**Name**: Alam Rithika

**Matriculation Number**: 2202550F

**Chosen Dataset**: Claiming Car Insurance

***Introduction:***

Predicting if car insurance can be claimed or not is a crucial task in the insurance industry, as it helps insurance companies assess the risk associated with each policyholder and make informed decisions regarding coverage and premiums. The ability to accurately predict whether a policyholder will file a claim in the next 6 months can significantly impact an insurance company's profitability and risk management strategies.

The primary objective of this project is to develop a predictive model that can effectively forecast whether a policyholder will file a car insurance claim within the next 6 months or not. By achieving this, insurance companies can proactively adjust their policies, assess potential risks, and optimize claim processing procedures, leading to more efficient operations and better customer satisfaction.

The dataset used for this project contains a comprehensive set of attributes related to policyholders and their respective car insurance policies. Each data point in the dataset corresponds to a policyholder, and the target variable indicates whether the policyholder filed a claim within the next 6 months or remained claim-free during that period.

Target variable ---> Is\_Claim: Binary variable indicating whether the policyholder filed a claim in the next 6 months (1 - Claimed, 0 - Not claimed).

***Data Exploration:***

In this data exploration phase, we used Python libraries such as NumPy, Pandas, Matplotlib, Seaborn, and Plotly to analyze the car insurance dataset. The dataset, stored as a Pandas DataFrame named df, contains information about policyholders, including policy tenure, age of the car, age of the car owner, city population density, car make and model, power, engine type, and more. The objective is to predict whether a policyholder will file a car insurance claim in the next six months.

1. Checking the Data

We began by examining the first and last five rows of the dataset using the head() and tail() functions. These operations allowed us to get a glimpse of the dataset's structure and content, which is crucial for an initial assessment and identifying any potential issues.Next, we checked the shape of the data using the shape attribute, which gives us the number of rows and columns in the dataset. This information helps us understand the size and complexity of the data, guiding subsequent analysis steps.

1. Understanding the Data Structure

To understand the dataset's structure, we inspected the column names using the columns attribute. This helped us identify the features available for analysis and provided insight into the dataset's organization. Additionally, we examined the data types of each column using the dtypes attribute, sorting the data types in ascending order. This categorization into numerical and categorical features is essential for data exploration and preprocessing.

1. Checking Data Summary

To obtain a concise summary of the DataFrame, we used the info() function. This summary provided information on the number of non-null values and data types for each column, helping us identify any missing data and gaining an overall understanding of the dataset's composition.

1. Statistical Summary of Numerical Features

We generated a comprehensive statistical summary for the numerical features using the describe() function. This summary includes essential statistics such as count, mean, standard deviation, minimum, 25th percentile (Q1), median (50th percentile or Q2), 75th percentile (Q3), and maximum values for each numerical attribute. The statistical summary helps us understand the distribution, central tendency, and range of the numerical features.

1. Statistical Summary of Categorical Features

To explore the categorical features, we utilized the describe(include="O") function. This summary provided us with the count, number of unique categories, the most frequent category, and its frequency for each categorical attribute. Understanding the distribution of categorical data is vital for drawing insights and making informed decisions during analysis.

1. Checking for Duplicate Records

In this part of the analysis, we aimed to identify and handle any duplicate records present in the DataFrame df. Duplicate records can negatively impact the integrity and accuracy of the analysis results, leading to skewed insights or model bias. To check for duplicate records, we used the duplicated() function, which returns a boolean array indicating whether each row is a duplicate of a previous row in the DataFrame. The variable duplicates stores this boolean array.Next, we checked if duplicates exist in the DataFrame by using the any() function on the duplicates variable. The any() function returns True if any element in the boolean array is True, indicating the presence of duplicate records.

Output Interpretation: The output of the first code snippet would be either "Duplicates exist in the DataFrame." or "No duplicates found in the DataFrame." based on whether duplicates are present or not.The output of the second code snippet, which directly checks for the existence of duplicates in the DataFrame, will return a boolean value. If True, it indicates the presence of duplicate records, and if False, it indicates the absence of duplicates.

Importance of Handling Duplicates: Handling duplicate records is essential to maintain data integrity and obtain accurate insights. Duplicate records can lead to erroneous statistical summaries, biased analysis, and inflated model performance. If duplicates are found in the dataset, appropriate actions should be taken, such as removing duplicates or consolidating information to ensure the dataset accurately represents the underlying data distribution.

1. Checking the Number of Unique Values

In this section, we aimed to understand the cardinality of each feature in the dataset. Cardinality refers to the number of unique values present in a feature. Understanding the number of unique values is essential to assess the diversity and variability of the data, especially for categorical features. For numerical features, we used the nunique() function directly on the DataFrame df, which returns the number of unique values in each numerical feature. This gives us insight into the diversity and distribution of values within each numerical feature.For categorical features, we applied the nunique() function to the subset of the DataFrame containing only the object and category data types. The resulting unique\_value\_counts variable holds the number of unique values for each categorical feature. Additionally, we printed the count of each categorical variable using a loop. The loop iterates over all categorical columns (i.e., object data type columns) and displays the value counts of each unique category. This provides an overview of the frequency distribution of each categorical variable.

Output Interpretation: The first output shows the number of unique values for each **numerical** feature. For example, if "age\_of\_car" has 49 unique values, it means there are 49 distinct car ages in the dataset. The second output, represented by the unique\_value\_counts variable, displays the number of unique values for each **categorical** feature. For instance, if "gender" has 2 unique values, it indicates there are two gender categories in the dataset (e.g., male and female). The third output consists of value counts for each unique category within every categorical variable. This allows us to observe the frequency distribution of each category, giving insights into the prevalence of different categories in the dataset.

Importance of Checking Unique Values: Checking the number of unique values for each feature helps in identifying features with low variability, which may not contribute significantly to the analysis or modeling process. It also helps in understanding the nature of categorical data and ensures that the dataset contains sufficient diversity to represent real-world scenarios accurately. Additionally, this step aids in identifying potential data quality issues, such as mislabeled categories or data entry errors.

1. Correlation Matrix and Heatmap

In this section, we performed a correlation analysis on the dataset to explore the relationships between numerical features. Correlation is a statistical measure that quantifies the strength and direction of a linear relationship between two variables. It helps us understand how different features in the dataset are related to each other, which is crucial for feature selection and dimensionality reduction.

Correlation Matrix: The first step is to calculate the correlation matrix using the corr() function on the DataFrame df. The correlation matrix is a square matrix where each cell represents the correlation coefficient between two variables. The correlation coefficient ranges from -1 to +1, where -1 indicates a perfect negative correlation, +1 indicates a perfect positive correlation, and 0 indicates no linear correlation.

Heatmap: To visualize the correlation matrix effectively, we used a heatmap, which is a graphical representation of data where individual values in a matrix are represented as colors. We utilized the sns.heatmap() function from the Seaborn library to plot the heatmap. In the heatmap, darker colors represent stronger positive correlations, while lighter colors represent stronger negative correlations.

Output Interpretation: The correlation matrix provides a comprehensive overview of the relationships between numerical features. By inspecting the matrix, we can observe which pairs of features are positively correlated, negatively correlated, or have little to no correlation. The heatmap further enhances the understanding of the correlation matrix by visually representing the correlation coefficients using a color scale. The intensity of the color indicates the strength of the correlation, making it easier to identify strong correlations (both positive and negative) among features.

Importance of Checking Correlations: Understanding the correlations between features is vital in various data analysis and machine learning tasks. High correlations between independent features may lead to multicollinearity, which can negatively impact the performance of some models. By examining the correlation matrix and heatmap, we can identify potential collinear features and decide whether to remove or combine them.

Additionally, correlation analysis helps in feature selection for predictive modeling. Features with low correlations to the target variable may have less predictive power and can be excluded from the modeling process. On the other hand, highly correlated features may be combined or used as indicators of important patterns in the data.Overall, checking the correlation matrix and heatmap allows us to gain valuable insights into the relationships between features, guiding us in making informed decisions during data preprocessing and model building.

The output:

A screenshot of a computer

Description automatically generated A table of numbers with numbers

Description automatically generated

A screenshot of a computer screen

Description automatically generated

***Data Transformation:***

1. Standardizing Age of Car and Age of Policyholder

The "age\_of\_car" and "age\_of\_policyholder" features represent the age of the car and the age of the policyholder, respectively. To standardize these features and make them more interpretable, we multiplied them by 10 and 100, respectively. This transformation scales the values to a more suitable range for analysis.

The values of these features are given as decimal fractions (between 0 and 1) & (between 0 and 100), which might not be as intuitive for interpretation and analysis.

- Multiplying by 10 for Age of Car: The "age\_of\_car" values were multiplied by 10 to transform them into more familiar units, such as years. Since the original values were fractions between 0 and 1, multiplying them by 10 converts them into a range of 0 to 10, representing the number of years since the car's manufacturing date. For example, a value of 0.05 in the original "age\_of\_car" column would be transformed to 0.05 \* 10 = 0.5, indicating that the car is approximately half a year old.

Before Transformation: After Transformation:

A screenshot of a table

Description automatically generated A screenshot of a cell phone

Description automatically generated

- Multiplying by 100 for Age of Policyholder: Similarly, the "age\_of\_policyholder" values were multiplied by 100 to convert them into years. The original values were fractions between 0 and 1, so multiplying them by 100 transforms them into a range of 0 to 100, representing the age of the policyholder in years. For instance, a value of 0.644231 in the original "age\_of\_policyholder" column would be transformed to 0.644231 \* 100 = 64.42, indicating that the policyholder is approximately 64 years and 4 months old.

Before Transformation: After Transformation:

A screenshot of a computer

Description automatically generated A screenshot of a cell phone

Description automatically generated

Purpose of doing this data transformation (assumptions included) :

- Improved Interpretability: By converting the "age\_of\_car" and "age\_of\_policyholder" values into more familiar units (years), we make the data more interpretable for stakeholders and analysts. Understanding the ages of the car and policyholder in years provides clearer insights into their relevance to car insurance claim prediction.

- Consistent Scaling: Standardizing the values of "age\_of\_car" and "age\_of\_policyholder" ensures that both features are on the same scale, which is important for many machine learning algorithms. Having consistent scaling prevents one feature from dominating the analysis or model training process due to differences in their original scale.

- Enhanced Model Performance: Standardization can improve the performance of machine learning models, especially those that are sensitive to the scale of the input features. Many algorithms, such as support vector machines and k-nearest neighbors, perform better when the features are on similar scales.

1. Removing columns policy\_id and policy\_tenure

The objective of dropping the "policy\_id" and "policy\_tenure" columns from the car insurance dataset is to enhance the data's relevancy and reduce unnecessary complexity in the analysis. The "policy\_id" column serves as a unique identifier for each policyholder, offering no predictive value regarding insurance claims. Since each policyholder is assigned a distinct identifier, this information does not contribute meaningfully to the analysis or modeling process.

Similarly, the "policy\_tenure" column represents the tenure or duration of each policy, which is not directly related to the prediction of insurance claims. While policy tenure could be an interesting factor in some analyses, our primary focus is on the features directly linked to potential claims, such as car age, policyholder age, and other relevant attributes.

Purpose of doing this data transformation (assumptions included) :

The decision to drop these columns is grounded in the assumption that the "policy\_id" and "policy\_tenure" information does not offer any insights relevant to predicting insurance claims. We made this assumption based on the understanding that these identifiers and duration values are not correlated with the likelihood of a policyholder filing a claim in the next six months.

By removing these irrelevant columns, we streamline the dataset and reduce its dimensionality, making it more manageable and easier to interpret. This process also facilitates improved model performance, as the absence of unrelated features prevents the model from considering noise or irrelevant information. Ultimately, the dropping of these unnecessary columns aligns with the overarching goal of optimizing the dataset for meaningful analysis and accurate predictive modelling.

1. String manipulation for columns max\_torque and max\_power

String manipulation is performed on the "max\_torque" and "max\_power" features to transform them into a suitable format for numerical analysis. These features represent the maximum torque and maximum power generated by the car and are currently presented as strings in the format "Nm@rpm" and "bhp@rpm," respectively.

The objective of this string manipulation is to extract the numeric values for torque and power and convert them into floating-point numbers, allowing us to perform numerical operations on these features accurately. To achieve this, the code uses a for loop to iterate through the values in the "max\_torque" and "max\_power" columns.

- For the "max\_torque" column: The code uses the "split" method to split the string at the "@" symbol, separating the numeric torque value from the rpm value. The first element of the resulting list contains the numeric torque value, and we use slicing ([:-2]) to remove the last two characters, which represent the unit "Nm." The second element of the list contains the rpm value, but we do not need it for further analysis. The extracted numeric torque values are then appended to the "max\_torque" list.

- For the "max\_power" column: The code follows the same procedure as above to extract the numeric power value from the string.Additionally, the code uses slicing ([:-3]) to remove the last three characters, which represent the unit "bhp."The extracted numeric power values are then appended to the "max\_power" list.

Before Transformation: After Transformation:

A screenshot of a computer

Description automatically generated A screenshot of a screenshot of a number

Description automatically generated

The output of this string manipulation is two new lists, "max\_torque" and "max\_power," containing the extracted numeric torque and power values, respectively. These lists will replace the original columns in the dataset, allowing us to work with these features as numerical values for further analysis, visualization, and modeling.

Purpose of doing this data transformation (assumptions included) :

- Numerical Analysis: The original format of "max\_torque" and "max\_power" is presented as strings with units (Nm@rpm and bhp@rpm, respectively). Such textual representations are not suitable for numerical analysis and calculations. By extracting and converting the numeric values from these strings into floating-point numbers, we make these features compatible with various numerical operations, statistical analyses, and machine learning algorithms.

- Feature Engineering: The extracted numeric torque and power values provide more granular and relevant information about the car's performance characteristics. By transforming these features into numerical format, we can engineer new features or derive meaningful insights based on the specific values of torque and power, such as creating torque-to-weight or power-to-displacement ratios.

- Model Compatibility: Many machine learning algorithms require numeric input features to make predictions effectively. By converting "max\_torque" and "max\_power" into numerical format, we ensure that these features can be used as inputs for classification models

- Improved Visualization: Numeric representations of "max\_torque" and "max\_power" can be effectively visualized using various plotting techniques, such as scatter plots or line charts. This allows us to explore the relationships between torque, power, and other features in the dataset and gain a better understanding of the data's underlying patterns.

Overall, the data transformation through string manipulation enhances the data's analytical capabilities, model compatibility, and visualization potential. By converting the textual format of "max\_torque" and "max\_power" into numeric values, we enable more robust analysis, exploration, and modeling of the car performance dataset, leading to more insightful and accurate results in predicting and understanding the car's performance characteristics.

***EDA***:

1. Univariate Analysis (Numerical)

I have created a function that is designed to perform univariate exploratory data analysis (EDA) on the DataFrame named 'df.' The primary objective of this function is to gain insights into the numerical columns of the dataset and identify any data issues or potential outliers.

Upon executing the code, the function systematically goes through each numerical column in the DataFrame 'data.' It begins by displaying the name of the column as a header to distinguish the analysis for each feature. The function then proceeds to provide a comprehensive statistical summary of the feature, including essential metrics such as mean, standard deviation, minimum, maximum, quartiles, and skewness.

Following the statistical summary, the code generates a box plot for each numerical column. This visualization aids in understanding the data distribution and helps identify any data points that might be considered outliers. (Outliers are data points that lie far from the bulk of the data and may indicate potential data quality issues.) . Additionally, the function implements the Interquartile Range (IQR) method to detect and count the number of outliers in each column. The IQR method is a robust technique used to identify data points that deviate significantly from the median of the data.

A number of numbers and symbols

Description automatically generated

A graph of a person with a graph

Description automatically generated

A black and white text

Description automatically generated

1. Univariate Analysis (Categorical)

The provided code contains a function named 'univariate\_eda\_cat' that performs univariate exploratory data analysis (EDA) on categorical columns within a pandas DataFrame named 'df.' The objective of this function is to gain insights into the distribution and unique values of each categorical feature in the dataset.

Upon executing the code, the function iterates through all columns in the DataFrame 'data' and checks if each column is categorical. If a column is found to be categorical, the function proceeds with the analysis for that specific feature

The first step of the analysis involves counting the number of unique values in the categorical column and displaying their respective counts. This provides a summary of the cardinality of each category within the feature. The function creates a dictionary to store each unique value as a key and its corresponding count as the value, and then prints this information.

Next, the code generates a bar plot for the categorical column, illustrating the distribution of each category. The x-axis represents the unique categorical values, and the y-axis shows the corresponding counts for each category. This visualization enables the user to visually identify the most prevalent and least common categories, offering valuable insights into the data distribution.

The 'univariate\_eda\_cat' function serves as an essential tool in understanding the distribution of categorical data within the dataset. By using this function, you can identify the unique values, count occurrences of each category, and visualize the distribution across categories. This analysis aids in detecting class imbalances, identifying dominant or rare categories, and assessing the representativeness of each category in the dataset. Moreover, this exploration is crucial for making informed decisions about data preprocessing and for selecting appropriate encoding strategies when building machine learning models based on categorical features. Overall, the function facilitates a deeper understanding of the dataset's categorical characteristics and enhances the quality of data-driven insights and modelling decisions.

Example:

A graph with numbers and a bar

Description automatically generatedA graph of a number of blue rectangular bars

Description automatically generated

1. Creating a function to check and deal with outliers

In this section, we aim to identify and handle outliers in the "age\_of\_car" and "age\_of\_policyholder" features of our car insurance dataset. Outliers are data points that deviate significantly from the rest of the data and can lead to biased model predictions if not appropriately handled. Our goal is to ensure the accuracy and reliability of our predictive model for insurance claim prediction by addressing outliers effectively.

1. The "outlier\_check" Function:

The "outlier\_check" function is a utility function designed to identify outliers using the Interquartile Range (IQR) method. This method is robust and effective in detecting outliers in numerical data. For each specified column, the function performs the following steps:

a. Calculate Quartiles: It computes the first quartile (Q1) and third quartile (Q3) for the column using the "quantile" function with "0.25" and "0.75" as arguments, respectively.

b. Calculate Interquartile Range (IQR): The function then computes the interquartile range (IQR) by subtracting Q1 from Q3.

c. Determine Outlier Boundaries: The lower bound for outliers is calculated as Q1 - 1.5 \* IQR, and the upper bound is calculated as Q3 + 1.5 \* IQR.

d. Count Outliers: The function counts the number of outliers in the column by filtering the data where the values are either less than the lower bound or greater than the upper bound.

The "outlier\_check" function returns the lower bound, upper bound, and the count of outliers for the specified column.

2. The "outlier\_analysis" Function:

The "outlier\_analysis" function is responsible for handling the outlier analysis for the selected columns, "age\_of\_car" and "age\_of\_policyholder." The function performs the following steps:

a. Iteration: The function iterates through the selected columns to analyze each one separately.

b. Outlier Check: For each column, the function calls the "outlier\_check" function to obtain the lower bound, upper bound, and the count of outliers.

c. Handling Outliers: If there are no outliers in a particular column, the function displays a message indicating the absence of outliers and proceeds to plot a box plot for visualization. Box plots help visualize the distribution of data and identify the presence of outliers. If outliers are present, the function proceeds to handle them.

d. Outlier Treatment: To handle outliers, the function replaces outlier values in the column with their respective lower or upper bound values. This process effectively normalizes the data distribution and mitigates the influence of extreme values, ensuring that our model's predictions are less biased.

e. Visualization: The function plots two box plots side by side for comparison - one before the outlier treatment and one after the treatment. This visualization allows us to assess the impact of the outlier handling process on the data distribution.

Purpose of handling outliers:

The primary objective of handling outliers in these features is to mitigate the adverse effects of extreme values on our machine learning model, especially in the context of predicting insurance claims. Outliers can significantly skew the statistical properties of the data and lead to biased model predictions. By replacing the outliers with the respective lower or upper bound values, we effectively normalize the data distribution while preserving its essential characteristics.

In conclusion, conducting an outlier analysis and handling outliers in the "age\_of\_car" and "age\_of\_policyholder" features are critical steps to ensure the robustness and accuracy of our predictive model. By making the data more suitable for training and minimizing the influence of extreme values, we improve the generalizability of the model and enhance its predictive capabilities for insurance claim predictions. The resulting model will be more reliable, providing valuable insights to insurance companies in assessing potential claims and managing risk efficiently.

A comparison of a graph

Description automatically generated

***Pre-Processing before modelling:***

1. Data Encoding

Data encoding is performed for the categorical features in our car insurance dataset. Data encoding is a crucial step in preparing the data for modeling as machine learning algorithms require numeric input. We use two encoding techniques - Ordinal Encoding and One-Hot Encoding - to transform the categorical features into a suitable numeric representation.

- Ordinal Encoding

We apply Ordinal Encoding to the "ncap\_rating" feature, which represents the safety rating of the car. This feature has ordered categories, i.e., "5", "4", "3", "2", and "0", indicating different safety ratings. Ordinal Encoding assigns a unique numeric value to each category based on the specified order. In this case, we use the "OrdinalEncoder" from Scikit-learn to map the categories to the integers 0 to 4, representing their safety ratings from lowest to highest.

- One Hot Encoding

We perform One-Hot Encoding on several categorical features with multiple unordered categories. One-Hot Encoding creates binary columns for each category in the original feature, where a "1" represents the presence of that category, and "0" indicates its absence. We apply One-Hot Encoding to the following features: "transmission\_type", "cylinder", "gear\_box" & etc.

For each feature, we convert it into multiple binary columns using the "OneHotEncoder" from Scikit-learn. The resulting encoded columns are appended to the "en\_df" DataFrame. After encoding all the specified features, we merge "en\_df" with the original dataset ("df") using the "index" column as a common key. This ensures that the encoded features are correctly aligned with the original data.

Objective of doing encoding:

The encoded DataFrame "df\_model\_en" contains the original numerical features and the newly encoded categorical features. This enriched dataset is now ready for modeling. By encoding the categorical features, we enable machine learning models to process them as numeric data, making it easier for the algorithms to identify patterns and relationships during training.

The "df\_model\_en" dataset can be used to build and evaluate predictive models for car insurance claim prediction. Various machine learning algorithms, such as logistic regression, decision trees, random forests, or XGBoost, can now be trained on this enriched data to predict insurance claim probabilities accurately. The data encoding process ensures that all relevant information from the categorical features is effectively represented and contributes to the overall performance of the models.

Before Encoding: After Encoding:

A screenshot of a computer

Description automatically generated A screenshot of a number

Description automatically generated

1. Splitting the data

In machine learning, it is essential to separate the dataset into two components: independent features (X) and the dependent feature or target variable (y). The independent features are the attributes or variables used to predict the target variable, while the dependent feature is the one we aim to predict based on the independent features.

X\_df: The variable "X\_df" contains the independent features, which are the attributes or characteristics of the data that will be used as input to the machine learning model. In this case, "X\_df" is created by dropping the column "is\_claim" from the preprocessed DataFrame "df\_model\_en." This results in a DataFrame that contains all the features except for the target variable "is\_claim."

y\_df: The variable "y\_df" represents the dependent feature, which is the target variable we want to predict. In this case, "y\_df" contains the values of the "is\_claim" column from the preprocessed DataFrame "df\_model\_en."

The reason for splitting the data into independent and dependent features is to ensure that we can train our machine learning model effectively. The model needs to learn from the input features (X\_df) and understand their relationships with the target variable (y\_df) during the training phase. By separating the features and the target variable, we are preparing the data for the modeling process, allowing us to use various machine learning algorithms to build a predictive model.

After splitting the data, we can proceed with further steps such as data scaling, model training, testing, and evaluation to develop a robust predictive model for the insurance claim predictions.

1. Handling imbalanced data

The target variable, "is\_claim," is balanced or imbalanced. A balanced dataset means that both classes (in this case, whether a claim is made or not) have roughly equal representation, while an imbalanced dataset means that one class significantly outweighs the other in terms of the number of samples. Handling imbalanced datasets is essential because many machine learning algorithms tend to perform poorly on such datasets, as they often focus more on the majority class, leading to biased predictions.

- minority\_class and majority\_class: These variables store the count of the two classes in the target variable "y\_data." The code uses the value\_counts() method to count the occurrences of each unique class and then sorts the values in ascending order.

- Checking Balance: The code compares the count of the minority class (class with fewer samples) multiplied by 2 with the count of the majority class (class with more samples). If the minority class is at least half the size of the majority class, the data is considered balanced.

- Handling Imbalance with SMOTE: If the data is imbalanced (i.e., the minority class is less than half the size of the majority class), the code prints a message indicating that the data is imbalanced. It then performs Synthetic Minority Over-sampling Technique (SMOTE) to balance the dataset. SMOTE is a technique that generates synthetic samples for the minority class by interpolating existing samples, thereby creating a more balanced dataset.

- Printing Results: The code prints the shape of the data before and after SMOTE and the value counts of each class to show the balancing effect of SMOTE.

Usefulness in Modeling:

Handling imbalanced data is crucial for building robust machine learning models. An imbalanced dataset can lead to biased predictions, where the model predominantly predicts the majority class and may not capture the patterns present in the minority class. By using SMOTE to balance the dataset, we ensure that both classes have sufficient representation, allowing the model to learn from both classes and make unbiased predictions.

Balancing the data with SMOTE improves the overall performance of the model, especially for classification problems like insurance claim prediction, where the distribution of claim and non-claim instances is typically imbalanced. This preprocessing step helps the model to learn from the minority class and provide more accurate predictions for both classes, leading to better generalization and higher predictive accuracy.

Before handling imbalanced data: After handling imbalanced data:

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

***Modelling:***

1. Model Overview

The purpose of this report is to evaluate the performance of a classification model for predicting the "is\_claim" target variable. The model aims to determine whether a policyholder will file a claim in the next 6 months based on various features. To accomplish this, we have developed a Python function called classification\_model. The function is designed to build, evaluate, and diagnose the performance of different classification models using key metrics.

2. Model Building and Evaluation

Function Definition: The function classification\_model is crucial in our analysis as it automates the model building and evaluation process. It takes two parameters as input: MODEL, representing the classification algorithm (e.g., Logistic Regression, Decision Tree, etc.), and train, which is the training dataset consisting of both the features and the target variable.

Model Fitting: Inside the function, the chosen classifier is initialized, and it is then trained on the training dataset using the train\_test\_split function. This allows us to divide the data into a training set and a testing set, enabling unbiased model evaluation.



Model Prediction: After the model is trained, it makes predictions on both the training and testing data to assess its performance on unseen data.

A close up of words

Description automatically generated

Metrics Calculation: The function calculates several key metrics to assess the model's effectiveness:

Accuracy Score: It indicates the overall accuracy of the model's predictions.

A close-up of a computer code

Description automatically generated

Precision Score: This metric measures the accuracy of positive predictions, focusing on correctly classified positive instances.

A close-up of a computer code

Description automatically generated

Recall Score: It quantifies the model's ability to identify positive instances, also known as the sensitivity or true positive rate.



F1 Score: The F1 score is the harmonic mean of precision and recall, offering a balanced evaluation measure for the model.



ROC AUC Score: It represents the area under the Receiver Operating Characteristic (ROC) curve, which illustrates the trade-off between sensitivity and specificity.



Model Check: The function further performs a model check to assess whether the model is overfitting or underfitting. It compares the training and testing accuracy and calculates the variance between them. The model is categorized based on this comparison as either having low bias, low variance; high bias, low variance; high bias, high variance; or low bias, high variance.

A white background with black text

Description automatically generated

Classification Report: The classification report is generated, containing precision, recall, F1-score, and support values for both the training and testing data. This detailed report provides insights into the model's performance for each class, helping to identify areas for improvement.

Confusion Matrix: The function plots the confusion matrix for both the training and testing data. The confusion matrix visually shows the model's performance in terms of true positives, true negatives, false positives, and false negatives, facilitating a deeper understanding of the model's strengths and weaknesses.

3. Purpose of the Code

The classification\_model function was created to streamline and standardize the process of model evaluation and diagnostics. By automating the model building and evaluation steps, it saves time and effort, making the analysis more efficient. The code allows us to evaluate different classification models quickly, compare their performances, and select the best-performing model for deployment.

4. Usefulness in Modeling

This function is highly valuable in the modeling process due to the following reasons:

Automation: It automates the model building and evaluation steps, making the process more efficient and reproducible.

Metrics Assessment: It calculates essential classification metrics, enabling a thorough assessment of the model's performance.

Diagnostic Information: The function provides insights into potential overfitting or underfitting issues, guiding model selection and tuning.

Reporting: The generated classification report and confusion matrix offer comprehensive diagnostic information for stakeholders to make informed decisions about model improvement.

5. Conclusion

The classification\_model function plays a vital role in evaluating and diagnosing the performance of classification models. By automating the model evaluation process, it simplifies the comparison and selection of the best model for deployment. The calculated metrics and diagnostic tools help stakeholders identify areas for model improvement and make data-driven decisions for model optimization. Overall, this function greatly enhances the efficiency and effectiveness of the model evaluation process in classification tasks

6. Model Building   
Importing Libraries: The code starts by importing the necessary libraries. For example, it imported the LogisticRegression class from the scikit-learn library. This class allows us to create a logistic regression model, which is suitable for binary classification tasks.

Model Building and Evaluation: The main part of the code calls the classification\_model function with LogisticRegression() as the first argument and X as the second argument. This function is a custom function designed to automate the process of model building and evaluation for classification tasks.

Function Arguments: The classification\_model function takes two main arguments:

- MODEL: This argument represents the machine learning classifier used for building the model. In this case, we pass LogisticRegression() as the value for this argument, indicating that we want to build a logistic regression model.

- train: This argument represents the input data (features) on which the model will be trained. In this code, X is provided as the value for this argument, which presumably contains the independent features used for prediction.

A close-up of a white background

Description automatically generated

Upon writing this code, the following output will be calculated:

**Output in next page**

A screenshot of a computer

Description automatically generated A screenshot of a graph

Description automatically generated

A blue squares with white numbers

Description automatically generated

7. Comparison with different models

It is to compare the performance metrics of different machine learning models. This can help you understand which model performs better on the task at hand and guide you in selecting the most suitable model for your specific problem. It's a common practice in machine learning to evaluate and compare models using various metrics to ensure they meet the desired criteria for performance.

After that create the model called model.pkl

Deployment through flask:

Overview

This report provides an overview of a Flask web application that predicts car insurance claim outcomes using a pre-trained machine learning model. The application is designed to take input from the user through a web form, process the input, and display the predicted result on the user interface.

Application Components

Libraries Used:

Numpy: A library for numerical computing in Python.

Flask: A web framework for Python to build web applications.

Joblib: A library for serializing Python objects, used to load the pre-trained machine learning model.

Web Application Setup:

The Flask app is created using Flask(\_\_name\_\_).

The pre-trained machine learning model is loaded into the model variable using joblib.load('model.pkl'). The 'model.pkl' file should have been previously trained and saved using joblib.dump().

Routes:

Root Route '/': This route is used to display the content from the 'index.html' template. The function home() is called, which renders the template using render\_template().

'/predict' Route: This route is used to receive form data from the user, which contains input features required for prediction.

HTTP Method: POST

Function: predict()

Functions:

a. home() Function:

This function renders the 'index.html' template using render\_template() when the root route is accessed.

The 'index.html' template should contain a form to take user input.

b. predict() Function:

This function is called when the '/predict' route is accessed via a POST request (form submission).

It retrieves the form data from the user and converts the input values to float data types.

The form data is then converted into a Numpy array.

The Numpy array is passed to the pre-trained machine learning model for prediction using model.predict().

The prediction result is formatted into text indicating whether "Car Insurance can be [prediction]".

The formatted prediction text is returned and displayed in the 'index.html' template.

Usage

Start the Flask application using app.run().

Access the web application through the browser or via a localhost URL.

The 'index.html' template is displayed, containing a form to input car insurance claim features.

Upon submitting the form, the application sends a POST request to the '/predict' route.

The machine learning model processes the input data and predicts the car insurance claim outcome.

The predicted outcome is displayed on the web page.

A screenshot of a car insurance claim

Description automatically generated

A screenshot of a computer screen

Description automatically generated

Conclusion:

In conclusion, we have successfully built and evaluated classification models to predict whether a policyholder is likely to file a claim in the next 6 months. The XGBoost model demonstrated promising results, providing a solid foundation for making data-driven decisions in insurance claim prediction. However, it is essential to continuously monitor and fine-tune the models as new data becomes available to ensure their ongoing accuracy and reliability.

Finally, to make the model accessible to users, we can integrate it into a web application using Flask, where users can input their information, and the model will provide a prediction on whether they are likely to file a claim. This would allow insurers and policyholders to make more informed decisions and enhance the overall insurance experience.