Faculty of Technology and Engineering
Computer Engineering

Machine Learning Project

Amir Hossein Safarkhan Moazeni
40314163101

Course Instructor
Fatemeh Bagheri

Fall 2024

Contents

1. Dataset

The "Customer Churn" dataset provides information related to the customers of a telecommunications company, including whether each customer has churned or not. This dataset consists of 3,150 records and 14 features for each customer. These features encompass the number of failed calls (Call Failures), whether the customer has filed a complaint (Complains), the duration of service usage by the customer (Subscription Length), the account balance (Charge Amount), the total seconds of service usage (Seconds of Use), the number of service uses (Frequency of Use), the frequency of SMS usage (Frequency of SMS), the number of unique contacts (Distinct Called Numbers), the age group of the customer (Age Group), the specific age of the customer (Age), the type of service plan (Tariff Plan), the customer's status (Status), the estimated value of the customer (Customer Value), and whether the customer has churned or not (Churn). The dataset contains no null values, although some potential outliers are observed.

In the initial stage of preprocessing any dataset, analyzing and understanding the data and its features is essential. Reviewing characteristics such as the minimum, maximum, median, and mean values, and then visualizing this information along with dataset records using Python's **matplotlib** library, serves this purpose. Methods such as box plots, bar charts, pie charts, and heatmaps are commonly employed for this analysis.

For more Information and Code, visit the Github Page.

1.1. Description of the Dataset

	Feature	Description
1	Call Failures	Number of failed calls
2	Complains	Binary indicator of customer complaints
3	Subscription Length	Total months of subscription
4	Charge Amount	Ordinal attribute of charge amount
5	Seconds of Use	Total seconds of calls
6	Frequency of Use	Total number of calls
7	Frequency of SMS	Total number of text messages
8	Distinct Called Numbers	Total number of distinct phone calls
9	Age Group	Ordinal attribute of age group
10	Tariff Plan	Binary indicator of tariff plan
11	Status	Binary indicator of customer status
12	Age	Ordinal attribute of age
13	Customer Value	Calculated value of customer

14	Churn	Binary indicator of churn

1.2. Samples of Dataset

Fe 1	Fe 2	Fe 3	Fe 4	Fe 5	Fe 6	Fe 7	Fe 8	Fe 9	Fe 10	Fe 11	Fe 12	Fe 13	Fe 14
8	0	38	0	4370	71	5	17	3	1	1	30	197.64	0
12	0	17	0	5793	131	64	27	1	2	1	15	677.82	0
23	1	33	0	955	47	16	17	2	1	2	25	117.09	1
0	1	10	0	1363	21	4	10	2	1	1	25	80.28	0
0	0	37	0	0	0	9	0	2	1	2	25	40.5	0
9	0	43	2	7893	135	401	49	3	1	1	30	1925	1

1.3. Basic Analysis of Dataset

Missing Value	Median	Min	Max	Instances	Feature
0	**6**	**0**	**36**	**3150**	**Call Failure**
0	**35**	**3**	**47**	**3150**	**Subscription Length**
0	**2990**	**0**	**1790**	**3150**	**Seconds of Use**
0	**54**	**0**	**255**	**3150**	**Frequency of use**
0	**21**	**0**	**522**	**3150**	**Frequency of SMS**
0	**21**	**0**	**97**	**3150**	**Distinct Called Numbers**
0	**228.48**	**0**	**2165.28**	**3150**	**Customer Value**
0	0 → Instances 2909 1 → Instances 241			**3150**	Complains
0	0 → Instances 1768 1 → Instances 617 2 → Instances 395 3 → Instances 199 4 → Instances 76 5 → Instances 30 6 → Instances 11			**3150**	Charge Amount

	7 → Instances 14 8 → Instances 19 9 → Instances 14 10 → Instances 7		
0	1 → Instances 123 2 → Instances 1037 3 → Instances 1425 4 → Instances 395 5 → Instances 170	**3150**	Age Group
0	1 → Instances 2905 2 → Instances 245	**3150**	Tariff Plan
0	15 → Instances 123 25 → Instances 1037 30 → Instances 1425 44 → Instances 395 55 → Instances 170	**3150**	Age
0	1 → Instances 2368 2 → Instances 782	**3150**	Status
0	0 → Instances 2655 1 → Instances 495	**3150**	Churn

1.4. Detailed analysis of Dataset

In this section, we analyze the discussed features. The graphs and visualizations presented here are generated using libraries such as matplotlib, seaborn, scipy, statsmodels and squarify.

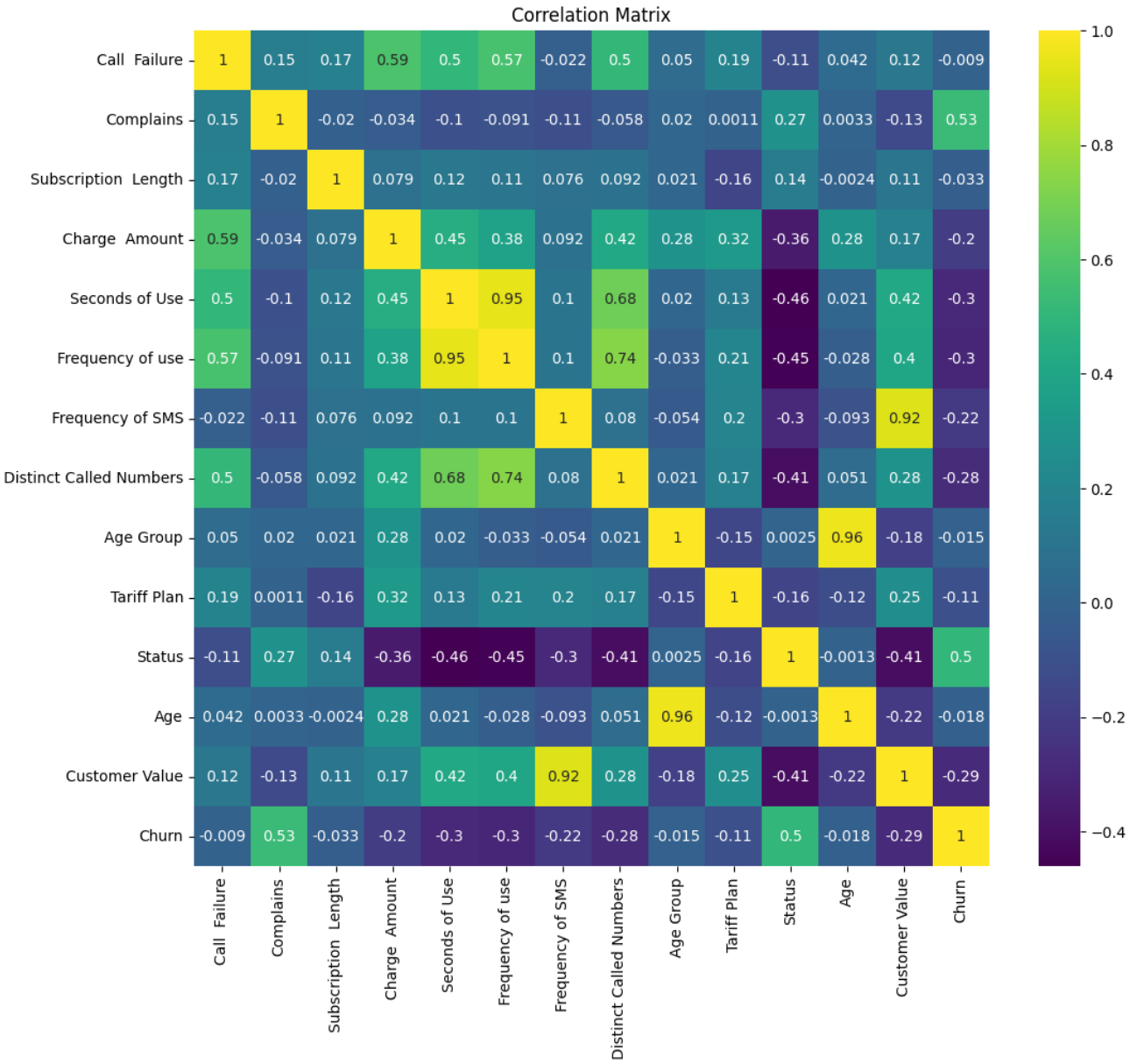


Figure 1. Correlation Matrix

We begin with the feature Call Failures. As shown in Figure 2, the majority of samples fall within the range of 0 to 5 failed calls, with data frequency decreasing as the number of failures increases. While this feature does not directly correlate with customer churn, the correlation matrix (Figure 1) reveals that the distribution of failed calls is closely related to features like charge amount, usage frequency, and duration of use.

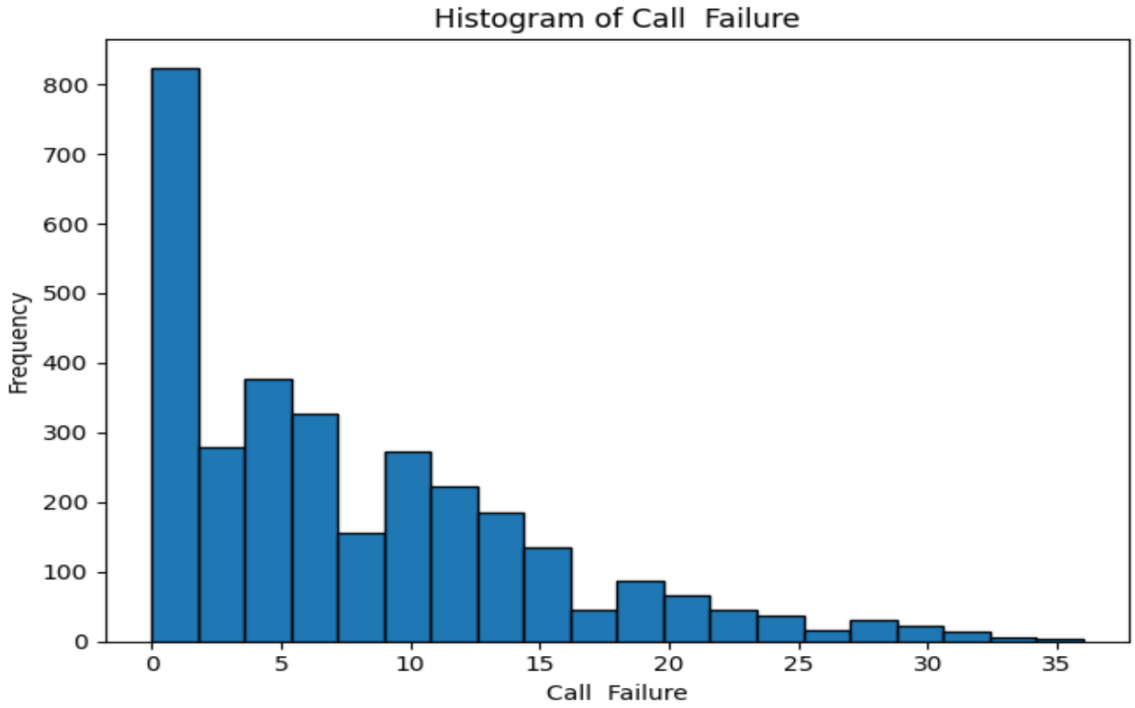


Figure 2. Call Failure Histogram

As seen in Figure 3, more customers with minimal service usage (fewer than 50 calls) are recorded as churned customers compared to people with higher frequency of use. However, this does not hold for power users (those with more than 200 calls), where failed calls never lead to churn. This figure also indicates that after 17 failed calls within a given time frame, customers who stop using the company's services are highly likely to churn.

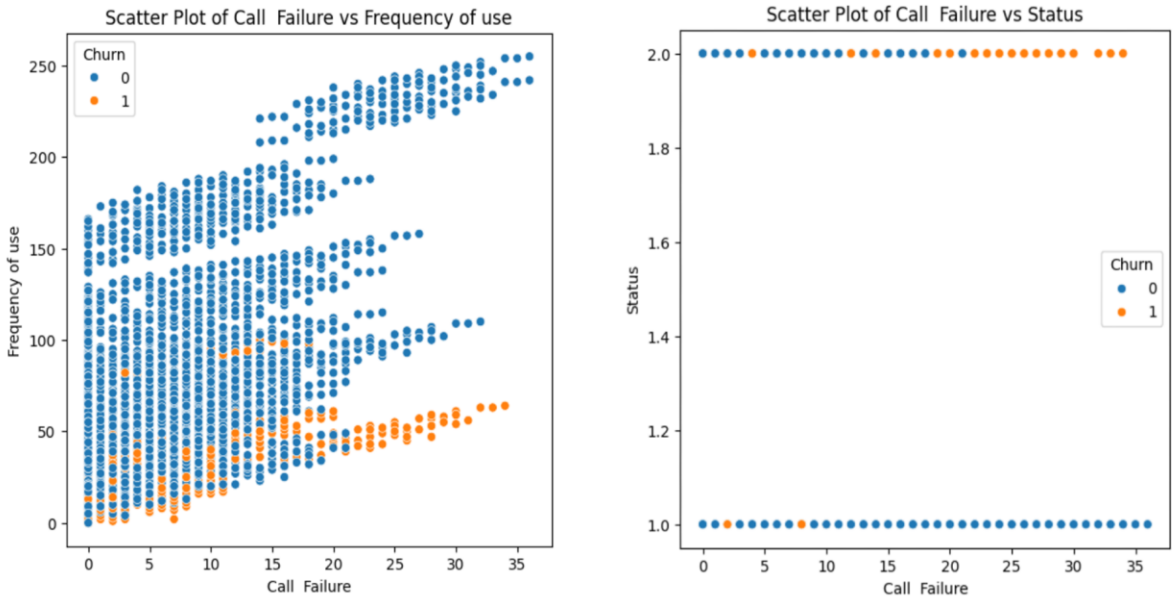
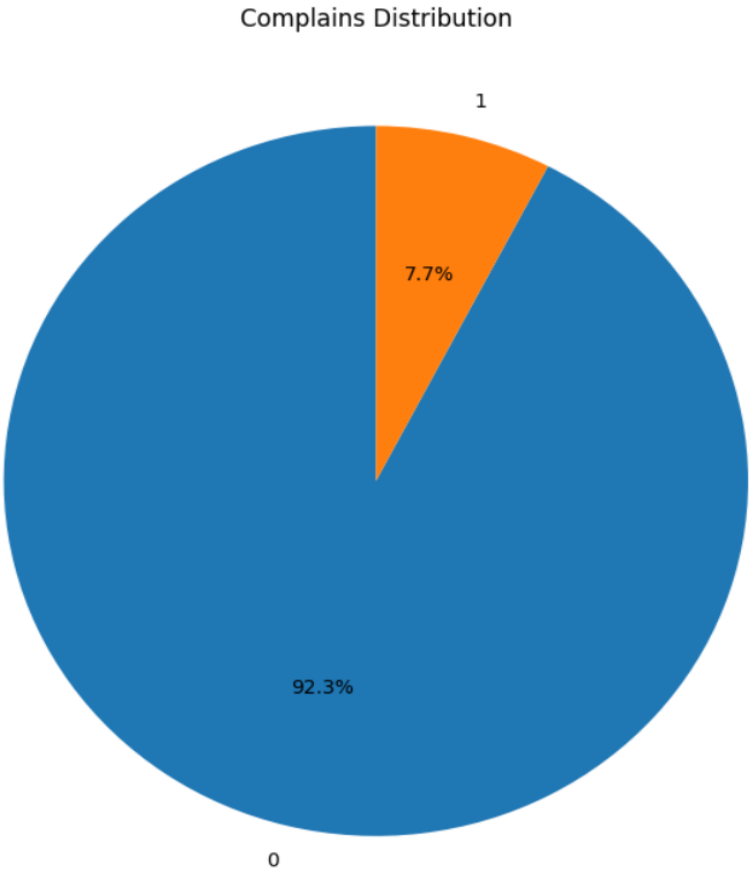


Figure 3. Scatter Plots of Call Failure vs Frequency of Use and Status

Next, we examine the Complaints feature, which indicates whether a customer has filed a complaint (value 1) or not (value 0). Although most customers of this company do not lodge complaints (Figure 4), Figure 5 demonstrates that the likelihood of chum increases significantly if a complaint is filed. However, as with failed calls, this trend does not apply to power users. This pattern extends to related features like duration of use, frequency of SMS, and customer value. As shown in the correlation matrix, complaints exhibit the highest correlation with customer churn, even for customers who have been using the company's services for a long time.



Complains	Count
0	2909
1	241

Figure 4. Distribution of Complains

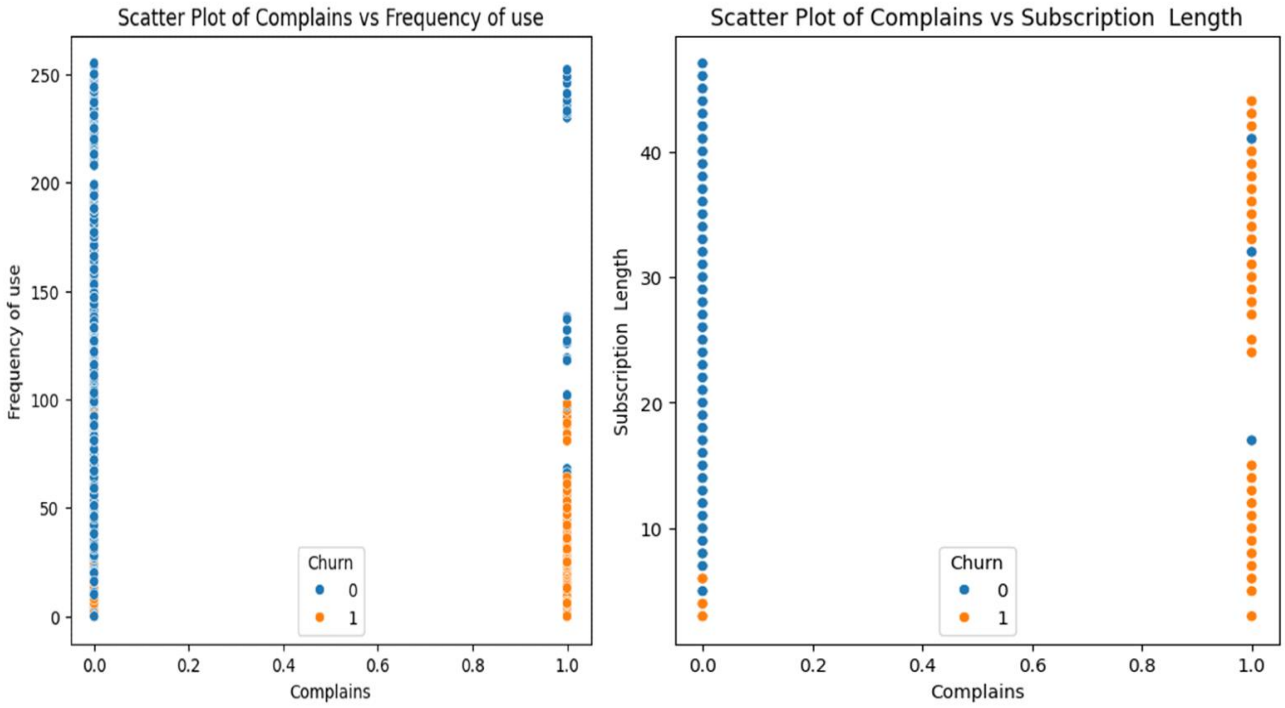


Figure 5. Scatter Plots of Complains vs Frequency of Use and Subscription Length

An analysis of the subscription length feature reveals that most customers, regardless of whether they churned, subscribe to the company's long-term service plans, as illustrated in Figure 6.

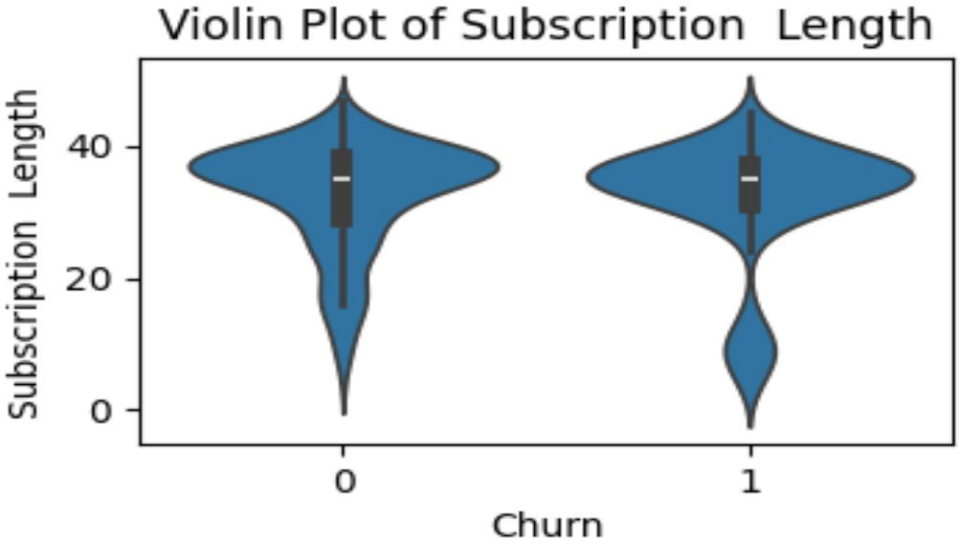
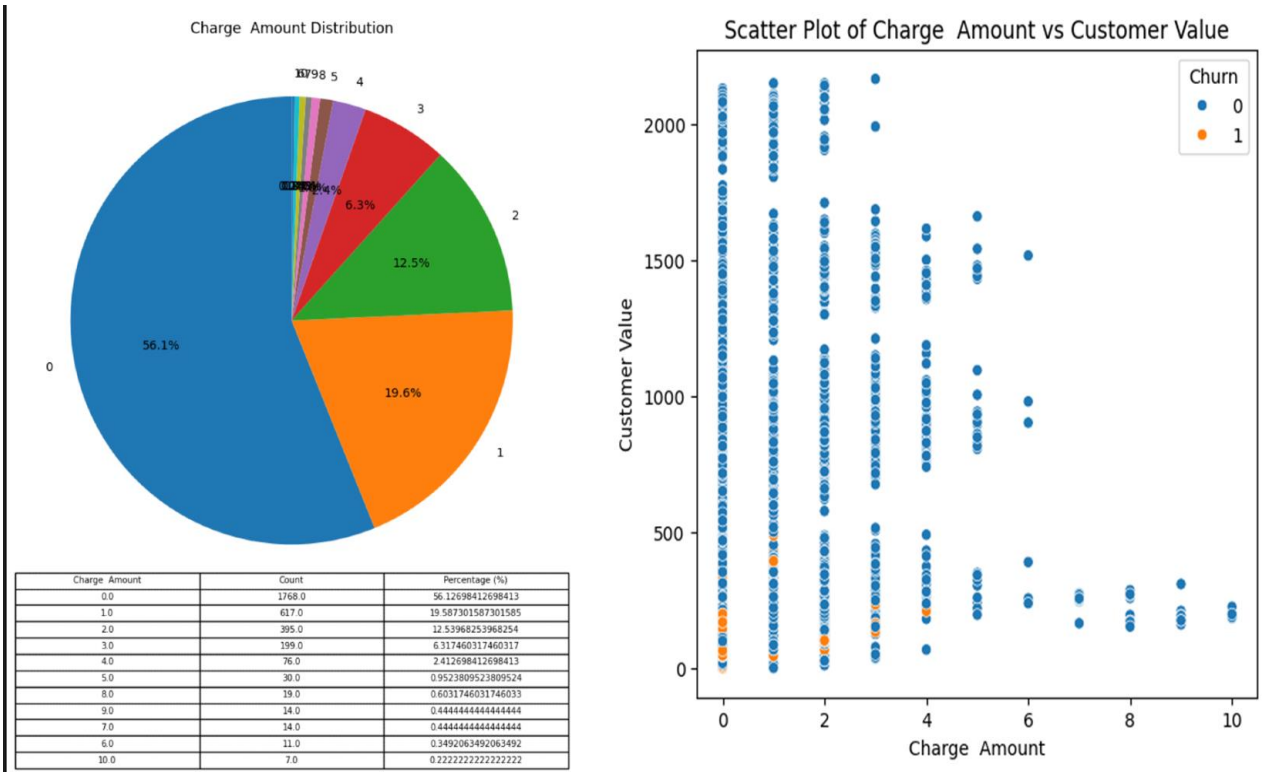


Figure 6. Violin Plot of Subscription Length

When examining the Charge Amount feature, it becomes evident from various charts, such as Figure 7, that most customers fall into the lower three tiers of service charges. However, a high charge amount does not necessarily equate to a higher customer value.



Charge Amount	Count	Percentage (%)
0.0	1768.0	56.12698412698413
1.0	617.0	19.587301587301585
2.0	395.0	12.53968253968254
3.0	199.0	6.317460317460317
4.0	76.0	2.412698412698413
5.0	30.0	0.9523809523809524
8.0	19.0	0.6031746031746033
9.0	14.0	0.4444444444444444
7.0	14.0	0.4444444444444444
6.0	11.0	0.3492063492063492
10.0	7.0	0.2222222222222222

Figure 7. Charge Distribution and Scatter Plot of Charge vs Customer Value

The Seconds of Use feature exhibits a linear relationship with the total number of calls. The highest usage times (a metric for identifying power users) are recorded for customers in the 25 and 30-year-old age groups, as shown in Figures 8 and then 9. A similar pattern is observed for the total number of calls, with the age distribution of the company's customers being more concentrated in these two age groups. The frequency of SMS usage follows a similar logic.

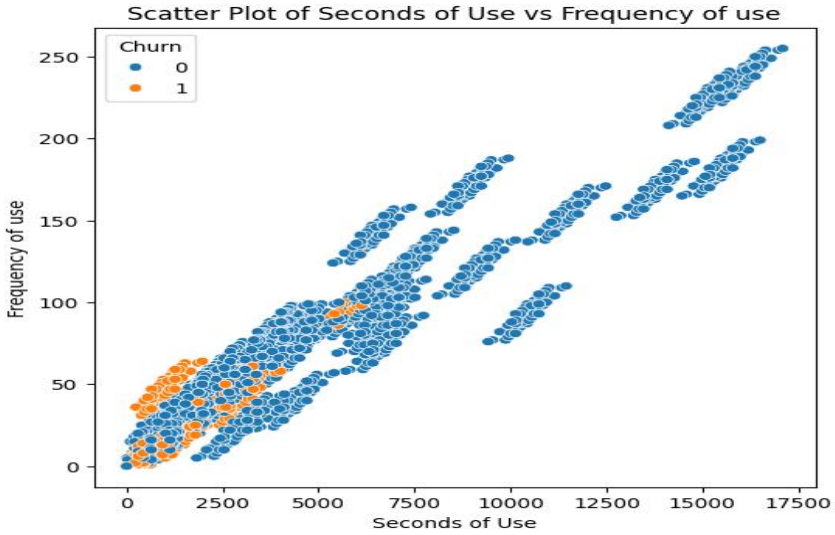


Figure 8. Scatter Plot of Seconds of Use vs Frequency of Use

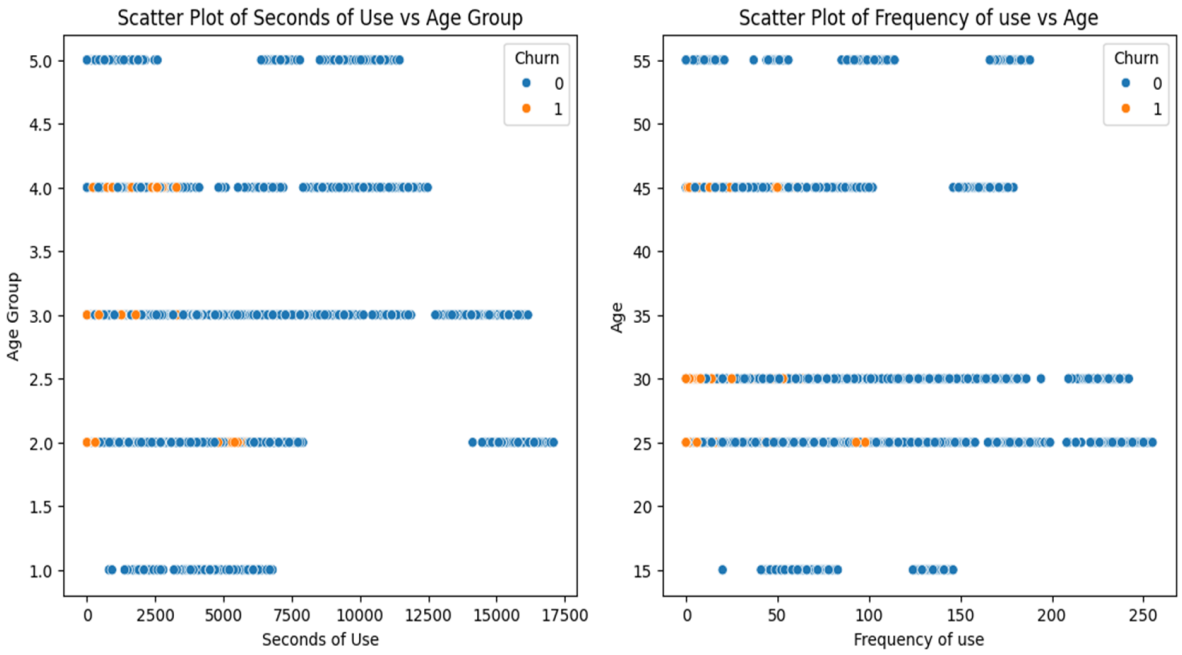


Figure 9. Scatter Plot of Seconds of Use vs Age and Age Groups

Interestingly, as depicted in Figure 10, there is a noticeable correlation between the number of SMS messages sent and customer value. According to the correlation matrix, SMS usage has the highest correlation with customer value.

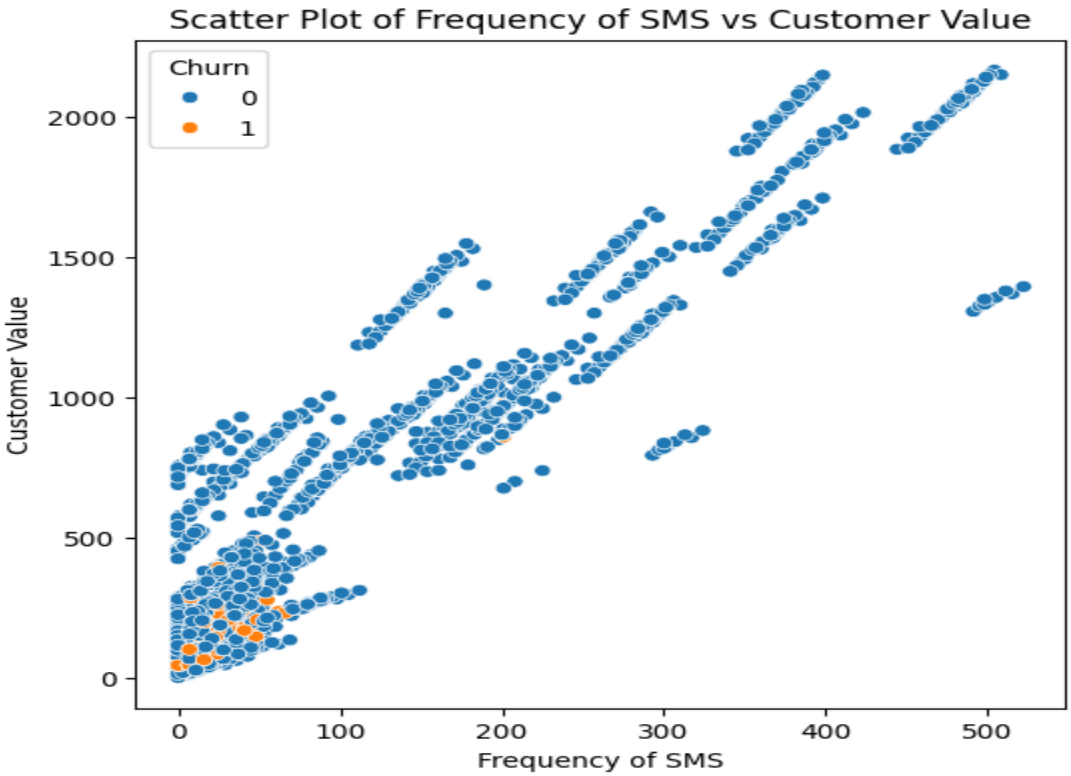


Figure 10. Scatter Plot of Frequency of SMS vs Customer Value

Continuing with the discussion of customer value, it is clear that individuals aged 25 and 30 are the company's most valuable customers, as shown in Figure 11.

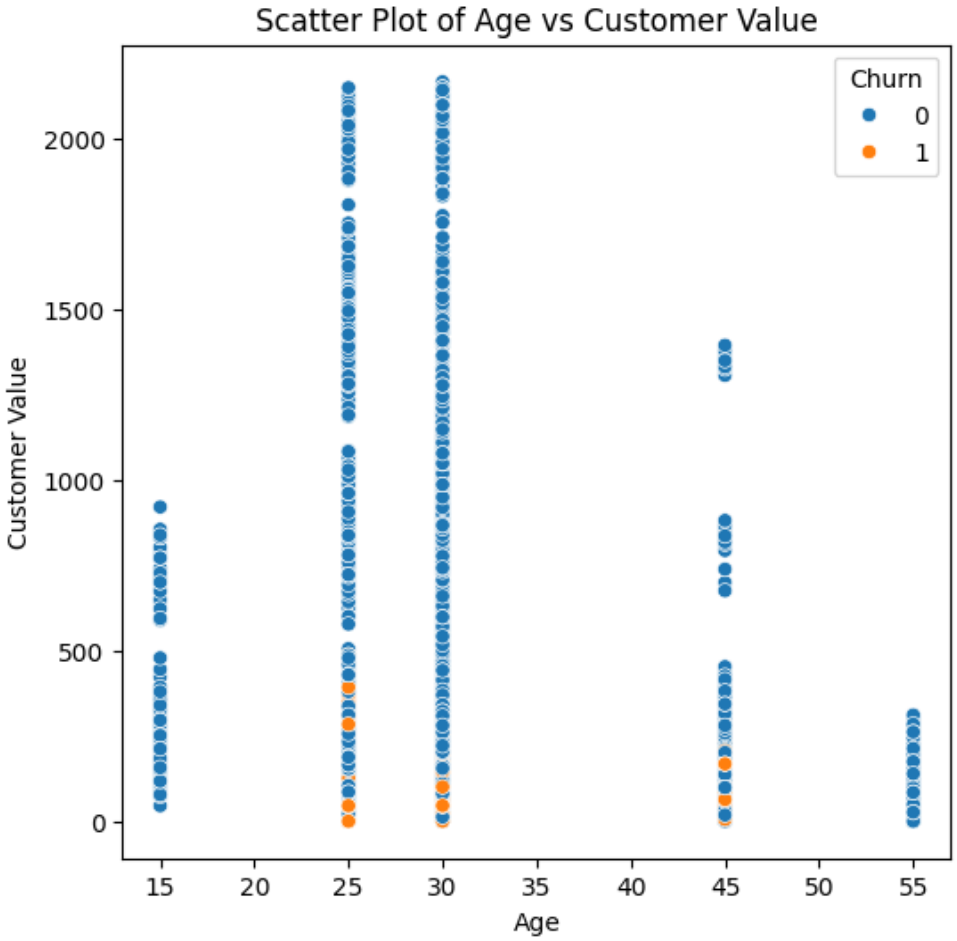
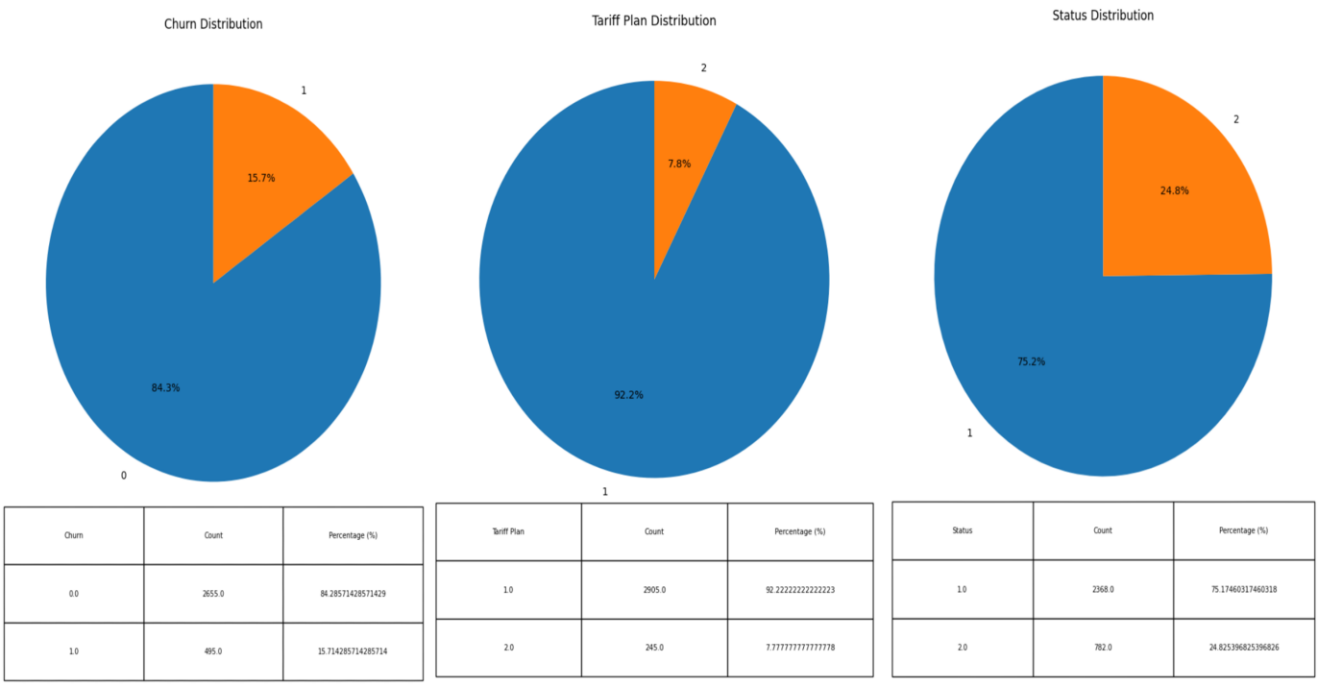


Figure 11. Scatter Plot of Age vs Customer Value

Finally, Figures 12 display the distribution of the churn feature along with Tariff Plan and Status.



Churn	Count	Percentage (%)
0.0	2655.0	84.28571428571429
1.0	495.0	15.714285714285714

Tariff Plan	Count	Percentage (%)
1.0	2905.0	92.22222222222223
2.0	245.0	7.777777777777778

Status	Count	Percentage (%)
1.0	2368.0	75.17460317460318
2.0	782.0	24.825396825396826

Figure 12. Distribution of Churn, Tariff Plan and Status

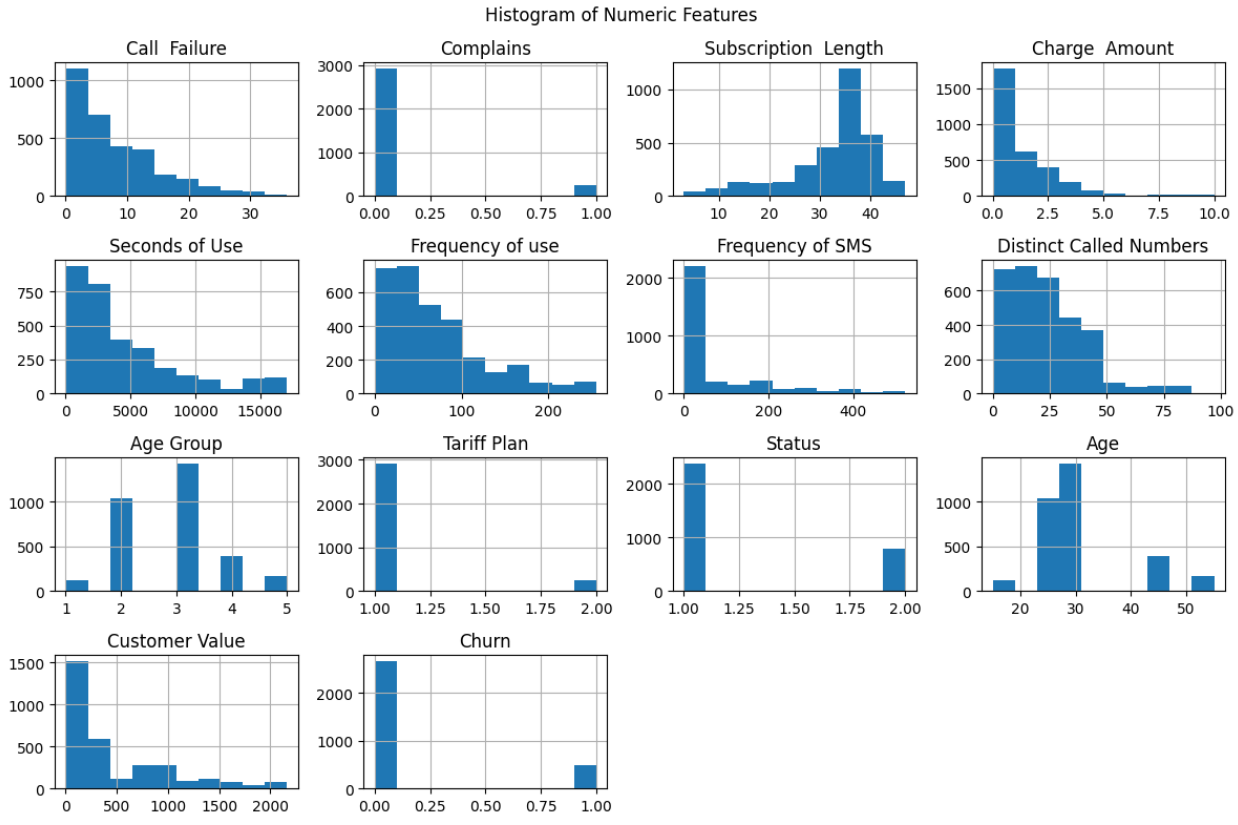


Figure 13. Histogram of Numeric Features

2. Preprocessing

The preprocessing phase involved several key steps to prepare the dataset for analysis and modeling. First, the dataset was checked for null values, though none were present. Outlier detection was conducted, and identified outliers were either removed or handled on a class-by-class basis to preserve meaningful variations. Data normalization was applied using appropriate scaling techniques to standardize the features. Finally, relevant features were selected, and the dataset was split into training and testing sets to ensure reliable model evaluation.

2.1. Handling Null Values

The documentation of the dataset and the initial data exploration confirmed that the dataset contains no null values. This ensured that no imputation or handling for missing data was necessary, allowing us to focus on other preprocessing tasks, such as outlier detection and feature normalization, to optimize the model's performance

2.2. Outlier Detection and Removal

Outliers can significantly impact machine learning models, especially those sensitive to extreme values. In this dataset, we identified a total of 154 outliers distributed across the target classes. Class 1 had 81 outliers, while Class 0 had 73 outliers.

To identify potential outliers, we examined specific patterns in the data. For example, we focused on records where the values of certain columns were consistently zero. Specifically, we observed that when "Seconds of Use" (Column 5) was zero, other columns related to usage behavior ("Frequency of use" and "Frequency of SMS") also tended to be zero. We hypothesized that these patterns might indicate abnormal usage behavior or possibly incorrect data entries and we implemented this code to find them.

```
# Access relevant columns
column_0 = dataset.iloc[:, 0]
column_4 = dataset.iloc[:, 4]
column_5 = dataset.iloc[:, 5]
column_13 = dataset.iloc[:, 13]

# Calculate zero counts and indices for Churn value 0
zero_count_ch0 = ((column_0 == 0) & (column_4 == 0) & (column_5 == 0) & (column_13 == 0)).sum()
print(f"Number of records with Churn value 0 in specified columns: {zero_count_ch0}")

zero_indices_ch0 = dataset[((column_0 == 0) & (column_4 == 0) & (column_5 == 0) & (column_13 == 0))].index.tolist()
print(f"Index of records with Churn value 0 in specified columns: {zero_indices_ch0}")

# Calculate zero counts and indices for Churn value 1
zero_count_ch1 = ((column_0 == 0) & (column_4 == 0) & (column_5 == 0) & (column_13 == 1)).sum()
print(f"Number of records with Churn value 1 in specified columns: {zero_count_ch1}")

zero_indices_ch1 = dataset[((column_0 == 0) & (column_4 == 0) & (column_5 == 0) & (column_13 == 1))].index.tolist()
print(f"Index of records with Churn value 1 in specified columns: {zero_indices_ch1}")
```

Figure 14. Identifying Outliers

To validate this, we calculated the deviation of these values from the global statistics of the entire dataset, including mean, median, and standard deviation, to evaluate how far these records deviated from the norm. Here's a summary of the global statistics:

- Mean: This metric provided an average value for each column, which we used as a reference to detect deviations in outlier records.

- Deviation from Mean: This measure helped identify how far a value deviates from the mean, enabling us to detect outliers more precisely.

$$\text{Deviation from Mean} = \text{Value} - \text{Mean}$$

- Median: The median allowed us to identify central tendencies in cases where the data was skewed, giving us insight into typical values that could contrast with outliers.

- Deviation from Median: This metric highlighted differences between a value and the central tendency in skewed distributions.

$$\text{Deviation from Median} = \text{Value} - \text{Median}$$

- Standard Deviation (std): This measure indicated the spread of values. We calculated how many standard deviations away from the mean each record's values were, marking records with extremely low or high standard deviations as potential outliers.

$$\text{Standard Deviation} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - Mean)^2}$$

- Standard Deviation from Mean **(z-score)**: This normalized measure quantified how far a value deviated from the mean in terms of standard deviations.

$$\text{Z-score} = \frac{\text{Value} - \text{Mean}}{\text{Standard Deviation}}$$

```python
# Select the relevant columns based on the initial code
column_0 = dataset.iloc[:, 0]
column_4 = dataset.iloc[:, 4]
column_5 = dataset.iloc[:, 5]



# Filter records based on specified conditions and get the indices
filtered_rows = dataset[(column_0 == 0) & (column_4 == 0) & (column_5 == 0)]

# Calculate global statistics for each of the specified columns
stats = {
    'mean': dataset.mean(),
    'median': dataset.median(),
    'std': dataset.std(),
    'min': dataset.min(),
    'max': dataset.max()
}

# Display statistics for reference
print("Global Dataset Statistics:")
print(pd.DataFrame(stats))

# Function to identify how far each value in the filtered rows is from the dataset's overall stats
def compare_to_global_stats(row, global_stats):
    deviations = {}
    for column in row.index:
        deviations[column] = {
            'value': row[column],
            'deviation_from_mean': row[column] - global_stats['mean'][column],
            'deviation_from_median': row[column] - global_stats['median'][column],
            'std_dev_from_mean': (row[column] - global_stats['mean'][column]) / global_stats['std'][column] if global_stats['std'][column] != 0 else 0
        }
    return deviations

# Compare each row in the filtered dataset to the global statistics and display the results
outlier_analysis = []
for idx, row in filtered_rows.iterrows():
    print(f"\nAnalysis for Row Index {idx}:")
    deviations = compare_to_global_stats(row, stats)
    outlier_analysis.append(deviations)
    print(pd.DataFrame(deviations).transpose())
```

Figure 15. Calculating Global and Outlier Statistics

For example, we analyzed record #18 in-depth, confirming that it deviated significantly across the following features:

- **Seconds of Use**: -1.065 standard deviations from the mean

- **Frequency of Use**: -1.209 standard deviations from the mean
- **Distinct Called Numbers**: -1.365 standard deviations from the mean
- **Customer Value**: -0.910 standard deviations from the mean

This in-depth comparison solidified the conclusion that records meeting these criteria had unusually low engagement, usage, or value, indicating that they did not align with the overall trends of the dataset.

2.3. Handling Outliers by Class

To understand the impact of outliers on model performance, we tested various configurations using a CART decision tree classifier. These included:

1. Retaining all records.
2. Removing all outliers across both classes.
3. Selectively removing outliers from only one class at a time.
4. Replacing outliers in one or both classes with mean or median values.
5. Using clustering-based techniques to identify and replace outliers.

Our experiments showed that the best results occurred when we selectively removed **Class 0** outliers. The dataset exhibited significant class imbalance (84% Class 0 vs. 16% Class 1), and **Class 1** outliers represented a substantial portion of the total **Class 1** instances. Removing these outliers entirely risked discarding valuable information. In contrast, **Class 0** outliers were more numerous and had a greater influence on model performance. By focusing on **Class 0**, we retained the integrity of the **Class 1** distribution and improved overall accuracy. Here are the results of each method with CART:

		Precision	Recall	F1-Score	Support
Default CART	0	0.95	0.96	0.96	531
	1	0.77	0.76	0.77	99
	Accuracy		0.93		630
	Macro		0.86		630
	Weighted		0.93		630
Class 1 Removed CART	0	0.98	0.98	0.98	531
	1	0.88	0.84	0.86	83
	Accuracy		0.96		614
	Macro	0.93	0.91	0.92	614
	Weighted		0.96		614
Class 0 Removed CART	0	0.97	0.97	0.97	517
	1	0.87	0.87	0.87	99
	Accuracy		0.96		616
	Macro		0.92		616

	Weighted	0.96			616
Class 0 and 1 Removed CART	0	0.97	0.98	0.98	517
	1	0.88	0.82	0.85	83
	Accuracy	0.96			600
	Macro	0.93	0.90	0.91	600
	Weighted	0.96			600
Class 0 and 1 Mean Replaced CART	0	0.97	0.98	0.97	531
	1	0.87	0.84	0.86	99
	Accuracy		0.96		630
	Macro	0.92	0.91	0.91	630
	Weighted	0.95	0.96	0.96	630
Class 0 and 1 Median Replaced CART	0	0.97	0.98	0.97	531
	1	0.86	0.83	0.85	99
	Accuracy		0.95		630
	Macro	0.92	0.90	0.91	630
	Weighted	0.95	0.95	0.95	630
Class 0 Removed and 1 Mean Replaced CART	0	0.97	0.98	0.98	517
	1	0.90	0.84	0.87	99
	Accuracy		0.96		616
	Macro	0.94	0.91	0.92	616
	Weighted	0.96	0.96	0.96	616
Class 1 Removed and 0 Mean Replaced CART	0	0.97	0.96	0.97	531
	1	0.77	0.81	0.79	83
	Accuracy		0.94		614
	Macro	0.87	0.88	0.88	614
	Weighted	0.94	0.94	0.94	614

Class 0 Removed and 1 Median Replaced CART	0	0.96	0.98	0.97	517
	1	0.88	0.81	0.84	99
	Accuracy		0.95		616
	Macro	0.92	0.89	0.91	616
	Weighted	0.95	0.95	0.95	616
Class 1 Removed and 0 Median Replaced CART	0	0.97	0.98	0.97	531
	1	0.84	0.82	0.83	83
	Accuracy		0.95		614
	Macro	0.91	0.90	0.90	614
	Weighted	0.95	0.95	0.95	614
Clustering Mean Replaced CART	0	0.97	0.99	0.98	531
	1	0.93	0.85	0.89	99
	Accuracy		0.97		630
	Macro	0.95	0.92	0.93	630
	Weighted	0.97	0.97	0.97	630
Clustering Median Replaced CART	0	0.97	0.98	0.98	531
	1	0.88	0.85	0.87	99
	Accuracy		0.96		630
	Macro	0.93	0.91	0.92	630
	Weighted	0.96	0.96	0.96	630

Removing **Class 0** outliers while retaining **Class 1** resulted in the highest balance between accuracy and class-specific metrics, achieving an average score of **0.92**. Meanwhile, clustering-based mean replacement showed potential, particularly for improving Class 1 recall, though it required more tuning to outperform selective removal.

This outlier handling strategy enhanced model accuracy while preserving critical information from the minority class, ultimately producing a cleaner dataset better suited for robust model building. Further exploration of hybrid methods, combining removal and clustering-based replacement, could provide additional improvements in future studies.

This outlier handling strategy helped improve model accuracy while balancing the need to retain sufficient data for effective model training. This preprocessing approach ultimately led to a cleaner dataset better suited for building a robust model.

2.4. Data Normalization

Normalization was a crucial step in this project, as the dataset contained columns with varied scales and distributions. Different normalization techniques were applied based on the nature and distribution of each feature to standardize values across columns.

1. Min-Max Scaling: For columns with high ranges and significant skewness, we applied Min-Max Scaling to bring all values into a range between 0 and 1. This technique preserved the shape of the data distribution while scaling it down, which is beneficial for machine learning models, especially when features have vastly different scales. The columns normalized using Min-Max Scaling were:

- Call Failure
- Charge Amount
- Seconds of Use
- Frequency of use
- Frequency of SMS
- Distinct Called Numbers
- Customer Value

By applying Min-Max Scaling, we transformed these columns so that their minimum value became 0 and maximum value became 1, while maintaining proportional differences between values.

2. Z-score Normalization (Standard Scaling): For columns with distributions closer to normal, we applied Z-score normalization, which standardized values by centering them around the mean and scaling by the standard deviation. This method transformed values to have a mean of 0 and a standard deviation of 1, effectively handling columns where data was approximately normally distributed. Z-score normalization was applied to:

- Subscription Length
- Age

This method is particularly effective when data is normally distributed, as it brings features to a comparable scale while preserving relative differences between them.

By combining these approaches, we ensured that our dataset was optimally scaled, enhancing the performance and interpretability of machine learning models applied to it. Min-Max Scaling made features with a high range and skewed distribution suitable for model input, while Z-score normalization provided stability for normally distributed features. This preprocessing step laid the groundwork for effective model training and accurate predictions, though we should note that some algorithms are inherently unaffected by values that are not normalized.

```
# Columns to normalize with Min-Max Scaling
min_max_cols = ['Call  Failure', 'Charge  Amount', 'Seconds of Use',
                'Frequency of use', 'Frequency of SMS',
                'Distinct Called Numbers', 'Customer Value']

# Columns to normalize with Z-score
z_score_cols = ['Subscription  Length', 'Age']

# Apply Min-Max Scaling
min_max_scaler = MinMaxScaler()
dataset[min_max_cols] = min_max_scaler.fit_transform(dataset[min_max_cols])

# Apply Z-score Normalization
z_score_scaler = StandardScaler()
dataset[z_score_cols] = z_score_scaler.fit_transform(dataset[z_score_cols])

# The normalized dataset is now updated
print(dataset.head())
```

Figure 16. Normalization of Dataset

2.5. Data Selection and Separation

Once the data was preprocessed and ready for modeling, the next step was to properly split the dataset into training and testing sets. This process is critical for building a robust model, as it helps evaluate the model's performance on unseen data, ensuring it generalizes well to new cases. Below are the steps and considerations involved in splitting the dataset:

1. Defining Features and Target Variables

The first step in data preparation is to define the feature columns (X) and the target column (y). In this case, the target variable is **Churn**, which indicates whether a customer churned (1) or not (0). The rest of the columns in the dataset represent the features, which include various customer behaviors and characteristics such as **Call Failure**, **Subscription Length**, **Age**, and others.

2. Splitting the Data

To evaluate the model's performance, it is necessary to split the dataset into training and testing sets. The training set is used to train the machine learning model, while the testing set is reserved for evaluating the model's performance on new, unseen data.

A common approach is to split the data into a **training set** and a **test set**, with the test set typically comprising around 20% of the total data. This split ensures that the model is exposed to enough data to learn from while also leaving enough unseen data to evaluate its generalization performance. In this case, we split the dataset into 80% for training and 20% for testing:

3. Maintaining Class Proportions (Stratified Split)

An important consideration when splitting the data is to maintain the proportions of the target variable across both the training and testing sets. In this case, **Churn** is an imbalanced classification problem, with a majority of customers not churning (Class 0) and a smaller portion of customers churning (Class 1). If we split the data randomly, the proportion of these classes could change, which might negatively impact the model's ability to learn and generalize.

To address this, we used **stratified sampling**, a technique that ensures the class distribution in both the training and test sets mirrors the original dataset's distribution. This approach helps prevent bias in training and ensures the model is tested on representative samples of both classes.

The proportions of **Class 0** and **Class 1** were as follows:

- **Training set proportions**:
  - Class 0: 2065 (83.9%)
  - Class 1: 396 (16.1%)
- **Test set proportions**:
  - Class 0: 517 (83.9%)
  - Class 1: 99 (16.1%)

As shown, the proportions in both sets were very similar to those in the original dataset, indicating that stratified sampling was successfully applied. This ensures that both training and test sets contain an appropriate mix of churn and non-churn instances, which is critical for training a model that can accurately predict churn.

```
# Define feature columns (X) and target column (y)
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

# Split the dataset into training and testing sets
def custom_train_test_split(X, y, test_size=0.2, stratify=None, random_state=None):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, stratify=stratify, random_state=random_state)
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = custom_train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# Function to print class proportions
def print_proportions(y):
    unique, counts = np.unique(y, return_counts=True)
    total = len(y)
    for val, count in zip(unique, counts):
        print(f"Class {val}: {count} ({count/total*100:.1f}%)")

print("\nTraining set proportions:")
print_proportions(y_train)
print("\nTest set proportions:")
print_proportions(y_test)
```

Figure 17. Building Training and Test Sets

3. Modeling

After preprocessing the dataset to ensure data quality and consistency, including steps like handling outliers, normalization, and class balancing, the next phase in our machine learning workflow is to develop and evaluate various models. This phase aims to determine which algorithms can best identify patterns and predict customer churn with high accuracy.

Our primary goal with modeling is to explore and analyze a range of classifiers, from simpler, interpretable models to more complex ensemble methods, each offering different strengths and trade-offs.

By testing a sequence of models, we can progressively address limitations observed in earlier approaches and move toward achieving higher accuracy and robustness.

3.1. Decision Tree

We began with basic decision tree algorithms, which are intuitive and interpretable but prone to overfitting. As we moved toward more sophisticated ensemble techniques like Random Forest, Gradient Boosting, and AdaBoost, we added layers of complexity and robustness, with each method attempting to mitigate the weaknesses of the previous models. This journey, from individual trees to ensembles, reflects a common path in predictive modeling—beginning with simpler baselines to understand the data structure, then advancing to methods that reduce variance and enhance generalization.

Below is a detailed breakdown of each model evaluated, providing insights into how they function, their strengths, weaknesses, and why they are suitable (or less suitable) for the task of predicting customer churn in this dataset.

- CART Classifier (Classification and Regression Tree) - Gini Impurity

Overview
The Classification and Regression Tree (CART) algorithm is a decision tree method that creates binary splits in the data. In classification tasks, it measures the "purity" of nodes using Gini impurity, which aims to minimize the probability of incorrect classifications at each split.

How It Works
CART iteratively splits data based on feature values to create two child nodes. Each split is chosen to maximize the homogeneity of the resulting child nodes, meaning that each node ideally contains observations from a single class. The splitting continues until certain stopping criteria are met, such as a maximum depth or minimum samples per node.

Strengths

- **Interpretability**: Decision trees are easily interpretable as they visually show the rules learned from the data.
- **No Need for Feature Scaling**: Decision trees do not require normalized or standardized data, as splits are made based on raw feature values.
- **Handles Non-Linear Relationships**: CART can handle non-linear relationships between features and targets by creating hierarchical splits.

Weaknesses

- **Overfitting**: CART is prone to overfitting, especially when the tree is grown to its full depth, capturing noise in the data.
- **Instability**: Small changes in data can lead to large changes in the structure of the tree.
- **Bias Towards Features with More Levels**: CART might favor features with many unique values, leading to biased splits.

- ID3 Classifier - Information Gain (Entropy)

Overview
ID3 (Iterative Dichotomiser 3) is an earlier decision tree algorithm that uses entropy to evaluate splits.

Entropy measures the amount of uncertainty or disorder in the data, and the algorithm chooses splits that maximize information gain.

How It Works
ID3 chooses features to split based on the highest information gain, which is the reduction in entropy (or disorder) after the split. This process continues recursively until the data is perfectly classified or other stopping criteria are met.

Strengths

- **Simple and Intuitive**: ID3 is straightforward and easy to understand.
- **Handles Non-Linear Data**: Like CART, ID3 can capture non-linear relationships by making successive splits.

Weaknesses

- **Overfitting**: ID3 is also prone to overfitting, especially when the tree depth is not restricted.
- **Bias Toward Features with Many Values**: ID3 may favor features with more levels (unique values), which can introduce biases in splits.
- **Computationally Expensive**: Calculating entropy and information gain for each feature at each split can be computationally intensive.

- C4.5 Classifier - An Enhancement of ID3

Overview
C4.5 is an improvement on ID3 that also uses entropy but addresses some of ID3's limitations. C4.5 is widely used as it optimizes the choice of splits and is less biased toward features with many unique values.

How It Works
Instead of pure information gain, C4.5 uses **gain ratio** to determine splits, which penalizes features with a large number of unique values, reducing bias. C4.5 also allows for multi-way splits rather than just binary splits, making it more flexible.

Strengths

- **Reduced Bias in Splits**: By using gain ratio, C4.5 reduces bias toward features with many unique values.
- **Pruning**: C4.5 includes a pruning step, which helps reduce overfitting by removing branches that do not improve the model's accuracy.
- **Handles Missing Values**: C4.5 can manage missing data by estimating the probable value of missing values based on other information.

Weaknesses

- **Complexity**: C4.5 is more complex and computationally expensive than ID3.
- **Memory Intensive**: Due to its calculations, C4.5 can be memory-intensive, especially with larger datasets.

- Extra Tree Classifier (Extremely Randomized Trees)

Overview
Extra Trees, or Extremely Randomized Trees, is a variant of decision trees where the splitting points are selected randomly rather than by an optimal criterion like Gini or entropy. It is often used in ensemble methods to add randomness to reduce overfitting.

How It Works
Extra Trees chooses random splits rather than optimizing them at each step, leading to a simpler and faster training process. This randomness helps reduce variance and can improve generalization by creating more diverse trees when used in an ensemble.

Strengths

- **Reduced Overfitting**: Extra Trees add randomness, which reduces the likelihood of overfitting.
- **Fast Training**: Since it does not optimize splits, Extra Trees are computationally faster to train than other decision trees.
- **Effective as an Ensemble**: Extra Trees show strong performance when combined with other models in an ensemble.

Weaknesses

- **Lower Interpretability**: The random splits make Extra Trees harder to interpret than traditional decision trees.
- **Reduced Accuracy as a Standalone Model**: Without optimization of splits, Extra Trees might have lower accuracy than fully optimized trees.

- Random Forest Classifier - Ensemble of Decision Trees

Overview
Random Forest is an ensemble learning method that builds multiple decision trees and combines their outputs to improve performance. It uses both bagging (sampling with replacement) and random feature selection to reduce variance and avoid overfitting.

How It Works
Random Forest generates multiple trees on different random samples of data and different subsets of features. The final prediction is made by averaging (for regression) or voting (for classification) across all the trees. This reduces the model's variance, making it more robust and accurate than a single decision tree.

Strengths

- **High Accuracy**: By combining multiple trees, Random Forest generally provides higher accuracy than a single tree.
- **Robust to Overfitting**: Due to ensemble averaging, Random Forest is less likely to overfit.
- **Feature Importance**: Random Forest provides insights into feature importance, making it useful for feature selection.

Weaknesses

- **Increased Complexity**: Random Forest is more complex than individual trees, making it harder to interpret.
- **Slower Predictions**: Predictions are slower than single-tree models since each input has to be processed by multiple trees.
- **Large Memory Requirement**: Random Forest requires more memory as it stores multiple trees.

- Gradient Boosting Classifier - Sequential Ensemble of Weak Learners

Overview
Gradient Boosting is a boosting algorithm that sequentially builds trees, each trying to correct the errors of the previous one. Instead of building trees independently, each new tree focuses on the residuals of the prior model, incrementally improving performance.

How It Works
Gradient Boosting fits new trees to the residual errors of the previous trees. The final prediction is a weighted sum of all trees. This approach helps focus on hard-to-predict instances, making Gradient Boosting highly accurate.

Strengths

- **High Accuracy**: Gradient Boosting often yields better accuracy by focusing on hard-to-predict examples.
- **Flexible**: It can use various loss functions, allowing customization for different tasks.

Weaknesses

- **Sensitive to Overfitting**: Gradient Boosting can overfit, especially if too many trees are used.
- **Slow Training**: Sequential training makes it slower compared to parallelizable methods like Random Forest.
- **Complex to Tune**: Gradient Boosting has many hyperparameters, which require careful tuning for optimal performance.

- AdaBoost Classifier - Adaptive Boosting with Decision Trees

Overview
AdaBoost is another boosting algorithm that combines weak learners, typically shallow trees. Each tree focuses more on misclassified points from the previous trees by assigning higher weights to these points.

How It Works
AdaBoost starts with equal weights on all data points. After each tree, the weights of misclassified points are increased, forcing the next tree to focus more on these hard examples. This adaptive process continues until the error is minimized or the maximum number of trees is reached.

Strengths

- **High Accuracy for Low-Complexity Models**: AdaBoost works well with weak learners and can yield high accuracy.

- **Focuses on Difficult Cases**: By focusing on hard-to-classify examples, AdaBoost improves on challenging areas.

Weaknesses

- **Sensitive to Noise**: AdaBoost may focus too heavily on noisy data, leading to overfitting.
- **Not Suitable for Highly Complex Learners**: AdaBoost generally works best with simple learners like shallow trees, as deeper trees may lead to overfitting.

- XGBoost - Extreme Gradient Boosting Algorithm

Overview

XGBoost is an optimized implementation of gradient boosting that uses advanced techniques to improve speed and performance. It is designed to handle large datasets efficiently and is highly flexible, making it a popular choice for machine learning competitions and real-world applications.

How ItsWorks

XGBoost builds an ensemble of decision trees in a sequential manner, where each tree attempts to correct the residual errors of the previous trees. It incorporates advanced features like regularization, sparsity awareness, and optimized memory usage to enhance performance. XGBoost uses a second-order approximation of the loss function, improving the precision of tree splitting and making it faster and more robust.

Key Features

- **Regularization**: Includes L1 and L2 regularization to reduce overfitting.
- **Sparsity Awareness**: Efficiently handles missing data and sparse inputs.
- **Parallelization**: Performs some operations in parallel, making training faster than traditional gradient boosting.
- **Tree Pruning**: Uses a "max depth" approach to prune trees backward, preventing unnecessary growth.
- **Custom Objectives**: Allows users to define custom loss functions for specific use cases.

Strengths

- **High Performance**: Offers superior accuracy and efficiency compared to other boosting algorithms.
- **Scalable**: Handles large datasets with millions of examples and features.
- **Flexibility**: Supports classification, regression, and ranking problems with customizable objectives.
- **Handles Missing Data**: Automatically learns the best way to handle missing values.

Weaknesses

- **Complexity**: Can be challenging to understand and implement due to its advanced features.
- **Sensitive to Hyperparameters**: Requires careful tuning of parameters like learning rate, max depth, and number of trees.
- **Computational Cost**: Though faster than traditional boosting, training can still be computationally expensive for very large datasets.

```python
# Function to train and evaluate a model
def train_and_evaluate_model(model, model_name):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    print(f"Results for {model_name}:")
    print("Accuracy:", accuracy_score(y_test, y_pred))
    print("Classification Report:\n", classification_report(y_test, y_pred))
    print("\n" + "="*50 + "\n")

# 1. CART (Classification and Regression Tree) - Gini impurity
cart_classifier = DecisionTreeClassifier(criterion='gini', random_state=42)
train_and_evaluate_model(cart_classifier, "CART Classifier (Gini)")

# 2. ID3 - Approximated using entropy as the criterion
id3_classifier = DecisionTreeClassifier(criterion='entropy', random_state=42)
train_and_evaluate_model(id3_classifier, "ID3 Classifier (Entropy)")

# 3. C4.5 - Approximated using entropy, similar to ID3
c4_5_classifier = DecisionTreeClassifier(criterion='entropy', random_state=42)
train_and_evaluate_model(c4_5_classifier, "C4.5 Classifier Approximation")

# 4. Extra Tree Classifier (another variant of decision trees)
extra_tree_classifier = ExtraTreeClassifier(random_state=42)
train_and_evaluate_model(extra_tree_classifier, "Extra Tree Classifier")

# 5. Random Forest Classifier
random_forest_classifier = RandomForestClassifier(random_state=42)
train_and_evaluate_model(random_forest_classifier, "Random Forest Classifier")

# 6. Gradient Boosting Classifier
gradient_boosting_classifier = GradientBoostingClassifier(random_state=42)
train_and_evaluate_model(gradient_boosting_classifier, "Gradient Boosting Classifier")

# 7. AdaBoost Classifier (with a Decision Tree as the base estimator)
adaboost_classifier = AdaBoostClassifier(estimator=DecisionTreeClassifier(), random_state=42)
train_and_evaluate_model(adaboost_classifier, "AdaBoost Classifier with Decision Tree")
```

Figure 18. Modeling

3.1.1. Improvements to the model

To maximize the accuracy and overall performance of the models, we performed hyperparameter tuning—a process that allows us to fine-tune the parameters of each model to identify the best configuration for the task at hand. While each model type (e.g., Decision Trees, Random Forest, Gradient Boosting, etc.) has a set of default hyperparameters, these defaults are rarely optimal. By adjusting parameters such as tree depth, the number of estimators, learning rate, and more, we can reduce issues like overfitting and underfitting, ultimately boosting prediction accuracy on unseen data. Here are the steps for Hyperparameter Tuning:

1. **Defining the Hyperparameter Grid**: For each model, we defined a grid of potential values for key hyperparameters. These included options such as:

- criterion: Determines the function to measure the quality of splits (e.g., 'gini' or 'entropy' for Decision Trees).
- max_depth: Specifies the maximum depth of the tree, helping to control overfitting.
- min_samples_split: The minimum number of samples required to split an internal node.
- n_estimators: The number of trees (or boosting rounds) in ensemble models like Random Forest and Gradient Boosting.
- learning_rate: Used in boosting methods to control the contribution of each additional tree.

2. **Applying Grid Search with Cross-Validation**: We used **GridSearchCV**, a powerful tool that performs an exhaustive search over specified hyperparameter values. For each model, GridSearchCV evaluated every possible combination of parameters from the defined grid using 5-fold cross-validation. This approach split the training data into five parts, training on four parts while validating on the fifth, rotating until every part has served as the validation set. Cross-validation helps to ensure that the model's performance is not dependent on a specific subset of the data, improving generalization.
3. **Selecting the Best Model Configuration**: After evaluating all parameter combinations, GridSearchCV identified the best-performing set of parameters based on accuracy. We then trained each model with this optimized configuration and evaluated its performance on the test set. This tuning process allowed each model to reach its potential, addressing specific limitations and making it more robust in predicting customer churn.

3.1.2. Improvements by Model

Below is a summary of the improvements applied to each model and the parameters that were optimized:

- **CART Classifier (Gini)**: The CART model was tuned by adjusting its criterion (gini vs. entropy), max_depth, and min_samples_split. This tuning helped reduce overfitting by limiting tree depth and setting a threshold for minimum splits.
- **ID3 Classifier (Entropy)**: The ID3 model, which approximates C4.5, was similarly optimized by tuning max_depth and min_samples_split. Since ID3 inherently uses entropy as its criterion, this tuning focused on managing tree complexity for better generalization.
- **Extra Tree Classifier**: This model, designed to create extremely randomized trees, was optimized by adjusting the max_depth and min_samples_split parameters. Extra Trees are naturally diverse, and tuning these parameters helped balance accuracy with generalization.
- **Random Forest Classifier**: For Random Forest, the number of trees (n_estimators), max_depth, and min_samples_split were tuned. Increasing the number of trees enhances the stability and accuracy of predictions, while controlling depth and splits reduces the risk of overfitting.
- **Gradient Boosting Classifier**: Gradient Boosting was fine-tuned by adjusting n_estimators, learning_rate, and max_depth. These parameters impact the model's ability to incrementally correct errors in previous trees. A lower learning rate combined with a larger number of trees generally improves performance but can increase computational cost.

- **AdaBoost Classifier with Decision Tree**: In the AdaBoost model, both the n_estimators and learning_rate parameters were tuned, along with max_depth of the base estimator (Decision Tree). AdaBoost adjusts misclassified instances by assigning them higher weights; tuning helped to prevent overfitting by limiting tree complexity and controlling the rate at which AdaBoost iterates over weak learners.

```
warnings.filterwarnings("ignore", category=UserWarning, module="skopt.optimizer.optimizer")

# Define classifiers and their respective parameter grids for tuning
classifiers_with_grids = {
    "CART": (DecisionTreeClassifier(random_state=42), {
        'max_depth': [5, 10, 15, None],
        'min_samples_split': [2, 5, 10]
    }),
    "ID3": (DecisionTreeClassifier(criterion='entropy', random_state=42), {
        'max_depth': [5, 10, 15, None],
        'min_samples_split': [2, 5, 10]
    }),
    "Extra Trees": (ExtraTreesClassifier(random_state=42), {
        'max_depth': [5, 10, 15, None],
        'min_samples_split': [2, 5, 10]
    }),
    "Random Forest": (RandomForestClassifier(random_state=42), {
        'n_estimators': [50, 100, 200],
        'max_depth': [5, 10, 15, None],
        'min_samples_split': [2, 5, 10]
    }),
    "Gradient Boosting": (GradientBoostingClassifier(random_state=42), {
        'n_estimators': [50, 100, 200],
        'learning_rate': [0.01, 0.1, 0.3],
        'max_depth': [3, 5, 10]
    })
}

# Initialize a dictionary to store results
results = {}

# Step 1: Evaluate each model without optimization
print("Evaluating models without optimization...\n")
for model_name, (clf, _) in classifiers_with_grids.items():
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    results[model_name] = {
        "Default": {
            "Accuracy": accuracy_score(y_test, y_pred),
            "Classification Report": classification_report(y_test, y_pred, output_dict=True)
        }
    }
    print(f"{model_name} - Default Accuracy: {results[model_name]['Default']['Accuracy']:.4f}")

# Step 2: Apply hyperparameter optimization
print("\nApplying hyperparameter optimization...\n")
for model_name, (clf, param_grid) in classifiers_with_grids.items():
    print(f"\nOptimizing {model_name}...\n")
    model_results = {}

    # Grid Search
    grid_search = GridSearchCV(clf, param_grid, cv=5, n_jobs=-1)
    grid_search.fit(X_train, y_train)
    model_results["Grid Search"] = {
        "Best Estimator": grid_search.best_estimator_,
        "Accuracy": accuracy_score(y_test, grid_search.best_estimator_.predict(X_test)),
        "Classification Report": classification_report(
            y_test, grid_search.best_estimator_.predict(X_test), output_dict=True
        )
    }

    # Randomized Search
    random_search = RandomizedSearchCV(clf, param_distributions=param_grid, n_iter=10, cv=5, random_state=42, n_jobs=-1)
    random_search.fit(X_train, y_train)
    model_results["Randomized Search"] = {
        "Best Estimator": random_search.best_estimator_,
        "Accuracy": accuracy_score(y_test, random_search.best_estimator_.predict(X_test)),
        "Classification Report": classification_report(
            y_test, random_search.best_estimator_.predict(X_test), output_dict=True
        )
    }

    # Bayesian Search
    try:
        bayes_search = BayesSearchCV(clf, search_spaces=param_grid, n_iter=32, cv=5, random_state=42, n_jobs=-1)
        bayes_search.fit(X_train, y_train)
        model_results["Bayesian Search"] = {
            "Best Estimator": bayes_search.best_estimator_,
            "Accuracy": accuracy_score(y_test, bayes_search.best_estimator_.predict(X_test)),
            "Classification Report": classification_report(
                y_test, bayes_search.best_estimator_.predict(X_test), output_dict=True
            )
        }
    except ValueError:
        print(f"{model_name} - Bayesian Search not applicable.")

    # Hyperband
    halving_search = HalvingRandomSearchCV(clf, param_distributions=param_grid, factor=2, cv=5, random_state=42, n_jobs=-1)
    halving_search.fit(X_train, y_train)
    model_results["Hyperband"] = {
        "Best Estimator": halving_search.best_estimator_,
        "Accuracy": accuracy_score(y_test, halving_search.best_estimator_.predict(X_test)),
        "Classification Report": classification_report(
            y_test, halving_search.best_estimator_.predict(X_test), output_dict=True
        )
    }

    results[model_name]["Optimized"] = model_results

# Step 3: Compare results
print("\nComparison of results:\n")
for model_name, result in results.items():
    default_accuracy = result["Default"]["Accuracy"]
    print(f"\n{model_name} - Default Accuracy: {default_accuracy:.4f}")
    for method, optimized_result in result["Optimized"].items():
        optimized_accuracy = optimized_result["Accuracy"]
        print(f"{model_name} ({method}) - Optimized Accuracy: {optimized_accuracy:.4f}")
```

Figure 19. Modeling using Hyperparameters

3.1.3. Results

The results of the modeling process were evaluated using several performance metrics to provide a comprehensive understanding of the model's effectiveness. These metrics include accuracy, precision, recall, and F1-scores for both classes (Class 0 and Class 1), as well as support values indicating the number of samples for each class.

Accuracy serves as an overall measure of the model's ability to correctly classify instances across all classes. However, to gain a deeper insight into the model's performance for each class, metrics like precision, recall, and F1-score were analyzed separately:

- Precision: Precision reflects the proportion of correctly predicted instances out of all instances predicted as belonging to a particular class. High precision indicates fewer false positives.

- Recall: Recall measures the proportion of correctly identified instances out of all true instances for a specific class. A high recall means fewer false negatives.

- F1-Score: The F1-score, the harmonic mean of precision and recall, balances these two metrics to provide a single performance measure, especially useful in cases of imbalanced datasets.

The support values for Class 0 and Class 1 reflect the number of actual samples in each class, offering insight into the class distribution within the dataset.

By analyzing these metrics, the model's strengths and areas for improvement were identified, ensuring a balanced evaluation of its predictive capabilities across both classes. Here are detailed tables for models ran on datasets mentioned.

- **Basic Decision Tree**

Classifier	Accuracy	Precision (Class 0)	Recall (Class 0)	F1-Score (Class 0)	Precision (Class 1)	Recall (Class 1)	F1-Score (Class 1)	Support (Class 0)	Support (Class 1)
CART (Gini)	0.9578	0.97	0.97	0.97	0.87	0.87	0.87	517	99
ID3 (Entropy)	0.9643	0.98	0.98	0.98	0.91	0.87	0.89	517	99
C4.5 Approx.	0.9643	0.98	0.98	0.98	0.91	0.87	0.89	517	99
Extra Trees	0.9594	0.97	0.98	0.98	0.88	0.87	0.87	517	99
Random Forest	0.9773	0.98	0.99	0.99	0.97	0.89	0.93	517	99
Gradient Boosting	0.9627	0.96	0.99	0.98	0.96	0.8	0.87	517	99
AdaBoost (DT)	0.9464	0.97	0.97	0.97	0.83	0.84	0.83	517	99
XGBoost	0.97	0.98	0.98	0.98	0.94	0.90	0.92	517	99

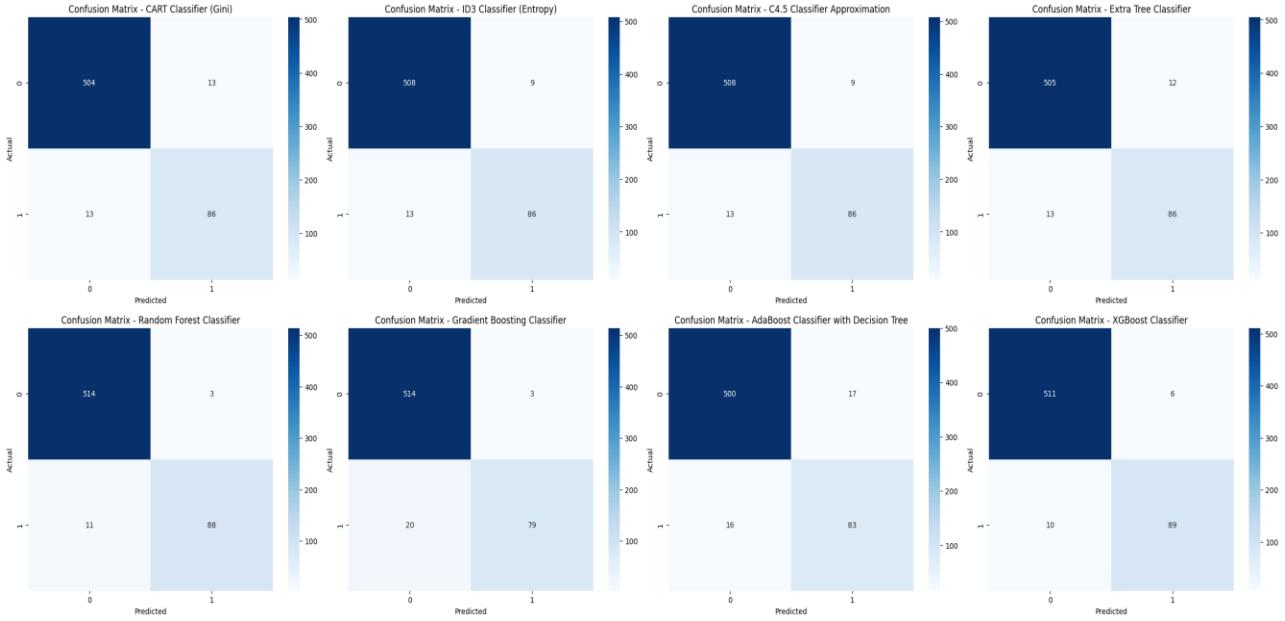


Figure 20. Confusion Matrices

- Hyperparameter Optimized

Classifier	Accuracy (Tuned)	Precision (Class 0)	Recall (Class 0)	F1-Score (Class 0)	Precision (Class 1)	Recall (Class 1)	F1-Score (Class 1)	Support (Class 0)	Support (Class 1)
CART (Gini)	0.9627	0.97	0.99	0.98	0.93	0.83	0.88	517	99
ID3 (Entropy)	0.9627	0.97	0.99	0.98	0.93	0.83	0.88	517	99
Extra Trees	0.9594	0.97	0.98	0.98	0.88	0.87	0.87	517	99
C4.5	0.9643	0.98	0.98	0.98	0.91	0.87	0.89	517	99
Random Forest	0.9773	0.98	0.99	0.99	0.97	0.89	0.93	517	99
Gradient Boosting	0.9605	0.98	0.99	0.99	0.97	0.91	0.94	517	99
AdaBoost (DT)	0.9421	0.99	0.99	0.99	0.96	0.93	0.94	517	99
XGBoost	0.97	0.98	0.99	0.98	0.94	0.94	0.92	517	99

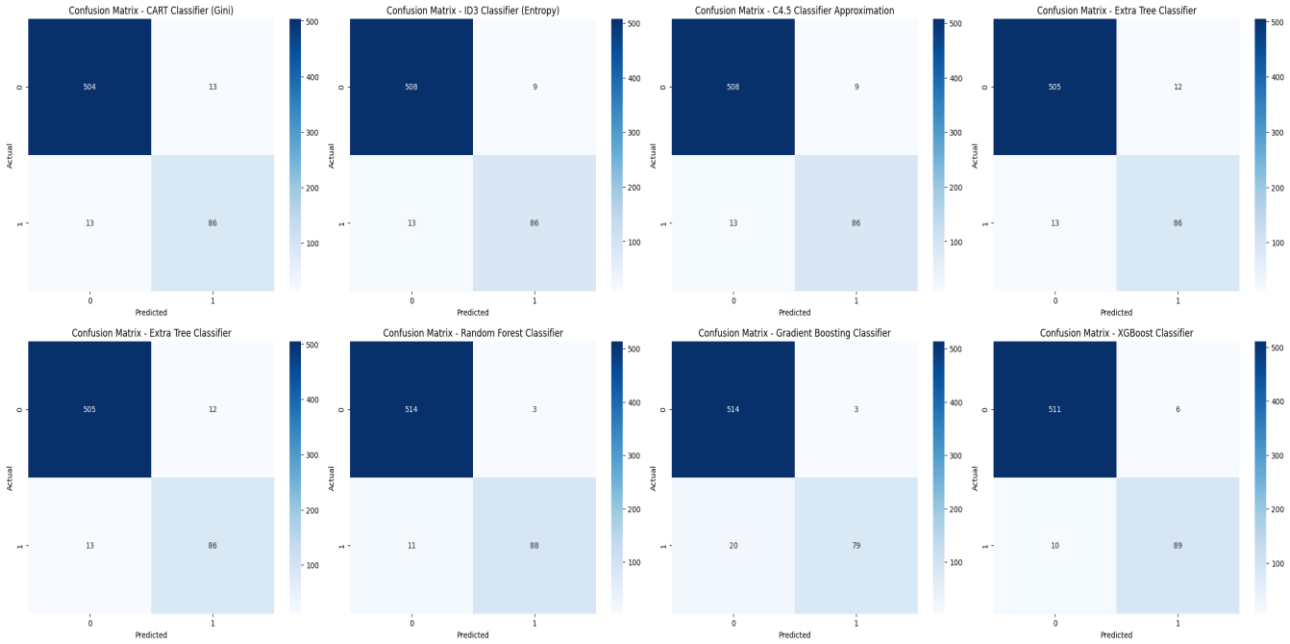


Figure 21. Confusion Matrices

3.1.4. Result Analysis

The modeling process revealed significant insights into churn prediction using various decision tree-based algorithms. Overall, decision tree ensembles such as Random Forest, Gradient Boosting, AdaBoost, and XGBoost consistently outperformed simpler tree-based methods like CART, ID3, and C4.5. Among the optimized models, Random Forest and XGBoost achieved the highest performance, with accuracies of 97.72% and 97.4%, respectively. These models demonstrated exceptional predictive power, particularly in handling the minority churn class (Class 1), which is crucial given the dataset's inherent imbalance. Gradient Boosting also performed strongly, achieving accuracy scores above 96% while maintaining a balance between precision and recall for both classes. In contrast, the simpler algorithms, while effective, were slightly less generalizable, which limited their overall predictive strength compared to ensemble methods.

Figure 22. Feature Importance Heatmap

The feature importance analysis revealed that certain features consistently played a critical role across all models. Features such as Status, Complains, and Frequency of Use emerged as the most impactful predictors of churn, as highlighted in the heatmap and bar graph. These features significantly influenced model performance, particularly in boosting the accuracy of ensemble-based algorithms like XGBoost. Other features, such as Tariff Plan and Customer Value, also contributed to the models' effectiveness, though to a lesser extent. Meanwhile, less significant features could be further analyzed to determine whether they can be simplified or removed altogether to improve computational efficiency without compromising predictive accuracy.

Despite the dataset's class imbalance, with Class 0 being significantly larger than Class 1, the models handled this challenge well. The optimized ensemble methods, particularly AdaBoost and Gradient Boosting, showed a strong ability to improve recall for the minority class while maintaining high precision and balanced F1-scores for both classes. These results underscore the value of ensemble techniques in addressing imbalance issues and ensuring robust classification performance across all customer groups.

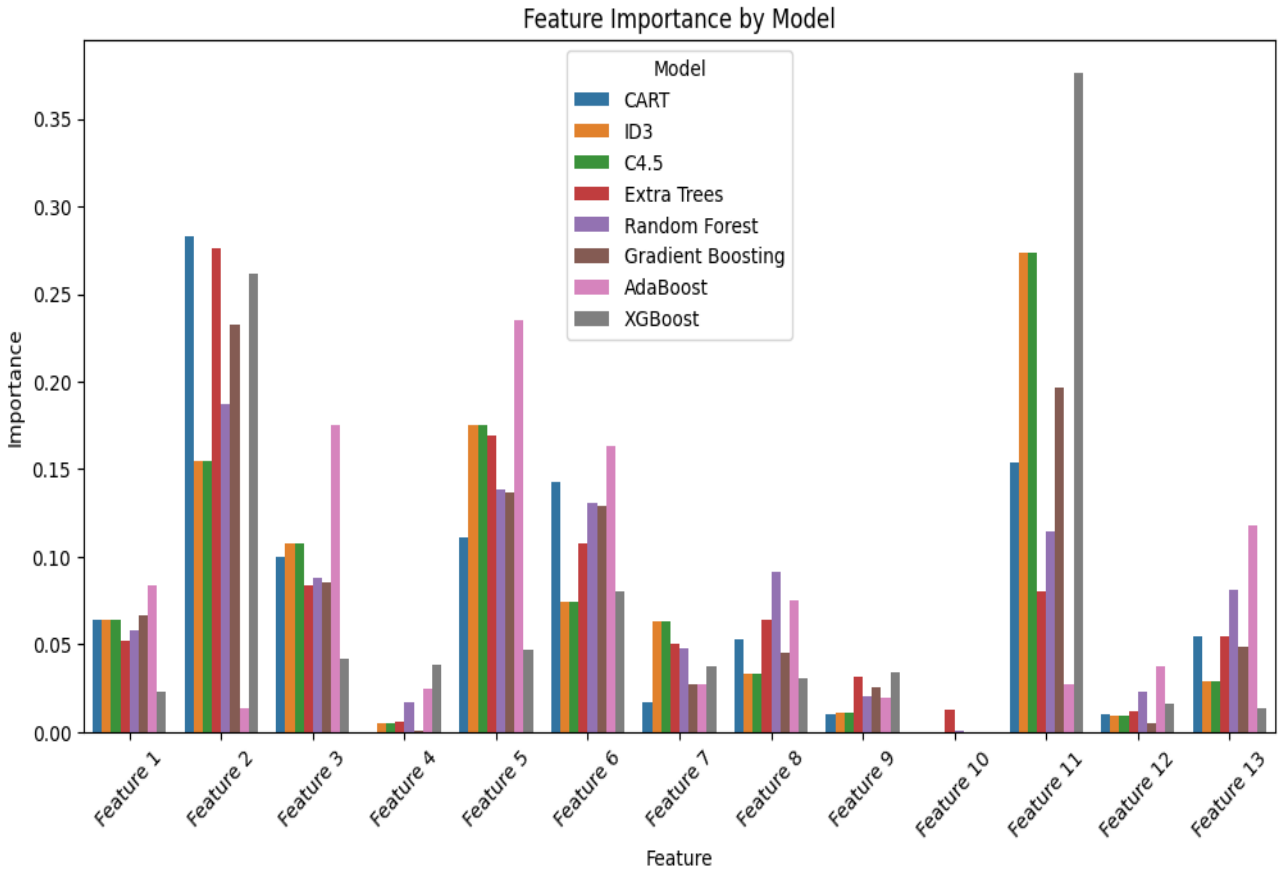


Figure 23. Feature Importance by Model

3.1.5. Tree Visualizations

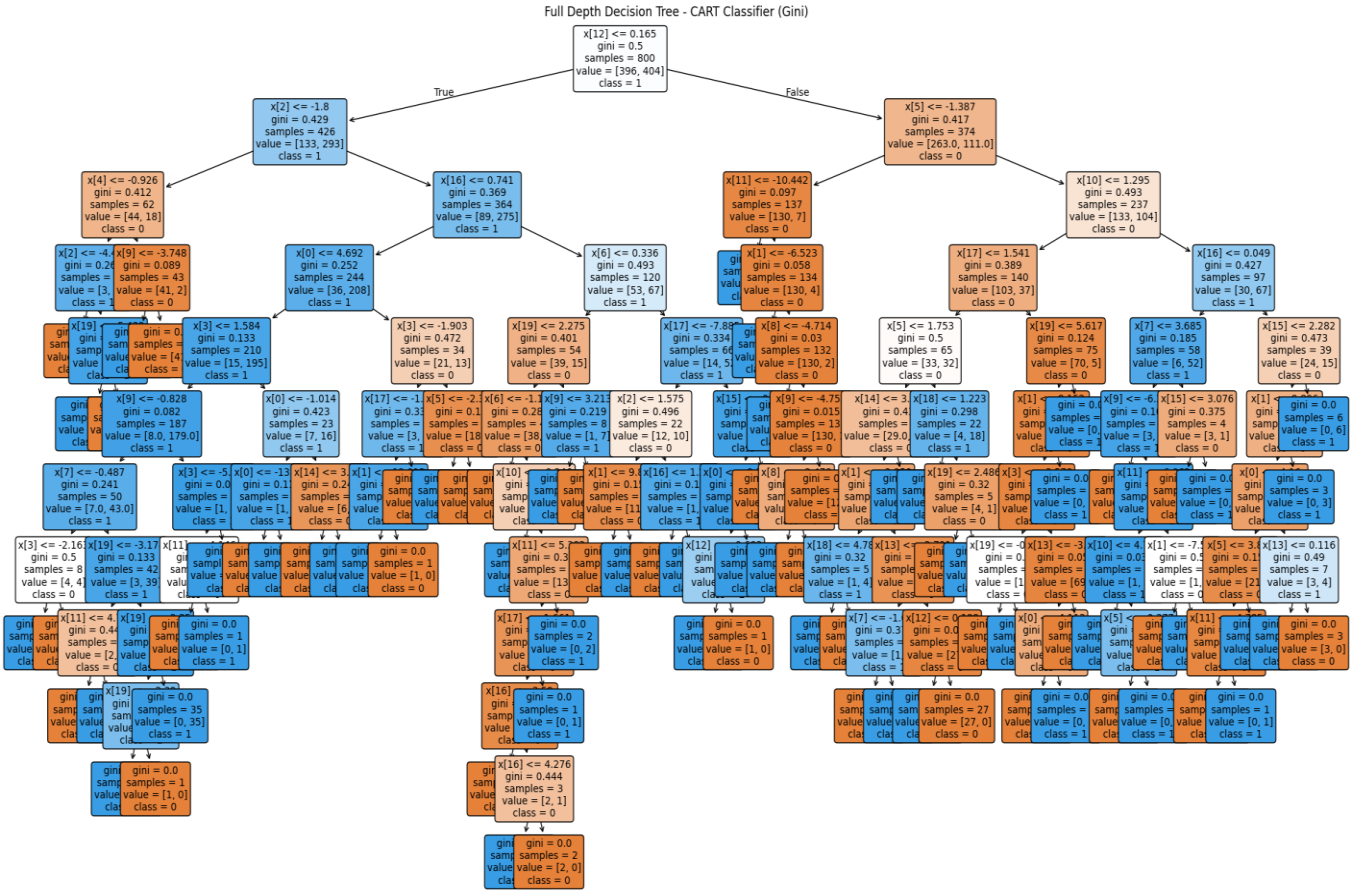


Figure 24. CART Decision Tree

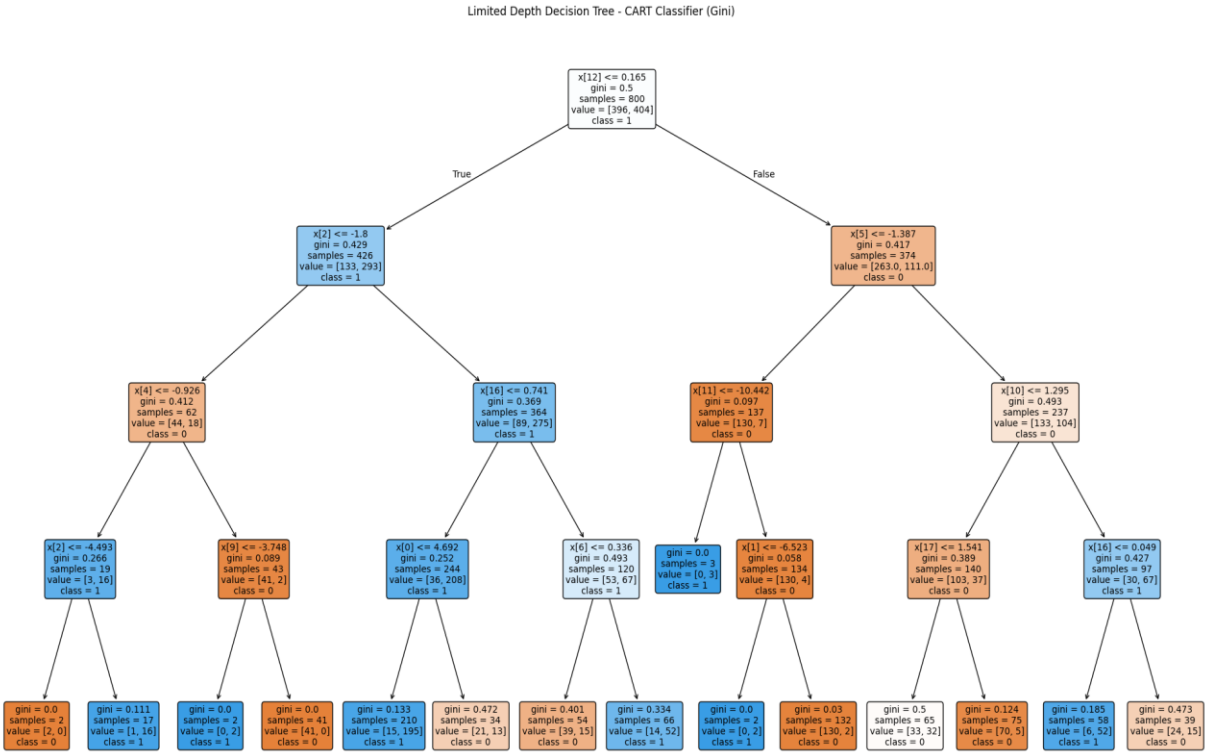


Figure 25. Mini CART Decision Tree

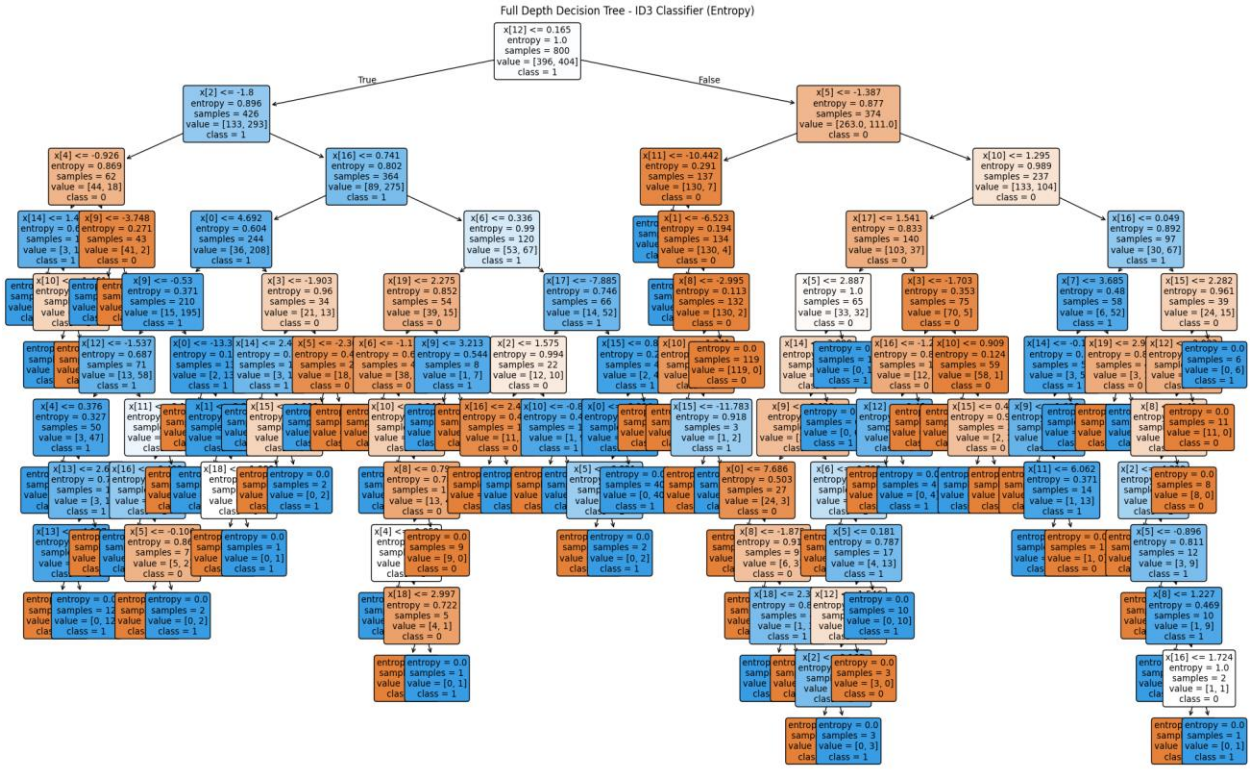


Figure 26. ID3 Decision Tree

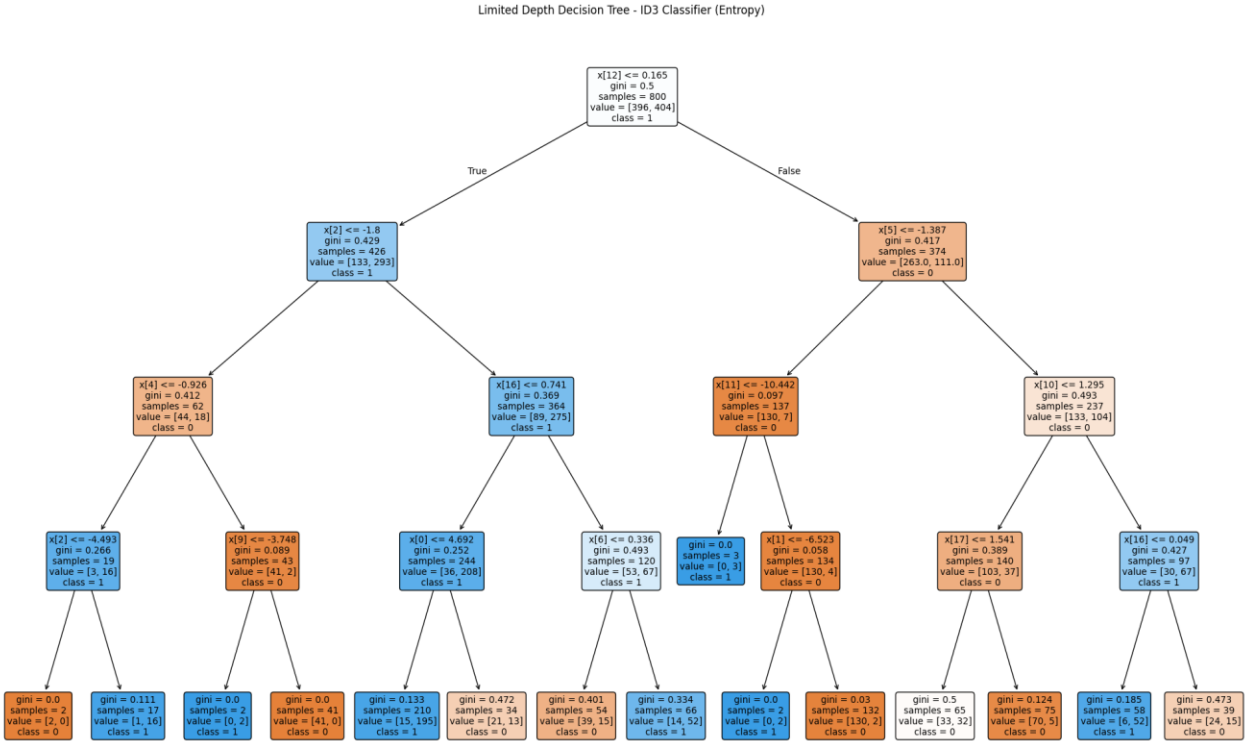


Figure 27. Mini ID3 Decision Tree

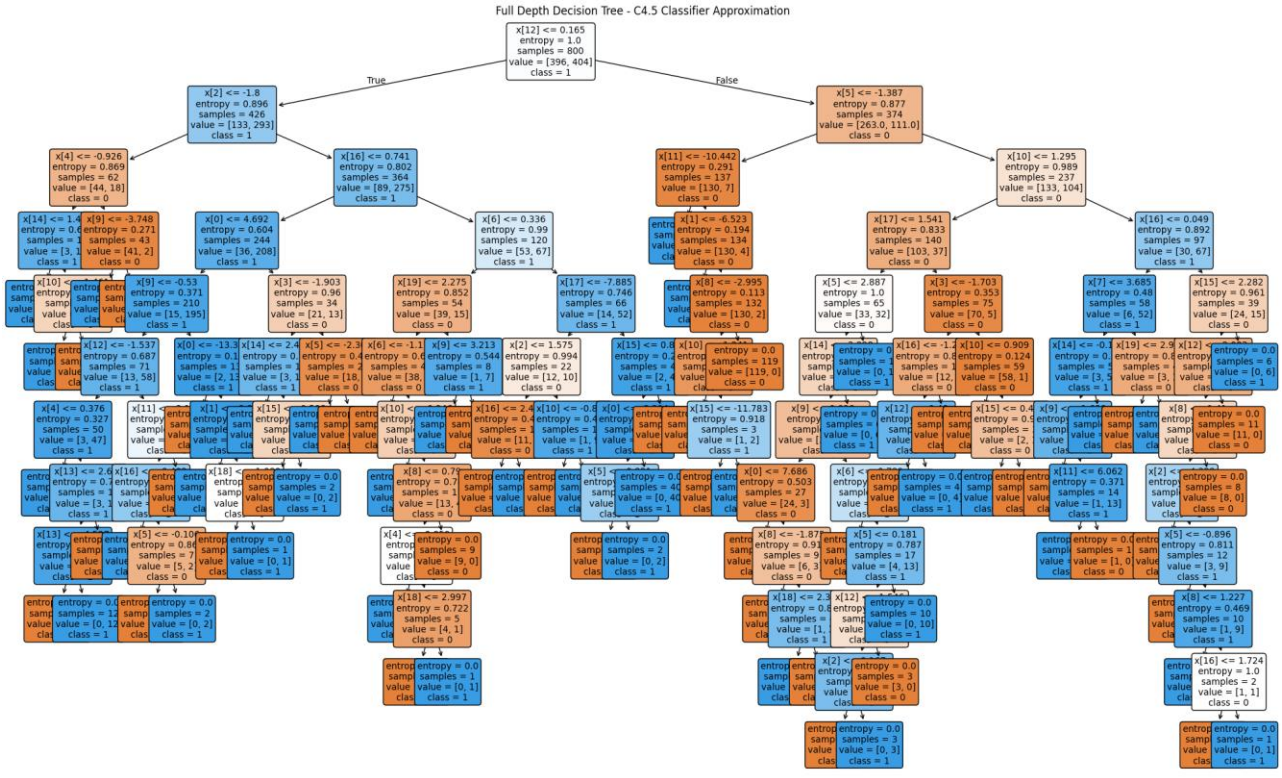


Figure 28. C4.5 Decision Tree

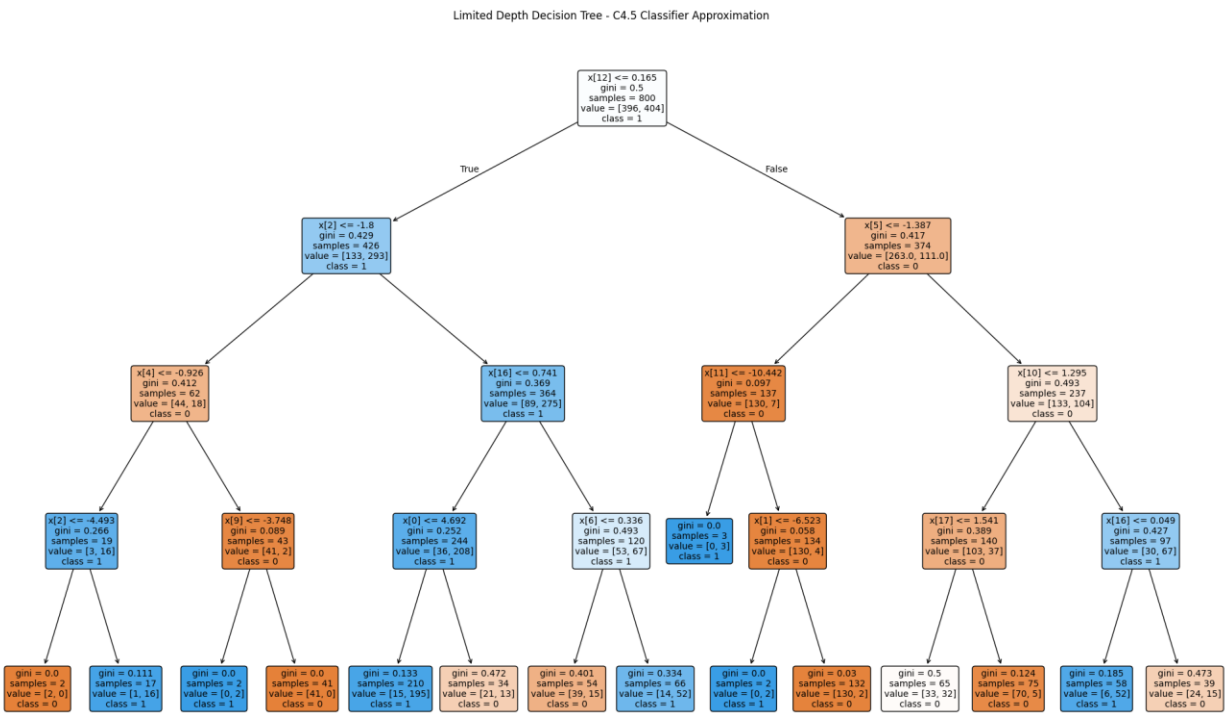


Figure 29. Mini C4.5 Decision Tree

3.2. KNN

The K-Nearest Neighbors (KNN) algorithm is a simple, non-parametric supervised learning method used for classification and regression tasks. It operates on the principle that similar instances exist in close proximity within the feature space. For classification, KNN assigns a class to a data point based on the majority class among its kkk nearest neighbors, where kkk is a user-defined parameter. The distance between data points is calculated using metrics such as Euclidean, Manhattan, or Minkowski distance.

KNN is effective for smaller datasets and non-linear relationships but can be computationally expensive for large datasets.

The provided code implements the KNN algorithm within a pipeline, performs hyperparameter optimization using GridSearchCV, evaluates the best model, and visualizes the results.

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier())
])

param_grid = {
    'knn__n_neighbors': [1, 3, 5, 7, 9, 11, 15, 21],
    'knn__weights': ['uniform', 'distance'],
    'knn__metric': ['euclidean', 'manhattan', 'chebyshev', 'minkowski'],
    'knn__p': [1, 2]
}

grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,          #
    n_jobs=-1,
    verbose=1
)

print("Starting Grid Search...")
grid_search.fit(X_train, y_train)
print("Grid Search Complete.\n")

results_df = pd.DataFrame(grid_search.cv_results_)

cols_of_interest = [
    'params',
    'mean_test_score',
    'std_test_score',
    'rank_test_score'
]
sorted_results_df = results_df[cols_of_interest].sort_values(
    by='mean_test_score',
    ascending=False
)

print("ALL CROSS-VALIDATION RESULTS (Sorted by mean_test_score):")
print(sorted_results_df.to_string(index=False), "\n")

print("BEST PARAMETERS (from cross-validation):", grid_search.best_params_)
print("BEST CV ACCURACY:", grid_search.best_score_, "\n")

best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

test_accuracy = accuracy_score(y_test, y_pred)
print("TEST ACCURACY (Best Model):", test_accuracy, "\n")

print("CONFUSION MATRIX (Best Model):")
print(confusion_matrix(y_test, y_pred), "\n")

print("CLASSIFICATION REPORT (Best Model):")
print(classification_report(y_test, y_pred))
```

Figure 30. Finding the best K value and Hyperparameter Tuning

The process begins with preparing the dataset. The target variable, which indicates whether a customer has churned, is separated from the feature set, and the data is split into training and testing subsets. To ensure fairness in evaluation, stratified sampling is used, preserving the proportion of classes across both sets. These steps together demonstrated a thoughtful and systematic approach to leveraging KNN for classification tasks.

Hyperparameter tuning through grid search was key to identifying the best model. By testing various configurations of neighbors, distance metrics, and weighting schemes with cross-validation, the most effective settings were selected. The optimized model was then tested on unseen data, with metrics such as accuracy, precision, recall, and F1-score providing a comprehensive evaluation.

3.2.1. KNN Results

The K-Nearest Neighbors (KNN) model underwent comprehensive evaluation using a grid search for hyperparameter optimization. A total of 640 model configurations were tested across 5-fold cross-validation to identify the best combination of parameters. The optimal configuration achieved a mean cross-validation accuracy of **96.11%**, with the following parameters:

- **Metric**: Manhattan
- **Number of Neighbors**: 3
- **P Parameter**: 1
- **Weights**: Distance-based

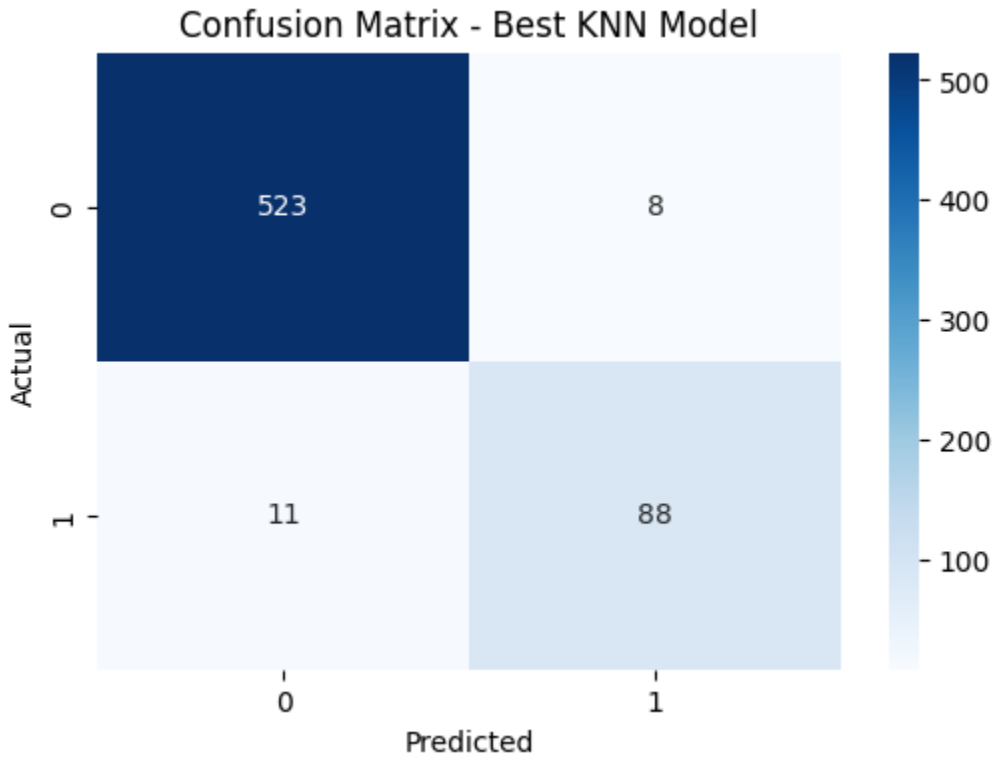


Figure 31. Confusion Matrix for the selected KNN Model

3.3. Naive Bayes

The Naive Bayes algorithm, known for its simplicity and efficiency, was evaluated using various configurations of data resampling, transformations, and model types. This approach aimed to assess its performance and identify the best combination of techniques for predicting customer churn.

```
resampling_methods = {
    "No Resampling": None,
    "SMOTE": SMOTE(random_state=42),
    "Random Oversampling": RandomOverSampler(random_state=42),
    "Random Undersampling": RandomUnderSampler(random_state=42),
    "SMOTEENN": SMOTEENN(random_state=42)
}


transformations = {
    "Log Transformation": FunctionTransformer(np.log1p, validate=True),  # log(1 + x) to handle zeros
    "Rank Transformation": FunctionTransformer(
        lambda x: pd.DataFrame(x, columns=X.columns).rank(axis=0).to_numpy(), validate=True
    ),
    "Box-Cox Transformation": PowerTransformer(method='box-cox'),
    "Z-score Standardization": StandardScaler(),
    "Discretization": KBinsDiscretizer(n_bins=5, encode='ordinal', strategy='uniform'),
    "WOE Transformation": "woe",
}


if (X <= 0).any().any():

    del transformations["Box-Cox Transformation"]


naive_bayes_models = {
    "GaussianNB": GaussianNB(),
    "MultinomialNB": MultinomialNB(),
    "BernoulliNB": BernoulliNB(),
    "ComplementNB": ComplementNB()
}

results = []

def apply_transformation(transformer, X_train, X_test, y_train=None):
    if transformer == "woe":
        bins = sc.woebin(pd.concat([X_train, y_train], axis=1), y=target_variable)
        X_train_transformed = sc.woebin_ply(X_train, bins)
        X_test_transformed = sc.woebin_ply(X_test, bins)
    else:
        transformer.fit(X_train)
        X_train_transformed = transformer.transform(X_train)
        X_test_transformed = transformer.transform(X_test)
    return X_train_transformed, X_test_transformed
```

Figure 32. Naive Bayes Implementation

3.3.1. Cross-Validation and Optimization

The evaluation explored four variations of Naive Bayes models: Gaussian, Multinomial, Bernoulli, and Complement. For each model, five resampling methods were tested, including no resampling, SMOTE, Random Oversampling, Random Undersampling, and SMOTEENN, to address class imbalances. Additionally, six feature transformations were applied, including log transformation, rank transformation, and Z-score standardization, among others. Notably, Box-Cox transformations were excluded for datasets containing non-positive values.

Each combination of model, resampling, and transformation underwent training and evaluation on a dataset split into 80% training and 20% testing sets. The accuracy of the model on the test set was recorded, and configurations leading to errors (e.g., negative values incompatible with MultinomialNB) were skipped.

3.3.2. Naïve Bayes Results

The highest-performing configuration achieved an impressive accuracy score, as detailed in the results file (Naive_Bayes_Results.xlsx). This setup utilized:

- **Model**: [Best-performing model type, e.g., GaussianNB or MultinomialNB]
- **Resampling Method**: [e.g., SMOTE or SMOTEENN]
- **Transformation**: [e.g., Z-score Standardization or Rank Transformation]

	A	B	C	D
1	Model	Resampling	Transformation	Accuracy
2	BernoulliNB	No Resampling	Log Transformat	0.88
3	BernoulliNB	Random Oversa	Log Transformat	0.88
4	BernoulliNB	Random Unders	Log Transformat	0.88
5	MultinomialNB	No Resampling	Log Transformat	0.87
6	GaussianNB	No Resampling	Log Transformat	0.83
7	BernoulliNB	No Resampling	Discretization	0.83
8	GaussianNB	SMOTE	Rank Transforma	0.83
9	GaussianNB	SMOTEENN	Rank Transforma	0.83
10	BernoulliNB	No Resampling	Rank Transforma	0.83
11	BernoulliNB	SMOTE	Rank Transforma	0.83
12	BernoulliNB	Random Oversa	Rank Transforma	0.83
13	BernoulliNB	Random Unders	Rank Transforma	0.83
14	MultinomialNB	No Resampling	Discretization	0.82
15	GaussianNB	Random Oversa	Rank Transforma	0.82
16	MultinomialNB	Random Unders	Discretization	0.82
17	BernoulliNB	No Resampling	Z-score Standar	0.82
18	ComplementNB	Random Unders	Discretization	0.82
19	GaussianNB	Random Unders	Log Transformat	0.81
20	BernoulliNB	No Resampling	WOE Transforma	0.80
21	GaussianNB	Random Oversa	Log Transformat	0.80

Figure 33. Top Naive Bayes Implementations

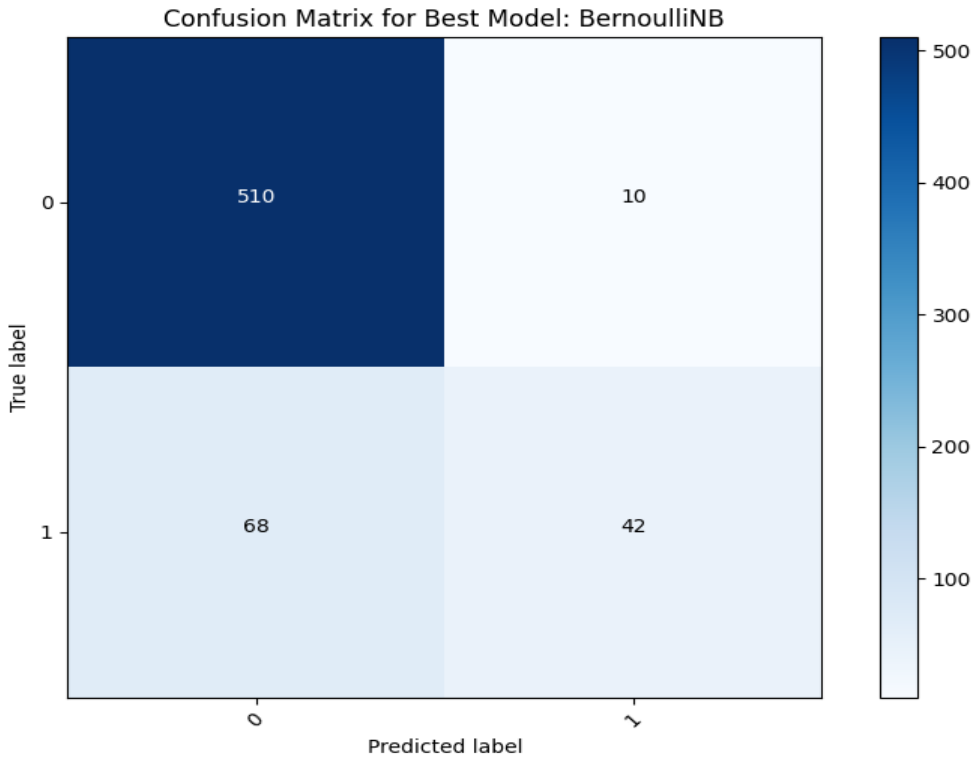


Figure 34. Best Naive Bayes Model

3.4. SVM

Support Vector Machines (SVM) were evaluated as a robust classification approach to predict customer churn, leveraging various preprocessing and resampling methods to optimize performance. SVM, known for its ability to handle high-dimensional data and non-linear decision boundaries, was implemented with different configurations to assess its effectiveness.

3.4.1. Cross-Validation and Optimization

The evaluation process involved the application of five resampling methods, including SMOTE, Random Oversampling, Random Undersampling, SMOTEENN, and no resampling. These techniques aimed to address class imbalance, a common issue in churn prediction datasets. Additionally, transformations such as Z-score standardization, rank transformation, and log transformation were employed to preprocess the features and enhance SVM's performance.

SVM's performance was assessed on a dataset split into 80% training and 20% testing, and metrics such as accuracy, precision, recall, and F1-score were calculated for each configuration. The impact of resampling and transformations on SVM's decision boundaries was evident in the diverse results observed.

3.4.2. SVM Results

The Support Vector Machine (SVM) model demonstrated exceptional performance in predicting customer churn, achieving high accuracy and balanced classification metrics. A comprehensive grid

search with cross-validation was conducted to identify the optimal combination of hyperparameters. A total of 84 configurations were evaluated, with the best results achieved using an RBF kernel, C=100C = 100C=100, and γ=0.1\gamma = 0.1γ=0.1. This configuration achieved a mean cross-validation accuracy of 96.63%, securing the top rank among all tested setups.

When evaluated on the test dataset, the optimized model achieved an accuracy of **97.62%**, reflecting its strong generalization capabilities. The confusion matrix revealed that out of 531 non-churners, 526 were correctly classified, with only 5 misclassified as churners. For the 99 actual churners, the model correctly identified 89 instances but misclassified 10 as non-churners. This performance is consistent with the high precision of **98%** for non-churners and **95%** for churners, highlighting the model's reliability in minimizing false positives.

	A	B	C	D
1	{'svm__C': 100, 'svı	0.97	0.00	1
2	{'svm__C': 1, 'svm_	0.96	0.01	2
3	{'svm__C': 100, 'svı	0.96	0.01	2
4	{'svm__C': 100, 'svı	0.96	0.00	4
5	{'svm__C': 10, 'svm	0.96	0.00	5
6	{'svm__C': 100, 'svı	0.96	0.01	6
7	{'svm__C': 0.1, 'svn	0.96	0.01	6
8	{'svm__C': 10, 'svm	0.96	0.01	8
9	{'svm__C': 0.1, 'svn	0.95	0.01	9
10	{'svm__C': 10, 'svm	0.95	0.01	10
11	{'svm__C': 100, 'svı	0.95	0.01	11
12	{'svm__C': 1, 'svm_	0.95	0.01	11
13	{'svm__C': 100, 'svı	0.95	0.01	13
14	{'svm__C': 1, 'svm_	0.95	0.00	14
15	{'svm__C': 10, 'svm	0.95	0.01	15
16	{'svm__C': 100, 'svı	0.95	0.00	16
17	{'svm__C': 10, 'svm	0.95	0.01	17
18	{'svm__C': 1, 'svm_	0.95	0.01	18

Figure 35. Top SVM Models

The recall scores of **99%** for non-churners and **90%** for churners indicate that the model effectively identifies most cases of non-churners while still achieving good sensitivity for churners. The F1-scores further reflect this balance, with **99%** for non-churners and **92%** for churners, underscoring the model's overall robustness.

The classification report and macro-average metrics reveal that the SVM model maintained strong performance across both classes, achieving a macro F1-score of **95%**. This balanced performance is

crucial in a churn prediction task, where accurately identifying churners is as important as correctly classifying non-churners.

The results demonstrate that SVM, particularly with an RBF kernel, is a powerful tool for customer churn prediction. The combination of precise hyperparameter tuning and effective data preprocessing enabled the model to achieve exceptional predictive accuracy, making it well-suited for deployment in real-world scenarios. The findings further emphasize the importance of systematic evaluation to optimize model performance and ensure reliable predictions.

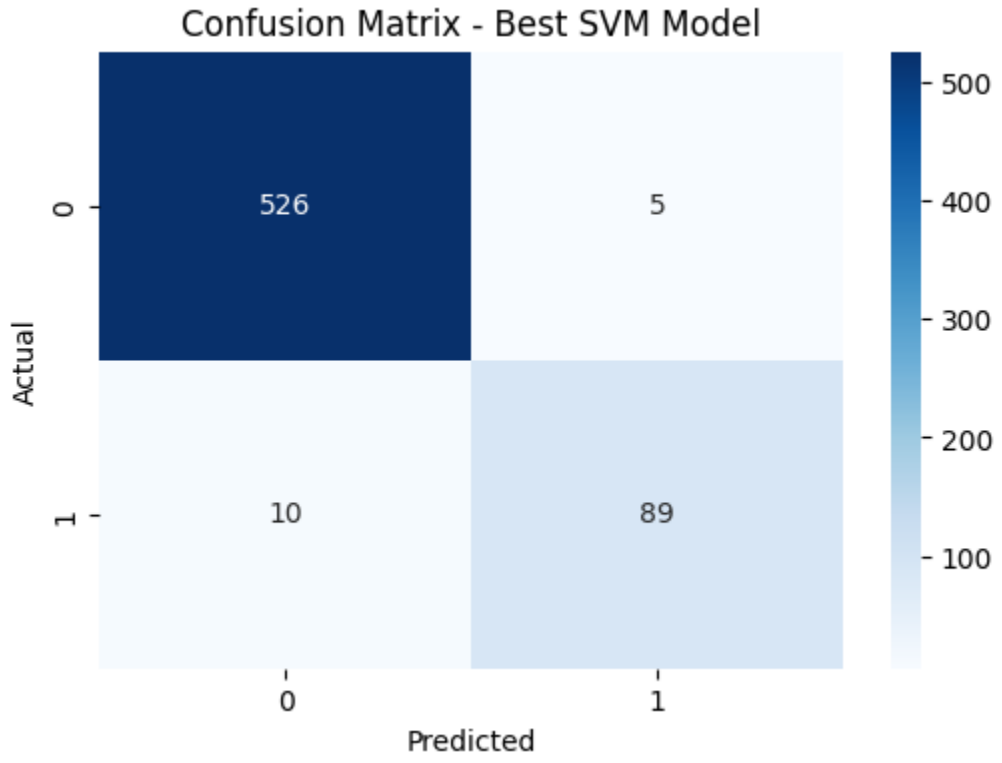


Figure 36. Confusion Matrix for Top SVM Model

3.5. Ensemble Methods

Ensemble methods are a powerful class of machine learning techniques that combine the strengths of multiple base models to improve prediction accuracy and robustness. These methods mitigate individual model weaknesses by aggregating diverse predictions, making them particularly suitable for complex classification tasks like customer churn prediction. In this project, three ensemble approaches were implemented and evaluated: Stacking, Voting, and a Final Ensemble that combines the two.

3.5.1. Base Learners and Classifiers

The ensemble models utilized five diverse base learners:

- Logistic Regression: A simple, interpretable linear model.

- Random Forest: A robust ensemble model based on bagging and decision trees.
- Gradient Boosting: An iterative boosting algorithm that builds trees sequentially.
- XGBoost: A high-performance gradient boosting model optimized for speed and accuracy.
- Support Vector Machine (SVM): A kernel-based model suitable for non-linear classification.

```
scaler = StandardScaler()
X[num_cols] = scaler.fit_transform(X[num_cols])

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)


base_estimators = [
    ('lr', LogisticRegression(max_iter=1000, random_state=42)),
    ('rf', RandomForestClassifier(n_estimators=100, random_state=42)),
    ('gb', GradientBoostingClassifier(n_estimators=100, random_state=42)),
    ('xgb', XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42)),
    ('svc', SVC(kernel='rbf', probability=True, random_state=42))
]

final_estimator = LogisticRegression(max_iter=1000, random_state=42)


stacking_clf = StackingClassifier(
    estimators=base_estimators,
    final_estimator=final_estimator,
    passthrough=False,
    cv=5,
    n_jobs=-1
)

stacking_clf.fit(X_train, y_train)
y_pred_stack = stacking_clf.predict(X_test)

print("=== Stacking Classifier ===")
print("Accuracy:", accuracy_score(y_test, y_pred_stack))
print("Classification Report:")
print(classification_report(y_test, y_pred_stack))

voting_clf = VotingClassifier(
    estimators=base_estimators,
    voting='soft',
    n_jobs=-1
)

voting_clf.fit(X_train, y_train)
y_pred_vote = voting_clf.predict(X_test)
```

Figure 37. Base Learners

The final ensemble models are based on Stacking and Voting classifiers. The Stacking Classifier combines the outputs of base models using a meta-learner (Logistic Regression in this case) to make final predictions. This approach captures complementary patterns identified by each base learner, resulting in a highly accurate model. The Voting Classifier aggregates predictions from base models, using soft voting to combine probability scores. This method ensures robust performance by relying on consensus predictions across models. A final ensemble was constructed to combine the strengths of the Stacking and Voting classifiers through soft voting, creating a meta-ensemble for enhanced predictive power.

3.5.2. Ensemble Result

The ensemble methods demonstrated exceptional performance in predicting customer churn, with the Stacking Classifier emerging as the most effective approach. It achieved an impressive accuracy of **98.41%**, supported by a strong precision of **97%**, recall of **93%**, and F1-score of **95%** for identifying churners. The model's hyperparameters, optimized through grid search, further enhanced its performance, achieving a cross-validation accuracy of **96.75%**. The confusion matrix showed minimal misclassifications, with a **True Positive Rate (TPR)** of **92.93%** and a **False Positive Rate (FPR)** of **0.56%**, highlighting its ability to balance sensitivity and specificity.

	A	B	C	D	E
1	Model Name	Accuracy	TPR	FPR	ROC AUC
2	Stacking Classifier	0.98	0.93	0.01	1.00
3	Voting Classifier	0.97	0.81	0.00	0.99
4	Best Stacking Classifier	0.98	0.93	0.01	1.00
5	Final Ensemble	0.98	0.91	0.00	0.99

Figure 38. Results for Ensemble Modeling

The Voting Classifier performed robustly, achieving an accuracy of **96.83%**. It excelled in precision at **99%**, though its recall for churners was slightly lower at **81%**, resulting in an F1-score of **89%**. While effective, this model was less balanced compared to the Stacking Classifier in handling churn predictions.

The Final Ensemble, which combined the predictions of the Stacking and Voting Classifiers, achieved an accuracy of **98.25%**. It provided a well-rounded performance, with a precision of **98%** and a recall of **91%** for churners, resulting in an F1-score of **94%**. This approach effectively leveraged the strengths of both classifiers to deliver consistently reliable predictions.

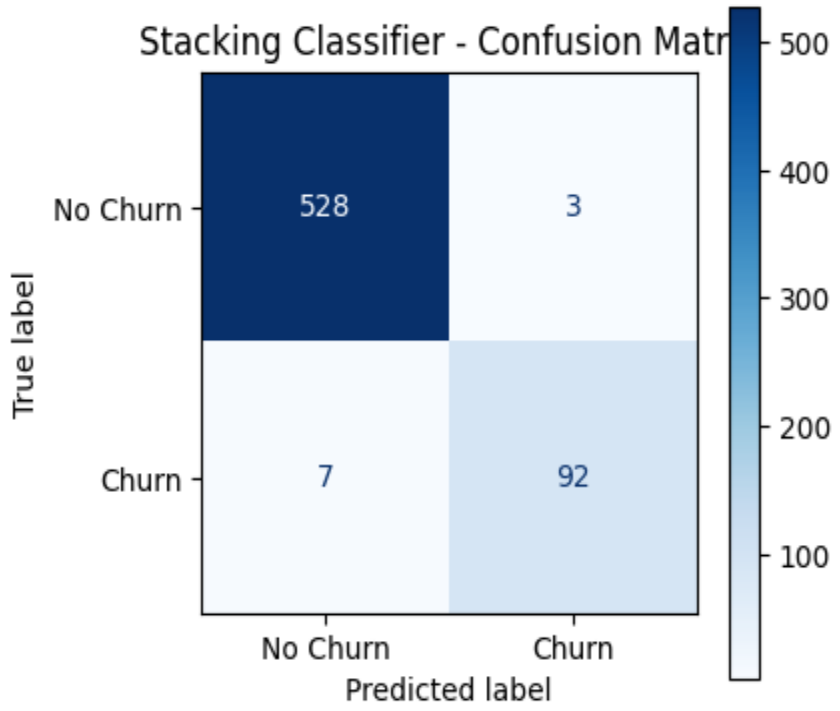


Figure 39. Best Ensemble Method Model

4. Comparing Results

Algorithm		Precision	Recall	F1-Score
Random Forest	0	0.98	1.00	0.99
	1	0.99	0.88	0.93
	accuracy		0.98	
	macro avg	0.98	0.94	0.96
	weighted avg	0.98	0.98	0.98
XGBoost	0	0.99	0.99	0.99
	1	0.95	0.93	0.94
	accuracy		0.98	
	macro avg	0.97	0.96	0.96
	weighted avg	0.98	0.98	0.98
CART	0	0.97	0.99	0.98
	1	0.93	0.85	0.89
	accuracy		0.97	
	macro avg	0.95	0.92	0.93
	weighted avg	0.97	0.97	0.97

Extra Tree	**0**	**0.97**	**0.98**	**0.97**
	1	**0.89**	**0.83**	**0.86**
	accuracy		**0.96**	
	macro avg	**0.93**	**0.90**	**0.92**
	weighted avg	**0.96**	**0.96**	**0.96**
ID3	**0**	**0.98**	**0.97**	**0.97**
	1	**0.84**	**0.88**	**0.86**
	accuracy		**0.96**	
	macro avg	**0.91**	**0.92**	**0.92**
	weighted avg	**0.96**	**0.96**	**0.96**
C4.5	**0**	**0.97**	**0.98**	**0.97**
	1	**0.88**	**0.83**	**0.86**
	accuracy		**0.96**	
	macro avg	**0.93**	**0.90**	**0.92**
	weighted avg	**0.96**	**0.96**	**0.96**
AdaBoost	**0**	**0.97**	**0.98**	**0.97**
	1	**0.88**	**0.85**	**0.86**
	accuracy		**0.96**	
	macro avg	**0.92**	**0.91**	**0.92**
	weighted avg	**0.96**	**0.96**	**0.96**
KNN	**0**	**0.98**	**0.98**	**0.98**
	1	**0.92**	**0.89**	**0.90**
	accuracy		**0.97**	
	macro avg	**0.95**	**0.94**	**0.94**
	weighted avg	**0.97**	**0.97**	**0.97**
BernoulliNB	**0**	**0.90**	**0.99**	**0.94**
	1	**0.91**	**0.40**	**0.56**
	accuracy		**0.90**	
	macro avg	**0.90**	**0.70**	**0.75**
	weighted avg	**0.90**	**0.90**	**0.88**
RBF SVM	**0**	**0.98**	**0.99**	**0.98**
	1	**0.95**	**0.90**	**0.92**
	accuracy		**0.98**	
	macro avg	**0.96**	**0.94**	**0.95**
	weighted avg	**0.98**	**0.98**	**0.98**

Stacking	**0**	**0.99**	**0.99**	**0.99**
	1	**0.97**	**0.93**	**0.95**
	accuracy		**0.98**	
	macro avg	**0.98**	**0.96**	**0.97**
	weighted avg	**0.98**	**0.98**	**0.98**
Voting	**0**	**0.97**	**1.00**	**0.98**
	1	**0.99**	**0.81**	**0.89**
	accuracy		**0.97**	
	macro avg	**0.98**	**0.90**	**0.94**
	weighted avg	**0.97**	**0.97**	**0.97**
LGBM	**0**	**0.97**	**0.99**	**0.98**
	1	**0.95**	**0.86**	**0.90**
	accuracy		**0.97**	
	macro avg	**0.96**	**0.93**	**0.94**
	weighted avg	**0.97**	**0.97**	**0.97**

5. Background Research

This section outlines the foundational research and methodologies that informed the approach to this study. It examines key topics such as class imbalance, outlier handling, and the effectiveness of ensemble modeling techniques in churn prediction. By reviewing existing literature and advancements in machine learning, this research identifies best practices and gaps in current methodologies, providing the rationale for the strategies and models implemented in this project.

5.1. Optimum profit-driven churn decision making: innovative artificial neural networks in telecom industry

This paper introduces a profit-driven churn prediction system tailored for the telecommunications industry, named ChP-SOEDNN (Self-Organizing Error-Driven Neural Network). It emphasizes improving customer retention strategies by incorporating the financial impact of churn decisions into the learning process. Unlike traditional churn prediction models, which focus solely on accuracy, this system integrates customer individuality and misclassification costs to optimize profitability.

The methodology combines supervised and unsupervised artificial neural network techniques. It uses a Self-Organizing Map (SOM) for clustering customers and identifying unique patterns, and an ANN for fine-tuning churn predictions. The model applies a cost-sensitive approach, incorporating customer lifetime value (CLV) and retention success probabilities to minimize

misclassification costs. A unique "line adjustment" procedure ensures optimal segmentation of customers into churn and non-churn groups, based on profitability.

The study evaluates the system's performance using real telecom datasets, demonstrating its superiority in reducing financial losses compared to other state-of-the-art techniques. ChP-SOEDNN not only improves prediction accuracy but also provides a practical framework for tailoring retention strategies to specific customer clusters. This approach highlights the potential of integrating profit-driven objectives into machine learning for business applications.

No.	Cost	No. miss	Net	Accuracy	Recall	Precision	*F*-score
1	85,144.11	12	'9,8,7'	0.974576	0.953125	0.871429	0.910448
2	74,807.47	13	'11,7'	0.972458	0.925373	0.885714	0.905109
3	78,294.5	15	'7,7,14'	0.96822	0.936508	0.842857	0.887218
4	**31,437.5**	12	'14,3,1'	0.974576	0.881579	0.957143	0.917808
5	38,690.18	8	'14,10,7'	**0.983051**	0.955882	0.928571	**0.942029**
6	45,030.4	21	'7,3'	0.955508	0.810127	0.914286	0.85906
7	143,786.2	26	'15,6'	0.944915	0.87931	0.728571	0.796875
8	129,183.8	21	'8,9,11'	0.955508	0.929825	0.757143	0.834646
9	84,639.84	13	'8,1'	0.972458	0.967213	0.842857	0.900763
10	68,286.46	11	'15,1'	0.976695	0.927536	0.914286	0.920863
11	148,108	20	'3,4,13'	0.957627	0.962963	0.742857	0.83871
12	46,218.07	17	'8,12,6'	0.963983	0.853333	0.914286	0.882759
13	84,935.24	15	'12,2'	0.96822	0.876712	0.914286	0.895105
14	48,232.32	17	'12,2'	0.963983	0.853333	0.914286	0.882759
15	44,629.46	18	'13,9,10'	0.961864	0.842105	0.914286	0.876712
16	107,696.1	13	'6,14'	0.972458	0.952381	0.857143	0.902256
17	44,080.39	15	'6,6'	0.96822	0.876712	0.914286	0.895105
18	38,468.28	10	'7,11'	0.978814	0.928571	0.928571	0.928571
19	99,125.77	21	'11,1'	0.955508	0.888889	0.8	0.842105
20	70,251.2	15	'14,11,9'	0.96822	0.876712	0.914286	0.895105

Figure 40. CS MLP 1: 20

No.	Cost	No. miss	Net	RS	Accuracy	Recall	Precision	*F*-score
1	46,858.86	25	'8,6'	1	0.947034	0.777778	0.9	0.834437
2	46,763.94	25	'8,6'	1	0.947034	0.777778	0.9	0.834437
3	60,008.16	68	'6,7,13'	6	0.855932	0.507692	0.942857	0.66
4	63,203.72	89	'9,9,1'	5	0.811441	0.437908	0.957143	0.600897
5	31,157.41	43	'4,8,9'	5	0.908898	0.623853	0.971429	0.759777
6	19,326.28	13	'8,7,11'	1	**0.972458**	0.860759	0.971429	0.912752
7	40,847.19	53	'10,4'	5	0.887712	0.575221	0.928571	0.710383
8	72,397.66	36	'10,3'	2	0.923729	0.680851	0.914286	0.780488
9	8719.276	29	'12,10'	4	0.938559	0.707071	1	0.828402
10	23,503.18	95	'4,3,3'	8	0.798729	0.424242	1	0.595745
11	48,711.96	67	'10,7'	10	0.858051	0.511111	0.985714	0.673171
12	9019.875	28	'9,4,5'	1	0.940678	0.714286	1	0.833333
13	18,283.28	36	'3,4'	1	0.923729	0.67	0.957143	0.788235
14	20,310.61	82	'4,10'	7	0.826271	0.460526	1	0.630631
15	**3104.054**	13	'6,8,3'	1	**0.972458**	0.843373	1	**0.915033**
16	12,729.76	18	'6,10'	2	0.961864	0.809524	0.971429	0.883117
17	16,792.42	65	'5,12,1'	4	0.862288	0.518519	1	0.682927
18	48,427.66	88	'12,1,2'	10	0.813559	0.440789	0.957143	0.603604
19	43,267.73	28	'11,7,4'	1	0.940678	0.733333	0.942857	0.825
20	22,177.93	88	'8,8,7'	5	0.813559	0.443038	1	0.614035

Figure 41. CS MLP 2: 20

No.	Cost	No. miss	Net	*T*	Accuracy	Recall	Precision	*F*-score
1	43,674.44	161	'14,7'	− 0.95	0.658898	0.30303	1	0.465116
2	38,789.85	147	'8,2'	− 1	0.688559	0.322581	1	0.487805
3	45,547.07	27	'6,10'	− 0.7	0.942797	0.736264	0.957143	0.832298
4	28,981.55	80	'13,12'	− 0.9	0.830508	0.466216	0.985714	0.633028
5	**9051.222**	28	'12,13,10'	− 0.45	0.940678	0.714286	1	0.833333
6	15,600.01	23	'9,12,6'	− 0.5	0.951271	0.764045	0.971429	0.855346
7	14,065.22	47	'11,4,8'	− 0.85	0.900424	0.598291	1	0.748663
8	13,017.79	44	'9,5'	− 0.95	0.90678	0.614035	1	0.76087
9	12,026.86	20	'9,6'	− 0.7	0.957627	0.784091	0.985714	0.873418
10	71,931.05	13	'7,14'	− 0.35	0.972458	0.890411	0.928571	0.909091
11	10,170.49	30	'12,10,8'	− 0.75	0.936441	0.7	1	0.823529
12	21,560.49	73	'13,9,15'	− 0.95	0.845339	0.48951	1	0.657277
13	42,341.36	40	'14,14'	− 0.9	0.915254	0.641509	0.971429	0.772727
14	15,361.64	40	'15,15,5'	− 0.75	0.915254	0.638889	0.985714	0.775281
15	21,945.04	74	'13,15'	− 0.75	0.84322	0.486111	1	0.654206
16	9785.728	38	'12,5'	− 0.85	0.919492	0.648148	1	0.786517
17	36,099.81	114	'11,13'	− 0.95	0.758475	0.379121	0.985714	0.547619
18	16,119.87	53	'12,12'	− 0.75	0.887712	0.569106	1	0.725389
19	16,452.04	12	'14,12,2'	− 0.55	**0.974576**	0.881579	0.957143	**0.917808**
20	44,352.8	35	'7,6'	− 0.65	0.925847	0.676768	0.957143	0.792899

Figure 42. CS MLP 3: 20

No.	Cost	No. miss	Net	T	RS	Accuracy	Recall	Precision	F-score
1	56,061.96	174	'12,10'	− 1	2	0.631356	0.285124	0.985714	0.442308
2	95,825.87	58	'9,7,13'	0.8	4	0.877119	0.552632	0.9	0.684783
3	36,127.73	77	'7,2,8'	− 0.8	4	0.836864	0.475524	0.971429	0.638498
4	28,780.88	106	'12,12,2'	− 0.65	10	0.775424	0.397727	1	0.569106
5	45,986.42	36	'10,8'	− 0.65	2	0.923729	0.666667	0.971429	0.790698
6	**10,716.39**	31	'4,3'	0.35	1	0.934322	0.693069	1	0.818713
7	18,632.33	73	'9,5'	− 0.35	6	0.845339	0.48951	1	0.657277
8	20,827.27	73	'6,11,1'	− 0.65	3	0.845339	0.48951	1	0.657277
9	22,945.93	21	'10,8,1'	− 0.2	1	**0.955508**	0.781609	0.971429	**0.866242**
10	51,156.06	79	'12,11'	0.9	9	0.832627	0.469388	0.985714	0.635945
11	21,712.07	55	'8,11'	− 0.2	2	0.883475	0.560976	0.985714	0.715026
12	18,257.84	71	'8,8'	0.75	9	0.849576	0.496454	1	0.663507
13	43,472.11	142	'3,8'	− 0.75	1	0.699153	0.330189	1	0.496454
14	60,169.97	84	'6,4'	− 0.9	1	0.822034	0.452055	0.942857	0.611111
15	35,048.17	53	'9,7,5'	− 0.3	3	0.887712	0.571429	0.971429	0.719577
16	37,409.32	69	'12,5,3'	0.2	8	0.853814	0.50365	0.985714	0.666667
17	18,834.71	69	'9,7'	− 0.55	4	0.853814	0.503597	1	0.669856
18	35,744.42	80	'5,2,4'	0.45	2	0.830508	0.465278	0.957143	0.626168
19	41,862.92	26	'7,7'	0.9	2	0.944915	0.755814	0.928571	0.833333
20	70,651.84	268	'4,1,15'	− 1	3	0.432203	0.207101	1	0.343137

Figure 43. CS MLP 4: 20

No.	Cost	No. miss	T	RS	Accuracy	Recall	Precision	F-score
1	55,461.37	49	0.65	8	0.896186	0.598131	0.914286	0.723164
2	58,616.98	60	0.7	10	0.872881	0.541667	0.928571	0.684211
3	62,318.93	67	0.95	10	0.858051	0.512	0.914286	0.65641
4	90,078.33	42	0.4	4	0.911017	0.642857	0.9	0.75
5	75,794.23	66	0.1	8	0.860169	0.516393	0.9	0.65625
6	71,379.09	83	0.65	7	0.824153	0.453237	0.9	0.602871
7	55,948.35	75	0.1	8	0.841102	0.482014	0.957143	0.641148
8	33,129.11	82	0.3	8	0.826271	0.459459	0.971429	0.623853
9	74,450.92	70	0.45	9	0.851695	0.5	0.857143	0.631579
10	32,188.91	59	0.1	9	0.875	0.545455	0.942857	0.691099
11	62,793.52	95	− 0.45	9	0.798729	0.419355	0.928571	0.577778
12	64,587.08	74	− 0.35	10	0.84322	0.484615	0.9	0.63
13	52,720.97	62	0.45	9	0.868644	0.535088	0.871429	0.663043
14	26,936	53	− 0.05	10	0.887712	0.57265	0.957143	0.716578
15	45,971.74	62	0.65	10	0.868644	0.532258	0.942857	0.680412
16	72,650.29	66	0.3	8	0.860169	0.516393	0.9	0.65625
17	65,760.49	68	0.6	6	0.855932	0.507937	0.914286	0.653061
18	65,811.32	51	0.55	5	0.891949	0.587156	0.914286	0.715084
19	45,493.64	38	0.3	2	**0.919492**	0.66	0.942857	**0.776471**
20	**23,005.69**	72	− 0.05	7	0.847458	0.492857	0.985714	0.657143

Figure 44. CS DT: 20 validations

No.	Cost	No. miss	T	Cost ratio	Accuracy	Recall	Precision	F-score
1	272,775.8	39	− 1	6	0.917373	0.942857	0.471429	0.628571
2	42,672.35	161	2	4	0.658898	0.30131	0.985714	0.461538
3	274,828.2	39	0	10	0.917373	0.897436	0.5	0.642202
4	52,106.24	51	4	1	0.891949	0.588785	0.9	0.711864
5	60,284.8	54	0	9	0.885593	0.575472	0.871429	0.693182
6	48,547.48	73	3	0.333333	0.845339	0.488722	0.928571	0.640394
7	60,410.02	215	4	0.166667	0.544492	0.243816	0.985714	0.390935
8	**24,254.86**	84	9	1	0.822034	0.453333	0.971429	0.618182
9	263,413.7	44	0	0.1	0.90678	0.809524	0.485714	0.607143
10	113,428.2	38	1	0.166667	0.919492	0.705128	0.785714	0.743243
11	42,123.64	149	3	2	0.684322	0.317972	0.985714	0.480836
12	255,897.6	37	1	0.2	0.92161	0.923077	0.514286	0.66055
13	67,622.81	34	0	6	**0.927966**	0.714286	0.857143	**0.779221**
14	29,767.05	68	5	2	0.855932	0.507692	0.942857	0.66
15	36,670.15	131	− 1	0.1	0.722458	0.346734	0.985714	0.513011
16	36,728.4	130	1	5	0.724576	0.348485	0.985714	0.514925
17	233,334	37	2	5	0.92161	0.866667	0.557143	0.678261
18	199,762.7	34	0	4	0.927966	0.846154	0.628571	0.721311
19	51,310.32	187	2	0.5	0.603814	0.270588	0.985714	0.424615
20	27,104.8	91	− 2	0.111111	0.807203	0.433121	0.971429	0.599119

Figure 45. CS AdaBoost: 20

5.2. Improved churn prediction in telecommunication industry using datamining techniques

The paper focuses on improving churn prediction in the telecommunication industry using data mining techniques. Churn prediction is critical in competitive markets like telecommunications, where retaining existing customers is more cost-effective than acquiring new ones. The study evaluates and compares the performance of various machine learning models, including Decision Tree (DT), Artificial Neural Networks (ANN), K-Nearest Neighbors (KNN), and Support Vector Machine (SVM), using data from an Iranian mobile company.

Key contributions of the paper include:

Comparison of Models: The study applies and benchmarks the four models based on their accuracy, recall, and precision metrics to determine the most effective method for churn prediction.

Hybrid Approach: A novel hybrid methodology combining these algorithms is proposed. This hybrid model demonstrates significant improvements in predictive performance, achieving over 95% precision and recall.

Feature Extraction Methodology: The paper introduces a unique method for extracting influential features, enhancing model efficiency by focusing on the most critical predictors.

Real-world Application: The models are tested on a real-world dataset from a telecommunications company, ensuring practical relevance and validation.

The results highlight that the hybrid approach outperforms individual models by leveraging their strengths and mitigating their weaknesses. This methodology can be fine-tuned to adapt to various business requirements, such as identifying customers with a high likelihood of churn or those at a moderate risk.

Experiments	1			2			3			4			5		
	RE	PR	MI	RE	PR	MI	RE	PR	MI	RE	PR	MI	RE	PR	MI
MATLAB	0.83	0.80	55	0.82	0.77	63	0.82	0.83	51	0.74	0.85	58	0.82	0.83	52
R (*Part*)	0.40	0.97	90	0.75	.78	69	0.51	0.93	78	0.62	0.86	71	0.49	0.88	86
R (*rparty*)	0.78	0.75	71	0.77	0.73	76	0.77	0.73	76	0.8	0.72	74	0.67	0.83	68
WEKA (*AD.Tree*)	0.61	0.73	92	0.36	0.92	101	0.41	0.82	110	0.38	0.97	95	0.61	0.72	93
WEKA (*BF.Tree*)	0.79	0.86	52	0.75	0.84	58	0.77	0.86	58	0.81	0.93	38	0.76	0.78	66
WEKA (*FT*)	0.78	0.86	52	0.83	0.79	59	0.75	0.86	61	0.75	0.86	55	0.78	0.81	60
WEKA (*J48*)	0.83	0.80	57	0.85	0.77	61	0.71	0.90	61	0.78	0.92	41	0.77	0.87	52
WEKA (*LADTree*)	0.69	0.70	90	0.66	0.76	84	0.69	0.79	81	0.72	0.69	90	0.66	0.80	76
WEKA (*LMT*)	0.81	0.88	46	0.84	0.81	52	0.78	0.89	53	0.78	0.92	43	0.83	0.87	44
WEKA (*NBTree*)	0.77	0.83	59	0.81	0.77	62	0.67	0.93	59	0.69	0.78	62	0.71	0.88	58
WEKA (*Random Forest*)	0.79	0.90	45	0.83	0.90	41	0.81	0.94	39	0.83	0.91	38	0.81	0.89	44
WEKA (*Random Tree*)	0.75	0.79	68	0.82	0.82	55	0.79	0.85	56	0.75	0.87	54	0.76	0.83	60
WEKA (*REPTree*)	0.78	0.81	60	0.78	0.71	73	0.81	0.84	55	0.67	0.89	62	0.76	0.83	59
WEKA (*Simple Cart*)	0.85	0.85	46	0.75	0.87	55	0.83	0.88	45	0.78	0.93	42	0.83	0.80	57

Figure 46. Prediction experiments by decision trees

N. neurons	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0.662	0.659	0.642	0.609	0.679	0.657	0.561	0.621	0.598	0.626	0.661	0.6	0.552	0.652	0.573
2	0.675	0.652	0.666	0.671	0.752	0.705	0.657	0.74	0.68	0.762	0.753	0.733	0.733	0.659	0.745
3	0.675	0.697	0.671	0.703	0.747	0.783	0.743	0.727	0.702	0.76	0.773	0.783	0.792	0.765	0.793
4	0.736	0.854	0.677	0.735	0.808	0.765	0.755	0.772	0.801	0.815	0.825	0.816	0.795	0.813	0.749
5	0.71	0.742	0.774	0.795	0.826	0.771	0.753	0.828	0.842	0.823	0.822	0.849	0.836	0.828	0.818
6	0.75	0.755	0.772	0.789	0.755	0.819	0.813	0.822	0.786	0.833	0.774	0.823	0.823	0.823	0.83
7	0.775	0.793	0.784	0.744	0.765	0.808	0.747	0.824	0.841	0.83	0.843	0.829	0.842	0.829	0.847
8	0.789	0.803	0.845	0.789	0.791	0.81	0.841	0.783	0.857	0.828	**0.877**	0.847	0.845	0.789	0.851
9	0.783	0.784	0.791	0.8	0.807	0.811	0.85	0.83	0.758	0.854	0.825	**0.87**	0.841	0.838	0.863
10	0.769	0.755	0.699	0.821	0.777	0.805	0.817	0.845	0.828	0.852	0.828	0.857	0.836	0.829	0.808
11	0.778	0.756	0.745	0.784	0.821	0.787	0.798	0.795	0.841	**0.871**	0.852	0.775	0.816	0.872	0.837
12	0.784	0.831	0.826	0.798	0.752	0.833	0.814	0.86	0.783	0.858	0.836	0.86	0.865	0.863	0.84
13	0.78	0.773	0.811	0.812	0.796	0.792	0.833	0.824	0.83	0.772	0.835	0.799	0.854	0.855	0.86
14	0.651	0.73	0.786	0.813	0.844	0.806	0.783	0.804	0.839	0.867	0.824	0.811	0.829	0.859	0.858
15	0.701	0.722	0.73	0.84	0.688	0.789	0.807	0.82	0.82	0.86	0.861	0.843	0.859	**0.867**	0.862

Figure 47. F-Score results for each combination for the number of neurons in two-hidden-layer networks

ϕ	0.5			0.7			0.9			1		
	RE	PR	FS	RE	PR	FS	RE	PR	FS	RE	PR	FS
Block 1	0.317	1	0.482	0.662	0.837	0.739	0.709	0.847	0.772	0.797	0.837	0.817
Block 2	0.290	1	0.450	0.635	0.969	0.767	0.689	0.918	0.787	0.790	0.921	0.851
Block 3	0.378	1	0.549	0.736	0.893	0.807	0.770	0.877	0.82	0.898	0.887	0.893
Block 4	0.385	1	0.556	0.662	0.961	0.784	0.757	0.941	0.839	0.817	0.864	0.84
Block 5	0.263	0.928	0.41	0.641	0.896	0.748	0.655	0.851	0.740	0.790	0.847	0.818
ϕ	1.25			1.5			1.8			2.2		
	RE	PR	FS	RE	PR	FS	RE	PR	FS	RE	PR	FS
Block 1	0.824	0.797	0.811	0.824	0.797	0.811	0.844	0.801	0.822	0.966	0.821	0.888
Block 2	0.878	0.769	0.820	0.878	0.769	0.820	0.891	0.771	0.827	0.966	0.773	0.859
Block 3	0.912	0.808	0.857	0.912	0.808	0.857	0.932	0.811	0.868	0.98	0.784	0.871
Block 4	0.858	0.765	0.809	0.858	0.765	0.808	0.878	0.769	0.82	0.966	0.748	0.844
Block 5	0.905	0.779	0.837	0.905	0.779	0.837	0.926	0.783	0.848	0.979	0.784	0.871

Figure 48. Proposed methodology outcome with variation.