Analysisis for Predicting and preventing customer churn for SyriaTel company

Objectives of the study

-Build a machine learning model that predicts customer churn

-Idenfiy factors that highly contribute to churn

-Provide inferential statistics and visualisations based on this data.

Data Understanding

This project uses a Churn in Telecom dataset from Kaggle. The target variable is in the churn column. The dataset encompasses various features, including both locational details (state and area_code) and plan specifics such as call minutes, charges, and customer service calls. Additionally, it indicates whether customers have international plans and/or voice mail plans. Through multiple model iterations, we analyzed subsets of these features along with aggregated versions to discern which ones most accurately predict customer churn. The data can be downloaded directly from kaggle or from this repo in Data.csv file.

Models used

For this project, the models we used are

Decisioin Tree Classifier

Logistic Regression Classifier

Random Forest Classifer

XGBoost Classifier

KNN Classifier

Metric to use to score the model

We used recall score to score our models and assesed the best performing model using the test data. We opted for recall since we wish to reduce the cost incurred by a lot of false negatives.It woulde be more costly for the company if the model predicted that a customer would stay with SyriaTel when in fact that would churn/leave. This would lead to a missed opportunity for the company to dedicate retention resources towards that customer and keeping their business.

Numerical data in 16/21 columns

account length - the duration of time the client has been active

area code - area code of client residence

number vmail messages - total vmail messages sent by client

total day minutes - total day minutes used by client

total day calls - total number of day calls made

total day charge - total charge for the day calls

total eve minutes - total evening minutes used by client

total eve calls - total number of evening calls made

total eve charge - total charge for the evening calls

total night minutes - total night minutes used by client

total night calls- total number of night calls made

total night charge - total charge forthe night calls

total intl minutes - total international minutes used by the client

total intl calls- total number of international calls made

total intl charge- total charge for the international calls

customer service calls - total number of calls made by client to the customer service

Categorical data in 5/21 columns

state - this is the state where the client resides

phone number - the phone contact of the client

international plan - for a client who has subscribed to an international plan

voice mail plan - for a client who has subscribed to a voicemail plan

churn - status of a client as either churned(True) or not churned(False) Syriatel company services

state full names and abbreviations

Alabama Ala. AL

Alaska Alaska AK

Arizona Ariz. AZ

Arkansas Ark. AR

California Calif. CA

Canal Zone C.Z. CZ

Colorado Colo. CO

Connecticut Conn. CT

Delaware Del. DE

District of Columbia D.C. DC

Florida Fla. FL

Georgia Ga. GA

Guam Guam GU

Hawaii Hawaii HI

Idaho Idaho ID

Illinois Ill. IL

Indiana Ind. IN

Iowa Iowa IA

Kansas Kan. KS

Kentucky Ky. KY

Louisiana La. LA

Maine Maine ME

Maryland Md. MD

Massachusetts Mass. MA

Michigan Mich. MI

Minnesota Minn. MN

Mississippi Miss. MS

Missouri Mo. MO

Montana Mont. MT

Nebraska Neb. NE

Nevada Nev. NV

New Hampshire N.H. NH

New Jersey N.J. NJ

New Mexico N.M. NM

New York N.Y. NY

North Carolina N.C. NC

North Dakota N.D. ND

Ohio Ohio OH

Oklahoma Okla. OK

Oregon Ore. OR

Pennsylvania Pa. PA

Puerto Rico P.R. PR

Rhode Island R.I. RI

South Carolina S.C. SC

South Dakota S.D. SD

Tennessee Tenn. TN

Texas Texas TX

Utah Utah UT

Vermont Vt. VT

Virgin Islands V.I. VI

Virginia Va. VA

Washington Wash. WA

West Virginia W.Va. WV

Wisconsin Wis. WI

Wyoming Wyo. WY

Data Cleaning and EXploratory Data Analysis

Importing necessary libraries

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.model_selection import  GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.impute import SimpleImputer
from imblearn.over_sampling import SMOTE
from sklearn.metrics import accuracy_score, precision_score, recall_score,
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
from sklearn.pipeline import make_pipeline
from sklearn.metrics import precision_recall_curve
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import average_precision_score
from sklearn.svm import SVC
from sklearn.datasets import make_classification
from sklearn.neighbors import KNeighborsClassifier
```

Loading the dataset

In [2]:
```
# using pandas to read the data
df= pd.read_csv('Data.csv')
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	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

Data Cleaning

In [3]:
```
# checking the rows and columns of our data
df.shape
```

Out[3]:
```
(3333, 21)
```

In [4]:
```
# checking the columns of our data
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]:
```python
# checking for null values
df.isnull().sum()
```

Out[5]:
```
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
```

There are no missing values in this dataset

In [6]:
```python
df.describe()
```

Out[6]:

	account length	area code	number vmail messages	total day minutes	total day calls	total day charge	t
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333
mean	101.064806	437.182418	8.099010	179.775098	100.435644	30.562307	200
std	39.822106	42.371290	13.688365	54.467389	20.069084	9.259435	50
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	166
50%	101.000000	415.000000	0.000000	179.400000	101.000000	30.500000	201
75%	127.000000	510.000000	20.000000	216.400000	114.000000	36.790000	235
max	243.000000	510.000000	51.000000	350.800000	165.000000	59.640000	363

The above dataframe gives the count, mean, std deviation,min and max value, and the 25th, 50th and 75th quartile

In [7]:
```
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
```

phone_number, international_plan and voice_mail_plan are strings and our target churn which is boolean data type but the other features are numeric

Dropping irrelevant columns

We will be dropping phone number column since we won't need it

In [8]:
```
# dropping phone number column
df.drop(['phone number'], axis=1, inplace=True)
```

Exploratory Data Analysis

Exploring Area Code

We will explore the area code to see if it has any relations with customer churn

In [9]:
```
df['area code'].value_counts()
```

Out[9]:
```
415    1655
510     840
408     838
Name: area code, dtype: int64
```

We have 3 area codes

In [10]:
```
from scipy.stats import pointbiserialr

# Calculate point-biserial correlation coefficient
correlation, p_value = pointbiserialr(df['area code'], df['churn'])

print("Point-biserial correlation coefficient:", correlation)
print("P-value:", p_value)
```

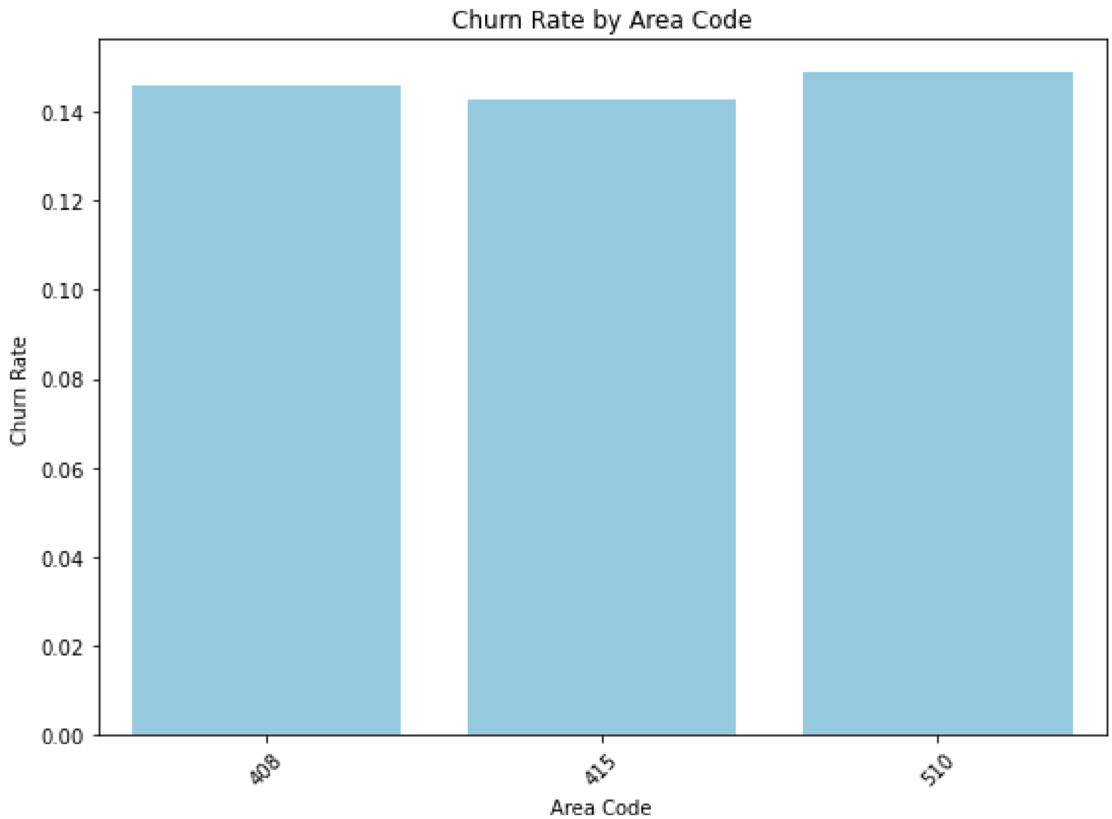
```
Point-biserial correlation coefficient: 0.006174233160678325
P-value: 0.7215998968003063
```

from the correlation coefficient we see that area code has a positive correlation of 0.7215998968003063 to the churn.

We will then plot a bar graph to show the churn rate of each area below.

In [11]:

```
#Calculate churn rate by area code
churn_rate = df.groupby("area code")["churn"].mean().reset_index()

#Plot churn rate by area code
plt.figure(figsize=(8, 6))
sns.barplot(data=churn_rate, x="area code", y="churn", color="skyblue")
plt.title("Churn Rate by Area Code")
plt.xlabel("Area Code")
plt.ylabel("Churn Rate")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



From the bar plot, we note that as much as the area code correlates with the churn, each area has close to the same rate of churn and thus making it a less relevant variable in our data set and hence we will drop it.

In [12]:

```
df.drop('area code', axis = 1, inplace = True)
```

Exploring correlations

We will inspect the columns (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) to see if there are any correlations between them.

In [13]:
```
df[['total day minutes', 'total day calls', 'total day charge', 'total eve
df.corr()
```

Out[13]:

	account length	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	total eve calls	tota ch
account length	1.000000	-0.004628	0.006216	0.038470	0.006214	-0.006757	0.019260	-0.00
number vmail messages	-0.004628	1.000000	0.000778	-0.009548	0.000776	0.017562	-0.005864	0.01
total day minutes	0.006216	0.000778	1.000000	0.006750	1.000000	0.007043	0.015769	0.00
total day calls	0.038470	-0.009548	0.006750	1.000000	0.006753	-0.021451	0.006462	-0.02
total day charge	0.006214	0.000776	1.000000	0.006753	1.000000	0.007050	0.015769	0.00
total eve minutes	-0.006757	0.017562	0.007043	-0.021451	0.007050	1.000000	-0.011430	1.00
total eve calls	0.019260	-0.005864	0.015769	0.006462	0.015769	-0.011430	1.000000	-0.01
total eve charge	-0.006745	0.017578	0.007029	-0.021449	0.007036	1.000000	-0.011423	1.00
total night minutes	-0.008955	0.007681	0.004323	0.022938	0.004324	-0.012584	-0.002093	-0.01
total night calls	-0.013176	0.007123	0.022972	-0.019557	0.022972	0.007586	0.007710	0.00
total night charge	-0.008960	0.007663	0.004300	0.022927	0.004301	-0.012593	-0.002056	-0.01
total intl minutes	0.009514	0.002856	-0.010155	0.021565	-0.010157	-0.011035	0.008703	-0.01
total intl calls	0.020661	0.013957	0.008033	0.004574	0.008032	0.002541	0.017434	0.00
total intl charge	0.009546	0.002884	-0.010092	0.021666	-0.010094	-0.011067	0.008674	-0.01
customer service calls	-0.003796	-0.013263	-0.013423	-0.018942	-0.013427	-0.012985	0.002423	-0.01
churn	0.016541	-0.089728	0.205151	0.018459	0.205151	0.092796	0.009233	0.09

From our correlation matrix we see that there is a perfect correlation of 1, between all the minutes and charge features and hence we may need to combine these features later.

Exploring customer churn column

In [14]:
```
#getting customer churn count
df['churn'].value_counts()
```

Out[14]:
```
False    2850
True      483
Name: churn, dtype: int64
```

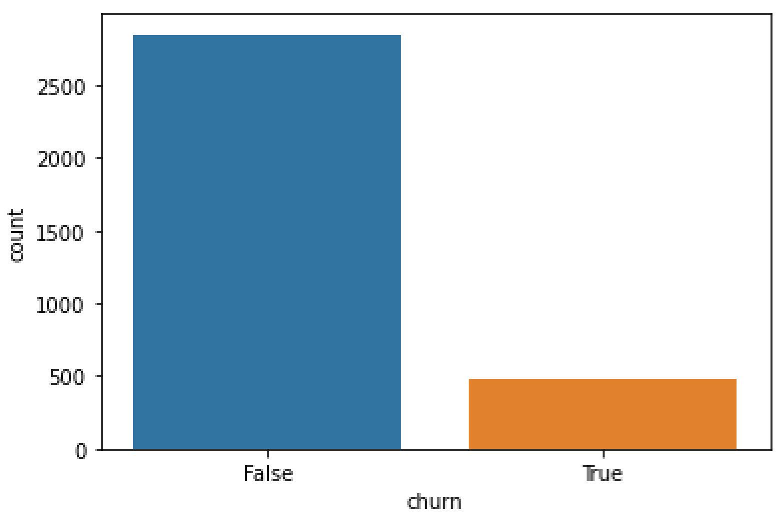
Here we note that there is a class imbalance in our dataset with class: True having 483 values and class: False having 2850 values. We will need to balance this later.

In [15]:
```
# percentage of customers that churn
churned=df[df['churn']==True].shape[0]
not_churned=df[df['churn']==False].shape[0]
print(churned/(churned+not_churned))
```

```
0.14491449144914492
```

In [16]:
```
#getting churn visualization
sns.countplot(x='churn', data=df)
```

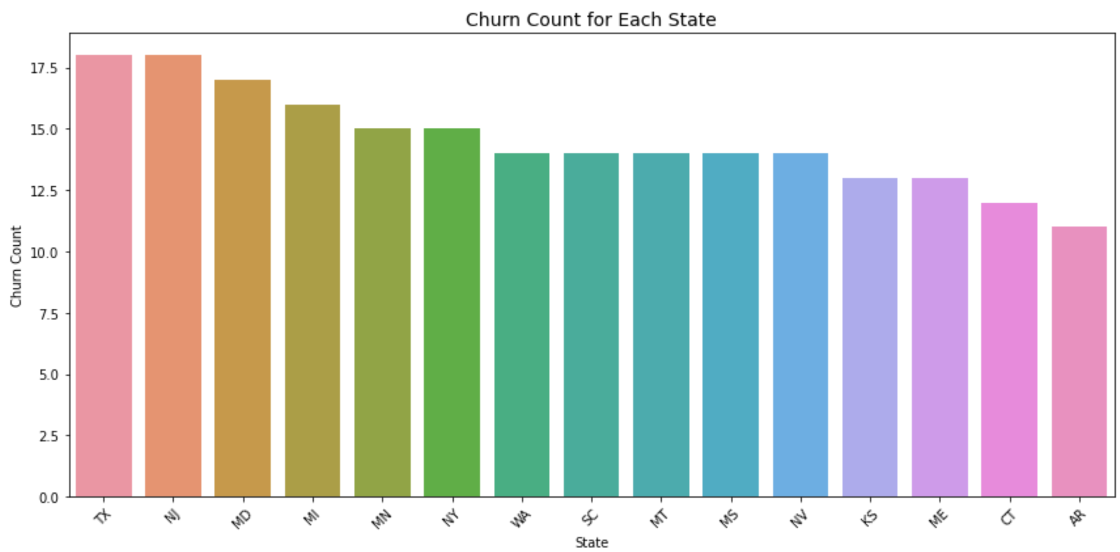
Out[16]: <AxesSubplot:xlabel='churn', ylabel='count'>



Bar plot of top 15 states with the highest churn rate

We will explore the top states with the highest churn rate from our dataset.

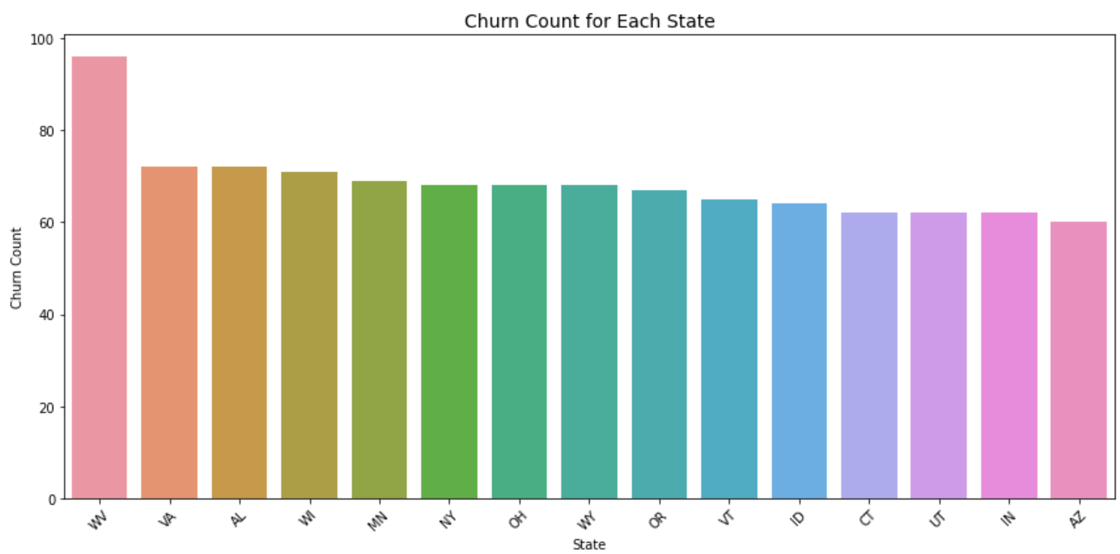
In [17]:
```
# bar plot of customers who churned
# getting churned df
churned_df = df[df['churn'] == True]

# getting churn counts for each state for churned customers
churn_counts = churned_df['state'].value_counts().sort_values(ascending=Fa
top_15=churn_counts.head(15)
# Set the size of the plot
plt.figure(figsize=(12, 6))

# Plot churn  for each state  in descending order
sns.barplot(x=top_15.index, y=top_15.values)

# Rotate x-axis labels for better readability
plt.xticks(rotation=45)

# Set labels
plt.xlabel('State')
plt.ylabel('Churn Count')
plt.title('Churn Count for Each State', fontsize=14)

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



State New Jersey has the highest churn rate followed by Texas from our visualization above.

Bar plot of customers who did not churn

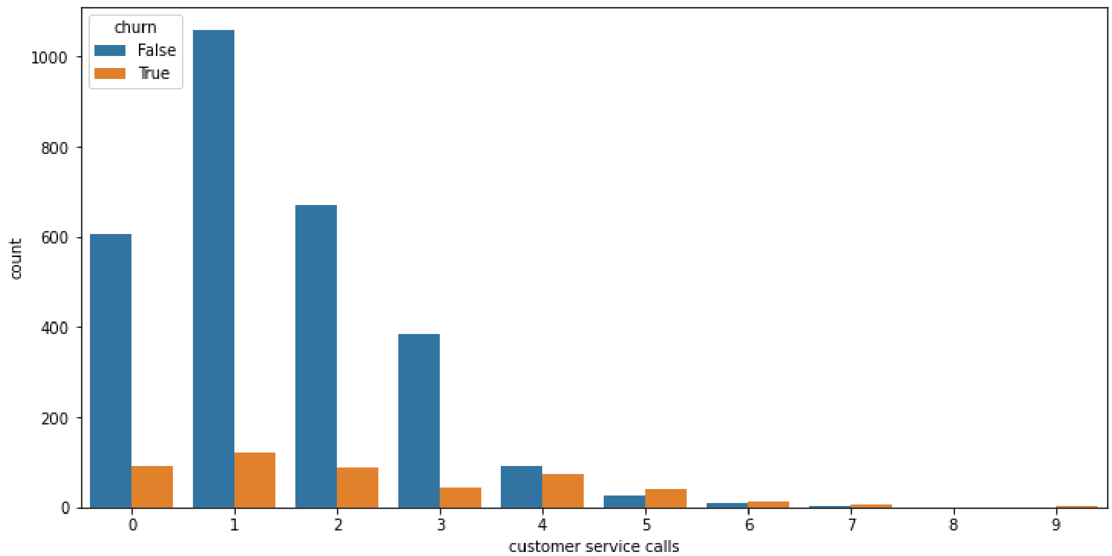
In [18]:
```
# bar plot of customers who did not churn
# getting customers that were retained df
non_churn_df = df[df['churn'] == False]

# getting non churn counts for each state for non churned customers
non_churn_counts = non_churn_df['state'].value_counts().sort_values(ascend
top_15=non_churn_counts.head(15)
# Set the size of the plot
plt.figure(figsize=(12, 6))

# Plot non churn  for each state  in descending order
sns.barplot(x=top_15.index, y=top_15.values)

# Rotate x-axis labels for better readability
plt.xticks(rotation=45)

# Set labels
plt.xlabel('State')
plt.ylabel('Churn Count')
plt.title('Churn Count for Each State', fontsize=14)

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



From the plot, the customers we retained the most are from West Virginia followed by Virginia.

Exploring the customer service calls column

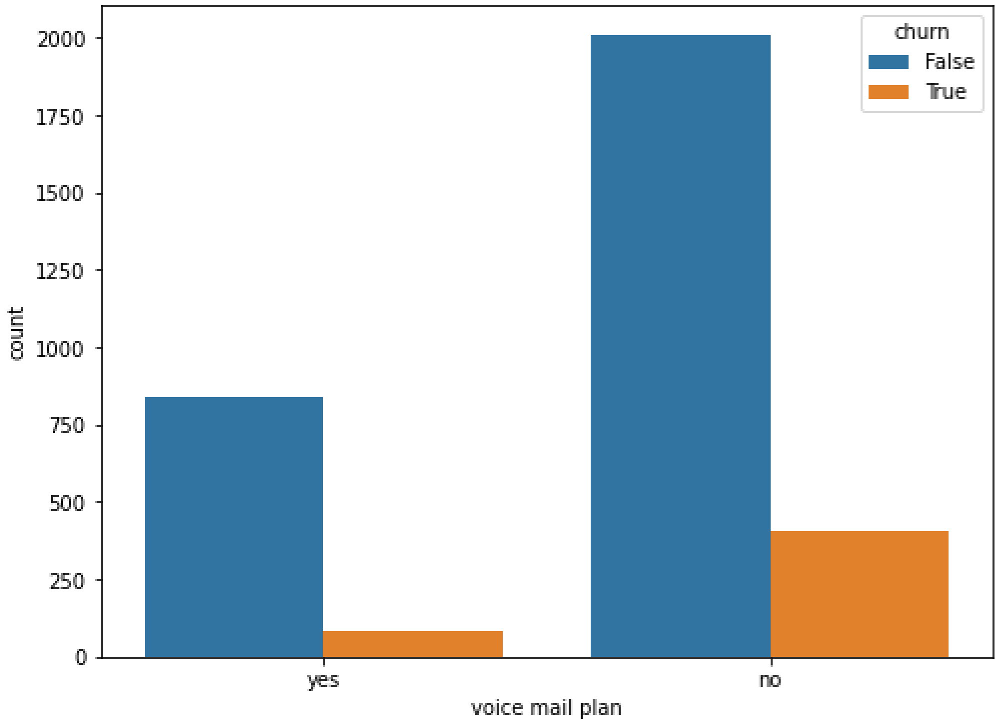
In [19]:
```
# Set the figure size
plt.figure(figsize=(12, 6))
sns.countplot(x='customer service calls', hue='churn', data=df)
#display
plt.show()
```



From the plot we see that the retained customers have higher calls but the churned ones equally contant customer service.

Exploring the voice mail churn column

In [20]:
```
# Set the figure size
plt.figure(figsize=(8, 6))
sns.countplot(x='voice mail plan', hue='churn', data=df)
# display
plt.show()
```

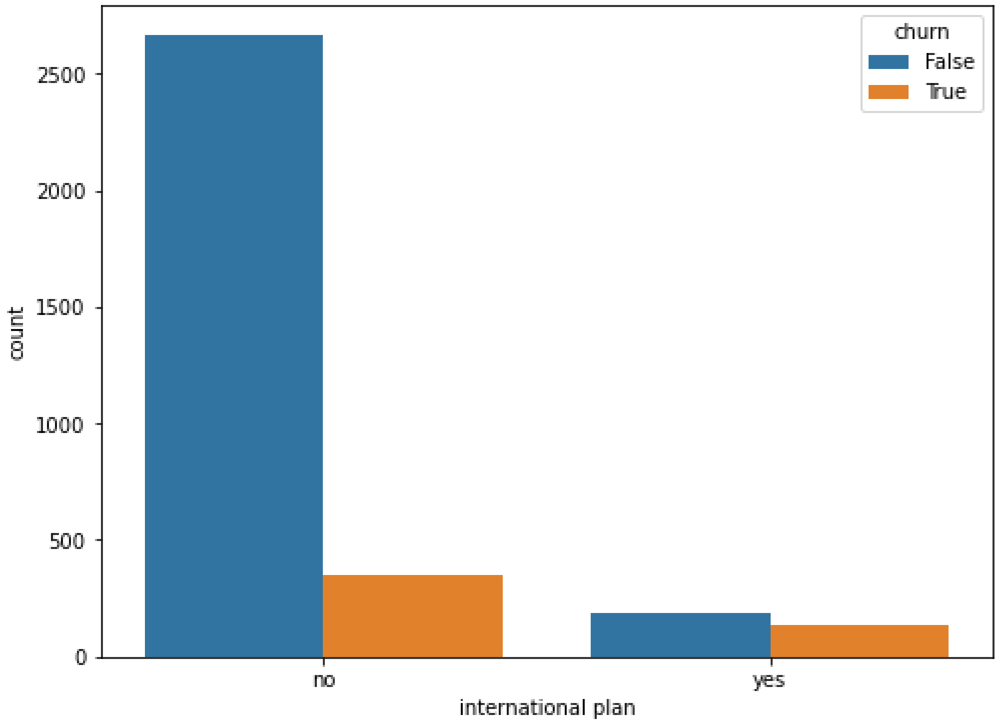


Most of the churned customers did not have a voice mail plan.

Exploring international plan column

In [21]:

```
# Set the figure size
plt.figure(figsize=(8, 6))
sns.countplot(x='international plan', hue='churn', data=df)
# display
plt.show()
```

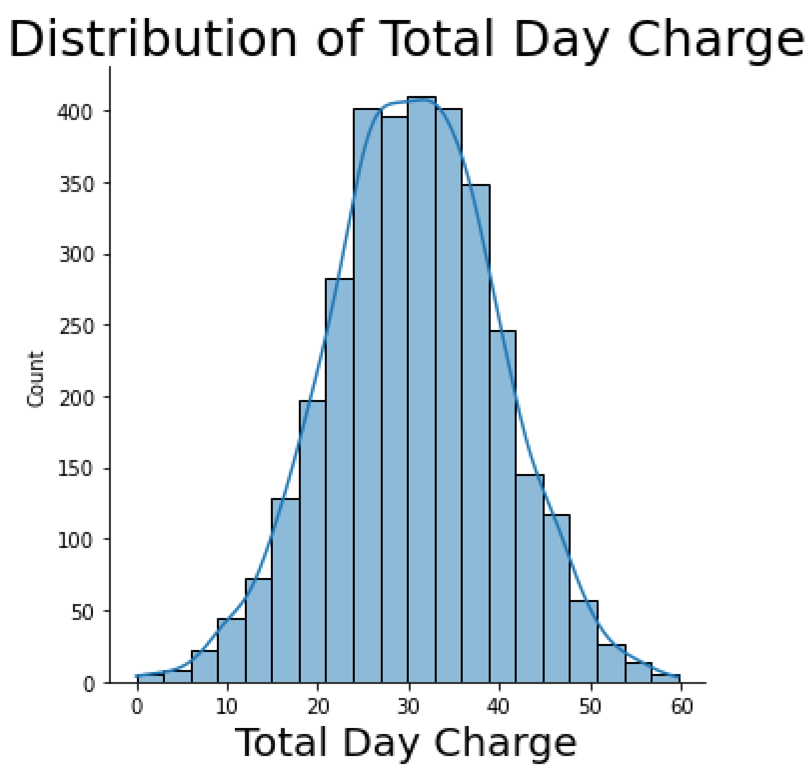


Most of the churned customers do not have an international plan.

Distribution of total day charge

In [22]:
```
# plotting displot
plt.figure(figsize=(15,8))
sns.displot(df['total day charge'], bins=20, kde=True)
#setting labels
plt.title('Distribution of Total Day Charge', fontsize = 25)
plt.xlabel('Total Day Charge', fontsize = 20)
plt.show()
```

```
<Figure size 1080x576 with 0 Axes>
```



From the distribution the data is normally distributed.

In [23]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3333 entries, 0 to 3332
Data columns (total 19 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   state                   3333 non-null   object 
 1   account length          3333 non-null   int64  
 2   international plan      3333 non-null   object 
 3   voice mail plan         3333 non-null   object 
 4   number vmail messages   3333 non-null   int64  
 5   total day minutes       3333 non-null   float64
 6   total day calls         3333 non-null   int64  
 7   total day charge        3333 non-null   float64
 8   total eve minutes       3333 non-null   float64
 9   total eve calls         3333 non-null   int64  
 10  total eve charge        3333 non-null   float64
 11  total night minutes     3333 non-null   float64
 12  total night calls       3333 non-null   int64  
 13  total night charge      3333 non-null   float64
 14  total intl minutes      3333 non-null   float64
 15  total intl calls        3333 non-null   int64  
 16  total intl charge       3333 non-null   float64
 17  customer service calls  3333 non-null   int64  
 18  churn                   3333 non-null   bool   
dtypes: bool(1), float64(8), int64(7), object(3)
memory usage: 472.1+ KB
```

We have some objects in the data that need to be transformed to numeric before modelling.

Encoding categorical data

We will first split our data into train and test splits before encoding it to prevent data leakage.

In [24]:

```
#Prepare the data
X = df.drop('churn', axis=1)
y = df['churn']
```

Binary encoding

In [25]:

```
# Split the data into training and testing sets
# we will use a test size of 0.2 and random state of 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r
```

We will do binary encoding using LabelEncoder from sklearn. And we will fit and transform categorical variables in training data and transform the same categorical variables in test data.

In [26]:
```
from sklearn.preprocessing import LabelEncoder

# Instantiate the label encoder
label_encoder = LabelEncoder()

# Perform label encoding on training data
X_train.loc[:, 'international plan'] = label_encoder.fit_transform(X_train
X_train.loc[:, 'voice mail plan'] = label_encoder.fit_transform(X_train['v
X_train.loc[:, 'international plan'] = X_train['international plan'].astyp
X_train.loc[:, 'voice mail plan'] = X_train['voice mail plan'].astype('int

# Apply the transformation to the testing data
X_test.loc[:, 'international plan'] = label_encoder.transform(X_test['inte
X_test.loc[:, 'voice mail plan'] = label_encoder.transform(X_test['voice m
```

One Hot Encoding

One hot encoding train data

We will perform one hot encoding on the state column to make it numerical. We will fit and transform the train set then transform the column as well in test set.

In [27]:
```
# instantiate ohe object
ohe = OneHotEncoder(sparse = False, handle_unknown = "ignore")

# fit ohe on small train data
ohe.fit(X_train[['state']])

# access the column names of the states
col_names = ohe.categories_[0]

# make a df with encoded states
train_state_encoded = pd.DataFrame(ohe.transform(X_train[["state"]]),
                                   index = X_train.index,
                                   columns = col_names)

# combine encoded states with X_t and drop old 'state' column
X_train = pd.concat([X_train.drop("state", axis = 1), train_state_encoded]
```

One hot encoding test data

In [28]:
```
# df with encoded states
test_state_encoded = pd.DataFrame(ohe.transform(X_test[['state']]),
                                  index = X_test.index,
                                  columns = col_names)

# combine encoded states with X_test and drop old 'state' column
X_test = pd.concat([X_test.drop("state", axis = 1), test_state_encoded], a
```

In [29]:
```
# checking first five rows of our data
X_train.head()
```

Out[29]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	total eve calls	to e cha
817	243	0	0	0	95.5	92	16.24	163.7	63	13
1373	108	0	0	0	112.0	105	19.04	193.7	110	16
679	75	1	0	0	222.4	78	37.81	327.0	111	27
56	141	0	0	0	126.9	98	21.57	180.0	62	15
1993	86	0	0	0	216.3	96	36.77	266.3	77	22

5 rows × 68 columns

In [30]:
```
X_train.tail()
```

Out[30]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	total eve calls	to e cha
1095	106	0	0	0	274.4	120	46.65	198.6	82	16
1130	122	0	0	0	35.1	62	5.97	180.8	89	15
1294	66	0	0	0	87.6	76	14.89	262.0	111	22
860	169	0	0	0	179.2	111	30.46	175.2	130	14
3174	36	0	1	43	29.9	123	5.08	129.1	117	10

5 rows × 68 columns

In [31]:

```
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2666 entries, 817 to 3174
Data columns (total 68 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   account length          2666 non-null   int64  
 1   international plan      2666 non-null   int64  
 2   voice mail plan         2666 non-null   int64  
 3   number vmail messages   2666 non-null   int64  
 4   total day minutes       2666 non-null   float64
 5   total day calls         2666 non-null   int64  
 6   total day charge        2666 non-null   float64
 7   total eve minutes       2666 non-null   float64
 8   total eve calls         2666 non-null   int64  
 9   total eve charge        2666 non-null   float64
 10  total night minutes     2666 non-null   float64
 11  total night calls       2666 non-null   int64  
 12  total night charge      2666 non-null   float64
 13  total intl minutes      2666 non-null   float64
 14  total intl calls        2666 non-null   int64  
 15  total intl charge       2666 non-null   float64
 16  customer service calls  2666 non-null   int64  
 17  AK                      2666 non-null   float64
 18  AL                      2666 non-null   float64
 19  AR                      2666 non-null   float64
 20  AZ                      2666 non-null   float64
 21  CA                      2666 non-null   float64
 22  CO                      2666 non-null   float64
 23  CT                      2666 non-null   float64
 24  DC                      2666 non-null   float64
 25  DE                      2666 non-null   float64
 26  FL                      2666 non-null   float64
 27  GA                      2666 non-null   float64
 28  HI                      2666 non-null   float64
 29  IA                      2666 non-null   float64
 30  ID                      2666 non-null   float64
 31  IL                      2666 non-null   float64
 32  IN                      2666 non-null   float64
 33  KS                      2666 non-null   float64
 34  KY                      2666 non-null   float64
 35  LA                      2666 non-null   float64
 36  MA                      2666 non-null   float64
 37  MD                      2666 non-null   float64
 38  ME                      2666 non-null   float64
 39  MI                      2666 non-null   float64
 40  MN                      2666 non-null   float64
 41  MO                      2666 non-null   float64
 42  MS                      2666 non-null   float64
 43  MT                      2666 non-null   float64
 44  NC                      2666 non-null   float64
 45  ND                      2666 non-null   float64
 46  NE                      2666 non-null   float64
 47  NH                      2666 non-null   float64
 48  NJ                      2666 non-null   float64
 49  NM                      2666 non-null   float64
 50  NV                      2666 non-null   float64
 51  NY                      2666 non-null   float64
```

```
 52   OH                         2666 non-null   float64
 53   OK                         2666 non-null   float64
 54   OR                         2666 non-null   float64
 55   PA                         2666 non-null   float64
 56   RI                         2666 non-null   float64
 57   SC                         2666 non-null   float64
 58   SD                         2666 non-null   float64
 59   TN                         2666 non-null   float64
 60   TX                         2666 non-null   float64
 61   UT                         2666 non-null   float64
 62   VA                         2666 non-null   float64
 63   VT                         2666 non-null   float64
 64   WA                         2666 non-null   float64
 65   WI                         2666 non-null   float64
 66   WV                         2666 non-null   float64
 67   WY                         2666 non-null   float64
dtypes: float64(59), int64(9)
memory usage: 1.4 MB
```

In [32]:

```
y_train.value_counts()
```

Out[32]:

```
False    2284
True      382
Name: churn, dtype: int64
```

Encoding target column to binary

In [33]:

```
# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Fit and transform train data
y_train = label_encoder.fit_transform(y_train)

# Transform test data (using the same label encoder fitted on train data)
y_test = label_encoder.transform(y_test)
```

Feature Engineering

Getting new features

We will come up with columns with the features, for total call duration, average charge per local and international calls, total charges and tenure years.

On Train data

In [34]:

```
# Create new feature for total call duration
X_train['total_call_duration'] = X_train['total day minutes'] + X_train['t
#getting average call charges for international
# Calculate average charge per call for international calls
X_train['average_charge_per_intl_call'] = X_train['total intl charge'] / X
#getting average call charges for local
# Calculate average charge per call for local calls
local_call_charges= ['total day charge', 'total eve charge', 'total night
total_local_call_charges= X_train[local_call_charges].sum(axis=1)
total_local_calls = X_train[['total day calls', 'total eve calls', 'total
X_train['average_charge_per_local_call'] = total_local_call_charges / tota

# Calculate total charges
X_train['total_charges'] = (X_train['total day charge'] + X_train['total e
                            X_train['total night charge'] + X_train['total intl

# Convert account length to tenure in years
X_train['tenure_years'] = X_train['account length'] / 12
```

In [35]:

```
X_train.head()
```

Out[35]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	total eve calls	to e cha
817	243	0	0	0	95.5	92	16.24	163.7	63	13
1373	108	0	0	0	112.0	105	19.04	193.7	110	16
679	75	1	0	0	222.4	78	37.81	327.0	111	27
56	141	0	0	0	126.9	98	21.57	180.0	62	15
1993	86	0	0	0	216.3	96	36.77	266.3	77	22

5 rows × 73 columns

On Test Data

In [36]:

```
# Create new feature for total call duration
X_test['total_call_duration'] = X_test['total day minutes'] + X_test['tota
#getting average call charges for international
# Calculate average charge per call for international calls
X_test['average_charge_per_intl_call'] = X_test['total intl charge'] / X_t
#getting average call charges for local
# Calculate average charge per call for local calls
local_call_charges= ['total day charge', 'total eve charge', 'total night
total_local_call_charges= X_test[local_call_charges].sum(axis=1)
total_local_calls = X_test[['total day calls', 'total eve calls', 'total n
X_test['average_charge_per_local_call'] = total_local_call_charges / total

# Calculate total charges
X_test['total_charges'] = (X_test['total day charge'] + X_test['total eve
                           X_test['total night charge'] + X_test['total intl c

# Convert account length to tenure in years
X_test['tenure_years'] = X_test['account length'] / 12
```

In [37]:

```
# checking missing data
X_train.isnull().sum()
```

Out[37]:

```
account length                  0
international plan              0
voice mail plan                 0
number vmail messages           0
total day minutes               0
                               ..
total_call_duration             0
average_charge_per_intl_call   14
average_charge_per_local_call   0
total_charges                   0
tenure_years                    0
Length: 73, dtype: int64
```

After feature engineering, we notice that there is a column with some missing data, the 'average_charge_per_intl_call' column. We will therefore replace it with the median since it is less sensitive to outliers.

Removing missing data

On train data

In [38]:
```
# initialize imputer
imputer = SimpleImputer(strategy='median')

# Selecting the column to impute
column = ['average_charge_per_intl_call']

# Fit the imputer
imputer.fit(X_train[column])

# Transform the column by replacing missing values with the median
X_train [column] = imputer.transform(X_train[column])
```

On test data

In [39]:
```
# initialize imputer
imputer = SimpleImputer(strategy='median')

# Selecting the column to impute
column = ['average_charge_per_intl_call']

# Fit the imputer
imputer.fit(X_test[column])

# Transform the column by replacing missing values with the median
X_test [column] = imputer.transform(X_test[column])
```

In [40]:
```
# ensuring null values have been replaced
X_train.isnull().sum()
```

Out[40]:
```
account length                  0
international plan              0
voice mail plan                 0
number vmail messages           0
total day minutes               0
                               ..
total_call_duration             0
average_charge_per_intl_call    0
average_charge_per_local_call   0
total_charges                   0
tenure_years                    0
Length: 73, dtype: int64
```

Dealing with class imbalance

In [41]:
```
# balancing the classes using SMOTE
# instantiating SMOTE
smote = SMOTE(random_state=42)
# fitting SMOTE on our data set
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train
```

In [42]:
```
# seeing if class imbalance has been solved
pd.Series(y_train_resampled).value_counts()
```

Out[42]:
```
1    2284
0    2284
dtype: int64
```

The missing data have been removed

Model Iterations

1). Logistic Regression Model.

**A). With Imbalanced Class Instances&Without Hyperparameter Tuning.

In [43]:
```
# Create a pipeline with StandardScaler and LogisticRegression
pipeline = make_pipeline(StandardScaler(), LogisticRegression(random_state
# Fit the model
pipeline.fit(X_train, y_train)
```

Out[43]:
```
▸ Pipeline
  ▸ StandardScaler
  ▸ LogisticRegression
```

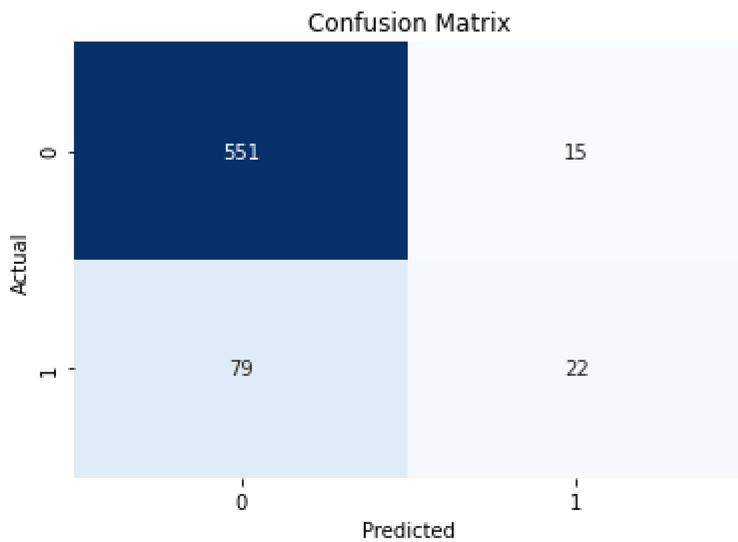
In [44]:
```
# Predictions on the testing set
y_pred = pipeline.predict(X_test)
```

In [45]:
```
# Evaluate model performance
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)
```

Confusion Matrix

In [46]:

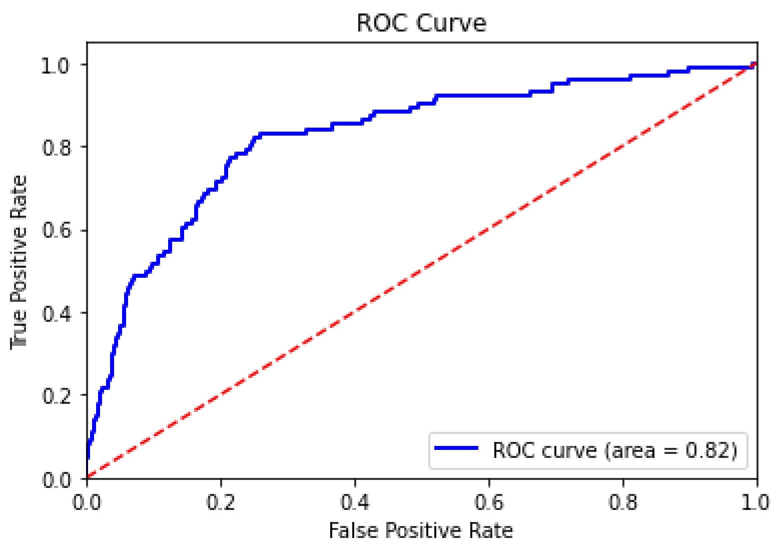
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
# Calculate confusion matrix
cm = confusion_matrix(y_test, y_pred)
# Plot confusion matrix
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g', cbar=False)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```



ROC Curve

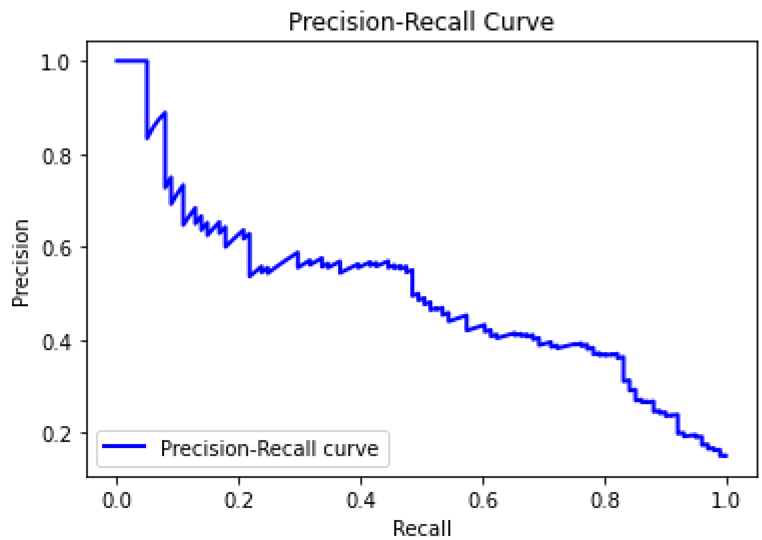
In [47]:

```
# Calculate probabilities for class 1
y_probs = pipeline.predict_proba(X_test)[:, 1]
# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_probs)
# Calculate AUC
roc_auc = auc(fpr, tpr)
# Plot ROC curve
plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (area = %0.2f)' %
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.show()
```



Precision-Recall Curve

In [48]:
```
# Calculate precision-recall curve
precision, recall, _ = precision_recall_curve(y_test, y_probs)
# Plot precision-recall curve
plt.plot(recall, precision, color='blue', lw=2, label='Precision-Recall cu
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend(loc='lower left')
plt.show()
```



In [49]:
```
# Print evaluation metrics
print("Accuracy:", accuracy)
print("Recall:", recall.mean())
print("F1 Score:", f1)
```

```
Accuracy: 0.8590704647676162
Recall: 0.7736853026620026
F1 Score: 0.31884057971014496
```

B).With Hyperparameter Tuning using GridSearchCV and Balanced class Instances:

In [50]:
```
# Define hyperparameters grid for Grid Search
param_grid ={
'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Regularization parameter
'penalty': ['l2']  # Only 'l2' penalty for lbfgs solver
}
```

In [51]:
```
# Train a Logistic Regression model with hyperparameter tuning using GridS
log_reg = LogisticRegression(solver='liblinear', random_state=42)
grid_search = GridSearchCV(log_reg, param_grid, cv=5, scoring='accuracy',
grid_search.fit(X_train_resampled, y_train_resampled)
```

Out[51]:
```
▸ GridSearchCV
▸ estimator: LogisticRegression
    ▸ LogisticRegression
```

In [52]:
```
# Best hyperparameters found
best_params = grid_search.best_params_
```

In [53]:
```
# Train the model with the best hyperparameters
best_log_reg = LogisticRegression(solver='liblinear', max_iter=1000, **bes
best_log_reg.fit(X_train_resampled, y_train_resampled)
```

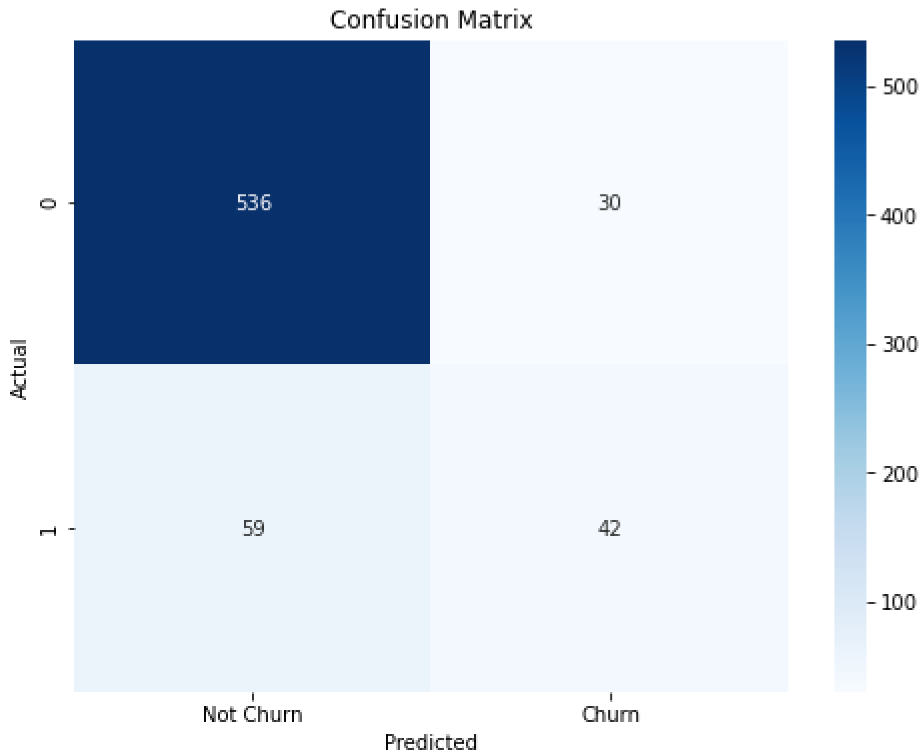
Out[53]:
```
▾ LogisticRegression
LogisticRegression(C=100, max_iter=1000, random_state=42, solver='libli
near')
```

In [54]:
```
# Predictions on the testing set
y_pred = best_log_reg.predict(X_test)
```

Confusion Matrix

In [55]:

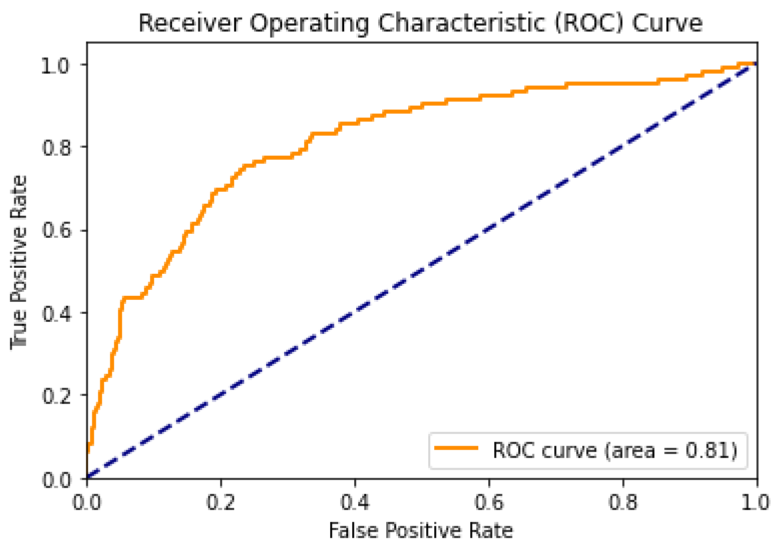
```
# Calculate confusion matrix
cm = confusion_matrix(y_test, y_pred)
# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Churn
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```



ROC Curve

In [56]:

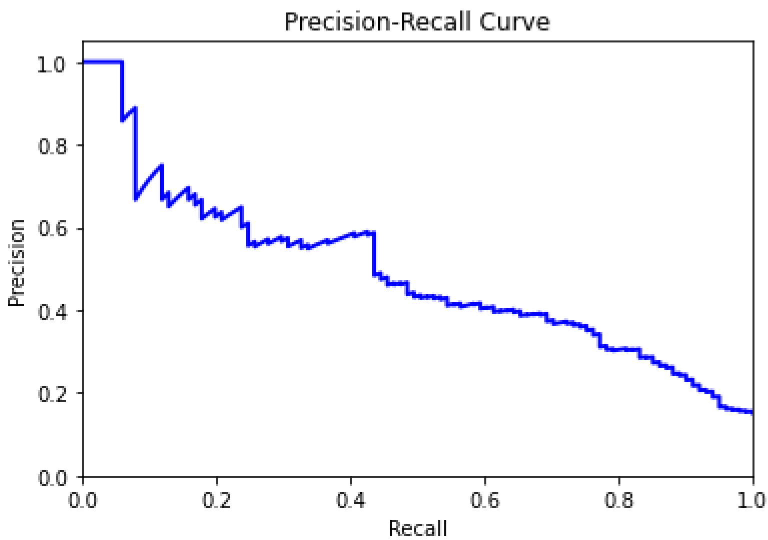
```
# Calculate predicted probabilities for the positive class
y_pred_proba = best_log_reg.predict_proba(X_test)[:, 1]
# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)
# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```



Precision-Recall Curve

In [57]:

```
# Calculate precision-recall curve
precision, recall, _ = precision_recall_curve(y_test, y_pred_proba)
# Calculate predicted probabilities for the positive class
y_pred_proba = best_log_reg.predict_proba(X_test)[:, 1]
# Plot precision-recall curve
plt.figure()
plt.plot(recall, precision, color='blue', lw=2)
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Precision-Recall Curve')
plt.show()
```



In [58]:

```
# Evaluate model performance
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)
# Print evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 0.8665667166416792
Precision: 0.5833333333333334
Recall: 0.4158415841584158
F1 Score: 0.4855491329479768
```

Decision Trees Model

With balanced Class Instances and Parameter tuning.

In [59]:
```
# Train Decision Tree Classifier
dt_classifier = DecisionTreeClassifier(random_state=42)
dt_classifier.fit(X_train_resampled, y_train_resampled)
```

Out[59]:
```
DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

In [60]:
```
# Hyperparameter Tuning
param_grid ={
'criterion': ['gini', 'entropy'],
'max_depth': [None, 10, 20, 30, 40, 50],
'min_samples_split': [2, 5, 10],
'min_samples_leaf': [1, 2, 4],
'max_features': [None, 'sqrt', 'log2']}  # Remove 'auto' as a value
grid_search = GridSearchCV(dt_classifier, param_grid, cv=5, scoring='accur
grid_search.fit(X_train_resampled, y_train_resampled)
```

Out[60]:
```
GridSearchCV
estimator: DecisionTreeClassifier
DecisionTreeClassifier
```

In [61]:
```
# Best hyperparameters found
best_params = grid_search.best_params_
```

In [62]:
```
# Train the model with the best hyperparameters
best_dt_classifier = DecisionTreeClassifier(**best_params, random_state=42
best_dt_classifier.fit(X_train_resampled, y_train_resampled)
```

Out[62]:
```
DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

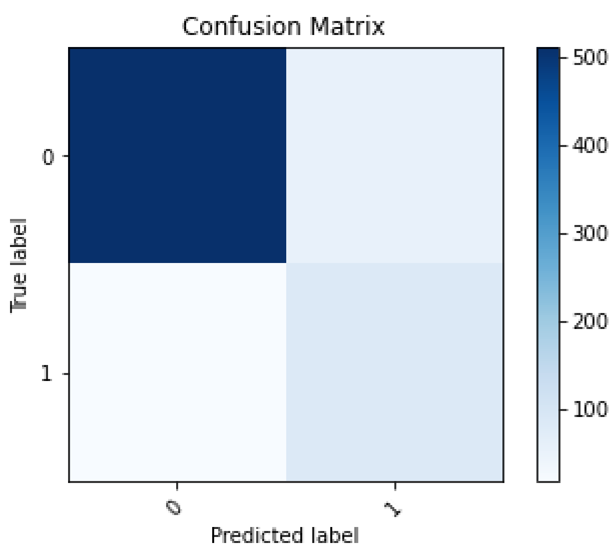
In [63]:
```
# Predictions on the testing set
y_pred = best_dt_classifier.predict(X_test)
```

In [64]:
```
# Evaluate model performance
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)
```

Confusion Matrix

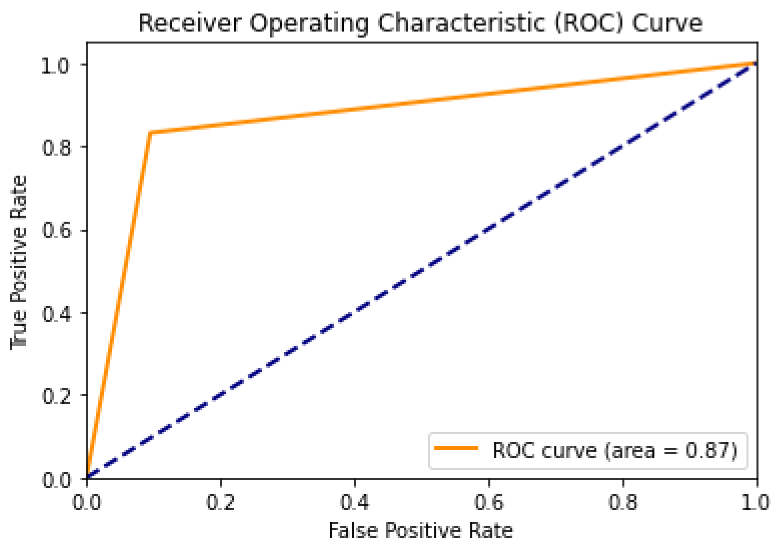
In [65]:

```
# Confusion Matrix
import numpy as np
cm = confusion_matrix(y_test, best_dt_classifier.predict(X_test))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
tick_marks = range(len(np.unique(y_test)))  # Assuming y_test contains the
plt.xticks(tick_marks, np.unique(y_test), rotation=45)
plt.yticks(tick_marks, np.unique(y_test))
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.show()
```



ROC Curve

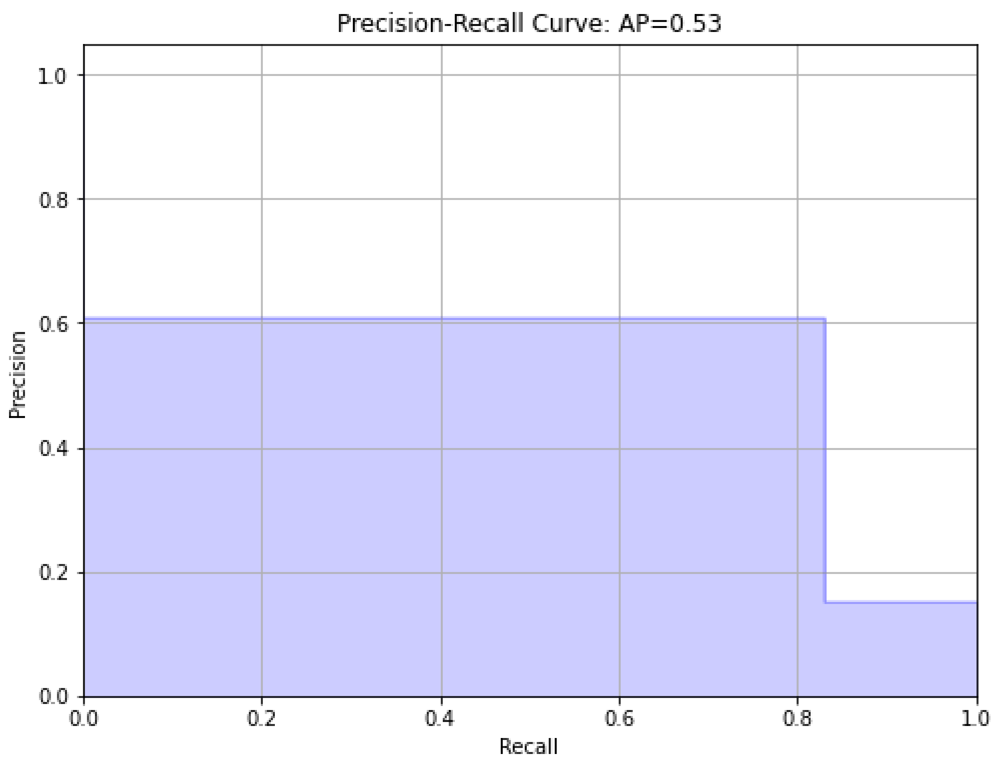
In [66]:
```
# Compute ROC curve and ROC area for each class
fpr, tpr, _ = roc_curve(y_test, best_dt_classifier.predict_proba(X_test)[:
roc_auc = auc(fpr, tpr)
# Plot ROC curve
plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange', lw=lw, label='ROC curve (area = %0.
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```



Precision-Recall Curve

In [67]:

```
# Predict probabilities
y_score = best_dt_classifier.predict_proba(X_test)[:, 1]
# Calculate precision-recall curve
precision, recall, _ = precision_recall_curve(y_test, y_score)
# Calculate average precision score
average_precision = average_precision_score(y_test, y_score)
# Plot Precision-Recall curve
plt.figure(figsize=(8, 6))
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2, color='b')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title(f'Precision-Recall Curve: AP={average_precision:.2f}')
plt.grid(True)
plt.show()
```



In [68]:

```
# Print evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 0.8935532233883059
Precision: [0.15142429 0.60869565 1.        ]
Recall: [1.         0.83168317 0.        ]
F1 Score: 0.7029288702928871
```

Random Forest Model and XGBoost before Any Tuning

We chose to use Random Forest and XGBoost since they are boost the performance of decision trees. And hence we wish to explore if the models will outperform decisiontree classifier as the baseline model.

We will run Random Forest classifier and XGBoost with all features and default parameters to see how it performs before tuning it.

In [69]:

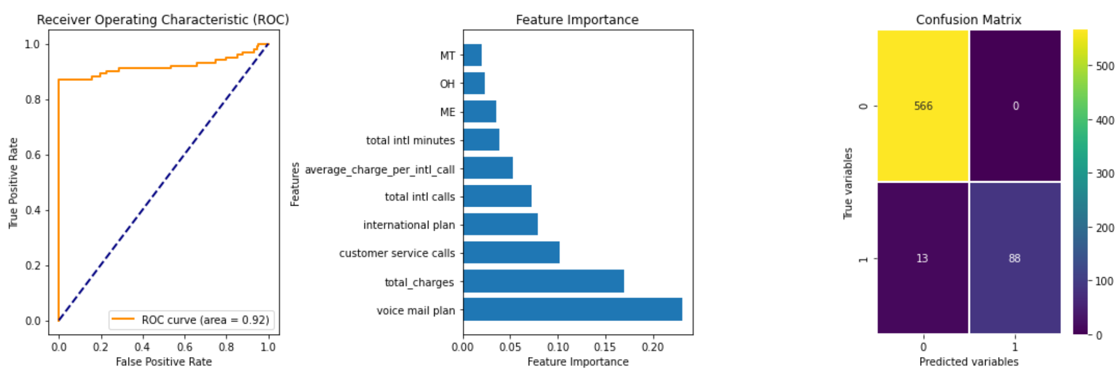
```
# definition a function for creating modesl
def create_models(seed=42):
    models =[]
    #appending the models to the model list.
    models.append((' XGB',  XGBClassifier(random_state=seed)))
    models.append(('random_forest', RandomForestClassifier(random_state=se
    return models
models= create_models()
```

In [70]:

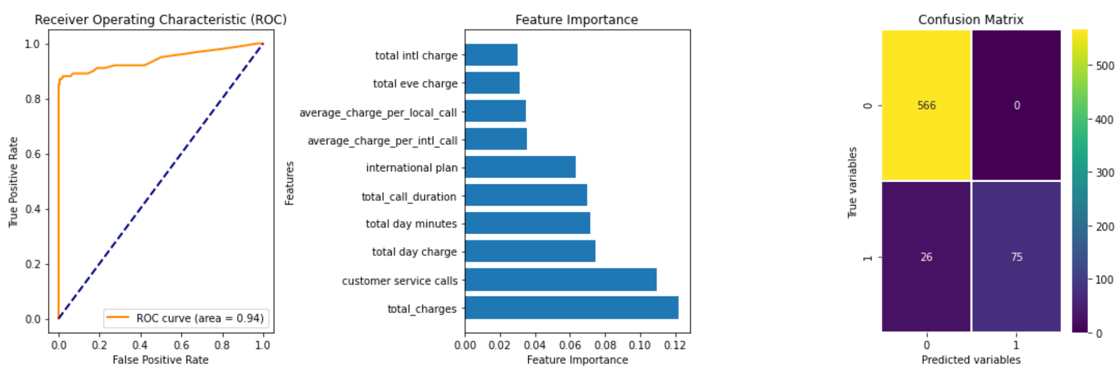
```
# creating a list of results, model name, and accuracy score
results = []
names = []
scoring = 'accuracy'

# Create a figure for each model
for name, model in models:
    # Create a figure with multiple subplots
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    # Fit model with training data
    model.fit(X_train, y_train)
    # Make predictions with testing data
    predictions = model.predict(X_test)
    # Calculate accuracy
    accuracy = accuracy_score(y_test, predictions)
    # Append model name and accuracy to the lists
    results.append(accuracy)
    names.append(name)
    # Print classifier accuracy
    print('Classifier: {}, Accuracy: {}'.format(name, accuracy))
    print(classification_report(y_test, predictions))

    # Calculate predicted probabilities for positive class
    y_proba = model.predict_proba(X_test)[:, 1]
    # getting fpr and tpr for roc
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    axes[0].plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (are
    axes[0].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    axes[0].set_xlabel('False Positive Rate')
    axes[0].set_ylabel('True Positive Rate')
    axes[0].set_title('Receiver Operating Characteristic (ROC)')
    axes[0].legend(loc="lower right")

    # Feature importance for models that support it
    if hasattr(model, 'feature_importances_'):
        feature_importance = model.feature_importances_
        # Zip feature names and their importance scores
        feature_importance_dict = dict(zip(X_train.columns, feature_import
        # Sort feature importance in descending order
        sorted_feature_importance = sorted(feature_importance_dict.items()
        # Print feature importance
        print('Top 10 Features Importance:')
        for feature, importance in sorted_feature_importance[:10]:
            print('{}: {:.4f}'.format(feature, importance))

        # Plot feature importance
        features = [x[0] for x in sorted_feature_importance[:10]]
        importance = [x[1] for x in sorted_feature_importance[:10]]
        axes[1].barh(features, importance)
        axes[1].set_xlabel('Feature Importance')
        axes[1].set_ylabel('Features')
        axes[1].set_title('Feature Importance')
```

```
    # Visualizing model performance using confusion matrix
    conf_matrix = confusion_matrix(y_test, predictions)
    sns.heatmap(conf_matrix, annot=True, cmap="viridis", fmt="d", linewidt
    axes[2].set_title("Confusion Matrix")
    axes[2].set_xlabel("Predicted variables")
    axes[2].set_ylabel("True variables")

    # Adjust layout and spacing
    plt.tight_layout()
    # Display the plot
    plt.show()
```

```
Classifier:  XGB, Accuracy: 0.9805097451274363
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       566
           1       1.00      0.87      0.93       101

    accuracy                           0.98       667
   macro avg       0.99      0.94      0.96       667
weighted avg       0.98      0.98      0.98       667

Top 10 Features Importance:
voice mail plan: 0.2306
total_charges: 0.1699
customer service calls: 0.1016
international plan: 0.0790
total intl calls: 0.0720
average_charge_per_intl_call: 0.0530
total intl minutes: 0.0386
ME: 0.0354
OH: 0.0232
MT: 0.0197
```



```
Classifier: random_forest, Accuracy: 0.9610194902548725
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       566
           1       1.00      0.74      0.85       101

    accuracy                           0.96       667
   macro avg       0.98      0.87      0.91       667
weighted avg       0.96      0.96      0.96       667

Top 10 Features Importance:
total_charges: 0.1223
customer service calls: 0.1094
total day charge: 0.0745
total day minutes: 0.0717
total_call_duration: 0.0696
international plan: 0.0632
average_charge_per_intl_call: 0.0356
average_charge_per_local_call: 0.0350
total eve charge: 0.0312
total intl charge: 0.0300
```



From the plots above, we notice that from the ROC curve, we have some false positives which affects the true positive rate for both Random Forest and XGBoost. We see that voice mail plan is the most important feature for XGBoost and total charges is the most important feature for Random Forest classifier.We also notice that the oversampled class which is 0, performs better than the undersampled class.

Evaluation On train data

In [71]:
```
# Create a figure for each model
for name, model in models:
    # Fit model with training data
    model.fit(X_train, y_train)
    # Make predictions with testing data
    predict_train = model.predict(X_train)
    # Calculate metrics
    accuracy = accuracy_score(y_train, predict_train)
    precision = precision_score(y_train, predict_train)
    recall = recall_score(y_train, predict_train)
    f1 = f1_score(y_train, predict_train)
    # Print classifier name and metrics
    print('Classifier:', name)
    print('Accuracy:', accuracy)
    print('Precision:', precision)
    print('Recall:', recall)
    print('F1-score:', f1)
```

```
Classifier:  XGB
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1-score: 1.0
Classifier: random_forest
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1-score: 1.0
```

From the metrics. All the metrics have perfect scores for the train model

Evaluation On test data

In [72]:
```
# Create a figure for each model for test data
for name, model in models:
    # Fit model with training data
    model.fit(X_train, y_train)
    # Make predictions with testing data
    predictions = model.predict(X_test)
    # Calculate metrics
    accuracy = accuracy_score(y_test, predictions)
    precision = precision_score(y_test, predictions)
    recall = recall_score(y_test, predictions)
    f1 = f1_score(y_test, predictions)
    # Print classifier name and metrics
    print('Classifier:', name)
    print('Accuracy:', accuracy)
    print('Precision:', precision)
    print('Recall:', recall)
    print('F1-score:', f1)
```

```
Classifier:  XGB
Accuracy: 0.9805097451274363
Precision: 1.0
Recall: 0.8712871287128713
F1-score: 0.9312169312169313
Classifier: random_forest
Accuracy: 0.9610194902548725
Precision: 1.0
Recall: 0.7425742574257426
F1-score: 0.8522727272727273
```

From the evaluation metrics above, XGBoost has an accuracy score of 0.9805097451274363, but a recall of 0.8712871287128713, for RandomForest, we have an accuracy score of 0.9610194902548725 which seems okay but recall is at 0.7425742574257426 this means that we have a low true positive rate for the two models. However we have class imbalance that could be affecting our performance metrics and hence we will first balance the classes and see how the models perform.But there is a huge difference on train data.

Random Forest Model after Parameter Tuning

We are going to run our model now, after tuning it to see if the performance improves after model tuning. We will use random search to find the best parameters for the model.

On train data and test data

In [73]:

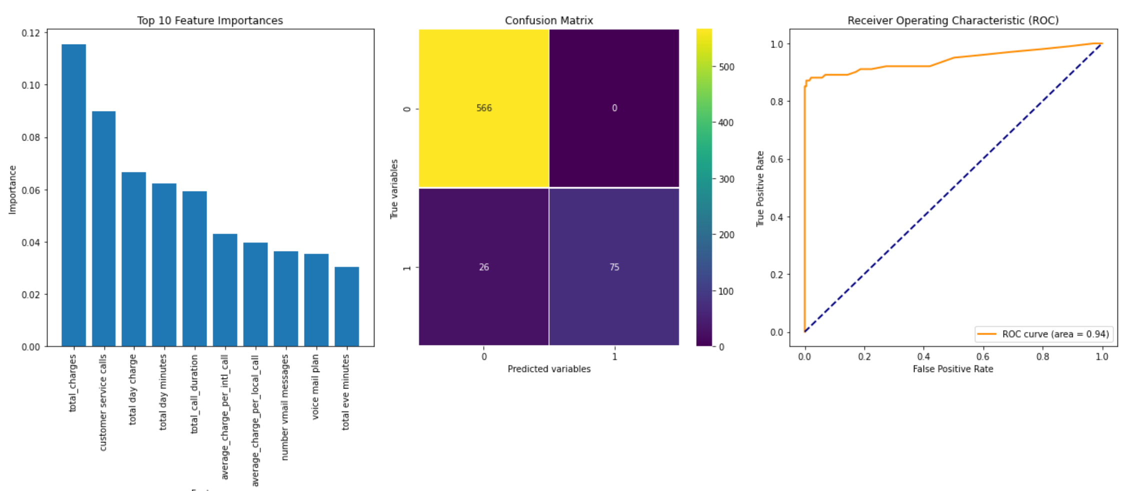
```
# Define a function to perform random search and evaluate the model
def perform_grid_search(classifier, param_grid):
    # Define the pipeline
    pipe = Pipeline([
        ('scaler', StandardScaler()),
        ('classifier', classifier)
    ])

    # Perform GridSearchCV
    random_search = GridSearchCV(estimator = pipe,
                       param_grid=param_grid,
                       scoring = 'accuracy',
                       cv=5)
    random_search.fit(X_train_resampled, y_train_resampled)

    # Get the best parameters
    best_params = random_search.best_params_
    print("Best Parameters:", best_params)

    # Evaluate the model on the test set
    y_pred = random_search.predict(X_test)
    y_pred_train = random_search.predict(X_train)

    # Calculate evaluation metrics for test
    accuracy_test = accuracy_score(y_test, y_pred)
    precision_test = precision_score(y_test, y_pred)
    recall_test = recall_score(y_test, y_pred)
    f1_test = f1_score(y_test, y_pred)

    # Calculate evaluation metrics for train        n
    accuracy_train = accuracy_score(y_train, y_train)
    precision_train = precision_score(y_train, y_pred_train)
    recall_train= recall_score(y_train, y_pred_train)
    f1_train = f1_score(y_train, y_pred_train)

    # Print evaluation metrics for test
    print("Test Accuracy:", accuracy_test)
    print("Test Precision:", precision_test)
    print("Test Recall:", recall_test)
    print("Test F1 Score:", f1_test)

    # Print evaluation metrics for train
    print("Train Accuracy:", accuracy_train)
    print("Train Precision:", precision_train)
    print("Train Recall:", recall_train)
    print("Train F1 Score:", f1_train)

    # Classification report
    print("Classification Report:")
    #print(classification_report(y_test, y_pred))


    return  best_params, accuracy, random_search
```

In [74]:

```
classifier= RandomForestClassifier(random_state=42)
param_grid = [{'classifier__max_depth': [None, 2,6,10],
        'classifier__min_samples_split': [5,10]}]
best_params, accuracy, random_search = perform_grid_search(classifier, par
```

```
Best Parameters: {'classifier__max_depth': None, 'classifier__min_sample
s_split': 5}
Test Accuracy: 0.95952023988006
Test Precision: 0.9868421052631579
Test Recall: 0.7425742574257426
Test F1 Score: 0.8474576271186441
Train Accuracy: 1.0
Train Precision: 1.0
Train Recall: 1.0
Train F1 Score: 1.0
Classification Report:
```

In [75]:
```
# Plot horizontal bar graph
plt.figure(figsize=(18, 8))
plt.subplot(1, 3, 1)
n_of_features = 10
feature_importances = random_search.best_estimator_['classifier'].feature_
# Get indices of top 10 features
indices = np.argsort(feature_importances)[::-1][:n_of_features]
# getting labels
plt.title("Top 10 Feature Importances")
plt.bar(range(n_of_features), feature_importances[indices], align="center'
plt.xticks(range(n_of_features), X_train.columns[indices], rotation=90)
plt.xlabel("Features")
plt.ylabel("Importance")

# visualizing model performance using confusion matrix
plt.subplot(1, 3, 2)
conf_matrix = confusion_matrix(y_test, predictions)
#plotting heatmap
sns.heatmap(conf_matrix, annot=True, cmap="viridis", fmt="d", linewidths=.
# setting the labels
plt.title("Confusion Matrix")
plt.xlabel("Predicted variables")
plt.ylabel("True variables")
# plot for ROC curve
plt.subplot(1, 3, 3)
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc="lower right")
# displaying the data
plt.tight_layout()
plt.show()
```



After tuning our model and solving class imbalance, Random Forest model, Accuracy is at 0.9565217391304348 which is a very slight decrease from the one with default parameters. Precision has dropped to 0.9390243902439024 and Recall has increased slightly to

0.7623762376237624 . For train data, the metrics still score 100% percent, a possibility of

XGBoost model after Parameter Tuning

Hyperparameters :learning rate,max_depth,min_child weight,subsample,number of trees(n_estimators). GridSearchCV - search through a predefined hyperparameter grid, and the best parameters are selected based on accuracy.

In [76]:

```
# Instantiate XGBClassifier
clf = XGBClassifier()
# Fit XGBClassifier
clf.fit(X_train_resampled, y_train_resampled)
# Predict on training and test sets
training_preds = clf.predict(X_train_resampled)
test_preds = clf.predict(X_test)
```

In [77]:

```
# Define the hyperparameter grid
param_grid = {
'learning_rate': [0.1, 0.2],
'max_depth': [6,8],
'min_child_weight': [1, 2],
'subsample': [0.5, 0.7],
'n_estimators': [100],
}
# Create the GridSearchCV object
grid_clf = GridSearchCV(clf, param_grid,scoring='accuracy', cv=5)

# Fit the GridSearchCV object to the data
grid_clf.fit(X_train_resampled, y_train_resampled)
grid_clf.fit(X_test,y_test)

best_parameters = grid_clf.best_params_

best_parameters
```

Out[77]:

```
{'learning_rate': 0.1,
 'max_depth': 8,
 'min_child_weight': 1,
 'n_estimators': 100,
 'subsample': 0.7}
```

After performing grid search and finding the best hyperparameters using GridSearchCV,the best model is used to make predictions on both the training and test sets.Accuracy scores are calculated for both the training and test sets

In [78]:

```
# Evaluate the model on the test set
y_pred = grid_clf.predict(X_test)

# Calculate evaluation metrics
train_accuracy = accuracy_score(y_train_resampled, training_preds)
train_recall = recall_score(y_train_resampled, training_preds)
train_precision = precision_score(y_train_resampled, training_preds)
train_f1 = f1_score(y_train_resampled, training_preds)
test_accuracy = accuracy_score(y_test, y_pred)
test_recall = recall_score(y_test, y_pred)
test_precision = precision_score(y_test, y_pred)
test_f1 = f1_score(y_test, y_pred)

# Print evaluation metrics
print("Training Accuracy:", train_accuracy)
print("Train Recall:", train_recall)
print("Train Precision:",train_precision)
print("Train F1 Score:", train_f1)
print("Test Accuracy:", test_accuracy)
print("Test Recall:", test_recall)
print("Test Precision:",test_precision)
print("Test F1 Score:", test_f1)

# Classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

```
Training Accuracy: 0.9995621716287215
Train Recall: 0.999124343257443
Train Precision: 1.0
Train F1 Score: 0.9995619798510731
Test Accuracy: 0.9985007496251874
Test Recall: 0.9900990099009901
Test Precision: 1.0
Test F1 Score: 0.9950248756218906
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       566
           1       1.00      0.99      1.00       101

    accuracy                           1.00       667
   macro avg       1.00      1.00      1.00       667
weighted avg       1.00      1.00      1.00       667
```

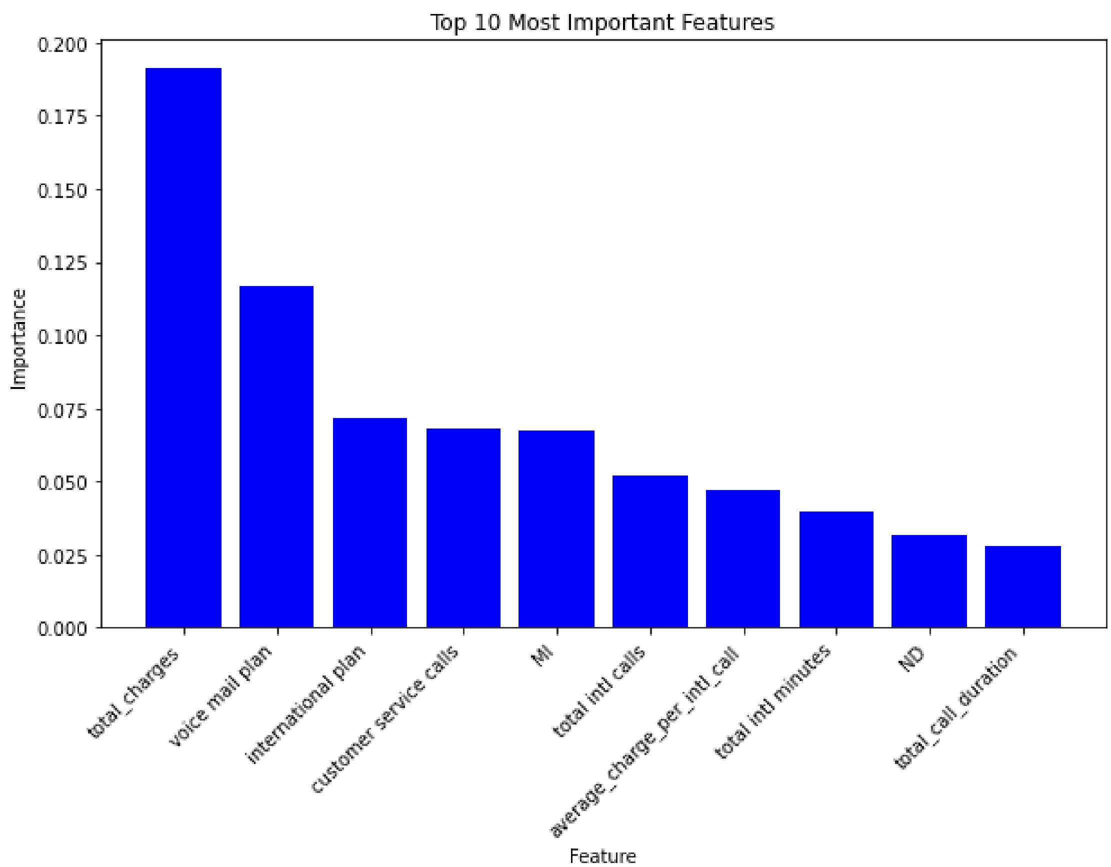
The model performs exceptionally well on unseen data in the testing dataset, maintaining a high accuracy of 99.85%.Recall metrics captures about 99.01% of the actual positive instances in the testing data.The model demonstrates exceptional accuracy, recall, and precision on both the training and testing datasets, indicating its ability to correctly identify positive instances.

In [79]:

```
# Get the best model
best_model = grid_clf.best_estimator_

# Get feature importances from the best model
feature_importance = best_model.feature_importances_

# Assuming X_train_resampled is your training feature matrix
# Create a DataFrame to associate feature names with their importances
feature_importance_df = pd.DataFrame({'Feature': X_train_resampled.columns


# Sort the DataFrame by importance in descending order
feature_importance_df = feature_importance_df.sort_values(by='Importance',

# Print or visualize the feature importances
print("Feature Importances:")
print(feature_importance_df)
```

```
Feature Importances:
                   Feature  Importance
71           total_charges    0.191405
2          voice mail plan    0.116485
1       international plan    0.071988
16  customer service calls    0.067753
39                      MI    0.067121
..                     ...         ...
35                      LA    0.000000
37                      MD    0.000000
38                      ME    0.000000
40                      MN    0.000000
72            tenure_years    0.000000

[73 rows x 2 columns]
```

In [80]:
```
#Plotting the feature importance for Top 10 most important columns
# Select the top 10 features
top_10_features = feature_importance_df.head(10)
# Plot the feature importance using a bar graph
plt.figure(figsize=(10, 6))
plt.bar(top_10_features['Feature'], top_10_features['Importance'], color='
plt.xlabel('Feature')
plt.ylabel('Importance')
plt.title('Top 10 Most Important Features')
plt.xticks(rotation=45, ha='right')
plt.show()
```



K-Nearest Model.

A).With Balanced Class Instances,without hyperparameters.

Training the model

In [81]:
```
# KNN classifier
knn_classifier =KNeighborsClassifier(metric='manhattan',
n_neighbors=3, weights='distance')
```

In [82]:

```
# Training the classifier
knn_classifier.fit(X_train_resampled, y_train_resampled)
```

Out[82]:

```
KNeighborsClassifier

KNeighborsClassifier(metric='manhattan', n_neighbors=3, weights='distan
ce')
```

In [83]:

```
#Predict values
y_pred =knn_classifier.predict(X_test)
y_pred_train =knn_classifier.predict(X_train)
```

In [84]:

```
# Calculate evaluation metrics for test
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)
# Print evaluation metrics
print("Test Accuracy:", accuracy)
print(" Test Precision:", precision)
print("Test Recall:", recall)
print("Test F1 Score:", f1)
# Classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))

# for train
# Calculate evaluation metrics for test
accuracy = accuracy_score(y_train, y_pred_train)
precision = precision_score(y_train, y_pred_train)
recall = recall_score(y_train, y_pred_train)
f1 = f1_score(y_train, y_pred_train)
# Print evaluation metrics
print("Train Accuracy:", accuracy)
print("Train Precision:", precision)
print("Train Recall:", recall)
print("Train F1 Score:", f1)
# Classification report
print("Classification Report:")
print(classification_report(y_train, y_pred_train))
```

```
Test Accuracy: 0.6881559220389805
 Test Precision: 0.26222222222222225
Test Recall: 0.5841584158415841
Test F1 Score: 0.3619631901840491
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.71      0.79       566
           1       0.26      0.58      0.36       101

    accuracy                           0.69       667
   macro avg       0.58      0.65      0.58       667
weighted avg       0.81      0.69      0.73       667

Train Accuracy: 1.0
Train Precision: 1.0
Train Recall: 1.0
Train F1 Score: 1.0
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2284
           1       1.00      1.00      1.00       382

    accuracy                           1.00      2666
   macro avg       1.00      1.00      1.00      2666
weighted avg       1.00      1.00      1.00      2666
```

B).B).With Balanced Class Instances and hyperparameters.**

In [85]:
```
# Define the parameter grid
param_grid ={
'n_neighbors': [3, 5, 7, 9, 11],
'weights': ['uniform', 'distance'],'metric': ['euclidean', 'manhattan', 'm
}
```

In [86]:
```
# KNN model
knn =KNeighborsClassifier()
```

In [87]:
```
# GridSearchCV
grid_search =GridSearchCV(knn, param_grid, cv=5, scoring='accuracy')
```

In [88]:
```
# Fit the model
grid_search.fit(X_train_resampled, y_train_resampled)
```

Out[88]:
```
▸ GridSearchCV
▸ estimator: KNeighborsClassifier
    ▸ KNeighborsClassifier
```

In [89]:
```
# Get the best parameters
best_params =grid_search.best_params_
print("Best Parameters:", best_params)
```

```
Best Parameters: {'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'd
istance'}
```

In [90]:
```
# Predict using the trained model
y_pred =grid_search.predict(X_test)
y_pred_train =grid_search.predict(X_train_resampled)
```

Evaluate Model Performance:

In [92]:

```
#for test data
print("Classification Report:")
print(classification_report(y_test, y_pred))
accuracy =accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
print("Test Accuracy:", accuracy )
print("Test Recall:", recall )
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("Classification Report:")
print(classification_report(y_test, y_pred))
# for train data
print("Classification Report:")
print(classification_report(y_train_resampled, y_pred_train))
accuracy =accuracy_score(y_train_resampled, y_pred_train)
recall = recall_score(y_train_resampled, y_pred_train)
print("Train Recall:", recall )
print("Train Accuracy:", accuracy )
print("Confusion Matrix:")
print(confusion_matrix(y_train_resampled, y_pred_train))
print("Classification Report:")
print(classification_report(y_train_resampled, y_pred_train))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.71      0.79       566
           1       0.26      0.58      0.36       101

    accuracy                           0.69       667
   macro avg       0.58      0.65      0.58       667
weighted avg       0.81      0.69      0.73       667

Test Accuracy: 0.6881559220389805
Test Recall: 0.5841584158415841
Confusion Matrix:
[[400 166]
 [ 42  59]]
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.71      0.79       566
           1       0.26      0.58      0.36       101

    accuracy                           0.69       667
   macro avg       0.58      0.65      0.58       667
weighted avg       0.81      0.69      0.73       667

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2284
           1       1.00      1.00      1.00      2284

    accuracy                           1.00      4568
   macro avg       1.00      1.00      1.00      4568
weighted avg       1.00      1.00      1.00      4568

Train Recall: 1.0
Train Accuracy: 1.0
Confusion Matrix:
[[2284    0]
 [   0 2284]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2284
           1       1.00      1.00      1.00      2284

    accuracy                           1.00      4568
   macro avg       1.00      1.00      1.00      4568
weighted avg       1.00      1.00      1.00      4568
```

Confusion matrix

In [93]:
```
# Create the confusion matrix for test
conf_matrix_test =confusion_matrix(y_test, y_pred)
# Print the confusion matrix
print("Confusion Matrix:")
print(conf_matrix)
```

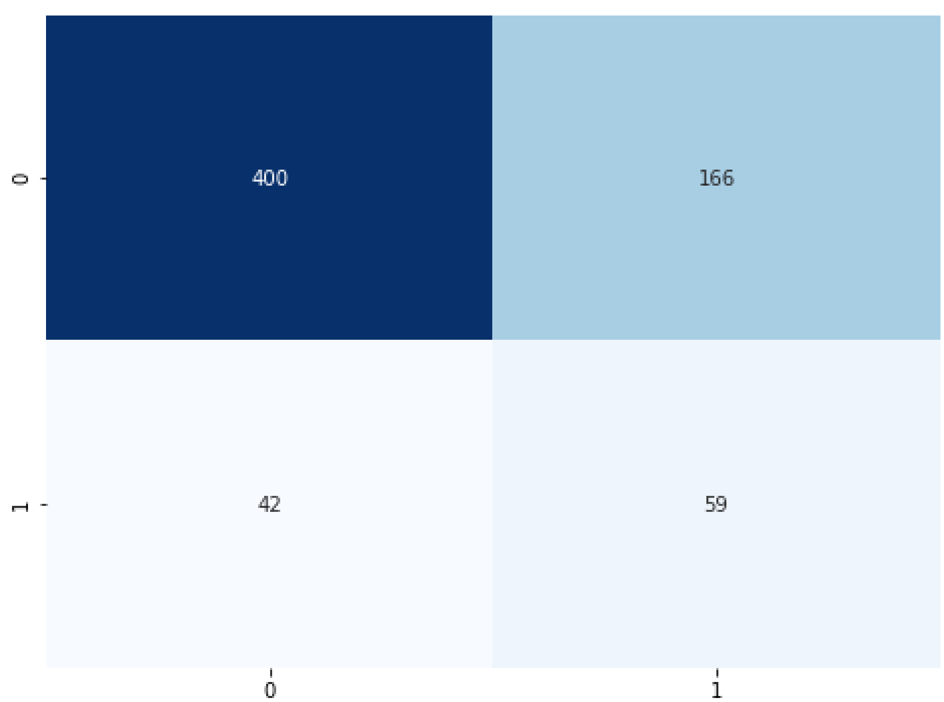
```
Confusion Matrix:
[[566   0]
 [ 26  75]]
```

In [94]:
```
# Create the confusion matrix for train
conf_matrix_train =confusion_matrix(y_train_resampled, y_pred_train)
# Print the confusion matrix
print("Confusion Matrix:")
print(conf_matrix)
```

```
Confusion Matrix:
[[566   0]
 [ 26  75]]
```

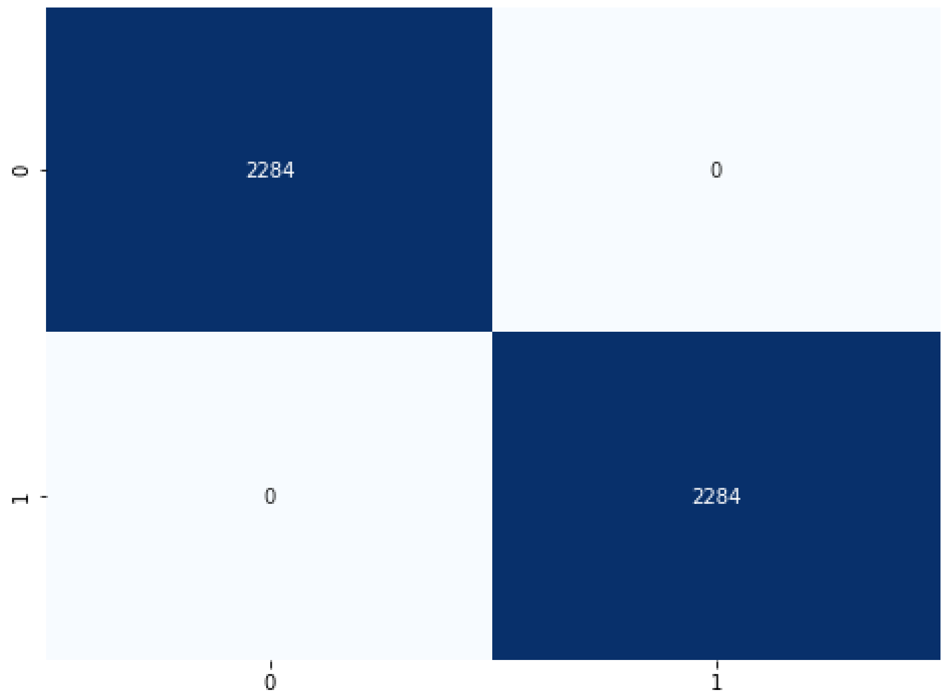
In [95]:
```
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_test, annot=True, fmt='d', cmap='Blues',
cbar=False)
```

Out[95]: <AxesSubplot:>



In [96]:
```
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_train, annot=True, fmt='d', cmap='Blues', cbar=False)
```

Out[96]: <AxesSubplot:>



Conclusions

Model evaluation.

From the models above,we used precision and recall on the test set before and after tuning to identify the best suitable model for the project.

For the baseline modelsLogistic regression had an acccuracy of 84 % and a recall of 77% before tuning,accuracy of 86% and a recall of 41% after tuning.

For decision trees with hyperparameter tuning,the model had accuracy of 0.8935532233883059 and recall of 0.83168317

Random forest had an accuracy of 95.65% and recall of 76.23% after tuning the model.

KNN before tuning has Test Accuracy: 0.6881559220389805 Test Recall: 0.5841584158415841 for train it is 100% performance on both metrics before tuning After tuning KNN had an accuracy of 0.6881559220389805 Recall: 0.5841584158415841 on test data and 100% for both on train data. Meaning there is absolutely no improvement .In general it is performing poorly.

XGboost had a precision of 100% and a recall of 87.12% on the test set before tuning and precision of 100%,recall of 99.00% after tuning the model.

For all the models all the metrics were 100% for train data.

Decision trees performs showing high precision and recall after tuning. Logistic regression was the least performing model with a lowest recall before and after tuning the model indicating that it is not effectively capturing positive instances Random forest shows improvement in precision after tuning, making it more precise while maintaining recall. XGBoost peforms better before tuning with a significant improvement on recall after tuning making it the most suitable ensemble model for prediction.

Best model

XGBooost

Training Accuracy: 0.9995621716287215

Train Recall: 0.999124343257443

Train Precision: 1.0

Train F1 Score: 0.9995619798510731

Test Accuracy: 0.9985007496251874

Test Recall: 0.9900990099009901

Test Precision: 1.0

Test F1 Score: 0.9950248756218906

XGBoost peforms better before tuning with a significant improvement on recall after tuning making it the most suitable ensemble model for prediction.

Feature importanace

Based on this model,the top three factors that contributed to customer churning are:Total charges,voicemail plan and the international plan.

Recommendations

Enhance Service Quality: Focus on improving the quality of service for features like total charges, voice mail plans, and international plans. Ensuring these services meet or exceed customer expectations can help retain existing customers and attract new ones.

Personalized Offerings: Utilize the insights gained from the analysis to tailor personalized offerings or promotions targeting customers who are at risk of churning. Offering incentives or discounts on international plans or voice mail services could encourage customers to stay with SyriaTel.

Customer Engagement: Implement strategies to increase customer engagement and satisfaction. This could include regular communication through personalized messages or emails, seeking feedback to address concerns promptly, and providing timely customer support.