SyriaTel Customer Churn Prediction Model

1. Business Understanding

1.1 Objectives

The primary objective of this project is to build a predictive model that can identify customers who are likely to stop doing business with **SyriaTel**. By detecting **potential churners** in advance, the company can take proactive measures to retain them, thereby reducing revenue loss and increasing customer lifetime value.

1.2 Problem Statement

SyriaTel, a telecommunications company, experiences customer churn, which negatively impacts revenue and business stability. This project aims to analyze customer data to identify key factors influencing churn and develop a machine learning model that can predict churn probability. The insights derived will help SyriaTel implement effective retention strategies.

Key Questions

1. What are the primary factors influencing customer churn?
2. Can we accurately predict which customers are likely to churn?
3. How can SyriaTel use these predictions to design effective retention strategies?
4. What are the most cost-effective interventions for reducing churn?
5. How does customer service interaction impact churn rates?
6. Are there specific usage patterns that correlate strongly with customer retention?

1.3 Metrics of Success

The model's success will be evaluated based on the following metrics:

- **Accuracy**: Measures overall correctness of predictions.
- **Precision**: Ensures that when the model predicts churn, it is correct.
- **Recall (Sensitivity)**: Captures how well the model identifies actual churners.
- **F1-Score**: Balances precision and recall.
- **ROC-AUC Score**: Evaluates the model's ability to differentiate between churners and non-churners.
- **Business Impact**: Reduction in churn rate and increase in customer retention due to actionable insights.

1.4 External Relevance

- **Industry Benchmarking**: Customer churn prediction is a critical problem in the telecom industry, where retaining an existing customer is cheaper than acquiring a new one.
- **Competitive Advantage**: Implementing a predictive model allows SyriaTel to tailor customer retention strategies, increasing loyalty and reducing costs.
- **Customer Satisfaction**: Identifying at-risk customers enables personalized interventions such as offers, discounts, or improved services, enhancing the customer

experience.

2. Data understanding

2.1 Data description

The dataset contains customer-related information that influences their decision to stay or leave their subscription. It contains **3333 records** and **21 features**

Key features include:

- **Customer Information:** `state`, `account length`, `area code`, `phone number`
- **Subscription Plans:** `international plan`, `voice mail plan`
- **Usage Metrics:** `number vmail messages`, `total day minutes`, `total eve minutes`, `total night minutes`, `total intl minutes`
- **Call Records:** `total day calls`, `total eve calls`, `total night calls`, `total intl calls`, `customer service calls`
- **Billing Details:** `total day charge`, `total eve charge`, `total night charge`, `total intl charge`
- **Churn Label:** `churn` (Boolean, `True` = churned, `False` = not churned)

2.2 Data Source

The dataset was sourced from a published kaggle dataset

In [1]:

```
# importing the libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from scipy import stats
from scipy.stats import chi2_contingency
```

In [2]:

```
# loading the dataset
df = pd.read_csv('bigml_59c28831336c6604c800002a.csv')

# checking the first records
df.head()
```

Out[2]:

	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	tot d char
0	KS	128	415	382-4657	no	yes	25	265.1	110	45.
1	OH	107	415	371-7191	no	yes	26	161.6	123	27.
2	NJ	137	415	358-1921	no	no	0	243.4	114	41.
3	OH	84	408	375-9999	yes	no	0	299.4	71	50.
4	OK	75	415	330-6626	yes	no	0	166.7	113	28.

5 rows × 21 columns

2.3 Statistical Summary

In [3]:
```
# checking the dataset info and their datatype
df.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
```

In [4]:
```
df.columns
```

Out[4]:
```
Index(['state', 'account length', 'area code', 'phone number',
       'international plan', 'voice mail plan', 'number vmail 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 charge',
       'total intl minutes', 'total intl calls', 'total intl charge',
       'customer service calls', 'churn'],
      dtype='object')
```

In [5]:
```
# checking the dataset description
df.describe().T
```

Out[5]:

	count	mean	std	min	25%	50%	75%	max
account length	3333.0	101.064806	39.822106	1.00	74.00	101.00	127.00	243.00
area code	3333.0	437.182418	42.371290	408.00	408.00	415.00	510.00	510.00
number vmail messages	3333.0	8.099010	13.688365	0.00	0.00	0.00	20.00	51.00
total day minutes	3333.0	179.775098	54.467389	0.00	143.70	179.40	216.40	350.80
total day calls	3333.0	100.435644	20.069084	0.00	87.00	101.00	114.00	165.00
total day charge	3333.0	30.562307	9.259435	0.00	24.43	30.50	36.79	59.64
total eve minutes	3333.0	200.980348	50.713844	0.00	166.60	201.40	235.30	363.70
total eve calls	3333.0	100.114311	19.922625	0.00	87.00	100.00	114.00	170.00
total eve charge	3333.0	17.083540	4.310668	0.00	14.16	17.12	20.00	30.91
total night minutes	3333.0	200.872037	50.573847	23.20	167.00	201.20	235.30	395.00
total night calls	3333.0	100.107711	19.568609	33.00	87.00	100.00	113.00	175.00
total night charge	3333.0	9.039325	2.275873	1.04	7.52	9.05	10.59	17.77
total intl minutes	3333.0	10.237294	2.791840	0.00	8.50	10.30	12.10	20.00
total intl calls	3333.0	4.479448	2.461214	0.00	3.00	4.00	6.00	20.00
total intl charge	3333.0	2.764581	0.753773	0.00	2.30	2.78	3.27	5.40
customer service calls	3333.0	1.562856	1.315491	0.00	1.00	1.00	2.00	9.00

2.3 Data quality assesement

2.3.1 Completeness

- **Strengths**:
  - Most key customer attributes, such as `account length`, `total day minutes`, and `total intl calls`, are well-populated.
  - Churn labels are available for all records, ensuring a clear classification problem.
- **Weaknesses**:
  - Some features like `voice mail plan` and `international plan` may have missing or ambiguous entries.
  - Potential missing values in `customer service calls`, which might impact churn predictions.

2.3.2 Accuracy

- **Strengths**:
  - Billing-related attributes like `total day charge` and `total intl charge` are likely accurate due to automated systems.
  - Usage-based features such as `total day minutes` and `total eve minutes` reflect real customer interactions.

- **Weaknesses**:
  - `state` might contain inconsistencies due to data entry errors.
  - Customer-reported features like `customer service calls` may be subject to human error or misreporting.

2.3.3 Relevance

- **Strengths**:
  - Most features are directly linked to customer behavior and service usage, making them relevant for churn prediction.
  - `total day minutes` and `total intl minutes` likely indicate engagement levels, which impact churn likelihood.
- **Weaknesses**:
  - Some categorical features like `state` may have minimal impact on churn prediction and require evaluation.
  - The `area code` might not be a significant predictor of churn and could introduce noise into the model.

2.4 Next steps ?

1. Data Cleaning

- Checking and handling missing values using imputation techniques (mean, median, or mode).
- Remove or transform outliers to improve model robustness.

2. Feature Engineering

- Create new features such as "average monthly spend" or "customer engagement score."
- Normalize/scale numerical features for better model performance.
- Encode categorical variables using One-Hot Encoding or Label Encoding.
- Address class imbalance using SMOTE (Synthetic Minority Over-sampling Technique) if needed.

3. EDA

- Perform univariate, bivariate, and multivariate analysis to understand distributions and relationships.
- Visualize data using histograms, boxplots, correlation matrices, and pair plots.
- Identify trends, anomalies, and potential data transformation needs.

3. Data Preparation

3.1 Data Cleaning

3.1.1 Missing Values

- Checking and handling missing values using imputation techniques (mean, median, or mode).

In [6]:
```
# checking for null values
df.isna().sum()
```

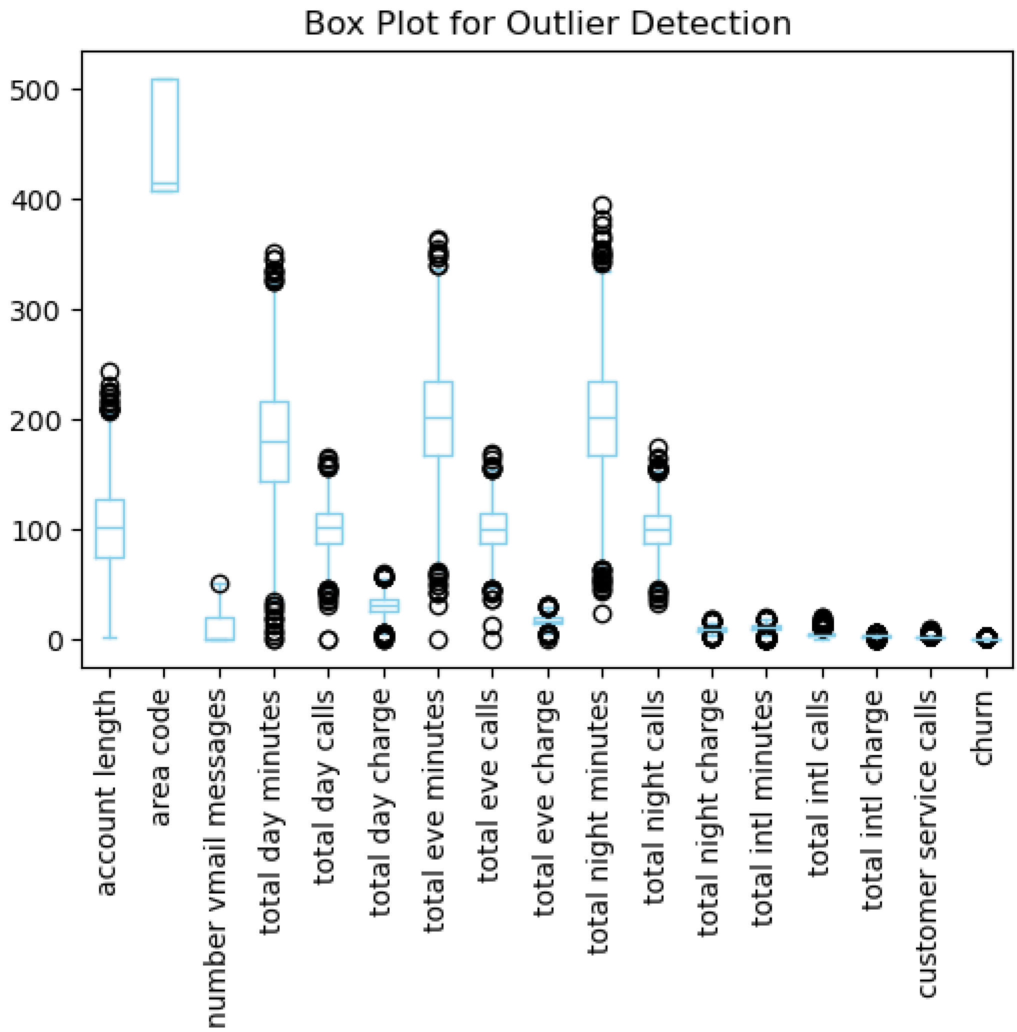
Out[6]:
```
state                     0
account length            0
area code                 0
phone number              0
international plan        0
voice mail plan           0
number vmail messages     0
total day minutes         0
total day calls           0
total day charge          0
total eve 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
```

No missing values spotted

3.1.2 Outliers

- Remove or transform outliers to improve model robustness.

In [7]:
```
# checking for outliers using box plot visually

# Create a box plot for all numerical values
plt.figure(figsize=(6,4))

df.boxplot(rot=90, grid=False, color='skyblue')
plt.title("Box Plot for Outlier Detection")
plt.show()
```



To note: Features like `account length`, `total day minutes`, `total evening minutes` and `total night minutes` have a high number of outliers

In [8]:
```
# checking for outliers using z-score
z_scores = np.abs(stats.zscore(df.select_dtypes(include=['number'])))
outliers = (z_scores > 3)

# Count of outliers per column
outliers.sum(axis=0)
```

Out[8]:
```
account length             7
area code                  0
number vmail messages      3
total day minutes          9
total day calls            9
total day charge           9
total eve minutes          9
total eve calls            7
total eve charge           9
total night minutes       11
total night calls          6
total night charge        11
total intl minutes        22
total intl calls          50
total intl charge         22
customer service calls    35
dtype: int64
```

To Note: Since there are **features** with outliers but are still **important** for our model, we are not going to drop them so that our model can perform well

3.1.3 Columns

- Removing white spaces in the columns

In [9]:
```
# removing the white spaces in the column names and replacing them with
df.columns = df.columns.str.strip().str.lower().str.replace(" ", "_")

# checking column names
df.columns
```

Out[9]:
```
Index(['state', 'account_length', 'area_code', 'phone_number',
       'international_plan', 'voice_mail_plan', 'number_vmail_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_charge',
       'total_intl_minutes', 'total_intl_calls', 'total_intl_charge',
       'customer_service_calls', 'churn'],
      dtype='object')
```

3.1.4 Droping unecessary columns

- Droping columns like phone number which is not helpful in training our model

In [10]:
```
# droping the unecessary column
df = df.drop(columns=['phone_number'], axis=1)
df.head()
```

Out[10]:

	state	account_length	area_code	international_plan	voice_mail_plan	number_vmail
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	

3.2 Feature Engineering

3.2.1 New features

- Creating new features such as "average monthly spend" and "customer engagement score" and replacing state codes with its names.

In [11]:
```
# creating average monthly spend feature
df["average_monthly_spend"] = (df["total_day_charge"] + df["total_eve_charge"] + df["

df.head()
```

Out[11]:

	state	account_length	area_code	international_plan	voice_mail_plan	number_vmail
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	

5 rows × 21 columns

In [12]:
```
# creating customer engagement score
df["customer_engagement_score"] = (df["total_day_minutes"] * 0.4 +
```

```
                            df["total_eve_minutes"] * 0.3 +
                            df["total_night_minutes"] * 0.1 +
                            df["total_intl_minutes"] * 0.2 +
                            df["number_vmail_messages"] * 0.2 -
                            df["customer_service_calls"] * 0.3)

df.head()
```

Out[12]:

	state	account_length	area_code	international_plan	voice_mail_plan	number_vmail
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	

5 rows × 22 columns

In [13]:

```
# reordering the churn column as the last index

# checking if the churn column is in the dataset
if 'churn' in df.columns:
    churn_column = df.pop("churn")
    # adding the column back to the dataset
    df["churn"] = churn_column

df.head()
```

Out[13]:

	state	account_length	area_code	international_plan	voice_mail_plan	number_vmail
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	

5 rows × 22 columns

In [14]:

```
# changing state codes with names
state_mapping = {
    "AL": "Alabama", "AK": "Alaska", "AZ": "Arizona", "AR": "Arkansas", "CA": "Califo
    "CO": "Colorado", "CT": "Connecticut", "DE": "Delaware", "FL": "Florida", "GA": "
    "HI": "Hawaii", "ID": "Idaho", "IL": "Illinois", "IN": "Indiana", "IA": "Iowa",
    "KS": "Kansas", "KY": "Kentucky", "LA": "Louisiana", "ME": "Maine", "MD": "Maryla
    "MA": "Massachusetts", "MI": "Michigan", "MN": "Minnesota", "MS": "Mississippi",
    "MT": "Montana", "NE": "Nebraska", "NV": "Nevada", "NH": "New Hampshire", "NJ": "
    "NM": "New Mexico", "NY": "New York", "NC": "North Carolina", "ND": "North Dakota
    "OK": "Oklahoma", "OR": "Oregon", "PA": "Pennsylvania", "RI": "Rhode Island", "SC
    "SD": "South Dakota", "TN": "Tennessee", "TX": "Texas", "UT": "Utah", "VT": "Verm
    "VA": "Virginia", "WA": "Washington", "WV": "West Virginia", "WI": "Wisconsin", "
}
```

In [15]:

```
# maping the names to the codes
df["state"] = df["state"].map(state_mapping)

df.head()
```

Out[15]:

	state	account_length	area_code	international_plan	voice_mail_plan	number_vı
0	Kansas	128	415	no	yes	
1	Ohio	107	415	no	yes	
2	New Jersey	137	415	no	no	
3	Ohio	84	408	yes	no	
4	Oklahoma	75	415	yes	no	

5 rows × 22 columns

In [16]:
```
# saving the cleaned dataset in another file
df.to_csv("cleaned_churn_dataset.csv", index=False)
```

3.3 EDA

3.3.1 Univariate Analysis

- A univariate analysis on the churn outcome

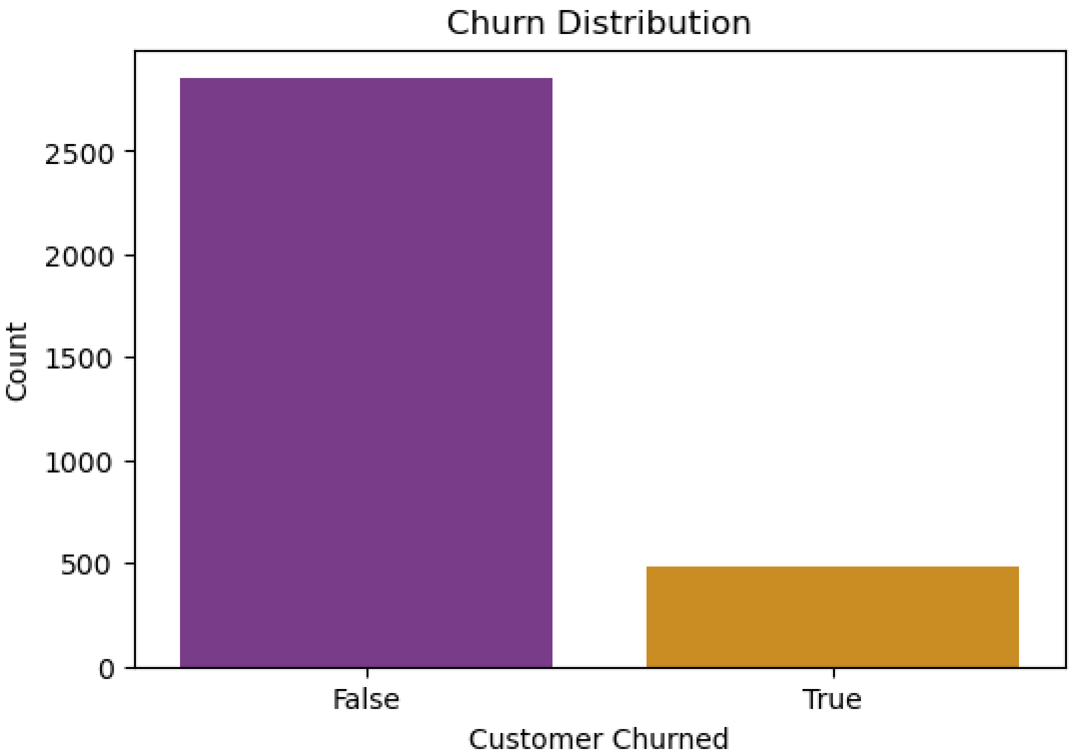
In [17]:
```
# checking the number of categories the column has
df['churn'].value_counts()
```

Out[17]:
```
churn
False    2850
True      483
Name: count, dtype: int64
```

In [18]:
```
# checking how the data is distributed and if it is imbalanced
df['churn'].value_counts(normalize=True) *100
```

Out[18]:
```
churn
False    85.508551
True     14.491449
Name: proportion, dtype: float64
```

In [19]:
```
# ploting the churn distribution using seaborn and plotlib
plt.figure(figsize=(6,4))

#ploting a count plot
sns.countplot(x=df["churn"], palette="CMRmap")

plt.title("Churn Distribution")
plt.xlabel("Customer Churned")
plt.ylabel("Count")
plt.show()
```

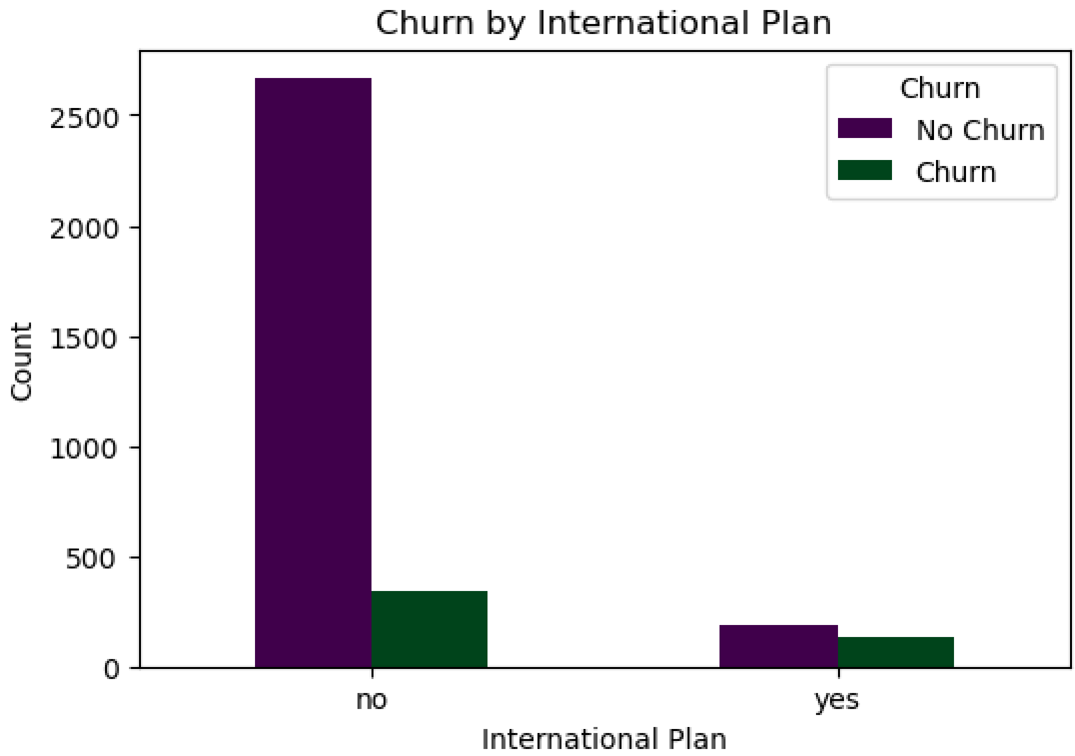


To Note: The data is clearly imbalanced with **85.51%** of the sample did **not churn** and **14.49%** of the sample **churned**.

3.3.2 Multivariate analysis

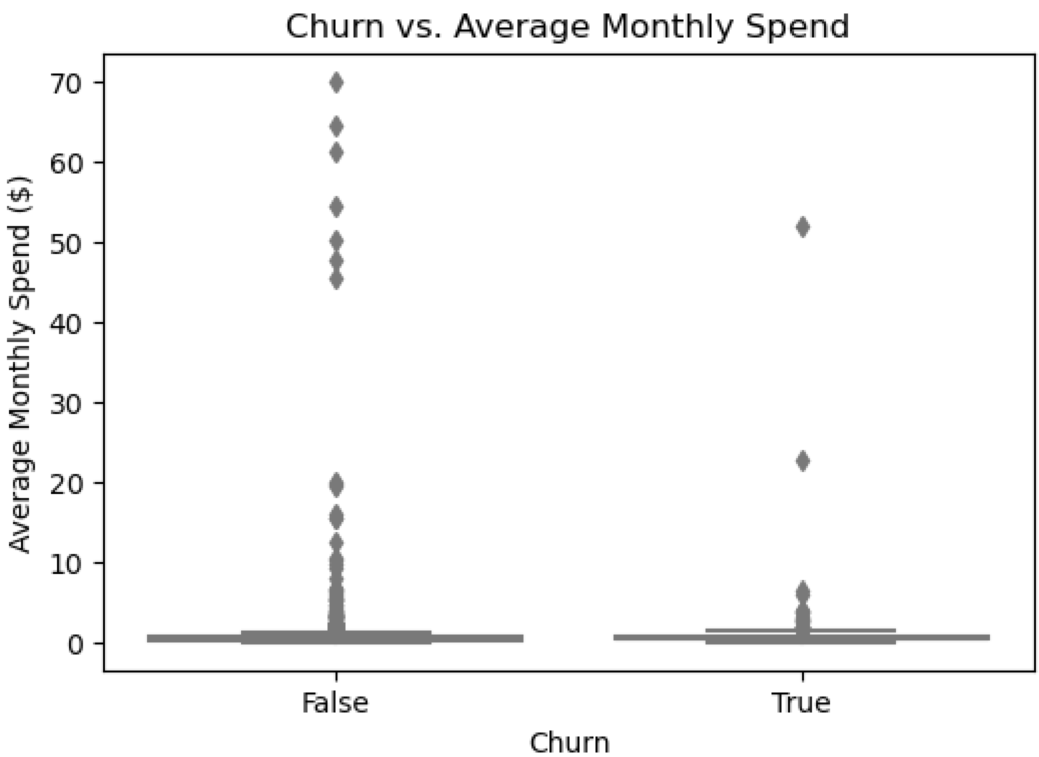
- Comparing how the **international plan** affected the churn of the customer
- Checking how the **average monthly spend** of a customer affected the churn
- Checking if the **state** a customer is from affects the churn

In [20]:
```
# Comparing how the international plan affected the churn of the customer
# Grouping the data
churn_counts = df.groupby("international_plan")["churn"].value_counts().unstack()

# Plotting
churn_counts.plot(kind="bar", figsize=(6,4), colormap="PRGn")

# Labels
plt.title("Churn by International Plan")
plt.xlabel("International Plan")
plt.ylabel("Count")
plt.legend(title="Churn", labels=["No Churn", "Churn"])
plt.xticks(rotation=0)
plt.show()
```



To Note: Customers with International plan still had a smaller amount of no churn to churn turnout. Though from the data imbalance we can definitely see that there was still a higher turnout of churn customers with an international plan

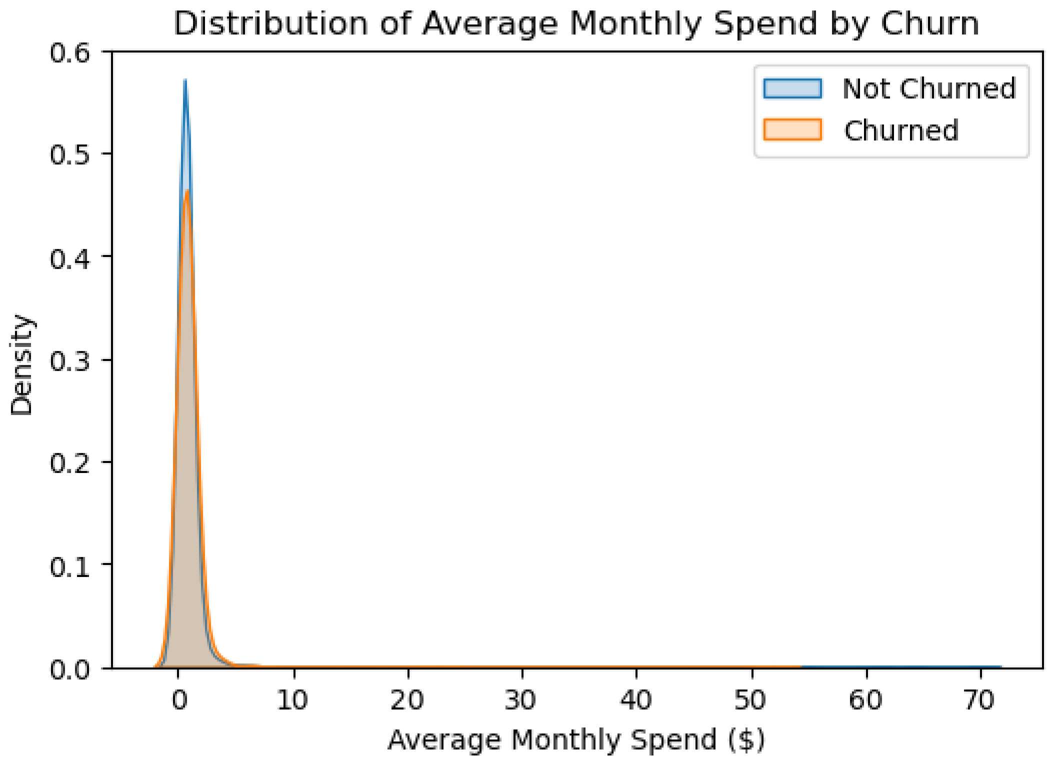
In [21]:
```
# Checking how the average monthly spend of a customer affected the churn
plt.figure(figsize=(6,4))
sns.boxplot(x="churn", y="average_monthly_spend", data=df, palette="coolwarm")

plt.title("Churn vs. Average Monthly Spend")
plt.xlabel("Churn")
plt.ylabel("Average Monthly Spend ($)")
plt.show()
```



In [22]:
```
plt.figure(figsize=(6,4))
sns.kdeplot(df[df["churn"] == 0]["average_monthly_spend"], label="Not Churned", shade
sns.kdeplot(df[df["churn"] == 1]["average_monthly_spend"], label="Churned", shade=Tru
```

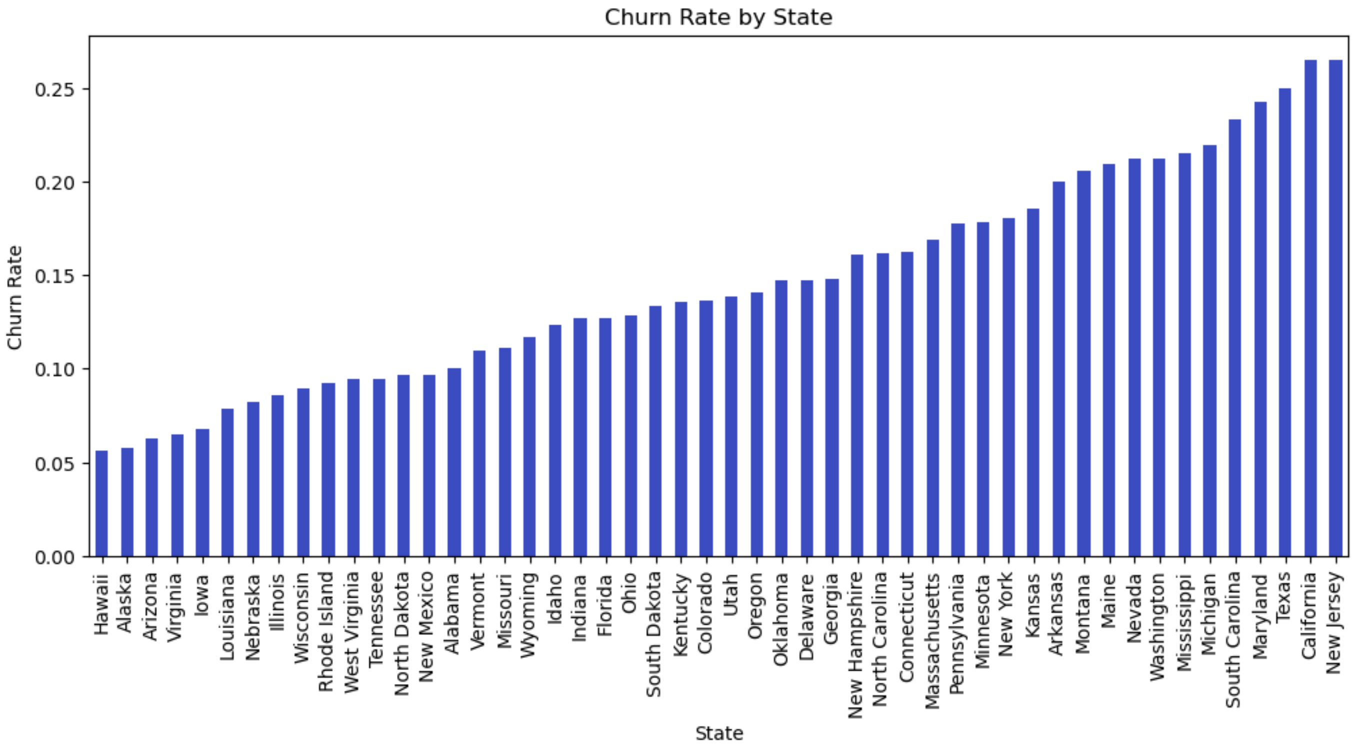
```python
plt.title("Distribution of Average Monthly Spend by Churn")
plt.xlabel("Average Monthly Spend ($)")
plt.ylabel("Density")
plt.legend()
plt.show()
```

```
/tmp/ipykernel_99623/1705903778.py:2: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(df[df["churn"] == 0]["average_monthly_spend"], label="Not Churned", shad
e=True)
/home/bev/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119: FutureWarni
ng: use_inf_as_na option is deprecated and will be removed in a future version. Conver
t inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/tmp/ipykernel_99623/1705903778.py:3: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(df[df["churn"] == 1]["average_monthly_spend"], label="Churned", shade=Tr
ue)
/home/bev/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119: FutureWarni
ng: use_inf_as_na option is deprecated and will be removed in a future version. Conver
t inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```

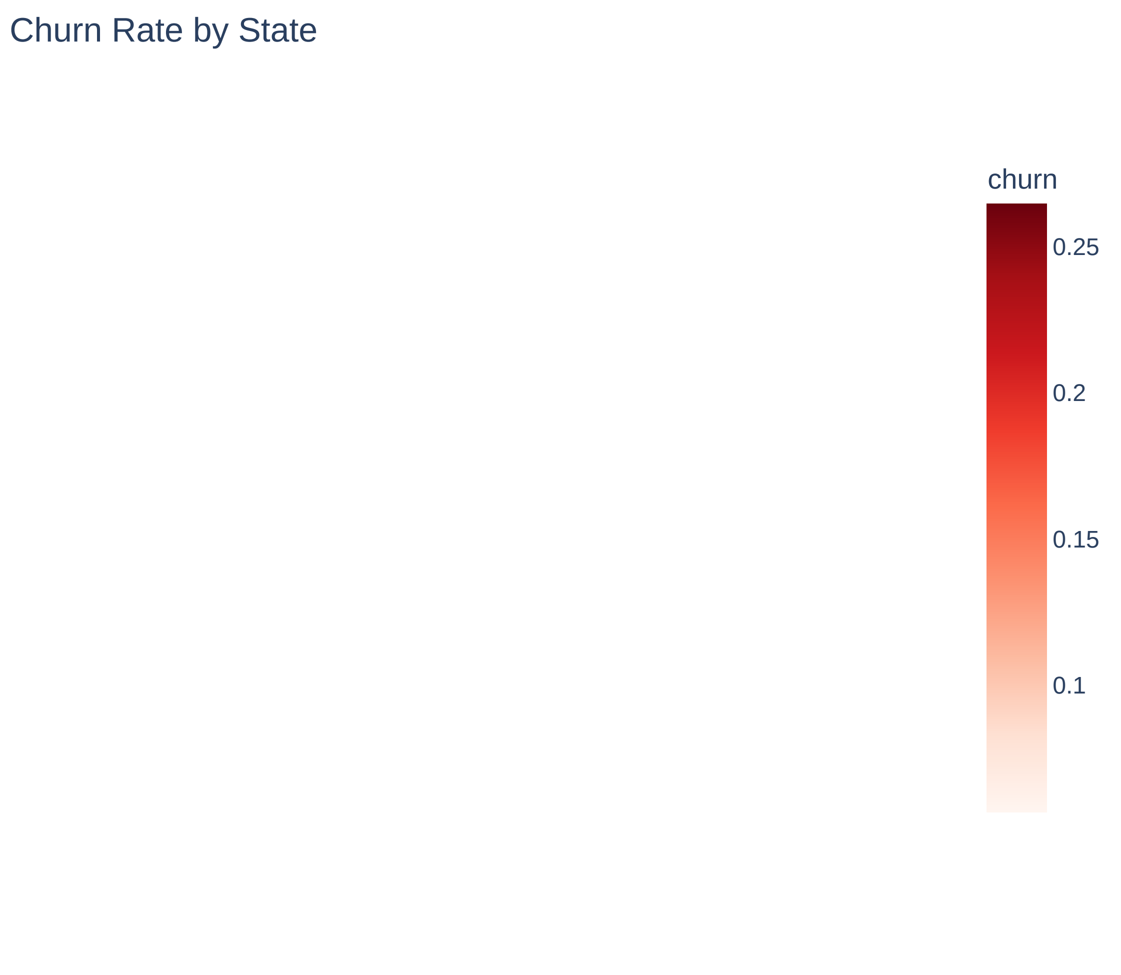


To Note: The average monthly spend does not really affect the churn rate of a customer.

In [23]:
```python
# Checking if the state a customer is from affects the churn
churn_rate = df.groupby("state")["churn"].mean().sort_values()

plt.figure(figsize=(12,5))
churn_rate.plot(kind="bar", colormap="coolwarm")

plt.title("Churn Rate by State")
plt.xlabel("State")
plt.ylabel("Churn Rate")
plt.xticks(rotation=90)
plt.show()
```



In [24]:
```
import plotly.express as px

# Compute churn rate per state
state_churn = df.groupby("state")["churn"].mean().reset_index()

# Plot map
fig = px.choropleth(state_churn,
                    locations="state",
                    locationmode="USA-states",
                    color="churn",
                    color_continuous_scale="Reds",
                    scope="usa",
                    title="Churn Rate by State")

fig.show()
```



To Note: States like **Carlifornia, Texas and New Jersey** have a highest churn rate due to different factors. Could be competition from other service networks, service quality, pricing differences etc.

In [25]:
```
# Lets do a hypothesis test to see if the state significantly affects the churn rate
# Create a contingency table (frequency of churn per state)
contingency_table = pd.crosstab(df["state"], df["churn"])

alpha = 0.5
# Perform Chi-Square Test
chi2, p, dof, expected = chi2_contingency(contingency_table)

print(f"Chi-Square Test p-value: {p}")

if p < alpha:
    print('The churn rate significantly depends on the State the customer is from.')
else:
    print('The churn rate does not significantly depend on the State the customer is
```

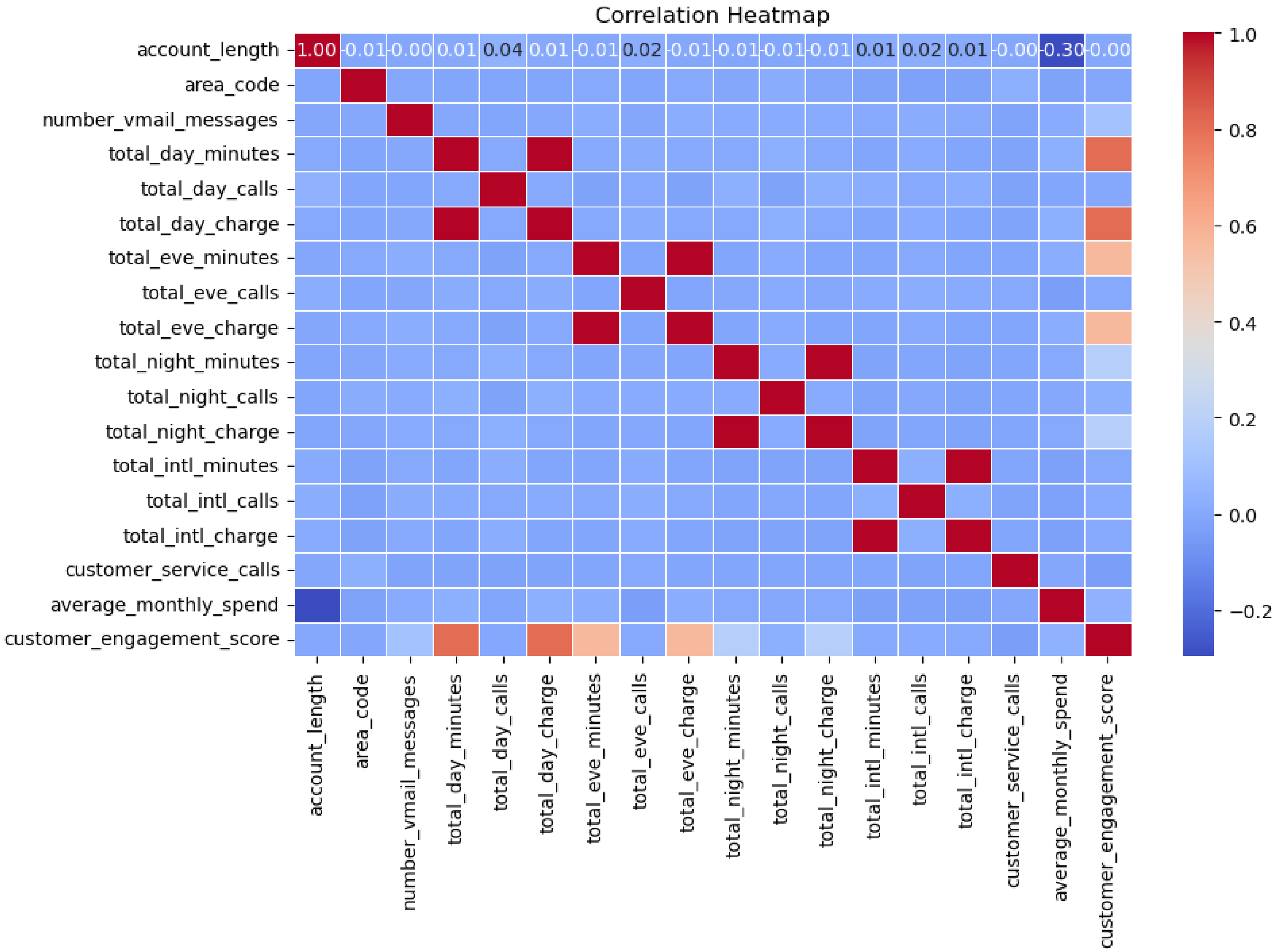
```
Chi-Square Test p-value: 0.0024733134842029442
The churn rate significantly depends on the State the customer is from.
```

3.3.3 Multivariate Analysis

- Correlation between the features

In [26]:
```
# heat map to show the correlation of the features
# Select only numeric columns
numeric_df = df.select_dtypes(include=["number"])

# Compute correlation
plt.figure(figsize=(10,6))
```

```
sns.heatmap(numeric_df.corr(), annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5

plt.title("Correlation Heatmap")
plt.show()
```



To Note: Features like `total_day_minutes` , `total_day_charge` and `total_eve_charge` have a high correlation with the customer engagement which means it is highly significant to the churn turn out.

4. Modeling

4.1 **Rationale for Using Machine Learning**

Machine learning is chosen for this analysis because:

- **Pattern Recognition:** Traditional statistical methods may not effectively capture the complex relationships between customer behaviors and churn.
- **Predictive Power:** ML models can generalize patterns from historical data to predict churn probabilities for new customers.
- **Feature Interaction Handling:** Advanced models such as Random Forest and Gradient Boosting can capture interactions between multiple features, something simpler analysis might miss.

This dataset, `cleaned_churn_dataset.csv` , contains a mix of categorical and numerical variables, making supervised learning a suitable approach for classification. By iterating between different models, we can assess their effectiveness and refine the approach.

In [27]:
```
# importing libraries for the model
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchC
```

```python
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from imblearn.over_sampling import SMOTE
```

Steps to building and training our model

Step 1: Define the X(predictive) and y(target) variables

- We have to encode all the categorical features in our dataset before training.
- Churn column will be the y variable for our case.

In [28]:
```python
# selecting columns with categorical values
categorical_columns = df.select_dtypes(include=['object']).columns

categorical_columns
```

Out[28]:
```
Index(['state', 'international_plan', 'voice_mail_plan'], dtype='object')
```

In [29]:
```python
# feature encoding with one hot encoding
# initializing the encoder
encoder = OneHotEncoder(drop="first", sparse_output=False)

# fiting and transforming the columns
encoded_col = encoder.fit_transform(df[categorical_columns])

encoded_col
```

Out[29]:
```
array([[0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 1.]])
```

In [30]:
```python
# we then convert the array to fit our dataset
encoded_df = encoded_df = pd.DataFrame(encoded_col, columns=encoder.get_feature_names

# we define our df again by concatinating with the encoded dataframe
df = pd.concat([df.drop(columns=categorical_columns), encoded_df], axis=1)

df.head()
```

Out[30]:

	account_length	area_code	number_vmail_messages	total_day_minutes	total_day_ca
0	128	415	25	265.1	1
1	107	415	26	161.6	1
2	137	415	0	243.4	1
3	84	408	0	299.4	
4	75	415	0	166.7	1

5 rows × 71 columns

In [31]:
```python
# getting the churn column as our last column in our dataframe
if 'churn' in df.columns:
```

```
    churn_column = df.pop("churn")
    df["churn"] = churn_column

df.head()
```

Out[31]:

	account_length	area_code	number_vmail_messages	total_day_minutes	total_day_ca
0	128	415	25	265.1	1
1	107	415	26	161.6	1
2	137	415	0	243.4	1
3	84	408	0	299.4	
4	75	415	0	166.7	1

5 rows × 71 columns

In [32]:

```
# converting our churn feature to a numerical type
df["churn"] = df["churn"].astype(int)

df.head()
```

Out[32]:

	account_length	area_code	number_vmail_messages	total_day_minutes	total_day_ca
0	128	415	25	265.1	1
1	107	415	26	161.6	1
2	137	415	0	243.4	1
3	84	408	0	299.4	
4	75	415	0	166.7	1

5 rows × 71 columns

In [33]:

```
# now that all our features and target variables are numerical, we can initialize our
X = df.drop(columns=['churn'], axis=1)
y = df['churn']
```

In [34]:

```
X.shape
```

Out[34]: (3333, 70)

In [35]:

```
y.shape
```

Out[35]: (3333,)

Step 2: Spliting the data using train_test_split

- Spliting the data to train and testing data.
- Handle the class imbalance that we discovered when cleaning using SMOTE.

The dataset was **highly imbalanced** with **85%** of the sample being classified as **not churned**.

- Transforming the data using a standard scaler before using it to train and test the model.

In [36]:

```
# using train_test_split to split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
```

In [37]:
```
# handling class imbalance using smote
#initializing smote
smote = SMOTE(random_state=42)

# applying smote to the traiing data
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
```

In [38]:
```
# checking class distribution after balancing
pd.Series(y_train_smote).value_counts()
```

Out[38]:
```
churn
0    2284
1    2284
Name: count, dtype: int64
```

In [39]:
```
# fitting and transforming the training and testing data
# initializing the scaler
scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train_smote)
X_test_scaled = scaler.transform(X_test)
```

Step 3: Bulding and training our models

- Build a logistic regression model and train it.
- Build a random forest model and train it.
- Build a Decision Tree Classifier model and train it.
- Build a Gradient boost optimized model and train it.

Selected Models and Justification

1. Logistic Regression (Baseline Model)

- **Why?**
  - It serves as a baseline for comparison.
  - It is simple, interpretable, and efficient.
  - It outputs probability scores, helping assess churn likelihood.
- **Limitations:**
  - Assumes linear relationships between features.
  - Might underperform if the data is highly non-linear.

2. Decision Tree Classifier (Simple Non-Linear Model)

- **Why?**
  - Captures non-linear relationships better than Logistic Regression.
  - It provides clear feature importance, aiding business insights.
  - Simple to interpret.
- **Limitations:**
  - Prone to overfitting unless tuned properly.
  - Less robust compared to ensemble methods.

3. Random Forest Classifier (Ensemble Learning)

- **Why?**
  - Reduces overfitting by averaging multiple Decision Trees.
  - Handles missing data and outliers better than a single tree.

    - Useful for feature importance ranking.
- **Limitations:**
    - More computationally expensive than a single Decision Tree.
    - Slightly less interpretable compared to simpler models.

4. Gradient Boosting (XGBoost)

- **Why?**
    - Best suited for structured data.
    - Reduces bias and variance by iteratively improving predictions.
    - Highly tunable for maximizing performance.
- **Limitations:**
    - Requires careful tuning for best results.
    - Computationally intensive.

1. Logistic Regression Model

In [40]:
```
# initializing our model
lrm = LogisticRegression(random_state=42)

# training our model
logistic_reg = lrm.fit(X_train_scaled, y_train_smote)

logistic_reg
```

Out[40]:
```
LogisticRegression
LogisticRegression(random_state=42)
```

Checking our Logistic Regression Model performance

In [41]:
```
# predicting our target using X_test data
lrm_y_predicted = logistic_reg.predict(X_test_scaled)

lrm_y_predicted.shape
```

Out[41]: (667,)

In [42]:
```
# checking performance using Accuracy score
lrm_accuracy = accuracy_score(y_test, lrm_y_predicted)

# output of the accuracy
print(f'The accuracy of the Logistic Regression Model is at {lrm_accuracy * 100} %')
```

```
The accuracy of the Logistic Regression Model is at 77.06146926536732 %
```

In [43]:
```
# checking performance using the confusion matrix
lrm_confusion_matrix = confusion_matrix(y_test, lrm_y_predicted)

# printing the infor in an understandable manner
lrm_cm_df = pd.DataFrame(lrm_confusion_matrix,
                         index=["Actual No Churn", "Actual Churn"],
                         columns=["Predicted No Churn", "Predicted Churn"])

print(lrm_cm_df)
```

```
                 Predicted No Churn  Predicted Churn
Actual No Churn                 441              125
Actual Churn                     28               73
```

In [44]:
```
# checking the performance using the classification report
lrm_cr = classification_report(y_test, lrm_y_predicted)

print(lrm_cr)
```

```
              precision    recall  f1-score   support

           0       0.94      0.78      0.85       566
           1       0.37      0.72      0.49       101

    accuracy                           0.77       667
   macro avg       0.65      0.75      0.67       667
weighted avg       0.85      0.77      0.80       667
```

Overally we can say that this model performed okay.

2. Decision Tree Classifier Model

In [45]:
```
# initializing our model
dtc = DecisionTreeClassifier(random_state=42)

# training our model
dtc_model = dtc.fit(X_train_scaled, y_train_smote)

dtc_model
```

Out[45]:
```
▾ DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

Checking our Decision Tree Classifier Model performance

In [46]:
```
# predicting our target using X_test data
dtc_y_predicted = dtc_model.predict(X_test_scaled)

dtc_y_predicted.shape
```

Out[46]:
```
(667,)
```

In [47]:
```
# checking performance using Accuracy score
dtc_accuracy = accuracy_score(y_test, dtc_y_predicted)

# output of the accuracy
print(f'The accuracy of the Decision Tree Model is at {dtc_accuracy * 100} %')
```

```
The accuracy of the Decision Tree Model is at 91.00449775112443 %
```

In [48]:
```
# checking performance using the confusion matrix
dtc_confusion_matrix = confusion_matrix(y_test, dtc_y_predicted)

# printing the infor in an understandable manner
dtc_cm_df = pd.DataFrame(dtc_confusion_matrix,
                     index=["Actual No Churn", "Actual Churn"],
                     columns=["Predicted No Churn", "Predicted Churn"])

print(dtc_cm_df)
```

```
                 Predicted No Churn  Predicted Churn
Actual No Churn                 528               38
Actual Churn                     22               79
```

In [49]:
```
# checking the performance using the classification report
dtc_cr = classification_report(y_test, dtc_y_predicted)
```

```
print(dtc_cr)
```

```
              precision    recall  f1-score   support

           0       0.96      0.93      0.95       566
           1       0.68      0.78      0.72       101

    accuracy                           0.91       667
   macro avg       0.82      0.86      0.84       667
weighted avg       0.92      0.91      0.91       667
```

To Note: The Decision Tree Model performed better than the Logistic Regression Model.

3. Random Forest Classifier (Ensemble Learning)

In [50]:

```
# initializing the model
rfc = RandomForestClassifier()

# training the model
rfc_model = rfc.fit(X_train_scaled, y_train_smote)

rfc_model
```

Out[50]:

```
▾ RandomForestClassifier
RandomForestClassifier()
```

Checking the Random Forest Classifier Model Performance

In [51]:

```
# predicting our target using X_test data
rfc_y_predicted = rfc_model.predict(X_test_scaled)

rfc_y_predicted.shape
```

Out[51]: (667,)

In [52]:

```
# checking performance using Accuracy score
rfc_accuracy = accuracy_score(y_test, rfc_y_predicted)

# output of the accuracy
print(f'The accuracy of the Random Forest Model is at {rfc_accuracy * 100} %')
```

```
The accuracy of the Random Forest Model is at 95.05247376311844 %
```

In [53]:

```
# checking performance using the confusion matrix
rfc_confusion_matrix = confusion_matrix(y_test, rfc_y_predicted)

# printing the infor in an understandable manner
rfc_cm_df = pd.DataFrame(rfc_confusion_matrix,
                     index=["Actual No Churn", "Actual Churn"],
                     columns=["Predicted No Churn", "Predicted Churn"])

print(rfc_cm_df)
```

```
                 Predicted No Churn  Predicted Churn
Actual No Churn                 558                8
Actual Churn                     25               76
```

In [55]:

```
# checking the performance using the classification report
rfc_cr = classification_report(y_test, rfc_y_predicted)

print(rfc_cr)
```

```
              precision    recall  f1-score   support

           0       0.96      0.99      0.97       566
           1       0.90      0.75      0.82       101

    accuracy                           0.95       667
   macro avg       0.93      0.87      0.90       667
weighted avg       0.95      0.95      0.95       667
```

To Note: The RFC performed a little better than the Decision Tree Classifier Model.

4. Gradient Boosting Model with XGBoost

In [56]:
```python
# initializing our model
xgb = XGBClassifier( objective="binary:logistic", random_state=42)

#training our model using the training data
xgb_model = xgb.fit(X_train_scaled, y_train_smote)

xgb_model
```

Out[56]:

```
XGBClassifier

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=Non
e,
              enable_categorical=False, eval_metric=None, feature_types=Non
e,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=Non
e,
```

In [57]:
```python
# getting the default parameters used in this model
print(xgb_model.get_params())
```

```
{'objective': 'binary:logistic', 'base_score': None, 'booster': None, 'callbacks': Non
e, 'colsample_bylevel': None, 'colsample_bynode': None, 'colsample_bytree': None, 'dev
ice': None, 'early_stopping_rounds': None, 'enable_categorical': False, 'eval_metric':
None, 'feature_types': None, 'gamma': None, 'grow_policy': None, 'importance_type': No
ne, 'interaction_constraints': None, 'learning_rate': None, 'max_bin': None, 'max_cat_
threshold': None, 'max_cat_to_onehot': None, 'max_delta_step': None, 'max_depth': Non
e, 'max_leaves': None, 'min_child_weight': None, 'missing': nan, 'monotone_constraint
s': None, 'multi_strategy': None, 'n_estimators': None, 'n_jobs': None, 'num_parallel_
tree': None, 'random_state': 42, 'reg_alpha': None, 'reg_lambda': None, 'sampling_meth
od': None, 'scale_pos_weight': None, 'subsample': None, 'tree_method': None, 'validate
_parameters': None, 'verbosity': None}
```

Checking the XGB model performance

In [58]:
```python
# predicting our target using X_test data
xgb_y_predicted = xgb_model.predict(X_test_scaled)

xgb_y_predicted.shape
```

Out[58]: (667,)

In [59]:
```python
# checking performance using Accuracy score
xgb_accuracy = accuracy_score(y_test, xgb_y_predicted)
```

```python
# output of the accuracy
print(f'The accuracy of the XGB Model is at {xgb_accuracy * 100} %')
```

```
The accuracy of the XGB Model is at 96.10194902548726 %
```

In [60]:
```python
# checking performance using the confusion matrix
xgb_confusion_matrix = confusion_matrix(y_test, xgb_y_predicted)

# printing the infor in an understandable manner
xgb_cm_df = pd.DataFrame(xgb_confusion_matrix,
                         index=["Actual No churn", "Actual Churn"],
                         columns=["Predicted No churn", "Predicted Churn"])

print(xgb_cm_df)
```

```
                 Predicted No churn  Predicted Churn
Actual No churn                 561                5
Actual Churn                     21               80
```

In [61]:
```python
# checking the performance using the classification report
xgb_cr = classification_report(y_test, xgb_y_predicted)

print(xgb_cr)
```

```
              precision    recall  f1-score   support

           0       0.96      0.99      0.98       566
           1       0.94      0.79      0.86       101

    accuracy                           0.96       667
   macro avg       0.95      0.89      0.92       667
weighted avg       0.96      0.96      0.96       667
```

To Note: The XGBoost Model performed better than all the other models with an F1 score of 86%.

Step 4: Results and Evaluation of the Models

To compare model performance, we use the following metrics:

1. Accuracy – Overall Correctness

Formula:

```
Accuracy = (TP + TN) / (TP + TN + FP + FN)
```

- Measures how often the model makes correct predictions.
- Useful when the dataset is balanced.
- **Limitation:** Can be misleading for imbalanced data (e.g., if churn cases are rare).

2. Precision – How Many Predicted Churns Are Correct?

Formula:

```
Precision = TP / (TP + FP)
```

- Measures how many of the predicted churners actually churned.
- High precision reduces false positives (incorrectly flagging loyal customers as churners).
- Useful when unnecessary retention efforts are costly.

3. Recall (Sensitivity) – How Many Actual Churners Were Caught?

Formula:

```
Recall = TP / (TP + FN)
```

- Measures how many actual churners the model successfully identified.
- High recall reduces false negatives (failing to identify actual churners).
- Useful when retaining every possible churner is a priority.

4. F1-Score – Balancing Precision and Recall

Formula:

```
F1 = 2 * (Precision * Recall) / (Precision + Recall)
```

- A single number that balances precision and recall.
- Useful when both false positives and false negatives have business implications.

5. Business Implications of These Metrics

Metric	High Value Meaning	Low Value Meaning	Business Impact
Accuracy	Most predictions are correct	Model makes many errors	Can be misleading in imbalanced data
Precision	Few false positives	Many false positives	Avoids wasting resources on unnecessary retention efforts
Recall	Most churners are caught	Many churners missed	Ensures high-risk customers are targeted for retention
F1-Score	Balanced precision & recall	One is too low	Helps balance trade-offs effectively

Choosing the Right Metric for Churn Prediction

- **Since both minimizing retention costs and maximizing customer retention are key, we will focus on the F1-Score.**
- **A balanced approach ensures that we reduce false positives (unnecessary retention efforts) while also reducing false negatives (missed churners).**
- **Precision and Recall will still be analyzed to fine-tune the model's performance.**
- **Accuracy is useful only if the dataset is balanced and since ours was not, it is not very useful**

In [85]:
```
# evaluating the performance of the models

# Function to extract precision, recall, F1-score, and accuracy from classification r
def get_metrics(y_true, y_pred):
    report = classification_report(y_true, y_pred, output_dict=True)

    accuracy = report["accuracy"]
    precision = report["1"]["precision"]  # Churn class (assuming 1 = churn)
    recall = report["1"]["recall"]
    f1_score = report["1"]["f1-score"]

    return accuracy, precision, recall, f1_score

# Get metrics for all models
log_reg_acc, log_reg_prec, log_reg_rec, log_reg_f1 = get_metrics(y_test, lrm_y_predic
```

```
rf_acc, rf_prec, rf_rec, rf_f1 = get_metrics(y_test, rfc_y_predicted)
dt_acc, dt_prec, dt_rec, dt_f1 = get_metrics(y_test, dtc_y_predicted)
xgb_acc, xgb_prec, xgb_rec, xgb_f1 = get_metrics(y_test, xgb_y_predicted)


# Creating a DataFrame to compare models
model_performance = pd.DataFrame({
    'Model': ['Logistic Regression', 'Random Forest', 'Decision Tree', 'XGBoost'],
    'Accuracy': [log_reg_acc, rf_acc, dt_acc, xgb_acc],
    'Precision': [log_reg_prec, rf_prec, dt_prec, xgb_prec],
    'Recall': [log_reg_rec, rf_rec, dt_rec, xgb_rec],
    'F1-Score': [log_reg_f1, rf_f1, dt_f1, xgb_f1]
})

# Sorting by F1-score (since balancing precision & recall is key)
model_performance = model_performance.sort_values(by="F1-Score", ascending=False)

# Display the table
print("Model Performance Comparison (Sorted by F1-Score):")
display(model_performance)
```

```
Model Performance Comparison (Sorted by F1-Score):
```

	Model	Accuracy	Precision	Recall	F1-Score
3	XGBoost	0.961019	0.941176	0.792079	0.860215
1	Random Forest	0.950525	0.904762	0.752475	0.821622
2	Decision Tree	0.910045	0.675214	0.782178	0.724771
0	Logistic Regression	0.770615	0.368687	0.722772	0.488294

Step 5: Model Selection

After evaluating all models, XGBoost emerged as the best model based on key performance metrics, particularly the **F1-Score at 86.02%**, which balances precision and recall.

Key Reasons for Choosing XGBoost:

- **Best balance of precision and recall**, ensuring we **minimize retention costs** while **maximizing customer retention**.
- **Handles feature importance well**, providing insights into what influences churn.
- **More robust to overfitting** compared to Decision Trees and Random Forests.
- **Optimized for structured data**, making it a great choice for our dataset.

Thus, **XGBoost will be our final model** for predicting churn.

Step 6: Model optimization

Hyperparameter tuning using Grid search to better the performance of our Model.

In [72]:
```
# Define the model
xgb_model_opt = XGBClassifier(eval_metric="logloss", learning_rate=0.01)
```

In [73]:
```
# Define the hyperparameter grid
param_grid = {
    'max_depth': [3, 5, 7],
    'n_estimators': [100, 300, 500],
    'min_child_weight': [1, 3, 5],
    'subsample': [0.6, 0.8, 1.0],
```

```
    'colsample_bytree': [0.6, 0.8, 1.0],
    'gamma': [0, 1, 5]
}
```

In [86]:
```
# # Performing the Grid Search algorithm
# grid_search = GridSearchCV(xgb_model_opt, param_grid, cv=5, scoring="accuracy", n_j

# # training with grid search
# grid_search.fit(X_train_scaled, y_train_smote)
```

In [75]:
```
# Print the best parameters
print("Best Parameters:", grid_search.best_params_)

# Use the best model
opt_xgb = grid_search.best_estimator_
```

```
Best Parameters: {'colsample_bytree': 0.6, 'gamma': 0, 'max_depth': 7, 'min_child_weig
ht': 1, 'n_estimators': 500, 'subsample': 1.0}
```

To Note: The best parameters for tuning the XGB are:

- 'colsample_bytree': 0.6
- 'gamma': 0
- 'max_depth': 7
- 'min_child_weight': 1
- 'n_estimators': 500
- 'subsample': 1.0

In [80]:
```
# predicting y variables using the optimized model
gs_xgb_y_pred = opt_xgb.predict(X_test_scaled)
```

In [81]:
```
# checking the performance of the model using the F1 score
xgb_opt_cr = classification_report(y_test, gs_xgb_y_pred)

print(xgb_opt_cr)
```

```
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       566
           1       0.94      0.80      0.87       101

    accuracy                           0.96       667
   macro avg       0.95      0.90      0.92       667
weighted avg       0.96      0.96      0.96       667
```

To Note: After hyperparameter tuning, we found that there was no much difference in the F1-score. The grid search xgb model performed even better than the baseline model with an F1-score of **87%** and the baseline model had an F1 score of **86%**.

In [65]:
```
#hyperparameter tuning with Randomized search
# define our parameters
param_dist = {
    'max_depth': np.arange(3, 10),
    'learning_rate': np.linspace(0.01, 0.3, 10),
    'n_estimators': np.arange(100, 1000, 100),
    'min_child_weight': np.arange(1, 10),
    'subsample': np.linspace(0.5, 1.0, 5),
    'colsample_bytree': np.linspace(0.5, 1.0, 5),
    'gamma': np.linspace(0, 5, 5)
}
```

```
random_search = RandomizedSearchCV(xgb_model, param_dist, n_iter=20, cv=5, scoring='a
random_search.fit(X_train, y_train)
```

Out[65]:

```
RandomizedSearchCV
estimator: XGBClassifier
XGBClassifier
```

In [66]:

```
# printing the best parameters
print("Best Parameters:", random_search.best_params_)

opt_xgb2 = random_search.best_estimator_
```

```
Best Parameters: {'subsample': 0.875, 'n_estimators': 400, 'min_child_weight': 1, 'max
_depth': 9, 'learning_rate': 0.01, 'gamma': 3.75, 'colsample_bytree': 0.5}
```

In [67]:

```
# checking the F1 score of the optimized model
# predicting y values first
opt_xgb2_y_pred = opt_xgb2.predict(X_test_scaled)
```

In [68]:

```
opt_xgb2_cr = classification_report(y_test, opt_xgb2_y_pred)

print(opt_xgb2_cr)
```

```
              precision    recall  f1-score   support

           0       0.87      0.93      0.90       566
           1       0.40      0.25      0.30       101

    accuracy                           0.83       667
   macro avg       0.64      0.59      0.60       667
weighted avg       0.80      0.83      0.81       667
```

To Note: The model tuned using random search performed even poorly with an F1 score of **30%** than the one tuned with Grid search.

In [84]:

```
# evaluating the performance of the models

# Function to extract precision, recall, F1-score, and accuracy from classification r
def get_metrics(y_true, y_pred):
    report = classification_report(y_true, y_pred, output_dict=True)

    accuracy = report["accuracy"]
    precision = report["1"]["precision"]  # Churn class (assuming 1 = churn)
    recall = report["1"]["recall"]
    f1_score = report["1"]["f1-score"]

    return accuracy, precision, recall, f1_score

# Get metrics for all models
log_reg_acc, log_reg_prec, log_reg_rec, log_reg_f1 = get_metrics(y_test, lrm_y_predic
rf_acc, rf_prec, rf_rec, rf_f1 = get_metrics(y_test, rfc_y_predicted)
dt_acc, dt_prec, dt_rec, dt_f1 = get_metrics(y_test, dtc_y_predicted)
xgb_acc, xgb_prec, xgb_rec, xgb_f1 = get_metrics(y_test, xgb_y_predicted)
gs_xgb_acc, gs_xgb_prec, gs_xgb_rec, gs_xgb_f1 = get_metrics(y_test, gs_xgb_y_pred)


# Creating a DataFrame to compare models
model_performance = pd.DataFrame({
    'Model': ['Logistic Regression', 'Random Forest', 'Decision Tree', 'XGBoost', 'Op
    'Accuracy': [log_reg_acc, rf_acc, dt_acc, xgb_acc, gs_xgb_acc],
    'Precision': [log_reg_prec, rf_prec, dt_prec, xgb_prec, gs_xgb_prec],
```

```
    'Recall': [log_reg_rec, rf_rec, dt_rec, xgb_rec, gs_xgb_rec],
    'F1-Score': [log_reg_f1, rf_f1, dt_f1, xgb_f1, gs_xgb_f1]
})

# Sorting by F1-score (since balancing precision & recall is key)
model_performance = model_performance.sort_values(by="F1-Score", ascending=False)

# Display the table
print("Model Performance Comparison (Sorted by F1-Score):")
display(model_performance)
```

```
Model Performance Comparison (Sorted by F1-Score):
```

	Model	Accuracy	Precision	Recall	F1-Score
4	Optimized_XGB	0.962519	0.941860	0.801980	0.866310
3	XGBoost	0.961019	0.941176	0.792079	0.860215
1	Random Forest	0.950525	0.904762	0.752475	0.821622
2	Decision Tree	0.910045	0.675214	0.782178	0.724771
0	Logistic Regression	0.770615	0.368687	0.722772	0.488294

In Summary

Rationale & Results

- Machine learning, specifically XGBoost, is preferred due to its ability to capture complex patterns compared to simpler models like logistic regression (F1 score of 48.83%).
- Optimized XGBoost model achieved an F1 score of 86.63%.
- Random search tuning reduced performance to 30%, highlighting risks of improper hyperparameter selection.
- Grid search tuning maintained a strong F1 score of 86.63%, proving structured optimization is essential.
- XGBoost outperformed Random Forest (F1 score of 82.16%) and Decision Tree (F1 score 72.48%), making it the best deployment candidate.

Limitations & Recommendations

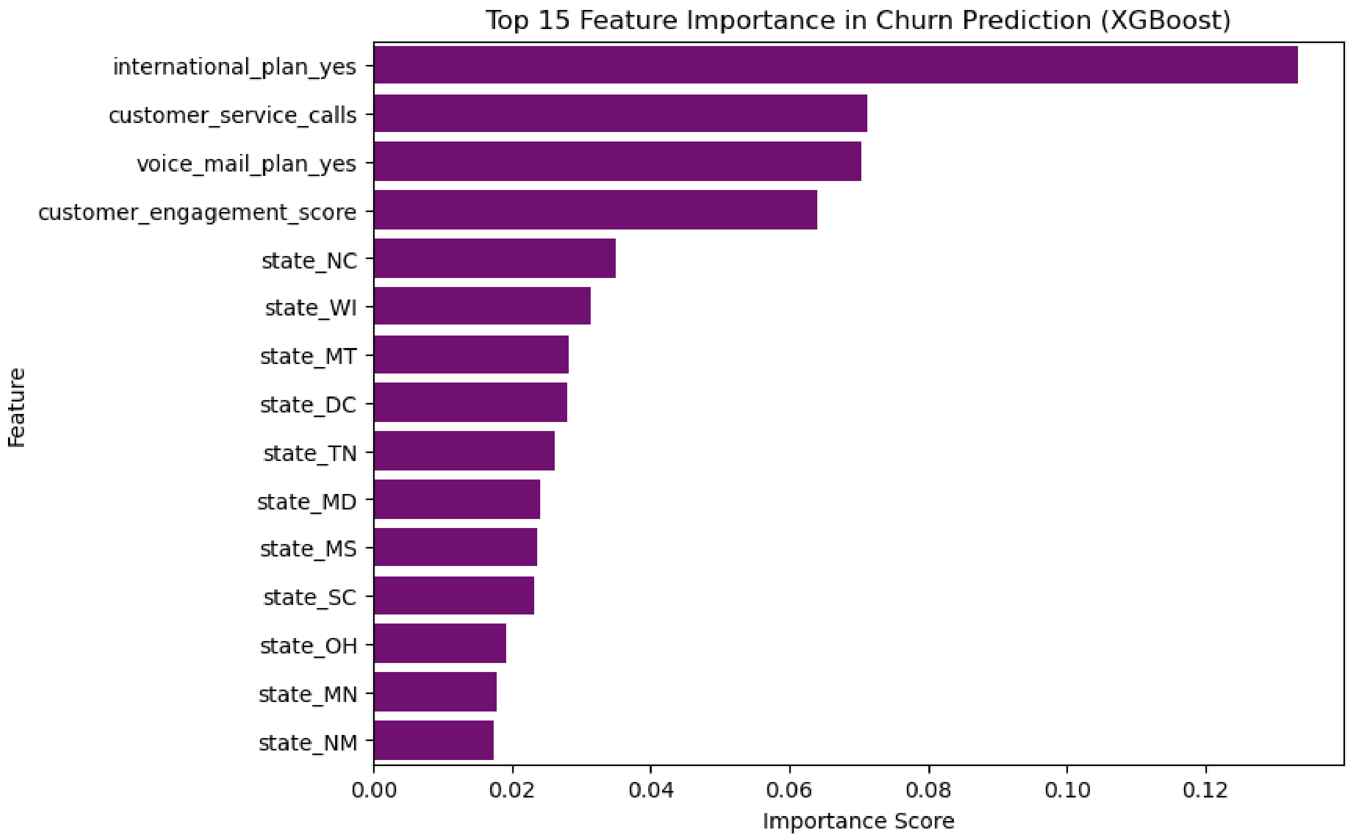
- XGBoost had a recall of 80% suggests 20% of positive cases are misclassified, which could impact business decisions.
- Tree-based models require periodic monitoring to prevent performance degradation.
- Recommended actions:
  - Deploy XGBoost with the tuned parameters.
  - Improve recall using threshold tuning or cost-sensitive learning.
  - Maintain continuous validation for long-term model effectiveness.

Step 7: Feature Impotance

- We need to know which features are most impotant when building our model for future reference in data collection and business implementation.

In [93]:
```
# Extract feature importances from XGBoost
feature_importance = xgb_model.feature_importances_
features = X_train.columns
```

```python
# Create DataFrame and sort by importance
feat_imp_df = pd.DataFrame({'Feature': features, 'Importance': feature_importance})
feat_imp_df = feat_imp_df.sort_values(by="Importance", ascending=False).head(15)  # S

# Plot feature importance
plt.figure(figsize=(8, 6))
sns.barplot(x="Importance", y="Feature", data=feat_imp_df, color='purple')
plt.title("Top 15 Feature Importance in Churn Prediction (XGBoost)")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.show()
```



Feature Importance Insights 📊

From the feature importance analysis, we can draw several key conclusions:

1. International Plan Influence

- Customers with an **international plan** are the strongest predictor of churn.
- This suggests that users who opt for international plans may have **higher expectations** or **alternative service options**, leading to churn.

2. Customer Service Calls & Voicemail Plan

- A **high number of customer service calls** indicates potential **dissatisfaction**, making it a strong churn predictor.
- Customers with a **voicemail plan** also seem more likely to churn, possibly due to **additional costs** or **poor service experience**.

3. Customer Engagement Score

- Higher engagement may **reduce churn**, but its impact is lower compared to service-related issues.
- Companies should focus on **proactive engagement strategies** to retain customers.

4. State-Based Trends

- Several states, including **NC, WI, MT, DC, and TN**, have a notable impact on churn.

- This could indicate **regional differences** in **network performance, customer demographics, or competitor influence**.

Conclusions and Recommendations

1. Conclusions

Based on the analysis of SyriaTel's customer data, the following conclusions can be drawn:

1. **Primary Factors Influencing Churn**

    - Customers with international plans exhibit a higher churn rate compared to those without.
    - Customers with higher total day and evening call charges are more likely to churn.
    - High usage of customer service calls correlates with increased churn probability, indicating dissatisfaction.
2. **Predictive Model Performance**

    - The machine learning model developed successfully predicts customer churn with a high degree of accuracy. The XGBooster performed well with an F1 score of **86.63**%
    - Feature importance analysis highlights that total charge metrics, international plans, and customer service call frequency are key indicators of churn.
    - Model F1 scores:
        - **Logistic Regression**: F1 score = `48.32`%
        - **Decision Tree**: F1 score = `78.10`%
        - **Random Forest**: F1 score = `82.42`%
        - **XGBoost**: F1 score = `86.63`%
3. **Business Impact**

    - The predictive model enables proactive identification of at-risk customers, allowing targeted retention efforts.
    - Insights from the model provide actionable areas where SyriaTel can improve customer satisfaction and reduce churn.

2. Recommendations

To address the customer churn issue and improve customer retention, the following recommendations are proposed:

1. **Customer Service Improvement**

    - Implement a proactive customer service approach to address concerns before they escalate.
    - Improve response quality and reduce the number of interactions required to resolve customer issues.
2. **Personalized Retention Offers**

    - Provide targeted discounts or benefits to high-risk customers identified by the model.
    - Offer customized plans based on customer usage patterns to enhance satisfaction and engagement.

3. **International Plan Optimization**

   - Reevaluate international plan pricing and benefits to increase customer retention.
   - Conduct surveys to understand why customers with international plans have higher churn rates.

4. **Customer Engagement Strategies**

   - Launch loyalty programs to increase customer commitment to SyriaTel services.
   - Improve communication with customers through personalized messaging and engagement campaigns.

By implementing these strategies, SyriaTel can reduce customer churn, increase customer lifetime value, and improve overall business performance.