University of Hertfordshire UH

School of Physics, Engineering and Computer Science

MSc Data Science Project
7PAM2002-0509-2024

Department of Physics, Astronomy and Mathematics

Data Science FINAL PROJECT REPORT

Project Title:

Machine Learning Prediction of Road Accident Severity with Urban factors and Weather conditions

Student Name and SRN:

AISHWARYA SHEKAR BABU

23093811

Supervisor: Krishna Kedia

Date Submitted: 28/08/2025

GitHub link: https://github.com/aishwarya-shekar-babu/Road-Accident-Prediction

Word Count: 4777

DECLARATION STATEMENT

This report is submitted in partial fulfilment of the requirement for the degree of Master of Science **in Data Science** at the University of Hertfordshire.

I have read the detailed guidance to students on academic integrity, misconduct and plagiarism information at Assessment Offences and Academic Misconduct and understand the University process of dealing with suspected cases of academic misconduct and the possible penalties, which could include failing the project or course.

I certify that the work submitted is my own and that any material derived or quoted from published or unpublished work of other persons has been duly acknowledged. (Ref. UPR AS/C/6.1, section 7 and UPR AS/C/5, section 3.6)

I did not use human participants in my MSc Project.

I hereby give permission for the report to be made available on module websites provided the source is acknowledged.

Student Name printed: AISHWARYA SHEKAR BABU

Student Name signature: Aishwarya Shekar Babu

Student SRN number: 23093811

UNIVERSITY OF HERTFORDSHIRE

SCHOOL OF PHYSICS, ENGINEERING AND COMPUTER SCIENCE

Abstract

This project investigates the use of predictive modeling to assess the severity of road accidents in the UK, incorporating environmental, temporal, spatial, and vehicle-related variables. Using the STATS19 dataset, accident and vehicle records were combined and processed through a pipeline including data imputation, categorical encoding, and feature engineering. To address the significant class imbalance, where minor accidents far exceed severe or fatal ones, an under-sampling approach was applied to improve model sensitivity to high-severity incidents. Four classifiers—Logistic Regression, Multilayer Perceptron, Random Forest, and XGBoost—were trained and optimized. Among them, XGBoost demonstrated the best results, with a recall of 0.78 and an ROC AUC of 0.71. The evaluation emphasized recall to reduce the risk of missing severe or fatal accidents. Feature importance analysis highlighted rural road types, carriageway layout, speed limits, lighting conditions, and adverse weather as key factors influencing accident outcomes. These findings illustrate the value of ensemble methods for enhancing road safety strategies, informing policy decisions, emergency response planning, and infrastructure risk management.

Table of Contents

1. Introduction 5

1.1 Research Aim 5

1.2 Objectives 5

1.3 Ethical considerations 6

2. Literature Review 6

2.1 Handling Imbalanced Data 6

2.3 Review of Methods for Traffic Crash Severity Prediction 7

2.4 Beyond Accuracy as an Evaluation Metric 7

3 Methodology 7

3.1 Brief Overview 7

3.2 Dataset Used 8

3.3 Review of Key Columns and Data Understanding 8

3.4 Data Pre-Processing: 8

3.5 Feature Engineering 9

3.6 Feature encoding and Class Imbalance Handling 9

3.7 Logistic Regression model 11

3.8 Multi-layer perceptron 12

3.9 Random forest 13

3.10 XGBOOST 13

4 Results 15

4.1 Evaluation metrics 15

4.2 Model results analysis 15

4.3 Feature importance and output 18

4.3 Summary 21

5 Conclusion and Future Work 22

1. Introduction

In the UK and worldwide, traffic accidents remain a major safety issue, resulting in many injuries and deaths each year. While some crashes are minor, serious and fatal ones place heavy demands on hospitals, emergency services, and public authorities. As a result, predicting accident severity is vital for efficient resource allocation, preparedness, and prevention.

The UK Department for Transport collects detailed crash data through the STATS19 database, which records factors such as road conditions, lighting, time of day, vehicle types, driver details, and casualty information. This comprehensive dataset is well-suited for machine learning methods aimed at understanding the variables that influence accident severity. In this study, accidents are classified as either Slight or Severe/Fatal. Since fatal cases are relatively rare, the original three categories (Slight, Serious, Fatal) are combined into two, and undersampling is used to reduce the dominance of slight cases.

For modelling, several approaches are employed, including Logistic Regression, Multilayer Perceptron, Random Forest, and XGBoost. Their effectiveness is measured using precision, recall, F1-score, and ROC-AUC, with recall emphasized to ensure that severe cases are not overlooked. Feature importance is also analyzed to highlight the most influential factors, providing valuable insights for policymakers and safety organizations in designing interventions and improving emergency response.

1.1 Research Aim

To determine the optimal machine learning approach for accurately predicting the severity of road accidents in the UK using environmental (e.g., weather), temporal (e.g., time of day, season), vehicle-related, and spatial (e.g., urban/rural) data, and to identify the key features that contribute most to severe or fatal outcomes.

1.2 Objectives

- Preprocess the STATS19 dataset by combining accident and vehicle records, handling missing values, creating relevant features, and addressing class imbalance.
- Apply multiple machine learning approaches—such as Logistic Regression, Multilayer Perceptron, Random Forest, and XGBoost—to predict accident severity.
- Evaluate these models using performance measures including recall, precision, F1-score, and ROC AUC, with priority given to recall to better capture severe and fatal cases.
- Determine the best-performing model and analyse feature importance to identify the key factors influencing serious or fatal accidents.

1.3 Ethical considerations

This project uses the publicly available UK STATS19 road accident dataset, which is fully anonymized with no personal identifiers, and geographic data reported at a coarse resolution to ensure GDPR compliance. The dataset is released under the Open Government Licence v3.0, permitting academic and research use with proper attribution. The original data was collected by UK police authorities under standardized lawful procedures, ensuring it is ethically sourced, legally compliant, and appropriate for academic analysis.

2. Literature Review

Accurately predicting accident severity is essential for improving road safety and enabling rapid emergency response. This study uses MLP, Logistic Regression, Random Forest, and XGBoost to classify accident outcomes while addressing challenges such as class imbalance and feature selection. Moreover, feature importance is examined to identify the key factors influencing crash severity, based on the UK STATS19 dataset (2005–2017).

2.1 Handling Imbalanced Data

In many classification problems, one class appears far more often than the other, creating an imbalance that makes it harder for traditional models to learn effectively. As noted by He and Garcia (2009), standard classifiers often become biased towards predicting the majority outcome, which results in poor recognition of rare but important cases. This imbalance is particularly evident in the STATS19 accident dataset, where minor crashes are recorded much more frequently than serious or fatal ones.
A range of techniques can be applied to address this challenge. One approach is **undersampling**, where a portion of the majority class is removed to make the classes more balanced. Another option is **oversampling**, which increases the number of minority class examples, sometimes by generating synthetic samples such as with the SMOTE algorithm. Each method comes with trade-offs: undersampling can discard valuable information, while oversampling may increase the chance of overfitting. For datasets like STATS19, which are relatively large, undersampling is often a practical choice since enough majority class cases remain available even after reduction.

2.2 Handling missing values and hidden data out of range

Missing data can appear in different forms, and the handling strategy depends on the underlying mechanism. According to Little and Rubin (2014), missing values can be classified as Missing Completely at Random (MCAR), Missing at Random (MAR), or Missing Not at Random (MNAR). These categories help determine whether deletion, imputation, or more advanced techniques are appropriate. A related issue arises when placeholders such as “-1,” “999,” or “Unknown” are used instead of true missing indicators. If left unprocessed, machine learning

algorithms may treat these as valid values, introducing errors. It is therefore necessary to recode such placeholders into nulls and apply appropriate imputation techniques before modeling.A unique but related problem is the presence of hidden missing values or out-of-range codes—such as -1, 999, or placeholders like “Unknown” or “Out of Range.” These values are often system-generated and not flagged as missing by default. If left unconverted, such entries can mislead machine learning models, particularly decision trees, which might treat these values as valid inputs and introduce artificial splits.
The literature recommends explicitly identifying and recoding such placeholders as nulls before applying imputation or exclusion strategies.

2.3 Review of Methods for Traffic Crash Severity Prediction

Logistic regression has long been a common approach for modelling accident severity because of its simplicity and ease of interpretation. It has been used to assess factors such as accident timing, road surface condition, crash category, and vehicle type (Hsu et al., 2020). However, traditional models typically focus on overall accuracy and often fail to effectively identify severe crashes. Recent research has therefore incorporated techniques like regularization, hyperparameter optimization, and imbalance-handling strategies to enhance predictive performance.
In comparison, modern machine learning methods—such as gradient boosting (e.g., XGBoost)—are capable of learning more complex, non-linear patterns. Studies combining advanced algorithms with preprocessing steps, including data cleaning and resampling, have demonstrated improved outcomes (Zhou, 2024). Beyond prediction, these methods also provide insights into feature importance, supporting policymaking and road safety planning.

2.4 Beyond Accuracy as an Evaluation Metric

Accuracy on its own can be misleading when applied to imbalanced datasets, as it may conceal the model’s inability to capture minority cases effectively. To address this, researchers increasingly emphasize the use of additional measures such as recall and precision for accident severity prediction. Ahmed et al. (2023), for example, found that Random Forest was more effective in detecting severe crashes when these metrics were prioritized over overall accuracy. In line with this evidence, the present study places greater weight on recall, since failing to identify severe or fatal accidents poses a higher risk than misclassifying slight ones.

3 Methodology

3.1 Brief Overview

This study was developed and run in Google Colaboratory, utilizing Python as the programming environment. For data processing and visualization, libraries such as NumPy, pandas, and matplotlib were applied, while scikit-learn, XGBoost, and TensorFlow/Keras were used

for machine learning tasks. The dataset, obtained from Kaggle and based on the UK Road Safety STATS19 records, contained accident and vehicle details. From these, relevant features were extracted and grouped into four categories: temporal (e.g., hour, month, season), spatial (urban vs. rural, location), environmental (weather, lighting, surface conditions), and driver/vehicle-related attributes.

3.2 Dataset Used

The dataset used in this analysis is the UK STATS19 collection, specifically the Accidents.csv and Vehicles.csv files covering 2005–2017, published by the UK Department for Transport under OGL v3.0. It was obtained from Kaggle. Accidents.csv contains details about each accident, including severity, date, time, location, weather, road conditions, and vehicle/casualty counts, while Vehicles.csv provides information on vehicle type, driver age and gender, and fuel type. The datasets were merged on the Accident_Index using an inner join to retain only accidents with at least one vehicle record, resulting in a comprehensive dataset that links accident characteristics with vehicle-specific details, enabling analysis of accident patterns, driver demographics, and contributing factors. The dataset link is provided in the references section.(Tsiaras, K., 2020. UK Road Safety Accidents and Vehicles (2005–2017))

3.3 Review of Key Columns and Data Understanding

The dataset integrates accident-level attributes such as severity, date, time, location, weather, casualties, and road or light conditions with vehicle-level details like vehicle type, driver age, gender, engine capacity, manoeuvre, and impact factors. Accident_Index serves as the primary key linking both sources. These key columns capture the circumstances, environment, and human–vehicle characteristics involved in each crash, offering a multidimensional perspective. Understanding these variables is crucial for exploring accident patterns, assessing risk factors, and supporting predictive modelling. The final structure provides a balanced representation of contextual, driver, and vehicle-related information for analysis.

3.4 Data Pre-Processing:

The raw UK STATS19 accident and vehicle datasets required extensive preprocessing to ensure quality and consistency. Columns with over 40% missing values or irrelevant information were dropped, while missing numeric values (e.g., Age of Vehicle, Driver IMD Decile, Engine Capacity) were imputed using the median. Categorical fields such as Vehicle Type and Sex of Driver were standardized by replacing hidden missing entries with "Unknown". Dates and times were parsed into proper formats, geographic coordinates were validated, and default placeholders (e.g., "00:00" for missing time, -1 for speed limits) were applied where necessary. Derived features like the Urban/Rural label were also created for consistency. The final cleaned dataset, comprising 29 key columns, provided a reliable foundation for further exploratory analysis and modeling.

3.5 Feature Engineering

Several new features were engineered to capture temporal, environmental, and contextual aspects of road accidents. Temporal features included Hour, Month, Year, Day_of_Week_Label, Season, and Time_of_Day, derived from accident timestamps. Environmental and road context features were standardized into categories such as Weather_Label, Road_Surface_Label, Road_Type_Label, Light_Label, and Speed_Bin. This step reduced noise from ambiguous values (e.g., "Data missing or out of range") and ensured consistent representations across the dataset.

3.6 Feature encoding and Class Imbalance Handling

All categorical features were converted into numeric form to make them suitable for machine learning models. Ordinal categories (such as season, time of day, and speed bands) were label encoded to preserve their natural order, whereas nominal variables (including weather conditions, lighting, road type, vehicle type, and driver age bands) were one-hot encoded to avoid artificial relationships between categories. Continuous numeric features were left unchanged but standardized when necessary.

The accident severity variable was recoded into two groups: slight versus severe/fatal. This simplification was chosen because fatal cases are relatively rare; combining them with serious accidents provided a more balanced binary outcome. To further address the imbalance, downsampling was applied to the slight accident category, ensuring that severe/fatal cases were sufficiently represented during model training.

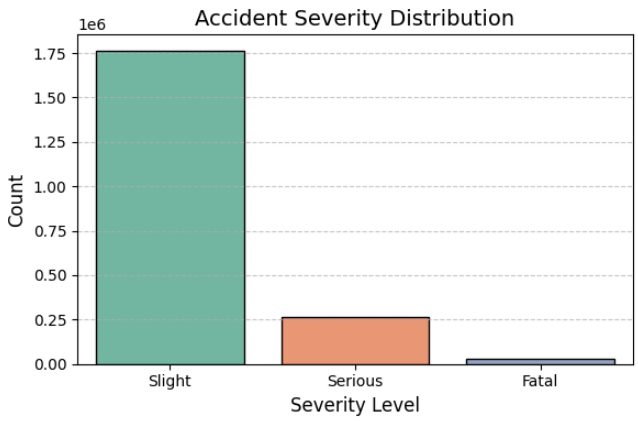


Figure 3.1 Distribution of accident severity with fatal incidents

In the original dataset, slight accidents made up the vast majority of cases, around 86 percent, whereas serious and fatal accidents together accounted for roughly 14 percent. Such an imbalance could lead predictive models to favour the majority class, reducing their ability to accurately detect severe outcomes.

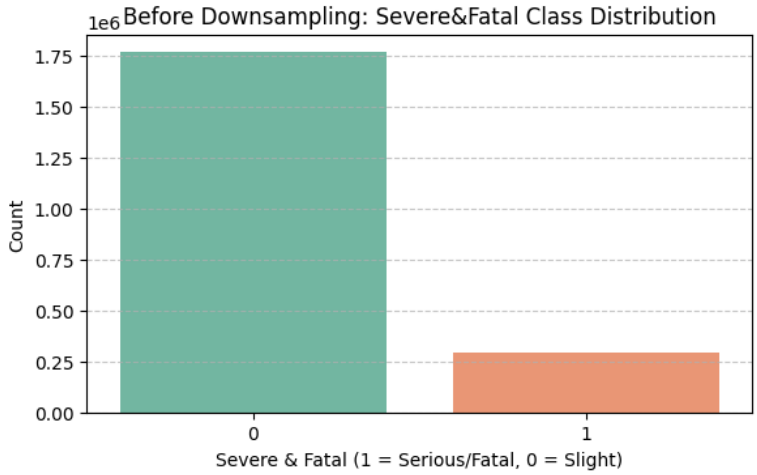


Figure 3.2 Distribution of accident severity before undersampling

To address this issue, the dataset was adjusted by downsampling the slight accident cases to approximately twice the number of severe/fatal accidents. Following this adjustment, slight accidents represented about 67 percent of the dataset, while severe and fatal accidents made up roughly 33 percent. This strategy retains all severe and fatal cases while reducing the dominance of slight accidents, creating a more balanced dataset that improves the model's ability to learn patterns associated with higher severity accidents.

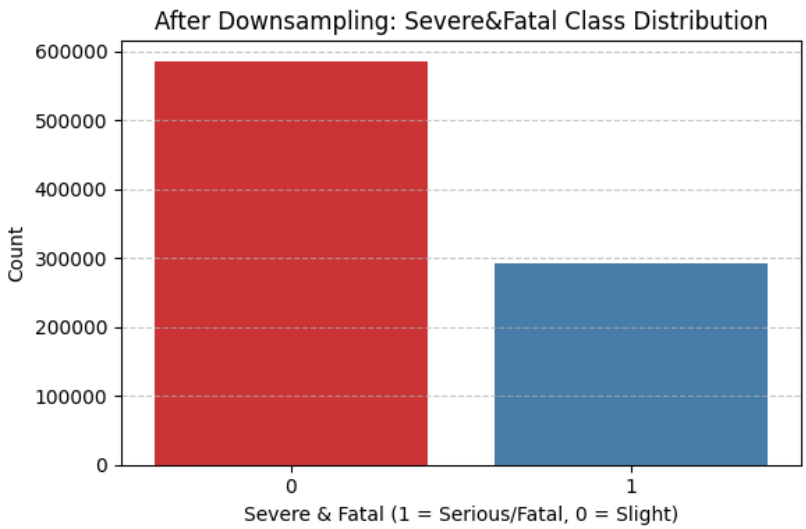


Figure 3.3 Distribution of accident severity after undersampling

Although **oversampling** and **SMOTE** were considered, they were not applied because oversampling risks overfitting by repeatedly exposing the model to identical severe/fatal cases, while SMOTE could generate unrealistic synthetic records due to the small and heterogeneous nature of high-severity accidents. Instead, undersampling was chosen as it reduces the dominance of slight accidents while preserving all real severe/fatal cases, ensuring the model learns from genuine patterns without bias from artificial data.

3.7 Logistic Regression model

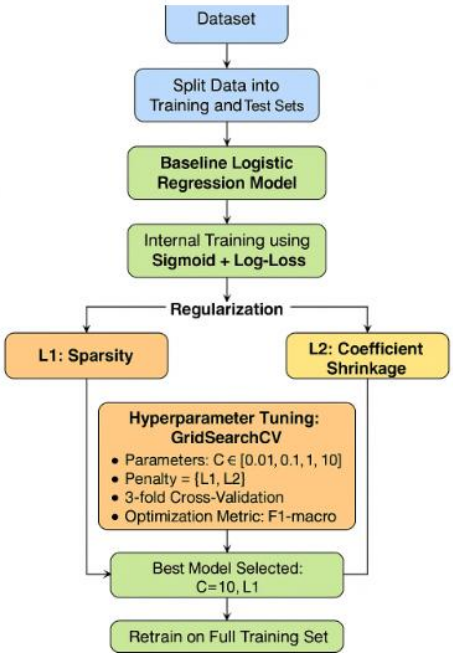


Figure 3.4 Logistic regression model

I implemented Logistic Regression as my baseline classifier. Logistic Regression predicts the probability that an observation belongs to the positive class using the **sigmoid function**:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \beta 0 + \beta 1x1 + \cdots + \beta nxn$$

where z is the linear combination of input features and learned coefficients (Google, 2025). The model internally minimizes log-loss (cross-entropy) during training:

$$\text{Log Loss} = -(ylog(p) + (1-y)log(1-p))$$

Here, y is the actual class label, while p is the predicted probability (GeeksforGeeks, 2025). This loss function penalizes misclassifications more heavily when the model is overconfident. To control complexity and improve generalization, **regularization** was introduced:

- **L1 (Lasso):** forces some coefficients to zero, producing a sparse solution.
- **L2 (Ridge):** shrinks coefficients smoothly, reducing their magnitude without eliminating them.

In scikit-learn, the parameter **C = 1/λ** determines the strength of regularization. Larger *C* values weaken the effect of regularization, allowing the model to capture more complex patterns (Scikit-learn, 2025). The **liblinear solver** was selected since it supports both L1 and L2 penalties, and the iteration limit was set to 500 to guarantee convergence.

Hyperparameter tuning was performed with **GridSearchCV (3-fold CV)**, using **F1-macro** as the optimization metric to achieve balanced performance across classes in the imbalanced dataset. While F1-macro was applied during the search, recall was prioritized in the final evaluation because missing severe/fatal accidents is more critical than generating false

positives. The best configuration was found at **C = 10 with L1 regularization**. This setup was retrained on the full dataset, and both class predictions and probability scores were generated on the test set using the default 0.5 threshold.

3.8 Multi-layer perceptron

After using logistic regression as a baseline, a Multi-Layer Perceptron (MLP) was implemented to investigate whether a deeper, non-linear model could capture patterns missed by linear methods. The MLP is a feed-forward neural network that includes several hidden layers, where each layer processes the input data before passing it forward. Within these layers, neurons transform inputs using activation functions, which allows the network to represent complex, non-linear relationships between features and the target output.

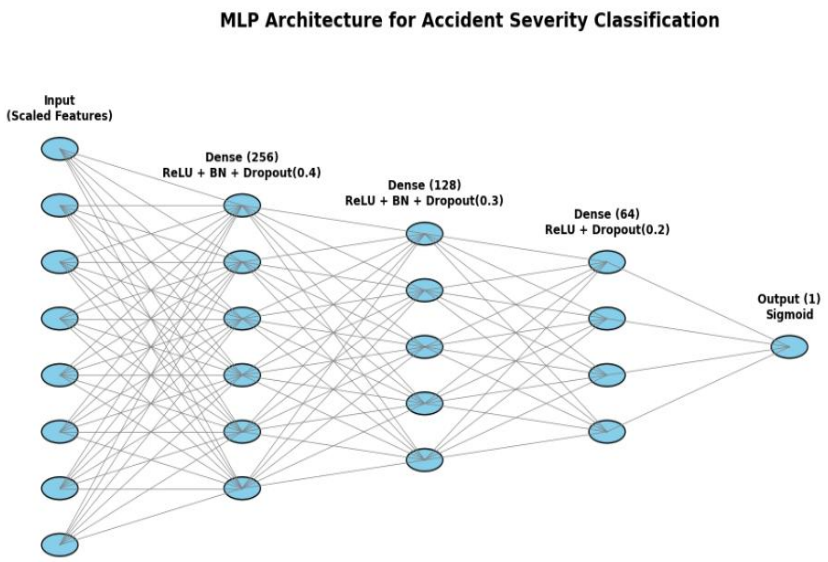


Figure 3.5 Multi-layer perceptron architecture

Since neural networks are highly sensitive to the scale of inputs, I normalised the dataset so that features were centred around zero and adjusted to a consistent scale. This preprocessing step stabilised training and allowed the model to converge more efficiently.

The architecture I designed consisted of three hidden layers with 256, 128, and 64 units. Each layer used the ReLU activation function, chosen for its efficiency and ability to minimise vanishing gradient issues. To improve generalisation and reduce overfitting, I added dropout after the hidden layers, with probabilities of 0.4, 0.3, and 0.2, respectively. I also incorporated batch normalization to keep the activations stable during training.

For the output, I included a single neuron with a sigmoid activation function, which produced probabilities between 0 and 1 for the positive class. I trained the model using the Adam optimiser with a learning rate of 0.0005, which provided a good balance between speed and

reliability. The binary cross-entropy loss function was selected, as it is the standard for binary classification tasks.

To prevent overfitting and unnecessary computation, I applied early stopping. This monitored the validation loss and stopped training if there was no improvement for five consecutive epochs, while keeping the best weights. Training was run for a maximum of 50 epochs with a batch size of 128, and I reserved 20% of the data for validation.

Once training was complete, I generated probability scores for the test set. These were then converted into binary predictions using a threshold of 0.5. I evaluated performance using accuracy, precision, recall, and F1-score, along with a confusion matrix to analyse errors in classification.

3.9 Random forest

After testing the MLP, I implemented a Random Forest classifier to take advantage of ensemble learning. Random Forest builds multiple decision trees; each trained on a randomly selected subset of the data and then combines their outputs. This ensemble method helps limit overfitting compared to a single decision tree and is effective at capturing complex, nonlinear feature relationships.

The model was trained with 100 trees, using a fixed random seed for reproducibility. Unlike neural networks, tree-based methods do not require feature scaling, so the model was applied directly to the raw data. At each split, trees used a random subset of features, which enhanced diversity across the forest and improved generalization to unseen cases.

For evaluation, both predicted labels and probability scores were generated on the test set. Performance was assessed through classification metrics (precision, recall, F1-score, accuracy) and error distribution was analysed using a confusion matrix.

3.10 XGBOOST

To improve predictive performance, XGBoost was applied as a gradient boosting algorithm that builds a sequence of decision trees, where each successive tree corrects the errors made by earlier ones through gradient-based optimization. This minimizes a differentiable objective function. Regularization terms (L1 and L2) are incorporated to reduce overfitting, while the method also handles missing values efficiently, making it well-suited for large-scale datasets. Hyperparameters were tuned using **RandomizedSearchCV** combined with **3-fold cross-validation** to enhance model reliability.

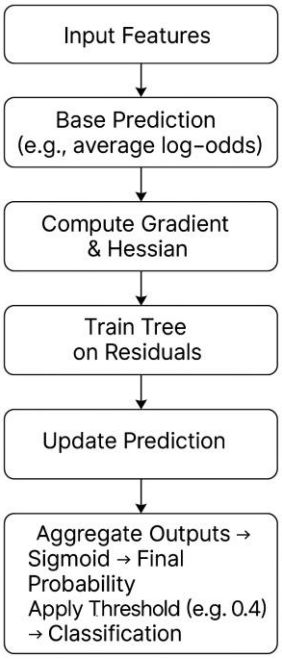


Figure 3.6 Xgboost workflow

The search covered the following parameters:

- Number of trees (n_estimators): 100, 200 – Sets the number of boosting rounds; more trees can improve accuracy but may risk overfitting (Gupta, 2021).
- Tree depth (max_depth): 3, 5, 7 – Defines the complexity of each tree; deeper structures capture interactions but may overfit (Gupta, 2021).
- Learning rate (eta): 0.01, 0.1, 0.2 – Controls how much each tree contributes; smaller values improve generalization but require more trees (Gupta, 2021).
- Subsample ratio: 0.8, 1 – Uses a fraction of training data per iteration to reduce overfitting (Gupta, 2021).
- Column sample per tree (colsample_bytree): 0.8, 1 – Randomly samples features for each tree, improving model robustness (Gupta, 2021).
- Gamma: 0, 1 – Minimum loss reduction required to create a split; higher values make the model more conservative (Gupta, 2021).
- Class weight scaling (scale_pos_weight): 1, 2 – Adjusts for class imbalance, giving more weight to minority classes (Gupta, 2021).

During hyperparameter tuning, the F1-macro metric was applied, which takes the harmonic mean of precision and recall for each class and averages the results. This ensured balanced evaluation across classes and reduced the tendency of the model to favor the majority class. Since the project's main focus is on avoiding missed severe/fatal cases, recall is chosen as the key performance metric

.

For evaluation, a custom probability threshold of 0.4 was used instead of the default 0.5. Adjusting the threshold allows balancing precision and recall; in this case, higher recall for severe/fatal crashes was prioritized to minimize false negatives, which are more critical than

false positives. Predicted probabilities were then converted into binary labels using this threshold.

The best hyperparameters found were subsample = 0.8, scale_pos_weight = 2, n_estimators = 200, max_depth = 7, learning_rate = 0.1, gamma = 0, and colsample_bytree = 0.8. Model performance was assessed with multiple metrics, including precision, recall, F1-score, and accuracy, along with confusion matrices to analyze classification errors.

4 Results

In this work, four models—**Logistic Regression, Random Forest, Gradient Boosting, and XGBoost**—were tested for accident severity prediction. Due to the strong imbalance between slight and severe/fatal cases, recall for the severe/fatal category was emphasized, since missing high-risk crashes carries greater consequences than false positives. Accuracy alone can give a misleading picture in imbalanced datasets; therefore, additional measures such as precision, recall, F1-score, confusion matrices, and ROC-AUC were used to provide a more reliable evaluation of model performance.

4.1 Evaluation metrics

To evaluate model, the following metrics were employed:

- **Recall (Sensitivity / True Positive Rate):** Recall represents the proportion of positive instances that are correctly detected. As the goal of this study is to identify severe/fatal accidents, recall is the most critical metric. Here, **TP** refers to true positives (severe/fatal crashes correctly predicted), and **FN** represents false negatives (severe/fatal crashes misclassified as slight).

$$\text{Recall} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FN}}$$

- **Receiver Operating Characteristic (ROC) Curve and AUC:** The ROC curve plots the True Positive Rate (Recall) against the False Positive Rate (FPR) at various thresholds:

$$\frac{FP}{FP + TN}$$

4.2 Model results analysis

Logistic Regression was tuned using grid search combined with 3-fold cross-validation. The optimal parameters were **C = 10, penalty = 'l1', and solver = 'liblinear'**. The model reached an overall accuracy of 70%, though performance varied considerably across classes:

- **Slight accidents (Class 0):** recall = 0.93, precision = 0.71, f1-score = 0.80

- **Severe/Fatal accidents (Class 1):** recall = 0.23, precision = 0.62, f1-score = 0.34

The confusion matrix (Figure 4.1) shows that most slight accidents were classified correctly (108,790 out of 117,095), but only 13,529 of 58,548 severe/fatal cases were identified, leaving 45,019 misclassified. The ROC curve produced an AUC of 0.665, reflecting only moderate ability to distinguish between classes. From a recall standpoint, Logistic Regression struggles to capture high-severity accidents, limiting its usefulness for practical accident prevention.

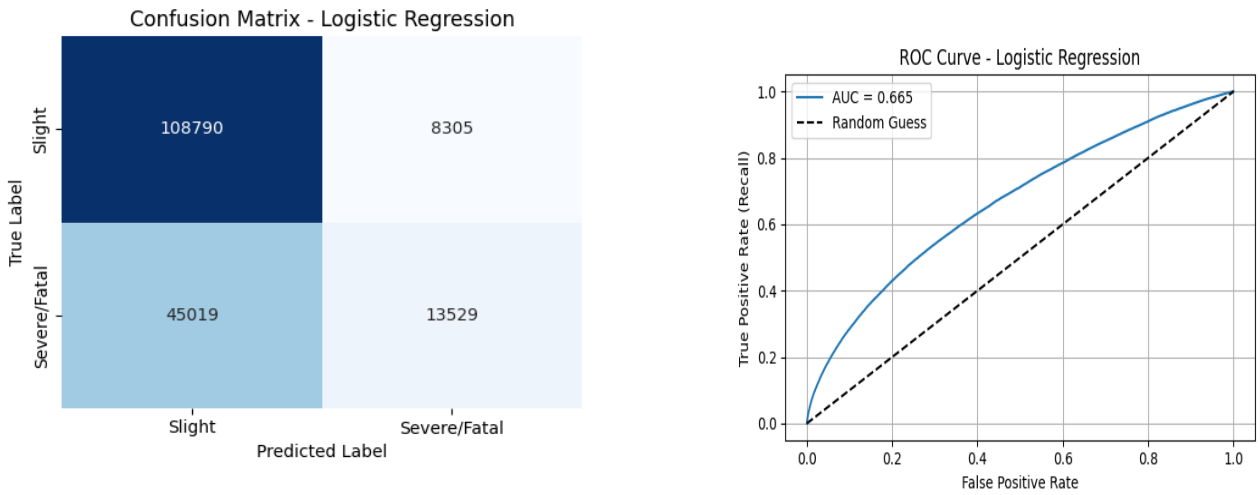


Figure 4.1 Logistic regression model results

The MLP model, with two hidden layers of 100 neurons each, improved recall for severe/fatal accidents (**Class 1) to 0.29, compared to 0.23** for Logistic Regression, while maintaining high recall for slight accidents (Class 0 = 0.91).
The confusion matrix shows that 17,172 of 58,548 severe/fatal accidents were correctly identified, compared to 13,529 with Logistic Regression. The ROC-AUC is 0.707, indicating moderate discriminative ability and a small improvement over Logistic Regression (0.665).
Interpretation: The neural network better captures minority-class patterns, enhancing recall for high-risk cases, although the majority of severe/fatal accidents remain undetected.

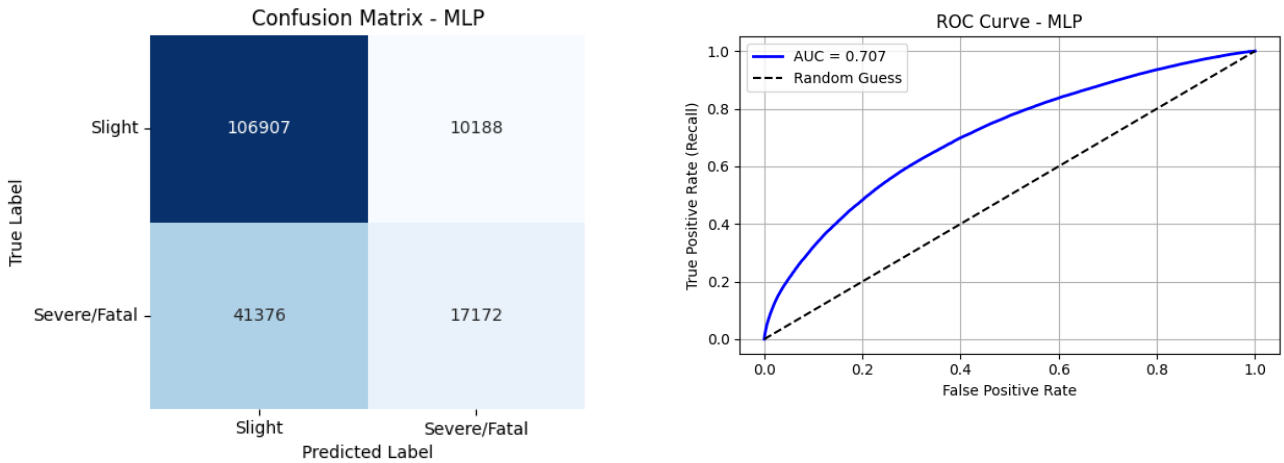


Figure 4.2 MLP model results

Random Forest reported a recall of **0.31** for severe/fatal crashes (Class 1) and **0.90** for slight crashes (Class 0), offering better detection of high-risk cases than Logistic Regression. Out of 58,548 severe/fatal cases, 18,323 were correctly identified, leaving 40,225 misclassified. The model reached a ROC-AUC of **0.699**, indicating only moderate separation ability. Compared to linear methods, Random Forest is better at recognizing minority-class patterns, giving a slight boost in severe/fatal case detection, though a large proportion of serious accidents still go undetected.

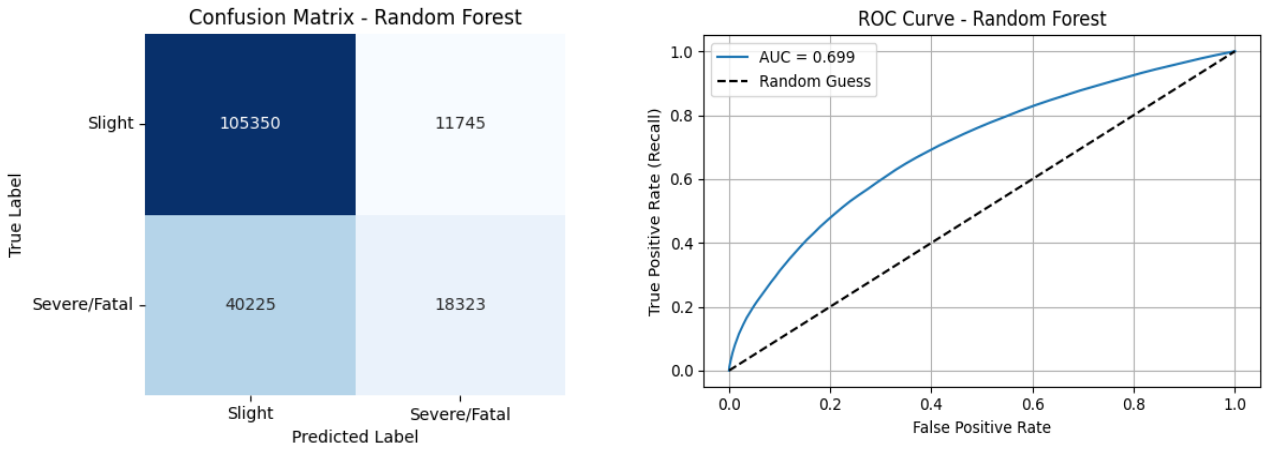


Figure 4.3 Random forest results

XGBoost, tuned with optimal hyperparameters, achieved strong performance with a ROC-AUC of 0.711. At a 0.4 threshold, it correctly identified 45,704 severe/fatal cases—far higher than Logistic Regression and MLP. It also achieved the best recall (0.78) for serious crashes, making it the most effective model for detecting high-risk accidents, despite an increase in false positives for slight cases.

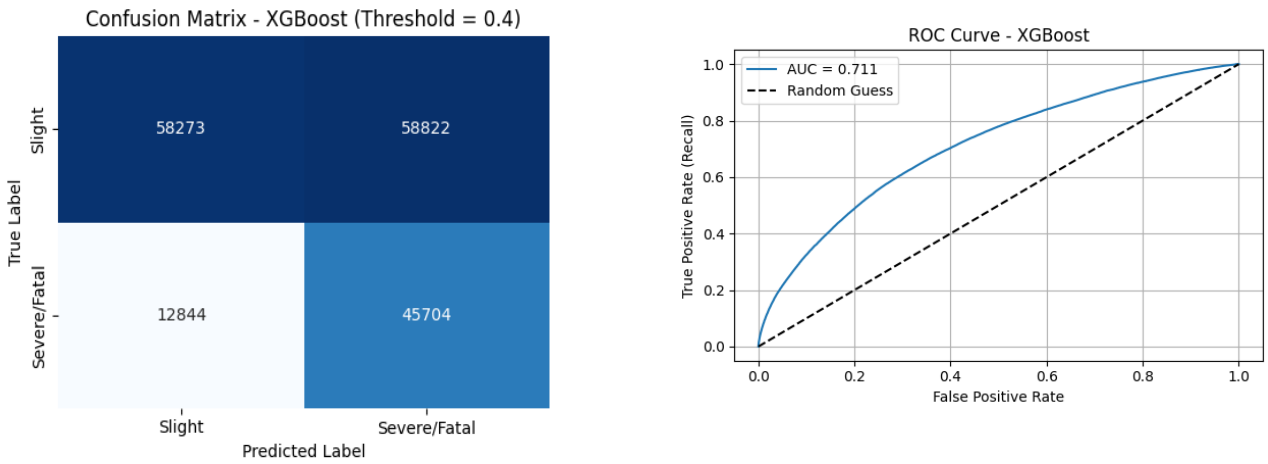


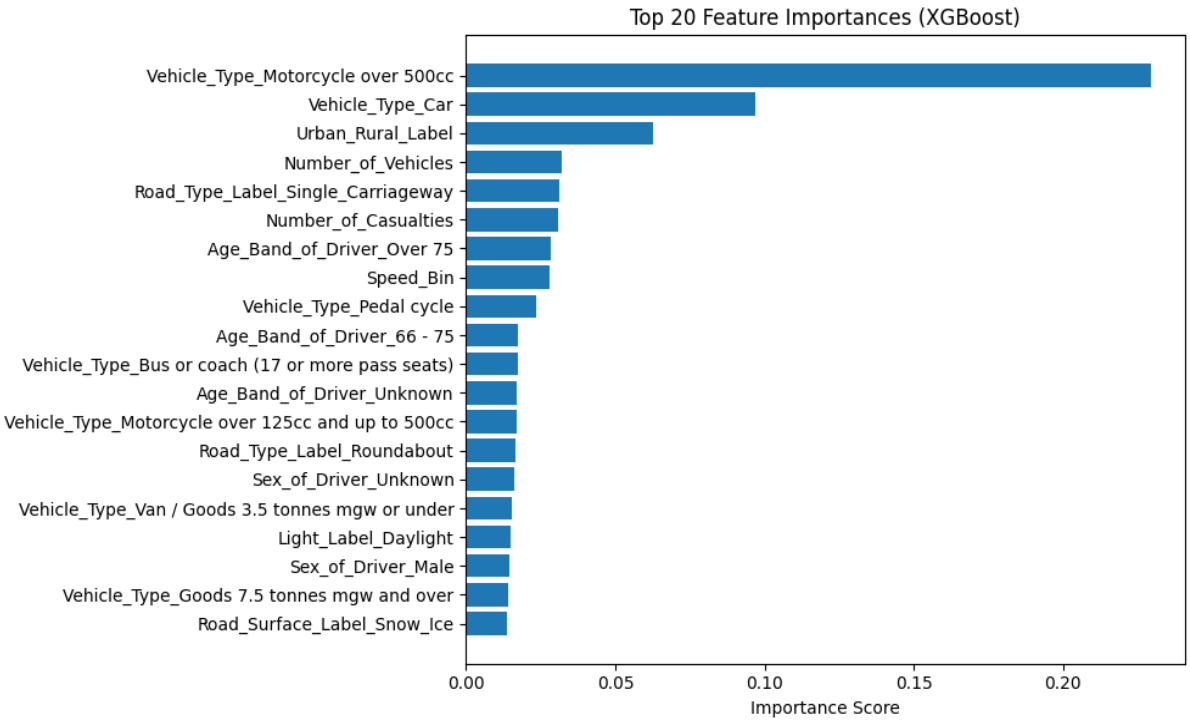
Figure 4.4 XGBoost results

Model	Precision	Recall	F1 Score	ROC AUC
XGBoost	0.437	0.781	0.561	0.711
Random Forest	0.609	0.313	0.414	0.699
MLP	0.620	0.307	0.410	0.706
Logistic Regression	0.620	0.231	0.337	0.665

Table 1: Classification report

4.3 Feature importance and output

The XGBoost feature importance analysis highlights that vehicle and driver-related attributes dominate accident severity prediction. Among all factors, motorcycles over 500cc had the highest influence, followed by cars and the urban–rural setting, indicating that both vehicle type and crash location strongly affect outcomes. Additional key predictors included the number of vehicles and casualties, as well as driver age groups, especially older drivers, suggesting demographic vulnerability. Road features such as single carriageways, roundabouts, and snow/ice surfaces also mattered but had lower influence compared to vehicle and driver characteristics. In contrast, the temporal and environmental feature analysis revealed that urban–rural differences, single carriageways, and speed were the most critical conditions shaping severity, with roundabouts, daylight, snow/ice, and rain also contributing. Time factors such as hour or day of week were less impactful, indicating that where and under what conditions an accident occurs is more important than when. Taken together, these results suggest that the most severe accidents often involve high-risk vehicles and vulnerable drivers in hazardous road or weather contexts, highlighting the combined effect of human, vehicular, and environmental factors.



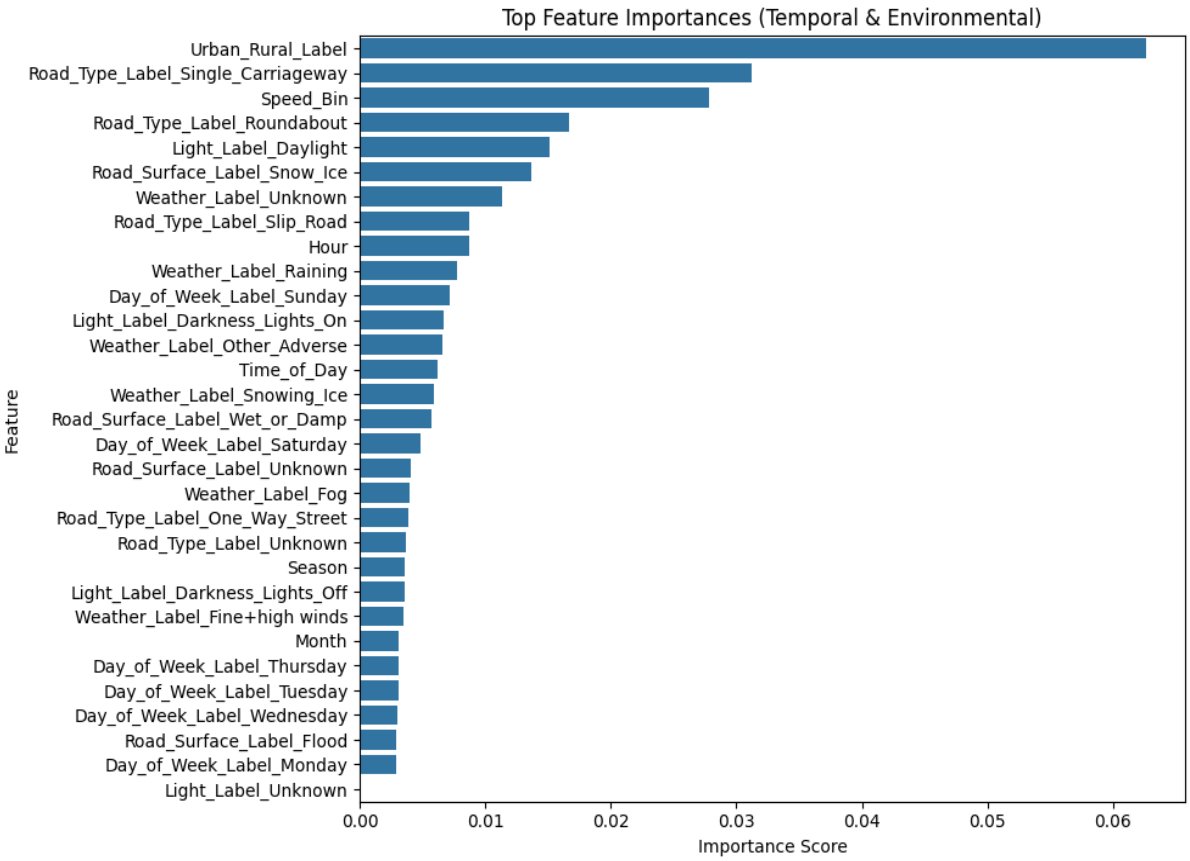


Figure 4.5 Feature importance from XGBoost model

The outputs from the Accident Severity Prediction interface demonstrate how different combinations of factors influence accident outcomes. In the first scenario, a rural single carriageway with high speed, rainy weather, poor lighting, wet road surface, and multiple casualties led the model to predict a Severe/Fatal outcome with a probability of 77.28%, showing how hazardous environments and risky conditions amplify severity. In contrast, the second scenario on an urban single carriageway with moderate speed, rainy but daylight conditions, and a younger driver resulted in a Slight prediction, with a lower 48.57% probability of Severe/Fatal. Comparing these cases highlights that while weather and surface conditions contribute to risk, factors such as road type, speed, lighting, and urban–rural context play a decisive role in shaping severity. This analysis reinforces the feature importance findings, showing that the interaction between vehicle, driver, and environmental conditions determines accident severity rather than any single factor alone.

The Accident Severity Prediction interface was initially developed with help of ChatGPT (OpenAI, 2023) to generate templates and suggestions, with all code reviewed and adapted

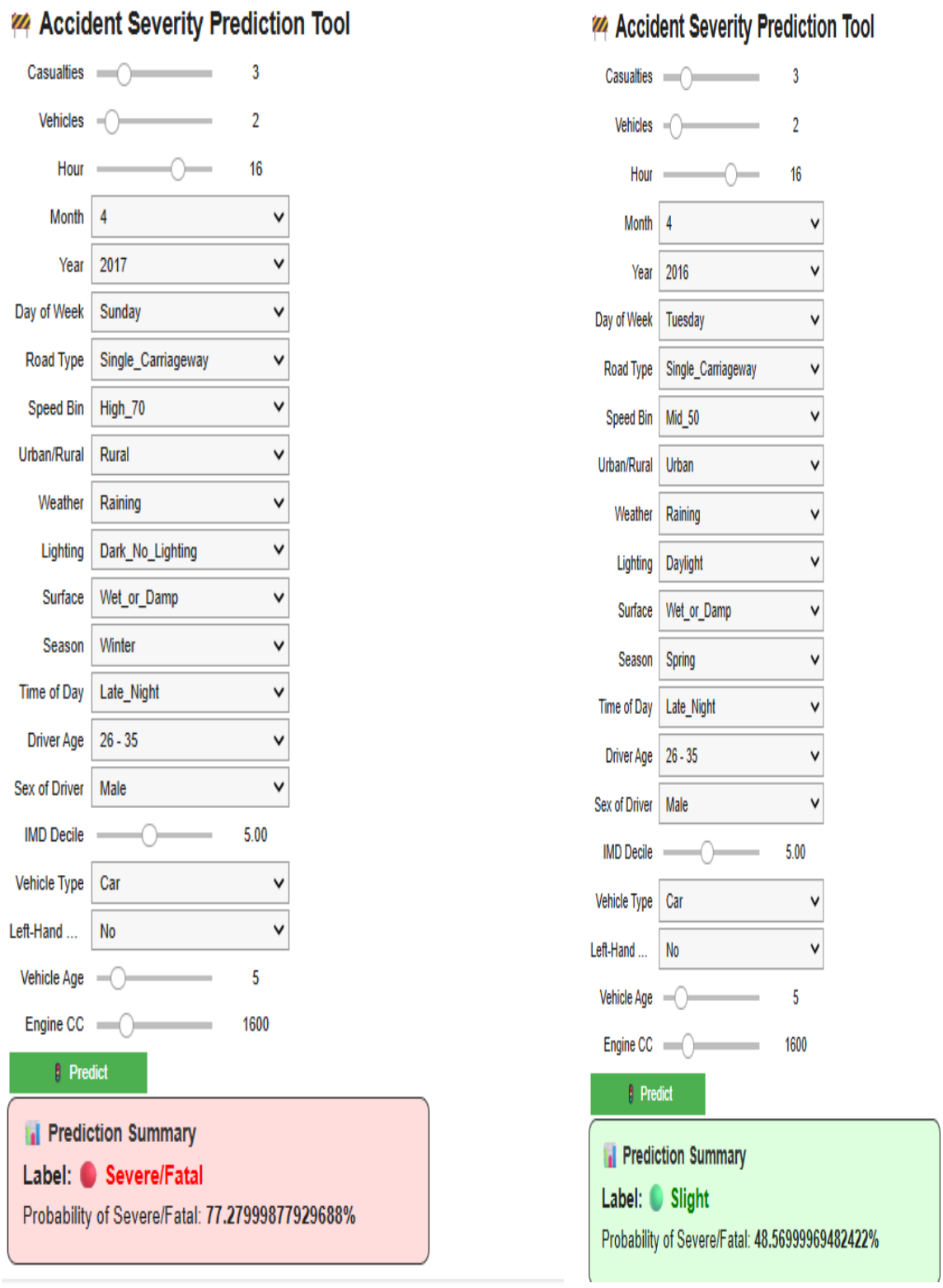


Figure 4.6 Output

4.3 Summary

Four predictive models—Logistic Regression, MLP, Random Forest, and XGBoost—were trained to classify accident severity as Slight or Severe/Fatal. Among them, XGBoost achieved the best performance, with a recall **of 0.78 for severe/fatal accidents (Class 1) and a ROC-AUC of 0.71.** This high recall is particularly important because it indicates the model's strong ability to correctly identify high-risk accidents, minimizing the number of severe incidents that go undetected. Detecting severe accidents accurately is crucial for emergency planning, resource allocation, and preventive measures, as missed cases could have serious real-world consequences.

Random Forest and MLP offered moderate improvements over Logistic Regression, with recalls of 0.31 and 0.28 for severe/fatal accidents, respectively. Logistic Regression, while simple and interpretable, had the lowest recall (0.23), highlighting its limitations in handling the highly imbalanced dataset.

Focusing on recall for the minority class ensures that high-severity accidents are prioritized in prediction, enabling authorities to prepare better emergency responses and implement targeted safety interventions.

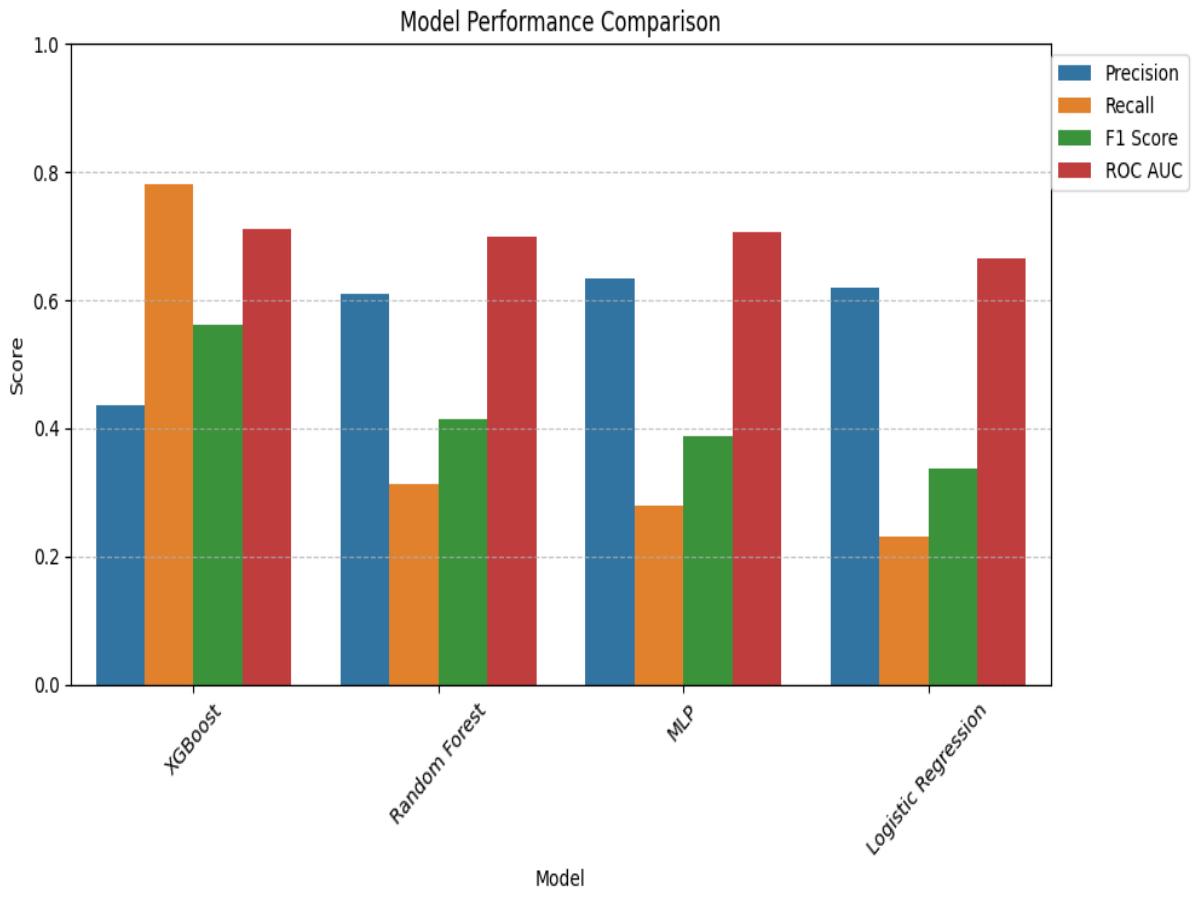


Figure 4.6 Model Comparison

5 Conclusion and Future Work

This project explored predicting accident severity on UK roads using the STATS19 dataset. Several machine learning models—Logistic Regression, MLP, Random Forest, and XGBoost—were implemented and compared. **XGBoost performed best**, achieving a recall of 0.78 for severe/fatal accidents, demonstrating strong ability to identify high-risk cases. Feature importance analysis highlighted road type, speed, lighting, and urban–rural context as key predictors, showing that accident severity arises from the interaction of multiple factors rather than any single variable. The project demonstrates that machine learning can provide actionable insights for road safety planning, emergency response, and policy interventions.

Limitations

1. **Data Imbalance:** Severe and fatal accidents are rare, requiring undersampling, which may lead to loss of information about slight accidents.
2. **Feature Limitations:** Some potentially influential factors, such as driver behavior history or real-time traffic conditions, were not included in the dataset.
3. **Model Interpretability:** While XGBoost performs best, its complexity makes it less interpretable than linear models.
4. **Static Dataset:** Predictions are based on historical data; real-time or changing conditions (e.g., weather, traffic) are not considered.

Future Work

1. **Incorporate Additional Data:** Including driver behavior, traffic density, or real-time environmental data could improve predictive accuracy.
2. **Explore Advanced Models:** Investigate other ensemble methods, deep learning architectures, or hybrid models for better minority-class detection.
3. **Real-Time Prediction Tool:** Extend the interface for dynamic, real-time predictions using live data feeds.
4. **Policy Simulation:** Use model insights to simulate interventions (e.g., speed limits, lighting improvements) and assess potential impact on accident severity.

References

Ahmed, S., Hossain, M.A., Ray, S.K., Bhuiyan, M.M.I. & Sabuj, S.R., 2023. A study on road accident prediction and contributing factors using explainable machine learning models: analysis and performance. *Transportation Research Interdisciplinary Perspectives*, 19, Article 100814. Available at: https://doi.org/10.1016/j.trip.2023.100814 [Accessed 28 August 2025].

GeeksforGeeks, 2025. *ML Cost Function in Logistic Regression / Regularization in Machine Learning*. Available at: https://www.geeksforgeeks.org/machine-learning/ml-cost-function-in-logistic-regression/ [Accessed 28 August 2025].

Google, 2025. *Machine Learning Crash Course: Logistic Regression Sigmoid Function*. Available at: https://developers.google.com/machine-learning/crash-course/logistic-regression/sigmoid-function [Accessed 28 August 2025].

Gupta, A., 2021. XGBoost Hyperparameters — Explained. *Medium*. Available at: https://amangupta16.medium.com/xgboost-hyperparameters-explained-bb6ce580501d [Accessed 28 August 2025].

He, H. & Garcia, E.A., 2009. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9), pp.1263–1284. Available at: https://doi.org/10.1109/TKDE.2008.239 [Accessed 28 August 2025].

Hsu, Y.T., Chang, S.C. & Hsu, T.H., 2020. Analysis of traffic accident severity at intersection using logistic regression model. *Journal of Engineering Research and Reports*, 13(4), pp.1–9. Available at: https://journaljerr.com/index.php/JERR/article/view/709 [Accessed 28 August 2025].

Little, R.J.A. & Rubin, D.B., 2014. *Statistical Analysis with Missing Data*. 3rd ed. Wiley. Available at: https://doi.org/10.1002/9781119013563 [Accessed 28 August 2025].

Scikit-learn, 2025. *Logistic Regression Documentation*. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html [Accessed 28 August 2025].

Tsiaras, K., 2020. UK Road Safety Accidents and Vehicles (2005–2017). *Kaggle* [online]. Available at: https://www.kaggle.com/datasets/tsiaras/uk-road-safety-accidents-and-vehicles [Accessed 27 August 2025].

Zhou, W., 2024. Traffic accident severity prediction based on data cleaning and machine learning (Random Forest / Xgboost). *Highlights in Science, Engineering and Technology*, 85, pp.376–388. Available at: https://doi.org/10.54097/h8cq6864 [Accessed 28 August 2025].

Department for Transport (DfT), 2025. Road Safety Data: Personal Injury Road Accidents, Great Britain. [online] Available at: https://data.gov.uk/dataset/cb7ae6f0-4be6-4935-9277-47e5ce24a11f/road-safety-data [Accessed 28 August 2025].

APPENDIX

Code

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import joblib
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
from sklearn.metrics import precision_recall_curve

# Path to your file
as_accident_df = pd.read_csv(
    '/content/drive/MyDrive/RoadAccidentData/archive/Accident_Information.csv',
    encoding='ISO-8859-1',
    dtype={'Accident_Index': str},
)
as_vehicle_df = pd.read_csv(
    '/content/drive/MyDrive/RoadAccidentData/archive-vehicle (1)/Vehicle_Information.csv',
    encoding='ISO-8859-1',
    dtype={'Accident_Index': str},
)
print(as_accident_df.columns)
print(as_vehicle_df.columns)

# Perform join to keep only accidents that have at least one vehicle record
as_accident_df = pd.merge(
    as_accident_df,
    as_vehicle_df,
    on='Accident_Index',
    how='inner'
)

print("final shape:", as_accident_df.shape)
display(as_accident_df.head())
#  Missing counts per column and it displays in ascending order
missing_counts = as_accident_df.isna().sum().sort_values(ascending=False)
print("Missing counts per column:")
print(missing_counts)

# 2. Missing values in percentage
```

```
missing_pct = (missing_counts / len(as_accident_df) * 100).sort_values(ascending=False)
print("\nMissing percentages per column:")
print(missing_pct)

# listing the columns to be dropped
cols_to_drop = [
    'Accident_Index',
    '1st_Road_Class',
    '1st_Road_Number',
    '2nd_Road_Class',
    '2nd_Road_Number',
    'Carriageway_Hazards',
    'Did_Police_Officer_Attend_Scene_of_Accident',
    'Junction_Control',
    'Junction_Detail',
    'Local_Authority_(Highway)',
    'Pedestrian_Crossing-Human_Control',
    'Pedestrian_Crossing-Physical_Facilities',
    'Special_Conditions_at_Site',
    'Driver_Home_Area_Type',
    'Hit_Object_in_Carriageway',
    'Hit_Object_off_Carriageway',
    'Journey_Purpose_of_Driver',
    'Junction_Location',
    'Propulsion_Code',
    'Skidding_and_Overturning',
    'Towing_and_Articulation',
    'Vehicle_Leaving_Carriageway',
    'Vehicle_Location.Restricted_Lane',
    'Vehicle_Manoeuvre',
    'X1st_Point_of_Impact',
    'Year_x',
    'Year_y'
]

# dropping columns listed above
as_accident_df = as_accident_df.drop(columns=cols_to_drop, errors='ignore')

print("Remaining columns:", as_accident_df.columns.tolist())
print("New shape:", as_accident_df.shape)

import pandas as pd

# Define column categories

vehicle_numeric = [
    'Age_of_Vehicle',
    'Driver_IMD_Decile',
    'Engine_Capacity_.CC.',
    'Vehicle_Reference'
```

```
]

vehicle_categorical = [
    'Age_Band_of_Driver',
    'make',
    'model',
    'Sex_of_Driver',
    'Vehicle_Type',
    'Was_Vehicle_Left_Hand_Drive'
]

# Define hidden missing tokens
hidden_missing = [
    'Unknown', 'Not known', 'Data missing or out of range',
    'nan', 'N/A', '-', '', 'None', 'null'
]

# Handle numeric columns
for col in vehicle_numeric:
    # Ensure numeric
    as_accident_df[col] = pd.to_numeric(as_accident_df[col], errors='coerce')
    # Fill NaNs with median
    median_val = as_accident_df[col].median()
    as_accident_df[col] = as_accident_df[col].fillna(median_val)

# Handle categorical columns with hidden missing values
for col in vehicle_categorical:
    # Convert to string, strip whitespace, and replace hidden missing with 'Unknown'
    cleaned = (
        as_accident_df[col]
        .astype(str)
        .str.strip()
        .replace(hidden_missing, 'Unknown')
    )
    as_accident_df[col] = cleaned

    # Create is_unknown flag column for feature importance analysis
    as_accident_df[f'{col}_is_unknown'] = cleaned == 'Unknown'

# Check for remaining missing values
missing_after = as_accident_df[vehicle_numeric + vehicle_categorical].isna().sum()
print("Missing values after imputation:\n", missing_after)

# Check how many 'Unknown' values exist per categorical column
print("Counts of 'Unknown' values per categorical column:")
for col in vehicle_categorical:
    unknown_count = (as_accident_df[col] == 'Unknown').sum()
    print(f"{col}: {unknown_count}")

print("New shape:", as_accident_df.shape)
```

```
# original accident-level columns
accident_cols = [
    'Accident_Severity','Date','Time','Latitude','Longitude',
    'Local_Authority_(District)','Urban_or_Rural_Area',
    'Road_Surface_Conditions','Road_Type','Speed_limit','Weather_Conditions',
    'Light_Conditions','Number_of_Casualties','Number_of_Vehicles',
    'Police_Force','Day_of_Week','InScotland'
]

# vehicle-level columns
vehicle_cols = [
    'Age_Band_of_Driver','Age_of_Vehicle','Driver_IMD_Decile',
    'Engine_Capacity_.CC.','Sex_of_Driver',
    'Vehicle_Reference','Vehicle_Type','Was_Vehicle_Left_Hand_Drive'
]

# merged DataFrame to include both sets
as_accident_level = as_accident_df[accident_cols + vehicle_cols].copy()
print("After subsetting (accident + vehicle):", as_accident_level.shape)
print(as_accident_level.columns.tolist())

#  Unique values in Urban_or_Rural_Area
print("Unique values in Urban_or_Rural_Area:")
print(as_accident_level['Urban_or_Rural_Area'].value_counts(dropna=False))

#  Unique values in Weather_Conditions
print("\nUnique values in Weather_Conditions:")
print(as_accident_level['Weather_Conditions'].value_counts(dropna=False))


#  Identify all object- or category-dtype columns
cat_cols = as_accident_level.select_dtypes(include=['object', 'category']).columns.tolist()

# Loop through and print unique values for each
for col in cat_cols:
    unique_vals = as_accident_level[col].unique()
    print(f"Column: {col}")
    print(f"  ({len(unique_vals)} unique values) → {unique_vals}\n")



# Fill missing Time with "00:00"
as_accident_level['Time'] = as_accident_level['Time'].fillna('00:00')

#  Latitude/Longitude to numeric
as_accident_level['Latitude'] = pd.to_numeric(
    as_accident_level['Latitude'], errors='coerce'
)
```

```
as_accident_level['Longitude'] = pd.to_numeric(
    as_accident_level['Longitude'], errors='coerce'
)

#  Now drop any row where either is NaN
before = as_accident_level.shape[0]
as_accident_level = as_accident_level.dropna(
    subset=['Latitude','Longitude'], how='any'
)


#Drop OS grid columns entirely
as_accident_level = as_accident_level.drop(columns=['Location_Easting_OSGR',
'Location_Northing_OSGR'], errors='ignore')

# Fill missing Speed_limit with -1
as_accident_level['Speed_limit'] = as_accident_level['Speed_limit'].fillna(-1)

#Fill missing Number_of_Casualties / Number_of_Vehicles with 0
as_accident_level['Number_of_Casualties'] = \
    as_accident_level['Number_of_Casualties'].fillna(0).astype(int)
as_accident_level['Number_of_Vehicles']   = \
    as_accident_level['Number_of_Vehicles'].fillna(0).astype(int)

# Fill missing InScotland with "Unknown"
as_accident_level['InScotland'] =
as_accident_level['InScotland'].fillna('Unknown').astype(str).str.strip()


# Fill all remaining categorical columns with "Unknown" if missing
for col in [
    'Local_Authority_(District)',
    'Urban_or_Rural_Area',
    'Road_Surface_Conditions',
    'Road_Type',
    'Weather_Conditions',
    'Light_Conditions',
    'Day_of_Week',
    'Police_Force'
]:
    as_accident_level[col] = as_accident_level[col].fillna('Unknown').astype(str).str.strip()



# 3.1 TEMPORAL FEATURES
as_accident_level['Hour'] = pd.to_datetime(
    as_accident_level['Date'] + ' ' + as_accident_level['Time'],
    format='%Y-%m-%d %H:%M',  # Time string already "HH:MM", Date "YYYY-MM-DD"
    errors='coerce'
).dt.hour
```

```
# If any Hour parse failed (unlikely), fill with median
as_accident_level['Hour'] =
as_accident_level['Hour'].fillna(as_accident_level['Hour'].median()).astype(int)

# Weekday and Month from Date_dt
as_accident_level['Month']       = as_accident_level['Date_dt'].dt.month

# 5.8 DAY OF WEEK: Day_of_Week_Label from Date_dt (ignore original Day_of_Week)
as_accident_level['Day_of_Week_Label'] = as_accident_level['Date_dt'].dt.day_name()

# Assuming 'Date_dt' is your datetime column
as_accident_level['Year'] = as_accident_level['Date_dt'].dt.year

# Season from Month
def month_to_season(m):
    if m in [12, 1, 2]:
        return 'Winter'
    elif m in [3, 4, 5]:
        return 'Spring'
    elif m in [6, 7, 8]:
        return 'Summer'
    else:
        return 'Autumn'
as_accident_level['Season'] = as_accident_level['Month'].apply(month_to_season)

# Time_of_Day from Hour
def time_of_day(h):
    if pd.isna(h):
        return 'Unknown'
    if h < 6:
        return 'Late_Night'
    elif h < 9:
        return 'Morning_Peak'
    elif h < 16:
        return 'Daytime'
    elif h < 19:
        return 'Evening_Peak'
    else:
        return 'Night'
as_accident_level['Time_of_Day'] = as_accident_level['Hour'].apply(time_of_day)

as_accident_level.drop(columns=['Date_dt'], inplace=True)



#  Define an exact mapping from each weather
weather_map = {
    'Fine no high winds':       'Fine',
    'Fine + high winds':        'Fine+high winds',
```

```
    'Raining no high winds':    'Raining',
    'Raining + high winds':     'Raining',
    'Snowing no high winds':    'Snowing_Ice',
    'Snowing + high winds':     'Snowing_Ice',
    'Fog or mist':              'Fog',
    'Other':                    'Other_Adverse',
    'Unknown':                  'Unknown',
    'Data missing or out of range': 'Unknown'
}

#  Ensure there are no trailing
as_accident_level['Weather_Conditions'] =
as_accident_level['Weather_Conditions'].astype(str).str.strip()

as_accident_level['Weather_Label'] = (
    as_accident_level['Weather_Conditions']
    .map(weather_map)
    .fillna('Unknown')
)

# Verify that the mapping worked correctly
print("Unique raw Weather_Conditions values:")
print(as_accident_level['Weather_Conditions'].unique(), "\n")

print("Unique mapped Weather_Label values:")
print(as_accident_level['Weather_Label'].unique())

print("\nValue counts for Weather_Label:")
print(as_accident_level['Weather_Label'].value_counts())


def map_road_surface(x):
    if x == 'Dry':
        return 'Dry'
    elif x == 'Wet or damp':
        return 'Wet_or_Damp'
    elif x in ['Snow', 'Frost or ice']:
        return 'Snow_Ice'
    elif x == 'Flood over 3cm. deep':
        return 'Flood'
    else:  # 'Data missing or out of range' or anything else
        return 'Unknown'

as_accident_level['Road_Surface_Label'] =
as_accident_level['Road_Surface_Conditions'].apply(map_road_surface)

print("Unique Road_Surface_Label values:",
as_accident_level['Road_Surface_Label'].unique())
```

```
def map_road_type(x):
    if x == 'Dual carriageway':
        return 'Dual_Carriageway'
    elif x == 'Single carriageway':
        return 'Single_Carriageway'
    elif x == 'One way street':
        return 'One_Way_Street'
    elif x == 'Roundabout':
        return 'Roundabout'
    elif x == 'Slip road':
        return 'Slip_Road'
    else:  # 'Unknown' or 'Data missing or out of range'
        return 'Unknown'

as_accident_level['Road_Type_Label'] =
as_accident_level['Road_Type'].apply(map_road_type)

print("Unique Road_Type_Label values:", as_accident_level['Road_Type_Label'].unique())

as_accident_level['Light_Conditions'] =
as_accident_level['Light_Conditions'].astype(str).str.strip()

def map_light(x):
    if x == 'Daylight':
        return 'Daylight'
    elif x == 'Darkness - lights lit':
        return 'Darkness_Lights_On'
    elif x == 'Darkness - lights unlit':
        return 'Darkness_Lights_Off'
    elif x in ['Darkness - lighting unknown', 'Darkness - no lighting']:
        return 'Dark_No_Lighting'
    else:  # 'Data missing or out of range' or anything else
        return 'Unknown'

as_accident_level['Light_Label'] = as_accident_level['Light_Conditions'].apply(map_light)

print("Unique Light_Label values:", as_accident_level['Light_Label'].unique())
def speed_bin(x):
    try:
        sl = float(x)
    except:
        return 'Unknown'
    if sl <= 30:
        return 'Low_30'
    elif sl <= 50:
        return 'Mid_50'
    elif sl <= 70:
        return 'High_70'
    else:
        return 'Very_High'
```

```
as_accident_level['Speed_Bin'] = as_accident_level['Speed_limit'].apply(speed_bin)

print("Unique Speed_Bin values:", as_accident_level['Speed_Bin'].unique())
# Now that we have mapped/engineered every column, drop the raw ones:
cols_to_drop = [
    'Date',
    'Time',
    'Urban_or_Rural_Area',
    'Road_Surface_Conditions',
    'Road_Type',
    'Speed_limit',
    'Weather_Conditions',
    'Light_Conditions',
    'Day_of_Week'
]

as_accident_level = as_accident_level.drop(columns=[c for c in cols_to_drop if c in
as_accident_level.columns],
                                errors='ignore')

print("\nFinal columns in accident_level:")
print(as_accident_level.columns.tolist())

print("\nFinal shape:", as_accident_level.shape)

# Ensure 'Accident_Severity' column exists and has the correct categories
order = ['Slight', 'Serious', 'Fatal']

plt.figure(figsize=(6, 4))
sns.countplot(
    data=as_accident_level,
    x='Accident_Severity',
    hue='Accident_Severity',          # Add this to avoid the future warning
    order=order,
    palette='Set2',                   # Choose a visually appealing palette
    edgecolor='black',
    legend=False                      # Remove redundant legend
)

plt.title('Accident Severity Distribution', fontsize=14)
plt.xlabel('Severity Level', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.tight_layout()
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()



plt.figure(figsize=(10, 6))
```

```
sns.countplot(data=as_accident_level, x='Weather_Label', hue='Accident_Severity',
palette='Set2')
plt.title('Accident Severity by Weather Condition')
plt.xlabel('Weather Condition')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.legend(title='Severity')
plt.tight_layout()
plt.show()

plt.figure(figsize=(8, 5))
sns.countplot(data=as_accident_level, x='Time_of_Day', hue='Accident_Severity',
palette='coolwarm')
plt.title('Accident Severity by Time of Day')
plt.xlabel('Time of Day')
plt.ylabel('Accident Count')
plt.legend(title='Severity')
plt.tight_layout()
plt.show()


plt.figure(figsize=(10, 5))

# Use a well-contrasted palette manually
custom_palette = {'Slight': '#1f77b4', 'Serious': '#ff7f0e', 'Fatal': '#d62728'}

sns.countplot(
    data=as_accident_level,
    x='Month',
    hue='Accident_Severity',
    palette=custom_palette,
    edgecolor='black'
)

plt.title('Monthly Accident Counts by Severity', fontsize=14)
plt.xlabel('Month (1=Jan, 12=Dec)')
plt.ylabel('Number of Accidents')
plt.xticks(ticks=range(0, 12), labels=[
    'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
    'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'
])
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Move legend outside the plot area to the top right
plt.legend(title='Severity', loc='upper left', bbox_to_anchor=(1, 1))

plt.tight_layout()
plt.show()
```

```
# Simply get a Python list of column names
final_columns = list(as_accident_level.columns)
print(final_columns)

as_accident_level.info ()


# Create the target variable
as_accident_level['Severe&Fatal'] = as_accident_level['Accident_Severity'].isin(['Serious',
'Fatal']).astype(int)

# Label encode ordinal/categorical columns
label_encode_cols = ['Season', 'Time_of_Day', 'Urban_Rural_Label', 'Speed_Bin']
label_encoders = {}

for col in label_encode_cols:
    le = LabelEncoder()
    as_accident_level[col] = le.fit_transform(as_accident_level[col].astype(str))
    label_encoders[col] = le

# One-hot encode non-ordinal categorical features
onehot_encode_cols = [
    'Weather_Label', 'Light_Label', 'Road_Surface_Label',
    'Road_Type_Label', 'Day_of_Week_Label',
    'Age_Band_of_Driver', 'Sex_of_Driver',
    'Vehicle_Type', 'Was_Vehicle_Left_Hand_Drive'
]

as_accident_encoded = pd.get_dummies(as_accident_level, columns=onehot_encode_cols,
drop_first=True)

# Define feature columns
features = [
    'Number_of_Casualties', 'Number_of_Vehicles', 'Hour', 'Month', 'Year',
    'Age_of_Vehicle', 'Driver_IMD_Decile', 'Engine_Capacity_.CC.', 'Vehicle_Reference',
] + label_encode_cols + [col for col in as_accident_encoded.columns if any(prefix + '_' in col
for prefix in onehot_encode_cols)]

# Create feature matrix and target
X = as_accident_encoded[features]
y = as_accident_encoded['Severe&Fatal']

plt.figure(figsize=(6, 4))
ax = sns.countplot(
    data=as_accident_level,
    x='Severe&Fatal',
    hue='Severe&Fatal',
    palette='Set2'
)
```

```
ax.legend_.remove()

plt.title('Distribution of Severe or Fatal Accidents', fontsize=14)
plt.xlabel('Severe & Fatal (1 = Serious/Fatal, 0 = Slight)', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

from sklearn.utils import resample

# Define features and target
X = as_accident_encoded[features]
y = as_accident_encoded['Severe&Fatal']

# Combine into one DataFrame
full_data = pd.concat([X, y], axis=1)

# Split into majority and minority classes
majority = full_data[full_data['Severe&Fatal'] == 0]
minority = full_data[full_data['Severe&Fatal'] == 1]

# Downsample majority class
majority_downsampled = resample(
    majority,
    replace=False,
    n_samples=len(minority) * 2,
    random_state=42
)

# Combine downsampled majority with all minority
balanced_df = pd.concat([majority_downsampled, minority])

# BEFORE DOWNSAMPLING
plt.figure(figsize=(6, 4))
sns.countplot(data=as_accident_level, x='Severe&Fatal', palette='Set2', hue='Severe&Fatal',
legend=False)
plt.title('Before Downsampling: Severe&Fatal Class Distribution')
plt.xlabel('Severe & Fatal (1 = Serious/Fatal, 0 = Slight)')
plt.ylabel('Count')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

print("Class counts before downsampling:")
print(as_accident_level['Severe&Fatal'].value_counts())
print(f"Severe/Fatal proportion: {as_accident_level['Severe&Fatal'].mean():.3%}")

# AFTER DOWNSAMPLING
balanced_df = pd.concat([majority_downsampled, minority])
```

```
plt.figure(figsize=(6, 4))
sns.countplot(data=balanced_df, x='Severe&Fatal', hue='Severe&Fatal', palette='Set1',
legend=False)
plt.title('After Downsampling: Severe&Fatal Class Distribution')
plt.xlabel('Severe & Fatal (1 = Serious/Fatal, 0 = Slight)')
plt.ylabel('Count')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

print("\nClass counts after downsampling:")
print(balanced_df['Severe&Fatal'].value_counts())
print(f"New Severe/Fatal proportion: {balanced_df['Severe&Fatal'].mean():.3%}")

from sklearn.model_selection import train_test_split

# Step 1: Separate features and target
X = balanced_df.drop(columns=['Severe&Fatal'])
y = balanced_df['Severe&Fatal']

# Step 2: Perform train-test split (only once)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)


from sklearn.linear_model import LogisticRegression
# Train Logistic Regression
log_clf = LogisticRegression(max_iter=500, solver='liblinear')
log_clf.fit(X_train, y_train)

# Predictions
y_pred = log_clf.predict(X_test)
y_proba = log_clf.predict_proba(X_test)[:, 1]

# Evaluation
print("\nLogistic Regression Report:")
print(classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("ROC AUC Score:", roc_auc_score(y_test, y_proba))


from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score,
precision_recall_curve, roc_curve
import matplotlib.pyplot as plt
import seaborn as sns

# 1. Define parameter grid
```

```
param_grid = {
    'C': [0.01, 0.1, 1, 10],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear']
}

# 2. Set up Logistic Regression and GridSearch
log_reg = LogisticRegression(max_iter=1000)
grid_search = GridSearchCV(log_reg, param_grid, cv=3, scoring='f1_macro', verbose=1,
n_jobs=-1)

# 3. Fit to training data
grid_search.fit(X_train, y_train)

# 4. Best model
best_log_clf = grid_search.best_estimator_

# 5. Predictions
y_pred_log = best_log_clf.predict(X_test)
y_proba_log = best_log_clf.predict_proba(X_test)[:, 1]



import joblib
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score,
precision_recall_curve

# Save the trained Logistic Regression model
joblib.dump(best_log_clf, 'logistic_regression_model.pkl')

# Compute evaluation metrics again if needed
cm = confusion_matrix(y_test, y_pred_log)
roc_auc = roc_auc_score(y_test, y_proba_log)
precision, recall, thresholds = precision_recall_curve(y_test, y_proba_log)

# Save the evaluation results in a dictionary
results = {
    'model_name': 'Logistic Regression',
    'confusion_matrix': cm,
    'roc_auc': roc_auc,
    'precision': precision,
    'recall': recall,
    'thresholds': thresholds,
    'classification_report': classification_report(y_test, y_pred_log, output_dict=True)
}

# Save the results dictionary to a file
joblib.dump(results, 'logistic_regression_results.pkl')
```

```
# 1. Imports
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score,
precision_recall_curve, roc_curve
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization, Input
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam

# 2. Scale data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 3. Build MLP model
mlp_model = Sequential([
    Input(shape=(X_train.shape[1],)),  # Input layer
    Dense(256, activation='relu'),
    BatchNormalization(),
    Dropout(0.4),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')  # Binary output
])

# 4. Compile model
mlp_model.compile(
    optimizer=Adam(learning_rate=0.0005),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# 5. Early stopping
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

# 6. Train model
history = mlp_model.fit(
    X_train_scaled, y_train,
    epochs=50,
    batch_size=128,
    validation_split=0.2,
    callbacks=[early_stop],
```

```
    verbose=1
)

# 7. Predictions
mlp_proba = mlp_model.predict(X_test_scaled).flatten()
mlp_pred = (mlp_proba > 0.5).astype(int)

print("MLP Classification Report:\n", classification_report(y_test, mlp_pred))

import joblib

# 1. Save the MLP model
mlp_model.save("mlp_model.h5")

# 2. Save the scaler (used for preprocessing)
joblib.dump(scaler, "mlp_scaler.pkl")

# 3. Save evaluation metrics
mlp_results = {
    'model_name': 'MLP',
    'confusion_matrix': cm_mlp,
    'roc_auc': roc_auc_score(y_test, mlp_proba),
    'classification_report': classification_report(y_test, mlp_pred, output_dict=True),
    'precision': precision,
    'recall': recall,
    'thresholds': thresholds
}
joblib.dump(mlp_results, "mlp_results.pkl")



from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score,
precision_recall_curve, roc_curve
import matplotlib.pyplot as plt
import seaborn as sns


# Train Random Forest
rf_model = RandomForestClassifier(
    n_estimators=100,
    random_state=42,
    n_jobs=-1
)
rf_model.fit(X_train, y_train)

# Predictions
y_pred_rf = rf_model.predict(X_test)
y_proba_rf = rf_model.predict_proba(X_test)[:, 1]
```

```
# Evaluation
print("\nRandom Forest Classification Report:")
print(classification_report(y_test, y_pred_rf))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))
print("ROC AUC Score:", roc_auc_score(y_test, y_proba_rf))



# Save the trained model
joblib.dump(rf_model, 'random_forest_model.pkl')

# Recalculate precision, recall, and thresholds for saving
precision_rf, recall_rf, thresholds_rf = precision_recall_curve(y_test, y_proba_rf)

# Save evaluation results
rf_results = {
    'model_name': 'Random Forest',
    'confusion_matrix': cm_rf,
    'roc_auc': roc_auc_score(y_test, y_proba_rf),
    'classification_report': classification_report(y_test, y_pred_rf, output_dict=True),
    'precision': precision_rf,
    'recall': recall_rf,
    'thresholds': thresholds_rf
}

# Save the evaluation dictionary
joblib.dump(rf_results, 'random_forest_results.pkl')


# Create a simple XGBoost classifier
xgb = XGBClassifier(
    eval_metric='logloss',     # evaluation metric
    random_state=42            # for reproducibility
)

# Train the model
xgb.fit(X_train, y_train)

# Predict on test set
y_pred = xgb.predict(X_test)
y_proba = xgb.predict_proba(X_test)[:, 1]



# Training data
X_to_use, y_to_use = X_train, y_train

# Define parameter grid
param_grid = {
```

```
    'n_estimators': [100, 200],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.8, 1],
    'colsample_bytree': [0.8, 1],
    'gamma': [0, 1],
    'scale_pos_weight': [1, 2]
}

xgb = XGBClassifier(
    eval_metric='logloss',
    random_state=42,
    n_jobs=-1
)

# RandomizedSearchCV
xgb_search = RandomizedSearchCV(
    estimator=xgb,
    param_distributions=param_grid,
    n_iter=10,
    scoring='f1_macro',
    cv=3,
    verbose=1,
    random_state=42,
    n_jobs=-1
)

# Train model
xgb_search.fit(X_to_use, y_to_use)
best_xgb = xgb_search.best_estimator_

# Predict probabilities
y_proba = best_xgb.predict_proba(X_test)[:, 1]

# Set your custom threshold
custom_threshold = 0.4
# Predict using custom threshold
y_pred_custom = (y_proba > custom_threshold).astype(int)


import joblib

# Save the trained model
joblib.dump(best_xgb, 'xgboost_model.pkl')
print("Model saved as 'xgboost_model.pkl'")

xgb_results = {
    'best_params': xgb_search.best_params_,
    'confusion_matrix': cm,
    'roc_auc': roc_auc_score(y_test, y_proba),
```

```
    'classification_report': classification_report(y_test, y_pred_custom, output_dict=True),
    'precision': precision,
    'recall': recall,
    'thresholds': thresholds
}

# Save the evaluation results
joblib.dump(xgb_results, 'xgboost_results.pkl')
print(" 📊 Evaluation results saved as 'xgboost_results.pkl'")



# Load the model
loaded_model = joblib.load('xgboost_model.pkl')
import pandas as pd

# === Sample Input 2 ===
sample_input_2 = {
    'Number_of_Casualties': 3,
    'Number_of_Vehicles': 1,
    'Hour': 2,
    'Month': 1,
    'Year': 2023,
    'Season': 'Winter',
    'Time_of_Day': 'Late_Night',
    'Urban_Rural_Label': 'Rural',
    'Speed_Bin': 'High_70',
    'Weather_Label': 'Raining',
    'Light_Label': 'Dark_No_Lighting',
    'Road_Surface_Label': 'Wet_or_Damp',
    'Road_Type_Label': 'Single_Carriageway',
    'Day_of_Week_Label': 'Sunday',
    'Age_Band_of_Driver': '26-35',
    'Sex_of_Driver': 'Male',
    'Vehicle_Type': 'Car',
    'Was_Vehicle_Left_Hand_Drive': 'No',
    'Age_of_Vehicle': 5,
    'Driver_IMD_Decile': 4.0,
    'Engine_Capacity_.CC.': 1600.0,
    'Vehicle_Reference': 12345
}

# Convert to DataFrame
sample_df2 = pd.DataFrame([sample_input_2])

#  Label encode
for col in label_encode_cols:
    if col in sample_df2.columns:
        sample_df2[col] = label_encoders[col].transform(sample_df2[col])
```

```
# One-hot encode
sample_df2 = pd.get_dummies(sample_df2, columns=onehot_encode_cols)

# Align columns to match training features
for col in X.columns:
    if col not in sample_df2.columns:
        sample_df2[col] = 0
sample_df2 = sample_df2[X.columns]

model = best_xgb
pred_class = model.predict(sample_df2)[0]
proba = model.predict_proba(sample_df2)[0][1]
label = "Severe/Fatal" if pred_class == 1 else "Slight"

# Step 5: Output
print("\n=== Manual Prediction Result ===")
print(f"Predicted Class: {pred_class} → {label}")
print(f"Probability of Severe/Fatal: {round(proba, 3)}")
```

```
sample_input_3 = {
        # Minor rain, well-lit street
        'Number_of_Casualties': 1,
        'Number_of_Vehicles': 2,
        'Hour': 16,
        'Month': 4,
        'Year': 2021,
        'Age_of_Vehicle': 5,
        'Driver_IMD_Decile': 6.0,
        'Engine_Capacity_.CC.': 1600.0,
        'Vehicle_Reference': 1,
        'Season': 'Spring',
        'Time_of_Day': 'Daytime',
        'Urban_Rural_Label': 'Urban',
        'Speed_Bin': 'Mid_50',
        'Weather_Label': 'Raining',
        'Light_Label': 'Daylight',
        'Road_Surface_Label': 'Wet_or_Damp',
        'Road_Type_Label': 'Single_Carriageway',
        'Day_of_Week_Label': 'Tuesday',
        'Age_Band_of_Driver': '26-35',
        'Sex_of_Driver': 'Male',
        'Vehicle_Type': 'Car',
        'Was_Vehicle_Left_Hand_Drive': 'No'
    }
sample_df = pd.DataFrame([sample_input_3])
```

```
# Apply label encoding
for col in label_encode_cols:
    sample_df[col] = label_encoders[col].transform(sample_df[col])

# One-hot encode
sample_df = pd.get_dummies(sample_df, columns=onehot_encode_cols)

# Align with training features
for col in X.columns:
    if col not in sample_df.columns:
        sample_df[col] = 0
sample_df = sample_df[X.columns]

# Predict
pred = best_xgb.predict(sample_df)[0]
proba = best_xgb.predict_proba(sample_df)[0][1]

label = "Severe/Fatal" if pred == 1 else "Slight"

print("Predicted Class:", pred, "→", label)
print("Probability of Severe/Fatal:", round(proba, 3))

sample_input_4 = {
        'Number_of_Casualties': 1,
        'Number_of_Vehicles': 1,
        'Hour': 5,
        'Month': 2,
        'Year': 2022,
        'Age_of_Vehicle': 8,
        'Driver_IMD_Decile': 7.0,
        'Engine_Capacity_.CC.': 1000.0,
        'Vehicle_Reference': 1,
        'Season': 'Winter',
        'Time_of_Day': 'Late_Night',
        'Urban_Rural_Label': 'Urban',
        'Speed_Bin': 'Low_30',
        'Weather_Label': 'Fine_No_Wind',
        'Light_Label': 'Dark_Street_Lights_Lit',
        'Road_Surface_Label': 'Dry',
        'Road_Type_Label': 'One_way_Street',
        'Day_of_Week_Label': 'Saturday',
        'Age_Band_of_Driver': '46-55',
        'Sex_of_Driver': 'Female',
        'Vehicle_Type': 'Car',
        'Was_Vehicle_Left_Hand_Drive': 'No'
    }
sample_df = pd.DataFrame([sample_input_4])

# Apply label encoding
for col in label_encode_cols:
```

```
    sample_df[col] = label_encoders[col].transform(sample_df[col])

# One-hot encode
sample_df = pd.get_dummies(sample_df, columns=onehot_encode_cols)

# Align with training features
for col in X.columns:
    if col not in sample_df.columns:
        sample_df[col] = 0
sample_df = sample_df[X.columns]

# Predict
pred = best_xgb.predict(sample_df)[0]
proba = best_xgb.predict_proba(sample_df)[0][1]

label = "Severe/Fatal" if pred == 1 else "Slight"

print("Predicted Class:", pred, "→", label)
print("Probability of Severe/Fatal:", round(proba, 3))

# Get feature importances from the trained model
importances = best_xgb.feature_importances_

# Match with feature names
feature_names = X_train.columns
feat_imp_df = pd.DataFrame({'Feature': feature_names, 'Importance': importances})

# Sort by importance
feat_imp_df = feat_imp_df.sort_values(by='Importance', ascending=False)

# Show top 15 features
print(feat_imp_df.head(15))
# Plot top 20 important features
plt.figure(figsize=(10, 6))
plt.barh(feat_imp_df['Feature'].head(20)[::-1], feat_imp_df['Importance'].head(20)[::-1])
plt.xlabel("Importance Score")
plt.title("Top 20 Feature Importances (XGBoost)")
plt.tight_layout()
plt.show()


# Get feature importances from the trained XGBoost model
feature_importances = pd.DataFrame({
    'Feature': X.columns,
    'Importance': best_xgb.feature_importances_
})

# Define relevant features
temporal_features = ['Hour', 'Month', 'Season', 'Time_of_Day', 'Day_of_Week_Label']
environmental_features = [
```

```
    'Urban_Rural_Label', 'Speed_Bin', 'Weather_Label',
    'Road_Surface_Label', 'Light_Label', 'Road_Type_Label'
]

# Match actual encoded feature names (like Weather_Label_Fine_No_Wind)
selected_features = [
    col for col in X.columns
    if any(feat in col for feat in temporal_features + environmental_features)
]

# Filter and sort
selected_df = feature_importances[feature_importances['Feature'].isin(selected_features)]
selected_df = selected_df.sort_values(by='Importance', ascending=False)  # DESCENDING


from sklearn.metrics import roc_auc_score, precision_score, recall_score, f1_score
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Collect predictions and probabilities from each model
models_info = {
    'Logistic Regression': {'y_pred': y_pred_log, 'y_proba': y_proba_log},
    'MLP': {'y_pred': mlp_pred, 'y_proba': mlp_proba},
    'Random Forest': {'y_pred': y_pred_rf, 'y_proba': y_proba_rf},
    'XGBoost': {'y_pred': y_pred_custom, 'y_proba': y_proba},
}

# Evaluate and store results
results = []
for name, data in models_info.items():
    prec = precision_score(y_test, data['y_pred'])
    rec = recall_score(y_test, data['y_pred'])
    f1 = f1_score(y_test, data['y_pred'])
    auc = roc_auc_score(y_test, data['y_proba'])

    results.append({
        'Model': name,
        'Precision': prec,
        'Recall': rec,
        'F1 Score': f1,
        'ROC AUC': auc
    })

# Convert to DataFrame and sort by ROC AUC
results_df = pd.DataFrame(results).sort_values(by='Recall', ascending=False)
print(results_df)

# Plot grouped bar chart
plt.figure(figsize=(10, 6))
```

```
sns.barplot(
    data=results_df.melt(id_vars='Model'),
    x='Model', y='value', hue='variable'
)
plt.title("Model Performance Comparison")
plt.ylabel("Score")
plt.ylim(0, 1)
plt.legend(loc='upper right', bbox_to_anchor=(1.15, 1))
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.savefig("model_performance_comparison.png")
plt.show()
```

Code for interface

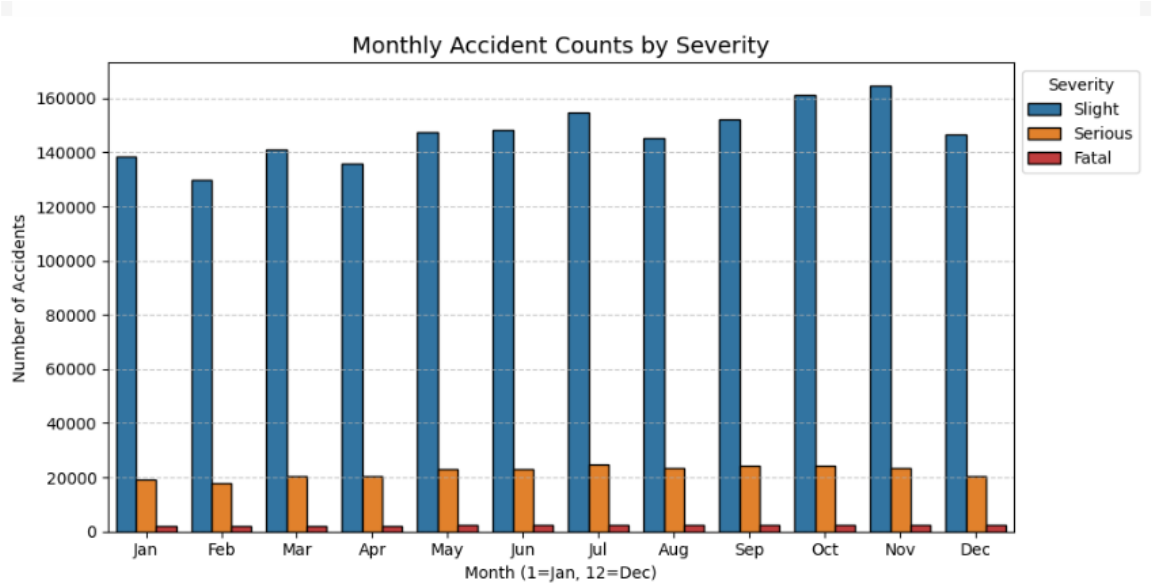
```
import joblib
import pandas as pd
import ipywidgets as widgets
from IPython.display import display, clear_output, HTML
import warnings
warnings.filterwarnings("ignore", category=UserWarning)

# === LOAD MODEL & ENCODERS ===
best_xgb = joblib.load("xgboost_model.pkl")
label_encoders = joblib.load("label_encoders.pkl")
onehot_encode_cols = joblib.load("onehot_cols.pkl")
X_columns = joblib.load("X_columns.pkl")
print("✅ Model, encoders, and column metadata loaded.")

# === OPTIONS ===
season_opts = ['Autumn', 'Spring', 'Summer', 'Winter']
time_opts = ['Daytime', 'Evening_Peak', 'Late_Night', 'Morning_Peak', 'Night']
urban_opts = ['Rural', 'Unknown', 'Urban']
speed_opts = ['High_70', 'Low_30', 'Mid_50']
weather_opts = ['Fine', 'Raining', 'Snowing_Ice', 'Fine+high winds', 'Other_Adverse',
'Unknown', 'Fog']
light_opts = ['Darkness_Lights_On', 'Daylight', 'Dark_No_Lighting', 'Darkness_Lights_Off',
'Unknown']
surface_opts = ['Dry', 'Wet_or_Damp', 'Snow_Ice', 'Flood', 'Unknown']
road_type_opts = ['Dual_Carriageway', 'Single_Carriageway', 'One_Way_Street',
'Roundabout', 'Slip_Road', 'Unknown']
day_opts = ['Wednesday', 'Thursday', 'Friday', 'Monday', 'Tuesday', 'Sunday', 'Saturday']
age_band_opts = ['36 - 45', '26 - 35', '46 - 55', '66 - 75', '16 - 20', '56 - 65', 'Unknown',
                 '21 - 25', 'Over 75', '11 - 15', '6 - 10', '0 - 5']
sex_opts = ['Male', 'Female', 'Unknown']
vehicle_type_opts = ['Bus or coach (17 or more pass seats)', 'Car', 'Motorcycle 125cc and
under',
```

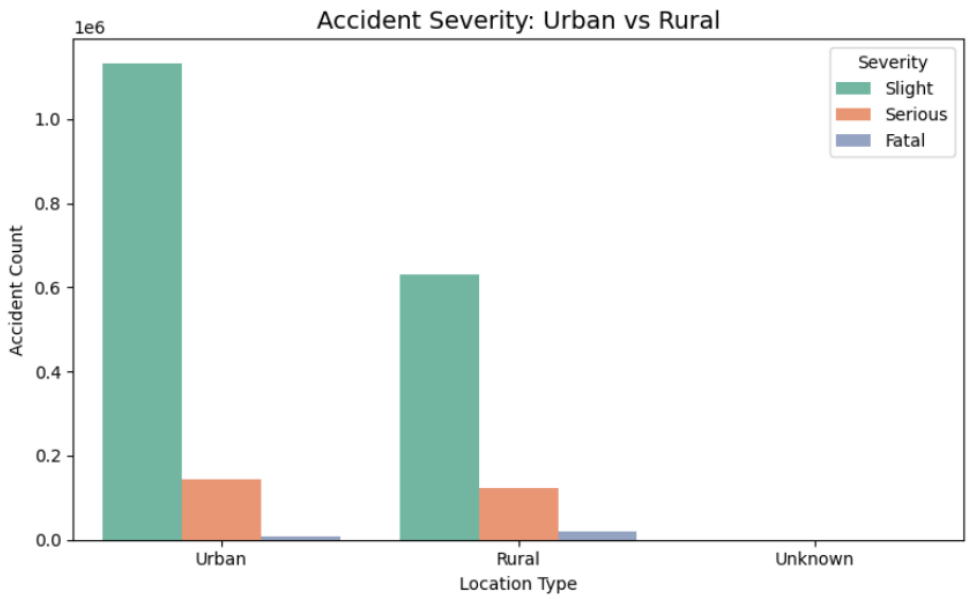
```
                'Motorcycle over 500cc','Motorcycle over 125cc and up to 500cc', 'Taxi/Private
hire car',
                'Goods 7.5 tonnes mgw and over','Van / Goods 3.5 tonnes mgw or under',
'Motorcycle 50cc and under',
                'Goods over 3.5t. and under 7.5t','Minibus (8 - 16 passenger seats)', 'Other
vehicle',
                'Agricultural vehicle', 'Pedal cycle', 'Tram','Ridden horse', 'Unknown',
                'Motorcycle - unknown cc', 'Mobility scooter','Goods vehicle - unknown weight',
'Electric motorcycle']
left_drive_opts = ['No', 'Yes', 'Unknown']

# === WIDGETS ===
casualties = widgets.IntSlider(description='Casualties', min=1, max=10, value=1)
vehicles = widgets.IntSlider(description='Vehicles', min=1, max=10, value=1)
hour = widgets.IntSlider(description='Hour', min=0, max=23, value=12)
month = widgets.Dropdown(description='Month', options=list(range(1, 13)), value=6)
year = widgets.Dropdown(description='Year', options=list(range(2005, 2026)), value=2023)
day = widgets.Dropdown(description='Day of Week', options=day_opts)
road_type = widgets.Dropdown(description='Road Type', options=road_type_opts)
speed_bin = widgets.Dropdown(description='Speed Bin', options=speed_opts)
urban = widgets.Dropdown(description='Urban/Rural', options=urban_opts)
weather = widgets.Dropdown(description='Weather', options=weather_opts)
light = widgets.Dropdown(description='Lighting', options=light_opts)
surface = widgets.Dropdown(description='Surface', options=surface_opts)
season = widgets.Dropdown(description='Season', options=season_opts)
time_of_day = widgets.Dropdown(description='Time of Day', options=time_opts)
age_band = widgets.Dropdown(description='Driver Age', options=age_band_opts)
sex = widgets.Dropdown(description='Sex of Driver', options=sex_opts)
imd = widgets.FloatSlider(description='IMD Decile', min=1, max=10, step=1, value=5.0)
vehicle_type = widgets.Dropdown(description='Vehicle Type', options=vehicle_type_opts)
left_drive = widgets.Dropdown(description='Left-Hand Drive', options=left_drive_opts)
vehicle_age = widgets.IntSlider(description='Vehicle Age', min=0, max=30, value=5)
engine_cc = widgets.IntSlider(description='Engine CC', min=500, max=5000, step=100,
value=1600)

predict_btn = widgets.Button(description=" 🚦 Predict", button_style='success')
output_box = widgets.Output()

# === LAYOUT ===
form_widgets = widgets.VBox([
    casualties, vehicles, hour, month, year, day,
    road_type, speed_bin, urban, weather, light, surface,
    season, time_of_day, age_band, sex, imd,
    vehicle_type, left_drive, vehicle_age, engine_cc,
    predict_btn, output_box
])

# === CALLBACK ===
def on_predict_clicked(b):
    input_data = {
```

```
    'Number_of_Casualties': casualties.value,
    'Number_of_Vehicles': vehicles.value,
    'Hour': hour.value,
    'Month': month.value,
    'Year': year.value,
    'Season': season.value,
    'Time_of_Day': time_of_day.value,
    'Urban_Rural_Label': urban.value,
    'Speed_Bin': speed_bin.value,
    'Weather_Label': weather.value,
    'Light_Label': light.value,
    'Road_Surface_Label': surface.value,
    'Road_Type_Label': road_type.value,
    'Day_of_Week_Label': day.value,
    'Age_Band_of_Driver': age_band.value,
    'Sex_of_Driver': sex.value,
    'Vehicle_Type': vehicle_type.value,
    'Was_Vehicle_Left_Hand_Drive': left_drive.value,
    'Age_of_Vehicle': vehicle_age.value,
    'Driver_IMD_Decile': imd.value,
    'Engine_Capacity_.CC.': engine_cc.value,
    'Vehicle_Reference': 12345
}

df = pd.DataFrame([input_data])

# === Apply Encoding (same as training) ===
for col in label_encoders:
    if col in df.columns:
        df[col] = label_encoders[col].transform(df[col])

df = pd.get_dummies(df, columns=onehot_encode_cols)

# Add missing columns to match training data
for col in X_columns:
    if col not in df.columns:
        df[col] = 0
df = df[X_columns]

# === Prediction ===
proba = best_xgb.predict_proba(df)[0][1]
threshold = 0.6
label = "Severe/Fatal" if proba >= threshold else "Slight"
emoji = " 🔴 " if label == "Severe/Fatal" else " 🟢 "
color = "#ffdddd" if label == "Severe/Fatal" else "#ddffdd"

with output_box:
    clear_output()
    display(HTML(f"""
        <div style="
```

```
            border: 2px solid #333;
            border-radius: 10px;
            padding: 15px;
            background-color: {color};
            font-family: Arial, sans-serif;
            width: 420px;
        ">
            <h3 style="margin-top:0; color:#333;">📊 Prediction Summary</h3>
            <p style="font-size:18px; font-weight:bold;">
                Label: {emoji} <span style="color:{'red' if label=='Severe/Fatal' else
'green'};">{label}</span>
            </p>
            <p style="font-size:16px;">
                Probability of Severe/Fatal: <b>{round(proba * 100, 2)}%</b>
            </p>
        </div>
    """))

predict_btn.on_click(on_predict_clicked)

# === DISPLAY ===
display(HTML("<h2>🚧 Accident Severity Prediction Tool</h2>"))
display(form_widgets)
```

Eda Plots On Raw Dataset





Testing with Manual inputs

```
sample_input_4 = {
        'Number_of_Casualties': 1,
        'Number_of_Vehicles': 1,
        'Hour': 5,
        'Month': 2,
        'Year': 2017,
        'Age_of_Vehicle': 8,
        'Driver_IMD_Decile': 7.0,
        'Engine_Capacity_.CC.': 1000.0,
        'Vehicle_Reference': 1,
        'Season': 'Winter',
        'Time_of_Day': 'Late_Night',
        'Urban_Rural_Label': 'Urban',
        'Speed_Bin': 'Low_30',
        'Weather_Label': 'Fine_No_Wind',
        'Light_Label': 'Dark_Street_Lights_Lit',
        'Road_Surface_Label': 'Dry',
        'Road_Type_Label': 'One_way_Street',
        'Day_of_Week_Label': 'Saturday',
        'Age_Band_of_Driver': '46-55',
        'Sex_of_Driver': 'Female',
        'Vehicle_Type': 'Car',
        'Was_Vehicle_Left_Hand_Drive': 'No'
    }
sample_df = pd.DataFrame([sample_input_4])

# Apply label encoding
for col in label_encode_cols:
    sample_df[col] = label_encoders[col].transform(sample_df[col])

# One-hot encode
sample_df = pd.get_dummies(sample_df, columns=onehot_encode_cols)

# Align with training features
for col in X.columns:
    if col not in sample_df.columns:
        sample_df[col] = 0
sample_df = sample_df[X.columns]

# Predict
pred = best_xgb.predict(sample_df)[0]
proba = best_xgb.predict_proba(sample_df)[0][1]

label = "Severe/Fatal" if pred == 1 else "Slight"

print("Predicted Class:", pred, "→", label)
print("Probability of Severe/Fatal:", round(proba, 3))
```

Predicted Class: 1 → Severe/Fatal
Probability of Severe/Fatal: 0.641