PGP-DSBA PROJECT REPORT

CAPSTONE PROJECT NOTES - 2

CUSTOMER CHURN PREDICTION

BY

ISHAAN SHAKTI JAYARAMAN

PGPDSBA.O.JULY24.A

TABLE OF CONTENTS

LIST OF FIGURES

MODEL BUILDING AND INTERPRETATION

1.1 Introduction to model building

Machine Learning model building involves selecting relevant data, preprocessing it, and choosing an appropriate algorithm to make predictions or classifications. The model is trained using labeled data, adjusting parameters to minimize errors and improve accuracy. Once optimized, the model is tested on unseen data to ensure it generalizes well before deployment.

The model needs to be built based on the dataset (Customer Churn Data.xlsx). In this business case, the need is to predict whether a given customer would churn or not. This is a binary classification problem with only two prediction outcomes '0' meaning the customer will not churn and '1' meaning the customer will churn.

The dataset has been prepared in the following steps to make it ready for model building.

- EDA was performed to explore the different dynamics of all the variables.
- Incorrect values such as integer values being stored as text values were all replaced with null values.
- All the null values were treated with either the median or mode of variable it is in.
- All outliers present in the numerical variables were treated.
- All categorical variables were encoded into numerical variables to make it easier to perform model building.
- The dataset was scaled using MinMax scaler.
- The dataset was split into training and testing data in the ration of 70:30.

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.

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.

1.2 Logistic Regression Model

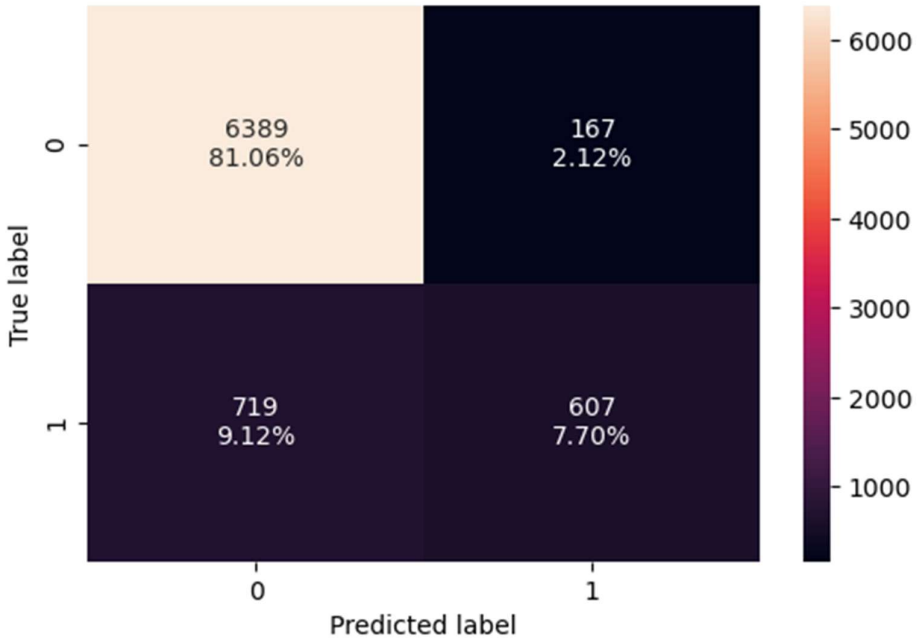
The Logistic Regression from scikit-learn is used to predict customer churn. Logistic Regression works by estimating probabilities of class membership and applying a threshold to classify observations. The model is built from the sklearn library in python.

Figure 1 - Logistic Regression Model - Performance Scores - Training Data

Training performance model 1:

	Accuracy	Recall	Precision	F1
0	0.887592	0.457768	0.784238	0.578095

Figure 2 - Confusion Matrix - Logistic Regression Model - Training Data



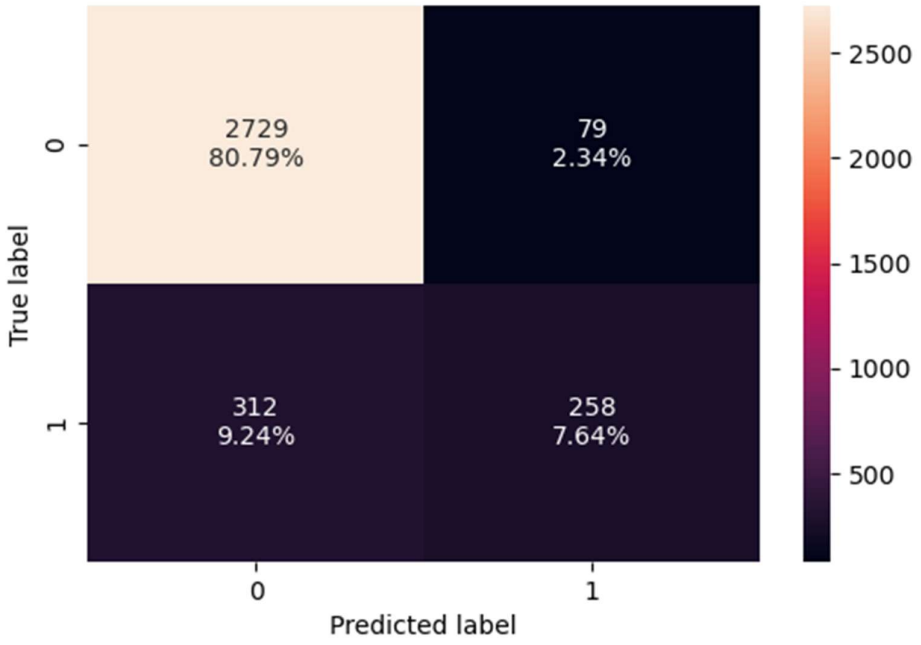
Accuracy score of 0.887 indicates good general performance. Recall score of 0.458 indicates low recall means many actual churners are missed. Precision score of 0.784 indicates relatively high precision which means few false positives. F1-Score of 0.578 is low indicating average balance between recall and precision.

Figure 3 - Logistic Regression Model - Performance Scores - Testing Data

Testing performance model 1:

	Accuracy	Recall	Precision	F1
0	0.884251	0.452632	0.765579	0.568908

Figure 4 - Confusion Matrix - Logistic Regression Model - Testing Data



Accuracy score of 0.884 which is similar to training, indicating good generalization. Recall score of 0.453 shows that the model is missing many actual churners. Precision score of 0.766 shows the model is more precise but struggles with recall. F1-Score of 0.569 is relatively low.

The model performs well in accuracy and precision but the low recall suggests it does not identify churners effectively.

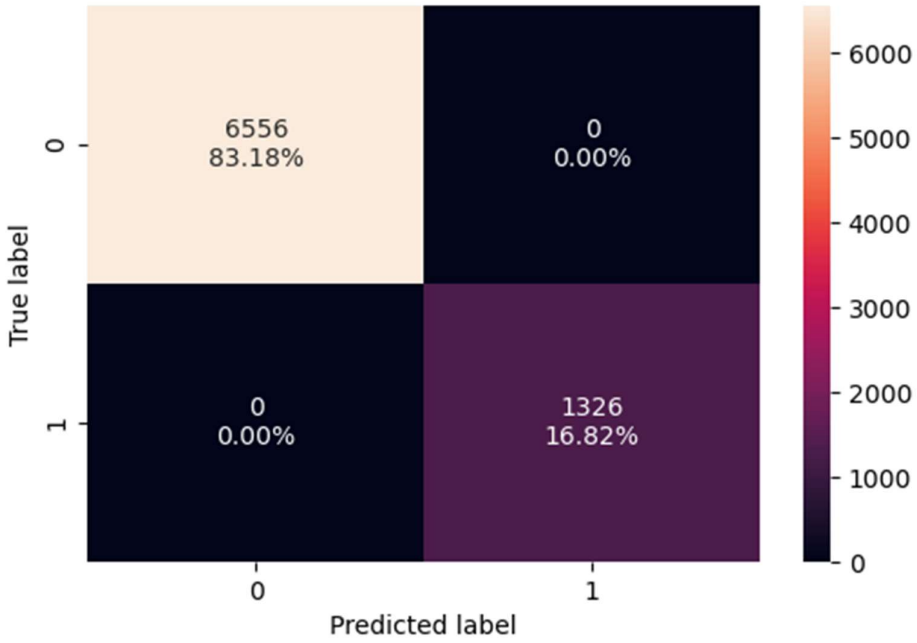
1.3 Decision Tree Model

A Decision Tree is a classification model that makes predictions by splitting data into branches based on feature values, ultimately leading to a decision at each leaf node. It mimics human decision-making by asking yes/no questions until a final classification is reached. The model is built from the sklearn library in python.

Figure 5 - Decision Tree Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	1.0	1.0	1.0	1.0

Figure 6 - Confusion Matrix - Decision Tree Model - Training Data

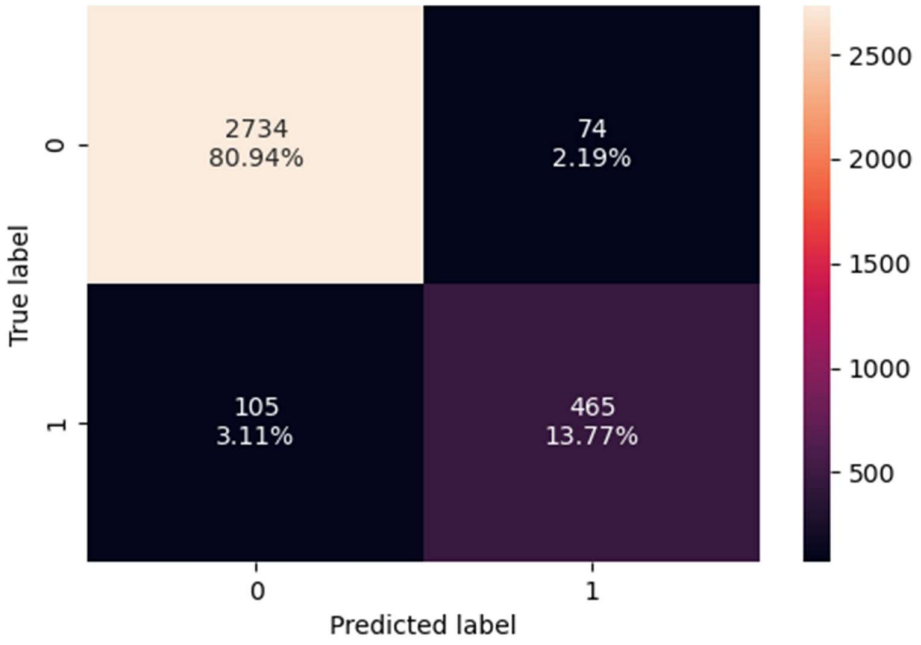


The decision tree model has achieved perfect scores across all metrics—accuracy, recall, precision, and F1-score on the training dataset. While this might seem ideal, it could be a sign of overfitting.

Figure 7 - Decision Tree Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.94701	0.815789	0.862709	0.838593

Figure 8 - Confusion Matrix - Decision Tree Model - Testing Data



On the testing dataset, the decision tree model performs well but not perfectly. The accuracy drops to 0.947, meaning it correctly predicts about 95% of test samples. However, recall decreases to 0.816, indicating some of actual churners were missed. Precision remains good at 0.863. The F1-score of 0.839 suggests that while the model is somewhat effective at predicting churn. The difference between the training scores and testing scores shows that the model may be too complex leading to loss of predictive power on test data.

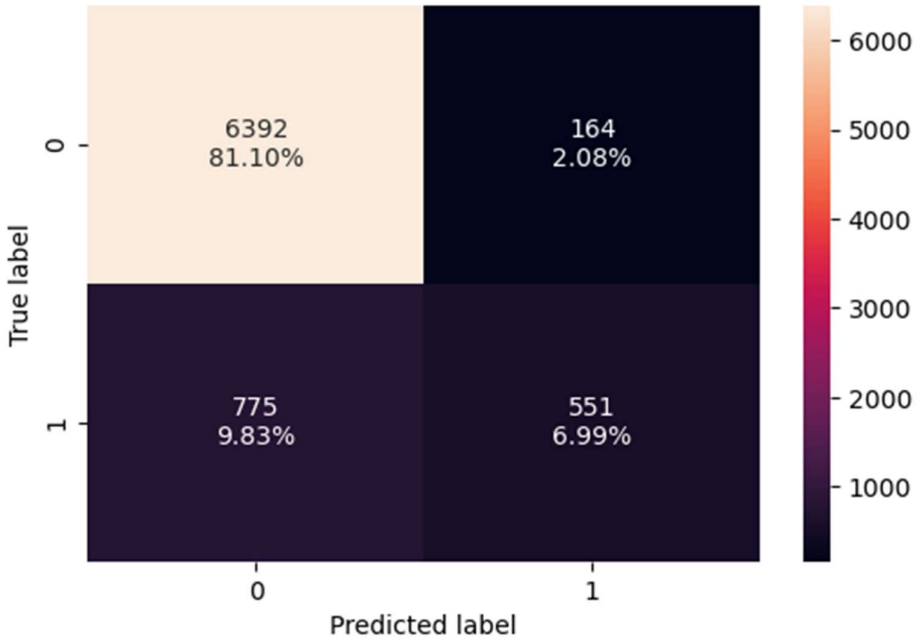
1.4 Linear Discriminant Analysis Model

Linear Discriminant Analysis is a classification technique that finds the best linear boundary to separate different classes by maximizing variance between them. It reduces dimensionality by projecting data onto a new axis while preserving class separability. The model is built from the sklearn library in python.

Figure 9 - Linear Discriminant Analysis Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	0.880868	0.415535	0.770629	0.539931

Figure 10 - Confusion Matrix - Linear Discriminant Analysis Model - Training Data

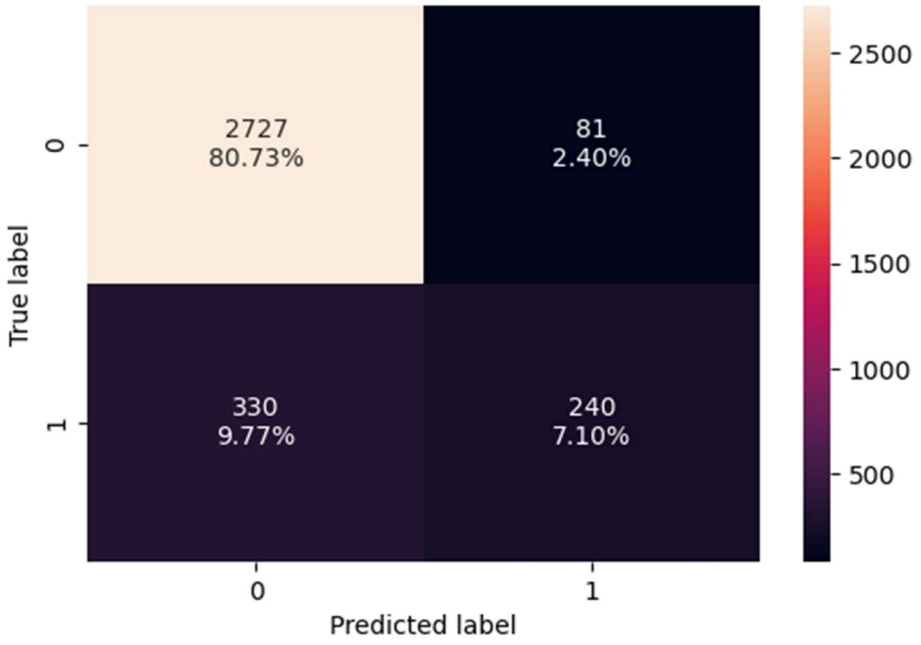


The model achieved an accuracy of 0.881, indicating strong overall performance in classifying churn and non-churn customers. However, its recall score suggests it struggles to identify a significant portion of actual churners, meaning some customers at risk of leaving were not correctly flagged. The precision score shows that when the model does predict churn, it is often correct, minimizing false alarms. The F1-score of 0.539 shows average balance between recall and precision.

Figure 11 - Linear Discriminant Analysis Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.87833	0.421053	0.747664	0.538721

Figure 12 - Confusion Matrix - Linear Discriminant Analysis Model - Testing Data



The model achieved 0.878 accuracy, effectively classifying churn cases. However, its recall score of 0.42 indicates it missed many actual churners, while precision score suggests that when it does predict churn, it can be accurate. The F1-score shows that while the model is useful improvements in recall need to be done which could enhance its ability to detect churners more reliably.

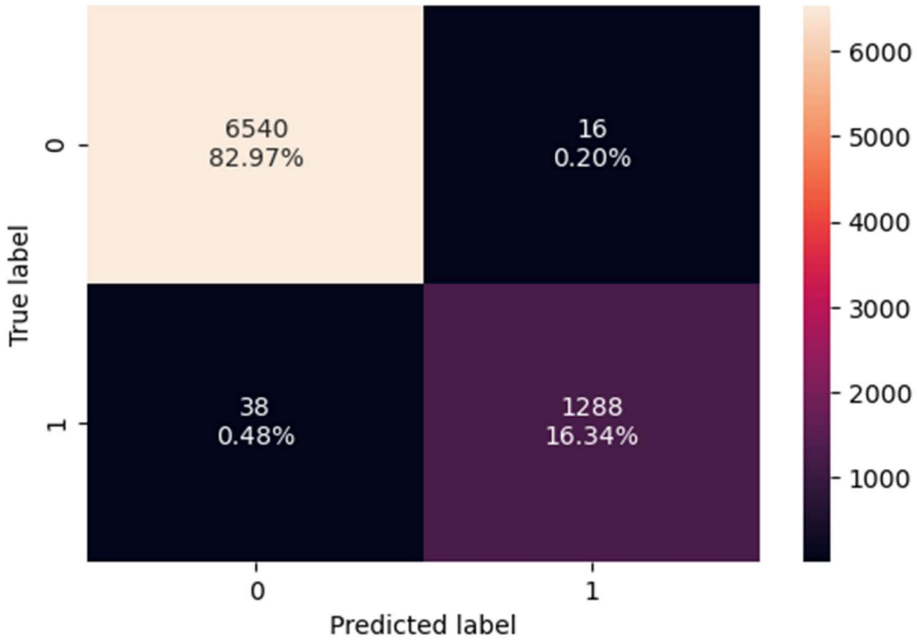
1.5 Artificial Neural Network Model

An Artificial Neural Network is a machine learning model designed to imitate the way the human brain processes information. It consists of layers of interconnected neurons that learn patterns from data, making it highly effective for complex problems like this customer churn prediction. ANN models are good for both classification and regression problems. The model is built from the sklearn library in python.

Figure 13 - Artificial Neural Network Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	0.993149	0.971342	0.98773	0.979468

Figure 14 - Confusion Matrix - Artificial Neural Network Model - Training Data

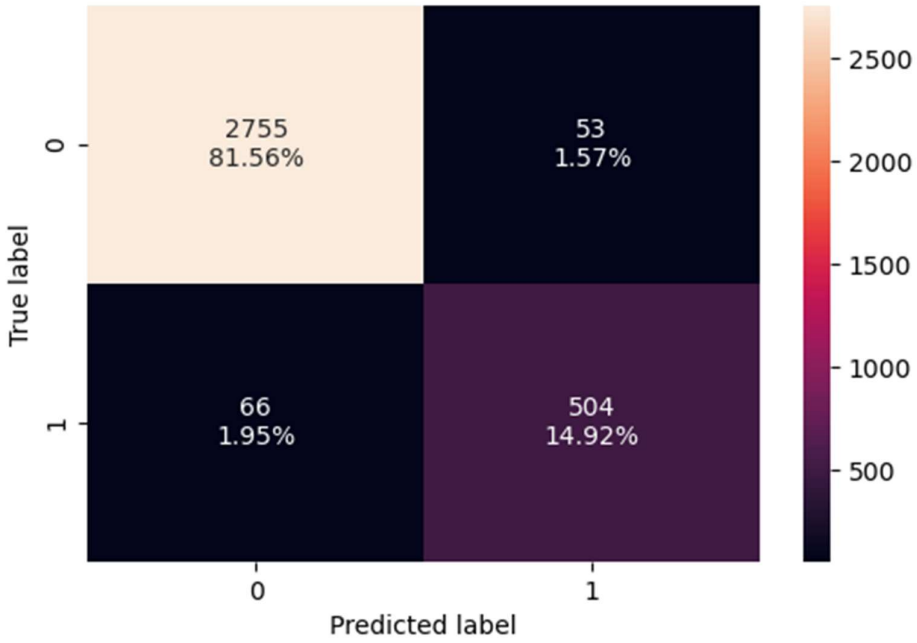


The model performance on the training data shows an accuracy score of 0.993, meaning it correctly classifies nearly all instances. Recall score of 0.971 indicates that it successfully identifies most actual churners, minimizing false negatives. Precision score of 0.987 ensures that when the model predicts a customer will churn, it is correct most of the time, reducing false positives. The F1-score of 0.979, which balances recall and precision shows that the model is highly effective at churn prediction. Overall, the model has performed well on the training data.

Figure 15 - Artificial Neural Network Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.964772	0.884211	0.904847	0.89441

Figure 16 - Confusion Matrix - Artificial Neural Network Model - Testing Data



The model performs well on the test data as well achieving an accuracy of 0.964, meaning it correctly classifies most cases. Its recall score of 0.884 indicates that it captures a large portion of actual churners, reducing the number of missed cases. Precision remains high at 0.905, showing that when the model predicts churn, it is usually correct. The F1-score of 0.894 shows that the model maintains a strong balance between identifying churners and ensuring accuracy in predictions. Overall, the model performs well on the test data and can be considered for use with tuning.

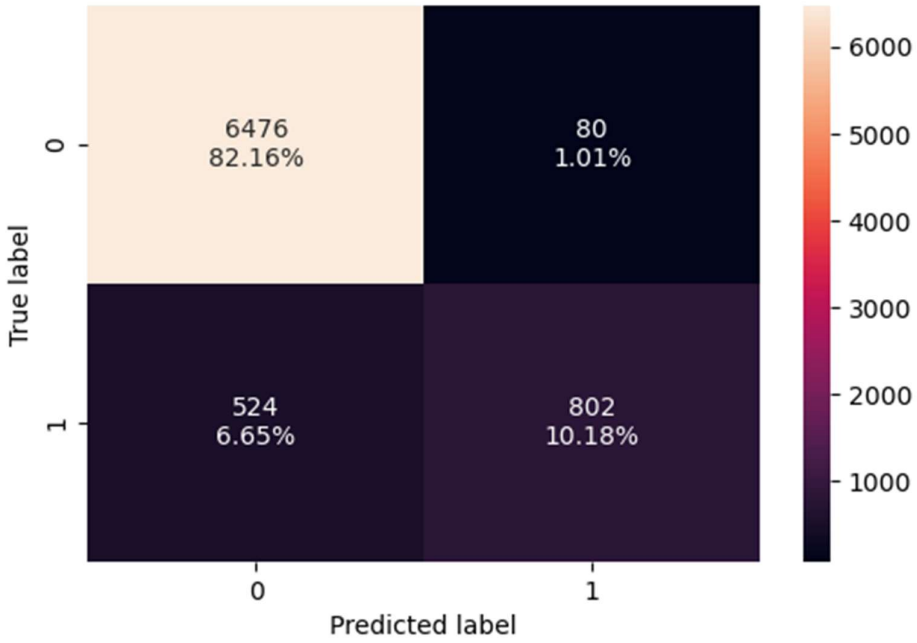
1.6 Support Vector Machine Model

A Support Vector Machine is a classification model that finds the optimal boundary between classes by maximizing the margin between them. It works well for both linear and non-linear data using kernels to transform features into higher-dimensional spaces. The model is built from the sklearn library in python.

Figure 17 - Support Vector Machine Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	0.92337	0.604827	0.909297	0.726449

Figure 18 - Confusion Matrix - Support Vector Machine Model - Training Data

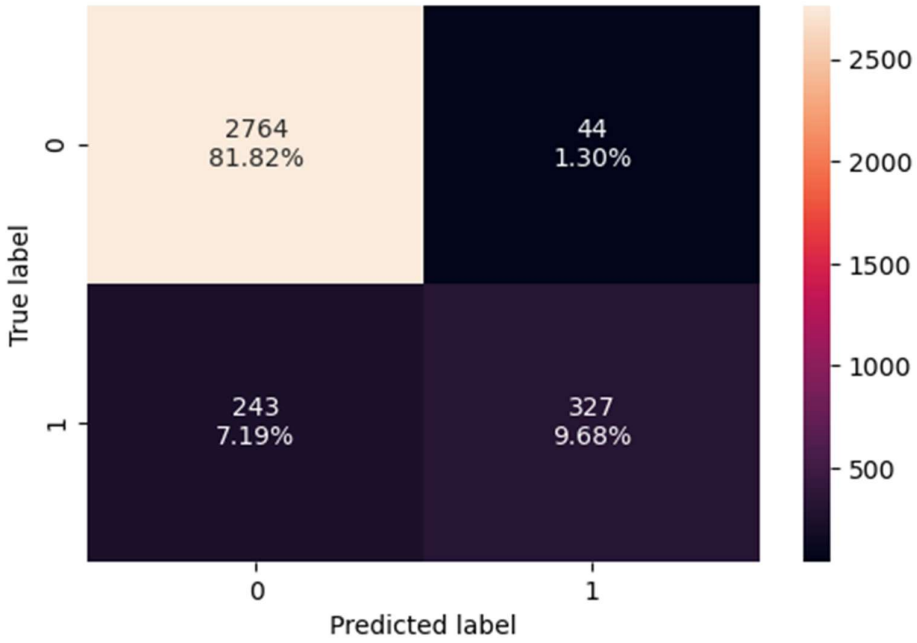


The model's performance on the training data shows an accuracy of 0.923, indicating it correctly classifies most cases. Its recall score of 0.605 percent suggests that while it identifies a good portion of actual churners, some may still be missed. Precision is high at 0.909, meaning that when the model predicts churn, it is usually correct with few false positives. The F1-score of 0.726 shows good balance recall and precision showing that the model is effective but not optimal.

Figure 19 - Support Vector Machine Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.915038	0.573684	0.881402	0.695005

Figure 20 - Confusion Matrix - Support Vector Machine Model - Testing Data



The model achieves an accuracy of 0.915 on the test data, correctly classifying most cases. Its recall of 0.573 indicates that it identifies a fair portion of actual churners, though some are still missed. Precision remains strong at 0.881, meaning that when the model predicts churn it is usually correct with few false positives. The F1-score of 0.695 shows an average balance between recall and precision, making the model somewhat usable but not optimal.

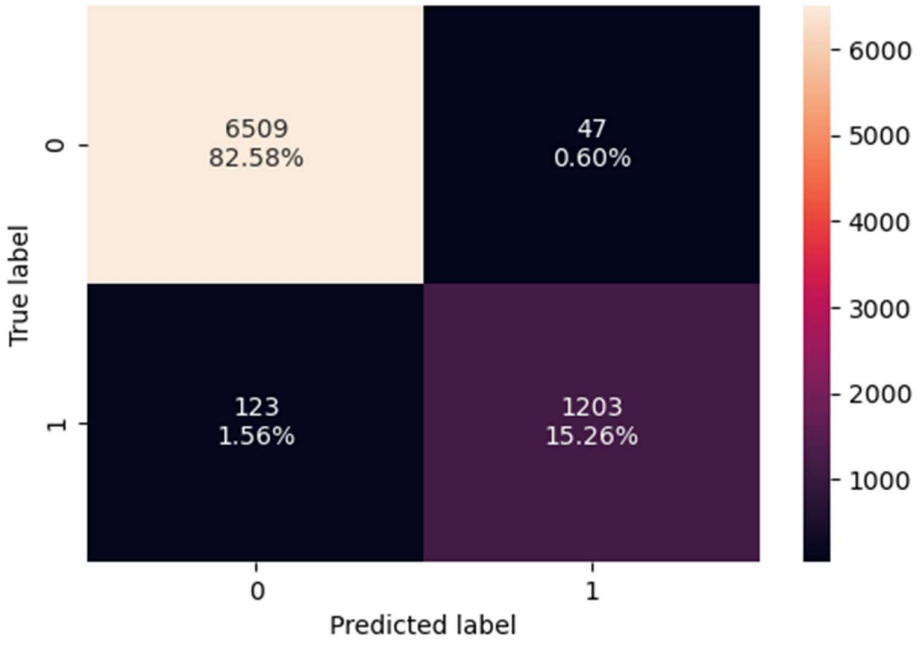
1.7 K-Nearest Neighbors Model

The K-Nearest Neighbors model is a classification algorithm that predicts the class of a data point based on the majority class of its closest neighbors. It works by calculating distances between points and assigning a label based on the most frequent category among K nearest neighbors. The model is built from the sklearn library in python.

Figure 21 - K-Nearest Neighbors Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	0.978432	0.90724	0.9624	0.934006

Figure 22 - Confusion Matrix - K-Nearest Neighbors Model - Training Data

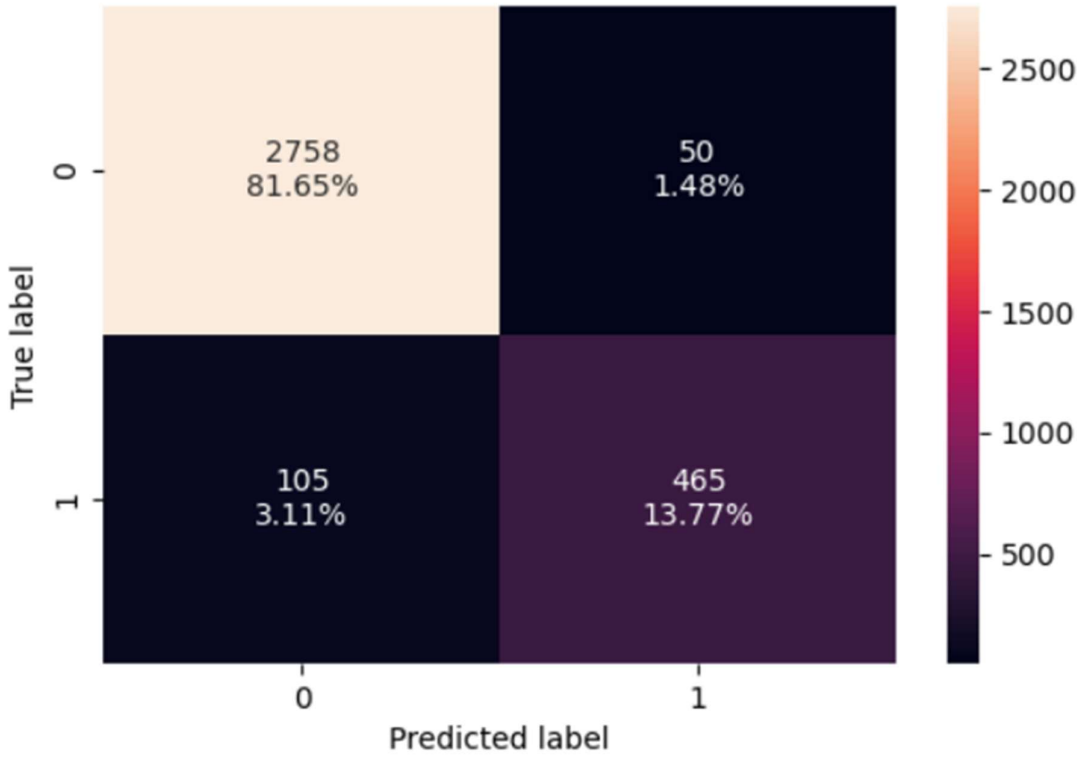


The model shows an accuracy of 0.978, indicating that it correctly classifies most cases. The high recall of 0.907 suggests that it successfully identifies a large portion of actual churners, minimizing missed cases. Precision at 0.962 shows that when the model predicts churn, it is highly reliable, making very few incorrect predictions. The F1-score of 0.934 shows very good balance between recall and precision. Overall, the model performs well on the training data.

Figure 23 - K-Nearest Neighbors Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.954115	0.815789	0.902913	0.857143

Figure 24 - Confusion Matrix - K-Nearest Neighbors Model - Testing Data



The model performance on the test data shows an accuracy of 0.954, indicating strong overall classification ability. Its recall of 0.816 shows that it correctly identifies most churners, though a small portion is missed. Precision remains high at 0.903, meaning that when it predicts churn, it is mostly accurate. The F1-score of 0.857 confirms a good balance between recall and precision. Overall, the model performs well on test data as well and can be considered for tuning to optimize the results.

MODEL TUNING & BUSINESS IMPLICATION

2.1 Ensemble Modeling

Ensemble modeling is a technique that combines multiple machine learning models to improve predictive accuracy and stability. It works by aggregating different models' strengths, reducing errors and enhancing generalization to unseen data.

The ensemble models which will be built for customer churn prediction problem are

- Random Forest Model
- Gradient Boosting Model

- Adaboosting Model

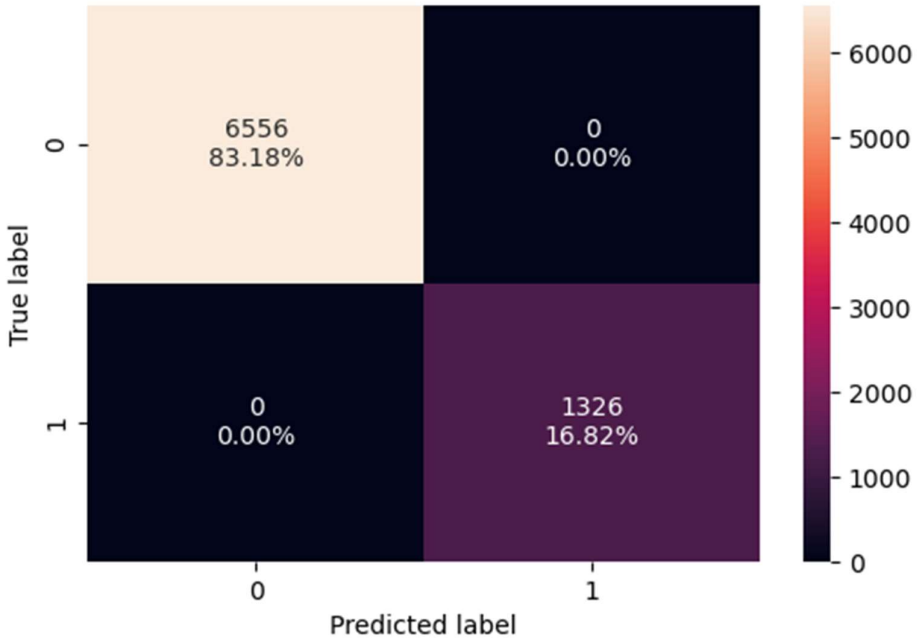
2.1.1 Random Forest Model

Random Forest is an ensemble learning method that builds multiple decision trees and combines their outputs to improve accuracy and stability. It reduces overfitting by averaging predictions, making it more robust than a single decision tree. The model is built from the sklearn library in python.

Figure 25 - Random Forest Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	1.0	1.0	1.0	1.0

Figure 26 - Confusion Matrix - Random Forest Model - Training Data

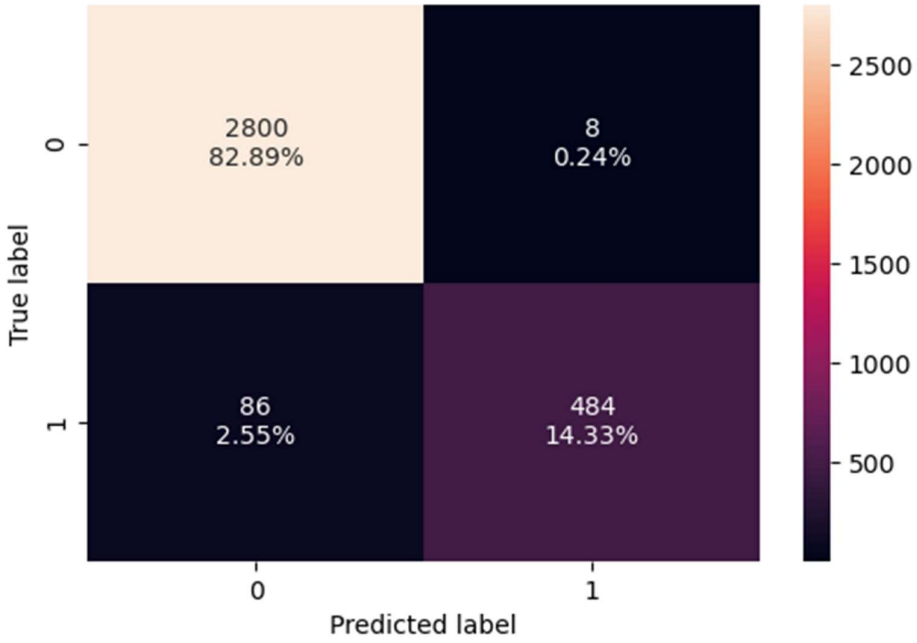


The model performs perfectly on the training data, correctly classifying churners and non-churners in the dataset. There could be overfitting which will be identified when it is performed on the test data.

Figure 27 - Random Forest Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.972173	0.849123	0.98374	0.911488

Figure 28 - Confusion Matrix - Random Forest Model - Testing Data



The model achieves an accuracy of 0.972, meaning it correctly classifies most cases. Its recall of 0.849 shows that it successfully identifies a high proportion of actual churners, reducing missed predictions. Precision is strong at 0.984, ensuring that when the model predicts churn, it is almost always correct. The F1-score of 0.911 reflects a well-balanced performance between identifying churners and maintaining accuracy in predictions. Overall, the model performs well on the test data and can be considered optimal to use with some tuning.

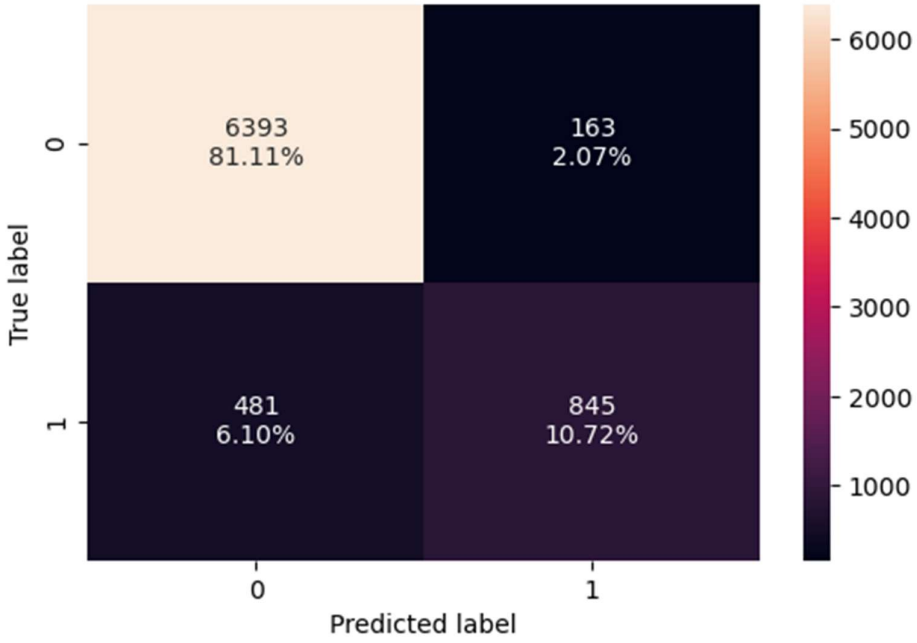
2.1.2 Gradient Boosting Model

Gradient Boosting is an ensemble learning technique that builds a series of weak models, usually decision trees, where each model corrects the errors of the previous one. It uses a gradient descent optimization approach to minimize the loss function, making predictions more accurate over time. The model is built from the sklearn library in python.

Figure 29 - Gradient Boosting Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	0.918295	0.637255	0.838294	0.724079

Figure 30 - Confusion Matrix - Gradient Boosting Model - Training Data

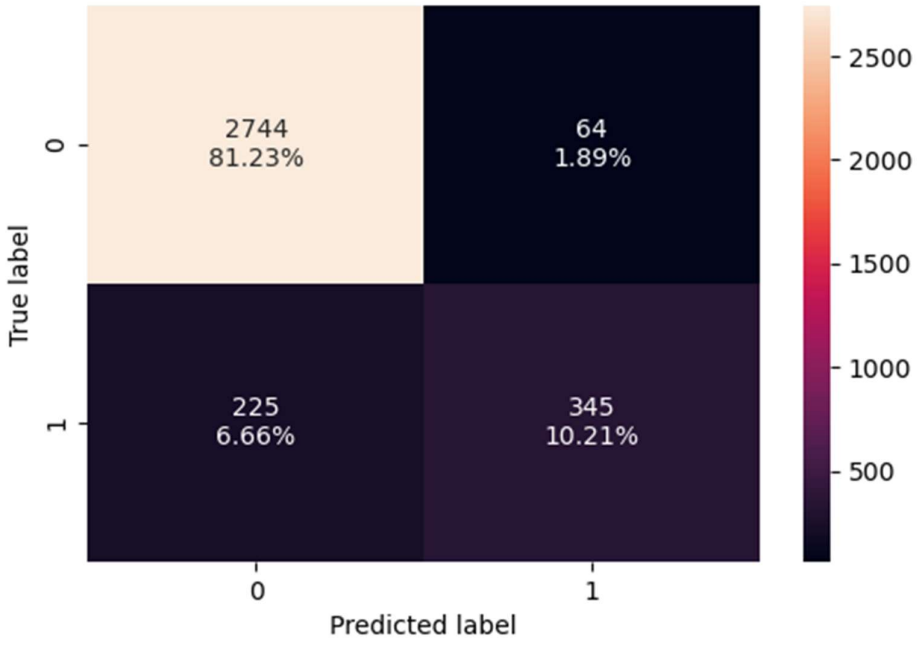


The model shows accuracy score of 0.918, correctly classifying a high number of cases. Its recall of 0.637 shows it successfully identifies a good portion of actual churners, though some are still missed. Precision is decent at 0.838, meaning that when the model predicts churn, it is usually correct with few false positives. The F1-score of 0.724 shows moderate balance between recall and precision, making the model effective while leaving room for recall improvement.

Figure 31 - Gradient Boosting Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.914446	0.605263	0.843521	0.704801

Figure 32 - Confusion Matrix - Gradient Boosting Model - Testing Data



The model shows an accuracy score of 0.914, correctly classifying most instances. Its recall score of 0.605 suggests it captures a decent portion of actual churners, though some are still missed. Precision score of 0.844 indicates that when it predicts churn, the prediction is mostly accurate with minimal false positives. The F1-score of 0.705 reflects a good balance between recall and precision, making the model somewhat effective on unseen data. The model could be optimal with some tuning.

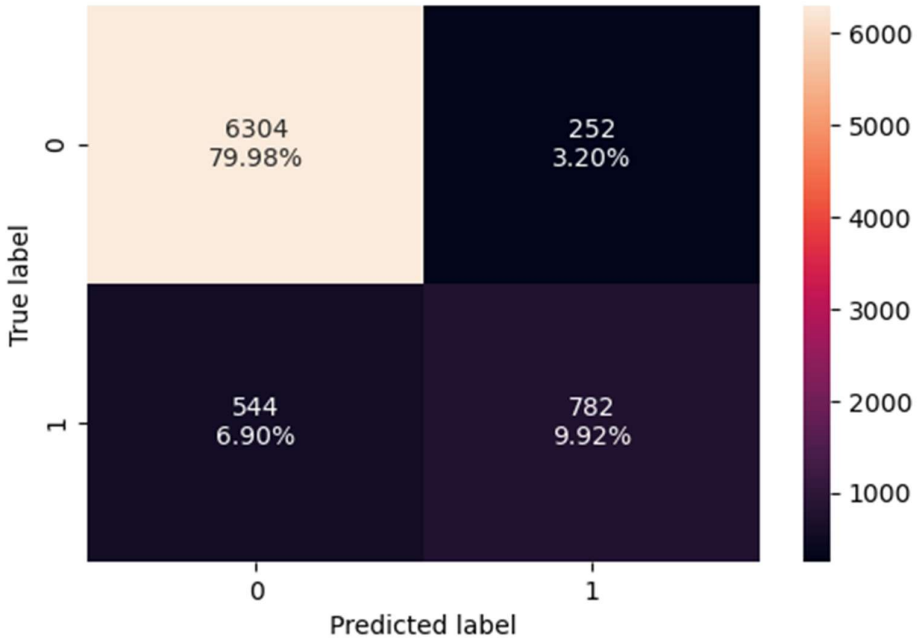
2.1.3 Adaboosting Model

Adaboost is an ensemble learning technique that improves weak classifiers by focusing on misclassified instances in each iteration. It assigns higher weights to hard-to-classify cases, refining predictions through multiple rounds. The model is built from the sklearn library in python.

Figure 33 - Adaboosting Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	0.89901	0.589744	0.756286	0.662712

Figure 34 - Confusion Matrix - Adaboosting Model - Training Data

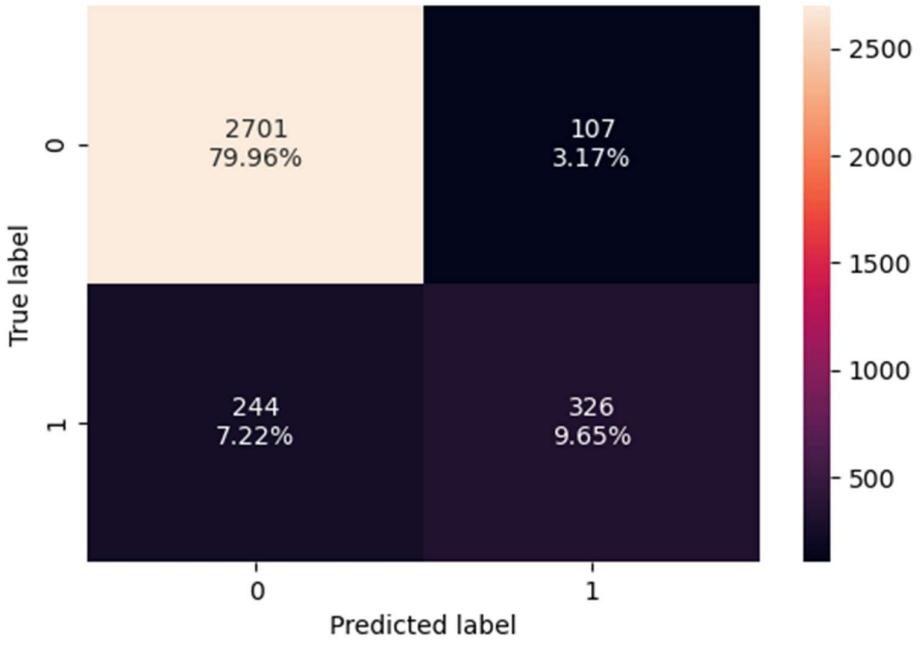


The model shows an accuracy score of 0.899, meaning it correctly classifies most cases. Its recall of 0.590 suggests that while it identifies a reasonable portion of actual churners, some are still missed. Precision stands at 0.756, indicating that when the model predicts churn, it is fairly reliable. The F1-score of 0.663 shows a moderate balance between recall and precision.

Figure 35 - Adaboosting Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.896092	0.57193	0.752887	0.65005

Figure 36 - Confusion Matrix - Adaboosting Model - Testing Data



The model shows an accuracy score of 0.896, correctly classifying most cases. Its recall of 0.572 suggests that it detects a good portion of actual churners, though some are still missed. Precision score of 0.753 indicates that when the model predicts churn, it is reliable with false positives. The F1-score of 0.650 reflects a moderate balance between recall and precision.

2.2 Model Comparison

All the models built will be 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 figures show the accuracy, recall, precision and F1 scores of all the models built.

Figure 37 - Training Performance Comparision - All ML Models

Testing performance comparison:

	Logistic Regression	Decision Tree	Linear Discriminant Analysis	Artifical Neural Network	Support Vector Machine	K-Nearest Neighbors	Random Forest	Gradient Boosting	Adaboosting
Accuracy	0.884251	0.947010	0.878330	0.964772	0.915038	0.954115	0.972173	0.914446	0.896092
Recall	0.452632	0.815789	0.421053	0.884211	0.573684	0.815789	0.849123	0.605263	0.571930
Precision	0.765579	0.862709	0.747664	0.904847	0.881402	0.902913	0.983740	0.843521	0.752887
F1	0.568908	0.838593	0.538721	0.894410	0.695005	0.857143	0.911488	0.704801	0.650050

Figure 38 - Testing Performance Comparision - All ML Models

Training performance comparison:

	Logistic Regression	Decision Tree	Linear Discriminant Analysis	Artifical Neural Network	Support Vector Machine	K-Nearest Neighbors	Random Forest	Gradient Boosting	Adaboosting
Accuracy	0.887592	0.947010	0.880868	0.993149	0.923370	0.978432	1.0	0.918295	0.899010
Recall	0.457768	0.815789	0.415535	0.971342	0.604827	0.907240	1.0	0.637255	0.589744
Precision	0.784238	0.862709	0.770629	0.987730	0.909297	0.962400	1.0	0.838294	0.756286
F1	0.578095	0.838593	0.539931	0.979468	0.726449	0.934006	1.0	0.724079	0.662712

Top 3 Optimal Models prioritizing the metric 'recall' which will be tuned are

Artificial Neural Network (ANN)

It shows a high accuracy along with good recall and F1-score, making it effective at identifying churners. The model can be tuned to get a higher score. The model is best for non-linear customer behavior trends.

Gradient Boosting

It shows high accuracy scores, good recall and strong precision scores. This model works well on imbalanced data and is less prone to overfitting than Random Forest. This model will be tuned using GridsearchCV to get more optimal scorers.

Random Forest

It shows very good accuracy scores, but may be overfitting, high recall scores and high precision scores. This model is great for feature importance analysis to understand why customers churn. Works well with structured, tabular data.

2.3 Model Tuning

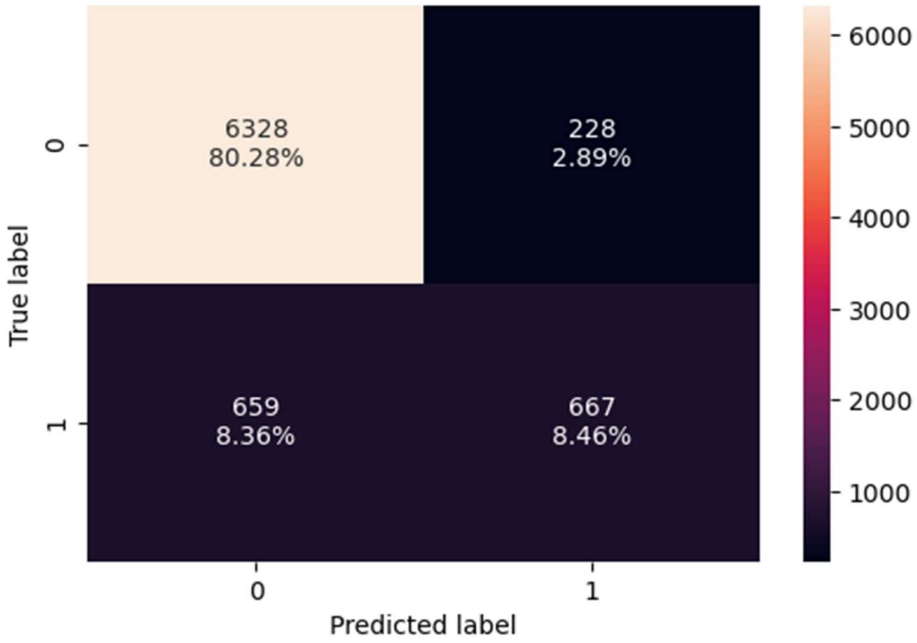
2.3.1 Artificial Neural Network Model

The model will be tuned by hyperparameter tuning for Multi-Layer Perceptron Neural Network using GridsearchCV. It tests different hidden layer sizes, activation functions, and optimizers. Uses 5-fold cross-validation for reliable tuning and trains the best model using the optimal parameters and evaluates performance.

Figure 39 - Tuned Artificial Neural Network Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	0.887465	0.503017	0.745251	0.60063

Figure 40 - Confusion Matrix - Tuned Artificial Neural Network Model - Training Data

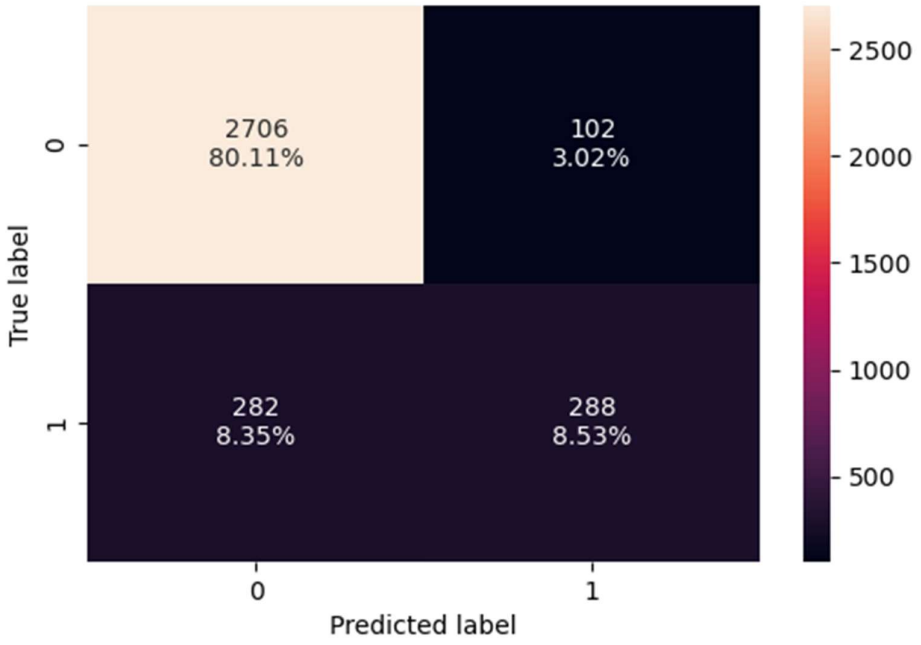


The tuned model shows an accuracy score of 0.887, meaning it correctly classifies a significant portion of instances. Its recall of 0.503 suggests that while it captures some actual churners but many are still missed. Precision score of 0.745 shows that when the model predicts churn, it is fairly reliable but with few false positives. The F1-score of 0.601 shows moderate balance between recall and precision.

Figure 41 - Tuned Artificial Neural Network Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.886323	0.505263	0.738462	0.6

Figure 42 - Confusion Matrix - Tuned Artificial Neural Network Model - Testing Data



The tuned model shows an accuracy score of 0.886 which correctly classifies most cases. Its recall score of 0.505 suggests it captures a moderate portion of actual churners though some are still missed. Precision score of 0.738 indicates that when the model predicts churn, it is generally reliable with few false positives. The F1-score of 0.600 reflects a good balance between recall and precision. Overall, the tuned model performance is worse than the normal model.

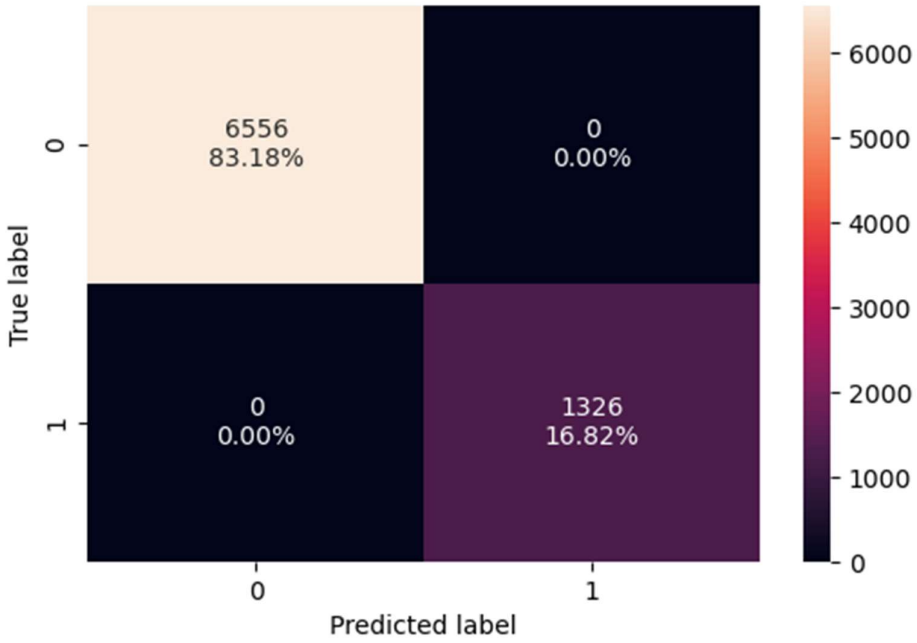
2.3.2 Gradient Boosting Model

The gradient boosting model is tuned using GridsearchCV by testing different parameter combinations. It evaluates various values for parameters like learning rate, number of trees, max depth, and subsample ratio to find the best-performing configuration

Figure 43 - Tuned Gradient Boosting Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	1.0	1.0	1.0	1.0

Figure 44 - Confusion Matrix - Tuned Gradient Boosting Model - Training Data

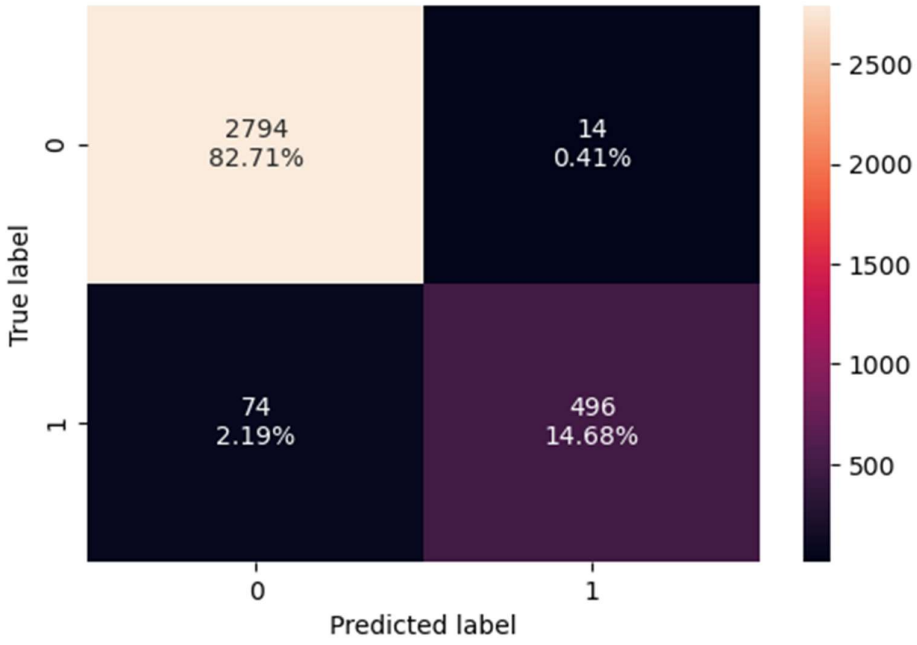


The tuned model performs perfectly on the training data which indicates a 100% correct prediction rate of churners and non-churners. The model needs to be performed on test data to identify any potential overfitting.

Figure 45 - Tuned Gradient Boosting Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.973949	0.870175	0.972549	0.918519

Figure 46 - Confusion Matrix - Tuned Gradient Boosting Model - Testing Data



The tuned model shows an accuracy score of 0.973, correctly classifying nearly all cases. Its recall score of 0.870 shows that it correctly identifies a high proportion of actual churners, with low missed cases. Precision score of 0.972 shows that when the model predicts churn, it is highly reliable with very few false positives. The F1-score of 0.918 shows very good balance between recall and precision. Overall, this model generalizes well without overfitting, making it a strong candidate.

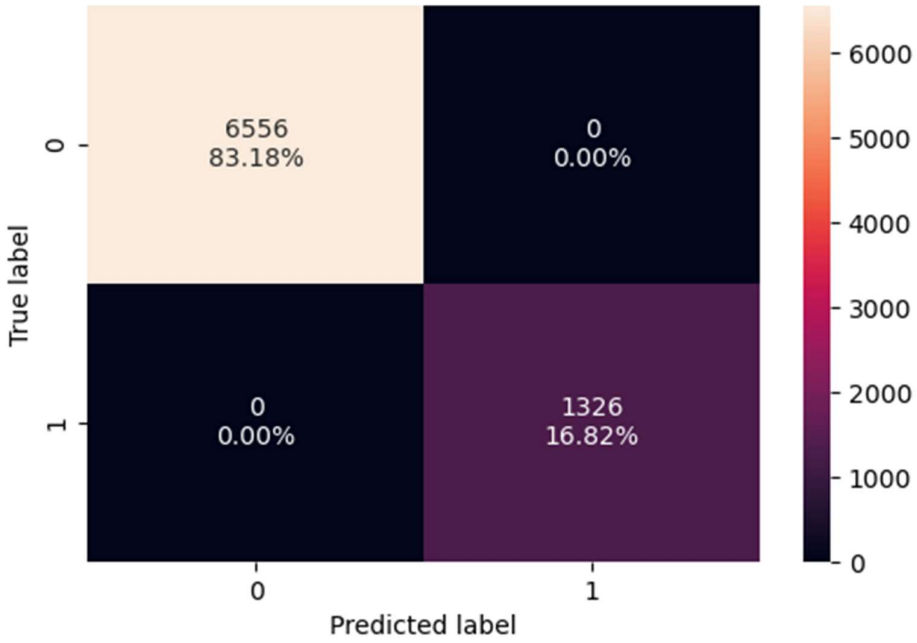
2.3.3 Random Forest Model

The random forest model can be tuned using GridsearchCV by searching for the best combination of values. It works by testing different settings for parameters like number of trees, maximum depth, minimum samples per split, and bootstrap options. GridSearchCV performs cross-validation which ensures the selected parameters generalize well to unseen data while preventing overfitting.

Figure 47 - Tuned Random Forest Model - Performance Scores - Training Data

	Accuracy	Recall	Precision	F1
0	1.0	1.0	1.0	1.0

Figure 48 - Confusion Matrix - Tuned Random Forest Model - Training Data

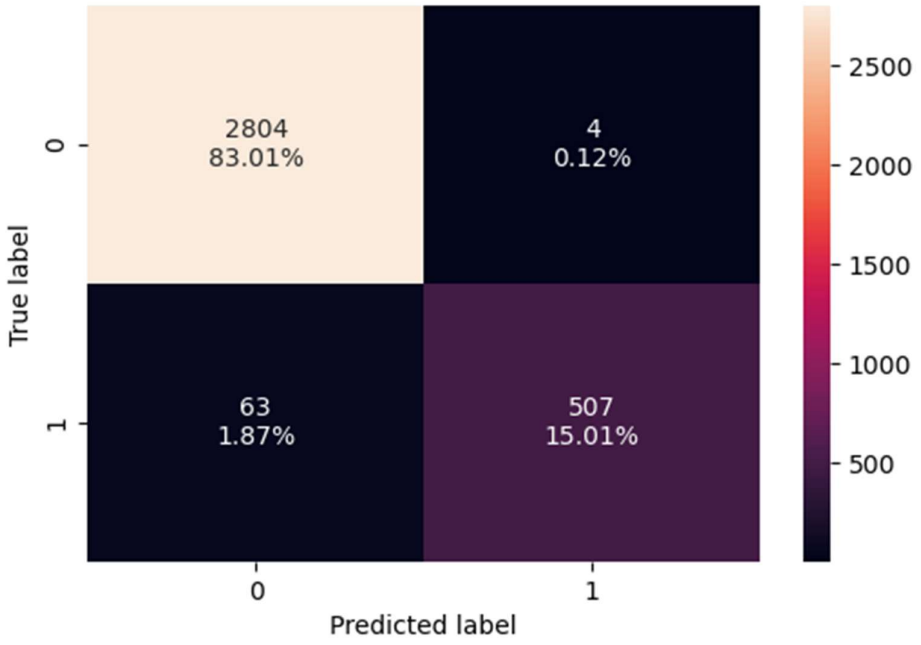


The tuned model performs perfectly on the training data which indicates a 100% correct prediction rate of churners and non-churners. The model needs to be performed on test data to identify any potential overfitting.

Figure 49 - Tuned Random Forest Model - Performance Scores - Testing Data

	Accuracy	Recall	Precision	F1
0	0.980166	0.889474	0.992172	0.93802

Figure 50 - Confusion Matrix - Tuned Random Forest Model - Testing Data



The tuned model shows an accuracy score of 0.980, correctly classifying nearly all cases. Its recall score of 0.889 shows that it correctly identifies a large proportion of actual churners, with low missed cases. Precision score of 0.992 shows that when the model predicts churn, it is highly reliable with very few false positives. The F1-score of 0.938 shows very good balance between recall and precision. Overall, this model performs better than the tuned gradient boost model making it the main contender for the optimal model.

2.4 Tuned Model Comparison

Figure 51 - Training Performance Comparison - Tuned ML Models

Training performance comparison:

	Tuned Artificial Neural Network	Tuned Gradient Boosting	Tuned Random Forest
Accuracy	0.887465	1.0	1.0
Recall	0.503017	1.0	1.0
Precision	0.745251	1.0	1.0
F1	0.600630	1.0	1.0

Figure 52 - Testing Performance Comparison - Tuned Models

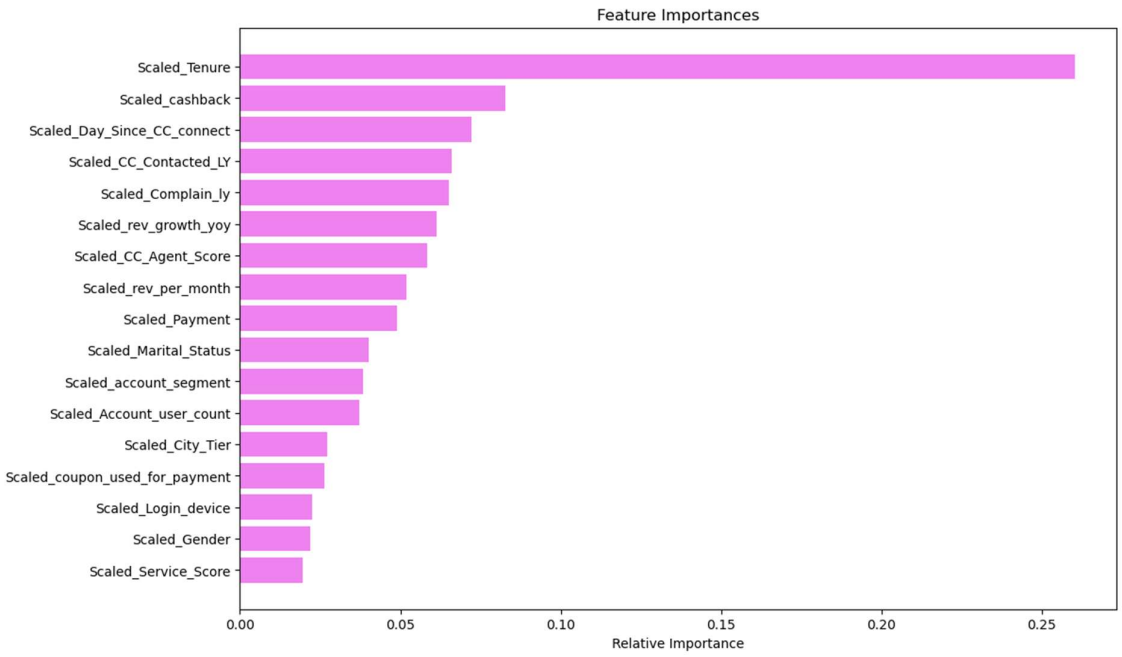
Testing performance comparison:

	Tuned Artificial Neural Network	Tuned Gradient Boosting	Tuned Random Forest
Accuracy	0.886323	0.973949	0.980166
Recall	0.505263	0.870175	0.889474
Precision	0.738462	0.972549	0.992172
F1	0.600000	0.918519	0.938020

On comparison of the 3 tuned models, we can see that the tuned random forest model is the most optimal for customer churn prediction with its perfect score on training data and very high score on testing data.

The tuned gradient boosting model scores lower than the tuned random forest however with its high scores its still an optimal model to use as well.

Figure 53 - Feature Importances Graph - Tuned Random Forest Model



On analysis of the feature importance, we can see that tenure, cashback, cc connect, last 12 months contact and complaints last year were the features which contributed to the most on building the random forest model. Service score, gender, login device had small impact on the model.

2.5 Implication of the Model on the Business

Utilizing the tuned random forest model will have the following implications on the business

- The model has very high prediction rate with nearly perfect accuracy on both training and test data, the model consistently classifies customers correctly, reducing the risk of misclassification.
- A recall of 0.889 ensures that most churners are identified, allowing the business to take proactive retention measures before losing valuable customers.
- A precision of 0.992 shows that when the model predicts churn, it is accurate most of the time which prevents unnecessary customer engagement efforts and saving resources.
- The test performance is very close to the training results, indicating that the tuned random forest model is not overfitted and can be confidently applied to real-world scenarios.
- The high model performance enables better-targeted retention strategies, reducing churn-related revenue losses and improving customer lifetime value.
- Accurate churn prediction allows businesses to stay ahead by refining customer service, pricing strategies, and personalized engagement to enhance loyalty.
- The model minimizes wasted effort on false churn predictions, ensuring marketing and customer service teams focus only on the right customers.

APPENDIX

Raw Code Snapshots:

```
# 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()
```

```
# 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')
```

```
# 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()
```

```
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 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()
```

```
# 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)
```

```
C:\Users\Ishaan Shakti\AppData\Local\Temp\ipykernel_11164\357238523.py:10: FutureWarning: Downcasting behavior in `replace` is deprecated and will be rem
oved in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_o
ption('future.no_silent_downcasting', True)`
  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)
```

```
C:\Users\Ishaan Shakti\AppData\Local\Temp\ipykernel_11164\3322179126.py:7: FutureWarning: Downcasting behavior in `replace` is deprecated and will be rem
oved in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_o
ption('future.no_silent_downcasting', True)`
  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)
```

```
C:\Users\Ishaan Shakti\AppData\Local\Temp\ipykernel_11164\2153673498.py:8: FutureWarning: Downcasting behavior in `replace` is deprecated and will be rem
oved in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_o
ption('future.no_silent_downcasting', True)`
  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)
```

```
# List of columns to scale
columns_to_scale = [
    'Churn', 'Tenure', 'City_Tier', 'CC_Contacted_LY', 'Payment',
    'Gender', 'Service_Score', 'Account_user_count', 'account_segment',
    'CC_Agent_Score', 'Marital_Status', 'rev_per_month', 'Complain_ly',
    'rev_growth_yoy', 'coupon_used_for_payment', 'Day_Since_CC_connect',
    'cashback', 'Login_device'
]

# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Apply scaling to each column in the list
for col in columns_to_scale:
    df[f'Scaled_{col}'] = scaler.fit_transform(df[[col]])
```

```
# creating new dataframe with scaled data
scaled_columns = [col for col in df.columns if col.startswith('Scaled_')]
df_scaled = df[scaled_columns]
df_scaled
```

```
df_scaled.info()
```

```
df['Churn'].value_counts()
```

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

```
# Checking the shape of training and test data
print('X_train',X_train.shape)
print('X_test',X_test.shape)
print('y_train',y_train.shape)
print('y_test',y_test.shape)
```

```
# 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=50)
lg_model.fit(X_train, y_train)
```

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

```
confusion_matrix_statsmodels(lg_model, X_train, y_train)
```

```
print("Training performance model 1:")
logistic_reg_perf_train = model_performance_classification_statsmodels(lg_model, X_train, y_train)
logistic_reg_perf_train
```

```
confusion_matrix_statsmodels(lg_model, X_test, y_test)
```

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

DECISION TREE MODEL ¶

```
from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier(random_state=1)
dt_model.fit(X_train, y_train)
```

```
decision_tree_perf_train = model_performance_classification_statsmodels(
    dt_model, X_train, y_train
)
decision_tree_perf_train
```

```
confusion_matrix_statsmodels(dt_model, X_train, y_train)
```

```
decision_tree_perf_test = model_performance_classification_statsmodels(
    dt_model, X_test, y_test
)
decision_tree_perf_test
```

```
confusion_matrix_statsmodels(dt_model, X_test, y_test)
```

LINEAR DISCRIMINANT ANALYSIS MODEL

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda_model = LinearDiscriminantAnalysis()
lda_model = lda_model.fit(X_train,y_train)
```

```
linear_discriminant_perf_train = model_performance_classification_statsmodels(
    lda_model, X_train, y_train
)
linear_discriminant_perf_train
```

```
confusion_matrix_statsmodels(lda_model, X_train, y_train)
```

```
linear_discriminant_perf_test = model_performance_classification_statsmodels(
    lda_model, X_test, y_test
)
linear_discriminant_perf_test
```

```
confusion_matrix_statsmodels(lda_model, X_test, y_test)
```

ARTIFICAL NEURAL NETWORK

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

```
artifical_neural_perf_train = model_performance_classification_statsmodels(
    ann_model, X_train, y_train
)
artifical_neural_perf_train
```

```
confusion_matrix_statsmodels(ann_model, X_train, y_train)
```

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

```
confusion_matrix_statsmodels(ann_model, X_test, y_test)
```

SUPPORT VECTOR MACHINE ¶

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

```
support_vector_perf_train = model_performance_classification_statsmodels(
    svm_model, X_train, y_train
)
support_vector_perf_train
```

```
confusion_matrix_statsmodels(svm_model, X_train, y_train)
```

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

```
confusion_matrix_statsmodels(svm_model, X_test, y_test)
```

K-NEAREST NEIGHBOUR MODEL

```
from sklearn.neighbors import KNeighborsClassifier
knn_model=KNeighborsClassifier()
knn_model.fit(X_train,y_train)
```

```
kn_neighbour_perf_train = model_performance_classification_statsmodels(
    knn_model, X_train, y_train
)
kn_neighbour_perf_train
```

```
confusion_matrix_statsmodels(knn_model, X_train, y_train)
```

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

```
confusion_matrix_statsmodels(knn_model, X_test, y_test)
```

ENSEMBLE MODELS

RANDOM FOREST MODEL

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

```
random_forest_perf_train = model_performance_classification_statsmodels(
    rf_model, X_train, y_train
)
random_forest_perf_train
```

```
confusion_matrix_statsmodels(rf_model, X_train, y_train)
```

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

```
confusion_matrix_statsmodels(rf_model, X_test, y_test)
```

GRADIENT BOOSTING MODEL ¶

```
from sklearn.ensemble import GradientBoostingClassifier
gbm_model = GradientBoostingClassifier(n_estimators=100, random_state=1)
gbm_model.fit(X_train, y_train)
```

```
gradient_boosting_perf_train = model_performance_classification_statsmodels(
    gbm_model, X_train, y_train
)
gradient_boosting_perf_train
```

```
confusion_matrix_statsmodels(gbm_model, X_train, y_train)
```

```
gradient_boosting_perf_test = model_performance_classification_statsmodels(
    gbm_model, X_test, y_test
)
gradient_boosting_perf_test
```

```
confusion_matrix_statsmodels(gbm_model, X_test, y_test)
```

ADABOOSTING MODEL

```
from sklearn.ensemble import AdaBoostClassifier
ada_model = AdaBoostClassifier(n_estimators=51, random_state=1)
adamodel = ada_model.fit(X_train, y_train)
```

```
C:\ProgramData\anaconda3\Lib\site-packages\sklearn\ensemble\_weight_boosting.py:519: FutureWarning: The SAMME.R algorithm (the default) is deprecated and
will be removed in 1.6. Use the SAMME algorithm to circumvent this warning.
  warnings.warn(
```

```
ada_boosting_perf_train = model_performance_classification_statsmodels(
    ada_model, X_train, y_train
)
ada_boosting_perf_train
```

```
confusion_matrix_statsmodels(ada_model, X_train, y_train)
```

```
ada_boosting_perf_test = model_performance_classification_statsmodels(
    ada_model, X_test, y_test
)
ada_boosting_perf_test
```

```
confusion_matrix_statsmodels(ada_model, X_test, y_test)
```

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, y_train)

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

```
tuned_ann_model = MLPClassifier(activation='tanh', alpha=0.0001, hidden_layer_sizes=100, solver='adam', tol=0.001, verbose=True,random_state=1, max_iter=

tuned_ann_model.fit(X_train, y_train)
```

```
tuned_ann_perf_train = model_performance_classification_statsmodels(
    tuned_ann_model, X_train, y_train
)
tuned_ann_perf_train
```

```
confusion_matrix_statsmodels(tuned_ann_model, X_train, y_train)
```

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

```
confusion_matrix_statsmodels(tuned_ann_model, X_test, y_test)
```

GRADIENT BOOSTING MODEL

```
param_grid = {
    'n_estimators': [50, 100, 200],  # Number of boosting stages
    'learning_rate': [0.01, 0.05, 0.1],  # Controls how much each tree contributes
    'max_depth': [3, 5, 7],  # Depth of individual trees
    'min_samples_split': [2, 5, 10],  # Minimum samples required to split a node
    'min_samples_leaf': [1, 2, 4],  # Minimum samples per leaf node
    'subsample': [0.8, 1.0],  # Fraction of samples used per tree (helps prevent overfitting)
}

# Initialize the Gradient Boosting classifier
gb_model1 = GradientBoostingClassifier(random_state=42)
# Perform Grid Search with 5-fold cross-validation
grid_search = GridSearchCV(estimator=gb_model1, param_grid=param_grid, cv=5, scoring='recall', verbose=2, n_jobs=-1)
grid_search.fit(X_train, y_train)  # Fit the model on training data

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

```
tuned_gb_model = GradientBoostingClassifier(learning_rate=0.1, max_depth=7,
                        min_samples_leaf=1,
                           min_samples_split=10, n_estimators=200,
                           random_state=1)
tuned_gb_model.fit(X_train, y_train)
```

```
tuned_gbm_perf_train = model_performance_classification_statsmodels(
    tuned_gb_model, X_train, y_train
)
tuned_gbm_perf_train
```

```
confusion_matrix_statsmodels(tuned_gb_model, X_train, y_train)
```

```
tuned_gbm_perf_test = model_performance_classification_statsmodels(
    tuned_gb_model, X_test, y_test
)
tuned_gbm_perf_test
```

```
confusion_matrix_statsmodels(tuned_gb_model, X_test, y_test)
```

RANDOM FOREST MODEL

```
param_grid = {
    'n_estimators': [50, 100, 200],  # Number of trees in the forest
    'max_depth': [None, 10, 20, 30],  # Maximum depth of the trees
    'min_samples_split': [2, 5, 10],  # Minimum samples required to split a node
    'min_samples_leaf': [1, 2, 4],  # Minimum samples required at a leaf node
    'bootstrap': [True, False]  # Whether bootstrap samples are used when building trees
}
# 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, y_train)  # 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, y_train)
```

```
tuned_rf_perf_train = model_performance_classification_statsmodels(
    tuned_rf_model, X_train, y_train
)
tuned_rf_perf_train
```

```
confusion_matrix_statsmodels(tuned_rf_model, X_train, y_train)
```

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

```
confusion_matrix_statsmodels(tuned_rf_model, X_test, y_test)
```

TUNED MODEL COMPARISON ¶

```
# training performance comparison

models_train_comp_df1 = pd.concat(
    [tuned_ann_perf_train.T,
        tuned_gbm_perf_train.T,
        tuned_rf_perf_train.T,
    ],
    axis=1,
)
models_train_comp_df1.columns = ["Tuned Artificial Neural Network",
    "Tuned Gradient Boosting",
    "Tuned Random Forest"
]
print("Training performance comparison:")
models_train_comp_df1
```

```
# testing performance comparison

models_test_comp_df1 = pd.concat(
    [tuned_ann_perf_test.T,
        tuned_gbm_perf_test.T,
        tuned_rf_perf_test.T,
    ],
    axis=1,
)
models_test_comp_df1.columns = ["Tuned Artificial Neural Network",
    "Tuned Gradient Boosting",
    "Tuned Random Forest"
]
print("Testing performance comparison:")
models_test_comp_df1
```

```
feature_names = X_train.columns
importances = tuned_rf_model.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12, 8))
plt.title("Feature Importances")
plt.barh(range(len(indices)), importances[indices], color="violet", align="center")
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel("Relative Importance")
plt.show()
```