SYRIATEL CUSTOMER CHURN PREDICTION: LEVERAGING DATA TO IMPROVE CUSTOMER RETENTION

Author: Evaclaire Munyika Wamitu

Student pace: Remote

Technical mentors: Asha Deen / Lucille Kaleha

BUSINESS UNDERSTANDING

SyriaTel, a leading telecommunications provider, is currently grappling with a significant business challenge: identifying customers who are likely to cancel their services in the near future. Customer churn is a pervasive issue in the telecommunications industry directly impacting both revenue and profitability. High churn rates can lead to substantial financial losses, increased customer acquisition costs and a negative impact on brand reputation.

By leveraging advanced predictive analytics and machine learning techniques, this project aims to develop a robust and reliable model to accurately predict customer churn. This predictive model will enable SyriaTel to proactively identify at-risk customers and implement targeted retention strategies thereby mitigating the financial repercussions of customer attrition.

Key stakeholders and their Interests

- **SyriaTel**: The primary stakeholder, focused on reducing customer churn and minimizing the financial losses associated with customer departures. The company aims to enhance customer satisfaction and loyalty, thereby securing long-term revenue growth.
- **Executives**: Senior management who are keen on using data-driven insights to steer strategic decisions. They are interested in high-level summaries of churn predictors and actionable insights that can inform company-wide policies and investment in retention initiatives.
- **Marketing Teams**: Responsible for designing and implementing customer retention campaigns. They require detailed understanding of the factors driving churn to create personalized marketing strategies aimed at retaining at-risk customers and maximizing customer lifetime value.
- **Customer Service Representatives**: Frontline employees who interact directly with customers. They need timely information about at-risk customers to provide proactive support and personalized service, thereby enhancing the customer experience and reducing churn.

By delivering deep insights into the underlying factors contributing to customer churn and identifying customers most at risk, the developed model will empower SyriaTel to make

Challenges

Developing a predictive model for customer churn in the telecommunications industry presents multifaceted challenges that necessitate a comprehensive understanding of both industry dynamics and advanced analytics techniques. Key hurdles include ensuring data quality, crafting informative features that capture nuanced customer behaviors and addressing class imbalance inherent in churn datasets. Additionally, selecting and evaluating appropriate machine learning algorithms while guarding against overfitting and ensuring model interpretability are crucial. By navigating these challenges with a methodical approach, the resulting model will not only accurately predict churn but also provide actionable insights, empowering SyriaTel to proactively manage customer attrition, enhance retention strategies and drive sustained business growth.

Proposed Solutions

To address the challenges inherent in developing a predictive model for customer churn in the telecommunications industry, several proposed solutions have been identified. Rigorous data cleaning and preprocessing techniques will be employed to ensure data quality and integration including handling missing values, outliers and inconsistencies. Feature engineering will involve extensive exploratory data analysis (EDA) to uncover meaningful patterns and correlations with new features engineered based on domain knowledge and insights from EDA to enhance predictive power. Techniques such as oversampling, undersampling and SMOTE will be utilized to mitigate class imbalance ensuring effective learning from both churned and non-churned instances.

Model selection will involve evaluating multiple machine learning algorithms and employing robust cross-validation techniques and performance metrics to identify the best-performing model. Finally, focusing on selecting interpretable algorithms and presenting findings in a clear and actionable manner will ensure stakeholders can make informed decisions and take targeted actions based on model predictions. These solutions collectively aim to develop a robust predictive model that provides actionable insights for SyriaTel and its stakeholders.

The final step will be deploying the trained model into a production environment ensuring seamless integration with SyriaTel's existing systems and processes. The model's performance will be monitored continuously and retrained and/or updated as needed to maintain its effectiveness.

Problem Statement

SyriaTel, a telecommunications company, is facing a significant challenge with customer churn which is causing substantial financial losses. As customers terminate their services and switch to competitors, SyriaTel's revenue streams are impacted and the company incurs additional costs associated with acquiring new customers to replace the ones who have churned.

Objectives

Main objective

The primary objective of this project is to develop a robust binary classification model capable of accurately predicting whether a customer will churn from SyriaTel in the near future. This model aims to identify potential churners proactively, enabling targeted retention strategies and improving customer satisfaction and loyalty.

Specific objectives

1. To perform data preparation by loading the data, preview the data, get summary statistics followed by in-depth exploratory data analysis (EDA) to address issues such as missing values, null values, outliers and duplicates.
2. To investigate relationships between numeric and categorical features and the target variable (customer churn) to identify potential predictors and gain insights into customer behavior in order to enhance the model's predictive power through effective feature engineering.
3. To evaluate and compare the performance of various machine learning algorithms such as logistic regression, decision trees, random forests and K-Nearest Neighbors for the binary classification task using appropriate evaluation metrics like ROC-AUC, precision, recall, accuracy and F1-score.
4. To implement techniques to address potential class imbalance such as oversampling, undersampling to ensure accurate prediction of both churned and non-churned customers.
5. Perform rigorous model validation using techniques such as GridSearchCV to assess the model's generalization performance.

DATA UNDERSTANDING

The dataset used in this project is called Churn in Telecom's from Kaggle (https://www.kaggle.com/datasets/becksddf/churn-in-telecoms-dataset).

The dataset provides customer activity data and information about whether they canceled their subscription with the telecom company. The goal is to develop predictive models that can help reduce financial losses caused by customers who do not remain with the company for an extended period.

The dataset consists of 3333 entries and 21 columns namely state, account length, area code, phone number, international plan, voicemail plan, number of voicemail messages, total day minutes, total day calls, total day charge, total evening minutes, total evening calls, total evening charge, total night minutes, total night calls, total night charge, total international minutes, total international calls, total international charge, customer service calls, and churn status.

Below is a detailed explanation of the dataset's column names:

- **State** - The state the customer resides.
- **Account Length** - Number of days the customer has had an account.
- **Area Code** - The area code of the customer.
- **Phone Number** - The phone number of the customer.
- **International Plan** - True if the customer has an international plan, false if they dont.
- **Voice Mail Plan** - True if the customer has a voice mail plan, false if they dont.
- **Number Vmail Messages** - the number of voicemails the customer has sent.
- **Total Day Minutes** - Total number of call minutes during the day.
- **Total Day Calls** - Total number of calls made by the customer during the day.
- **Total Day Charge** - Total amount of money charged to the customer by the Syriatel for calls made during the day.
- **Total Eve Minutes** - Total number of customer call minutes in the evening.
- **Total Eve Calls** - Total number of calls the customer has made in the evening.
- **Total Eve Charge** - Total amount of money charged to the customer by the Syriatel for calls made in the evening.
- **Total Night Minutes** - Total number of customer call minutes in the night time.
- **Total Night Calls** - Total number of calls the customer has made in the night time.
- **Total Night Charge** - Total amount of money the customer was charged by Syriatel for calls made in the night time.
- **Total Intl Minutes** - Total number of minutes the customer has accrued on international calls.
- **Total Intl Calls** - Total number of international calls the customer has made.
- **Total Intl Charge** - Total amount of money the customer was charged by Syriatel for international calls made.
- **Customer Service Calls** - Number of calls the customer has made to customer service.
- **Churn** - True if the customer terminated their contract,false if they didnt.

DATA PREPARATION

This section will involve preparing the data for exploratory data analysis (EDA) and later modelling. The first step will be to import all the libraries relevant to this project then load the dataset into a pandas dataframe, preview the data and check for any missing, null or

duplicate values

In [1]:
```
# Import relevant libraries

# Data manipulation
import pandas as pd
import numpy as np
from scipy import stats

# Data visualization
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.graph_objs as go
import plotly.express as px
%matplotlib inline

# Data modelling
from sklearn.model_selection import train_test_split, cross_val_score,
from imblearn.over_sampling import SMOTE
from sklearn.metrics import accuracy_score, f1_score, precision_score,
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.utils.metaestimators import available_if
from statsmodels.stats.outliers_influence import variance_inflation_fa

# Load supervised learning algorithms
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
import xgboost as xgb
from xgboost import XGBClassifier
from sklearn.svm import SVC

# Filter future warnings
import warnings
warnings.filterwarnings('ignore')
```

Below I created a DataExplorer class to facilitate the loading, exploration and basic analysis of data from a csv file using pandas. The class includes several methods namely:

- load_data(): This method loads the CSV file into a pandas DataFrame and handles potential file loading errors.
- get_shape(): This method prints the dimensions of the DataFrame, indicating the number of rows and columns.
- summarize_info(): This method provides a summary of the DataFrame's columns, including data types and non-null counts.
- describe_data(): This method outputs descriptive statistics of the DataFrame, such as mean, standard deviation, and quartiles for numerical columns.
- display_column_types(): This method identifies and displays the numerical and categorical columns in the DataFrame.

In [2]:
```
import pandas as pd

class DataExplorer:
    '''
    A class to handle and explore pandas DataFrames.

    Attributes:
        file_path (str): The file path of the csv file to load.
        data (pandas.DataFrame): The loaded DataFrame.

    Methods:
        load_data(): Load the CSV file into a DataFrame.
        get_shape(): Print the number of rows and columns in the DataF
        summarize_info(): Print a summary of the DataFrame columns.
        describe_data(): Print descriptive statistics of the DataFrame
    '''

    def __init__(self, file_path):
        '''
        Initialize the DataExplorer object.

        Args:
            file_path (str): The file path of the csv file to load.
        '''
        self.file_path = file_path
        self.data = None

    def load_data(self):
        '''
        Load the csv file into a DataFrame.

        Returns:
            None
        '''
        print('Loading the customer data csv file...')
        try:
            self.data = pd.read_csv(self.file_path)
            print(f'Dataset loaded successfully from {self.file_path}\
        except FileNotFoundError:
            print(f'Error: The file \'{self.file_path}\' was not found
        except Exception as e:
            print(f'Error: An unexpected error occurred: {e}')

    def get_shape(self):
        '''
        Print the number of rows and columns in the DataFrame.

        Returns:
            None
        '''
        if self.data is not None:
            rows, columns = self.data.shape
            print(f'The DataFrame has {rows} rows and {columns} columr
        else:
            print('Error: No data loaded yet. Please call the load_dat


    def summarize_info(self):
        '''
        Print a summary of the DataFrame columns.
```

```
        Returns:
            None
        '''
        print('Summarizing the DataFrame info')
        print('------------------------------')
        if self.data is not None:
            print(self.data.info())
        else:
            print('Error: No data loaded yet. Please call the load_dat

    def describe_data(self):
        '''
        Print descriptive statistics of the DataFrame.

        Returns:
            None
        '''
        print('\nDescribing the DataFrame data')
        print('-----------------------------')

        if self.data is not None:
            print(self.data.describe())
        else:
            print('Error: No data loaded yet. Please call the load_dat

    def display_column_types(self):
        '''
        Display numerical and categorical columns.

        Returns:
            None
        '''
        print('\nDisplaying numerical and categorical columns')
        print('--------------------------------------------')
        if self.data is not None:
            numerical_columns = self.data.select_dtypes(include='numbe
            categorical_columns = self.data.select_dtypes(include='obj
            print(f'Numerical Columns: {numerical_columns}\n')
            print(f'Categorical Columns: {categorical_columns}\n')
        else:
            print('Error: No data loaded yet. Please call the load_dat



# Instantiate
file_path = 'customer_data.csv'
data_explorer = DataExplorer(file_path)

# Load data
data_explorer.load_data()

# Get dimensions
data_explorer.get_shape()

# Summarize info
data_explorer.summarize_info()

# Describe data
data_explorer.describe_data()

# Display numerical and categorical columns
```

```
data_explorer.display_column_types()
```

```
Loading the customer data csv file...
Dataset loaded successfully from customer_data.csv

The DataFrame has 3333 rows and 21 columns.

Summarizing the DataFrame info
------------------------------
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3333 entries, 0 to 3332
Data columns (total 21 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   state                   3333 non-null   object 
 1   account length          3333 non-null   int64  
 2   area code               3333 non-null   int64  
 3   phone number            3333 non-null   object 
 4   international plan      3333 non-null   object 
 5   voice mail plan         3333 non-null   object 
 6   number vmail messages   3333 non-null   int64  
 7   total day minutes       3333 non-null   float64
 8   total day calls         3333 non-null   int64  
 9   total day charge        3333 non-null   float64
 10  total eve minutes       3333 non-null   float64
 11  total eve calls         3333 non-null   int64  
 12  total eve charge        3333 non-null   float64
 13  total night minutes     3333 non-null   float64
 14  total night calls       3333 non-null   int64  
 15  total night charge      3333 non-null   float64
 16  total intl minutes      3333 non-null   float64
 17  total intl calls        3333 non-null   int64  
 18  total intl charge       3333 non-null   float64
 19  customer service calls  3333 non-null   int64  
 20  churn                   3333 non-null   bool   
dtypes: bool(1), float64(8), int64(8), object(4)
memory usage: 524.2+ KB
None

Describing the DataFrame data
-----------------------------
       account length    area code  number vmail messages  total day
minutes  \
count     3333.000000  3333.000000            3333.000000        333
3.000000
mean       101.064806   437.182418               8.099010         17
9.775098
std         39.822106    42.371290              13.688365          5
4.467389
min          1.000000   408.000000               0.000000
0.000000
25%         74.000000   408.000000               0.000000         14
3.700000
50%        101.000000   415.000000               0.000000         17
9.400000
75%        127.000000   510.000000              20.000000         21
6.400000
max        243.000000   510.000000              51.000000         35
0.800000

       total day calls  total day charge  total eve minutes  total e
ve calls  \
count      3333.000000       3333.000000        3333.000000      333
```

```
3.000000
mean        100.435644         30.562307         200.980348       10
0.114311
std          20.069084          9.259435          50.713844        1
9.922625
min           0.000000          0.000000           0.000000
0.000000
25%          87.000000         24.430000         166.600000        8
7.000000
50%         101.000000         30.500000         201.400000       10
0.000000
75%         114.000000         36.790000         235.300000       11
4.000000
max         165.000000         59.640000         363.700000       17
0.000000

       total eve charge  total night minutes  total night calls  \
count       3333.000000          3333.000000        3333.000000
mean          17.083540           200.872037         100.107711
std            4.310668            50.573847          19.568609
min            0.000000            23.200000          33.000000
25%           14.160000           167.000000          87.000000
50%           17.120000           201.200000         100.000000
75%           20.000000           235.300000         113.000000
max           30.910000           395.000000         175.000000

       total night charge  total intl minutes  total intl calls  \
count         3333.000000         3333.000000       3333.000000
mean             9.039325           10.237294          4.479448
std              2.275873            2.791840          2.461214
min              1.040000            0.000000          0.000000
25%              7.520000            8.500000          3.000000
50%              9.050000           10.300000          4.000000
75%             10.590000           12.100000          6.000000
max             17.770000           20.000000         20.000000

       total intl charge  customer service calls
count        3333.000000             3333.000000
mean            2.764581                1.562856
std             0.753773                1.315491
min             0.000000                0.000000
25%             2.300000                1.000000
50%             2.780000                1.000000
75%             3.270000                2.000000
max             5.400000                9.000000

Displaying numerical and categorical columns
------------------------------------------------
Numerical Columns: Index(['account length', 'area code', 'number vma
il messages',
       'total day minutes', 'total day calls', 'total day charge',
       'total eve minutes', 'total eve calls', 'total eve charge',
       'total night minutes', 'total night calls', 'total night char
ge',
       'total intl minutes', 'total intl calls', 'total intl charg
e',
       'customer service calls'],
      dtype='object')

Categorical Columns: Index(['state', 'phone number', 'international
```

```
plan', 'voice mail plan'], dtype='object')
```

Data Cleaning

Next I created a DataCleaner class to handle and clean the data. This class provides several methods for preprocessing the data, making it more suitable for analysis and later modeling. The following are the methods included in the class:

- check_missing(): This method checks for columns with missing values.
- drop_missing(threshold=0.5): This method drops columns with missing values exceeding a specified threshold (default is 50%).
- fill_missing(method='mean'): This method fills missing values using the mean. The mean is proposed for this instance because it minimizes the disruption to the existing data structure.
- drop_duplicates(): This method removes duplicate rows from the DataFrame.

In [3]:
```
import pandas as pd

class DataCleaner:
    '''
    A class to handle and clean pandas DataFrames.

    Attributes:
        data (pandas.DataFrame): The DataFrame to clean.

    Methods:
        drop_missing(threshold=0.5): Drop columns with missing values
        fill_missing(method='mean'): Fill missing values using a speci
        drop_duplicates(): Drop duplicate rows in the DataFrame.
        encode_categorical(): Encode categorical columns using one-hot
        normalize_columns(): Normalize numerical columns.
        clean_column_names(): Remove whitespace from column names and
    '''

    def __init__(self, data):
        '''
        Initialize the DataCleaner object.

        Args:
            data (pandas.DataFrame): The DataFrame to clean.
        '''
        self.data = data

    def check_missing(self):
        '''
        Check columns with missing values.

        Returns:
            None
        '''
        print("\nChecking columns with missing values")
        print('-----------------------------------------------\n')
        missing_values = self.data.isna().sum()
        print(f'Number of missing values: \n{missing_values}')

    def drop_missing(self, threshold=0.5):
        '''
        Drop columns with missing values above a given threshold.

        Args:
            threshold (float): The threshold above which columns will

        Returns:
            None
        '''
        print(f'\nDropping columns with more than {threshold*100}% mis
        print('-----------------------------------------------\n')
        missing_fraction = self.data.isnull().mean()
        columns_to_drop = missing_fraction[missing_fraction > threshol
        self.data.drop(columns=columns_to_drop, inplace=True)
        print(f'Columns dropped: {columns_to_drop.tolist()}')

    def fill_missing(self, method='mean'):
        '''
        Fill missing values using a specified method.

        Args:
```

```
            method (str): The method to use for filling missing values

        Returns:
            None
        '''
        if method == 'mean':
            self.data.fillna(self.data.mean(), inplace=True)
        elif method == 'median':
            self.data.fillna(self.data.median(), inplace=True)
        elif method == 'mode':
            self.data.fillna(self.data.mode().iloc[0], inplace=True)
        else:
            print(f'Error: Method {method} not recognized.')

    def drop_duplicates(self):
        '''
        Drop duplicate rows in the DataFrame.

        Returns:
            None
        '''
        initial_shape = self.data.shape
        self.data.drop_duplicates(inplace=True)
        final_shape = self.data.shape
        print(f'Dropped {initial_shape[0] - final_shape[0]} duplicate

    def clean_column_names(self):
        '''
        Removes whitespace from column names and replaces it with unde

        Returns:
            None
        '''
        self.data.columns = self.data.columns.str.replace(' ', '_')
        print('New column names:')
        print('-----------------------------------------\n')
        print(self.data.columns.tolist())

# Instantiate
data_cleaner = DataCleaner(data_explorer.data)

# Clean column names
data_cleaner.clean_column_names()

# Check for missing values
data_cleaner.check_missing()

# Drop columns with more than 50% missing values
data_cleaner.drop_missing(threshold=0.5)

# Fill missing values using the mean method
data_cleaner.fill_missing(method='mean')

# Drop duplicate rows
data_cleaner.drop_duplicates()
```

```
New column names:
--------------------------------------------

['state', 'account_length', 'area_code', 'phone_number', 'internatio
nal_plan', 'voice_mail_plan', 'number_vmail_messages', 'total_day_mi
nutes', 'total_day_calls', 'total_day_charge', 'total_eve_minutes',
'total_eve_calls', 'total_eve_charge', 'total_night_minutes', 'total
_night_calls', 'total_night_charge', 'total_intl_minutes', 'total_in
tl_calls', 'total_intl_charge', 'customer_service_calls', 'churn']

Checking columns with missing values
--------------------------------------------

Number of missing values:
state                     0
account_length            0
area_code                 0
phone_number              0
international_plan        0
voice_mail_plan           0
number_vmail_messages     0
total_day_minutes         0
total_day_calls           0
total_day_charge          0
total_eve_minutes         0
total_eve_calls           0
total_eve_charge          0
total_night_minutes       0
total_night_calls         0
total_night_charge        0
total_intl_minutes        0
total_intl_calls          0
total_intl_charge         0
customer_service_calls    0
churn                     0
dtype: int64

Dropping columns with more than 50.0% missing values...
--------------------------------------------

Columns dropped: []
Dropped 0 duplicate rows.
```

Now that we have removed all the whitespace in our column names and there are no missing, null or duplicate values, we proceed to carry out our exploratory data analysis (EDA).

Exploratory Data Analysis

Univariate data analysis

Involves the study of a single variable, focusing on its distribution, characteristics and patterns within the dataset. In the context of this project, this approach will enable us to analyse the distribution and properties of each variable identifying any potential concerns such as the presence of outliers. We will begin by plotting histograms for the numeric features to understand their distributions.

First step is to have a general overview of the DataFrame by previewing the first five columns and assigning the variable 'data' to our DataCleaner class output for ease of reference.

In [4]:
```
data = data_cleaner.data
data.head()
```

Out[4]:

	state	account_length	area_code	phone_number	international_plan	voice_mail_plan	numbe
0	KS	128	415	382-4657	no	yes	
1	OH	107	415	371-7191	no	yes	
2	NJ	137	415	358-1921	no	no	
3	OH	84	408	375-9999	yes	no	
4	OK	75	415	330-6626	yes	no	

5 rows × 21 columns

In [5]:
```
data.area_code.nunique()
```

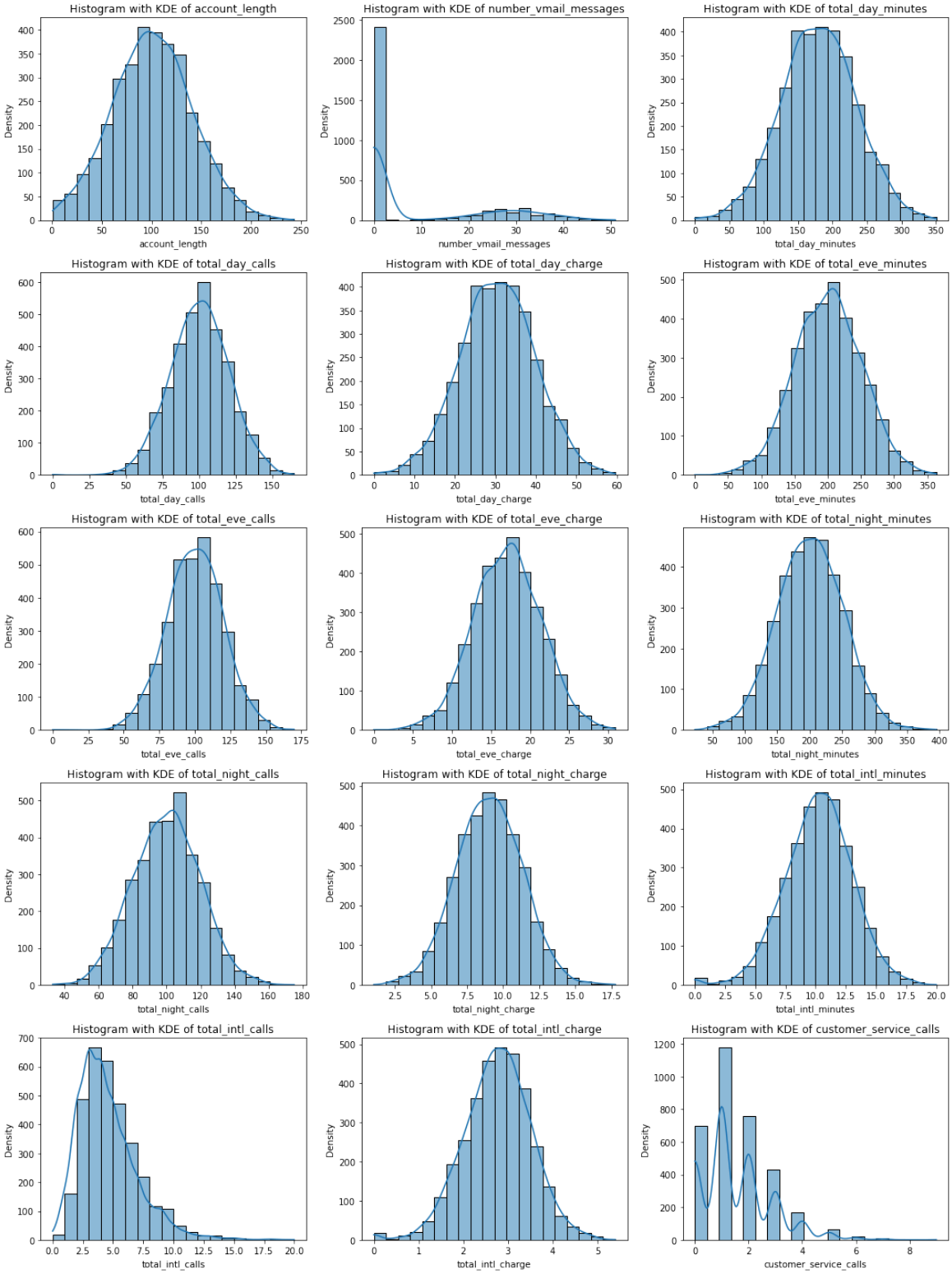
Out[5]: 3

I was curious about the 'area_code' column having only 3 unique values. The area_code variable likely represents categories or groups rather than continuous numerical values hence we shall convert it from an integer to an object.

In [6]:
```
data['area_code'] = data['area_code'].astype('object')
data.area_code.dtype
```

Out[6]: dtype('O')

In [7]:
```
def plot_numeric_columns(data):
    '''
    Plots histograms and kernel density estimates for numeric columns

    Args:
        data (pd.DataFrame): The input DataFrame containing numeric co

    Returns:
        None
    '''
    # Select numeric columns
    numeric_columns = data.select_dtypes(include=['int64', 'float64'])
    num_columns = numeric_columns.columns

    # Create subplots layout
    num_subplots = len(num_columns)
    subplots_per_row = 3
    num_rows = (num_subplots - 1) // subplots_per_row + 1

    # Create subplots and display histograms with KDEs
    fig, axes = plt.subplots(num_rows, subplots_per_row, figsize=(15,

    # Iterate creating subplots
    for i, column in enumerate(num_columns):
        row_idx = i // subplots_per_row
        col_idx = i % subplots_per_row
        ax = axes[row_idx, col_idx]

        # Plot histograms with kernel density estimate
        sns.histplot(data[column], bins=20, kde=True, edgecolor='black
        ax.set_title(f'Histogram with KDE of {column}')
        ax.set_xlabel(column)
        ax.set_ylabel('Density')


    # Display plots
    plt.tight_layout()
    plt.show()


plot_numeric_columns(data)
```



The univariate analysis of the numeric features in our dataset shows histograms with Kernel Density Estimation (KDE) curves that provide valuable insights into the underlying distributions of the numeric variables. Majority of the numeric features appear to follow a normal distribution, indicating a symmetric and bell-shaped pattern. This suggests that these variables can be easily interpreted and modeled.

However, a few features stand out as exceptions to this general trend. The 'area_code' variable exhibits a distribution that is concentrated around the lower end of the scale with most subscribers coming from area codes between 0 and 420. This could be an indication of a geographical bias in our customer base.

The 'number_vmail_messages' feature also displays a unique distribution with a large concentration of customers having close to 0 voicemail messages and a small portion of outliers sending over 2,000 voicemails. This suggests that the voicemail usage among the customers is highly skewed with a few heavy users driving the overall distribution.

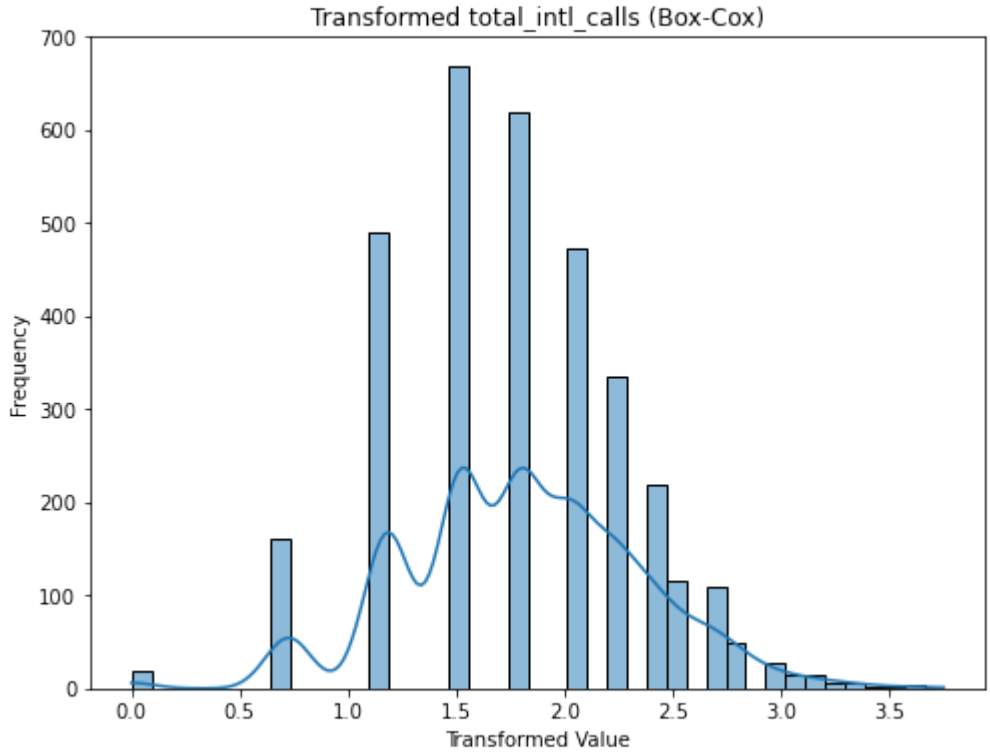
The 'total_intl_calls' variable exhibits a right-skewed distribution indicating that majority of customers make relatively few international calls while a small subset of customers account for a disproportionately high number of international calls of up to around 600 calls.

Lastly, the 'customer_service_calls' feature also demonstrates a right-skewed distribution with most customers making only 1 customer service call per day and a small number of customers making significantly more calls.

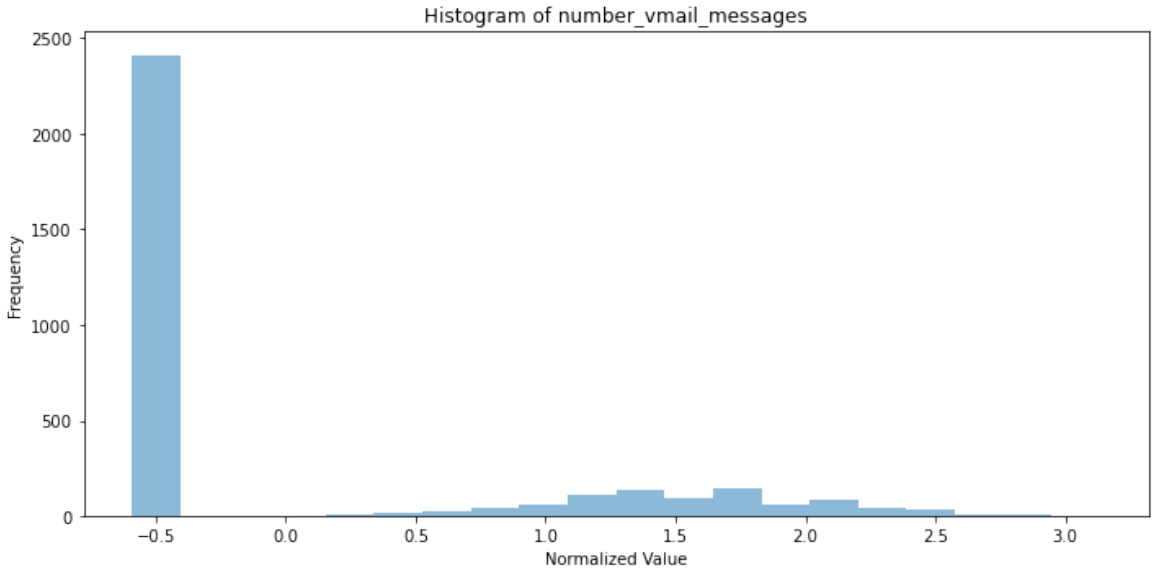
To deal with the negative skewness of the 'total_intl_calls' variable, we will apply a box-cox transformation since Box-Cox is a family of power transformations that includes the square root and log transformations as special cases. It can handle both positive and negative skewness by adjusting the lambda parameter and is designed to find the best transformation to approximate normality. It can effectively handle moderate skewness.

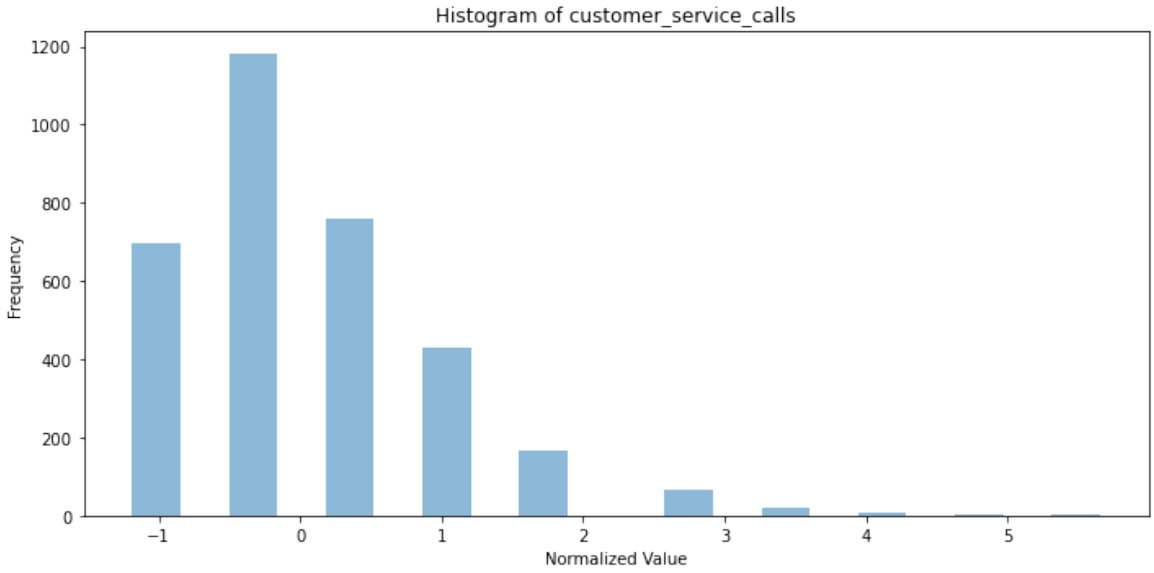
In [8]:
```
from scipy.stats import boxcox

class DataProcessor:
    def __init__(self, data):
        self.data = data
    '''
    Class to perform box-cox transformation and normalize data in the

    Attributes:
        data (pd.DataFrame): The input DataFrame containing numeric co

    Returns:
        None
    '''

    def boxcox_transform(self, column_name):
        '''
        Transform a skewed column in the DataFrame using Box-Cox trans

        Args:
            column_name (str): The name of the column to transform.

        Returns:
            pandas.DataFrame: The DataFrame with the transformed colum
        '''
        # Create a copy of the DataFrame to avoid modifying the origir
        transformed_data = self.data.copy()

        # Apply Box-Cox transformation
        # Adding 1 to handle zero values
        transformed_data[column_name], _ = boxcox(self.data[column_nam

        return transformed_data

    def normalize_variables(self, columns):
        '''
        Normalize numerical variables in the DataFrame using StandardS

        Args:
            columns (list): A list of column names to normalize.

        Returns:
            pandas.DataFrame: The DataFrame with the normalized columr
        '''
        # Create a copy of the DataFrame to avoid modifying the origir
        normalized_data = self.data.copy()

        # Initialize StandardScaler
        scaler = StandardScaler()

        # Normalize specified columns
        normalized_data[columns] = scaler.fit_transform(self.data[colu

        return normalized_data

    def plot_transformed_variable(self, column_name):
        '''
        Plot the histogram of a transformed variable after applying Bo

        Args:
            column_name (str): The name of the transformed column.
```

```
        Returns:
            None
        '''
        # Apply Box-Cox transformation
        transformed_data = self.boxcox_transform(column_name)

        # Plot histogram of the transformed variable
        plt.figure(figsize=(8, 6))
        sns.histplot(transformed_data[column_name], kde=True)
        plt.title(f'Transformed {column_name} (Box-Cox)')
        plt.xlabel('Transformed Value')
        plt.ylabel('Frequency')
        plt.show()
        # Display skewness
        print(f'Skewness = {transformed_data[column_name].skew()}')

    def plot_normalized_columns(self, columns):
        '''
        Plot histograms of normalized columns on separate axes.

        Args:
            columns (list): A list of column names to plot.

        Returns:
            None
        '''
        # Normalize specified columns
        normalized_data = self.normalize_variables(columns)

        # Create subplots
        fig, axes = plt.subplots(len(columns), 1, figsize=(10, len(col

        # Plot histograms of normalized columns on separate axes
        for i, column in enumerate(columns):
            axes[i].hist(normalized_data[column], bins=20, alpha=0.5)
            axes[i].set_title(f'Histogram of {column}')
            axes[i].set_xlabel('Normalized Value')
            axes[i].set_ylabel('Frequency')

        # Display transformed plots
        plt.tight_layout()
        plt.show()

# Instantiate
data_processor = DataProcessor(data)

# Plot transformed variable
data_processor.plot_transformed_variable('total_intl_calls')

# Plot histograms of the normalized columns
data_processor.plot_normalized_columns(['number_vmail_messages', 'cust
```




```
Skewness = 0.005816369249351684
```

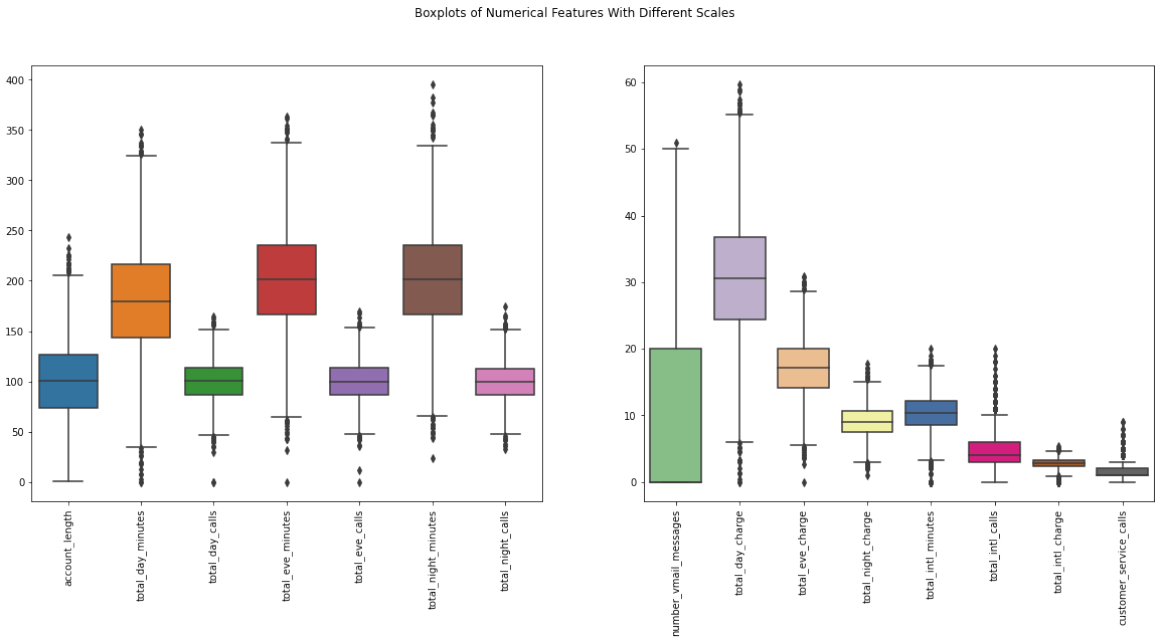




The 'total_intl_calls' plot now appears to have a skewness of 0 after the box-cox transformation while number_vmail_messages and customer_service_calls have been normalized.

The next step was to identify outliers in the numeric data. Outliers can greatly impact some models so it is important to identify them early and decide on the next course of action before the modelling process. From the histograms above we can note that the some columns have larger values than other putting them on different scales. For that reason, we are going to plot two sets of box plots to show the outliers as per their respective scales.

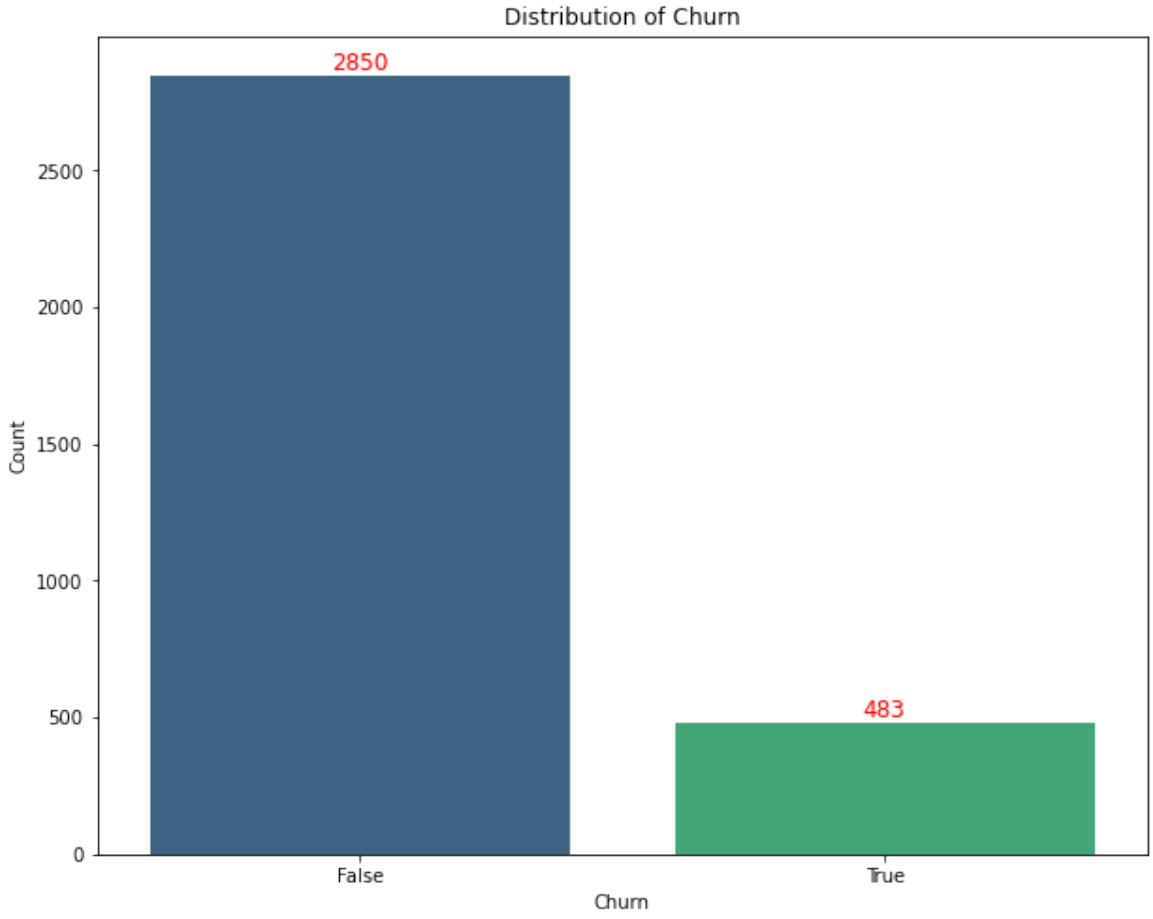
In [9]:
```
def plot_grouped_boxplots(data):

    '''
    Generates grouped boxplots for two sets of numerical features in t

    Args:
        data : pd.DataFrame
        The DataFrame containing the features to be plotted.

    Returns:
        None
    '''

    # Identify columns for the first group of boxplots
    cols_x = ['account_length', 'total_day_minutes', 'total_day_calls'
              'total_eve_minutes', 'total_eve_calls', 'total_night_mir

    # Identify columns for the second group of boxplots
    cols_y = ['number_vmail_messages', 'total_day_charge', 'total_eve_
              'total_intl_minutes', 'total_intl_calls', 'total_intl_ch

    # Initialize a figure with a single row and two columns for the su
    fig, axes = plt.subplots(1, 2, figsize=(20, 8))

    # Generate a boxplot for the first group of columns on the first s
    sns.boxplot(data=data[cols_x], ax=axes[0], palette='tab10')
    axes[0].set_xticklabels(axes[0].get_xticklabels(), rotation=90)

    # Generate a boxplot for the second group of columns on the second
    sns.boxplot(data=data[cols_y], ax=axes[1], palette='Accent')
    axes[1].set_xticklabels(axes[1].get_xticklabels(), rotation=90)

    # Set the overall title for the figure
    fig.suptitle('Boxplots of Numerical Features With Different Scales

    # Display the plot
    plt.show()

plot_grouped_boxplots(data)
```



From the boxplots we can clearly see that the data contained many outliers especially in the 'total_night_minutes', 'total_intl_calls' and 'customer_service_calls' columns. Outliers in certain features may be indicative of unusual but valid customer behavior or market-specific conditions therefore I decided to leave them as is and see how they would affect the outcome of our models.

Next I went on to analyze the distribution of the categorical variables starting with the target variable 'Churn' so as to assess the customer attrition situation.

In [10]:
```
def plot_churn_distribution(data):
    '''
    Plots the distribution of the 'churn' column.

    Args:
        data (pd.DataFrame): The input DataFrame.

    Returns:
        None
    '''
    # Set size of the plot
    plt.figure(figsize=(10, 8))
    sns.countplot(data=data, x='churn', palette='viridis')
    counts = data['churn'].value_counts()

    # Iterate to add count values on top of the bars
    for i, v in enumerate(counts):
        plt.text(i, v, str(v), color='red', ha='center', va='bottom',

    plt.title('Distribution of Churn')
    plt.xlabel('Churn')
    plt.ylabel('Count')
    plt.show()

plot_churn_distribution(data)
```



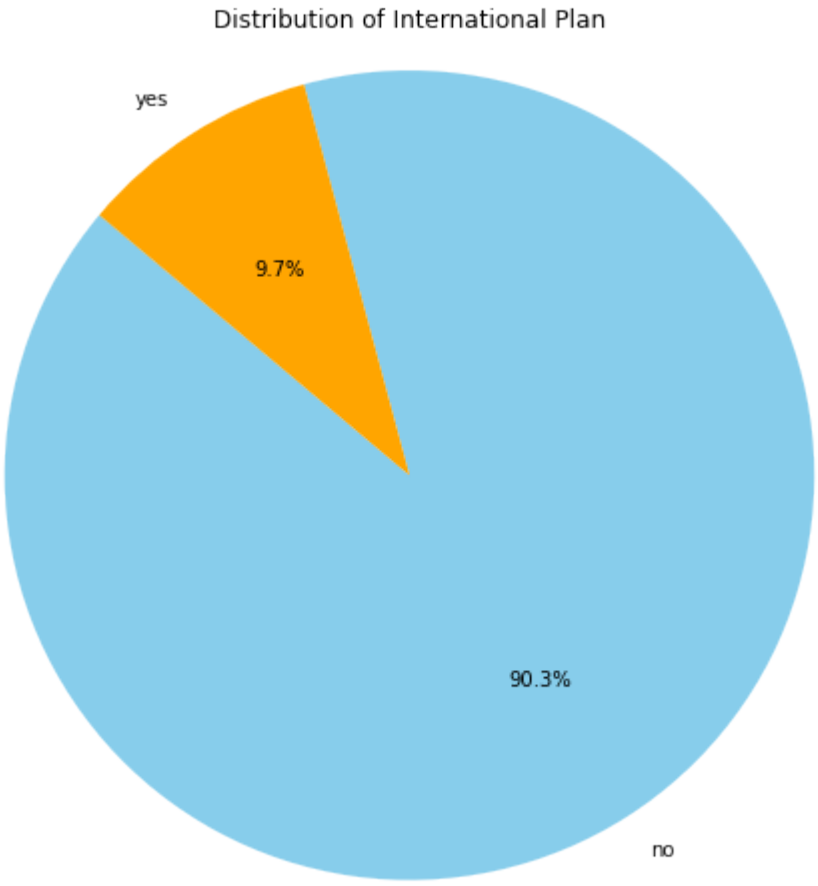
Of the 3,333 customers, 2850 remained at SyriaTel, representing about 85.51% of their clientele. 483 terminated their contract which represents approximately 14.49% of the total customers. This imbalance in the binary classes highlights a potential issue that needs to be addressed before modeling in order to avoid bias and inaccurate predictions.

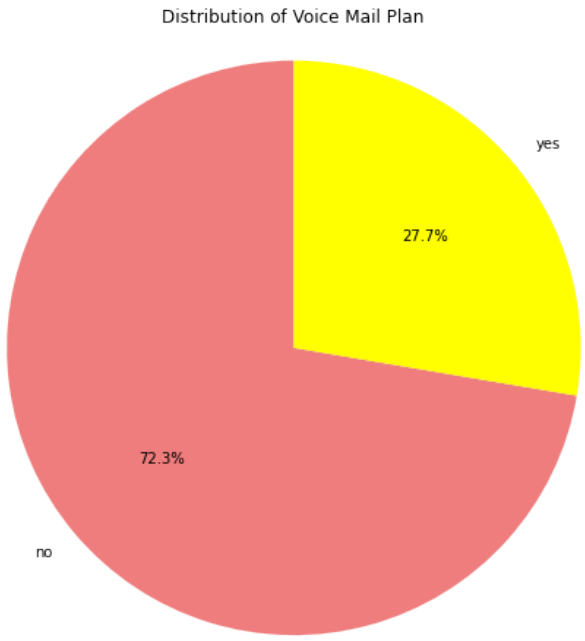
Next we are going to analyse the distribution of the other categorical columns namely 'state', 'international plan' and 'voice mail plan' by creating a CategoricalDistributionPlotter class that will enable us to plot visualizations to show their different distributions. The 'Phone number' column will, however, not be represented as it doesnt contain binary values.

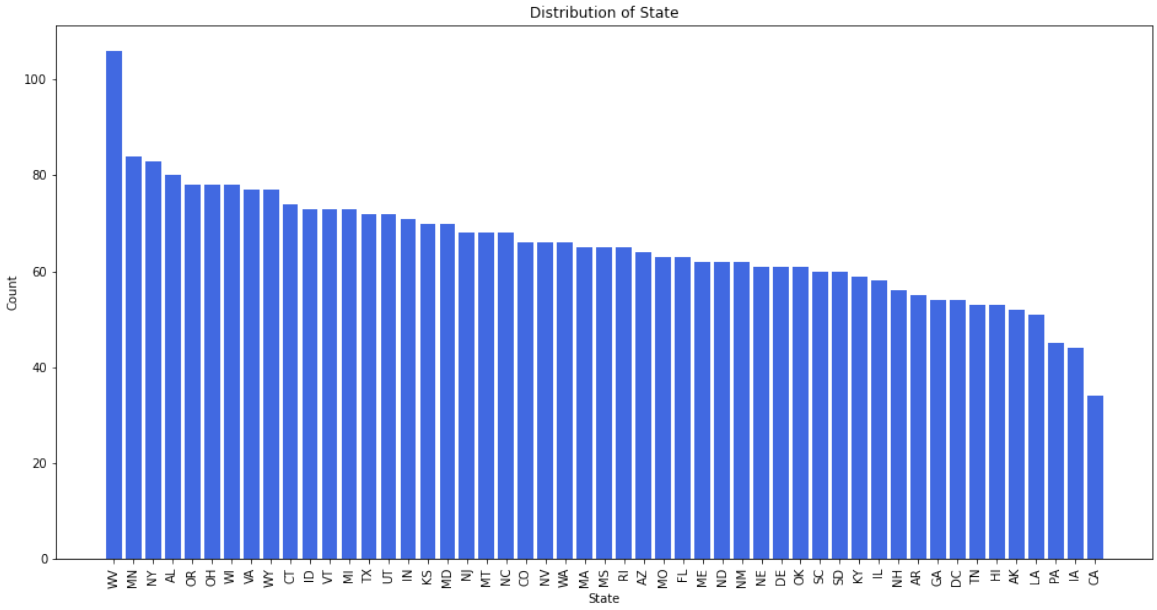
In [11]:
```
class CategoricalDistributionPlotter:
    '''
    Class used to perform univariate data analysis on categorical feat

    Attributes:
        data (pd.DataFrame): The input DataFrame containing numeric co

    Returns:
        None
    '''

    def __init__(self, data):
        self.data = data

    def plot_international_plan_distribution(self):
        '''
        Plots the distribution of the 'international plan' categorical

        Args:
            None

        Returns:
            None
        '''
        # Get the counts of each category
        feature_counts = self.data['international_plan'].value_counts(

        # Create labels and sizes for the pie chart
        labels = feature_counts.index
        sizes = feature_counts.values

        # Plot the pie chart
        plt.figure(figsize=(8, 8))
        plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=14
        plt.title('Distribution of International Plan')
        # Ensure that pie is drawn as a circle.
        plt.axis('equal')
        plt.show()

    def plot_voice_mail_plan_distribution(self):
        '''
        Plots the distribution of the 'voice mail plan' categorical va

        Args:
            None

        Returns:
            None
        '''
        # Get the counts of each category
        feature_counts = self.data['voice_mail_plan'].value_counts()

        # Create labels and sizes for the pie chart
        labels = feature_counts.index
        sizes = feature_counts.values

        # Plot the pie chart
        plt.figure(figsize=(14, 8))
        plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90
        plt.title('Distribution of Voice Mail Plan')
        plt.axis('equal')
```

```
        plt.show()

    def plot_state_distribution(self):
        '''
        Plots the distribution of the 'state' categorical variable.

        Args:
            None

        Returns:
            None
        '''
        feature_counts = self.data['state'].value_counts().reset_index
        feature_counts.columns = ['state', 'Count']
        plt.figure(figsize=(16, 8))

        # Plot the bar chart
        plt.bar(feature_counts['state'], feature_counts['Count'], colo
        plt.xticks(feature_counts['state'], rotation=90)
        plt.title('Distribution of State')
        plt.xlabel('State')
        plt.ylabel('Count')
        plt.show()
```

In [12]:
```
# Instantiate
plotter = CategoricalDistributionPlotter(data)
plotter.plot_international_plan_distribution()
plotter.plot_voice_mail_plan_distribution()
plotter.plot_state_distribution()
```





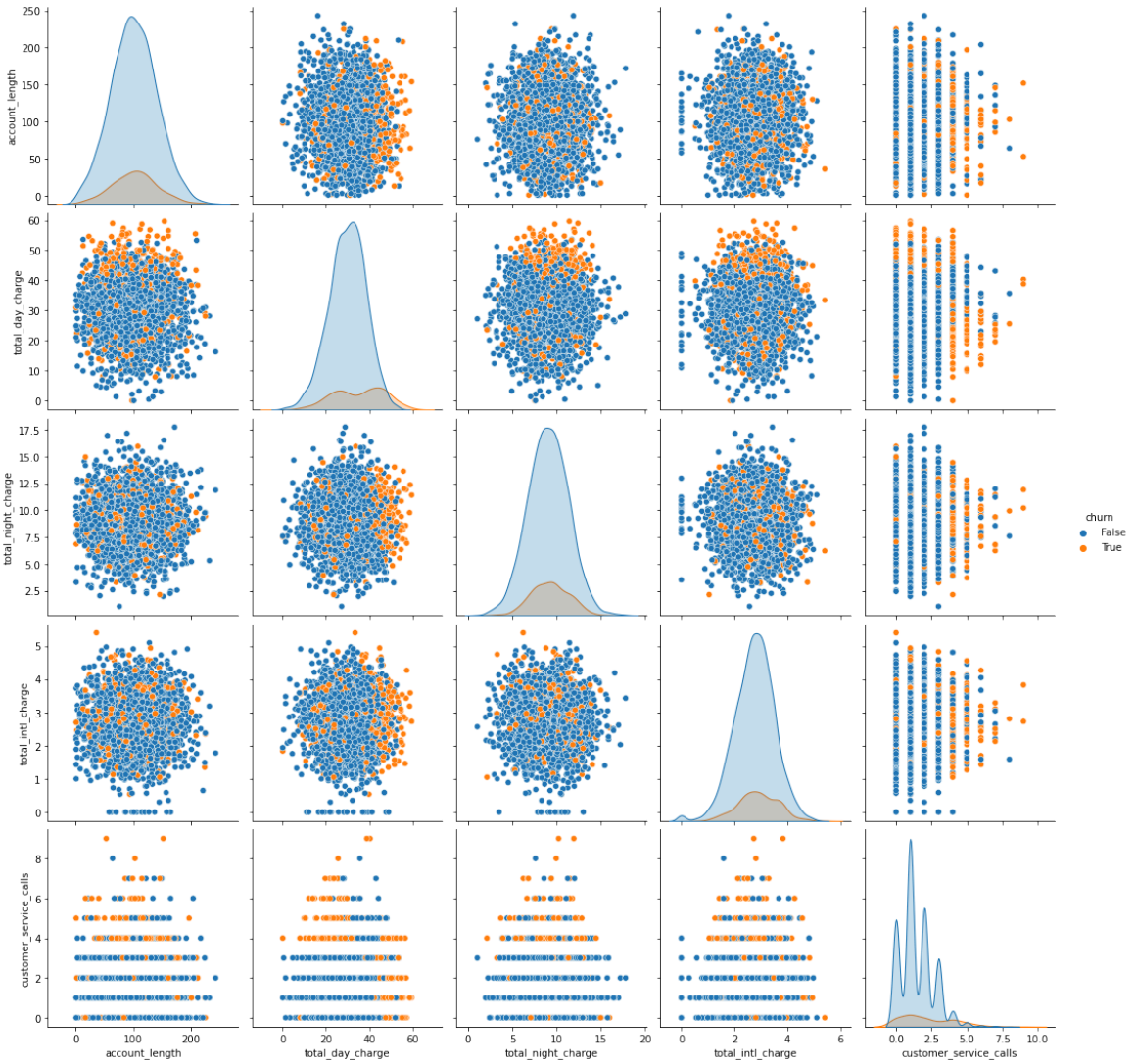


From the first pie chart we can infer that 90% of SyriaTel's clientelle does not have an international phone plan while only 9.7% do. The question we can ask ourselves is whether having an international plan or lack thereof is directly correlated with customer churn. The same can be asked concerning customers who have a voicemail plan who make up 27.7% of SyriaTel's customers while 72.3% do not have a voicemail plan. Lastly, we can see that the highest number of the customers are from the State of West Virginia followed by Minnesota, NewYork and Alabama. The lowest number of customers are from the State of Carlifornia, Iowa and Pennsylvania respectively.

Bivariate Analysis

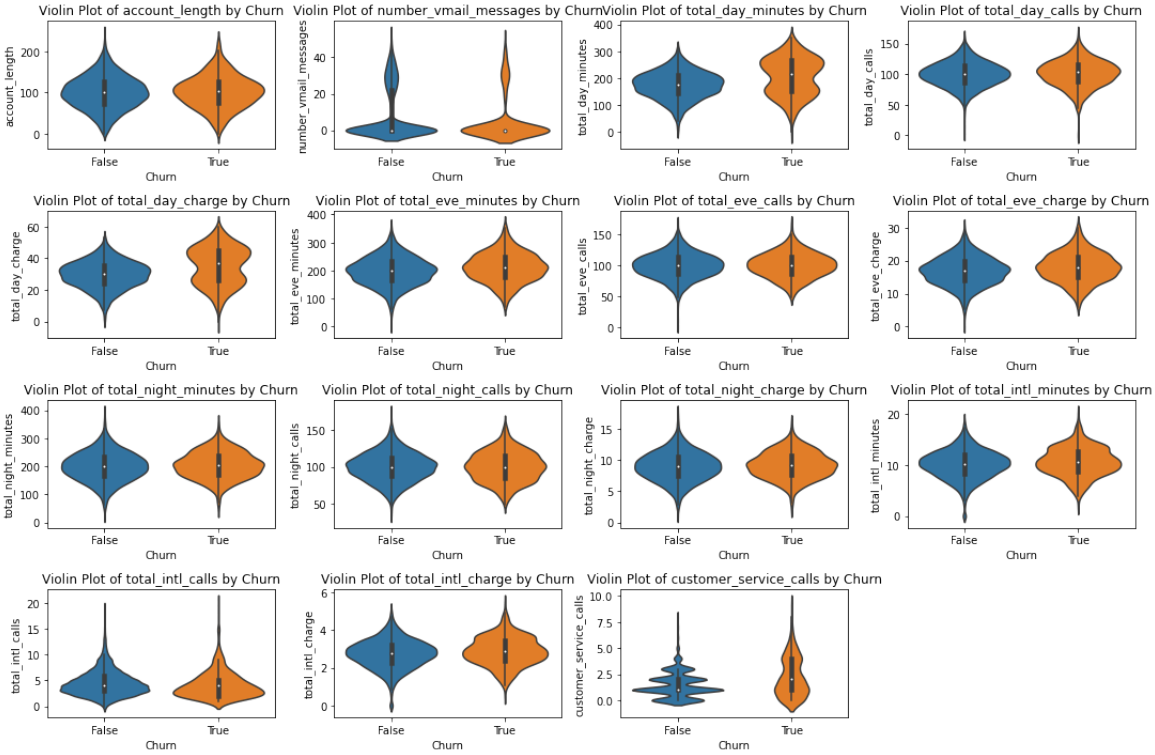
Bivariate analysis involves exploring the relationships between two variables. The first step is to analyse the relationship of some of the numeric variables with relation to 'churn'. I decided to analyse the following features: 'account_length', 'total_day_charge', 'total_night_charge', 'total_intl_charge' and 'customer_service_calls' to see how they correlated with churn.

In [13]:
```
#plot pairplots for numeric variables
num_cols = data[['account_length', 'total_day_charge', 'total_night_ch
                 'customer_service_calls', 'churn']]
sns.pairplot(num_cols, hue='churn',height=3.0);
plt.show();
```



From the pairplot above it is clear that as the customer service calls increase so does the customer churn rate. The same is true for the total night charge, the total day charge and the total international charge. We can also see that the longer a customer has had an account the lower occurence of attrition. Below we plot violin plots to further visualise the distribution of the numeric variables within our dataset.

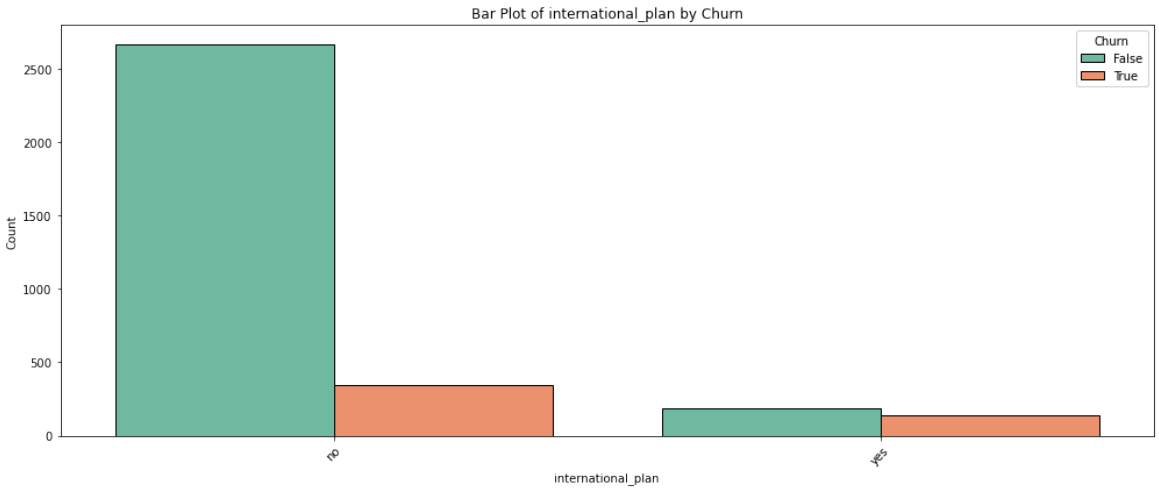
In [14]:
```
# List numeriic variables
numeric_variables = ['account_length', 'number_vmail_messages', 'total
                     'total_eve_minutes', 'total_eve_calls', 'total_ev
                     'total_night_charge', 'total_intl_minutes', 'tota

# Create violin plots for each numeric variable
plt.figure(figsize=(15, 10))
for i, variable in enumerate(numeric_variables):
    plt.subplot(4, 4, i+1)
    sns.violinplot(x='churn', y=variable, data=data)
    plt.title(f'Violin Plot of {variable} by Churn')
    plt.xlabel('Churn')
    plt.ylabel(variable)
# Diisplay plots
plt.tight_layout()
plt.show()
```



The violin plots reveal that account length, evening usage, night-time usage and international usage are not significant factors in predicting customer churn. However, customers who do not churn tend to have a higher number of voicemail messages suggesting that frequent use of voicemail services may reduce churn. There is a noticeable increase in total day minutes, calls and charges for customers who churn indicating that high day-time usage could be linked to churn. Most importantly, customers who churn tend to make more customer service calls, suggesting that high interaction with customer service, possibly due to issues or dissatisfaction, is strongly associated with a higher likelihood of churn.

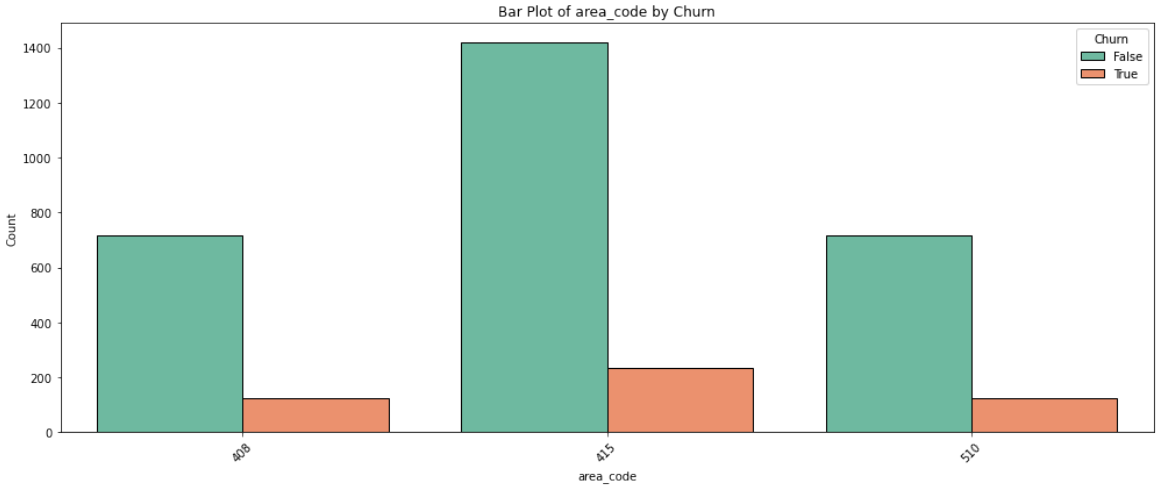
Here we defined a function that would enable us to see the relationship between churn and the categorical variables in the dataset.

In [15]:
```
def analyze_categorical_variable(variable, data):
    '''
    Create bar plots for the categorical variables.

    Args:
        variable (str): The name of the categorical variable.
        data (pd.DataFrame): The DataFrame containing the data.

    Returns:
        None
    '''
    # Create stacked bar plot
    plt.figure(figsize=(14, 6))
    sns.countplot(x=variable, hue='churn', data=data, palette='Set2',
    plt.title(f'Bar Plot of {variable} by Churn')
    plt.xlabel(variable)
    plt.ylabel('Count')
    plt.legend(title='Churn', loc='upper right')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

analyze_categorical_variable('international_plan', data)
```



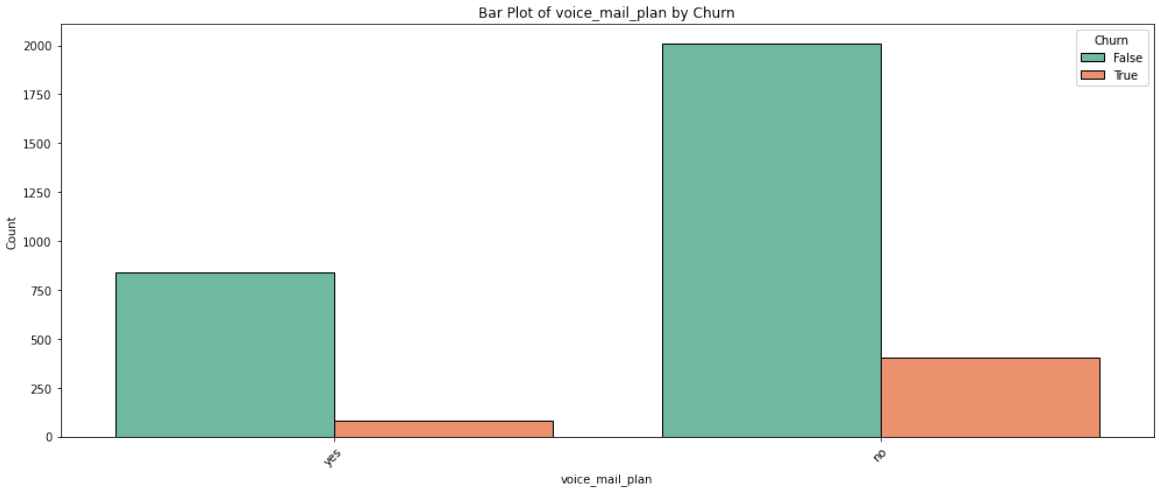
A large percentage of customers without an international plan stuck with their service provider, SyriaTel as compared to those that left. In comparison, the number of people that left SyriaTel and had an international plan was almost equivalent to those that stayed.

In [16]:
```
analyze_categorical_variable('area_code', data)
```



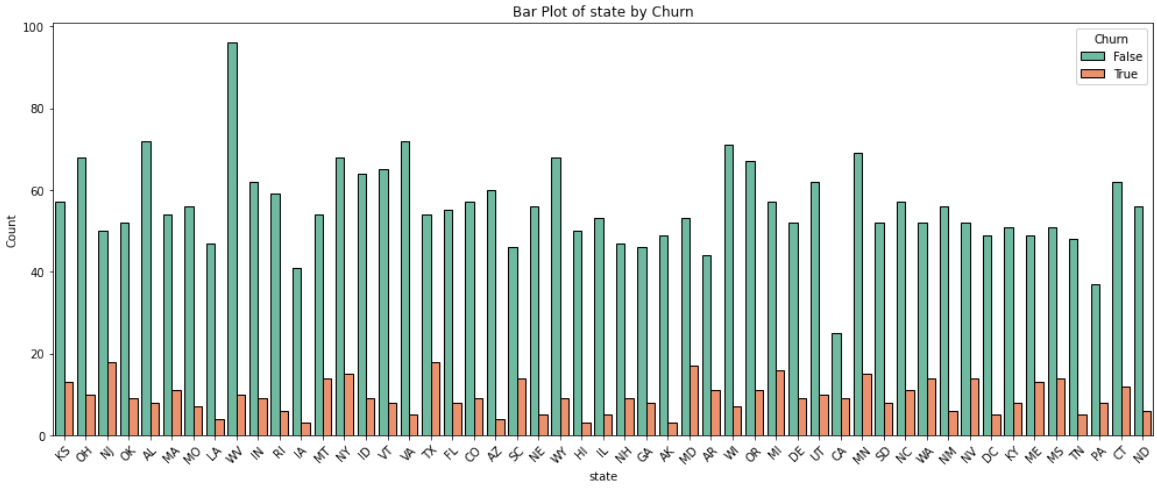
Here we can see that area code 415 experienced the highest churn rate and the highest customer retention rate. This might be due to a number of factors such as higher population or better network infrastructure. Area code 408 and 510 show almost similar trends in terms of customer attrition.

In [17]:
```
analyze_categorical_variable('voice_mail_plan', data)
```



We can see that a higher number of customers without a voicemail plan stayed with SyriaTel but at the same time they make up the larger proportion of those that ended their SyriaTel subscription.

In [18]:
```
analyze_categorical_variable('state', data)
```

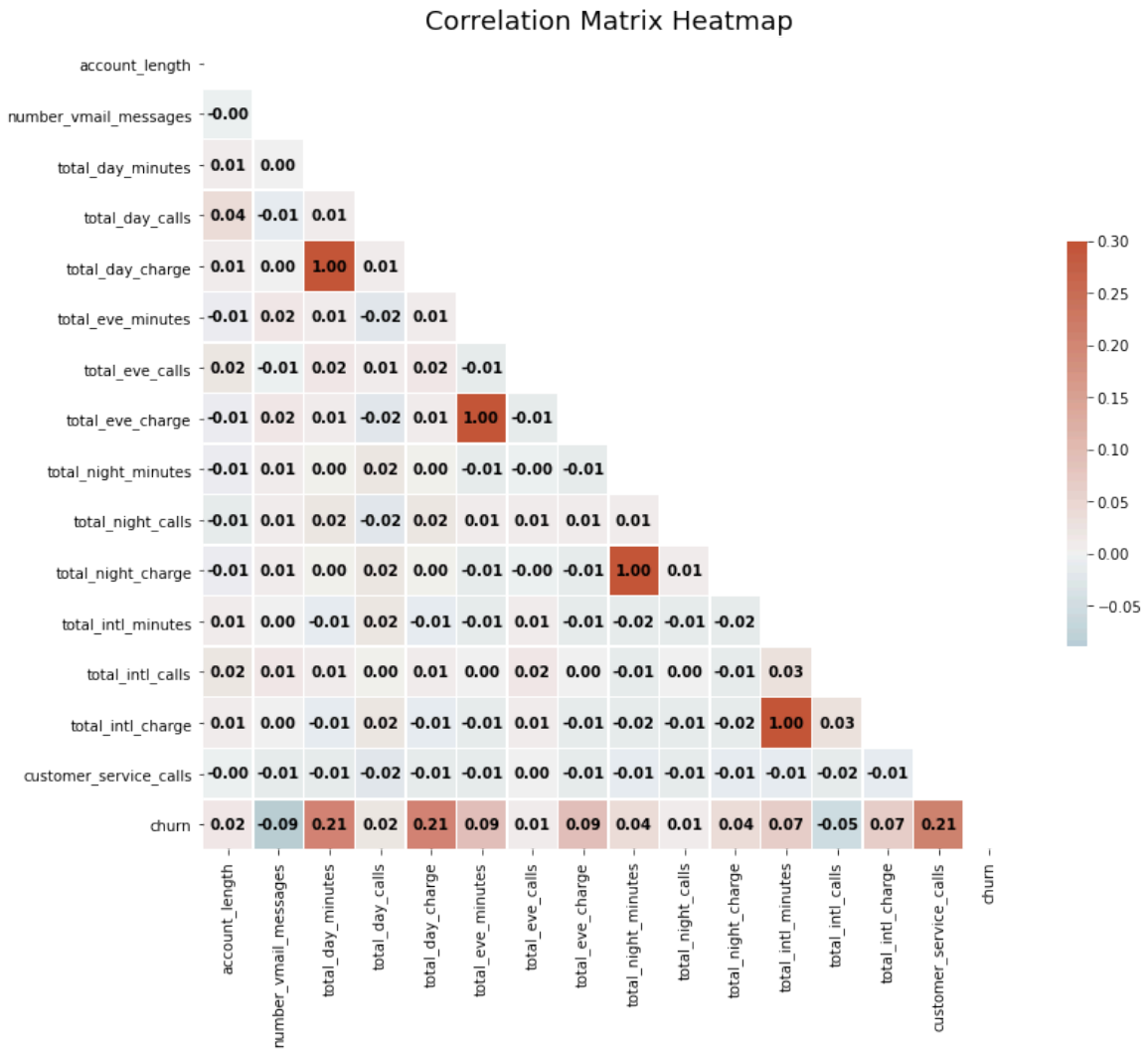


From the plot above we can see that the states of New Jersey, Texas,South Carolina, maryland, Michigan and Minnesota had the highest customer attrition whereas West Virginia, Alabama, Ohio, Virginia, Wisconsin and Minnesota had the lowest churn rates.

Multivariate Analysis

Multivariate analysis involves analyzing data with multiple variables simultaneously. It helps uncover relationships, patterns and interactions among those variables providing valuable insights for the modelling stage. We decided to go with a correlation heatmap that would show the features with the highest correlation.

In [19]:
```
def plot_correlation_heatmap(data):
    '''
    Plot a correlation matrix heatmap.

    Args:
        data (pandas.DataFrame): The DataFrame containing the data.

    Returns:
        None
    '''
    # Correlation matrix
    corr_matrix = data.corr()

    # Create a mask for upper triangle
    mask = np.triu(np.ones_like(corr_matrix, dtype=bool))

    # Set up the matplotlib figure
    plt.figure(figsize=(12, 10))

    # Generate custom diverging colormap
    cmap = sns.diverging_palette(220, 20, as_cmap=True)

    # Plot the heatmap
    sns.heatmap(corr_matrix, mask=mask, cmap=cmap, vmax=.3, center=0,
                square=True, linewidths=.5, cbar_kws={"shrink": .5},
                annot=True, fmt='.2f', annot_kws={"size": 10, "weight'
    plt.title('Correlation Matrix Heatmap', fontsize=18)
    plt.tight_layout()
    plt.show()

plot_correlation_heatmap(data)
```



The correalation heatmap shows a lot of very weak positive and negative correlations between the numeric variables in our dataset.

We can, however, take note of the perfect positive correlations of 1.0 between 'total_day_charge' and 'total_day_minutes', 'total_eve_charge' and 'total_eve_minutes', 'total_night_charge' and 'total_night_minutes' and 'total_intl_charge' and 'total_intl_minutes'.

It can be infered that they are perfectly correlated because the higher the call minutes, the higher the call charges and vice versa.

Feature Engineering

Checking for multicollinearity

We will conduct correlation analysis by computing the absolute correlation values between all pairs of variables in the DataFrame and sort them in descending order. The aim is to filter the data to show correlations greater than 0.75 and less than 1 which indicate strong positive correlations. Highly correlated features will then be paired and one feature will eventually be dropped from each pair which will aid in tackling multicolllinearity in our data.

In [20]:
```
# Compute absolute correlation between variable pairs
corr_data=data.corr().abs().stack().reset_index().sort_values(0, ascer

# Zip the feature names columns
corr_data['correlated_features'] = list(zip(corr_data.level_0, corr_da

# Set index
corr_data.set_index(['correlated_features'], inplace = True)

# Drop temporary columns
corr_data.drop(columns=['level_1', 'level_0'], inplace = True)

# Name correlation column
corr_data.columns = ['correlation']

# Filter for correlations >0.75 and <1
corr_data[(corr_data.correlation>.75) & (corr_data.correlation <1)]
```

Out[20]:

correlated_features	correlation
(total_day_charge, total_day_minutes)	1.000000
(total_day_minutes, total_day_charge)	1.000000
(total_eve_charge, total_eve_minutes)	1.000000
(total_eve_minutes, total_eve_charge)	1.000000
(total_night_minutes, total_night_charge)	0.999999
(total_night_charge, total_night_minutes)	0.999999
(total_intl_charge, total_intl_minutes)	0.999993
(total_intl_minutes, total_intl_charge)	0.999993

We can see that there is high multicollinearity between most of the features in our dataset. Multicollinearity in regression models results in inflated standard errors, unstable coefficients, reduced interpretability and an increased risk of overfitting. Highly correlated predictors hinder disentangling of their individual effects on the response variable leading to less precise coefficient estimates and wider confidence intervals.

In [21]:
```
import pandas as pd

def calculate_correlation_pairs(data):
    '''
    Calculate correlation pairs and filter based on a threshold.

    Args:
        data: pandas DataFrame : The input data containing correlation

    Returns:
        pandas DataFrame : A DataFrame with pairs of correlated featur
    '''

    # Calculate absolute correlation coefficients
    corr_matrix = data.corr().abs()

    # Stack the correlation matrix to create pairs
    stacked_corr = corr_matrix.stack().reset_index()

    # Rename columns
    stacked_corr.columns = ['feature_1', 'feature_2', 'correlated_feat

    # Filter based on a threshold (ie. > 0.75)
    filtered_corr = stacked_corr[(stacked_corr.correlated_features > 0

    return filtered_corr


correlation_pairs = calculate_correlation_pairs(data)
correlation_pairs
```

Out[21]:

	feature_1	feature_2	correlated_features
36	total_day_minutes	total_day_charge	1.000000
66	total_day_charge	total_day_minutes	1.000000
87	total_eve_minutes	total_eve_charge	1.000000
117	total_eve_charge	total_eve_minutes	1.000000
138	total_night_minutes	total_night_charge	0.999999
168	total_night_charge	total_night_minutes	0.999999
189	total_intl_minutes	total_intl_charge	0.999993
219	total_intl_charge	total_intl_minutes	0.999993

In [22]:
```
def drop_correlated_features(data, threshold=0.9):
    '''
    Drop one feature from each pair of highly correlated features in t

    Args:
        data (pandas.DataFrame): The input data as a DataFrame.
        threshold (float): The correlation threshold above which featu

    Returns:
        pandas.DataFrame: The DataFrame with one feature dropped from
    '''

    # Calculate absolute correlation matrix
    corr_matrix = data.corr().abs()
    # Select upper triangle of correlation matrix
    upper_tri = corr_matrix.where(np.triu(np.ones(corr_matrix.shape),
    # Find features with correlation greater than the threshold
    to_drop = [column for column in upper_tri.columns if any(upper_tri
    # Drop one feature from each highly correlated pair
    to_drop_pairs = [
        ('total_day_charge', 'total_day_minutes'),
        ('total_eve_charge', 'total_eve_minutes'),
        ('total_night_charge', 'total_night_minutes'),
        ('total_intl_charge', 'total_intl_minutes')
    ]

    # Iterate over each pair in to_drop_pairs list
    for pair in to_drop_pairs:
        # Check if both elements of the pair are in to_drop list
        if pair[0] in to_drop and pair[1] in to_drop:
            # Drop the second element from to_drop list
            to_drop.remove(pair[1])# Iterate over each pair in to_drop

    data = data.drop(to_drop, axis=1)
    return data

# Drop one feature from each highly correlated pair
cleaned_data = drop_correlated_features(data)
cleaned_data.head()
```

Out[22]:

	state	account_length	area_code	phone_number	international_plan	voice_mail_plan	numbe
0	KS	128	415	382-4657	no	yes	
1	OH	107	415	371-7191	no	yes	
2	NJ	137	415	358-1921	no	no	
3	OH	84	408	375-9999	yes	no	
4	OK	75	415	330-6626	yes	no	

Dropping the phone number feature as it has no direct significance on customer attrition as it is a personal attribute or identifier.

In [23]:
```
cleaned_data = cleaned_data.drop('phone_number', axis=1)
cleaned_data
```

Out[23]:

	state	account_length	area_code	international_plan	voice_mail_plan	number_vmail_mess
0	KS	128	415	no	yes	
1	OH	107	415	no	yes	
2	NJ	137	415	no	no	
3	OH	84	408	yes	no	
4	OK	75	415	yes	no	
...	...	...	...	...	...	
3328	AZ	192	415	no	yes	
3329	WV	68	415	no	no	
3330	RI	28	510	no	no	
3331	CT	184	510	yes	no	
3332	TN	74	415	no	yes	

3333 rows × 16 columns

We then proceed to label encode and one hot encode our data. It is key to note that LabelEncoder is suitable for ordinal data which is categorical variables with clear ordering such as 'yes' and 'no', while OneHotEncoder is more appropriate for nominal categorical data which is data without a natural ordering such as the values in the state column.

In [24]:
```
def preprocess_data(data):
    '''
    Preprocesses the input DataFrame by converting columns with 'yes'
    and one-hot encodes the 'state' column.

    Parameters:
    - data: DataFrame to be preprocessed.

    Returns:
    - preprocessed_data: Preprocessed DataFrame.
    '''

    # Convert columns with 'yes' or 'no' to binary using LabelEncoder
    label_encoder = LabelEncoder()
    cols_to_encode = ['churn', 'international_plan', 'voice_mail_plan'
    data[cols_to_encode] = data[cols_to_encode].apply(label_encoder.fi

    # One-hot encode the 'state' column
    preprocessed_data = pd.get_dummies(data, columns=['state'])

    return preprocessed_data

cleaned_data = preprocess_data(cleaned_data)
cleaned_data.head()
```

Out[24]:

	account_length	area_code	international_plan	voice_mail_plan	number_vmail_messages	tot
0	128	415	0	1	25	
1	107	415	0	1	26	
2	137	415	0	0	0	
3	84	408	1	0	0	
4	75	415	1	0	0	

5 rows × 66 columns

Our data is now ready for modelling.

MODELLING

In order to identify the most effective machine learning model and its optimal parameters for our classification task, we will adopt a comprehensive approach. We will construct and assess the performance of multiple models namely:

- **Logistic Regression**
- **K-Nearest Neighbors**
- **Decision Trees**
- **Random Forest**

We will then employ evaluation techniques such as gridsearch cross-validation and performance metrics such as accuracy, precision, recall and f1 score, confusion matrix and ROC-AUC curves to objectively compare the predictive capabilities of these models.

Once we have a baseline understanding of each models' performance, we shall then engage in hyperparameter tuning. Hyperparameters are the configurable settings that govern the behavior and complexity of machine learning algorithms. We will then be able to fine-tune the models to maximize their predictive accuracy while mitigating issues like overfitting or

Logistic Regression

A logistic regression model is a statistical method used for binary classification, which predicts the probability of an event occurring based on input features. In this case, we aim to predict customer churn within SyriaTel Company with the aim of improving customer retention.

- NB: Class 0 = Not churn

  Class 1 = Churn

In [25]:
```
# Define X and y
X = cleaned_data.drop('churn', axis=1)
y = cleaned_data['churn']

# Split data into 75% train and 25% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.

# Initialize the scaler and fit on the training data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize SMOTE
smote = SMOTE(random_state=20)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_scal

# Initialize the logistic regression model
logreg = LogisticRegression()

# Fit the model on the resampled training data
logreg.fit(X_train_resampled, y_train_resampled)

# Predict churn for the train and test data
y_train_pred = logreg.predict(X_train_resampled)
y_test_pred = logreg.predict(X_test_scaled)
```

Checking to ensure that class imbalance has been resolved by SMOTE

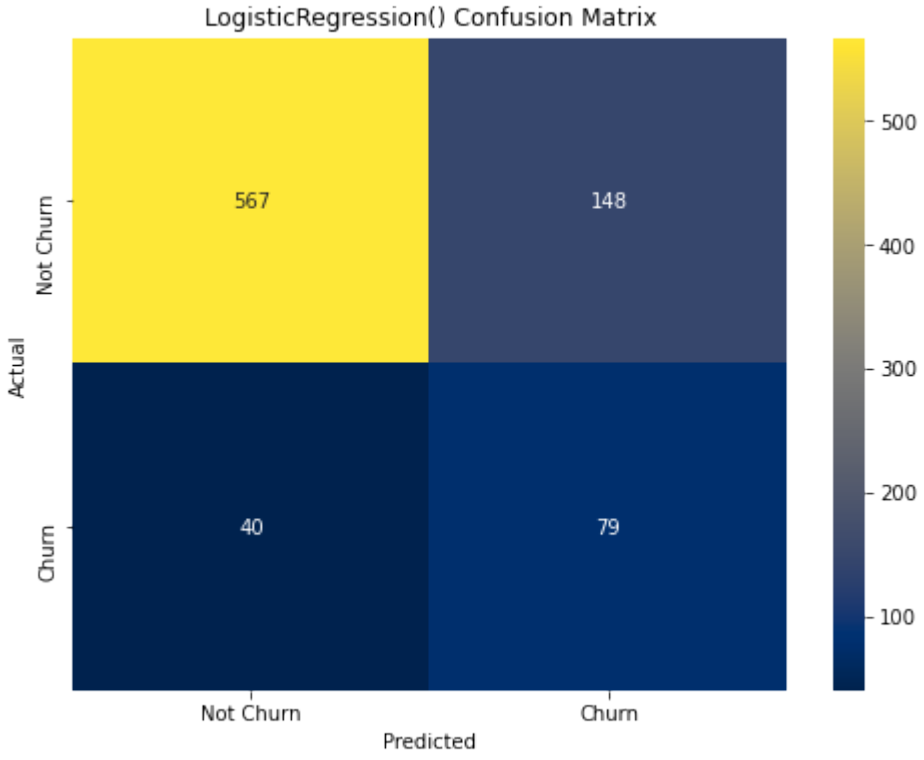
In [26]:
```
y_train_resampled.value_counts()
```

Out[26]:
```
1    2135
0    2135
Name: churn, dtype: int64
```

SMOTE worked perfectly, now to evaluate our model metrics and plot the relevant visualizations.

In [27]:
```
def evaluate_model(model, X_train, y_train, X_test, y_test):
    # Predict churn for the train and test data
    y_train_pred = model.predict(X_train_resampled)
    y_test_pred = model.predict(X_test_scaled)

    # Calculate accuracy of the model for train and test data
    train_accuracy = accuracy_score(y_train_resampled, y_train_pred)
    test_accuracy = accuracy_score(y_test, y_test_pred)

    # Print the train and test scores
    print(f'Train Accuracy: {train_accuracy:.2f}')
    print(f'Test Accuracy: {test_accuracy:.2f}')
    print('-----------------------------------\n')

    # Print the classification report for test data
    print('Classification Report (Test Data):')
    print(classification_report(y_test, y_test_pred))
    print('-----------------------------------\n')

    # Print the confusion matrix for test data
    print('Confusion Matrix (Test Data):')
    print(confusion_matrix(y_test, y_test_pred))
    print('-----------------------------------\n')

    # Print the model scores
    print(f'Train Score: {model.score(X_train_resampled, y_train_resam
    print(f'Test Score: {model.score(X_test_scaled, y_test):.2f}')

    # Calculate confusion matrix
    cm = confusion_matrix(y_test, y_test_pred)

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='cividis',
                xticklabels=['Not Churn', 'Churn'],
                yticklabels=['Not Churn', 'Churn'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(f'{model} Confusion Matrix')
    plt.show()

# Evaluate model metrics
evaluate_model(logreg, X_train_resampled, y_train_resampled, X_test_sc
```

```
Train Accuracy: 0.78
Test Accuracy: 0.77
-----------------------------------

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.93      0.79      0.86       715
           1       0.35      0.66      0.46       119

    accuracy                           0.77       834
   macro avg       0.64      0.73      0.66       834
weighted avg       0.85      0.77      0.80       834

-----------------------------------

Confusion Matrix (Test Data):
[[567 148]
 [ 40  79]]
-----------------------------------

Train Score: 0.78
Test Score: 0.77
```



The logistic regression model achieved a train accuracy of approximately 78% and a test accuracy of around 77%.

For class 0 (the majority class), precision is high (93%) indicating that when the model predicts class 0 it is mostly correct. However, recall (sensitivity) is lower (79%) suggesting that some actual class 0 instances are missed.

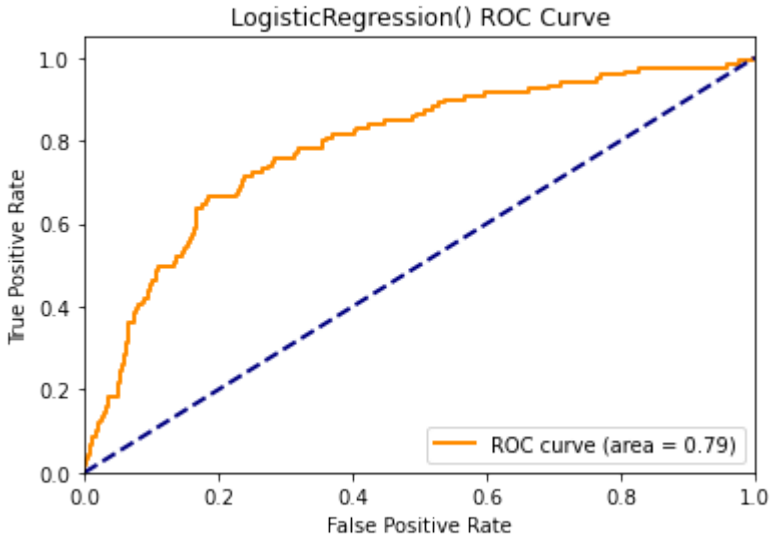
For class 1 (the minority class) precision is low (35%) meaning that when the model predicts class 1 it often makes mistakes. However, recall is higher (66%) indicating that the model captures a reasonable proportion of actual class 1 instances.

The weighted average F1-score is 0.80, which balances precision and recall across both classes.

The confusion matrix shows that the model correctly predicted 567 instances of class 0 and 79 instances of class 1. However, it misclassified 148 instances of class 0 as class 1.

The model seems to perform reasonably well but could benefit from further tuning or

In [28]:
```
from sklearn.metrics import auc

def plot_roc_curve(model, X_test, y_test):

    # Predict the probabilities for the positive class
    y_test_prob = model.predict_proba(X_test)[:, 1]

    # Compute the ROC curve
    fpr, tpr, thresholds = roc_curve(y_test, y_test_prob)
    roc_auc = auc(fpr, tpr)

    # Plot the ROC curve
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (ar
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.0])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'{model} ROC Curve')
    plt.legend(loc='lower right')
    plt.show()


# Evaluate the logistic regression model and plot the ROC curve
plot_roc_curve(logreg, X_test_scaled, y_test)
```

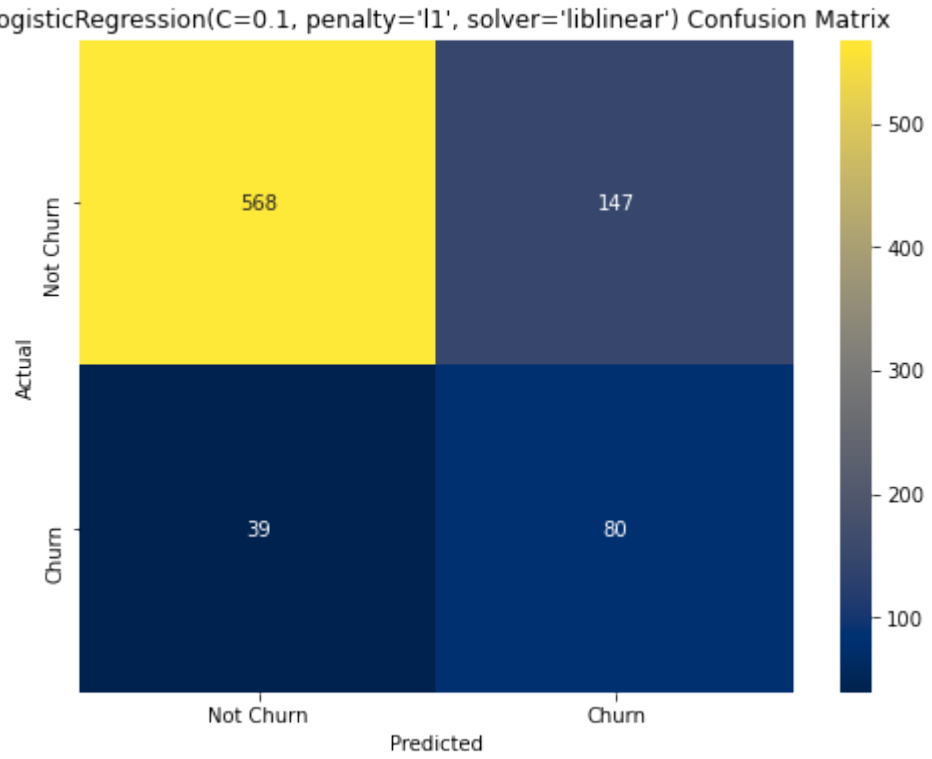


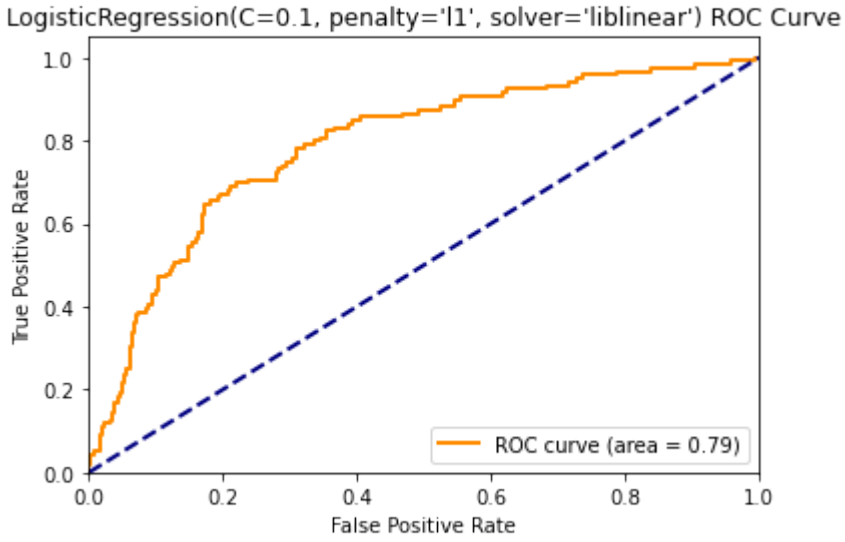
The next step is to carry out hyperparameter tuning on the model. We will specify a parameter grid for hyperparameter tuning. We will then perform grid search cross-validation (GridsearchCV) to find the best combination of hyperparameters and then fit the grid search to the training data and obtain the best hyperparameters for our model.

Hyperparameter Tuning

In [29]:
```
def tune_hyperparameters(model, param_grid, X_train, y_train, cv=5):
    '''
    Perform hyperparameter tuning using GridSearchCV.

    Parameters:
    - model: The classification model to tune.
    - param_grid: The hyperparameter grid to search.
    - X_train: Training data features.
    - y_train: Training data labels.
    - cv: Number of cross-validation folds (default is 5).

    Returns:
    - best_model: The model with the best hyperparameters.
    - best_params: The best hyperparameters found.
    '''
    grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
    grid_search.fit(X_train, y_train)

    best_model = grid_search.best_estimator_
    best_params = grid_search.best_params_

    print(f'Best Parameters: {best_params}')
    return best_model, best_params


# Define the logistic regression model
logreg = LogisticRegression()

# Define the hyperparameter grid for logistic regression
param_grid_logreg = {
    'C': [0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear']
}

# Perform hyperparameter tuning
best_logreg, best_params_logreg = tune_hyperparameters(logreg, param_g

# Evaluate the best logistic regression model
evaluate_model(best_logreg, X_train_resampled, y_train_resampled, X_te

# Plot the ROC curve for the best logistic regression model
plot_roc_curve(best_logreg, X_test_scaled, y_test)
```

```
Best Parameters: {'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'}
Train Accuracy: 0.78
Test Accuracy: 0.78
-----------------------------------

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.94      0.79      0.86       715
           1       0.35      0.67      0.46       119

    accuracy                           0.78       834
   macro avg       0.64      0.73      0.66       834
weighted avg       0.85      0.78      0.80       834

-----------------------------------

Confusion Matrix (Test Data):
[[568 147]
 [ 39  80]]
-----------------------------------

Train Score: 0.78
Test Score: 0.78
```





After hyperparameter tuning, the logistic regression model indicates that the best Parameters are: {'C': 0.1, 'penalty': 'L1', 'solver': 'liblinear'}, train accuracy of 78% and test accuracy of 78%.

For class 0 (the majority class), precision is high (94%) indicating that when the model predicts class 0, it is usually correct. However, recall (sensitivity) is lower (79%) suggesting that some actual class 0 instances were missed.

For class 1 (the minority class), precision is low (35%) meaning that when the model predicts class 1 it often makes mistakes. However, recall is higher (67%) indicating that the model captures a reasonable proportion of actual class 1 instances.

The weighted average F1-score is 0.80, which balances precision and recall across both classes.

The confusion matrix shows that the model correctly predicted 568 instances of class 0 and 80 instances of class 1. However, it misclassified 147 instances of class 0 as class 1.

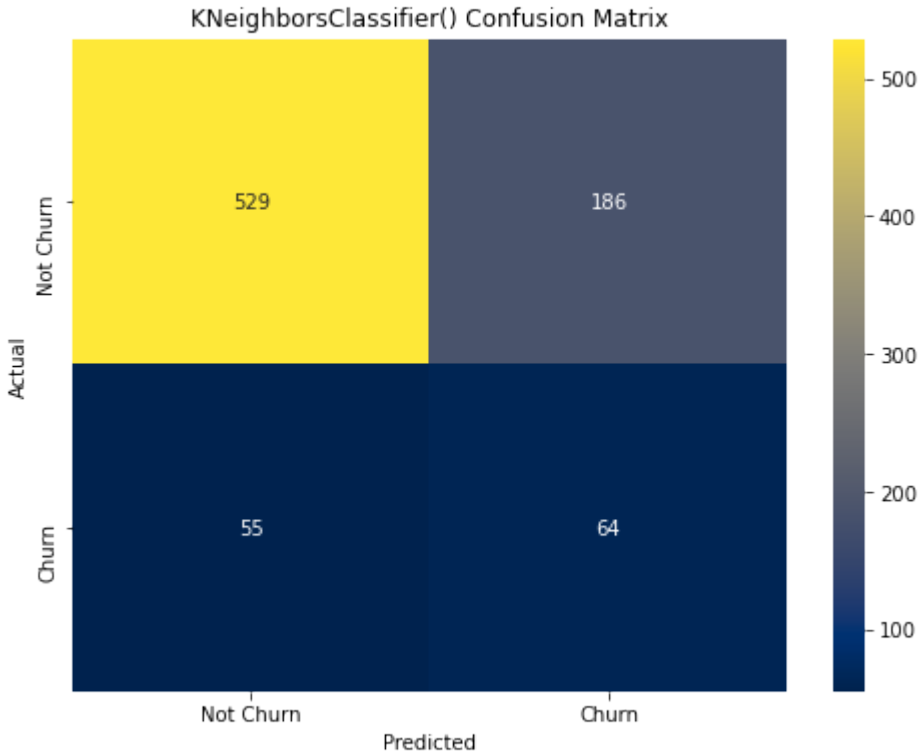
The tuned model performs similar to the previous one but there's still room for improvement, especially for class 1 predictions.

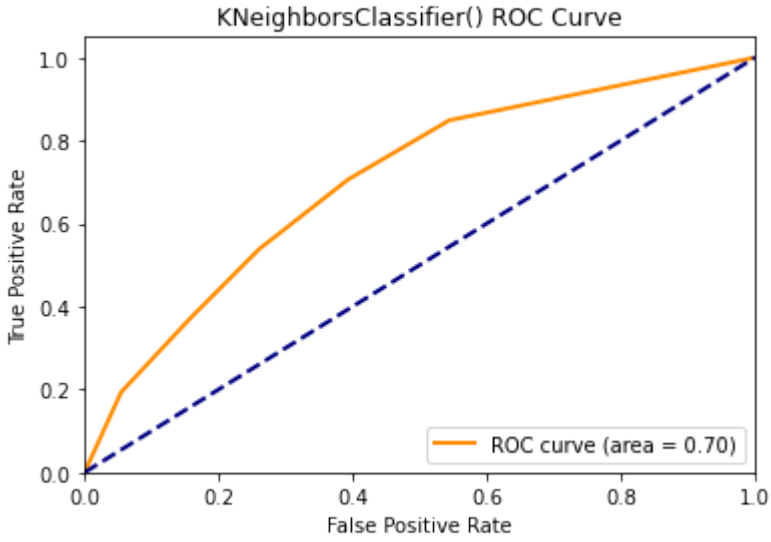
K-Nearest Neighbors

K-Nearest Neighbors (KNN) is a simple and intuitive machine learning algorithm that classifies new data points based on the majority vote of their nearest neighbors in the training dataset. We shall go ahead and assess its ability to predict whether customers will churn or not.

In [30]:
```
# Initialize KNN classifier
knn = KNeighborsClassifier(n_neighbors=5)

# Fit model to training data
knn.fit(X_train_resampled, y_train_resampled)

# Evaluate knn metrics
evaluate_model(knn, X_train_resampled, y_train_resampled, X_test_scale

# Plot knn ROC curve
plot_roc_curve(knn, X_test_scaled, y_test)
```

```
Train Accuracy: 0.84
Test Accuracy: 0.71
-----------------------------------

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.91      0.74      0.81       715
           1       0.26      0.54      0.35       119

    accuracy                           0.71       834
   macro avg       0.58      0.64      0.58       834
weighted avg       0.81      0.71      0.75       834

-----------------------------------

Confusion Matrix (Test Data):
[[529 186]
 [ 55  64]]
-----------------------------------

Train Score: 0.84
Test Score: 0.71
```





The KNN model achieved a train accuracy of 84% and a test accuracy of 71%. This indicates that when the model was trained on the training data, it correctly predicted the class label 84% of the time. Similarly, when evaluated on new, unseen data (the test set), it maintained a predictive accuracy of 71%.

In the classification report, the precision was high at 91%. for the majority class (class 0)-no churn. This means that out of all instances predicted as class 0, 91% were actually class 0. However, the recall was moderate at 74%, indicating that the model correctly identified 74% of all actual class 0 instances.

Conversely, for the minority class (class 1)-churn, which likely represents the event or positive class, the precision was low at 26%. This implies that out of all instances predicted as class 1, only 26% were actually class 1. However, the recall was higher at 54%, suggesting that the model captured 54% of all actual churn instances.

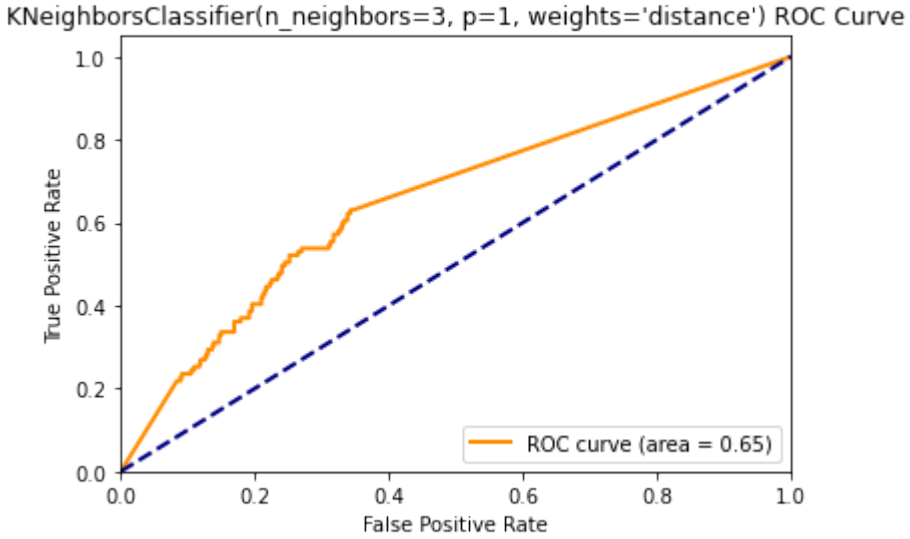
The weighted average F1-score was 0.75 indicating a balance between precision and recall across both classes. The confusion matrix revealed that the model correctly predicted 529 instances of class 0 and 64 instances of class 1, but misclassified 186 instances of class 0 as class 1.

Overall, the model performed better than the hyperparameter tuned logistic regression model in terms of accuracy but performed worse in terms of precision, recall.

The next step was to do hyperparameter tuning on the KNN model and assess how it would affect its performance.

Hyperparameter Tuning

In [31]:
```
# Define hyperparameters
param_grid_knn = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'p': [1, 2]  # 1 for Manhattan distance, 2 for Euclidean distance
}

# Perform hyperparameter tuning
best_knn, best_params_knn = tune_hyperparameters(knn, param_grid_knn,

# Evaluate the best knn model
evaluate_model(best_knn, X_train_resampled, y_train_resampled, X_test_

# Plot the ROC curve for the best knn model
plot_roc_curve(best_knn, X_test_scaled, y_test)
```



After hyperparameter tuning the KNN model, significant improvements are evident in both training and testing accuracies. The best parameters obtained, with (k = 3) neighbors, (p = 1) (indicating Manhattan distance), and using distance-based weighting, demonstrate a refined model configuration. The model now achieves a perfect training accuracy of 100%, indicating that it perfectly fits the training data.

In testing, the accuracy remains relatively high at 74%, suggesting that the model generalizes well to unseen data. However, the classification report highlights persistent challenges in correctly identifying instances of the minority class (class 1) with low precision and moderate recall. This imbalance is reflected in the confusion matrix where a considerable number of class 1 instances are misclassified as class 0.

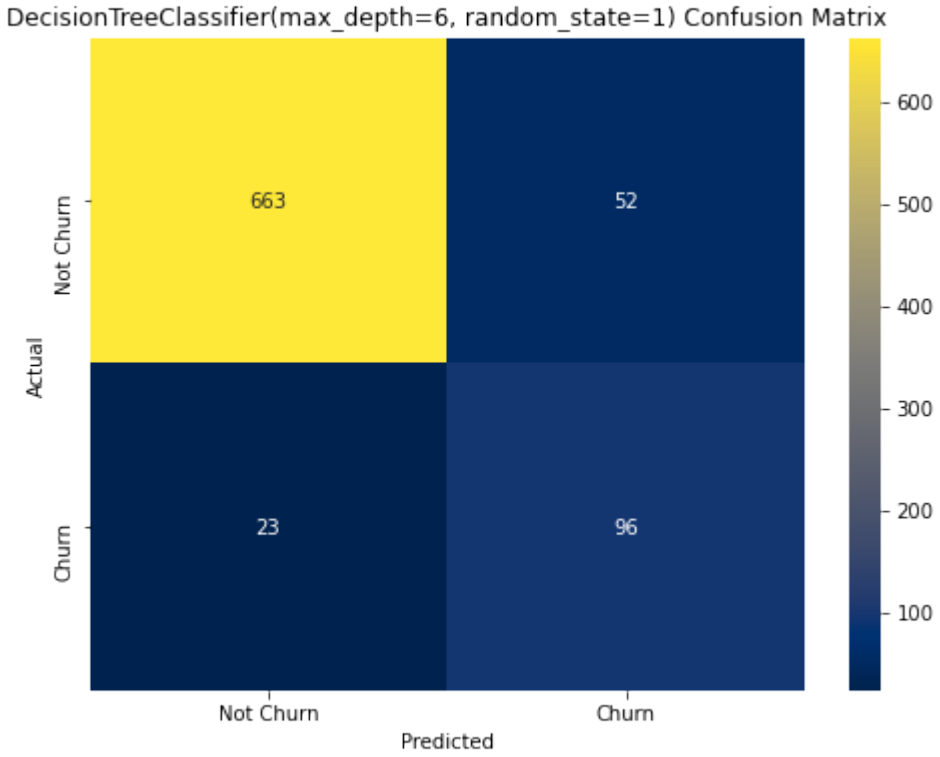
Despite these shortcomings, the model exhibits improved overall performance post-tuning showcasing the effectiveness of parameter optimization in enhancing predictive capabilities.

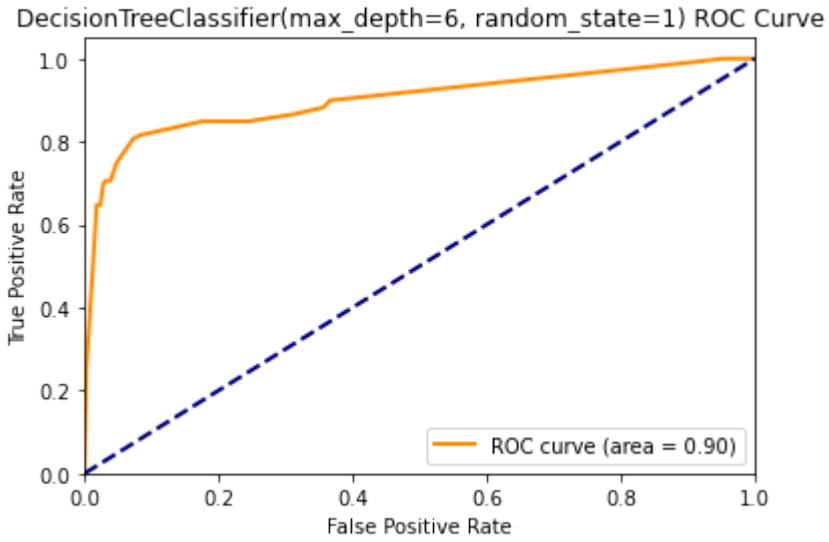
Decision Trees Classifier

A decision tree classifier is a predictive model that learns simple decision rules inferred from the data features to predict the target variable's class. We will go ahead and assess its ability at predicting customer attrition.

In [32]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Instantiate a DecisionTreeClassifier with a maximum depth
dt = DecisionTreeClassifier(max_depth=6, random_state=1)

# Fit the decision tree to the training set
dt.fit(X_train_resampled, y_train_resampled)

# Evaluate the best decision tree model
evaluate_model(dt, X_train_resampled, y_train_resampled, X_test_scaled

# Plot the ROC curve for the best decision tree model
plot_roc_curve(dt, X_test_scaled, y_test)
```

```
Train Accuracy: 0.93
Test Accuracy: 0.91
-----------------------------------

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.97      0.93      0.95       715
           1       0.65      0.81      0.72       119

    accuracy                           0.91       834
   macro avg       0.81      0.87      0.83       834
weighted avg       0.92      0.91      0.91       834

-----------------------------------

Confusion Matrix (Test Data):
[[663  52]
 [ 23  96]]
-----------------------------------

Train Score: 0.93
Test Score: 0.91
```





The decision tree classifier demonstrates robust performance with high accuracies of 93% on the training set and 91% on the test set indicating effective learning and generalization capabilities.

The classification report further highlights its ability to distinguish between classes, achieving precision and recall scores of 97% and 93% respectively for the majority class (class 0) and 65% and 81% for the minority class (class 1).

Although the model shows stronger performance in classifying instances of the majority class, it still achieves respectable metrics for the minority class, contributing to an overall balanced performance.

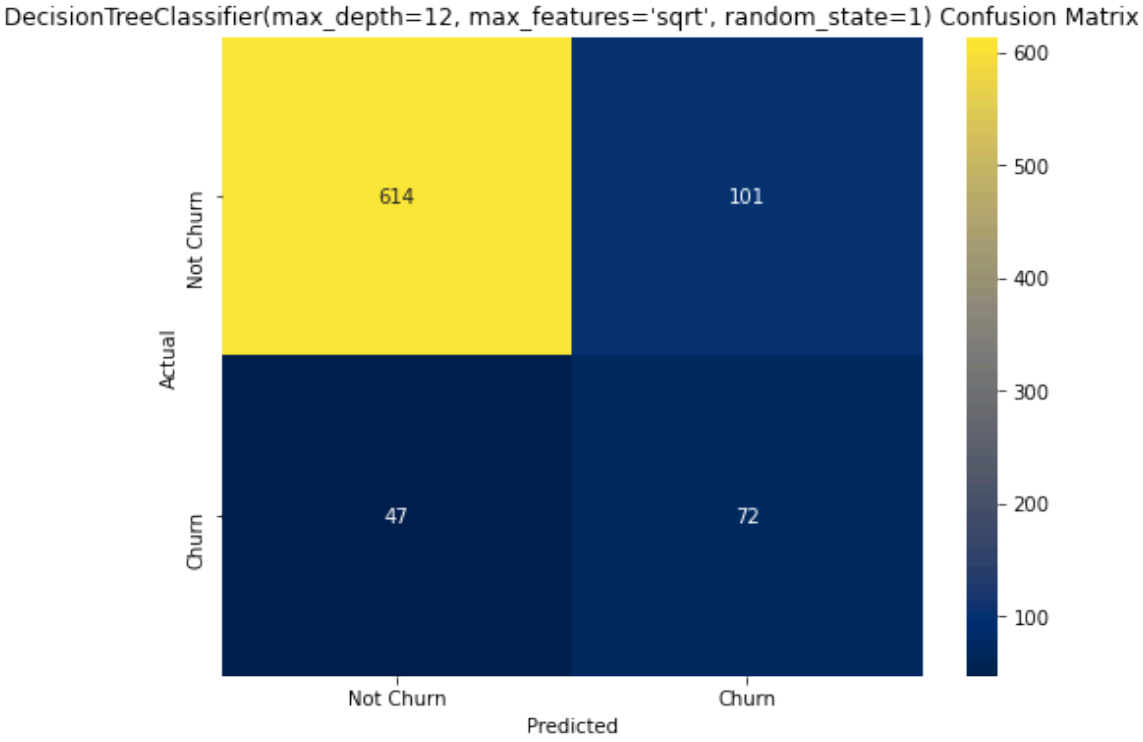
The confusion matrix illustrates the model's ability to correctly classify the majority of instances, with a relatively low number of misclassifications. The majority of instances belonging to class 0 were correctly classified with 663 instances correctly identified as class

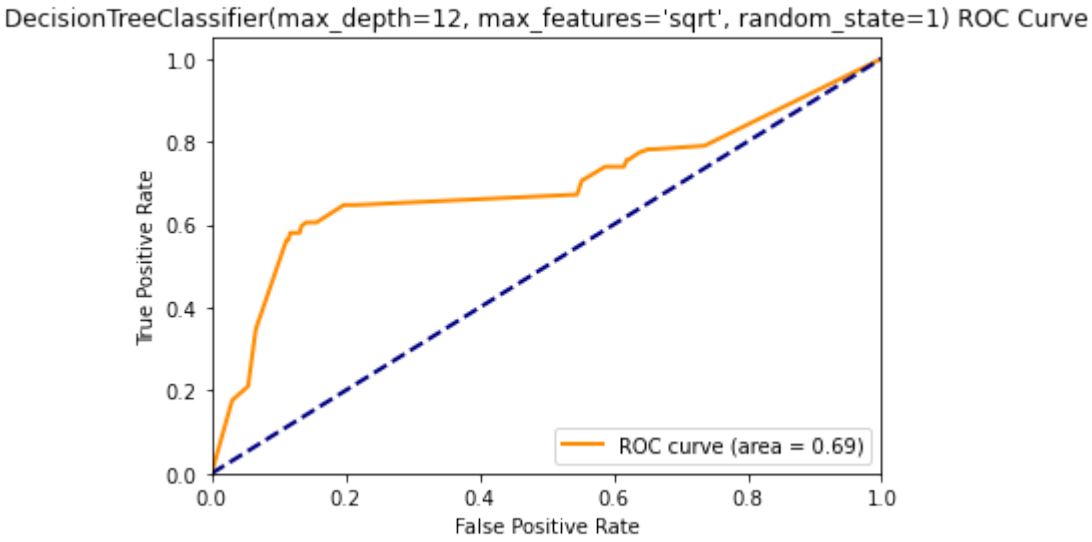
0 and only 52 instances misclassified as class 1. This indicates a high level of accuracy in distinguishing instances of class 0. Similarly, for class 1, the model correctly classified 96

Hyperparameter Tuning

In [33]:
```
# Define the parameter grid
param_grid_dt = {
    'criterion': ['gini', 'entropy'],
    'max_features': ['sqrt', 'log2'],
    'min_samples_split': [2, 5, 10],
    'max_depth': [2, 4, 6, 8, 10, 12],
    'min_samples_leaf': [1, 2, 3, 4, 5, 6]

}

# Perform hyperparameter tuning
best_dt, best_params_dt = tune_hyperparameters(dt, param_grid_dt, X_tr

# Evaluate the best decision tree model
evaluate_model(best_dt, X_train_resampled, y_train_resampled, X_test_s

# Plot the ROC curve for the best decision tree model
plot_roc_curve(best_dt, X_test_scaled, y_test)
```

```
Best Parameters: {'criterion': 'gini', 'max_depth': 12, 'max_feature
s': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2}
Train Accuracy: 0.89
Test Accuracy: 0.82
-----------------------------------

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.93      0.86      0.89       715
           1       0.42      0.61      0.49       119

    accuracy                           0.82       834
   macro avg       0.67      0.73      0.69       834
weighted avg       0.86      0.82      0.84       834

-----------------------------------

Confusion Matrix (Test Data):
[[614 101]
 [ 47  72]]
-----------------------------------

Train Score: 0.89
Test Score: 0.82
```





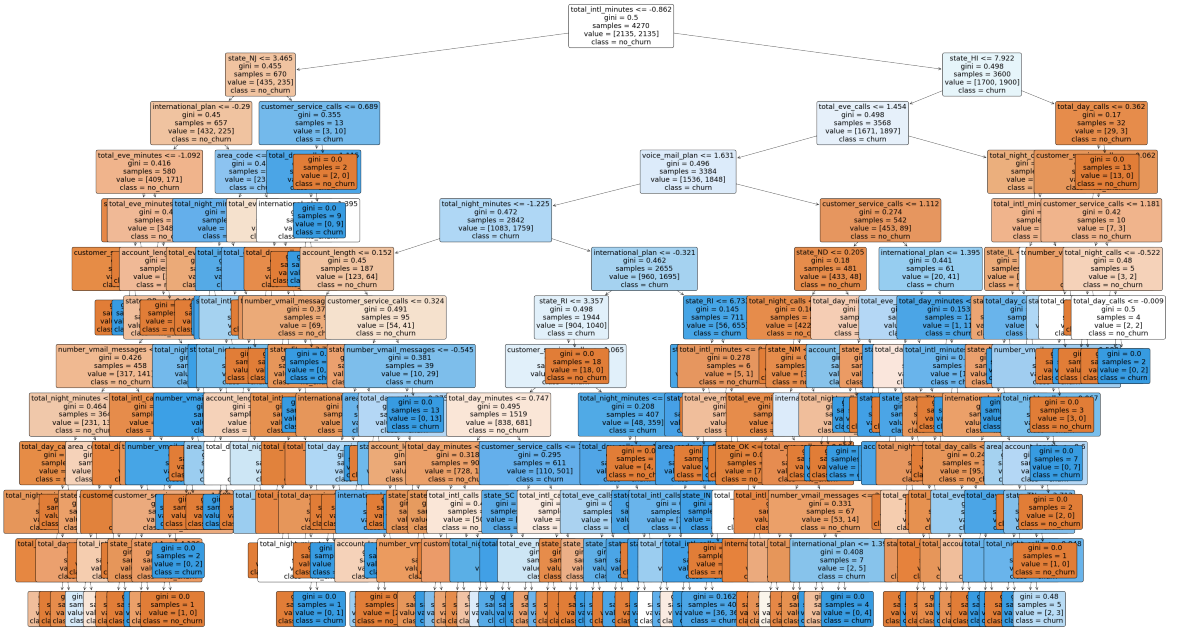
The optimized decision tree model decreased performance with a train accuracy of 89% down from 93% and a test accuracy of 82% from 91%.

While it maintains reliable predictive capabilities, there is a slight performance drop on unseen data. The classification report reveals that the model performs well for the majority class with precision of 93% and recall of 86% but struggles with the minority class with precision of 42% and recall of 61%.

The overall test accuracy of 82% suggests that while the decision tree model is effective, further tuning or alternative approaches may be needed to enhance its performance, especially in predicting the minority class.

In some cases, hyperparameter tuning might lead to overfitting where the model becomes too complex and starts to capture noise in the training data eg. when the max_depth is too high it allows the tree to create too many splits, fitting the training data very closely but performing poorly on unseen data and that might be the case.

In [34]:
```
from sklearn.tree import plot_tree

# Plot the decision tree
plt.figure(figsize=(50,30))  # Adjust the figure size as needed
plot_tree(best_dt, filled=True, feature_names=X_test.columns, class_na
          rounded=True, fontsize=18)
# Display decision tree
plt.show()
```

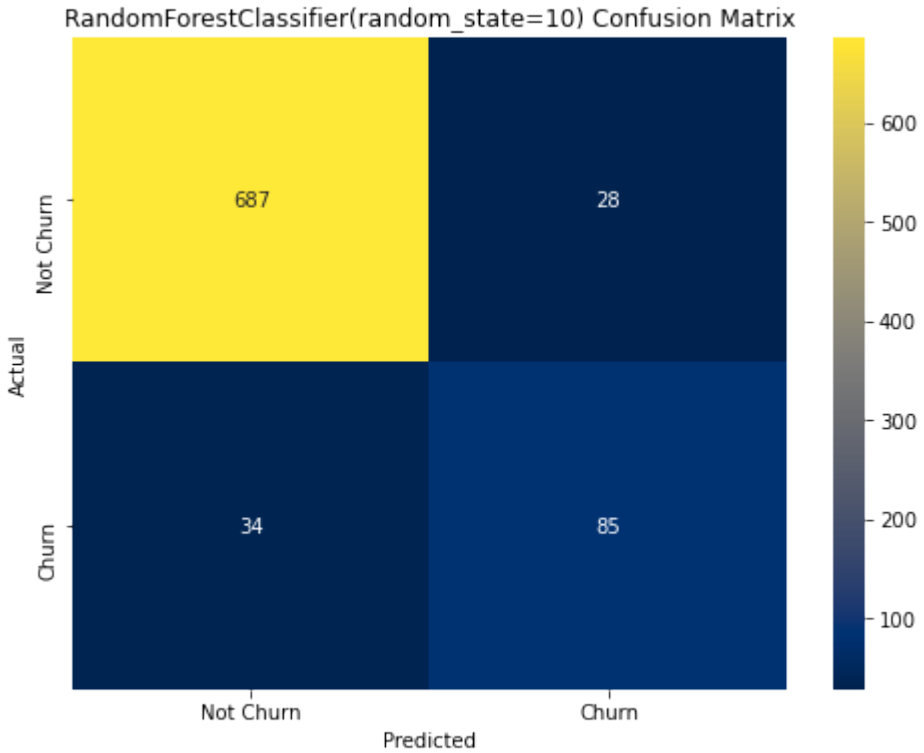


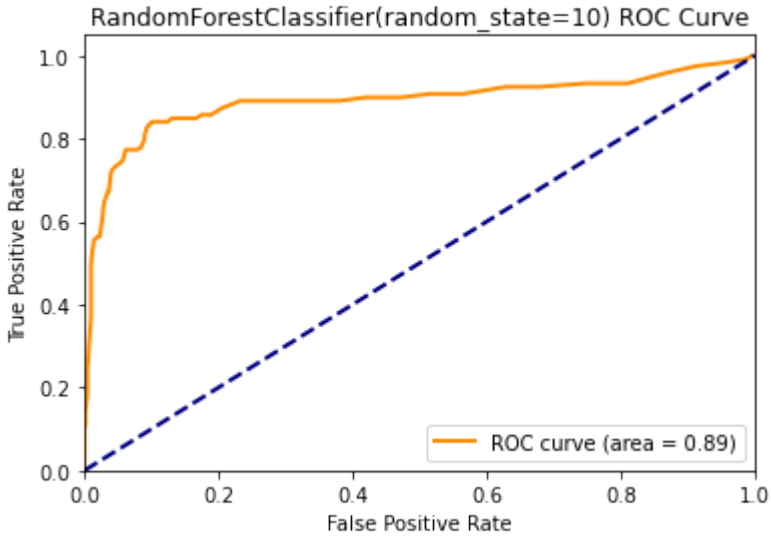
Random Forest Classifier

Random Forest classifier is an ensemble learning method that constructs multiple decision trees during training and outputs the class that is the mode of the classes (classification) or the mean prediction (regression) of the individual trees. We opted for a value of 100 for the n-estimators since the dataset wasn't considerably large.

In [35]:
```
# Initialize the Random Forest classifier with 100 estimators and a ra
rf = RandomForestClassifier(n_estimators=100, random_state=10)

# Train the classifier using the resampled training data to handle cla
rf.fit(X_train_resampled, y_train_resampled)

# Evaluate Random Forest model metrics
evaluate_model(rf, X_train_resampled, y_train_resampled, X_test_scaled

# Plot Random Forest ROC curve
plot_roc_curve(rf, X_test_scaled, y_test)
```

```
Train Accuracy: 1.00
Test Accuracy: 0.93
-----------------------------------

Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.95      0.96      0.96       715
           1       0.75      0.71      0.73       119

    accuracy                           0.93       834
   macro avg       0.85      0.84      0.84       834
weighted avg       0.92      0.93      0.92       834

-----------------------------------

Confusion Matrix (Test Data):
[[687  28]
 [ 34  85]]
-----------------------------------

Train Score: 1.00
Test Score: 0.93
```





With a training accuracy of 100% and a test accuracy of 93%, the model demonstrates excellent generalization performance, indicating a high level of fitting to the training data while effectively capturing the underlying patterns in unseen test data.

The classification report showcases strong precision (0.95 for class 0, 0.75 for class 1) and recall (0.96 for class 0, 0.71 for class 1) scores for both classes with class 0 exhibiting higher precision but slightly lower recall compared to class 1.

The F1-score, which balances precision and recall is also notable with values of 0.96 for class 0 and 0.73 for class 1. Although the model maintains a good balance between precision and recall for both classes, there are still instances of misclassification, particularly for class 1.

However, the confusion matrix highlights the model's ability to correctly classify a majority of instances, with 687 instances of class 0 and 85 instances of class 1 correctly classified, and only 28 instances of class 0 and 34 instances of class 1 misclassified. Overall, these results suggest that the model effectively discriminates between classes while achieving high overall accuracy, showcasing its robustness and effectiveness in classification tasks.

EVALUATION

Now that we have assessed the performance of our models, it is time to evaluate the different ROC curves after hyperparameter tuning and to analyse feature importance in order to come up with our conclusions and recommendations. We create a dataframe that displays each model's test data after hyperparameter tuning since all the models we tested showed a slight improvement in performance after hyperparameter tuning was done.

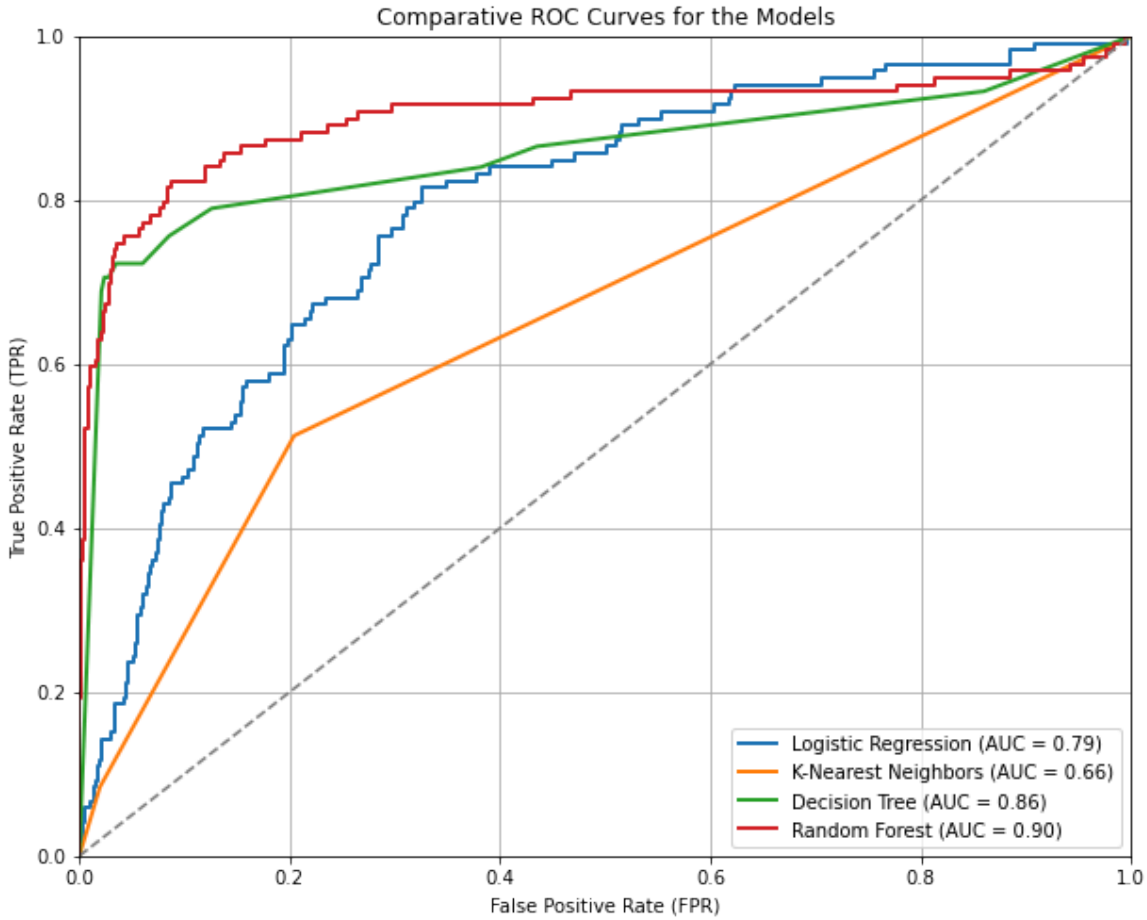
In [36]:
```
model_performance = pd.DataFrame({'Model':['Logistic Regression',
                                           'K-Nearest Neighbors',
                                           'Decision Trees',
                                           'Random Forest'],
                                  'Accuracy':[0.78,0.74,0.91,0.93],
                                  'Precision':[0.64,0.57,0.81,0.85],
                                  'Recall':[0.73,0.60,0.87,0.84],
                                  'F1 Score':[0.66,0.57,0.83,0.83]
                                   })

model_performance
```

Out[36]:

	Model	Accuracy	Precision	Recall	F1 Score
0	Logistic Regression	0.78	0.64	0.73	0.66
1	K-Nearest Neighbors	0.74	0.57	0.60	0.57
2	Decision Trees	0.91	0.81	0.87	0.83
3	Random Forest	0.93	0.85	0.84	0.83

Next we plot the individual AUC-ROC curves of the models on one axis to compare their performance.

In [37]:
```
# Initialize the classifiers
classifiers = [
    LogisticRegression(),
    KNeighborsClassifier(n_neighbors=3),
    DecisionTreeClassifier(max_depth=8, random_state=1),
    RandomForestClassifier(n_estimators=50, max_depth=10, random_state
]

# List model names
names = ['Logistic Regression', 'K-Nearest Neighbors', 'Decision Tree'

# Set size of the figure
plt.figure(figsize=(10, 8))

# Loop through each classifier and plot its ROC curve
for clf, name in zip(classifiers, names):
    clf.fit(X_train_resampled, y_train_resampled)
    y_prob = clf.predict_proba(X_test_scaled)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_test, y_prob)
    roc_auc = roc_auc_score(y_test, y_prob)
    plt.plot(fpr, tpr, lw=2, label=f'{name} (AUC = {roc_auc:.2f})')

# Plot the diagonal line representing a random classifier
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Comparative ROC Curves for the Models')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```



The plot above displays a comparative AUC-ROC curve analysis for our four classification models namely Logistic Regression, K-Nearest Neighbors (KNN), Decision Tree, and Random Forest. Each ROC curve illustrates the relationship between the true positive rate (TPR) and the false positive rate (FPR) across different threshold settings.

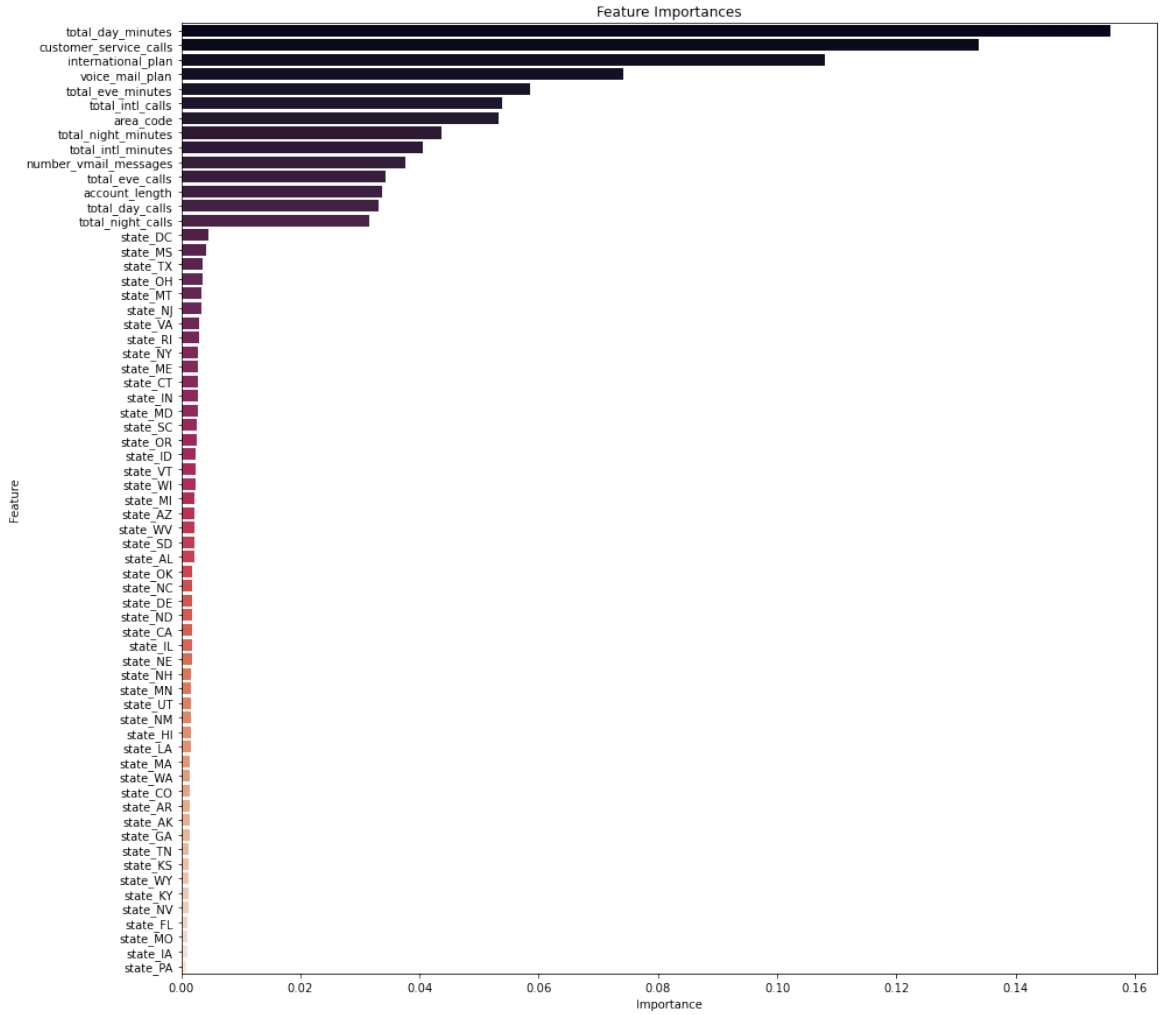
- The Random Forest model exhibits the highest area under the curve (AUC) value of 0.90 indicating superior performance in distinguishing between classes.
- The Decision Tree model follows with an AUC of 0.86 demonstrating relatively strong predictive capability.
- Logistic Regression shows moderate effectiveness with an AUC of 0.79.
- K-Nearest Neighbors model has the lowest AUC of 0.66 suggesting it is the least effective in this comparison.

The diagonal dashed line represents a random classifier with an AUC of 0.50. Overall, the Random Forest and Decision Tree models outperform Logistic Regression and K-Nearest Neighbors in this analysis with Random Forest Classifier standing out as the best!

Feature Importance

Feature importance reveals how much each feature influences the model's predictions. When a feature has a high importance value, it means that it plays a bigger role in shaping the model's predictions. In simpler terms, it tells us which factors matter most in determining the outcome according to the model's perspective. In this scenario we want to assess which features held the most importance according to the Random Forest Classifier model.

In [38]:
```
# Extract feature importances
feature_importances = rf.feature_importances_

# Create a DataFrame for feature importances
feature_importance_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': feature_importances
})

# Sort the DataFrame by importance
feature_importance_df = feature_importance_df.sort_values(by='Importar

# Plot feature importances as a histogram
plt.figure(figsize=(15, 15))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df, p
plt.title('Feature Importances')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.show()

# Display the DataFrame
feature_importance_df
```



Out[38]:

	Feature	Importance
5	total_day_minutes	0.155832
13	customer_service_calls	0.133837
2	international_plan	0.107924
3	voice_mail_plan	0.074227
7	total_eve_minutes	0.058581
...	...	...
47	state_NV	0.001187
23	state_FL	0.001121
38	state_MO	0.001021
26	state_IA	0.000999
52	state_PA	0.000775

65 rows × 2 columns

Save to Pickle

Pickle is a module in Python that allows you to serialize and deserialize Python objects. In simpler terms, it's a way to convert a Python object into a byte stream and vice versa. When you serialize an object using pickle, you convert it into a byte stream that can be written to a file or sent over a network.

In [39]:
```
import pickle

# Make predictions on the testing data
y_pred = rf.predict(X_test_scaled)

# Your trained Random Forest model
rf_model = y_pred

# Save the model to a pickle file with the name "RandomForest_Model.pk
with open("RandomForest_Model.pkl", "wb") as file:
    pickle.dump(rf_model, file)
```

CONCLUSIONS

In conclusion, of all the models we have tested, we can confidently say that Random Forest Classifier is the best at accurately predicting customer attrition at SyriaTel. It boasts a training accuracy of 100% and a test accuracy of 93% showing that the model demonstrates excellent generalization performance which indicates a high level of fitting to the training data while effectively capturing the underlying patterns in unseen test data without overfitting or underfitting.

In terms of feature importance, the most important features that greatly influence churn are such as:

1. Total day minutes
2. Number of customer service calls
3. International plan
4. Voicemail plan
5. Total evening minutes
6. Total international calls
7. Number of voicemail messages
8. Total evening calls
9. Account length
10. Total day calls
11. Total night calls

- Call usage features: We can conclude that call usage features have the highest importance scores. This suggests that customers' calling behavior including the duration and frequency of calls made during different times of the day as well as their subscription to international and voice mail plans, are strong predictors of churn.
- Customer tenure and service characteristics: Features like 'account_length', 'total_eve_calls', 'total_day_calls', and 'total_night_calls' also have relatively high importance scores indicating that customer tenure and the overall usage of various services play a significant role in predicting churn.
- Geographical location: The feature 'area_code' has a moderate importance score suggesting that customers' geographical locations could influence their churn behavior.
- State-level features: Most of the State features have relatively low importance scores indicating that they may not be as influential in predicting churn compared to other features.
- Feature Selection and Model Optimization: Although the random forest classifier can handle a large number of features, it may be beneficial to perform feature selection or dimensionality reduction to improve the model's performance and interpretability.

RECOMMENDATIONS

- We recommend that SyriaTel leverage the power of the Random Forest Classifier as the primary predictive model for customer churn analysis. This robust machine learning algorithm has demonstrated exceptional performance, boasting an impressive ROC curve and outstanding metrics across accuracy, F1-score, recall, and precision when evaluated on the test dataset. Consequently, the Random Forest Classifier is optimally positioned to accurately classify customers as either potential churners or loyal subscribers, providing SyriaTel with a valuable tool for proactive retention strategies and maximizing customer lifetime value.
- SyriaTel should closely monitor call usage patterns and develop targeted retention strategies for customers exhibiting high call usage or a sudden change in their calling behavior.
- SyriaTel should focus on improving customer satisfaction and offering attractive loyalty programs or discounts to long-term customers to reduce the likelihood of churn.
- SyriaTel should analyze regional differences in customer churn rates and tailor their marketing and customer service strategies accordingly, addressing region-specific needs and preferences.
- While state-level features should not be disregarded entirely, SyriaTel should prioritize the more important features mentioned above when developing their churn prediction and prevention strategies.

- In the spirit of model improvement, it would be prudent to consider techniques like recursive feature elimination or principal component analysis to identify and retain the most relevant features, potentially improving the model's accuracy and efficiency. Machine learning techniques such as XGBoost and Support Vector Machines (SVM) could also be explored in order to assess their customer churn prediction capabilities.

Overall, by focusing on the most important features, particularly those related to call usage, customer tenure, and service characteristics, SyriaTel can develop targeted strategies to identify and retain customers at risk of churning, ultimately improving their customer retention and revenue.