PGP-DSBA PROJECT REPORT

CAPSTONE FINAL REPORT SUBMISSION

CUSTOMER CHURN PREDICTION

BY

ISHAAN SHAKTI JAYARAMAN

PGPDSBA.O.JULY24.A

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

1.1 Problem Statement

In a highly competitive market, an E-Commerce company is facing increasing difficulty in retaining its existing accounts, where each account may represent multiple individual customers, thus amplifying the business impact of churn. To address this, the company aims to develop a predictive model to identify accounts at high risk of churn and implement a targeted, cost-effective retention campaign. The model must accurately forecast potential churners using historical account data and customer behavior, enabling the company to proactively engage them with segmented offers. These offers must strike a balance between being attractive enough to retain customers while ensuring financial sustainability, as overly generous incentives risk rejection by the revenue assurance team.

1.2 Project Objective

The need for this project arises from the critical business challenge of customer retention in a highly competitive E-Commerce environment, where losing a single account can lead to the loss of multiple customers and substantial revenue. The objective is to develop a data-driven churn prediction model which is essential to identify at-risk customers, allowing the company to intervene before churn occurs. Additionally, integrating intelligent segmentation and targeted campaign offers ensures that retention efforts are focused on the right accounts, using incentives that are both persuasive to the customer and financially viable for the company.

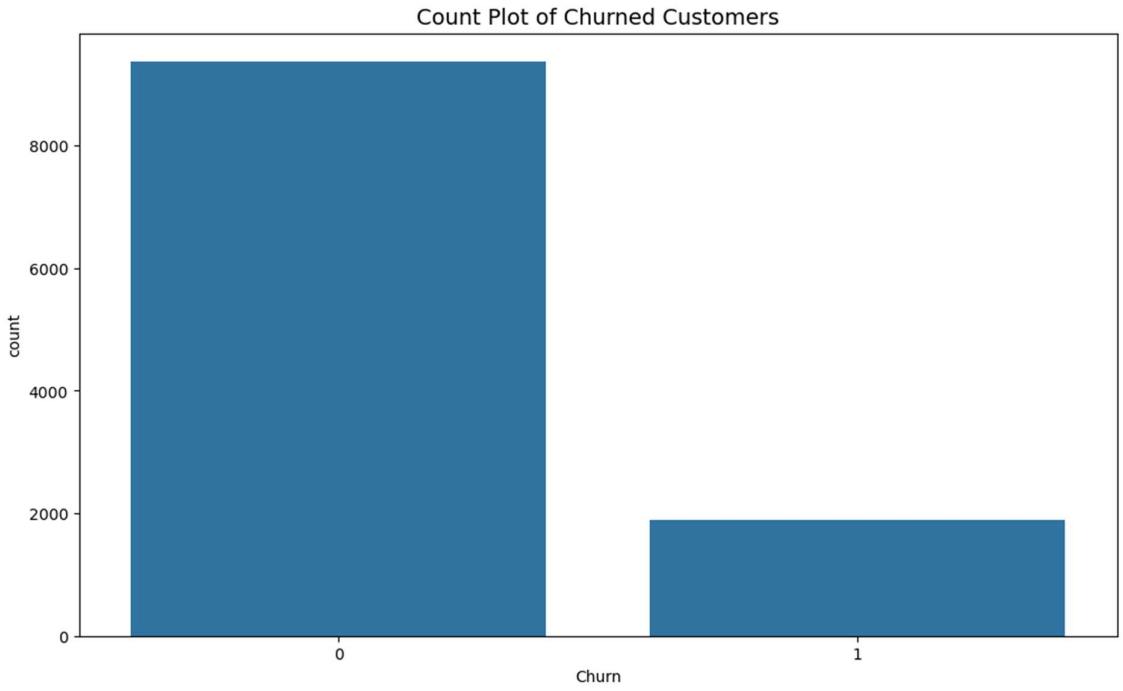
1.3 Data Overview

Data represents 11,260 unique account IDs, covering a wide range of customer demographics such as gender and marital status etc. Key variables like "CC_Contacted_L12m", "rev_per_month", "Complain_l12m", "rev_growth_yoy", "coupon_used_l12m", "Day_Since_CC_connect", and "cashback_l12m" implies a 12-month period for data collection. The dataset consists of 19 variables, with 18 independent and 1 dependent variable, which tracks customer churn. It provides insights into the services used by customers, their payment options, and basic personal information. The dataset includes both categorical and numerical variables.

EDA AND BUSINESS IMPLICATION

2.1 Univariate Analysis

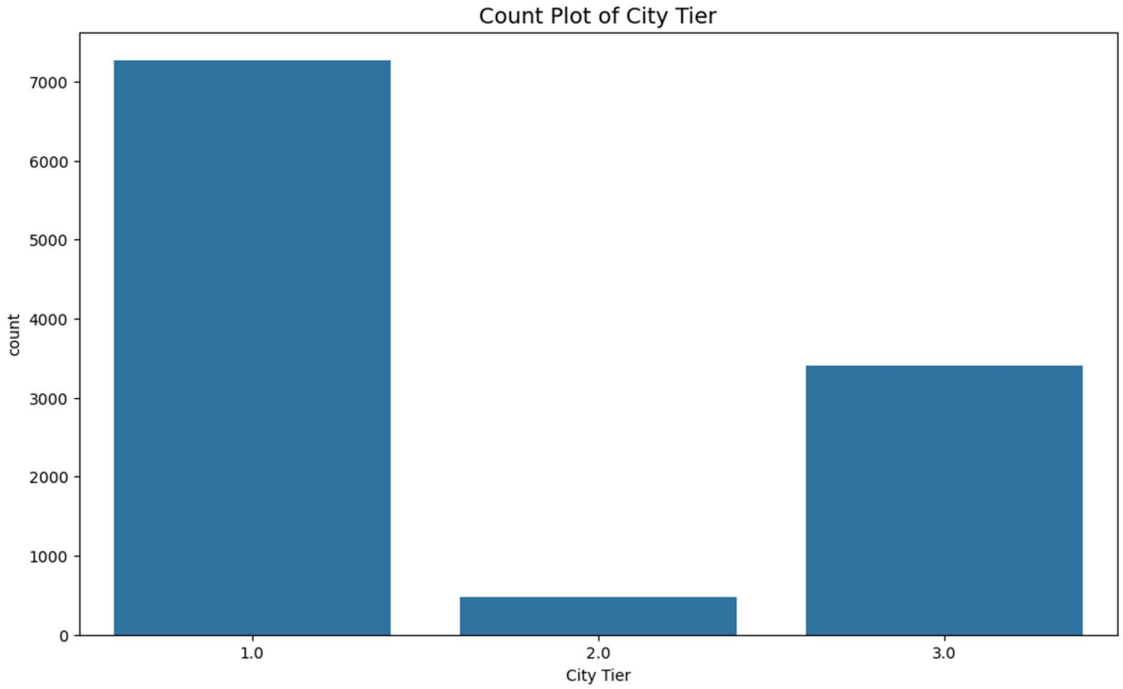
Figure 1 - Countplot of Churned Customers



Insights:

- There are over 8000 customer accounts who have not churned while there are less than 2000 customer accounts who have churned.
- There is a large difference between churned and non-churned accounts, this shows there a significant data imbalance this is our key indicator used for modeling.

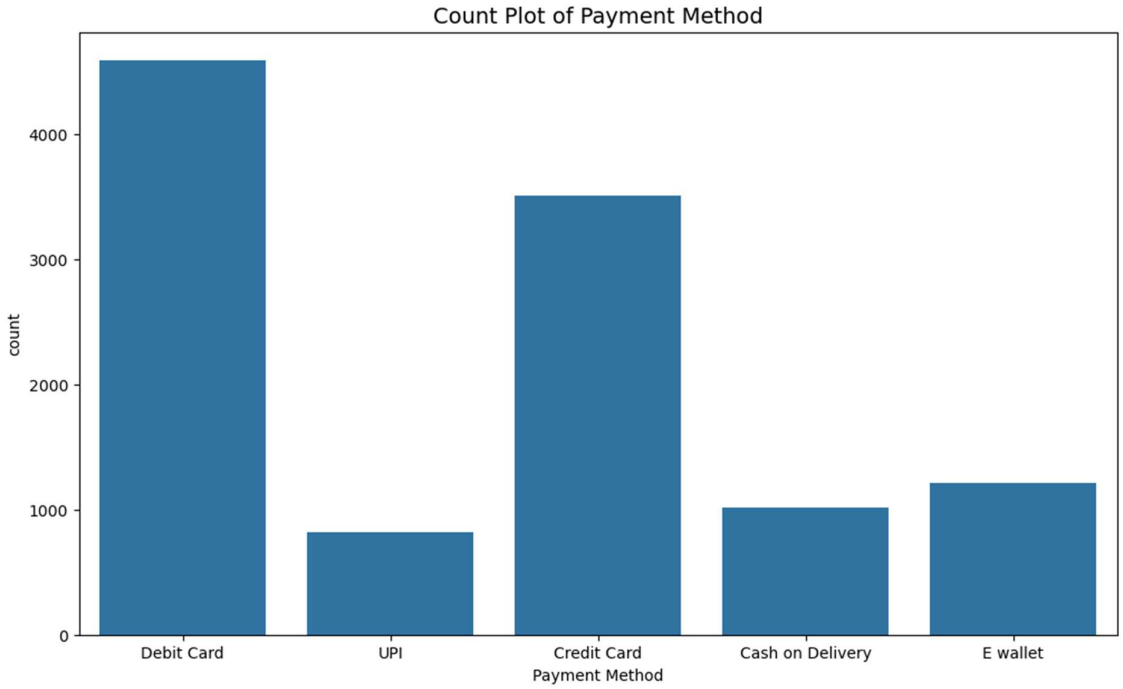
Figure 2 - Countplot of City Tier



Insights:

- A large proportion of customers come from city tier 1, indicating a high population density in this area.
- The lowest number of customers are from tier 2 cities, less than around 500 customers.
- There are around 3000-4000 customers in tier 3 cities.

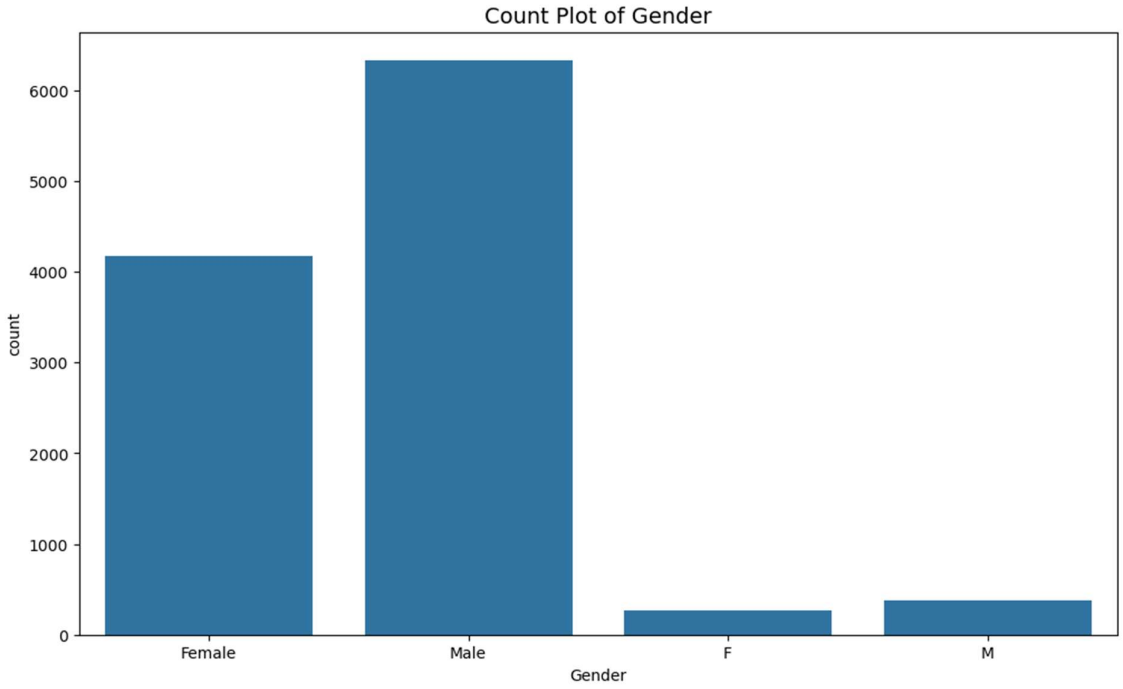
Figure 3 - Countplot of Payment Method



Insights:

- Debit and Credit Cards are the most popular payment methods among customers.
- UPI, cash on delivery and e-wallet are the least used payment methods.

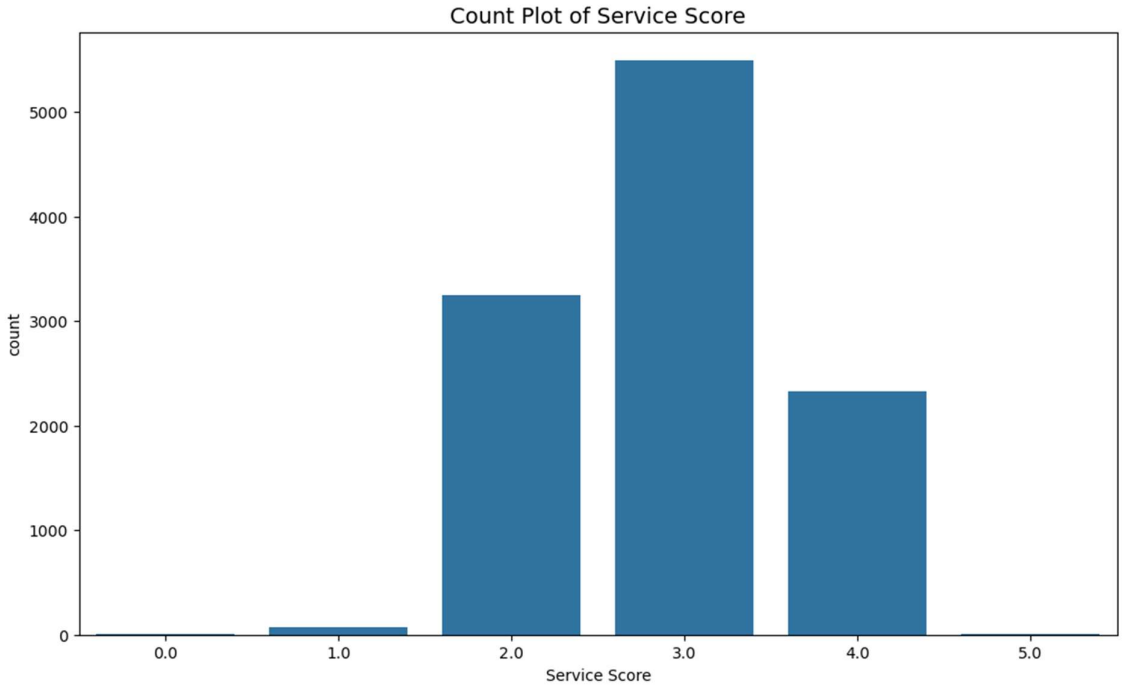
Figure 4 - Countplot of Gender



Insights:

- There are around 6000 male customers compared to around 4000 female customers.
- Though there are more male customers, the difference is not that significant to female customers.

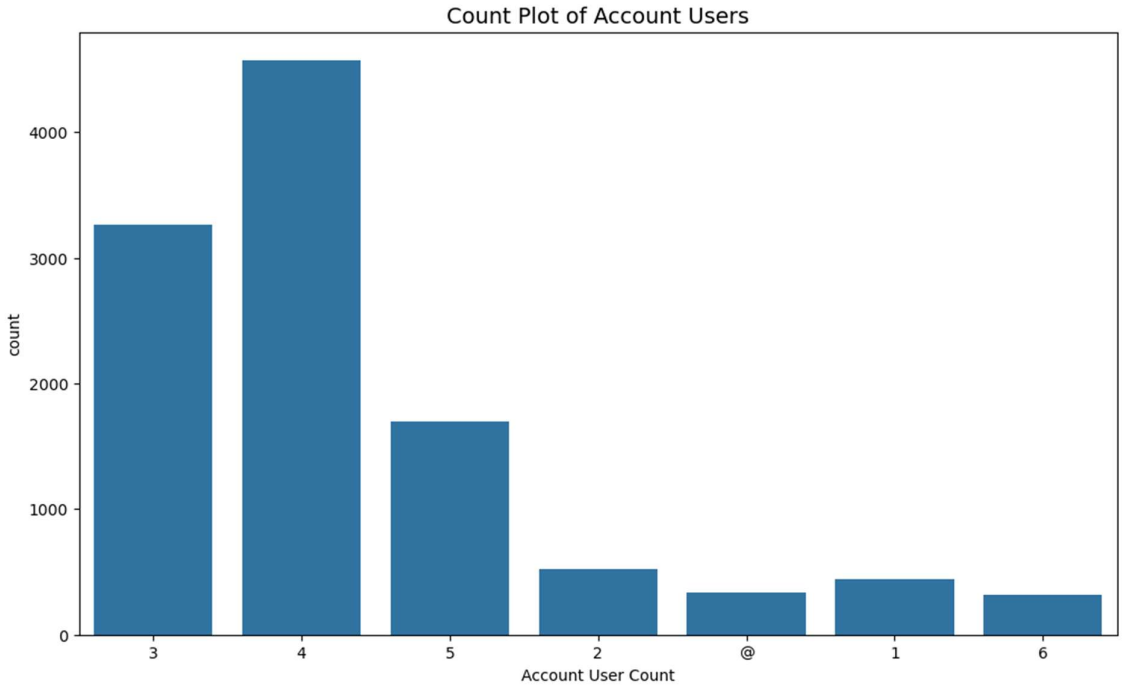
Figure 5 - Countplot of Service Score



Insights:

- Customers rate the service score around 3 which can be considered decent but 4-5 rating is always the target.
- Very few to no customers have rated a service score 5 which is the highest score, this could indicate decline in service quality.
- Majority of customers have rated the service 3 and below.

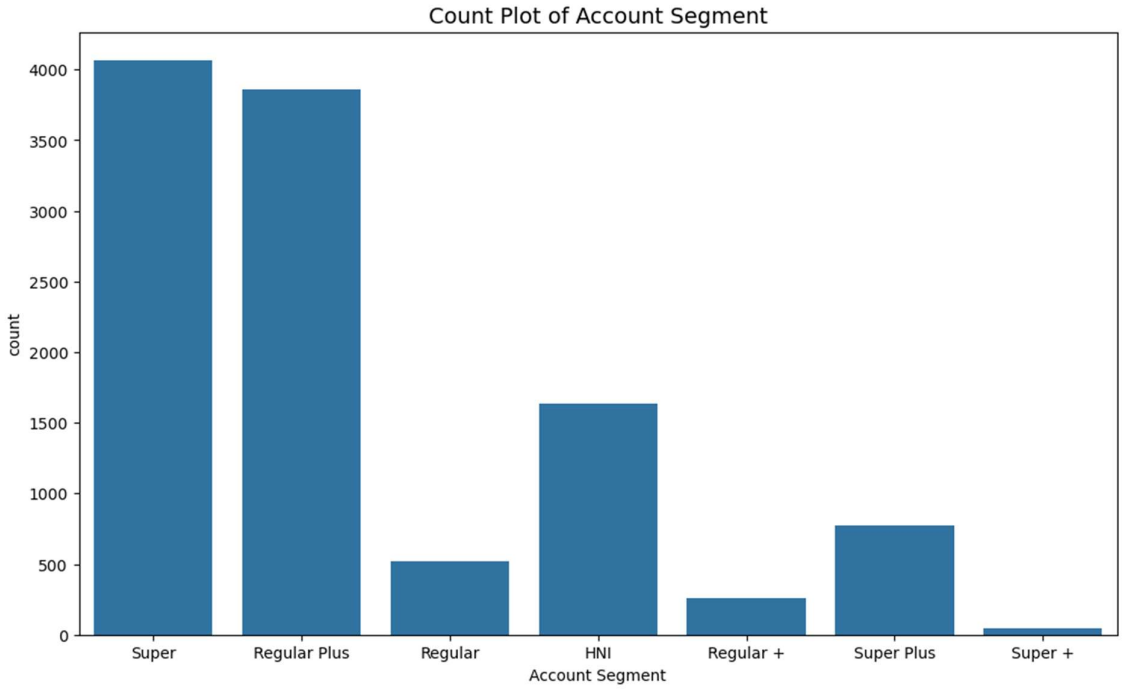
Figure 6 - Countplot of Account Users



Insights:

- The highest number of users in a single account is 4 users followed by 3.
- Account users with a total of 6 are the lowest.
- There are around 1000 customers who have account users of 1 and 2.

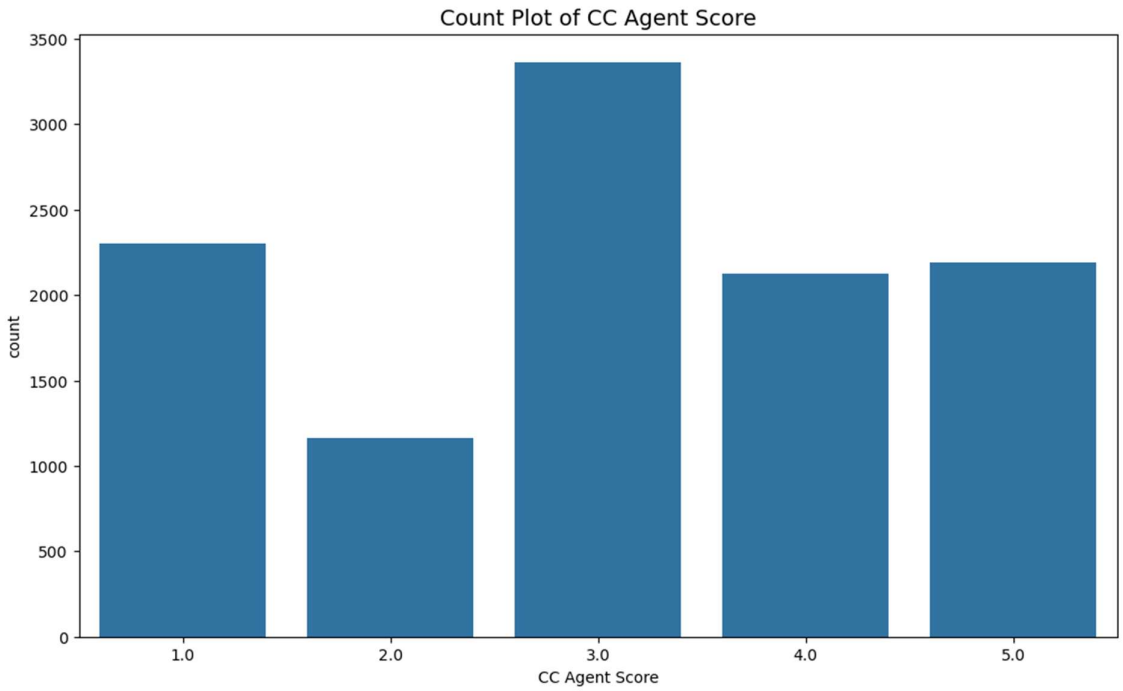
Figure 7 - Countplot of Account Segment



Insights:

- The majority of customers belong to the Super segment, with the Regular segment having the fewest customers.
- Majority of the customer base are on the Regular Plus and Super segment.
- There are around 1700 customers in the HNI Account segment.
- Very few customers in the Regular segment.

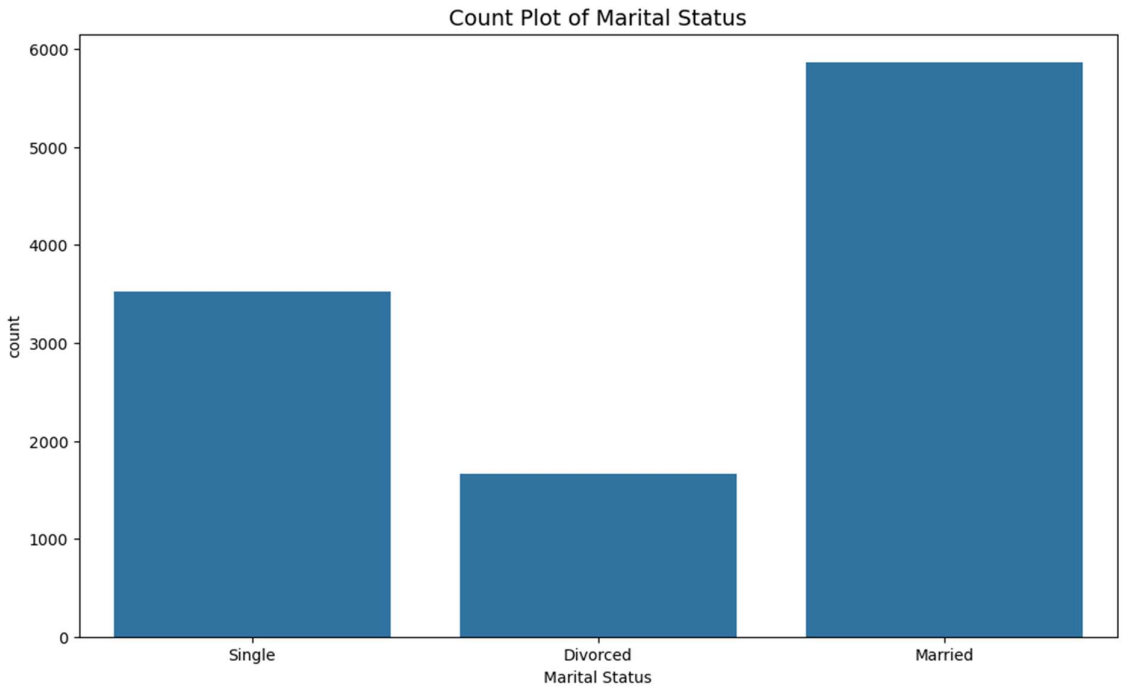
Figure 8 - Countplot of CC Agent Score



Insights:

- Majority of the customers have rated customer care agents with a score and less.
- There are around 2000 customers who rate customer score 4 and 5.
- Agent score 3 is the most common received score.
- There are also a significant number of agents who received score of 1, indicating some poor customer satisfaction.

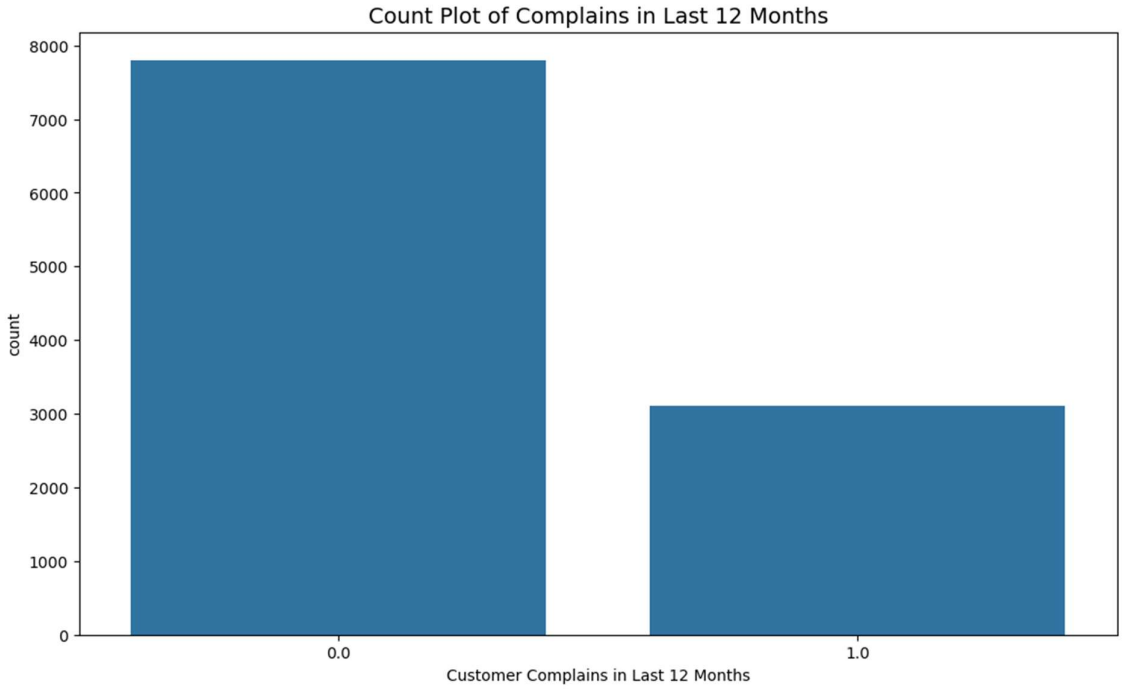
Figure 9 - Countplot of Marital Status



Insights:

- Most customers who use the dth services are married.
- There are around 3500 customers who are single.
- Divorced customers form the smallest group in the demograph.

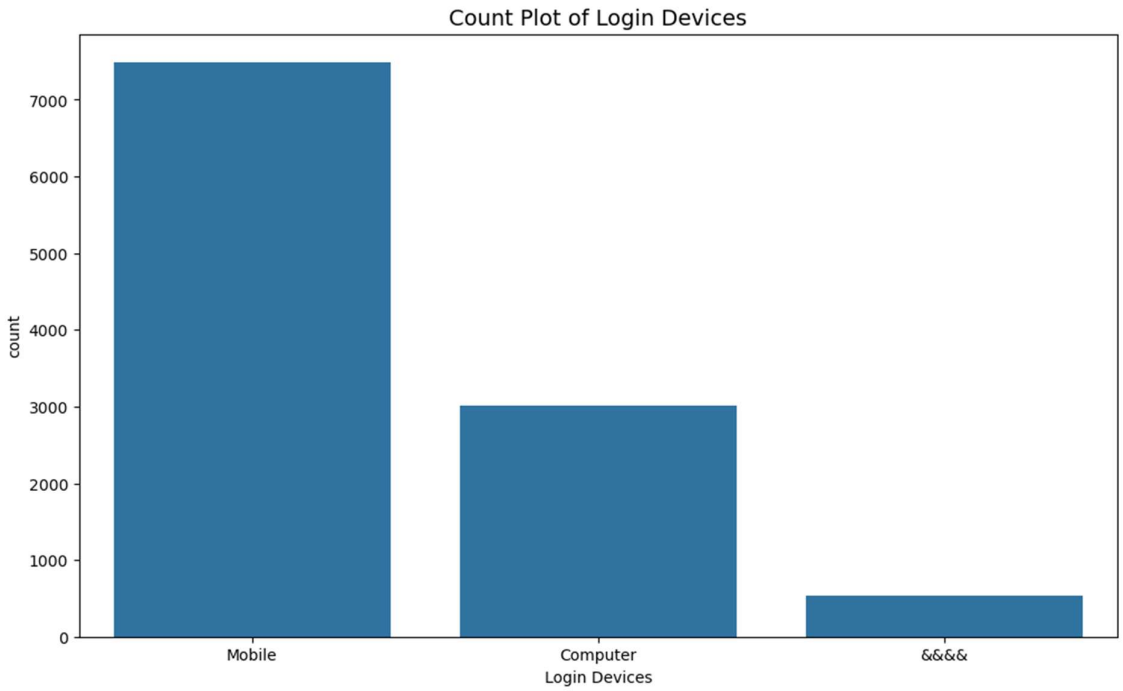
Figure 10 - Countplot of Complains in the Last 12 Months



Insights:

- There have been around 3000 complains made by customer accounts in the last 12 months.
- Majority of the customer base have not filed a complaint in the last 12 months.

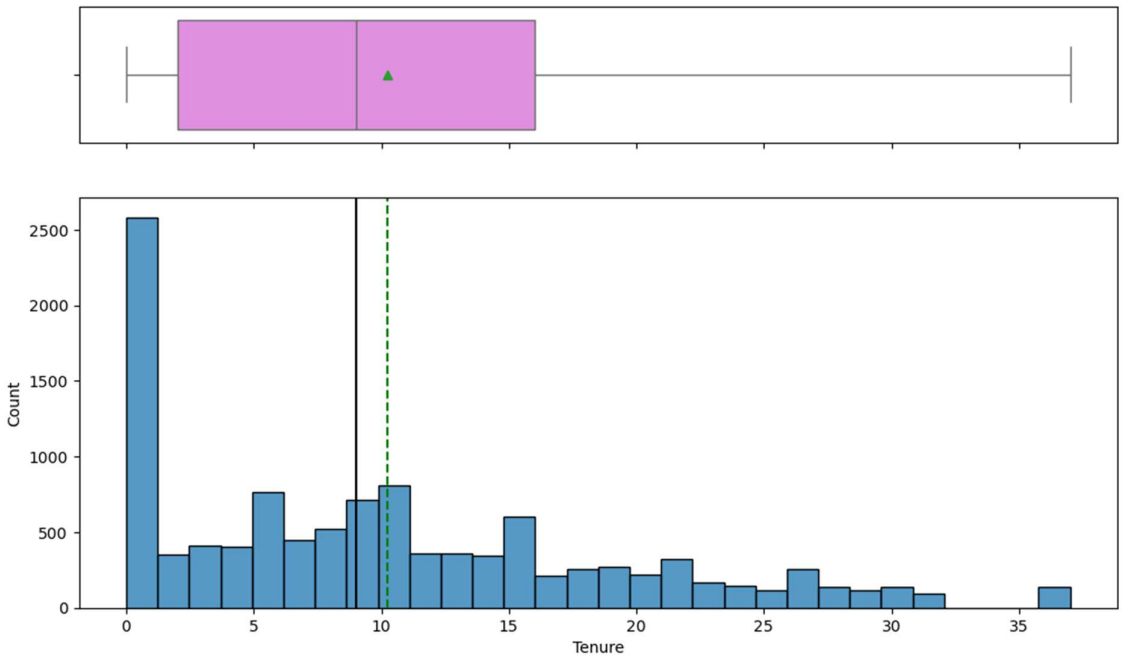
Figure 11 - Countplot of Login Devices



Insights:

- Mobile devices are the preferred choice for customers when accessing DTH services.
- Around 3000 customers use computers for the DTH service.

Figure 12 - Histogram Boxplot of Tenure



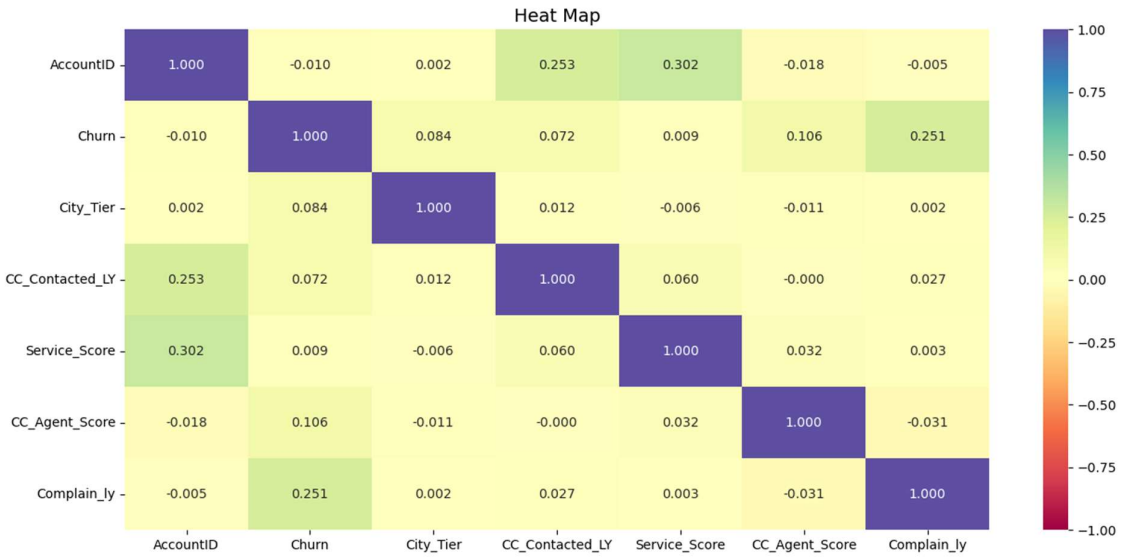
Insights:

- On average, Customers have a tenure of around 10 years.
- There are large number of customers with a tenure of less than 3/2, this shows many customers leave the service within 2 years.

2.2 Bivariate Analysis

2.2.1 Heatmap

Figure 13 - Heatmap of all Numerical Variables

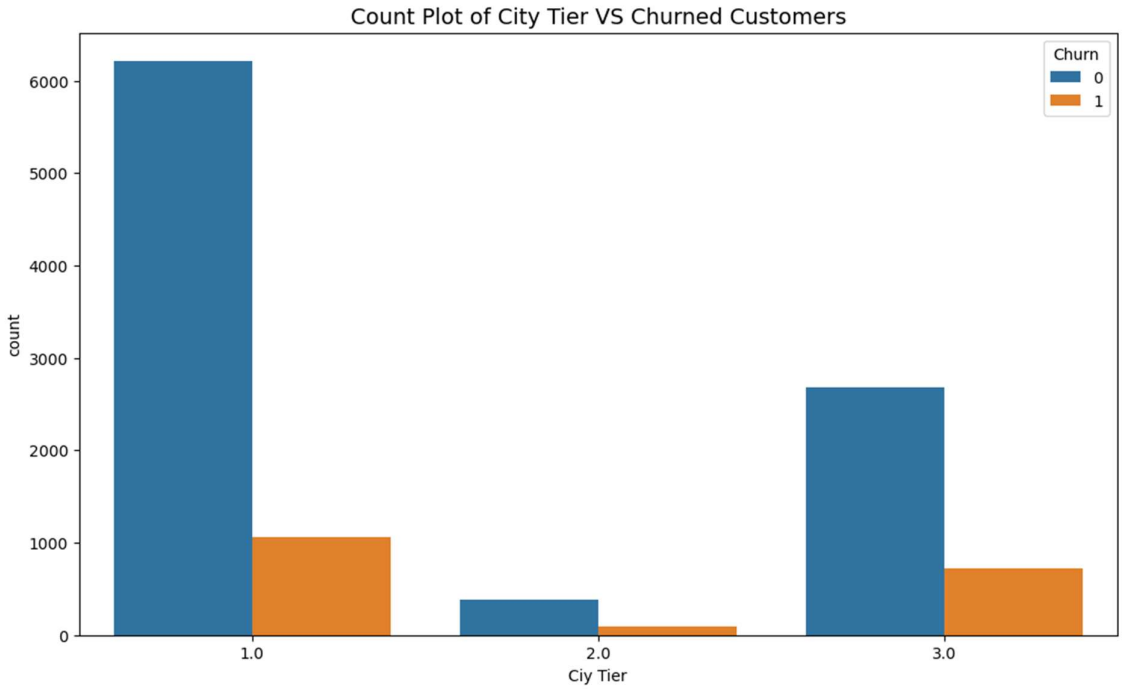


Observations of Heatmap:

- From the heatmap we can see that maximum correlation in the dataset is between Churn and Complain_ly which shows that customers who made a complaint in the last 12 months had a higher chance of churning.
- Correlation between service score and AccountID can be ignored as AccountID is completely unique values.
- There is little correlation among the other variables.

2.2.2 All Variables VS Churn

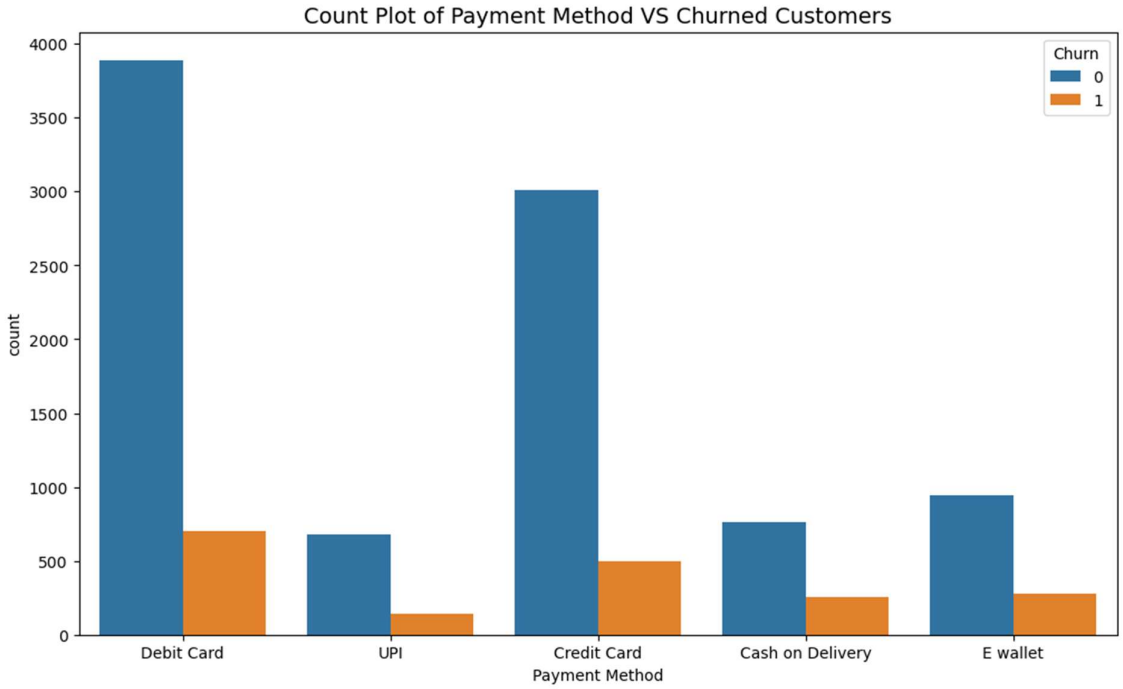
Figure 14 - Countplot of City Tier vs Churned Customers



Insights:

- City tier 1 experiences the highest churn rate compared to city tiers 2 and 3. This is logical as tier 1 cities form the majority group of customers.
- There are almost similar number of customers who have churned between tier 1 and tier 3 cities.

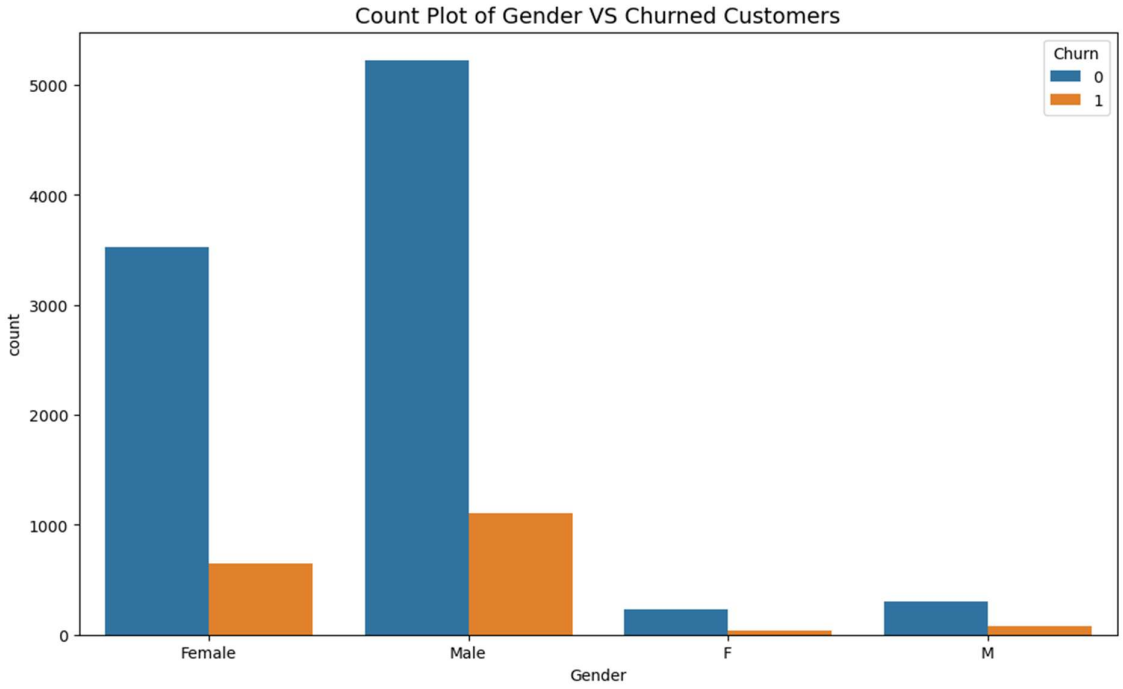
Figure 15 - Countplot of Payment Method vs Churned Customers



Insights:

- Majority of the customers who churned have used Credit and Debit card as primary payment method.

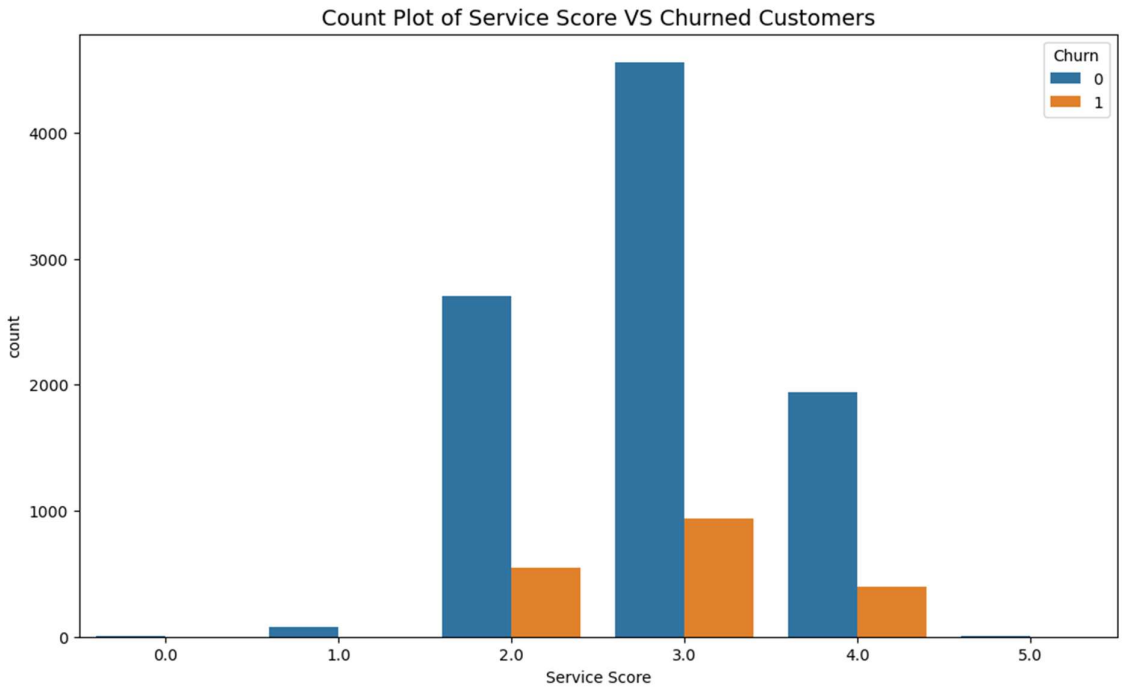
Figure 16 - Countplot of Gender Vs Churned Customers



Insights:

- Male customers have a higher churn rate than female customers.

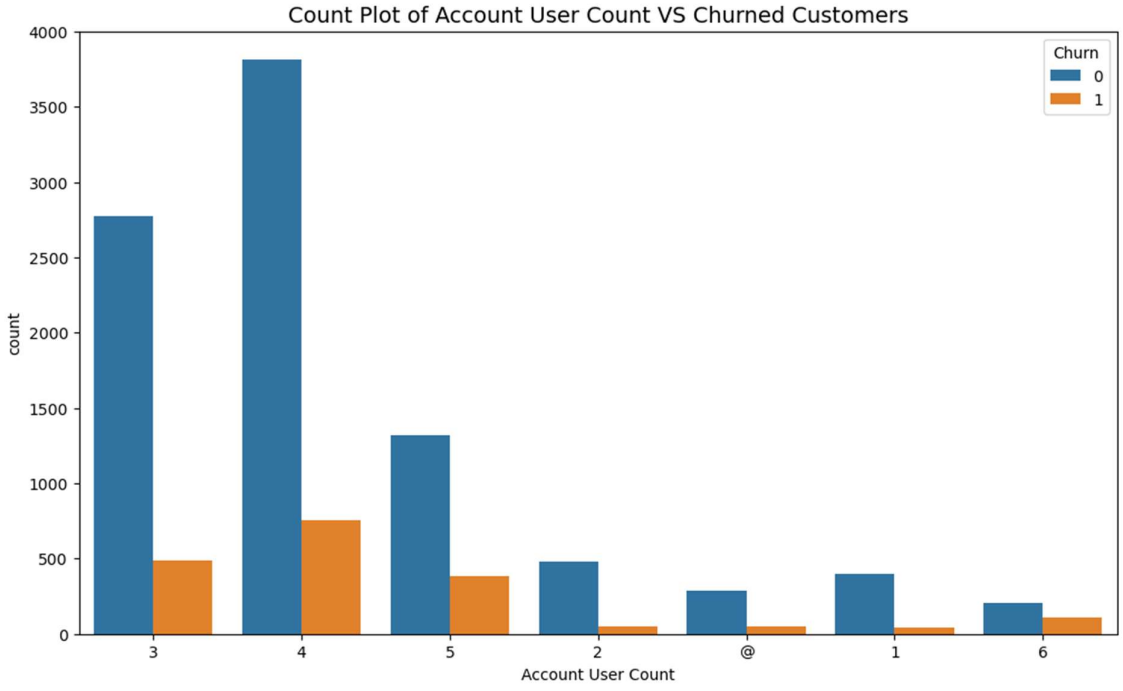
Figure 17 - Countplot of Service Score vs Churned Customers



Insights:

- Service score rating of 3 and below is most common and has the highest number of churned customers.

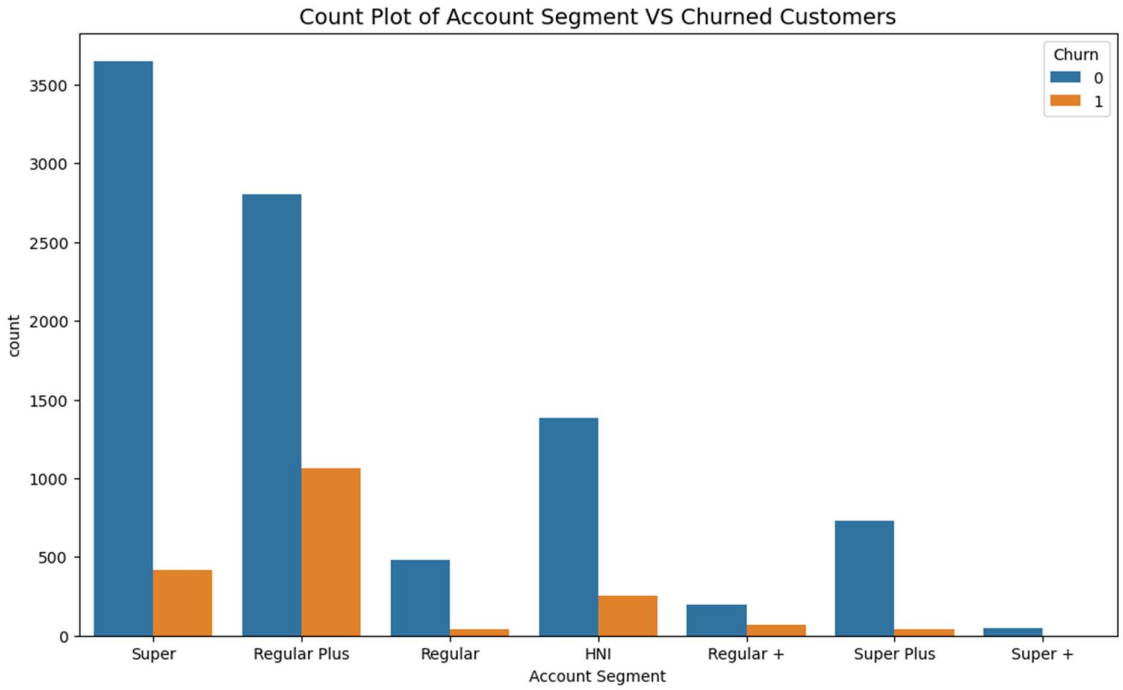
Figure 18 - Countplot of Account User Count vs Churned Customers



Insights:

- Majority of churned customers have a total of 3 and more account users.
- Since majority of these account have a significant number of users, there is an impact in the business when these customers churn.

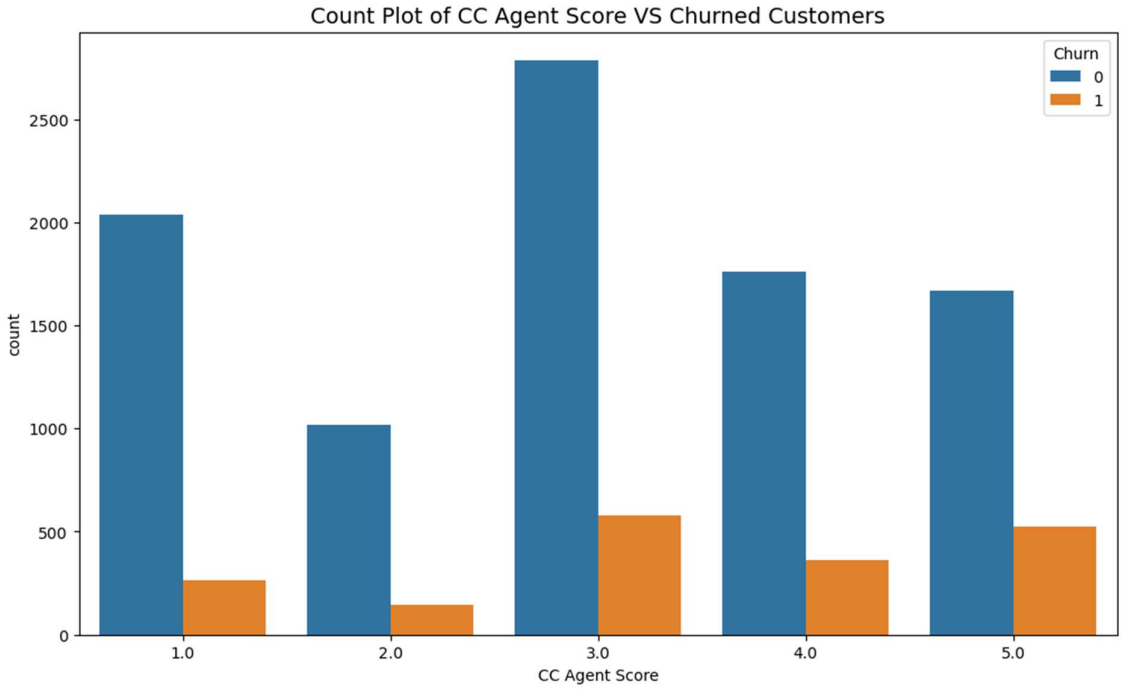
Figure 19 - Countplot of Account Segment vs Churned Customers



Insights:

- The Regular Plus segment shows the highest churn of customers. This could indicate potential poor value in the plan.
- Super Plus and Super segment have a very low churn rate compared to other segments.

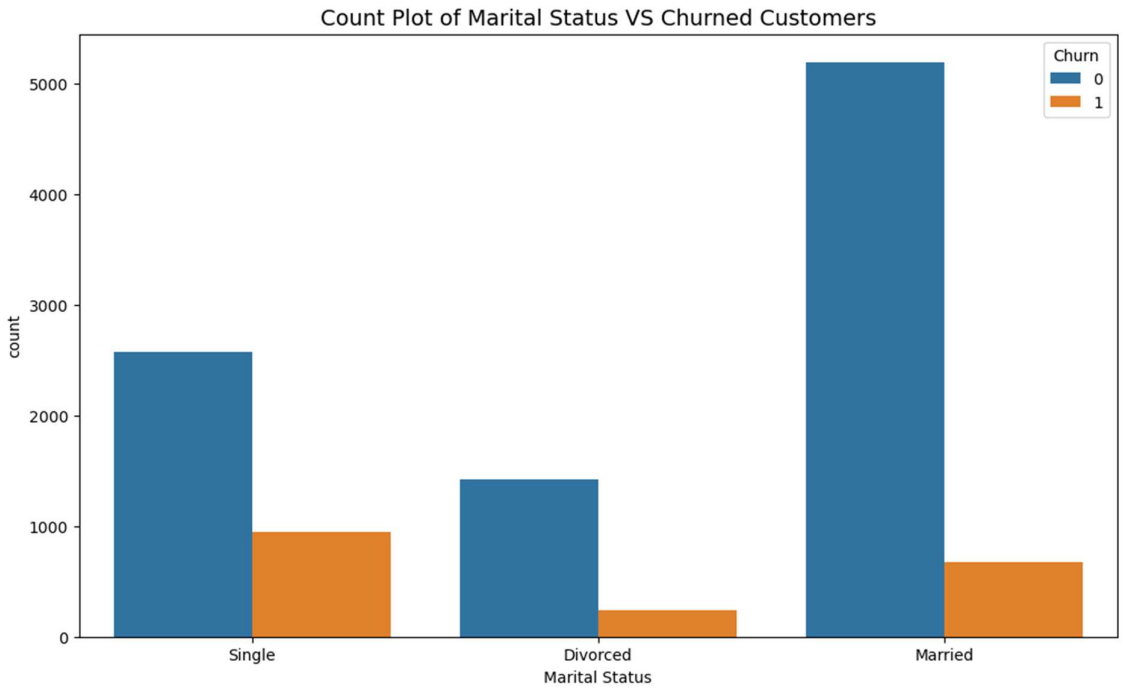
Figure 20 - Countplot of CC Agent Score vs Churned Customers



Insights:

- Majority of customers who have churned have rated an agent score of 3 and below.
- There are a significant number of churned customers who have rated agent score of 4 and above as well.

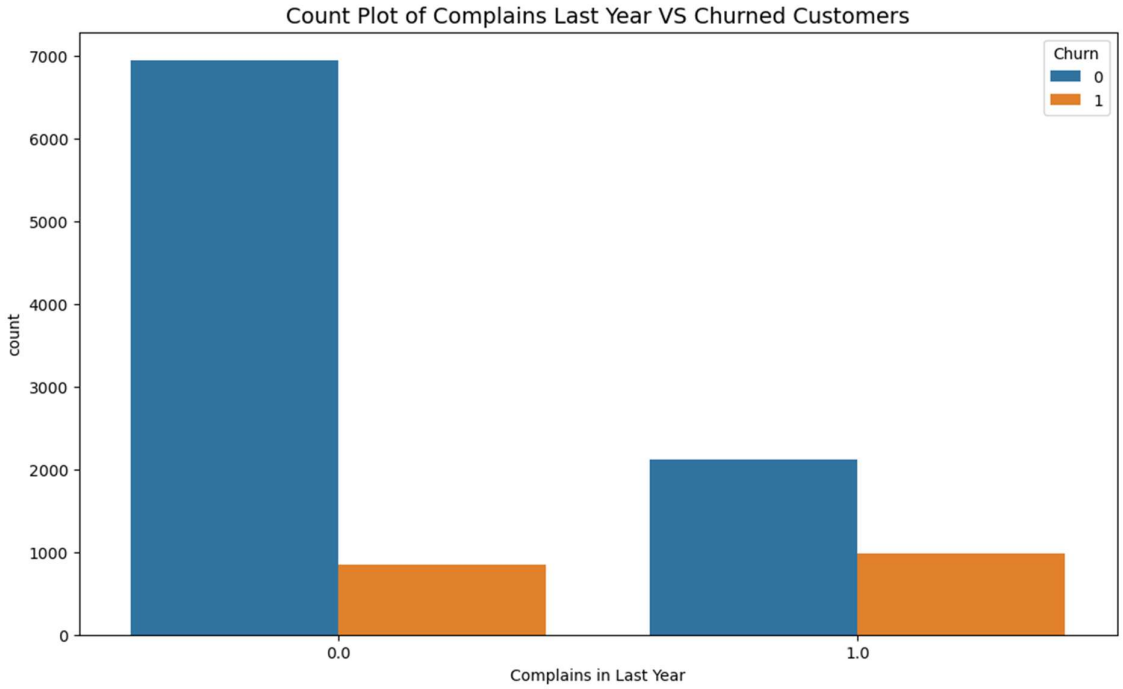
Figure 21 - Countplot of Marital Status vs Churned Customers



Insights:

- Single customers are more prone to churn than those who are divorced or married.
- There are lower number of churned customers in the divorced and married demography.

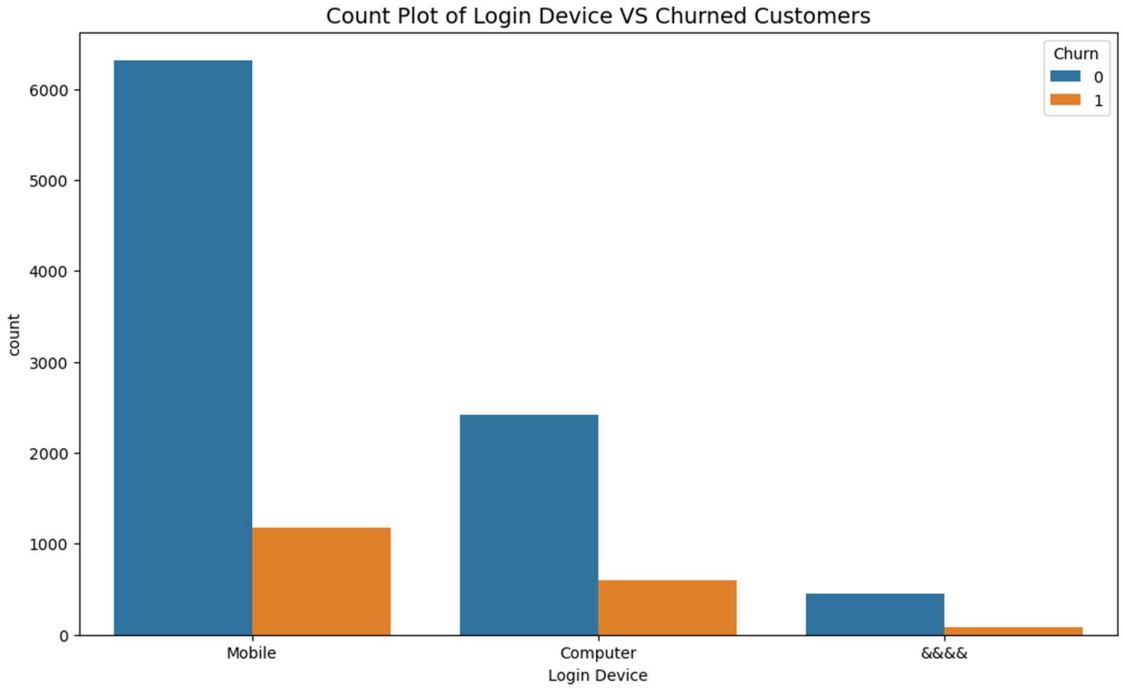
Figure 22 - Countplot of Complains in Last Year vs Churned Customers



Insights:

- Most of the customers who churned have raised a complain in the last 12 months.
- There are around 1000 customers who have churned but have not raised any complaint in the last 12 months.

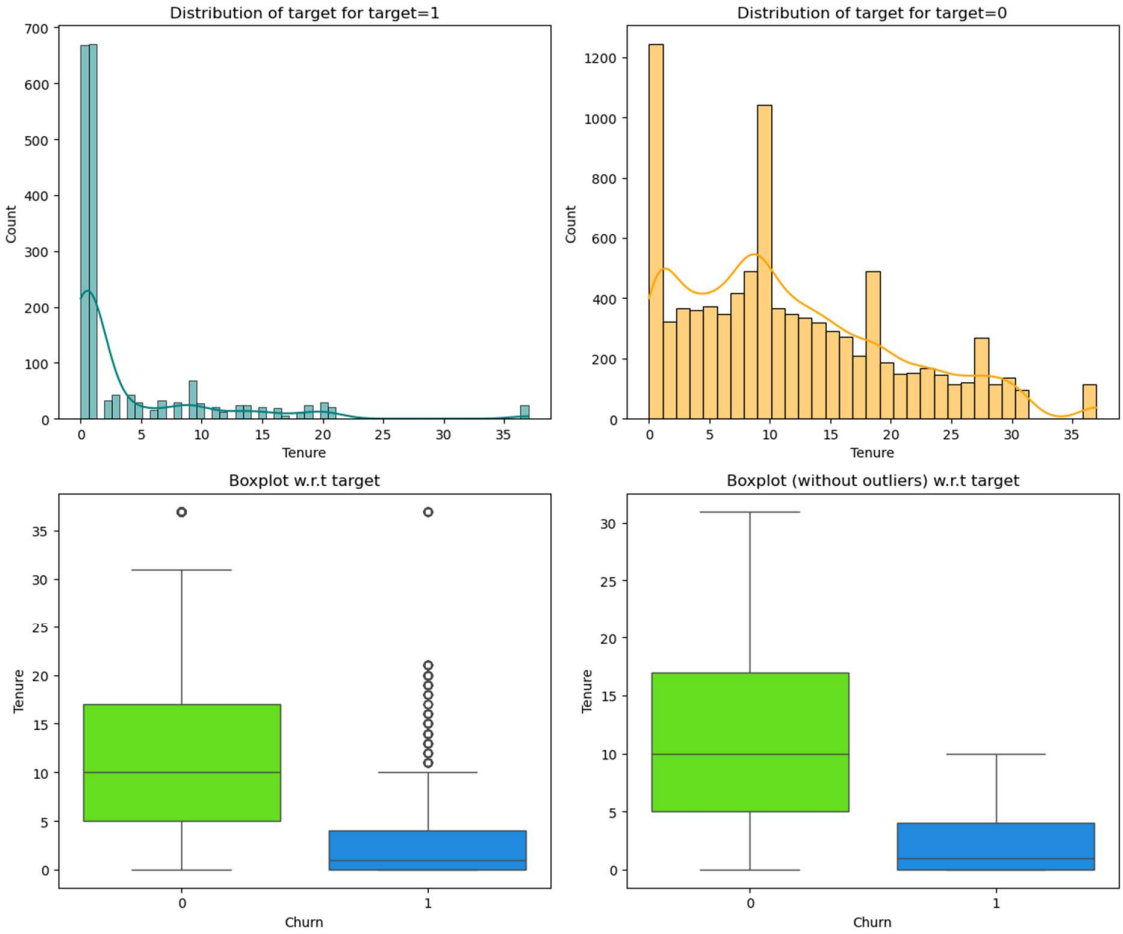
Figure 23 - Countplot of Login Devices vs Churned Customers



Insights:

- Customers accessing services through mobile devices show higher churn rates than computers.

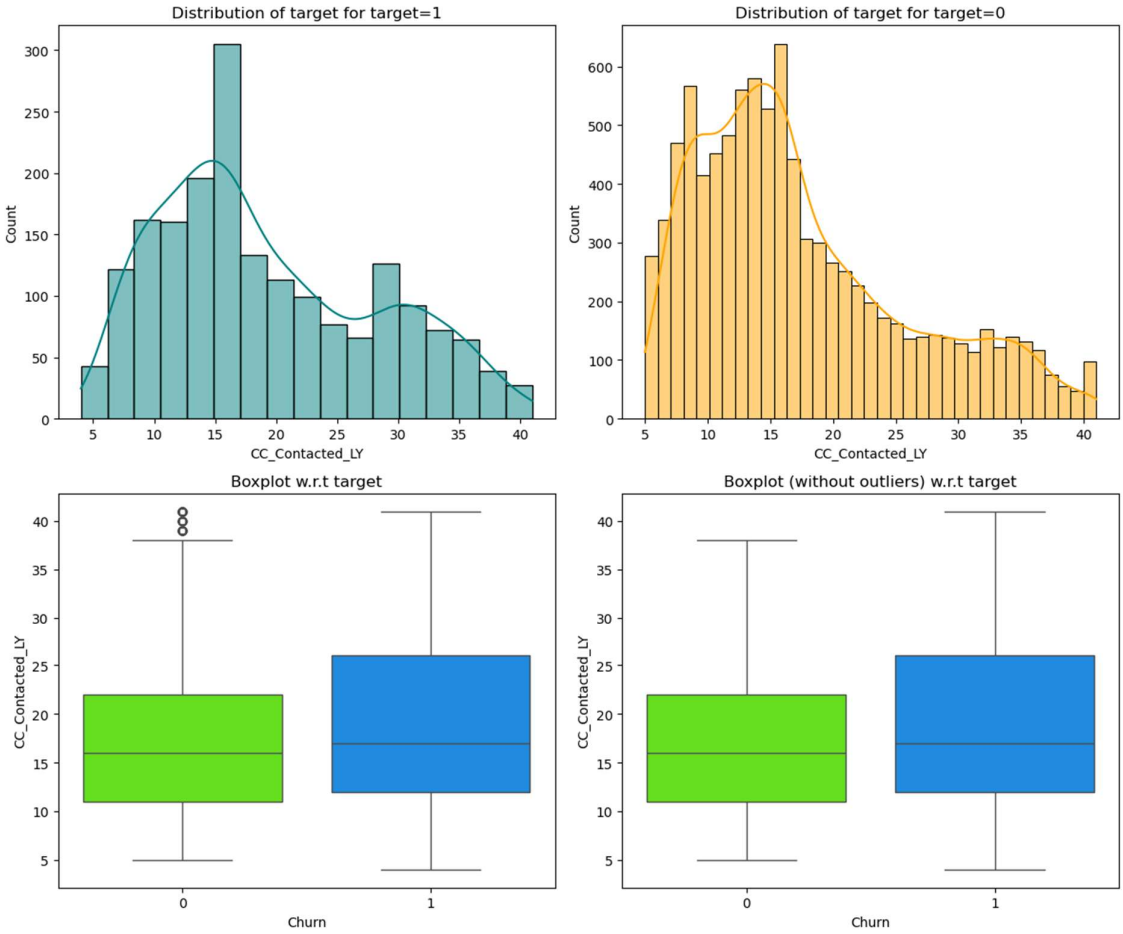
Figure 24 - Histogram Boxplot of Tenure vs Churned Customers



Insights:

- A significant portion of churned customers had a tenure of less than 5 years, indicating early departure from using the services of the company.

Figure 25 - Histogram Boxplot of Customer Contact vs Churned Customers



Insights:

- Majority of the customers who have churned have had some contact with the customer service and raised a complaint in the last 12 months.

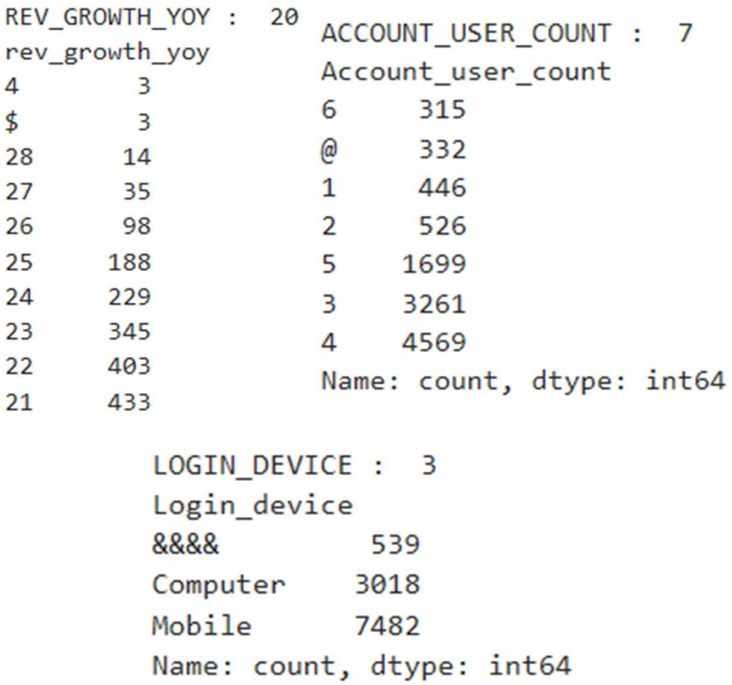
DATA CLEANING AND PRE-PROCESSING

3.1 Data Cleaning

Irregular values and variables that have no impact on the study can be removed from the dataset. In this case. “AccountID” variables contains all unique values but it will not be removed for now as one ID can have the same data for different variables. The variable will be removed.

On inspection of all the unique values in the dataset, there can be seen a number of incorrect entries for some variables

Figure 26 - Snapshot of Incorrect Values in the Variables



From the snippet of some variables showing incorrect data such as “$”, “@”, “&&&&” and more, these incorrect values will be replaced with NaN or missing value and will be treated as a null value.

3.2 Missing Value Treatment

Figure 27 - Total number of Missing Values in the Variables

```
AccountID                   0
Churn                       0
Tenure                      0
City_Tier                 112
CC_Contacted_LY           102
Payment                   109
Gender                    108
Service_Score              98
Account_user_count        444
account_segment            97
CC_Agent_Score            116
Marital_Status            212
rev_per_month             791
Complain_ly               357
rev_growth_yoy              3
coupon_used_for_payment     3
Day_Since_CC_connect      358
cashback                  471
Login_device              760
dtype: int64
```

The missing values need to treated so that it does not impact future analysis and model building, The following steps are taken

- The missing values in the variables, City_Tier, Payment, Gender, Service_Score, account_segment, CC_Agent_Score, Marital_Status, Complain_ly and Login_device will be replaced with the Mode on the data as they are considered as categorical variables.
- The missing values in the variables, CC_Contacted_LY, Account_user_count, rev_per_month, rev_growth_yoy, coupon_used_for_payment, Day_Since_CC_connect and cashback will be replaced with the Median of the data as they are continuous numerical variables.

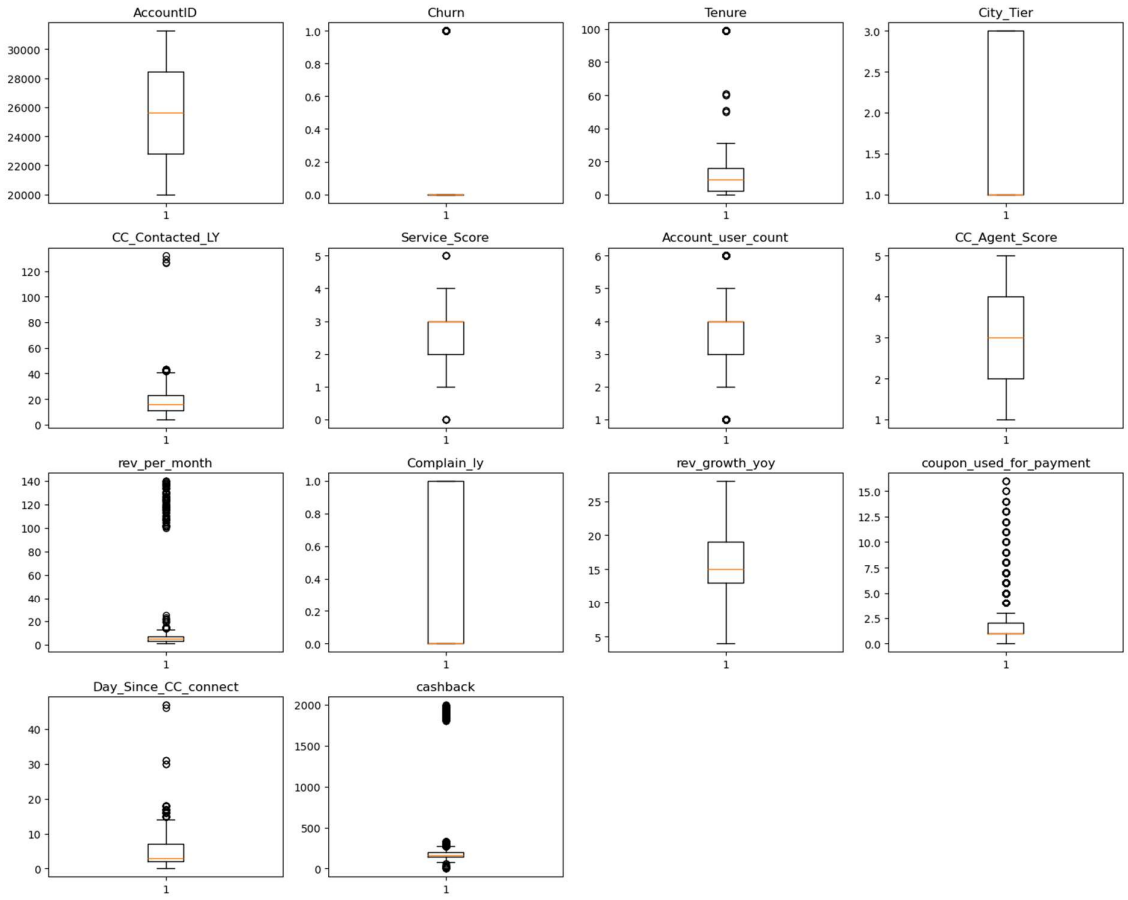
Figure 28 - Missing Values in the Variables Post Treatment

```
AccountID                0
Churn                    0
Tenure                   0
City_Tier                0
CC_Contacted_LY          0
Payment                  0
Gender                   0
Service_Score            0
Account_user_count       0
account_segment          0
CC_Agent_Score           0
Marital_Status           0
rev_per_month            0
Complain_ly              0
rev_growth_yoy           0
coupon_used_for_payment  0
Day_Since_CC_connect     0
cashback                 0
Login_device             0
dtype: int64
```

Following treatment, there are no more missing values in the dataset.

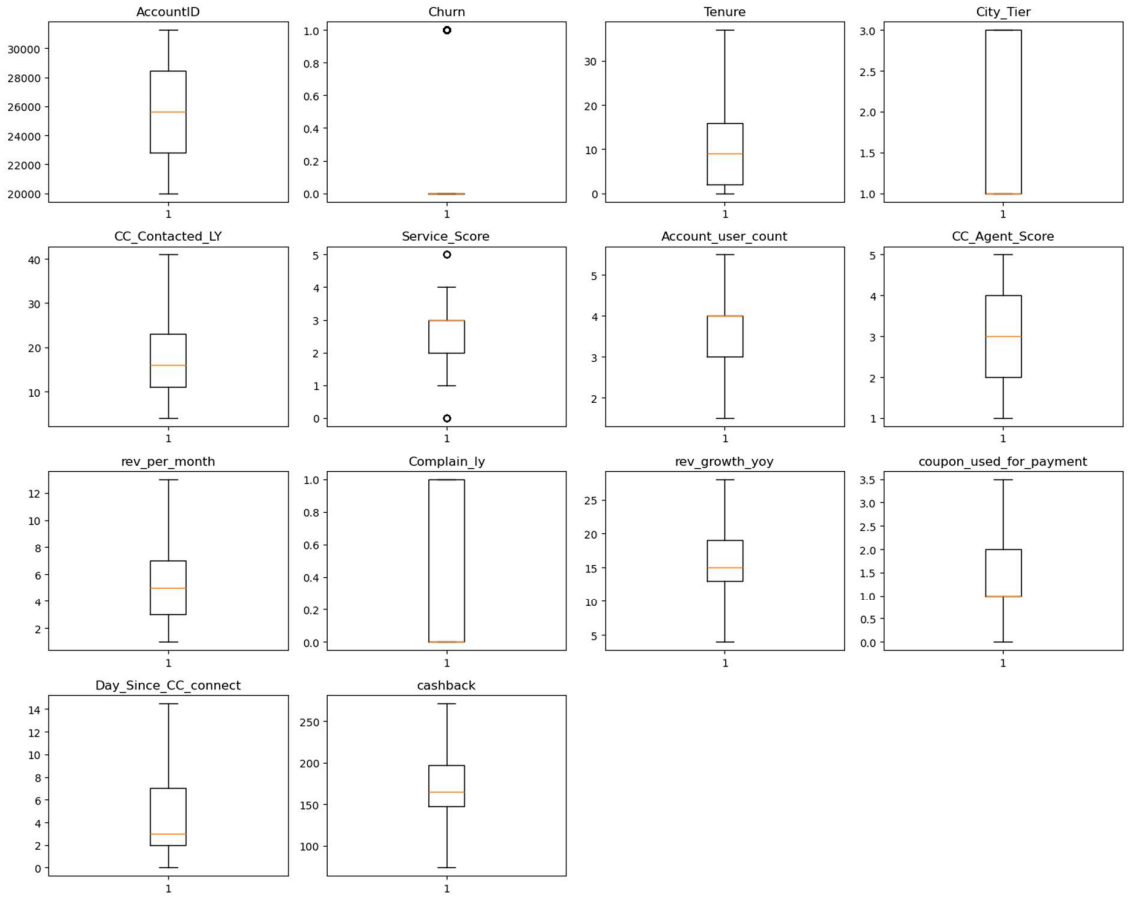
3.3 Outlier Treatment

Figure 29 - Boxplots of Numerical Variables with Outlier



There are a number of outliers present in the variables of the dataset and will need to be treated, Churn which is our target variable and Service_Score shows outlier however as it only contains categorical values, this will not be treated. The outliers present in the other variables will be treated with the upper and lower range of the variable.

Figure 30 - Boxplot of Numerical Variables Post Treatment



3.4 Data Preparation

Data transformation is the process of altering the values in a dataset to improve the performance of the model. All categorical variables were encoded into numerical variables to perform model building. The dataset is then split into training and testing data in the ratio of 70:30.

There is a huge imbalance in our key variable “Churn”, there are 9364 entries of non-churned customers and 1896 entries of churned customers. This imbalance will affect the performance of the model so we use a technique called SMOTE or Synthetic Minority Over-sampling Technique which is a technique used to handle class imbalance in datasets by creating synthetic examples of the minority class. Instead of simply duplicating samples, it generates new ones by interpolating between existing ones, which helps the model learn better decision boundaries without overfitting.

The values of each of the variable need to be normalized as the data ranges are very varied for each of the variables which can be seen by comparing the ranges of some numerical variables like cashback with a variable like coupon_used_for_payment or rev_per_month. The dataset will be scaled using the MinMax scaler from the sklearn library in python.

3.5 Feature Selection

Feature selection is the process of identifying and choosing the most relevant input variables for the predictive model to improve performance, reduce complexity, and avoid overfitting. For this we use VIF or Variance Inflation Factor method which quantifies how much the variance of a regression coefficient is inflated due to multicollinearity with other features. A VIF value greater than 5 or 10 typically indicates high multicollinearity, which can destabilize model coefficients. We drop these redundant predictors and improve model interpretability.

We drop the columns 'Service_Score', 'cashback' and 'rev_growth_yoy' as they have VIF values greater than 5.
The dataset is now pre-processed and ready for Model Building.

MODEL BUILDING

The model will be built on the training data and then tested on the unseen data also known as the testing data to check the performance of the model. Each model will be evaluated on these metrics.

- Accuracy which measures the percentage of correctly classified instances out of all predictions. Higher accuracy indicates better overall model performance.
- Precision which reflects how many of the predicted positive cases are actually correct, ensuring minimal false positives. Important when incorrect predictions have significant consequences.
- Recall which captures how well the model identifies actual positive cases, focusing on minimizing incorrect predictions. It is useful when missing positive cases is costly.
- F1 which is the harmonic mean of precision and recall, balancing both to give a single performance score.
- ROC-AUC which measures a model's ability to distinguish between classes, with a score closer to 1.0 indicating better performance.

For the case of customer churn prediction, the most important metric is Recall which ensures that most actual churners are detected, minimizing missed opportunities for customer retention. A high recall helps businesses intervene before customers leave. Precision score is also important to ensure all retention efforts target the right customers instead of wasting resources on those unlikely to churn.

The following models are built.

Individual Models:

1. **Logistic Regression:**

   This model offers a fast, transparent baseline with probabilistic outputs which is easy to interpret which factors increase churn odds. The model is built by loading the logistic regression class from the scikit-learn library in Python.

Figure 31 - Logistic Regression Evaluation Scores

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Logistic Regression	0.788	0.814	0.774	0.793	0.765	0.784	0.400	0.530	0.841

   The Logistic Regression model demonstrates moderate training performance with an accuracy of 0.788, recall of 0.814, precision of 0.774, and an F1 score of 0.793, none of which reach a high-performance threshold. On the test data, the accuracy remains at 0.765, but precision falls to 0.400, suggesting the model struggles to limit false positives. The recall holds at 0.784, though the F1 score drops to 0.530, indicating an imbalanced trade-off between precision and recall. These results imply the model may not generalize well to unseen data. However, with an AUC score of 0.841, it retains a reasonable ability to distinguish between classes

2. **Decision Tree:**

   This model captures nonlinear interactions and decision thresholds and it is inherently very easily explainable. The model is built by loading the decision tree classifier from the scikit-learn library in Python.

Figure 32 - Decision Tree Evaluation Scores

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Decision Tree Model	1.000	1.000	1.000	1.000	0.906	0.867	0.671	0.757	0.890

The Decision Tree model shows perfect performance on the training set, with maximum scores of 1.000 for accuracy, recall, precision, and F1 indicating that it fits the training data exactly with no errors. However, the test results tell a different story: while accuracy remains high at 0.906 and recall at 0.867, precision drops to 0.671, leading to an F1 score of 0.757. This gap between training and test performance signals likely overfitting, where the model memorized patterns from the training data but struggles with generalization. Still, an AUC of 0.890 suggests it maintains solid class separation ability on unseen data.

3. **Linear Discriminant Analysis:**

This model finds the projection that best separates churners from non-churners under a Gaussian assumption, reducing noise and dimensionality. The model is built by loading the linear discriminant analysis classifier from the scikit-learn library in Python.

Figure 33 - Linear Discriminant Analysis Evaluation Scores

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Linear Discriminant Analysis	0.783	0.824	0.762	0.791	0.745	0.788	0.378	0.511	0.836

The Linear Discriminant Analysis model shows moderate performance on the training set, with accuracy at 0.783, recall at 0.824, precision at 0.762, and an F1 score of 0.791—none reaching high-performance levels. On the test set, performance declines, with accuracy at 0.745 and F1 score at 0.511. Precision drops significantly to 0.378, indicating the model generates many false positives, while recall remains relatively steady at 0.788. This imbalance suggests the model struggles to maintain prediction quality on new data. Nevertheless, an AUC score of 0.836 indicates it still has a fair capacity to distinguish between classes.

4. **Artificial Neural Network:**

The model learns complex, high-order feature interactions that simpler models might miss. The model is built by loading the MLP classifier from the scikit-learn library in Python.

Figure 34 - Artificial Neural Network Evaluation Scores

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Artificial Neural Network	0.955	0.963	0.948	0.955	0.911	0.861	0.691	0.767	0.959

The Artificial Neural Network (ANN) model shows strong training performance, with accuracy at 0.955, recall at 0.963, precision at 0.948, and an F1 score of 0.955—

indicating effective and well-balanced learning on the training data. On the test set, the model maintains a high accuracy of 0.911, though precision drops to 0.691, suggesting some increase in false positives. The recall remains high at 0.861, and the F1 score of 0.767 reflects a reasonable balance between precision and recall, albeit weaker than in training. This performance gap suggests a slight overfitting tendency. However, the high AUC of 0.959 confirms that the model retains excellent class separation capabilities on new data.

5. **Support Vector Machine:**

The model excels in high-dimensional spaces and can carve out tight decision boundaries which are helpful when churn-driving signals are subtle. The model is built by loading the SVM classifier from the scikit-learn library in Python.

Figure 35 - Support Vector Machine Evaluation Scores

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Support Vector Machine	0.900	0.914	0.889	0.901	0.875	0.844	0.590	0.695	0.922

The Support Vector Machine (SVM) model shows solid performance on the training set, achieving an accuracy of 0.900, recall of 0.914, precision of 0.889, and an F1 score of 0.901, indicating strong and consistent learning. On the test set, accuracy remains high at 0.875, and recall stays relatively strong at 0.844, but precision drops noticeably to 0.590, pointing to a rise in false positives. This results in an F1 score of 0.695, showing some imbalance between correctly identified and incorrectly predicted positives. This discrepancy between training and testing suggests the model may not generalize its precision as effectively. However, the AUC of 0.922 highlights the model's strong ability to distinguish between the classes on unseen data.

6. **K-Nearest Neighbors:**

The model is a nonparametric, instance-based approach that flags a customer as high-risk if similar profiles in history have churned. The model is built by loading the K neighbors' classifier from the scikit-learn library in Python.

Figure 36 - K-Nearest Neighbors Evaluation Scores

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
K-Nearest Neighbors	0.955	0.975	0.938	0.956	0.901	0.916	0.644	0.757	0.964

The K-Nearest Neighbors (KNN) model performs very well on the training set, achieving high scores across all metrics—accuracy at 0.955, recall at 0.975, precision at 0.938, and an F1 score of 0.956—indicating the model fits the training data closely

and captures most patterns effectively. On the test set, it retains strong recall at 0.916 and accuracy at 0.901, though precision dips to 0.644, suggesting an increase in false positives when applied to new data. The resulting F1 score of 0.757 shows a moderate balance between precision and recall. Despite this drop in precision, the AUC of 0.964 confirms the model maintains excellent capability in distinguishing between the classes.

Ensemble Models:

1. **Random Forest:**

   The model aggregates many de-correlated trees to reduce variance and robust to noisy features and unlikely to overfit on churn's complex patterns. The model is built by loading the random forest classifier from the scikit-learn library in Python.

Figure 37 - Random Forest Evaluation Scores

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Random Forest Model	1.000	1.000	1.000	1.000	0.955	0.886	0.854	0.870	0.984

The Random Forest model shows flawless performance on the training set, with perfect scores of 1.000 across accuracy, recall, precision, and F1—indicating it learned the training data completely without errors. On the test set, the model maintains strong generalization, achieving an accuracy of 0.955 and a high recall of 0.886, which means it correctly identifies most positive cases. Precision remains solid at 0.854, suggesting a relatively low false positive rate. The F1 score of 0.870 reflects a well-balanced prediction capability on unseen data. Additionally, an AUC of 0.984 confirms excellent class separation, showing the model handles varying decision thresholds very effectively.

2. **Gradient Boosting:**

   The model sequentially corrects its own mistakes, honing in on hard-to-predict churn cases and often pushing overall accuracy/recall higher. The model is built by loading the gradient boosting classifier from the scikit-learn library in Python.

Figure 38 - Gradient Boosting Evaluation Scores

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Gradient Boosting	0.871	0.863	0.878	0.870	0.866	0.784	0.575	0.663	0.914

The Gradient Boosting model delivers reasonably strong training performance, with accuracy at 0.871 and F1 score at 0.870, showing that it fits the training data

effectively while maintaining a healthy balance between recall (0.863) and precision (0.878). However, on the test set, performance softens—accuracy dips slightly to 0.866, and recall remains fairly strong at 0.784, but precision falls to 0.575, indicating the model struggles more with false positives when exposed to unseen data. The resulting F1 score of 0.663 reflects this decline in balance between sensitivity and specificity. Still, the AUC score of 0.914 suggests the model continues to exhibit solid class discrimination and remains fairly reliable across varying threshold levels.

3. **Adaboosting:**

The model reweights misclassified customers on each round, sharpening focus on borderline churners and improving discrimination where it matters most. The model is built by loading the adaboost classifier from the scikit-learn library in Python.

Figure 39 - Adaboosting Evaluation Scores

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Adaboosting	0.825	0.817	0.831	0.824	0.829	0.758	0.495	0.599	0.884

The Adaboosting model performs moderately well on the training set, with accuracy at 0.825, recall at 0.817, precision at 0.831, and an F1 score of 0.824—showing reasonably balanced learning but not reaching top-tier performance. On the test data, accuracy slightly improves to 0.829, and recall stays respectable at 0.758, but precision drops to 0.495, suggesting it makes a fair number of false positive predictions. The F1 score drops to 0.599, indicating an overall decline in how well the model balances correctness and completeness on unseen data. However, the AUC of 0.884 suggests the model still has solid ability to distinguish between the classes across various decision thresholds.

MODEL VALIDATION

All the models built will be validated and compared with each other and the most optimal models will be tuned to get better scores and identify the most optimal model which will be used to predict customer churning.

The below table show the accuracy, recall, precision, F1 scores and ROC-AUC score of all the models built

Figure 40 - Model Evaluation Scores Comparison

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Logistic Regression	0.788	0.814	0.774	0.793	0.765	0.784	0.400	0.530	0.841
Decision Tree Model	1.000	1.000	1.000	1.000	0.906	0.867	0.671	0.757	0.890
Linear Discriminant Analysis	0.783	0.824	0.762	0.791	0.745	0.788	0.378	0.511	0.836
Artificial Neural Network	0.955	0.963	0.948	0.955	0.911	0.861	0.691	0.767	0.959
Support Vector Machine	0.900	0.914	0.889	0.901	0.875	0.844	0.590	0.695	0.922
K-Nearest Neighbors	0.955	0.975	0.938	0.956	0.901	0.916	0.644	0.757	0.964
Random Forest Model	1.000	1.000	1.000	1.000	0.955	0.886	0.854	0.870	0.984
Gradient Boosting	0.871	0.863	0.878	0.870	0.866	0.784	0.575	0.663	0.914
Adaboosting	0.825	0.817	0.831	0.824	0.829	0.758	0.495	0.599	0.884

The models highlighted in green show good results, the models can be tuned to identify

Figure 41 - Tuned Models Evaluation Scores Comparison

Model Algorithm	Training Scores				Testing Scores				
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1	AUC
Tuned Artificial Neural Network	0.999	1.000	0.998	0.999	0.958	0.923	0.844	0.882	0.983
Tuned Random Forest	1.000	1.000	1.000	1.000	0.955	0.889	0.849	0.869	0.986
Tuned Decision Tree	0.999	0.998	0.999	0.999	0.914	0.846	0.704	0.768	0.888
Tuned K-Nearest Neighbours	1.000	1.000	1.000	1.000	0.943	0.944	0.771	0.849	0.979

On comparison of the 4 tuned models, we can see that the tuned artificial neural network model is the most optimal for customer churn prediction with its high score on training data and testing data.

The tuned knn model is also shows strong potential though it has a lower precision score despite the higher recall score. Overall, the tuned ANN model is the best all-rounder while the recall score isn't that much lower compared to tuned KNN model.

5.1 Best Model – Tuned Artificial Neural Network

An Artificial Neural Network is a machine learning model inspired by how the human brain processes information. It consists of layers of interconnected nodes that pass data through

weighted connections. Input data is transformed as it flows through the network, allowing the model to learn complex patterns.

The model is tuned using GridsearchCV by searching for the best combination of values. It works by testing different settings for hyperparameters like hidden layer sizes, maximum iterations, solver algorithm, tolerance etc. GridSearchCV performs cross-validation which ensures the selected parameters generalize well to unseen data while preventing overfitting.

The model is selected because it delivered the strongest results across all evaluation metrics, It Effectively balances Recall and Precision and it performs well even on unseen data after tuning. The tuned model shows improved scores overall especially the recall score compared to the base model.

Figure 42 - Tuned ANN Model Best Hyperparameters

Hyperparameter	Best Value	Description
activation	tanh	Activation function that introduces non-linearity
alpha	0.01	L2 regularization term that helps prevent overfitting.
hidden_layer_sizes	(100, 200, 150)	Number of neurons in each hidden layer; defines the ANN's architecture.
max_iter	1000	Maximum number of training iterations (epochs).
solver	adam	Optimization algorithm used for weight updates
tol	0.001	Tolerance for stopping criteria.

The above table shows the best hyperparameters that were found using GridSearchCV method, the model is tuned based on those hyperparameters

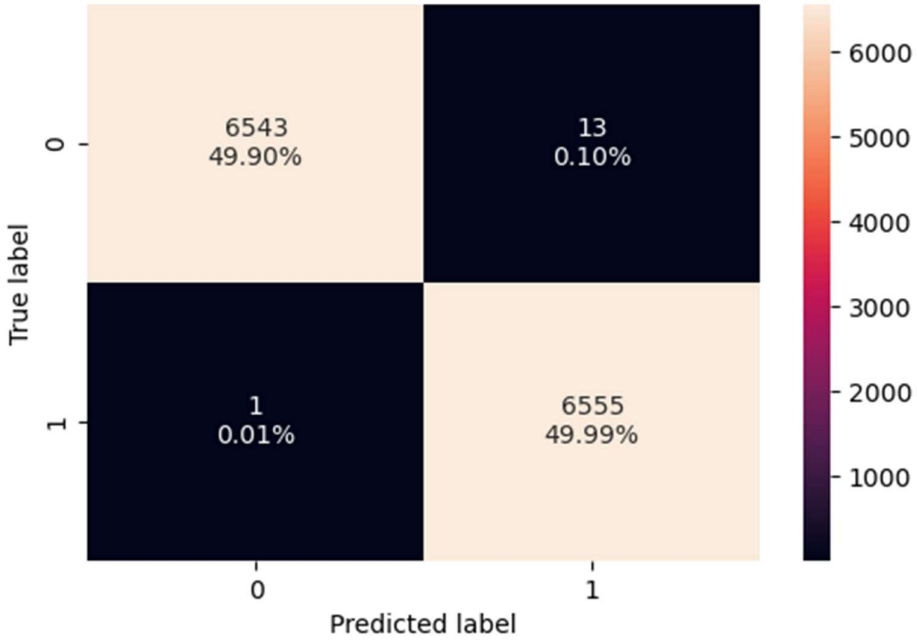
Figure 43 - Training Evaluation Scores - Base vs Tuned

TRAINING SCORES		
Metric	**Base Model**	**Tuned Model**
Accuracy	95.48%	99.89%
Recall	96.26%	99.98%
Precision	94.77%	99.80%
F1 Score	95.51%	99.89%

The above figure shows the training scores of the tuned ANN model compared to the base training set, The training scores of the model suggest it performs nearly flawlessly on the training data. With an accuracy of 0.999, it correctly classifies almost every instance, and a perfect recall of 1.000 means it doesn't miss any actual positive cases. Its precision of 0.998 shows it makes very few false positive predictions, and the F1 score of 0.999 reflects an good balance between precision and recall. While these results point to a highly effective model,

such near-perfect scores can also hint at possible overfitting, so it's important to compare them with test set performance to ensure the model generalizes well.

Figure 44 - Confusion Matrix - Training Set



The confusion matrix shows a classification model performing exceptionally well, with class 0 predicted correctly 49.90% of the time and only misclassified as class 1 in 0.10% of cases. Similarly, class 1 is correctly predicted 49.99% of the time with just 0.01% misclassified as class 0. This near-perfect distribution across the diagonal (true positives for both classes) indicates that the model is highly accurate and balanced in distinguishing between the two classes, with negligible error rates and no evident bias toward one class over the other.

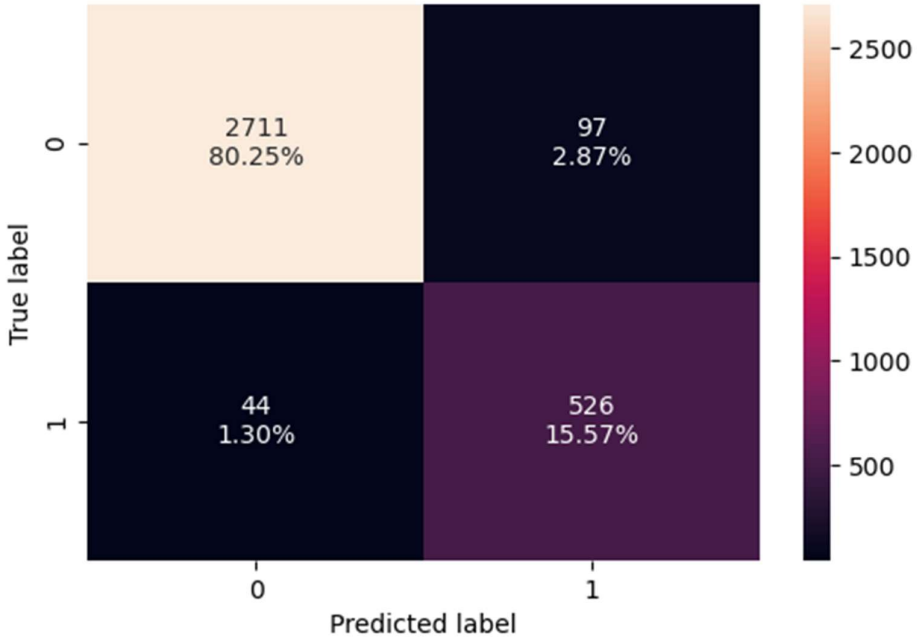
Figure 45 - Testing Evaluation Scores - Base vs Tuned

TESTING SCORES		
Metric	**Base Model**	**Tuned Model**
Accuracy	91.15%	95.83%
Recall	86.14%	92.28%
Precision	69.06%	84.43%
F1 Score	76.66%	88.18%
ROC AUC	0.959	0.983

The above table shows the comparison between the scores of the base and tuned model. The test scores for the tuned ANN model indicate strong generalization to unseen data. With an accuracy of 0.958, the model correctly predicts 95.8% of the test samples, and a recall of

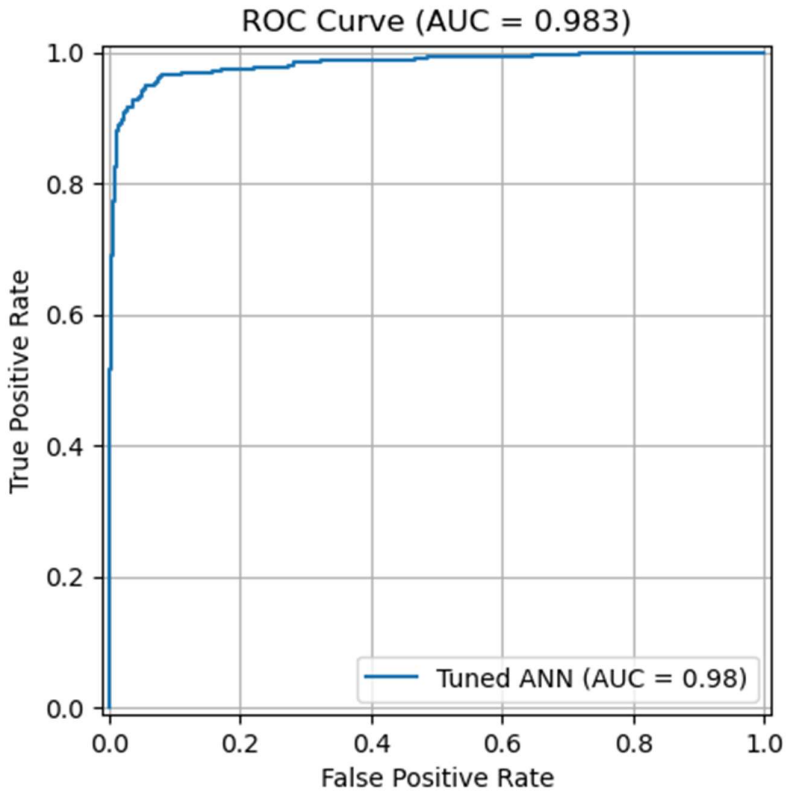
0.923 show it successfully captures most of the actual positives. Precision score of 0.844 implies there are some false positives, but still a high rate of correct positive predictions. The F1 score of 0.882 shows a solid balance between precision and recall. Lastly, the AUC score of 0.983 demonstrates excellent discriminatory power between classes, indicating the model maintains robust predictive performance beyond the training data.

Figure 46 - Confusion Matrix - Testing Set



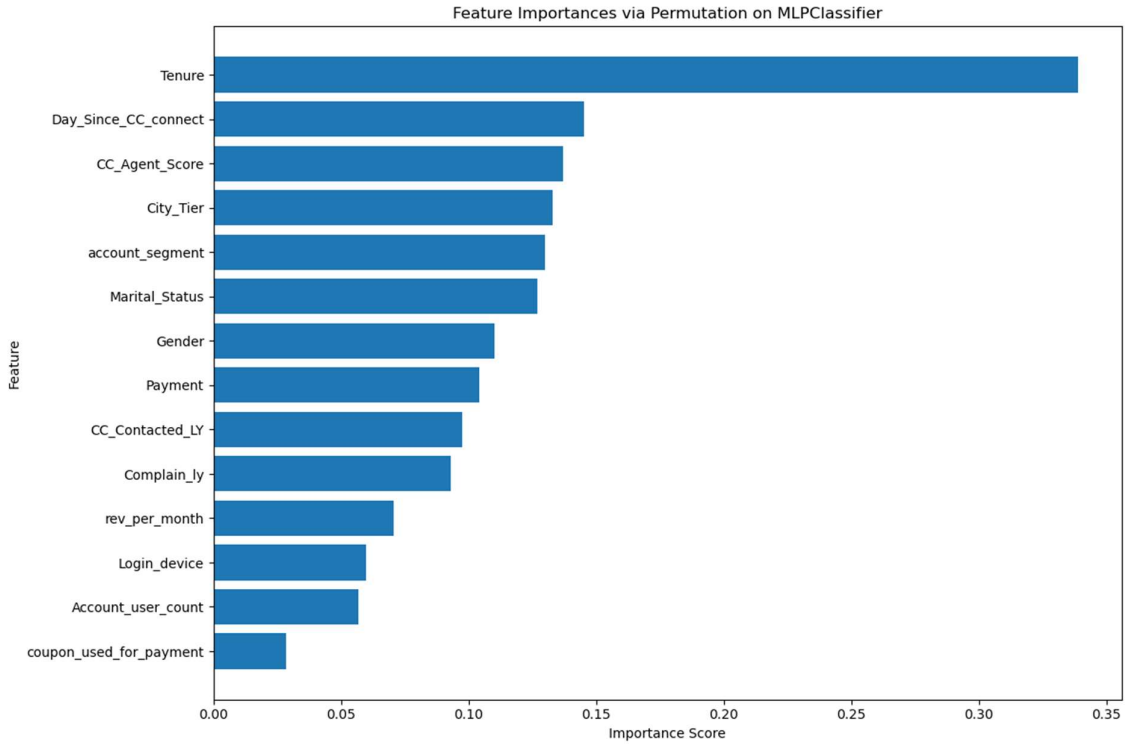
The confusion matrix shows how the model performs on the test data. It correctly classifies class 0 in 80.25% of cases and misclassifies it as class 1 in 2.87% of instances. Similarly, it correctly classifies class 1 in 15.57% of cases while misclassifying 1.30% of them as class 0. The model appears to favor class 0 slightly, given its higher accuracy there, but overall, still captures class 1 reasonably well. While there's a small imbalance, the overall misclassification rates remain low, indicating that the model is still performing effectively.

Figure 47 - Tuned Model ROC-AUC Graph



An AUC or Area Under the Curve score of 0.983 for the tuned model indicates excellent ability to distinguish between the two classes across all classification thresholds. It means that there's a 98.3% chance the model will correctly rank a randomly chosen positive instance higher than a randomly chosen negative one. This high AUC reflects strong overall performance, suggesting that the model is not only accurate but also robust in handling class imbalances or varying thresholds for decision-making.

Figure 48 - Tuned ANN Model Feature Importances



The permutation feature importance function is used on this model to obtain the feature importance scores of all the variables. On analysis of the feature importance of the tuned ANN model, we can see that tenure, days since customer care connect, agent score and city tier were the features which contributed to the most on building the random forest model. Tenure contributed to almost 30% of the model's performance implying tenure places a significant role on whether a customer will churn or not. Login device, coupon used for payment, account user count contributes to around 5% of the model's performance.

FINAL RECOMMENDATIONS

Based on the insights made during EDA and observations from model building, These are the final recommendations made to the business to reduce customer churning.

1. Strengthen Early Tenure Engagement

- Implement onboarding campaigns and tailored welcome journeys for new users in their first 60 days.
- Offer first-month perks or guided product tours to boost early satisfaction and reduce churn risk.

2. Proactively Use Support Interaction Data

- Set up automated alerts for repeated or low-rated support interactions to trigger retention workflows.
- Train frontline agents to identify early signs of dissatisfaction and escalate cases flagged by the churn model.

3. Upgrade Customer Support Experience

- Invest in agent training focused on empathy, resolution speed, and communication.
- Introduce post-interaction surveys and link agent KPIs to customer satisfaction scores.

4. Utilize Churn Prediction Model Output

- Integrate churn scores into CRM to prioritize outreach efforts for high-risk segments.
- Personalize interventions like offers or follow-up calls based on churn likelihood and customer profile.

5. Reevaluate and Refocus Retention Campaigns

- Analyze whether current cashback or coupon initiatives resonate with at-risk segments.
- Shift investment towards tactics targeting newly onboarded users and low-engagement customers.

6. Expand Market Presence Strategically

- Roll out geo-targeted campaigns and distribution in underpenetrated tier-2 cities.
- Partner with local influencers or digital payment platforms for regional adoption.

7. Design Segment-Specific Offers

- Develop tailored loyalty plans for married customers and high-value repeat buyers.
- Offer flexible subscriptions or bundles based on past spending behavior and product preferences.

8. Improve Satisfaction Measurement & Service Quality

- Launch a comprehensive customer expectations survey across touchpoints.
- Use results to reshape service workflows, reduce friction, and plug quality gaps.

APPENDIX

RAW CODE

```
[ ]: # importing the necessary libararies
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
```

```
[ ]: # loading the dataset
df = pd.read_excel('C:\\Users\\Ishaan Shakti\\Documents\\Great Lakes\\Capstone Project\\Project Work\\Customer Churn Data.xlsx', sheet_name = 'Data for D
df.head()
```

```
[ ]: df.tail()
```

```
[ ]: df.shape
```

```
[ ]: df.info()
```

```
[ ]: # viewing the statistical summary
df.describe().T
```

```
[ ]: df.isnull().sum()
```

```
[ ]: df.duplicated().sum()
```

UNIVARIATE ANALYSIS

```
[ ]: # function to plot a boxplot and a histogram along the same scale.

def histogram_boxplot(data, feature, figsize=(12, 7), kde=False, bins=None):
    """
    Boxplot and histogram combined

    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (12,7))
    kde: whether to the show density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(
        nrows=2,  # Number of rows of the subplot grid= 2
        sharex=True,  # x-axis will be shared among all subplots
        gridspec_kw={"height_ratios": (0.25, 0.75)},
        figsize=figsize,
    )  # creating the 2 subplots
    sns.boxplot(
        data=data, x=feature, ax=ax_box2, showmeans=True, color="violet"
    )  # boxplot will be created and a star will indicate the mean value of the column
    sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2, bins=bins, palette="winter"
    ) if bins else sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2
    )  # For histogram
    ax_hist2.axvline(
        data[feature].mean(), color="green", linestyle="--"
    )  # Add mean to the histogram
    ax_hist2.axvline(
        data[feature].median(), color="black", linestyle="-"
    )  # Add median to the histogram
```

[ ]:
```
# function to create labeled barplots


def labeled_barplot(data, feature, perc=False, n=None):
    """
    Barplot with percentage at the top

    data: dataframe
    feature: dataframe column
    perc: whether to display percentages instead of count (default is False)
    n: displays the top n category levels (default is None, i.e., display all levels)
    """

    total = len(data[feature])  # length of the column
    count = data[feature].nunique()
    if n is None:
        plt.figure(figsize=(count + 1, 5))
    else:
        plt.figure(figsize=(n + 1, 5))

    plt.xticks(rotation=90, fontsize=15)
    ax = sns.countplot(
        data=data,
        x=feature,
        palette="Paired",
        order=data[feature].value_counts().index[:n].sort_values(),
    )

    for p in ax.patches:
        if perc == True:
            label = "{:.1f}%".format(
                100 * p.get_height() / total
            )  # percentage of each class of the category
        else:
            label = p.get_height()  # count of each level of the category

        x = p.get_x() + p.get_width() / 2  # width of the plot
        y = p.get_height()  # height of the plot

        ax.annotate(
            label,
            (x, y),
            ha="center",
            va="center",
            size=12,
            xytext=(0, 5),
            textcoords="offset points",
        )  # annotate the percentage

    plt.show()  # show the plot
```

[ ]:
```
# function to plot stacked bar chart

def stacked_barplot(data, predictor, target):
    """
    Print the category counts and plot a stacked bar chart

    data: dataframe
    predictor: independent variable
    target: target variable
    """
    count = data[predictor].nunique()
    sorter = data[target].value_counts().index[-1]
    tab1 = pd.crosstab(data[predictor], data[target], margins=True).sort_values(
        by=sorter, ascending=False
    )
    print(tab1)
    print("-" * 120)
    tab = pd.crosstab(data[predictor], data[target], normalize="index").sort_values(
        by=sorter, ascending=False
    )
    tab.plot(kind="bar", stacked=True, figsize=(count + 1, 5))
    plt.legend(
        loc="lower left", frameon=False,
    )
    plt.legend(loc="upper left", bbox_to_anchor=(1, 1))
    plt.show()
```

```
def distribution_plot_wrt_target(data, predictor, target):

    fig, axs = plt.subplots(2, 2, figsize=(12, 10))

    target_uniq = data[target].unique()

    axs[0, 0].set_title("Distribution of target for target=" + str(target_uniq[0]))
    sns.histplot(
        data=data[data[target] == target_uniq[0]],
        x=predictor,
        kde=True,
        ax=axs[0, 0],
        color="teal",
    )

    axs[0, 1].set_title("Distribution of target for target=" + str(target_uniq[1]))
    sns.histplot(
        data=data[data[target] == target_uniq[1]],
        x=predictor,
        kde=True,
        ax=axs[0, 1],
        color="orange",
    )

    axs[1, 0].set_title("Boxplot w.r.t target")
    sns.boxplot(data=data, x=target, y=predictor, ax=axs[1, 0], palette="gist_rainbow")

    axs[1, 1].set_title("Boxplot (without outliers) w.r.t target")
    sns.boxplot(
        data=data,
        x=target,
        y=predictor,
        ax=axs[1, 1],
        showfliers=False,
        palette="gist_rainbow",
    )

    plt.tight_layout()
    plt.show()
```

[ ]:
```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Churn')
plt.title('Count Plot of Churned Customers', fontsize=14)
plt.xlabel('Churn')
plt.show()
```

[ ]:
```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'City_Tier')
plt.title('Count Plot of City Tier', fontsize=14)
plt.xlabel('City Tier')
plt.show()
```

[ ]:
```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Payment')
plt.title('Count Plot of Payment Method', fontsize=14)
plt.xlabel('Payment Method')
plt.show()
```

[ ]:
```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Gender')
plt.title('Count Plot of Gender', fontsize=14)
plt.xlabel('Gender')
plt.show()
```

[ ]:
```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Account_user_count')
plt.title('Count Plot of Account_user_count', fontsize=14)
plt.xlabel('Account_user_count')
plt.show()
```

[ ]:
```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Account_user_count')
plt.title('Count Plot of Account Users', fontsize=14)
plt.xlabel('Account User Count')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'account_segment')
plt.title('Count Plot of Account Segment', fontsize=14)
plt.xlabel('Account Segment')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'CC_Agent_Score')
plt.title('Count Plot of CC Agent Score', fontsize=14)
plt.xlabel('CC Agent Score')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Marital_Status')
plt.title('Count Plot of Marital Status', fontsize=14)
plt.xlabel('Marital Status')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Complain_ly')
plt.title('Count Plot of Complains in Last 12 Months', fontsize=14)
plt.xlabel('Customer Complains in Last 12 Months')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Login_device')
plt.title('Count Plot of Login Devices', fontsize=14)
plt.xlabel('Login Devices')
plt.show()
```

```
histogram_boxplot(df, 'Tenure')
```

BIVARIATE ANALYSIS

```
cols_list = df.select_dtypes(include=np.number).columns.tolist()
plt.figure(figsize=(15, 7))
sns.heatmap(df[cols_list].corr(), annot=True, vmin=-1, vmax=1, fmt=".3f", cmap="Spectral")
plt.title('Heat Map', fontsize=14)
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'City_Tier',hue = 'Churn')
plt.title('Count Plot of City Tier VS Churned Customers', fontsize=14)
plt.xlabel('Ciy Tier')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Payment',hue = 'Churn')
plt.title('Count Plot of Payment Method VS Churned Customers', fontsize=14)
plt.xlabel('Payment Method')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Gender',hue = 'Churn')
plt.title('Count Plot of Gender VS Churned Customers', fontsize=14)
plt.xlabel('Gender')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Service_Score',hue = 'Churn')
plt.title('Count Plot of Service Score VS Churned Customers', fontsize=14)
plt.xlabel('Service Score')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Account_user_count',hue = 'Churn')
plt.title('Count Plot of Account User Count VS Churned Customers', fontsize=14)
plt.xlabel('Account User Count')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'account_segment',hue = 'Churn')
plt.title('Count Plot of Account Segment VS Churned Customers', fontsize=14)
plt.xlabel('Account Segment')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'CC_Agent_Score',hue = 'Churn')
plt.title('Count Plot of CC Agent Score VS Churned Customers', fontsize=14)
plt.xlabel('CC Agent Score')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Marital_Status',hue = 'Churn')
plt.title('Count Plot of Marital Status VS Churned Customers', fontsize=14)
plt.xlabel('Marital Status')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Complain_ly',hue = 'Churn')
plt.title('Count Plot of Complains Last Year VS Churned Customers', fontsize=14)
plt.xlabel('Complains in Last Year')
plt.show()
```

```
plt.figure(figsize = (12, 7))
sns.countplot(data = df,x = 'Login_device',hue = 'Churn')
plt.title('Count Plot of Login Device VS Churned Customers', fontsize=14)
plt.xlabel('Login Device')
plt.show()
```

```
distribution_plot_wrt_target(df,'Tenure','Churn')
```

```
distribution_plot_wrt_target(df,'CC_Contacted_LY','Churn')
```

REMOVAL OF UNWANTED VARIABLES

```
# Function to check all values in dataset
for column in df.columns:
    if df[column].dtype == 'object':
        print(column.upper(),': ',df[column].nunique())
        print(df[column].value_counts().sort_values())
        print('\n')
```

```
# removing unwanted variables in Tenure
df['Tenure'] = df['Tenure'].replace('#',np.NaN)
df['Tenure'] = df['Tenure'].astype('Int64')
df["Tenure"].unique()
```

```
# replace null values with median
df['Tenure'] = df['Tenure'].fillna(df['Tenure'].median())
```

```
# removing unwanted variables in Gender
df['Gender'] = df['Gender'].replace('F','Female')
df['Gender'] = df['Gender'].replace('M','Male')
```

```
df["Gender"].unique()
```

```
# removing unwanted variables in Account User Count
df['Account_user_count'] = df['Account_user_count'].replace('@',np.NaN)
df['Account_user_count'] = df['Account_user_count'].astype('float64')
```

```
df["Account_user_count"].unique()
```

```
# removing unwanted variables in Rev Per Month
df['rev_per_month'] = df['rev_per_month'].replace('+',np.NaN)
df['rev_per_month'] = df['rev_per_month'].astype('float64')
df["rev_per_month"].unique()
```

```
# removing unwanted variables in Rev Growth yoy'
df['rev_growth_yoy'] = df['rev_growth_yoy'].replace('$',np.NaN)
df['rev_growth_yoy'] = df['rev_growth_yoy'].astype('float64')
df["rev_growth_yoy"].unique()
```

```
# removing unwanted variables in Coupon Used For Payment
df['coupon_used_for_payment'] = df['coupon_used_for_payment'].replace('#',np.NaN)
df['coupon_used_for_payment'] = df['coupon_used_for_payment'].replace('$',np.NaN)
df['coupon_used_for_payment'] = df['coupon_used_for_payment'].replace('*',np.NaN)
df['coupon_used_for_payment'] = df['coupon_used_for_payment'].astype('float64')
df["coupon_used_for_payment"].unique()
```

```
# removing unwanted variables in Day Since CC Connect
df['Day_Since_CC_connect'] = df['Day_Since_CC_connect'].replace('$',np.NaN)
df['Day_Since_CC_connect'] = df['Day_Since_CC_connect'].astype('float64')
df["Day_Since_CC_connect"].unique()
```

```
# removing unwanted variables in Cashback
df['cashback'] = df['cashback'].replace('$',np.NaN)
df['cashback'] = df['cashback'].astype('float64')
df["cashback"].unique
```

```
# removing unwanted variables in Login Device
df['Login_device'] = df['Login_device'].replace('&&&&',np.NaN)
df["Login_device"].unique()
```

MISSING VALUE TREATMENT

```
df.isnull().sum()
```

```
# treatment in City Tier
df['City_Tier'] = df['City_Tier'].fillna(df['City_Tier'].mode()[0])
```

```
# treatment in CC_Contacted_LY
df['CC_Contacted_LY'] = df['CC_Contacted_LY'].fillna(df['CC_Contacted_LY'].median())
```

```
# treatment in Payment
df['Payment'] = df['Payment'].fillna(df['Payment'].mode()[0])
```

```
# treatment in Gender
df['Gender'] = df['Gender'].fillna(df['Gender'].mode()[0])
```

```
# treatment in Service_Score
df['Service_Score'] = df['Service_Score'].fillna(df['Service_Score'].mode()[0])
```

```
# treatment in Account_user_count
df['Account_user_count'] = df['Account_user_count'].fillna(df['Account_user_count'].median())
```

```
# treatment in account_segment
df['account_segment'] = df['account_segment'].fillna(df['account_segment'].mode()[0])
```

```
# treatment in CC_Agent_Score
df['CC_Agent_Score'] = df['CC_Agent_Score'].fillna(df['CC_Agent_Score'].mode()[0])
```

```
# treatment in Marital_Status
df['Marital_Status'] = df['Marital_Status'].fillna(df['Marital_Status'].mode()[0])
```

```
# treatment in rev_per_month
df['rev_per_month'] = df['rev_per_month'].fillna(df['rev_per_month'].median())
```

```
# treatment in Complain_ly
df['Complain_ly'] = df['Complain_ly'].fillna(df['Complain_ly'].mode()[0])
```

```
# treatment in rev_growth_yoy
df['rev_growth_yoy'] = df['rev_growth_yoy'].fillna(df['rev_growth_yoy'].median())
```

```
# treatment in coupon_used_for_payment
df['coupon_used_for_payment'] = df['coupon_used_for_payment'].fillna(df['coupon_used_for_payment'].median())
```

```
# treatment in Day_Since_CC_connect
df['Day_Since_CC_connect'] = df['Day_Since_CC_connect'].fillna(df['Day_Since_CC_connect'].median())
```

```
# treatment in cashback
df['cashback'] = df['cashback'].fillna(df['cashback'].median())
```

```
# treatment in Login_device
df['Login_device'] = df['Login_device'].fillna(df['Login_device'].mode()[0])
```

```
df.isnull().sum()
```

OUTLIER TREATMENT

```
# outlier detection using boxplot
numeric_columns = df.select_dtypes(include=np.number).columns.tolist()

plt.figure(figsize=(15, 12))

for i, variable in enumerate(numeric_columns):
    plt.subplot(4, 4, i + 1)
    plt.boxplot(df[variable], whis=1.5)
    plt.tight_layout()
    plt.title(variable)

plt.show()
```

```
# treating outliers
def outlier_treatment(col):
    sorted(col)
    Q1,Q3 = col.quantile([0.25,0.75])
    IQR = Q3-Q1
    lower_range = Q1 - (1.5 * IQR)
    upper_range = Q3 + (1.5 * IQR)
    return lower_range, upper_range
```

```
# list of columns to apply the outlier treatment
columns_to_process = [
    'Tenure', 'CC_Contacted_LY', 'Account_user_count', 'cashback', 'rev_per_month',
    'Day_Since_CC_connect', 'coupon_used_for_payment', 'rev_growth_yoy'
]

for col in columns_to_process:
    lw, up = outlier_treatment(df[col])
    df[col] = np.clip(df[col], lw, up)
```

```
[ ]: # check outliers post treatment
numeric_columns = df.select_dtypes(include=np.number).columns.tolist()

plt.figure(figsize=(15, 12))

for i, variable in enumerate(numeric_columns):
    plt.subplot(4, 4, i + 1)
    plt.boxplot(df[variable], whis=1.5)
    plt.tight_layout()
    plt.title(variable)

plt.show()
```

VARIABLE TRANSFORMATION

```
[ ]: # encoding payment variable
payment_mapping = {
    'Debit Card': 1,
    'UPI': 2,
    'Credit Card': 3,
    'Cash on Delivery': 4,
    'E wallet': 5
}

df['Payment'] = df['Payment'].replace(payment_mapping)
```

```
[ ]: # encoding gender variable
gender_mapping = {
    'Female': 1, 'F': 1,
    'Male': 2, 'M': 2
}

df['Gender'] = df['Gender'].replace(gender_mapping)
```

```
[ ]: # encoding account segment variable
account_segment_mapping = {
    'Super': 1,
    'Regular Plus': 2, 'Regular +': 2,
    'Regular': 3,
    'HNI': 4,
    'Super Plus': 5, 'Super +': 5
}

df['account_segment'] = df['account_segment'].replace(account_segment_mapping)
```

```
[ ]: # encoding marital status variable
marital_status_mapping = {
    'Single': 1,
    'Divorced': 2,
    'Married': 3
}

df['Marital_Status'] = df['Marital_Status'].replace(marital_status_mapping)
```

```
[ ]: # encoding login device variable
login_device_mapping = {
    'Mobile': 1,
    'Computer': 2
}

df['Login_device'] = df['Login_device'].replace(login_device_mapping)
```

```
[ ]: df = df.drop(['AccountID'], axis=1)
```

MODEL BUILDING

```
# Create X and Y for modeling
X = df.drop(['Churn'], axis=1)
y = df['Churn']
```

```
# Splitting data into train and test data set
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
```

```
X_train = X_train.astype(int)
y_train = y_train.astype(int)
X_train.info()
```

SMOTE

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
```

DATA SCALING

```
# Initialize the MinMaxScaler
scaler = MinMaxScaler()
# Apply scaling
X_train_scaled = scaler.fit_transform(X_train_smote)
X_test_scaled = scaler.transform(X_test)
```

VIF

```
#Checking for multicollinearity using VIF
from statsmodels.stats.outliers_influence import variance_inflation_factor
X_train_scaled_df = pd.DataFrame(X_train_scaled, columns=X.columns)
vif = pd.DataFrame()
vif["Feature"] = X_train_scaled_df.columns
vif["VIF"] = [variance_inflation_factor(X_train_scaled_df.values, i)
              for i in range(X_train_scaled_df.shape[1])]

print(vif)
```

```
X_train_scaled_df = X_train_scaled_df.drop(['Service_Score'], axis=1)
vif = pd.DataFrame()
vif["Feature"] = X_train_scaled_df.columns
vif["VIF"] = [variance_inflation_factor(X_train_scaled_df.values, i)
              for i in range(X_train_scaled_df.shape[1])]

print(vif)
```

```
X_train_scaled_df = X_train_scaled_df.drop(['cashback'], axis=1)
vif = pd.DataFrame()
vif["Feature"] = X_train_scaled_df.columns
vif["VIF"] = [variance_inflation_factor(X_train_scaled_df.values, i)
              for i in range(X_train_scaled_df.shape[1])]

print(vif)
```

```
X_train_scaled_df = X_train_scaled_df.drop(['rev_growth_yoy'], axis=1)
vif = pd.DataFrame()
vif["Feature"] = X_train_scaled_df.columns
vif["VIF"] = [variance_inflation_factor(X_train_scaled_df.values, i)
              for i in range(X_train_scaled_df.shape[1])]

print(vif)
```

```
X_train_scaled_df = pd.DataFrame(X_train_scaled, columns=X.columns)
X_test_scaled_df = pd.DataFrame(X_test_scaled, columns=X.columns)

drop_cols = ['Service_Score', 'cashback', 'rev_growth_yoy']
X_train_final = X_train_scaled_df.drop(drop_cols, axis=1)
X_test_final = X_test_scaled_df.drop(drop_cols, axis=1)
```

```
# defining a function to compute different metrics to check performance of a classification model built using statsmodels
def model_performance_classification_statsmodels(
    model, predictors, target, threshold=0.5
):
    """
    Function to compute different metrics to check classification model performance

    model: classifier
    predictors: independent variables
    target: dependent variable
    threshold: threshold for classifying the observation as class 1
    """

    # checking which probabilities are greater than threshold
    pred_temp = model.predict(predictors) > threshold
    # rounding off the above values to get classes
    pred = np.round(pred_temp)

    acc = accuracy_score(target, pred)  # to compute Accuracy
    recall = recall_score(target, pred)  # to compute Recall
    precision = precision_score(target, pred)  # to compute Precision
    f1 = f1_score(target, pred)  # to compute F1-score

    # creating a dataframe of metrics
    df_perf = pd.DataFrame(
        {"Accuracy": acc, "Recall": recall, "Precision": precision, "F1": f1,},
        index=[0],
    )

    return df_perf
```

```
def confusion_matrix_statsmodels(model, predictors, target, threshold=0.5):
    """
    To plot the confusion_matrix with percentages

    model: classifier
    predictors: independent variables
    target: dependent variable
    threshold: threshold for classifying the observation as class 1
    """
    y_pred = model.predict(predictors) > threshold
    cm = confusion_matrix(target, y_pred)
    labels = np.asarray(
        [
            ["{0:0.0f}".format(item) + "\n{0:.2%}".format(item / cm.flatten().sum())]
            for item in cm.flatten()
        ]
    ).reshape(2, 2)

    plt.figure(figsize=(6, 4))
    sns.heatmap(cm, annot=labels, fmt="")
    plt.ylabel("True label")
    plt.xlabel("Predicted label")
```

LOGISTIC REGRESSION MODEL

```
# To get diferent metric scores
from sklearn.metrics import (
    f1_score,
    accuracy_score,
    recall_score,
    precision_score,
    confusion_matrix,
    roc_auc_score,
    precision_recall_curve,
    roc_curve,
    make_scorer,
)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

lg_model = LogisticRegression(random_state=1, max_iter=1000)
lg_model.fit(X_train_final, y_train_smote)
```

```
confusion_matrix_statsmodels(lg_model, X_train_final, y_train_smote)
```

```
print("Training performance model 1:")
logistic_reg_perf_train = model_performance_classification_statsmodels(lg_model, X_train_final, y_train_smote)
logistic_reg_perf_train
```

```
confusion_matrix_statsmodels(lg_model, X_test_final, y_test)
```

```
print("Testing performance model 1:")
logistic_reg_perf_test=model_performance_classification_statsmodels(lg_model, X_test_final, y_test)
logistic_reg_perf_test
```

```
[ ]: from sklearn.metrics import roc_curve, auc, RocCurveDisplay

# Get probability estimates for the positive class
y_proba = lg_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Logistic Regression').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid(True)
plt.show()
```

DECISION TREE MODEL

```
[ ]: from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier(random_state=1)
dt_model.fit(X_train_final, y_train_smote)
```

```
[ ]: decision_tree_perf_train = model_performance_classification_statsmodels(
    dt_model, X_train_final, y_train_smote
)
decision_tree_perf_train
```

```
[ ]: confusion_matrix_statsmodels(dt_model, X_train_final, y_train_smote)
```

```
[ ]: decision_tree_perf_test = model_performance_classification_statsmodels(
    dt_model, X_test_final, y_test
)
decision_tree_perf_test
```

```
[ ]: confusion_matrix_statsmodels(dt_model, X_test_final, y_test)
```

```
[ ]: from sklearn.metrics import roc_curve, auc, RocCurveDisplay

# Get probability estimates for the positive class
y_proba = dt_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Decision Tree').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid(True)
plt.show()
```

LINEAR DISCRIMINANT ANALYSIS MODEL

```
[ ]: from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda_model = LinearDiscriminantAnalysis()
lda_model = lda_model.fit(X_train_final, y_train_smote)
```

```
[ ]: linear_discriminant_perf_train = model_performance_classification_statsmodels(
    lda_model, X_train_final, y_train_smote
)
linear_discriminant_perf_train
```

```
[ ]: confusion_matrix_statsmodels(lda_model, X_train_final, y_train_smote)
```

```
[ ]: linear_discriminant_perf_test = model_performance_classification_statsmodels(
    lda_model, X_test_final, y_test
)
linear_discriminant_perf_test
```

```
[ ]: confusion_matrix_statsmodels(lda_model, X_test_final, y_test)
```

```
[ ]:

# Get probability estimates for the positive class
y_proba = lda_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Linear Discriminant Analysis').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid(True)
plt.show()
```

ARTIFICAL NEURAL NETWORK

```
from sklearn.neural_network import MLPClassifier
ann_model = MLPClassifier(random_state=1, max_iter=1000)
ann_model.fit(X_train_final, y_train_smote)
```

```
artifical_neural_perf_train = model_performance_classification_statsmodels(
    ann_model, X_train_final, y_train_smote
)
artifical_neural_perf_train
```

```
confusion_matrix_statsmodels(ann_model, X_train_final, y_train_smote)
```

```
artifical_neural_perf_test = model_performance_classification_statsmodels(
    ann_model, X_test_final, y_test
)
artifical_neural_perf_test
```

```
confusion_matrix_statsmodels(ann_model, X_test_final, y_test)
```

```
# Get probability estimates for the positive class
y_proba = ann_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Artificial Neural Network').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid(True)
plt.show()
```

SUPPORT VECTOR MACHINE

```
from sklearn import svm
svm_model = svm.SVC(random_state=1, probability=True)
svm_model.fit(X_train_final, y_train_smote)
```

```
support_vector_perf_train = model_performance_classification_statsmodels(
    svm_model, X_train_final, y_train_smote
)
support_vector_perf_train
```

```
confusion_matrix_statsmodels(svm_model, X_train_final, y_train_smote)
```

```
support_vector_perf_test = model_performance_classification_statsmodels(
    svm_model, X_test_final, y_test
)
support_vector_perf_test
```

```
confusion_matrix_statsmodels(svm_model, X_test_final, y_test)
```

```
# Get probability estimates for the positive class
y_proba = svm_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Support Vector Machine').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid(True)
plt.show()
```

K-NEAREST NEIGHBOUR MODEL

```
from sklearn.neighbors import KNeighborsClassifier
knn_model=KNeighborsClassifier()
knn_model.fit(X_train_final, y_train_smote)
```

```
kn_neighbour_perf_train = model_performance_classification_statsmodels(
    knn_model, X_train_final, y_train_smote
)
kn_neighbour_perf_train
```

```
confusion_matrix_statsmodels(knn_model, X_train_final, y_train_smote)
```

```
kn_neighbour_perf_test = model_performance_classification_statsmodels(
    knn_model, X_test_final, y_test
)
kn_neighbour_perf_test
```

```
confusion_matrix_statsmodels(knn_model, X_test_final, y_test)
```

```
# Get probability estimates for the positive class
y_proba = knn_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='K Nearest Neighbour').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid(True)
plt.show()
```

ENSEMBLE MODELS

RANDOM FOREST MODEL

```
from sklearn.ensemble import RandomForestClassifier
rf_model = RandomForestClassifier(n_estimators=100, random_state=1)
rf_model.fit(X_train_final, y_train_smote)
```

```
random_forest_perf_train = model_performance_classification_statsmodels(
    rf_model, X_train_final, y_train_smote
)
random_forest_perf_train
```

```
confusion_matrix_statsmodels(rf_model, X_train_final, y_train_smote)
```

```
random_forest_perf_test = model_performance_classification_statsmodels(
    rf_model, X_test_final, y_test
)
random_forest_perf_test
```

```
confusion_matrix_statsmodels(rf_model, X_test_final, y_test)
```

```
# Get probability estimates for the positive class
y_proba = rf_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Random Forest').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid(True)
plt.show()
```

GRADIENT BOOSTING MODEL

```
[ ]: from sklearn.ensemble import GradientBoostingClassifier
gbm_model = GradientBoostingClassifier(n_estimators=100, random_state=1)
gbm_model.fit(X_train_final, y_train_smote)
```

```
[ ]: gradient_boosting_perf_train = model_performance_classification_statsmodels(
    gbm_model, X_train_final, y_train_smote
)
gradient_boosting_perf_train
```

```
[ ]: confusion_matrix_statsmodels(gbm_model, X_train_final, y_train_smote)
```

```
[ ]: gradient_boosting_perf_test = model_performance_classification_statsmodels(
    gbm_model, X_test_final, y_test
)
gradient_boosting_perf_test
```

```
[ ]: confusion_matrix_statsmodels(gbm_model, X_test_final, y_test)
```

```
[ ]: # Get probability estimates for the positive class
y_proba = gbm_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Gradient Boosting').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid(True)
plt.show()
```

ADABOOSTING MODEL

```
[ ]: from sklearn.ensemble import AdaBoostClassifier
ada_model = AdaBoostClassifier(n_estimators=51, random_state=1)
adamodel = ada_model.fit(X_train_final, y_train_smote)
```

```
[ ]: ada_boosting_perf_train = model_performance_classification_statsmodels(
    ada_model, X_train_final, y_train_smote
)
ada_boosting_perf_train
```

```
[ ]: confusion_matrix_statsmodels(ada_model, X_train_final, y_train_smote)
```

```
[ ]: ada_boosting_perf_test = model_performance_classification_statsmodels(
    ada_model, X_test_final, y_test
)
ada_boosting_perf_test
```

```
[ ]: confusion_matrix_statsmodels(ada_model, X_test_final, y_test)
```

```
[ ]: # Get probability estimates for the positive class
y_proba = ada_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='AdaBoosting').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid(True)
plt.show()
```

MODEL COMPARISON

```
[ ]: # training performance comparison

models_train_comp_df = pd.concat(
    [logistic_reg_perf_train.T,
        decision_tree_perf_train.T,
        linear_discriminant_perf_train.T,
        artifical_neural_perf_train.T,
        support_vector_perf_train.T,
        kn_neighbour_perf_train.T,
        random_forest_perf_train.T,
        gradient_boosting_perf_train.T,
        ada_boosting_perf_train.T
    ],
    axis=1,
)
models_train_comp_df.columns = ["Logistic Regression",
    "Decision Tree",
    "Linear Discriminant Analysis",
    "Artifical Neural Network",
    "Support Vector Machine",
    "K-Nearest Neighbors",
    "Random Forest",
    "Gradient Boosting",
    "Adaboosting",
]
print("Training performance comparison:")
models_train_comp_df
```

```
[ ]: # testing performance comparison

models_test_comp_df = pd.concat(
    [logistic_reg_perf_test.T,
        decision_tree_perf_test.T,
        linear_discriminant_perf_test.T,
        artifical_neural_perf_test.T,
        support_vector_perf_test.T,
        kn_neighbour_perf_test.T,
        random_forest_perf_test.T,
        gradient_boosting_perf_test.T,
        ada_boosting_perf_test.T
    ],
    axis=1,
)
models_test_comp_df.columns = ["Logistic Regression",
    "Decision Tree",
    "Linear Discriminant Analysis",
    "Artifical Neural Network",
    "Support Vector Machine",
    "K-Nearest Neighbors",
    "Random Forest",
    "Gradient Boosting",
    "Adaboosting",
]
print("Testing performance comparison:")
models_test_comp_df
```

MODEL TUNING

ANN MODEL TUNING

```
[ ]: from sklearn.model_selection import GridSearchCV
param_grid = {
    'hidden_layer_sizes': [(80,), (90,), (100,), (100, 200, 150)],  # Different hidden layer configurations
    'max_iter': [1000, 2000],  # Maximum number of iterations
    'solver': ['sgd', 'adam', 'lbfgs'],  # Optimization algorithms
    'tol': [0.01, 0.001],  # Tolerance for stopping criterion
    'activation': ['identity', 'logistic', 'tanh', 'relu'],  # Activation functions
    'alpha': [0.0001, 0.001, 0.01]  # L2 regularization strength
}
# Initialize the MLPClassifier
mlp = MLPClassifier(random_state=1)

# Perform Grid Search with 5-fold cross-validation
grid_search = GridSearchCV(estimator=mlp, param_grid=param_grid, cv=5, scoring='f1', verbose=2, n_jobs=-1)
grid_search.fit(X_train_final, y_train_smote)

# Print the best parameters found
print("Best Parameters:", grid_search.best_params_)
```

```
[ ]: tuned_ann_model = MLPClassifier(**grid_search.best_params_, random_state=1,)

tuned_ann_model.fit(X_train_final, y_train_smote)
```

```
[ ]: tuned_ann_perf_train = model_performance_classification_statsmodels(
    tuned_ann_model, X_train_final, y_train_smote
)
tuned_ann_perf_train
```

```
[ ]: confusion_matrix_statsmodels(tuned_ann_model, X_train_final, y_train_smote)
```

```
tuned_ann_perf_test = model_performance_classification_statsmodels(
    tuned_ann_model, X_test_final, y_test
)
tuned_ann_perf_test
```

```
confusion_matrix_statsmodels(tuned_ann_model, X_test_final, y_test)
```

```
from sklearn.metrics import roc_curve, auc, RocCurveDisplay
import matplotlib.pyplot as plt

# Get probability estimates for the positive class
y_proba = tuned_ann_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Tuned ANN').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid()
plt.show();
```

RANDOM FOREST MODEL

```
param_grid = {
    'n_estimators':      [200, 500, 800],
    'max_depth':         [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf':  [1, 2, 4],
    'max_features':      ['sqrt', 'log2'],
    'class_weight':      ['balanced', 'balanced_subsample', {0:1, 1:5}],
    'criterion':         ['gini', 'entropy']
}
# Initialize the Random Forest classifier
rf_model = RandomForestClassifier(random_state=42)
# Perform Grid Search with 5-fold cross-validation
grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid, cv=5, scoring='f1', verbose=2, n_jobs=-1)
grid_search.fit(X_train_final, y_train_smote)  # Fit the model on training data
# Print the best parameters found
print("Best Parameters:", grid_search.best_params_)
```

```
# Train final model using the best parameters
tuned_rf_model = RandomForestClassifier(**grid_search.best_params_, random_state=42)
tuned_rf_model.fit(X_train_final, y_train_smote)
```

```
tuned_rf_perf_train = model_performance_classification_statsmodels(
    tuned_rf_model, X_train_final, y_train_smote
)
tuned_rf_perf_train
```

```
confusion_matrix_statsmodels(tuned_rf_model, X_train_final, y_train_smote)
```

```
tuned_rf_perf_test = model_performance_classification_statsmodels(
    tuned_rf_model, X_test_final, y_test
)
tuned_rf_perf_test
```

```
confusion_matrix_statsmodels(tuned_rf_model, X_test_final, y_test)
```

```
# Get probability estimates for the positive class
y_proba = tuned_rf_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Tuned Random Forest').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid()
plt.show();
```

DECISION TREE MODEL

```
param_grid = {
    'max_depth': [3, 5, 10, 20, 30],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4, 6],
    'criterion': ['gini', 'entropy']
}

dt = DecisionTreeClassifier(random_state=42)
grid_search = GridSearchCV(estimator=dt, param_grid=param_grid, cv=5, scoring='f1')
grid_search.fit(X_train_final, y_train_smote)

print("Best params:", grid_search.best_params_)
```

```
# Train final model using the best parameters
tuned_dt_model = DecisionTreeClassifier(**grid_search.best_params_, random_state=1)
tuned_dt_model.fit(X_train_final, y_train_smote)
```

```
tuned_dt_perf_train = model_performance_classification_statsmodels(
    tuned_dt_model, X_train_final, y_train_smote
)
tuned_dt_perf_train
```

```
confusion_matrix_statsmodels(tuned_dt_model, X_train_final, y_train_smote)
```

```
tuned_dt_perf_test = model_performance_classification_statsmodels(
    tuned_dt_model, X_test_final, y_test
)
tuned_dt_perf_test
```

```
confusion_matrix_statsmodels(tuned_dt_model, X_test_final, y_test)
```

```
# Get probability estimates for the positive class
y_proba = tuned_dt_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Tuned Decision Tree').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid()
plt.show()
```

K-NEAREST NEIGHBOUR MODEL

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([
    ('knn', KNeighborsClassifier())
])
param_grid = {
  'knn__n_neighbors': [3,5,7,9,11,15],
  'knn__weights': ['uniform','distance'],
  'knn__p': [1,2],
  'knn__leaf_size': [20,30,40],
  'knn__metric': ['minkowski','manhattan','chebyshev']
}

grid = GridSearchCV(
    estimator=pipe,
    param_grid=param_grid,
    cv=5,
    scoring='f1',     # focus on catching all churners
    n_jobs=-1,
    verbose=2
)

grid.fit(X_train_final, y_train_smote)
print("Best KNN params:", grid.best_params_)
```

```
# Train final model using the best parameters
best_knn_params = {
    'n_neighbors': 3,
    'weights': 'distance',
    'p': 1,
    'metric': 'minkowski',
    'leaf_size': 30
}
tuned_knn_model = KNeighborsClassifier(**best_knn_params)
tuned_knn_model.fit(X_train_final, y_train_smote)
```

```
tuned_knn_perf_train = model_performance_classification_statsmodels(
    tuned_knn_model, X_train_final, y_train_smote
)
tuned_knn_perf_train
```

```
confusion_matrix_statsmodels(tuned_knn_model, X_train_final, y_train_smote)
```

```
tuned_knn_perf_test = model_performance_classification_statsmodels(
    tuned_knn_model, X_test_final, y_test
)
tuned_knn_perf_test
```

```
confusion_matrix_statsmodels(tuned_knn_model, X_test_final, y_test)
```

```
# Get probability estimates for the positive class
y_proba = tuned_knn_model.predict_proba(X_test_final)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc, estimator_name='Tuned KNN').plot()
plt.title(f"ROC Curve (AUC = {roc_auc:.3f})")
plt.grid()
plt.show()
```

TUNED MODEL COMPARISON

```
# training performance comparison

models_train_comp_df1 = pd.concat(
    [tuned_ann_perf_train.T,
        tuned_rf_perf_train.T,
        tuned_dt_perf_train.T,
        tuned_knn_perf_train.T
    ],
    axis=1,
)
models_train_comp_df1.columns = ["Tuned Artificial Neural Network",
    "Tuned Random Forest",
    "Tuned Decision Tree",
    "Tuned K-Nearest Neighbours"
]
print("Training performance comparison:")
models_train_comp_df1
```

```
# testing performance comparison

models_test_comp_df1 = pd.concat(
    [tuned_ann_perf_test.T,
        tuned_rf_perf_test.T,
        tuned_dt_perf_test.T,
        tuned_knn_perf_test.T
    ],
    axis=1,
)
models_test_comp_df1.columns = ["Tuned Artificial Neural Network",
    "Tuned Random Forest",
    "Tuned Decision Tree",
    "Tuned K-Nearest Neighbours"
]
print("Testing performance comparison:")
models_test_comp_df1
```

```
from sklearn.inspection import permutation_importance
result = permutation_importance(tuned_ann_model, X_train_final, y_train_smote,
                                scoring='recall', n_repeats=10, random_state=42, n_jobs=-1)

# Extract feature importances
importances = result.importances_mean
feature_names = X_train_final.columns

# Sort in descending order
sorted_indices = np.argsort(importances)[::-1]
sorted_importances = importances[sorted_indices]
sorted_feature_names = feature_names[sorted_indices]

# Reverse bar positions to put most important at top
plt.figure(figsize=(12, 8))
plt.title("Feature Importances via Permutation on MLPClassifier")
plt.barh(range(len(sorted_importances)), sorted_importances[::-1], align="center")
plt.yticks(range(len(sorted_feature_names)), sorted_feature_names[::-1])
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.tight_layout()
plt.show()
```