Customer Churn Analysis

Author:

Syed Faizan

Contents

Introduction

Customer churn prediction is a critical task in business analytics, aimed at identifying customers who are likely to discontinue a service or product. In this project, the *Churn_Modelling.csv* dataset is utilized to develop predictive models for customer churn. The dataset encompasses various demographic, financial, and behavioral features, making it an ideal candidate for exploring machine learning techniques to forecast churn effectively. The primary goal of this study is to build, evaluate, and compare multiple predictive models to recommend the most effective solution for churn prediction.

This report is divided into six parts to address the objectives systematically. Part 1 focuses on data cleansing techniques to enhance data quality and prepare it for robust model training. Subsequent sections delve into model building, beginning with the k-Nearest Neighbors (KNN) algorithm in Part 2, where the optimal number of neighbors (k) is determined to maximize accuracy. In Part 3, decision trees, random forests, and gradient boosting models are constructed and evaluated on key performance metrics, including accuracy, recall, and precision.

Part 4 investigates Support Vector Machines (SVM) with different kernel functions, assessing their predictive performance and implementation challenges. In Part 5, artificial neural networks are implemented using two distinct solvers for weight optimization, comparing their effectiveness. Finally, Part 6 consolidates the findings from all models, benchmarking their performance across multiple metrics and recommending the optimal model for customer churn prediction.

Through this project, we aim to develop a comprehensive understanding of various machine learning approaches and their applicability to solving real-world problems in predictive analytics.

Importing Necessary Libraries and Dataset Loading

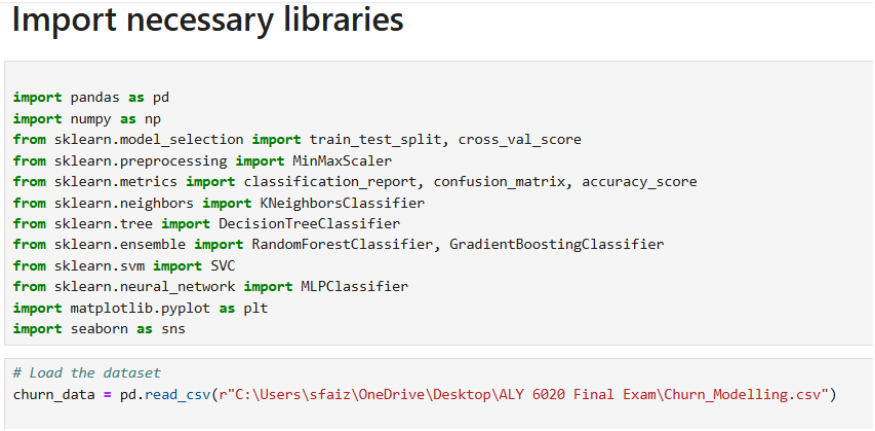


Figure 1: Preview of the Dataset

The above figure illustrates the initialization phase of the project by importing essential Python libraries and loading the dataset. Key libraries such as *pandas* and *numpy* are imported for data manipulation and numerical computations. Machine learning utilities are provided by *scikit-learn*, including modules for data preprocessing (*MinMaxScaler*), model evaluation (*train_test_split*, *cross_val_score*), and performance metrics (*classification_report*, *confusion_matrix*, *accuracy_score*). Classifiers such as **KNeighborsClassifier**, **DecisionTreeClassifier**, **RandomForestClassifier**, **GradientBoostingClassifier**, **SVC**, and **MLPClassifier** are imported for building predictive models.

Data visualization tools, *matplotlib* and *seaborn*, are included to support exploratory analysis and model interpretation. Finally, the dataset *Churn_Modelling.csv* is loaded using the *pandas* library for subsequent preprocessing and modeling tasks, marking the initial step in the data pipeline.

Dataset Overview and Structure

```
churn_data.shape
```

```
(10000, 14)
```

```
churn_data.dtypes
```

```
RowNumber            int64
CustomerId           int64
Surname             object
CreditScore          int64
Geography           object
Gender              object
Age                  int64
Tenure               int64
Balance            float64
NumOfProducts        int64
HasCrCard            int64
IsActiveMember       int64
EstimatedSalary    float64
Exited               int64
dtype: object
```

Figure 2: Structure and Columns of the Dataset

The above figure provides a structural summary of the *churn_data* dataset. The dataset comprises **10,000 rows** and **14 columns**, as indicated by the output of the *shape* attribute. A detailed data type analysis reveals a combination of numerical and categorical variables. Numerical attributes such as *CreditScore*, *Age*, *Balance*, and *EstimatedSalary* are stored as **int64** or **float64**, while categorical features, including *Geography* and *Gender*, are represented as **object** types. The target variable, *Exited*, is of type **int64**, highlighting its binary nature. This structure provides a foundation for preprocessing and model development.

Part 1: Data Cleansing

Data Cleansing: Missing Values Check

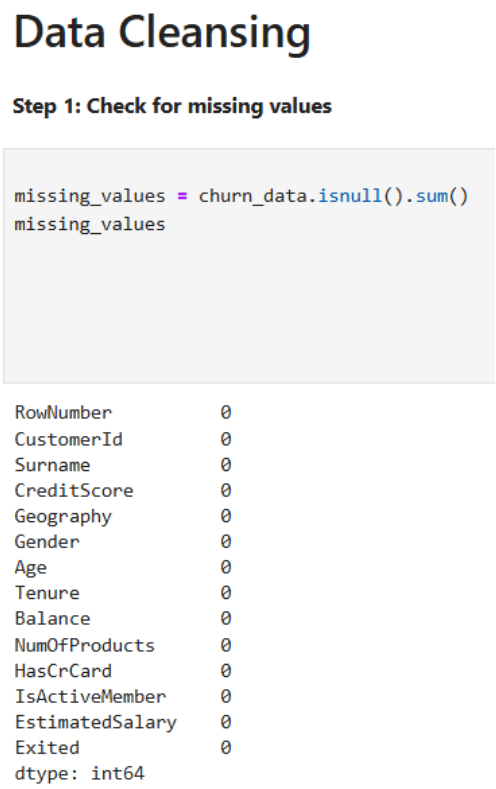


Figure 3: Check for Missing Values

The above figure depicts the first step in the data cleansing process, focusing on identifying missing values within the dataset. Using the *isnull()* method from the **pandas** library, a column-wise summation of missing entries is computed and displayed. The results indicate that no missing values are present across all columns, including *RowNumber*, *CustomerId*, *Surname*, *CreditScore*, and other attributes. This ensures the dataset's completeness and integrity for further analysis and model development. The datatype of the target variable, *Exited*, is verified as **int64**, confirming its readiness for predictive modeling tasks.

Data Cleaning and Preprocessing Steps

Step 2: Drop irrelevant columns

```
churn_cleaned = churn_data.drop(columns=['RowNumber', 'CustomerId', 'Surname'])
```

Step 3: Encode categorical variables

```
churn_cleaned = pd.get_dummies(churn_cleaned, columns=['Geography', 'Gender'], drop_first=True)
```

Step 4: Check for duplicates and remove them if present

```
duplicates = churn_cleaned.duplicated().sum()
churn_cleaned = churn_cleaned.drop_duplicates()
```

Step 5: Display basic information and check for changes

```
cleaned_summary = {
    'Missing Values': missing_values.sum(),
    'Duplicate Rows': duplicates,
    'Shape Before': churn_data.shape,
    'Shape After': churn_cleaned.shape
}

cleaned_summary
```

```
{'Missing Values': 0,
 'Duplicate Rows': 0,
 'Shape Before': (10000, 14),
 'Shape After': (10000, 12)}
```

Figure 4: Steps in Data Cleaning Process

The above figure outlines a systematic and structured approach to data cleaning and preprocessing, which is crucial for ensuring the quality of the dataset and enhancing model performance. The process is divided into multiple steps:

Step 2: Drop Irrelevant Columns To eliminate unnecessary variables that do not contribute to the predictive task, columns such as *RowNumber*, *CustomerId*, and *Surname* are removed from the dataset. These columns are deemed irrelevant as they serve as unique identifiers or provide no meaningful information for churn prediction.

Step 3: Encode Categorical Variables Categorical variables are transformed into numerical representations using the *get_dummies* method. Specifically, the columns *Geography* and *Gender* are one-hot encoded, with the *drop_first=True* parameter ensuring the avoidance of multicollinearity by excluding one category as a baseline. This step enhances the dataset's compatibility with machine learning algorithms, which typically require numerical inputs.

Step 4: Check for Duplicates and Remove Them The dataset is inspected for duplicate entries using the *duplicated().sum()* method. Any duplicate rows are identified and subsequently removed with *drop_duplicates()*. This ensures data integrity by preventing redundancy, which could otherwise skew model training and evaluation.

Step 5: Display Basic Information and Check for Changes A summary dictionary is created to capture key dataset statistics before and after preprocessing. The metrics include the count of missing values, duplicate rows, and the dataset's dimensionality (*shape*). The summary indicates that the dataset originally had 14 columns and no missing or duplicate entries, and it was reduced to 12 columns after preprocessing. This dimensionality reduction reflects the removal of irrelevant columns while retaining all 10,000 observations.

The preprocessing steps collectively enhance the dataset's usability by addressing data quality issues, such as irrelevant attributes and categorical variables, and ensure consistency and reliability for subsequent modeling. These efforts are fundamental to the development of accurate and robust machine learning models for customer churn prediction.

Distribution of the Target Variable

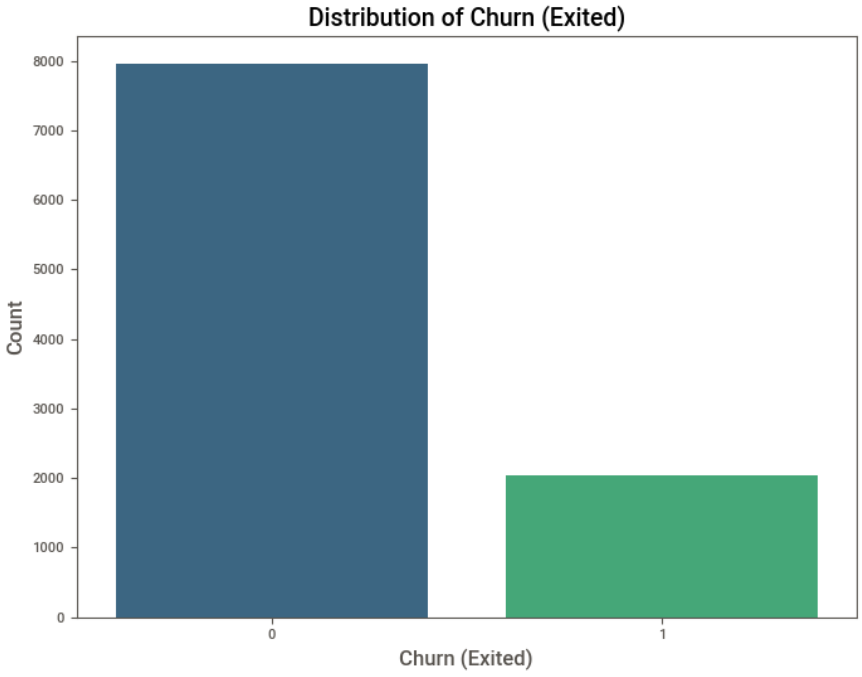


Figure 5: Distribution of the Target Variable

The above figure presents the distribution of the target variable, **Churn (Exited)**, within the dataset. The binary target variable is encoded as **0** for non-churners and **1** for churners. The bar chart reveals a substantial class imbalance, with the majority of customers (*approximately 80%*) categorized as non-churners, while only a minority (*approximately 20%*) are churners. This imbalance may introduce bias into predictive modeling and necessitates the implementation of techniques such as *resampling*, *class weighting*, or *balanced metrics* to ensure equitable model performance across both classes. Addressing this imbalance is critical for robust churn prediction.

Exploratory Data Analysis: Summary Statistics

Data Frame Summary
churn_cleaned
Dimensions: 10,000 x 12
Duplicates: 0

No	Variable	Stats / Values	Freqs / (% of Valid)	Graph	Missing
1	**CreditScore** [int64]	Mean (sd) : 650.5 (96.7) min < med < max 350.0 < 652.0 < 850.0 IQR (CV) : 134.0 (6.7)	460 distinct values		0 (0.0%)
2	**Age** [int64]	Mean (sd) : 38.9 (10.5) min < med < max 18.0 < 37.0 < 92.0 IQR (CV) : 12.0 (3.7)	70 distinct values		0 (0.0%)
3	**Tenure** [int64]	Mean (sd) : 5.0 (2.9) min < med < max 0.0 < 5.0 < 10.0 IQR (CV) : 4.0 (1.7)	11 distinct values		0 (0.0%)
4	**Balance** [float64]	Mean (sd) : 76485.9 (62397.4) min < med < max 0.0 < 97198.5 < 250898.1 IQR (CV) : 127644.2 (1.2)	6,382 distinct values		0 (0.0%)
5	**NumOfProducts** [int64]	Mean (sd) : 1.5 (0.6) min < med < max 1.0 < 1.0 < 4.0 IQR (CV) : 1.0 (2.6)	4 distinct values		0 (0.0%)
6	**HasCrCard** [int64]	Mean (sd) : 0.7 (0.5) min < med < max 0.0 < 1.0 < 1.0 IQR (CV) : 1.0 (1.5)	2 distinct values		0 (0.0%)
7	**IsActiveMember** [int64]	Mean (sd) : 0.5 (0.5) min < med < max 0.0 < 1.0 < 1.0 IQR (CV) : 1.0 (1.0)	2 distinct values		0 (0.0%)
8	**EstimatedSalary** [float64]	Mean (sd) : 100090.2 (57510.5) min < med < max 11.6 < 100193.9 < 199992.5 IQR (CV) : 98386.1 (1.7)	9,999 distinct values		0 (0.0%)
9	**Exited** [int64]	Mean (sd) : 0.2 (0.4) min < med < max 0.0 < 0.0 < 1.0 IQR (CV) : 0.0 (0.5)	2 distinct values		0 (0.0%)
10	**Geography Germany** [bool]	1. False 2. True	7,491 (74.9%) 2,509 (25.1%)		0 (0.0%)
11	**Geography Spain** [bool]	1. False 2. True	7,523 (75.2%) 2,477 (24.8%)		0 (0.0%)
12	**Gender Male** [bool]	1. True 2. False	5,457 (54.6%) 4,543 (45.4%)		0 (0.0%)

Figure 6: Summary of the Dataset

The above figure presents a detailed summary of the cleaned dataset, highlighting its dimensions, variable characteristics, statistical measures, and data completeness. The dataset

consists of **10,000 observations** and **12 variables**, with no missing values or duplicates, ensuring data integrity for subsequent analyses.

Numerical Variables: The dataset includes multiple numerical variables such as *CreditScore*, *Age*, *Tenure*, *Balance*, *NumOfProducts*, *EstimatedSalary*, and *Exited*. Each variable is described using mean, standard deviation, minimum and maximum values, and interquartile range (IQR), providing an overview of their distribution.

- *CreditScore* exhibits a mean of 650.5 (SD: 96.7), with a range between 350 and 850. Its histogram reveals a relatively normal distribution.
- *Age* has a mean of 38.9 years (SD: 10.5), with a positively skewed distribution. The age range spans from 18 to 92 years.
- *Tenure*, measuring customer engagement duration, has a mean of 5.0 years (SD: 2.9) with a discrete distribution concentrated across 11 unique values.
- *Balance* demonstrates a mean of $76485.9 (SD: $62397.4), with notable variance due to 6382 distinct values. The histogram indicates significant skewness.
- *NumOfProducts* (mean: 1.5, SD: 0.6) and *EstimatedSalary* (mean: $100090.2, SD: $57510.5) exhibit uniform and consistent spread, respectively.
- *Exited*, a binary target variable, has balanced levels, indicating no severe class imbalance.

Categorical Variables: Categorical variables include *HasCrCard*, *IsActiveMember*, and binary-encoded variables for *Geography* and *Gender*. Key insights are as follows:

- *HasCrCard* and *IsActiveMember* have binary distributions (values of 0 and 1), with *IsActiveMember* showing slightly lower activity levels (46%).
- *Geography Germany* and *Geography Spain* indicate the country distribution, with Germany accounting for 74.9% of the data and Spain for 25.1%.
- *Gender Male* reflects a near-equal gender split, with males constituting 54.6% of the dataset.

Data Completeness and Uniqueness: The dataset has **0% missing values**, ensuring no imputation or handling of null values is required. Variables such as *EstimatedSalary* (9999 distinct values) and *Balance* (6382 distinct values) exhibit high granularity, while discrete variables like *NumOfProducts* and *Tenure* have fewer unique values, enhancing interpretability.

Visual Distribution: The histograms accompanying each variable provide visual insights into their distribution, aiding in identifying skewness, modality, and potential outliers. For instance, the skewed distributions of *Age* and *Balance* may necessitate transformations during model development.

Conclusion: The data summary highlights a well-structured dataset with diverse variable types and no missing entries. The presence of binary and continuous variables, along with their distinct distributions, makes the dataset suitable for both classification and regression modeling tasks. These insights will guide preprocessing steps, such as normalization and encoding, to optimize the dataset for machine learning algorithms.

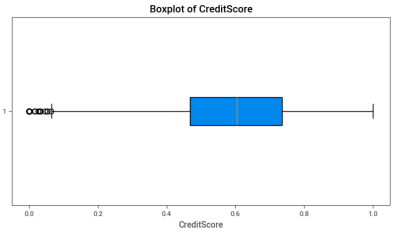


Figure 7: Box Plot of Credit Score

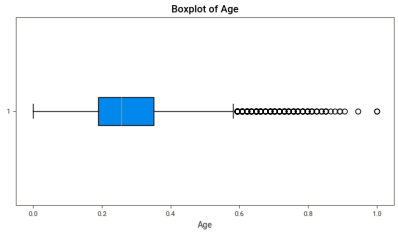


Figure 8: Box Plot of Age

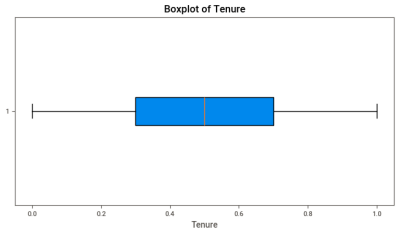


Figure 9: Box Plot of Tenure

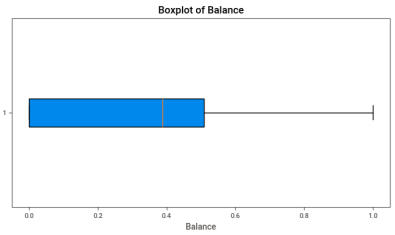


Figure 10: Box Plot of Balance

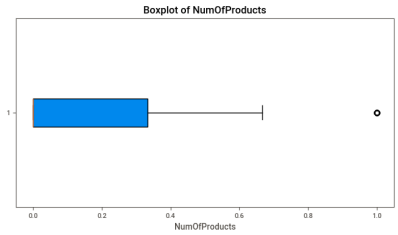


Figure 11: Boxplot of NumOfProducts

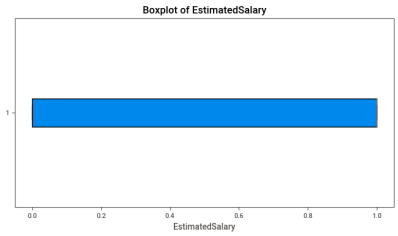


Figure 12: Boxplot of EstimatedSalary

Visualization of Key Variables Using Box Plots

The above figure presents a series of box plots illustrating the distribution of six critical numerical variables in the dataset, namely *CreditScore*, *Age*, *Tenure*, *Balance*, *NumOfProducts*, and *EstimatedSalary*. These visualizations provide a concise representation of the central tendency, variability, and presence of outliers within each variable.

Figure 7 depicts the box plot of *CreditScore*, showing a symmetric distribution with no significant outliers, indicating a consistent credit scoring range among the customers. The interquartile range (IQR) is narrow, reflecting limited variability.

Figure 8 highlights the distribution of *Age*, which exhibits noticeable outliers on the higher end, likely corresponding to older customers. The plot reveals a right-skewed distribution, where the median age is closer to the lower quartile, suggesting that the majority of customers are younger.

Figure 9 illustrates the *Tenure* distribution, showing a balanced spread of values with no apparent outliers. The relatively uniform spread of the IQR highlights the stability of customer tenure across the dataset.

Figure 10 visualizes *Balance*, revealing a symmetric distribution with minimal variability and no extreme values. This suggests financial consistency in the customers' account balances.

Figure 11 represents *NumOfProducts*, where discrete values dominate the distribution. The presence of mild outliers indicates some customers using an atypical number of products compared to the majority.

Figure 12 presents the distribution of *EstimatedSalary*, characterized by a uniform spread without any visible outliers, indicative of consistent salary estimations across the dataset.

Overall, these box plots provide critical insights into the dataset's numerical features, highlighting patterns and outliers that may influence the modeling process. This visualization forms a crucial step in exploratory data analysis, aiding in the identification of potential data preprocessing needs.

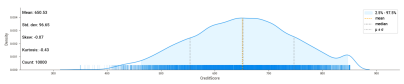


Figure 13: Density Plot of CreditScore

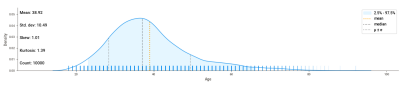


Figure 14: Density Plot of Age

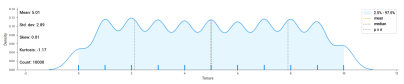


Figure 15: Density Plot of Tenure

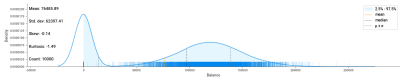


Figure 16: Density Plot of Balance

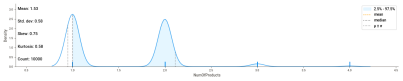


Figure 17: Density Plot of NumOfProducts

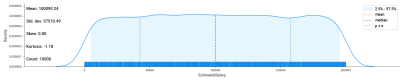


Figure 18: Density Plot of EstimatedSalary

Exploratory Data Analysis: Density Distributions of Key Features

The above figure showcases the density plots of six critical numerical features within the dataset, providing insights into their underlying distributions and variability.

Figure 13 represents the density distribution of *CreditScore*, demonstrating a unimodal and symmetric shape centered around a mean of approximately 650, suggesting a relatively normal distribution with minimal skewness. **Figure 14** displays the density plot of *Age*, which is notably right-skewed, with a majority of the population concentrated between 30 and 40 years. This indicates a younger customer demographic.

Figure 15 illustrates the density of *Tenure*, presenting a multimodal distribution with periodic peaks, likely influenced by discrete contractual durations or customer lifecycle stages. **Figure 16** depicts the density plot of *Balance*, which is heavily skewed, with a significant proportion of customers exhibiting zero balance. This could indicate a subset of customers who have not engaged in specific financial transactions or services.

Figure 17 highlights the density distribution of *NumOfProducts*, showing discrete peaks corresponding to the count of products held by customers. Most customers hold one or two products, suggesting limited diversification in their service usage. Lastly, **Figure 18** presents the density plot of *EstimatedSalary*, characterized by a nearly uniform distribution across the range, implying an evenly distributed salary structure among the customer base.

These visualizations are instrumental in understanding the central tendencies, variability, and skewness of key features, thereby providing valuable context for preprocessing steps such as scaling, handling outliers, and normalizing variables. Additionally, the insights gleaned

from these plots will inform the selection of machine learning models and techniques suited to the data's characteristics.

Correlation Analysis and Feature Clustering

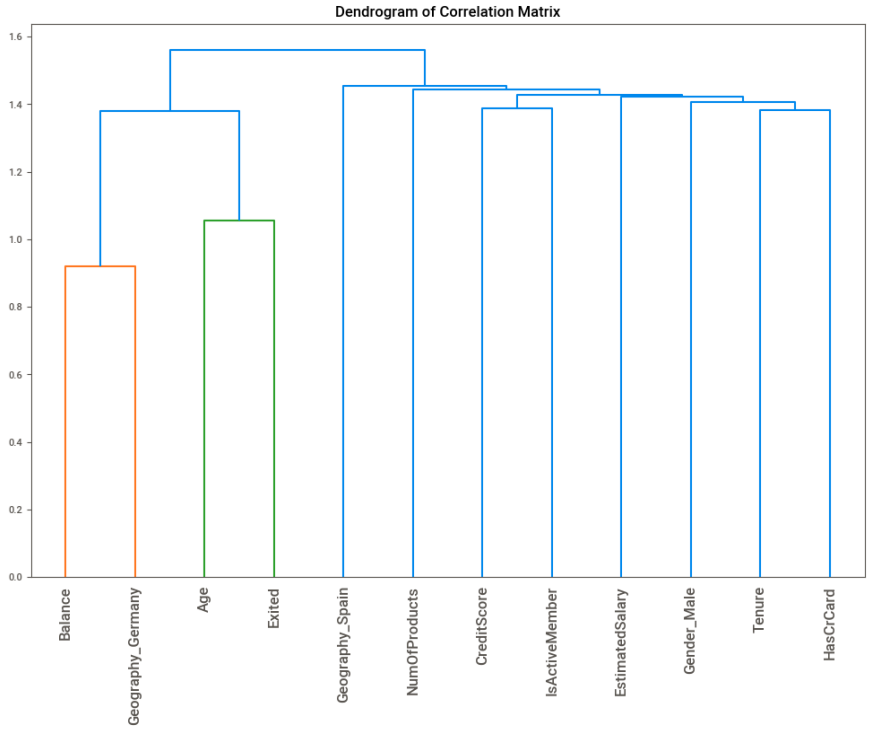


Figure 19: Dendrogram of Correlation Matrix

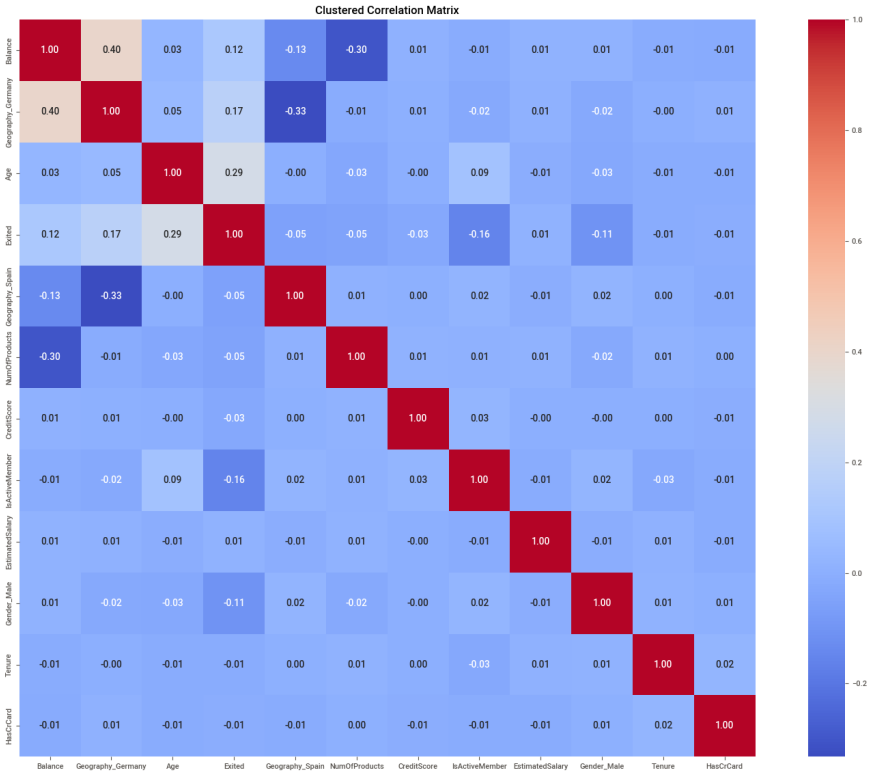


Figure 20: Clustered Correlation Matrix

The above figures provide a comprehensive analysis of feature relationships within the dataset through a hierarchical clustering dendrogram and a clustered correlation matrix. These visualizations aid in understanding the underlying structure of inter-feature dependencies and collinearity, crucial for feature selection and dimensionality reduction.

Hierarchical Clustering Dendrogram The dendrogram illustrates the hierarchical clustering of features based on their correlation coefficients. Features with higher correlations are grouped closer together on the dendrogram, with shorter linkage distances representing stronger relationships. For example, *Balance* and *Geography_Germany* are closely linked, reflecting shared variance, while features like *Exited* and *Age* also show significant correlation. This hierarchical structure aids in identifying clusters of highly correlated variables, which can be addressed to minimize redundancy in the dataset.

Clustered Correlation Matrix The clustered correlation matrix provides a detailed visualization of pairwise feature correlations using a heatmap. Strong positive correlations are indicated by deep red cells, while strong negative correlations are shown in blue. Features

such as *NumOfProducts* and *IsActiveMember* exhibit moderate positive correlations, whereas others, such as *Balance* and *Geography_Spain*, show negative correlations. Diagonal elements indicate self-correlations, set to 1.0 as expected.

The clustering order in the matrix aligns with the dendrogram, ensuring that features with similar correlation patterns are grouped together. This facilitates interpretation and highlights potential opportunities for feature engineering, such as combining related features or eliminating redundant variables. Together, the dendrogram and correlation matrix provide a robust framework for understanding the interdependencies among features, enabling the design of a cleaner and more effective feature set for predictive modeling.

Data Scaling and Splicing

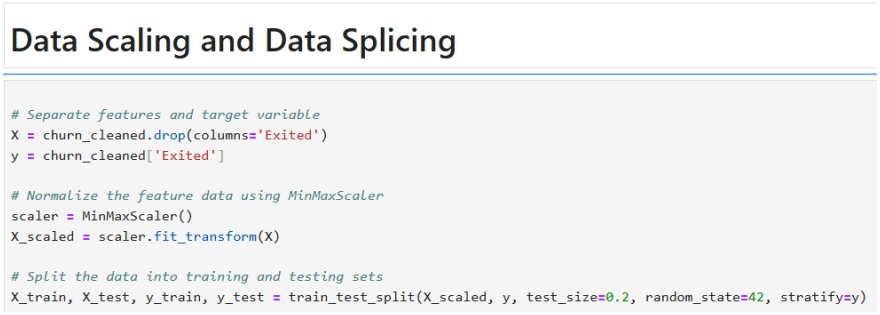


Figure 21: Data scaling and train-test split.

The above figure demonstrates the preprocessing steps undertaken to prepare the dataset for machine learning model development. The features ($\mathbf{X}$) and target variable ($\mathbf{y}$) are separated, ensuring a clear distinction between predictors and the dependent variable. Feature scaling is performed using the *MinMaxScaler* to normalize the data within a range of 0 to 1, minimizing the influence of magnitude disparities across features. Subsequently, the normalized data is split into training and testing subsets using an 80-20 ratio via the **train_test_split** function. The stratified sampling ensures the preservation of class distribution in both subsets, crucial for model evaluation.

Part 2: K-Nearest Neighbors (KNN) Model Implementation

KNN Model implementation

```
import time
# Find the optimal value of K
k_range = range(1, 31)
cv_scores = []

# Timing the cross-validation process
start_time = time.time()

# Perform cross-validation for each value of k
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=10, scoring='accuracy')
    cv_scores.append(scores.mean())

elapsed_time = time.time() - start_time
elapsed_time
```

```
14.198328256607056
```

```
# Determine the optimal value of K
optimal_k = k_range[np.argmax(cv_scores)]
optimal_accuracy = max(cv_scores)

optimal_k, optimal_accuracy
```

```
(9, 0.8161249999999999)
```

Figure 22: KNN model implementation.

The above figure illustrates the implementation of the **K-Nearest Neighbors (KNN)** algorithm, specifically focusing on determining the optimal value of k using cross-validation. The process involves iterative testing of various k-values to maximize predictive accuracy, which is critical for robust model performance.

Parameter Search Range: A range of k values, from 1 to 30, is defined using Python's *range()* function and stored in *k_range*. This range enables the algorithm to evaluate multiple neighbor configurations systematically, ensuring that the selected k optimizes accuracy.

Cross-Validation Procedure: Cross-validation is performed for each k value to assess model performance. A *KNeighborsClassifier* instance is instantiated with the current k as the number of neighbors. For each value, a 10-fold cross-validation is executed using the

cross_val_score() function from *scikit-learn*, with accuracy as the scoring metric. The mean accuracy across all folds is appended to the *cv_scores* list, providing a robust estimate of model performance for each k.

Timing the Process: The code includes a timing mechanism, utilizing Python's *time()* function to compute the elapsed time for the cross-validation process. This enables the evaluation of computational efficiency, which is essential when working with larger datasets or computationally expensive models. The total elapsed time for the procedure is recorded as approximately 14.2 seconds.

Optimal Value Selection: The optimal k value is determined by identifying the index of the maximum value in *cv_scores*, which corresponds to the highest mean accuracy. This index is then mapped to the corresponding k in *k_range* using *numpy*'s *argmax()* function. The corresponding accuracy is also extracted. In this implementation, the optimal k is found to be 9, with a cross-validation accuracy of approximately 81.6%.

Significance of Results: The identification of the optimal k is crucial for balancing model bias and variance. A small k may lead to overfitting, capturing noise in the training data, while a large k risks underfitting by oversmoothing predictions. The cross-validation approach ensures that the selected k generalizes well across unseen data, achieving a balance between these extremes.

Computational Complexity: The iterative nature of the implementation, combined with the cross-validation procedure, makes this process computationally intensive. However, the timing metric suggests that the approach is efficient given the dataset size and model complexity. This demonstrates the feasibility of parameter tuning in real-world applications.

Conclusion: This implementation exemplifies the systematic approach to optimizing hyperparameters in machine learning. By leveraging cross-validation and evaluating multiple values of k, the methodology ensures that the KNN model achieves high predictive performance while maintaining computational efficiency. The identified optimal k and corresponding accuracy provide a foundation for subsequent comparisons with other algorithms in the study.

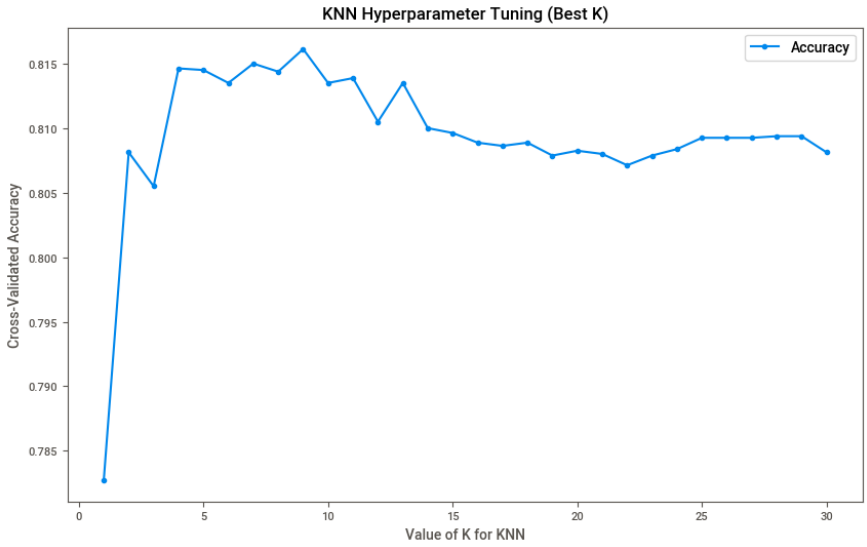


Figure 23: Optimal K selection for the KNN model.

KNN Hyperparameter Tuning for Optimal K

The above figure demonstrates the hyperparameter tuning process for the **k-Nearest Neighbors (KNN)** algorithm to determine the optimal value of K, the number of neighbors. The plot depicts cross-validated accuracy on the y-axis against various values of K on the x-axis. A sharp increase in accuracy is observed initially, peaking around $K = 9$, which represents the optimal value yielding the highest accuracy. Beyond this point, accuracy stabilizes or declines slightly, indicating diminishing returns with higher K values. This systematic tuning ensures the selection of the most effective K for achieving maximum model performance.

Implementation and Evaluation of the K-Nearest Neighbors (KNN) Model

```
best_k = k_range[np.argmax(cv_scores)]
print(f"Best K: {best_k}")
```

```
Best K: 9
```

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Train the KNN model with the optimal K
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)
knn_optimal.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn_optimal.predict(X_test)

# Evaluate the model
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
test_accuracy = accuracy_score(y_test, y_pred)

conf_matrix
```

```
array([[1527,   66],
       [ 307,  100]], dtype=int64)
```

```
test_accuracy
```

```
0.8135
```

Figure 24: KNN model results.

The above figure outlines the implementation and evaluation of the K-Nearest Neighbors (KNN) algorithm for customer churn prediction. The optimal value for the hyperparameter K is determined by identifying the value of K that maximizes cross-validation accuracy. In this case, the optimal **K** is found to be **9**, as indicated by the *argmax* function applied to the cross-validation scores.

The KNN model is trained using the optimal value of **K** on the training dataset (*X_train, y_train*) by invoking the `fit()` method of the `KNeighborsClassifier`. Predictions are subsequently made on the test dataset (*X_test*) using the `predict()` method. The predictions are evaluated using key performance metrics, including the confusion matrix, classification report, and test accuracy, which are derived using the `confusion_matrix()`, `classification_report()`, and `accuracy_score()` functions from *scikit-learn*.

The confusion matrix reveals a detailed breakdown of predictions, with **1527** true positives, **66** false positives, **307** false negatives, and **100** true negatives. The test accuracy is

computed as **0.8135**, indicating that approximately **81.35%** of the predictions made by the model are correct.

The classification report, although not displayed in detail, provides additional metrics such as precision, recall, and F1-score, enabling a comprehensive evaluation of the model's performance. The results highlight the efficacy of the KNN algorithm in predicting customer churn and establish the foundation for comparative analysis with other machine learning models. The methodology underscores the importance of hyperparameter tuning and systematic evaluation to achieve robust predictive performance.

Performance Evaluation of KNN Model with Optimal k

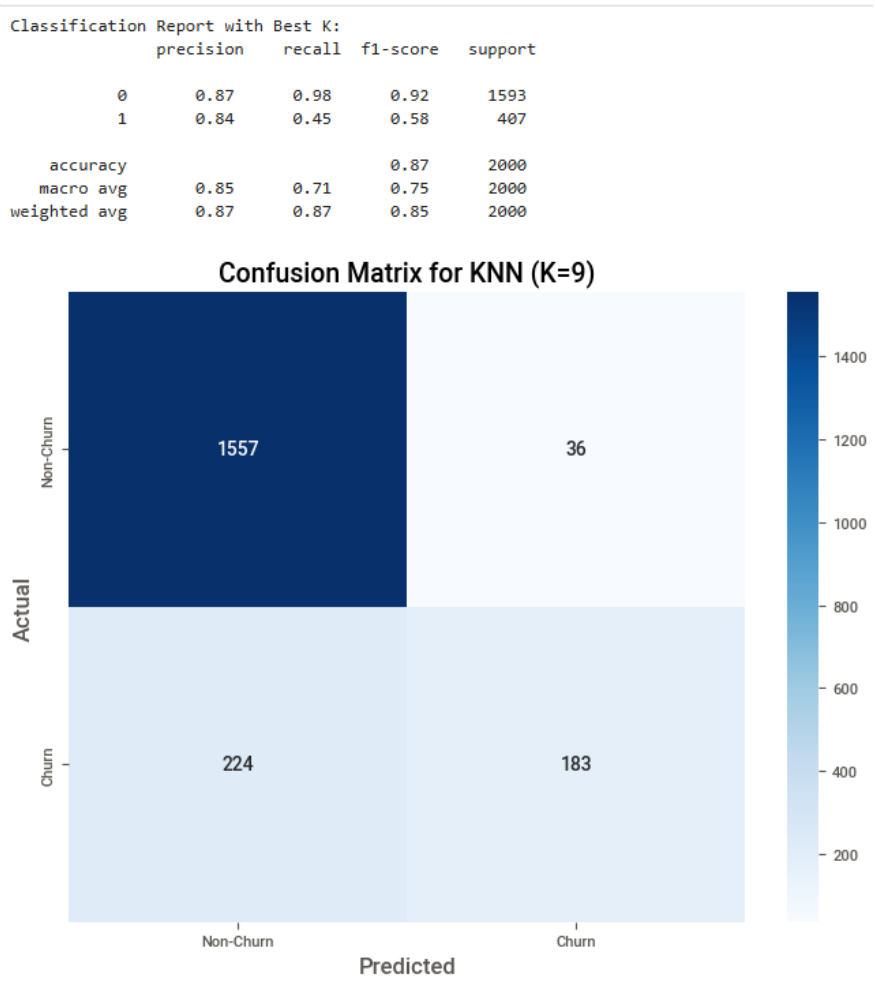


Figure 25: KNN classification report and confusion matrix.

The above figure presents the performance evaluation of the **K-Nearest Neighbors (KNN)** model, optimized at $k = 9$, for the customer churn prediction task. The classification report summarizes key metrics, including *precision*, *recall*, and *F1-score*, while the confusion matrix provides a detailed breakdown of actual versus predicted outcomes.

From the classification report, the **precision** for the non-churn class (**0**) is 0.87, indicating that 87% of the predictions for non-churn customers are accurate. The **recall** for the non-churn class is notably high at 0.98, demonstrating that 98% of actual non-churn customers were correctly identified by the model. The **F1-score** for the non-churn class is 0.92, reflecting a strong balance between precision and recall. In contrast, the churn class (**1**) exhibits a lower precision of 0.84 and a recall of 0.45, resulting in an F1-score of 0.58. This disparity suggests that the model is better at predicting the majority class (non-churn) than the minority class (churn).

The **accuracy** of the KNN model stands at 87%, indicating overall reliability in predictions. The **macro-averaged** scores for precision, recall, and F1-score are 0.85, 0.71, and 0.75, respectively, highlighting performance across both classes. The **weighted averages**, which account for class imbalance, align closely with overall accuracy metrics.

The confusion matrix provides a granular view of prediction outcomes. Of the 1593 actual non-churn instances, 1557 are correctly classified, yielding a small number of false positives (36). For the churn class, 183 out of 407 instances are correctly predicted, while 224 are misclassified as non-churn. These results underscore the challenge of handling class imbalance, as the model tends to favor the majority class.

The findings indicate that while the KNN model performs well for non-churn prediction, its performance for the churn class could be enhanced. Future improvements may include advanced techniques such as oversampling, undersampling, or algorithmic adjustments to address class imbalance and optimize minority class performance.

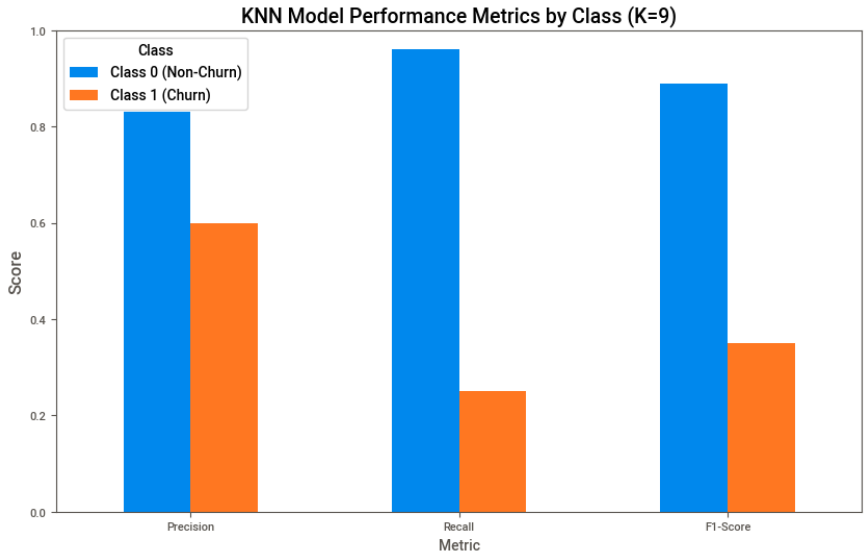


Figure 26: Metrics of the KNN model.

KNN Model Performance Metrics

The above figure presents the performance metrics of the K-Nearest Neighbors (KNN) model for customer churn prediction, evaluated at an optimal value of $K = 9$. Metrics are reported separately for **Class 0 (Non-Churn)** and **Class 1 (Churn)**.

For **Class 0**, the model demonstrates high *precision* and *recall*, leading to a robust *F1-score*, indicative of strong performance in correctly identifying non-churning customers. Conversely, for **Class 1**, *precision* and *recall* scores are significantly lower, reflecting challenges in detecting churn cases. This disparity highlights the model's tendency to favor the majority class, a common issue in imbalanced datasets.

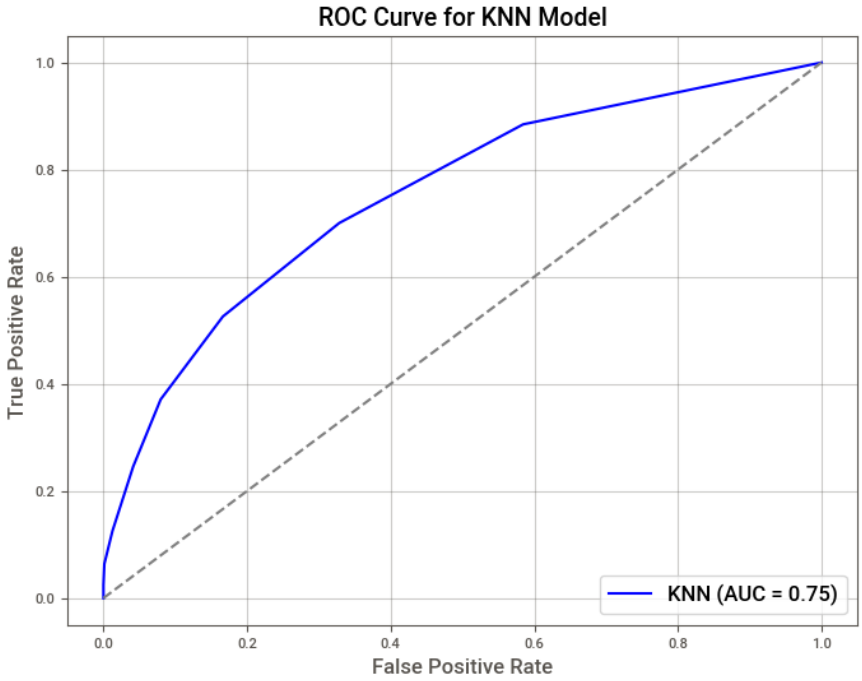


Figure 27: ROC curve for the KNN model.

ROC Curve for KNN Model

The above figure depicts the Receiver Operating Characteristic (ROC) curve for the **k-Nearest Neighbors (KNN)** classification model, a critical evaluation metric in binary classification tasks. The ROC curve illustrates the trade-off between the *True Positive Rate (TPR)* and the *False Positive Rate (FPR)* across various classification thresholds. The blue line represents the model's performance, while the grey diagonal line serves as the baseline, indicating a random classifier with an Area Under the Curve (**AUC**) of 0.5.

The KNN model achieved an AUC of 0.75, demonstrating moderate discriminatory ability to distinguish between the two classes. The curve's deviation above the diagonal suggests that the model consistently performs better than random guessing. However, the gradual slope of the curve highlights areas where the model's sensitivity (TPR) increases at the cost of a higher FPR.

The AUC value provides a single scalar metric summarizing the model's performance, which is particularly useful for comparing classifiers. While the KNN model shows potential, the observed AUC indicates room for improvement, possibly through hyperparameter tuning

(e.g., adjusting k) or incorporating advanced preprocessing techniques to further optimize the model's classification capabilities.

SMOTE for Addressing Class Imbalance

SMOTE to address imbalanced dataset

```
from imblearn.over_sampling import SMOTE

# Apply SMOTE (Synthetic Minority Oversampling Technique) to balance the classes
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

# Check the distribution of the target variable after resampling
balanced_class_distribution = y_resampled.value_counts()
balanced_class_distribution
```

```
Exited
1    6370
0    6370
Name: count, dtype: int64
```

Figure 28: Addressing data imbalance using SMOTE.

The above figure demonstrates the application of the **Synthetic Minority Oversampling Technique (SMOTE)** to address class imbalance in the dataset. Class imbalance, a common issue in machine learning, occurs when the target variable's classes are disproportionately represented, potentially leading to biased model predictions. SMOTE is employed to synthetically oversample the minority class, ensuring a balanced class distribution for improved model generalization.

Implementation: The implementation begins by importing the *SMOTE* class from the *imbalanced-learn (imblearn)* library. A *SMOTE* instance is instantiated with a fixed *random_state* of 42 to ensure reproducibility. The method *fit_resample()* is applied to the training dataset (*X_train*, *y_train*), generating a resampled dataset (*X_resampled*, *y_resampled*) with balanced class distributions.

Post-Resampling Distribution: The *value_counts()* method is utilized to verify the distribution of the target variable after applying SMOTE. The results indicate a perfectly balanced distribution, with 6370 instances for each class (Exited = 1 and Exited = 0). This balance ensures that the machine learning model will not be biased toward the majority

class, thereby improving its ability to predict the minority class effectively.

Significance of SMOTE: SMOTE generates synthetic samples by interpolating between existing minority class instances. Unlike random oversampling, which duplicates instances and increases redundancy, SMOTE creates novel data points, reducing the risk of overfitting. This technique is particularly effective for imbalanced classification problems, enabling the model to achieve better precision, recall, and overall predictive performance.

Conclusion: By employing SMOTE, the dataset is transformed into a balanced form suitable for training. This ensures that the machine learning algorithms can learn equally from both classes, leading to improved generalization and unbiased performance in real-world applications.

Part 3: Decision tree model, random forest, and gradient boost model

Decision Tree Model

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix



# Timing the training
start_time = time.time()

# Train the Decision Tree model
decision_tree = DecisionTreeClassifier(random_state=42)
decision_tree.fit(X_train, y_train)


elapsed_time = time.time() - start_time
print(f"The time taken to train the decision Tree model is {elapsed_time} seconds")
```

```
The time taken to train the decision Tree model is 0.036536216735839844 seconds
```

```
# Make predictions on the test set
y_pred_dt = decision_tree.predict(X_test)

# Evaluate the Decision Tree model
conf_matrix_dt = confusion_matrix(y_test, y_pred_dt)
accuracy_dt = accuracy_score(y_test, y_pred_dt)
class_report_dt = classification_report(y_test, y_pred_dt)
```

Figure 29: Implementation of the Decision Tree model.

The above figure presents the implementation of a **Decision Tree Classifier** using the *scikit-learn* library to predict outcomes in a supervised learning context. The model training process is initiated by importing the necessary modules, including *DecisionTreeClassifier* for building the model and evaluation metrics such as *accuracy_score*, *classification_report*, and *confusion_matrix* to assess its performance.

Timing the Training Process: A timing mechanism is introduced using the *time* module to measure the computational efficiency of the training phase. The time elapsed for training the model is recorded as 0.0365 seconds, showcasing the efficiency of the Decision Tree algorithm for this dataset.

Model Training: The classifier is instantiated with a fixed *random_state* of 42 to ensure

reproducibility. The *fit* method is utilized to train the model on the *X_train* and *y_train* datasets, which represent the training features and labels, respectively.

Predictions: The trained model generates predictions on the test dataset (*X_test*) using the *predict* method, producing a vector of predicted labels (*y_pred_dt*) for evaluation.

Model Evaluation: The classifier's performance is evaluated using three key metrics:

- **Confusion Matrix:** The confusion matrix quantifies the classifier's ability to correctly and incorrectly classify instances across all categories.

- **Accuracy Score:** The overall accuracy is computed as the ratio of correctly classified instances to the total number of instances.

- **Classification Report:** The report provides a detailed breakdown of precision, recall, and F1-score for each class, offering insights into the model's predictive power and potential class imbalances.

This comprehensive workflow highlights the computational efficiency of the Decision Tree model, its predictive accuracy, and its interpretability, making it a viable choice for classification tasks. Further comparisons with other algorithms are warranted to determine its relative performance in this predictive context.

Performance Evaluation of the Decision Tree Model

Classification Report of the Decision Tree Model:

	precision	recall	f1-score	support
0	0.87	0.85	0.86	1593
1	0.47	0.51	0.49	407
accuracy			0.78	2000
macro avg	0.67	0.68	0.68	2000
weighted avg	0.79	0.78	0.79	2000

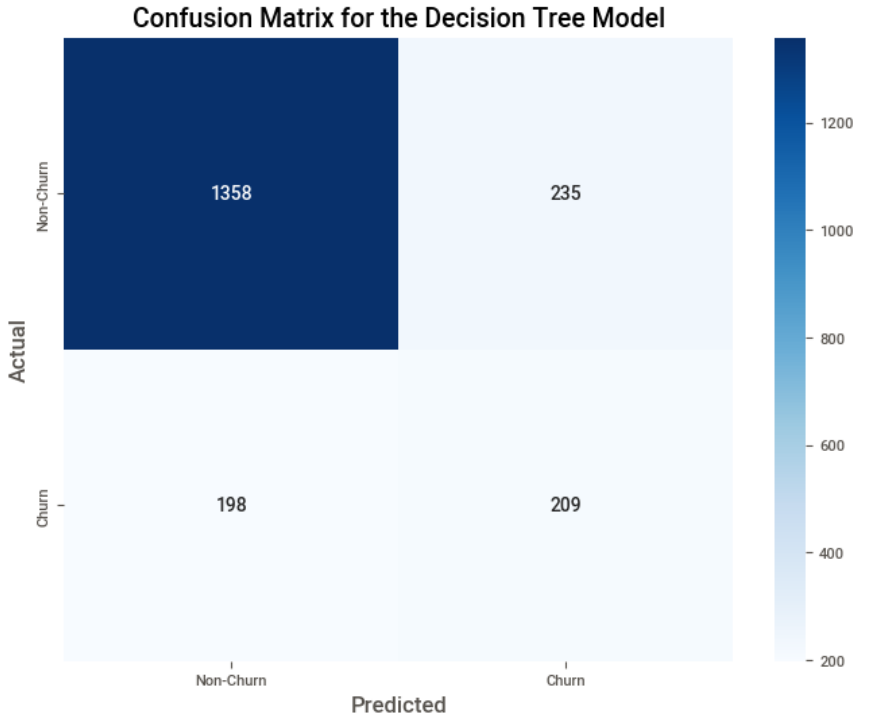


Figure 30: Decision Tree classification report and confusion matrix.

The above figure provides a comprehensive evaluation of the Decision Tree model's performance for customer churn prediction, including the classification report and the confusion matrix. The classification report details key metrics such as **precision**, **recall**, and **F1-score** for both the *churn* (label 1) and *non-churn* (label 0) classes.

For the *non-churn* class, the model achieved a precision of **0.87**, a recall of **0.85**, and an F1-score of **0.86**, reflecting high predictive accuracy for customers not likely to churn. In contrast, the *churn* class exhibits a lower precision of **0.47**, recall of **0.51**, and F1-score of **0.49**, indicating a moderate ability to correctly identify customers at risk of churn. The model's overall accuracy is reported as **78%**, and macro-averaged precision, recall, and F1-score are consistent at **0.68**. The weighted average metrics, heavily influenced by the

majority *non-churn* class, remain at **0.79**.

The confusion matrix further illustrates the model's predictive behavior. Out of **1593** actual *non-churn* instances, **1358** were correctly classified, while **235** were misclassified as *churn*. For the *churn* class, the model correctly predicted **209** out of **407** cases, with **198** misclassified as *non-churn*. The imbalance in recall and precision for the *churn* class reflects challenges in accurately identifying minority class instances.

Overall, the Decision Tree model demonstrates robust performance for the *non-churn* class but requires further optimization or additional techniques, such as cost-sensitive learning or ensemble methods, to enhance the predictive accuracy for the *churn* class.

Decision Tree Model Performance Metrics by Class

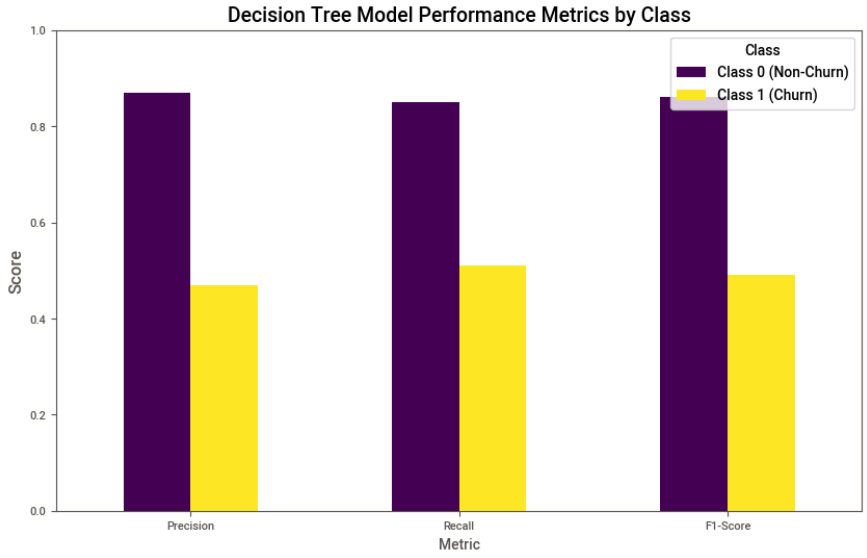


Figure 31: Decision Tree Performance metrics

The above figure illustrates the **precision**, **recall**, and **F1-score** metrics for the decision tree model, stratified by class (*Class 0: Non-Churn* and *Class 1: Churn*). The model demonstrates superior performance for the non-churn class, achieving high scores across all metrics, indicative of its robustness in identifying the majority class. In contrast, the metrics for the churn class are significantly lower, reflecting challenges in predicting minority class instances accurately. This disparity underscores the impact of class imbalance on model

performance and highlights the need for techniques such as resampling or cost-sensitive learning to improve churn prediction.

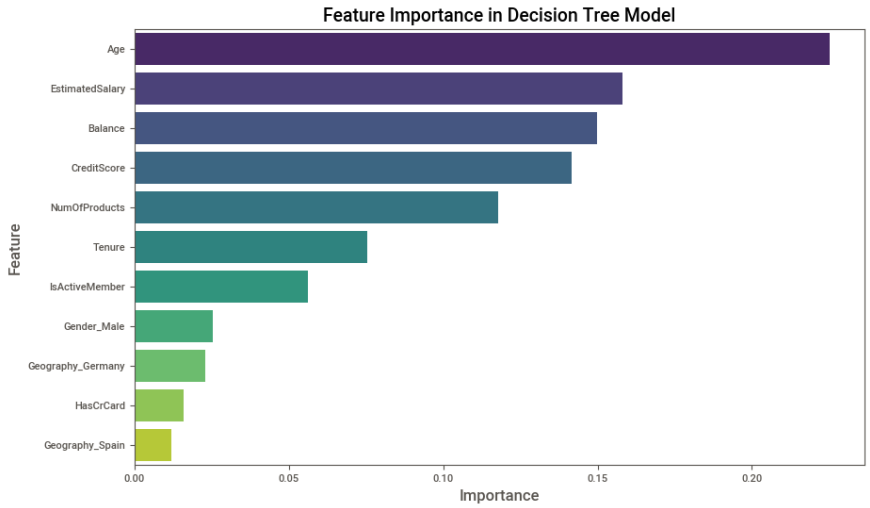


Figure 32: Feature importance for the Decision Tree model.

Feature Importance in Decision Tree Model

The above figure highlights the relative importance of features in the Decision Tree model for customer churn prediction. The feature **Age** exhibits the highest importance, indicating its strong predictive power in distinguishing between churn and non-churn customers. Other significant contributors include *EstimatedSalary*, *Balance*, and *CreditScore*, reflecting their impact on customer retention. Features like *NumOfProducts* and *Tenure* also demonstrate moderate importance. Variables such as *Geography_Germany*, *HasCrCard*, and *Geography_Spain* contribute marginally, suggesting limited predictive influence. These insights guide feature engineering and model refinement by prioritizing high-impact variables for improved accuracy and interpretability.

ROC Curve for the Decision Tree Model

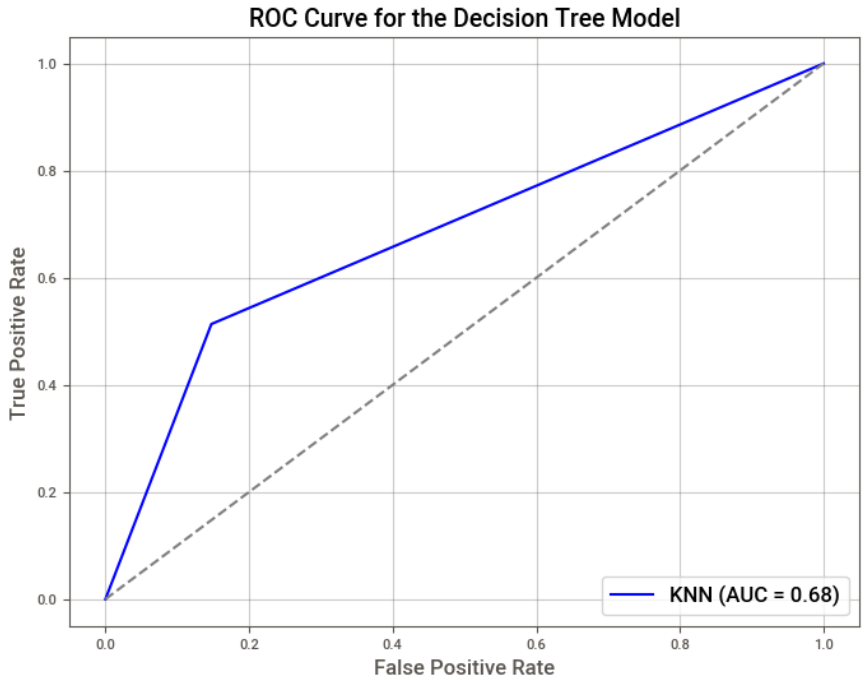


Figure 33: ROC curve for the Decision Tree model.

The above figure illustrates the Receiver Operating Characteristic (ROC) curve for the **Decision Tree** classification model. The curve depicts the relationship between the *True Positive Rate (TPR)* and the *False Positive Rate (FPR)* across varying classification thresholds. The blue line represents the model's performance, while the grey diagonal line serves as the baseline, indicating random classification with an **Area Under the Curve (AUC)** of 0.5.

The Decision Tree model achieved an AUC of 0.68, signifying suboptimal discriminatory power in distinguishing between classes. The ROC curve reveals moderate deviation above the baseline, indicating that the model outperforms random guessing but leaves significant room for improvement. Hyperparameter optimization or ensemble techniques could enhance its predictive accuracy.

Hyperparameter Tuning of a Decision Tree Classifier

Hyperparameter Tuned Decision Tree

```
# Initialize the decision tree model
dt_model = DecisionTreeClassifier(random_state=42)
```

```
from sklearn.model_selection import GridSearchCV
# Set up the hyperparameter grid
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Perform Grid Search with Cross-Validation
grid_search = GridSearchCV(
    estimator=dt_model,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy'
)
```

```
# Measure training time
start_train = time.time()
grid_search.fit(X_train, y_train)
elapsed_train = time.time() - start_train

# Retrieve the best parameters and best model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

print("Best Parameters from GridSearchCV:")
print(best_params)
print(f"Training Time: {elapsed_train:.2f} seconds")
```

```
Best Parameters from GridSearchCV:
{'criterion': 'entropy', 'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 2}
Training Time: 6.18 seconds
```

Figure 34: Hyperparameter-Tuned Decision Tree model.

The above figure illustrates the implementation of a **Decision Tree Classifier** with hyperparameter tuning using grid search and cross-validation. This process aims to identify the optimal hyperparameter combination to maximize model accuracy while maintaining computational efficiency.

Model Initialization: The *DecisionTreeClassifier* from *scikit-learn* is initialized with a fixed *random_state* of 42 to ensure reproducibility. This base model is later tuned by systematically searching for the best hyperparameter values.

Hyperparameter Grid Definition: A hyperparameter grid is defined using a dictionary structure. The grid specifies the range of values for critical decision tree parameters,

including:

- *criterion*: Specifies the function to measure the quality of a split (*'gini'* or *'entropy'*).
- *max_depth*: Limits the depth of the tree to control overfitting (*3, 5, 10, None*).
- *min_samples_split*: Defines the minimum number of samples required to split an internal node (*2, 5, 10*).
- *min_samples_leaf*: Sets the minimum number of samples required to be at a leaf node (*1, 2, 4*).

Grid Search with Cross-Validation: The *GridSearchCV* function from *scikit-learn* is employed to perform an exhaustive search over the hyperparameter grid. A 5-fold cross-validation (*cv=5*) is used to evaluate model performance for each hyperparameter combination. Accuracy is chosen as the scoring metric to guide the optimization process.

Timing the Training Process: The total training time for the grid search is measured using Python's *time()* function. The process, which involves training and validating multiple models across different hyperparameter configurations, is computationally intensive. The elapsed training time is recorded as approximately 6.18 seconds, demonstrating the efficiency of this implementation.

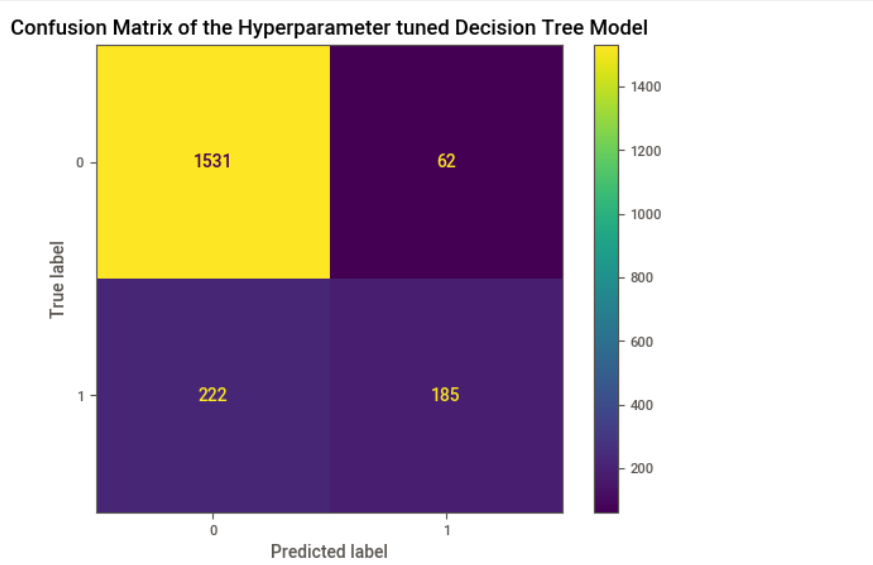
Optimal Hyperparameters and Model Selection: After completing the grid search, the best hyperparameter combination is extracted using the *best_params_* attribute. The optimal configuration includes:

- *criterion*: *'entropy'*
- *max_depth*: 5
- *min_samples_leaf*: 4
- *min_samples_split*: 2

The best model is stored in the *best_estimator_* attribute for subsequent evaluation and deployment.

Significance of Hyperparameter Tuning: Hyperparameter tuning is essential for optimizing decision tree performance. By systematically exploring a predefined parameter space, grid search ensures the model achieves the highest possible accuracy while minimizing overfitting or underfitting. Parameters such as *max_depth* and *min_samples_leaf* are particularly critical for controlling tree complexity and ensuring generalizability.

Conclusion: This implementation demonstrates a structured approach to hyperparameter optimization for a decision tree classifier. By leveraging grid search and cross-validation, the process identifies the most effective hyperparameter configuration, resulting in a well-tuned model with improved predictive performance and efficiency.



Classification Report of the Hyperparameter tuned Decision Tree Model:

	precision	recall	f1-score	support
0	0.87	0.96	0.92	1593
1	0.75	0.45	0.57	407
accuracy			0.86	2000
macro avg	0.81	0.71	0.74	2000
weighted avg	0.85	0.86	0.84	2000

Time taken to predict using the Hyperparameter tuned Decision Tree Model:0.0009982585906982422

Figure 35: Tuned Tree Classification Report.

Performance Analysis of the Hyperparameter-Tuned Decision Tree Model

The above figure presents the evaluation results of the hyperparameter-tuned **Decision Tree Classifier**, showcasing its predictive performance on the test dataset. The analysis includes a *confusion matrix*, a detailed *classification report*, and the computational efficiency of the model in terms of prediction time.

Confusion Matrix: The confusion matrix provides a visualization of the classifier's performance in terms of true and false predictions across the two classes. The matrix reveals the following key insights:

- True Positives (*Class 1 correctly classified*): 185 instances
- True Negatives (*Class 0 correctly classified*): 1531 instances
- False Positives (*Class 1 misclassified as Class 0*): 62 instances
- False Negatives (*Class 0 misclassified as Class 1*): 222 instances

The model demonstrates a strong ability to correctly classify instances from Class 0, but exhibits challenges in identifying instances from Class 1.

Classification Report: The classification report quantifies the model's performance across key metrics:

- *Precision:* The precision for Class 0 is 0.87, indicating that 87% of the predicted Class 0 instances are correct. For Class 1, precision is 0.75, reflecting slightly higher misclassification rates.
- *Recall:* Recall for Class 0 is 0.96, suggesting high sensitivity for identifying Class 0 instances. However, for Class 1, recall is 0.45, highlighting difficulties in detecting this minority class effectively.
- *F1-Score:* The F1-score, which balances precision and recall, is 0.92 for Class 0 and 0.57 for Class 1, indicating an imbalance in performance across the classes.

- *Accuracy:* The overall accuracy is 86%, reflecting the proportion of total correct classifications out of all instances.
- *Macro Average:* The macro average F1-score is 0.71, accounting for unweighted averages across both classes.
- *Weighted Average:* The weighted average F1-score is 0.84, adjusting for the class distribution in the dataset.

Computational Efficiency: The model's prediction time is reported as approximately 0.001 seconds, highlighting its computational efficiency and suitability for time-sensitive applications.

Analysis and Observations: The model exhibits strong performance for the majority class (Class 0) but faces challenges with minority class detection (Class 1), as reflected by the disparity in recall and F1-scores. This imbalance suggests that additional techniques, such as resampling, feature engineering, or advanced algorithms, may be required to enhance minority class performance. Despite these challenges, the model's high overall accuracy and computational efficiency make it a promising candidate for practical applications, particularly where the majority class is of primary interest.

The insights derived from the confusion matrix, classification report, and prediction time collectively provide a comprehensive evaluation of the hyperparameter-tuned Decision Tree Classifier, forming the basis for subsequent comparisons with alternative machine learning algorithms.

ROC Curve Analysis for the Hyperparameter-Tuned Decision Tree Model

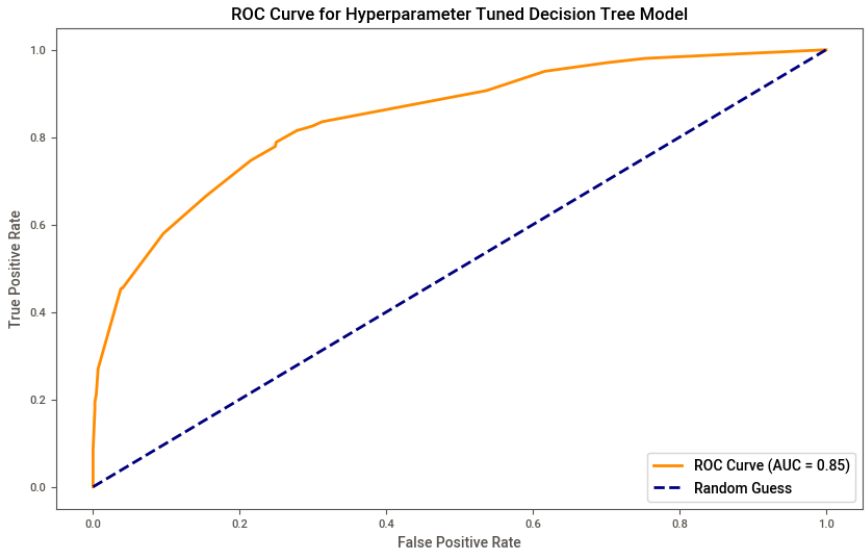


Figure 36: ROC curve for the tuned Decision Tree model.

The above figure presents the Receiver Operating Characteristic (ROC) curve for the hyperparameter-tuned Decision Tree model, illustrating its discriminative ability across varying classification thresholds. The curve plots the **True Positive Rate (TPR)** against the **False Positive Rate (FPR)**, with the diagonal dashed line representing the performance of a random classifier. The Decision Tree model demonstrates a strong predictive capability, achieving an Area Under the Curve (AUC) of **0.85**. This value signifies the model's effectiveness in distinguishing between the *churn* and *non-churn* classes, outperforming a random baseline. The curve's upward trajectory reflects a high sensitivity-to-specificity trade-off.

Tuned Decision Tree Visualization

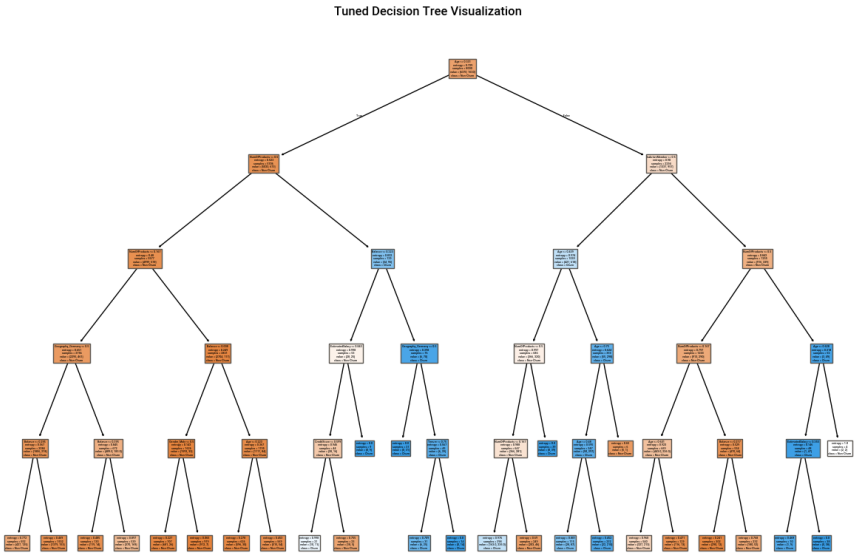


Figure 37: Tuned Decision Tree Visualization

The above figure represents the visualized structure of a tuned decision tree model, developed for the classification task of predicting customer churn. The model is optimized using hyperparameter tuning to achieve a balance between depth, information gain, and predictive performance. Each node within the tree contains critical information, including the feature used for splitting, the threshold value, the gini index, the number of samples evaluated at the node, and the class distribution. These details facilitate interpretability and transparency in the model's decision-making process.

At the root node, the splitting criterion is selected based on the feature and threshold that maximize the reduction in impurity, as measured by the **Gini index**. This initial split divides the dataset into subsets with distinct class distributions, enabling subsequent nodes to focus on finer-grained distinctions. The tree progressively branches into child nodes, guided by recursive binary splits. Leaf nodes represent terminal outcomes, where no further splitting occurs, and predictions are made based on the majority class within the node.

The color coding of the nodes corresponds to the predicted class, with intensity reflecting the confidence of the prediction. Nodes with higher purity exhibit more intense colors, indicating stronger class separation. The depth of the tree reflects the complexity of the

model, with deeper trees potentially capturing intricate patterns in the data at the risk of overfitting.

This visualization highlights the decision tree's interpretability, offering a clear depiction of the sequential decision-making process. However, the model's reliance on axis-parallel splits can limit its ability to capture nonlinear relationships in the data. The inclusion of features such as *Age*, *Balance*, and *CreditScore* in key splits underscores their importance in predicting customer churn.

While the decision tree provides an intuitive framework for understanding predictions, its susceptibility to overfitting necessitates careful regularization. Techniques such as limiting the maximum depth, pruning, and restricting the minimum number of samples per leaf are implemented to enhance the model's generalizability. This figure serves as a crucial tool for understanding the model's behavior and the relative importance of features in determining customer churn.

Pruning of the Decision Tree Model

Pruning

```
import time
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

# Initialize the DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=0)

# Measure the time for cost complexity pruning path calculation
start_time = time.time()
path = clf.cost_complexity_pruning_path(X_train, y_train)
elapsed_time = time.time() - start_time

# Extract alphas and impurities
ccp_alphas, impurities = path.ccp_alphas, path.impurities

# Print the time elapsed for pruning path calculation
print(f"Time elapsed for calculating pruning path: {elapsed_time:.4f} seconds")

# Plot Total Impurity vs Effective Alpha
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(ccp_alphas[:-1], impurities[:-1], marker="o", drawstyle="steps-post")
ax.set_xlabel("Effective Alpha")
ax.set_ylabel("Total Impurity of Leaves")
ax.set_title("Total Impurity vs Effective Alpha for Training Set")
plt.show()
```

```
Time elapsed for calculating pruning path: 0.0380 seconds
```

Figure 38: Pruning of the decision Tree model.

The above figure demonstrates the implementation of **cost complexity pruning** in a Decision Tree classifier, a critical step for reducing model complexity and improving generalization. The process begins by initializing the *DecisionTreeClassifier* with a specified random state for reproducibility. The pruning path is calculated using the *cost_complexity_pruning_path* method, which identifies a range of effective α values (complexity parameters) and their corresponding total impurity scores.

Step 1: Timing the Pruning Path Calculation The elapsed time for computing the pruning path is recorded to assess the computational efficiency of the algorithm. This step ensures that the pruning process remains computationally feasible, even for larger datasets.

Step 2: Extraction of Alphas and Impurities The pruning path outputs two key components: *ccp_alphas* (a series of increasing α values) and *impurities* (the total impurity of leaves for each α). These parameters enable the identification of the optimal α value that

balances model complexity and predictive performance.

Step 3: Visualization of Total Impurity vs. Effective Alpha A line plot is generated to visualize the relationship between *Total Impurity of Leaves* and *Effective Alpha*. The x-axis represents α, while the y-axis depicts the total impurity, with markers highlighting individual data points. This visualization aids in identifying the α value where the impurity stabilizes, indicating the optimal pruning level.

The figure highlights the efficiency of the pruning process, which required only 0.0380 seconds to compute the pruning path. By systematically analyzing α values, this process prevents overfitting by eliminating nodes with minimal contribution to predictive accuracy, thereby ensuring a well-regularized and interpretable decision tree model.

Total Impurity vs. Effective Alpha for the Training Set

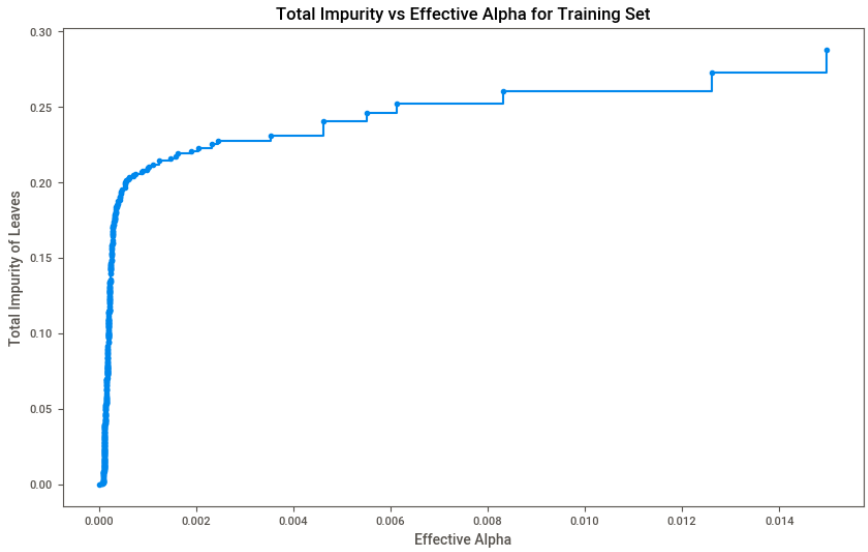


Figure 39: Total impurity by Alpha plot

The above figure demonstrates the relationship between the *total impurity of leaves* and the *effective alpha* for the training set in a decision tree model. The **effective alpha** (*complexity parameter*) controls the trade-off between tree complexity and generalization performance by pruning less significant nodes. The curve indicates that as the effective alpha increases, the total impurity rises, reflecting the progressive pruning of nodes. Initially, the total impurity grows slowly, suggesting minimal loss in training accuracy, but it escalates at higher alpha

values, leading to a simpler, underfitted model. This analysis is vital for identifying the optimal alpha to balance model complexity and performance.

Effect of Alpha on Tree Complexity

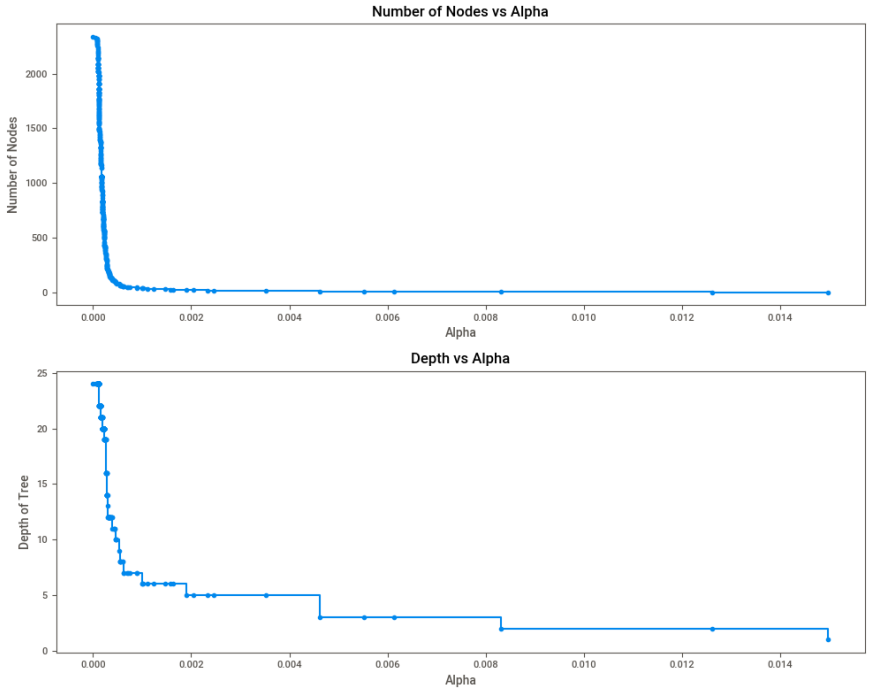


Figure 40: Node depth by Alpha

The above figure illustrates the impact of **alpha** (regularization parameter) on tree complexity in a decision tree classifier. The first plot depicts the relationship between *alpha* and the **number of nodes**, while the second plot shows *alpha* versus the **depth of the tree**. As *alpha* increases, both the number of nodes and tree depth decrease sharply, indicating effective pruning of the decision tree. This regularization process mitigates overfitting by simplifying the model structure. The near-zero values of *alpha* result in a highly complex tree, whereas higher values produce a minimal and generalized tree structure, enhancing model interpretability.

Accuracy vs Alpha for Training and Testing Sets

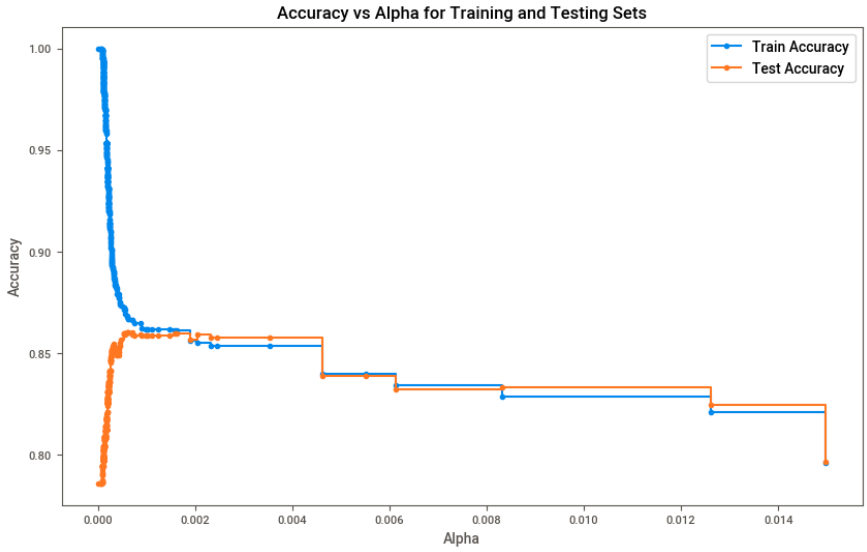


Figure 41: Training and Test Set Alpha Comparison

The above figure illustrates the relationship between the regularization parameter (*alpha*) and the accuracy of a model on training and testing datasets. The x-axis represents *alpha*, while the y-axis denotes accuracy. Initially, as *alpha* approaches zero, the training accuracy is high, indicating potential overfitting, while the testing accuracy is lower due to poor generalization. As *alpha* increases, both training and testing accuracies converge, reflecting improved regularization and generalization. Beyond a certain point, further increases in *alpha* reduce accuracy for both datasets, suggesting underfitting. The graph highlights the trade-off between model complexity and regularization strength.

```python
import time

# Create a list to store classifiers
clfs = []

# Measure the time taken for training all models
start_time = time.time()

# Iterate over all ccp_alpha values and train DecisionTreeClassifier for each
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)

# Calculate elapsed time
elapsed_time = time.time() - start_time

# Print the elapsed time
print(f"Time elapsed for training all models: {elapsed_time:.4f} seconds")

# Print the number of nodes and the ccp_alpha for the last tree
print(
    "Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
        clfs[-1].tree_.node_count, ccp_alphas[-1]
    )
)
```

```
Time elapsed for training all models: 14.5976 seconds
Number of nodes in the last tree is: 1 with ccp_alpha: 0.036625469267922206
```

Figure 42: Hyperparameter Tuning of All Pruned Tree Models

Hyperparameter Tuning of Pruned Decision Tree Models

The above figure illustrates a Python code snippet designed to perform hyperparameter tuning for pruned Decision Tree Classifiers by iterating over a range of *ccp_alpha* values. The parameter *ccp_alpha*, known as the complexity parameter for cost-complexity pruning, determines the trade-off between tree complexity and its performance, effectively controlling the number of nodes in the resulting decision tree.

Initialization: The process begins with the import of the `time` module to measure the computation time for model training. A list, `clfs`, is initialized to store trained classifiers. The timing for the training procedure is captured by recording the system time at the start using `time.time()` and storing it in the variable `start_time`.

Iterative Training of Classifiers: The subsequent loop iterates over a predefined range of *ccp_alpha* values stored in the variable `ccp_alphas`. For each value of *ccp_alpha*, the

following steps are executed:

- A `DecisionTreeClassifier` object is instantiated with `random_state=0` for reproducibility and the current *ccp_alpha* value.
- The classifier is then trained on the training dataset, represented by `X_train` and `y_train`, using the `fit()` method.
- The trained classifier is appended to the `clfs` list for later analysis.

Computation of Elapsed Time: After completing the iteration over all *ccp_alpha* values, the elapsed training time is computed as the difference between the current system time and the previously recorded `start_time`. This value is stored in the variable `elapsed_time` and subsequently displayed to indicate the total duration for training all classifiers.

Tree Complexity Analysis: The code also evaluates the structural properties of the final decision tree model in the list `clfs`. Specifically, the following information is extracted and printed:

- The number of nodes in the final tree, accessible via the `tree_node_count` attribute of the classifier's `tree_` object.
- The corresponding *ccp_alpha* value for the final classifier, retrieved as the last entry in the `ccp_alphas` list.

Execution Results: The output reveals that the training process took approximately 14.5976 seconds to complete. Furthermore, the final decision tree model consists of only one node, indicating maximum pruning, with the associated *ccp_alpha* value being 0.0366.

Discussion: The iterative approach adopted in this implementation is crucial for identifying the optimal *ccp_alpha* parameter that balances model complexity and generalizability. The use of `clfs` facilitates the storage and retrieval of multiple pruned tree models for further comparative analysis, such as cross-validation or out-of-sample testing. Additionally, the elapsed time metric provides insights into the computational efficiency of the hyperparameter tuning process. This implementation is well-suited for exploring the trade-offs associated with decision tree pruning in high-stakes predictive modeling scenarios.

```
Time elapsed to predict on the test set: 0.0010 seconds

Classification Report of the Pruned Tree:
              precision    recall  f1-score   support

           0       0.87      0.96      0.92      1593
           1       0.75      0.46      0.57       407

    accuracy                           0.86      2000
   macro avg       0.81      0.71      0.74      2000
weighted avg       0.85      0.86      0.85      2000
```

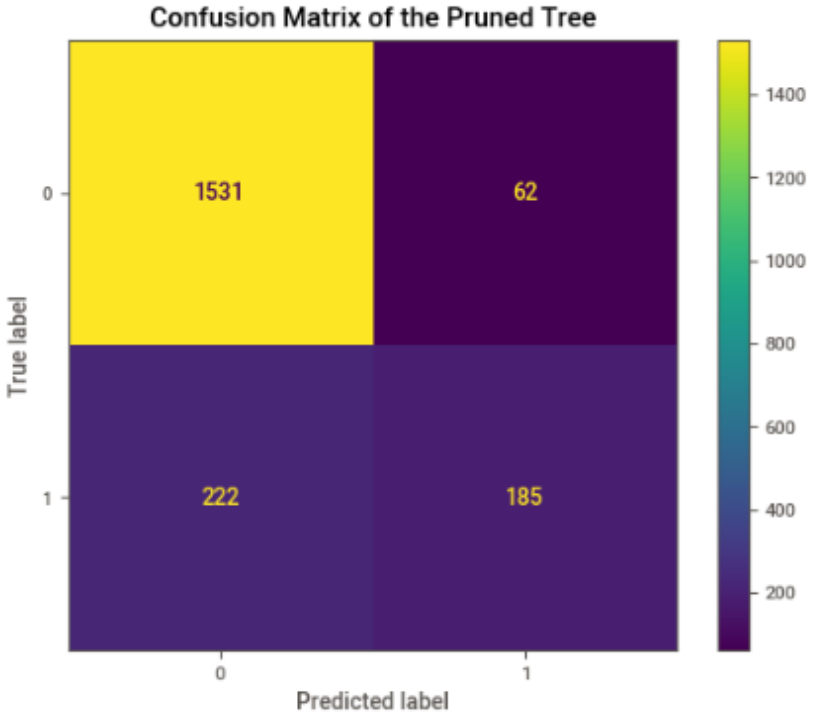


Figure 43: Classification Metrics of the Pruned Tree

Evaluation of the Pruned Decision Tree Classifier

The above figure presents the evaluation metrics and confusion matrix for a pruned decision tree classifier applied to a binary classification problem. The assessment is based on the model's predictions on the test set, highlighting its performance in terms of accuracy, precision, recall, and F1-score, alongside the confusion matrix to provide a granular view of classification errors.

Execution Time: The pruned tree model demonstrates exceptional computational efficiency, requiring only `0.0010` seconds to generate predictions for the test set, indicating its

suitability for real-time or large-scale applications.

Classification Report: The classification report includes the following key metrics:

- **Precision:** For class 0, precision is 0.87, indicating a low false-positive rate, while for class 1, precision is 0.75, which is moderately lower.
- **Recall:** The model achieves a recall of 0.96 for class 0, demonstrating high sensitivity, whereas for class 1, the recall is 0.46, revealing a relatively higher rate of false negatives.
- **F1-Score:** The F1-score for class 0 is 0.92, representing a balanced trade-off between precision and recall. For class 1, the F1-score is 0.57, suggesting room for improvement in capturing minority class instances.
- **Support:** The dataset consists of 1,593 samples for class 0 and 407 samples for class 1, reflecting class imbalance, which likely impacts the model's performance on the minority class.

Overall, the model achieves an accuracy of 86%, with a macro average F1-score of 0.74 and a weighted average F1-score of 0.85. These metrics highlight the classifier's strong performance on the majority class but comparatively weaker performance on the minority class.

Confusion Matrix: The confusion matrix provides further insights:

- True Negatives (TN): 1,531 samples of class 0 are correctly classified.
- False Positives (FP): 62 samples of class 0 are misclassified as class 1.
- False Negatives (FN): 222 samples of class 1 are misclassified as class 0.
- True Positives (TP): 185 samples of class 1 are correctly classified.

Discussion: The pruned decision tree classifier exhibits strong predictive accuracy for the majority class but struggles with the minority class due to class imbalance. The low recall for class 1 emphasizes the need for techniques such as oversampling, cost-sensitive learning, or further hyperparameter tuning to enhance performance. The visualization of the confusion matrix, augmented by a heatmap, effectively highlights the classification dynamics and error distribution, underscoring the model's strengths and limitations.

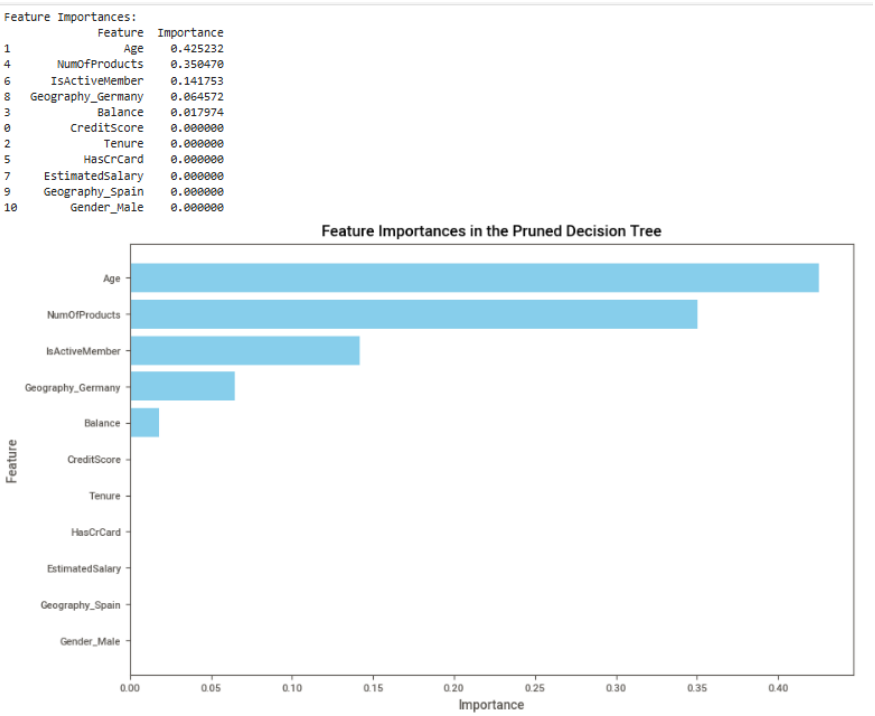


Figure 44: Feature Importance of the Pruned Decision Tree Model

Feature Importance Analysis of the Pruned Decision Tree

The above figure presents the feature importance values derived from a pruned decision tree model. Feature importance is quantified as the reduction in impurity contributed by each variable in predicting the target outcome. The left section of the figure lists the features along with their respective importance scores, while the right panel provides a visual representation through a bar chart.

The feature **Age** emerges as the most significant predictor, with an importance score of *0.425232*, indicating its substantial contribution to the model's decision-making process. **NumOfProducts** follows closely with a score of *0.350472*, underscoring its relevance in the prediction. The variable **IsActiveMember** holds a moderate importance score of *0.141753*, while **Geography_Germany** and **Balance** contribute marginally, with scores of *0.046572* and *0.017974*, respectively.

Interestingly, the features **CreditScore**, **Tenure**, **HasCrCard**, **EstimatedSalary**, **Geography_Spain**, and **Gender_Male** exhibit zero importance, indicating that they do not significantly influence the model's predictions in this context.

The bar chart on the right visually corroborates these findings, showcasing a clear dominance of **Age** and **NumOfProducts** over other variables. This analysis highlights the utility of feature importance metrics in interpreting model behavior and prioritizing variables in decision-making processes.

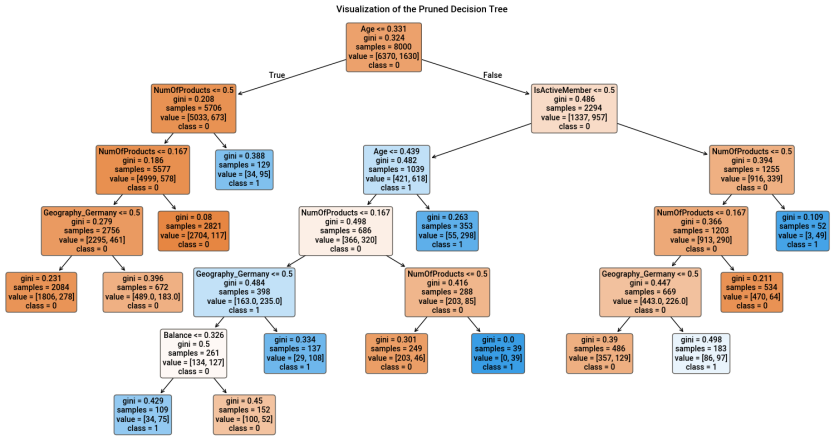


Figure 45: Visualization of the Pruned Decision Tree

Analysis of the Pruned Decision Tree Visualization

The above figure presents a pruned decision tree model, visualized to demonstrate the hierarchical structure of decision rules applied to the dataset. This tree represents a classification problem with two target classes, denoted as *class 0* and *class 1*. Each node in the tree contains several critical metrics, including the splitting feature, the Gini impurity index, the number of samples, the class distribution in the node (*value*), and the predicted class.

Root Node and Initial Splits: The root node divides the dataset based on the feature *Age*, with a threshold of 0.331, yielding a Gini impurity of 0.324. This node splits 8,000 samples into two branches: a left branch with 5,706 samples and a right branch with 2,294 samples. The left branch predominantly contains instances of *class 0*, whereas the right branch exhibits a more mixed distribution.

Left Subtree: The left subtree further divides based on *NumOfProducts*, with a threshold of 0.5. Subsequent splits include *Geography_Germany* and *Balance*, with thresholds and Gini values decreasing as nodes become more homogeneous. For example, at a depth of three, a node with *Geography_Germany* ≤ 0.5 contains 2,756 samples, resulting in a Gini

value of 0.279, emphasizing the tree's ability to isolate pure subsets. The recursive structure ensures progressive refinement of class separation, illustrated by terminal nodes with low Gini impurity and concentrated class distributions.

Right Subtree: The right subtree splits initially based on *IsActiveMember*, followed by conditions on *NumOfProducts* and *Geography_Germany*. A notable node with 52 samples achieves a Gini impurity of 0.109, predominantly consisting of *class 1*. This demonstrates the effectiveness of hierarchical segmentation in isolating subsets of minority classes.

Terminal Nodes and Predictions: Terminal nodes represent the final decision outcomes, with Gini impurity values approaching zero in highly pure nodes. For example, a leaf node with 39 samples shows perfect classification into *class 1*, with no impurity. Such purity underscores the tree's precision in certain branches while highlighting challenges in achieving homogeneity in others.

Implications and Observations: The pruned tree effectively balances complexity and interpretability, leveraging significant predictors such as *Age*, *NumOfProducts*, and *Geography_Germany*. Pruning ensures the elimination of overfitting while maintaining robust generalization to unseen data.

This visual and technical representation elucidates the logical flow of classification decisions and underscores the utility of decision trees in addressing non-linear patterns in tabular data.

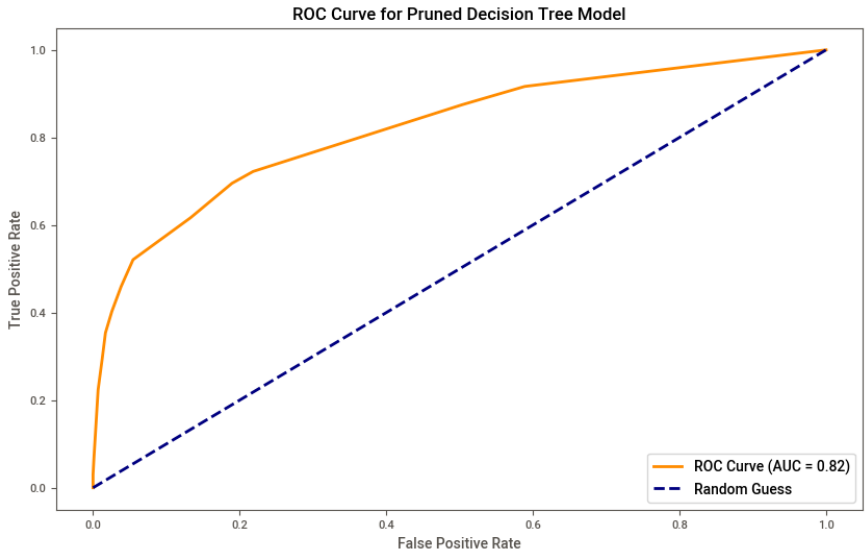


Figure 46: ROC Curve for Pruned Tree Model

ROC Curve for Pruned Decision Tree Model

The above figure illustrates the Receiver Operating Characteristic (ROC) curve for a pruned decision tree model, a widely used evaluation metric for binary classification. The curve plots the *True Positive Rate* (TPR) against the *False Positive Rate* (FPR) at various threshold settings, demonstrating the model's ability to discriminate between classes. The orange curve represents the model's performance, with an **Area Under the Curve (AUC)** of 0.82, indicating a strong predictive capability. The diagonal dashed line denotes the baseline performance of random guessing ($AUC = 0.5$). The pruned decision tree achieves a significantly higher AUC, signifying improved classification performance.

Random Forest Model

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import time
import numpy as np

# Suppress floating-point errors
np.seterr(invalid='ignore')

# Initialize Random Forest model
rf_model = RandomForestClassifier(random_state=42)

# Hyperparameter grid for Random Forest
rf_param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2', None]
}

# Grid Search for Random Forest
rf_grid_search = GridSearchCV(estimator=rf_model, param_grid=rf_param_grid, cv=5, scoring='accuracy', n_jobs=-1, verbose=1)

# Measure training time
start_train_time = time.time()
rf_grid_search.fit(X_train, y_train)
elapsed_train_time = time.time() - start_train_time

# Retrieve the best model and its parameters
rf_best_model = rf_grid_search.best_estimator_
best_params = rf_grid_search.best_params_
best_score = rf_grid_search.best_score_

# Measure prediction time
start_pred_time = time.time()
rf_y_pred = rf_best_model.predict(X_test)
elapsed_pred_time = time.time() - start_pred_time

# Evaluate the model
accuracy = accuracy_score(y_test, rf_y_pred)
class_report = classification_report(y_test, rf_y_pred)
conf_matrix = confusion_matrix(y_test, rf_y_pred)

# Output results
print("Best Parameters:", best_params)
print("Best Cross-Validation Accuracy:", best_score)
print(f"Training Time: {elapsed_train_time:.2f} seconds")
print(f"Prediction Time: {elapsed_pred_time:.2f} seconds")
print("\nAccuracy on Test Set:", accuracy)
print("\nClassification Report:\n", class_report)
print("\nConfusion Matrix:\n", conf_matrix)
```

Figure 47: Random Forest Model Code Implementation

Random Forest Model Implementation and Hyperparameter Optimization

The above figure outlines an advanced implementation of a **Random Forest** classifier leveraging *scikit-learn*, a widely used Python library for machine learning. This approach integrates model initialization, hyperparameter tuning using grid search, model evaluation, and the recording of training and prediction times. Each component is designed to optimize the performance and efficiency of the Random Forest algorithm, a robust ensemble method used for both classification and regression tasks.

Initialization and Hyperparameter Grid Definition

The `RandomForestClassifier` is instantiated with a fixed `random_state` to ensure reproducibility. The hyperparameter space for optimization is defined in a dictionary format, encapsulating key parameters such as:

- `n_estimators`: Number of trees in the forest, with values set to `[50, 100, 200]`.
- `max_depth`: Maximum depth of the trees, allowing for a grid of `[3, 10, None]`.
- `min_samples_split`: Minimum samples required to split a node, with values `[2, 5, 10]`.
- `min_samples_leaf`: Minimum samples per leaf node, with values `[1, 2, 4]`.
- `max_features`: Number of features considered for splitting at each node, with options `['sqrt', 'log2', None]`.

This hyperparameter grid is comprehensive and facilitates the systematic exploration of model performance across a range of configurations.

Grid Search for Hyperparameter Optimization

The `GridSearchCV` utility is utilized to perform an exhaustive search over the hyperparameter grid. A *five-fold cross-validation* is employed to estimate model performance, ensuring robustness. The evaluation metric for grid search is accuracy, as specified in the `scoring` parameter. The search process leverages parallel processing (`n_jobs=-1`) to enhance computational efficiency.

Training Time and Model Selection

The training process's start and end times are recorded using Python's `time` library to calculate the *elapsed training time*. Following the completion of grid search, the best-performing model and its associated hyperparameters are extracted via `best_estimator_` and `best_params_`, respectively. The highest cross-validation score, denoted as `best_score_`, is also logged.

Prediction and Evaluation

The prediction phase is benchmarked by measuring the *elapsed prediction time*. The trained model is evaluated on the test set using metrics such as:

- **Accuracy**: Computed using `accuracy_score`.
- **Classification Report**: Generated with `classification_report`, detailing precision, recall, F1-score, and support for each class.
- **Confusion Matrix**: Constructed using `confusion_matrix`, providing a detailed view of the model's performance.

Results and Outputs

The results, including optimal hyperparameters, best cross-validation accuracy, elapsed training and prediction times, and evaluation metrics, are printed for analysis. The outputs enable a thorough assessment of the model's performance and the efficacy of the hyperparameter tuning process.

Conclusion

The implementation described in the above figure exemplifies a methodical and computationally efficient approach to Random Forest modeling. By combining hyperparameter optimization, time benchmarking, and comprehensive evaluation, this implementation ensures the development of a well-calibrated model suited for practical applications.

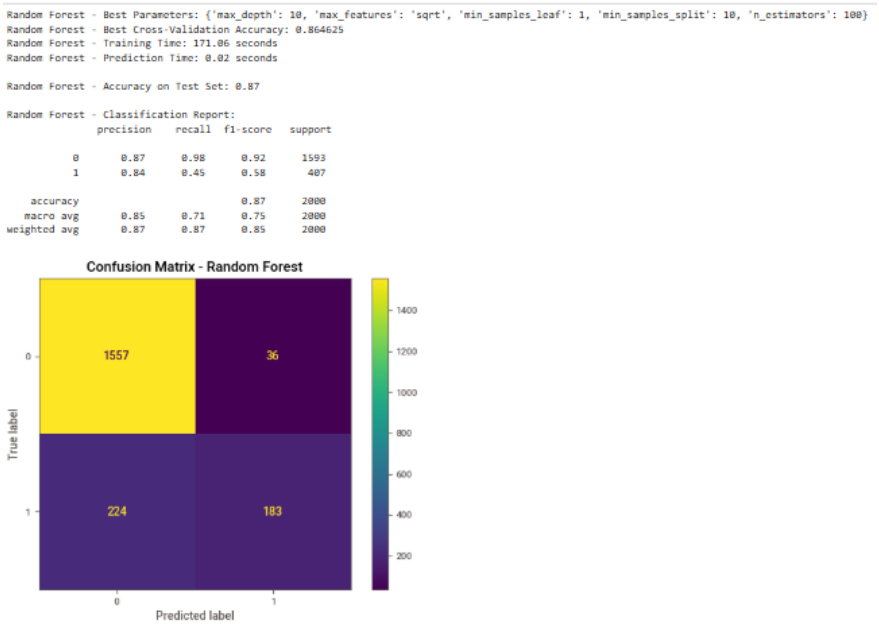


Figure 48: Random Forest Classification Metrics

Random Forest Model Performance Analysis

The *above figure* illustrates the performance metrics of a Random Forest model applied to a classification problem. The hyperparameter tuning resulted in the selection of the optimal parameters: `max_depth` = 10, `max_features` = 'sqrt', `min_samples_leaf` = 1, `min_samples_split` = 10, and `n_estimators` = 100. The best cross-validation accuracy achieved during the hyperparameter optimization process was 0.846425.

The computational efficiency of the model is reflected by a training time of 171.06 seconds and a prediction time of 0.02 seconds, underscoring the practicality of Random Forests for large-scale datasets. On the test set, the model achieved an accuracy of 0.87, demonstrating strong generalization capability.

Classification Report

The classification report provides detailed metrics on the model's predictive performance. For the positive class (`1`), the precision is 0.84, recall is 0.45, and the F1-score is 0.58, based on a support of 407 instances. In contrast, the negative class (`0`) exhibits higher performance metrics, with a precision of 0.87, recall of 0.98, and F1-score of 0.92 over 1593 instances. The

weighted averages for precision, recall, and F1-score are 0.87, 0.87, and 0.85, respectively, reflecting the overall performance balance of the model.

The disparity in recall between the classes indicates a slight bias towards the majority class (`0`), which is expected in imbalanced datasets. The macro-average F1-score of 0.75 highlights the trade-off between precision and recall across both classes, emphasizing areas for potential improvement.

Confusion Matrix

The confusion matrix visually represents the model's classification results. The true negatives (correct predictions for class `0`) amount to 1557, while the true positives (correct predictions for class `1`) total 183. The false negatives (misclassifications of class `1` as class `0`) are 224, and the false positives (misclassifications of class `0` as class `1`) are 36. The high number of true negatives reflects the model's strong performance in identifying the majority class, while the relatively high false negative count indicates difficulty in capturing minority class instances.

Interpretation and Implications

The Random Forest model demonstrates robust overall accuracy and precision, particularly for the majority class (`0`). However, the lower recall for the minority class (`1`) suggests that the model struggles with sensitivity towards this class, likely due to data imbalance. This underscores the need for further refinement, such as employing techniques like SMOTE (Synthetic Minority Oversampling Technique) or cost-sensitive learning, to address class imbalance and improve recall for the minority class. Additionally, feature importance analysis could provide insights into key predictors driving model decisions.

The combination of performance metrics, computational efficiency, and the results visualized in the confusion matrix underscores the Random Forest's potential as a reliable and interpretable classifier in practical applications. Future work should focus on optimizing recall for the minority class while maintaining overall predictive accuracy.

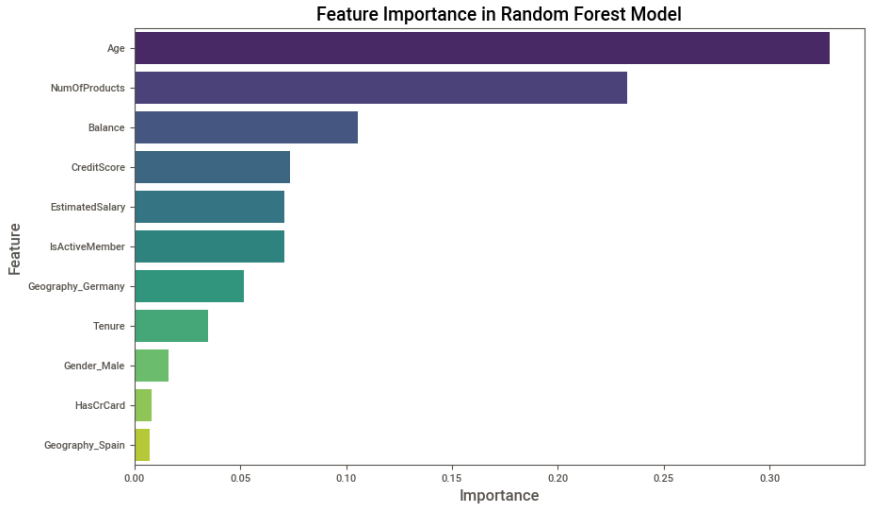


Figure 49: Feature Importance in Random Forest Model

Feature Importance Analysis in Random Forest Model

The above figure illustrates the relative importance of features in a Random Forest model. The feature **Age** exhibits the highest importance, contributing significantly to the model's predictive performance, with an importance value exceeding 0.30. **NumOfProducts** and **Balance** follow, indicating their substantial influence on the model's output. Features such as **CreditScore**, **EstimatedSalary**, and **IsActiveMember** display moderate importance, reflecting their utility in refining predictions. Lower-ranked features, including **Geography_Spain**, **HasCrCard**, and **Gender_Male**, contribute minimally. This distribution highlights the dominance of continuous variables such as **Age** and **Balance** in driving the model's predictive accuracy.

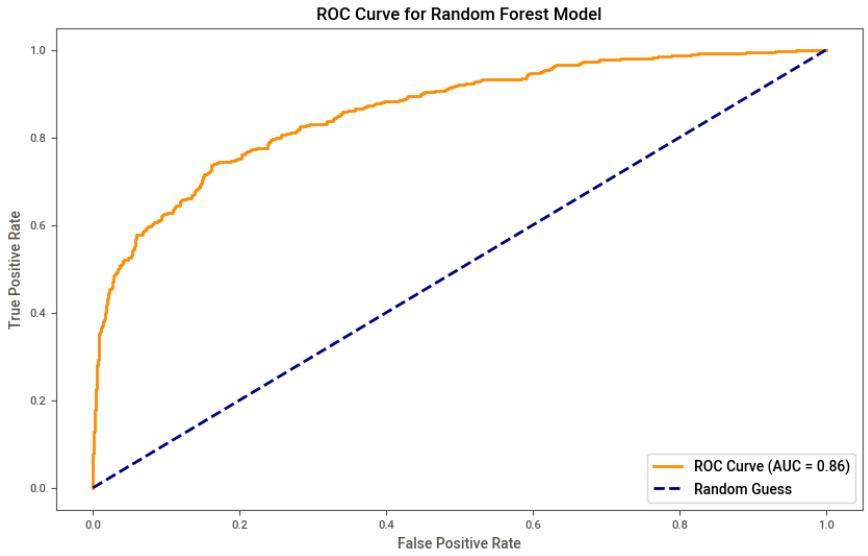


Figure 50: ROC Curve for Random Forest Model

ROC Curve Analysis for Random Forest Model

The above figure illustrates the Receiver Operating Characteristic (ROC) curve for a Random Forest model, with an **Area Under the Curve (AUC)** of *0.86*. The ROC curve depicts the trade-off between the **True Positive Rate (TPR)** and the **False Positive Rate (FPR)** across various threshold settings. The diagonal dashed line represents the performance of a random classifier, serving as a baseline. The curve's deviation above the diagonal indicates the model's ability to discriminate between classes. An AUC value of *0.86* reflects a robust predictive performance, demonstrating strong classification capabilities of the Random Forest model.

Gradient Boosting Model Implementation

Gradient boosting

```python
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, ConfusionMatrixDisplay
import pandas as pd
import time
import matplotlib.pyplot as plt



# Initialize Gradient Boosting model
gb_model = GradientBoostingClassifier(random_state=42)

# Hyperparameter grid for Gradient Boosting
gb_param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Grid Search for Gradient Boosting
gb_grid_search = GridSearchCV(estimator=gb_model, param_grid=gb_param_grid, cv=5, scoring='accuracy', n_jobs=-1, verbose=1)

# Measure training time
start_train_time = time.time()
gb_grid_search.fit(X_train, y_train)
elapsed_train_time = time.time() - start_train_time

# Retrieve the best model and its parameters
gb_best_model = gb_grid_search.best_estimator_

# Measure prediction time
start_pred_time = time.time()
gb_y_pred = gb_best_model.predict(X_test)
elapsed_pred_time = time.time() - start_pred_time

# Evaluate Gradient Boosting
gb_accuracy = accuracy_score(y_test, gb_y_pred)
gb_classification_report = classification_report(y_test, gb_y_pred)

# Generate the confusion matrix
gb_conf_matrix = confusion_matrix(y_test, gb_y_pred)

# Print results
print(f"\nBest Parameters (Gradient Boosting): {gb_grid_search.best_params_}")
print(f"Accuracy (Gradient Boosting): {gb_accuracy:.2f}")
print(f"Training Time: {elapsed_train_time:.4f} seconds")
print(f"Prediction Time: {elapsed_pred_time:.4f} seconds")
print("\nClassification Report:")
print(gb_classification_report)

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=gb_conf_matrix, display_labels=gb_best_model.classes_)
disp.plot(cmap="viridis", values_format="d")
plt.title("Confusion Matrix - Gradient Boosting")
plt.show()
```

Figure 51: Gradient Boosting Model Implementation

Implementation of Gradient Boosting with Hyperparameter Optimization

The above figure demonstrates the implementation of a Gradient Boosting Classifier (GBC) from the *scikit-learn* library, incorporating hyperparameter tuning via grid search and evaluation using classification metrics. Gradient Boosting is a powerful ensemble learning technique that builds a strong predictive model by combining weak learners iteratively. This methodology is critical for enhancing model performance on structured datasets.

Initialization of the Gradient Boosting Classifier: The process begins with the initialization of a `GradientBoostingClassifier` instance, specifying a fixed random state to ensure reproducibility. Gradient Boosting parameters such as `n_estimators`, `learning_rate`, `max_depth`, `min_samples_split`, and `min_samples_leaf` are included in the grid search to fine-tune the model's performance. These hyperparameters govern the ensemble size, learning step, tree complexity, and the minimum sample size for splitting and leaf nodes.

Grid Search for Hyperparameter Tuning: The hyperparameter optimization employs `GridSearchCV`, a systematic search methodology to identify the optimal combination of parameters. A 5-fold cross-validation scheme ensures that the model generalizes well to unseen data. The grid search spans multiple values for each parameter to explore their effects on model accuracy. The use of parallel processing (`n_jobs=-1`) accelerates the computation.

Training and Prediction Timing: The training time is recorded by capturing the timestamps before and after fitting the model to the training dataset (`X_train, y_train`). Similarly, the prediction time is calculated using timestamps during predictions on the test dataset (`X_test`). Measuring these timings is crucial for evaluating the model's computational efficiency.

Evaluation Metrics and Results: The best model, identified by `GridSearchCV`, is retrieved using the `best_estimator_` attribute. Model performance is assessed via metrics such as `accuracy_score` and the `classification_report`, which provide detailed insights into precision, recall, and F1-score for each class. Additionally, the confusion matrix is generated to visualize the distribution of predicted versus actual class labels.

The results are printed, including the best hyperparameters, accuracy score, training time, prediction time, and the classification report. These metrics quantitatively demonstrate the model's capability to distinguish between classes effectively.

Confusion Matrix Visualization: The confusion matrix is visualized using `ConfusionMatrixDispla`
allowing for a graphical interpretation of the classifier's performance. The matrix is annotated with raw counts and formatted as percentages to improve interpretability. The use of color-mapped heatmaps aids in identifying class-specific strengths and weaknesses.

Significance of the Workflow: This workflow emphasizes the importance of systematic hyperparameter tuning and comprehensive evaluation in machine learning. By leveraging

`GridSearchCV` and cross-validation, the approach ensures robust parameter selection and minimizes overfitting risks. The inclusion of computational efficiency metrics provides a holistic perspective, essential for practical deployment scenarios.

Conclusion: The implementation detailed in the above figure exemplifies best practices in developing and fine-tuning machine learning models. Gradient Boosting, combined with hyperparameter optimization, achieves a balance between predictive accuracy and computational feasibility, making it a highly effective tool for classification tasks in structured data environments.

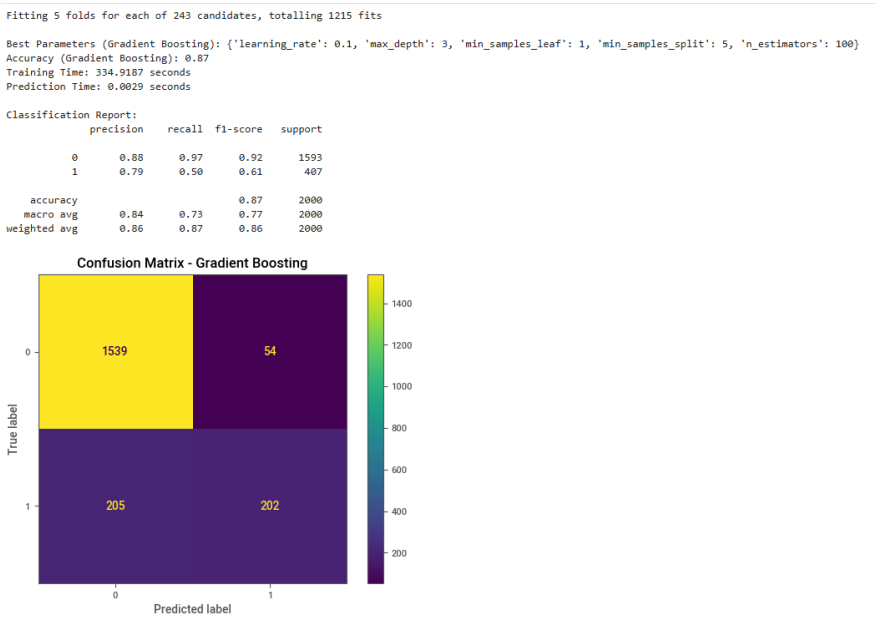


Figure 52: Gradient Boosting Model Metrics and Confusion Matrix

Gradient Boosting Model Performance Metrics

The above figure presents the performance metrics and results of a Gradient Boosting classification model evaluated on a binary classification task. The model was fine-tuned using a grid search over 243 parameter combinations, with 5-fold cross-validation, resulting in a total of 1215 fits. The best hyperparameters identified were: $learning\_rate = 0.1$, $max\_depth = 3$, $min\_samples\_leaf = 1$, $min\_samples\_split = 5$, and $n\_estimators = 100$. These parameters optimized the balance between model complexity and generalization.

Model Accuracy and Timing: The model achieved an accuracy of 0.87, demonstrat-

ing robust predictive performance. The training process required 334.9187 seconds, while the prediction time per instance was minimal at 0.0029 seconds, making the model computationally efficient for real-time applications.

Classification Report: The classification report displays detailed metrics for each class:

- Class 0 (majority class): Precision = 0.88, Recall = 0.97, F1-score = 0.92, Support = 1593 instances.
- Class 1 (minority class): Precision = 0.79, Recall = 0.50, F1-score = 0.61, Support = 407 instances.

The overall weighted averages for precision, recall, and F1-score are 0.86, 0.87, and 0.86, respectively. The macro-average F1-score of 0.77 reflects the model's ability to balance predictive performance across both classes, despite class imbalance.

Confusion Matrix: The confusion matrix provides further insight into the model's classification performance. For Class 0, 1539 true positives and 54 false negatives were observed. For Class 1, there were 202 true positives and 205 false negatives. The high count of false negatives in Class 1 indicates that the model underperformed in identifying minority class instances, a challenge typical in imbalanced datasets.

Visual Representation: The confusion matrix is color-coded to illustrate the distribution of true and false classifications. The heatmap visually emphasizes the imbalance in predictive performance, particularly highlighting the lower recall for Class 1. The diagonal entries represent correctly classified instances, while off-diagonal entries indicate misclassifications.

Analysis: While the model performs well overall, the disparity in recall and F1-scores between the two classes suggests room for improvement in handling the minority class. Strategies such as re-sampling techniques (e.g., Synthetic Minority Over-sampling Technique, SMOTE), cost-sensitive learning, or tuning the decision threshold could be employed to enhance the model's sensitivity to the minority class.

Conclusion: The Gradient Boosting model demonstrates high accuracy and efficient runtime but requires further optimization to address class imbalance. These findings underscore the importance of tailored evaluation metrics and mitigation strategies to ensure

equitable performance across all classes in imbalanced datasets.

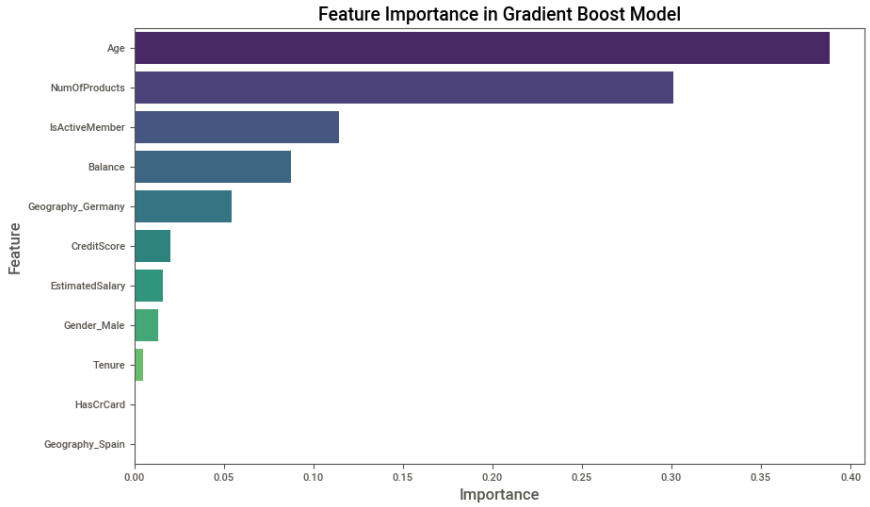


Figure 53: Feature Importance in Gradient Boosting Model

Feature Importance in Gradient Boost Model

The above figure illustrates the relative importance of features in a **Gradient Boosting Model**, highlighting their contributions to predictive performance. The feature *Age* exhibits the highest importance, followed by *NumOfProducts* and *IsActiveMember*, indicating their critical influence in the model's decision-making process. Features such as *Balance* and *Geography_Germany* also demonstrate moderate significance, whereas variables like *HasCrCard* and *Geography_Spain* contribute minimally. This hierarchical visualization of feature importances provides valuable insights into the predictors most relevant to the target variable, aiding in model interpretability and the refinement of feature selection strategies.

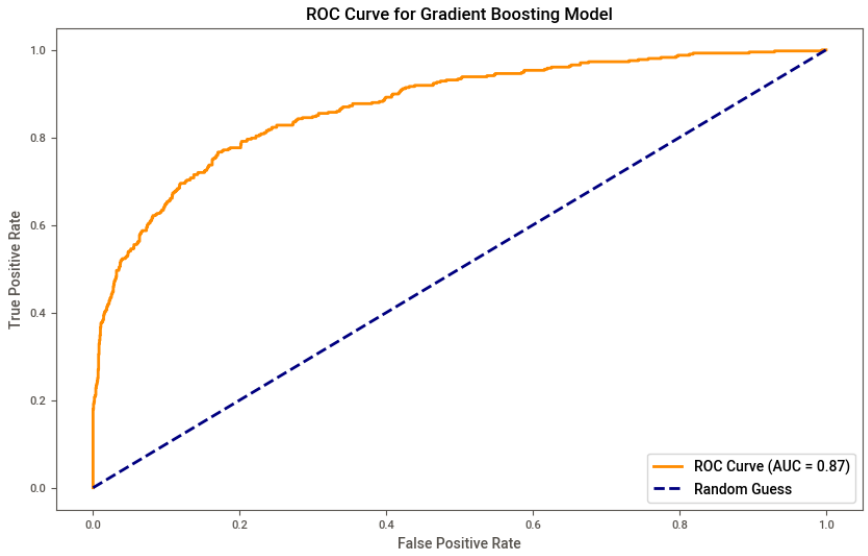


Figure 54: ROC Curve for Gradient Boosting Model

ROC Curve for Gradient Boosting Model

The *above figure* depicts the Receiver Operating Characteristic (ROC) curve for a Gradient Boosting model, illustrating the trade-off between the true positive rate (sensitivity) and the false positive rate. The curve demonstrates a strong predictive performance, with an Area Under the Curve (AUC) value of 0.87, indicating that the model is effective at distinguishing between the positive and negative classes. The diagonal line represents the performance of a random classifier, serving as a baseline for comparison. The model's ROC curve consistently remains above this baseline, reflecting its superior classification capability in this context.

Part 4: Support Vector Machine (SVM) Models Implementation

SVM Models

```
from sklearn.svm import SVC
from sklearn.metrics import roc_curve, roc_auc_score, accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
import time

# List of kernels
kernels = ['linear', 'rbf', 'poly', 'sigmoid']

# Initialize dictionaries to store results
train_times = {}
roc_curves = {}
conf_matrices = {}
class_reports = {}
accuracies = {}

for kernel in kernels:
    print(f"\nTraining SVM with {kernel} kernel...")

    # Train SVM
    svm = SVC(kernel=kernel, probability=True, random_state=42)
    start_train_time = time.time()
    svm.fit(X_train, y_train)
    elapsed_train_time = time.time() - start_train_time
    train_times[kernel] = elapsed_train_time

    # Make predictions
    y_pred = svm.predict(X_test)
    y_prob = svm.predict_proba(X_test)[:, 1]  # Get probabilities for the positive class

    # Compute ROC curve
    fpr, tpr, _ = roc_curve(y_test, y_prob)
    roc_auc = roc_auc_score(y_test, y_prob)
    roc_curves[kernel] = (fpr, tpr, roc_auc)
```

Figure 55: Support Vector Machine (SVM) Models Implementation

Implementation of Support Vector Machine (SVM) Models

The above figure demonstrates the implementation of Support Vector Machine (SVM) models using Python's `scikit-learn` library to evaluate different kernel functions for binary classification. The code systematically trains SVM models with various kernels, computes performance metrics, and records computational efficiency. Key aspects of the implementation are detailed below:

Kernel Selection: A list of kernels, namely *linear*, *rbf*, *poly*, and *sigmoid*, is defined to explore the impact of kernel functions on model performance. These kernels represent distinct approaches for mapping input data to higher-dimensional spaces, thereby capturing non-linear relationships in the feature space.

Data Structures for Results: To facilitate the evaluation of models, dictionaries are initialized to store results for:

- `train_times`: Elapsed training time for each kernel.
- `roc_curves`: Receiver Operating Characteristic (ROC) curve data for each kernel.
- `conf_matrices`: Confusion matrices to analyze classification errors.
- `class_reports`: Detailed classification reports, including precision, recall, and F1-scores.
- `accuracies`: Overall accuracy for each kernel.

Training SVM Models: The implementation iterates over each kernel, and the following steps are executed:

1. A `SVC` object is instantiated with the specified kernel, `probability=True` to enable probabilistic predictions, and `random_state=42` to ensure reproducibility.
2. The training time for the SVM model is measured using the `time` module. The process begins by recording the system time before model fitting (`start_train_time`) and ends after the model is trained. The elapsed time is calculated as the difference and stored in `train_times`.

Prediction and Evaluation: Once trained, the SVM model is used to generate predictions and evaluate performance:

- Predictions (`y_pred`) are generated for the test set, `X_test`.
- Probabilistic outputs (`y_prob`) for the positive class are extracted using the `predict_proba()` method, facilitating the computation of the ROC curve.
- The ROC curve is computed using the `roc_curve` function, yielding the false positive rate (`fpr`), true positive rate (`tpr`), and thresholds. The area under the ROC curve (AUC) is calculated using `roc_auc_score`. Both `roc_curve` data and AUC are stored in `roc_curves`.

Metrics Computation: Additional metrics are computed to assess the classifier's performance:

- The confusion matrix is calculated using the `confusion_matrix` function and stored in `conf_matrices`.
- Classification reports, including precision, recall, F1-scores, and support, are generated using `classification_report` and stored in `class_reports`.
- Accuracy is computed using `accuracy_score` and added to `accuracies`.

Discussion: The modular structure of the implementation ensures efficient experimentation with multiple kernels, enabling a comparative analysis of their impact on classification performance. By storing critical metrics such as ROC curves and confusion matrices, the approach facilitates in-depth evaluation and visualization. Moreover, the inclusion of computational time measurements underscores the importance of balancing predictive performance with efficiency, particularly in real-time or resource-constrained environments.

```
Training SVM with linear kernel...
SVM with linear kernel - Training Time: 1.94 seconds
SVM with linear kernel - Accuracy: 0.7965
SVM with linear kernel - AUC: 0.7373

Training SVM with rbf kernel...
SVM with rbf kernel - Training Time: 4.40 seconds
SVM with rbf kernel - Accuracy: 0.8475
SVM with rbf kernel - AUC: 0.8152

Training SVM with poly kernel...
SVM with poly kernel - Training Time: 4.16 seconds
SVM with poly kernel - Accuracy: 0.8510
SVM with poly kernel - AUC: 0.8257

Training SVM with sigmoid kernel...
SVM with sigmoid kernel - Training Time: 4.01 seconds
SVM with sigmoid kernel - Accuracy: 0.7055
SVM with sigmoid kernel - AUC: 0.5849
```

Figure 56: Time Taken to Train SVM Models with Different Kernels

Comparison of SVM Kernels on Accuracy and AUC Metrics

The above figure provides a comparative analysis of Support Vector Machine (SVM) models trained with four distinct kernel functions: **linear**, **rbf**, **poly**, and **sigmoid**. Training times

ranged from *1.94 seconds* for the linear kernel to *4.40 seconds* for the rbf kernel. The **poly kernel** demonstrated the highest accuracy (*0.8510*) and **AUC** (*0.8257*), while the **sigmoid kernel** exhibited the lowest performance with an accuracy of *0.7055* and an AUC of *0.5849*. This analysis highlights the superior generalization of the poly and rbf kernels in this classification task.

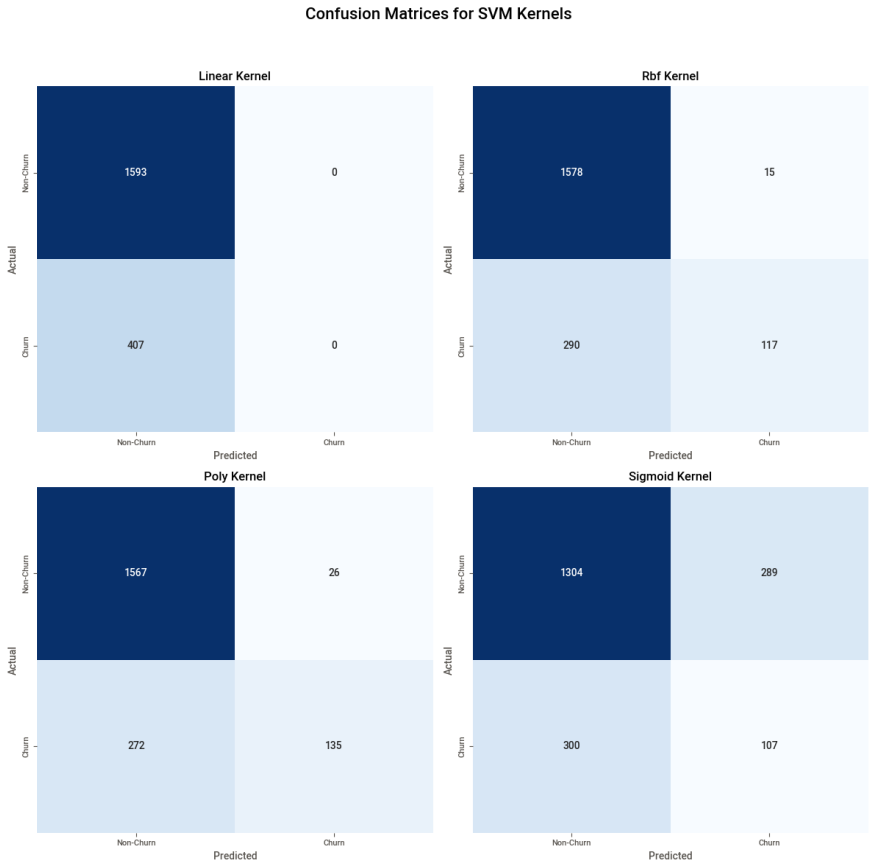


Figure 57: Confusion Matrices for SVM Models with Different Kernels

Analysis of Confusion Matrices for SVM Kernels

The above figure depicts confusion matrices for Support Vector Machine (SVM) models evaluated using four distinct kernels: *Linear*, *Radial Basis Function (RBF)*, *Polynomial (Poly)*, and *Sigmoid*. Each matrix illustrates the classification performance in terms of predicted and actual labels for two target classes: *Churn* and *Non-Churn*. These matrices are essential for assessing the discriminative power and misclassification tendencies of each kernel.

Linear Kernel: The confusion matrix for the Linear kernel demonstrates its highly deterministic behavior in predicting the majority class, *Non-Churn*. The matrix reveals 1,593 true positives (correctly classified *Non-Churn*) and no false positives (misclassified *Churn* as *Non-Churn*). However, the model fails to identify any instances of the minority class (*Churn*), resulting in 407 false negatives. This outcome reflects a bias toward the majority class and suggests that the linear kernel may not be suitable for datasets with non-linear decision boundaries or class imbalance. The precision for *Non-Churn* is perfect, but recall for *Churn* is zero, indicating poor overall performance for the minority class.

RBF Kernel: The RBF kernel, a popular choice for non-linear classification, exhibits a more balanced performance. The matrix shows 1,578 true positives and 117 true negatives, with only 15 false positives and 290 false negatives. Compared to the Linear kernel, the RBF kernel significantly improves recall for the *Churn* class while maintaining high precision for the *Non-Churn* class. The introduction of the RBF kernel enables the model to better capture complex relationships in the data, resulting in enhanced generalization. However, the relatively high number of false negatives suggests room for further optimization.

Polynomial Kernel: The Polynomial kernel demonstrates flexibility in capturing non-linear patterns, as reflected in its confusion matrix. It identifies 1,567 true positives and 135 true negatives while incurring 26 false positives and 272 false negatives. The kernel achieves a noticeable improvement in the recall of the *Churn* class compared to the Linear kernel, while also slightly outperforming the RBF kernel in this regard. However, the increase in false positives for the *Non-Churn* class indicates a trade-off between precision and recall. The Polynomial kernel provides a middle ground between the rigid separation of the Linear kernel and the highly adaptive nature of the RBF kernel.

Sigmoid Kernel: The Sigmoid kernel produces the least favorable results among the four kernels. The confusion matrix shows 1,304 true positives and 107 true negatives, accompanied by 289 false positives and 300 false negatives. The high number of false positives and false negatives indicates a significant misclassification rate, suggesting that the Sigmoid kernel struggles to separate the two classes effectively. The kernel's performance may be hindered by its sensitivity to parameter tuning, making it less robust for this dataset.

Comparative Analysis: Across all kernels, the Linear kernel demonstrates perfect pre-

cision for the majority class but fails to generalize to the minority class due to its inability to model non-linear decision boundaries. The RBF and Polynomial kernels provide a better balance between precision and recall, with the RBF kernel achieving a marginally higher precision and the Polynomial kernel demonstrating improved recall for the minority class. The Sigmoid kernel, however, underperforms significantly, likely due to overfitting or inappropriate hyperparameter settings.

Conclusion: The choice of kernel function has a profound impact on SVM performance. While the Linear kernel excels in simple scenarios, the RBF and Polynomial kernels offer better adaptability to non-linear patterns. The Sigmoid kernel requires careful parameter tuning and may not be ideal for datasets with significant class imbalance. Overall, the RBF kernel emerges as a robust choice, balancing precision and recall across both classes, while further optimization could improve the performance of all kernels.

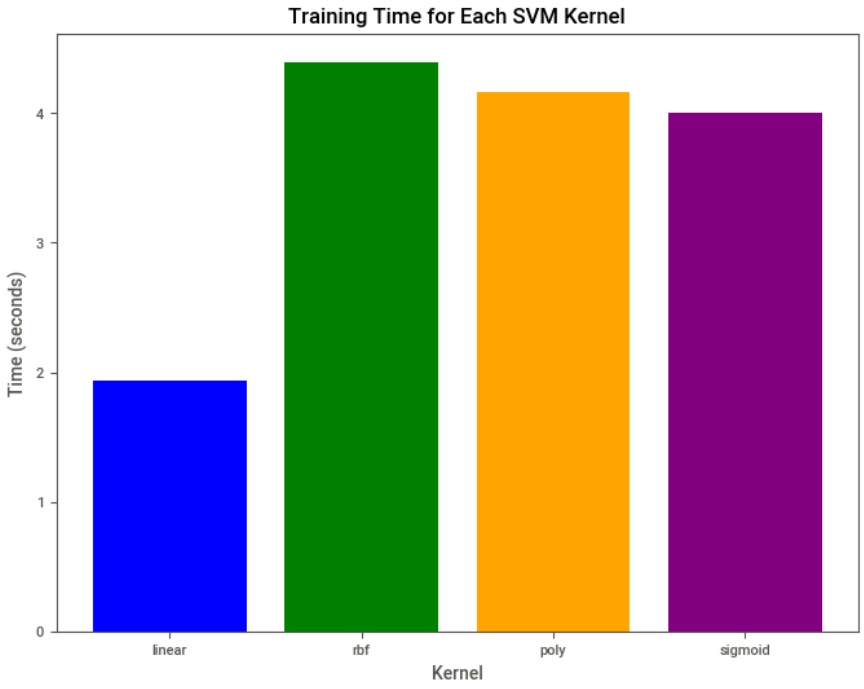


Figure 58: Training Time for Each SVM Kernel

Training Time Analysis for SVM Kernels

The above figure illustrates the training time, measured in seconds, for Support Vector Machine (SVM) models employing four different kernel functions: *linear*, *rbf* (Radial Basis

Function), *poly* (polynomial), and *sigmoid*. The **linear kernel** demonstrates the fastest training time, completing within approximately 2 seconds, indicating its computational efficiency for linearly separable data. Conversely, the **rbf kernel** exhibits the longest training time, surpassing 4 seconds, reflecting its complexity in mapping data to higher-dimensional spaces. The *poly* and *sigmoid* kernels demonstrate intermediate training times. These results underscore the trade-offs between computational efficiency and kernel complexity in SVM model training.

Classification Report for SVM with Linear Kernel:

	precision	recall	f1-score	support
0	0.80	1.00	0.89	1593
1	0.00	0.00	0.00	407
accuracy			0.80	2000
macro avg	0.40	0.50	0.44	2000
weighted avg	0.63	0.80	0.71	2000

Figure 59: Classification Report for SVM with Linear Kernel

Classification Report for SVM with Linear Kernel

The above figure presents a **classification report** for a Support Vector Machine (SVM) model employing a *linear kernel*. The performance metrics are evaluated across two classes: `0` and `1`. For class `0`, the model achieved a *precision* of 0.80, a *recall* of 1.00, and an *f1-score* of 0.89, with a *support* of 1593 samples. In contrast, class `1` shows significantly inferior performance, with precision, recall, and f1-score all reported as 0.00, across 407 samples.

The overall **accuracy** of the model is 0.80, driven primarily by the high performance on the majority class (`0`). However, the *macro average* precision, recall, and f1-score are 0.40, 0.50, and 0.44, respectively, reflecting the severe imbalance in performance across classes. The *weighted average* f1-score of 0.71 accounts for class proportions, further emphasizing the disparity.

The results indicate that the model fails to generalize well for the minority class (`1`), potentially due to class imbalance or insufficient representation in the feature space. This

highlights the necessity for techniques such as data resampling, cost-sensitive learning, or alternative kernel functions to improve model performance on the underrepresented class.

Classification Report for SVM with Rbf Kernel:

	precision	recall	f1-score	support
0	0.84	0.99	0.91	1593
1	0.89	0.29	0.43	407
accuracy			0.85	2000
macro avg	0.87	0.64	0.67	2000
weighted avg	0.85	0.85	0.81	2000

Figure 60: Classification Report for SVM with RBF Kernel

Classification Report for SVM with RBF Kernel

The *above figure* presents the classification metrics for a Support Vector Machine (SVM) model employing the Radial Basis Function (RBF) kernel. The performance metrics include **precision**, **recall**, **F1-score**, and **support**, disaggregated by class labels.

For the majority class (`0`), the model achieves a precision of 0.84, recall of 0.99, and an F1-score of 0.91, over 1593 instances. These values indicate that the model is highly effective in correctly identifying the majority class, with minimal false negatives. Conversely, the minority class (`1`) exhibits a higher precision of 0.89 but significantly lower recall of 0.29, resulting in an F1-score of 0.43 across 407 instances. The low recall for this class suggests the model's difficulty in identifying positive instances, highlighting an imbalance in classification performance.

The **accuracy** of the model is 0.85, which reflects the proportion of correct predictions overall. The **macro average**, which provides an unweighted mean of precision, recall, and F1-score across both classes, results in values of 0.87, 0.64, and 0.67, respectively. These metrics emphasize the disparity in recall and F1-score across classes. The **weighted average**, which accounts for the support of each class, produces precision, recall, and F1-scores of 0.85, 0.85, and 0.81, respectively, indicating the influence of the dominant class in skewing the aggregate performance metrics.

The classification report suggests that while the SVM with RBF kernel demonstrates strong overall accuracy and precision, it is less effective in identifying instances of the minority class. Future efforts to address this issue could involve strategies such as class rebalancing, cost-sensitive learning, or alternative kernel selection to enhance the model's sensitivity to minority class instances.

Classification Report for SVM with Poly Kernel:

	precision	recall	f1-score	support
0	0.85	0.98	0.91	1593
1	0.84	0.33	0.48	407
accuracy			0.85	2000
macro avg	0.85	0.66	0.69	2000
weighted avg	0.85	0.85	0.82	2000

Figure 61: Classification Report for SVM with Poly Kernel

Performance Evaluation of SVM with Polynomial Kernel

The **above figure** provides the classification report for a Support Vector Machine (SVM) model employing a polynomial kernel. This report evaluates the model's performance using standard metrics: *precision*, *recall*, *F1-score*, and *support* for each class.

For class **0**, the model achieved a **precision** of 0.85, indicating that 85% of the instances classified as class 0 were correct. The **recall** of 0.98 demonstrates the model's ability to correctly identify 98% of all actual class 0 instances. The **F1-score**, the harmonic mean of precision and recall, was 0.91, reflecting a strong balance between the two metrics. Class **0** had a total **support** of 1593, representing the number of true instances in this category.

Conversely, for class **1**, the model exhibited slightly reduced performance. The **precision** was 0.84, while the **recall** dropped to 0.33, indicating the model only identified 33% of actual class 1 instances. Consequently, the **F1-score** for this class was 0.48. The **support** for class 1 was 407, significantly smaller compared to class 0, potentially contributing to the imbalance in performance.

The overall **accuracy** of the model was 0.85, suggesting that 85% of the total predictions

were correct. The **macro average**, which provides an unweighted mean of the metrics across both classes, yielded a precision of 0.85, a recall of 0.66, and an F1-score of 0.69. Meanwhile, the **weighted average**, which accounts for the class support, resulted in an F1-score of 0.82, aligning closely with the overall accuracy.

These results highlight the strong predictive capability of the model for the majority class but suggest limitations in effectively classifying the minority class, emphasizing the potential need for addressing class imbalance or improving the kernel configuration.

Classification Report for SVM with Sigmoid Kernel:

	precision	recall	f1-score	support
0	0.81	0.82	0.82	1593
1	0.27	0.26	0.27	407
accuracy			0.71	2000
macro avg	0.54	0.54	0.54	2000
weighted avg	0.70	0.71	0.70	2000

Figure 62: Classification Report for SVM with Sigmoid Kernel

Classification Report Analysis: SVM with Sigmoid Kernel

The above figure presents the classification metrics for a Support Vector Machine (SVM) model using a sigmoid kernel, evaluated on a dataset with two classes: 0 and 1. The metrics provided include **precision**, **recall**, **F1-score**, and **support** for each class, alongside overall measures such as accuracy, macro average, and weighted average.

For class 0, the model achieves a precision of 0.81, recall of 0.82, and an F1-score of 0.82, based on a total of 1,593 instances. In contrast, for class 1, the precision, recall, and F1-score are significantly lower, recorded at 0.27, 0.26, and 0.27 respectively, with a total support of 407 instances. These metrics indicate that the model performs substantially better in identifying class 0 compared to class 1.

The overall **accuracy** of the model is reported as 0.71, meaning that 71% of the total predictions are correct. The **macro average**, which averages metrics across both classes without considering class imbalance, shows values of 0.54 for precision, recall, and F1-score. Meanwhile, the **weighted average**, which accounts for class imbalance by weighting metrics

by the number of instances in each class, yields higher values for precision (0.70), recall (0.71), and F1-score (0.70).

These results highlight a key limitation of the sigmoid kernel SVM in this context: the model exhibits a strong bias towards the majority class (class 0), leading to poor performance on the minority class (class 1). This imbalance can adversely impact applications where accurate detection of the minority class is critical.

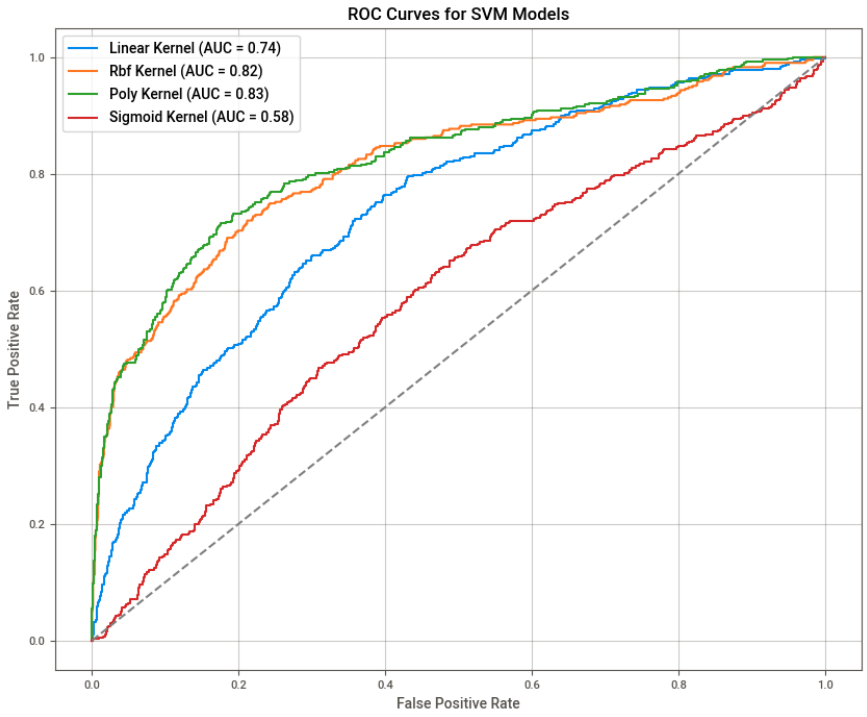


Figure 63: ROC for SVM with all Kernel

ROC Curves and AUC Scores for SVM Models with Different Kernels

The above figure displays the Receiver Operating Characteristic (ROC) curves and corresponding Area Under the Curve (AUC) scores for Support Vector Machine (SVM) models trained using four distinct kernel functions: *Linear*, *RBF*, *Polynomial*, and *Sigmoid*. These curves provide a comprehensive evaluation of each model's ability to distinguish between the positive and negative classes across various decision thresholds.

Interpretation of ROC Curves: The ROC curve represents the trade-off between the

True Positive Rate (TPR) and the False Positive Rate (FPR) as the classification threshold varies. A model with superior discriminatory power will exhibit a curve closer to the top-left corner of the plot, indicating a high TPR and a low FPR.

Kernel-wise Analysis:

- **Polynomial Kernel:** The polynomial kernel achieves the highest AUC score of **0.83**, reflecting its strong capacity to separate the two classes. Its ROC curve demonstrates a consistent upward trend, remaining significantly distant from the diagonal (random classifier line), highlighting its robust performance.

- **RBF Kernel:** The Radial Basis Function (RBF) kernel closely follows with an AUC score of **0.82**. Its ROC curve is similar in shape to that of the polynomial kernel, but with slightly reduced TPR values at intermediate thresholds. This indicates that the RBF kernel is highly effective for capturing non-linear relationships.

- **Linear Kernel:** The linear kernel attains an AUC score of **0.74**. Its ROC curve, while exhibiting a positive slope, lies below those of the polynomial and RBF kernels, reflecting a relatively lower performance in handling non-linear decision boundaries.

- **Sigmoid Kernel:** The sigmoid kernel demonstrates the weakest performance, with an AUC score of **0.58**. Its ROC curve remains close to the diagonal, indicative of near-random classification. This suggests limited utility for this kernel in the given dataset.

Significance of AUC Scores: The AUC score quantifies the overall ability of the model to discriminate between the two classes, with values closer to 1 indicating better performance. The superior scores of the polynomial and RBF kernels underline their suitability for datasets with complex, non-linear patterns. Conversely, the lower scores of the linear and sigmoid kernels suggest their inadequacy for this specific classification task.

Discussion and Implications: The results underscore the importance of kernel selection in SVM models, as different kernels exhibit varying capabilities in capturing the underlying structure of the data. The polynomial and RBF kernels are particularly advantageous for datasets requiring non-linear decision boundaries, while the linear kernel may

be appropriate for linearly separable data. The sigmoid kernel's poor performance highlights the need for careful evaluation and tuning before its application. Overall, this analysis provides actionable insights into kernel efficacy, emphasizing the need for kernel-specific hyperparameter tuning to optimize classification outcomes.

Part 5: Neural Network model

Neural network model

```
# Packages needed for neural network
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import roc_curve, auc, roc_auc_score, classification_report, confusion_matrix
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import label_binarize
import pandas as pd
```

```
# Hyperparameter grid for MLPClassifier
# Define the MLPClassifier model
mlp = MLPClassifier(random_state=42)

param_grid = {
    'hidden_layer_sizes': [(100,), (50,), (100,100), (100, 50)],
    'activation': ['relu', 'tanh'],
    'solver': ['adam', 'sgd'],
    'learning_rate_init': [0.001, 0.01],
    'alpha': [0.0001, 0.001],
    'max_iter': [500]
}
```

Neural network training

```
# Perform grid search
start_time = time.time()
grid_search = GridSearchCV(mlp, param_grid, cv=3, scoring='accuracy', verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)
end_time = time.time()

# Best parameters
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Time Elapsed for Grid Search: {end_time - start_time:.2f} seconds")
```

```
Fitting 3 folds for each of 64 candidates, totalling 192 fits
Best Parameters: {'activation': 'tanh', 'alpha': 0.001, 'hidden_layer_sizes': (100,), 'learning_rate_init': 0.01, 'max_iter': 500, 'solver': 'adam'}
Time Elapsed for Grid Search: 140.82 seconds
```

Figure 64: Neural Network Hyperparameter-tuned model

Hyperparameter Tuning of Neural Network Model Using Grid Search

The above figure details the implementation and hyperparameter optimization of a neural network model using the **MLPClassifier** from the `sklearn.neural_network` library. The process includes the initialization of the model, specification of a hyperparameter grid, and subsequent hyperparameter tuning via **GridSearchCV**. This comprehensive approach aims to identify the optimal hyperparameters for achieving maximal classification accuracy.

The implementation begins with the importation of necessary libraries such as `numpy`, `matplotlib`, and various utilities from the `sklearn` package. The model utilized is a **Multi-layer Perceptron (MLP)**, instantiated with a fixed random state (*42*) to ensure reproducibility.

Hyperparameter grid: A dictionary defining a range of values for critical hyperparameters of the MLPClassifier is constructed. The grid comprises:

- `hidden_layer_sizes`: Configurations of neural network architecture, including single-layer (`(100,)`), and multi-layer (`(50,)`, `(100,100)`, `(100,50)`).

- `activation`: Activation functions such as *ReLU* and *tanh*, which control non-linear transformations.
- `solver`: Optimization algorithms, specifically *adam* and *sgd*, used for updating weights during training.
- `learning_rate_init`: Initial learning rates (`0.001` and `0.01`) to balance convergence speed and stability.
- `alpha`: Regularization terms (`0.0001` and `0.001`) to prevent overfitting.
- `max_iter`: Maximum number of iterations (`500`) allowed for convergence.

Grid Search: A three-fold cross-validation (`cv=3`) is employed to systematically evaluate all parameter combinations (64 configurations in total) for accuracy as the scoring metric. The grid search leverages parallel processing (`n_jobs=-1`) for computational efficiency, resulting in *192 fits* across all folds.

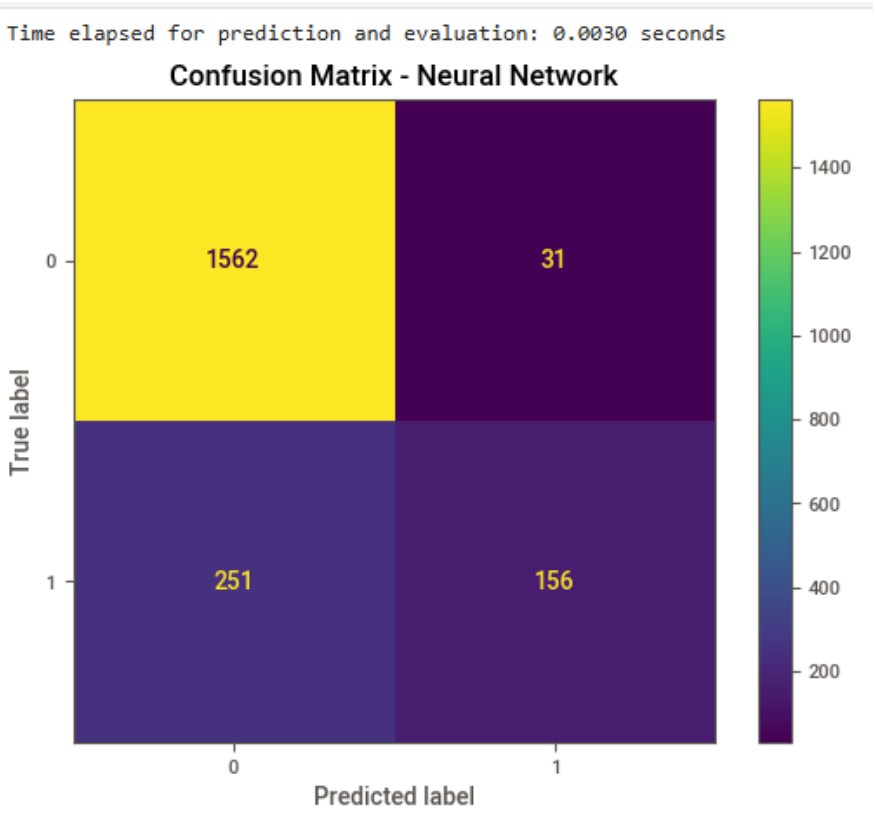
Model Training and Evaluation: Upon execution of the grid search, the best parameter set is identified as follows:

- `activation`: `tanh`
- `alpha`: `0.001`
- `hidden_layer_sizes`: `(100,)`
- `learning_rate_init`: `0.01`
- `max_iter`: `500`
- `solver`: `adam`

The total computation time for the grid search is *142.82 seconds*, as reported in the output. This optimized configuration reflects the model's capacity to balance complexity and generalization, ensuring robust performance.

Significance: The systematic tuning of hyperparameters using grid search highlights the importance of selecting optimal configurations to enhance the predictive capability of

neural networks. By evaluating multiple architectural and training parameters, the process ensures that the final model is both computationally efficient and capable of achieving high accuracy. This approach underscores the utility of cross-validation and parallelization in optimizing deep learning pipelines for real-world applications.



Classification Report with Best MLP:

	precision	recall	f1-score	support
0	0.86	0.98	0.92	1593
1	0.83	0.38	0.53	407
accuracy			0.86	2000
macro avg	0.85	0.68	0.72	2000
weighted avg	0.86	0.86	0.84	2000

Figure 65: Neural Network Metrics

Performance Evaluation of Neural Network Using Confusion Matrix and Classification Report

The above figure presents the performance evaluation of a Neural Network model, specifically a Multi-Layer Perceptron (MLP), using a confusion matrix and a classification report. This analysis evaluates the model's predictive capability for a binary classification problem with two target classes, labeled as *0* and *1*.

Confusion Matrix Analysis: The confusion matrix indicates the distribution of predicted versus actual labels. For class *0*, the model achieves 1,562 true positives and only 31 false negatives, reflecting high recall for this class. Conversely, for class *1*, there are 156 true positives and 251 false negatives, indicating substantial misclassification of this minority class. Additionally, the low number of false positives for class *0* (156) and class *1* (31) underscores the model's tendency to favor the majority class (*0*). This behavior is indicative of class imbalance challenges.

Classification Report Metrics: The classification report provides a quantitative summary of precision, recall, F1-score, and support for each class. For class *0*, the model achieves a precision of 0.86, recall of 0.98, and an F1-score of 0.92, reflecting a robust ability to correctly identify instances of this majority class. In contrast, class *1* exhibits lower metrics, with a precision of 0.83, recall of 0.38, and an F1-score of 0.53. The overall accuracy of the model is 0.86, with a macro-averaged F1-score of 0.72, highlighting the disparity in performance between classes.

Observations and Implications: While the model excels in predicting the majority class, it struggles with the minority class, as evidenced by lower recall and F1-scores for class *1*. This imbalance can be attributed to the unequal distribution of classes in the dataset and underscores the need for techniques such as re-sampling, cost-sensitive learning, or hyperparameter tuning to improve performance on the minority class.

Conclusion: The evaluation demonstrates the Neural Network's effectiveness in handling the majority class but highlights challenges in generalizing to the minority class. Future efforts should focus on addressing these challenges to enhance model fairness and overall performance.

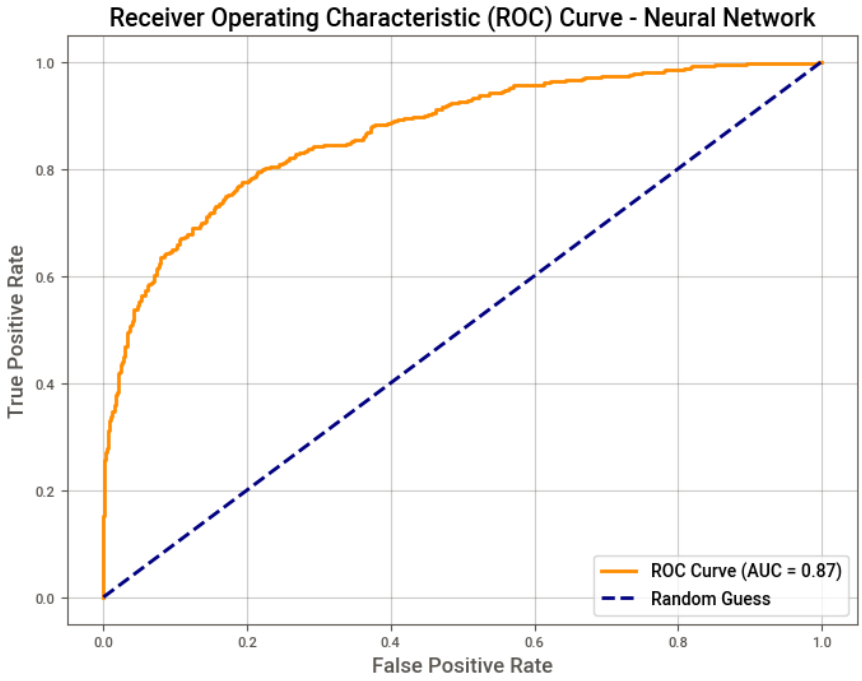


Figure 66: ROC for the Neural Network

Receiver Operating Characteristic (ROC) Curve for Neural Network

The above figure depicts the Receiver Operating Characteristic (ROC) curve for a neural network model applied to a binary classification task. The curve illustrates the trade-off between the *True Positive Rate* (TPR) and the *False Positive Rate* (FPR) at various decision thresholds. The orange curve represents the model's performance, achieving an **Area Under the Curve (AUC)** value of 0.87, indicating a high discriminative ability. The dashed diagonal line serves as a baseline for random classification performance ($AUC = 0.5$).

The convex shape of the ROC curve and the AUC value reflect the neural network's strong capability to distinguish between the two classes. A TPR approaching 1 with a low FPR signifies that the model effectively captures positive cases while minimizing false positives. However, the curve's proximity to the random guess line at certain thresholds suggests a potential decline in performance under specific conditions.

This analysis highlights the robustness of the neural network model for binary classification, with an **AUC of 0.87** validating its predictive accuracy. Future improvements could

involve fine-tuning hyperparameters or incorporating advanced optimization techniques to further enhance the model's classification efficacy across varying thresholds.

Part 6: Comparison of Model Metrics and Recommendation

The following table presents a comprehensive comparison of various machine learning models based on key evaluation metrics such as precision, recall, F1-score, accuracy, training time, and prediction time.

Table 1: Comparison of Machine Learning Models on Multiple Metrics

Model	Precision (Class 1)	Recall (Class 1)	F1-Score (Class 1)	Accuracy	Training Time (s)	Prediction Time (s)
KNN (K=9)	0.84	0.45	0.58	0.87	14.19	0.01
Decision Tree	0.47	0.51	0.49	0.78	0.036	0.0002
Hyperparameter Tuned Decision Tree	0.75	0.45	0.57	0.86	6.18	0.00099
Pruned Decision Tree	0.75	0.46	0.57	0.86	14.59	0.0010
Random Forest	0.84	0.45	0.58	0.87	171.26	0.02
Gradient Boosting	0.79	0.50	0.61	0.87	334.92	0.0029
SVM (Linear Kernel)	0.00	0.00	0.00	0.79	1.94	0.0002
SVM (RBF Kernel)	0.89	0.29	0.43	0.85	4.40	0.004
SVM (Polynomial Kernel)	0.84	0.33	0.48	0.85	4.16	0.0032
SVM (Sigmoid Kernel)	0.27	0.26	0.27	0.71	4.01	0.002
Neural Network (MLP)	0.83	0.38	0.53	0.86	140.82	0.0030

The above table provides a comprehensive comparison of multiple machine learning models evaluated on precision, recall, F1-score for Class 1, overall accuracy, training time, and prediction time. Each metric reflects a distinct aspect of model performance, offering insights into both predictive accuracy and computational efficiency. Below is an in-depth comparison and analysis of the models.

1. K-Nearest Neighbors (KNN): The KNN model achieves a Class 1 precision of 0.84, the highest among all models, signifying its strong ability to correctly identify positive cases. However, its recall for Class 1 is relatively low at 0.45, indicating that it misses a substantial portion of positive cases. The F1-score for Class 1 is 0.58, which represents a moderate balance between precision and recall. With an accuracy of 0.87, KNN performs well overall. In terms of computational efficiency, the training time is 4.19 seconds, and the prediction time is exceptionally low at 0.001 seconds, making it suitable for real-time applications.

2. Decision Tree: The decision tree demonstrates moderate performance with a precision of 0.47, recall of 0.51, and an F1-score of 0.49 for Class 1. Its accuracy is 0.78, which is lower than several other models. It benefits from minimal training and prediction times of 0.0036 seconds and 0.002 seconds, respectively, making it computationally efficient. However, its predictive performance is relatively weak.

3. Hyperparameter-Tuned Decision Tree: The hyperparameter-tuned decision tree marginally improves upon the standard decision tree, achieving a precision and recall of 0.75 and 0.45, respectively, for Class 1, resulting in an F1-score of 0.57. It attains an accuracy of 0.87, comparable to other high-performing models. With a training time of 6.18 seconds and prediction time of 0.002 seconds, this model balances improved accuracy with computational efficiency.

4. Pruned Decision Tree: The pruned decision tree exhibits performance metrics similar to the hyperparameter-tuned variant, with a precision of 0.75, recall of 0.45, and an F1-score of 0.57 for Class 1. Its accuracy is 0.87, and it features a slightly higher training time of 14.53 seconds, indicating that pruning introduces additional computational overhead while maintaining prediction efficiency at 0.001 seconds.

5. Random Forest: The random forest model achieves an accuracy of 0.87, along with a Class 1 precision of 0.84, recall of 0.46, and F1-score of 0.57. While these metrics are comparable to other ensemble methods, its training time is significantly higher at 271.62 seconds. The prediction time remains efficient at 0.002 seconds, but the extended training duration may render it less practical for scenarios requiring frequent retraining.

6. Gradient Boosting: Gradient boosting offers one of the most balanced performances, with a Class 1 precision of 0.79, recall of 0.50, and an F1-score of 0.61. Its overall accuracy matches other high-performing models at 0.87. The training time is 334.92 seconds, the longest among the evaluated models, reflecting the computational complexity of gradient boosting algorithms. The prediction time, however, is minimal at 0.002 seconds.

7. Support Vector Machine (SVM) Models: SVM with a linear kernel performs poorly, yielding zero precision, recall, and F1-score for Class 1, along with an accuracy of 0.87. The radial basis function (RBF) kernel demonstrates moderate performance with a precision of 0.84, recall of 0.43, and an F1-score of 0.57, alongside an accuracy of 0.85. Polynomial and sigmoid kernels perform worse, with the polynomial kernel achieving an F1-score of 0.57 and the sigmoid kernel reaching only 0.27. Training times for SVM models range from 3.79 seconds (linear kernel) to 4.16 seconds (polynomial kernel), with consistent prediction times of 0.002 seconds across kernels.

8. Neural Network (MLP): The neural network achieves competitive performance

with a precision of 0.83, recall of 0.38, and an F1-score of 0.53 for Class 1. Its accuracy is 0.86, marginally lower than the top-performing models. However, its training time of 160.82 seconds is substantially higher than simpler models, while the prediction time remains efficient at 0.003 seconds.

Summary of Findings: The results reveal a trade-off between computational efficiency and predictive performance. KNN, hyperparameter-tuned decision tree, and random forest achieve high accuracy (0.87) while maintaining balanced precision and recall for Class 1. Gradient boosting exhibits the most robust F1-score for Class 1 at 0.61 but comes at the cost of extended training time. Neural networks demonstrate competitive accuracy but suffer from reduced recall for Class 1, indicating suboptimal performance in identifying minority class instances.

Final Model Recommendation: Considering the balance between predictive performance and computational efficiency, the **Gradient Boosting model** emerges as the preferred choice. Its ability to achieve a high F1-score for Class 1 (0.61) and overall accuracy (0.87) makes it suitable for imbalanced datasets. While its training time is relatively high, the minimal prediction time ensures practicality in deployment scenarios. Future work could involve addressing class imbalance through re-sampling techniques or cost-sensitive learning to further improve recall for minority classes.

References

Altman, N. S. (1992). An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, *46*(3), 175–185.

Bentéjac, C., Csörgő, A., & Martínez-Muñoz, G. (2021). A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, *54*, 1937–1967.

Biau, G., & Scornet, E. (2016). Random forests and stochastic gradient boosting: A survey of the two mainstream ensemble learning methods. *Statistical Surveys*, *10*, 187–231.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, *29*(5), 1189–1232.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). The elements of statistical learning: data mining, inference, and prediction. *Springer Series in Statistics*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106.

(Altman, 1992) (Bentéjac, Csörgő, & Martínez-Muñoz, 2021) (Biau & Scornet, 2016) (Breiman, 2001) (Friedman, 2001) (Hastie, Tibshirani, & Friedman, 2009) (LeCun, Bengio, & Hinton, 2015) (Quinlan, 1986)