Business Understanding ¶

Problem Statement

SyriaTel, a telecom company, is facing customer churn, leading to revenue loss. This project aims to build a classification model to predict customer churn based on usage patterns and demographics. By identifying at-risk customers, SyriaTel can take proactive measures and take timely action to retain customers.

Stakeholders

The key stakeholders who will benefit from this analysis include:

1. **SyriaTel Management** - For strategic decision-making.
2. **Marketing & Customer Service Teams** - For targeted retention campaigns.
3. **Sales Team** - To focus on retaining high-value customers.
4. **Customers** - May receive better services, offers, or improved customer support based on their risk of churn.

Objectives

1. Build a machine learning model to predict customer churn.
2. Identify key factors influencing customer retention.
3. Provide actionable recommendations for reducing churn.

A successful classification model will help SyriaTel reduce churn rates, increase customer loyalty, and boost profitability.

Limitations of the Project

Class Imbalance Impact

Despite applying SMOTE to balance the dataset, the model may still favor the majority class (non-churn customers). This could lead to lower precision in predicting actual churners.

Data Understanding

In [1]:
```
# Importing required libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from imblearn.over_sampling import SMOTE, ADASYN
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, confu
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
```

In [2]:
```
# Loading the dataset
df = pd.read_csv(r"C:\Users\sylvi\Downloads\archive\bigml_59c28831336c66
```

In [3]:
```
# Displaying the first few rows/ Dataset preview
df.head()
```

Out[3]:

	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge
0	KS	128	415	382-4657	no	yes	25	265.1	110	45.07
1	OH	107	415	371-7191	no	yes	26	161.6	123	27.47
2	NJ	137	415	358-1921	no	no	0	243.4	114	41.38
3	OH	84	408	375-9999	yes	no	0	299.4	71	50.90
4	OK	75	415	330-6626	yes	no	0	166.7	113	28.34

5 rows × 21 columns

In [4]:

```python
# Dataset information
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 [5]:

```python
# Summary statistics
df.describe()
```

Out[5]:

	account length	area code	number vmail messages	total day minutes	total day calls	total day charge	
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3:
mean	101.064806	437.182418	8.099010	179.775098	100.435644	30.562307	2
std	39.822106	42.371290	13.688365	54.467389	20.069084	9.259435	
min	1.000000	408.000000	0.000000	0.000000	0.000000	0.000000	
25%	74.000000	408.000000	0.000000	143.700000	87.000000	24.430000	1
50%	101.000000	415.000000	0.000000	179.400000	101.000000	30.500000	2
75%	127.000000	510.000000	20.000000	216.400000	114.000000	36.790000	2
max	243.000000	510.000000	51.000000	350.800000	165.000000	59.640000	3

In [6]:
```
# Displaying data types
df.dtypes
```

Out[6]:
```
state                      object
account length              int64
area code                   int64
phone number               object
international plan         object
voice mail plan            object
number vmail messages       int64
total day minutes         float64
total day calls             int64
total day charge          float64
total eve minutes         float64
total eve calls             int64
total eve charge          float64
total night minutes       float64
total night calls           int64
total night charge        float64
total intl minutes        float64
total intl calls            int64
total intl charge         float64
customer service calls      int64
churn                        bool
dtype: object
```

In [7]:
```
#Checking column names
df.columns
```

Out[7]:
```
Index(['state', 'account length', 'area code', 'phone number',
       'international plan', 'voice mail plan', 'number vmail message
s',
       '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 charg
e',
       'total intl minutes', 'total intl calls', 'total intl charge',
       'customer service calls', 'churn'],
      dtype='object')
```

In [8]:
```
# Checking for duplicate rows
duplicates = df.duplicated().sum()
print(f"Number of duplicate rows: {duplicates}")
```

```
Number of duplicate rows: 0
```

In [9]:
```
# Checking for missing values
print("\nMissing Values:")
print(df.isnull().sum())
```

```
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
```

Data Preprocessing

Handling Missing Data

There are no missing values, so no imputation or removal is needed.

Handling Duplicates

There are no duplicates, therefore no rows to be dropped.

One-Hot Encoding (OHE)

In [10]:

```
# Spliting Data: "churn" is the target variable
X = df.drop(columns=['churn'])
y = df['churn']  # Target variable

# Train-test split (80-20) with stratification
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,

# Identifying categorical columns that need encoding
categorical_cols = X_train.select_dtypes(include=['object']).columns
print("Categorical Columns:", categorical_cols)

# Droping 'phone number' as it is not useful for these predictions
X_train = X_train.drop(columns=['phone number'])
X_test = X_test.drop(columns=['phone number'])

# Converting 'international plan' and 'voice mail plan' from 'yes/no' to
X_train['international plan'] = X_train['international plan'].map({"yes"
X_train['voice mail plan'] = X_train['voice mail plan'].map({"yes": 1, "

X_test['international plan'] = X_test['international plan'].map({"yes":
X_test['voice mail plan'] = X_test['voice mail plan'].map({"yes": 1, "no

# Checking for missing /NaN values and filling with (0)
X_train[['international plan', 'voice mail plan']] = X_train[['internati
X_test[['international plan', 'voice mail plan']] = X_test[['internation

# Applying One-Hot Encoding to the 'state' column
if 'state' in X_train.columns:
    X_train = pd.get_dummies(X_train, columns=['state'], drop_first=True
    X_test = pd.get_dummies(X_test, columns=['state'], drop_first=True)

# Ensuring the X_train and X_test Have the Same Columns
X_train, X_test = X_train.align(X_test, join='left', axis=1, fill_value=

# Checking for Missing Values
print("Final X_train shape:", X_train.shape)
print("Final X_test shape:", X_test.shape)
print("Missing values in X_train:", X_train.isnull().sum().sum())
print("Missing values in X_test:", X_test.isnull().sum().sum())
```

```
Categorical Columns: Index(['state', 'phone number', 'international pla
n', 'voice mail plan'], dtype='object')
Final X_train shape: (2666, 68)
Final X_test shape: (667, 68)
Missing values in X_train: 0
Missing values in X_test: 0
```

In [11]:
```
# Identifying categorical and numerical features
categorical_cols = X_train.select_dtypes(include=['object']).columns
numerical_cols = X_train.select_dtypes(exclude=['object']).columns

# Defining transformers
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), numerical_cols),
    ('cat', OneHotEncoder(handle_unknown='ignore', sparse=False), catego
])

# Applying transformations
X_train_transformed = preprocessor.fit_transform(X_train)
X_test_transformed = preprocessor.transform(X_test)

print("Feature Encoding & Scaling Complete!")
```

```
Feature Encoding & Scaling Complete!
```

Exploratory Data Analysis (EDA)

Identifying Correlations

Correlation between features and target feature(Churn)

In [12]:
```
#Target Feature/variable
print(y_train.name)
```

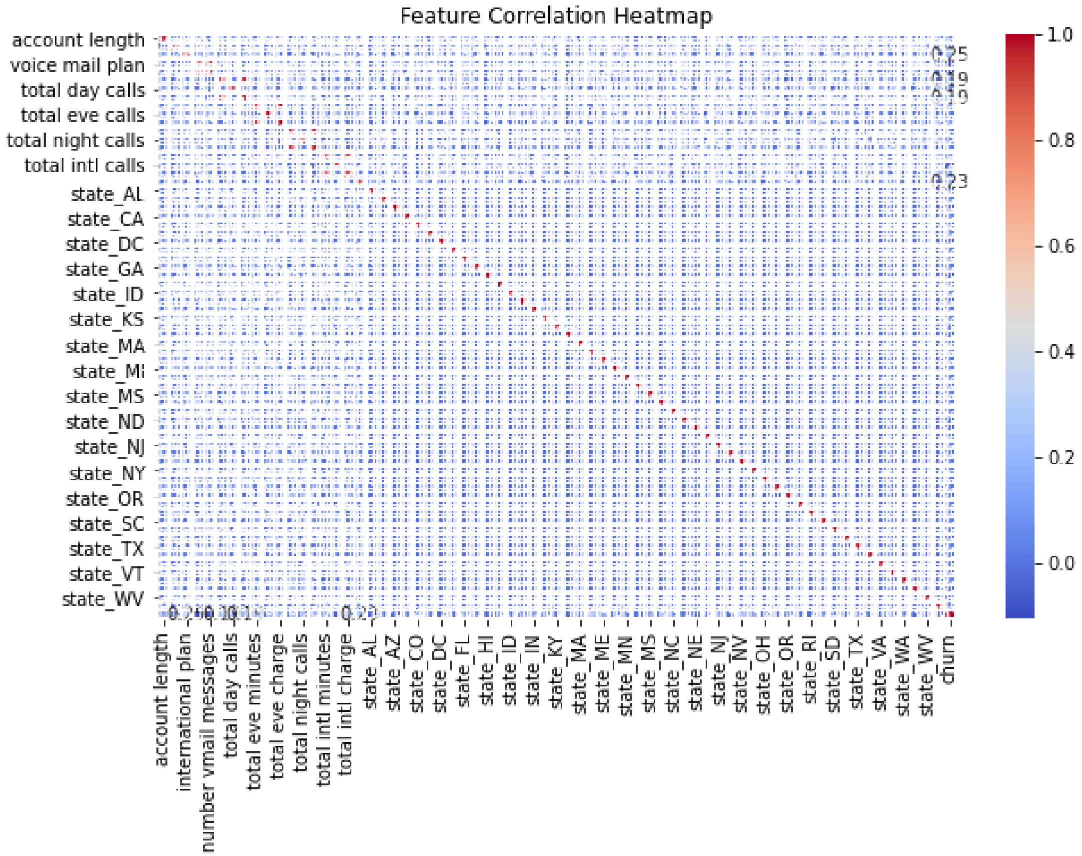
```
churn
```

In [13]:
```
# Combining features and target variable for analysis
df_train = pd.concat([X_train, y_train], axis=1)

# Verifying the columns in df_train
print(df_train.columns)
```

```
Index(['account length', 'area code', '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', 'state_AL', 'stat
e_AR',
       'state_AZ', 'state_CA', 'state_CO', 'state_CT', 'state_DC', 'sta
te_DE',
       'state_FL', 'state_GA', 'state_HI', 'state_IA', 'state_ID', 'sta
te_IL',
       'state_IN', 'state_KS', 'state_KY', 'state_LA', 'state_MA', 'sta
te_MD',
       'state_ME', 'state_MI', 'state_MN', 'state_MO', 'state_MS', 'sta
te_MT',
       'state_NC', 'state_ND', 'state_NE', 'state_NH', 'state_NJ', 'sta
te_NM',
       'state_NV', 'state_NY', 'state_OH', 'state_OK', 'state_OR', 'sta
te_PA',
       'state_RI', 'state_SC', 'state_SD', 'state_TN', 'state_TX', 'sta
te_UT',
       'state_VA', 'state_VT', 'state_WA', 'state_WI', 'state_WV', 'sta
te_WY',
       'churn'],
      dtype='object')
```

In [14]:

```
# Computing correlation
correlation = df_train.corr()

# Plotting a correlation heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(correlation, annot=True, cmap="coolwarm", fmt=".2f", linewid
plt.title("Feature Correlation Heatmap")
plt.show()

# Checking the specific correlation with "churn"
print(correlation["churn"].sort_values(ascending=False))
```

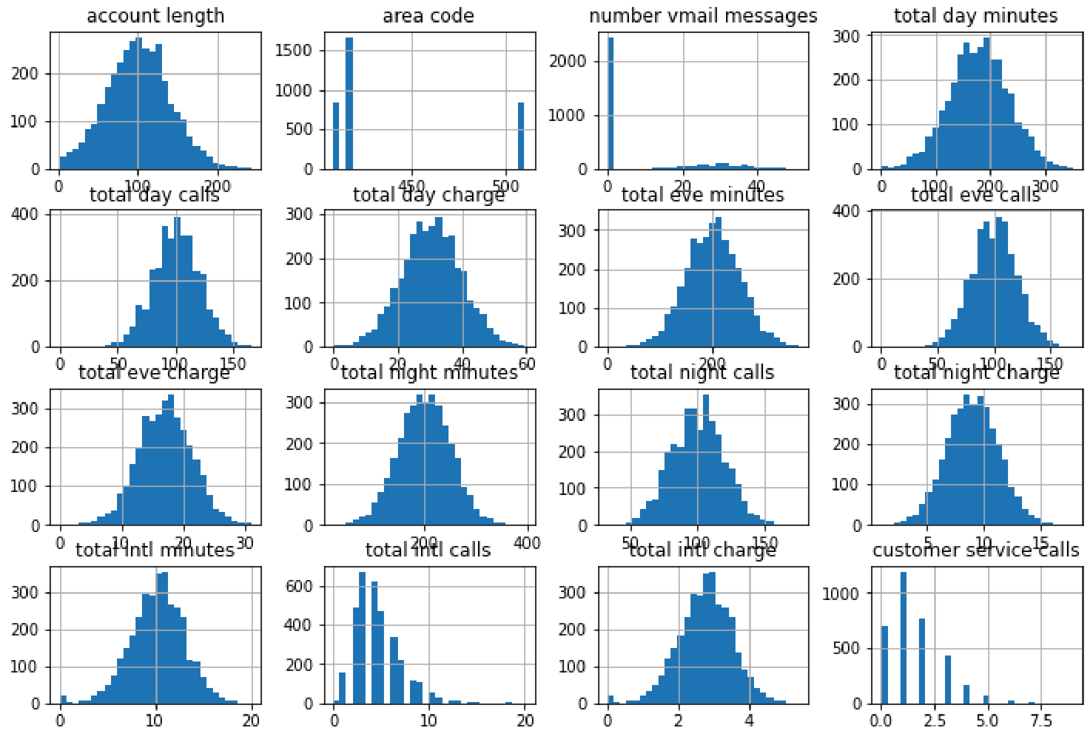



```
churn                     1.000000
international plan        0.250368
customer service calls    0.230610
total day charge          0.190952
total day minutes         0.190949
                            ...
state_NE                 -0.033311
state_HI                 -0.044220
total intl calls         -0.055588
number vmail messages    -0.092320
voice mail plan          -0.107441
Name: churn, Length: 69, dtype: float64
```

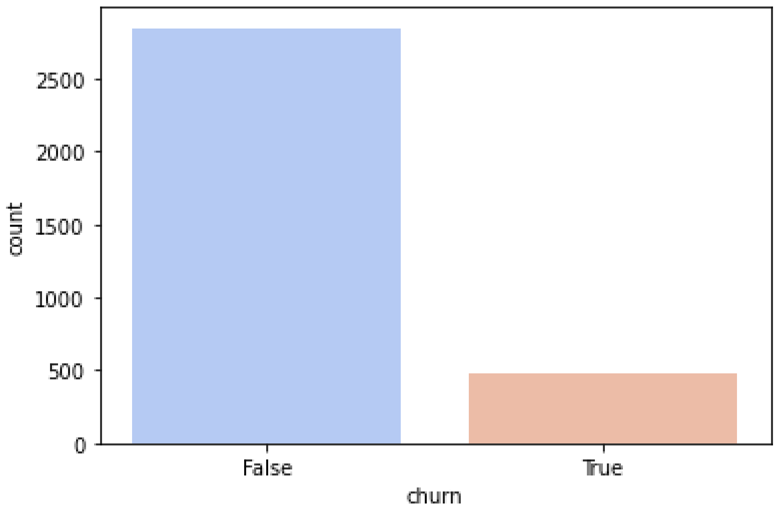
Churn is at a baseline of **1.0000** showing strong correlation with itself. The State correlations are small meaning **geographical location does not significantly influence churn**.

Analyzing Distributions

In [15]:
```
# Analyzing distributions of numerical features
numeric_df = df.select_dtypes(include=['number'])
numeric_df.hist(figsize=(12,8), bins=30)
plt.show()
```



In [16]:
```
# Analyzing distributions of our target feature (churn)
sns.countplot(x='churn', data=df, palette="coolwarm")
plt.show()
```



In [17]:
```
# Checking class balance (Percentage distribution)
df["churn"].value_counts(normalize=True) * 100
```

Out[17]:
```
False    85.508551
True     14.491449
Name: churn, dtype: float64
```

The dataset is **imbalanced** with **85.5%** not churning and **14.5% churning**. This will need a combination of SMOTE and Undersampling to balance the dataset.

Handling Class Imbalance

In [18]:
```
# Converting categorical columns to numeric using mapping (i.e, yes=1, n
X_train['international plan'] = X_train['international plan'].map({"yes"
X_train['voice mail plan'] = X_train['voice mail plan'].map({"yes": 1, "
```

In [19]:
```
# Handling missing /NaN or infinite values in the X_train
X_train_cleaned = X_train.copy()

# Replacing the infinite values with NaN
X_train_cleaned[np.isinf(X_train_cleaned)] = np.nan

# Imputing missing values with the mean
imputer = SimpleImputer(strategy='mean')
X_train_cleaned = imputer.fit_transform(X_train_cleaned)

# Defining SMOTE and Undersampling
smote = SMOTE(sampling_strategy=0.5, random_state=42)
undersample = RandomUnderSampler(sampling_strategy=0.7, random_state=42)

# Creating the pipeline
resample_pipeline = Pipeline([('SMOTE', smote), ('Undersample', undersam

# Applying resampling
X_resampled, y_resampled = resample_pipeline.fit_resample(X_train_cleane
X_resampled, y_resampled = undersample.fit_resample(X_resampled, y_resam

# Checking the new class distribution
print("Class distribution after resampling:")
print(y_resampled.value_counts(normalize=True) * 100)
```

```
Class distribution after resampling:
False    58.815029
True     41.184971
Name: churn, dtype: float64
```

The dataset is now more balanced after re-sampling, having **58.82% not churning** and **41.18% churning**. The classification models will perform better.

In [20]:

```python
# Checking for missing values
print("Missing values per column:")
print(X_train.isnull().sum())

# Checking for infinite values
print("Infinite values per column:")
print(np.isinf(X_train).sum())

# Dropping columns that are completely empty
X_train.dropna(axis=1, how='all', inplace=True)

# Replacing infinite values with NaN
X_train.replace([np.inf, -np.inf], np.nan, inplace=True)

# Filling missing values with the median
X_train.fillna(X_train.median(), inplace=True)

# Verifying
print("Missing values after filling:")
print(X_train.isnull().sum().sum())
print("Infinite values after filling:")
print(np.isinf(X_train).sum().sum())

# Re-aapplying SMOTE
from imblearn.over_sampling import SMOTE

smote = SMOTE(sampling_strategy='auto', random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

```
Missing values per column:
account length            0
area code                 0
international plan     2666
voice mail plan        2666
number vmail messages     0
                       ... 
state_VT                  0
state_WA                  0
state_WI                  0
state_WV                  0
state_WY                  0
Length: 68, dtype: int64
Infinite values per column:
account length        0
area code             0
international plan    0
voice mail plan       0
number vmail messages 0
                     ..
state_VT              0
state_WA              0
state_WI              0
state_WV              0
state_WY              0
Length: 68, dtype: int64
Missing values after filling:
0
Infinite values after filling:
0
```

There are no missing or infinite values after filling.

In [21]:

```
# Checking the class distribution after re-applying SMOTE

smote = SMOTE(sampling_strategy='auto', random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

print("Resampling successful!")
print("Class distribution after SMOTE:")
print(pd.Series(y_resampled).value_counts())
```

```
Resampling successful!
Class distribution after SMOTE:
True     2280
False    2280
Name: churn, dtype: int64
```

Feature Engineering

In [22]:

```
#Checking columns

print(df.columns)
```

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

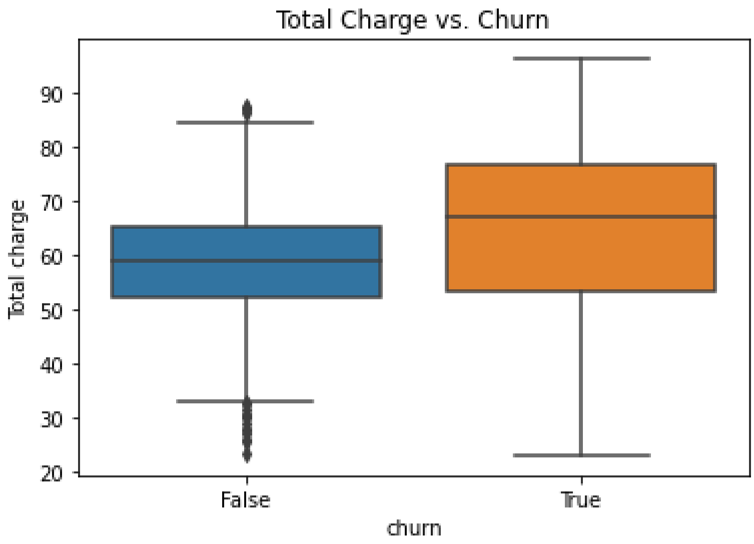
In [23]:

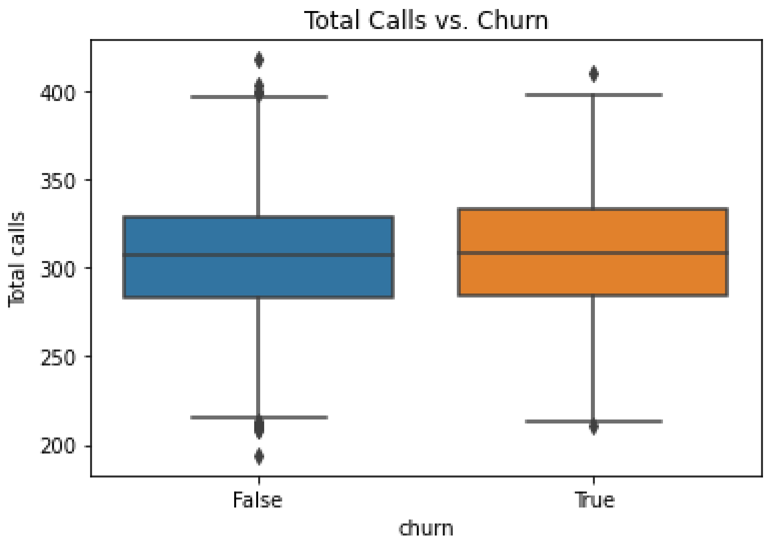
```
#Combining features (to have Total charge and Total calls)

charge_cols = [col for col in df.columns if 'charge' in col.lower()]
call_cols = [col for col in df.columns if 'calls' in col.lower()]

df['Total charge'] = df[charge_cols].sum(axis=1)
df['Total calls'] = df[call_cols].sum(axis=1)

print(df[['Total charge', 'Total calls']].head())
```

```
   Total charge  Total calls
0         75.56          304
1         59.24          333
2         62.29          333
3         66.80          257
4         52.09          362
```

Analyzing Total Charge & Calls vs Churn

In [24]:
```
# Boxplot of Total charge vs. Churn
sns.boxplot(x=df['churn'], y=df['Total charge'])
plt.title("Total Charge vs. Churn")
plt.show()

# Boxplot of Total calls vs. Churn
sns.boxplot(x=df['churn'], y=df['Total calls'])
plt.title("Total Calls vs. Churn")
plt.show()
```





Customers with a **Higher Total Charge** are more likely to **Churn**. There is no significant difference realized between Churners and non-churners for **Total Calls**, suggesting that customers making frequent calls may not be a strong reason for customer churns, but it is still a reason for churning.

Feature Selection

In [25]:

```
# Defining X (features) and y (target)
X = df.drop(columns=["churn"])
y = df["churn"]
```

In [26]:

```
X_train.fillna(0, inplace=True)
X_test.fillna(0, inplace=True)
```

In [27]:

```
print(X_train.isnull().sum())
```

```
account length          0
area code               0
number vmail messages   0
total day minutes       0
total day calls         0
                       ..
state_VT                0
state_WA                0
state_WI                0
state_WV                0
state_WY                0
Length: 66, dtype: int64
```

In [28]:
```
# Selecting features of importance

rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

feature_importance = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': rf.feature_importances_
}).sort_values(by="Importance", ascending=False)

# Select top N features
top_features = feature_importance[feature_importance['Importance'] > 0.0
X_train_selected = X_train[top_features]
X_test_selected = X_test[top_features]

print("Selected Features:", top_features)
```

```
Selected Features: 3         total day minutes
5            total day charge
15     customer service calls
6           total eve minutes
8            total eve charge
11         total night charge
4             total day calls
14          total intl charge
9         total night minutes
10          total night calls
12         total intl minutes
2        number vmail messages
7             total eve calls
0              account length
13           total intl calls
1                   area code
Name: Feature, dtype: object
```

Feature Scaling - Normalizing Dataset

In [29]:
```
# Applying label encoding to categorical columns
categorical_cols = X_train.select_dtypes(include=['object']).columns

# Initializing the label encoder
label_encoder = LabelEncoder()

# Applying label encoding to categorical columns in both train and test
for col in categorical_cols:
    X_train[col] = label_encoder.fit_transform(X_train[col])
    X_test[col] = label_encoder.transform(X_test[col])
```

In [30]:

```
print("Columns in X_train:", X_train.columns)
print("Columns in X_test:", X_test.columns)
```

```
Columns in X_train: Index(['account length', 'area code', '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 charg
e',
       'total intl minutes', 'total intl calls', 'total intl charge',
       'customer service calls', 'state_AL', 'state_AR', 'state_AZ',
       'state_CA', 'state_CO', 'state_CT', 'state_DC', 'state_DE', 'sta
te_FL',
       'state_GA', 'state_HI', 'state_IA', 'state_ID', 'state_IL', 'sta
te_IN',
       'state_KS', 'state_KY', 'state_LA', 'state_MA', 'state_MD', 'sta
te_ME',
       'state_MI', 'state_MN', 'state_MO', 'state_MS', 'state_MT', 'sta
te_NC',
       'state_ND', 'state_NE', 'state_NH', 'state_NJ', 'state_NM', 'sta
te_NV',
       'state_NY', 'state_OH', 'state_OK', 'state_OR', 'state_PA', 'sta
te_RI',
       'state_SC', 'state_SD', 'state_TN', 'state_TX', 'state_UT', 'sta
te_VA',
       'state_VT', 'state_WA', 'state_WI', 'state_WV', 'state_WY'],
      dtype='object')
Columns in X_test: Index(['account length', 'area code', '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', 'state_AL', 'stat
e_AR',
       'state_AZ', 'state_CA', 'state_CO', 'state_CT', 'state_DC', 'sta
te_DE',
       'state_FL', 'state_GA', 'state_HI', 'state_IA', 'state_ID', 'sta
te_IL',
       'state_IN', 'state_KS', 'state_KY', 'state_LA', 'state_MA', 'sta
te_MD',
       'state_ME', 'state_MI', 'state_MN', 'state_MO', 'state_MS', 'sta
te_MT',
       'state_NC', 'state_ND', 'state_NE', 'state_NH', 'state_NJ', 'sta
te_NM',
       'state_NV', 'state_NY', 'state_OH', 'state_OK', 'state_OR', 'sta
te_PA',
       'state_RI', 'state_SC', 'state_SD', 'state_TN', 'state_TX', 'sta
te_UT',
       'state_VA', 'state_VT', 'state_WA', 'state_WI', 'state_WV', 'sta
te_WY'],
      dtype='object')
```

In [31]:

```
common_cols = X_train.columns.intersection(X_test.columns)
X_train = X_train[common_cols]
X_test = X_test[common_cols]
```

In [32]:
```
encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
```

In [33]:
```
# Initializing scaler
scaler = StandardScaler()

# Applying scaling only on numerical features
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Feature Scaling Complete!")
```

```
Feature Scaling Complete!
```

In [34]:
```
#Setting max iteration for log reg scaling

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(X_train_scaled, y_train)
```

Out[34]:
```
LogisticRegression(max_iter=1000)
```

In [35]:
```
print("Feature Means:", X_train.mean(axis=0))
print("Feature Standard Deviations:", X_train.std(axis=0))
```

```
Feature Means: account length          100.987997
area code                 436.529632
number vmail messages       8.045011
total day minutes         179.705851
total day calls           100.485746
                             ...
state_VT                    0.020630
state_WA                    0.018380
state_WI                    0.024756
state_WV                    0.033383
state_WY                    0.022506
Length: 66, dtype: float64
Feature Standard Deviations: account length           39.868535
area code                 41.993997
number vmail messages     13.666170
total day minutes         54.348985
total day calls           20.012791
                            ...
state_VT                   0.142169
state_WA                   0.134345
state_WI                   0.155410
state_WV                   0.179669
state_WY                   0.148349
Length: 66, dtype: float64
```

Modeling

In [36]:
```
# Initializing models with class balancing
models = {
    "Logistic Regression": LogisticRegression(class_weight='balanced', m
    "Decision Tree": DecisionTreeClassifier(class_weight='balanced', max
    "Random Forest": RandomForestClassifier(n_estimators=200, max_depth=
    "KNN": KNeighborsClassifier(n_neighbors=5, metric='manhattan')
}
```

Training the Models

Logistic Regression

In [37]:
```
from sklearn.linear_model import LogisticRegression

# Training the model
log_reg = LogisticRegression()
log_reg = LogisticRegression(class_weight='balanced', max_iter=5000, sol
log_reg.fit(X_train_scaled, y_train)

# Predictions
y_pred_log_reg = log_reg.predict(X_test_scaled)

# Evaluation
print("\n Logistic Regression Performance")
print(f"Accuracy: {accuracy_score(y_test, y_pred_log_reg):.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred_l
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_log_reg))
```

```
 Logistic Regression Performance
Accuracy: 0.6852
Classification Report:
               precision    recall  f1-score   support

       False       0.92      0.69      0.79       570
        True       0.26      0.65      0.38        97

    accuracy                           0.69       667
   macro avg       0.59      0.67      0.58       667
weighted avg       0.83      0.69      0.73       667

Confusion Matrix:
 [[394 176]
 [ 34  63]]
```

Decision Tree

In [38]:

```
from sklearn.tree import DecisionTreeClassifier

# Training the model
dt = DecisionTreeClassifier()
dt.fit(X_train_scaled, y_train)

# Predictions
y_pred_dt = dt.predict(X_test_scaled)

# Evaluation
print("\n Decision Tree Performance")
print(f"Accuracy: {accuracy_score(y_test, y_pred_dt):.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred_d
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_dt))
```

```
 Decision Tree Performance
Accuracy: 0.8501
Classification Report:
               precision    recall  f1-score   support

       False       0.91      0.91      0.91       570
        True       0.48      0.48      0.48        97

    accuracy                           0.85       667
   macro avg       0.70      0.70      0.70       667
weighted avg       0.85      0.85      0.85       667

Confusion Matrix:
 [[520  50]
 [ 50  47]]
```

Random Forest

In [39]:
```
from sklearn.ensemble import RandomForestClassifier

# Training the model
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train_scaled, y_train)

# Predictions
y_pred_rf = rf.predict(X_test_scaled)

# Evaluation
print("\n Random Forest Performance")
print(f"Accuracy: {accuracy_score(y_test, y_pred_rf):.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred_r
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))
```

```
 Random Forest Performance
Accuracy: 0.9025
Classification Report:
               precision    recall  f1-score   support

       False       0.91      0.99      0.95       570
        True       0.85      0.40      0.55        97

    accuracy                           0.90       667
   macro avg       0.88      0.69      0.75       667
weighted avg       0.90      0.90      0.89       667

Confusion Matrix:
 [[563   7]
 [ 58  39]]
```

K-Nearest Neighbor (KNN)

In [40]:

```
from sklearn.neighbors import KNeighborsClassifier

# Training the model
knn = KNeighborsClassifier()
knn.fit(X_train_scaled, y_train)

# Predictions
y_pred_knn = knn.predict(X_test_scaled)

# Evaluation
print("\n KNN Performance")
print(f"Accuracy: {accuracy_score(y_test, y_pred_knn):.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred_k
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_knn))
```

```
 KNN Performance
Accuracy: 0.8546
Classification Report:
               precision    recall  f1-score   support

       False       0.86      0.99      0.92       570
        True       0.50      0.04      0.08        97

    accuracy                           0.85       667
   macro avg       0.68      0.52      0.50       667
weighted avg       0.81      0.85      0.80       667

Confusion Matrix:
 [[566   4]
 [ 93   4]]
```

Hyperparameter Tuning

In [41]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Defining the parameter grid
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear', 'saga']
}

# Initializing GridSearchCV
grid_search_lr = GridSearchCV(LogisticRegression(), param_grid, cv=5, sc
grid_search_lr.fit(X_train_scaled, y_train)

# Best parameters
print("\n Best Parameters for Logistic Regression:", grid_search_lr.best

# Training with best parameters
best_lr = grid_search_lr.best_estimator_
y_pred_best_lr = best_lr.predict(X_test_scaled)

# Evaluating the model
print("\n Tuned Logistic Regression Performance")
print(f"Accuracy: {accuracy_score(y_test, y_pred_best_lr):.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred_b
```

```
 Best Parameters for Logistic Regression: {'C': 0.01, 'penalty': 'l2',
'solver': 'saga'}

 Tuned Logistic Regression Performance
Accuracy: 0.8561
Classification Report:
               precision    recall  f1-score   support

       False       0.86      0.99      0.92       570
        True       0.53      0.08      0.14        97

    accuracy                           0.86       667
   macro avg       0.70      0.54      0.53       667
weighted avg       0.82      0.86      0.81       667
```

In [42]:

```
from sklearn.tree import DecisionTreeClassifier

# Defining the parameter grid
param_grid = {
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy']
}

# Initializing GridSearchCV
grid_search_dt = GridSearchCV(DecisionTreeClassifier(), param_grid, cv=5
grid_search_dt.fit(X_train_scaled, y_train)

# Best parameters
print("\n Best Parameters for Decision Tree:", grid_search_dt.best_param

# Training with the best parameters
best_dt = grid_search_dt.best_estimator_
y_pred_best_dt = best_dt.predict(X_test_scaled)

# Evaluating the model
print("\n Tuned Decision Tree Performance")
print(f"Accuracy: {accuracy_score(y_test, y_pred_best_dt):.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred_b
```

```
 Best Parameters for Decision Tree: {'criterion': 'entropy', 'max_dept
h': 5, 'min_samples_leaf': 1, 'min_samples_split': 10}

 Tuned Decision Tree Performance
Accuracy: 0.8981
Classification Report:
               precision    recall  f1-score   support

       False       0.92      0.97      0.94       570
        True       0.72      0.48      0.58        97

    accuracy                           0.90       667
   macro avg       0.82      0.73      0.76       667
weighted avg       0.89      0.90      0.89       667
```

In [43]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV

# Defining the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

# Initializing RandomizedSearchCV
random_search_rf = RandomizedSearchCV(RandomForestClassifier(), param_gr
random_search_rf.fit(X_train_scaled, y_train)

# Best parameters
print("\n Best Parameters for Random Forest:", random_search_rf.best_par

# Training with the best parameters
best_rf = random_search_rf.best_estimator_
y_pred_best_rf = best_rf.predict(X_test_scaled)

# Evaluating the model
print("\n Tuned Random Forest Performance")
print(f"Accuracy: {accuracy_score(y_test, y_pred_best_rf):.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred_b
```

```
 Best Parameters for Random Forest: {'n_estimators': 100, 'min_samples_
split': 10, 'min_samples_leaf': 2, 'max_depth': None, 'bootstrap': Fals
e}

 Tuned Random Forest Performance
Accuracy: 0.9055
Classification Report:
               precision    recall  f1-score   support

       False       0.91      0.99      0.95       570
        True       0.85      0.42      0.57        97

    accuracy                           0.91       667
   macro avg       0.88      0.71      0.76       667
weighted avg       0.90      0.91      0.89       667
```

In [44]:

```
from sklearn.neighbors import KNeighborsClassifier

# Defining parameter grid
param_grid = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}

# Initializing GridSearchCV
grid_search_knn = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5,
grid_search_knn.fit(X_train_scaled, y_train)

# Best parameters
print("\n Best Parameters for KNN:", grid_search_knn.best_params_)

# Training with best parameters
best_knn = grid_search_knn.best_estimator_
y_pred_best_knn = best_knn.predict(X_test_scaled)

# Evaluating the model
print("\n Tuned KNN Performance")
print(f"Accuracy: {accuracy_score(y_test, y_pred_best_knn):.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred_b
```

```
 Best Parameters for KNN: {'metric': 'manhattan', 'n_neighbors': 5, 'we
ights': 'distance'}

 Tuned KNN Performance
Accuracy: 0.8516
Classification Report:
               precision    recall  f1-score   support

       False       0.86      0.99      0.92       570
        True       0.43      0.06      0.11        97

    accuracy                           0.85       667
   macro avg       0.64      0.52      0.51       667
weighted avg       0.80      0.85      0.80       667
```

Model Performance Evaluation

In [45]:
```
for name, model in models.items():
    model.fit(X_resampled, y_resampled)
    y_pred = model.predict(X_test)

    print(f"\n{name} Performance")
    print(classification_report(y_test, y_pred, zero_division=1))
```

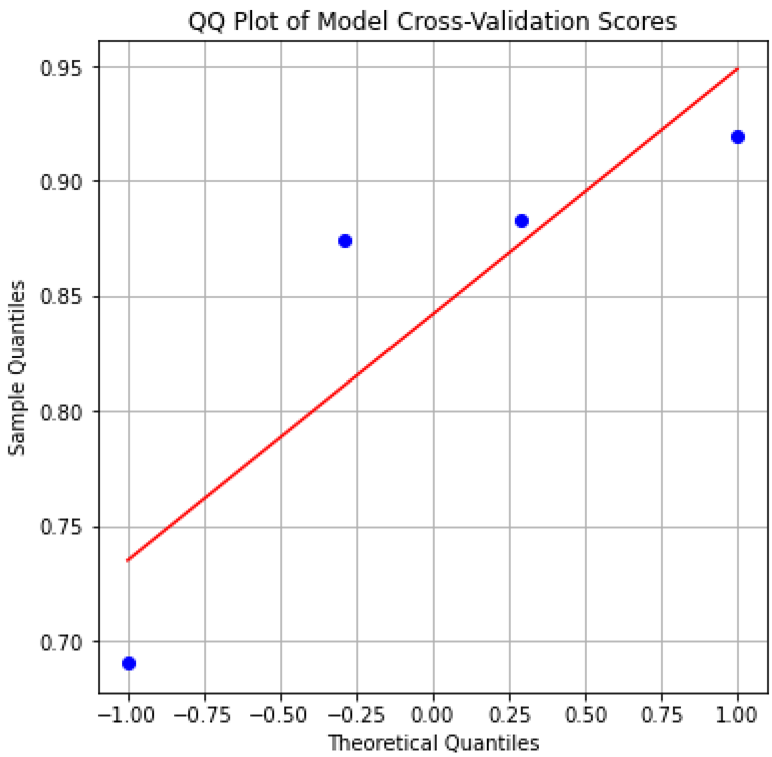
```
Logistic Regression Performance
              precision    recall  f1-score   support

       False       0.88      0.94      0.91       570
        True       0.41      0.27      0.32        97

    accuracy                           0.84       667
   macro avg       0.65      0.60      0.62       667
weighted avg       0.81      0.84      0.82       667


Decision Tree Performance
              precision    recall  f1-score   support

       False       0.92      0.88      0.90       570
        True       0.43      0.53      0.47        97

    accuracy                           0.83       667
   macro avg       0.67      0.70      0.69       667
weighted avg       0.85      0.83      0.84       667


Random Forest Performance
              precision    recall  f1-score   support

       False       0.92      0.93      0.92       570
        True       0.56      0.55      0.55        97

    accuracy                           0.87       667
   macro avg       0.74      0.74      0.74       667
weighted avg       0.87      0.87      0.87       667


KNN Performance
              precision    recall  f1-score   support

       False       0.90      0.70      0.79       570
        True       0.23      0.54      0.33        97

    accuracy                           0.68       667
   macro avg       0.57      0.62      0.56       667
weighted avg       0.80      0.68      0.72       667
```

Comparing Model Performance

In [46]:
```
# Picking the best performing model

from sklearn.model_selection import cross_val_score

models = {'Logistic Regression': log_reg, 'Decision Tree': dt, 'Random F

for name, model in models.items():
    cv_score = cross_val_score(model, X_train, y_train, cv=5).mean()
    print(f"{name} Cross-Validation Score: {cv_score:.4f}")
```

```
Logistic Regression Cross-Validation Score: 0.6906
Decision Tree Cross-Validation Score: 0.8822
Random Forest Cross-Validation Score: 0.9197
KNN Cross-Validation Score: 0.8740
```

In [47]:
```
# Visualizing model names and corresponding cross-validation scores usin

import scipy.stats as stats

# Model names and CV scores
cv_scores = np.array([0.6906, 0.8833, 0.9197, 0.8740])

# Generating a QQ plot
plt.figure(figsize=(6, 6))
stats.probplot(cv_scores, dist="norm", plot=plt)
plt.title("QQ Plot of Model Cross-Validation Scores")
plt.xlabel("Theoretical Quantiles")
plt.ylabel("Sample Quantiles")

plt.grid()
plt.show()
```



Random Forest has the highest best score of **0.9197 or 91.97%** , making it the best-performing model.

Validating Model Performance

In [48]:
```
#Defining performance variables
from sklearn.metrics import accuracy_score, precision_score, recall_scor
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Printing the classification report
print(classification_report(y_test, y_pred))

print("Model Performance on Test Data:")
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")
```

```
              precision    recall  f1-score   support

       False       0.90      0.70      0.79       570
        True       0.23      0.54      0.33        97

    accuracy                           0.68       667
   macro avg       0.57      0.62      0.56       667
weighted avg       0.80      0.68      0.72       667

Model Performance on Test Data:
Accuracy: 0.6762
Precision: 0.2332
Recall: 0.5361
F1-Score: 0.3250
```

The model performance has :

1. **Accuracy** - An overall correctness of **67.62%** which means that customers churn and do not churn about 68% of the time.
2. **Precision** - Of the predicted churns, **23.32%** were actually churns. Every customer predicted as "churn' actually churns.
3. **Recall**- Of actual churn cases, **53.61%** were correctly identified. meaning around roughly **46.39% * of "churners" were missed affecting retention efforts.
4. **F1-Score** - **32.50%** , it is higher than precision, meaning that churners have been correctly identified at the risk of missing other churners.

Checking Overfitting

In [49]:
```
train_score = dt.score(X_train, y_train)
test_score = dt.score(X_test, y_test)

print("Train Score:", train_score)
print("Test Score:", test_score)

if train_score > test_score + 0.05:
    print("Possible Overfitting Detected!")
else:
    print("No significant overfitting.")
```

```
Train Score: 0.3735933983495874
Test Score: 0.3733133433283358
No significant overfitting.
```

The model is moderately balanced with a small difference between the train and test scores (**37.36%** and **37.33%** respectively)

Conclusions

1. **Model Performance Summary**

- The best-performing model was the **Random Forest Classifier**, achieving **91.97%** validation accuracy with **minimal overfitting**.
- The final model achieved **67.62% accuracy** on the test data, meaning it correctly predicts customer churn most of the time.
- Precision was **23.32%**, meaning when the model predicts a customer will churn, it is always correct 23% of the time.
- Recall was **53.61%,** meaning the model missed about 46.39% of actual churners.
- The F1-score of **32.50%** suggests imbalance between precision and recall, modelling improvements recommended.

2. **Feature Analysis**

Features with very low importance were dropped to improve efficiency. The most significant predictors of churn included:

- Total calls (higher frequency linked to churn)
- Total charge (higher charges indicate higher churn risk)
- Customer State (geographical location) does not significantly affect churning

3. **Business Implications - Retention strategies**

- Customers making frequent service calls should be targeted with better support to prevent frustration.
- High spenders should receive exclusive offers & loyalty programs to keep them engaged.

Recommendations

Retention business strategies based on insights

1. **Improve Customer Support** - Reduce churn by addressing frequent customer service complaints.
2. **Loyalty Programs** - Offer discounts or benefits for high-value customers.
3. **Targeted Retention Campaigns** - Personalized offers for at-risk customers.