Final Report – Car Insurance Claims Data Science Project

In the insurance industry, accurately predicting which clients will file claims is crucial. For car insurance, numerous variables describe each policy case, including car features (make, model, airbags, fuel type, sensors, age of the car, torque, and power) and policyholder demographics (age, city, population density).

Predicting if a policy will be claimed involves two types of errors: false negatives (predicting that the policy won't be claimed when it actually is) and false positives (predicting that the policy will be claimed when it isn't). Each type of error incurs a cost, in car insurance, false negatives carry a significantly higher cost than false positives—ranging from 5 to 50 times higher, due to the potential for significant financial loss and poor risk management. Thus, the goal is to develop a model that minimizes a cost function based on these error costs.

To achieve this, I performed data wrangling, exploratory data analysis, preprocessing, modeling, model evaluation, and optimization.

The Data Wrangling

The dataset used for this project is available on Kaggle and contains 44 columns and 58,592 rows. Each row represents a policy record. The dataset includes a policy ID, policy tenure, car features (such as age, make, segment, fuel type, and transmission type), demographic features (such as the age of the policyholder, city, and population density), and a target variable indicating whether the policy was claimed or not.

	policy_id	policy_tenure	age_of_car	age_of_policyholder	area_cluster	population_density	make	segment	model	fuel_type	...
0	ID00001	0.515874	0.05	0.644231	C1	4990	1	A	M1	CNG	...
1	ID00002	0.672619	0.02	0.375000	C2	27003	1	A	M1	CNG	...
2	ID00003	0.841110	0.02	0.384615	C3	4076	1	A	M1	CNG	...
3	ID00004	0.900277	0.11	0.432692	C4	21622	1	C1	M2	Petrol	...
4	ID00005	0.596403	0.11	0.634615	C5	34738	2	A	M3	Petrol	...

In this step, I assessed the data types of each column, identifying integer, float, and categorical types. Additionally, I encoded all Yes-No values to True-False, as this format is preferred by scikit-learn. I split the max power and max torque columns because both contained two quantities in the form of Nm-rpm for max torque and bhp-rpm for max power.

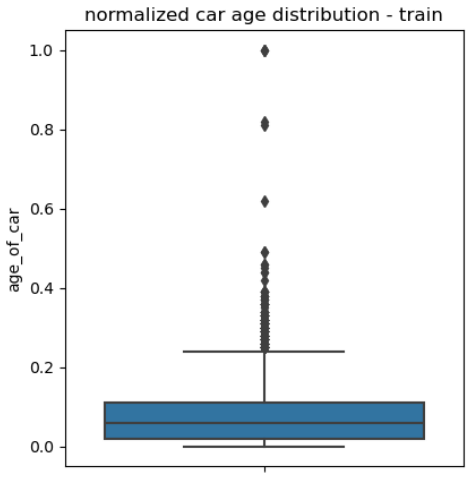
```
#train
pd.set_option('future.no_silent_downcasting', True)
train_convert[boolean_cols] = train_convert[boolean_cols].replace({"Yes": True, "No": False}).infer_objects()
train_convert["is_claim"] = train_convert["is_claim"].replace({1:True, 0:False}).infer_objects()
```

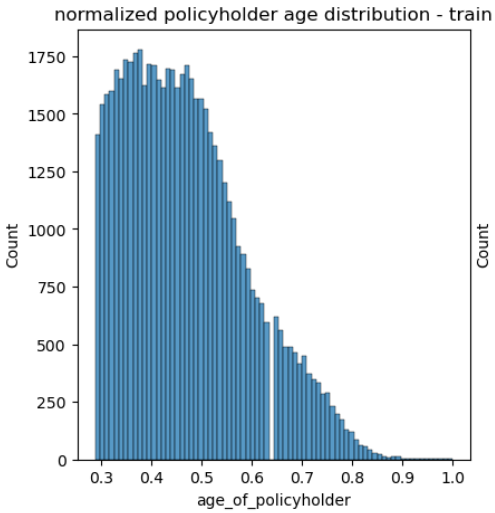
```
#train max_torque
train_convert[["max_torque_Nm", "max_torque_rpm"]] = train_convert["max_torque"].str.split("@", expand = True)
train_convert["max_torque_Nm"] = train_convert["max_torque_Nm"].str.strip("Nm").astype("float")
train_convert["max_torque_rpm"] = train_convert["max_torque_rpm"].str.strip("rpm").astype("float")

#train max_power
train_convert[["max_power_bhp", "max_power_rpm"]] = train_convert["max_power"].str.split("@", expand = True)
train_convert["max_power_bhp"] = train_convert["max_power_bhp"].str.strip("bph").astype("float")
train_convert["max_power_rpm"] = train_convert["max_power_rpm"].str.strip("rpm").astype("float")
```

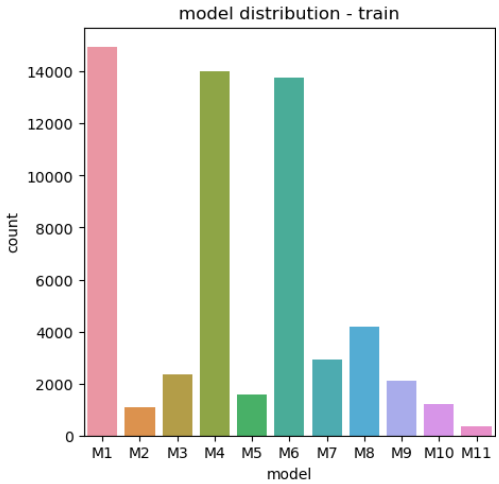
I checked for missing values and found none. I also verified the uniqueness of IDs and checked for duplicates. Fortunately, the dataset was very clean and healthy.

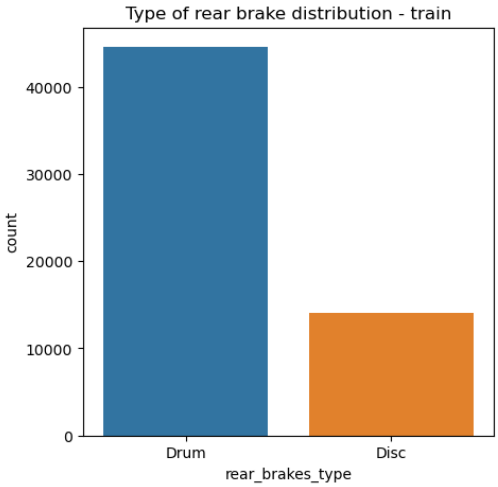
Exploratory Data Analysis

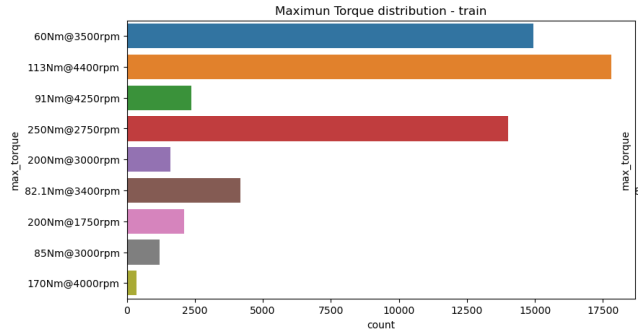
My Exploratory Data Analysis (EDA) consisted of three parts. First, I explored each variable separately, creating numerous plots to visually examine the data. For numerical columns, I made histograms and box plots, while for categorical columns, I used count plots. Among the categorical columns, there were several Boolean columns (True-False), for which I also used count plots, including the target variable. My goal was to understand the distributions, identify outliers, check for normality, and assess imbalance. Notably, the target variable (is_claim) exhibited considerable class imbalance, as the majority of policies were not claimed. Here are some of the most interesting plots:

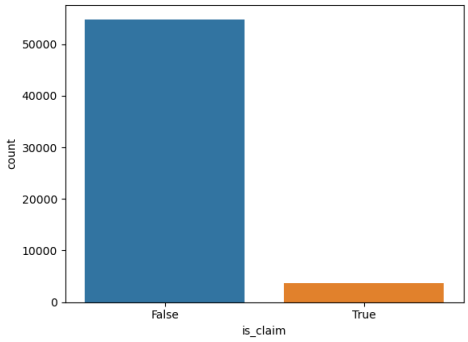




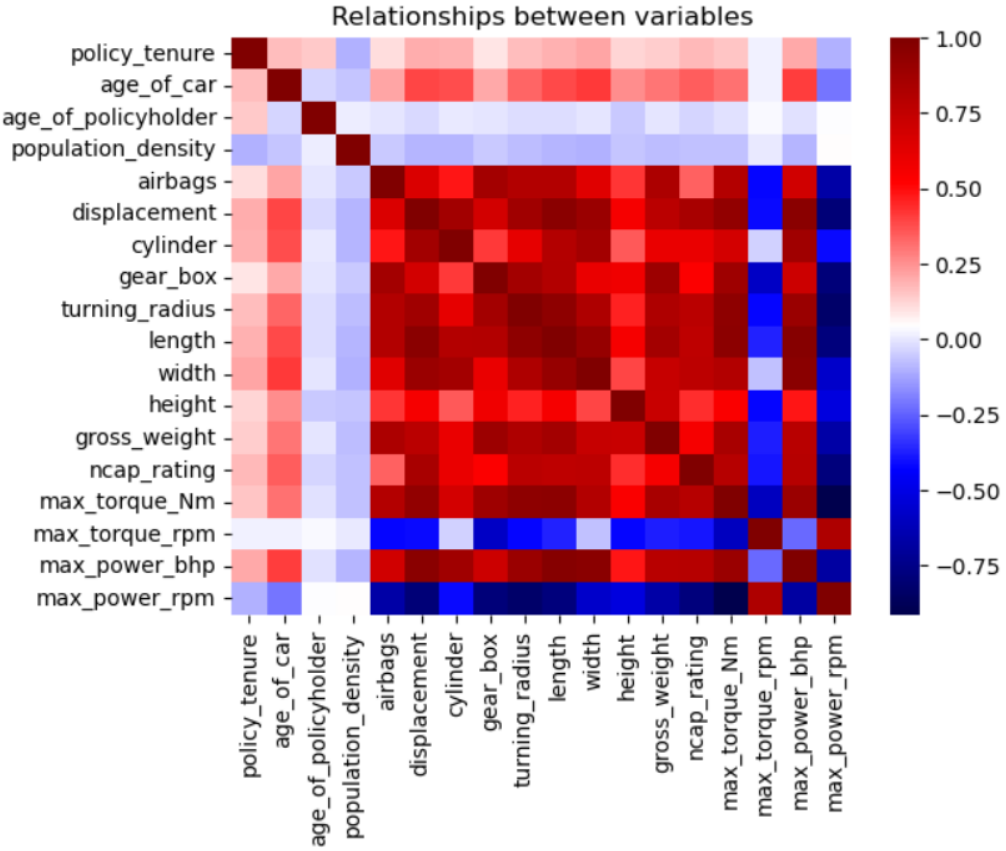






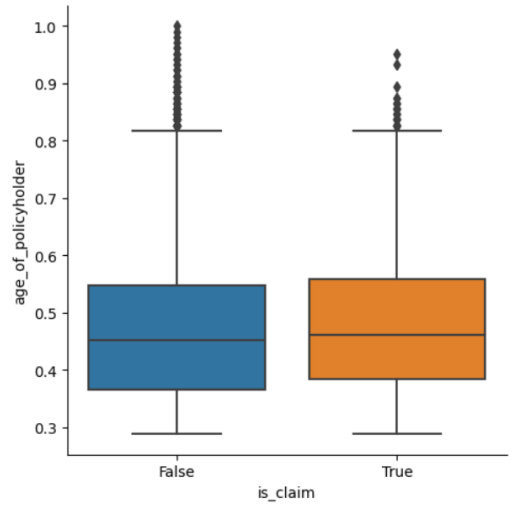


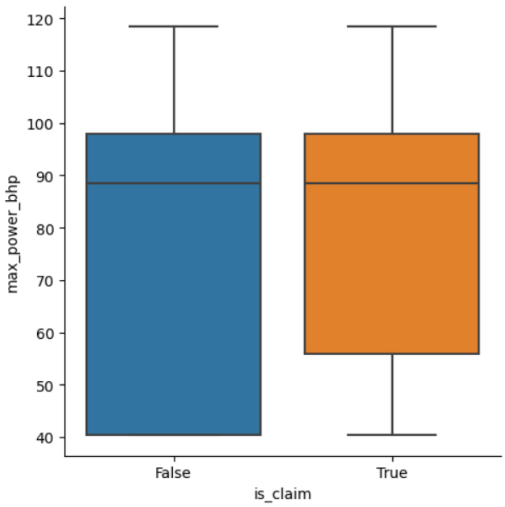
I explored the relationships between the numeric variables using a heatmap. The heatmap revealed several instances of both direct and inverse correlations among the variables:



I also examined the relationship between the target variable and all the predictors. For numeric variables, I used boxplots to compare their distributions based on the target variable. For categorical variables, I used data frames to compare the categories in relation to the target variable:

engine_type	1.5 Turbocharged Revotron	1.0 SCe	K10C	G12B	F8D Petrol Engine	i-DTEC	1.5 L U2 CRDi	K Series Dual jet	1.2 L K Series Engine	1.5 Turbocharged Revotorq	1.2 L K12N Dualjet
is_claim											
False	0.958678	0.94606	0.941529	0.93962	0.938587	0.937086	0.935725	0.931838	0.931633	0.927409	0.925926
True	0.041322	0.05394	0.058471	0.06038	0.061413	0.062914	0.064275	0.068162	0.068367	0.072591	0.074074





is_adjustable_steering	False	True
is_claim		
False	0.940258	0.933288
True	0.059742	0.066712

is_front_fog_lights	False	True
is_claim		
False	0.939426	0.933565
True	0.060574	0.066435

I conducted null hypothesis significance tests to evaluate if there was a statistically significant difference in the proportions when considering the target variable:

```
stat, pval = proportions_ztest([15, 3733], [348 + 15, 54496 + 3733])

if pval < 0.05:
    print("Reject the null hypothesis: The proportions are different")
else:
    print("Fail to reject the null hypothesis: There is not enough evidence to affirm that the proportions are different")
print(pval)
```

```
Fail to reject the null hypothesis: There is not enough evidence to affirm that the proportions are different
0.0769413619339836
```

```
segment  is_claim
A        False       16275
         True         1046
B1       False        3929
         True          244
B2       False       17058
         True         1256
C1       False        3329
         True          228
C2       False       13117
         True          901
Utility  False        1136
         True           73
Name: count, dtype: int64
```

```
stat, pval = proportions_ztest([244, 1256], [3929 + 244, 17058 + 1256])

if pval < 0.05:
    print("Reject the null hypothesis: The proportions are different")
else:
    print("Fail to reject the null hypothesis: There is not enough evidence to affirm that the proportions are different")
print(pval)
```

```
Reject the null hypothesis: The proportions are different
0.018164796314432102
```

I also conducted hypothesis tests for the means. I used parametric tests in cases where variance homogeneity was proven and non-parametric (permutations) tests where the distributions didn't pass a Levene's Variance Homogeneity test:

```
sample1 = train[train["is_claim"] == True].max_torque_Nm
sample2 = train[train["is_claim"] == False].max_torque_Nm

# Variance Homogeneity Test
stat_levene, pval_levene = levene(sample1, sample2)

if pval_levene < 0.05: # If Variance Homogeneity use t test to compare distributions
    print("Variance Homogeneity Assumption")
    stat, pval = ttest_ind(sample1, sample2)
    if pval < 0.05:
        print("Reject the null hypothesis: The distributions means are different")
    else:
        print("Fail to reject the null hypothesis: There is not enough evidence to affirm that the distributions means are
    print(pval)
else: # Use permutations to compare distributions
    print("No Variance Homogeneity")
    perm_distributions_comparison(sample1, sample2)
```

```
No Variance Homogeneity
Fail to reject the null hypothesis: There is not enough evidence to affirm that the distributions means are different
0.154
```

```
sample1 = train[train["is_claim"] == True].policy_tenure
sample2 = train[train["is_claim"] == False].policy_tenure

# Variance Homogeneity Test
stat_levene, pval_levene = levene(sample1, sample2)

if pval_levene < 0.05: # If Variance Homogeneity use t test to compare distributions
    print("Variance Homogeneity Assumption")
    stat, pval = ttest_ind(sample1, sample2)
    if pval < 0.05:
        print("Reject the null hypothesis: The distributions means are different")
    else:
        print("Fail to reject the null hypothesis: There is not enough evidence to affirm that the distributions means are
    print(pval)
else: # Use permutations to compare distributions
    print("No Variance Homogeneity")
    perm_distributions_comparison(sample1, sample2)
```

```
Variance Homogeneity Assumption
Reject the null hypothesis: The distributions means are different
3.0181155800813767e-81
```

The EDA revealed influential predictors and correlations among the variables.

Preprocessing

As the dataset was clean, preprocessing was straightforward. I scaled the features, created dummy variables for categorical columns, and split the data into train and test sets.

pandas.get_dummies:

```
train_dummies = pd.get_dummies(train.drop(columns=["policy_id"]))
```

Scale numeric columns with StandardScaler:

```
train_dummies_numeric = train_dummies.select_dtypes(include=["int", "float"])
train_dummies_non_numeric = train_dummies.select_dtypes(exclude=["int", "float"])
```

```
scaler = StandardScaler()
```

```
train_dummies_numeric_scaled = scaler.fit_transform(train_dummies_numeric)
```

```
train_dummies_numeric_scaled = pd.DataFrame(train_dummies_numeric_scaled, columns=train_dummies_numeric.columns)
train_pre = pd.concat([train_dummies_non_numeric, train_dummies_numeric_scaled], axis=1)
train_pre.shape
```

```
(58592, 119)
```

Split in training and test sets:

```
X = train_pre.drop(columns=["is_claim"])
y = train_pre["is_claim"]
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,stratify=y, random_state=24)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

Modelling, Evaluation and Optimization

Baseline Models

Since this is a classification task, I trained the following baseline models: K-Nearest Neighbors Classifier, Decision Tree Classifier, and Logistic Regression.

```
knn_base = KNeighborsClassifier()
tree_base = DecisionTreeClassifier(random_state = 19)
logreg_base = LogisticRegression(max_iter = 500, random_state = 19)
base_models = [knn_base, tree_base, logreg_base]
```

I evaluated the models using the Classification Report, which includes metrics such as precision, recall, and F1 score, as well as the Confusion Matrix, which shows the number of correctly and incorrectly predicted cases for both classes.

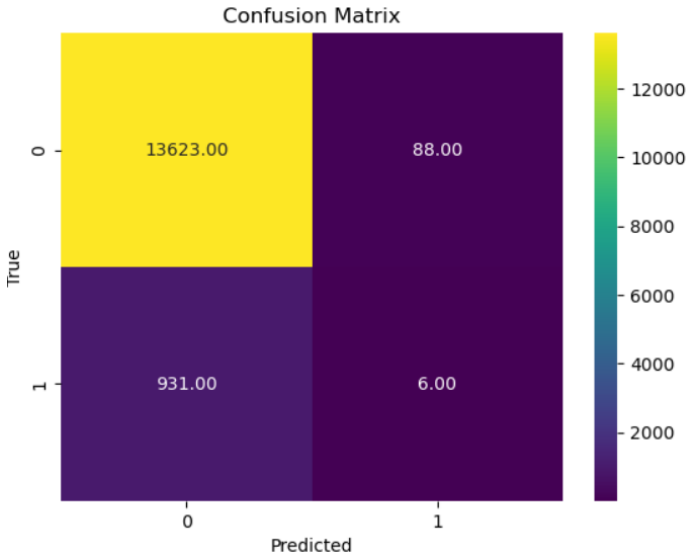
```
KNeighborsClassifier() Classification Report and Confusion Matrix
------------------------------------------------------------------
              precision    recall  f1-score   support

       False       0.94      1.00      0.97     13711
        True       0.15      0.01      0.02       937

    accuracy                           0.93     14648
   macro avg       0.54      0.50      0.49     14648          [[13666    45]
weighted avg       0.89      0.93      0.90     14648           [  929     8]]

DecisionTreeClassifier(random_state=19) Classification Report and Confusion Matrix
---------------------------------------------------------------------------
              precision    recall  f1-score   support

       False       0.94      0.92      0.93     13711
        True       0.07      0.09      0.08       937

    accuracy                           0.87     14648
   macro avg       0.50      0.51      0.50     14648
weighted avg       0.88      0.87      0.88     14648

[[12680  1031]
 [  856    81]]
```

```
LogisticRegression(max_iter=500, random_state=19) Classification Report and Confusion Matrix
--------------------------------------------------------------------------
              precision    recall  f1-score   support

       False       0.94      1.00      0.97     13711
        True       0.00      0.00      0.00       937

    accuracy                           0.94     14648
   macro avg       0.47      0.50      0.48     14648
weighted avg       0.88      0.94      0.91     14648

[[13711     0]
 [  937     0]]
```

Random Forest Classifier (default hyperparameters)

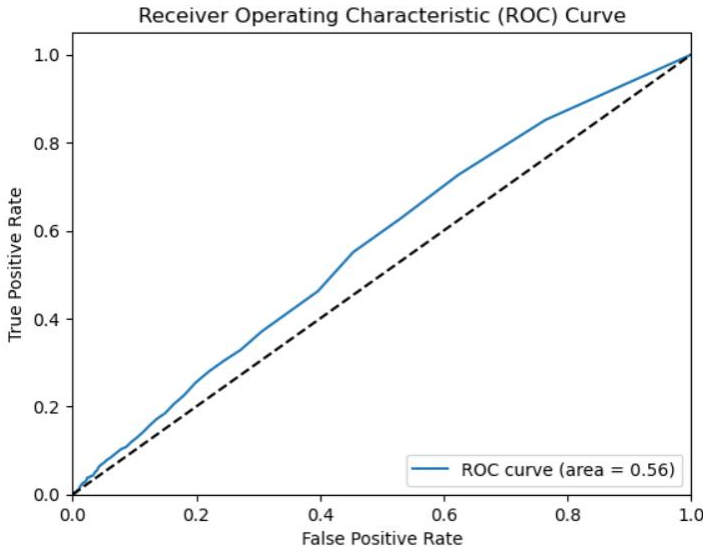
Ensemble algorithms often outperform simple classifiers. Thus, I proceeded with a Random Forest Classifier, an ensemble method that fits multiple decision trees on various subsets of the dataset and averages their predictions to enhance performance. I began with a Random Forest Classifier using default parameters and evaluated its performance using the Classification Report and Confusion Matrix.

```
Classification Report
--------------------------------------------------------
              precision    recall  f1-score   support

       False       0.94      0.99      0.96     13711
        True       0.06      0.01      0.01       937

    accuracy                           0.93     14648
   macro avg       0.50      0.50      0.49     14648
weighted avg       0.88      0.93      0.90     14648
```



Another useful classification metric I used was the Receiver Operating Characteristic (ROC) Curve, which illustrates the model's performance across all classification thresholds. The ROC curve is summarized with the Area Under the Curve (AUC), where a higher AUC

indicates better performance, with 1 being the ideal score. The ROC curve and AUC for the baseline Random Forest Classifier are as follows:



Random Forest Classifier (hyperparameter tuning with Random Search)

Hyperparameters define how a machine learning model learns, making it crucial to find the optimal settings to steer the model's learning in the desired direction. Since the F1 score balances attention between false negatives and false positives, it serves as the scoring metric for Random Search. Random Search is a hyperparameter tuning algorithm that defines a search space as a bounded domain of hyperparameter values and randomly samples points within that domain.

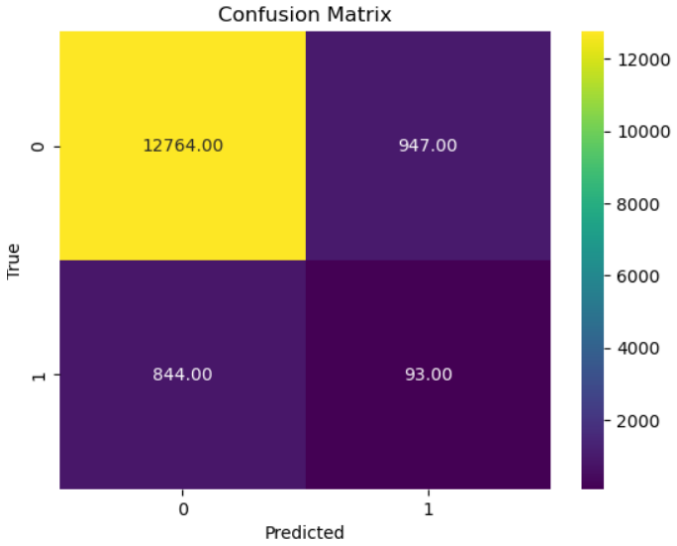
In this process, Random Search performs cross-validation to identify the best hyperparameters. For this task, I set up a 5-fold cross-validation. The estimator used was a Random Forest Classifier, with the F1 score as the scoring metric. The parameter distribution for the Random Search is shown in the following image:

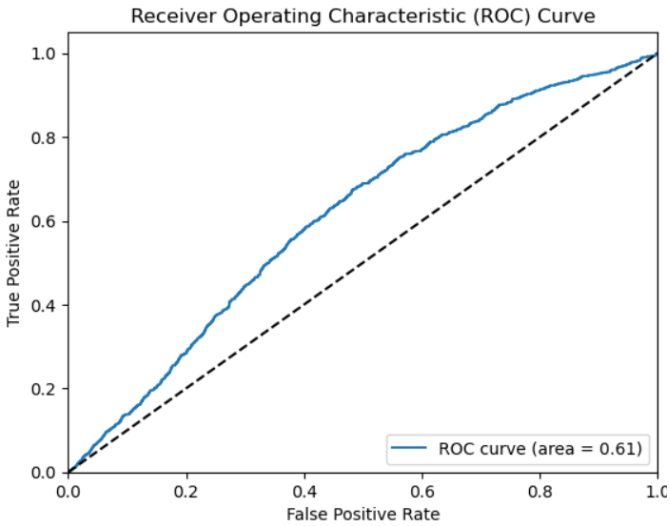
```
RandomizedSearchCV(cv=5, estimator=RandomForestClassifier(random_state=19),
                   param_distributions={'class_weight': ['balanced',
                                                         {0: 1, 1: 10}],
                                        'max_depth': [None, 10, 20],
                                        'min_samples_leaf': [1, 2],
                                        'min_samples_split': [2, 5],
                                        'n_estimators': [100, 200]},
                   scoring='f1')
```

The best hyperparameters identified through the Random Search process, along with the classification report, confusion matrix, ROC curve, and AUC for the best estimator, are presented below:

```
{'n_estimators': 100, 'min_samples_split': 2, 'min_samples_leaf': 1,
 'max_depth': 20, 'class_weight': 'balanced'}
```

	precision	recall	f1-score	support
False	0.94	0.93	0.93	13711
True	0.09	0.10	0.09	937
accuracy			0.88	14648
macro avg	0.51	0.52	0.51	14648
weighted avg	0.88	0.88	0.88	14648

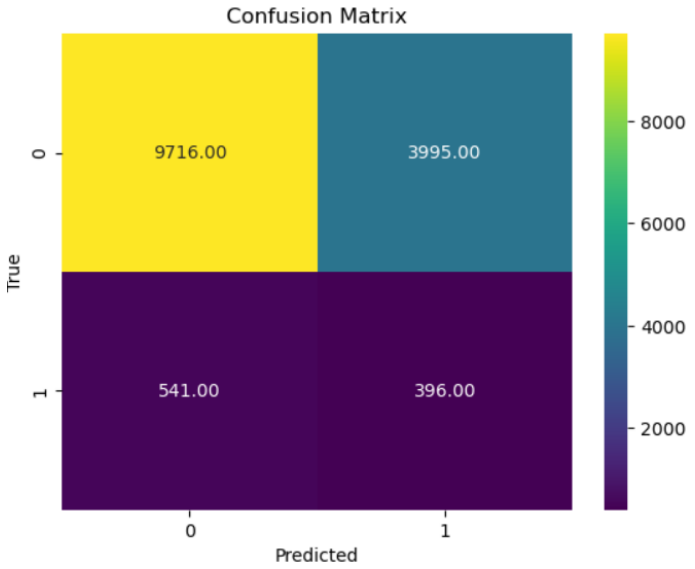


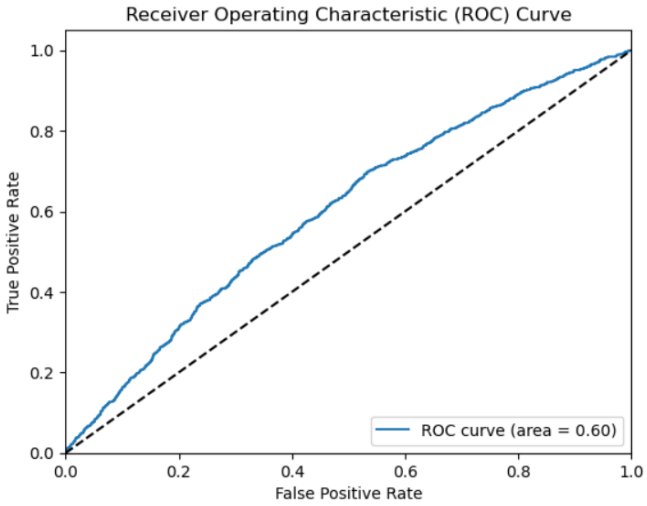


Extreme Gradient Boosting (default hyperparameters)

Extreme Gradient Boosting (XGBoost) is a state-of-the-art ensemble method known for its superior performance in various regression and classification tasks. Typically, it uses trees as base estimators and employs boosting, an iterative approach to optimize performance. The XGBoost library offers an API compatible with the scikit-learn environment. I trained an XGBClassifier with the scale_pos_weight parameter to address class imbalance in the dataset. The evaluation metrics for this model are as follows:

	precision	recall	f1-score	support
False	0.95	0.71	0.81	13711
True	0.09	0.42	0.15	937
accuracy			0.69	14648
macro avg	0.52	0.57	0.48	14648
weighted avg	0.89	0.69	0.77	14648



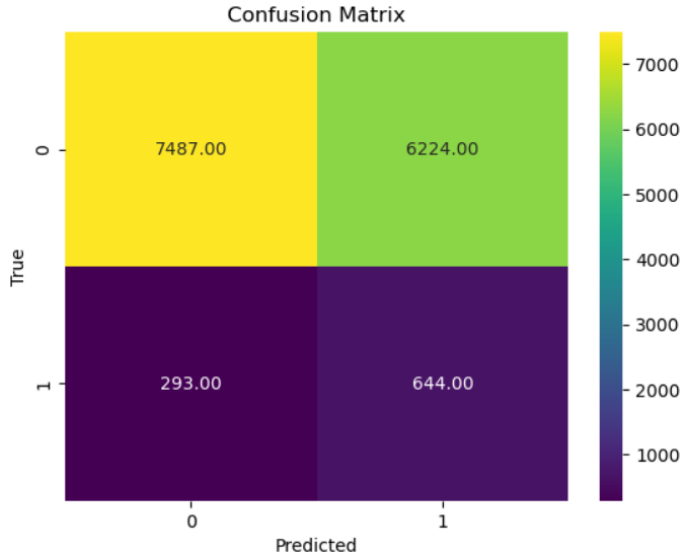


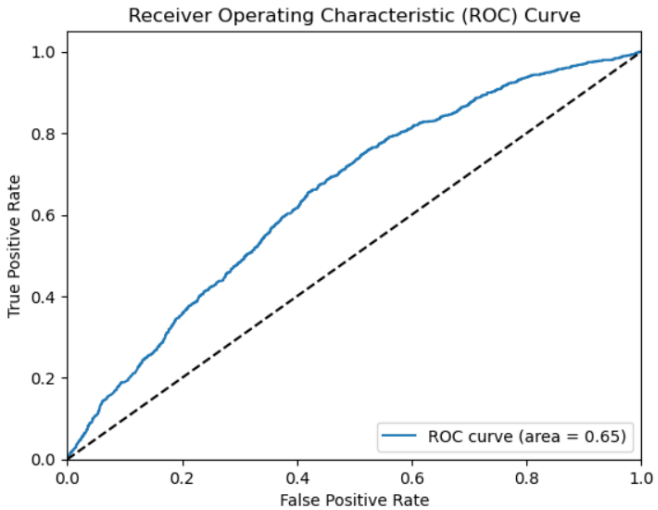
Extreme Gradient Boosting (hyperparameter tuning with Random Search)

Similar to the Random Forest Classifier, the XGBoost Classifier can significantly benefit from hyperparameter tuning. To optimize the performance of the XGBoost model, I conducted a Random Search for hyperparameter tuning. The scoring metric used for the was the F1 score. The following outlines the parameter distributions and results:

```python
param_dist = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [3, 4, 5, 6, 7, 8, 9, 10],
    'learning_rate': [0.01, 0.05, 0.1, 0.15, 0.2],
    'gamma': [0, 0.1, 0.2, 0.3, 0.4, 0.5]
}
```

```
{'n_estimators': 200, 'max_depth': 6, 'learning_rate': 0.01, 'gamma': 0}
              precision    recall  f1-score   support

       False       0.96      0.55      0.70     13711
        True       0.09      0.69      0.17       937

    accuracy                           0.56     14648
   macro avg       0.53      0.62      0.43     14648
weighted avg       0.91      0.56      0.66     14648
```





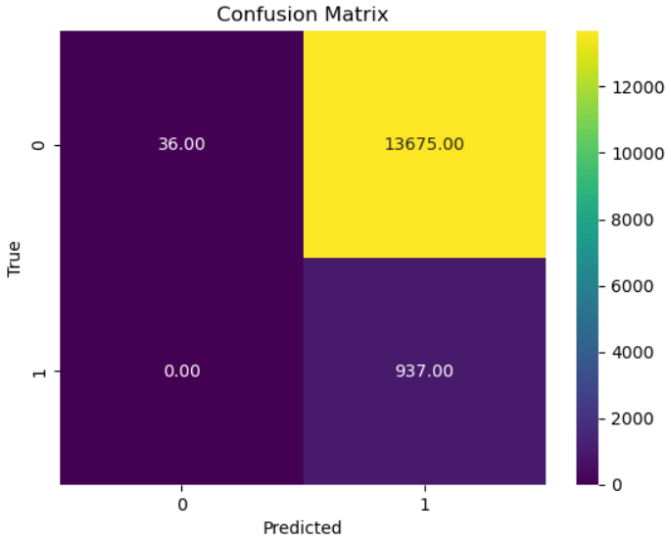
XGB + Synthetic Minority Over-sampling Technique (SMOTE)

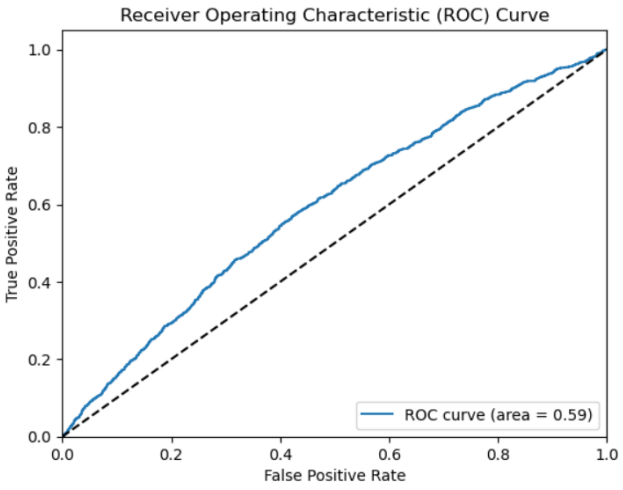
SMOTE is a powerful technique used to address class imbalance in machine learning datasets by generating synthetic samples for the minority class. This helps to balance the dataset and can lead to better model performance, especially in cases where the target variable is heavily skewed. In this step, I applied SMOTE to resample the data and then trained an XGBoost Classifier on the resampled dataset:

```
smote = SMOTE()
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

The metrics for this base XGB with SMOTE are the following:

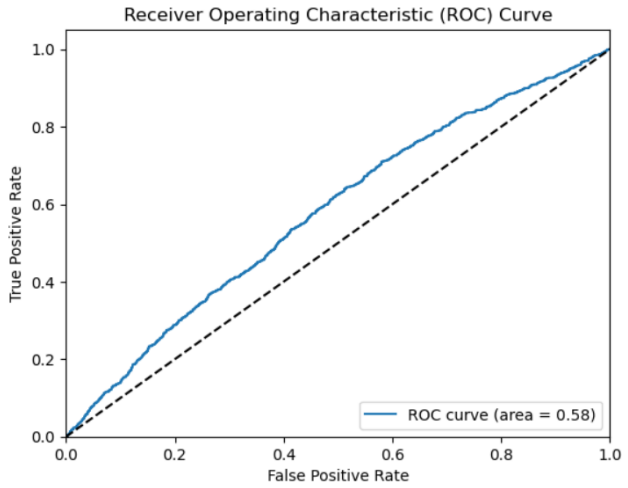
	precision	recall	f1-score	support
False	1.00	0.00	0.01	13711
True	0.06	1.00	0.12	937
accuracy			0.07	14648
macro avg	0.53	0.50	0.06	14648
weighted avg	0.94	0.07	0.01	14648





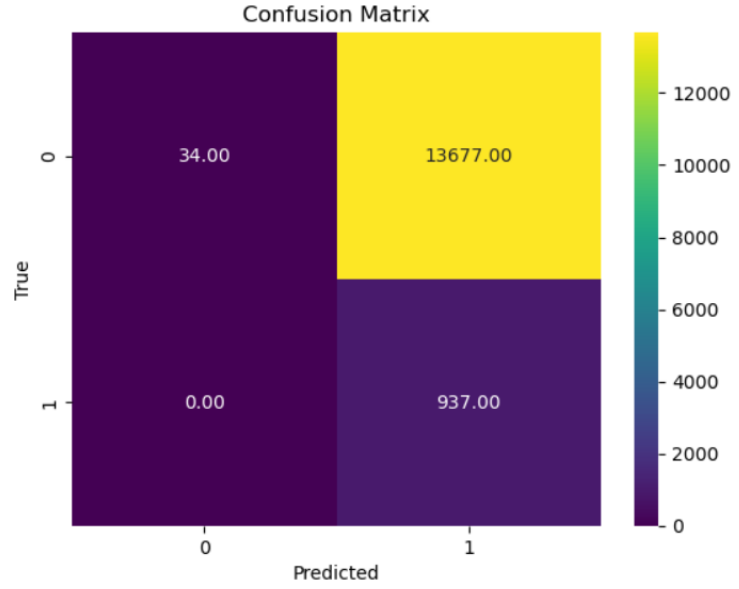
XGB + SMOTE (hyperparameter tuning with Random Search)

The initial XGB + SMOTE model resulted in an extreme reduction of false negatives, achieving zero false negatives. However, this also led to a significant increase in false positives. To address this, I performed hyperparameter tuning using Random Search to find the optimal parameters that balance both false positives and false negatives more effectively.



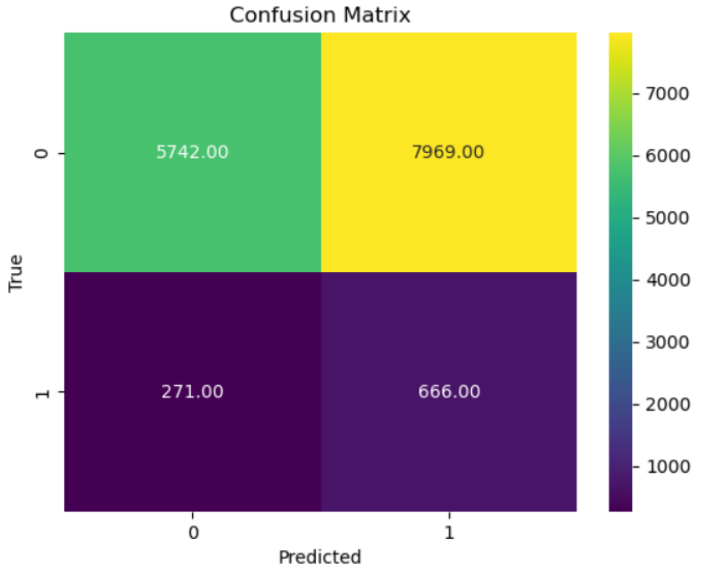
To optimize the classification performance of the model, I explored different classification thresholds. Initially, I identified the optimal threshold using the precision-recall curve. This threshold maximizes the model's ability to balance precision and recall, crucial for minimizing false positives and false negatives in the predictions:

	precision	recall	f1-score	support
False	1.00	0.00	0.00	13711
True	0.06	1.00	0.12	937
accuracy			0.07	14648
macro avg	0.53	0.50	0.06	14648
weighted avg	0.94	0.07	0.01	14648



To further refine the model's performance, I explored the optimal classification threshold using the Receiver Operating Characteristic (ROC) curve. This threshold maximizes the true positive rate while minimizing the false positive rate, enhancing the model's ability to discriminate between classes effectively:

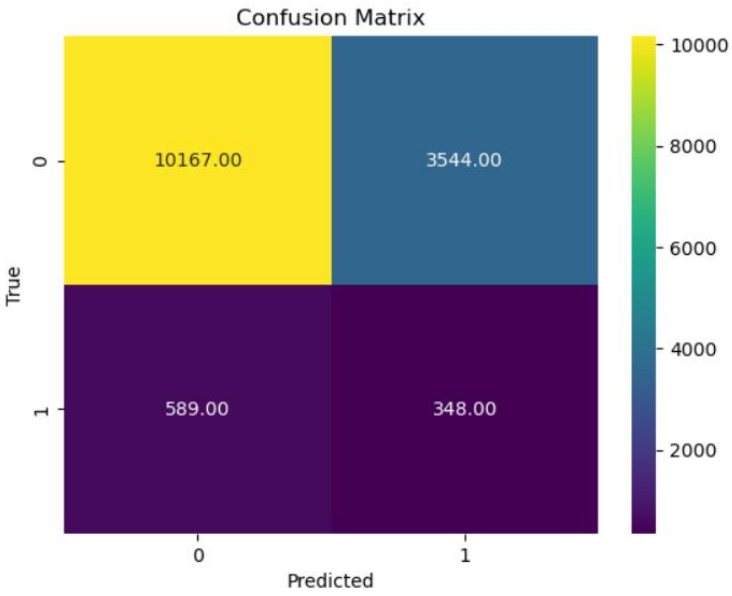
	precision	recall	f1-score	support
False	0.95	0.42	0.58	13711
True	0.08	0.71	0.14	937
accuracy			0.44	14648
macro avg	0.52	0.56	0.36	14648
weighted avg	0.90	0.44	0.55	14648

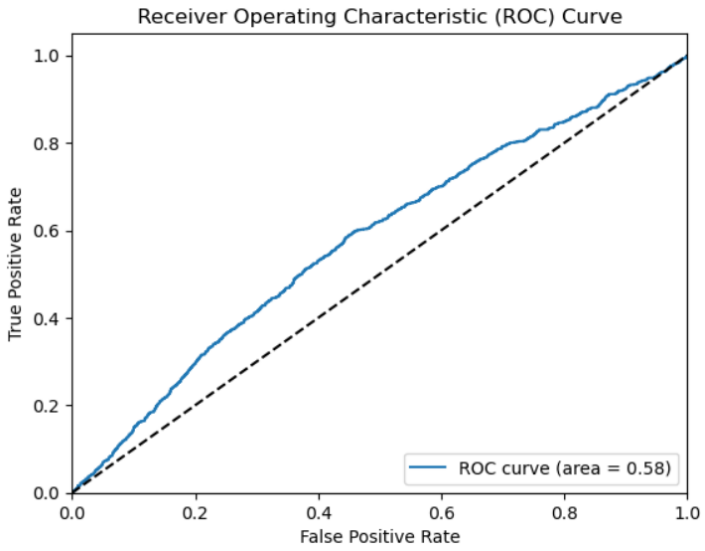


Neural Network: Multilayer Perceptron Classifier (MPC)

The Multilayer Perceptron (MLP) Classifier is a fundamental neural network architecture capable of learning complex relationships in data, particularly useful for solving non-linearly separable problems. In this project, I employed the MLP Classifier provided by scikit-learn to explore its performance on the insurance claim prediction task:

	precision	recall	f1-score	support
False	0.95	0.74	0.83	13711
True	0.09	0.37	0.14	937
accuracy			0.72	14648
macro avg	0.52	0.56	0.49	14648
weighted avg	0.89	0.72	0.79	14648





Business Modelling

To determine the optimal model for a Car Insurance company, it is crucial to consider the financial implications of prediction errors. In this context, false positives and false negatives carry different costs for the company. A cost function was designed to quantify these costs:

C = (CFP * FP) + (CFN * FN)

Where:

- CFP: The cost of a false positive
- CFN: The cost of a false negative
- FP: The number of false positives
- FN: The number of false negatives

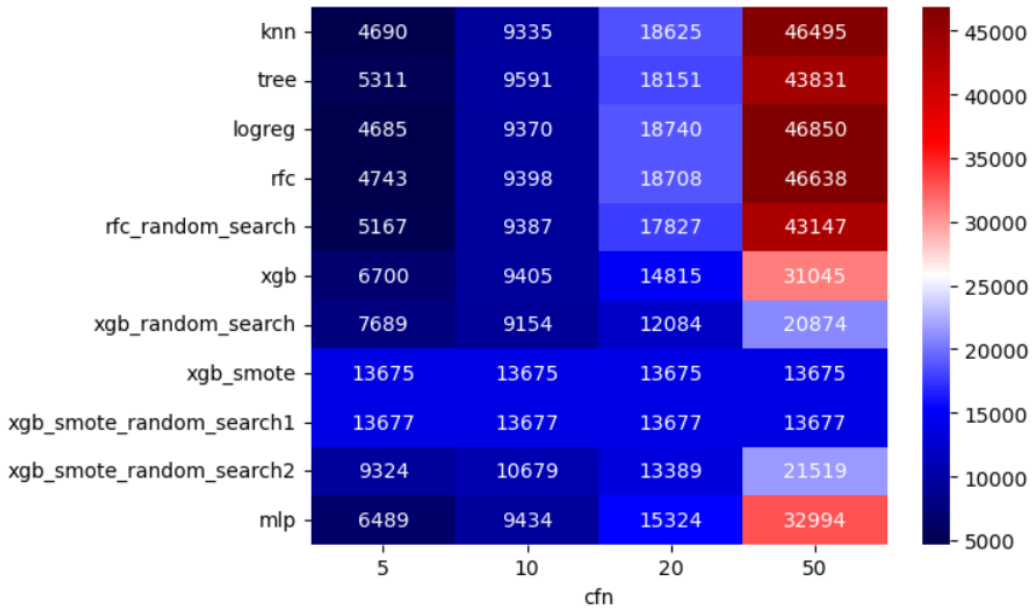
```
def business_cost(cfp, cfn, fp, fn):
    C = (cfp * fp + cfn * fn)
    return C
```

Here is a summary of the FP and FN counts for each developed model:

```
base_knn = {"fp": 45, "fn": 929}
base_tree = {"fp": 1031, "fn": 856}
base_logreg = {"fp": 0, "fn": 937}

base_rfc = {"fp": 88, "fn": 931}
rfc_random_search = {"fp": 947, "fn": 844}

base_xgb = {"fp": 3995, "fn": 541}
xgb_random_search = {"fp": 6224, "fn": 293}

xgb_smote = {"fp": 13675, "fn": 0}
xgb_smote_random_search1 = {"fp": 13677, "fn": 0}
xgb_smote_random_search2 = {"fp": 7969, "fn": 271}

mlp = {"fp": 3544, "fn": 589}
```

In the context of insurance claims, the cost associated with false negatives (CFN) is typically significantly higher than that of false positives (CFP). To assess the impact of varying these costs, I explored different ratios of CFN to CFP, ranging from 5 to 50. The results of the cost function, represented as a heatmap, illustrate how different ratios influence the overall cost in this scenario:

```
cfp = 1
cfn_list = [5, 10, 20, 50]
```



- When the CFN is 5 times the CFP, the best models are: 1 Base Logistic Regression, 2 Base KNN, 3 Base Random Forest Classifier.
- When the CFN is 10 times the CFP, the best models are: 1 XGB (Random Search), 2 Base KNN, 3 Base Logistic Regression.
- When the CFN is 20 times the CFP, the best models are: 1 XGB (Random Search), 2 XGB (SMOTE + Random Search2), 3 XGB (SMOTE).
- When the CFN is 50 times the CFP, the best models are: 1 XGB (SMOTE), 2 XGB (SMOTE + Random Search1), 3 XGB (Random Search).

Based on these findings, the most probable scenarios, occurring with CFN values between 10- and 20-times CFP, suggest XGB (Random Search) as the optimal model. This model consistently demonstrates strong performance, particularly highlighted by its highest Area Under the Curve (AUC) in the Receiver Operating Characteristic (ROC) analysis, showcasing robustness across various decision thresholds.

Conclusion

The goal of this project was to develop a model for predicting car insurance claims based on policy, car, and demographic features. Given the nature of insurance claims, where false negatives incur significantly higher costs than false positives, the focus was on optimizing for this scenario.

Using a dataset comprising 44 features and 58,592 policy records, I conducted thorough data wrangling and exploratory data analysis. This process uncovered meaningful patterns and relationships within the data, guiding subsequent preprocessing steps to prepare for modeling.

I trained and evaluated several models, beginning with three baseline models and progressing to ensemble methods such as Random Forest and XGBoost, which were further refined through hyperparameter optimization using Random Search. Additionally, I employed the SMOTE technique to address class imbalance and explored the performance of a Multilayer Perceptron Classifier.

Evaluation metrics, particularly ROC and AUC, were used to compare model performance. The XGBoost Classifier, optimized through Random Search, emerged as the superior model. It consistently demonstrated the highest Area Under the Curve (AUC) in the ROC analysis, showcasing robustness across various decision thresholds and effectively minimizing our targeted cost function.

As a final recommendation, improving the dataset's class imbalance would likely enhance model performance further. Additionally, with increased computational resources, conducting a Complete Grid Search for hyperparameter tuning could yield even more refined results. Lastly, establishing a precise business cost function with exact ratios between false positives and false negatives would provide deeper insights for decision-making.