modality separately. For instance, an image encoder would process an image, while a text encoder would process text.

Once the uni modal encoders have processed the input data, the next component of the

architecture is the fusion network. The fusion network's primary role is to combine the features extracted by the uni modal encoders from the different modalities into a single representation. This step is critical in achieving success in these models. Various fusion techniques, such as concatenation, attention mechanisms, and cross-modal interactions, have been proposed for this purpose.

Finally, the last component of the multi modal architecture is the classifier, which makes predictions based on the fused data. The classifier's role is to classify the fused representation into a specific output category or make a prediction based on the input. The classifier is trained on the specific task and is responsible for making the final decision.

One of the benefits of multi modal architectures is their modularity which allows for flexibility in combining different modalities and adapting to new tasks and inputs. By combining information from multiple modalities, a multi modal model offers more dynamic predictions and better performance compared to uni modal AI systems. For instance, a model that can process both audio and visual data can better understand speech than a model that only processes audio data.

Encoders in multi modal models

Encoders are pivotal in translating raw data from different modalities into a compatible feature space. Each encoder is specialized:

- **Image encoders:** Convolutional Neural Networks (CNNs) are a common choice for processing image data. They excel in capturing spatial hierarchies and patterns within images.
- **Text encoders:** Transformer-based models like BERT or GPT have revolutionized

text encoding. They capture contextual information and nuances in language, far surpassing previous models.

- **Audio encoders:** Models like WaveNet and DeepSpeech translate raw audio into meaningful features, capturing rhythm, tone, and content.

Fusion in multi modal models

Fusion strategies integrate information from various encoders, ensuring that the multi modal model capitalizes on the strengths of each modality:

- **Early fusion:** Integrates raw data or initial features from each modality before passing them through the model, promoting early interaction between modalities.
- **Late fusion**: Combines the outputs of individual modality-specific models, allowing each modality to be processed independently before integration.
- **Hybrid fusion:** Combines aspects of both early and late fusion, aiming to balance the integration of modalities at different stages of processing.
- **Cross-modal fusion:** Goes beyond simple combination, often employing attention or other mechanisms to dynamically relate features from different modalities, enhancing the model's ability to capture inter-modal relationships [1].

CARLA (CAR Learning to Act) is an open simulator for urban driving, developed as an open-source layer over Unreal Engine 4. Technically, it operates similarly to, as an open source layer over Unreal Engine 4 that provides sensors in the form of RGB cameras (with customizable positions), ground truth depth maps, ground truth semantic segmentation maps with 12 semantic classes designed for driving (road, lane marking, traffic sign, sidewalk and so on), bounding boxes for dynamic objects in the environment, and measurements of the agent itself (vehicle location and orientation)

[8].

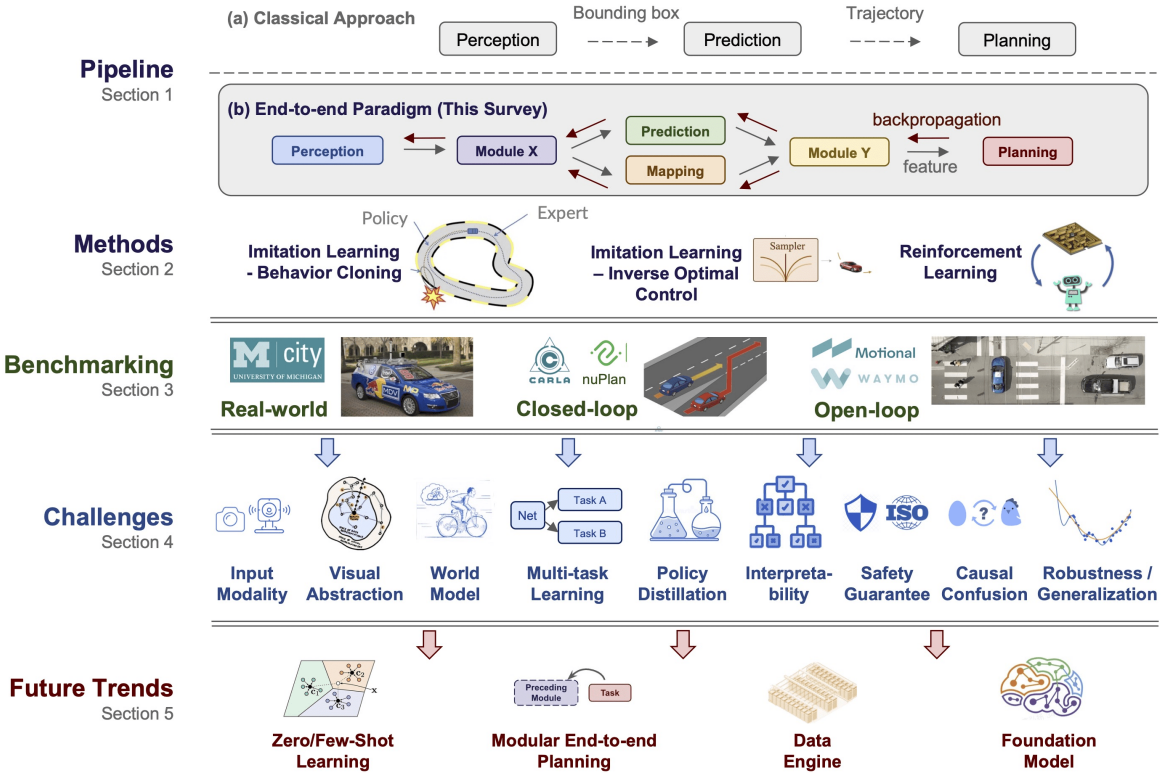


Fig 3 - Architecture of CARLA simulator

Perception in the context of multi modal AI for autonomous electric vehicles (EVs) refers to the ability of the vehicle to interpret and understand its environment using data from multiple sensors [2]. These sensors can include cameras, LiDAR, radar, ultrasonic sensors, and others. The goal of perception is to create a comprehensive and accurate representation of the vehicle's surroundings to enable safe and effective navigation and decision-making.

Key Components of Perception in Multi modal AI for Autonomous EVs

1. **Sensor Fusion**:
    - Combining data from multiple sensors to get a more accurate and reliable understanding of the environment than any single sensor could provide alone.

- Example: Combining LiDAR data for precise distance measurements with camera images for object recognition.

2. **Object Detection and Classification**:
   - Identifying and categorizing objects in the vehicle's vicinity, such as other vehicles, pedestrians, cyclists, road signs, and obstacles.
   - Techniques: Convolutional Neural Networks (CNNs), Region-based CNNs (R-CNNs), YOLO (You Only Look Once), and SSD (Single Shot Multi box Detector).
3. **Tracking and Prediction**:
   - Continuously monitoring the position and movement of detected objects and predicting their future trajectories.
   - Techniques: Kalman Filters, Particle Filters, and Recurrent Neural Networks (RNNs) such as LSTMs (Long Short-Term Memory networks).
4. **Semantic Segmentation**:
   - Classifying each pixel in the sensor data to understand the scene at a fine-grained level (e.g., distinguishing between road, sidewalk, vehicles, pedestrians).
   - Techniques: Fully Convolutional Networks (FCNs), U-Net, SegNet, and DeepLab.
5. **Localization and Mapping**:
   - Determining the precise location of the vehicle within its environment and creating a map of the surroundings.
   - Techniques: Simultaneous Localization and Mapping (SLAM), Visual Odometer, and GPS integration.
6. **Environmental Understanding**:
   - Interpreting the scene to understand contextual information, such as traffic light status, road markings, and traffic signs.
   - Techniques: Computer Vision algorithms, Natural Language Processing (for interpreting textual information), and Rule-based systems [10].

The auto industry has invested a lot of time and money in advanced driver-assistance systems (ADAS). This is extremely positive. It's been responsible for producing faster processors and more complex sensors with smaller footprints. In a recent study of people who bought cars with ADAS, over half said it prevented a crash in the first 90 days of ownership.

That being said, it's different in the motorcycle industry. This is because motorcycles are inherently different from any other vehicle on the road.

This difference can be characterized by the following unique vehicle dynamics:

- How they shift lanes
- If and how they are recognized by drivers
- The ways in which they maneuver
- How they accelerate & brake
- How other road users navigate around them

So, while automotive ADAS have made great headway in reducing car accidents, riders are still in need of more protection. Motorcycles are no longer solely for recreation, they are also becoming increasingly common for the same day-to-day use as any vehicle on four wheels. So, shouldn't riders be offered the same level of safety? This is where Damon's CoPilot Advancing Warning System for Motorcycles (AWSM) comes in. Our 360º protection system incorporates advanced camera vision, object detection, and processor segmentation. This is all complemented by a ground-breaking motorcycle AI engine [6].

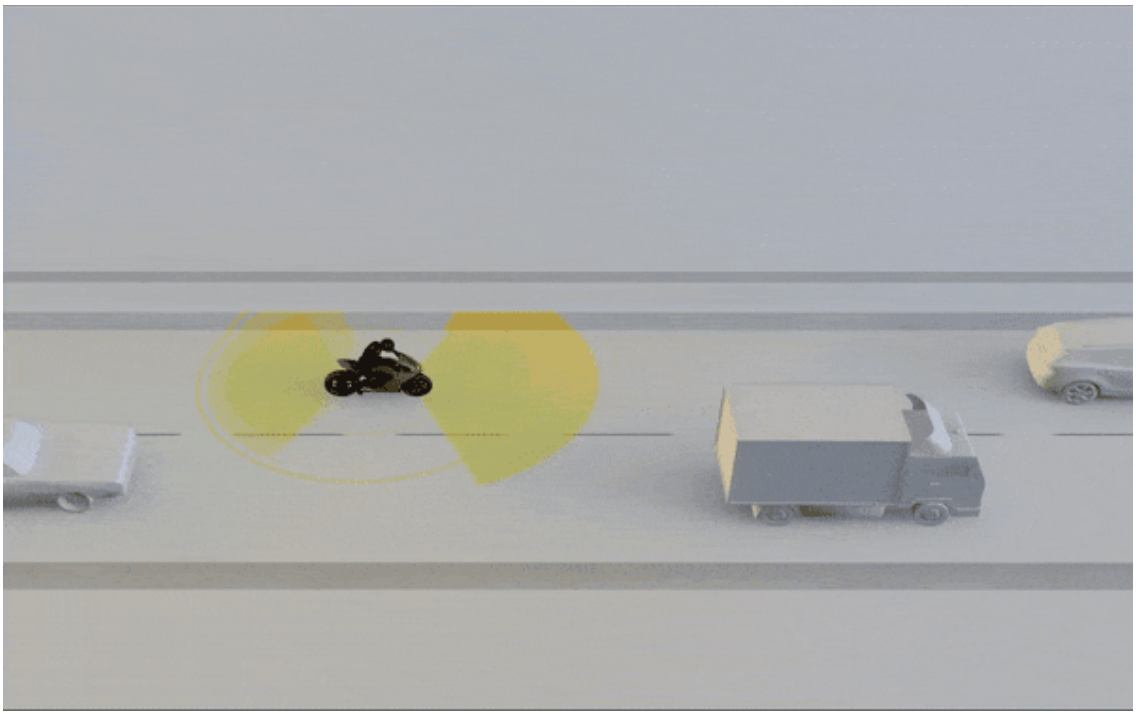


Fig 5 - Figure depicting AWSM in motor cycles

To build a model that predicts collision warnings, collision directions, and vehicle current speed using the provided features, you can consider using a stacking approach. Here's a possible way to implement it:

Step 1: Feature Engineering

1. **Extract relevant features:** From the given features, extract relevant information that can help in predicting the targets. For example:
    - Calculate the relative distance and velocity between the vehicle and the nearest vehicle or object.
    - Calculate the time to collision (TTC) based on the distance and velocity.
    - Extract features from the battery SOC, such as the rate of change or the remaining capacity.
2. **Create interaction terms:** Create interaction terms between features that might be relevant for predicting the targets. For example, the interaction between

vehicle speed and distance to the nearest vehicle.

Step 2: Base Models

1. **Collision Warning:** Train a binary classification model, such as:
    - Logistic Regression
    - Decision Trees
    - Random Forest
    - Support Vector Machines (SVM)
    - Neural Networks
2. **Collision Direction:** Train a multi-class classification model, such as:
    - Decision Trees
    - Random Forest
    - SVM
    - Neural Networks
3. **Vehicle Current Speed:** Train a regression model, such as:
    - Linear Regression
    - Decision Trees
    - Random Forest
    - Neural Networks

Step 3: Stacking

1. **Train a meta-model:** Train a meta-model that takes the predictions of the base models as input features. This can be a simple model, such as:
    - Logistic Regression
    - Decision Trees
    - Random Forest
    - Neural Networks

2. **Stacking configuration:** Configure the stacking approach by deciding how to combine the predictions of the base models. Some options include:
    - Stacking with averaging: Take the average of the predictions from each base model.
    - Stacking with voting: Take the majority vote of the predictions from each base model.
    - Stacking with weighted averaging: Assign weights to each base model based on their performance and take a weighted average of their predictions.

In Random Forest algorithm, `n_estimators` and `max_depth` are two important hyper parameters that significantly affect the behavior and performance of the model.

1. **n_estimators**:
    - **Significance**: `n_estimators` defines the number of decision trees in the forest.
    - **Effect**:
        - Increasing `n_estimators` usually improves the performance of the model until a certain point.
        - More trees generally lead to better generalization and robustness of the model.
        - However, adding too many trees can increase the computational cost and might lead to over fitting if not controlled properly.
    - **Rule of Thumb**: Increasing `n_estimators` will increase the model's complexity and training time but may improve performance up to a certain point. It's essential to find a balance where increasing `n_estimators` doesn't lead to diminishing returns.
2. **max_depth**:
    - **Significance**: `max_depth` controls the maximum depth of each decision tree in the forest.
    - **Effect**:
        - Controlling `max_depth` helps in limiting the depth of each tree in the forest.
        - Deeper trees can capture more complex relationships in the data

but are prone to over fitting.
    - Shallower trees are simpler and less likely to over fit but may not capture complex patterns in the data.
- **Rule of Thumb**: Controlling `max_depth` is crucial to prevent over fitting. It's often a good practice to start with a shallow tree and increase depth gradually while monitoring model performance on validation data.

Test Methodology:

Data plays a crucial role in building of any model, so first data has been prepared using various models and logics

Dataset Overview

The dataset contains 9000 entries with the following 11 columns:

1. **obj1_cord**: Coordinates of the our vehicle.
2. **obj1**: Class of the vehicle (EV).
3. **obj2_cord**: Coordinates of the other object (another vehicle or an obstacle).
4. **obj2**: Class of the other vehicle (e.g., truck, car).
5. **distance**: Distance between `obj1` and `obj2`.
6. **Others_speed**: Speed of the other vehicle (obj2).
7. **Current_speed**: Speed of the current vehicle (obj1)
8. **Battery SOC**: State of Charge of the vehicle's battery.
9. **Nearest Battery distance**: Distance to the nearest battery station.
10. **collide warn**: Collision warning status.
11. **Collision Direction**: Direction of the potential collision.

Data Issues and Cleaning

1. **Missing Values**:
    - `Others_speed` has many missing values.
    - `Current_speed` has many missing values.
    - `Nearest Battery distance` has some missing values.

    - `collide warn` and `Collision Direction` have many missing values.
2. **Data Types**:
    - The coordinate columns (`obj1_cord`, `obj2_cord`) are stored as strings and need to be split into latitude and longitude for further analysis.

Data Preparation Steps

1. **Split Coordinates**: Separate latitude and longitude for `obj1_cord` and `obj2_cord`.
2. **Handle Missing Values**: Decide on a strategy to handle missing data (e.g., imputation, removal).
3. **Feature Engineering**: Extract additional features if needed.
4. **Encode Categorical Variables**: Convert categorical variables into numerical values if needed.
5. **Normalize/Standardize Features**: Scale features for better model performance.

Conditions to evaluate the current speed of the vehicle

- **Traffic Light**:
    - Speed depends upon the traffic light, whether it is red, green or yellow
    - Whenever the traffic light is red, the vehicle slows down and stops until the state of traffic light is changed
- **Battery SOC**:
    - Higher SOC might allow for slightly higher speeds to reach the charging station.
    - Let's assume that for every 10% increase in SOC, the speed can increase by 5%.
- **Distance to Nearest Station**:
    - The closer the station, the less conservative the speed can be.
    - Let's assume a linear decrease in speed as the distance to the nearest

station increases, with a reduction of 2 km/h for every 1 km increase in distance.

Navigation:

The primary aim of this Navigation is to simulate a journey from place A to place B, identifying and marking the locations of battery stations along the route.

Key Steps and Logic:

1. **Setting Up Coordinates:** The journey starts at place A and ends at place B, with coordinates specified for both locations.
2. **Graph and Route Calculation:** Using a map graph and Dijkshatra algorithm, the code calculates the shortest bike path between the starting and ending points.
3. **Battery Station Generation:** Along this calculated route, battery stations are placed at regular intervals. The code ensures that the stations are evenly spaced, making sure to insert new stations if the distance between points on the route exceeds the specified interval.
4. **Map Creation:** The map includes markers for the start and end points, as well as for the generated battery stations.
5. **Bike Movement Simulation:** The bike's movement along the route is simulated. For each point along the route, the code calculates the distance to the nearest battery station and updates the CSV file accordingly. The distance to the nearest station is stored in the DataFrame.

Data Preprocessing:

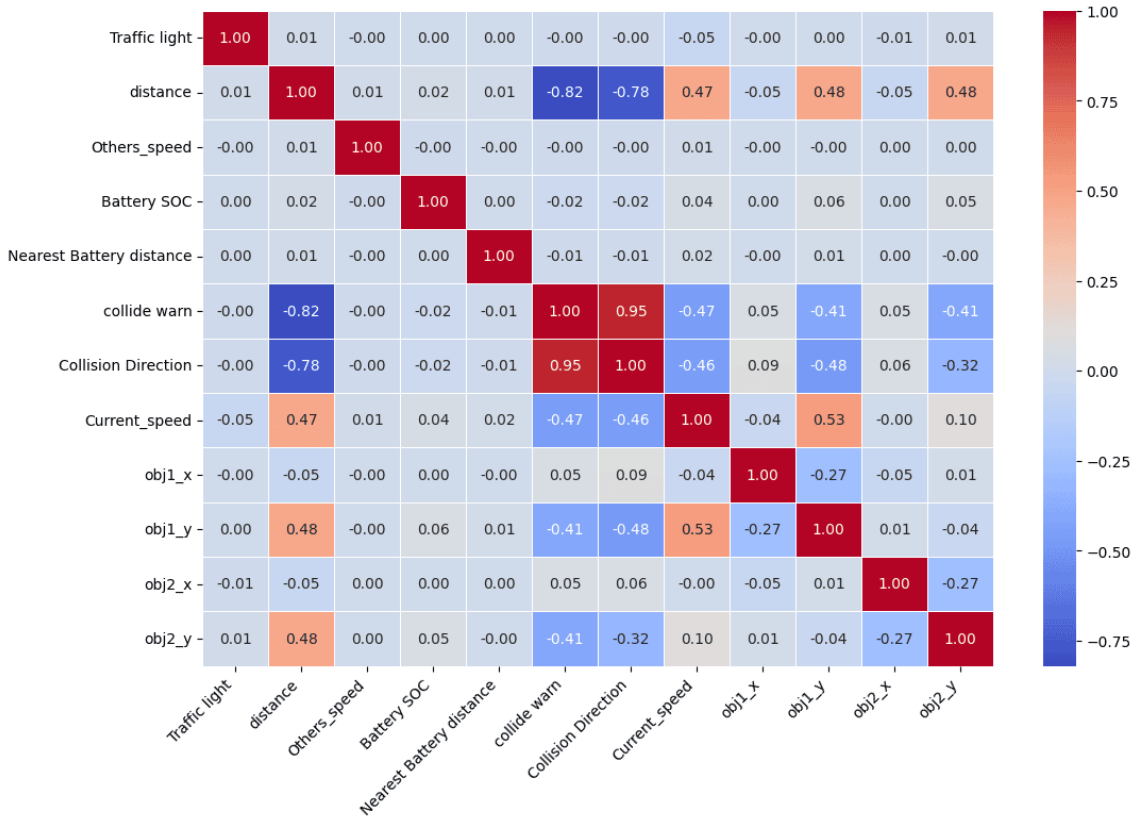


Figure depicting co-relation between multiple columns of the dataset (new_data.csv)

Data preprocessing is the initial step in model building, as described in the dataset overview section, there are multiple features in the dataset, using which model is to be trained on. Initially encoded traffic light colors as green-1, yellow-2, red-3 and null values are filled with 0, similarly encoded collision warn and collision direction features.

Data Transformation

Numerical columns are processed and transformed using Simple Imputer where as Categorical columns are transformed using both Simple Imputer and Label Encoder.

How `LabelEncoder` Works

`LabelEncoder` works by assigning a unique integer to each unique category (label) in a dataset. It maps each category to an integer between 0 and n-1, where n is the number of unique categories. This mapping allows machine learning algorithms, which generally require numerical input, to process categorical data.

Steps Involved in Label Encoding

1. **Identify Unique Categories**: The encoder first identifies all unique categories in the categorical feature.
2. **Assign Integer Labels**: Each unique category is assigned a unique integer.
3. **Transform Data**: The original categorical values are replaced with their corresponding integer labels.

How SimpleImputer Works:

1. **Identify Missing Values**: SimpleImputer identifies missing values (NaN) in the dataset.
2. **Imputation Strategy**: It then replaces these missing values with a specified statistic (mean, median, mode, or a constant value).
3. **Fit the Imputer**: In scikit-learn's convention, you first fit the imputer on the dataset to compute the statistic for imputation.
4. **Transform the Data**: Once the imputer is fitted, you can transform your dataset, replacing missing values with the computed statistic.

Parameters of SimpleImputer:

- **strategy**: Specifies the imputation strategy. It can be:
  - `"mean"`: Replaces missing values with the mean along each column.
  - `"median"`: Replaces missing values with the median along each column.
  - `"most_frequent"`: Replaces missing values with the most frequent value along each column.
  - `"constant"`: Replaces missing values with a constant specified by the `fill_value` parameter.
- **fill_value**: Used when strategy is `"constant"`. Specifies the constant value to replace missing values.

Model Selection

1. ***Collision Warning:*** For predicting collision warnings, you can use a binary classification model, such as:
    - Logistic Regression
    - Decision Trees
    - Random Forest
    - Support Vector Machines (SVM)
    - Neural Networks

Any of the above mentioned models can be used, but here Logistic Regression is used because Logistic regression is computationally efficient, both in terms of training and prediction. This can be particularly advantageous when dealing with large datasets or when deploying the model in real-time systems where quick predictions are essential.

Logistic Regression is a statistical method used for binary classification that models the probability of a given input belonging to one of two categories. Unlike linear regression, which predicts a continuous outcome, logistic regression predicts a discrete outcome using a logistic function to model the probability. The output is a value between 0 and 1, representing the probability that the given input belongs to the positive class. By applying a threshold (commonly 0.5), the model can classify inputs into two distinct classes. This technique is widely used in fields such as machine learning, medicine, and social sciences for tasks like spam detection, disease diagnosis, and customer churn prediction.

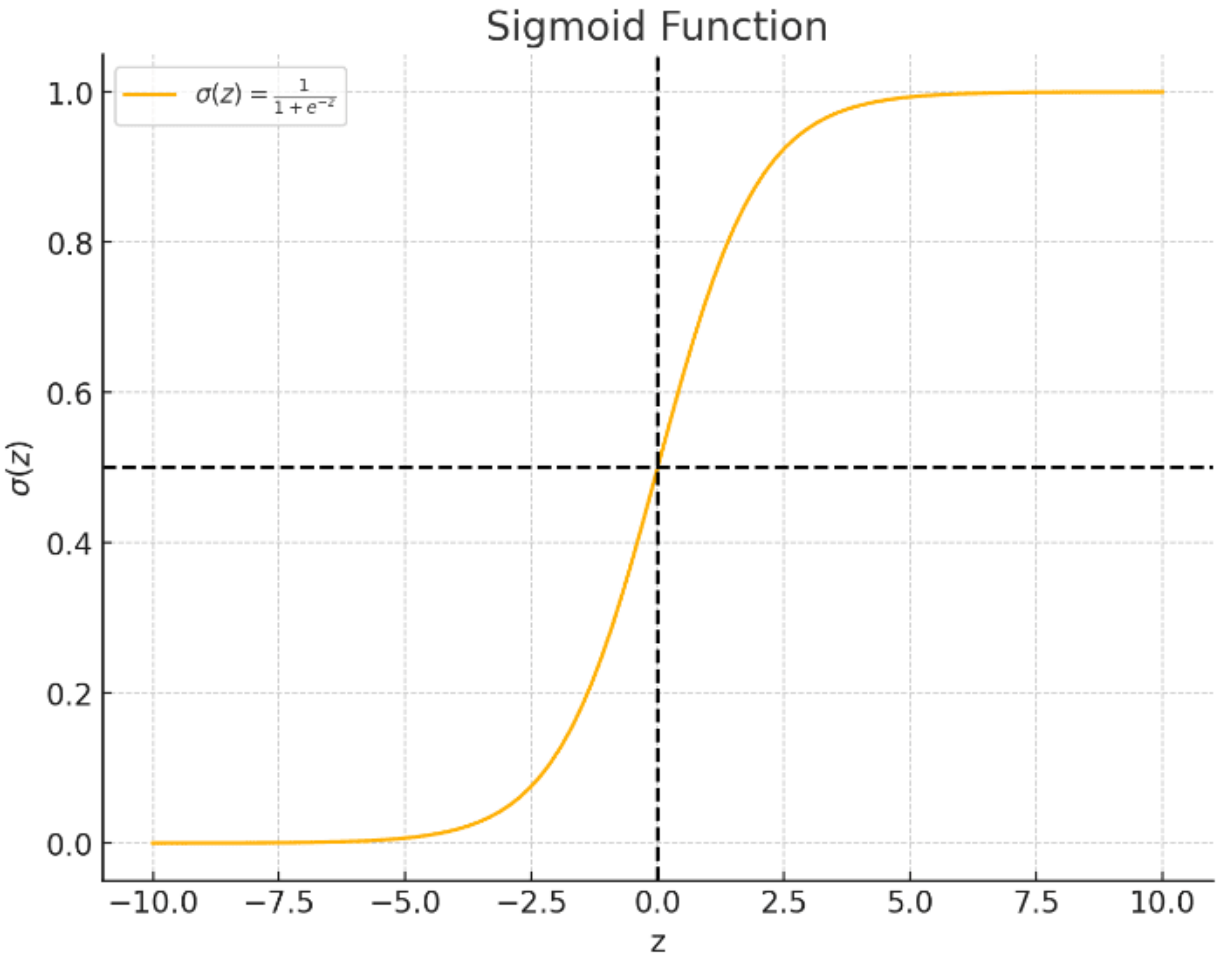


Fig 6 - Sigmoid curve depicting the functioning of Logistic Regression

- The function used by Logistic Regression for producing probabilistic predictions is Sigmoid function.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

- The vertical dashed line at z=0 indicates the point where the probability is 0.5.
- The horizontal dashed line at σ(z)=0.5 shows the threshold commonly used for binary classification.

Cost Function (Log Loss):

To measure the error between predicted probabilities and actual labels, logistic

regression uses the log loss (or cross-entropy loss) function:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right)$$

Where:

- m is the number of training examples.
- $h_\theta(x^{(i)})$ is the predicted probability that $x^{(i)}$ belongs to class 1.
- $y^{(i)}$ is the actual label (0 or 1) of the i-th training example.

2. ***Collision Direction:*** For predicting collision directions, you can use a multi-class classification model, such as:
    - Decision Trees
    - Random Forest
    - SVM
    - Neural Networks

Random Forest is often the preferred choice for multi-class classification tasks due to its accuracy, robustness against over fitting, ability to handle missing data, and feature importance insights. Its ease of use and scalability make it suitable for a wide range of applications, including predicting collision directions in various scenarios

Random Forest Classifier:

A Random Forest is an ensemble learning method that is primarily used for classification and regression tasks. It constructs multiple decision trees during training and outputs the mode of the classes (classification) or mean prediction (regression) of the individual trees.

Key Concepts

1. **Ensemble Learning**: Combining multiple models (trees in this case) to improve the overall performance.
2. **Decision Tree**: A tree-like model of decisions and their possible consequences.

3. **Bootstrap Aggregation (Bagging)**: Randomly sampling with replacement to create multiple datasets from the original dataset.

How It Works

1. **Data Sampling**:
    - The algorithm selects random samples from the original dataset with replacement (bootstrap samples).
2. **Decision Tree Construction**:
    - For each bootstrap sample, a decision tree is constructed.
    - At each node, a random subset of features is selected, and the best split is chosen from this subset (instead of considering all features).
3. **Voting/Averaging**:
    - For classification tasks, each tree in the forest votes for a class, and the class with the most votes is chosen.
    - For regression tasks, the mean prediction of all trees is taken.
3. ***Vehicle Current Speed:*** For predicting vehicle current speed, you can use a regression model, such as:
    - Linear Regression
    - Decision Trees
    - Random Forest
    - Neural Networks

When to Use Random Forest Regressor:

- **Complex Non-linear Relationships**: When the relationship between input features and target variable is non-linear and complex.
- **Medium to Large Datasets**: Random Forests perform well on medium to large-sized datasets due to their efficiency and parallelization capabilities.
- **Feature Importance Insights**: When understanding which features are most influential for predictions is important.

Random Forest is a supervised learning algorithm that builds multiple decision trees during training and outputs the mean prediction (regression) or the mode of the

predictions (classification) of the individual trees.

- **Ensemble Learning**: Random Forest is an ensemble method, meaning it combines multiple models to improve performance and robustness over a single estimator.
- **Decision Trees**: The base model used in Random Forest is the decision tree, which learns simple decision rules from the data features to predict the target variable.
- **Regression Task**: In the context of regression, Random Forest Regressor predicts a continuous output (numeric value) based on input features.

How Does Random Forest Regressor Work?

1. **Bootstrap Sampling (Bagging)**:
    - Random Forest builds multiple decision trees by randomly selecting a subset of data points (with replacement) from the training set. This process is called bootstrap sampling.
    - Each tree is trained independently on these bootstrap samples.
2. **Random Feature Selection**:
    - During the construction of each decision tree, at each node, a random subset of features is selected as potential candidates for splitting.
    - This randomness helps to decorrelate the trees, leading to more diverse and robust models.
3. **Decision Tree Construction**:
    - Each decision tree is grown fully (or up to a specified depth) based on the selected features.
    - Nodes are split based on criteria such as Gini impurity (for classification) or variance reduction (for regression).
4. **Prediction**:
    - For a new input instance, the Random Forest Regressor aggregates the predictions from all individual trees (mean of predictions for regression) to produce the final prediction.
    - This aggregation reduces over fitting and improves generalization compared to a single decision tree.

Model Training and Evaluation

1. Split the data: Split the dataset into training and testing sets (e.g., 70% for

training and 30% for testing).

2. Train the models: Train each model using the training data and tune hyper parameters using the validation set.
3. Evaluate the models: Evaluate the performance of each model using metrics relevant to the problem, such as:
    - Accuracy, Precision, Recall, and F1-score for collision warning and direction
    - Mean Absolute Error (MAE) or Mean Squared Error (MSE) for vehicle current speed
4. Select the best model: Select the best-performing model for each target attribute based on the evaluation metrics.

The process outlined involves three different machine learning tasks, each utilizing data preprocessing, model training, and evaluation. Here's a step-by-step breakdown:

First Task: Predicting Collision Warning

- **Data Loading**: Data is read from a file named 'objects.csv'.
- **Data Preparation**:
    - 'Traffic light' is converted to numerical values (green=1, yellow=2, red=3, missing=4, and any other missing values=0).
    - Missing values in 'Others_speed' are filled with the average value of the column.
    - 'collide warn' is converted to numerical values (Collision Warning=1, otherwise=0, and any missing values=0).
- **Feature and Target Selection**:
    - Features: 'Traffic light', 'distance', 'Others_speed', and 'Current_speed'.
    - Target: 'collide warn'.
- **Data Splitting**: The data is divided into training and testing sets (80% training, 20% testing).
- **Model Creation and Training**: A logistic regression model is created and trained using the training data.

- **Prediction and Evaluation**: The model makes predictions on the test data, and its accuracy is calculated and displayed.
- **Model Saving**: The trained model is saved to a file named 'lr_model.pkl'.

Second Task: Predicting Collision Direction

- **Data Loading**: Data is read from a file named 'updated_data.csv'.
- **Data Preparation**:
  - 'Traffic light' and 'collide warn' are converted to numerical values similarly to the first task.
  - 'Collision Direction' is mapped to numerical values (north=1, east=2, south=3, etc.).
  - Coordinates for two objects are extracted from a text format and split into separate columns.
  - Missing values in 'Others_speed' and 'Nearest Battery distance' are filled with their respective average values.
- **Feature and Target Selection**:
  - Features include the numerical columns such as object coordinates, speeds, distance, traffic light status, etc.
  - Target: 'Collision Direction'.
- **Data Splitting**: The data is divided into training and testing sets.
- **Model Creation and Training**: A Random Forest classifier is created with settings to prevent over fitting (like limiting tree depth and the number of estimators) and trained using the training data.
- **Prediction and Evaluation**: The model's accuracy is evaluated on both training and test data.
- **Model Saving**: The trained model is saved to a file named 'rf_model.pkl'.

Third Task: Current Speed Prediction

- **Loading the Dataset:**
  - The code starts by loading a dataset (`updated_data.csv`) using Pandas, which is a Python library for data manipulation and analysis.
- **Data Preprocessing Steps:**
  - **Mapping Categorical Values to Numerical Values:**
    - Several columns (`Traffic light`, `collide warn`, `Collision`

`Direction`) are transformed from categorical (textual) to numerical values. For example, traffic light colors like 'green', 'red', and 'yellow' are mapped to 1, 3, and 2 respectively.

- **Extracting Coordinates:**
  - Columns (`obj1_cord`, `obj2_cord`) containing coordinates in a specific format (e.g., "(x, y)") are parsed to extract numerical coordinates (`obj1_x`, `obj1_y`, `obj2_x`, `obj2_y`) and converted to float data type. The original coordinate columns are then dropped (`obj1_cord`, `obj2_cord`).
- **Handling Missing Values:**
  - Missing values in `Others_speed` and `Nearest Battery distance` columns are filled with the mean of their respective columns.
- **Label Encoding:**
  - Categorical columns (`obj1` and `obj2`) are encoded using LabelEncoder, which assigns a unique numerical value to each category within the column. Missing values are filled with a placeholder value (e.g., 'missing') before encoding.

- **Defining Features (X) and Target (y):**
  - The features (`X`) are defined as all columns in the dataset except for the target variable (`Current_speed`), which represents the current speed.
  - The target variable (`y`) is defined as `Current_speed`.
- **Identifying Numerical and Categorical Columns:**
  - Numerical columns (`numerical_columns`) include features that contain numerical data or have been transformed into numerical format.
  - Categorical columns (`categorical_columns`) include features that contain categorical data which has been label encoded.
- **Preprocessing:**
  - **Numerical Data:**
    - Missing values in numerical columns are imputed (filled) with the mean of each respective column using SimpleImputer.
    - The data is then standardized (scaled) using StandardScaler, which normalizes numerical data to have a mean of 0 and a standard deviation of 1.
- **Combining Preprocessing Steps:**
  - ColumnTransformer is used to combine the preprocessing steps for numerical data (`numerical_transformer`) with the original categorical

columns (`remainder='passthrough'`). This ensures that all data preprocessing steps are applied correctly to each subset of features.

- **Splitting the Dataset:**
  - The dataset (`X`) is split into training and testing sets (`X_train`, `X_test`, `y_train`, `y_test`) using train_test_split. This allows evaluation of the model's performance on unseen data.
- **Training the Random Forest Regressor:**
  - A RandomForestRegressor model is initialized with a specified random state for reproducibility.
  - The model is trained using the training data (`X_train`, `y_train`) to learn patterns between features and the target variable (`Current_speed`).
- **Predicting and Evaluating the Model:**
  - The trained model predicts the target variable (`Current_speed`) for the test data (`X_test`).
  - Model performance is evaluated using Mean Squared Error (MSE) and R-squared (R^2) Score metrics, which measure the accuracy and goodness of fit of the predictions compared to the actual values (`y_test`).
- **Saving the Model:**
  - The trained RandomForestRegressor model is saved as a file (`rf_reg.pkl`) using joblib, allowing it to be reused later for making predictions on new data.

Label Encoder:

The primary purpose of using a Label Encoder is to convert categorical data so that it can be used as input for machine learning algorithms, which typically work with numerical data. By assigning a unique numerical label to each unique category within a column of categorical data, the Label Encoder facilitates numerical computation and analysis.

How Label Encoder Works

1. **Mapping Categories to Numbers:**
   - The Label Encoder scans through the categorical data in a column and assigns a unique integer to each distinct category. For instance, if a column contains three categories 'red', 'green', and 'blue', the Label Encoder might assign them numerical labels 0, 1, and 2 respectively.
2. **Handling Missing Values:**
   - Before encoding, missing values in the categorical data are often replaced

with a placeholder value, such as 'missing', to ensure completeness during the encoding process.

3. **Encoding Process:**
    - Once the categories are mapped to numerical labels, the Label Encoder transforms the original categorical column into a new column containing these numerical labels.
4. **Application to Machine Learning:**

The transformed numerical labels can then be fed into machine learning algorithms for tasks like classification or regression. Algorithms such as decision trees or neural networks can then interpret these labels as input features for making predictions.

Warning System:

Data Preparation:

1. **Data Generation Functions:**
    - Several functions are defined to randomly simulate different aspects of driving conditions, like weather (dry, wet, icy, snowy), visibility (good, fair, poor), traffic lights (red, yellow, green), and others.
2. **Driving Rules Dictionary:**
    - There's a set of rules defined in the script that are important for safe driving. Each rule has a score indicating its priority.
3. **Main Function (`indian_two_wheeler_rules`):**
    - This function checks if the simulated driving conditions violate any of the predefined rules.
    - It generates messages advising on safe driving practices based on the violations detected.
    - The function calculates the safest speed for turning angles and braking distances based on simulated conditions.
4. **Data Generation Loop:**
    - The script runs a loop to generate a large number of random driving scenarios.
    - For each scenario, it calculates various driving parameters like speed, distance from vehicles, road conditions, etc.
    - It uses the `indian_two_wheeler_rules` function to determine the most important safety message for each scenario based on the predefined rules.
5. **Saving Data:**

    - The generated scenarios and their corresponding safety messages are stored in a file called `two_wheeler_driving_conditions.csv`.
    - This file can be used later for analysis or to train models on driving safety.
6. **Conclusion:**
    - The script is designed to simulate and analyze safe driving scenarios for two-wheelers under different conditions without using technical terms. Its purpose is to promote awareness and adherence to safe driving practices through simulated scenarios and messages.

Model Building:

- **Loading the Dataset:**
    - The script starts by loading a dataset (`two_wheeler_driving_conditions.csv`) containing simulated driving scenarios for two-wheelers.
- **Separating Features and Target Variable:**
    - It separates the dataset into two parts:
        - **Features (`X`)**: These are the characteristics of each driving scenario, like speed, weather, and traffic signals.
        - **Target variable (`y`)**: This represents the output message that advises on safe driving practices based on each scenario.
- **Defining Feature Types:**
    - Features are categorized into three types:
        - **Categorical features**: Describe qualitative aspects like traffic light color, road conditions, and visibility.
        - **Boolean features**: Represent yes/no or true/false attributes such as whether it's raining, daytime, or if the vehicle is on a highway.
        - **Numeric features**: Quantitative variables such as speed, distance from other vehicles, and turning angle.
- **Preprocessing Pipelines:**
    - **Boolean Features**: These are transformed into numerical form (0 for false, 1 for true) since many machine learning models work with numerical data.
    - **Numeric Features**: Missing values are filled with the median (middle value) of each feature and then scaled (standardized) to ensure all features contribute equally to the model.
    - **Categorical Features**: Missing values are replaced with the most frequent value and then encoded into numerical values using OrdinalEncoder.
- **ColumnTransformer:**

    - This combines all the preprocessing steps into a single transformer (`preprocessor`) that applies the appropriate transformations to each type of feature.
- **Data Transformation:**
    - The original dataset (`X`) undergoes transformation through the `preprocessor`, converting all features into a format suitable for training a machine learning model.
- **Splitting Data:**
    - The transformed data (`X`) and target variable (`y`) are split into training and testing sets using `train_test_split`. This division ensures the model can be trained on a portion of the data and tested on another to evaluate its performance.
- **Defining and Training the Model:**
    - A Random Forest classifier model is chosen as it's suitable for classification tasks (predicting categories or classes).
    - The model is trained on the training data (`X_train`, `y_train`) using the `fit` method.
- **Model Evaluation:**
    - The trained model predicts output messages for both the training and testing datasets.
    - **Training Accuracy**: Measures how well the model predicts the training data it was trained on.
    - **Testing Accuracy**: Assesses how accurately the model predicts output messages on new, unseen data (testing set).
    - **Classification Report**: Provides a detailed evaluation of the model's performance, including metrics like precision, recall, and F1-score for each output message category.
- **Saving the Model:**
    - Finally, the trained model is saved into a file (`warn_model.pkl`) using `joblib.dump`, allowing it to be used later for making predictions on new driving scenarios.

Ordinal encoder:

An Ordinal Encoder is a type of data transformation tool used in machine learning to convert categorical values into numerical values, specifically when the categorical values have a natural order or hierarchy.

Functionality:

- **Mapping Categorical to Numerical:** Ordinal Encoder assigns a unique numerical value to each category in a categorical feature. These numerical values maintain the order and relationship between categories.
- **Example Scenario:** Suppose you have a categorical feature like "road conditions" with categories: "dry", "wet", "icy", and "snowy". Ordinal Encoder will assign numerical values like 0, 1, 2, and 3 respectively, preserving the order from least severe to most severe condition.
- **Transformation Principle:** When applied to a dataset, Ordinal Encoder scans through the categorical feature, identifies unique categories, and maps them to integers in ascending order based on their position in the list of unique categories.

Model Retraining

- **Load Data**: It reads a CSV file named 'predictions.csv' into a data structure that holds the data in a table format.
- **Load Model**: It loads a pre-trained machine learning model from a file named 'lr_model.pkl'.
- **Filter Data**: It filters the rows of the data to include only those rows where the value in the 'collision_probability_score' column is less than 0.70.
- **Update Column**: In the filtered rows, it sets the value in the 'collision_probability' column to 1 for rows where the 'distance' column value is less than 0.1.
- **Prepare Data for Training**: It selects specific columns ('Traffic light', 'distance', 'Others_speed', and 'Current_speed') to be used as input for the model and the 'collision_probability' column as the output.
- **Train Model**: It retrains the previously loaded model using the selected input columns and the output column.
- **Save Model**: It saves the retrained model back to the file 'lr_model.pkl'.
- **Confirmation**: It prints a message confirming that the model has been saved.

Integrating all the models in the interface

HTML & JavaScript Part

Overall Structure

The HTML and JavaScript section creates a web page that allows users to choose between two types of predictions: collision prediction and speed prediction. It provides forms to input data for each prediction type and buttons to submit the data.

Selecting Prediction Type

- A dropdown menu lets users select either "Collision Prediction" or "Speed Prediction."
- A button labeled "Proceed" shows the appropriate form based on the selection.

Collision Prediction Form

- Users enter the coordinates and labels of two vehicles, the color of the traffic light, the distance between the vehicles, the speed of other vehicles, and the current speed.
- When the form is submitted, the input data is sent to the server using a fetch request to a specified URL.
- The server responds with a prediction about the likelihood of a collision and, if applicable, the direction of the collision.
- The result is displayed on the web page.

Speed Prediction Form

- Users enter similar details to the collision form but also include the battery state of charge, the distance to the nearest battery station, collision warning probability, and collision direction.
- When the form is submitted, the input data is sent to the server using a fetch request to another specified URL.
- The server responds with a predicted speed, which is then displayed on the webpage.

Flask (Python) Part

Setting Up the Server

- The server is created using Flask, a lightweight web framework.
- Cross-Origin Resource Sharing (CORS) is enabled to allow requests from

different domains.

Loading Models

- Pre-trained models are loaded from files. These models are used to make predictions based on the input data from the web forms.

Home Route

- A simple route that returns a message indicating the server is running.

Collision Prediction Endpoint

- This endpoint processes data from the collision prediction form.
- The server receives the input data, logs it, and ensures all required information is present.
- The data is converted into a format suitable for the prediction models.
- An initial prediction is made to determine if a collision is likely.
- If a collision is predicted, further analysis determines the direction of the collision.
- The results, including the probability of a collision and the direction if applicable, are saved in a CSV file and returned to the web page.

Speed Prediction Endpoint

- This endpoint processes data from the speed prediction form.
- Similar to the collision prediction endpoint, the server receives, logs, and verifies the input data.
- The data is converted and processed for the prediction model.
- A speed prediction is made and returned to the web page.
- The results are saved in a CSV file for future reference.

Running the Server

- The server is set to run in debug mode, which allows for easy troubleshooting and development.

The webpage allows users to choose between predicting collisions or speed. They enter relevant details into a form, which sends the data to a Flask server. The server processes the data, uses pre-trained models to make predictions, and returns the results to be displayed on the webpage. The predictions are also saved in CSV files for future analysis.

Results:

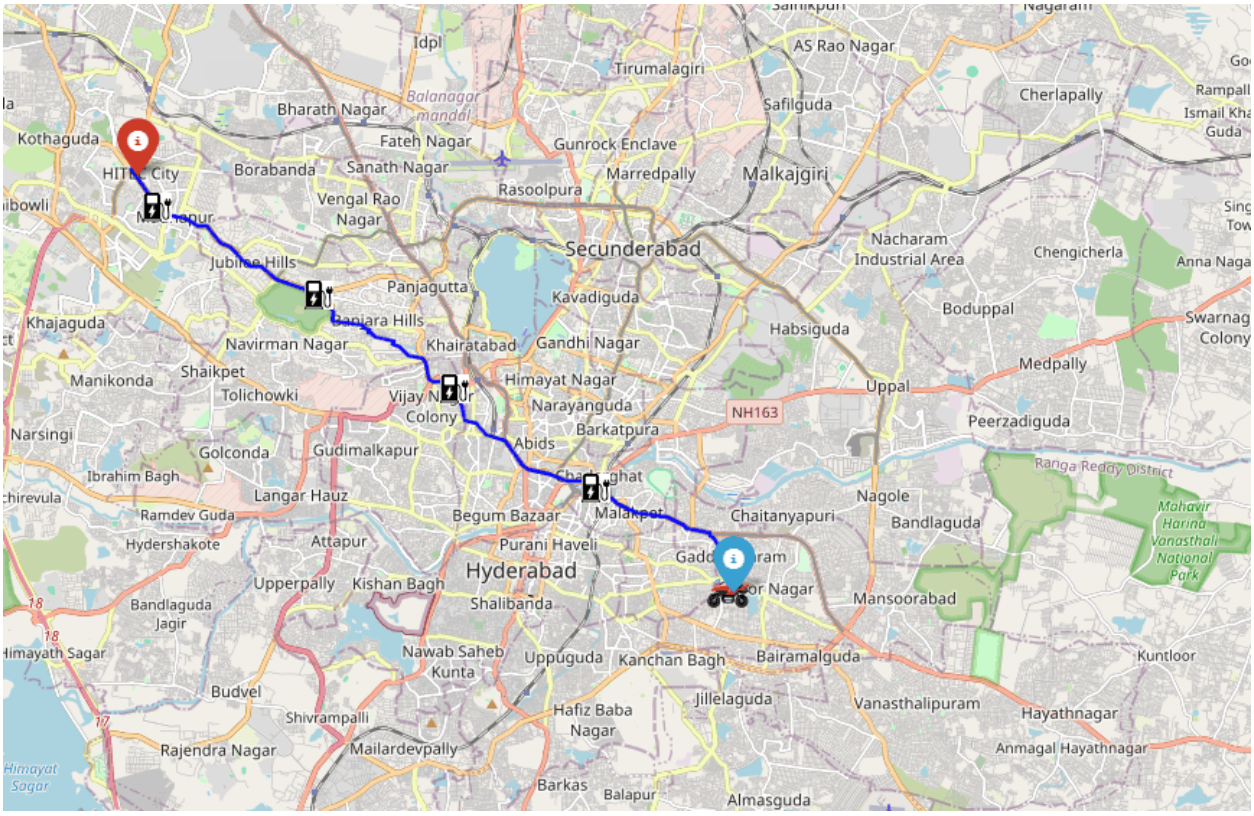


Fig 10 - Shortest route with battery stations displaying along the path

In this interface, the path shows the shortest path between two places namely Hitech city and Saroor Nagar, but whenever the distance between the source and destination places is large shortest path is not displayed properly and this interface is not providing any information about time to travel and driving directions. So in order to address all these issues, slightly enhanced the interface as shown below.

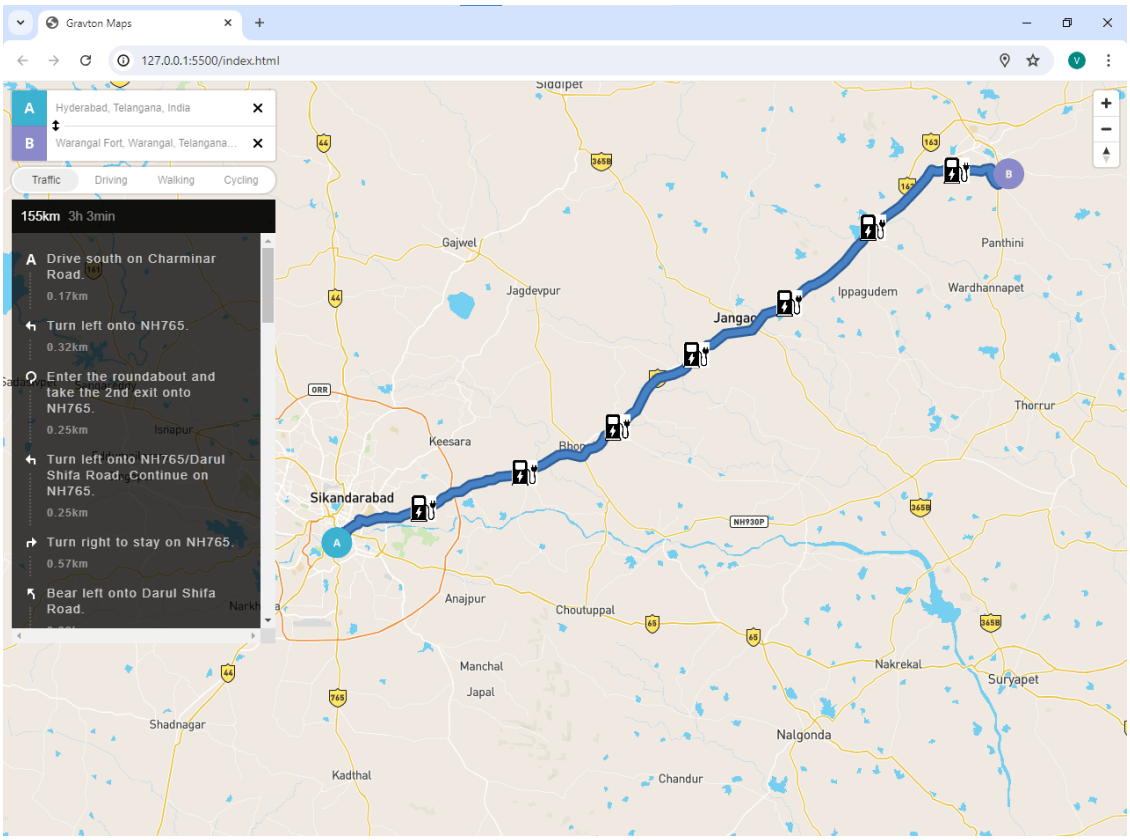


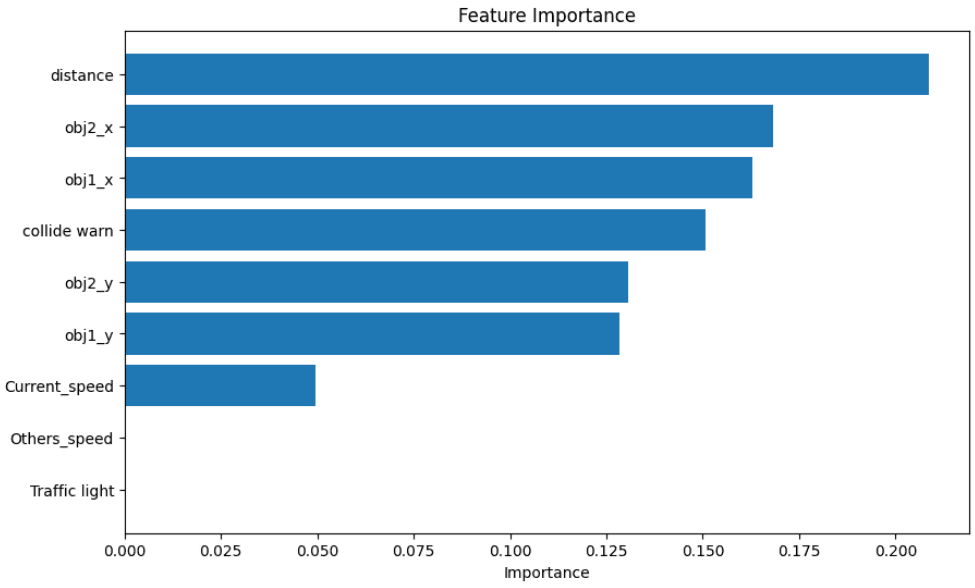
Fig 11 - Shortest route with battery stations displaying along the path, along with driving instructions.

As mentioned earlier, this interface is enhanced version of the previous one, here the shortest path is displayed easily no matter how much the distance between source and destination. In the figure, source is Hyderabad, India and destination is warangal fort, in the interface it is showing the distance as 155km and approximate time to travel is 3h 3min, which varies from the type of vehicle used for driving, it also shows the directions to travel and battery stations are located at every 5km along the path.

obj1_cord	obj1	obj2_cord	obj2	Traffic light	distance	Others_speed	Battery SOC	Nearest Battery distance	collide warn	Collision Direction	Current_speed
(1090, 1101)	truck	(3082, 1384)	car	yellow	0.532342191	0	99.99722222	15.39713236			5
(1090, 1101)	truck	(1527, 1157)	car		0.116568388	0	99.99444444	15.29144075			10
(1090, 1101)	truck	(1553, 1068)	car		0.122812831	0	99.99166667	15.24029661			15
(3082, 1384)	car	(1090, 1101)	truck		0.532342191	0	99.98888889	15.2217284			20
(3082, 1384)	car	(1527, 1157)	car		0.415787752	18.85	99.98611111	15.18473722			25
(3082, 1384)	car	(1553, 1068)	car		0.413097237	32.09	99.98333333	15.14615898			30
(1527, 1157)	car	(1090, 1101)	truck		0.116568388	11.92	99.98055556	15.07098162			35
(1527, 1157)	car	(3082, 1384)	car		0.415787752	11.92	99.97777778	15.04602584			40
(1527, 1157)	car	(1553, 1068)	car		0.024532166	18.85	99.975	15.01452508	Collision Warning!	northwest	35
(1553, 1068)	car	(1090, 1101)	truck		0.122812831	32.09	99.97222222	15.01606178			40
(1553, 1068)	car	(3082, 1384)	car		0.413097237	11.92	99.96944444	14.99636178			45
(1553, 1068)	car	(1527, 1157)	car		0.024532166	11.92	99.96666667	14.98427157	Collision Warning!	southeast	40
(1085, 1105)	truck	(3063, 1371)	car		0.528056832	18.85	99.96388889	14.94023009			45
(1085, 1105)	truck	(1528, 1164)	car		0.118245352	32.09	99.96111111	14.90759571			50
(1085, 1105)	truck	(1554, 1075)	car		0.124343173	11.92	99.95833333	14.84790719			55
(3063, 1371)	car	(1085, 1105)	truck		0.528056832	11.92	99.95555556	14.82314524			60
(3063, 1371)	car	(1528, 1164)	car		0.409811603	18.85	99.95277778	14.81365729			60
(3063, 1371)	car	(1554, 1075)	car		0.406864857	32.09	99.95	14.76528419			60
(1528, 1164)	car	(1085, 1105)	truck		0.118245352	11.92	99.94722222	14.74884159			60
(1528, 1164)	car	(3063, 1371)	car		0.409811603	11.92	99.94444444	14.72240898			60
(1528, 1164)	car	(1554, 1075)	car	green	0.024532166	18.85	99.94166667	14.71441342	Collision Warning!	northwest	55
(1554, 1075)	car	(1085, 1105)	truck		0.124343173	32.09	99.93888889	14.69721273			60
(1554, 1075)	car	(3063, 1371)	car		0.406864857	11.92	99.93611111	14.6754627			60
(1554, 1075)	car	(1528, 1164)	car		0.024532166	11.92	99.93333333	14.65229774	Collision Warning!	southeast	55
(1081, 1112)	truck	(3043, 1358)	car		0.523176931	18.85	99.93055556	14.64535083			60
(1081, 1112)	truck	(1529, 1171)	car		0.119556818	32.09	99.92777778	14.6421607			60
(1081, 1112)	truck	(1551, 1077)	car		0.124698477	11.92	99.925	14.63601968			60
(1081, 1112)	truck	(1548, 1057)	car		0.124414371	11.92	99.92222222	14.6331622			60
(1081, 1112)	truck	(1545, 668)	truck		0.169917694	18.85	99.91944444	14.63121973			60
(3043, 1358)	car	(1081, 1112)	truck		0.523176931	32.09	99.91666667	14.62604066			60
(3043, 1358)	car	(1529, 1171)	car		0.403623103	11.92	99.91388889	14.62151197			60
(3043, 1358)	car	(1551, 1077)	car		0.401698536	11.92	99.91111111	14.61324928			60
(3043, 1358)	car	(1548, 1057)	car		0.403489619	18.85	99.90833333	14.63510903			60
(3043, 1358)	car	(1545, 668)	truck		0.436370299	32.09	99.90555556	14.6955368			60
(1529, 1171)	car	(1081, 1112)	truck		0.119556818	11.92	99.90277778	14.73619026			60

Fig 7 - A sample of the dataset that is being prepared is shown

Data Analysis:



Feature Importance Analysis for Collision Direction

In this analysis, the Random Forest Classifier is used to predict the target variable `collision_direction`. The bar plot provided above illustrates the importance of each feature in determining the direction of a potential collision. Here's a detailed breakdown of each feature's significance:

Overall Analysis

- **Distance and Object Positions**: The most influential features are those related to the distances and positions of surrounding objects, indicating their critical role in determining the direction from which a collision is likely to occur.
- **Collision Warning**: The presence of a collision warning is a significant predictor of the collision direction, highlighting its importance in the model.

Current Speed and Traffic Conditions: While these factors are relevant, their impact on predicting the specific direction of a collision is relatively minor compared to spatial features.

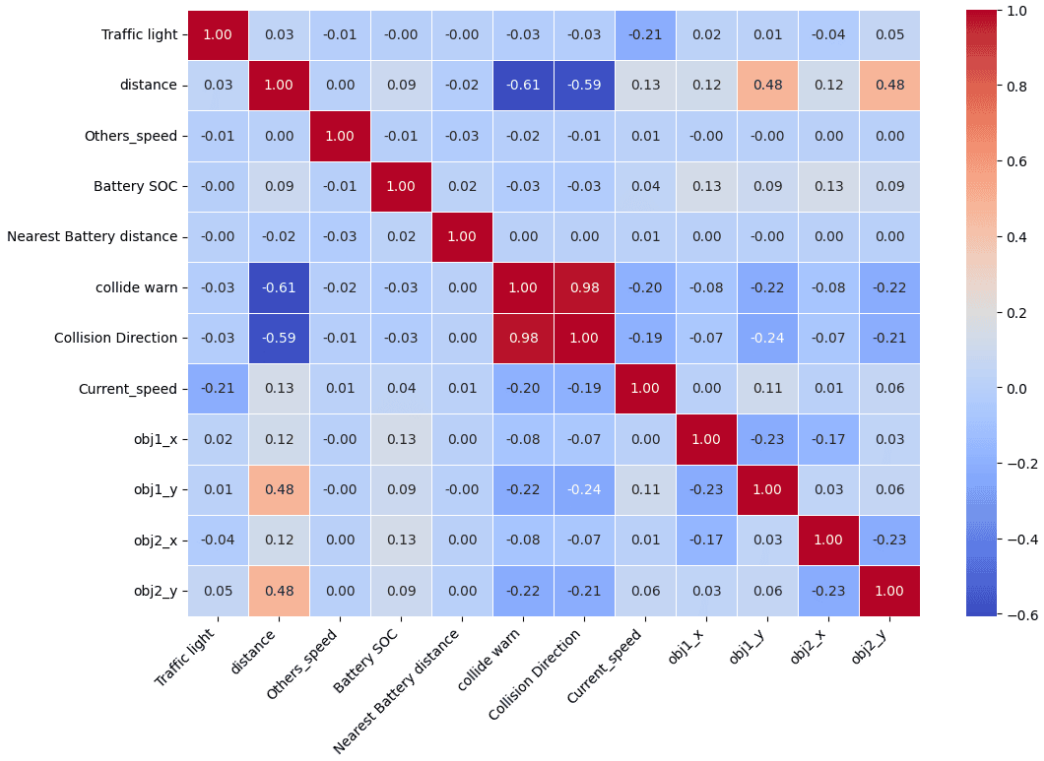


Figure depicting co-relation between multiple columns of the dataset (obj.csv)

i) Metrics for Collision warn:

In order to predict the collision warnings, Logistic regression has been used which gave the accuracy of `Accuracy:0.9948148148148148` upon testing it for dataset 1.

DATASET 1 (obj.csv):

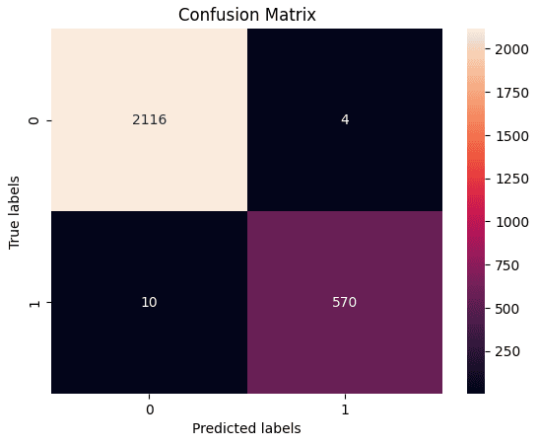


Fig 11 - Confusion matrix representing the number of TP, TN, FP, FN for first dataset

TP - These are the cases where the model predicted the positive class correctly, and the actual class is also positive. (570)

TN - These are the cases where the model predicted the negative class correctly, and the actual class is also negative. (2116)

FP - These are the cases where the model predicted the positive class incorrectly (said it's positive), but the actual class is negative. (4)

FN - These are the cases where the model predicted the negative class incorrectly (said it's negative), but the actual class is positive. (10)

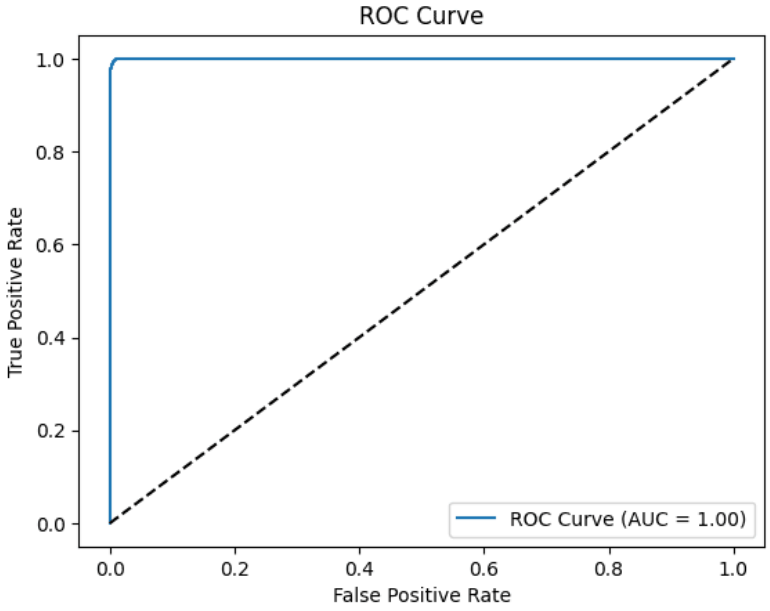


Fig 12 - Figure representing the ROC Curve of the Logistic Regression model for first dataset

The Receiver Operating Characteristic (ROC) curve is a graphical plot that illustrates the diagnostic ability of a binary classification model across different discrimination thresholds. The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings.

Here's what TPR and FPR represent:

- **True Positive Rate (TPR)**: Also known as Sensitivity or Recall, it is the ratio of correctly predicted positive instances to all actual positive instances.

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

TPR tells you how good the model is at predicting the positive class when the actual outcome is positive.

- **False Positive Rate (FPR)**: It is the ratio of incorrectly predicted negative instances to all actual negative instances.

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

FPR tells you how often the model predicts the positive class when the actual outcome is negative.

In the ROC curve:

- The x-axis represents the False Positive Rate (FPR).
- The y-axis represents the True Positive Rate (TPR).

```
Precision: 0.9930313588850174
Recall: 0.9827586206896551
F1 Score: 0.9878682842287695
Accuracy: 0.9948148148148148
```

DATASET 2 (new_data.csv):

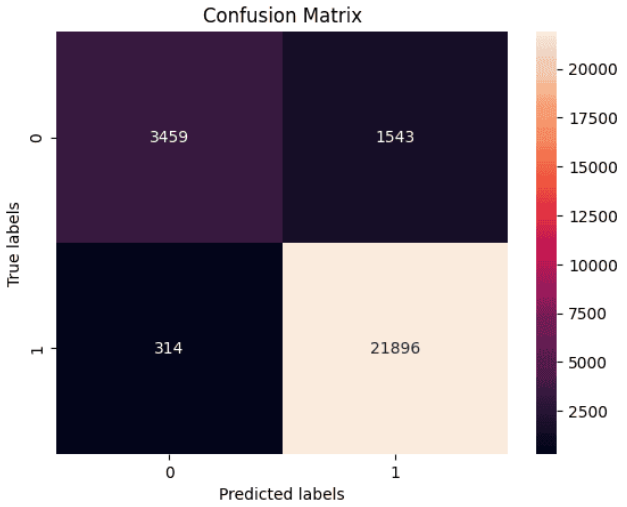


Fig 13 - Confusion matrix representing the number of TP, TN, FP, FN for second dataset

In this case, the model is trained on 80% of first dataset and it is tested against 20% of first dataset and 100% of second dataset.

TP - These are the cases where the model predicted the positive class correctly, and the actual class is also positive. (21896)

TN - These are the cases where the model predicted the negative class correctly, and the actual class is also negative. (3459)

FP - These are the cases where the model predicted the positive class incorrectly (said it's positive), but the actual class is negative. (1543)

FN - These are the cases where the model predicted the negative class incorrectly (said

it's negative), but the actual class is positive. (314)

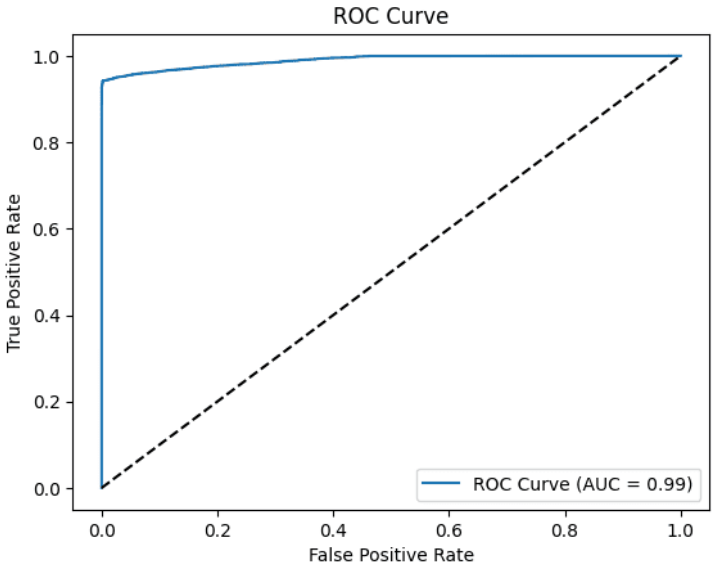


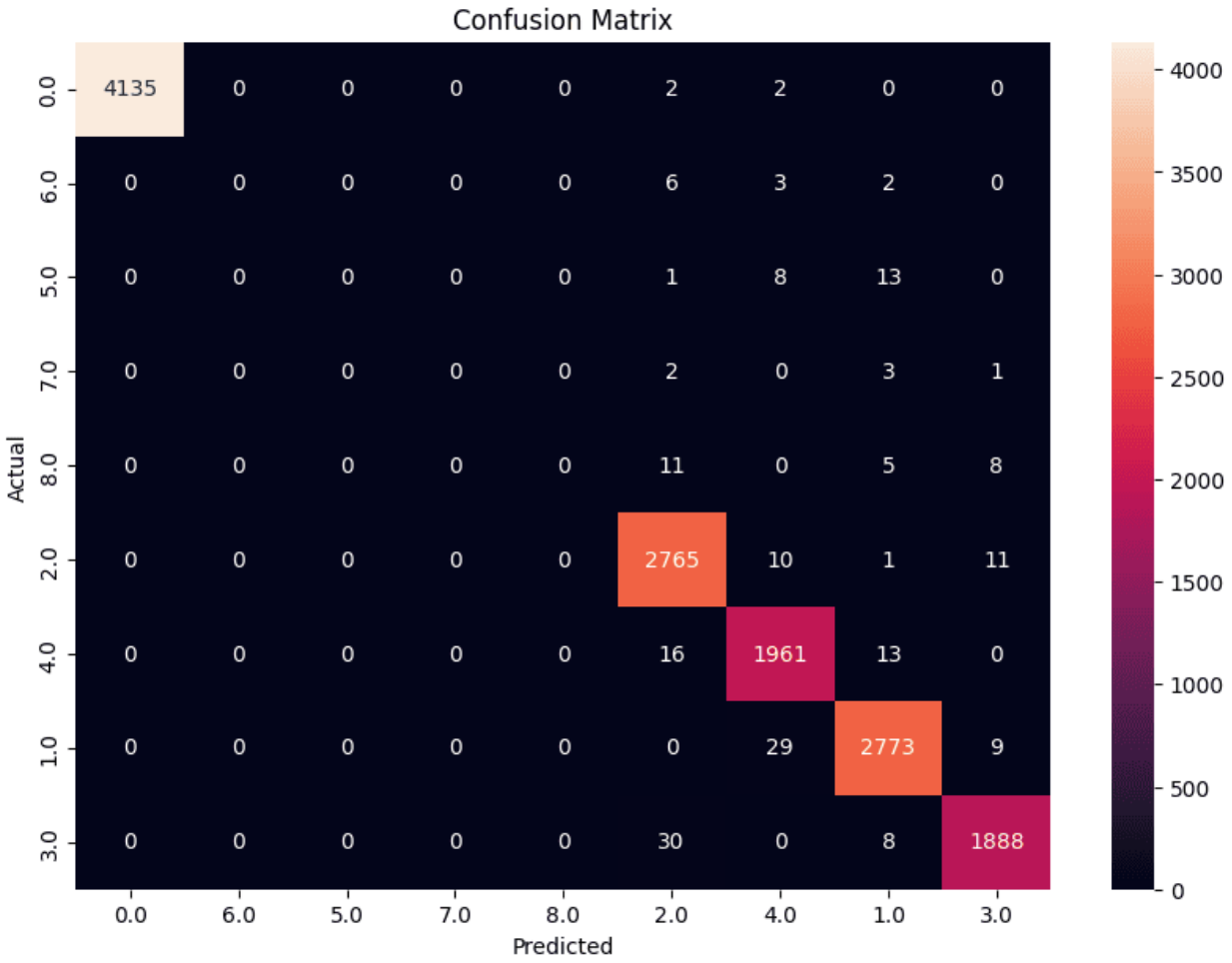
Fig 14 - Figure representing the ROC Curve of the Logistic Regression model for second dataset

- **High AUC (0.99)**:
  - An AUC of 0.99 indicates excellent performance of the classifier.
  - It means the model has a high true positive rate and a low false positive rate.
  - The closer the curve is to the top-left corner, the better the model's performance.
- **Shape of the ROC Curve**:
  - The curve starts at (0,0), which means at threshold 1, the model classifies all instances as negative (no false positives and no true positives).
  - As the threshold decreases, the true positive rate increases as more positive instances are correctly classified.
  - Ideally, the curve should rise quickly towards the top-left corner, indicating a high true positive rate with a low false positive rate.

ii) Metrics for Collision Direction:

Dataset (new_data.csv):

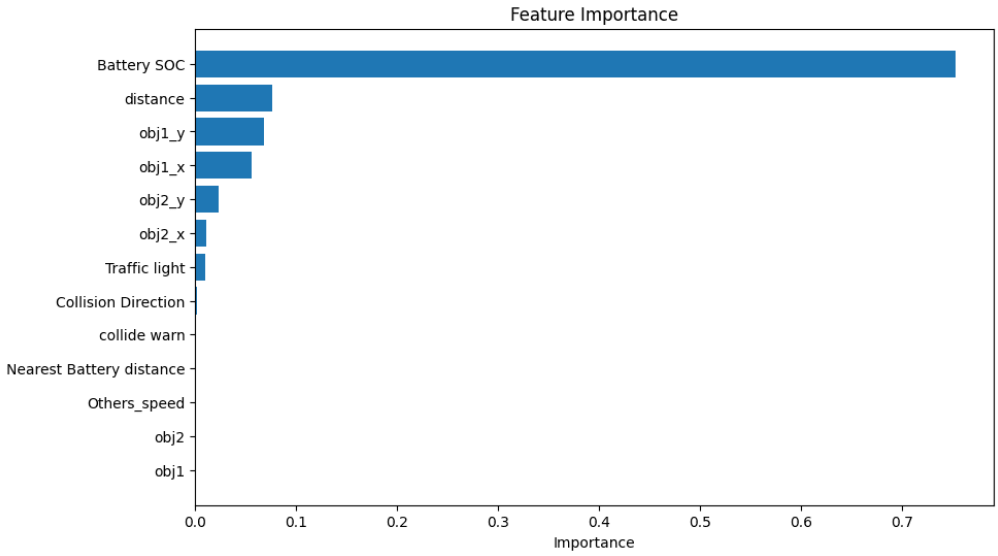
```
Training accuracy: 0.9905034450074733
Test accuracy: 0.9858559346748323
```



iii) Metrics for Speed Prediction

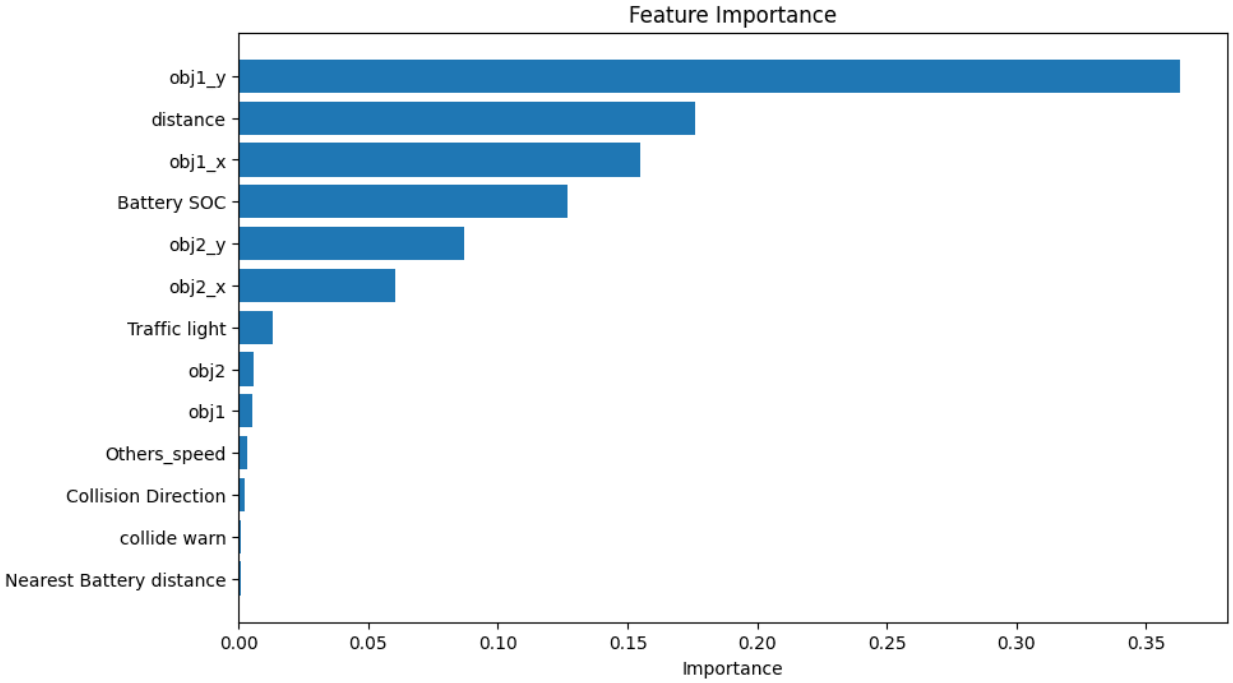
Dataset 1(objects.csv)

Mean Squared Error: 35.45745454712475
R^2 Score: 0.9109505409809976



Dataset 2 (new_data.csv)

Mean Squared Error: 75.215185833944
R^2 Score: 0.70161049124892



Warning System

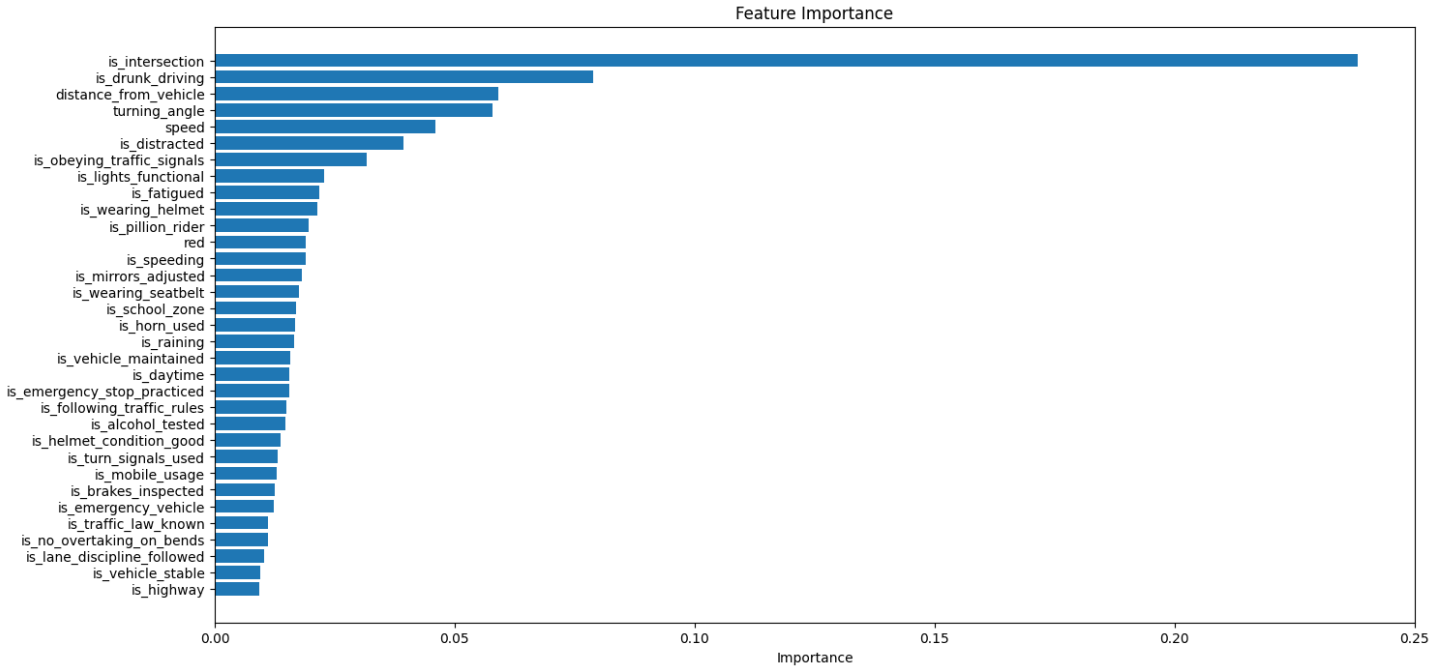
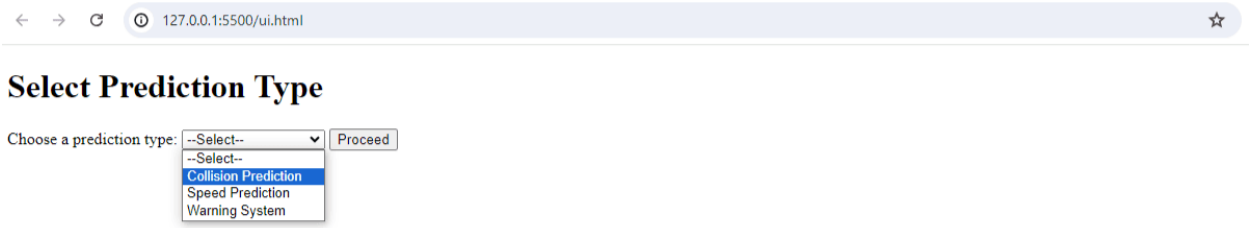
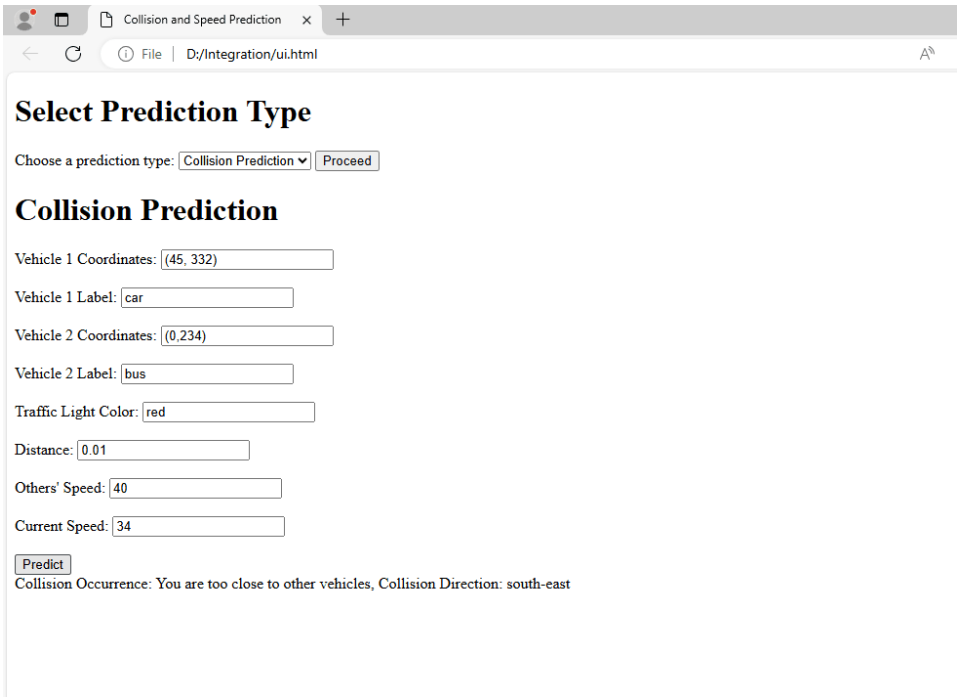


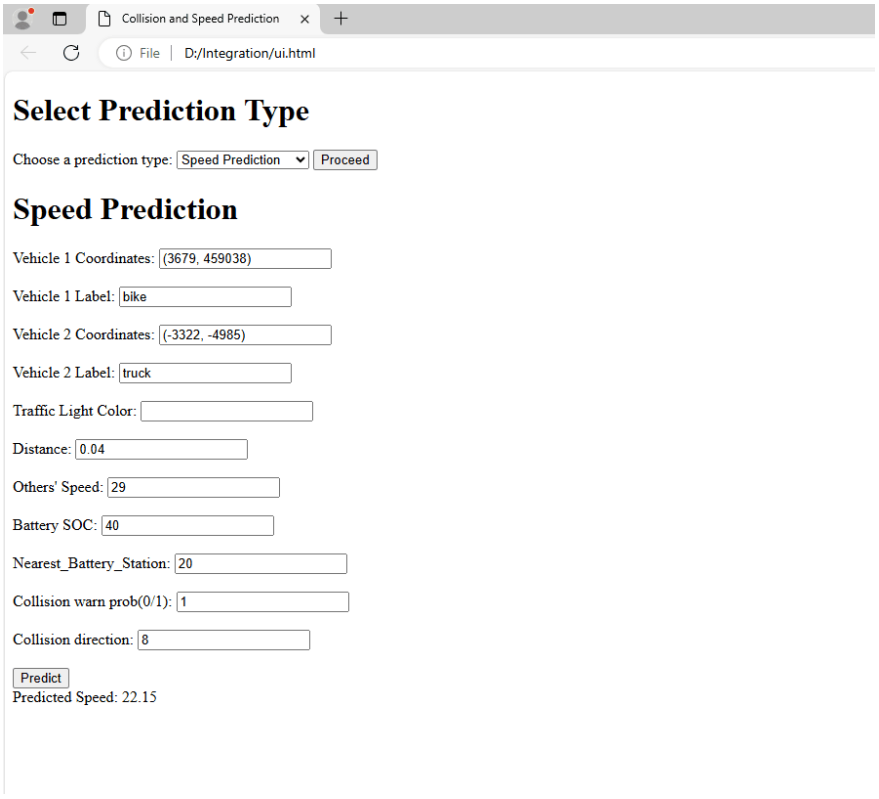
Figure representing the dependencies of the output warning system.



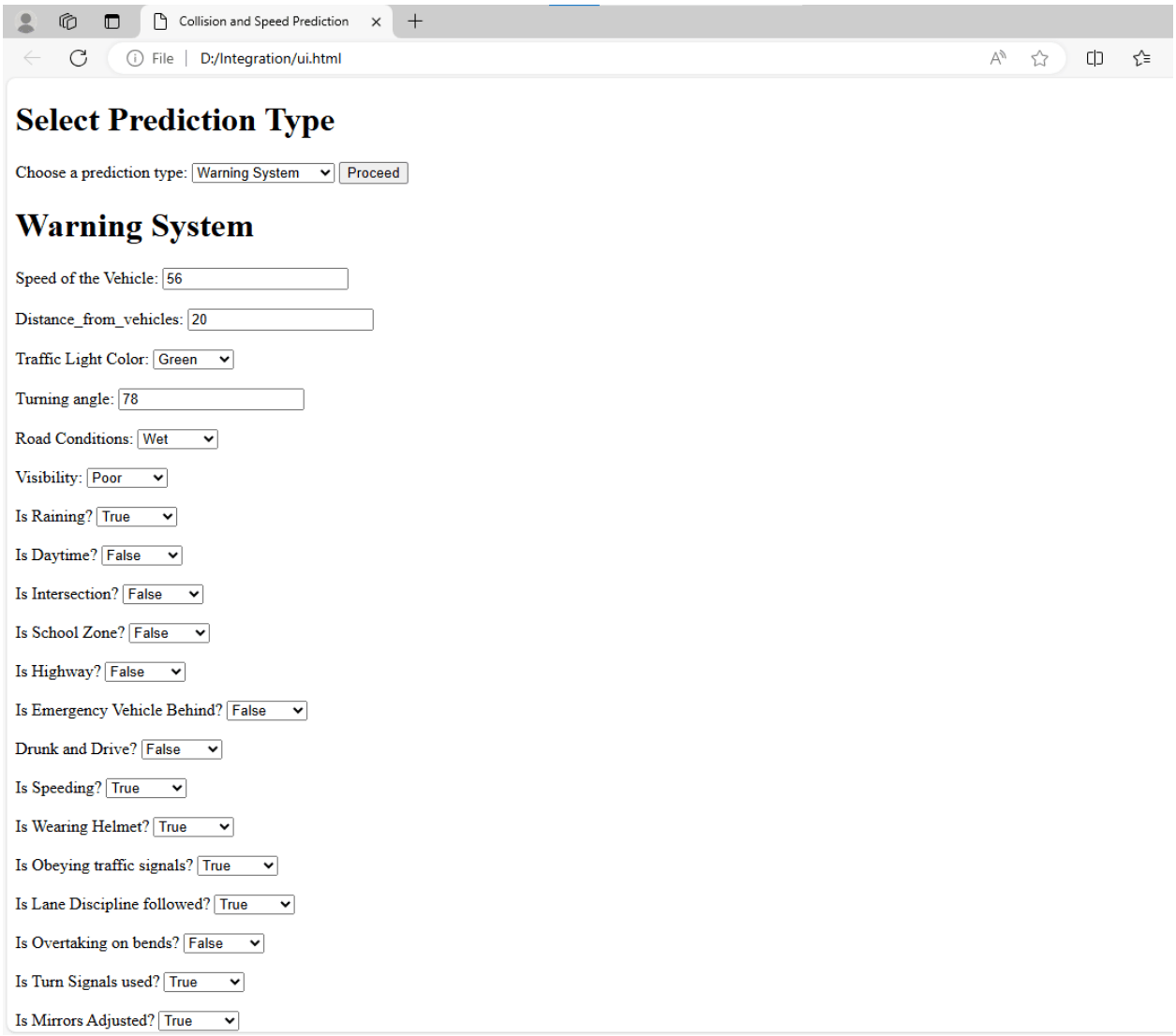
Interface representing the selection of 3 prediction types



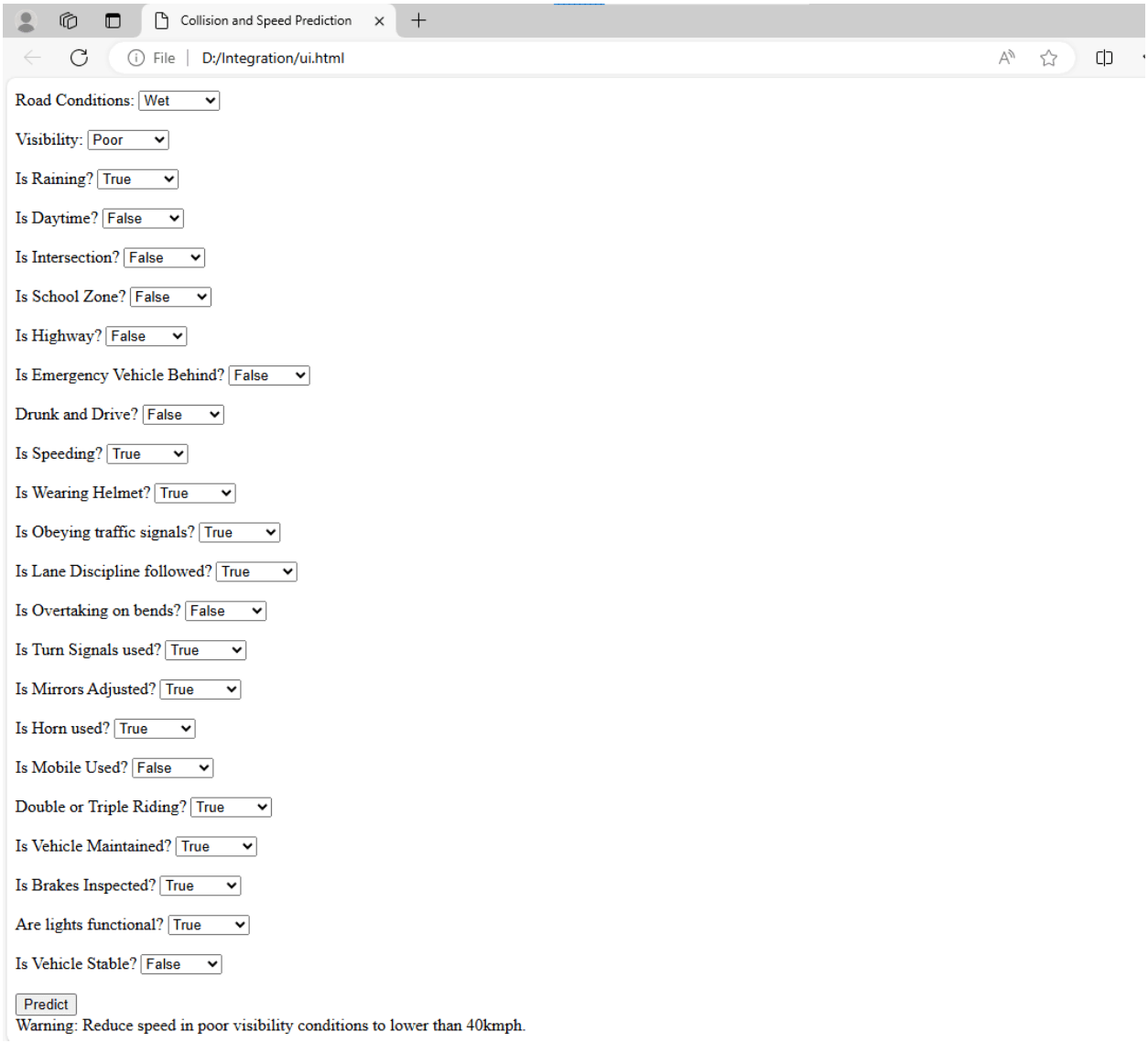
Interface for Collision Prediction



Interface for Speed Prediction



Interface for Warning System



Interface for Warning system

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	obj1	obj2	Traffic ligh	distance	Others_sp	Current_s	obj1_x	obj1_y	obj2_x	obj2_y	collision_	coll_accur	collision_	col_dir_acc
2	car	truck	3	0.01	30	50	3	4	2	3	1	0.979771	7	0.484905
3	car	truck	3	0.01	30	50	893	808	2	3	1	0.979771	8	0.451577
4	car	truck	3	0.01	30	50	893	808	1527	1157	1	0.979771	7	0.356687
5	car	truck	3	0.09	30	50	893	808	1527	1157	0	0.472272	0	0
6	car	truck	3	0.05	30	50	893	808	1527	1157	1	0.868138	6	0.400638
7	car	truck	3	0.05	57	26	893	808	1527	1157	1	0.918851	6	0.414299
8	car	truck	3	0.05	57	26	1553	1068	-2	-3	1	0.918851	8	0.847305
9	bike	truck	3	0.07	100	48	10000	34425	34425	10000	1	0.705006	5	0.43751
10	bike	truck	1	0.07	100	48	10000	34425	34425	10000	1	0.75659	5	0.43862
11	bike	car	1	0.07	100	48	3	4	34425	10000	1	0.75659	6	0.762887
12	bike	car	1	0.08	100	48	3	4	34425	10000	1	0.653661	6	0.754698
13	bike	car	1	0.09	100	48	3	4	34425	10000	1	0.534014	6	0.763605
14	bike	car	1	0.1	100	48	3	4	34425	10000	0	0.410321	0	0
15	bike	car	2	0.5	100	60	1553	1068	34425	10000	0	9.90E-10	0	0
16	car	bike	3	0	79	75	1553	1068	-2	-3	1	0.976914	8	0.676669
17	truck	bike	3	0.05	36	24	-3	-4	1527	1157	1	0.92364	6	0.755469
18	truck	bike	3	0	18	29	-3	-4	930	733	1	0.992501	6	0.58968
19	bicycle	truck	1	0.09	57	34	90	50	1527	1157	1	0.623687	6	0.749892
20	truck	bike	3	0.05	45	23	90	50	39	5000	1	0.924728	5	0.740096
21	bicycle	bike	1	0	57	54	90	50	1527	1157	1	0.989264	6	0.632002
22	car	bike	1	5	34	12	-54	-7348	930	733	0	1.17E-106	0	0
23	bicycle	bike	1	1	32	16	1553	1068	39	5000	0	4.97E-20	0	0
24	bike	truck	3	0.07	78	34	-54	-7348	39	5000	1	0.772308	5	0.835088
25	bike	truck	3	0	8	65	90	50	39	5000	1	0.982788	5	0.721628
26	bicycle	truck	3	0.08	21	43	90	50	1527	1157	1	0.63672	6	0.747992
27	bike	truck	3	0.02	23	13	1553	1068	39	5000	1	0.986086	5	0.415099
28	bike	bike	3	0.02	23	34	-54	-7348	2	3	1	0.977362	7	0.485192
29	car	truck	1	0.01	78	54	1553	1068	930	733	1	0.982113	8	0.479866
30	bike	truck	3	0.06	53	45	1453	1068	1527	1107	1	0.815038	5	0.433862
31	bicycle	bike	0	7	65	3	90	50	1527	1157	0	7.43E-150	0	0
32	bicycle	truck	3	0.01	-0.01	0.01	-54	-7348	-2	-3	1	0.993861	5	0.462797
33	bike	car	2	0.06	12	56	-54	-7348	1527	1157	1	0.800852	6	0.742472
34	bicycle	truck	1	0.07	56	89	90	50	930	733	1	0.551281	6	0.457202
35	bicycle	bike	3	0.02	90	78	0	-7348	39	0	1	0.935032	7	0.502419
36	car	bus	3	0.01	40	34	45	332	0	234	1	0.985921	7	0.476016

Testing data that has been collected from the interface

Object detection using YOLOV8

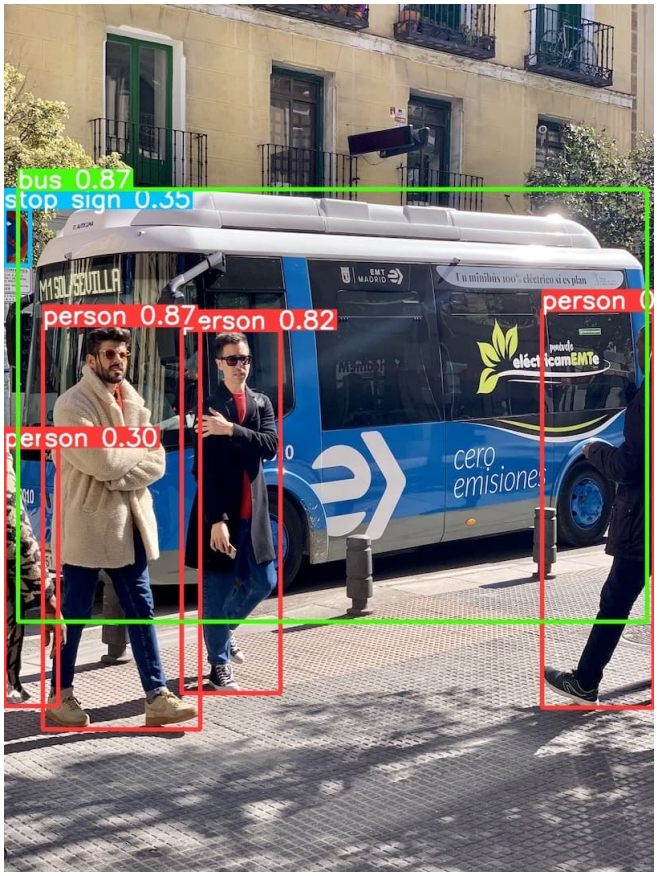


Fig 13 - Object detection with accuracy using yolov8

```
The below objects are closer to your bus
person at distance 0.07999521948883079
person at distance 0.050412666355612076
```

Conclusion:

Logistic regression model can be integrated onto NVIDIA chips, particularly using NVIDIA's GPU technology. Custom implementations of logistic regression can be written using CUDA for direct control over GPU resources. NVIDIA GPUs are well-suited for high-performance computing tasks due to their parallel processing capabilities. This is particularly beneficial for stacking models, as the process often involves training multiple models independently before combining their outputs. GPUs can handle the extensive computational load required for training complex models such as deep neural networks, gradient boosting machines, or other machine learning algorithms more

efficiently than traditional CPUs. In conclusion, stacking models on NVIDIA GPUs leverages the parallel processing power and optimized libraries available for these chips, leading to significant gains in training efficiency and predictive performance. By carefully managing resources and employing advanced optimization techniques, developers can harness the full potential of ensemble learning on high-performance GPU hardware.

References:

1. https://www.leewayhertz.com/multimodal-model/
2. https://github.com/opendrivelab/end-to-end-autonomous-driving?tab=readme-ov-file
3. https://www.researchgate.net/publication/355123219_Artificial_Intelligence_AI_Framework_for_Multi-Modal_Learning_and_Decision_Making_towards_Autonomous_and_Electric_Vehicles
4. https://medium.com/divya-krishnan-design/ui-ux-for-autonomous-vehicle-interface-to-build-trust-de7f4c545c3b
5. https://datasetninja.com/bdd100k
6. https://damon.com/blog/motorcycle-ai-create-safest-electric-motorcycle/
7. https://ebikelabs.com/ebikelabs-ai-for-e-bikes/
8. https://link.springer.com/article/10.1007/s11042-021-11437-3
9. https://www.visteon.com/machine-learning-algorithms-in-autonomous-cars/
10. https://neptune.ai/blog/self-driving-cars-with-convolutional-neural-networks-cnn
11. https://www.instructables.com/How-You-Can-Make-a-Fully-Autonomous-Self-

Driving-V/

12. https://medium.com/analytics-vidhya/evaluating-a-random-forest-model-9d165595ad56