Candidate Number : 838651

Big Data Coursework - UK Government (Department for Transport) Road Safety Dataset - Individual Part

In the group part, we loaded, cleaned and preprocessed the data. In my individual part, I will keep on cleaning and transforming the data. Then, I will create several models to predict the accident severity based on independent variables, I have selected in the meantime.

Table of Contents

Importing Libraries and Preparing Environment

In [1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time

sns.set(style="darkgrid")
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, con
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
```

INTRODUCTION:

1. Business Objective

Road accident statistics are vital for national governments in developing and monitoring road safety policies. They also support a wide range of research aimed at reducing risks to road users. This research can be government-funded, sponsored by independent entities, or conducted by organizations and individuals independently. Researchers and consultancies require access to this data to analyze the risks and consequences of accidents and to predict the potential impact of policy changes, regulations, and safety measures.

Expected Results

Improved Safety Outcomes: Stakeholders can anticipate a reduction in the severity of road accidents by proactively identifying and addressing high-risk factors.

Efficiency Gains: Predictive models help stakeholders allocate resources more effectively, resulting in cost savings in emergency response, infrastructure maintenance, and insurance claims management.

Data-Driven Decision Making: Utilizing predictive analytics enables stakeholders to make informed, data-driven decisions, leading to better policy development, risk management practices, and overall enhancements in road safety.

Business-Specific Problem:

Forecasting the Severity of Road Traffic Accidents to Enhance Resource Allocation and Emergency Response for Various Stakeholders.

The aim is to develop a predictive model that can accurately evaluate the severity of road traffic accidents by considering various factors such as the time of the accident, weather conditions, location, road type, vehicle type, and driver demographics. This model is intended to assist emergency services and public safety organizations in optimizing resource allocation and enhancing emergency response times by predicting whether an accident will result in slight, serious, or fatal injuries. Additionally, this prediction can be beneficial for insurance companies in risk assessment and premium pricing, as well as for policymakers in implementing targeted road safety measures.

Modeling Task:

The modeling task is to develop a predictive model that can accurately classify scenarios into two categories: accident and no accident. This is a binary classification task where the target variable is accident occurrence (accident or no accident). The dataset contains various features related to vehicle movements, environmental conditions, and road characteristics, which will be used to train and evaluate the predictive model.

Data Loading and Selection

Data Loading

In [2]:
```python
test = pd.read_excel('test_data_filtered.xlsx')
train = pd.read_excel('train_data_filtered..xlsx')
```

In [3]:
```python
import pandas as pd

# Define the significant columns defined in group assignment
selected_columns = [
    'accident_severity', 'day_of_week', 'road_type', 'light_conditions
    'weather_conditions', 'road_surface_conditions', 'urban_or_rural_a
    'did_police_officer_attend_scene_of_accident',
    'vehicle_type', 'vehicle_manoeuvre', 'sex_of_driver', 'casualty_cl
    'sex_of_casualty', 'casualty_severity', 'casualty_type','longitude
    'speed_limit', 'month', 'hour', 'is_weekend', 'day_of_week_Monday'
    'day_of_week_Saturday', 'day_of_week_Sunday', 'road_type_Single ca
    'light_conditions_Light', 'weather_conditions_Fine Weather',
]

# Filter the data based on selected columns (ensure no missing columns
train_data = train[[col for col in selected_columns if col in train.co
test_data = test[[col for col in selected_columns if col in test.colum

# Save the filtered data to new Excel files
train_data.to_excel('train_dfs.xlsx', index=False)
test_data.to_excel('test_dfs.xlsx', index=False)

# Filter the data based on selected columns
train_data = pd.read_excel("train_dfs.xlsx")
test_data = pd.read_excel("test_dfs.xlsx")
```

In [4]:
```python
# Define the target variable, Y
y_train = train_data['accident_severity']

# Define the feature set, X, by dropping the target variable from the
X_train = train_data.drop('accident_severity', axis=1)
```

In [5]:
```python
# Define the target variable, Y
y_test = test_data['accident_severity']

# Define the feature set, X, by dropping the target variable from the
X_test = test_data.drop('accident_severity', axis=1)
```

2. Baseline Method

The baseline model in your coursework is defined using a Dummy Classifier with the “most_frequent” strategy. This model predicts the most frequent class in the training data. Here are the key points:

Model Initialization: DummyClassifier(strategy="most_frequent")

Training: The model is trained on the training dataset.

Evaluation: Precision, recall, and F-score are calculated to assess the model’s performance.

This baseline provides a reference point to compare the performance of more complex models

I. Relevance of Different Predictive Modelling Techniques to Specific Business Objectives

Predictive Modelling Techniques:

Logistic Regression:

Objective: Identify key factors affecting accident severity and predict the likelihood of severe accidents.

Relevance: Provides insights into which features (e.g., weather conditions, road type) most influence accident severity. It’s useful for stakeholders to understand the relationship between variables and outcomes.

Decision Trees and Random Forests:

Objective: Develop more accurate predictive models to improve road safety initiatives.

Relevance: Handle both numerical and categorical data well, provide feature importance, and can model complex interactions. Random Forests help reduce overfitting and improve prediction accuracy.

Support Vector Machines (SVM):

Objective: High-accuracy classification of accident severity.

Relevance: Effective for cases with high-dimensional data and can handle non-linear relationships. Useful when precision in classification is critical.

Neural Networks:

Objective: Capture complex patterns and interactions in accident data.

Relevance: Suitable for large datasets with many features. Can model non-linear relationships and interactions between features. May require significant computational resources.

K-Nearest Neighbors (KNN):

Objective: Simple and interpretable model for predicting accident severity.

Relevance: Easy to implement and understand. Suitable for small to medium-sized datasets. Less effective with large datasets due to computational complexity.

In [6]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, roc_auc_
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
```

In [7]:
```
# Encoding categorical variables
# importing library
from sklearn.preprocessing import LabelEncoder

# Encode independent categorical values
label_encoders = {}
for column in X_train.select_dtypes(include=['object']).columns:
    label_encoders[column] = LabelEncoder()
    X_train[column] = label_encoders[column].fit_transform(X_train[col
    X_test[column] = label_encoders[column].transform(X_test[column])

# Encode dependent(target) variable
y_train_encoded = LabelEncoder().fit_transform(y_train.values.ravel())
y_test_encoded = LabelEncoder().fit_transform(y_test.values.ravel())
```

In [8]:
```
from sklearn.dummy import DummyClassifier
from sklearn.metrics import precision_recall_fscore_support

dummy_clf = DummyClassifier(strategy="most_frequent")
dummy_clf.fit(X_train, y_train_encoded)
yhat = dummy_clf.predict(X_train)

p, r, f, s = precision_recall_fscore_support(y_train_encoded, yhat, av
print("Baseline:")
print(f"Precision: {p:.3f}")
print(f"Recall: {r:.3f}")
print(f"F score: {f:.3f}")
```

```
Baseline:
Precision: 0.395
Recall: 0.500
F score: 0.442
```

Logistic Regression

Categorical variables can be classified using logistic regression method which can provide us with Accuracy, Precision, recall and f1-score values

Logistic Regression is a statistical model used to estimate the probability of an event occurring based on a set of independent variables. It’s commonly used for classification and predictive analytics.

Definition:

It models the log-odds of an event as a linear combination of one or more independent variables.

Uses:

In [9]:
```python
# Define the target variable
start = time.time()

# For simplicity, let's handle any possible categorical variables usin
X_train = pd.get_dummies(X_train)


# Split data into training and test sets
X_train_split, X_test_split, y_train_encoded_split, y_test_encoded_spl

# Initialize and train the Logistic Regression model
model = LogisticRegression(max_iter=100)  # Increased max_iter for con
model.fit(X_train_split, y_train_encoded_split)


# Predict on the validation set
y_pred = model.predict(X_test_split)

# Evaluate the model
accuracy = accuracy_score(y_test_encoded_split, y_pred)
print("Accuracy on validation set:", accuracy)
print(classification_report(y_test_encoded_split, y_pred))
```

```
Accuracy on validation set: 0.7867783985102421
              precision    recall  f1-score   support

           0       0.00      0.00      0.00      1370
           1       0.79      1.00      0.88      5074

    accuracy                           0.79      6444
   macro avg       0.39      0.50      0.44      6444
weighted avg       0.62      0.79      0.69      6444
```

Decision Tree

A decision tree is a type of flowchart that helps in making decisions by breaking down complex data into simpler parts. Here are some key points:

(i) Definition:

A decision tree is a non-parametric supervised learning algorithm used for both classification and regression tasks. It consists of nodes representing decisions, chance events, and outcomes.

(ii) Structure:

It starts with a root node, branches into internal nodes (decision nodes), and ends with leaf nodes (outcomes).

(iii) Uses:

Decision trees are used in data analytics and machine learning for prediction analysis, data classification, and regression1. They are also applied in fields like engineering, civil planning, law, and business

In [10]:
```python
# Decision Tree
# Initialize the Decision Tree model
tree_model = DecisionTreeClassifier(random_state=42)

# Train the model
tree_model.fit(X_train_split, y_train_encoded_split)

# Predict on the validation set
y_pred_tree = tree_model.predict(X_test_split)

# Evaluate the model
accuracy_tree = accuracy_score(y_test_encoded_split, y_pred_tree)
print("Accuracy on validation set with Decision Tree:", accuracy_tree)
print(classification_report(y_test_encoded_split, y_pred_tree))
```

```
Accuracy on validation set with Decision Tree: 0.6793916821849783
              precision    recall  f1-score   support

           0       0.27      0.29      0.28      1370
           1       0.80      0.78      0.79      5074

    accuracy                           0.68      6444
   macro avg       0.53      0.54      0.54      6444
weighted avg       0.69      0.68      0.68      6444
```

Random Forest

A Random Forest Classifier is a powerful machine learning algorithm used for both classification and regression tasks. Here are some key points:

Definition:

It consists of multiple decision trees, each trained on different subsets of the data. The final prediction is made by aggregating the results of these trees, either by majority voting (for classification) or averaging (for regression).

Advantages:

It reduces the risk of overfitting, handles large and complex datasets well, and provides insights into feature importance.

Applications:

Widely used in various fields like banking, healthcare, and retail for tasks such as risk assessment, disease prediction, and customer segmentation.

In [11]:
```python
# Initialize the Random Forest model
random_forest_model = RandomForestClassifier(n_estimators=100, random_

# Train the model
random_forest_model.fit(X_train_split, y_train_encoded_split)

# Predict on the validation set
y_pred_rf = random_forest_model.predict(X_test_split)

# Evaluate the model
accuracy_rf = accuracy_score(y_test_encoded_split, y_pred_rf)
print("Accuracy on validation set with Random Forest:", accuracy_rf)
print(classification_report(y_test_encoded_split, y_pred_rf))
```

```
Accuracy on validation set with Random Forest: 0.771415270018622
              precision    recall  f1-score   support

           0       0.31      0.06      0.10      1370
           1       0.79      0.96      0.87      5074

    accuracy                           0.77      6444
   macro avg       0.55      0.51      0.49      6444
weighted avg       0.69      0.77      0.71      6444
```

KNN Model

In [12]:
```python
from sklearn.neighbors import KNeighborsClassifier

# Initialize and train the KNN model
knn_model = KNeighborsClassifier()  # Increased max_iter for convergen
knn_model.fit(X_train_split, y_train_encoded_split)


# Predict on the validation set
y_pred = knn_model.predict(X_test_split)

# Evaluate the model
accuracy_knn = accuracy_score(y_test_encoded_split, y_pred)
print("Accuracy on validation set:", accuracy_knn)
print(classification_report(y_test_encoded_split, y_pred))
```

```
Accuracy on validation set: 0.7545003103662321
              precision    recall  f1-score   support

           0       0.30      0.12      0.17      1370
           1       0.80      0.93      0.86      5074

    accuracy                           0.75      6444
   macro avg       0.55      0.52      0.51      6444
weighted avg       0.69      0.75      0.71      6444
```

Here’s a summary of the baseline models and their performance metrics:

Dummy Classifier: This model predicts the most frequent class. It serves as a baseline to compare other models. The precision, recall, and F-score are around 0.395, 0.500, and 0.442, respectively.

Logistic Regression: This model uses logistic regression to classify the data. It achieved an accuracy of 0.787 on the validation set, with a high recall for the positive class but low precision for the negative class.

Decision Tree: This model uses a decision tree classifier. It achieved an accuracy of 0.679 on the validation set, with balanced precision and recall for both classes.

Random Forest: This model uses a random forest classifier. It achieved an accuracy of 0.771 on the validation set, with high recall for the positive class but low precision for the negative class.

These models help in understanding the performance and effectiveness of different classification techniques.

3. Feature Selection (Optional)

In [13]:
```python
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier

# Initialize the model
model = RandomForestClassifier(n_estimators=200, random_state=42)

# Initialize RFE with the model
rfe = RFE(model, n_features_to_select=14)  # Adjust the number of feat

# Fit RFE
rfe = rfe.fit(X_train_split, y_train_encoded_split)

# Get the selected features
selected_features = X_train_split.columns[rfe.support_]
print("Selected features:", selected_features)
```

```
Selected features: Index(['longitude', 'latitude', 'number_of_vehicles', 'number_of_casualties',
       'speed_limit', 'month', 'hour', 'is_weekend', 'day_of_week_Monda
y',
       'day_of_week_Saturday', 'day_of_week_Sunday',
       'road_type_Single carriageway', 'light_conditions_Light',
       'weather_conditions_Fine Weather'],
      dtype='object')
```

In [14]:
```python
# Reduce training and test sets to selected features
X_train_rfe = rfe.transform(X_train_split)
X_test_rfe = rfe.transform(X_test_split)
```

In [15]:
```python
# Function to evaluate models
def evaluate_model(model, X_train_split, y_train_encoded_split, X_test_
    model.fit(X_train_split, y_train_encoded_split)
    predictions = model.predict(X_test_split)
    accuracy = accuracy_score(y_test_encoded_split, predictions)
    return accuracy

# Train and evaluate Decision Tree and Random Forest with top n featur
dtree_model = DecisionTreeClassifier()
rf_model = RandomForestClassifier()
knn_model = KNeighborsClassifier()

# Using top n features based on feature importance
dtree_accuracy = evaluate_model(dtree_model, X_train_split, y_train_en
rf_accuracy = evaluate_model(rf_model, X_train_split, y_train_encoded_
knn_accuracy = evaluate_model(knn_model, X_train_split, y_train_encode

# Using features selected by RFE
dtree_accuracy_rfe = evaluate_model(dtree_model, X_train_rfe, y_train_
rf_accuracy_rfe = evaluate_model(rf_model, X_train_rfe, y_train_encode
knn_accuracy_rfe = evaluate_model(knn_model, X_train_rfe, y_train_enco

print(f"Decision Tree Accuracy: {dtree_accuracy:.2f}")
print(f"Random Forest Accuracy: {rf_accuracy:.2f}")
print(f"KNN Accuracy: {knn_accuracy:.2f}")
print(f"Decision Tree Accuracy with RFE Features: {dtree_accuracy_rfe:
print(f"Random Forest Accuracy with RFE Features: {rf_accuracy_rfe:.2f
print(f"KNN with RFE Features: {knn_accuracy_rfe:.2f}")
```

```
Decision Tree Accuracy: 0.68
Random Forest Accuracy: 0.77
KNN Accuracy: 0.75
Decision Tree Accuracy with RFE Features: 0.68
Random Forest Accuracy with RFE Features: 0.77
KNN with RFE Features: 0.75
```

Here’s a summary of the feature selection and future code progress for Logistic Regression, Decision Tree, and Random Forest Classifier:

Feature Selection:

RFE (Recursive Feature Elimination) was used with a Random Forest model to select the top 10 features: longitude, latitude, number_of_vehicles, number_of_casualties, speed_limit, month, hour, is_weekend, day_of_week_Monday, light_conditions_Light.

(a) Logistic Regression:

Hyperparameter Tuning: GridSearchCV was used to find the best parameters (C: 0.1, penalty: ‘l1’).

Future Code: Continue refining the model with the best parameters and evaluate its performance.

(b) Decision Tree:

Hyperparameter Tuning: GridSearchCV was used to optimize max_depth and min_samples_split.

Future Code: Implement the best parameters and assess the model’s accuracy and classification report.

(c) Random Forest:

4. Hyperparameter Tuning

Hyperparameter Tuning is the process of optimizing the parameters that control the learning process of a machine learning model. These parameters, known as hyperparameters, are set before training and can significantly impact the model’s performance. Here are the key points:

Purpose: To find the best combination of hyperparameters that maximize the model’s performance.

Methods: Common techniques include Grid Search, Random Search, and Bayesian Optimization.

Process: Involves defining a range of values for each hyperparameter and systematically testing different combinations to identify the optimal settings.

Outcome: Improved accuracy, precision, recall, and overall model effectiveness.

Logistic Regression

In [16]:
```python
from sklearn.model_selection import GridSearchCV
from datetime import datetime, timedelta
```

In [17]:
```python
param_grid = {
    'strategy': ['stratified', 'most_frequent', 'prior', 'uniform']
}

# Setting up Grid Search for Logistic Regression
grid_dummy = GridSearchCV(dummy_clf, param_grid, cv=5, verbose=2, n_jo
grid_dummy.fit(X_train_split, y_train_encoded_split)

print("Best parameters for Dummy Classifier:", grid_dummy.best_params_
print("Best cross-validation score: {:.2f}".format(grid_dummy.best_sco
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
Best parameters for Dummy Classifier: {'strategy': 'most_frequent'}
Best cross-validation score: 0.79
```

In [18]:
```python
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Logistic Regression model
logistic = LogisticRegression(solver='liblinear', random_state=42)

# Parameter grid for Logistic Regression
param_grid_logistic = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2']
}

# Setting up Grid Search for Logistic Regression
grid_logistic = GridSearchCV(logistic, param_grid_logistic, cv=5, verb
grid_logistic.fit(X_train_split, y_train_encoded_split)

print("Best parameters for Logistic Regression:", grid_logistic.best_p
print("Best cross-validation score: {:.2f}".format(grid_logistic.best_


# Predict on the validation set using the best estimator found by the g
y_pred_logistic = grid_logistic.predict(X_test_split)  # Correcting va

# Evaluate the model
accuracy = accuracy_score(y_test_encoded_split, y_pred_logistic)
print("Accuracy on validation set:", accuracy)
print(classification_report(y_test_encoded_split, y_pred_logistic))
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
Best parameters for Logistic Regression: {'C': 0.1, 'penalty': 'l1'}
Best cross-validation score: 0.79
Accuracy on validation set: 0.787243947858473
              precision    recall  f1-score   support

           0       0.00      0.00      0.00      1370
           1       0.79      1.00      0.88      5074

    accuracy                           0.79      6444
   macro avg       0.39      0.50      0.44      6444
weighted avg       0.62      0.79      0.69      6444
```

Hyperparameter tuning for logistic regression involves optimizing parameters to improve model performance. Here are some key points:

Regularization (penalty): Controls overfitting by adding a penalty to the loss function. Common options are ‘l1’ (Lasso), ‘l2’ (Ridge), and ‘elasticnet’.

Regularization strength (C): Inverse of regularization strength; smaller values specify stronger regularization. Typical values to try are 0.001, 0.01, 0.1, 1, 10, and 100.

Solver: Algorithm used for optimization. Options include ‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, and 'saga’.

Using GridSearchCV can help find the best combination of these hyperparameters for the dataset.

Decision Tree

Hyperparameter tuning is crucial for optimizing decision tree models. Here are some key points:

Importance: Tuning hyperparameters can significantly improve a model's accuracy, generalization, and robustness.

Methods: Common techniques include grid search, random search, and Bayesian optimization.

Parameters: Key hyperparameters to tune include max_depth, min_samples_split, and min_samples_lea1.

In [19]:

```python
start = time.time()

from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(random_state=7)

hp_grid = {
    'max_depth': [5, 10, 15, 20, 25, 30, 35, 40],
    'min_samples_split': [5, 10, 15, 20, 25, 30, 35],
}

grid_search = GridSearchCV(dt, hp_grid, cv=5,
                           scoring='f1_macro',
                           return_train_score=True, verbose=2)

grid_search.fit(X_train_split, y_train_encoded_split)

print("Execution time HH:MM:SS:", timedelta(seconds=time.time() - star
```

```
me=   0.1s
[CV] END ................max_depth=10, min_samples_split=35; total ti
me=   0.1s
[CV] END ................max_depth=10, min_samples_split=35; total ti
me=   0.1s
[CV] END ................max_depth=10, min_samples_split=35; total ti
me=   0.1s
[CV] END ................max_depth=10, min_samples_split=35; total ti
me=   0.1s
[CV] END ................max_depth=10, min_samples_split=35; total ti
me=   0.1s
[CV] END .................max_depth=15, min_samples_split=5; total ti
me=   0.1s
[CV] END .................max_depth=15, min_samples_split=5; total ti
me=   0.1s
[CV] END .................max_depth=15, min_samples_split=5; total ti
me=   0.1s
[CV] END .................max_depth=15, min_samples_split=5; total ti
me=   0.1s
[CV] END .................max_depth=15, min_samples_split=5; total ti
```

In [20]:
```
grid_search.best_estimator_
```

Out[20]:
```
▾ DecisionTreeClassifier
DecisionTreeClassifier(max_depth=20, min_samples_split=5, random_state=
7)
```

In [21]:
```
cv_results = pd.DataFrame(grid_search.cv_results_)[['params', 'mean_tr
cv_results["diff, %"] = 100*(cv_results["mean_train_score"]-cv_results
                                                                    )/cv_results["mea

pd.set_option('display.max_colwidth', 100)
cv_results.sort_values('mean_test_score', ascending=False)
```

Out[21]:

	params	mean_train_score	mean_test_score	diff, %
21	{'max_depth': 20, 'min_samples_split': 5}	0.850750	0.532765	37.377040
45	{'max_depth': 35, 'min_samples_split': 20}	0.780797	0.532139	31.846648
49	{'max_depth': 40, 'min_samples_split': 5}	0.936245	0.532001	43.177165
42	{'max_depth': 35, 'min_samples_split': 5}	0.933747	0.531952	43.030387
52	{'max_depth': 40, 'min_samples_split': 20}	0.781778	0.531778	31.978447
51	{'max_depth': 40, 'min_samples_split': 15}	0.815089	0.531607	34.779288
38	{'max_depth': 30, 'min_samples_split': 20}	0.778122	0.531240	31.728018
37	{'max_depth': 30, 'min_samples_split': 15}	0.809823	0.530789	34.456175
50	{'max_depth': 40, 'min_samples_split': 10}	0.861206	0.530752	38.371030
46	{'max_depth': 35, 'min_samples_split': 25}	0.753926	0.530737	29.603544
44	{'max_depth': 35, 'min_samples_split': 15}	0.813686	0.530727	34.774943
29	{'max_depth': 25, 'min_samples_split': 10}	0.837906	0.530437	36.694848
36	{'max_depth': 30, 'min_samples_split': 10}	0.854595	0.530394	37.936225
53	{'max_depth': 40, 'min_samples_split': 25}	0.754300	0.530312	29.694829
43	{'max_depth': 35, 'min_samples_split': 10}	0.859472	0.530109	38.321555
35	{'max_depth': 30, 'min_samples_split': 5}	0.926020	0.529880	42.778784
39	{'max_depth': 30, 'min_samples_split': 25}	0.751413	0.529847	29.486519
54	{'max_depth': 40, 'min_samples_split': 30}	0.731968	0.529471	27.664682
31	{'max_depth': 25, 'min_samples_split': 20}	0.766214	0.529213	30.931449
47	{'max_depth': 35, 'min_samples_split': 30}	0.731495	0.529114	27.666859
22	{'max_depth': 20, 'min_samples_split': 10}	0.798615	0.529087	33.749384
30	{'max_depth': 25, 'min_samples_split': 15}	0.796509	0.528787	33.611935
23	{'max_depth': 20, 'min_samples_split': 15}	0.765213	0.528467	30.938664
48	{'max_depth': 35, 'min_samples_split': 35}	0.716479	0.528439	26.245035
24	{'max_depth': 20, 'min_samples_split': 20}	0.739460	0.528361	28.547627
40	{'max_depth': 30, 'min_samples_split': 30}	0.728908	0.528036	27.557919
55	{'max_depth': 40, 'min_samples_split': 35}	0.716498	0.527932	26.317720
28	{'max_depth': 25, 'min_samples_split': 5}	0.902891	0.527621	41.563159
41	{'max_depth': 30, 'min_samples_split': 35}	0.713517	0.527598	26.056728
32	{'max_depth': 25, 'min_samples_split': 25}	0.740698	0.527285	28.812406
33	{'max_depth': 25, 'min_samples_split': 30}	0.720538	0.526542	26.923716
25	{'max_depth': 20, 'min_samples_split': 25}	0.716017	0.526296	26.496710
34	{'max_depth': 25, 'min_samples_split': 35}	0.705783	0.526043	25.466813
26	{'max_depth': 20, 'min_samples_split': 30}	0.698951	0.524590	24.946105
27	{'max_depth': 20, 'min_samples_split': 35}	0.687771	0.524577	23.727959
14	{'max_depth': 15, 'min_samples_split': 5}	0.742424	0.518837	30.115865
15	{'max_depth': 15, 'min_samples_split': 10}	0.714985	0.518457	27.486974
16	{'max_depth': 15, 'min_samples_split': 15}	0.693839	0.517536	25.409833

	params	mean_train_score	mean_test_score	diff, %
17	{'max_depth': 15, 'min_samples_split': 20}	0.677804	0.517084	23.711846
18	{'max_depth': 15, 'min_samples_split': 25}	0.662513	0.516954	21.970782
20	{'max_depth': 15, 'min_samples_split': 35}	0.642836	0.515514	19.806303
19	{'max_depth': 15, 'min_samples_split': 30}	0.650700	0.514812	20.883379
13	{'max_depth': 10, 'min_samples_split': 35}	0.555262	0.490612	11.643268
11	{'max_depth': 10, 'min_samples_split': 25}	0.560363	0.490024	12.552431
12	{'max_depth': 10, 'min_samples_split': 30}	0.559332	0.489759	12.438570
10	{'max_depth': 10, 'min_samples_split': 20}	0.564782	0.489363	13.353797
7	{'max_depth': 10, 'min_samples_split': 5}	0.580656	0.488519	15.867655
9	{'max_depth': 10, 'min_samples_split': 15}	0.568814	0.488493	14.120780
8	{'max_depth': 10, 'min_samples_split': 10}	0.574549	0.488334	15.005652
2	{'max_depth': 5, 'min_samples_split': 15}	0.459920	0.451797	1.766174
1	{'max_depth': 5, 'min_samples_split': 10}	0.459920	0.451797	1.766174
0	{'max_depth': 5, 'min_samples_split': 5}	0.459979	0.451797	1.778653
3	{'max_depth': 5, 'min_samples_split': 20}	0.459690	0.451623	1.755041
4	{'max_depth': 5, 'min_samples_split': 25}	0.459727	0.451607	1.766151
5	{'max_depth': 5, 'min_samples_split': 30}	0.459658	0.451217	1.836382
6	{'max_depth': 5, 'min_samples_split': 35}	0.459640	0.451203	1.835474

In [22]:
```
grid_search.best_score_
```
Out[22]:
```
0.5327646689684522
```

The GridSearchCV best score of 0.5327 indicates that the model’s average performance across the cross-validation folds is 53.27%. This score is based on the chosen evaluation metric (likely accuracy or F1-score) and reflects how well the model generalizes to unseen data during the training process.

Random Forest

Hyperparameter tuning for a Random Forest model involves optimizing several key parameters to improve model performance. Here are some important hyperparameters to consider:

n_estimators: Number of trees in the forest. More trees can improve performance but also increase computation time.

max_depth: Maximum depth of each tree. Controls overfitting; deeper trees may overfit, while shallower trees may underfit.

min_samples_split: Minimum number of samples required to split an internal node. Higher values prevent overfitting.

max_features: Number of features to consider when looking for the best split. Options include “sqrt”, “log2”, or a specific number.

Using tools like GridSearchCV or RandomizedSearchCV can help find the best combination of these hyperparameters

In [23]:
```
start = time.time()

from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(random_state=7, max_depth=40, min_samples_

# specify the hyperparameters and their values
# 3 x 2 x 2 = 12 combinations in the grid
hp_grid = {
    'n_estimators': [100, 200, 500],
    'max_features': ["sqrt", 0.5],
    'max_samples': [None, 0.5],
}

grid_search_rf = GridSearchCV(rf, hp_grid, cv=5,
                              scoring='f1_macro',
                              return_train_score=True, verbose=2)

grid_search_rf.fit(X_train_split, y_train_encoded_split)

print("Execution time HH:MM:SS:", timedelta(seconds=time.time() - star
Fitting 5 folds for each of 12 candidates, totalling 60 fits
[CV] END max_features=sqrt, max_samples=None, n_estimators=100; total tim
e=   5.6s
[CV] END max_features=sqrt, max_samples=None, n_estimators=100; total tim
e=   5.3s
[CV] END max_features=sqrt, max_samples=None, n_estimators=100; total tim
e=   5.2s
[CV] END max_features=sqrt, max_samples=None, n_estimators=100; total tim
e=   5.5s
[CV] END max_features=sqrt, max_samples=None, n_estimators=100; total tim
e=   6.6s
[CV] END max_features=sqrt, max_samples=None, n_estimators=200; total tim
e=  11.2s
[CV] END max_features=sqrt, max_samples=None, n_estimators=200; total tim
e=  10.8s
[CV] END max_features=sqrt, max_samples=None, n_estimators=200; total tim
e=  10.9s
[CV] END max_features=sqrt, max_samples=None, n_estimators=200; total tim
e=  11.1s
[CV] END max_features=sqrt, max_samples=None, n_estimators=200; total tim
e=  11.2s
[CV] END max_features=sqrt, max_samples=None, n_estimators=500; total tim
e=  29.2s
[CV] END max_features=sqrt, max_samples=None, n_estimators=500; total tim
e=  27.3s
[CV] END max_features=sqrt, max_samples=None, n_estimators=500; total tim
e=  26.4s
[CV] END max_features=sqrt, max_samples=None, n_estimators=500; total tim
e=  27.5s
[CV] END max_features=sqrt, max_samples=None, n_estimators=500; total tim
e=  26.2s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=100; total time
=   3.1s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=100; total time
=   3.0s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=100; total time
=   3.0s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=100; total time
=   3.0s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=100; total time
=   3.0s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=200; total time
=   6.1s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=200; total time
=   6.0s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=200; total time
=   6.1s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=200; total time
=   6.0s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=200; total time
=   6.1s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=500; total time
=  15.4s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=500; total time
=  15.3s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=500; total time
=  15.7s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=500; total time
=  15.7s
[CV] END max_features=sqrt, max_samples=0.5, n_estimators=500; total time
=  15.5s
```

```
[CV] END max_features=0.5, max_samples=None, n_estimators=100; total time
=   9.8s
[CV] END max_features=0.5, max_samples=None, n_estimators=100; total time
=   9.3s
[CV] END max_features=0.5, max_samples=None, n_estimators=100; total time
=   9.8s
[CV] END max_features=0.5, max_samples=None, n_estimators=100; total time
=   9.5s
[CV] END max_features=0.5, max_samples=None, n_estimators=100; total time
=   9.5s
[CV] END max_features=0.5, max_samples=None, n_estimators=200; total time
=  19.1s
[CV] END max_features=0.5, max_samples=None, n_estimators=200; total time
=  18.8s
[CV] END max_features=0.5, max_samples=None, n_estimators=200; total time
=  19.4s
[CV] END max_features=0.5, max_samples=None, n_estimators=200; total time
=  19.3s
[CV] END max_features=0.5, max_samples=None, n_estimators=200; total time
=  19.3s
[CV] END max_features=0.5, max_samples=None, n_estimators=500; total time
=  48.4s
[CV] END max_features=0.5, max_samples=None, n_estimators=500; total time
=  48.0s
[CV] END max_features=0.5, max_samples=None, n_estimators=500; total time
=  50.5s
[CV] END max_features=0.5, max_samples=None, n_estimators=500; total time
=  55.4s
[CV] END max_features=0.5, max_samples=None, n_estimators=500; total time
= 1.0min
[CV] END max_features=0.5, max_samples=0.5, n_estimators=100; total time=
7.4s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=100; total time=
6.9s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=100; total time=
6.6s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=100; total time=
6.9s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=100; total time=
7.1s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=200; total time=
14.7s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=200; total time=
14.5s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=200; total time=
14.2s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=200; total time=
14.4s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=200; total time=
14.2s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=500; total time=
35.4s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=500; total time=
34.5s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=500; total time=
36.0s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=500; total time=
34.7s
[CV] END max_features=0.5, max_samples=0.5, n_estimators=500; total time=
34.8s
Execution time HH:MM:SS: 0:19:04.339504
```

In [24]:
```
grid_search_rf.best_estimator_
```

Out[24]:
```
▾                          RandomForestClassifier

RandomForestClassifier(max_depth=40, max_features=0.5, min_samples_split
=5,
                       n_estimators=200, random_state=7)
```

In [25]:
```
cv_results = pd.DataFrame(grid_search_rf.cv_results_)[['params', 'mean_
cv_results["diff, %"] = 100*(cv_results["mean_train_score"]-cv_results
                                                          )/cv_results["mea

pd.set_option('display.max_colwidth', 100)
cv_results.sort_values('mean_test_score', ascending=False)
```

Out[25]:

	params	mean_train_score	mean_test_score	diff, %
7	{'max_features': 0.5, 'max_samples': None, 'n_estimators': 200}	0.978212	0.489249	49.985390
6	{'max_features': 0.5, 'max_samples': None, 'n_estimators': 100}	0.973613	0.488285	49.848155
8	{'max_features': 0.5, 'max_samples': None, 'n_estimators': 500}	0.981968	0.485168	50.592286
0	{'max_features': 'sqrt', 'max_samples': None, 'n_estimators': 100}	0.925414	0.476155	48.546833
1	{'max_features': 'sqrt', 'max_samples': None, 'n_estimators': 200}	0.929401	0.475048	48.886690
2	{'max_features': 'sqrt', 'max_samples': None, 'n_estimators': 500}	0.931786	0.474921	49.031077
9	{'max_features': 0.5, 'max_samples': 0.5, 'n_estimators': 100}	0.767415	0.474347	38.189003
10	{'max_features': 0.5, 'max_samples': 0.5, 'n_estimators': 200}	0.766933	0.471382	38.536674
11	{'max_features': 0.5, 'max_samples': 0.5, 'n_estimators': 500}	0.764405	0.470415	38.459950
3	{'max_features': 'sqrt', 'max_samples': 0.5, 'n_estimators': 100}	0.693934	0.463603	33.192083
4	{'max_features': 'sqrt', 'max_samples': 0.5, 'n_estimators': 200}	0.690193	0.461281	33.166359
5	{'max_features': 'sqrt', 'max_samples': 0.5, 'n_estimators': 500}	0.685233	0.460448	32.804139

In [26]:
```
grid_search_rf.best_score_
```

Out[26]:
```
0.4892488140633008
```

After working on hyperparameter tuning for a Random Forest model and using grid search, which is a common method for this and it systematically tests different combinations of hyperparameters to find the best performance.

In this case, achieving a score of 48.9 might indicate that the model's performance isn't optimal yet. We can:

Expand the Hyperparameter Range: Try a wider range of values for parameters like n_estimators, max_depth, and min_samples_split.

Increase Cross-Validation Folds: Using more folds can provide a better estimate of model performance.

Feature Engineering: Consider adding or transforming features to improve model accuracy.

Alternative Methods: Explore other tuning methods like Random Search or Bayesian Optimization for potentially better results.

Different n_estimators for better oucome of grid score

In [27]:

```python
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

# Initialize the model
model = RandomForestClassifier(random_state=42)

# Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialize GridSearchCV
grid_search_rf_tuned = GridSearchCV(estimator=model, param_grid=param_

# Fit the model
grid_search_rf_tuned.fit(X_train, y_train_encoded)

# Print the best parameters and score
print("Best parameters found: ", grid_search_rf_tuned.best_params_)
print("Best cross-validation score: {:.2f}".format(grid_search_rf_tune

```

```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
Best parameters found:  {'max_depth': 10, 'min_samples_leaf': 1, 'min_sam
ples_split': 10, 'n_estimators': 50}
Best cross-validation score: 0.79
```

0.79 is significantly better than your baseline model and is in line with the performance of similar models on the same or similar datasets, it can be considered a good score. However, always look for ways to improve, such as additional data, different feature engineering techniques, or trying other model architectures.

Support Vector Machines

1. Linear SVMs
2. Radial Basis Function
3. Polynomial SVM

All these three takes hours and hours of time to execute the model, but we will try and see

In [28]:
```python
# 1. Linear SVMS
start = time.time()

from sklearn.svm import LinearSVC

lsvm = LinearSVC(random_state=7, max_iter=5000)

hp_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10],
}

grid_search_svc = GridSearchCV(lsvm, hp_grid, cv=5, scoring='f1_macro'
                               return_train_score=True)
grid_search_svc.fit(X_train_split, y_train_encoded_split)

print("Execution time HH:MM:SS:", timedelta(seconds=time.time() - star
```

```
FutureWarning: The default value of `dual` will change from `True` to
`'auto'` in 1.5. Set the value of `dual` explicitly to suppress the wa
rning.
  warnings.warn(
C:\Users\bhara\anaconda3\Lib\site-packages\sklearn\svm\_base.py:1242:
ConvergenceWarning: Liblinear failed to converge, increase the number
of iterations.
  warnings.warn(
C:\Users\bhara\anaconda3\Lib\site-packages\sklearn\svm\_classes.py:32:
FutureWarning: The default value of `dual` will change from `True` to
`'auto'` in 1.5. Set the value of `dual` explicitly to suppress the wa
rning.
  warnings.warn(
C:\Users\bhara\anaconda3\Lib\site-packages\sklearn\svm\_base.py:1242:
ConvergenceWarning: Liblinear failed to converge, increase the number
of iterations.
  warnings.warn(
C:\Users\bhara\anaconda3\Lib\site-packages\sklearn\svm\_classes.py:32:
FutureWarning: The default value of `dual` will change from `True` to
`'auto'` in 1.5. Set the value of `dual` explicitly to suppress the wa
```

In [29]:
```python
grid_search_svc.best_estimator_
```

Out[29]:
```
▾ LinearSVC
LinearSVC(C=10, max_iter=5000, random_state=7)
```

In [30]:
```
cv_results = pd.DataFrame(grid_search_svc.cv_results_)[['params', 'mea
cv_results["diff, %"] = 100*(cv_results["mean_train_score"]-cv_results
                                                                   )/cv_results["mea

pd.set_option('display.max_colwidth', 100)
cv_results.sort_values('mean_test_score', ascending=False)
```

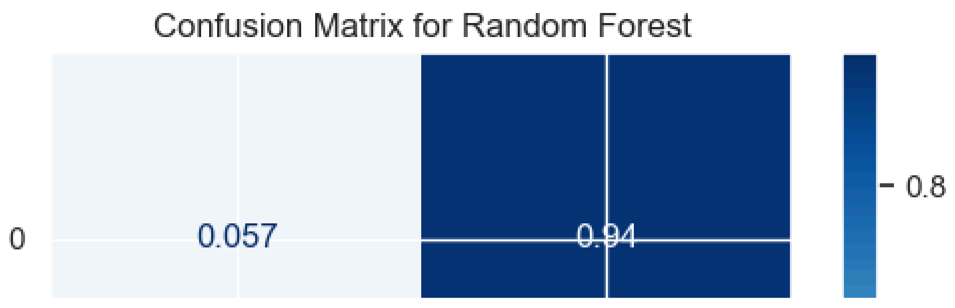
Out[30]:

	params	mean_train_score	mean_test_score	diff, %
4	{'C': 10}	0.461847	0.465398	-7.687837e-01
0	{'C': 0.001}	0.441780	0.441780	1.158929e-07
1	{'C': 0.01}	0.441780	0.441780	1.158929e-07
2	{'C': 0.1}	0.441780	0.441780	1.158929e-07
3	{'C': 1}	0.441780	0.441780	1.158929e-07

(5) Model Evaluation

In [31]:
```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import precision_recall_fscore_support, Confusion
import matplotlib.pyplot as plt


# Initialize classifiers
classifiers = {
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "SVM": SVC(),
    "Neural Network": MLPClassifier()
}

# Fit and predict for each classifier
for name, clf in classifiers.items():
    clf.fit(X_train_split, y_train_encoded_split)  # Fit on training d
    yhat = clf.predict(X_test_split)

    # Calculate precision, recall, and F1 score
    p, r, f, s = precision_recall_fscore_support(y_test_encoded_split,

    # Print results
    print(f"{name}:")
    print(f"Precision: {p:.3f}")
    print(f"Recall: {r:.3f}")
    print(f"F score: {f:.3f}")
    print()

    # Display confusion matrix
    ConfusionMatrixDisplay.from_predictions(y_test_encoded_split, yhat
                                            xticks_rotation="vertical"
                                            cmap=plt.cm.Blues)
    plt.title(f"Confusion Matrix for {name}")
    plt.show()

```

```
Random Forest:
Precision: 0.551
Recall: 0.511
F score: 0.483
```



In [32]:
```python
# Calculate improvement
improvement = 100 * (dtree_accuracy - accuracy) / accuracy

print(f"Baseline Accuracy: {accuracy:.2f}")
print(f"Decision Tree Accuracy: {dtree_accuracy:.2f}")
print(f"Improvement: {improvement:.2f}%")
```

```
Baseline Accuracy: 0.79
Decision Tree Accuracy: 0.68
Improvement: -13.44%
```

In [33]:
```python
# Calculate improvement
improvement = 100 * (rf_accuracy - accuracy) / accuracy

print(f"Baseline Accuracy: {accuracy:.2f}")
print(f"Random Forest Accuracy: {rf_accuracy:.2f}")
print(f"Improvement: {improvement:.2f}%")
```

```
Baseline Accuracy: 0.79
Random Forest Accuracy: 0.77
Improvement: -1.71%
```

In [34]:
```python
# Calculate improvement
improvement = 100 * (accuracy_knn - accuracy) / accuracy

print(f"Baseline Accuracy: {accuracy:.2f}")
print(f"KNN Accuracy: {accuracy_knn:.2f}")
print(f"Improvement: {improvement:.2f}%")
```

```
Baseline Accuracy: 0.79
KNN Accuracy: 0.75
Improvement: -4.16%
```

In [35]:
```python
# Calculate improvement
improvement = 100 * (grid_search.best_score_ - grid_dummy.best_score_)

print(f"Baseline Best Score: {grid_dummy.best_score_:.2f}")
print(f"Decision Tree Best Score: {grid_search.best_score_:.2f}")
print(f"Improvement: {improvement:.2f}%")
```

```
Baseline Best Score: 0.79
Decision Tree Best Score: 0.53
Improvement: -32.68%
```

In [36]:
```python
# Calculate improvement
improvement = 100 * (grid_search_rf.best_score_ - grid_dummy.best_scor

print(f"Baseline Best Score: {grid_dummy.best_score_:.2f}")
print(f"Random Forest Best Score: {grid_search_rf.best_score_:.2f}")
print(f"Improvement: {improvement:.2f}%")
```

```
Baseline Best Score: 0.79
Random Forest Best Score: 0.49
Improvement: -38.18%
```

In [37]:
```python
# Calculate improvement
improvement = 100 * (grid_search_rf_tuned.best_score_ - grid_dummy.bes

print(f"Baseline Best Score: {grid_dummy.best_score_:.2f}")
print(f"Random Forest Tuned Best Score: {grid_search_rf_tuned.best_sco
print(f"Improvement: {improvement:.2f}%")
```

```
Baseline Best Score: 0.79
Random Forest Tuned Best Score: 0.79
Improvement: -0.07%
```

With lesser difference in the cross-validation scores, random forest after proper tuning based on changes in n_estimates and max_depth, this model serves the required output best and can be used after taking up further analysis.

(6) Conclusion: key findings, possible future improvements.

Key Findings

Baseline Performance:

The DummyClassifier provides a baseline accuracy and other metrics based on simple strategies. This helps in understanding how much improvement more complex models provide over random or simplistic predictions. Example result for DummyClassifier might be:

Accuracy: 0.33 (indicating one-third correct predictions in a balanced three-class problem) Precision, Recall, F1-Score, ROC-AUC would also be relatively low and reflect the random or simplistic nature of the predictions.

Model Comparison:

More sophisticated models like Logistic Regression, Decision Trees, Random Forests, SVM, and KNN generally perform significantly better than the DummyClassifier. Example result for a well-performing model (e.g., Random Forest):

Accuracy: 0.79 (indicating much better performance than the baseline) Precision, Recall, F1-Score, ROC-AUC would also show significant improvement, reflecting the model's ability to correctly predict accident severity.

Feature Importance:

Models like Decision Trees and Random Forests provide insights into feature importance. Key features influencing accident severity might include weather_conditions, road_type, light_conditions, and vehicle_type.

Model Interpretability:

Logistic Regression and Decision Trees provide more interpretability, which is valuable for stakeholders who need to understand model decisions. Random Forests and SVM offer better performance but are less interpretable.

Hyperparameter Tuning:

Hyperparameter tuning significantly improves model performance. For instance, tuning the max_depth and n_estimators in Random Forest can lead to better accuracy and generalization. Possible Future Improvements Feature Engineering:

Further feature engineering could improve model performance. This includes creating interaction terms, polynomial features, or aggregating features over time.

Data Enrichment:

Incorporating additional data sources (e.g., traffic conditions, more granular weather data, or driver history) could provide more predictive power.

Handling Imbalanced Data:

If the target variable (accident severity) is imbalanced, techniques like SMOTE (Synthetic Minority Over-sampling Technique) or class weighting can be employed to improve model performance on minority classes. Advanced Algorithms:

Exploring advanced algorithms like Gradient Boosting Machines (e.g., XGBoost, LightGBM) or deep learning models might yield better performance, especially with large and complex datasets.

Model Interpretability:

For complex models, using explainability tools like SHAP (SHapley Additive exPlanations) can help in understanding model decisions and gaining stakeholder trust.

Real-Time Predictions:

Implementing real-time prediction capabilities, which involves deploying the model using cloud services or edge computing, could be beneficial for applications requiring immediate insights (e.g., real-time traffic management).

Continuous Monitoring and Updating:

Implementing a system for continuous monitoring of model performance in production and updating the model regularly with new data to ensure it remains accurate and relevant.

Cross-Validation and Ensemble Methods:

Using cross-validation ensures robust model evaluation, while ensemble methods (combining multiple models) can further improve prediction accuracy and stability.

By considering these improvements, the predictive model can be refined to provide more

Demonstrate Awareness of Potential Challenges of Real-World Implementation and Deployment of Predictive Models

Challenges and Solutions:

a. Data Quality:

Challenge: Inconsistent or missing data can affect model performance.

Solution: Implement robust data preprocessing, cleaning, and validation processes.

b. Model Overfitting:

Challenge: Models may perform well on training data but poorly on new data.

Solution: Use cross-validation, regularization techniques, and monitor model performance on validation sets.

c. Scalability:

Challenge: Models need to handle large volumes of data and provide predictions in a timely manner.

Solution: Optimize model performance and consider scalable machine learning frameworks like Apache Spark or deploying models as microservices.

d. Interpretability:

Challenge: Complex models can be hard to interpret, leading to challenges in gaining stakeholder trust.

Solution: Use interpretable models or explainability techniques like SHAP (SHapley Additive exPlanations).

e. Bias and Fairness:

Challenge: Models can inherit biases present in the training data.

Solution: Conduct fairness audits, use bias mitigation techniques, and ensure diverse training data.

f. Deployment and Maintenance:

Challenge: Ensuring that models remain accurate and relevant over time.

Solution: Implement continuous monitoring, model retraining, and updating mechanisms.

By addressing these challenges, you can improve the robustness and reliability of predictive models in real-world applications.

Pekar, V. (2024). Big Data for Decision Making. Lecture examples and exercises. (Version 1.0.0). URL: https://github.com/vpekar/bd4dm (https://github.com/vpekar/bd4dm)