BUSINESS UNDERSTANDING

OBJECTIVE

SyriaTel, a telecommunication's company, seeks to understand potential customer churn and develop strategies to enhance retention.This project analyzes customer behavior,satisfaction levels from service calls,account length, and incurred charges to identify predictive patterns of churn.These insights will help the company implement targeted incentives, improve service quality, and enhance customer satisfaction which will result in maximizing retention and profitability

KEY BUSINESS QUESTIONS

1.What is the average account length of churned compared to retained customers?

2.Do customers with multiple service complaints have a higher likelihood of churning?

3.How do charges and pricing plans affect customer churn?

METRICS OF SUCCESS

1.Churn Rate

2.Average Account Length

3.Service Call Frequency

DATA UNDERSTANDING

OVERVIEW

This dataset contains customer's information such as the account length,service usage,charges and customer service calls and churn status.These features will be used in analysis to help the company implement targeted incentives, improve service quality, and enhance customer satisfaction which will result in maximizing retention and profitability

Importing Libaries

In [44]:
```
# we will import the necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn import tree
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import GridSearchCV


from sklearn.tree import DecisionTreeClassifier,plot_tree
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_
```

In [45]:
```
#We will need to load dataset we are going to use
data = pd.read_csv("bigml_59c28831336c6604c800002a.csv")
#We then check the top 5 rows of the data to see the flow of the data
data.head()
```

Out[45]:

	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	...	to ca
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 [46]:
```
# checking the bottom 5 rows to ensure flow of data
data.tail()
```

Out[46]:

	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	...
3328	AZ	192	415	414-4276	no	yes	36	156.2	77	26.55	...
3329	WV	68	415	370-3271	no	no	0	231.1	57	39.29	...
3330	RI	28	510	328-8230	no	no	0	180.8	109	30.74	...
3331	CT	184	510	364-6381	yes	no	0	213.8	105	36.35	...
3332	TN	74	415	400-4344	no	yes	25	234.4	113	39.85	...

5 rows × 21 columns

Observation

There is uniformity in the dataset

In [47]:
```
#checking the shape
print(f"The dataset has records {data.shape[0]} and {data.shape[1]} variables
```

```
The dataset has records 3333 and 21 variables
```

In [48]:
```
#checking the data type
data.dtypes
```

Out[48]:
```
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 [49]:
```
#checking data type
data.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 [50]:
```
#changing some variables to category
data["area code"] = data["area code"].astype("category")
data["phone number"] = data["phone number"].astype("category")
data.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   category
 3   phone number            3333 non-null   category
 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), category(2), float64(8), int64(7), object(3)
memory usage: 637.1+ KB
```

In [51]:
```
#checking column names
data.columns
```

Out[51]:
```
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')
```

Data Preparation

Data Cleaning

In [52]:
```
#checking for missing values
data.isna().sum()
```

Out[52]:
```
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
```

In [53]:
```
#Checking for any duplicates
data.duplicated().sum()
```

Out[53]:
```
0
```

Observation

This dataset is clean.It has no missing values and no duplicates

Exploratotry Data Analysis

Univariate Analysis

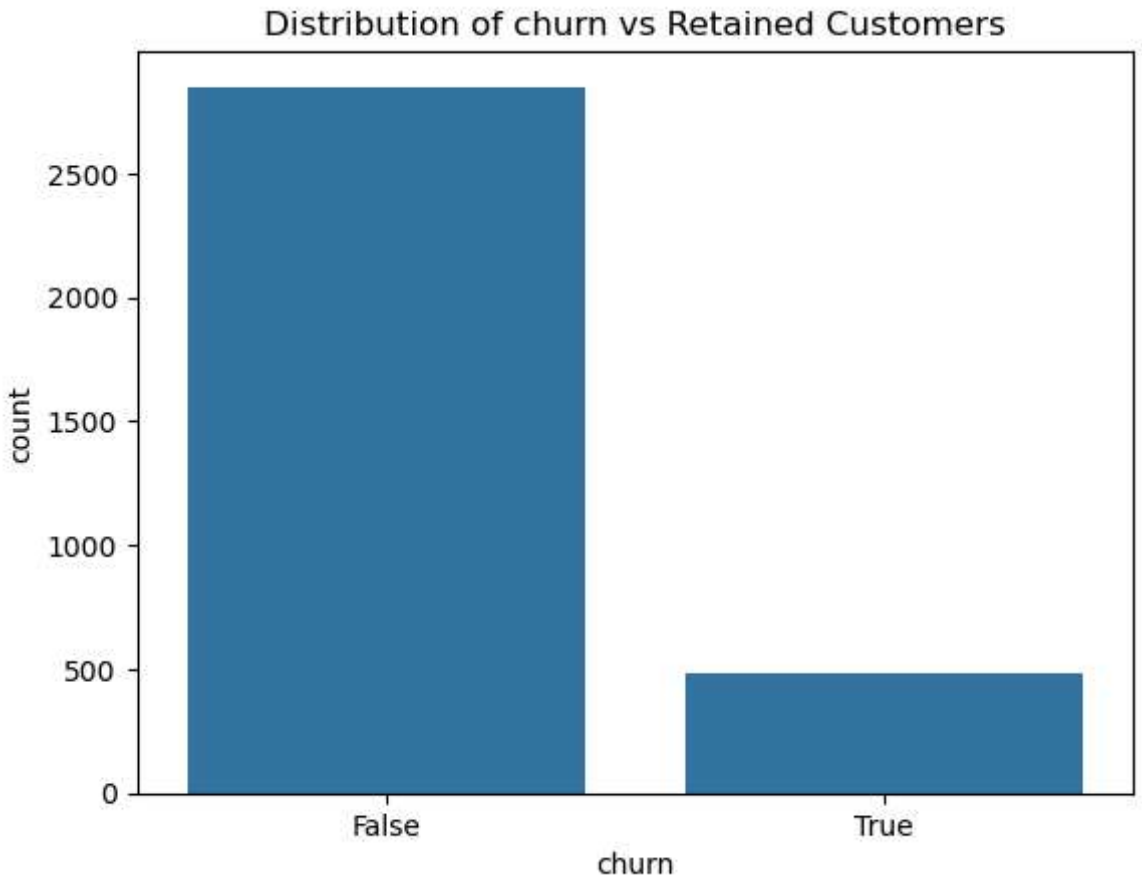
In [54]:
```
# checking for the summary of all numerical features
data.describe()
```

Out[54]:

	account length	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	total c
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000
mean	101.064806	8.099010	179.775098	100.435644	30.562307	200.980348	100.114
std	39.822106	13.688365	54.467389	20.069084	9.259435	50.713844	19.922
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
25%	74.000000	0.000000	143.700000	87.000000	24.430000	166.600000	87.000
50%	101.000000	0.000000	179.400000	101.000000	30.500000	201.400000	100.000
75%	127.000000	20.000000	216.400000	114.000000	36.790000	235.300000	114.000
max	243.000000	51.000000	350.800000	165.000000	59.640000	363.700000	170.000

In [55]:
```
# checking churn distribution
print(data['churn'].value_counts())

# Plot for churn distribution
sns.countplot(x='churn', data=data)
plt.title('Distribution of churn vs Retained Customers')
plt.show()
```

```
churn
False    2850
True      483
Name: count, dtype: int64
```



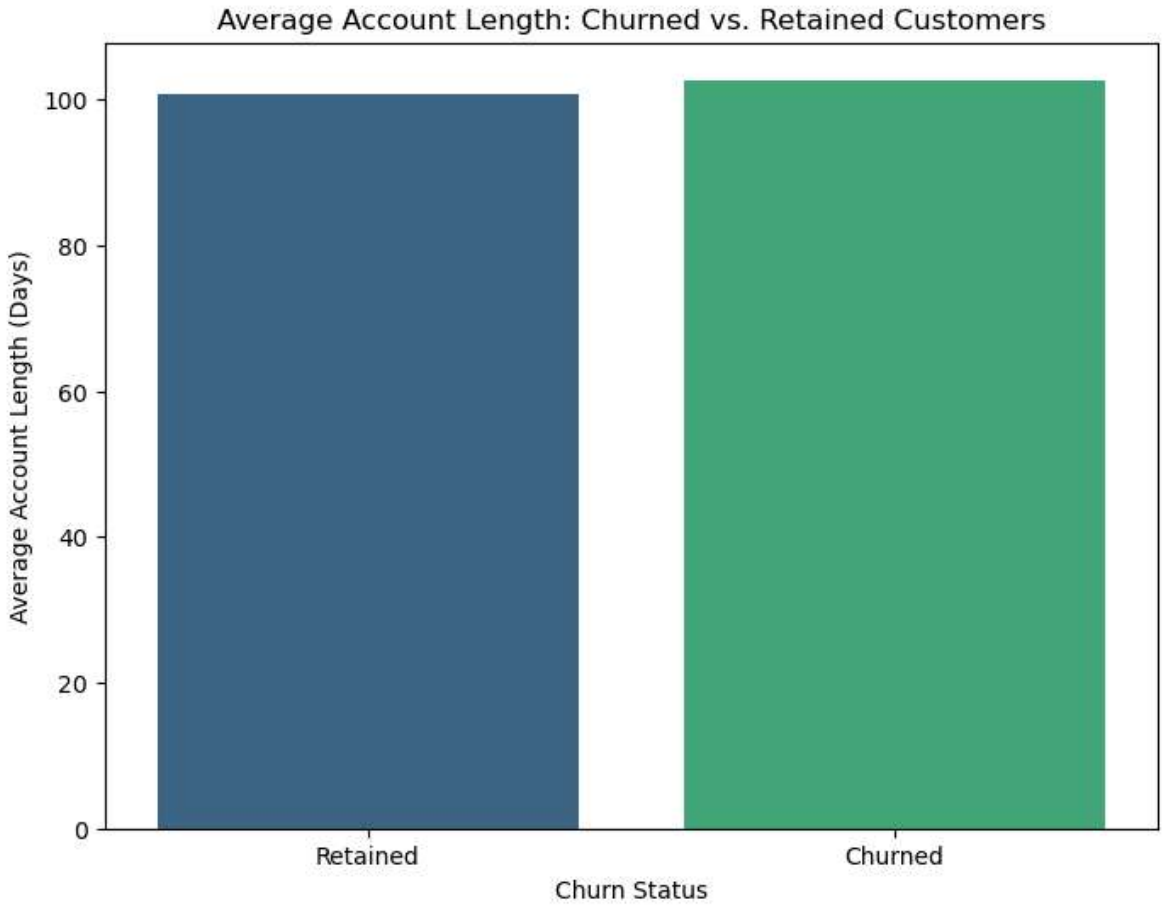
Observation

The above plot shows a significantly high level of False churn rates

In [56]:
```
# checking for a summary avearge of churn rate,Account length ,charges and se
# Churn rate
churn_rate = data['churn'].mean() * 100
print(f"\nChurn Rate: {churn_rate:.2f}%")

# Average account length
avg_account_length = data['account length'].mean()
print(f"Average Account Length: {avg_account_length:.2f} days")

# Average monthly charges
avg_day_charge = data['total day charge'].mean()
avg_eve_charge = data['total eve charge'].mean()
avg_night_charge = data['total night charge'].mean()
avg_intl_charge = data['total intl charge'].mean()
print(f"\nAverage Charges:")
print(f"Day: ${avg_day_charge:.2f}, Evening: ${avg_eve_charge:.2f}, Night: ${

# Service call frequency
avg_service_calls = data['customer service calls'].mean()
print(f"\nAverage Customer Service Calls: {avg_service_calls:.2f}")
```

```
Churn Rate: 14.49%
Average Account Length: 101.06 days

Average Charges:
Day: $30.56, Evening: $17.08, Night: $9.04, International: $2.76

Average Customer Service Calls: 1.56
```

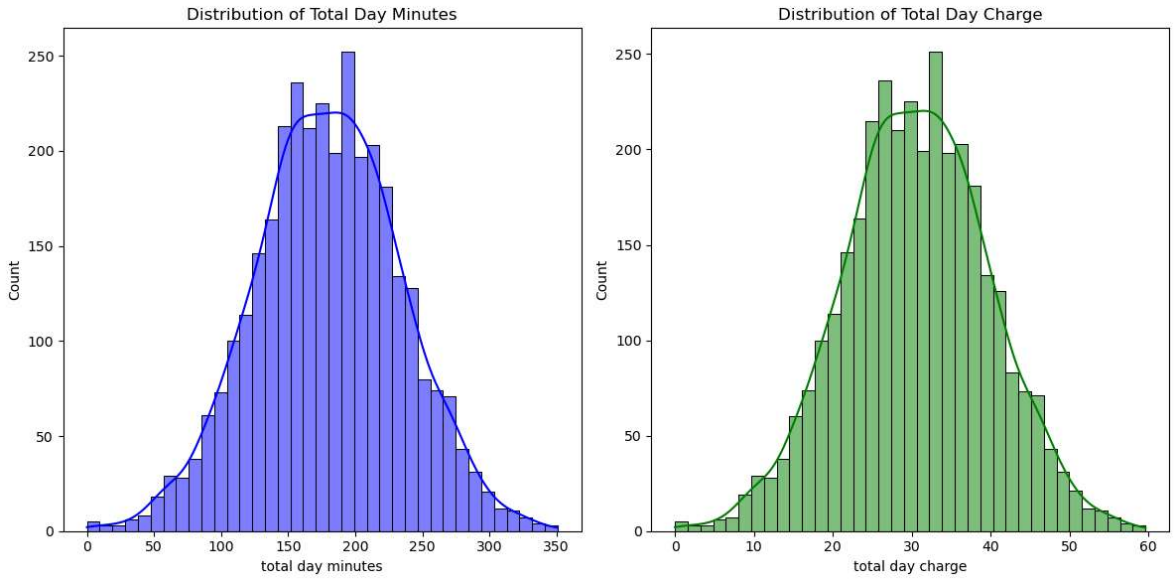
In [57]:
```
# Calculating average account length for churned and retained customers
avg_account_length_by_churn = data.groupby('churn')['account length'].mean()

# Converting mean to DataFrame for visualization
avg_account_length_df = avg_account_length_by_churn.reset_index()

# Ploting the average account length versus churn rate
plt.figure(figsize=(8, 6))
sns.barplot(data=avg_account_length_df, x='churn', y='account length', hue='c

plt.title('Average Account Length: Churned vs. Retained Customers')
plt.xlabel('Churn Status')
plt.ylabel('Average Account Length (Days)')
plt.xticks(ticks=[0, 1], labels=['Retained', 'Churned'])
plt.show()
```



Observation

On average, the number of False churn rates is slightly lower compared to churn rates

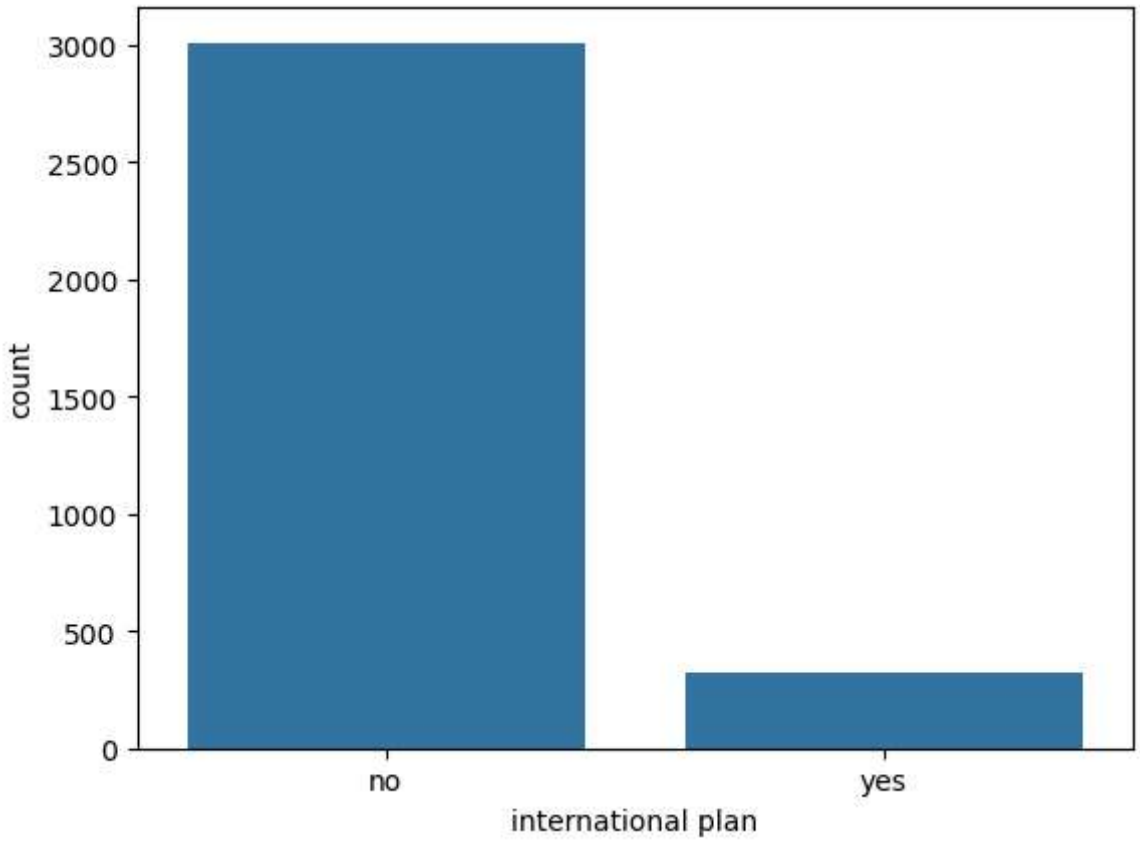
In [58]:
```
# Ploting distributions
plt.figure(figsize=(12, 6))

# Total day minutes
plt.subplot(1, 2, 1)
sns.histplot(data['total day minutes'], kde=True, color='blue')
plt.title('Distribution of Total Day Minutes')

# Total day charge
plt.subplot(1, 2, 2)
sns.histplot(data['total day charge'], kde=True, color='green')
plt.title('Distribution of Total Day Charge')

plt.tight_layout()
plt.show()
```



Observation

The total day minutes and total day charge show a normal distribution

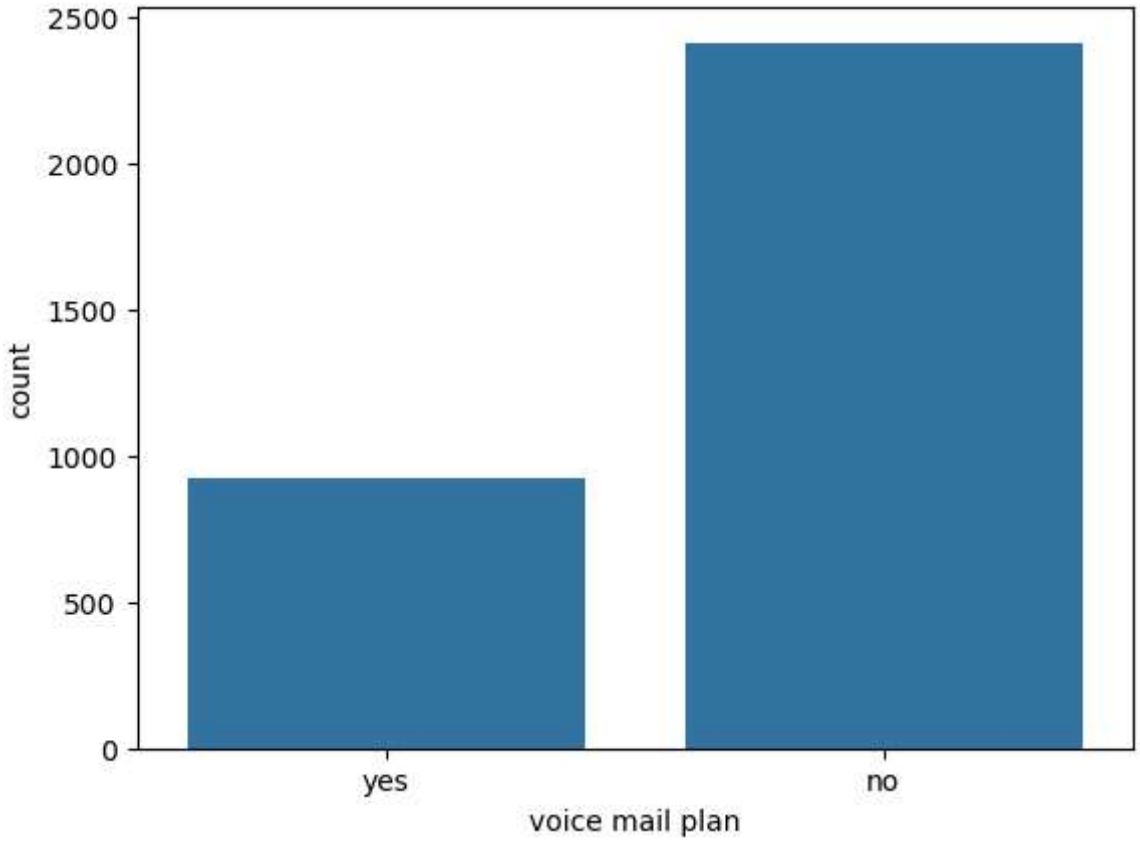
In [59]:
```
# bar plot of International plan count
sns.countplot(x='international plan', data=data);
```



Observation

Very few customers use the International plan

In [60]:
```
# bar plot of voice mail plan count
sns.countplot(x='voice mail plan', data=data);
```

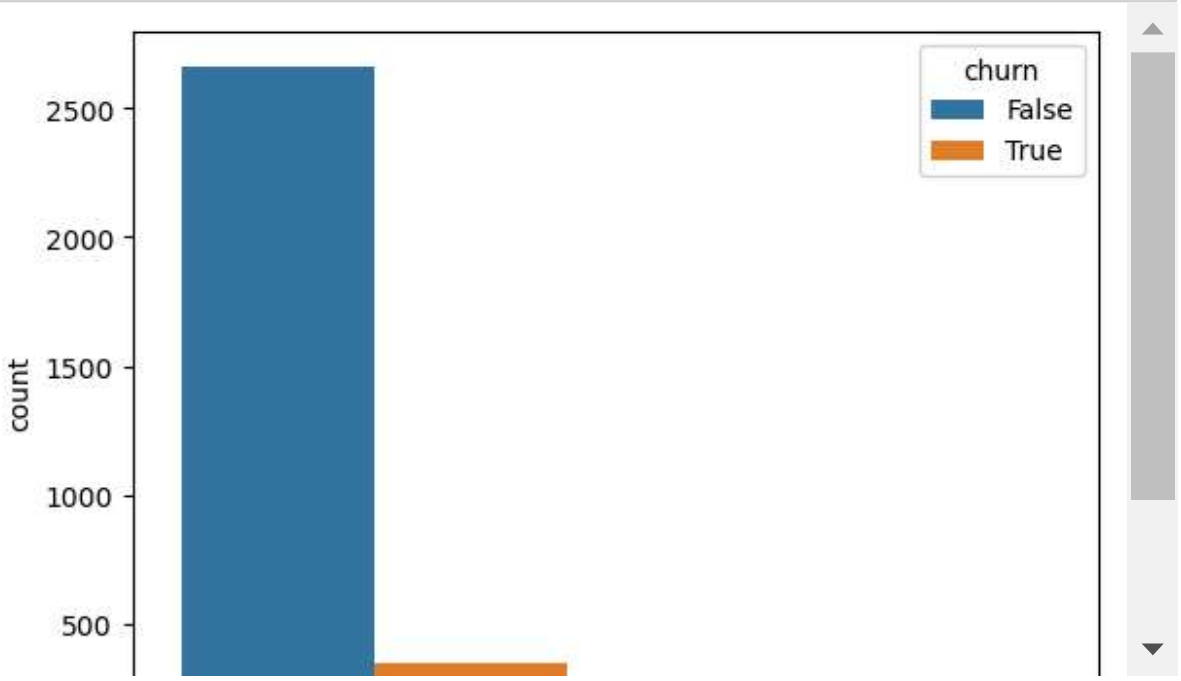


Observation

Very few customers use the Voice mail plan

Bivariate Analysis

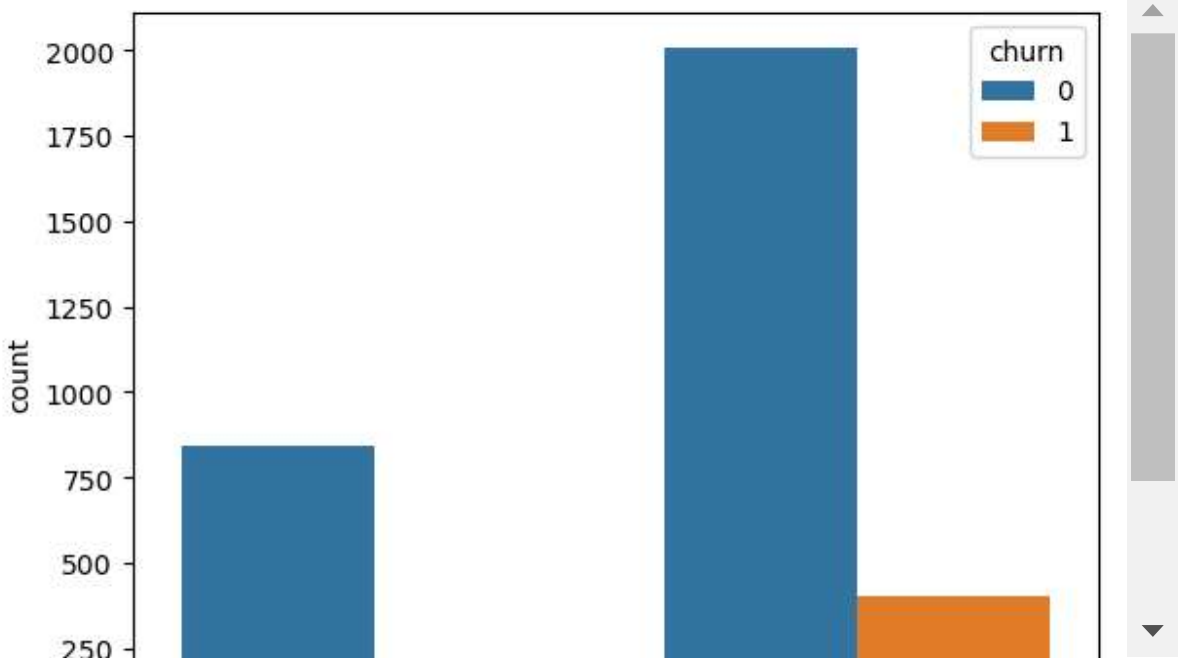
In [61]:
```
# ploting the relationship between International plan and Churn rate
sns.countplot(x='international plan', hue='churn', data=data);
```



Observation

Many customers with no International plan are also high churners. Most churners are likely local customers

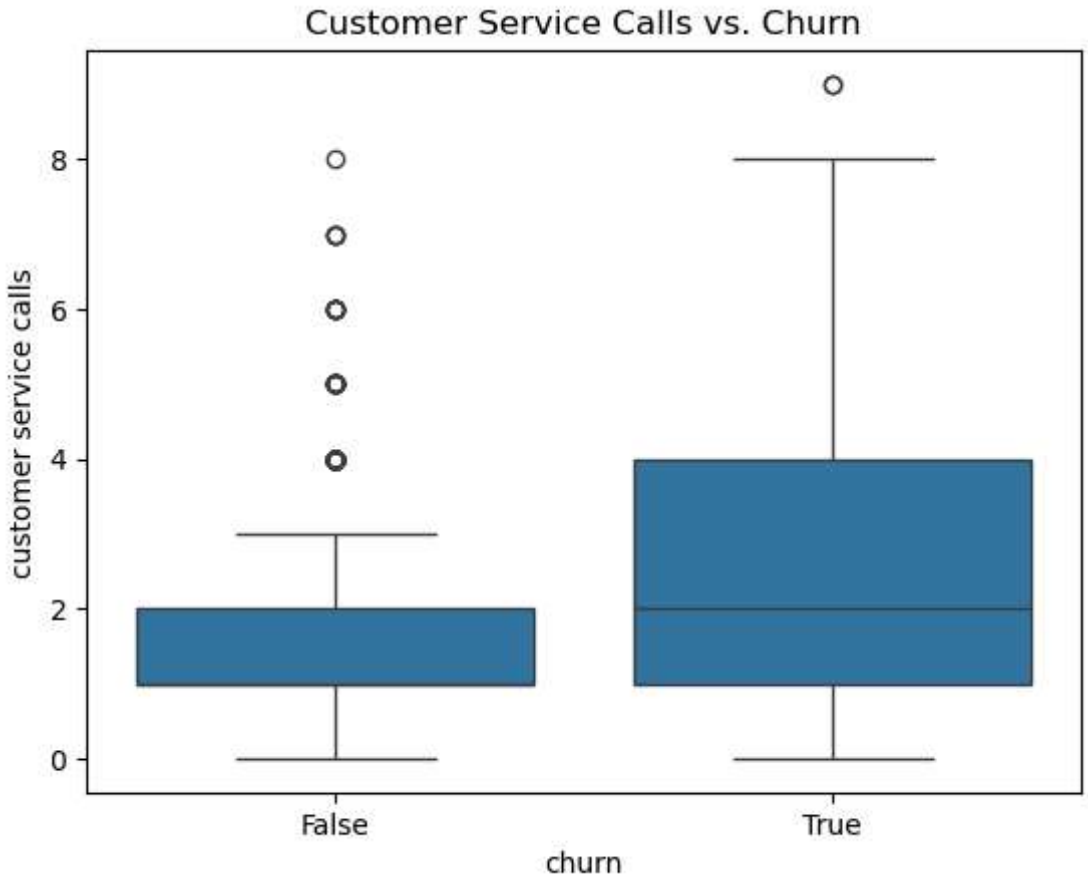
In [79]:
```
# ploting the relationship between Voice mail plan and Churn rate
sns.countplot(x='voice mail plan', hue='churn', data=data);
```



Observation

Most customers who have no voice mail plan are also high churners

In [63]:
```
# Boxplot of customer service calls vs. churn
sns.boxplot(x='churn', y='customer service calls', data=data)
plt.title("Customer Service Calls vs. Churn")
plt.show()
```

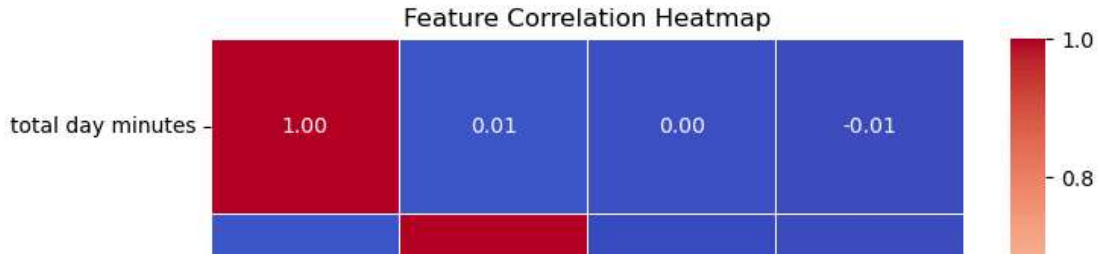


Observation

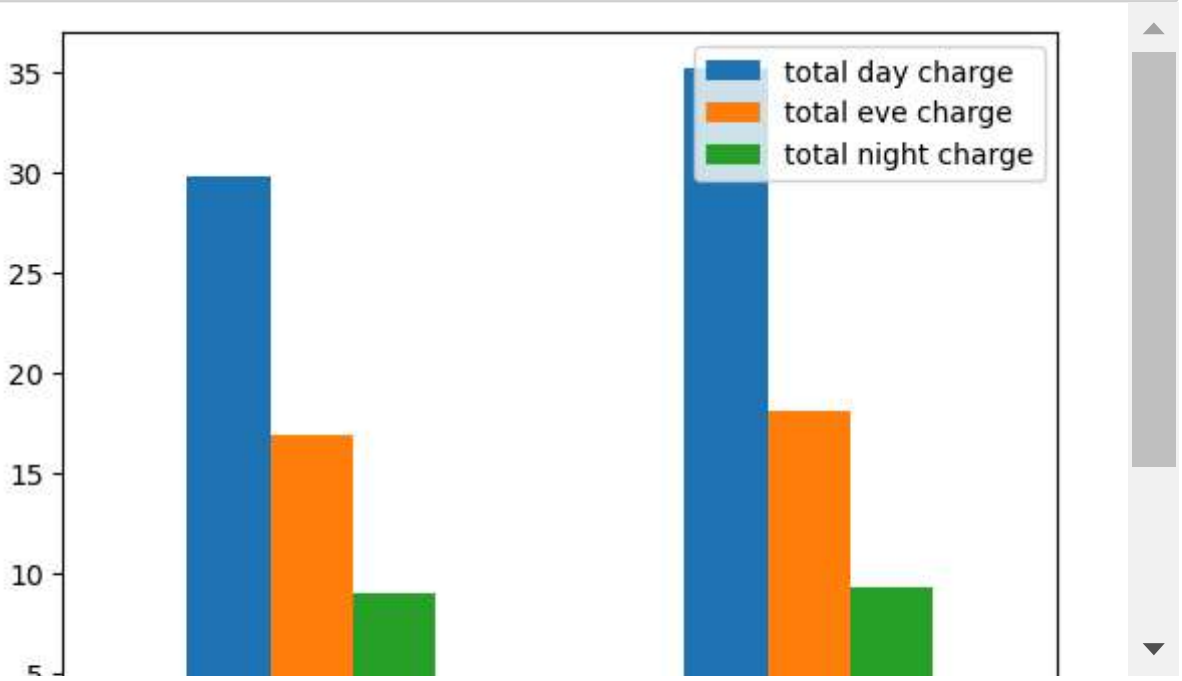
A higher number of churners have made multiple customer service calls

In [64]:
```
# Select relevant columns for correlation analysis
cols = ['total day minutes', 'total eve minutes', 'total night minutes', 'tot
corr_matrix = data[cols].corr()

# Display correlation matrix
print(corr_matrix)

# Visualizing correlation heatmap
plt.figure(figsize=(8,6))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=0
plt.title("Feature Correlation Heatmap")
plt.show()
```

```
                      total day minutes  total eve minutes  \
total day minutes             1.000000           0.007043
total eve minutes             0.007043           1.000000
total night minutes           0.004323          -0.012584
total intl minutes           -0.010155          -0.011035

                      total night minutes  total intl minutes
total day minutes                0.004323           -0.010155
total eve minutes               -0.012584           -0.011035
total night minutes              1.000000           -0.015207
total intl minutes              -0.015207            1.000000
```

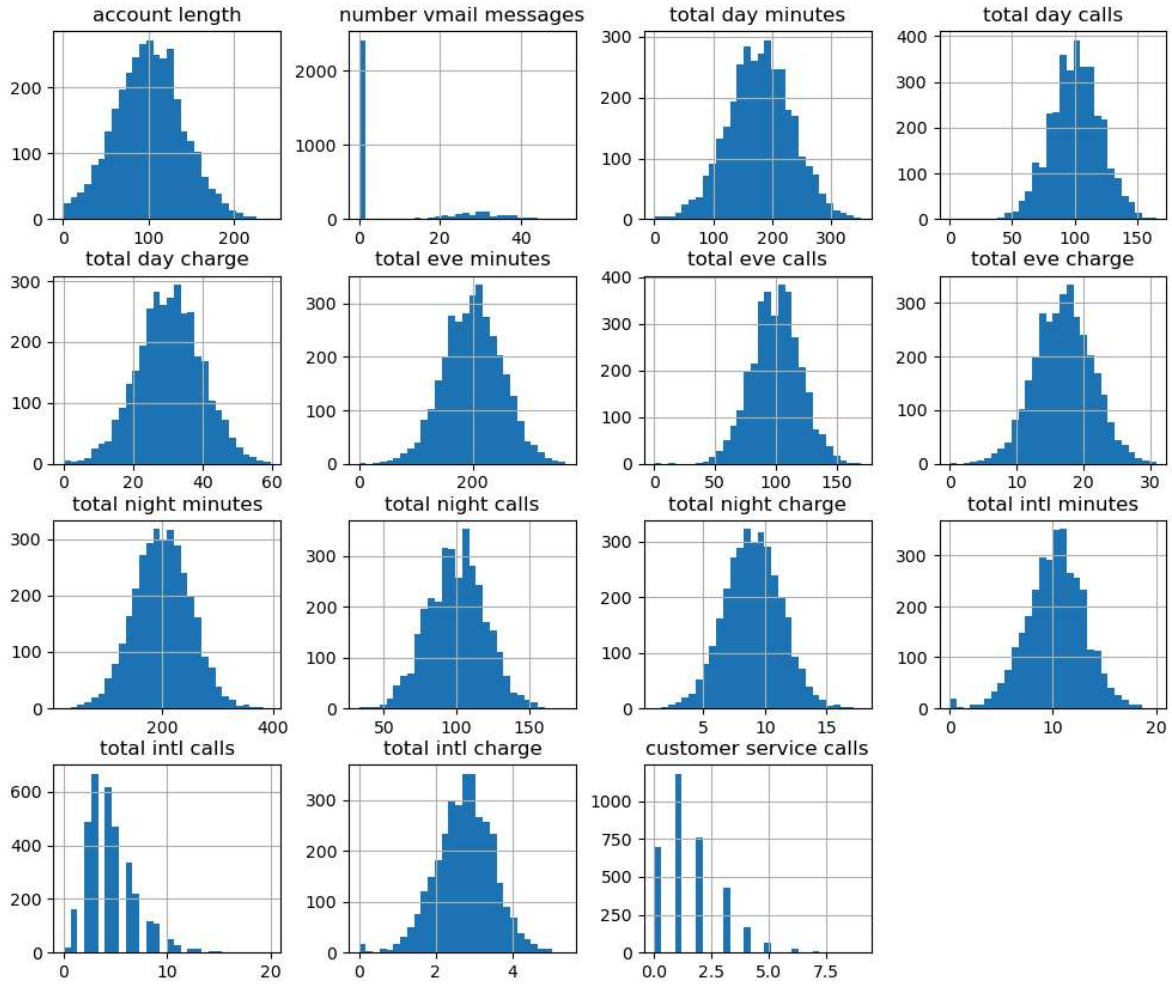


In [65]:
```
data.groupby('churn')[['total day charge', 'total eve charge', 'total night c
```



Preprocessing

In [66]:
```
# Distribution of numerical features
data.hist(figsize=(12, 10), bins=30)
plt.show()
```



In [67]:
```
# Dropping the phone column since i do not need it for modeling
data.drop(columns=['phone number'], inplace=True)
```

In [68]:
```
#changing churn variable into an integer considering it is the target
data['churn'] = data['churn'].astype(int)

# Separating 'churn' from the rest of the DataFrame to allow oneHotEncoding
churn = data['churn']
data_features = data.drop(columns=['churn'])
```

One Hot Encoding

In [69]:
```
#using one-hot encoding to the selected feature columns
df= pd.get_dummies(data_features, columns=['state', 'international plan', 'vo

# adding 'churn' back to the encoded DataFrame
df['churn'] = churn
```

In [70]:
```
# Checking the encoded DataFrame
df.head()
```

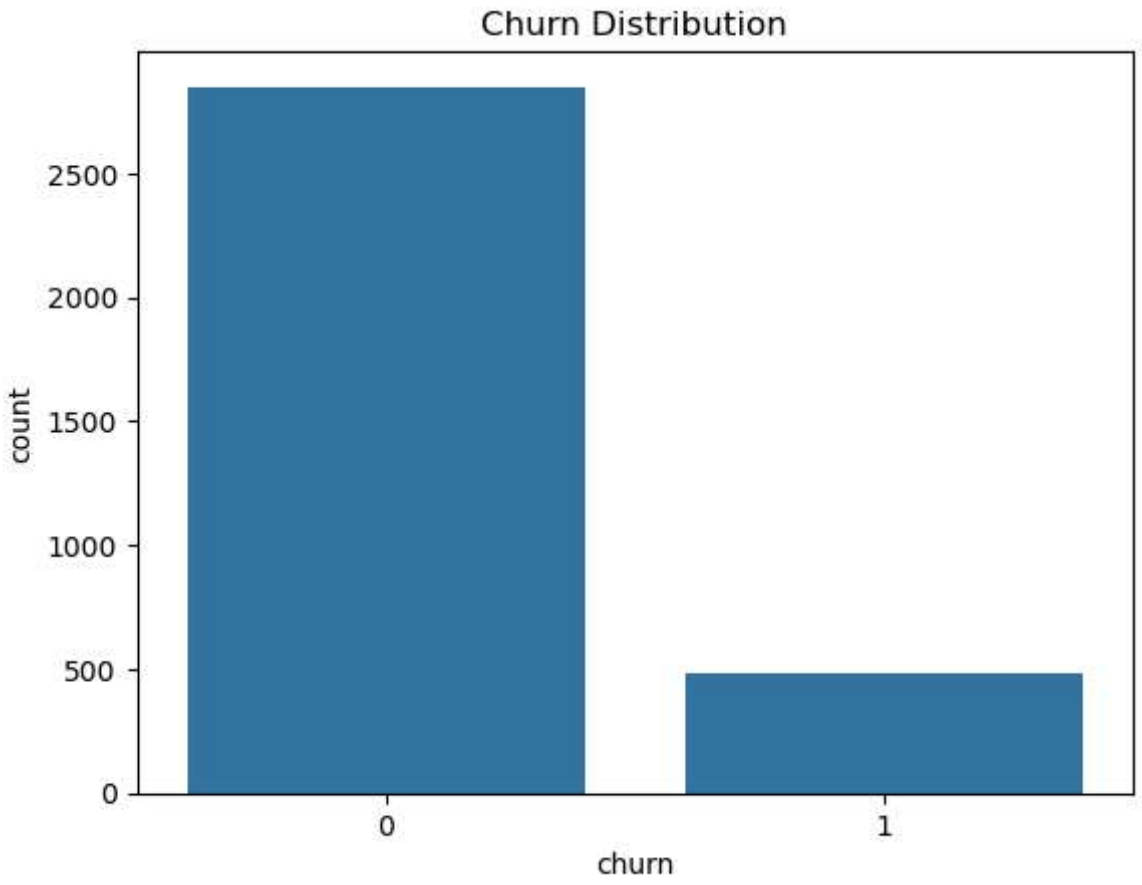
Out[70]:

	account length	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	...	state
0	128	25	265	110	45	197	99	16	244	91	...	
1	107	26	161	123	27	195	103	16	254	103	...	
2	137	0	243	114	41	121	110	10	162	104	...	
3	84	0	299	71	50	61	88	5	196	89	...	
4	75	0	166	113	28	148	122	12	186	121	...	

5 rows × 70 columns

In [71]:
```
# Display churn distribution
print(df['churn'].value_counts())

# Plot churn distribution
sns.countplot(x='churn', data=df)
plt.title('Churn Distribution')
plt.show()
```

```
churn
0    2850
1     483
Name: count, dtype: int64
```



Observations:

Highly imbalanced churn distribution.

MODELING

In [72]:
```
#split the data into target and predictor
X = df.drop("churn", axis=1)
y = df["churn"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rand

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

In [73]:
```
# Train a logistic regression model
model = LogisticRegression(max_iter=1000, random_state=42)
model.fit(X_train_scaled, y_train)

# Evaluate the model
y_pred = model.predict(X_test_scaled)
y_pred_proba = model.predict_proba(X_test_scaled)[:, 1]

print("Baseline Logistic Regression Accuracy:", accuracy_score(y_test, y_pred
print("ROC AUC Score:", roc_auc_score(y_test, y_pred_proba))
print(classification_report(y_test, y_pred))
```

```
Baseline Logistic Regression Accuracy: 0.86
ROC AUC Score: 0.8185408523798254
              precision    recall  f1-score   support

           0       0.88      0.97      0.92       857
           1       0.52      0.22      0.31       143

    accuracy                           0.86      1000
   macro avg       0.70      0.59      0.62      1000
weighted avg       0.83      0.86      0.84      1000
```

In [74]:
```
# Train a decision tree classifier
tree = DecisionTreeClassifier(random_state=42)
tree.fit(X_train, y_train)

# Evaluate the model
y_pred_tree = tree.predict(X_test)
y_pred_proba_tree = tree.predict_proba(X_test)[:, 1]

print("Decision Tree Accuracy:", accuracy_score(y_test, y_pred_tree))
print("ROC AUC Score:", roc_auc_score(y_test, y_pred_proba_tree))
print(classification_report(y_test, y_pred_tree))
```

```
Decision Tree Accuracy: 0.917
ROC AUC Score: 0.8408784914035788
              precision    recall  f1-score   support

           0       0.96      0.95      0.95       857
           1       0.70      0.73      0.72       143

    accuracy                           0.92      1000
   macro avg       0.83      0.84      0.83      1000
weighted avg       0.92      0.92      0.92      1000
```

In [75]:
```
# Hyperparameter tuning for the decision tree
param_grid = {
    'max_depth': [3, 5, 7, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

grid_search = GridSearchCV(DecisionTreeClassifier(random_state=42), param_gri
grid_search.fit(X_train, y_train)

# Best parameters and model
best_tree = grid_search.best_estimator_
y_pred_best_tree = best_tree.predict(X_test)
y_pred_proba_best_tree = best_tree.predict_proba(X_test)[:, 1]

print("Best Decision Tree Accuracy:", accuracy_score(y_test, y_pred_best_tree
print("Best ROC AUC Score:", roc_auc_score(y_test, y_pred_proba_best_tree))
print(classification_report(y_test, y_pred_best_tree))
```

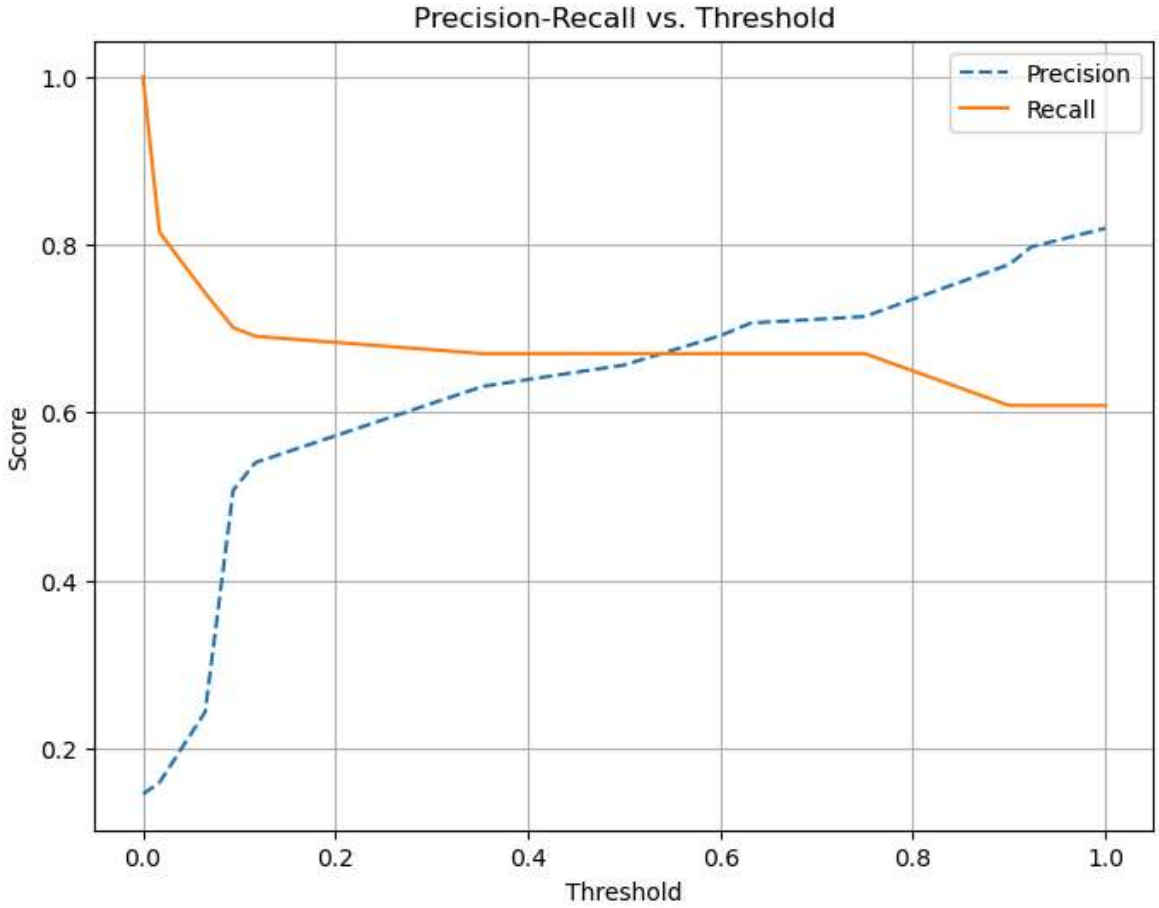
```
Best Decision Tree Accuracy: 0.939
Best ROC AUC Score: 0.9051292931106234
              precision    recall  f1-score   support

           0       0.94      0.99      0.97       857
           1       0.89      0.65      0.75       143

    accuracy                           0.94      1000
   macro avg       0.92      0.82      0.86      1000
weighted avg       0.94      0.94      0.93      1000
```

In [76]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, roc_auc_score
from sklearn.model_selection import train_test_split

# Splitting dataset (Replace X, y with your actual feature set and target var
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand

# Define Decision Tree with class weighting to improve recall
dt_model = DecisionTreeClassifier(
    class_weight={0: 1, 1: 3},  # Increase weight for class 1 (churners)
    max_depth=10,  # Prevent overfitting
    min_samples_split=5,
    min_samples_leaf=2,
    random_state=42
)

# Train the model
dt_model.fit(X_train, y_train)

# Predict probabilities
y_probs = dt_model.predict_proba(X_test)[:, 1]  # Get probabilities for class

# Adjust decision threshold for higher recall
threshold = 0.4  # Default is 0.5, lower favors recall
y_pred = (y_probs >= threshold).astype(int)

# Evaluate performance
print("Optimized Decision Tree Accuracy:", accuracy_score(y_test, y_pred))
print("ROC AUC Score:", roc_auc_score(y_test, y_probs))
print(classification_report(y_test, y_pred))
```

```
Optimized Decision Tree Accuracy: 0.9010494752623688
ROC AUC Score: 0.776171097847712
              precision    recall  f1-score   support

           0       0.94      0.94      0.94       570
           1       0.66      0.67      0.66        97

    accuracy                           0.90       667
   macro avg       0.80      0.81      0.80       667
weighted avg       0.90      0.90      0.90       667
```

In [77]:
```
from sklearn.metrics import precision_recall_curve, accuracy_score, roc_auc_s
import numpy as np
import matplotlib.pyplot as plt

# Get precision-recall curve values
precision, recall, thresholds = precision_recall_curve(y_test, y_probs)

# Find the best threshold (maximize F1-score)
f1_scores = 2 * (precision * recall) / (precision + recall + 1e-9)  # Avoid d
best_threshold = thresholds[np.argmax(f1_scores)]

print("Best Threshold:", best_threshold)

# Apply the new threshold
y_pred_opt = (y_probs >= best_threshold).astype(int)

# Evaluate the new model
print("Optimized Decision Tree Accuracy:", accuracy_score(y_test, y_pred_opt)
print("ROC AUC Score:", roc_auc_score(y_test, y_probs))
print(classification_report(y_test, y_pred_opt))

# Plot precision-recall curve
plt.figure(figsize=(8, 6))
plt.plot(thresholds, precision[:-1], label="Precision", linestyle='dashed')
plt.plot(thresholds, recall[:-1], label="Recall")
plt.xlabel("Threshold")
plt.ylabel("Score")
plt.title("Precision-Recall vs. Threshold")
plt.legend()
plt.grid()
plt.show()
```

```
Best Threshold: 1.0
Optimized Decision Tree Accuracy: 0.9235382308845578
ROC AUC Score: 0.776171097847712
              precision    recall  f1-score   support

           0       0.94      0.98      0.96       570
           1       0.82      0.61      0.70        97

    accuracy                           0.92       667
   macro avg       0.88      0.79      0.83       667
weighted avg       0.92      0.92      0.92       667
```


Precision-Recall vs. Threshold
Precision
Recall
Score
Threshold
1.0
0.8
0.6
0.4
0.2
0.0
0.2
0.4
0.6
0.8
1.0

In [78]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, roc_auc_sc

# Define Random Forest Model
rf_model = RandomForestClassifier(
    n_estimators=100,  # Number of trees
    max_depth=10,  # Prevent overfitting
    min_samples_split=5,
    min_samples_leaf=2,
    class_weight={0: 1, 1: 3},  # Increase weight for churners
    random_state=42,
    n_jobs=-1  # Use all CPU cores for faster training
)

# Train the model
rf_model.fit(X_train, y_train)

# Predict probabilities
y_probs_rf = rf_model.predict_proba(X_test)[:, 1]

# Find the best threshold using precision-recall tradeoff
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(y_test, y_probs_rf)
f1_scores = 2 * (precision * recall) / (precision + recall + 1e-9)
best_threshold_rf = thresholds[np.argmax(f1_scores)]
print("Best Threshold for Random Forest:", best_threshold_rf)

# Apply the best threshold
y_pred_rf = (y_probs_rf >= best_threshold_rf).astype(int)

# Evaluate performance
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
print("ROC AUC Score:", roc_auc_score(y_test, y_probs_rf))
print(classification_report(y_test, y_pred_rf))
```

```
Best Threshold for Random Forest: 0.4235850242586436
Random Forest Accuracy: 0.9280359820089955
ROC AUC Score: 0.8970157352143244
              precision    recall  f1-score   support

           0       0.95      0.96      0.96       570
           1       0.76      0.73      0.75        97

    accuracy                           0.93       667
   macro avg       0.86      0.85      0.85       667
weighted avg       0.93      0.93      0.93       667
```

KEY OBSERVATIONS OF THE VARIOUS MODELS

1. Baseline Logistic Regression Model

It has an accuracy of 86% which is generally good. An ROC AUC score of 82% which is quite good The recall for the positive class (instances of churn) is very low at 22% It also has an F1 score of 0.31 showing that the model struggles to identify customers who are likely to churn. This is because the dataset is highly imbalanced and therefore adressing the imbalance or using a better model will give us better results.

2. Decision Tree Classifier

Here the accuracy is at 92% which is improved The ROC AUC score is 84% which is better than 82% of the Logistic Regression Model The recall is at 72% which means it is able to notice more instances of churners . The F1 score is at 72% which is a balanced trade-off between Precision and Recall.This makes the model more effective . Having a recall from 22 % to 72 % is a huge adjustment. Can we better refine our model by tuning it ? We shall use Hyperparameter tuning to see how our model works.

2b . Decision Tree Classifier(with Hyperparameter tuning)

After tuning, the Accuracy score is at 94% is very good. The ROC AUC score is 90% which means it distinguishes between the classes very well. The Recall however has dropped to 65% .This is could be due to over pruning.The disadvantage is that it will miss more instances of Churning The F1 on the hand has improved from 72% to 75% which means the perfomance is more balanced . Having a sudden drop in recall means the model is still not efficient and therefore needs fixing.

3. Optimized Decision Tree Classifier(with best threshhold)

The accuracy is 92 % slightly lower but still good. The ROC AUC score is 77% is also lower but still good. The Recall has a slight drop to 61 % The precision has increased to 82 % .This is good because it means the instances of False positives will decrease . Making this a better model.

4. Random Forest Classifier

It has an accuracy score of 93% which is better than the Decission Tree Classifier The ROC AUC score is at 90% which is very good. Recall is 73% which is a good improvement .It will be able to identify more churners Precision is at 76% which is a slight drop but good for handling the False Postives. the F1 score is at 75% which is an increase making the model more balanced at this point.

Conclusion

The analysis of various models reveals that while Logistic Regression struggles with imbalanced data and low recall, tree-based models like Decision Tree and Random Forest perform significantly better. Hyperparameter tuning improved the Decision Tree's performance, but it still faced challenges with recall. The Random Forest Classifier emerged as the most balanced model, with high accuracy, ROC AUC, and recall, making it the best choice for identifying churners effectively

Recommendation

Adopt the Random Forest Model:

It provides the best balance of accuracy (93%), ROC AUC (90%), and recall (73%), making it the most reliable model for churn prediction.