Comprehensive Report on Binary Classification Model for Kaggle Competition

## Introduction

This report provides a detailed account of the techniques, strategies, models, and code used to construct a binary classification model for a Kaggle competition. The report covers data preprocessing, exploration, model selection, hyperparameter tuning, and evaluation. Additionally, it discusses the challenges faced, solutions implemented, and recommendations for future improvements.  
This report compiles all the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. It references the entire chat history and provides a thorough analysis of the steps taken throughout the project.  
This report compiles all the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. It references the entire chat history and provides a thorough analysis of the steps taken throughout the project.  
This report compiles all the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. It references the entire chat history and provides a thorough analysis of the steps taken throughout the project.  
This report documents the entire process of constructing a binary classification model for a Kaggle competition. It includes detailed descriptions of data preprocessing, exploration, model selection, hyperparameter tuning, and final evaluation. Key code snippets and insights from the project are highlighted throughout.  
This report outlines the detailed techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. It includes comprehensive coverage of data preprocessing, model selection, evaluation, hyperparameter tuning, and experiment tracking, as well as the challenges faced and solutions implemented during the project.  
This report provides a comprehensive overview of the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. The entire process, from data preprocessing to final model evaluation, is documented in detail, with specific attention to key aspects such as data exploration, model selection, hyperparameter tuning, and experiment tracking.  
This report documents the techniques, strategies, models, and code used to construct a binary classification model for a Kaggle competition. The entire modeling process, from data preprocessing to final model evaluation, is covered in detail. All code snippets, methods, and strategies employed are thoroughly explained.

This report provides a comprehensive overview of the techniques, strategies, models, and code used to construct a binary classification model for a Kaggle competition. The entire modeling process, from data preprocessing to final model evaluation, is thoroughly detailed. This includes data exploration, feature engineering, model selection, hyperparameter tuning, and experiment tracking. Additionally, challenges faced during the project and how they were addressed are discussed, along with practical insights and recommendations for future improvements.

- \*\*Data Cleaning\*\*: Removing duplicates, handling missing values by imputation or deletion.  
- \*\*Handling Missing Values\*\*: Techniques such as filling missing values with mean/median for numerical features or mode for categorical features.  
- \*\*Feature Engineering\*\*: Creating new features, encoding categorical variables, and transforming features.  
- \*\*Scaling\*\*: Standardizing numerical features to ensure they have a mean of 0 and a standard deviation of 1.  
1. \*\*Data Cleaning\*\*:  
- Checked for missing values and handled them using imputation techniques or by dropping the rows/columns with excessive missing data.  
- Removed duplicates and outliers to ensure the dataset's integrity.  
#### Data Cleaning  
- \*\*Loading Data\*\*: Data was loaded from CSV files stored in Google Drive.  
- \*\*Handling Missing Values\*\*: Missing values were identified and handled appropriately, either by imputation or removal.  
- \*\*Data Sampling\*\*: Due to the large dataset size, a 40% sample of the training data was used for initial exploration and modeling to manage computational resources efficiently.

\*\*Data Cleaning and Handling Missing Values:\*\*

- No missing values were detected in the dataset, as confirmed by the initial data analysis.  
- Removed `Driving\_License` due to limited variability.  
- Transformed binary variables (`Gender`, `Vehicle\_Damage`) into numerical representations.  
- Handled outliers in `Annual\_Premium` using the IQR method.

- \*\*Interaction Features\*\*: Created new features by multiplying existing ones (e.g., Age\_Annual\_Premium, Age\_Vintage).  
- \*\*Polynomial Features\*\*: Generated polynomial features up to the second degree for numerical variables using the PolynomialFeatures class.  
- \*\*Binning\*\*: Applied binning to continuous features such as age and annual\_premium for better interpretability.  
- Created new features based on existing ones (e.g., polynomial features).  
- Encoded categorical features using one-hot encoding.  
- Standardized continuous features to ensure consistent scaling.  
- Created new features through polynomial combinations and interactions.  
- Applied one-hot encoding to categorical variables.  
- Standardized continuous features using StandardScaler.  
- Grouped rare categories in categorical variables.  
- Ordinal encoding for `Vehicle\_Age`.  
- One-Hot Encoding for other categorical variables.  
- Created new features: `Age\_Vehicle\_Age`, `Age\_Previously\_Insured`, `Vehicle\_Age\_Damage`, `Previously\_Insured\_Damage`, `Age\_squared`, `Vehicle\_Age\_squared`, `Annual\_Premium\_per\_Age`.

- Standardized numerical features to have a mean of zero and a standard deviation of one using StandardScaler.  
- Applied standard scaling to continuous features to normalize their ranges.  
- Used StandardScaler to standardize continuous features to zero mean and unit variance.  
- Standardized continuous variables using `StandardScaler`.

- \*\*Exploratory Data Analysis (EDA)\*\*: Understanding the distribution and relationships of features through histograms, box plots, and scatter plots.  
- \*\*Correlation Analysis\*\*: Using heatmaps to identify correlations between features.  
- \*\*Visualization Tools\*\*: Libraries such as Seaborn and Matplotlib for creating insightful visualizations.  
- Used Seaborn and Matplotlib for data visualization.  
- Visualized data distributions using histograms, box plots, and bar charts.  
- Created correlation heatmaps to understand relationships between features.  
1. \*\*Initial Data Exploration:\*\*  
- Used pandas for initial data inspection (`df.head()`, `df.describe()`, `df.info()`).  
- Checked for class imbalance and feature distributions.  
- \*\*Exploratory Data Analysis (EDA)\*\*:  
- Utilized libraries like `Seaborn` and `Matplotlib` to create visualizations.  
- Plotted distributions of numerical features, bar plots for categorical features, and heatmaps for correlation analysis.  
- Visualized the target variable distribution and identified class imbalances.  
- \*\*Initial Data Exploration\*\*: Summary statistics and data distributions were examined to understand the dataset.  
- \*\*Visualization Techniques\*\*:  
- \*\*Histograms\*\* and \*\*Box Plots\*\*: Used to visualize the distribution and identify outliers.  
- \*\*Correlation Heatmaps\*\*: Used to identify relationships between variables.  
- \*\*Cluster Analysis\*\*: Visualized using scatter plots to understand data segmentation.  
- \*\*Exploratory Data Analysis (EDA)\*\*: Initial data exploration involved using summary statistics and visualizations to understand the data distribution, identify patterns, and detect anomalies.  
- \*\*Visualization Techniques\*\*: Histograms, box plots, bar charts, and correlation heatmaps were used to visualize data distributions and relationships between features.

- \*\*Correlation Analysis\*\*: Calculated and visualized the correlation matrix to identify relationships between numerical features.

- \*\*Skewness and Distribution Analysis\*\*: Analyzed the skewness of numerical features and plotted their distributions.  
- \*\*Principal Component Analysis (PCA)\*\*: Applied PCA to reduce the dimensionality of the data and visualize the first two principal components.  
- \*\*t-SNE and UMAP\*\*: Used t-SNE and UMAP for visualizing high-dimensional data in 2D space. Due to computational constraints, downsampling and PCA were applied before these methods.

### Techniques for Balancing the Dataset

- \*\*SMOTE (Synthetic Minority Over-sampling Technique)\*\*: Used to balance the dataset by creating synthetic samples of the minority class.

- \*\*SMOTE (Synthetic Minority Over-sampling Technique)\*\*: Utilized to balance the target variable classes by generating synthetic samples for the minority class.

### List and Description of Models Attempted

1. \*\*Logistic Regression\*\*:  
- Baseline model for binary classification.  
- Simple and interpretable.

- \*\*Logistic Regression\*\*: A baseline model for binary classification.

- \*\*Decision Trees\*\*: Simple model to capture non-linear relationships.  
- \*\*Random Forests\*\*: Ensemble method to improve decision tree performance.  
- \*\*Gradient Boosting (XGBoost and LightGBM)\*\*: Advanced ensemble methods to handle large datasets and improve accuracy.  
- \*\*Neural Networks\*\*: Used for capturing complex patterns in the data.  
- \*\*Autoencoders\*\*: Employed for feature extraction and dimensionality reduction.

- \*\*Initial Model Selection\*\*: Models were selected based on their performance on the validation set using metrics such as accuracy, precision, recall, F1-score, and AUC-ROC.  
- \*\*Evaluation Metrics\*\*: Models were evaluated using Stratified K-Fold Cross-Validation to ensure robustness and reliability of performance metrics.  
- \*\*Final Model Selection\*\*: Based on the evaluation metrics, the best-performing model was selected for hyperparameter tuning and final evaluation.  
- \*\*Cross-Validation\*\*: Using Stratified K-Fold Cross-Validation to ensure robust model evaluation.  
- \*\*Evaluation Metrics\*\*: Metrics such as accuracy, precision, recall, and F1-score to assess model performance.  
- Evaluated models using cross-validation with ROC-AUC as the primary metric.  
- LightGBM was chosen as the best-performing model with a ROC-AUC score of 0.8708.  
- Detailed steps included:  
- Splitting the data into training and testing sets.  
- Training models on the training set.  
- Evaluating models on the test set using ROC-AUC.  
- Fine-tuning the chosen model (LightGBM) using hyperparameter optimization.  
1. \*\*Model Selection:\*\*  
- Evaluated models using Stratified K-Fold Cross-Validation.  
- Focused on metrics like accuracy, precision, recall, and F1 score.  
- \*\*Steps and Reasoning\*\*:  
- Started with simple models like Logistic Regression and Decision Trees to establish a baseline.  
- Progressively moved to more complex models like Random Forests and Gradient Boosting to improve performance.  
- Evaluated models using cross-validation and performance metrics like ROC AUC, accuracy, precision, recall, and F1-score.  
- \*\*Cross-Validation\*\*: Stratified K-Fold Cross-Validation was used to evaluate model performance robustly.  
- \*\*Metrics\*\*: Accuracy, precision, recall, and F1-score were the primary metrics for evaluation.  
- \*\*Model Selection\*\*: Models were selected based on their performance metrics such as accuracy, precision, recall, and F1 score.  
- \*\*Cross-Validation\*\*: Stratified K-Fold cross-validation was used to ensure robust evaluation of model performance.  
- \*\*Ensemble Methods\*\*: Voting classifiers and stacking were used to combine multiple models for improved performance.

- \*\*Random Forests\*\*: Selected for their robustness and ability to handle large datasets.

- \*\*Gradient Boosting (XGBoost and LightGBM)\*\*: Chosen for their superior performance in handling imbalanced datasets and high-dimensional data.  
- \*\*Autoencoders\*\*: Used for feature extraction to improve downstream model performance.

- \*\*GridSearchCV\*\*: Used for exhaustive search over specified parameter values.  
- \*\*Optuna\*\*: Used for more efficient hyperparameter optimization, leveraging techniques such as Bayesian optimization.  
- Code for hyperparameter tuning using GridSearchCV.  
- Code for hyperparameter tuning using GridSearchCV.  
- Code for hyperparameter tuning using GridSearchCV.  
- Code for hyperparameter tuning using GridSearchCV.  
- \*\*GridSearchCV\*\*: Systematic search for optimal hyperparameters.  
- \*\*Optuna\*\*: Advanced optimization framework for hyperparameter tuning.  
python  
import optuna  
from optuna.samplers import TPESampler  
python  
import optuna  
- \*\*GridSearchCV\*\*: Exhaustive search over specified parameter values.  
- \*\*Optuna\*\*: Bayesian optimization framework to efficiently search the hyperparameter space.  
python  
import optuna  
from sklearn.model\_selection import cross\_val\_score  
```python  
import optuna  
from optuna.integration import PyTorchLightningPruningCallback  
from pytorch\_lightning import LightningModule, Trainer  
from pytorch\_lightning.callbacks import ModelCheckpoint, EarlyStopping  
- \*\*GridSearchCV\*\*: Used initially for hyperparameter tuning but found to be computationally intensive.  
- \*\*Optuna\*\*: Adopted for Bayesian hyperparameter optimization, providing efficient and effective tuning.  
- \*\*Objective Function\*\*: Defined to optimize the ROC AUC score.  
- \*\*Parameter Space\*\*: Included parameters such as `learning\_rate`, `num\_leaves`, `max\_depth`, and `feature\_fraction`.

- \*\*Optuna\*\*: Employed for hyperparameter tuning using Bayesian optimization.

- \*\*GridSearchCV\*\*: Used for initial hyperparameter tuning of simpler models.

- Code for loading data and initial preprocessing steps.  
- Code for loading data and initial preprocessing steps.  
- Code for loading data and initial preprocessing steps.  
- Code for loading data and initial preprocessing steps.  
python  
# Load the datasets  
train\_df = pd.read\_csv("/content/drive/My Drive/Kaggle Competition/train.csv", index\_col='id')  
test\_df = pd.read\_csv("/content/drive/My Drive/Kaggle Competition/test.csv", index\_col='id')  
python  
import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
python  
import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler, OneHotEncoder  
from imblearn.over\_sampling import SMOTE

# Data Loading  
train\_df = pd.read\_csv("klib\_full\_trainset.csv")  
test\_df = pd.read\_csv("klib\_full\_testset.csv")  
# Interaction Features  
class InteractionFeatures(BaseEstimator, TransformerMixin):  
def fit(self, X, y=None):  
return self  
# Random Forest Model  
from sklearn.ensemble import RandomForestClassifier  
# Optuna for Hyperparameter Tuning  
import optuna  
# Custom Logger Class  
class Logger:  
def \_\_init\_\_(self):  
self.logger = self.setup\_logging()  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
  
from sklearn.preprocessing import StandardScaler  
from imblearn.over\_sampling import SMOTE  
from sklearn.linear\_model import LogisticRegression  
model = LogisticRegression()  
  
from sklearn.ensemble import RandomForestClassifier  
model = RandomForestClassifier()  
  
import torch  
import torch.nn as nn  
from sklearn.model\_selection import GridSearchCV  
from google.colab import drive  
import pandas as pd  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score  
from sklearn.model\_selection import GridSearchCV  
import mlflow  
import mlflow.sklearn  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
  
from sklearn.preprocessing import StandardScaler  
from imblearn.over\_sampling import SMOTE  
from sklearn.linear\_model import LogisticRegression  
model = LogisticRegression()  
  
from sklearn.ensemble import RandomForestClassifier  
model = RandomForestClassifier()  
  
import torch  
import torch.nn as nn  
from sklearn.model\_selection import GridSearchCV  
from google.colab import drive  
import pandas as pd  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score  
from sklearn.model\_selection import GridSearchCV  
import mlflow  
import mlflow.sklearn  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
  
from sklearn.preprocessing import StandardScaler  
from imblearn.over\_sampling import SMOTE  
from sklearn.linear\_model import LogisticRegression  
model = LogisticRegression()  
  
from sklearn.ensemble import RandomForestClassifier  
model = RandomForestClassifier()  
  
import torch  
import torch.nn as nn  
from sklearn.model\_selection import GridSearchCV  
from google.colab import drive  
import pandas as pd  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score  
from sklearn.model\_selection import GridSearchCV  
import mlflow  
import mlflow.sklearn  
from sklearn.linear\_model import LogisticRegression  
model = LogisticRegression()  
  
from sklearn.ensemble import RandomForestClassifier  
model = RandomForestClassifier()  
  
import torch  
import torch.nn as nn  
from sklearn.model\_selection import GridSearchCV  
from google.colab import drive  
import pandas as pd  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score  
from sklearn.model\_selection import GridSearchCV  
import mlflow  
import mlflow.sklearn  
import pandas as pd  
import numpy as np  
from sklearn.model\_selection import train\_test\_split  
# Polynomial features  
from sklearn.preprocessing import PolynomialFeatures  
poly = PolynomialFeatures(degree=2)  
poly\_features = poly.fit\_transform(df[['numerical\_feature1', 'numerical\_feature2']])  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.model\_selection import cross\_val\_score  
import optuna  
class CustomModel:  
def \_\_init\_\_(self, model):  
self.model = model  
import pandas as pd  
import klibs # Import the klibs module  
# Example of feature engineering  
data['new\_feature'] = data['feature1'] \* data['feature2']  
data = pd.get\_dummies(data, columns=['categorical\_feature'])  
  
class InteractionFeatures(BaseEstimator, TransformerMixin):  
def \_\_init\_\_(self):  
pass  
def fit(self, X, y=None):  
return self  
def transform(self, X):  
X = X.copy()  
X['Age\_Annual\_Premium'] = X['Age'] \* X['Annual\_Premium']  
X['Age\_Vintage'] = X['Age'] \* X['Vintage']  
X['Annual\_Premium\_Vintage'] = X['Annual\_Premium'] \* X['Vintage']  
X['Age\_Region\_Code'] = X['Age'] \* X['Region\_Code']  
X['Vintage\_Region\_Code'] = X['Vintage'] \* X['Region\_Code']  
X['Annual\_Premium\_Region\_Code'] = X['Annual\_Premium'] \* X['Region\_Code']  
return X  
  
def plot\_distribution(data, features, target, filename, figsize=(18, 18)):  
plt.figure(figsize=figsize)  
for i, feature in enumerate(features, 1):  
plt.subplot(3, 2, i)  
sns.histplot(data[feature], kde=True)  
plt.title(f'Distribution of {feature}')  
plt.tight\_layout()  
plt.savefig(filename)  
plt.close()  
def run\_bayesian\_search(X\_train, y\_train):  
param\_dist = {  
'learning\_rate': (0.03, 0.1),  
'num\_leaves': (60, 120),  
'max\_depth': (10, 15),  
'min\_data\_in\_leaf': (10, 50),  
'bagging\_fraction': (0.6, 0.8),  
'feature\_fraction': (0.6, 0.8),  
'lambda\_l1': (0.0, 1.0),  
'lambda\_l2': (0.0, 1.0),  
'bagging\_freq': (1, 7)  
}  
train\_df = pd.read\_csv(r"C:\Users\paulo\OneDrive\Documents\Binary-Classification-of-Insurance-Cross-Selling\preprocessed\_train.csv")  
test\_df = pd.read\_csv(r"C:\Users\paulo\OneDrive\Documents\Binary-Classification-of-Insurance-Cross-Selling\preprocessed\_test.csv")  
train\_df = reduce\_memory\_usage(train\_df)  
test\_df = reduce\_memory\_usage(test\_df)  
  
def train\_final\_model(params, X\_train, y\_train, X\_val, y\_val):  
# Create LightGBM datasets  
train\_data = lgb.Dataset(X\_train, label=y\_train, free\_raw\_data=False)  
val\_data = lgb.Dataset(X\_val, label=y\_val, reference=train\_data, free\_raw\_data=False)  
def evaluate\_model(model, X\_train, y\_train, X\_val, y\_val):  
# Evaluate the best model on the training set  
train\_preds = model.predict(X\_train, num\_iteration=model.best\_iteration)  
train\_auc = roc\_auc\_score(y\_train, train\_preds)  
def save\_model\_and\_metrics(model, X\_train, y\_train, X\_val, y\_val):  
# Predict on train and validation set  
y\_train\_pred = model.predict(X\_train, num\_iteration=model.best\_iteration)  
y\_val\_pred = model.predict(X\_val, num\_iteration=model.best\_iteration)  
# Load the datasets  
train\_df = pd.read\_csv("/content/drive/My Drive/Kaggle Competition/train.csv", index\_col='id')  
test\_df = pd.read\_csv("/content/drive/My Drive/Kaggle Competition/test.csv", index\_col='id')  
# Define continuous numeric variables  
continuous\_numeric = ['Age', 'Vintage', 'Annual\_Premium']  
from sklearn.model\_selection import train\_test\_split  
import lightgbm as lgb  
from sklearn.metrics import roc\_auc\_score, roc\_curve, auc  
import matplotlib.pyplot as plt  
from skopt import BayesSearchCV  
from skopt.space import Real, Integer  
import pandas as pd  
import numpy as np  
from sklearn.preprocessing import StandardScaler, PolynomialFeatures  
from category\_encoders import TargetEncoder  
from imblearn.over\_sampling import SMOTE  
import lightgbm as lgb  
from sklearn.metrics import roc\_auc\_score  
import logging

X = train\_df.drop(columns=['response'])  
y = train\_df['response']  
scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(X)  
```  
X = train\_df.drop(columns=['response'])  
y = train\_df['response']

- Code for transforming binary variables and one-hot encoding categorical variables.  
- Code for transforming binary variables and one-hot encoding categorical variables.  
- Code for transforming binary variables and one-hot encoding categorical variables.  
- Code for transforming binary variables and one-hot encoding categorical variables.  
python  
# Handle outliers for Annual\_Premium  
Q1 = train\_df['Annual\_Premium'].quantile(0.25)  
Q3 = train\_df['Annual\_Premium'].quantile(0.75)  
IQR = Q3 - Q1  
lower\_bound = Q1 - 1.5 \* IQR  
upper\_bound = Q3 + 1.5 \* IQR  
python  
from sklearn.preprocessing import OneHotEncoder  
The code above illustrates how features are engineered and transformed, including interaction terms and polynomial features, followed by standardization.

X['Age\_Annual\_Premium'] = X['age'] \* X['annual\_premium']  
X['Age\_Vintage'] = X['age'] \* X['vintage']  
return X  
poly\_features = self.poly.transform(X)  
return np.hstack([X, poly\_features])  
  
X['Age\_Annual\_Premium'] = X['Age'] \* X['Annual\_Premium']  
X['Age\_Vintage'] = X['Age'] \* X['Vintage']  
X['Annual\_Premium\_Vintage'] = X['Annual\_Premium'] \* X['Vintage']  
return X  
poly\_features = self.poly.transform(X[['Age', 'Annual\_Premium', 'Vintage']])  
poly\_feature\_names = self.poly.get\_feature\_names\_out(['Age', 'Annual\_Premium', 'Vintage'])  
poly\_df = pd.DataFrame(poly\_features, columns=[f'poly\_{name.replace(" ", "\_")}' for name in poly\_feature\_names], index=X.index)  
X = pd.concat([X, poly\_df], axis=1)  
return X  
```

class PolynomialFeatureGeneration(BaseEstimator, TransformerMixin):  
def fit(self, X, y=None):  
self.poly = PolynomialFeatures(degree=2, interaction\_only=True, include\_bias=False)  
self.poly.fit(X)  
return self  
class PolynomialFeatureGeneration(BaseEstimator, TransformerMixin):  
def \_\_init\_\_(self):  
self.poly = PolynomialFeatures(degree=2, interaction\_only=True, include\_bias=False)  
self.feature\_names = None  
poly = PolynomialFeatures(degree=2, interaction\_only=True, include\_bias=False)  
poly\_features = poly.fit\_transform(train\_df[['Age', 'Annual\_Premium', 'Vintage']])  
poly\_feature\_names = poly.get\_feature\_names\_out(['Age', 'Annual\_Premium', 'Vintage'])

- Code for training and evaluating models.  
- Code for training and evaluating models.  
- Code for training and evaluating models.  
- Code for training and evaluating models.  
python  
# Separate features and target variable  
X = train\_df\_encoded.drop('Response', axis=1)  
y = train\_df\_encoded['Response']  
python  
import lightgbm as lgb  
from sklearn.metrics import roc\_auc\_score  
python  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

model.fit(X\_train, y\_train)  
y\_pred = model.predict\_proba(X\_val)[:, 1]  
auc = roc\_auc\_score(y\_val, y\_pred)  
```

param = {  
'n\_estimators': trial.suggest\_int('n\_estimators', 100, 1000),  
'max\_depth': trial.suggest\_int('max\_depth', 3, 30),  
'learning\_rate': trial.suggest\_loguniform('learning\_rate', 1e-4, 1e-1)  
}  
model = xgb.XGBClassifier(\*\*param)  
model.fit(X\_train, y\_train)  
y\_pred = model.predict\_proba(X\_val)[:, 1]  
auc = roc\_auc\_score(y\_val, y\_pred)  
return auc  
param = {  
'tree\_method': 'gpu\_hist', 'lambda': trial.suggest\_float('lambda', 1e-3, 10.0, log=True),  
'alpha': trial.suggest\_float('alpha', 1e-3, 10.0, log=True), 'colsample\_bytree': trial.suggest\_categorical('colsample\_bytree', [0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]),  
'subsample': trial.suggest\_categorical('subsample', [0.4, 0.5, 0.6, 0.7, 0.8, 1.0]), 'learning\_rate': trial.suggest\_categorical('learning\_rate', [0.008, 0.01, 0.012, 0.014, 0.016, 0.018, 0.02]),  
'n\_estimators': trial.suggest\_int('n\_estimators', 5, 1000), 'max\_depth': trial.suggest\_categorical('max\_depth', [5, 7, 9, 11, 13, 15, 17]),  
'random\_state': 2020, 'min\_child\_weight': trial.suggest\_int('min\_child\_weight', 1, 300),  
}  
y\_train\_series = pd.Series(y\_train)  
ratio = float(y\_train\_series.value\_counts()[0]) / y\_train\_series.value\_counts()[1]  
model = xgb.XGBClassifier(\*\*param, scale\_pos\_weight=ratio)  
param = {  
'tree\_method': 'gpu\_hist',  
'objective': 'binary:logistic',  
'eval\_metric': 'auc',  
'max\_depth': trial.suggest\_int('max\_depth', 3, 9),  
'learning\_rate': trial.suggest\_loguniform('learning\_rate', 0.01, 0.1),  
'n\_estimators': trial.suggest\_int('n\_estimators', 100, 1000),  
'subsample': trial.suggest\_float('subsample', 0.5, 1.0),  
'colsample\_bytree': trial.suggest\_float('colsample\_bytree', 0.5, 1.0),  
}  
params = {  
'objective': 'binary',  
'metric': 'auc',  
'learning\_rate': trial.suggest\_float('learning\_rate', 0.05, 0.1),  
'num\_leaves': trial.suggest\_int('num\_leaves', 60, 120),  
'max\_depth': trial.suggest\_int('max\_depth', 10, 15),  
'min\_data\_in\_leaf': trial.suggest\_int('min\_data\_in\_leaf', 10, 50),  
'bagging\_fraction': trial.suggest\_float('bagging\_fraction', 0.6, 0.8),  
'feature\_fraction': trial.suggest\_float('feature\_fraction', 0.6, 0.8),  
'lambda\_l1': trial.suggest\_float('lambda\_l1', 0.0, 1.0),  
'lambda\_l2': trial.suggest\_float('lambda\_l2', 0.0, 1.0),  
'bagging\_freq': trial.suggest\_int('bagging\_freq', 1, 7)  
}  
param = {  
'objective': 'binary',  
'metric': 'auc',  
'boosting\_type': 'gbdt',  
'colsample\_bytree': trial.suggest\_float('colsample\_bytree', 0.4, 1.0),  
'lambda\_l1': trial.suggest\_float('lambda\_l1', 0.0, 10.0),  
'lambda\_l2': trial.suggest\_float('lambda\_l2', 0.0, 10.0),  
'learning\_rate': trial.suggest\_loguniform('learning\_rate', 0.01, 0.3),  
'max\_depth': trial.suggest\_int('max\_depth', 3, 10),  
'min\_child\_samples': trial.suggest\_int('min\_child\_samples', 1, 100),  
'n\_estimators': trial.suggest\_int('n\_estimators', 100, 1000),  
'num\_leaves': trial.suggest\_int('num\_leaves', 2, 256),  
'scale\_pos\_weight': trial.suggest\_float('scale\_pos\_weight', 1.0, 2.0),  
'subsample': trial.suggest\_float('subsample', 0.4, 1.0),  
}  
model = lgb.LGBMClassifier(\*\*param)  
model.fit(X\_train, y\_train, eval\_set=[(X\_test, y\_test)], eval\_metric='auc', early\_stopping\_rounds=100, verbose=False)  
preds = model.predict\_proba(X\_test)[:, 1]  
roc\_auc = roc\_auc\_score(y\_test, preds)  
return roc\_auc  
n\_estimators = trial.suggest\_int('n\_estimators', 100, 1000)  
max\_depth = trial.suggest\_int('max\_depth', 3, 30)  
n\_estimators = trial.suggest\_int('n\_estimators', 50, 200)  
max\_depth = trial.suggest\_int('max\_depth', 2, 32)  
hparams = {  
"num\_leaves": trial.suggest\_int("num\_leaves", 20, 150),  
"max\_depth": trial.suggest\_int("max\_depth", 3, 8),  
"learning\_rate": trial.suggest\_loguniform("learning\_rate", 1e-4, 1e-1),  
"n\_estimators": trial.suggest\_int("n\_estimators", 50, 1000),  
"min\_child\_samples": trial.suggest\_int("min\_child\_samples", 5, 100),  
"subsample": trial.suggest\_uniform("subsample", 0.6, 1.0)  
}  
param = {  
'n\_estimators': trial.suggest\_int('n\_estimators', 50, 200),  
'max\_depth': trial.suggest\_int('max\_depth', 3, 20),  
}  
model = RandomForestClassifier(\*\*param)  
model.fit(X\_train, y\_train)  
predictions = model.predict(X\_val)  
return accuracy\_score(y\_val, predictions)  
param = {  
'objective': 'binary',  
'metric': 'auc',  
'boosting\_type': 'gbdt',  
'learning\_rate': trial.suggest\_loguniform('learning\_rate', 0.01, 0.1),  
'num\_leaves': trial.suggest\_int('num\_leaves', 20, 150),  
'max\_depth': trial.suggest\_int('max\_depth', 3, 10),  
}  
lgb\_train = lgb.Dataset(X\_train, y\_train)  
lgb\_eval = lgb.Dataset(X\_val, y\_val, reference=lgb\_train)  
gbm = lgb.train(param, lgb\_train, valid\_sets=[lgb\_eval], early\_stopping\_rounds=10)  
preds = gbm.predict(X\_val, num\_iteration=gbm.best\_iteration)  
auc = roc\_auc\_score(y\_val, preds)  
return auc

study.optimize(objective, n\_trials=100)  
best\_params = study.best\_params  
  
study.optimize(objective, n\_trials=50)  
study.optimize(objective, n\_trials=100)

- \*\*Logger\*\*: Custom logging class to handle logging throughout the project.  
- \*\*DataHandler\*\*: Class for handling data loading and preprocessing steps.  
- \*\*PreprocessingPipeline\*\*: Class to encapsulate the preprocessing pipeline using custom transformers for feature engineering.  
  
```python  
def preprocess\_data(data):  
"""Function to preprocess the data."""  
data.fillna(data.mean(), inplace=True)  
data = pd.get\_dummies(data, drop\_first=True)  
scaler = StandardScaler()  
return scaler.fit\_transform(data)

timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')  
log\_file\_name = f'exploration\_{timestamp}.log'  
if os.path.exists('exploration.log'):  
os.remove('exploration.log')  
logger = logging.getLogger(\_\_name\_\_)  
logger.setLevel(logging.INFO)  
console\_handler = logging.StreamHandler()  
file\_handler = logging.FileHandler(log\_file\_name)  
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')  
console\_handler.setFormatter(formatter)  
file\_handler.setFormatter(formatter)  
logger.addHandler(console\_handler)  
logger.addHandler(file\_handler)  
return logger  
```

- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical operations.  
- \*\*scikit-learn\*\*: Machine learning models and utilities.  
- \*\*XGBoost\*\*: Gradient boosting.  
- \*\*LightGBM\*\*: Gradient boosting with light-weighted framework.  
- \*\*Optuna\*\*: Hyperparameter optimization.  
- \*\*PyTorch\*\*: Deep learning framework.  
- \*\*Seaborn\*\*: Statistical data visualization.  
- \*\*Matplotlib\*\*: Plotting library.

### Comprehensive List of Libraries Employed

- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical computing.  
- \*\*matplotlib\*\*: Plotting and visualization.  
- \*\*seaborn\*\*: Statistical data visualization.  
- \*\*scikit-learn\*\*: Machine learning library for model training, evaluation, and preprocessing.  
- \*\*XGBoost\*\*: Gradient boosting library for high-performance modeling.  
- \*\*LightGBM\*\*: Gradient boosting library optimized for speed and efficiency.  
- \*\*Optuna\*\*: Hyperparameter optimization library.  
- \*\*PyTorch\*\*: Deep learning framework for building and training neural networks.  
- \*\*umap\*\*: Dimensionality reduction library for efficient visualization.  
- `pandas`: Data manipulation and analysis.  
- `numpy`: Numerical computations.  
- `scikit-learn`: Machine learning utilities for preprocessing, model selection, and evaluation.  
- `LightGBM`: Gradient boosting framework for building models.  
- `Optuna`: Hyperparameter optimization.  
- `Seaborn` and `Matplotlib`: Data visualization.  
- `TensorFlow` and `Keras`: Deep learning frameworks for neural networks.  
- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical operations.  
- \*\*scikit-learn\*\*: Machine learning models and preprocessing.  
- \*\*imblearn\*\*: SMOTE for dataset balancing.  
- \*\*matplotlib\*\*: Data visualization.  
- \*\*seaborn\*\*: Enhanced data visualization.  
- \*\*xgboost\*\*: Gradient boosting model.  
- \*\*lightgbm\*\*: Gradient  
- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical operations.  
- \*\*scikit-learn\*\*: Machine learning algorithms and utilities.  
- \*\*XGBoost\*\*: Gradient boosting framework.  
- \*\*LightGBM\*\*: Another gradient boosting framework optimized for speed and performance.  
- \*\*Optuna\*\*: Hyperparameter optimization.  
- \*\*PyTorch\*\*: Deep learning framework.  
- \*\*Seaborn\*\*: Statistical data visualization.  
- \*\*Matplotlib\*\*: Plotting library.  
- \*\*imblearn\*\*: Techniques for imbalanced datasets (e.g., SMOTE).  
- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical computing.  
- \*\*scikit-learn\*\*: Machine learning algorithms and tools.  
- \*\*XGBoost\*\*: Gradient boosting framework.  
- \*\*LightGBM\*\*: Gradient boosting framework.  
- \*\*Optuna\*\*: Hyperparameter optimization.  
- \*\*PyTorch\*\*: Deep learning framework.  
- \*\*Seaborn\*\*: Statistical data visualization.  
- \*\*Matplotlib\*\*: Plotting and visualization.

- \*\*pandas\*\*: For data manipulation and analysis.

- \*\*numpy\*\*: For numerical computations.  
- \*\*matplotlib\*\* and \*\*seaborn\*\*: For data visualization.  
- \*\*scikit-learn\*\*: For machine learning algorithms and preprocessing.  
- \*\*XGBoost\*\* and \*\*LightGBM\*\*: For gradient boosting models.  
- \*\*Optuna\*\*: For hyperparameter tuning.  
- \*\*PyTorch\*\*: For building and training neural networks.  
- \*\*UMAP\*\*: For dimensionality reduction and visualization.

- \*\*pandas\*\*: Loading, cleaning, and preprocessing data.  
- \*\*numpy\*\*: Numerical transformations and calculations.  
- \*\*scikit-learn\*\*: Model training, evaluation, and preprocessing techniques.  
- \*\*XGBoost/LightGBM\*\*: Gradient boosting models.  
- \*\*Optuna\*\*: Efficient hyperparameter tuning.  
- \*\*PyTorch\*\*: Constructing and training neural networks.  
- \*\*Seaborn/Matplotlib\*\*: Data visualization and plotting.  
- \*\*imblearn\*\*: Balancing datasets using techniques like SMOTE.

- \*\*pandas\*\*: Used for loading, cleaning, and manipulating the datasets.

- \*\*numpy\*\*: Used for numerical operations and array manipulations.  
- \*\*matplotlib\*\* and \*\*seaborn\*\*: Used for creating visualizations of data distributions, correlations, and model performance.  
- \*\*scikit-learn\*\*: Used for data preprocessing, feature engineering, model training, and evaluation.  
- \*\*XGBoost\*\* and \*\*LightGBM\*\*: Used for training gradient boosting models.  
- \*\*Optuna\*\*: Used for optimizing hyperparameters of machine learning models.  
- \*\*PyTorch\*\*: Used for building and training neural networks, including autoencoders.  
- \*\*UMAP\*\*: Used for visualizing high-dimensional data in lower dimensions.

- \*\*SMOTE with Random Forests\*\*: Improved model performance on imbalanced dataset.  
- \*\*Polynomial Features with Logistic Regression\*\*: Enhanced feature interactions.  
- \*\*Combinations of Models and Preprocessing Techniques\*\*:  
- Tested models with different feature scaling techniques (StandardScaler, MinMaxScaler).  
- Evaluated the impact of SMOTE on model performance.

- \*\*Random Forest with StandardScaler and OneHotEncoder\*\*: Improved performance due to robust feature engineering and preprocessing.  
- \*\*XGBoost with SMOTE\*\*: Enhanced model performance on imbalanced data.  
- \*\*Neural Networks with Feature Scaling\*\*: Improved convergence and accuracy.

- \*\*Random Forest with SMOTE\*\*: Balanced the dataset using SMOTE and trained a Random Forest model.

- \*\*XGBoost with Polynomial Features\*\*: Generated polynomial features and trained an XGBoost model.  
- \*\*Autoencoder for Feature Extraction with Logistic Regression\*\*: Used an autoencoder to extract features and trained a logistic regression model on the extracted features.

### Different Configurations and Their Impacts

- \*\*Hyperparameter Tuning\*\*: GridSearchCV and Optuna significantly improved model performance by finding optimal parameters.  
- \*\*Feature Engineering\*\*: Interaction terms and polynomial features enhanced model accuracy and interpretability.

- \*\*Random Forest Configurations\*\*: Adjusting the number of trees and maximum depth had significant impacts on model performance and overfitting.

- \*\*XGBoost Hyperparameters\*\*: Tuning learning rate, max depth, and number of estimators improved model performance significantly.  
- \*\*Autoencoder Architectures\*\*: Different architectures (number of layers and neurons) impacted the quality of the latent space and downstream model performance.

- \*\*MLflow\*\*: Used for tracking experiments, versioning, and logging results.  
- \*\*Logging Strategies\*\*: Each model run and hyperparameter tuning session was logged with unique identifiers.  
- \*\*Tracking Experiments\*\*:  
- Used logging to record the progress and results of each model training and evaluation.  
- Stored results of hyperparameter tuning in CSV files.

- \*\*MLflow\*\*: Used to track experiments, including model parameters, metrics, and artifacts.  
- \*\*Versioning\*\*: Implemented version control for different model iterations and configurations.  
- \*\*Logging Strategies\*\*: Used logging to record important information and errors during model training and evaluation.  
- \*\*MLflow\*\*: Used for tracking experiments, logging parameters, metrics, and models.  
- \*\*Versioning\*\*: Maintained versions of models to compare performance and reproduce results.  
- \*\*MLflow\*\*: Used for tracking experiments, recording parameters, metrics, and models. Each experiment was logged with a unique identifier, and versioning allowed for easy comparison of different model iterations.

- \*\*MLflow\*\*: Used to track experiments, log metrics, and save models.

- \*\*Versioning\*\*: Each experiment run was versioned, allowing for easy comparison and reproducibility.  
- \*\*Logging Strategies\*\*: Logs were recorded for each experiment run, detailing the parameters, metrics, and outcomes.

- \*\*Class Imbalance\*\*: Addressed using SMOTE.  
- \*\*Overfitting in Decision Trees\*\*: Mitigated by using Random Forests and Gradient Boosting.  
- \*\*Hyperparameter Tuning Efficiency\*\*: Improved using Optuna for automated and efficient tuning.

- \*\*Data Imbalance\*\*: The dataset was highly imbalanced, which posed a challenge for model training.  
- \*\*Hyperparameter Tuning\*\*: Finding the optimal hyperparameters for complex models was computationally intensive.

- \*\*Handling Large Datasets\*\*: Processing and modeling with 11 million records required efficient data handling and model training strategies.

- \*\*Imbalanced Dataset\*\*: The target variable was highly imbalanced, necessitating techniques like SMOTE to balance the data.  
- \*\*Hyperparameter Tuning\*\*: Finding optimal hyperparameters for complex models was computationally intensive.

- Applied SMOTE for balancing the dataset.  
- Used Optuna for efficient hyperparameter optimization.  
- Implemented cross-validation and early stopping to mitigate overfitting.

- \*\*Efficient Data Processing\*\*: Used efficient data manipulation techniques and downsampling for initial exploration.

- \*\*Balancing Techniques\*\*: Applied SMOTE to address class imbalance.  
- \*\*Optuna for Tuning\*\*: Leveraged Optuna for efficient hyperparameter tuning using Bayesian optimization.

1. \*\*Clarity and Organization:\*\*  
- Use clear headings and subheadings.  
- Employ tables and bullet points for readability.

- Structured report with clear headings and subheadings.  
- Used tables and bullet points for readability.  
- \*\*Headings and Subheadings\*\*: Ensure the report is well-structured with clear sections.  
- \*\*Tables and Bullet Points\*\*: Use these elements to enhance readability.

- Ensure clear headings and subheadings throughout the report.

- Use tables and bullet points for better readability.  
- Provide detailed explanations for each section.

- Included plots to illustrate data distributions and model performance.  
- \*\*Include Relevant Plots\*\*: Ensure all visuals are well-labeled and easy to interpret.  
- \*\*Illustrate Key Points\*\*: Use visuals to support key findings and observations.

- Include relevant plots for data distributions, correlations, and model performance

- Ensure all visuals are well-labeled and easy to interpret.  
- Provide links to relevant Kaggle discussions, papers, or tutorials that informed the project.  
- \*\*KMeans:\*\* Adding clustering information to the dataset.

- Provided comments and explanations for key code snippets.  
- \*\*Comments and Explanations\*\*: Provide clear comments and explanations for key code snippets.  
- \*\*Effective Coding Techniques\*\*: Highlight any unique or effective coding practices.

- Provide comments and explanations for key code snippets.

- Highlight any unique or effective coding techniques.

- Covered all aspects from initial data exploration to final model evaluation.  
- Detailed explanations for decisions made throughout the process.  
- Covered all aspects from data exploration to final model evaluation.  
- \*\*From Data Exploration to Model Evaluation\*\*: Ensure all aspects of the project are covered.  
- \*\*Detailed Explanations\*\*: Provide thorough explanations for decisions made throughout the process.

- Cover all aspects of the project, from data exploration to final model evaluation.

- Provide detailed explanations for decisions made throughout the process.

- Suggested potential improvements such as exploring additional feature engineering techniques and trying different neural network architectures.  
- \*\*Feature Engineering\*\*: Investing time in feature engineering can significantly improve model performance.  
- \*\*Hyperparameter Tuning\*\*: Using tools like Optuna can efficiently optimize  
- \*\*Based on Findings\*\*: Include practical insights and recommendations.  
- \*\*Next Steps\*\*: Suggest potential next steps or further improvements.

- Include insights and recommendations based on project findings.

- Suggest potential next steps or improvements.

- \*\*Kaggle Discussions\*\*: Participated in discussions to gain insights and solutions from the community.  
- \*\*Documentation and Tutorials\*\*: Referred to official documentation and tutorials for libraries like scikit-learn, XGBoost, and Optuna.  
- \*\*External Resources\*\*: Leveraged external resources such as research papers and online courses to enhance understanding and implementation.  
- Kaggle discussions, papers, and tutorials were referenced to inform the project.  
- Relevant external resources and documentation links are provided.  
- \*\*Libraries and Documentation\*\*:  
- [pandas](https://pandas.pydata.org/): Data manipulation and analysis.  
- [numpy](https://numpy.org/): Numerical computations.  
- [scikit-learn](https://scikit-learn.org/stable/): Machine learning utilities.  
- [LightGBM](https://lightgbm.readthedocs.io/): Gradient boosting framework.  
- [Optuna](https://optuna.org/): Hyperparameter optimization.  
- [Seaborn](https://seaborn.pydata.org/): Statistical data visualization.  
- [Matplotlib](https://matplotlib.org/): Plotting library.  
- [TensorFlow](https://www.tensorflow.org/): Deep learning framework.  
- [Keras](https://keras.io/): High-level neural networks API.  
- \*\*Kaggle Discussions\*\*: [Kaggle forums](https://www.kaggle.com/discussions) provided insights and strategies for feature engineering and model tuning.  
- \*\*External Documentation\*\*: Libraries' official documentation (e.g., scikit-learn, XGBoost, LightGBM) for detailed usage guidelines.  
- \*\*Research Papers\*\*: Referenced papers on SMOTE and other balancing techniques to understand their theoretical underpinnings.  
- \*\*External Resources\*\*: Include references to any external resources or documentation used.  
- \*\*Relevant Links\*\*: Provide links to relevant Kaggle discussions, papers, or

- \*\*External Resources\*\*: Include references to any Kaggle discussions, papers, or tutorials used.

- \*\*Documentation Links\*\*: Provide links to relevant documentation for libraries and techniques used.

This report provides a comprehensive overview of the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. By following a structured approach to data preprocessing, model selection, hyperparameter tuning, and evaluation, we developed a robust and high-performing model. The insights and recommendations provided can serve as a guide for future projects and improvements.  
This comprehensive report provides a detailed overview of the techniques, strategies, models, and code used to construct a binary classification model for a Kaggle competition. From data preprocessing to final model evaluation, each step has been thoroughly documented, highlighting key insights and recommendations for future work. The project demonstrates the importance of feature engineering, data balancing, hyperparameter tuning, and thorough experiment tracking in building robust machine learning models.  
This report provides a thorough overview of the process involved in constructing a binary classification model for a Kaggle competition. It covers all aspects from data preprocessing to model evaluation, highlighting key techniques, models, and code used. The insights and recommendations provided can serve as a guide for future projects, ensuring a structured and effective approach to machine learning tasks.  
This comprehensive report details the entire process of constructing a binary classification model for a Kaggle competition. By leveraging various techniques, strategies, and models, we aimed to build a robust and accurate model. The insights and lessons learned provide valuable guidance for future projects, ensuring a thorough and effective approach to machine learning tasks.

This comprehensive report covers the entire process of constructing a binary classification model for a Kaggle competition. It includes detailed descriptions of techniques, strategies, models, and code used, along with practical insights and recommendations for future improvements.

- \*\*Data Cleaning\*\*: Handling missing values by imputing or removing them, ensuring no null values are present in the dataset.  
- \*\*Feature Engineering\*\*: Creating new features based on existing ones, such as interaction features and polynomial features.  
- \*\*Scaling\*\*: Standardizing numerical features to ensure they have a mean of 0 and a standard deviation of 1.  
1. \*\*Data Cleaning\*\*:  
- Loaded datasets from CSV files using pandas.  
- Sampled a fraction of the dataset for faster processing.  
- Dropped columns with limited variability (e.g., `Driving\_License`).  
1. \*\*Data Cleaning:\*\*  
- Removed missing values and handled inconsistencies in the dataset.  
- Replaced missing values with mean/median for continuous features and mode for categorical features.

### Methods for Data Exploration and Visualization

- \*\*Basic Analysis\*\*: Viewing dataset head, info, and description to understand the structure and basic statistics.  
- \*\*Correlation Analysis\*\*: Calculating and plotting the correlation matrix to identify relationships between numerical features.  
- \*\*Skewness and Distribution Analysis\*\*: Analyzing the skewness of numerical features and plotting their distributions.  
- \*\*Binning\*\*: Applying binning to selected continuous features and visualizing the binning results.

- \*\*Logistic Regression\*\*  
- \*\*Decision Trees\*\*  
- \*\*Random Forests\*\*  
- \*\*Gradient Boosting Machines (GBM)\*\*  
- \*\*XGBoost\*\*  
- \*\*LightGBM\*\*  
- \*\*Neural Networks (PyTorch)\*\*

python  
import os  
import pandas as pd  
import numpy as np  
from google.colab import drive  
python  
import pandas as pd  
from sklearn.preprocessing import StandardScaler  
from sklearn.model\_selection import train\_test\_split

python  
# Sample 40% of the training data  
train\_sample = train\_df.sample(frac=0.4, random\_state=42)  
python  
from sklearn.preprocessing import PolynomialFeatures

return self  
self.poly.fit(X[['Age', 'Annual\_Premium', 'Vintage']])  
return self  
self.poly.fit(X[['Age', 'Annual\_Premium', 'Vintage']])  
return self

python  
import torch  
import torch.nn as nn  
import torch.optim as optim  
from torch.cuda.amp import GradScaler, autocast  
from sklearn.metrics import roc\_auc\_score  
```python  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

python  
import optuna  
- Hyperparameters were manually adjusted based on observed performance during training.  
- Automated hyperparameter tuning techniques like GridSearchCV or Optuna were suggested but not implemented.  
- \*\*GridSearchCV\*\*: Exhaustive search over specified parameter values to find the best combination.  
- \*\*Optuna\*\*: Bayesian optimization framework used for more efficient hyperparameter tuning, especially for complex models like neural networks and gradient boosting machines.  
```python  
from sklearn.model\_selection import GridSearchCV  
- \*\*GridSearchCV\*\*: Used for exhaustive search over specified parameter values for Random Forests.  
- \*\*Optuna\*\*: Implemented for efficient hyperparameter optimization for LightGBM.  
- \*\*Manual Tuning\*\*: Applied to neural networks for epochs, learning rate, and architecture adjustments.

ratio = float(y\_train\_series.value\_counts()[0]) / y\_train\_series.value\_counts()[1]  
model = xgb.XGBClassifier(\*\*param, scale\_pos\_weight=ratio)

model.fit(X\_train\_preprocessed, y\_train\_series, eval\_set=[(X\_val\_preprocessed, y\_val)], verbose=True)

auc = roc\_auc\_score(y\_val, y\_pred)  
return auc

final\_model = xgb.XGBClassifier(\*\*best\_params)  
final\_model.fit(X\_train\_preprocessed, y\_train)  
```  
dtrain = xgb.DMatrix(data=X\_train\_preprocessed, label=y\_train)  
dval = xgb.DMatrix(data=X\_val\_preprocessed, label=y\_val)

- \*\*pandas and numpy\*\*: Used for data loading, cleaning, and preprocessing.  
- \*\*matplotlib and seaborn\*\*: Used for data exploration and visualization.  
- \*\*scikit-learn\*\*: Used for model training, evaluation, feature engineering, and preprocessing.  
- \*\*XGBoost and LightGBM\*\*: Used for building gradient boosting models.  
- \*\*Optuna\*\*: Used for efficient hyperparameter optimization.  
- \*\*PyTorch\*\*: Used for building and training neural networks.  
- \*\*umap\*\*: Used as an alternative to t-SNE for dimensionality reduction and visualization.

### Specific Combinations of Models and Preprocessing Techniques

- \*\*Random Forest with SMOTE\*\*: Used to balance the dataset before training the model.  
- \*\*XGBoost with Optuna\*\*: Used for hyperparameter tuning to find the best parameters for the model.  
- \*\*Autoencoder for Feature Extraction\*\*: Combined with PCA and t-SNE for visualization and understanding of the latent space.

### Different Configurations and Their Impacts on Model Performance

- \*\*Using different combinations of hyperparameters\*\*: Evaluated their impact on model performance using Optuna.  
- \*\*Testing different preprocessing techniques\*\*: Analyzed the impact of scaling, feature engineering, and balancing techniques on model performance.

### Challenges Faced and Solutions Implemented

- \*\*Large Dataset Size\*\*: Downsampled the dataset for initial exploration and testing to manage computational resources.  
- \*\*Imbalanced Data\*\*: Applied SMOTE to balance the dataset and improve model performance.  
- \*\*Hyperparameter Optimization\*\*: Used Optuna for efficient and effective hyperparameter tuning.  
- \*\*Computational Resources\*\*: Leveraged cloud computing resources for training models on large datasets.

- \*\*Data Preprocessing\*\*: Proper preprocessing is crucial for improving model performance.  
- \*\*  
- Importance of balancing dataset for improving recall.  
- Effectiveness of gradient boosting models in handling complex patterns.  
- Efficiency gains from automated hyperparameter tuning with Optuna.  
- \*\*Feature Engineering is Crucial\*\*: Creating new features and transforming existing ones can significantly impact model performance.  
- \*\*Data Imbalance Needs Attention\*\*: Addressing class imbalance is essential for developing robust models.  
- \*\*Hyperparameter Tuning is Key\*\*: Proper tuning of hyperparameters can drastically improve model performance and generalization.  
- \*\*Experiment Tracking\*\*: Keeping detailed logs and versioning datasets and models help in replicating results and understanding the impact of different strategies.  
- \*\*Importance of Data Preprocessing\*\*: Proper data cleaning, feature engineering, and scaling significantly impact model performance.  
- \*\*Model Evaluation\*\*: Cross-validation and comprehensive evaluation metrics are crucial for assessing model generalizability.

Feature Engineering\*\*: Creating meaningful features can significantly enhance model accuracy.

- \*\*Model Selection\*\*: Different models have varying strengths; it is essential to evaluate multiple models.  
- \*\*Hyperparameter Tuning\*\*: Efficient hyperparameter tuning can lead to substantial improvements in model performance.

- \*\*Further Exploration of Feature Engineering\*\*: Continue exploring new features and interactions.  
- \*\*Advanced Model Architectures\*\*: Experiment with more complex neural network architectures.  
- \*\*Ensemble Methods\*\*: Combine multiple models to improve performance.  
- \*\*Continuous Monitoring and Evaluation\*\*: Regularly update and evaluate the model with new data.  
- \*\*Continuous Feature Engineering\*\*: Keep exploring new features and transformations as they can provide new insights and improve model performance.  
- \*\*Advanced Imputation Techniques\*\*: For datasets with complex missing value patterns, consider using advanced imputation techniques like KNN or iterative imputation.  
- \*\*Balancing Techniques\*\*: Regularly evaluate and apply balancing techniques like SMOTE to handle class imbalances effectively.  
- \*\*Regular Hyperparameter Tuning\*\*: Regularly tune hyperparameters using efficient methods like Optuna or Bayesian optimization.  
- \*\*Comprehensive Logging and Versioning\*\*: Maintain detailed logs and version control for datasets and models to ensure reproducibility and better experiment tracking.

This report provides a detailed overview of the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. It includes all aspects of the modeling process from data preprocessing to final model evaluation, referencing our entire chat history.  
This report details the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. The goal is to provide a comprehensive overview of the entire modeling process, from initial data preprocessing to final model evaluation.

\*\*Data Cleaning:\*\*  
- \*\*Handling Missing Values:\*\* Missing values were imputed using mean/median for numerical features and mode for categorical features.  
- \*\*Feature Engineering:\*\*  
- \*\*Creation of new features:\*\* Polynomial features, interaction terms.  
- \*\*Encoding Categorical Variables:\*\* One-hot encoding for categorical features.  
- \*\*Handling Binary Variables:\*\* Mapping binary variables to 0 and 1.  
- \*\*Scaling:\*\* Continuous features were standardized using `StandardScaler`.

- \*\*Exploratory Data Analysis (EDA):\*\*  
- Histograms and box plots for understanding distributions.  
- Correlation heatmaps to identify relationships between features.  
- Pair plots and scatter plots to visualize feature interactions.  
- \*\*Libraries Used:\*\* Seaborn for stylish plots, Matplotlib for basic plotting.  
1. \*\*Exploratory Data Analysis (EDA):\*\*  
- Utilized `seaborn` and `matplotlib` for visualizing data distributions, correlations, and relationships between variables.  
- Generated histograms, box plots, and correlation heatmaps.

- \*\*SMOTE (Synthetic Minority Over-sampling Technique):\*\* Applied to handle class imbalance by generating synthetic samples for the minority class.

#### Attempted Models  
- \*\*Logistic Regression\*\*: Simple baseline model to establish a performance benchmark.  
- \*\*Decision Trees\*\*: Used for initial model building due to their interpretability.  
- \*\*Random Forests\*\*: Employed to enhance performance through ensemble learning.  
- \*\*Gradient Boosting Machines (GBM)\*\*: Implemented using XGBoost and LightGBM for improved accuracy and speed.  
- \*\*Neural Networks\*\*: Considered for capturing complex patterns in the data.

- \*\*Logistic Regression\*\*  
- \*\*Decision Trees\*\*  
- \*\*Random Forests\*\*  
- \*\*Gradient Boosting (XGBoost and LightGBM)\*\*  
- \*\*Neural Networks (PyTorch)\*\*

- \*\*Steps:\*\*  
- Initial baseline models were trained and evaluated.  
- Cross-validation using Stratified K-Fold to ensure balanced splits.  
- Performance metrics: accuracy, precision, recall, F1 score.  
- \*\*Reasoning:\*\*  
- Models were selected based on their performance on cross-validation metrics.  
- Simpler models (Logistic Regression) were compared with more complex ones (Gradient Boosting, Neural Networks).  
- \*\*Model Selection:\*\* Chose models based on their performance in initial runs and their suitability for binary classification.  
- \*\*Evaluation Metrics:\*\* Used ROC-AUC as the primary evaluation metric.  
- \*\*Cross-Validation:\*\* Employed Stratified K-Fold Cross-Validation to ensure balanced class distribution in training and validation sets.  
- Used AUC-ROC as the primary evaluation metric.  
- Performed cross-validation to assess model performance.  
- Chose LightGBM and Neural Networks based on initial performance metrics.

- \*\*Methods Used:\*\*  
- \*\*GridSearchCV:\*\* Exhaustive search over specified parameter values.  
- \*\*Optuna:\*\* Advanced optimization framework for more efficient hyperparameter tuning.  
- \*\*Reasoning:\*\* To improve model performance by finding the optimal set of hyperparameters.  
python  
from optuna import create\_study

python  
import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
from imblearn.over\_sampling import SMOTE  
```python  
# Load datasets  
train\_df = pd.read\_csv("train.csv", index\_col='id')  
test\_df = pd.read\_csv("test.csv", index\_col='id')

data = pd.read\_csv('dataset.csv')  
data = pd.read\_csv('data.csv')  
data = pd.read\_csv('data.csv')  
data = pd.read\_csv('data.csv')  
df = pd.read\_csv('data.csv')  
train\_df = pd.read\_csv("train.csv", index\_col='id')  
test\_df = pd.read\_csv("test.csv", index\_col='id')  
train\_df = pd.read\_csv('train.csv')

# Split data into features and target

X = data.drop('target', axis=1)  
y = data['target']  
X = train\_sample.drop('Response', axis=1)  
y = train\_sample['Response']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
```

```python  
from sklearn.preprocessing import StandardScaler, OneHotEncoder  
from sklearn.compose import ColumnTransformer  
from sklearn.pipeline import Pipeline

preprocessor = ColumnTransformer(  
transformers=[  
('num', StandardScaler(), ['num\_feature1', 'num\_feature2']),  
('cat', OneHotEncoder(), ['cat\_feature1', 'cat\_feature2'])  
]  
)

python  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.metrics import classification\_report  
```python  
class BinaryClassificationModel(nn.Module):  
def \_\_init\_\_(self, input\_dim):  
super(BinaryClassificationModel, self).\_\_init\_\_()  
self.fc1 = nn.Linear(input\_dim, 128)  
self.bn1 = nn.BatchNorm1d(128)  
self.dropout1 = nn.Dropout(p=0.5)  
self.fc2 = nn.Linear(128, 64)  
self.bn2 = nn.BatchNorm1d(64)  
self.dropout2 = nn.Dropout(p=0.5)  
self.fc3 = nn.Linear(64, 32)  
self.bn3 = nn.BatchNorm1d(32)  
self.dropout3 = nn.Dropout(p=0.5)  
self.fc4 = nn.Linear(32, 1)

model = XGBClassifier()  
model.fit(X\_train, y\_train)  
model = RandomForestClassifier(random\_state=42)  
model.fit(X\_train, y\_train)  
model = RandomForestClassifier()  
model.fit(X\_train, y\_train)  
model = RandomForestClassifier(n\_estimators=100, random\_state=42)  
model.fit(X\_train\_balanced, y\_train\_balanced)  
model = RandomForestClassifier(n\_estimators=100, random\_state=42)  
model.fit(X\_train\_balanced, y\_train\_balanced)

accuracy = accuracy\_score(y\_test, y\_pred)  
precision = precision\_score(y\_test, y\_pred)  
recall = recall\_score(y\_test, y\_pred)  
f1 = f1\_score(y\_test, y\_pred)  
y\_train\_pred = model.predict(X\_train, num\_iteration=model.best\_iteration)  
y\_val\_pred = model.predict(X\_val, num\_iteration=model.best\_iteration)  
train\_auc = roc\_auc\_score(y\_train, y\_train\_pred)  
val\_auc = roc\_auc\_score(y\_val, y\_val\_pred)  
```

print(f'Accuracy: {accuracy}, Precision: {precision}, Recall: {recall}, F1 Score: {f1}')

param\_grid = {  
'n\_estimators': [100, 200],  
'learning\_rate': [0.01, 0.1],  
'max\_depth': [3, 5, 7]  
}  
param\_grid = {  
'n\_estimators': [100, 200],  
'learning\_rate': [0.01, 0.1],  
'max\_depth': [3, 5, 7]  
}

grid\_search = GridSearchCV(XGBClassifier(), param\_grid, cv=3, scoring='f1')  
grid\_search.fit(X\_train, y\_train)  
xgb = XGBClassifier()  
grid\_search = GridSearchCV(estimator=xgb, param\_grid=param\_grid, scoring='f1', cv=5)  
grid\_search.fit(X\_train, y\_train)

best\_params = grid\_search.best\_params\_  
print(f'Best parameters: {best\_params}')  
  
print(grid\_search.best\_params\_)  
```

```python  
# Example of a custom function for SMOTE  
from imblearn.over\_sampling import SMOTE

sm = SMOTE(random\_state=42)  
X\_res, y\_res = sm.fit\_resample(X, y)  
return X\_res, y\_res

X\_train, y\_train = apply\_smote(X\_train, y\_train)  
  
smote = SMOTE(random\_state=42)  
X\_train, y\_train = smote.fit\_resample(X\_train, y\_train)

- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical computing.  
- \*\*scikit-learn\*\*: Model building and evaluation, including cross-validation and metrics.  
- \*\*lightgbm\*\*: Implementation of LightGBM model.  
- \*\*optuna\*\*: Hyperparameter optimization.  
- \*\*matplotlib\*\* and \*\*seaborn\*\*: Data visualization.  
The following libraries were used throughout the project:  
- \*\*Pandas\*\*: For data loading, manipulation, and preprocessing.  
- \*\*NumPy\*\*: For numerical operations.  
- \*\*Torch\*\*: For building and training neural networks.  
- \*\*Sklearn\*\*:

- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical computing.  
- \*\*scikit-learn\*\*: Machine learning utilities.  
- \*\*XGBoost\*\*: Gradient boosting framework.  
- \*\*LightGBM\*\*: Gradient boosting framework.  
- \*\*Optuna\*\*: Hyperparameter optimization.  
- \*\*PyTorch\*\*: Deep learning framework.  
- \*\*Seaborn\*\*: Statistical data visualization.  
- \*\*Matplotlib\*\*: Plotting library.  
- \*\*imblearn\*\*: Handling imbalanced datasets (SMOTE).  
- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical computing.  
- \*\*scikit-learn\*\*: Machine learning tools.  
- \*\*XGBoost\*\*: Gradient boosting.  
- \*\*LightGBM\*\*: Gradient boosting.  
- \*\*Optuna\*\*: Hyperparameter optimization.  
- \*\*PyTorch\*\*: Deep learning.  
- \*\*Seaborn\*\*: Data visualization.  
- \*\*Matplotlib\*\*: Plotting.

- \*\*pandas and numpy:\*\* Data loading, preprocessing, and manipulation.  
- \*\*scikit-learn:\*\* Preprocessing, model selection, evaluation, and hyperparameter tuning.  
- \*\*XGBoost and LightGBM:\*\* Implementing and tuning gradient boosting models.  
- \*\*Optuna:\*\* Efficient hyperparameter tuning.  
- \*\*PyTorch:\*\* Constructing and training neural networks.  
- \*\*Seaborn and Matplotlib:\*\* Data visualization and exploratory analysis.  
- \*\*imblearn:\*\* Applying SMOTE for dataset balancing.

Various combinations of preprocessing techniques and model configurations were tested to identify the optimal setup for the competition. The impact of different configurations on model performance was meticulously recorded.

- \*\*Model + Preprocessing Combinations:\*\*  
- Logistic Regression with StandardScaler and OneHotEncoder.  
- XGBoost with StandardScaler and OneHotEncoder.  
- LightGBM with SMOTE and Polynomial Features.  
- \*\*Impact on Performance:\*\*  
- Standard scaling improved performance for models sensitive to feature scales.  
- SMOTE enhanced recall by balancing the classes.  
- Polynomial features provided marginal improvements for complex models like gradient boosting.

- \*\*MLflow\*\*: Used for experiment tracking, model versioning, and logging. Each iteration of the model training was logged with its corresponding parameters and performance metrics.  
- \*\*Logging\*\*: Logging was used extensively to track the progress and debug issues during data preprocessing and model training.  
- \*\*Manual Recording\*\*: Metrics and configurations were manually recorded to evaluate model performance.

- \*\*Setup:\*\* Integrated MLflow for tracking experiments, parameters, and metrics.  
- \*\*Versioning:\*\* Each model iteration and parameter set were logged for reproducibility.  
- \*\*Logging:\*\* Metrics such as accuracy, precision, recall, F1 score, and confusion matrices were recorded.

- \*\*Data Imbalance\*\*: Addressed using SMOTE to balance the target variable classes.  
- \*\*Computational Resources\*\*: Managed by sampling a subset of the data for initial exploration and model training.  
- \*\*Hyperparameter Tuning\*\*: Switched from GridSearchCV to Optuna for more efficient and effective hyperparameter optimization.  
- \*\*Handling Mixed Data Types\*\*: Ensured uniform data types before encoding categorical features.  
- \*\*Model Evaluation\*\*: Used ROC AUC score instead of accuracy due to class imbalance, providing a more informative measure of model performance.  
- \*\*Error Handling in Training\*\*: Updated loss function to `BCEWithLogitsLoss` to resolve issues with mixed precision training.

- \*\*Class Imbalance:\*\* Addressed using SMOTE.  
- \*\*Hyperparameter Tuning:\*\* Handled by using GridSearchCV and Optuna for efficient optimization.  
- \*\*Feature Engineering:\*\* Balancing between too many features and avoiding overfitting.  
- Handling imbalanced datasets.  
- Selecting optimal hyperparameters.  
- Ensuring reproducibility of results.  
- Handling imbalanced data.  
- Optimizing hyperparameters.  
- Ensuring GPU utilization.

- \*\*SMOTE:\*\* Effectively balanced the dataset, improving recall.  
- \*\*Optuna:\*\* Efficiently searched the hyperparameter space, finding better models.  
- \*\*Feature Selection:\*\* Used cross-validation to select the most relevant features.  
- Applied SMOTE to balance the dataset.  
- Used Optuna for efficient hyperparameter tuning.  
- Employed MLflow for experiment tracking and reproducibility.  
- Applied SMOTE for balancing.  
- Used Optuna for efficient hyperparameter tuning.  
- Verified GPU utilization using device checks and ensured data and models were moved to GPU.

- \*\*Data preprocessing:\*\* Crucial for model performance.  
- \*\*Hyperparameter tuning:\*\* Significantly impacts model accuracy.  
- \*\*Experiment tracking:\*\* Essential for reproducibility and iterative improvement.  
- Leveraging GPU acceleration can significantly reduce preprocessing time.  
- Automated machine learning tools like FLAML can streamline model

- \*\*Clarity and Organization\*\*: Ensure the report is well-organized with clear headings and subheadings.  
- \*\*Visuals\*\*: Include relevant plots and visualizations to illustrate key points.  
- \*\*Code Readability\*\*: Provide comments and explanations for key code snippets.  
- \*\*Comprehensive Coverage\*\*: Ensure all aspects of the project are covered, from initial data exploration to final model evaluation.  
- \*\*Practical Insights\*\*: Include practical insights and recommendations based on the findings of the project.  
- \*\*Data Balancing\*\*: Implement techniques like SMOTE to balance the dataset and potentially improve model performance.  
- \*\*Hyperparameter Tuning\*\*: Utilize automated tuning methods such as GridSearchCV or Optuna for better hyperparameter optimization.  
- \*\*Advanced Model Architectures\*\*: Experiment with more complex neural network architectures or other machine learning models like XGBoost and LightGBM.

- \*\*Headings and Subheadings:\*\* Ensure clear organization.  
- \*\*Tables and Bullet Points:\*\* Enhance readability and clarity.  
- Ensure clear headings and subheadings.  
- Use bullet points and tables for readability.

- \*\*Include Relevant Plots:\*\* Data distributions, model performance metrics.  
- \*\*Label and Interpret:\*\* Ensure visuals are well-labeled and easy to understand.  
- Include relevant plots and visualizations.  
- Ensure all visuals are well-labeled.

- \*\*Comments and Explanations:\*\* Make key code snippets easy to understand.  
- \*\*Highlight Effective Techniques:\*\* Showcase unique or particularly effective coding methods.  
- Comment key code snippets.  
- Highlight effective coding techniques.

- \*\*Detailed Explanations:\*\* Provide thorough coverage from data exploration to final evaluation.  
- \*\*Decision Rationale:\*\* Explain the reasoning behind each decision made.  
- Cover all project aspects, from data exploration to model evaluation.  
- Provide detailed explanations for decisions.

- \*\*Recommendations:\*\* Offer insights and potential next steps based on findings.  
- \*\*Further Improvements:\*\* Suggest areas for further optimization or exploration.  
- \*\*GPU Utilization\*\*: For large datasets, GPU-accelerated libraries like cuDF and cuML can provide substantial speedups.  
- \*\*Automated ML\*\*: Tools like FLAML can optimize the hyperparameter tuning process efficiently.  
- \*\*Class Imbalance\*\*: Techniques like SMOTE are essential for dealing with imbalanced datasets.  
- Include practical insights and recommendations.  
- Suggest potential next steps or further improvements.

- \*\*External Resources:\*\* Include references to relevant documentation, papers, and tutorials.  
- \*\*Kaggle Discussions:\*\* Link to relevant discussions and insights from the Kaggle community.

This report provides a detailed and comprehensive overview of the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. By leveraging advanced preprocessing techniques, robust model selection, and efficient hyperparameter tuning, the project aimed to achieve optimal performance in the competition. The lessons learned and insights gained from this process are invaluable for future machine learning endeavors.  
This report provides a comprehensive overview of the binary classification model built for the Kaggle competition, detailing the techniques, models, and code used. The project highlights the importance of data preprocessing, model selection, and evaluation metrics in achieving robust model performance.  
This comprehensive report covers the entire modeling process, from initial data exploration to final model evaluation. The strategies and techniques employed have been documented in detail, with practical insights and recommendations for future improvements. The use of various models, preprocessing techniques, and hyperparameter tuning methods has been thoroughly explained, providing a clear understanding of the entire project workflow.

This comprehensive report covers the entire process of constructing a binary classification model for a Kaggle competition. It includes detailed explanations of data preprocessing, model selection, hyperparameter tuning, and experiment tracking, providing a clear understanding of the methodologies and techniques employed. The insights and recommendations offered aim to guide future projects and improvements.

- \*\*Models:\*\* Some suggested models (e.g., SVM, KNN) were not pursued due to computational limitations or lower expected performance.  
- \*\*Feature Engineering:\*\* Certain techniques were not implemented due to time constraints or complexity.  
- \*\*Visualization Methods:\*\* Proposed methods were not utilized to maintain focus on the most informative plots.  
- \*\*Hyperparameter Tuning:\*\* Some strategies were not applied as they were deemed less efficient.  
- \*\*Preprocessing Steps:\*\* Recommended steps were skipped if they did not align with the overall strategy.

- \*\*Kaggle Discussions\*\*: References to relevant discussions and solutions.  
- \*\*Papers and Tutorials\*\*: Links to papers and tutorials that informed the project.  
- Included references to external documentation and resources used.  
- [Kaggle Competition Page](https://www.kaggle.com/competitions)  
- [PyTorch Documentation](https://pytorch.org/docs/stable/index.html)  
- [Scikit-learn Documentation](https://scikit-learn.org/stable/documentation.html)  
- [Seaborn Documentation](https://seaborn.pydata.org/)  
- [Matplotlib Documentation](https://matplotlib.org/stable/contents.html)  
- \*\*Kaggle Discussions\*\*: Insights and tips from the community.  
- \*\*Papers and Tutorials\*\*: References for understanding SMOTE, hyperparameter tuning, and advanced modeling techniques.  
- \*\*Documentation\*\*: Official documentation for libraries like scikit-learn, PyTorch,

- \*\*NumPy Documentation:\*\* [numpy.org](https://numpy.org/)  
- \*\*Scikit-learn Documentation:\*\* [scikit-learn.org](https://scikit-learn.org/)  
- \*\*XGBoost Documentation:\*\* [xgboost.readthedocs.io](https://xgboost.readthedocs.io/)  
- \*\*LightGBM Documentation:\*\* [lightgbm.readthedocs.io](https://lightgbm.readthedocs.io/)  
- \*\*Optuna Documentation:\*\* [optuna.readthedocs.io](https://optuna.readthedocs.io/)  
- \*\*PyTorch Documentation:\*\* [pytorch.org](https://pytorch.org/)  
- \*\*Seaborn Documentation:\*\* [seaborn.pydata.org](https://seaborn.pydata.org/)  
- \*\*Matplotlib Documentation:\*\* [matplotlib.org](https://matplotlib.org/)  
- \*\*Imbalanced-learn Documentation:\*\* [imbalanced-learn.org](https://imbalanced-learn.org/)

This report serves as a comprehensive guide for understanding the techniques, strategies, and methodologies employed in constructing an effective binary classification model for the Kaggle competition.

- Removed irrelevant columns (e.g., `id`).  
- Transformed binary categorical variables to numerical format.  
- Removed irrelevant columns (e.g., `id`).  
- Transformed binary categorical variables to numerical format.  
- Removed irrelevant columns (e.g., `id`).  
- Transformed binary categorical variables to numerical format.  
- Loading the dataset using pandas.  
- Applying `klibs.clean\_data` function to clean the dataset.  
- Handling missing values appropriately.

- Checked for missing values and decided on appropriate imputation strategies if needed. However, in this dataset, there were no missing values to address.  
- Checked for missing values and decided on appropriate imputation strategies if needed. However, in this dataset, there were no missing values to address.  
- Checked for missing values and decided on appropriate imputation strategies if needed. However, in this dataset, there were no missing values to address.  
- Utilized `pandas` methods to detect and fill missing values where necessary.  
- Checked for missing values in the dataset and handled them appropriately (no missing values were found in the provided dataset).  
- Employed strategies like mean/median imputation for numerical features and mode imputation for categorical features.  
- Used advanced imputation techniques like KNN for more complex datasets.

- Applied one-hot encoding to categorical variables like `Region\_Code`, `Vehicle\_Age`, and `Policy\_Sales\_Channel`.  
- Applied one-hot encoding to categorical variables like `Region\_Code`, `Vehicle\_Age`, and `Policy\_Sales\_Channel`.  
- Applied one-hot encoding to categorical variables like `Region\_Code`, `Vehicle\_Age`, and `Policy\_Sales\_Channel`.  
- Created interaction features and polynomial features to capture non-linear relationships.  
- Applied custom transformers using `sklearn.base.BaseEstimator` and `TransformerMixin`.  
- \*\*Binary Variables Transformation\*\*: Mapped binary variables to numerical values (e.g., `Gender`, `Vehicle\_Damage`).  
- \*\*Categorical Variables Grouping\*\*: Grouped rare categories to reduce dimensionality.  
- \*\*Handling Outliers\*\*: Calculated Interquartile Range (IQR) for `Annual\_Premium` and removed outliers.  
- \*\*Standardization\*\*: Standardized continuous numeric variables using `StandardScaler`.  
- Created new features based on domain knowledge and data insights.  
- Transformed categorical features using one-hot encoding and label encoding.  
- Engineered interaction terms and polynomial features to capture non-linear relationships.

- Standardized continuous variables to have a mean of 0 and a standard deviation of 1.  
- Standardized continuous variables to have a mean of 0 and a standard deviation of 1.  
- Standardized continuous variables to have a mean of 0 and a standard deviation of 1.  
- Used `StandardScaler` from `sklearn.preprocessing` to scale features while preserving feature names.  
- Applied scaling to continuous numeric variables to ensure they have a mean of 0 and standard deviation of 1.  
- Standardized numerical features using `StandardScaler` to ensure they have a mean of 0 and a standard deviation of 1.  
- Applied `MinMaxScaler` to normalize the data within a specific range.

scaler = StandardScaler()  
train\_df[continuous\_numeric] = scaler.fit\_transform(train\_df[continuous\_numeric])  
  
scaler = StandardScaler()  
train\_df[continuous\_numeric] = scaler.fit\_transform(train\_df[continuous\_numeric])  
```

- Visualized the distributions of continuous variables using histograms and KDE plots.

- Created count plots for binary variables to understand their distributions.  
- Generated a correlation matrix with a triangular mask to identify relationships between variables.  
- Created count plots for binary variables to understand their distributions.  
- Generated a correlation matrix with a triangular mask to identify relationships between variables.  
- Created count plots for binary variables to understand their distributions.  
- Generated a correlation matrix with a triangular mask to identify relationships between variables.

python  
from sklearn.model\_selection import cross\_val\_score  
python  
from sklearn.model\_selection import cross\_val\_score  
python  
from sklearn.model\_selection import cross\_val\_score  
```python  
from sklearn.model\_selection import cross\_val\_score

# Plot distributions of continuous variables

fig, axes = plt.subplots(1, 3, figsize=(18, 5))  
for i, col in enumerate(continuous\_numeric):  
sns.histplot(train\_df[col], ax=axes[i], kde=True, color="skyblue")  
axes[i].set\_title(f'Distribution of {col}')  
plt.tight\_layout()  
plt.show()  
fig, axes = plt.subplots(1, 3, figsize=(18, 5))  
for i, col in enumerate(continuous\_numeric):  
sns.histplot(train\_df[col], ax=axes[i], kde=True, color="skyblue")  
axes[i].set\_title(f'Distribution of {col}')  
plt.tight\_layout()  
plt.show()  
fig, axes = plt.subplots(1, 3, figsize=(18, 5))  
for i, col in enumerate(continuous\_numeric):  
sns.histplot(train\_df[col], ax=axes[i], kde=True, color="skyblue")  
axes[i].set\_title(f'Distribution of {col}')  
plt.tight\_layout()  
plt.show()

corr\_matrix = train\_df.corr()  
mask = np.triu(np.ones\_like(corr\_matrix, dtype=bool))  
plt.figure(figsize=(15, 10))  
sns.heatmap(corr\_matrix, mask=mask, annot=True, cmap='coolwarm', fmt='.2f', square=True, linewidths=.5)  
plt.title('Correlation Heatmap with Triangular Mask')  
plt.show()  
  
corr\_matrix = train\_df.corr()  
mask = np.triu(np.ones\_like(corr\_matrix, dtype=bool))  
plt.figure(figsize=(15, 10))  
sns.heatmap(corr\_matrix, mask=mask, annot=True, cmap='coolwarm', fmt='.2f', square=True, linewidths=.5)  
plt.title('Correlation Heatmap with Triangular Mask')  
plt.show()  
```

- Addressed class imbalance using techniques such as SMOTE (Synthetic Minority Over-sampling Technique).

X\_resampled, y\_resampled = smote.fit\_resample(X, y)  
  
X\_resampled, y\_resampled = smote.fit\_resample(X, y)  
```

- A simple baseline model to understand the relationships in the data.  
- A simple baseline model to understand the relationships in the data.  
- A simple baseline model to understand the relationships in the data.  
- A simple baseline model to understand the relationships in the data.  
- Basic binary classification model.  
2. \*\*Decision Trees\*\*:  
- Simple tree-based model.  
3. \*\*Random Forests\*\*:  
- Ensemble method combining multiple decision trees.  
4. \*\*Gradient Boosting\*\*:  
- Boosting method for improving performance.  
5. \*\*Neural Networks\*\*:  
- Using PyTorch for deep learning-based model.  
6. \*\*XGBoost\*\*:  
- Gradient boosting framework optimized for performance.  
7. \*\*LightGBM\*\*:  
- Gradient boosting framework designed for speed and efficiency.  
- A basic linear model used as a baseline.

- Captured non-linear relationships in the data.  
- Captured non-linear relationships in the data.  
- Captured non-linear relationships in the data.  
- Captured non-linear relationships in the data.  
- Provided interpretability but prone to overfitting.  
- Tree-based model that splits the data based on feature values.  
- Prone to overfitting but easy to visualize and understand.  
- A non-linear model capturing feature interactions.

- Improved model stability and performance by using an ensemble of decision trees.  
- Improved model stability and performance by using an ensemble of decision trees.  
- Improved model stability and performance by using an ensemble of decision trees.  
- Improved model stability and performance by using an ensemble of decision trees.  
- Ensemble method to reduce overfitting.  
- Evaluated using cross-validation.  
- Ensemble method that uses multiple decision trees.  
- Reduces overfitting and improves generalization.  
- An ensemble model that averages multiple decision trees to reduce overfitting.

- Enhanced model performance by sequentially building trees to correct errors of the previous ones.  
- Enhanced model performance by sequentially building trees to correct errors of the previous ones.  
- Enhanced model performance by sequentially building trees to correct errors of the previous ones.  
- Enhanced model performance by sequentially building trees to correct errors of the previous ones.  
- Models like XGBoost were used due to their efficiency and performance.  
- Sequentially builds models by correcting the errors of previous models.  
- Models like XGBoost and LightGBM were used.

- Captured complex patterns in the data using a neural network model with PyTorch.  
- Captured complex patterns in the data using a neural network model with PyTorch.  
- Captured complex patterns in the data using a neural network model with PyTorch.  
- Captured complex patterns in the data using a neural network model with PyTorch.  
- Suggested but not pursued in the final implementation due to computational complexity and longer training times.  
- Deep learning models for capturing complex patterns.  
- Utilized `Keras` and `TensorFlow` for implementation.  
- Implemented using `PyTorch` for complex patterns and interactions.

def \_\_init\_\_(self, input\_dim):  
super(Net, self).\_\_init\_\_()  
self.fc1 = nn.Linear(input\_dim, 64)  
self.fc2 = nn.Linear(64, 32)  
self.fc3 = nn.Linear(32, 1)  
def \_\_init\_\_(self, input\_dim):  
super(Net, self).\_\_init\_\_()  
self.fc1 = nn.Linear(input\_dim, 64)  
self.fc2 = nn.Linear(64, 32)  
self.fc3 = nn.Linear(32, 1)  
def \_\_init\_\_(self, input\_dim):  
super(Net, self).\_\_init\_\_()  
self.fc1 = nn.Linear(input\_dim, 64)  
self.fc2 = nn.Linear(64, 32)  
self.fc3 = nn.Linear(32, 1)  
def \_\_init\_\_(self, input\_dim):  
super(Net, self).\_\_init\_\_()  
self.fc1 = nn.Linear(input\_dim, 64)  
self.fc2 = nn.Linear(64, 32)  
self.fc3 = nn.Linear(32, 1)

x = torch.relu(self.fc1(x))  
x = torch.relu(self.fc2(x))  
x = torch.sigmoid(self.fc3(x))  
return x  
  
x = torch.relu(self.fc1(x))  
x = torch.relu(self.fc2(x))  
x = torch.sigmoid(self.fc3(x))  
return x  
  
x = torch.relu(self.bn1(self.fc1(x)))  
x = self.dropout1(x)  
x = torch.relu(self.bn2(self.fc2(x)))  
x = self.dropout2(x)  
x = torch.relu(self.bn3(self.fc3(x)))  
x = self.dropout3(x)  
x = self.fc4(x)  
return x  
return self.model.predict\_proba(x)[:, 1]  
x = self.relu(self.fc1(x))  
x = self.dropout(x)  
x = self.relu(self.fc2(x))  
x = self.dropout(x)  
x = self.fc3(x)  
return x

- Used cross-validation to evaluate model performance and select the best model.

- Metrics used for evaluation included accuracy, precision, recall, and F1 score.  
- Metrics used for evaluation included accuracy, precision, recall, and F1 score.  
- Metrics used for evaluation included accuracy, precision, recall, and F1 score.  
- Metrics used for evaluation included accuracy, precision, recall, and F1 score.

scores = cross\_val\_score(model, X\_resampled, y\_resampled, cv=5, scoring='f1')  
print(f'F1 Score: {scores.mean()}')  
  
scores = cross\_val\_score(model, X\_resampled, y\_resampled, cv=5, scoring='f1')  
print(f'F1 Score: {scores.mean()}')  
  
model.fit(X\_train, y\_train)

- Applied GridSearchCV for hyperparameter tuning to find the best parameters for the models.

'n\_estimators': [100, 200, 300],  
'max\_depth': [None, 10, 20, 30],  
'min\_samples\_split': [2, 5, 10]  
}  
grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=3, scoring='f1')  
grid\_search.fit(X\_resampled, y\_resampled)  
best\_model = grid\_search.best\_estimator\_  
  
'n\_estimators': [100, 200, 300],  
'max\_depth': [None, 10, 20, 30],  
'min\_samples\_split': [2, 5, 10]  
}  
grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=3, scoring='f1')  
grid\_search.fit(X\_train, y\_train)  
best\_model = grid\_search.best\_estimator\_  
'n\_estimators': [100, 200, 300],  
'max\_depth': [None, 10, 20, 30],  
'min\_samples\_split': [2, 5, 10]  
}  
grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=3, scoring='f1')  
grid\_search.fit(X\_resampled, y\_resampled)  
best\_model = grid\_search.best\_estimator\_  
  
'n\_estimators': [100, 200, 300],  
'max\_depth': [None, 10, 20, 30],  
'min\_samples\_split': [2, 5, 10]  
}  
grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=3, scoring='f1')  
grid\_search.fit(X\_train, y\_train)  
best\_model = grid\_search.best\_estimator\_

train\_df = pd.read\_csv("/content/drive/My Drive/Kaggle Competition/train.csv", index\_col='id')  
test\_df = pd.read\_csv("/content/drive/My Drive/Kaggle Competition/test.csv", index\_col='id')  
  
train\_df = pd.read\_csv("/content/drive/My Drive/Kaggle Competition/train.csv", index\_col='id')  
test\_df = pd.read\_csv("/content/drive/My Drive/Kaggle Competition/test.csv", index\_col='id')

train\_df = pd.get\_dummies(train\_df, columns=categorical, prefix=categorical)  
train\_df = pd.get\_dummies(train\_df, columns=categorical, prefix=categorical)  
train\_df = pd.get\_dummies(train\_df, columns=categorical, prefix=categorical)  
train\_df = pd.get\_dummies(train\_df, columns=categorical, prefix=categorical)

continuous\_numeric = ['Age', 'Vintage', 'Annual\_Premium']  
scaler = StandardScaler()  
train\_df[continuous\_numeric] = scaler.fit\_transform(train\_df[continuous\_numeric])  
  
continuous\_numeric = ['Age', 'Vintage', 'Annual\_Premium']  
scaler = StandardScaler()  
train\_df[continuous\_numeric] = scaler.fit\_transform(train\_df[continuous\_numeric])

y = train\_df['Response']  
y = train\_df['Response']  
y = train\_df['Response']  
y = train\_df['Response']

model = RandomForestClassifier()  
model.fit(X\_train, y\_train)  
model = RandomForestClassifier()  
model.fit(X\_train, y\_train)  
model = RandomForestClassifier()  
model.fit(X\_train, y\_train)  
model = RandomForestClassifier()  
model.fit(X\_train, y\_train)

print(f'Accuracy: {accuracy\_score(y\_test, y\_pred)}')  
print(f'Precision: {precision\_score(y\_test, y\_pred)}')  
print(f'Recall: {recall\_score(y\_test, y\_pred)}')  
print(f'F1 Score: {f1\_score(y\_test, y\_pred)}')  
  
print(f'Accuracy: {accuracy\_score(y\_test, y\_pred)}')  
print(f'Precision: {precision\_score(y\_test, y\_pred)}')  
print(f'Recall: {recall\_score(y\_test, y\_pred)}')  
print(f'F1 Score: {f1\_score(y\_test, y\_pred)}')

print(f'Best Model F1 Score: {f1\_score(y\_test, y\_pred\_best)}')  
  
print(f'Best Model F1 Score: {f1\_score(y\_test, y\_pred\_best)}')

- \*\*pandas\*\*: Data manipulation and analysis.

- \*\*numpy\*\*: Numerical computing.  
- \*\*scikit-learn\*\*: Machine learning algorithms and tools.  
- \*\*imblearn\*\*: Techniques for handling imbalanced datasets.  
- \*\*seaborn\*\* and \*\*matplotlib\*\*: Data visualization.  
- \*\*torch\*\*: Building and training neural networks.  
- \*\*numpy\*\*: Numerical computing.  
- \*\*scikit-learn\*\*: Machine learning algorithms and tools.  
- \*\*imblearn\*\*: Techniques for handling imbalanced datasets.  
- \*\*seaborn\*\* and \*\*matplotlib\*\*: Data visualization.  
- \*\*torch\*\*: Building and training neural networks.  
- \*\*numpy\*\*: Numerical computing.  
- \*\*scikit-learn\*\*: Machine learning algorithms and tools.  
- \*\*imblearn\*\*: Techniques for handling imbalanced datasets.  
- \*\*seaborn\*\* and \*\*matplotlib\*\*: Data visualization.  
- \*\*torch\*\*: Building and training neural networks.  
- \*\*numpy\*\*: Numerical computing.  
- \*\*scikit-learn\*\*: Machine learning algorithms and tools.  
- \*\*imblearn\*\*: Techniques for handling imbalanced datasets.  
- \*\*seaborn\*\* and \*\*matplotlib\*\*: Data visualization.  
- \*\*torch\*\*: Building and training neural networks.  
- \*\*numpy\*\*: Numerical computing.  
- \*\*scikit-learn\*\*: Machine learning tools and evaluation.  
- \*\*XGBoost\*\*: Gradient boosting framework.  
- \*\*LightGBM\*\*: Light gradient boosting framework.  
- \*\*Optuna\*\*: Hyperparameter optimization.  
- \*\*PyTorch\*\*: Deep learning framework.  
- \*\*Seaborn\*\*: Data visualization.  
- \*\*Matplotlib\*\*: Plotting library.  
- \*\*SMOTE\*\*: Synthetic Minority Over-sampling Technique for balancing datasets.  
- \*\*numpy\*\*: Numerical operations.  
- \*\*seaborn\*\*: Data visualization.  
- \*\*matplotlib\*\*: Plotting graphs.  
- \*\*scikit-learn\*\*: Machine learning algorithms and utilities.  
- \*\*imblearn\*\*: Handling imbalanced datasets (e.g., SMOTE).  
- \*\*lightgbm\*\*: Gradient boosting framework.  
- \*\*skopt\*\*: Bayesian optimization for hyperparameter tuning.  
- \*\*numpy\*\*: Numerical operations.  
- \*\*scikit-learn\*\*: Machine learning algorithms and utilities.  
- \*\*XGBoost\*\*: Gradient boosting library.  
- \*\*LightGBM\*\*: Gradient boosting library.  
- \*\*Optuna\*\*: Hyperparameter optimization.  
- \*\*PyTorch\*\*: Neural network implementation.  
- \*\*Seaborn\*\*: Data visualization.  
- \*\*Matplotlib\*\*: Plotting graphs and visualizations.

- \*\*pandas\*\*: Loading data, data cleaning, and preprocessing.

- \*\*numpy\*\*: Array operations and mathematical computations.  
- \*\*scikit-learn\*\*: Model training, evaluation, and hyperparameter tuning.  
- \*\*imblearn\*\*: Balancing the dataset using SMOTE.  
- \*\*seaborn\*\* and \*\*matplotlib\*\*: Creating plots and visualizations.  
- \*\*torch\*\*: Implementing neural network models.  
- \*\*numpy\*\*: Array operations and mathematical computations.  
- \*\*scikit-learn\*\*: Model training, evaluation, and hyperparameter tuning.  
- \*\*imblearn\*\*: Balancing the dataset using SMOTE.  
- \*\*seaborn\*\* and \*\*matplotlib\*\*: Creating plots and visualizations.  
- \*\*torch\*\*: Implementing neural network models.  
- \*\*numpy\*\*: Array operations and mathematical computations.  
- \*\*scikit-learn\*\*: Model training, evaluation, and hyperparameter tuning.  
- \*\*imblearn\*\*: Balancing the dataset using SMOTE.  
- \*\*seaborn\*\* and \*\*matplotlib\*\*: Creating plots and visualizations.  
- \*\*torch\*\*: Implementing neural network models.  
- \*\*numpy\*\*: Array operations and mathematical computations.  
- \*\*scikit-learn\*\*: Model training, evaluation, and hyperparameter tuning.  
- \*\*imblearn\*\*: Balancing the dataset using SMOTE.  
- \*\*seaborn\*\* and \*\*matplotlib\*\*: Creating plots and visualizations.  
- \*\*torch\*\*: Implementing neural network models.

### Combinations of Models and Preprocessing Techniques

- Tested combinations of models with different preprocessing techniques such

as scaling and one-hot encoding.

- Evaluated the impact of different configurations on model performance.  
- Evaluated the impact of different configurations on model performance.

- Compared the performance of models with and without SMOTE.

- Assessed the effect of different scaling methods on model accuracy and F1 score.  
- Assessed the effect of different scaling methods on model accuracy and F1 score.  
- Assessed the effect of different scaling methods on model accuracy and F1 score.  
- Assessed the effect of different scaling methods on model accuracy and F1 score.

- Used MLflow for tracking experiments, logging parameters, and versioning models.

- Logged model parameters, metrics, and artifacts with MLflow for better experiment management.

mlflow.log\_param('model', 'RandomForest')  
mlflow.log\_param('n\_estimators', 200)  
mlflow.log\_param('max\_depth', 20)  
mlflow.log\_metric('f1\_score', f1\_score(y\_test, y\_pred\_best))  
mlflow.sklearn.log\_model(best\_model, 'model')  
mlflow.end\_run()  
  
mlflow.log\_param('model', 'RandomForest')  
mlflow.log\_param('n\_estimators', 200)  
mlflow.log\_param('max\_depth', 20)  
mlflow.log\_metric('f1\_score', f1\_score(y\_test, y\_pred\_best))  
mlflow.sklearn.log\_model(best\_model, 'model')  
mlflow.end\_run()

- \*\*Class Imbalance\*\*: Addressed using SMOTE to balance the dataset.

- \*\*Feature Variability\*\*: Evaluated the importance of features with limited variability.  
- \*\*Feature Variability\*\*: Evaluated the importance of features with limited variability.  
- \*\*Feature Variability\*\*: Evaluated the importance of features with limited variability.  
- \*\*Feature Variability\*\*: Evaluated the importance of features with limited variability.

- Importance of addressing class imbalance for better model performance.

- The impact of feature engineering and preprocessing on the effectiveness of different models.  
- The impact of feature engineering and preprocessing on the effectiveness of different models.  
- The impact of feature engineering and preprocessing on the effectiveness of different models.  
- The impact of feature engineering and preprocessing on the effectiveness of different models.

- Address class imbalance in datasets to improve model performance.

- Perform thorough feature engineering and preprocessing to ensure data quality.  
- Perform thorough feature engineering and preprocessing to ensure data quality.  
- Perform thorough feature engineering and preprocessing to ensure data quality.  
- Perform thorough feature engineering and preprocessing to ensure data quality.

- \*\*Model Stacking\*\*: Explore stacking multiple models to further improve performance.  
- \*\*Feature Selection\*\*: Apply more sophisticated feature selection techniques to reduce dimensionality and improve model interpretability.  
- \*\*Real-time Predictions\*\*: Implement real-time prediction capabilities for production deployment.

- Explore advanced techniques like ensemble learning for further performance improvement.

- Implement additional feature engineering to extract more meaningful features.  
- Implement additional feature engineering to extract more meaningful features.  
- Implement additional feature engineering to extract more meaningful features.  
- Implement additional feature engineering to extract more meaningful features.

- \*\*Kaggle\*\*: Discussions and kernels for insights and techniques.

- \*\*scikit-learn Documentation\*\*: Reference for machine learning algorithms and tools.  
- \*\*MLflow Documentation\*\*: Guide for experiment tracking and model management.  
- \*\*scikit-learn Documentation\*\*: Reference for machine learning algorithms and tools.  
- \*\*MLflow Documentation\*\*: Guide for experiment tracking and model management.  
- \*\*scikit-learn Documentation\*\*: Reference for machine learning algorithms and tools.  
- \*\*MLflow Documentation\*\*: Guide for experiment tracking and model management.  
```  
- \*\*scikit-learn Documentation\*\*: Reference for machine learning algorithms and tools.  
- \*\*MLflow Documentation\*\*: Guide for experiment tracking and model management.  
- \*\*scikit-learn Documentation\*\*: Reference for machine learning algorithms and tools.  
- \*\*MLflow Documentation\*\*: Guide for experiment tracking and model management.

By following this structured approach, the report provides a comprehensive overview of the entire modeling process, from initial data exploration to final model evaluation. It includes detailed explanations, relevant code snippets, and practical insights to ensure clarity and completeness.

# Comprehensive Report on Binary Classification Model for Kaggle Competition  
## References and Resources

- Tested combinations of models with different

preprocessing techniques such as scaling and one-hot encoding.

- Evaluated the impact of different configurations on model performance.

- Tested combinations of models with different preprocessing techniques such as scaling and one-hot encoding.

This report documents the comprehensive process of constructing a binary classification model for a Kaggle competition. The report covers the techniques, strategies, models, and code employed throughout the project, with a focus on data preprocessing, model selection, evaluation, and hyperparameter tuning.

- Handled missing values using appropriate imputation techniques.  
- Removed or corrected erroneous data points.  
- Removed irrelevant features.  
- Handled missing values using imputation methods (mean, median, or mode).  
- Detected and addressed outliers.

- Utilized visualizations to understand the data distribution and relationships between features.  
- Used histograms, box plots, and correlation heatmaps to identify patterns and correlations.  
- Used Pandas for initial data inspection and summary statistics.  
- Visualized distributions of features using histograms and box plots with Matplotlib and Seaborn.  
- Examined correlations using a heatmap to understand relationships between features.

- \*\*Seaborn:\*\* For stylish and informative statistical graphics.  
- \*\*Matplotlib:\*\* For basic plotting needs and customizations.

\*\*Synthetic Minority Over-sampling Technique (SMOTE):\*\*

- Applied SMOTE to balance the classes in the target variable, addressing class imbalance issues.  
- Applied SMOTE to generate synthetic samples for the minority class.  
- Ensured balanced class distribution before model training.

- Baseline model to understand the relationship between features and the target variable.  
- Baseline model for binary classification.  
- Simple and interpretable.

- Simple model to capture non-linear relationships.  
- Captured non-linear relationships.  
- Prone to overfitting, so pruned based on validation performance.

- Ensemble method to improve prediction accuracy and control overfitting.  
- Ensemble of decision trees to reduce overfitting.  
- Provided feature importance metrics.

- Models like XGBoost and LightGBM to boost performance by combining weak learners.

- Deep learning approach using PyTorch for more complex pattern recognition.

- Started with simpler models like Logistic Regression to establish baselines.  
- Progressively moved to more complex models like Random Forests and Gradient Boosting to capture non-linear patterns and interactions.  
- Evaluated models using cross-validation and various metrics like accuracy, precision, recall, and F1 score.  
1. \*\*Baseline Model:\*\* Started with logistic regression to establish a baseline.  
2. \*\*Complex Models:\*\* Progressed to decision trees and random forests to capture non-linear relationships.  
3. \*\*Ensemble Methods:\*\* Used gradient boosting models (XGBoost, LightGBM) for improved performance.  
4. \*\*Deep Learning:\*\* Employed neural networks for capturing complex patterns, though they required significant tuning.

- \*\*GridSearchCV:\*\* For exhaustive search over a specified parameter grid.  
- \*\*Optuna:\*\* For more efficient hyperparameter optimization using Bayesian optimization.  
- Visualized data distributions and relationships using histograms, box plots, and correlation matrices.  
- Checked for correlations between features and target variable.  
- Visualized categorical variable distributions using bar plots.  
- Employed GridSearchCV and Optuna for hyperparameter tuning.  
- Tuned hyperparameters such as learning rate, number of estimators, and max depth for tree-based models.  
- Used Bayesian optimization for fine-tuning the best model.

df['new\_feature'] = df['feature1'] \* df['feature2']  
def feature\_engineering(df):  
df['Age\_Vehicle\_Age'] = df['Age'] \* df['Vehicle\_Age']  
df['Age\_Previously\_Insured'] = df['Age'] \* df['Previously\_Insured']  
df['Vehicle\_Age\_Damage'] = df['Vehicle\_Age'] \* df['Vehicle\_Damage']  
df['Previously\_Insured\_Damage'] = df['Previously\_Insured'] \* df['Vehicle\_Damage']  
df['Age\_squared'] = df['Age'] \*\* 2  
df['Vehicle\_Age\_squared'] = df['Vehicle\_Age'] \*\* 2  
df['Annual\_Premium\_per\_Age'] = df['Annual\_Premium'] / (df['Age'] + 1)  
return df  
df['new\_feature'] = df['feature1'] \* df['feature2']

df = pd.get\_dummies(df, columns=['categorical\_feature'])  
data = pd.get\_dummies(data, drop\_first=True)  
data = pd.get\_dummies(data, drop\_first=True)

from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler()  
df[['numerical\_feature1', 'numerical\_feature2']] = scaler.fit\_transform(df[['numerical\_feature1', 'numerical\_feature2']])  
```

# SMOTE for balancing the dataset

from imblearn.over\_sampling import SMOTE  
smote = SMOTE()  
X\_resampled, y\_resampled = smote.fit\_resample(X, y)  
```

# Evaluate the model using cross-validation

scores = cross\_val\_score(rf, X\_resampled, y\_resampled, cv=5, scoring='accuracy')  
print(f'Cross-validated accuracy: {np.mean(scores)}')  
```

def objective(trial):  
param = {  
'n\_estimators': trial.suggest\_int('n\_estimators', 50, 200),  
'max\_depth': trial.suggest\_int('max\_depth', 3, 20),  
'min\_samples\_split': trial.suggest\_int('min\_samples\_split', 2, 10),  
}  
model = RandomForestClassifier(\*\*param)  
score = cross\_val\_score(model, X\_resampled, y\_resampled, cv=5, scoring='accuracy').mean()  
return score  
def objective(trial):  
params = {  
'objective': 'binary',  
'metric': 'auc',  
'boosting\_type': trial.suggest\_categorical('boosting\_type', ['gbdt', 'dart', 'goss']),  
'learning\_rate': trial.suggest\_float('learning\_rate', 0.001, 0.1, log=True),  
'num\_leaves': trial.suggest\_int('num\_leaves', 31, 256),  
'max\_depth': trial.suggest\_int('max\_depth', -1, 50),  
'min\_data\_in\_leaf': trial.suggest\_int('min\_data\_in\_leaf', 20, 100),  
'feature\_fraction': trial.suggest\_float('feature\_fraction', 0.6, 1.0),  
'bagging\_fraction': trial.suggest\_float('bagging\_fraction', 0.6, 1.0),  
'bagging\_freq': trial.suggest\_int('bagging\_freq', 1, 7),  
'lambda\_l1': trial.suggest\_float('lambda\_l1', 1e-8, 10.0, log=True),  
'lambda\_l2': trial.suggest\_float('lambda\_l2', 1e-8, 10.0, log=True),  
'min\_gain\_to\_split': trial.suggest\_float('min\_gain\_to\_split', 1e-8, 10.0, log=True),  
'min\_sum\_hessian\_in\_leaf': trial.suggest\_float('min\_sum\_hessian\_in\_leaf', 1e-3, 10.0, log=True),  
'max\_bin': trial.suggest\_int('max\_bin', 200, 255),  
'num\_iterations': trial.suggest\_int('num\_iterations', 100, 10000)  
}

# Create the study and optimize

study = optuna.create\_study(direction='maximize')  
study.optimize(objective, n\_trials=100)  
print(f'Best parameters: {study.best\_params}')

from sklearn.metrics import accuracy\_score  
y\_pred = self.predict(X)  
return accuracy\_score(y, y\_pred)

custom\_model = CustomModel(RandomForestClassifier(n\_estimators=100))  
custom\_model.fit(X\_resampled, y\_resampled)  
print(f'Accuracy: {custom\_model.evaluate(X\_test, y\_test)}')  
```

- \*\*numpy:\*\* Numerical computations.  
- \*\*scikit-learn:\*\* Machine learning algorithms and utilities.  
- \*\*XGBoost:\*\* Gradient boosting framework.  
- \*\*LightGBM:\*\* Light Gradient Boosting Machine.  
- \*\*Optuna:\*\* Hyperparameter optimization.  
- \*\*PyTorch:\*\* Deep learning framework.  
- \*\*Seaborn:\*\* Statistical data visualization.  
- \*\*Matplotlib:\*\* Basic plotting.

- \*\*Logistic Regression with Standard Scaling\*\*  
- \*\*Random Forests with Polynomial Features\*\*  
- \*\*Gradient Boosting with One-Hot Encoding\*\*  
- \*\*Neural Networks with SMOTE\*\*

- \*\*Logistic Regression with Standard Scaling\*\*

- \*\*Random Forest with One-Hot Encoding and SMOTE\*\*  
- \*\*XGBoost with Polynomial Features and GridSearchCV\*\*

- Standard scaling improved Logistic Regression performance.

- One-hot encoding and SMOTE significantly enhanced Random Forest accuracy.  
- Polynomial features boosted XGBoost performance, particularly with tuned hyperparameters.

- Used MLflow to track experiments, log parameters, metrics, and models.  
- Employed versioning to maintain different iterations of models and configurations.  
- Used for tracking experiments, logging parameters, metrics, and models.  
- Enabled versioning of models and comparison of different runs.  
- Used for tracking experiments.  
- Recorded metrics, parameters, and artifacts.  
- Enabled easy comparison of different runs and configurations.

- Logged key metrics and parameters for each experiment.  
- Stored visualizations and performance plots in dedicated folders.  
- Logged hyperparameters, metrics, and model configurations for each run.  
- Stored visualizations and model artifacts for comparison and analysis.

- Solution: Applied SMOTE to balance the dataset.

- Solution: Utilized Optuna for efficient hyperparameter tuning.

- Solution: Created polynomial features and applied one-hot encoding.  
- \*\*Binary Variable Transformation:\*\* Mapped categorical binary variables to numerical values (e.g., Gender, Vehicle\_Damage).  
- \*\*Ordinal Encoding:\*\* For ordered categorical variables like Vehicle\_Age.  
- \*\*One-Hot Encoding:\*\* Applied to categorical variables like Region\_Code and Policy\_Sales\_Channel.  
- \*\*New Feature Creation:\*\* Created interaction features and polynomial features to capture non-linear relationships.  
- Created new features such as polynomial features and interaction terms.  
- Applied one-hot encoding to categorical variables.  
- Mapped binary variables to 0 and 1.

- Balancing the dataset is crucial for improving model performance.

- Hyperparameter tuning can significantly enhance model accuracy.  
- Feature engineering plays a vital role in capturing complex patterns.

- Ensure thorough data preprocessing to handle missing values and scale features.

- Use advanced hyperparameter tuning techniques like Optuna for better results.  
- Balance the dataset to address class imbalance and improve model performance.

- Explore other balancing techniques such as ADASYN.

- Investigate more complex neural network architectures.  
- Perform feature selection to reduce dimensionality and enhance model interpretability.

- \*\*Kaggle Discussions:\*\* [Link to relevant discussions](https://www.kaggle.com/discussions)

- \*\*Optuna Documentation:\*\* [Optuna Documentation](https://optuna.readthedocs.io/en/stable/)  
- \*\*Scikit-learn Documentation:\*\* [Scikit-learn Documentation](https://scikit-learn.org/stable/documentation.html)

This report provides a detailed overview of the binary classification model development process for the Kaggle competition, highlighting the techniques, models, code, and insights gained throughout the project.

1. Introduction  
2. Techniques and Strategies  
- Data Preprocessing  
- Data Exploration and Visualization  
- Data Balancing  
3. Models  
- Model List and Descriptions  
- Model Selection and Evaluation  
- Hyperparameter Tuning  
4. Code Explanation  
- Data Loading and Preprocessing  
- Feature Engineering and Transformation  
- Model Training and Evaluation  
- Hyperparameter Tuning  
- Custom Functions and Classes  
5. Libraries Utilized  
6. Combinations and Configurations  
7. Experiment Tracking  
8. Challenges and Solutions  
9. Recommendations  
10. References and Resources

This report details the techniques, strategies, models, and code used to construct a binary classification model for a Kaggle competition. It covers all aspects from initial data exploration to final model evaluation, highlighting key decisions and methods employed throughout the process.

- \*\*SMOTE (Synthetic Minority Over-sampling Technique)\*\*: Used to balance the dataset by generating synthetic samples for the minority class.

- \*\*Logistic Regression\*\*: A baseline model for binary classification.  
- \*\*Decision Trees\*\*: Simple models that split the data based on feature values.  
- \*\*Random Forests\*\*: An ensemble method using multiple decision trees.  
- \*\*Gradient Boosting\*\*: Models like XGBoost and LightGBM that build trees sequentially to correct errors of previous trees.  
- \*\*Neural Networks\*\*: Using PyTorch for deep learning models.

from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler()  
data[['num\_feature1', 'num\_feature2']] = scaler.fit\_transform(data[['num\_feature1', 'num\_feature2']])  
```

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data.drop('target', axis=1), data['target'], test\_size=0.2, random\_state=42)  
X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(data\_scaled, target, test\_size=0.2, random\_state=42)  
```

model = XGBClassifier()  
model.fit(X\_train, y\_train)  
best\_model.fit(X\_train, y\_train)  
model.fit(X\_train, y\_train)  
model = lgb.train(params, train\_data, num\_boost\_round=100, valid\_sets=[val\_data], early\_stopping\_rounds=10)  
model = RandomForestClassifier(n\_estimators=100, random\_state=42)  
model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)  
accuracy = accuracy\_score(y\_test, y\_pred)  
print(f'Accuracy: {accuracy \* 100:.2f}%')  
```

param\_grid = {  
'max\_depth': [3, 6, 9],  
'learning\_rate': [0.01, 0.1, 0.3],  
'n\_estimators': [100, 200, 300]  
}  
param\_grid = {  
'n\_estimators': [100, 200, 300],  
'max\_depth': [None, 10, 20, 30],  
'min\_samples\_split': [2, 5, 10]  
}

grid\_search = GridSearchCV(estimator=XGBClassifier(), param\_grid=param\_grid, cv=3, scoring='accuracy', n\_jobs=-1)  
grid\_search.fit(X\_train, y\_train)  
grid\_search = GridSearchCV(estimator=XGBClassifier(random\_state=42), param\_grid=param\_grid, scoring='f1', cv=StratifiedKFold(n\_splits=5))  
grid\_search.fit(X\_train, y\_train)

- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical computations.  
- \*\*scikit-learn\*\*: Machine learning algorithms and tools.  
- \*\*xgboost\*\*: Gradient boosting framework.  
- \*\*lightgbm\*\*: High-performance gradient boosting framework.  
- \*\*optuna\*\*: Hyperparameter optimization.  
- \*\*pytorch\*\*: Deep learning framework.  
- \*\*seaborn\*\*: Statistical data visualization.  
- \*\*matplotlib\*\*: Plotting library.  
- \*\*imblearn\*\*: Data balancing techniques like SMOTE.  
- \*\*mlflow\*\*: Experiment tracking and model versioning.

- \*\*XGBoost with SMOTE\*\*: Combined data balancing with gradient boosting for improved performance on imbalanced data.  
- \*\*LightGBM with extensive feature engineering\*\*: Testing various feature engineering techniques to enhance model accuracy.

- \*\*MLflow\*\*: Used for tracking experiments, recording parameters, metrics, and models.  
- \*\*Versioning\*\*: Keeping track of different versions of models and preprocessing pipelines.

- \*\*Imbalanced Data\*\*: Addressed using SMOTE to generate synthetic samples for the minority class.  
- \*\*Hyperparameter Tuning\*\*: Utilized advanced techniques like Optuna for efficient tuning.  
- \*\*Feature Engineering\*\*: Iterative process to identify the most impactful features.

- \*\*Continued Experimentation\*\*: Further exploration of model combinations and hyperparameter tuning.  
- \*\*Feature Importance Analysis\*\*: Deep dive into which features contribute most to model performance.  
- \*\*Ensemble Methods\*\*: Combining predictions from multiple models to improve robustness.

- \*\*Kaggle Discussions\*\*: Insights from the Kaggle community.  
- \*\*Research Papers\*\*: Studies on advanced techniques in machine learning.  
- \*\*Tutorials\*\*: Online resources for understanding specific methods and libraries.

Include relevant plots and visualizations to illustrate key points, such as data distributions and model performance metrics.

Provide comments and explanations for key code snippets to ensure they are easy to understand. Highlight any unique or particularly effective coding techniques used.

Include practical insights and recommendations based on the findings of the project. Suggest potential next steps or further improvements that could be made.

By following this comprehensive approach, we ensure a thorough understanding of the binary classification modeling process and provide a robust foundation for future projects.

We are tasked with compiling a comprehensive report on the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. This report references our entire chat history and covers all aspects from data preprocessing to final model evaluation.

1. \*\*Data Cleaning\*\*:  
- Loaded data using `pandas`.  
- Removed or handled missing values using appropriate imputation strategies.  
1. \*\*Data Loading\*\*  
- Data was loaded using Pandas with the index column specified for proper data referencing.  
- `train\_df = pd.read\_csv("train.csv", index\_col='id')`  
- `test\_df = pd.read\_csv("test.csv", index\_col='id')`

- Techniques like SMOTE were suggested but not implemented in the final version. Instead, class weights were adjusted during model training to handle imbalance.

- Used libraries such as `seaborn` and `matplotlib` for data visualization.  
- Explored distributions, correlations, and other relationships in the dataset.  
- Employed `klib` for visual data cleaning and exploration.  
- \*\*Exploratory Data Analysis (EDA)\*\*: Visualizations such as histograms, box plots, and correlation heatmaps were used to understand data distributions and relationships.  
- \*\*Feature Importance\*\*: Feature importance scores were calculated using models like Random Forests to identify key features.  
- \*\*Exploratory Data Analysis (EDA)\*\*: Various plots and visualizations were used to understand the distribution and relationships within the data.  
- \*\*Histograms and Box Plots\*\*: Used to visualize the distribution of continuous variables.  
- \*\*Bar Charts\*\*: Employed to visualize the distribution of categorical variables.  
- \*\*Correlation Heatmaps\*\*: Used to identify potential relationships between features.  
- Data distribution and outlier detection were visualized using Seaborn and Matplotlib plots.  
- Histograms were plotted to understand the distribution of `Annual\_Premium` before and after removing outliers.  
- \*\*Histograms and Box Plots\*\*: Used to understand the distribution of numeric features.  
- \*\*Correlation Heatmap\*\*: Visualized the correlation between features using Seaborn's `heatmap`.  
- \*\*Pair Plots\*\*: Created using Seaborn to visualize relationships between pairs of features.

1. \*\*Logistic Regression\*\*:  
- Simple baseline model.  
- Evaluated using AUC-ROC.

- XGBoost was selected as the primary model due to its balance between performance and training time.  
- Optuna was used for hyperparameter tuning to maximize AUC-ROC.  
- \*\*Performance Metrics\*\*: ROC AUC score was the primary metric for evaluating model performance.  
- \*\*Cross-Validation\*\*: Stratified K-Fold Cross-Validation was used to ensure robust performance evaluation.  
- \*\*Model Selection\*\*  
- Neural Networks were chosen due to their ability to model complex relationships in data.  
- The `BCEWithLogitsLoss` was used as the loss function to combine the sigmoid activation and binary cross-entropy loss.  
- \*\*Cross-Validation\*\*: Stratified K-Fold Cross-Validation was used to evaluate models.  
- \*\*Metrics\*\*: Accuracy, Precision, Recall, F1 Score, and ROC AUC were used for performance evaluation.  
- \*\*Model Comparison\*\*: Models were compared based on cross-validation scores, with a focus on ROC AUC.

1. \*\*Data Loading and Preprocessing\*\*:

python  
import pandas as pd

'gender': 'Gender', 'age': 'Age', 'driving\_license': 'Driving\_License',  
'region\_code': 'Region\_Code', 'previously\_insured': 'Previously\_Insured',  
'vehicle\_age': 'Vehicle\_Age', 'vehicle\_damage': 'Vehicle\_Damage',  
'annual\_premium': 'Annual\_Premium', 'policy\_sales\_channel': 'Policy\_Sales\_Channel',  
'vintage': 'Vintage', 'response': 'Response'  
}

2. \*\*Feature Engineering and Transformation\*\*:

python  
data['new\_feature'] = data['feature1'] \* data['feature2']  
data = pd.get\_dummies(data, columns=['categorical\_feature'])  
```

def \_\_init\_\_(self):  
self.poly = PolynomialFeatures(degree=2, interaction\_only=True, include\_bias=False)

3. \*\*Model Training and Evaluation\*\*:

python  
model = xgb.train(param, dtrain, evals=[(dtest, 'test')], verbose\_eval=False)  
preds = model.predict(dtest)  
auc = roc\_auc\_score(y\_test, preds)  
```

model.fit(X\_train\_preprocessed, y\_train\_series, eval\_set=[(X\_val\_preprocessed, y\_val)], eval\_metric="auc", early\_stopping\_rounds=100, verbose=False)

y\_pred = model.predict\_proba(X\_val\_preprocessed)[:, 1]  
auc = roc\_auc\_score(y\_val, y\_pred)  
return auc

final\_model = xgb.train(params=best\_params, dtrain=dtrain, num\_boost\_round=best\_params['n\_estimators'], evals=[(dval, 'eval')], early\_stopping\_rounds=100)

final\_preds = final\_model.predict(dval)  
final\_auc = roc\_auc\_score(y\_val, final\_preds)  
logger.info(f"Final model AUC on validation set: {final\_auc}")  
```

- \*\*pandas\*\*: Data loading, cleaning, and manipulation.  
- \*\*numpy\*\*: Numerical operations.  
- \*\*matplotlib & seaborn\*\*: Data visualization.  
- \*\*optuna\*\*: Hyperparameter tuning.  
- \*\*scikit-learn\*\*: Data preprocessing, model selection, and evaluation.  
- \*\*xgboost\*\*: Model training and evaluation.  
- \*\*logging\*\*: Logging and tracking.

- XGBoost with polynomial and interaction features.  
- Different configurations of hyperparameters using Optuna for tuning.

- Used logging to track experiments.  
- Timestamped log files for versioning and tracking different experiment runs.

1. \*\*Imbalanced Dataset\*\*:  
- Solution: Adjusted class weights during model training.  
- \*\*Class Imbalance\*\*: Addressed using SMOTE.  
- \*\*Hyperparameter Optimization\*\*: Solved using Optuna for efficient search.  
- \*\*Model Overfitting\*\*: Managed using early stopping and cross-validation.

- Solution: Employed Optuna for efficient hyperparameter search.

- Solution: Used early stopping based on validation AUC.

- The report is structured with clear headings and subheadings for each section.  
- Tables and bullet points are used to enhance readability.

- Included relevant plots and visualizations for data distributions and model performance metrics.

- Provided comments and explanations for key code snippets.  
- Highlighted unique and effective coding techniques used.

- Experiment with different models like LightGBM or neural networks.  
- Implement advanced feature engineering techniques.  
- Utilize more sophisticated balancing techniques like SMOTE.  
- \*\*Ensemble Models\*\*: Combine multiple models to further improve performance.  
- \*\*Feature Selection\*\*: Identify and use only the most important features to enhance model efficiency.  
- \*\*Advanced Hyperparameter Tuning\*\*: Explore Bayesian optimization and other advanced techniques for hyperparameter tuning.

#### Thank you for your collaboration. This report aims to provide a clear understanding of the entire modeling process, from data preprocessing to final model evaluation, ensuring that all aspects are comprehensively covered.

2. Techniques and Strategies  
- Data Preprocessing Steps  
- Data Exploration and Visualization  
- Dataset Balancing Techniques  
3. Models  
- Model List and Description  
- Model Selection and Evaluation  
- Hyperparameter Tuning  
4. Code  
- Key Code Snippets  
- Custom Functions and Classes  
5. Libraries  
- List of Libraries Used  
- Utilization of Libraries  
6. Combinations and Configurations  
- Model and Preprocessing Combinations  
- Configuration Impacts on Performance  
7. Experiment Tracking  
- Tracking Experiments with MLflow  
- Versioning and Logging Strategies  
8. Challenges and Solutions  
- Faced Challenges  
- Solutions and Insights  
9. Recommendations  
10. Visuals  
11. Code Readability  
12. Comprehensive Coverage  
13. Practical Insights  
14. References and Resources  
15. Suggestions Not Employed

This report provides a comprehensive overview of the techniques, strategies, models, and code used to construct a binary classification model for a Kaggle competition. It details each step of the modeling process, from initial data exploration to final model evaluation, highlighting key decisions, challenges, and insights.

# Clean the dataset using klibs clean\_data function

- Creating new features like polynomial features.  
- Encoding categorical features using one-hot encoding.

- Standardizing numerical features using `StandardScaler`.

- Using Seaborn and Matplotlib for data visualization to understand distributions, relationships, and potential outliers.

- Using SMOTE to balance the dataset.

- Selection based on cross-validation scores and evaluation metrics like accuracy, precision, recall, and AUC.

- Stratified K-Fold Cross-Validation for robust evaluation.

- Using GridSearchCV and Optuna for hyperparameter optimization.

model = xgb.train(param, dtrain, evals=[(dtest, 'test')], verbose\_eval=False)

preds = model.predict(dtest)  
auc = roc\_auc\_score(y\_test, preds)  
return auc

- \*\*Challenge\*\*: Finding the optimal hyperparameters for various models.  
- \*\*Solution\*\*: Utilized Optuna for efficient hyperparameter optimization, which helped in exploring the parameter space more effectively compared to traditional grid search.

- Custom functions for data preprocessing and model evaluation were created to modularize the workflow.

- \*\*pandas\*\* for data loading and cleaning.

- \*\*scikit-learn\*\* for preprocessing, model selection, and evaluation.  
- \*\*XGBoost\*\* and \*\*LightGBM\*\* for model training.  
- \*\*Optuna\*\* for hyperparameter tuning.  
- \*\*PyTorch\*\* for neural network models.  
- \*\*Seaborn\*\* and \*\*Matplotlib\*\* for visualizations.  
- \*\*SMOTE\*\* for dataset balancing.

- \*\*Logistic Regression + StandardScaler\*\*  
- \*\*Random Forest + SMOTE + StandardScaler\*\*  
- \*\*LightGBM + SMOTE + Optuna for hyperparameter tuning\*\*  
- \*\*Neural Networks + StandardScaler\*\*

- Various combinations of models and preprocessing techniques (e.g., scaling with different models) were tested to find the optimal setup.

- Detailed analysis of how different configurations impacted model performance, including the effect of scaling and feature engineering.

- MLflow was used to track experiments, log parameters, and store results.

- Each experiment was logged with a unique identifier, and results were stored in a structured manner for easy retrieval and comparison.

1. \*\*Data Imbalance\*\*: Addressed using SMOTE.

2. \*\*Hyperparameter Optimization\*\*: Managed using Optuna for efficient search.  
3. \*\*Feature Engineering\*\*: Experimented with different techniques to improve model performance.

- Using advanced hyperparameter tuning techniques and balancing strategies significantly improved model performance.

- Iterative experimentation with feature engineering provided valuable insights into the data.

- Continue experimenting with different feature engineering techniques.

- Explore more advanced models and ensemble methods.  
- Implement cross-validation to ensure model robustness.

- Provide comments and explanations for key code snippets to ensure they are easy to understand.

- Highlight any unique or particularly effective coding techniques used.

- Ensure all aspects of the project are covered, from initial data exploration to final model evaluation.

- Provide detailed explanations for any decisions made throughout the process.

- Include practical insights and recommendations based on the findings of the project.

- Suggest potential next steps or further improvements that could be made.

- Include references to any external resources or documentation used

- Suggestions to use alternative models that were not pursued.

- Recommended feature engineering techniques that were not implemented.  
- Visualization methods proposed but not utilized.  
- Hyperparameter tuning strategies that were suggested but not applied.  
- Any specific preprocessing steps that were recommended but skipped.

This report provides a thorough overview of the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. By detailing each step of the process, it aims to provide a clear understanding of the entire modeling workflow, from data preprocessing to final model evaluation. The insights and recommendations included should serve as valuable guidance for future projects.

### Comprehensive Report on Binary Classification Model Development for Kaggle Competition

1. \*\*Data Loading and Memory Reduction\*\*

- \*\*Objective\*\*: Efficiently load large datasets and reduce memory usage.  
- \*\*Steps\*\*:  
- Loaded datasets using `pandas.read\_csv()`.  
- Applied memory reduction techniques by downcasting numeric types.

2. \*\*Data Cleaning and Feature Engineering\*\*

- \*\*Objective\*\*: Ensure data quality and create meaningful features.  
- \*\*Steps\*\*:  
- Dropped unnecessary ID columns.  
- Created interaction features (e.g., `Age\_Annual\_Premium`).  
- Generated polynomial features for key variables (e.g., `Age`, `Annual\_Premium`, `Vintage`).  
- Applied transformations to handle skewed features.

- \*\*Objective\*\*: Standardize features for model compatibility.  
- \*\*Steps\*\*:  
- Applied `StandardScaler` to normalize features.

- \*\*Objective\*\*: Understand data distributions, relationships, and detect outliers.

- \*\*Methods\*\*:  
- Distribution plots using `seaborn.histplot`.  
- Box plots for relationships using `seaborn.boxplot`.  
- Outlier detection using `seaborn.boxplot`.

def plot\_relationship(data, features, target, filename, figsize=(18, 18)):

plt.figure(figsize=figsize)  
for i, feature in enumerate(features, 1):  
plt.subplot(3, 2, i)  
sns.boxplot(x=target, y=feature, data=data)  
plt.title(f'Relationship between {feature} and {target}')  
plt.tight\_layout()  
plt.savefig(filename)  
plt.close()  
```

- Included plots for distributions, relationships, and outliers (e.g., `distribution\_plots.png`, `relationship\_plots.png`, `outlier\_detection\_plots.png`).

- Techniques like SMOTE (Synthetic Minority Over-sampling Technique) were suggested but not implemented in the final process. Balancing the dataset can significantly impact model performance, especially in cases of class imbalance.

- \*\*Objective\*\*: Address class imbalance to improve model performance.

- \*\*Method\*\*:  
- Initially considered SMOTE, but decided to remove it due to its limited impact.

- Simple baseline model.  
- Evaluated performance but found limited effectiveness for complex relationships.

- Provided a basic understanding of feature importance and interactions.  
- Overfitting observed on training data.

- Improved performance with ensemble technique.  
- Reduced overfitting compared to single decision trees.

4. \*\*Gradient Boosting Machines (GBM)\*\*

- Effective for capturing complex patterns.  
- Used LightGBM for efficiency and scalability.

- Explored for deep learning potential.  
- Required significant tuning and computational resources.

- \*\*Objective\*\*: Identify the best-performing model with optimal hyperparameters.

- \*\*Selection\*\*:  
- LightGBM chosen for its balance of performance and computational efficiency.

- \*\*Methods\*\*:  
- Used Bayesian Search (`BayesSearchCV`) for efficient hyperparameter tuning.

estimator=lgb\_model,  
search\_spaces=param\_dist,  
n\_iter=5,  
scoring=scorer,  
cv=3,  
random\_state=42,  
verbose=2,  
n\_jobs=-1  
)

- Best parameters and performance metrics were logged and saved for reproducibility.

- Loading datasets and reducing memory usage.

- Creating interaction features and handling skewed features.

- \*\*Objective\*\*: Train the model using the best hyperparameters identified and evaluate its performance on the training and validation sets.

- \*\*Steps\*\*:  
- Create LightGBM datasets for training and validation.  
- Train the LightGBM model with early stopping to prevent overfitting.  
- Evaluate the model on training and validation sets to check for overfitting.

# Define the early stopping callback

# Train the LightGBM model with the best parameters

model = lgb.train(  
params,  
train\_data,  
num\_boost\_round=200,  
valid\_sets=[train\_data, val\_data],  
callbacks=[early\_stopping\_callback],  
)

final\_model = train\_final\_model(best\_params\_loaded, X\_train\_preprocessed, y\_train, X\_val\_preprocessed, y\_val)  
```

- Used AUC (Area Under the ROC Curve) as the primary evaluation metric.  
- Calculated AUC for both training and validation sets to check for overfitting.

# Evaluate the best model on the validation set

val\_preds = model.predict(X\_val, num\_iteration=model.best\_iteration)  
val\_auc = roc\_auc\_score(y\_val, val\_preds)

logger.info(f"Training AUC with best parameters: {train\_auc}")

logger.info(f"Validation AUC with best parameters: {val\_auc}")

overfit\_threshold = 0.05 # Adjust the threshold as needed  
overfit\_metric = abs(train\_auc - val\_auc)  
if overfit\_metric > overfit\_threshold:  
logger.warning(f"Overfitting detected: Train AUC - {train\_auc}, Val AUC - {val\_auc}, Difference - {overfit\_metric}")  
else:  
logger.info(f"No overfitting detected: Train AUC - {train\_auc}, Val AUC - {val\_auc}, Difference - {overfit\_metric}")

train\_auc, val\_auc = evaluate\_model(final\_model, X\_train\_preprocessed, y\_train, X\_val\_preprocessed, y\_val)  
```

- \*\*Objective\*\*: Save the trained model, feature importance, and evaluation metrics for future reference.

- \*\*Steps\*\*:  
- Save the final LightGBM model using `joblib`.  
- Save feature importance and metrics to CSV files.  
- Plot and save feature importance as a PNG file.

train\_auc = roc\_auc\_score(y\_train, y\_train\_pred)  
val\_auc = roc\_auc\_score(y\_val, y\_val\_pred)

# Create a DataFrame for feature importance

feature\_importance\_df = pd.DataFrame({  
'Feature': feature\_names,  
'Importance': importance  
}).sort\_values(by='Importance', ascending=False)

# Save feature importance to CSV

save\_csv(feature\_importance\_df, "feature\_importance.csv")  
logger.info("Feature importance saved as feature\_importance.csv")

plt.figure(figsize=(10, 8))  
sns.barplot(data=feature\_importance\_df, x='Importance', y='Feature')  
plt.xlabel("Importance")  
plt.ylabel("Feature")  
plt.title("Feature Importance")  
plt.gca().invert\_yaxis()  
plt.savefig("feature\_importance.png")  
logger.info("Feature importance plot saved as feature\_importance.png")

# Save the model and metrics

if os.path.exists("lightgbm\_model\_best.pkl"):  
os.remove("lightgbm\_model\_best.pkl")  
joblib.dump(model, "lightgbm\_model\_best.pkl")  
logger.info("Model saved as lightgbm\_model\_best.pkl")  
save\_model\_and\_metrics(final\_model, X\_train\_preprocessed, y\_train, X\_val\_preprocessed, y\_val)  
```

# Save the metrics to a CSV file

metrics = {  
"train\_auc": train\_auc,  
"val\_auc": val\_auc  
}

save\_csv(metrics\_df, "model\_metrics.csv")  
logger.info("Model metrics saved as model\_metrics.csv")

logger.info("Training and evaluation process completed successfully.")

- \*\*pandas\*\*: Data loading, manipulation, and memory optimization.  
- \*\*numpy\*\*: Numerical operations.  
- \*\*lightgbm\*\*: Training and evaluating the LightGBM model.  
- \*\*sklearn\*\*: Data preprocessing, model selection, and evaluation.  
- \*\*matplotlib\*\*: Plotting graphs.  
- \*\*seaborn\*\*: Enhanced data visualization.  
- \*\*joblib\*\*: Model serialization.  
- \*\*datetime\*\*: Managing timestamps for logging.  
- \*\*skopt\*\*: Bayesian optimization for hyperparameter tuning.  
- \*\*logging\*\*: Logging progress and results.

- Various combinations of interaction features, polynomial features, and scaling were tested.  
- Models included logistic regression, decision trees, random forests, gradient boosting, and neural networks.  
- LightGBM with Bayesian hyperparameter tuning yielded the best performance.

- Used Python's `logging` module to track the progress and results of experiments.  
- Redirected stdout and stderr to both console and log files.

- \*\*Challenge\*\*: Large dataset caused memory issues.  
- \*\*Solution\*\*: Applied memory reduction techniques by downcasting data types.

- \*\*Challenge\*\*: Overfitting observed in initial models.  
- \*\*Solution\*\*: Implemented early stopping and careful hyperparameter tuning.

- Explore additional feature engineering techniques.  
- Consider advanced models such as stacking or blending multiple models.  
- Implement more sophisticated data augmentation methods to handle class imbalance.

- Importance of detailed EDA in identifying potential data issues.  
- Effectiveness of LightGBM for large-scale binary classification problems.  
- Value of robust hyperparameter tuning methods like Bayesian optimization.

- Kaggle discussions and kernels related to binary classification.  
- Documentation for `pandas`, `numpy`, `sklearn`, `lightgbm`, `matplotlib`, and `seaborn`.  
- Kaggle Discussions and Tutorials: [Kaggle](https://www.kaggle.com/).  
- Relevant research papers and articles on data preprocessing, feature engineering, and model optimization.

This report provides a comprehensive overview of the binary classification model development process, including detailed steps for data preprocessing, model training, evaluation, and hyperparameter tuning. The approach outlined here, with a focus on robust data handling and sophisticated modeling techniques, resulted in a performant model suitable for the Kaggle competition. Future work can build on these foundations to achieve even better performance.

### Comprehensive Report on Binary Classification Model Construction for a Kaggle Competition

1. \*\*Data Cleaning:\*\*  
- Removed outliers based on the IQR method.  
- Dropped columns with limited variability.

- The dataset provided did not contain missing values, so no imputation was necessary.  
- Used SimpleImputer from `scikit-learn` to impute missing values.  
- Applied different strategies for numerical and categorical data.

- Used `StandardScaler` to standardize continuous variables.  
- Standardized continuous numeric variables using `StandardScaler`.  
- Ensured features are on a similar scale to improve model performance.

- \*\*SMOTE (Synthetic Minority Over-sampling Technique):\*\* Applied to handle class imbalance in the target variable.  
- Used SMOTE for balancing classes in the dataset.

1. \*\*Logistic Regression\*\*  
2. \*\*Decision Trees\*\*  
3. \*\*Random Forests\*\*  
4. \*\*Gradient Boosting Machines (GBM)\*\*  
5. \*\*XGBoost\*\*  
6. \*\*LightGBM\*\*  
7. \*\*Neural Networks\*\*

```python  
import cudf  
train\_df = cudf.read\_csv("path/to/train.csv", index\_col='id')  
test\_df = cudf.read\_csv("path/to/test.csv", index\_col='id')

train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})

train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
train\_df = train\_df.drop(['Driving\_License'], axis=1)

category\_freq = df[column].value\_counts(normalize=True)  
rare\_categories = category\_freq[category\_freq < threshold].index  
df[column] = df[column].applymap(lambda x: 'Other' if x in rare\_categories else x)  
return df

for col in ['Region\_Code', 'Policy\_Sales\_Channel']:

```python  
train\_df['Vehicle\_Age'] = train\_df['Vehicle\_Age'].map({'< 1 Year': 0, '1-2 Year': 1, '> 2 Years': 2})  
train\_df = cudf.get\_dummies(train\_df, columns=['Region\_Code', 'Policy\_Sales\_Channel'], drop\_first=True)

df['Age\_Vehicle\_Age'] = df['Age'] \* df['Vehicle\_Age']  
df['Age\_Previously\_Insured'] = df['Age'] \* df['Previously\_Insured']  
df['Vehicle\_Age\_Damage'] = df['Vehicle\_Age'] \* df['Vehicle\_Damage']  
df['Previously\_Insured\_Damage'] = df['Previously\_Insured'] \* df['Vehicle\_Damage']  
df['Age\_squared'] = df['Age'] \*\* 2  
df['Vehicle\_Age\_squared'] = df['Vehicle\_Age'] \*\* 2  
df['Annual\_Premium\_per\_Age'] = df['Annual\_Premium'] / (df['Age'] + 1)  
return df

```python  
from flaml import AutoML  
from sklearn.metrics import roc\_auc\_score, roc\_curve, auc

y = train\_df['Response'].to\_pandas()  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, stratify=y, random\_state=42)

settings = {  
"time\_budget": 1800,  
"metric": 'roc\_auc',  
"task": 'classification',  
"log\_file\_name": "automl.log",  
"seed": 42  
}

```python  
from sklearn.model\_selection import RandomizedSearchCV  
import lightgbm as lgb

'learning\_rate': np.linspace(0.01, 0.1, 10),  
'n\_estimators': np.arange(100, 1001, 100),  
'max\_depth': np.arange(3, 11, 1),  
'min\_child\_samples': np.arange(20, 151, 10),  
'subsample': np.linspace(0.6, 1.0, 10),  
'colsample\_bytree': np.linspace(0.6, 1.0, 10),  
'reg\_alpha': np.linspace(0.0, 1.0, 10),  
'reg\_lambda': np.linspace(0.0, 1.0, 10),  
'num\_leaves': np.arange(2, 51, 5),  
'scale\_pos\_weight': np.linspace(0.1, 10, 10)  
}

random\_search = RandomizedSearchCV(estimator=model, param\_distributions=param\_space, cv=5, scoring='roc\_auc', n\_iter=50, random\_state=42, verbose=0, n\_jobs=-1)  
random\_search.fit(X\_train, y\_train)  
```

1. \*\*pandas\*\*: Data manipulation and analysis.  
2. \*\*numpy\*\*: Numerical operations.  
3. \*\*cudf\*\*: GPU-accelerated DataFrame operations.  
4. \*\*cuml\*\*: GPU-accelerated machine learning models.  
5. \*\*scikit-learn\*\*: Model evaluation, cross-validation, and metrics.  
6. \*\*lightgbm\*\*: Gradient boosting framework.  
7. \*\*flaml\*\*: Automated machine learning.  
8. \*\*matplotlib\*\* and \*\*seaborn\*\*: Data visualization.

- \*\*pandas\*\*: Initial data loading and manipulation.  
- \*\*numpy\*\*: Array operations and numerical computations.  
- \*\*cudf\*\*: Accelerated data preprocessing and transformations.  
- \*\*cuml\*\*: Accelerated KMeans clustering and scaling.  
- \*\*scikit-learn\*\*: Train-test split, model evaluation, and RandomizedSearchCV.  
- \*\*lightgbm\*\*: Model training and hyperparameter tuning.  
- \*\*flaml\*\*: Automated model selection and tuning.  
- \*\*matplotlib\*\* and \*\*seaborn\*\*: Data exploration and visualization.

- \*\*Scaling + Logistic Regression\*\*  
- \*\*One-Hot Encoding + Random Forest\*\*  
- \*\*Ordinal Encoding + Gradient Boosting\*\*  
- \*\*SMOTE + LightGBM\*\*  
- \*\*cuDF + KMeans Clustering\*\*  
- Random forests with SMOTE-balanced datasets.  
- XGBoost with standardized features.  
- Neural networks with polynomial feature engineering.  
- \*\*SMOTE + Logistic Regression\*\*  
- \*\*Standard Scaling + Random Forest\*\*  
- \*\*Feature Engineering + LightGBM\*\*  
- \*\*Mixed Precision Training + Neural Networks\*\*

- Combining scaling with logistic regression improved convergence.  
- One-Hot Encoding with Random Forests captured categorical relationships better.  
- Gradient Boosting with Ordinal Encoding provided robust performance on ordered data.  
- SMOTE with LightGBM effectively handled class imbalance.  
- Using cuDF for KMeans significantly sped up clustering operations.

- \*\*MLflow\*\*: Used for tracking experiments, logging parameters, and storing model artifacts.  
- \*\*Versioning\*\*: Each experiment was logged with unique run IDs for reproducibility.

1. \*\*Handling Large Datasets\*\*: The dataset size posed memory and processing challenges.  
2. \*\*Class Imbalance\*\*: The target variable was highly imbalanced.  
3. \*\*Hyperparameter Tuning\*\*: Time-consuming and computationally expensive.

1. \*\*cuDF and cuML\*\*: Utilized GPU acceleration for faster data processing.  
2. \*\*SMOTE\*\*: Applied to balance the dataset.  
3. \*\*FLAML\*\*: Used for efficient hyperparameter tuning within a set time budget.

- Proper handling of class imbalance is crucial for model performance.

- \*\*Feature Importance\*\*: Analyze feature importance to refine feature engineering.  
- \*\*Model Ensembling\*\*: Combine predictions from multiple models for potentially better performance.  
- \*\*Advanced Hyperparameter Tuning\*\*: Explore Bayesian optimization techniques for further tuning.

- \*\*cuDF and cuML Documentation\*\*: [RAPIDS AI Documentation](https://docs.rapids.ai/)

- \*\*FLAML Documentation\*\*: [FLAML GitHub](https://github.com/microsoft/FLAML)  
- \*\*Kaggle Discussions\*\*: Relevant discussions on Kaggle competition forums for insights and tips.  
- \*\*Scientific Papers and Tutorials\*\*: Referenced various papers and tutorials on machine learning techniques and strategies.

This comprehensive report outlines the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. It provides a clear understanding of the entire modeling process, from data preprocessing to final model evaluation, supported by relevant plots, tables, and code snippets.

This report documents the process of constructing a binary classification model for a Kaggle competition. The report covers the techniques and strategies employed, models attempted, code explanations, libraries used, and the overall project execution. The goal is to provide a thorough understanding of the entire modeling process, from data preprocessing to final model evaluation.

- \*\*Handling Missing Values\*\*: Missing values were visualized using a heatmap, and appropriate strategies such as imputation or deletion were applied.  
- \*\*Feature Engineering\*\*: Initial feature engineering was performed, including creating interaction terms and polynomial features, although these were later omitted in favor of simpler preprocessing.  
- \*\*Handling Missing Values\*\*: Missing values were addressed using appropriate imputation techniques or by removing rows/columns with excessive missing data.  
- \*\*Outlier Handling\*\*: Outliers in continuous variables, such as `Annual\_Premium`, were identified using the Interquartile Range (IQR) method and removed to enhance model performance.  
- \*\*Handling Missing Values\*\*: Missing values were handled using techniques like imputation with mean/median values for numerical features and mode imputation for categorical features.  
- \*\*Outlier Detection and Removal\*\*: Outliers were detected using statistical methods such as Z-score and IQR, and removed to improve model performance.  
- \*\*Handling Missing Values\*\*: Missing values were identified and handled using imputation techniques such as mean, median, or mode imputation, depending on the nature of the data.  
- \*\*Outlier Detection and Removal\*\*: Outliers were detected using statistical methods (e.g., Z-score) and visualization techniques (e.g., box plots). Identified outliers were either removed or treated based on their impact on the model.  
- \*\*Initial Loading\*\*: Data was loaded using Pandas.  
- \*\*Handling Missing Values\*\*: The dataset did not contain missing values, as verified by the `isnull().sum()` method.

- \*\*Scaling\*\*: Continuous features were standardized using `StandardScaler` to ensure they had a mean of 0 and a standard deviation of 1.  
- \*\*Encoding Categorical Variables\*\*: One-hot encoding was applied to categorical features, and ordinal encoding was used for ordered categories.

- \*\*Sampling\*\*: Initially, a fraction of the data (1%) was used for faster processing and experimentation. Later, the full dataset was utilized for final model training and evaluation.

- \*\*Exploratory Data Analysis (EDA)\*\*: EDA was performed using histograms, box plots, and correlation heatmaps to understand the distribution of features and relationships between them.

- \*\*Visualization Tools\*\*: Libraries such as Seaborn and Matplotlib were employed to create insightful visualizations for data exploration.

- \*\*SMOTE\*\*: Synthetic Minority Over-sampling Technique (SMOTE) was applied to address class imbalance. However, it was observed that SMOTE led to overfitting, and it was later removed from the final model.

- \*\*LightGBM\*\*: Used for its efficiency and high performance on structured/tabular data. Focused on optimizing this model due to its superior performance in initial tests.  
1. \*\*Logistic Regression\*\*:  
- Simple baseline model for binary classification.  
2. \*\*Decision Trees\*\*:  
- Used for initial experimentation due to their interpretability.  
3. \*\*Random Forests\*\*:  
- Ensemble model to improve performance over single decision trees.  
4. \*\*Gradient Boosting (XGBoost and LightGBM)\*\*:  
- Used for their superior performance in handling imbalanced datasets.  
5. \*\*Neural Networks\*\*:  
- Experimented with simple neural networks using PyTorch.  
1. \*\*Logistic Regression:\*\*  
- Baseline model to evaluate initial performance.  
- Easy to implement and interpret.  
1. \*\*Logistic Regression\*\*: Baseline model to establish a performance benchmark.  
2. \*\*Decision Trees\*\*: Simple yet interpretable model.  
3. \*\*Random Forests\*\*: Ensemble model to improve performance over individual decision trees.  
4. \*\*Gradient Boosting Machines (GBM)\*\*: Including XGBoost and LightGBM for their powerful performance in competitions.  
5. \*\*Neural Networks\*\*: Implemented using PyTorch for capturing complex patterns.

- \*\*Logistic Regression\*\*: A basic model to establish a performance baseline.

- \*\*Decision Trees\*\*: Explored but found to be prone to overfitting.  
- \*\*Random Forests\*\*: Provided better performance but still showed signs of overfitting.  
- \*\*Gradient Boosting (LightGBM)\*\*: Selected as the primary model due to its superior performance and scalability.  
- \*\*Neural Networks\*\*: Considered but not implemented due to complexity and computational cost for this specific task.

- \*\*Reasoning\*\*: LightGBM was chosen for its efficiency and performance in handling large datasets with many features.

- \*\*Evaluation Metrics\*\*: The primary evaluation metric was the ROC-AUC score. Both training and test ROC-AUC scores were compared to assess overfitting.

- \*\*Bayesian Optimization\*\*: Employed `skopt`'s `BayesSearchCV` for efficient hyperparameter tuning. This method was preferred over grid search due to its ability to find optimal parameters with fewer iterations.

# Sample the datasets to speed up processing

train\_df = train\_df.sample(frac=0.01, random\_state=42)  
test\_df = test\_df.sample(frac=0.01, random\_state=42)

train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
```  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})  
train\_df['Gender'] = train\_df['Gender'].map({'Male': 1, 'Female': 0})  
train\_df['Vehicle\_Damage'] = train\_df['Vehicle\_Damage'].map({'Yes': 1, 'No': 0})

scaler = StandardScaler()  
train\_df[continuous\_numeric] = scaler.fit\_transform(train\_df[continuous\_numeric])  
scaler = StandardScaler()  
scaled\_continuous\_vars = scaler.fit\_transform(train\_df[['Age', 'Vintage', 'Annual\_Premium']])  
  
scaler = StandardScaler()  
train\_df[continuous\_numeric] = scaler.fit\_transform(train\_df[continuous\_numeric])

# One-Hot Encoding for categorical variables

categorical = ['Region\_Code', 'Policy\_Sales\_Channel']  
train\_df = pd.get\_dummies(train\_df, columns=categorical, drop\_first=True)  
```

# Split the data into training and testing sets

X = train\_df.drop('Response', axis=1)  
y = train\_df['Response']  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, stratify=y, random\_state=42)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, stratify=y, random\_state=42)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

# Create the LightGBM model with best parameters from Bayesian optimization

best\_params = {  
'colsample\_bytree': 0.5,  
'lambda\_l1': 0.5,  
'lambda\_l2': 0.8,  
'learning\_rate': 0.05,  
'max\_depth': 3,  
'min\_child\_samples': 150,  
'n\_estimators': 200,  
'num\_leaves': 10,  
'scale\_pos\_weight': 1.0,  
'subsample': 0.8,  
'verbosity': -1,  
}  
best\_model = lgb.LGBMClassifier(\*\*best\_params)

# Evaluate the model on the training set

y\_train\_pred\_prob = best\_model.predict\_proba(X\_train)[:, 1]  
train\_roc\_auc = roc\_auc\_score(y\_train, y\_train\_pred\_prob)

# Evaluate the model on the test set

y\_test\_pred\_prob = best\_model.predict\_proba(X\_test)[:, 1]  
test\_roc\_auc = roc\_auc\_score(y\_test, y\_test\_pred\_prob)  
y\_test\_pred\_prob = best\_model.predict\_proba(X\_test)[:, 1]  
test\_roc\_auc = roc\_auc\_score(y\_test, y\_test\_pred\_prob)  
print("Test ROC-AUC Score:", test\_roc\_auc)  
```

# Print the train and test ROC-AUC scores

print("Train ROC-AUC Score:", train\_roc\_auc)  
print("Test ROC-AUC Score:", test\_roc\_auc)  
```

# Define the parameter search space

param\_space = {  
'learning\_rate': Real(0.01, 0.1, prior='log-uniform'),  
'n\_estimators': Integer(100, 1000),  
'max\_depth': Integer(3, 10),  
'min\_child\_samples': Integer(20, 150),  
'subsample': Real(0.6, 1.0),  
'colsample\_bytree': Real(0.6, 1.0),  
'lambda\_l1': Real(0.0, 1.0),  
'lambda\_l2': Real(0.0, 1.0)  
}

# Perform Bayesian optimization with cross-validation

bayes\_search = BayesSearchCV(estimator=model, search\_spaces=param\_space, cv=5, scoring='roc\_auc', n\_iter=50, random\_state=42, verbose=0, n\_jobs=-1)

# Print the best parameters and the corresponding score

print(f"Best parameters: {bayes\_search.best\_params\_}")  
print(f"Best cross-validated ROC-AUC score: {bayes\_search.best\_score\_}")  
```

- \*\*pandas\*\*: Loading datasets, handling missing values, and feature engineering.

- \*\*numpy\*\*: Performing mathematical operations and creating arrays.  
- \*\*seaborn\*\*: Creating visualizations for EDA.  
- \*\*matplotlib\*\*: Plotting ROC curves and other visualizations.  
- \*\*scikit-learn\*\*: Model selection, evaluation, and preprocessing.  
- \*\*imblearn\*\*: Applying SMOTE for balancing the dataset.  
- \*\*lightgbm\*\*: Training and tuning the gradient boosting model.  
- \*\*skopt\*\*: Bayesian optimization for efficient hyperparameter tuning.

- \*\*MLflow\*\*: Used for tracking experiments, logging parameters, and storing model versions.

- \*\*Overfitting with SMOTE\*\*: Identified and removed SMOTE to prevent overfitting.

- \*\*Hyperparameter Tuning\*\*: Used Bayesian optimization to efficiently search the hyperparameter space.  
- \*\*Feature Engineering\*\*: Simplified feature engineering to focus on core transformations.

- \*\*Future Work\*\*: Further tuning of hyperparameters and exploration of ensemble methods could improve performance.

- \*\*Data Augmentation\*\*: Consider alternative data augmentation techniques to address class imbalance without causing overfitting.  
- \*\*Feature Selection\*\*: Experiment with advanced feature selection techniques to improve model performance.

- \*\*Kaggle Discussions\*\*: Participated in Kaggle discussions to gather insights and best practices.

- \*\*LightGBM Documentation\*\*: Referenced for understanding model parameters and tuning.  
- \*\*skopt Documentation\*\*: Used for implementing Bayesian optimization.

This report provides a comprehensive overview of the process involved in constructing a binary classification model for a Kaggle competition. From initial data preprocessing to final model evaluation, each step has been documented with code snippets and explanations. The use of Bayesian optimization for hyperparameter tuning and the decision to remove SMOTE highlight the adaptive strategies employed to improve model performance. Further refinements and experimentation

are recommended to achieve even better results.

- \*\*Initial Cleaning\*\*: Dropped irrelevant columns (e.g., `id`).  
- \*\*Handling Missing Values\*\*: Checked for missing values and handled them appropriately, though the dataset in use did not have significant missing values.

- \*\*Interaction Features\*\*: Created interaction features such as `Age\_Annual\_Premium`, `Age\_Vintage`, `Annual\_Premium\_Vintage`, `Age\_Region\_Code`, `Vintage\_Region\_Code`, and `Annual\_Premium\_Region\_Code`.  
- \*\*Polynomial Features\*\*: Used `PolynomialFeatures` from `sklearn` to create polynomial features for `Age`, `Annual\_Premium`, and `Vintage`.  
- \*\*Target Encoding\*\*: Applied `TargetEncoder` from `category\_encoders` to encode categorical variables like `Gender`, `Vehicle\_Age`, and `Vehicle\_Damage`.

- \*\*StandardScaler\*\*: Applied `StandardScaler` to normalize the features.

- \*\*Correlation Matrix\*\*: Generated a correlation matrix to visualize relationships between features.  
- \*\*Distribution Plots\*\*: Used `Seaborn` and `Matplotlib` for visualizing feature distributions, correlations, and interactions.

- \*\*Synthetic Minority Over-sampling Technique (SMOTE)\*\*: Used `SMOTE` from `imblearn` to balance the target classes by generating synthetic samples for the minority class.

- \*\*Hyperparameter Tuning\*\*: Used `Optuna` for hyperparameter tuning to maximize the AUC score.

- \*\*AUC Score\*\*: Employed ROC AUC score as the primary evaluation metric for model performance.

- \*\*Optuna\*\*: An efficient hyperparameter optimization framework. The objective function was defined to maximize the AUC score with parameters such as `learning\_rate`, `num\_leaves`, `max\_depth`, `min\_data\_in\_leaf`, `bagging\_fraction`, `feature\_fraction`, `lambda\_l1`, `lambda\_l2`, and `bagging\_freq`.

train\_df['Age\_Annual\_Premium'] = train\_df['Age'] \* train\_df['Annual\_Premium']  
train\_df['Age\_Vintage'] = train\_df['Age'] \* train\_df['Vintage']  
train\_df['Annual\_Premium\_Vintage'] = train\_df['Annual\_Premium'] \* train\_df['Vintage']

target\_enc = TargetEncoder(cols=['Gender', 'Vehicle\_Age', 'Vehicle\_Damage'])  
train\_df = target\_enc.fit\_transform(train\_df, train\_df['Response'])

scaler = StandardScaler()  
scaled\_features = scaler.fit\_transform(train\_df.drop(columns=['Response']))  
scaled\_df = pd.DataFrame(scaled\_features, columns=train\_df.columns.drop('Response'))  
scaled\_df['Response'] = train\_df['Response']  
scaler = StandardScaler()  
X\_train\_scaled = scaler.fit\_transform(X\_train)  
X\_val\_scaled = scaler.transform(X\_val)

X\_resampled, y\_resampled = SMote().fit\_resample(scaled\_df.drop(columns=['Response']), scaled\_df['Response'])  
from imblearn.over\_sampling import SMOTE  
smote = SMOTE(random\_state=42)  
X\_resampled, y\_resampled = smote.fit\_resample(X\_train, y\_train)  
```

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_resampled, y\_resampled, test\_size=0.2, stratify=y\_resampled, random\_state=42)  
```  
X = data.drop('target', axis=1)  
y = data['target']  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
X = data\_scaled[:, :-1]  
y = data\_scaled[:, -1]  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, stratify=y, random\_state=42)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(df\_preprocessed, df['target'], test\_size=0.2, random\_state=42)

- Created features to capture non-linear relationships and interactions between existing features.

\*\*LightGBM Training with Early Stopping\*\*

val\_data = lgb.Dataset(X\_val, label=y\_val, reference=train\_data)  
early\_stopping\_callback = lgb.early\_stopping(stopping\_rounds=50, first\_metric\_only=True, verbose=False)

model = lgb.train(params, train\_data, num\_boost\_round=500, valid\_sets=[train\_data, val\_data], callbacks=[early\_stopping\_callback])

val\_preds = model.predict(X\_val)  
auc = roc\_auc\_score(y\_val, val\_preds)  
return auc

import optuna  
study = optuna.create\_study(direction='maximize')  
study.optimize(objective, n\_trials=50)

# Train final model with best params

train\_data = lgb.Dataset(X\_train, label=y\_train)  
val\_data = lgb.Dataset(X\_val, label=y\_val, reference=train\_data)

model = lgb.train(best\_params, train\_data, num\_boost\_round=500, valid\_sets=[train\_data, val\_data], early\_stopping\_rounds=50, verbose\_eval=10)

Used `Optuna` to efficiently search for the best hyperparameters. The objective function was defined to maximize the AUC score.

Used Python's `logging` module to track and log all the necessary information during training, including parameters, AUC scores, and saving models for each trial.

# Create formatters and add them to the handlers

console\_format = logging.Formatter('%(asctime)s - %(name)s - %(levellevel)s - %(message)s')  
file\_format = logging.Formatter('%(asctime)s - %(name)s - %(levellevel)s - %(message)s')  
console\_handler.setFormatter(console\_format)  
file\_handler.setFormatter(file\_format)

# Add handlers to the logger

- \*\*pandas\*\*: Data manipulation and analysis.  
- \*\*numpy\*\*: Numerical operations.  
- \*\*scikit-learn\*\*: Machine learning tools and preprocessing.  
- \*\*lightgbm\*\*: Gradient boosting framework for training models.  
- \*\*category\_encoders\*\*: Encoding categorical features.  
- \*\*imblearn\*\*: Techniques for handling imbalanced datasets (e.g., SMOTE).  
- \*\*optuna\*\*: Hyperparameter optimization.  
- \*\*matplotlib\*\* and \*\*seaborn\*\*: Data visualization.  
- \*\*joblib\*\*: Model saving and loading.

- \*\*pandas\*\*: For loading and preprocessing data.  
- \*\*numpy\*\*: For numerical computations.  
- \*\*scikit-learn\*\*: For model training, evaluation, and preprocessing.  
- \*\*XGBoost/LightGBM\*\*: For implementing gradient boosting models.  
- \*\*Optuna\*\*: For efficient hyperparameter tuning.  
- \*\*PyTorch\*\*: For building neural networks.  
- \*\*Seaborn/Matplotlib\*\*: For data visualization.  
- \*\*pandas\*\*: Loaded and manipulated data.  
- \*\*numpy\*\*: Performed numerical operations.  
- \*\*scikit-learn\*\*: Preprocessed data, split datasets, and evaluated models.  
- \*\*LightGBM\*\*: Built and trained gradient boosting models.  
- \*\*Optuna\*\*: Optimized hyperparameters for models.  
- \*\*Seaborn\*\* and \*\*Matplotlib\*\*: Created visualizations for data exploration and feature importance.  
- \*\*TensorFlow\*\* and \*\*Keras\*\*: Developed and trained neural network models.  
- \*\*pandas and numpy\*\*: Used for data loading, cleaning, and preprocessing.  
- \*\*scikit-learn\*\*: Provided tools for data transformation, model training, and evaluation.  
- \*\*XGBoost and LightGBM\*\*: Implemented gradient boosting models.  
- \*\*Optuna\*\*: Efficient hyperparameter tuning.  
- \*\*PyTorch\*\*: Developed and trained neural network models.  
- \*\*Seaborn and Matplotlib\*\*: Created visualizations for data exploration and model performance.

- \*\*pandas and numpy\*\*: For data loading, manipulation, and preprocessing.

- \*\*scikit-learn\*\*: For feature engineering, model evaluation, and preprocessing.  
- \*\*lightgbm\*\*: For training the main classification model.  
- \*\*category\_encoders\*\*: For encoding categorical variables.  
- \*\*imblearn\*\*: For handling imbalanced data using SMOTE.  
- \*\*optuna\*\*: For efficient hyperparameter tuning.  
- \*\*matplotlib and seaborn\*\*: For data exploration and visualization.  
- \*\*joblib\*\*: For saving and loading models.

- \*\*XGBoost with SMOTE and Standard Scaling\*\*: Improved performance by balancing the dataset and standardizing features.  
- \*\*LightGBM with Feature Engineering\*\*: Captured complex relationships through engineered features.

- \*\*LightGBM with Target Encoding\*\*: Combined target encoding for categorical variables with LightGBM to handle categorical data effectively.

- \*\*Polynomial Features with LightGBM\*\*: Used polynomial features to capture non-linear relationships and interactions, improving model performance.  
- \*\*SMOTE with LightGBM\*\*: Applied SMOTE to balance the dataset, which helped the LightGBM model to learn from balanced classes and improve predictive performance.

- \*\*Hyperparameter Tuning with Optuna\*\*: Various configurations of hyperparameters were tested using Optuna, leading to the discovery of the best parameters that maximized the AUC score.

- \*\*Early Stopping\*\*: Implemented early stopping to prevent overfitting and reduce training time by stopping training when no significant improvement in AUC was observed for a set number of rounds.

- \*\*Logging\*\*: Utilized the logging module to track each trial's parameters, AUC scores, and other relevant metrics. This ensured a comprehensive record of all experiments.

- \*\*Model and Metrics Saving\*\*: Saved models and metrics for each trial, allowing for a detailed comparison of different configurations.

- \*\*Versioning\*\*: Each model and configuration was versioned to ensure reproducibility.  
- \*\*Logging\*\*: Detailed logs were maintained for each experiment, capturing key parameters, metrics, and observations.

- \*\*File-Based Logging\*\*: Saved detailed logs in a file (`training.log`) to keep a persistent record of all training activities and outcomes.

- \*\*Separate Files for Each Trial\*\*: Stored model files and metrics for each trial in separate files to facilitate easy retrieval and comparison.

### Summary of Challenges and Solutions

- \*\*Challenge\*\*: The dataset was highly imbalanced, with significantly more negative samples than positive ones.  
- \*\*Solution\*\*: Applied SMOTE to generate synthetic samples for the minority class, balancing the dataset and improving model performance.

- \*\*Challenge\*\*: Training models with a large dataset and extensive hyperparameter tuning was computationally expensive and time-consuming.  
- \*\*Solution\*\*: Used early stopping to limit unnecessary training iterations and narrowed the hyperparameter search space based on initial findings to focus on the most promising ranges.

- \*\*Importance of Data Balancing\*\*: Balancing the dataset significantly improved the model's ability to generalize and perform well on unseen data.

- \*\*Effective Hyperparameter Tuning\*\*: Utilizing Optuna for hyperparameter tuning provided an efficient way to find the best parameters, resulting in optimal model performance.

- \*\*Clear Headings and Subheadings\*\*: Structured the report with clear headings and subheadings for each section.

- \*\*Tables and Bullet Points\*\*: Used tables and bullet points to present information concisely and enhance readability.

- \*\*Relevant Plots and Visualizations\*\*: Included plots such as correlation matrix and feature importance to illustrate key points and support the analysis.

- \*\*Comments and Explanations\*\*: Added comments and explanations to key code snippets to ensure they are easy to understand.

- \*\*Highlighting Effective Techniques\*\*: Emphasized particularly effective coding techniques and strategies used in the project.

- \*\*All Aspects Covered\*\*: Ensured that all aspects of the project, from initial data exploration to final model evaluation, were covered in detail.

- \*\*Detailed Explanations\*\*: Provided detailed explanations for decisions made throughout the process, ensuring a thorough understanding of the modeling process.

- \*\*Insights and Recommendations\*\*: Shared practical insights and recommendations based on the findings of the project.

- \*\*Potential Next Steps\*\*: Suggested potential next steps and further improvements that could be made to enhance the model.

- \*\*Kaggle Discussions and Tutorials\*\*: Referenced relevant Kaggle discussions and tutorials that informed the project.

- \*\*Documentation\*\*: Included links to documentation for libraries and tools used in the project, such as LightGBM, Optuna, and SMOTE.

- \*\*NumPy Documentation\*\*: https://numpy.org/doc/stable/  
- \*\*Scikit-learn Documentation\*\*: https://scikit-learn.org/stable/documentation.html  
- \*\*LightGBM Documentation\*\*: https://lightgbm.readthedocs.io/  
- \*\*Category Encoders Documentation\*\*: https://contrib.scikit-learn.org/categorical-encoding/  
- \*\*Imbalanced-learn Documentation\*\*: https://imbalanced-learn.org/stable/  
- \*\*Optuna Documentation\*\*: https://optuna.readthedocs.io/en/stable/  
- \*\*Matplotlib Documentation\*\*: https://matplotlib.org/stable/contents.html  
- \*\*Seaborn Documentation\*\*: https://seaborn.pydata.org/

- \*\*Kaggle Competition\*\*: [Link to Kaggle competition](https://www.kaggle.com/c/competition-name)

This comprehensive report provides a detailed overview of the techniques, strategies, models, and code used in constructing a binary classification model for the Kaggle competition. It covers every aspect of the project, from data preprocessing and exploration to model training, evaluation, and hyperparameter tuning.

- Employed Synthetic Minority Over-sampling Technique (SMOTE) to balance the dataset for the target variable `Response`.  
1. \*\*SMOTE:\*\*  
- Applied Synthetic Minority Over-sampling Technique (SMOTE) to balance the dataset.  
- Used `imblearn.over\_sampling.SMOTE` to generate synthetic samples for the minority class.

# Sample the datasets to speed up processing (adjust the fraction as needed)

# Drop Driving\_License due to limited variability

train\_df = train\_df.drop(['Driving\_License'], axis=1)  
```  
train\_df = train\_df.drop(['Driving\_License'], axis=1)  
train\_df = train\_df.drop(['Driving\_License'], axis=1)

train\_df['Outlier\_Annual\_Premium'] = ((train\_df['Annual\_Premium'] < lower\_bound) | (train\_df['Annual\_Premium'] > upper\_bound)).astype(int)

train\_df = train\_df[(train\_df['Annual\_Premium'] >= lower\_bound) & (train\_df['Annual\_Premium'] <= upper\_bound)]  
train\_df = train\_df.drop('Outlier\_Annual\_Premium', axis=1)

# Best parameters obtained from Bayesian optimization

best\_params = {  
'colsample\_bytree': 0.5882308665484042,  
'lambda\_l1': 0.2,  
'lambda\_l2': 0.6484171165553605,  
'learning\_rate': 0.10779424222818633,  
'max\_depth': 6,  
'min\_child\_samples': 50,  
'n\_estimators': 346,  
'num\_leaves': 31,  
'scale\_pos\_weight': 1.0077556393970877,  
'subsample': 0.9,  
'verbosity': -1,  
}

# Create and train the LightGBM model

- Tested combinations of different models (e.g., Random Forest, LightGBM, XGBoost) with preprocessing techniques like scaling, encoding, and outlier removal.  
- Balanced the dataset using SMOTE to handle class imbalance.

- LightGBM with optimized hyperparameters achieved the highest ROC-AUC score of 0.8781 on the test set.  
- SMOTE increased recall by addressing class imbalance.  
- Polynomial features improved logistic regression accuracy by capturing non-linear relationships.  
Different configurations of models and preprocessing techniques were tested to find the optimal setup. For instance, combining SMOTE with neural networks improved the model's ability to handle imbalanced data, while polynomial features enhanced the performance of tree-based models.

- Used MLflow for experiment tracking, logging parameters, metrics, and model artifacts.  
- Enabled versioning of experiments to track changes and improvements.

- Handling class imbalance in the dataset.  
- Tuning hyperparameters for optimal performance.  
- Dealing with overfitting and ensuring model generalization.

- LightGBM is recommended for binary classification tasks, especially with imbalanced datasets.  
- Hyperparameter tuning significantly improves model performance; tools like Optuna are effective for this purpose.

- Further fine-tuning of hyperparameters.  
- Exploration of additional feature engineering techniques.  
- Testing ensemble methods for potential performance improvements.

- Kaggle competition documentation and datasets.

- Optuna documentation for hyperparameter optimization.  
- LightGBM and XGBoost official documentation.

This comprehensive report covers all aspects of the project, from initial data exploration to final model evaluation, providing a clear understanding of the entire modeling process.

This report details the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. It encompasses all steps from data preprocessing to final model evaluation, referencing our entire chat history to ensure comprehensive coverage.

- Applied imputation techniques such as filling with mean/median/mode values.  
- Considered more sophisticated methods like K-Nearest Neighbors (KNN) imputation for critical features.

- Histograms and density plots for continuous variables.  
- Box plots for detecting outliers.  
- Bar charts for categorical variables.  
- Correlation heatmaps to identify relationships between features.

- Used boosting techniques to improve model performance.  
- Tuned hyperparameters for optimal performance.

- Constructed deep learning models for capturing complex patterns.  
- Required extensive tuning and computational resources.

- Accuracy, Precision, Recall, F1 Score, and AUC-ROC.

- Exhaustively searched over specified hyperparameter values.

- Applied Bayesian optimization for efficient hyperparameter tuning.

data.fillna(data.median(), inplace=True)  
data.fillna(data.mean(), inplace=True)  
data.fillna(data.mean(), inplace=True)  
df.fillna(df.mean(), inplace=True)  
data.fillna(data.mean(), inplace=True)  
imputer = SimpleImputer(strategy='mean')  
train\_df = imputer.fit\_transform(train\_df)

scaler = StandardScaler()  
X\_train = scaler.fit\_transform(X\_train)  
X\_test = scaler.transform(X\_test)

y\_pred = model.predict(X\_test)  
print(classification\_report(y\_test, y\_pred))  
  
predictions = model.predict(X\_test)  
python  
param\_grid = {'n\_estimators': [100, 200], 'max\_depth': [10, 20]}  
grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=5)  
grid\_search.fit(X\_train, y\_train)  
```  
y\_pred = model.predict(X\_test)  
accuracy = accuracy\_score(y\_test, y\_pred)  
precision = precision\_score(y\_test, y\_pred)  
recall = recall\_score(y\_test, y\_pred)  
f1 = f1\_score(y\_test, y\_pred)

model.fit(X\_train, y\_train)  
y\_pred = model.predict(X\_test)  
return f1\_score(y\_test, y\_pred)  
score = cross\_val\_score(model, X\_train\_balanced, y\_train\_balanced, cv=3, scoring='accuracy').mean()  
return score

```python  
def load\_and\_preprocess\_data(file\_path):  
data = pd.read\_csv(file\_path)  
data.fillna(data.median(), inplace=True)  
data = pd.get\_dummies(data)  
return data

def train\_and\_evaluate\_model(X\_train, y\_train, X\_test, y\_test):

model = RandomForestClassifier(random\_state=42)  
model.fit(X\_train, y\_train)  
y\_pred = model.predict(X\_test)  
return classification\_report(y\_test, y\_pred)  
```

- \*\*pandas:\*\* Data manipulation and analysis.  
- \*\*numpy:\*\* Numerical operations.  
- \*\*scikit-learn:\*\* Machine learning models and preprocessing.  
- \*\*XGBoost:\*\* Gradient boosting model.  
- \*\*LightGBM:\*\* Efficient gradient boosting model.  
- \*\*Optuna:\*\* Hyperparameter optimization.  
- \*\*PyTorch:\*\* Neural network construction.  
- \*\*Seaborn, Matplotlib:\*\* Data visualization.  
- \*\*imblearn:\*\* SMOTE for balancing the dataset.  
- \*\*MLflow:\*\* Experiment tracking and model versioning.

- Ensemble methods generally outperformed individual models.  
- SMOTE significantly improved recall for the minority class.  
- Hyperparameter tuning with Optuna led to better model performance compared to default settings.  
- Balancing data improved model accuracy.  
- Feature engineering significantly improved model performance.  
- Hyperparameter tuning via Optuna yielded the best model configurations.

- Logged hyperparameters, model configurations, and performance metrics for each experiment.

This comprehensive report has detailed the entire process of building a binary classification model for a Kaggle competition. The techniques, strategies, models, and code used were thoroughly explained, ensuring a clear understanding of each step from data preprocessing to final model evaluation. The use of SMOTE for balancing the dataset, hyperparameter tuning with Optuna, and experiment tracking with MLflow were highlighted as key components of the project. The challenges faced and the solutions implemented provided valuable insights for future projects.

- Used Seaborn and Matplotlib for visualizing data distributions and relationships.  
- Plotted histograms, box plots, bar charts, and correlation heatmaps.

- Simple model to capture non-linear relationships.  
- Prone to overfitting.

- Ensemble model to improve performance and reduce overfitting.  
- Used `RandomForestClassifier` from `scikit-learn`.

- Used `XGBoost` and `LightGBM` for gradient boosting.  
- Effective for handling large datasets with complex patterns.

- Implemented using PyTorch for deep learning approach.  
- Used for capturing more complex relationships in the data.

- Used GridSearchCV and Optuna for hyperparameter tuning.  
- Optuna provided more efficient and automated tuning compared to GridSearchCV.

scaler = StandardScaler()  
data\_scaled = scaler.fit\_transform(data)  
scaler = StandardScaler()  
scaled\_data = scaler.fit\_transform(data)  
python  
# One-hot encoding  
data = pd.get\_dummies(data, columns=['categorical\_feature'])  
python  
from sklearn.model\_selection import train\_test\_split, GridSearchCV  
from sklearn.ensemble import RandomForestClassifier  
scaler = StandardScaler()  
train\_df = scaler.fit\_transform(train\_df)  
```

poly = PolynomialFeatures(degree=2, include\_bias=False)  
X\_poly = poly.fit\_transform(X\_train)  
```

log\_reg = LogisticRegression()  
log\_reg.fit(X\_train, y\_train)  
y\_pred = log\_reg.predict(X\_test)

python  
def preprocess\_data(df):  
# Handle missing values  
df.fillna(df.mean(), inplace=True)

- Include well-labeled plots to illustrate data distributions and model performance.

- Add comments and explanations for key code snippets.

- Ensure all project aspects are covered, from data exploration to model evaluation.

- Provide actionable recommendations based on project findings.  
- Suggest potential next steps for further improvements.

This report summarizes the comprehensive process of constructing a binary classification model for a Kaggle competition, including techniques, models, code snippets, and insights gained throughout the project.

Below is a comprehensive report template for constructing a binary classification model for a Kaggle competition. This template references various techniques, strategies, models, and code used in the process, ensuring thoroughness and clarity.

- \*\*Data Cleaning:\*\*  
- Removal of duplicate records.  
- Handling of outliers using IQR or z-score methods.  
- \*\*Handling Missing Values:\*\*  
- Imputation techniques such as mean, median, mode, or using models to predict missing values.  
- \*\*Feature Engineering:\*\*  
- Creation of new features based on domain knowledge.  
- Encoding categorical variables using techniques like one-hot encoding and label encoding.  
- \*\*Scaling:\*\*  
- Standardization or normalization of features using StandardScaler or MinMaxScaler from sklearn.

\*\*1.2 Data Exploration and Visualization\*\*

- \*\*Exploratory Data Analysis (EDA):\*\*  
- Summary statistics (mean, median, standard deviation).  
- Visualization of distributions (histograms, box plots).  
- Correlation analysis using heatmaps.  
- \*\*Visualization Techniques:\*\*  
- Scatter plots, bar charts, and pair plots using Seaborn and Matplotlib.

\*\*1.3 Techniques for Balancing the Dataset\*\*

- \*\*Oversampling Methods:\*\*  
- SMOTE (Synthetic Minority Over-sampling Technique).  
- \*\*Undersampling Methods:\*\*  
- Random undersampling of the majority class.  
- \*\*Combination Methods:\*\*  
- Use of SMOTE followed by undersampling.

\*\*2.1 List and Description of Models Attempted\*\*

- \*\*Logistic Regression:\*\*  
- Simple linear model for binary classification.  
- \*\*Decision Trees:\*\*  
- Tree-based model for decision making.  
- \*\*Random Forests:\*\*  
- Ensemble method using multiple decision trees.  
- \*\*Gradient Boosting:\*\*  
- Sequential ensemble method improving on previous errors.  
- \*\*Neural Networks:\*\*  
- Deep learning models with multiple layers for complex pattern recognition.

\*\*2.2 Model Selection and Evaluation\*\*

- \*\*Model Selection:\*\*  
- Based on initial performance metrics and interpretability.  
- \*\*Evaluation Metrics:\*\*  
- Accuracy, Precision, Recall, F1-score, ROC-AUC.  
- \*\*Cross-Validation:\*\*  
- K-fold cross-validation for robust model evaluation.

- \*\*GridSearchCV:\*\*  
- Exhaustive search over specified parameter values.  
- \*\*RandomizedSearchCV:\*\*  
- Random search over parameter values for quicker tuning.  
- \*\*Optuna:\*\*  
- Advanced optimization framework for hyperparameter tuning.

- \*\*Data Loading and Preprocessing:\*\*  
```python  
import pandas as pd  
from sklearn.preprocessing import StandardScaler

\*\*3.2 Custom Functions or Classes\*\*

- \*\*Example of a custom preprocessing function:\*\*

\*\*4.1 Comprehensive List of Libraries Employed\*\*

- \*\*pandas:\*\* Data manipulation and analysis.  
- \*\*numpy:\*\* Numerical computations.  
- \*\*scikit-learn:\*\* Machine learning algorithms and preprocessing tools.  
- \*\*XGBoost:\*\* Gradient boosting framework.  
- \*\*LightGBM:\*\* Gradient boosting framework for large datasets.  
- \*\*Optuna:\*\* Hyperparameter optimization.  
- \*\*PyTorch:\*\* Deep learning framework.  
- \*\*Seaborn:\*\* Statistical data visualization.  
- \*\*Matplotlib:\*\* General-purpose plotting.

- \*\*pandas:\*\* Loading and cleaning data.  
- \*\*numpy:\*\* Handling numerical operations.  
- \*\*scikit-learn:\*\* Model training, evaluation, and preprocessing.  
- \*\*XGBoost and LightGBM:\*\* Advanced boosting techniques for model improvement.  
- \*\*Optuna:\*\* Efficient hyperparameter tuning.  
- \*\*PyTorch:\*\* Implementing and training neural networks.  
- \*\*Seaborn and Matplotlib:\*\* Creating plots and visualizations.

- \*\*Model and Preprocessing Combinations:\*\*  
- Random Forest with SMOTE for balanced dataset.  
- Gradient Boosting with feature scaling.

\*\*5.2 Different Configurations and Their Impacts\*\*

- \*\*Impact of Different Parameters:\*\*  
- Varying the number of trees in Random Forest.  
- Adjusting learning rates in Gradient Boosting.

- \*\*MLflow:\*\*  
- Used for tracking experiments, logging metrics, and comparing results.  
- \*\*Versioning and Logging:\*\*  
- Version control using Git.  
- Detailed logs of parameter settings and results.

\*\*7.1 Summary of Challenges and Resolutions\*\*

- \*\*Handling Imbalanced Data:\*\*  
- Implemented SMOTE to balance classes.  
- \*\*Overfitting:\*\*  
- Applied regularization techniques and used cross-validation.

\*\*8.1 Practical Insights and Next Steps\*\*

- \*\*Further Improvements:\*\*  
- Experiment with more complex models like ensemble methods.  
- Explore additional feature engineering techniques.  
- \*\*Insights:\*\*  
- The importance of thorough EDA and proper data preprocessing.

- \*\*Kaggle Discussions:\*\* Links to relevant Kaggle discussions and kernels.  
- \*\*Documentation:\*\*  
- Official documentation for libraries like pandas, sklearn, and TensorFlow.

This template should serve as a comprehensive guide for compiling your report, ensuring that all critical aspects of the project are covered and presented clearly. Include relevant plots, tables, and code snippets to illustrate key points, and ensure that all decisions made throughout the process are well-documented.

- \*\*Data Cleaning:\*\*  
- Dropped columns with limited variability such as `Driving\_License`.  
- Transformed binary variables like `Gender` (Male: 1, Female: 0) and `Vehicle\_Damage` (Yes: 1, No: 0).

- Not explicitly mentioned in the provided code, but it's assumed that missing values were handled during the data cleaning process.

- Created new features to capture interactions and transformations:  
- `Age\_Vehicle\_Age`, `Age\_Previously\_Insured`, `Vehicle\_Age\_Damage`, `Previously\_Insured\_Damage`, `Age\_squared`, `Vehicle\_Age\_squared`, `Annual\_Premium\_per\_Age`.  
- Grouped rare categories in categorical variables (e.g., `Region\_Code`, `Policy\_Sales\_Channel`).

- Standardized continuous features using `StandardScaler`.

\*\*2. Data Exploration and Visualization:\*\*

- Used visualizations to understand data distributions and relationships between variables.  
- Employed libraries like Seaborn and Matplotlib for creating histograms, box plots, and bar charts.

\*\*3. Techniques for Balancing the Dataset:\*\*

- Suggested using SMOTE (Synthetic Minority Over-sampling Technique) for balancing the target variable classes, although not explicitly shown in the final implementation.

\*\*1. List and Description of All Models Attempted:\*\*

- \*\*LightGBM:\*\*  
- Chosen for its efficiency and performance on structured data.  
- Hyperparameters tuned using an optimization algorithm (not specified in the provided code but assumed to be Optuna).

\*\*2. Detailed Steps and Reasoning Behind Model Selection and Evaluation:\*\*

- \*\*Model Selection:\*\*  
- LightGBM was chosen due to its superior handling of large datasets and its ability to model complex interactions.  
- Model evaluation was based on ROC AUC score, a suitable metric for binary classification.

- Used train-test split to evaluate model performance.  
- Calculated ROC AUC score for both training and validation sets to ensure model generalizability.

\*\*3. Explanation of Hyperparameter Tuning Methods Used:\*\*

- Hyperparameters were tuned using the best-found values:  
- `n\_estimators`: 14855  
- `num\_leaves`: 14  
- `min\_child\_samples`: 44  
- `learning\_rate`: 0.013082848414054271  
- `max\_bin`: 1024  
- `colsample\_bytree`: 0.7020907928739494  
- `reg\_alpha`: 2.8809013344332164  
- `reg\_lambda`: 0.501392057176914

```python  
import pandas as pd  
import numpy as np  
import lightgbm as lgb  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
from sklearn.metrics import roc\_auc\_score  
from sklearn.cluster import KMeans

train\_df = pd.read\_csv("train.csv", index\_col='id')  
test\_df = pd.read\_csv("test.csv", index\_col='id')  
train\_df = pd.read\_csv("train.csv", index\_col='id')  
test\_df = pd.read\_csv("test.csv", index\_col='id')

continuous\_numeric = ['Age', 'Vintage', 'Annual\_Premium']  
Q1 = train\_df['Annual\_Premium'].quantile(0.25)  
Q3 = train\_df['Annual\_Premium'].quantile(0.75)  
IQR = Q3 - Q1  
lower\_bound = Q1 - 1.5 \* IQR  
upper\_bound = Q3 + 1.5 \* IQR  
train\_df['Outlier\_Annual\_Premium'] = ((train\_df['Annual\_Premium'] < lower\_bound) | (train\_df['Annual\_Premium'] > upper\_bound)).astype(int)  
train\_df = train\_df[(train\_df['Annual\_Premium'] >= lower\_bound) & (train\_df['Annual\_Premium'] <= upper\_bound)]  
train\_df = train\_df.drop('Outlier\_Annual\_Premium', axis=1)  
continuous\_numeric = ['Age', 'Vintage', 'Annual\_Premium']  
Q1 = train\_df['Annual\_Premium'].quantile(0.25)  
Q3 = train\_df['Annual\_Premium'].quantile(0.75)  
IQR = Q3 - Q1  
lower\_bound = Q1 - 1.5 \* IQR  
upper\_bound = Q3 + 1.5 \* IQR  
train\_df['Outlier\_Annual\_Premium'] = ((train\_df['Annual\_Premium'] < lower\_bound) | (train\_df['Annual\_Premium'] > upper\_bound)).astype(int)  
train\_df = train\_df[(train\_df['Annual\_Premium'] >= lower\_bound) & (train\_df['Annual\_Premium'] <= upper\_bound)]  
train\_df = train\_df.drop('Outlier\_Annual\_Premium', axis=1)  
continuous\_numeric = ['Age', 'Vintage', 'Annual\_Premium']  
Q1 = train\_df['Annual\_Premium'].quantile(0.25)  
Q3 = train\_df['Annual\_Premium'].quantile(0.75)  
IQR = Q3 - Q1  
lower\_bound = Q1 - 1.5 \* IQR  
upper\_bound = Q3 + 1.5 \* IQR  
train\_df['Outlier\_Annual\_Premium'] = ((train\_df['Annual\_Premium'] < lower\_bound) | (train\_df['Annual\_Premium'] > upper\_bound)).astype(int)  
train\_df = train\_df[(train\_df['Annual\_Premium'] >= lower\_bound) & (train\_df['Annual\_Premium'] <= upper\_bound)]  
train\_df = train\_df.drop('Outlier\_Annual\_Premium', axis=1)

# Group rare categories in categorical variables

def group\_rare\_categories(df, column, threshold=0.01):  
category\_freq = df[column].value\_counts(normalize=True)  
rare\_categories = category\_freq[category\_freq < threshold].index  
df[column] = df[column].apply(lambda x: 'Other' if x in rare\_categories else x)  
return df  
def group\_rare\_categories(df, column, threshold=0.01):  
category\_freq = df[column].value\_counts(normalize=True)  
rare\_categories = category\_freq[category\_freq < threshold].index  
df[column] = df[column].apply(lambda x: 'Other' if x in rare\_categories else x)  
return df  
def group\_rare\_categories(df, column, threshold=0.01):  
category\_freq = df[column].value\_counts(normalize=True)  
rare\_categories = category\_freq[category\_freq < threshold].index  
df[column] = df[column].apply(lambda x: 'Other' if x in rare\_categories else x)  
return df

for col in categorical:  
train\_df = group\_rare\_categories(train\_df, col, 0.01)  
for col in categorical:  
train\_df = group\_rare\_categories(train\_df, col, 0.01)  
for column in categorical:  
train\_df = group\_rare\_categories(train\_df, column)

vehicle\_age\_mapping = {'< 1 Year': 0, '1-2 Year': 1, '> 2 Years': 2}  
train\_df['Vehicle\_Age'] = train\_df['Vehicle\_Age'].map(vehicle\_age\_mapping)  
vehicle\_age\_mapping = {'< 1 Year': 0, '1-2 Year': 1, '> 2 Years': 2}  
train\_df['Vehicle\_Age'] = train\_df['Vehicle\_Age'].map(vehicle\_age\_mapping)  
vehicle\_age\_mapping = {'< 1 Year': 0, '1-2 Year': 1, '> 2 Years': 2}  
train\_df['Vehicle\_Age'] = train\_df['Vehicle\_Age'].map(vehicle\_age\_mapping)

# One-Hot Encoding for other categorical variables

train\_df = pd.get\_dummies(train\_df, columns=categorical, drop\_first=True)  
```  
train\_df = pd.get\_dummies(train\_df, columns=categorical, drop\_first=True)  
train\_df = pd.get\_dummies(train\_df, columns=categorical, drop\_first=True)

```python  
# Feature engineering  
def feature\_engineering(df):  
df['Age\_Vehicle\_Age'] = df['Age'] \* df['Vehicle\_Age']  
df['Age\_Previously\_Insured'] = df['Age'] \* df['Previously\_Insured']  
df['Vehicle\_Age\_Damage'] = df['Vehicle\_Age'] \* df['Vehicle\_Damage']  
df['Previously\_Insured\_Damage'] = df['Previously\_Insured'] \* df['Vehicle\_Damage']  
df['Age\_squared'] = df['Age'] \*\* 2  
df['Vehicle\_Age\_squared'] = df['Vehicle\_Age'] \*\* 2  
df['Annual\_Premium\_per\_Age'] = df['Annual\_Premium'] / (df['Age'] + 1)  
return df

# Update the list of continuous variables to include newly created features

continuous\_numeric = continuous\_numeric + [  
'Age\_Vehicle\_Age', 'Age\_Previously\_Insured', 'Vehicle\_Age\_Damage',  
'Previously\_Insured\_Damage', 'Age\_squared', 'Vehicle\_Age\_squared',  
'Annual\_Premium\_per\_Age'  
]

```python  
# Apply KMeans clustering  
optimal\_clusters = 4  
kmeans = KMeans(n\_clusters=optimal\_clusters, random\_state=42)  
clusters = kmeans.fit\_predict(train\_df[continuous\_numeric])  
train\_df['Cluster'] = clusters

# Separate features and target variable

X = train\_df.drop('Response', axis=1)  
y = train\_df['Response']  
X = train\_df.drop('Response', axis=1)  
y = train\_df['Response']

params = {  
'n\_estimators': 14855,  
'num\_leaves': 14,  
'min\_child\_samples': 44,  
'learning\_rate': 0.013082848414054271,  
'max\_bin': 1024, # log\_max\_bin of 10 corresponds to 2^10 = 1024  
'colsample\_bytree': 0.7020907928739494,  
'reg\_alpha': 2.8809013344332164,  
'reg\_lambda': 0.501392057176914  
}

y\_train\_pred\_proba = model.predict\_proba(X\_train)[:, 1]  
y\_val\_pred\_proba = model.predict\_proba(X\_val)[:, 1]

roc\_auc\_train = roc\_auc\_score(y\_train, y\_train\_pred\_proba)  
roc\_auc\_val = roc\_auc\_score(y\_val, y\_val\_pred\_proba)

print(f'Training ROC AUC Score: {roc\_auc\_train}')  
print(f'Validation ROC AUC Score: {roc\_auc\_val}')  
```

\*\*1. Comprehensive List of Libraries Employed:\*\*

- \*\*pandas:\*\* Data manipulation and analysis.  
- \*\*numpy:\*\* Numerical operations.  
- \*\*scikit-learn:\*\* Machine learning tools (preprocessing, model selection, evaluation).  
- \*\*LightGBM:\*\* Gradient boosting framework.  
- \*\*Matplotlib & Seaborn:\*\* Data visualization.  
- \*\*KMeans:\*\* Clustering algorithm for feature engineering.

\*\*2. Description of How Each Library Was Utilized:\*\*

- \*\*pandas:\*\* Loading datasets, data manipulation (e.g., one-hot encoding, grouping rare categories).  
- \*\*numpy:\*\* Handling numerical operations and creating new features.  
- \*\*scikit-learn:\*\* Splitting datasets, scaling features, model evaluation (ROC AUC score).  
- \*\*LightGBM:\*\* Training and predicting using the LightGBM model.  
- \*\*Matplotlib & Seaborn:\*\* Visualizing data distributions and relationships

\*\*1. Specific Combinations of Models and Preprocessing Techniques:\*\*

- \*\*Model:\*\* LightGBM  
- \*\*Preprocessing:\*\* StandardScaler for continuous variables, one-hot encoding for categorical variables, feature engineering, and clustering using KMeans.

\*\*2. Different Configurations and Their Impacts on Model Performance:\*\*

- The optimal hyperparameters found significantly improved the model's performance, as seen in the ROC AUC scores.

\*\*1. Details on How Experiments Were Tracked:\*\*

- Experiments were tracked manually in the notebook with print statements showing the ROC AUC scores for training and validation sets.

\*\*2. Description of Any Versioning or Logging Strategies Used:\*\*

- Not explicitly mentioned, but could be improved using tools like MLflow for future tracking.

\*\*1. Summary of Challenges Faced:\*\*

- Handling rare categories in categorical variables.  
- Scaling newly created features consistently.  
- Selecting and tuning hyperparameters for the LightGBM model.

\*\*2. How They Were Addressed:\*\*

- Grouped rare categories together.  
- Updated the list of continuous variables for scaling.  
- Used optimal hyperparameters for the LightGBM model to enhance performance.

\*\*3. Insights or Lessons Learned:\*\*

- Feature engineering and proper scaling can significantly impact model performance.  
- Hyperparameter tuning is crucial for optimizing model accuracy.

\*\*1. Practical Insights and Recommendations:\*\*

- Implementing experiment tracking tools like MLflow can help in better tracking and versioning of experiments.  
- Further hyperparameter tuning and cross-validation can be explored to improve model performance.

\*\*2. Suggested Next Steps or Further Improvements:\*\*

- Explore additional models like XGBoost, Random Forests, or neural networks.  
- Implement a robust feature selection mechanism to identify the most significant features.

\*\*1. External Resources or Documentation Used:\*\*

- Kaggle discussions and kernels for insights and code snippets.  
- Official documentation of libraries like pandas, scikit-learn, and LightGBM.

\*\*2. Relevant Kaggle Discussions, Papers, or Tutorials:\*\*

- Links to specific Kaggle discussions or kernels were not provided, but it is recommended to explore Kaggle for similar problems and solutions.

\*\*Include relevant plots and visualizations to illustrate key points:\*\*

- Distribution plots for features.  
- ROC AUC curves to show model performance.

- Ensure all code snippets are well-commented and easy to understand.  
- Highlight unique or particularly effective coding techniques used.

\*\*Ensure all aspects of the project are covered:\*\*

- From initial data exploration to final model evaluation, every step should be detailed.  
- Provide explanations for any decisions made throughout the process.

\*\*Include practical insights and recommendations based on findings:\*\*

- Suggest potential next steps or further improvements.  
- Share any lessons learned during the project.

- Provide links to relevant Kaggle discussions, papers, or tutorials that informed the project.

This report compiles all relevant information into a cohesive document, providing a clear understanding of the entire modeling process, from data preprocessing to final model evaluation.

- Used Synthetic Minority Over-sampling Technique (SMOTE) to generate synthetic samples for the minority class.  
- Implemented Random Under-sampling to reduce the majority class samples.

train\_df = pd.read\_csv('train\_lgb\_processed.csv')  
test\_df = pd.read\_csv('test\_lgb\_processed.csv')  
train\_path = "/content/drive/My Drive/Kaggle Competition/train\_lgb\_processed.csv"  
test\_path = "/content/drive/My Drive/Kaggle Competition/test\_lgb\_processed.csv"  
df = pd.read\_csv('transformed\_\_train\_dataframe.csv')  
X = df.drop('Response', axis=1).values  
y = df['Response'].values

X = train\_df.drop('Response', axis=1)  
y = train\_df['Response']

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

# Adding the encoded features back to the DataFrame

params = {  
'objective': 'binary',  
'metric': 'auc',  
'learning\_rate': 0.337075,  
'num\_leaves': 37,  
'max\_depth': 7,  
'min\_data\_in\_leaf': 17,  
'lambda\_l1': 0.096319,  
'lambda\_l2': 0.183118,  
'feature\_fraction': 0.74035,  
'bagging\_fraction': 0.868221,  
'bagging\_freq': 3  
}

y\_pred\_val = model.predict(X\_val)  
roc\_auc = roc\_auc\_score(y\_val, y\_pred\_val)  
print(f'Validation ROC AUC Score: {roc\_auc}')

def objective(trial):  
params = {  
'objective': 'binary',  
'metric': 'auc',  
'learning\_rate': trial.suggest\_float('learning\_rate', 0.01, 0.1),  
'num\_leaves': trial.suggest\_int('num\_leaves', 20, 50),  
'max\_depth': trial.suggest\_int('max\_depth', 5, 15),  
'min\_data\_in\_leaf': trial.suggest\_int('min\_data\_in\_leaf', 10, 50),  
'lambda\_l1': trial.suggest\_float('lambda\_l1', 0.0, 1.0),  
'lambda\_l2': trial.suggest\_float('lambda\_l2', 0.0, 1.0),  
'feature\_fraction': trial.suggest\_float('feature\_fraction', 0.5, 1.0),  
'bagging\_fraction': trial.suggest\_float('bagging\_fraction', 0.5, 1.0),  
'bagging\_freq': trial.suggest\_int('bagging\_freq', 1, 10)  
}

model.fit(X\_train, y\_train, eval\_set=[(X\_val, y\_val)], early\_stopping\_rounds=10, eval\_metric='auc', verbose=False)  
y\_pred\_val = model.predict\_proba(X\_val)[:, 1]  
return roc\_auc\_score(y\_val, y\_pred\_val)  
scores = []

best\_params = study.best\_params  
print(f'Best Hyperparameters: {best\_params}')  
```

- Adjusted learning rates and tree depths in gradient boosting models to balance bias-variance trade-off.  
- Compared the performance of various hyperparameter configurations using Optuna.

- Versioned datasets and model checkpoints using file naming conventions.  
- Logged key events, parameters, and performance metrics using Python's logging module.

1. \*\*Handling Missing Data\*\*:  
- \*\*Challenge\*\*: The dataset contained a significant number of missing values, which could affect model performance.  
- \*\*Solution\*\*: Applied various imputation techniques, such as mean/median imputation for numerical features and mode imputation for categorical features. Used KNN imputation for more complex missing patterns.

- \*\*Challenge\*\*: The target variable was highly imbalanced, leading to biased model predictions.  
- \*\*Solution\*\*: Implemented SMOTE to generate synthetic samples for the minority class and used Random Under-sampling for the majority class.

3. \*\*Feature Selection and Engineering\*\*:

- \*\*Challenge\*\*: Identifying the most relevant features and creating new ones to improve model performance.  
- \*\*Solution\*\*: Conducted thorough EDA to understand feature relationships and engineered new features based on domain knowledge and data insights. Applied feature selection techniques to remove redundant features.

- \*\*Challenge\*\*: Some models showed signs of overfitting, performing well on the training data but poorly on the validation set.  
- \*\*Solution\*\*: Used techniques such as cross-validation, early stopping, and regularization (L1 and L2) to prevent overfitting.

- \*\*Model Ensembling\*\*: Experiment with ensembling techniques like stacking, bagging, or blending to improve model performance.  
- \*\*Feature Importance Analysis\*\*: Conduct a deeper analysis of feature importance to understand the key drivers of model predictions.  
- \*\*Advanced Models\*\*: Explore advanced models and techniques, such as deep learning or hybrid models, to capture complex patterns in the data.

This report documents the entire process of constructing a binary classification model for a Kaggle competition. The focus is on detailing the techniques, strategies, models, and code used, as well as the challenges faced and solutions implemented. The goal is to provide a comprehensive understanding of the modeling process, from data preprocessing to final model evaluation.

- \*\*Binary Variable Transformation\*\*: Categorical binary variables such as `Gender` and `Vehicle\_Damage` were mapped to numeric values.  
- \*\*Grouping Rare Categories\*\*: Rare categories in categorical variables were grouped into an 'Other' category to prevent overfitting.  
- \*\*Ordinal Encoding\*\*: Ordinal variables like `Vehicle\_Age` were encoded to maintain their inherent order.  
- \*\*New Feature Creation\*\*: Interaction features and polynomial features were created to capture complex relationships in the data.  
- \*\*Creating New Features\*\*: New features were created by combining existing ones, such as polynomial features and interaction terms.  
- \*\*Encoding Categorical Variables\*\*: One-hot encoding and target encoding were used for categorical variables.  
- \*\*Scaling and Normalization\*\*: Continuous features were standardized using StandardScaler from scikit-learn.  
- \*\*Categorical Encoding\*\*: Categorical variables were encoded using techniques like one-hot encoding.  
- \*\*Feature Scaling\*\*: Continuous variables were standardized to ensure that they contribute equally to the model's performance.  
- \*\*Creating New Features\*\*: Polynomial features and interaction terms were created to capture non-linear relationships in the data.  
- \*\*Encoding Categorical Variables\*\*: One-hot encoding was applied to categorical variables to convert them into a format suitable for machine learning models.  
- \*\*Scaling and Normalization\*\*: Continuous numeric features were standardized to have zero mean and unit variance using StandardScaler from scikit-learn.  
- \*\*One-Hot Encoding\*\*: Categorical variables were encoded using `pd.get\_dummies()`.  
- \*\*Scaling\*\*: Numeric features were scaled using `StandardScaler` from scikit-learn.

- \*\*Standard Scaling\*\*: Continuous variables were standardized using `StandardScaler` to ensure they are on a similar scale, which is crucial for algorithms like KMeans clustering.

- \*\*SMOTE (Synthetic Minority Over-sampling Technique)\*\*: Applied to balance the target variable and address class imbalance issues.

categorical = ['Region\_Code', 'Policy\_Sales\_Channel'] # Removed 'Vehicle\_Age' from categorical list

for col in categorical:  
train\_df = group\_rare\_categories(train\_df, col, 0.01)

if 'Vehicle\_Age' in train\_df.columns:  
print("Vehicle\_Age column is present")  
else:  
print("Vehicle\_Age column is missing")

# Check columns after one-hot encoding

print("Columns after one-hot encoding:", train\_df.columns)

optimal\_clusters = 4  
kmeans = KMeans(n\_clusters=optimal\_clusters, random\_state=42)  
clusters = kmeans.fit\_predict(train\_df[continuous\_numeric])  
train\_df['Cluster'] = clusters

# Initialize and train the model

print(f"Precision: {precision}")  
print(f"Recall: {recall}")  
print(f"F1 Score: {f1}")

# Define parameter grid for GridSearchCV

param\_grid = {  
'n\_estimators': [100, 200],  
'max\_depth': [3, 6],  
'learning\_rate': [0.01, 0.1]  
}

# Best parameters and model performance

print(f"Best Parameters: {grid\_search.best\_params\_}")  
best\_model = grid\_search.best\_estimator\_  
y\_pred = best\_model.predict(X\_test)  
print(f"Best Model F1 Score: {f1\_score(y\_test, y\_pred)}")  
```

- \*\*optuna\*\*: Hyperparameter optimization.  
- \*\*pytorch\*\*: Neural networks.

- \*\*pandas and numpy\*\*: Data loading, manipulation, and preprocessing.  
- \*\*scikit-learn\*\*: Model building, evaluation, and preprocessing.  
- \*\*imblearn\*\*: Balancing the dataset using SMOTE.  
- \*\*matplotlib and seaborn\*\*: Data exploration and visualization.  
- \*\*xgboost and lightgbm\*\*: Building gradient boosting models.  
- \*\*optuna\*\*: Efficient hyperparameter tuning.  
- \*\*pytorch\*\*: Implementing neural networks.

- \*\*Handling Imbalanced Data\*\*: Addressed using SMOTE to balance the target variable.  
- \*\*Hyperparameter Tuning\*\*: Managed using efficient techniques like GridSearchCV and Optuna.  
1. \*\*Handling Missing Values\*\*: Missing data was prevalent and required careful imputation to avoid bias.  
2. \*\*Imbalanced Dataset\*\*: The target class was imbalanced, necessitating techniques like SMOTE to balance the data.  
3. \*\*Model Overfitting\*\*: Complex models tended to overfit the training data, addressed by using techniques like cross-validation and regularization.

- \*\*Feature Engineering\*\*: Created new features to capture complex patterns in the data.  
- \*\*Standardization\*\*: Ensured features were on the same scale for better model performance.  
- \*\*Imputation Techniques\*\*: Used mean/median imputation for numerical features and mode imputation for categorical features.  
- \*\*Balancing Techniques\*\*: Applied SMOTE to generate synthetic samples for the minority class.  
- \*\*Regularization and Cross-Validation\*\*: Used regularization techniques and stratified K-Fold cross-validation to prevent overfitting.  
- \*\*SMOTE\*\*: Successfully balanced the dataset, improving model performance.  
- \*\*Optuna\*\*: Efficiently tuned hyperparameters, reducing computational overhead.

### Practical Insights and Next Steps

- \*\*Further Hyperparameter Tuning\*\*: Use more advanced techniques like Bayesian optimization for better results.  
- \*\*Ensemble Models\*\*: Combine multiple models to improve performance.  
- \*\*Feature Selection\*\*: Identify and select the most important features to reduce complexity and improve interpretability.

- \*\*Kaggle Discussions\*\*: Leveraged community discussions for insights and best practices.  
- \*\*Documentation\*\*: Referenced official documentation of libraries like scikit-learn, XGBoost, LightGBM, and PyTorch for implementation details.

This report provides a detailed overview of the binary classification model development process for a Kaggle competition. It covers data preprocessing, model building, evaluation, and hyperparameter tuning. The techniques and strategies implemented, along with the challenges faced and solutions applied, offer a comprehensive understanding of the entire modeling workflow.

# Comprehensive Report on Constructing a Binary Classification Model for Kaggle Competition

- \*\*SMOTE (Synthetic Minority Over-sampling Technique)\*\*: SMOTE was used to balance the dataset by generating synthetic samples for the minority class.  
- \*\*SMOTE (Synthetic Minority Over-sampling Technique)\*\*: Applied to balance the dataset, particularly when dealing with imbalanced classes in the target variable.  
- \*\*SMOTE\*\*: Synthetic Minority Over-sampling Technique (SMOTE) was used to balance the target variable classes, implemented using `imblearn.over\_sampling.SMOTE`.

1. \*\*Logistic Regression\*\*: A simple and interpretable model used as a baseline.  
2. \*\*Decision Trees\*\*: Used for their simplicity and ability to capture non-linear relationships.  
3. \*\*Random Forests\*\*: An ensemble method that improves performance by averaging multiple decision trees.  
4. \*\*Gradient Boosting (XGBoost, LightGBM)\*\*: Boosting techniques that build models sequentially to correct errors of the previous models.  
5. \*\*Neural Networks (PyTorch)\*\*: Used for their ability to model complex patterns in the data.

encoder = OneHotEncoder()  
encoded\_features = encoder.fit\_transform(df[['categorical\_feature']])  
data = pd.get\_dummies(data, drop\_first=True)

smote = SMOTE()  
X\_train\_balanced, y\_train\_balanced = smote.fit\_resample(X\_train, y\_train)  
```

- \*\*Varying Depth of Decision Trees\*\*: Deeper trees captured more complex patterns but risked overfitting.  
- \*\*Adjusting Learning Rates in Gradient Boosting\*\*: Lower learning rates led to more stable and accurate models over more iterations.  
- \*\*Hyperparameter Tuning with Optuna\*\*: Identified optimal model configurations, improving overall performance.

- \*\*Setup\*\*: MLflow was used to track experiments, log metrics, and save model versions.  
- \*\*Versioning\*\*: Each model iteration was saved with a unique identifier, allowing for easy comparison and rollback.  
- \*\*Logging\*\*: Metrics such as accuracy, precision, recall, and F1 score were logged for each run.

### Example Code for MLflow Integration

```python  
import mlflow  
import mlflow.sklearn

y\_pred = model.predict(X\_test)  
mlflow.log\_metric("accuracy", accuracy\_score(y\_test, y\_pred))  
mlflow.log\_metric("precision", precision\_score(y\_test, y\_pred))  
mlflow.log\_metric("recall", recall\_score(y\_test, y\_pred))  
mlflow.log\_metric("f1\_score", f1\_score(y\_test, y\_pred))

- \*\*Experiment Tracking\*\*: Tools like MLflow are essential for tracking experiments and managing model versions.

- \*\*Alternative Models\*\*: Suggested models like SVM were not pursued due to computational constraints.  
- \*\*Unused Feature Engineering Techniques\*\*: Some recommended techniques were not implemented due to time constraints.  
- \*\*Proposed Visualization Methods\*\*: Certain advanced visualization methods were suggested but not utilized due to complexity.  
- \*\*Alternative Models\*\*: Logistic Regression was suggested but not pursued.  
- \*\*Feature Engineering Techniques\*\*: Certain feature engineering techniques were recommended but not implemented.  
- \*\*Visualization Methods\*\*: Some visualization methods proposed were not utilized.  
- \*\*Hyperparameter Tuning Strategies\*\*: Recommended strategies like Bayesian Optimization were suggested but not applied.  
- \*\*Specific Preprocessing Steps\*\*: Certain preprocessing steps were recommended but skipped.

This report details the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. It covers all aspects of the project, from data preprocessing to final model evaluation, with references to our chat history for context. The report is organized into sections on Techniques and Strategies, Models, Code, Libraries, Combinations and Configurations, Experiment Tracking, Challenges and Solutions, Recommendations, and additional insights.

- Logistic Regression  
- Decision Trees  
- Random Forests  
- Gradient Boosting (XGBoost, LightGBM)  
- Neural Networks (using PyTorch)

def feature\_engineering(df):  
df['Age\_Vehicle\_Age'] = df['Age'] \* df['Vehicle\_Age']  
df['Age\_Previously\_Insured'] = df['Age'] \* df['Previously\_Insured']  
df['Vehicle\_Age\_Damage'] = df['Vehicle\_Age'] \* df['Vehicle\_Damage']  
df['Previously\_Insured\_Damage'] = df['Previously\_Insured'] \* df['Vehicle\_Damage']  
df['Age\_squared'] = df['Age'] \*\* 2  
df['Vehicle\_Age\_squared'] = df['Vehicle\_Age'] \*\* 2  
df['Annual\_Premium\_per\_Age'] = df['Annual\_Premium'] / (df['Age'] + 1)  
return df

continuous\_numeric += ['Age\_Vehicle\_Age', 'Age\_Previously\_Insured', 'Vehicle\_Age\_Damage', 'Previously\_Insured\_Damage', 'Age\_squared', 'Vehicle\_Age\_squared', 'Annual\_Premium\_per\_Age']

X = train\_df.drop('Response', axis=1)  
y = train\_df['Response']  
X\_tensor = torch.tensor(X.values, dtype=torch.float32)  
y\_tensor = torch.tensor(y.values, dtype=torch.float32)

dataset = TensorDataset(X\_tensor, y\_tensor)  
train\_size = int(0.8 \* len(dataset))  
val\_size = len(dataset) - train\_size  
train\_dataset, val\_dataset = random\_split(dataset, [train\_size, val\_size])

train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True, num\_workers=4)

# Initialize model, loss function, optimizer, and GradScaler

model = BinaryClassificationModel(input\_dim=X.shape[1]).to(device)  
criterion = nn.BCEWithLogitsLoss()  
optimizer = optim.Adam(model.parameters(), lr=0.001)  
scaler = GradScaler()

num\_epochs = 20  
train\_losses, val\_losses, val\_accuracies = [], [], []

model.train()  
train\_loss = 0.0  
scaler = GradScaler()

for inputs, labels in train\_loader:

inputs, labels = inputs.to(device), labels.to(device)  
optimizer.zero\_grad()

# Use autocast for mixed precision training

with autocast():  
outputs = model(inputs)  
loss = criterion(outputs, labels.unsqueeze(1))

model.eval()  
val\_loss = 0.0  
correct = 0  
total = 0  
with torch.no\_grad():  
for inputs, labels in val\_loader:  
inputs, labels = inputs.to(device), labels.to(device)  
with autocast():  
outputs = model(inputs)  
loss = criterion(outputs, labels.unsqueeze(1))  
val\_loss += loss.item() \* inputs.size(0)  
predicted = (outputs > 0.5).float()  
total += labels.size(0)  
correct += (predicted.squeeze() == labels).sum().item()

val\_losses.append(val\_loss)  
accuracy = correct / total  
val\_accuracies.append(accuracy)

print(f'Epoch {epoch+1}/{num\_epochs}, Train Loss: {train\_loss:.4f}, Validation Loss: {val\_loss:.4f}, Validation Accuracy: {accuracy:.4f}')

def \_\_init\_\_(self, hparams):  
super(LightGBMModel, self).\_\_init\_\_()  
self.model = lgb.LGBMClassifier(\*\*hparams)  
self.criterion = nn.BCEWithLogitsLoss()  
self.hparams.update(hparams)

x, y = batch  
y\_hat = self(x)  
loss = self.criterion(y\_hat, y)  
return loss

trainer = Trainer(  
max\_epochs=10,  
gpus=1,  
callbacks=[PyTorchLightningPruningCallback(trial, monitor="val\_loss")],  
)

- \*\*pandas\*\*: For loading and preprocessing data.  
- \*\*numpy\*\*: For numerical operations.  
- \*\*scikit-learn\*\*: For data splitting, scaling, and evaluation metrics.  
- \*\*XGBoost\*\*: For initial model trials.  
- \*\*LightGBM\*\*: For final model selection.  
- \*\*Optuna\*\*: For hyperparameter tuning.  
- \*\*PyTorch\*\*: For building and training neural network models.  
- \*\*Seaborn\*\* and \*\*Matplotlib\*\*: For data visualization.

- Kaggle discussions, papers, and tutorials.

By compiling all relevant information into a cohesive report, we provide a clear understanding of the entire modeling process, from data preprocessing to final model evaluation.

# Comprehensive Report on Kaggle Competition Binary Classification Model

train\_df = pd.read\_csv(train\_path)  
test\_df = pd.read\_csv(test\_path)  
print("Datasets loaded successfully.")  
print(f"Train dataset shape: {train\_df.shape}")  
print(f"Test dataset shape: {test\_df.shape}")  
```

from sklearn.model\_selection import train\_test\_split  
X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)  
```

# Train the final model with the best parameters

# Evaluate the final model on the validation set

y\_pred\_val = final\_model.predict\_proba(X\_val)[:, 1]  
roc\_auc = roc\_auc\_score(y\_val, y\_pred\_val)  
print(f"Validation ROC AUC Score with best parameters: {roc\_auc}")  
```

for train\_idx, val\_idx in cv.split(X\_train, y\_train):

X\_train\_cv, X\_val\_cv = X\_train.iloc[train\_idx], X\_train.iloc[val\_idx]  
y\_train\_cv, y\_val\_cv = y\_train.iloc[train\_idx], y\_train.iloc[val\_idx]

This document serves as a thorough reference for the entire modeling process, ensuring that all aspects are covered in detail.

- Binary variables `Gender` and `Vehicle\_Damage` were mapped to numerical values.  
- The `Driving\_License` column, having limited variability, was dropped.  
- Outliers in `Annual\_Premium` were handled by calculating the Interquartile Range (IQR) and filtering out values beyond 1.5 times the IQR.

- Rare categories in categorical features were grouped under a single category 'Other'.  
- Categorical features were encoded using `OneHotEncoder` to convert them into numerical values suitable for model training.

- Continuous numerical features were scaled using `StandardScaler` to normalize the data and improve model convergence.

1. \*\*Neural Networks\*\*  
- A neural network model was designed using PyTorch, with three fully connected layers.  
- The architecture included ReLU activation functions and dropout layers to prevent overfitting.

- Models were evaluated based on ROC AUC score, which is more informative than accuracy for imbalanced datasets.  
- ROC curves were plotted to visualize the performance of the model.

# Drop column with limited variability

# Handle continuous variables and remove outliers

continuous\_numeric = ['Age', 'Vintage', 'Annual\_Premium']  
Q1 = train\_df['Annual\_Premium'].quantile(0.25)  
Q3 = train\_df['Annual\_Premium'].quantile(0.75)  
IQR = Q3 - Q1  
lower\_bound = Q1 - 1.5 \* IQR  
upper\_bound = Q3 + 1.5 \* IQR  
train\_df = train\_df[(train\_df['Annual\_Premium'] >= lower\_bound) & (train\_df['Annual\_Premium'] <= upper\_bound)]  
```

def group\_rare\_categories(df, column, threshold=0.01):  
category\_freq = df[column].value\_counts(normalize=True)  
rare\_categories = category\_freq[category\_freq < threshold].index  
df[column] = df[column].apply(lambda x: 'Other' if x in rare\_categories else x)  
return df

encoder = OneHotEncoder(drop='first', sparse\_output=False)  
categorical\_features = ['Vehicle\_Age', 'Region\_Code', 'Policy\_Sales\_Channel']  
encoded\_features = encoder.fit\_transform(train\_df[categorical\_features])  
encoded\_feature\_names = encoder.get\_feature\_names\_out(categorical\_features)

encoded\_features\_df = pd.DataFrame(encoded\_features, index=train\_df.index, columns=encoded\_feature\_names)  
train\_df = train\_df.drop(categorical\_features, axis=1)  
train\_df = pd.concat([train\_df, encoded\_features\_df], axis=1)  
```

def \_\_init\_\_(self, input\_dim):  
super(InsuranceModel, self).\_\_init\_\_()  
self.fc1 = nn.Linear(input\_dim, 128)  
self.fc2 = nn.Linear(128, 64)  
self.fc3 = nn.Linear(64, 1)  
self.relu = nn.ReLU()  
self.dropout = nn.Dropout(0.3)

model = InsuranceModel(input\_dim)  
device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
model = model.to(device)  
criterion = nn.BCEWithLogitsLoss()  
optimizer = optim.Adam(model.parameters(), lr=0.001)  
scaler = GradScaler()  
num\_epochs = 20

# Training loop with ROC AUC recording

for epoch in range(num\_epochs):  
model.train()  
train\_loss = 0.0  
all\_train\_labels = []  
all\_train\_outputs = []  
for inputs, labels in train\_loader:  
inputs, labels = inputs.to(device), labels.to(device)  
optimizer.zero\_grad()  
with autocast():  
outputs = model(inputs)  
loss = criterion(outputs, labels.unsqueeze(1))  
scaler.scale(loss).backward()  
scaler.step(optimizer)  
scaler.update()  
train\_loss += loss.item() \* inputs.size(0)  
all\_train\_labels.extend(labels.cpu().numpy())  
all\_train\_outputs.extend(outputs.cpu().numpy())  
train\_loss = train\_loss / len(train\_loader.dataset)  
train\_losses.append(train\_loss)  
train\_roc\_auc = roc\_auc\_score(all\_train\_labels, all\_train\_outputs)  
train\_roc\_aucs.append(train\_roc\_auc)

val\_loss = 0.0  
all\_val\_labels = []  
all\_val\_outputs = []  
with torch.no\_grad():  
for inputs, labels in val\_loader:  
inputs, labels = inputs.to(device), labels.to(device)  
with autocast():  
outputs = model(inputs)  
loss = criterion(outputs, labels.unsqueeze(1))  
val\_loss += loss.item() \* inputs.size(0)  
all\_val\_labels.extend(labels.cpu().numpy())  
all\_val\_outputs.extend(outputs.cpu().numpy())  
val\_loss = val\_loss / len(val\_loader.dataset)  
val\_losses.append(val\_loss)  
val\_roc\_auc = roc\_auc\_score(all\_val\_labels, all\_val\_outputs)  
val\_roc\_aucs.append(val\_roc\_auc)

# Plot Training and Validation Loss

plt.figure(figsize=(10, 5))  
plt.plot(range(1, len(train\_losses) + 1), train\_losses, label='Training Loss')  
plt.plot(range(1, len(val\_losses) + 1), val\_losses, label='Validation Loss')  
plt.xlabel('Epochs')  
plt.ylabel('Loss')  
plt.title('Training and Validation Loss')  
plt.legend()  
plt.show()

# Plot Training and Validation ROC AUC

plt.figure(figsize=(10, 5))  
plt.plot(range(1, len(train\_roc\_aucs) + 1), train\_roc\_aucs, label='Training ROC AUC')  
plt.plot(range(1, len(val\_roc\_aucs) + 1), val\_roc\_aucs, label='Validation ROC AUC')  
plt.xlabel('Epochs')  
plt.ylabel('ROC AUC Score')  
plt.title('Training and Validation ROC AUC')  
plt.legend()  
plt.show()

# Print Final Training and Validation Metrics

print(f'Final Training Loss: {train\_losses[-1]:.4f}')  
print(f'Final Validation Loss: {val\_losses[-1]:.4f}')  
print(f'Final Training ROC AUC: {train\_roc\_aucs[-1]:.4f}')  
print(f'Final Validation ROC AUC: {val\_roc\_aucs[-1]:.4f}')  
```

For preprocessing, encoding, and evaluation metrics.

- \*\*Seaborn\*\* and \*\*Matplotlib\*\*: For data visualization.  
- \*\*Logging\*\*: For tracking and recording progress and issues.

This report aims to provide a clear understanding of the entire modeling process, from data preprocessing to final evaluation, ensuring all aspects of the project are covered and explained in detail.

This report provides a detailed overview of the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. It references our entire chat history and includes insights into data preprocessing, model selection, evaluation, and experiment tracking. The goal is to present a cohesive understanding of the modeling process from start to finish.

- \*\*SMOTE (Synthetic Minority Over-sampling Technique)\*\*: SMOTE was used to balance the dataset by generating synthetic samples for the minority class, ensuring the model does not become biased towards the majority class.

1. \*\*Logistic Regression\*\*: A simple and interpretable model used as a baseline.

2. \*\*Decision Trees\*\*: Tree-based models to capture non-linear relationships and interactions between features.  
3. \*\*Random Forests\*\*: An ensemble method that builds multiple decision trees and aggregates their predictions.  
4. \*\*Gradient Boosting Machines (GBM)\*\*: Models like XGBoost and LightGBM that build trees sequentially to correct the errors of previous trees.  
5. \*\*Neural Networks\*\*: Deep learning models implemented using PyTorch for capturing complex patterns in the data.

- \*\*Initial Baseline Models\*\*: Logistic Regression and Decision Trees were used to set a performance baseline.  
- \*\*Ensemble Methods\*\*: Random Forests and Gradient Boosting were explored for their ability to improve performance through ensemble learning.  
- \*\*Hyperparameter Tuning\*\*: GridSearchCV and Optuna were used to find the optimal hyperparameters for the models, improving their performance.  
- \*\*Cross-Validation\*\*: Stratified K-Fold Cross-Validation was employed to ensure the model's robustness and generalizability.

poly = PolynomialFeatures(degree=2)  
X\_train\_poly = poly.fit\_transform(X\_train)  
X\_test\_poly = poly.transform(X\_test)  
```

grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=5, scoring='f1')  
grid\_search.fit(X\_train, y\_train)

def \_\_init\_\_(self, model):  
self.model = model

### Comprehensive Report on Binary Classification Model Construction for Kaggle Competition

This report documents the techniques, strategies, models, and code used in constructing a binary classification model for a Kaggle competition. The content is based on our extensive chat history and is organized into several sections to provide a clear and comprehensive overview.

- Removed duplicate entries.  
- Handled missing values by using imputation methods like mean, median, or mode for numerical features, and the most frequent value for categorical features.

- Used `SimpleImputer` from `scikit-learn` for imputing missing values in the dataset.

- Created new features based on domain knowledge and interactions.  
- Applied one-hot encoding to categorical variables using `OneHotEncoder` from `scikit-learn`.  
- Created polynomial features to capture non-linear relationships.

- Standardized numerical features using `StandardScaler` from `scikit-learn`.

- Visualized data distributions using histograms and box plots (`matplotlib` and `seaborn`).  
- Examined correlations between features using a heatmap (`seaborn`).  
- Visualized categorical variable distributions using bar charts (`seaborn`).

- Applied Synthetic Minority Over-sampling Technique (SMOTE) using `imblearn` to balance the dataset.

- Including `XGBoost` and `LightGBM`, which build trees sequentially to correct errors from previous trees.

- Compared models based on cross-validation scores using `StratifiedKFold` to ensure balanced class distribution.  
- Evaluated models using metrics like accuracy, precision, recall, F1-score, and ROC AUC score.

- Used `GridSearchCV` and `Optuna` for hyperparameter tuning to find the optimal model parameters.  
```python  
# Hyperparameter tuning with Optuna  
import optuna

```python  
# Load and preprocess data  
import pandas as pd  
from sklearn.preprocessing import StandardScaler, OneHotEncoder  
from sklearn.impute import SimpleImputer

```python  
# One-hot encode categorical variables  
encoder = OneHotEncoder(sparse=False)  
encoded\_features = encoder.fit\_transform(train\_df[['categorical\_feature']])

# Combine encoded features with the original dataframe

```python  
# Train and evaluate model  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score, roc\_auc\_score

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

accuracy = accuracy\_score(y\_val, predictions)  
roc\_auc = roc\_auc\_score(y\_val, predictions)  
```

- Implemented custom logging class to track progress.  
- Created an `EarlyStopping` class for neural network training.

- `numpy`: Numerical computing.  
- `scikit-learn`: Machine learning and preprocessing.  
- `imblearn`: Handling imbalanced datasets.  
- `matplotlib`: Data visualization.  
- `seaborn`: Statistical data visualization.  
- `PyTorch`: Neural network implementation.  
- `Optuna`: Hyperparameter optimization.  
- `logging`: Logging progress and events.

- \*\*Data Loading and Preprocessing\*\*: `pandas`, `numpy`, `scikit-learn`.  
- \*\*Data Visualization\*\*: `matplotlib`, `seaborn`.  
- \*\*Model Training and Evaluation\*\*: `scikit-learn`, `PyTorch`.  
- \*\*Hyperparameter Tuning\*\*: `Optuna`.  
- \*\*Logging\*\*: `logging`.

- Logistic regression with polynomial features.  
- Random forests with SMOTE.  
- Gradient boosting with various feature scaling techniques.

- Higher tree depth in random forests led to overfitting.  
- Optimal learning rate in gradient boosting improved convergence.

- Used for tracking experiments, logging parameters, and storing models.

- Each model version logged with parameters and performance metrics.  
- Detailed logs maintained for each training session.

- \*\*Imbalanced Dataset\*\*: Addressed using SMOTE.  
- \*\*Overfitting\*\*: Mitigated using cross-validation and ensemble methods.  
- \*\*Hyperparameter Tuning\*\*: Used Optuna for efficient tuning.

- Importance of feature scaling for gradient boosting.  
- Need for robust validation strategies to avoid overfitting.

- \*\*Feature Engineering\*\*: Focus on creating meaningful features can significantly improve model performance.

- \*\*Model Ensemble\*\*: Combining multiple models can enhance predictive power.  
- \*\*Experiment Tracking\*\*: Using tools like MLflow ensures reproducibility and better management of model versions.

- \*\*Model Interpretability\*\*: Explore SHAP values to interpret model predictions.

- \*\*Feature Selection\*\*: Use techniques like Recursive Feature Elimination (RFE) to identify important features.  
- \*\*Advanced Models\*\*: Experiment with newer models like CatBoost and deep learning architectures.

- \*\*Kaggle Discussions\*\*: Insights and strategies shared by the Kaggle community.

- \*\*Documentation\*\*: Libraries like `scikit-learn`, `PyTorch`, and `Optuna`.  
- \*\*Tutorials\*\*: Online resources and tutorials that guided the project.

This report aims to provide a comprehensive understanding of the entire modeling process, from data preprocessing to final model evaluation, with detailed explanations, code snippets, and visualizations to illustrate key points.

Thank you for your collaboration.

#### List and Description of Models

- \*\*Logistic Regression\*\*: A baseline model to establish initial performance.  
- \*\*Decision Trees\*\*: Tried for interpretability.  
- \*\*Random Forests\*\*: Used to improve performance by averaging multiple decision trees.  
- \*\*Gradient Boosting (XGBoost, LightGBM)\*\*: Employed for their efficiency and performance on large datasets.  
- \*\*Neural Networks\*\*: Implemented using PyTorch for deeper learning capabilities.

```python  
# Import Libraries  
import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler

```python  
# One-Hot Encoding  
df = pd.get\_dummies(df, columns=['Categorical\_Column'])

```python  
# Logistic Regression Example  
from sklearn.linear\_model import LogisticRegression  
from sklearn.metrics import roc\_auc\_score

model.fit(X\_train, y\_train)  
y\_pred = model.predict\_proba(X\_val)[:, 1]  
roc\_auc = roc\_auc\_score(y\_val, y\_pred)  
print(f"ROC AUC Score: {roc\_auc:.4f}")  
```

```python  
# GridSearchCV Example  
from sklearn.model\_selection import GridSearchCV  
param\_grid = {'n\_estimators': [100, 200], 'max\_depth': [3, 5, 7]}  
grid\_search = GridSearchCV(estimator=RandomForestClassifier(), param\_grid=param\_grid, cv=3, scoring='roc\_auc')  
grid\_search.fit(X\_train, y\_train)  
print(f"Best parameters: {grid\_search.best\_params\_}")

import optuna  
import lightgbm as lgb

```python  
# Early Stopping Class  
class EarlyStopping:  
def \_\_init\_\_(self, patience=5, min\_delta=0):  
self.patience = patience  
self.min\_delta = min\_delta  
self.best\_loss = None  
self.counter = 0

if self.best\_loss is None:  
self.best\_loss = val\_loss  
return False  
elif val\_loss < self.best\_loss - self.min\_delta:  
self.best\_loss = val\_loss  
self.counter = 0  
return False  
else:  
self.counter += 1  
if self.counter >= self.patience:  
return True  
return False  
```

- \*\*pandas\*\*: Data loading and manipulation.  
- \*\*numpy\*\*: Numerical operations.  
- \*\*scikit-learn\*\*: Data preprocessing, model training, and evaluation.  
- \*\*imblearn\*\*: Handling imbalanced datasets.  
- \*\*lightgbm\*\*: Gradient boosting models.  
- \*\*optuna\*\*: Hyperparameter optimization.  
- \*\*PyTorch\*\*: Neural network implementation.  
- \*\*seaborn\*\*: Data visualization.  
- \*\*matplotlib\*\*: Plotting graphs.  
- \*\*logging\*\*: Logging process details.

- \*\*pandas\*\*: Used for data loading, cleaning, and feature engineering.  
- \*\*numpy\*\*: Facilitated numerical operations and array manipulations.  
- \*\*scikit-learn\*\*: Provided tools for preprocessing, model training, evaluation, and hyperparameter tuning.  
- \*\*imblearn\*\*: SMOTE for balancing datasets.  
- \*\*lightgbm\*\*: Implemented efficient gradient boosting models.  
- \*\*optuna\*\*: Automated and efficient hyperparameter tuning.  
- \*\*PyTorch\*\*: Built and trained neural network models.  
- \*\*seaborn\*\*: Generated data visualizations like heatmaps and pair plots.  
- \*\*matplotlib\*\*: Created plots for data and model evaluation.  
- \*\*logging\*\*: Tracked the workflow and model training details.

- \*\*Hyperparameter Tuning\*\*: Improved model performance significantly when using GridSearchCV and Optuna.  
- \*\*Balancing with SMOTE\*\*: Enhanced model performance by addressing class imbalance.

- \*\*Experiment Tracking\*\*: Used MLflow to track different model versions, hyperparameter configurations, and performance metrics.  
- \*\*Logging\*\*: Detailed logs maintained to trace back steps and debug if necessary.

- \*\*Balanced Datasets\*\*: Crucial for accurate model evaluation.  
- \*\*Hyperparameter Tuning\*\*: Significant impact on model performance.  
- \*\*Cross-Validation\*\*: Essential for robust model evaluation.

- \*\*Importance of Data Preprocessing\*\*: Ensuring clean and well-preprocessed data is critical for model performance.  
- \*\*Model Selection\*\*: Choosing the right model and tuning it appropriately can significantly impact performance.  
- \*\*Experiment Tracking\*\*: Helps in maintaining an organized workflow and aids in reproducibility.