Predicting Taxi accidents severity

August 15, 2025

1 Predicting Severity of Taxi Accidents in the UK - Part 2: Model Development

2 Problem definition

Taxi and ride hailing apps face the risk of accidents which can affect passenger satisfaction, claims against the company and pricing. Traditional operational strategies ignore pre-trip information such as weather, road conditions and other key details. This leads to missed opportunities for risk reduction, cost optimisation and safer fleet management

Goal of project

Predict severity of taxi collisions for ride hailing apps before trip begins, using pre-accident data available at time of booking

Potential Benefits

- Better routing to avoid risky areas - If the model predicts a ride to have a severe accident, re-rotuing can be done to prevent this
- Insurance cost reduction - If the model is implemented and riskier routes are avoided, overall risk factor of the taxi company will reduce which it can use to negotiate lower insurance premiums
- Proactive driver training - driver profiles with higher chances of severe accidents can be called for retraining to avoid future accidents
- Better pricing models based on pre-booking information - this will make the company more competitive and allow great profits#

Note: EDA and pre-processing are done in part 1 of the file

[1]:
```python
# Importing relevant libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import time

# To evaluate model accuracy, we use precision, recall and f-scores
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import classification_report, confusion_matrix
```

```
# Importing sns and matplotlib for better visualisation of confusion matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Hiding warnings for better presentation
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```
# Importing data files
train_df = pd.read_csv('processed_train_data.csv')
test_df = pd.read_csv('processed_test_data.csv')
```

[3]:
```
# Checking train_df columns

train_df.head()
```

[3]:
```
   age_of_driver_scaled  hour_scaled  day_of_week  first_road_class  \
0             -0.258043     1.350882          7.0               6.0
1              0.386947    -1.916732          1.0               6.0
2              0.879176    -0.516326          7.0               6.0
3              0.488788     0.572879          5.0               3.0
4             -0.784218     0.417278          7.0               3.0

   road_type  urban_or_rural_area  accident_year  sex_of_driver  \
0        6.0                  1.0         2022.0            1.0
1        6.0                  1.0         2021.0            1.0
2        2.0                  1.0         2022.0            3.0
3        6.0                  1.0         2023.0            1.0
4        3.0                  1.0         2023.0            1.0

   age_of_driver  driver_imd_decile  …  time_of_day  road_surface_label  \
0           40.0                1.0  …          2.0                 1.0
1           40.0                1.0  …          2.0                 1.0
2           40.0                2.0  …          1.0                 3.0
3           52.0                3.0  …          1.0                 1.0
4           37.0                1.0  …          1.0                 2.0

   light_condition_label  weather_conditions_grouped  region_number  hour  \
0                    2.0                         1.0            4.0  22.0
1                    3.0                         1.0            2.0   1.0
2                    1.0                         1.0            3.0  10.0
3                    1.0                         1.0            5.0  17.0
4                    1.0                         2.0            5.0  16.0

   speed_limit_grouped  special_conditions_at_site_grouped  \
0                    2                                   1
```

```
1                    2                                    1
2                    1                                    1
3                    2                                    1
4                    1                                    1

   propulsion_code_grouped  accident_severity
0                        1                  1
1                        1                  1
2                        2                  1
3                        3                  1
4                        1                  1

[5 rows x 21 columns]
```

[4]:
```
# Note that age and hour were scaled and standardised during the group work␣
↪stage.
```

[5]:
```
train_df.shape
```

[5]:
```
(9984, 21)
```

3 Feature Selection

[6]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9984 entries, 0 to 9983
Data columns (total 21 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   age_of_driver_scaled          9984 non-null   float64
 1   hour_scaled                   9984 non-null   float64
 2   day_of_week                   9984 non-null   float64
 3   first_road_class              9984 non-null   float64
 4   road_type                     9984 non-null   float64
 5   urban_or_rural_area           9984 non-null   float64
 6   accident_year                 9984 non-null   float64
 7   sex_of_driver                 9984 non-null   float64
 8   age_of_driver                 9984 non-null   float64
 9   driver_imd_decile             9984 non-null   float64
 10  month                         9984 non-null   float64
 11  time_of_day                   9984 non-null   float64
 12  road_surface_label            9984 non-null   float64
 13  light_condition_label         9984 non-null   float64
 14  weather_conditions_grouped    9984 non-null   float64
 15  region_number                 9984 non-null   float64
 16  hour                          9984 non-null   float64
 17  speed_limit_grouped                   9984 non-null   int64
 18  special_conditions_at_site_grouped    9984 non-null   int64
 19  propulsion_code_grouped               9984 non-null   int64
 20  accident_severity                     9984 non-null   int64
dtypes: float64(17), int64(4)
memory usage: 1.6 MB
```

We have 21 columns but not all will be necessary. Based on the insights from the EDA performed earlier (part 1), a few columns will be removed. These are:

1. Accident Year - This has little relevance to the taxi companies as this does not play a role on the accident's circumstances.
2. The first 2 columns (scaled variables) - Scaling will only be re-done for models which require it.
3. Road surface - this is often based on the weather of the day therefore having either one should suffice.
4. Sex of driver - This could cause legal issues if companies use gender information to bias against a certain gender or use sex as a means of meeting any of the goals mentioned above.

```
[7]: # Removing these columns from the training data
train_df.drop(['accident_year', 'age_of_driver_scaled', 'hour_scaled',␣
 ↪'road_surface_label', 'sex_of_driver'], axis = 1, inplace = True)

# Doing the same for testing data
test_df.drop(['accident_year', 'age_of_driver_scaled', 'hour_scaled',␣
 ↪'road_surface_label', 'sex_of_driver'], axis = 1, inplace = True)
```

The model will be trained with the remaining features. Once the best models are selected, further feature selection will be done to reduce model complexity

```
[8]: # specifying correct data types for all columns
train_df.astype('int64').dtypes
test_df.astype('int64').dtypes
```

```
[8]: day_of_week                           int64
first_road_class                      int64
road_type                             int64
urban_or_rural_area                   int64
age_of_driver                         int64
driver_imd_decile                     int64
month                                 int64
time_of_day                           int64
light_condition_label                 int64
weather_conditions_grouped            int64
region_number                         int64
hour                                  int64
speed_limit_grouped                   int64
```

```
special_conditions_at_site_grouped    int64
propulsion_code_grouped               int64
accident_severity                     int64
dtype: object
```

3.1 Adjusting Target Variable values

Currently, 1 represents not severe and 0 represents severe.

Inherently, sklearn's evaluation modules (for precision, recall and f-score) consider 1 to be positive. In our case, it might be more useful to have severe cases as positive since they have more critical consequences in relation to route planning, pricing and insurance. If we set severe to be 1, then we can use recall to better evaluate the models.

[9]:
```
# Interchannging 1 to be 0 and 0 to be 1

train_df = train_df.replace({0:1, 1:0})

# Applying the same to the test set

test_df = test_df.replace({0:1, 1:0})
```

3.2 Splitting target and predictor variables

[10]:
```
# For training data
X_train = train_df.drop('accident_severity', axis = 1)
y_train = train_df['accident_severity'].copy()

# For test data
X_test = test_df.drop('accident_severity', axis = 1)
y_test = test_df['accident_severity'].copy()
```

NOTE: Some methods may require dummy variables while some may not. While the data will be the same, a seperate training set will be made for when binary variables are made so we can preserve the original dataset for methods which do not need dummy variables.

[11]:
```
X_train_binary = X_train.copy()
X_test_binary = X_test.copy()
```

4 Modifying variables

4.1 Creating Binary variables

[12]:
```
# Creating dummy variables for use in models that require dummies

# Importing the relevant library
from sklearn.preprocessing import OneHotEncoder

```

```
one_hot_encoder = OneHotEncoder(drop = 'first', sparse_output = False)

# Defining categorical columns
cat_columns = ['day_of_week', 'first_road_class', 'road_type',␣
 ↪'urban_or_rural_area',
              'driver_imd_decile', 'month', 'time_of_day',␣
 ↪'light_condition_label',
              'weather_conditions_grouped', 'region_number',␣
 ↪'speed_limit_grouped',
               'special_conditions_at_site_grouped', 'propulsion_code_grouped']

# Making dummy variables
cat_values_train = X_train_binary[cat_columns].to_numpy()
transformed_train = one_hot_encoder.fit_transform(cat_values_train)

# Giving these new columns names
new_col_names = one_hot_encoder.get_feature_names_out(cat_columns)

# Creating a new dataframe for the dummy variables
encoded_train = pd.DataFrame(transformed_train, columns = new_col_names, index␣
 ↪= X_train_binary.index)

# Dropping the old columns which are not converted to binary and join the new␣
 ↪columns
X_train_binary.drop(columns = cat_columns, inplace = True)
X_train_binary = pd.concat([X_train_binary, encoded_train], axis = 1)
```

[13]:
```
# Applying the same encoding to the test set
cat_values_test = X_test_binary[cat_columns].to_numpy()
transformed_test = one_hot_encoder.transform(cat_values_test)

# Creating new dataframe for test set dummy variables
encoded_test = pd.DataFrame(transformed_test, columns = new_col_names, index =␣
 ↪X_test_binary.index)

# Dropping old columns from test set and joining new columns
X_test_binary.drop(columns = cat_columns, inplace = True)
X_test_binary = pd.concat([X_test_binary, encoded_test], axis = 1)
```

We can now test different models. While there are many models to use, the following are tested on this data, alongside their reasoning:

1. Naive Bayes - Fast and effective to begin testing with. Also good to use with categorical features
2. Logistic Regression - Also a simple and fast technique to use for basic binary classification
3. Linear Support Vector Machines (LSVM) - Works well to avoid overfitting (ensured through hyper-parameter tuning) in spaces with many inputs

4. Decision Trees - Easy to interprete and effective on non-linear relationships
5. CATBoost - Designed specifically for categorical variables hence understands categorical variables better for model building. Also may perform well on my imbalanced dataset
6. Random Forest - Handles both categorical and numerical variables well. Chances of overfitting are also reduced since it uses the outputs of many decision trees to make a final prediction

5 Setting Baseline

[14]:
```
y_train.value_counts()
```

[14]:
```
accident_severity
0    8328
1    1656
Name: count, dtype: int64
```

- 0 stands for Not Severe
- 1 stands for Severe

The data is imbalanced with 'not severe' accidents representing only 16% of the values.

For the baseline, we can use the mode. All accidents will be classified as not severe. This will allow us to see the impact of imbalanced data and subsequently give a good starting point of the recall and f-score

[15]:
```
# Importing dummy classifier module
from sklearn.dummy import DummyClassifier
```

[16]:
```
# Defining the dummy classifier model
dummy_clf = DummyClassifier(strategy = 'most_frequent')

# Fitting dummy classifier on the model
dummy_clf.fit(X_train, y_train)

# Predicting outcomes using dummy classifier
yhat_baseline = dummy_clf.predict(X_train)

# Evaluating the baseline model
precision_baseline, recall_baseline, fscore_baseline, support_baseline =␣
 ↪precision_recall_fscore_support(y_train, yhat_baseline, average = 'macro',␣
 ↪zero_division = 0)
print('Baseline Model Performance')
print(f'precision = {precision_baseline:.3f}')
print(f'recall = {recall_baseline:.3f}')
print(f'fscore = {fscore_baseline:.3f}')
```

```
Baseline Model Performance
precision = 0.417
```

```
recall = 0.500
fscore = 0.455
```

[17]:
```
pd.Series(yhat_baseline).value_counts()
```

[17]:
```
0    9984
Name: count, dtype: int64
```

The performance of this baseline is very poor. It predicted all cases as 'Not severe', as expected.

The baseline predicts only the majority class. This gives a misleadingly high value of 0.5 due to macro averaging but it completely failed to identify any severe cases (A zero-division error occurs since all values are in the majority class of 'Not Severe')

6 Naive Bayes model

While the ideal choice is Bernoulli Naive Bayes (given the output is in 1 or 0), the complement Naive Bayes model works better on imbalanced data. Both will be tested

6.0.1 Bernoulli Naive Bayes model

[18]:
```
# importing Bernoulli model
from sklearn.naive_bayes import BernoulliNB
```

[19]:
```
# Defining and fitting the model

bernoulli_model = BernoulliNB()
bernoulli_model.fit(X_train_binary, y_train)

# Predicting on training set
yhat_train_bernoulli = bernoulli_model.predict(X_train_binary)
```

[20]:
```
precision_BNB, recall_BNB, fscore_BNB, support_BNB =␣
 ↪precision_recall_fscore_support(y_train, yhat_train_bernoulli, average =␣
 ↪'macro')
print('Bernoulli Naive Bayes Model Performance')
print(f'precision = {precision_BNB:.3f}')
print(f'recall = {recall_BNB:.3f}')
print(f'fscore = {fscore_BNB:.3f}')
```

```
Bernoulli Naive Bayes Model Performance
precision = 0.655
recall = 0.519
fscore = 0.499
```

6.0.2 Complement Naive Bayes Model

```
[21]: # Importing the Naive model
from sklearn.naive_bayes import ComplementNB
```

```
[22]: # Defining and fitting model

complement_NB_model = ComplementNB()
complement_NB_model.fit(X_train_binary, y_train)

# Predicting on training set

yhat_train_complementNB = complement_NB_model.predict(X_train_binary)
```

```
[23]: precision_CNB, recall_CNB, fscore_CNB, support_CNB =␣
  ↪precision_recall_fscore_support(y_train, yhat_train_complementNB, average =␣
  ↪'macro')
print('Complement Naive Bayes Model Performance')
print(f'precision = {precision_CNB:.3f}')
print(f'recall = {recall_CNB:.3f}')
print(f'fscore = {fscore_CNB:.3f}')
```

```
Complement Naive Bayes Model Performance
precision = 0.545
recall = 0.580
fscore = 0.513
```

This Complement Naive Bayes model has a slightly better f score and recall on the training set.

In our case, severe accidents (labelled 1) are more important to correctly guess since their consequences are more impactful. Recall is the most important metric for us to evaluate since we want to know how many, of all severe accidents, can we predict

7 Logistic Regression Model

```
[24]: # importing logistic regression module
from sklearn.linear_model import LogisticRegression

# Defining the model
log_model = LogisticRegression(random_state = 7, max_iter = 1000, class_weight␣
  ↪= 'balanced')
```

Note on imbalanced data handling

As mentioned earlier, the data is highly imbalanced. Different techniques for handling this were considered but not used e.g.:

1. SMOTE: Since most predictors are categorical, SMOTE does not seem to be very useful.

2. Oversampling: will also create no real benefit to the model since this will simply lead to duplication of the data.

3. SMOTENC: This works well with categorical data, however when tested, was creating invalid categorical classes e.g. 0 when there is no 0 class.

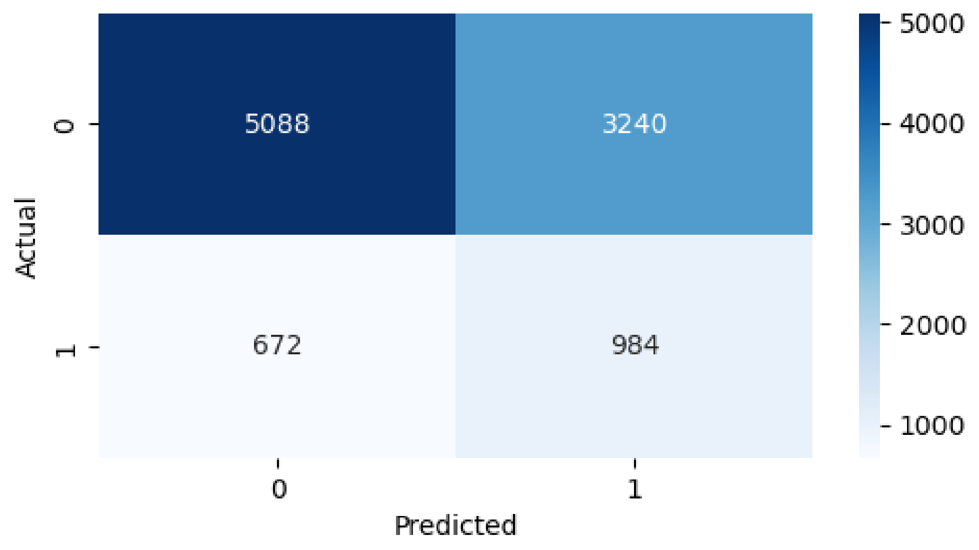
Instead, the **class_weight** attribute is added to the model (and subsequent models). This allows the models to give more weight to the under represented class, enabling a more balanced prediction model

[25]:
```
# Applying the model
log_model.fit(X_train_binary, y_train)

# Getting predicted valued on training model
yhat_log= log_model.predict(X_train_binary)
```

[26]:
```
# Evaluating model performance

precision_log, recall_log, fscore_log, support_log =␣
 ↪precision_recall_fscore_support(y_train, yhat_log, average = 'macro')
print('Logistic Regression Model Performance')
print(f'precision = {precision_log:.3f}')
print(f'recall = {recall_log:.3f}')
print(f'fscore = {fscore_log:.3f}')
```

```
Logistic Regression Model Performance
precision = 0.558
recall = 0.603
fscore = 0.529
```

[27]:
```
# Getting the confusion matrix for better analysis
cm = confusion_matrix(y_train, yhat_log)

#plotting the confusion matrix
plt.figure(figsize=(6,3))
sns.heatmap(cm, annot=True, fmt = 'd', cmap='Blues', xticklabels = [0,1],␣
 ↪yticklabels = [0,1])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```



This model seems to have performed better than the Naive Bayes model based on the fscore and recall. When tested without setting class_weight, the performance was extremely poor, hence class_weight was included

However, the models prediction on 'Severe' accidents was moderate. It did classify most of the accidents clearly but still left out 40.6% of the severe cases

[28]:
```
# To avoid having to duplicate the code for the confusion matric everytime, we␣
↪save this as a function
def cm_maker(y_train, yhat):
    '''
    Used to make the confusion matrix. y_train are the training y values and␣
↪yhat are the predicted y values of the model used
    '''
    cm = confusion_matrix(y_train, yhat)

    #plotting the confusion matrix
    plt.figure(figsize=(6,3))
    sns.heatmap(cm, annot=True, fmt = 'd', cmap='Blues', xticklabels = [0,1],␣
↪yticklabels = [0,1])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()
```

8 Linear Support Vector Machine (LSVM)

[29]:
```
# Importing the LSVM module
from sklearn.svm import LinearSVC

# We will also do some hyperparameter tuning at this point so we import the␣
 ↪gridsearch library
# Note: We are using Bayesian Optimisation

!pip install scikit-optimize
from skopt import BayesSearchCV
```

```
Requirement already satisfied: scikit-optimize in
c:\users\mites\anaconda3\lib\site-packages (0.10.2)
Requirement already satisfied: joblib>=0.11 in
c:\users\mites\anaconda3\lib\site-packages (from scikit-optimize) (1.4.2)
Requirement already satisfied: pyaml>=16.9 in c:\users\mites\anaconda3\lib\site-
packages (from scikit-optimize) (25.1.0)
Requirement already satisfied: numpy>=1.20.3 in
c:\users\mites\anaconda3\lib\site-packages (from scikit-optimize) (1.26.4)
Requirement already satisfied: scipy>=1.1.0 in
c:\users\mites\anaconda3\lib\site-packages (from scikit-optimize) (1.13.1)
Requirement already satisfied: scikit-learn>=1.0.0 in
c:\users\mites\anaconda3\lib\site-packages (from scikit-optimize) (1.5.1)
Requirement already satisfied: packaging>=21.3 in
c:\users\mites\anaconda3\lib\site-packages (from scikit-optimize) (24.1)
Requirement already satisfied: PyYAML in c:\users\mites\anaconda3\lib\site-
packages (from pyaml>=16.9->scikit-optimize) (6.0.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in
c:\users\mites\anaconda3\lib\site-packages (from scikit-learn>=1.0.0->scikit-
optimize) (3.5.0)
```

[30]:
```
# Defining the LVSM model
lsvm = LinearSVC(random_state = 5, C=5, max_iter = 10000, class_weight =␣
 ↪'balanced')
# Class_weight = balanced helps us ensure the minority class is given importance

# Definining the hyperparameters to tune
hp_grid = {
    'C' : [0.05, 0.5, 1, 2, 5], # The c parameter allows us to control␣
 ↪overfitting issues
    'max_iter': [1000, 2000, 5000, 10000],
    'class_weight': ['balanced', None]
}

# Starting the timer to evaluate how long it takes to run the grid search
start_time = time.time()
```

```
# Running the Bayesian grid search
bayes_grid_search = BayesSearchCV(
    lsvm, hp_grid, n_iter = 10, random_state = 5, scoring = 'f1_macro', cv =␣
 ↪15, verbose = 0
)
# Verbose has been set to 0 to ensure a cleaner PDF output with only necessary␣
 ↪information
# This is true for all subsequent models as well

# Training the model
bayes_grid_search.fit(X_train_binary, y_train)

# Stopping the time
end_time = time.time()

# Finding time taken
print(f'time taken = {end_time - start_time:.2f} seconds')
```

```
time taken = 20.51 seconds
```

[31]:
```
# Finding the best model and its score
print(bayes_grid_search.best_params_)
print(bayes_grid_search.best_score_)
```

```
OrderedDict({'C': 5, 'class_weight': 'balanced', 'max_iter': 5000})
0.5140966467522717
```

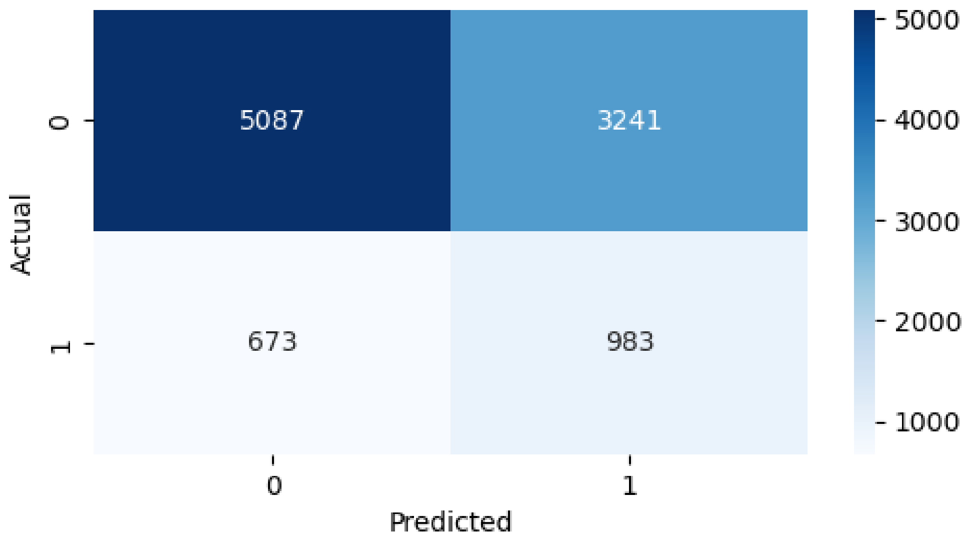
[32]:
```
# We now get the specific values for LVSM with c=0.5 and 1000 iterations
lsvm = LinearSVC(C=5, max_iter=5000, random_state = 5, class_weight =␣
 ↪'balanced')

# Fitting model on training data
lsvm.fit(X_train_binary, y_train)

# Getting predicted values
yhat_lsvm = lsvm.predict(X_train_binary)

# Evaluating model performance
precision_lsvm, recall_lsvm, fscore_lsvm, support_lsvm =␣
 ↪precision_recall_fscore_support(y_train, yhat_lsvm, average = 'macro')
print('Linear SVC Model Performance')
print(f'precision = {precision_lsvm:.3f}')
print(f'recall = {recall_lsvm:.3f}')
print(f'fscore = {fscore_lsvm:.3f}')
```

```
Linear SVC Model Performance
precision = 0.558
```

```
recall = 0.602
fscore = 0.528
```

The results with the Bayes Search and when the ideal model are seperately saved are slightly different. This is because cross validation was used in the Bayes Search stage. When finally training and evaluating on the full dataset with only the final model, cross validation was not done hence there is a possibility of overfitting.

[34]:
```
# Visualising results on the confusion matrix
cm_maker(y_train, yhat_lsvm)
```



The performance of the LVSM is similar to that of the Logisitic Regression model in terms of the f-score as well as the confusion matrix

Although we used balanced weighting in this stage, we could also manually define weightings. To determine if this is useful, we test with 2 specific weight categories

8.0.1 Testing with severe categories holding twice the weight

[35]:
```
# Specifying model with severe accidents having twice the weight
lsvm2 = LinearSVC(C=5, max_iter=5000, random_state = 5, class_weight = {0:1, 1:
 ↪2})

# Fitting model on training data
lsvm2.fit(X_train_binary, y_train)

# Getting predicted values
yhat_lsvm2 = lsvm2.predict(X_train_binary)

# Evaluating model performance
```

```
precision_lsvm2, recall_lsvm2, fscore_lsvm2, support_lsvm2 =␣
  ↪precision_recall_fscore_support(y_train, yhat_lsvm2, average = 'macro')
print('Linear SVM Model Performance (class 1 holds twice the weight)')
print(f'precision = {precision_lsvm2:.3f}')
print(f'recall = {recall_lsvm2:.3f}')
print(f'fscore = {fscore_lsvm2:.3f}')
```

```
Linear SVM Model Performance (class 1 holds twice the weight)
precision = 0.625
recall = 0.524
fscore = 0.510
```

8.0.2 Testing with severe categories holding five times the weight

[36]:
```
# Specifying model with severe accidents having 5 times the weight
# Five times is used because non-severe cases are approximately 5 times the␣
  ↪severe cases
lsvm3 = LinearSVC(C=5, max_iter=5000, random_state = 5, class_weight = {0:1, 1:
  ↪5})

# Fitting model on training data
lsvm3.fit(X_train_binary, y_train)

# Getting predicted values
yhat_lsvm3 = lsvm3.predict(X_train_binary)

# Evaluating model performance
precision_lsvm3, recall_lsvm3, fscore_lsvm3, support_lsvm3 =␣
  ↪precision_recall_fscore_support(y_train, yhat_lsvm3, average = 'macro')
print('Linear SVM Model Performance (Class 1 holds five times the weight')
print(f'precision = {precision_lsvm3:.3f}')
print(f'recall = {recall_lsvm3:.3f}')
print(f'fscore = {fscore_lsvm3:.3f}')
```

```
Linear SVM Model Performance (Class 1 holds five times the weight
precision = 0.558
recall = 0.602
fscore = 0.530
```

When double the weight was used, the model performance detoriated. However, when 5 times the weight was used, the model performed almost similarly to when ‘balanced’ weightings are used. This suggests that the ‘balanced’ model can be used for evaluation instead of manual evaluation - the difference is too small to manually encode the weights on all subsequent models when performing Bayes Search for optimal parameters.

9 Decision Trees

[37]:
```
# Importing the relevant library
from sklearn.tree import DecisionTreeClassifier
```

[38]:
```
# Defining the model
dtree = DecisionTreeClassifier(random_state=5)
# No parameters specified since we will do this by hyperparameter tuning

# Specifiying the hyper parameter tuning grid
hp_grid = {
    'max_depth': [5, 8, 10, 15, None],
    'min_samples_split': [50, 100, 200, 500],
    'class_weight' : ['balanced', None],
    'max_leaf_nodes': [10, 20, 30, 50, 100, 150, 200]
}

# Starting timer
start_time = time.time()

# Running the Bayes Search
bayes_grid_search = BayesSearchCV(
    dtree, hp_grid, n_iter = 25, cv = 15, scoring = 'recall', random_state =5,␣
  ↪verbose = 0
)

# Training model
# Note that we use the original X_train set not X_train_binary. Decision trees␣
  ↪are able to handly these variables well
# without the need for binary encoding
bayes_grid_search.fit(X_train, y_train)

# Ending timer
end_time = time.time()

print(f'Time taken = {end_time - start_time:.2f} seconds')
```

```
Time taken = 47.32 seconds
```

[39]:
```
print(bayes_grid_search.best_params_)
print(bayes_grid_search.best_score_)
```

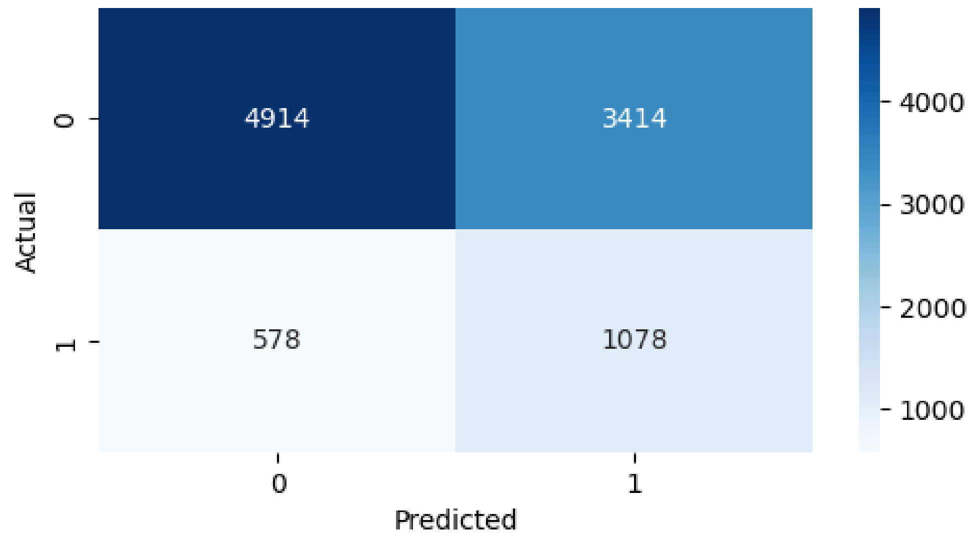
```
OrderedDict({'class_weight': 'balanced', 'max_depth': 15, 'max_leaf_nodes': 50,
'min_samples_split': 500})
0.5512039312039313
```

[40]:
```
# Saving the model with the best parameters
```

```
dtree = DecisionTreeClassifier(class_weight = 'balanced', max_depth = 15,␣
  ↪max_leaf_nodes = 50, min_samples_split = 500, random_state = 7)

# Training model on best parameters
dtree.fit(X_train, y_train)

# Prediciting values on training set
yhat_dtree = dtree.predict(X_train)

# Evaluating the model
precision_dtree, recall_dtree, fscore_dtree, support_dtree =␣
  ↪precision_recall_fscore_support(y_train, yhat_dtree, average = 'macro')
print('Decision Tree Model Performance')
print(f'precision = {precision_dtree:.3f}')
print(f'recall = {recall_dtree:.3f}')
print(f'fscore = {fscore_dtree:.3f}')
```

```
Decision Tree Model Performance
precision = 0.567
recall = 0.621
fscore = 0.531
```

[41]:
```
cm_maker(y_train, yhat_dtree)
```



The decision tree has higher recall and fscores than the lvsm model. The model also did a very good job in identifying severe accidents with 1078 correct predictions, leaving out only 34.9% of the severe cases in its prediction.

Further refinement of the inputs may make the performance better e.g. using the only the important features and tuning the model more

10 8. CATBoost model

CATBoost models are gradient boosting hence use multiple decision trees to make a prediction. They work well with categorical data and since most of our accident related data is categorical, this is an ideal model to use

[42]:
```
# Importing relevant library
!pip install catboost
from catboost import CatBoostClassifier
```

```
Requirement already satisfied: catboost in c:\users\mites\anaconda3\lib\site-
packages (1.2.8)
Requirement already satisfied: graphviz in c:\users\mites\anaconda3\lib\site-
packages (from catboost) (0.21)
Requirement already satisfied: matplotlib in c:\users\mites\anaconda3\lib\site-
packages (from catboost) (3.9.2)
Requirement already satisfied: numpy<3.0,>=1.16.0 in
c:\users\mites\anaconda3\lib\site-packages (from catboost) (1.26.4)
Requirement already satisfied: pandas>=0.24 in
c:\users\mites\anaconda3\lib\site-packages (from catboost) (2.2.2)
Requirement already satisfied: scipy in c:\users\mites\anaconda3\lib\site-
packages (from catboost) (1.13.1)
Requirement already satisfied: plotly in c:\users\mites\anaconda3\lib\site-
packages (from catboost) (5.24.1)
Requirement already satisfied: six in c:\users\mites\anaconda3\lib\site-packages
(from catboost) (1.16.0)
Requirement already satisfied: python-dateutil>=2.8.2 in
c:\users\mites\anaconda3\lib\site-packages (from pandas>=0.24->catboost)
(2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in
c:\users\mites\anaconda3\lib\site-packages (from pandas>=0.24->catboost)
(2024.1)
Requirement already satisfied: tzdata>=2022.7 in
c:\users\mites\anaconda3\lib\site-packages (from pandas>=0.24->catboost)
(2023.3)
Requirement already satisfied: contourpy>=1.0.1 in
c:\users\mites\anaconda3\lib\site-packages (from matplotlib->catboost) (1.2.0)
Requirement already satisfied: cycler>=0.10 in
c:\users\mites\anaconda3\lib\site-packages (from matplotlib->catboost) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
c:\users\mites\anaconda3\lib\site-packages (from matplotlib->catboost) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in
c:\users\mites\anaconda3\lib\site-packages (from matplotlib->catboost) (1.4.4)
Requirement already satisfied: packaging>=20.0 in
c:\users\mites\anaconda3\lib\site-packages (from matplotlib->catboost) (24.1)
Requirement already satisfied: pillow>=8 in c:\users\mites\anaconda3\lib\site-
packages (from matplotlib->catboost) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
```

```
c:\users\mites\anaconda3\lib\site-packages (from matplotlib->catboost) (3.1.2)
Requirement already satisfied: tenacity>=6.2.0 in
c:\users\mites\anaconda3\lib\site-packages (from plotly->catboost) (8.2.3)
```

[44]:
```
# CATBoost needs us to specifiy categorical variables so we save this in a␣
 ↪variable
cat_variables = [
    'day_of_week', 'first_road_class', 'road_type', 'urban_or_rural_area',␣
 ↪'driver_imd_decile',
    'time_of_day', 'light_condition_label', 'weather_conditions_grouped',␣
 ↪'region_number', 'speed_limit_grouped',
    'special_conditions_at_site_grouped', 'propulsion_code_grouped'
]

# Converting the column to string type to avoid float issues since CATBoost␣
 ↪does not recognise float values.
for col in cat_variables:
    X_train[col] = X_train[col].astype(str)

# Matching the column names to the indices
cat_variables_indices = [X_train.columns.get_loc(col) for col in cat_variables]

# Calling and specifying the CATboost model
catboost_model = CatBoostClassifier(cat_features = cat_variables_indices,
                                    auto_class_weights = 'Balanced',
                                    iterations = 100,
                                    verbose = 0, # Set as 0 to ensure cleaner␣
 ↪output on the saved PDF
                                    random_state = 5) # more hyperparameters␣
 ↪are specified later using hyper parameter tuning

# Specifying the hyperparameters to tune
hp_grid = {
    'learning_rate': [0.01, 0.1, 0.3],
    'depth': [3, 5, 7]
}

# Starting timer
start_time = time.time()

# Running the Bayes Search
bayes_search_CATboost = BayesSearchCV(
    catboost_model, hp_grid, cv=15, n_iter = 25, verbose = 0, scoring = 'recall'
)

# Training model
bayes_search_CATboost.fit(X_train, y_train)
```

```
# End timer
end_time = time.time()

# Print time taken
print(f'Time taken = {end_time - start_time:.2f} seconds')
```

```
Time taken = 2120.19 seconds
```

[45]:
```
print(bayes_search_CATboost.best_params_)
print(bayes_search_CATboost.best_score_)
```

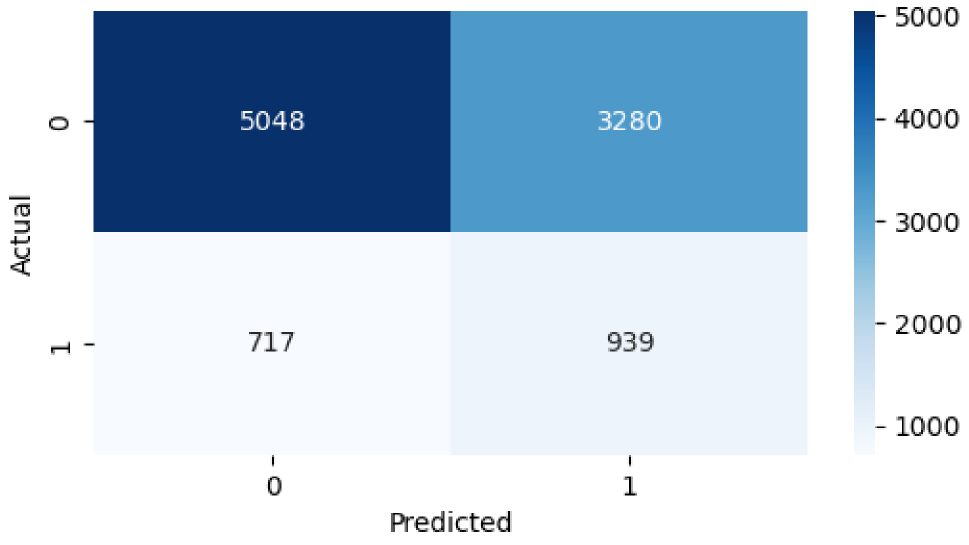
```
OrderedDict({'depth': 3, 'learning_rate': 0.01})
0.5332077532077532
```

[46]:
```
# Saving the model with the best parameters
catboost_model =  CatBoostClassifier(cat_features = cat_variables_indices,
                                    auto_class_weights = 'Balanced',
                                    iterations = 100,
                                     verbose = 0,
                                    depth = 3,
                                    learning_rate = 0.01,
                                    random_state = 5)

# Starting timer for training time
start_time = time.time()

# Training the model with training data
catboost_model.fit(X_train, y_train)

# Predicting yhat values using this model
yhat_CATboost = catboost_model.predict(X_train)

# Ending timer
end_time = time.time()

# Evaluating the model
precision_CATboost, recall_CATboost, fscore_CATboost, support_CATboost =␣
 ↪precision_recall_fscore_support(y_train, yhat_CATboost, average = 'macro')
print('CATBoost Model Performance')
print(f'precision = {precision_CATboost:.3f}')
print(f'recall = {recall_CATboost:.3f}')
print(f'fscore = {fscore_CATboost:.3f}')
# Time taken to train model on best parameter
print(f'Time taken = {end_time - start_time:.2f} seconds')
```

```
CATBoost Model Performance
precision = 0.549
recall = 0.587
```

```
fscore = 0.518
Time taken = 4.30 seconds
```

[47]:
```
cm_maker (y_train, yhat_CATboost)
```



The CATBoost model took significantly longer to train (3.29 seconds AFTER identifying the best model parameters) and used up much more resources than the decision tree. However, its performance was worse than the decision tree with a recall score of just 0.587.

Its correct classification of severe cases was also quite low at only 939 correct class 1 predictions but it was better at determining the class 0 cases when compared to the decision trees.

There is a possibility that the CATboost model may be a better option but given the limit on time and processing power, the hyperparameters for this cannot be tuned any further

11 9. Random Forest

[48]:
```
# Importing the reevant library
from sklearn.ensemble import RandomForestClassifier
```

[50]:
```
# Saving the model with initial hyperparamters
rf = RandomForestClassifier(random_state=5)

# Setting up the hyperparameter grid
hp_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 10, 15], # Max depth of 15 set since this was optimal for␣
 ↪the decision tree on its own
    'class_weight': [None, 'balanced'],
    'min_samples_split': [10, 20], # slightly higher numbers used to prevent␣
 ↪overfitting
}

# Starting the timer
start_time = time.time()

# Running the Bayes Search
bayes_search_rf = BayesSearchCV(
    rf, hp_grid, n_iter = 25, random_state = 5, verbose = 0, scoring =␣
 ↪'recall', cv=15
)

# Training the model
bayes_search_rf.fit(X_train, y_train)

# End timer
end_time = time.time()

# Print time taken
print(f'Time taken = {end_time - start_time:.2f} seconds')
```

```
Time taken = 825.28 seconds
```

[51]:
```
print(bayes_search_rf.best_params_)
print(bayes_search_rf.best_score_)
```

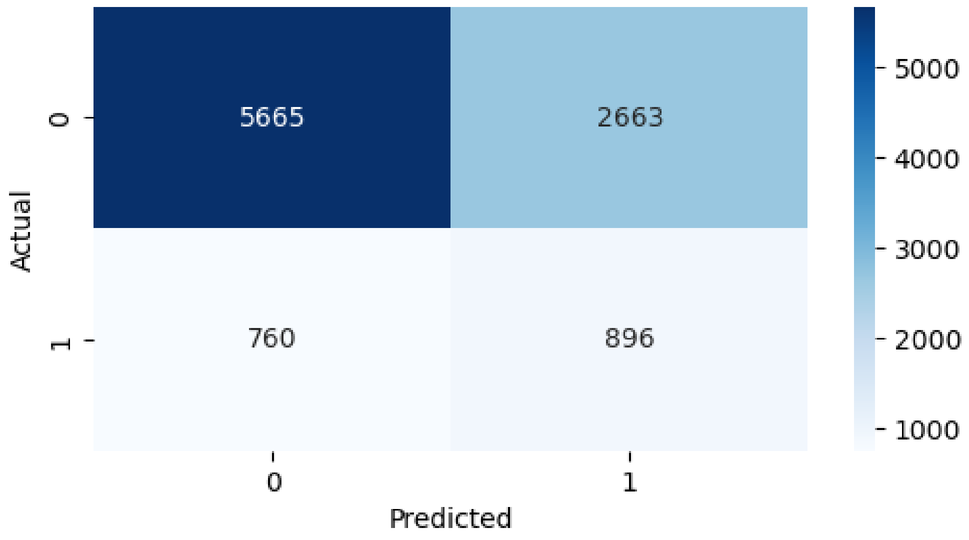
```
OrderedDict({'class_weight': 'balanced', 'max_depth': 5, 'min_samples_split':
17, 'n_estimators': 200})
0.4733715533715534
```

[52]:
```
# Defining the random forest with the best paramaters
rf = RandomForestClassifier(class_weight = 'balanced', max_depth = 5,␣
 ↪min_samples_split = 17, n_estimators = 200, random_state = 5)

# Starting the timer to evaluate training time
start_time = time.time()

# Training the rf with the training set
rf.fit(X_train, y_train)

# Predicting using the rf on the training set
yhat_rf = rf.predict(X_train)

# End timer
end_time = time.time()

# Evaluating the model
```

```
precision_rf, recall_rf, fscore_rf, support_rf =␣
 ↪precision_recall_fscore_support(y_train, yhat_rf, average = 'macro')
print('Random Forest Model Performance')
print(f'precision = {precision_rf:.3f}')
print(f'recall = {recall_rf:.3f}')
print(f'fscore = {fscore_rf:.3f}')
# Time taken to train model on best parameter
print(f'Time taken = {end_time - start_time:.2f} seconds')
```

```
Random Forest Model Performance
precision = 0.567
recall = 0.611
fscore = 0.556
Time taken = 2.25 seconds
```

[53]:
```
cm_maker(y_train, yhat_rf)
```



The recall score on the specified random forest is quite decent at 0.611. However, only 896 actually severe accidents were predicted as severe. This is the lowest from all models.

On the contrary, the model predicted non-severe cases really well with the highest correct predictions of non-severe cases from all models (5665), making the random forest good at predicting the majority class.

However, when using the Bayes Search to optimise the hyperparameters, it was really low. This shows a sign of overfitting on the training data

12 Refining top 2 models

[54]:
```
# Making a table of precision, recall and fscores

model_scores = {
    'Baseline (mode)': [precision_baseline, recall_baseline, fscore_baseline],
    'Naive Bayes (Bernoulli)': [precision_BNB, recall_BNB, fscore_BNB],
    'Naive Bayes (Complement)': [precision_CNB, recall_CNB, fscore_CNB],
    'Logistic Regression': [precision_log, recall_log, fscore_log],
    'LSVM': [precision_lsvm, recall_lsvm, fscore_lsvm],
    'Decision Trees': [precision_dtree, recall_dtree, fscore_dtree],
    'CATBoost': [precision_CATboost, recall_CATboost, fscore_CATboost],
    'Random Forest' :[precision_rf, recall_rf, fscore_rf]
}

models_summary = pd.DataFrame(model_scores).transpose()
models_summary.columns=['Precision', 'Recall', 'Fscore']
models_summary
```

[54]:
```
                          Precision    Recall    Fscore
Baseline (mode)            0.417067  0.500000  0.454784
Naive Bayes (Bernoulli)    0.655408  0.519235  0.498883
Naive Bayes (Complement)   0.545370  0.580076  0.513295
Logistic Regression        0.558144  0.602577  0.528505
LSVM                       0.557939  0.602215  0.528264
Decision Trees             0.567369  0.620512  0.530913
CATBoost                   0.549097  0.586588  0.518022
Random Forest              0.566734  0.610649  0.555802
```

Based on this, the 2 best models are Random Forests and Decision Trees. These models have the highest f scores and the best recall scores as well. Further refinement will be done on these models

12.1 Top feature selection

Since computational resources are limited, the most important features first need to be identified in order to reduce model training time. Identifying the top features also ensures unnecessary information is not fed into the model which can create cases of overfitting or unnecessary prediction features which worsen model quality.

Recursive Feature elimination is used to determine the most important features.

This will be run on the initially trained decision tree and random forest models.

[55]:
```
# Importing the relevant library

from sklearn.feature_selection import RFE
```

[56]:
```
#Identifying the most important features as per the trained decision tree model
feature_selector_dtree = RFE(dtree, n_features_to_select = 10)
```

```
feature_selector_dtree.fit(X_train, y_train)

# Identifying the top features
feature_selector_dtree.get_feature_names_out()
```

```
[56]: array(['road_type', 'age_of_driver', 'month', 'time_of_day',
              'light_condition_label', 'weather_conditions_grouped',
              'region_number', 'hour', 'speed_limit_grouped',
              'special_conditions_at_site_grouped'], dtype=object)
```

```
[57]: #Identifying the most important features as per the trained random forest model
feature_selector_rf = RFE(rf, n_features_to_select = 10)
feature_selector_rf.fit(X_train, y_train)

# Identifying the top features
feature_selector_rf.get_feature_names_out()
```

```
[57]: array(['road_type', 'urban_or_rural_area', 'age_of_driver', 'month',
              'time_of_day', 'light_condition_label', 'region_number', 'hour',
              'speed_limit_grouped', 'special_conditions_at_site_grouped'],
             dtype=object)
```

While 9 features match in both models, the decision tree considers 'weather' important and the rf considers 'urban/rural area' more important. We need to select the same 10 features for both models. In this case, we will pick urban/rural since weather conditions could somewhat be inferred by the month variable (certain months are known to have certain weather conditions - this was shown in the group assignment).

Ideally though we would have picked all features if computing time was not a concern.

```
[58]: # Defining new training and testing set with above features

X_train = X_train[['road_type', 'urban_or_rural_area', 'age_of_driver', 'month',
       'time_of_day', 'light_condition_label', 'region_number', 'hour',
       'speed_limit_grouped', 'special_conditions_at_site_grouped']]

X_test = X_test[['road_type', 'urban_or_rural_area', 'age_of_driver', 'month',
       'time_of_day', 'light_condition_label', 'region_number', 'hour',
       'speed_limit_grouped', 'special_conditions_at_site_grouped']]
```

12.2 Refined Decision Tree

We can now commit more time to testing more hyperparameters on the decision tree

```
[59]: dtree = DecisionTreeClassifier(random_state = 5, class_weight = 'balanced')

# Defining hyperparameter grid

hp_grid = {
    'max_depth': [5,8,10,15,20, None],
    'min_samples_split': [20, 50, 100, 200, 400, 500],
    'max_leaf_nodes': [30, 40, 50, 80, 100, 150]
}
# start timer
start_time = time.time()

# Running the bayesian search, but with more iterations
bayes_search_dt = BayesSearchCV(
    dtree, hp_grid, cv = 20, n_iter = 30, verbose = 0, scoring = 'recall',␣
  ↪random_state=5
)
# More iterations were set with a larger cv for more reliable estimates

bayes_search_dt.fit(X_train, y_train)

# Ending timer
end_time = time.time()

print(f'Time taken = {end_time - start_time:.2f} seconds')
```

```
Time taken = 95.29 seconds
```

[60]:
```
bayes_search_dt.best_estimator_
```

[60]:
```
DecisionTreeClassifier(class_weight='balanced', max_depth=20,
                       max_leaf_nodes=150, min_samples_split=500,
                       random_state=5)
```

[61]:
```
# This decision tree model is saved based on best settings
dtree = DecisionTreeClassifier(random_state = 5, class_weight = 'balanced',␣
  ↪max_depth=20,
                               max_leaf_nodes=150, min_samples_split=500)
# Getting training time
start_time = time.time()

# Fitting this decision tree on the training data
dtree.fit(X_train, y_train)

# Making predictions on the training data
yhat_train_dtree = dtree.predict(X_train)

# Ending timer
end_time = time.time()

# Saving the  evaluation metrics
```

```
precision_dtree_train, recall_dtree_train, fscore_dtree_train,␣
↪support_dtree_train = precision_recall_fscore_support(y_train,␣
↪yhat_train_dtree, average = 'macro')

# Getting training time
print(f'Decision tree training time = {end_time - start_time:.2f} seconds')
```

```
Decision tree training time = 0.08 seconds
```

12.2.1 Applying model on test data

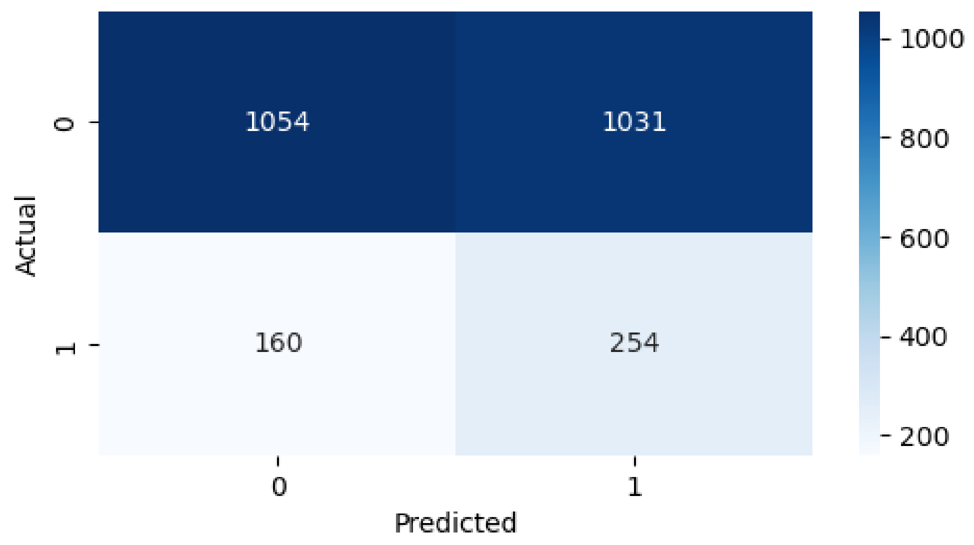
[62]:
```
# Running decision tree on test data
yhat_test_dtree = dtree.predict(X_test)

# Saving the  evaluation metrics
precision_dtree_test, recall_dtree_test, fscore_dtree_test, support_dtree_test␣
↪= precision_recall_fscore_support(y_test, yhat_test_dtree, average = 'macro')
```

12.2.2 Evaluating decision tree

[63]:
```
model_scores = {
    'Training Data': [precision_dtree_train, recall_dtree_train,␣
↪fscore_dtree_train],
    'Test data': [precision_dtree_test, recall_dtree_test, fscore_dtree_test]
}

models_summary = pd.DataFrame(model_scores).transpose()
models_summary.columns=['Precision', 'Recall', 'Fscore']
models_summary
```

[63]:
```
               Precision    Recall    Fscore
Training Data   0.569580  0.624112  0.535642
Test data       0.532935  0.559521  0.468990
```

[64]:
```
# Getting confusion matrix for test data
cm_maker(y_test, yhat_test_dtree)
```



Model interpretation done at the end

12.3 Refined Random Forest

[65]:
```
rf = RandomForestClassifier(random_state=5, class_weight = 'balanced')

# Setting up the hyperparameter grid with more options for all parameters
hp_grid = {
    'n_estimators': [100, 150, 200, 250, 300],
    'max_depth': [5, 8, 11, 15],
    'min_samples_split': [10, 20, 30],
}

# starting timer
start_time = time.time()

# Running the bayes search
bayes_search_rf = BayesSearchCV(
    rf, hp_grid, n_iter = 30, cv = 20, verbose = 0, random_state = 5, scoring =␣
  ↪'recall'
)

# Fitting the search to training data
bayes_search_rf.fit(X_train, y_train)

# Ending the time
end_time = time.time()

print(f'Time taken = {end_time - start_time:.2f} seconds')
```

```
Time taken = 1149.64 seconds
```

[66]:
```
bayes_search_rf.best_params_
```

[66]:
```
OrderedDict([('max_depth', 5),
             ('min_samples_split', 30),
             ('n_estimators', 100)])
```

[67]:
```
# This decision tree model is saved based on best settings
rf = RandomForestClassifier(random_state=5, class_weight = 'balanced',␣
  ↪max_depth= 5, min_samples_split = 30, n_estimators= 100)

# Getting training time
start_time = time.time()

# Fitting this decision tree on the training data
rf.fit(X_train, y_train)

# Making predictions on the training data
yhat_train_rf = rf.predict(X_train)

# Ending timer
```

```
end_time = time.time()

# Saving the  evaluation metrics
precision_rf_train, recall_rf_train, fscore_rf_train, support_rf_train =␣
 ↪precision_recall_fscore_support(y_train, yhat_train_rf, average = 'macro')

# Getting training time
print(f'Random Forest training time = {end_time - start_time:.2f} seconds')
```

```
Random Forest training time = 1.02 seconds
```

12.3.1 Applying model on test data

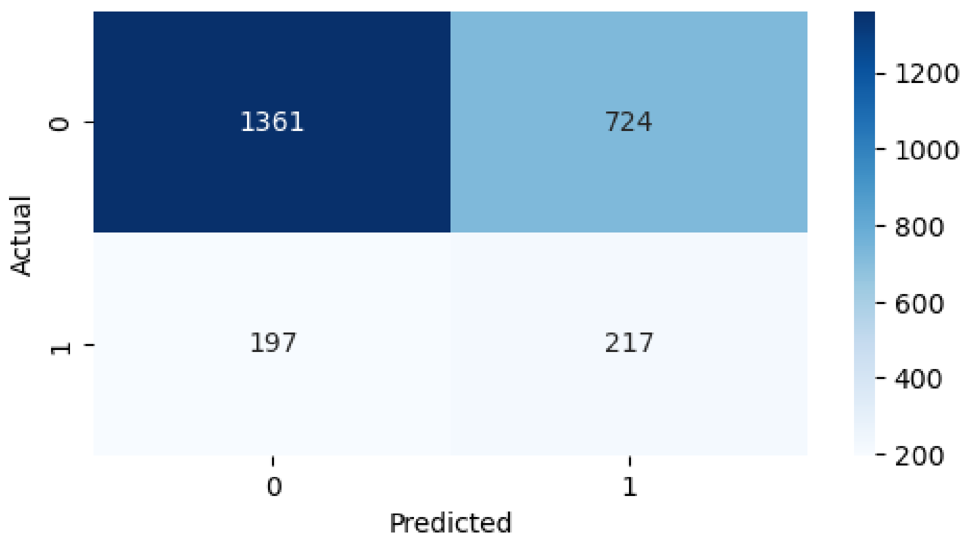
[68]:
```
# Running decision tree on test data
yhat_test_rf = rf.predict(X_test)

# Saving the  evaluation metrics
precision_rf_test, recall_rf_test, fscore_rf_test, support_rf_test =␣
 ↪precision_recall_fscore_support(y_test, yhat_test_rf, average = 'macro')
```

12.3.2 Evaluating Random Forest

[69]:
```
model_scores = {
    'Training Data': [precision_rf_train, recall_rf_train, fscore_rf_train],
    'Test data': [precision_rf_test, recall_rf_test, fscore_rf_test]
}

models_summary = pd.DataFrame(model_scores).transpose()
models_summary.columns=['Precision', 'Recall', 'Fscore']
models_summary
```

[69]:
```
               Precision    Recall    Fscore
Training Data   0.565642  0.611247  0.550171
Test data       0.552081  0.588456  0.533741
```

[70]:
```
# Plotting confusion matrix
cm_maker(y_test, yhat_test_rf)
```



12.4 Final Model Interpretation

After feature selection and some further tuning, the decision tree performed slightly better on the training data than with the previous inputs, suggesting model imporovement. The same was not true for the random forest model, but the difference in model scores was too small to make a significant difference.

On the test data, the random forest model performed better than the Decision Tree model with an F-score of 0.533 and a higher recall score as well. It also made better predicitions with the majority class of non-severe accidents with 65.3% correct predictions in this class.

However, the decision tree was better at predicting severe cases correctly with 61.4% severe cases predicted correctly.

Overall, the random forest got 63% of the predictions correctly while the decision tree got only 52.3% of the predictions correct. This makes the random forest much better at prediction of severity of cases

13 Conclusion and Recommendations

Conclusion

The aim of this project was to build a machine learning model that can effectively predict the severity of an accident for taxi and ride hailing companies during trip booking. This can allow them to price the trips accordingly, assist them in negotiating insurance prices and plan out better routes.

Using data from the UK Department of Transport, decision trees and random forest models were considered to be the best in predicting accident severities. The data used ranged from 2019 to 2023 and was filtered to include only taxis or private car hires. Overall, random forests were considered to be most suitable for this scenario

Limitations and future improvements

1. Because of limited computing power, some key variables were left out such as latitude and longitude which would give better location information. As such, the goal of route planning may not be achieved with the models created above but the other goals can be met.

2. For the same constraint above, limited hyper parameters were tuned on the different models tested with. For instance, the CATBoost model which I anticipated would provide the best results did not perform as well. There is a possibility that if more hyperparameters were tuned, better models might have been created.

3. An accuracy score of 63% with an fscore of 0.533 is not the best model to use. Adding more recent data (2024) is essential and trying out better models might be critical if the model is to be used in real life.