PHASE 3 PROJECT

Overview

This project aims to develop a binary classification model to predict customer churn for SyriaTel, a telecommunications company. By identifying customers likely to stop using the company's services soon, the business can implement targeted retention strategies to minimize revenue loss. The analysis will explore customer behavior, usage patterns, and demographics to uncover actionable insights.

Leveraging machine learning techniques, the model will identify key features that contribute to churn, enabling SyriaTel to address root causes and enhance customer satisfaction. This proactive approach not only mitigates financial losses but also strengthens customer loyalty and competitive positioning.

1. Problem Definition

- **Objective**: Develop a machine learning model to predict customer churn (binary classification: `Yes / No`), helping SyriaTel proactively address and mitigate churn risks.
- **Outcome**: Enable SyriaTel to identify patterns in customer behavior and usage, equipping them with actionable insights to improve customer retention and reduce revenue loss.
- **Metric for Success**: Select the most relevant evaluation metric, such as **Accuracy** (overall prediction correctness), **Precision** (proportion of true churn predictions), **Recall** (ability to detect all churn cases), **F1-Score** (balance between Precision and Recall), or **AUC-ROC** (model performance across thresholds), aligning with SyriaTel’s business priorities.

2. Data Collection

- **Source**: The dataset for this project is sourced from Kaggle. It contains customer data, including demographics, account information, and usage patterns, relevant to predicting churn.
- **File Format**: CSV (Comma-Separated Values).
- **Link to Dataset**: *(https://www.kaggle.com/datasets/becksddf/churn-in-telecoms-dataset)*.

Importing the Required Libraries

In [1]:

```
#importing necessary libraries

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

```
import matplotlib.pyplot as plt
import plotly.express as px
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_a
```

3. Data Preparation

3.1 Creating DataFrame & Data Understanding

In [2]:

```
df_churn = pd.read_csv("churn_telecom.csv")
df_churn.head()
```

Out[2]:

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

5 rows × 21 columns

3.1.1 Checking for thr shape of the data

- To know the number of rows and columns in the dataframe

In [3]:

```
# Check shape of dataframe - 3333 rows and 21 columns
df_churn.shape
```

Out[3]: (3333, 21)

3.1.3 Checking the description

In [4]:

```
df_churn.describe()
```

Out[4]:

	account length	area code	number vmail	total day minutes	total day calls	total c char

		messages				
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000(
mean	101.064806	437.182418	8.099010	179.775098	100.435644	30.562:
std	39.822106	42.371290	13.688365	54.467389	20.069084	9.2594
min	1.000000	408.000000	0.000000	0.000000	0.000000	0.000(
25%	74.000000	408.000000	0.000000	143.700000	87.000000	24.430(
50%	101.000000	415.000000	0.000000	179.400000	101.000000	30.500(
75%	127.000000	510.000000	20.000000	216.400000	114.000000	36.790(
max	243.000000	510.000000	51.000000	350.800000	165.000000	59.640(

3.1.4 Checking the info

- To know the data types in the columns.

In [5]:
```
df_churn.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
```

3.2 Data Cleaning

This section focuses on preparing the dataset for exploratory data analysis (EDA) and modeling. The steps include:

- Identifying and removing duplicate rows.

- Handling missing values to ensure data consistency.
- Eliminating irrelevant columns that do not contribute meaningfully to the analysis.

In [6]:
```
# Checking for duplicated rows, no duplicated rows to deal with.
df_churn.duplicated().sum()
```

Out[6]:
```
0
```

> No duplicates in the data

In [7]:
```
# Checking for missing values, no missing values.

df_churn.isnull().sum()
```

Out[7]:
```
state                     0
account length            0
area code                 0
phone number              0
international plan        0
voice mail plan           0
number vmail messages     0
total day minutes         0
total day calls           0
total day charge          0
total eve minutes         0
total eve calls           0
total eve charge          0
total night minutes       0
total night calls         0
total night charge        0
total intl minutes        0
total intl calls          0
total intl charge         0
customer service calls    0
churn                     0
dtype: int64
```

> No missing data

3.2.1 Handling irrelevant columns

- Dropping `phone number`, `state`, and `area code` as they are of little significance to the models.

In [8]:
```
# Removing customer number feature as it is contact information on the cli

df_churn.drop(['phone number', 'state', 'area code'],axis=1,inplace=True)
df_churn.head()
```

Out[8]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	tota eve call

0	128	no	yes	25	265.1	110	45.07	197.4	9
1	107	no	yes	26	161.6	123	27.47	195.5	10
2	137	no	no	0	243.4	114	41.38	121.2	11
3	84	yes	no	0	299.4	71	50.90	61.9	8
4	75	yes	no	0	166.7	113	28.34	148.3	12

> Checking the shape now after dropping irrevlevant columns.

In [9]:
```
df_churn.shape
```

Out[9]:
```
(3333, 18)
```

3.2.2 Detecting categorical columns

In [10]:
```
categoricals = df_churn.select_dtypes("object")

for col in categoricals:
    print(df_churn[col].value_counts(), "\n")
```

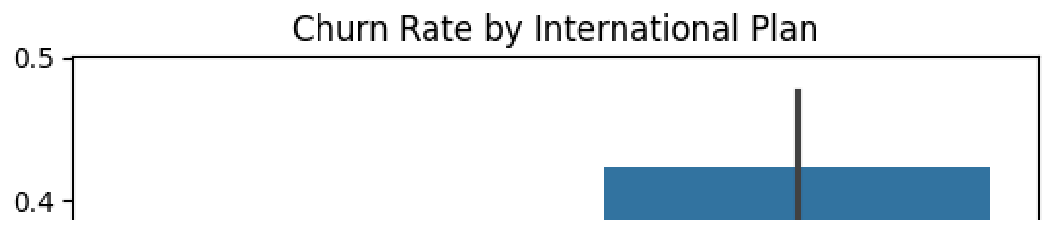
```
international plan
no     3010
yes     323
Name: count, dtype: int64

voice mail plan
no     2411
yes     922
Name: count, dtype: int64
```

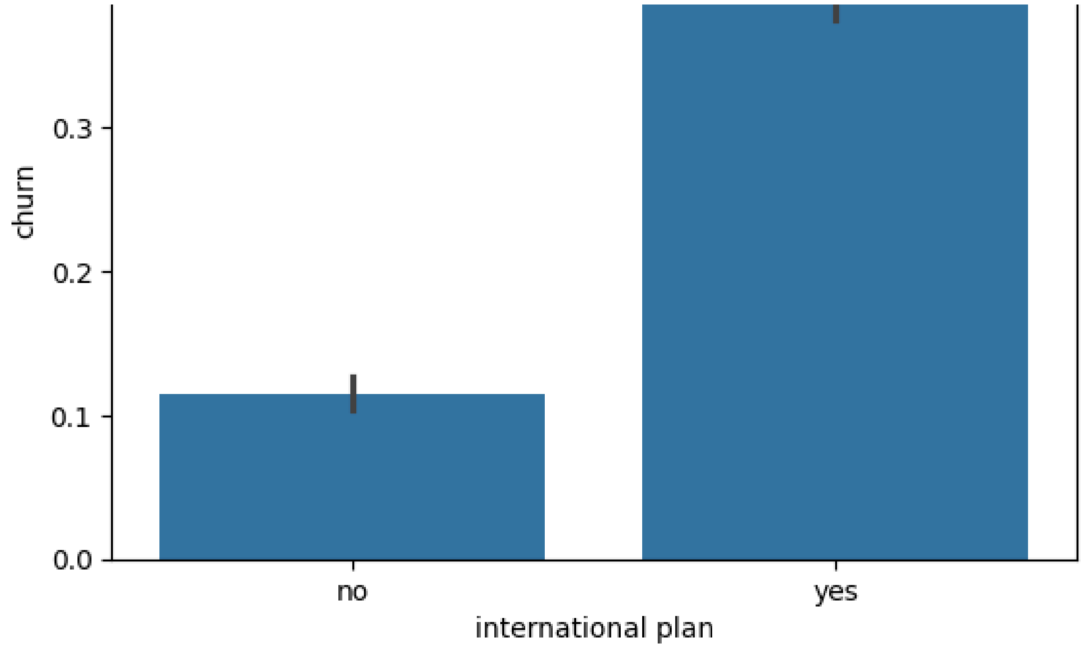
3.2.3 Analyzing Churn Rate by International Plan

- Here, I calculate the average churn rate for customers based on whether they have an international plan and visualizes the results using a bar plot.

In [11]:
```
print(df_churn.groupby('international plan')['churn'].mean())
sns.barplot(x='international plan', y='churn', data=df_churn)
plt.title('Churn Rate by International Plan')
plt.show()
```

```
international plan
no     0.114950
yes    0.424149
Name: churn, dtype: float64
```





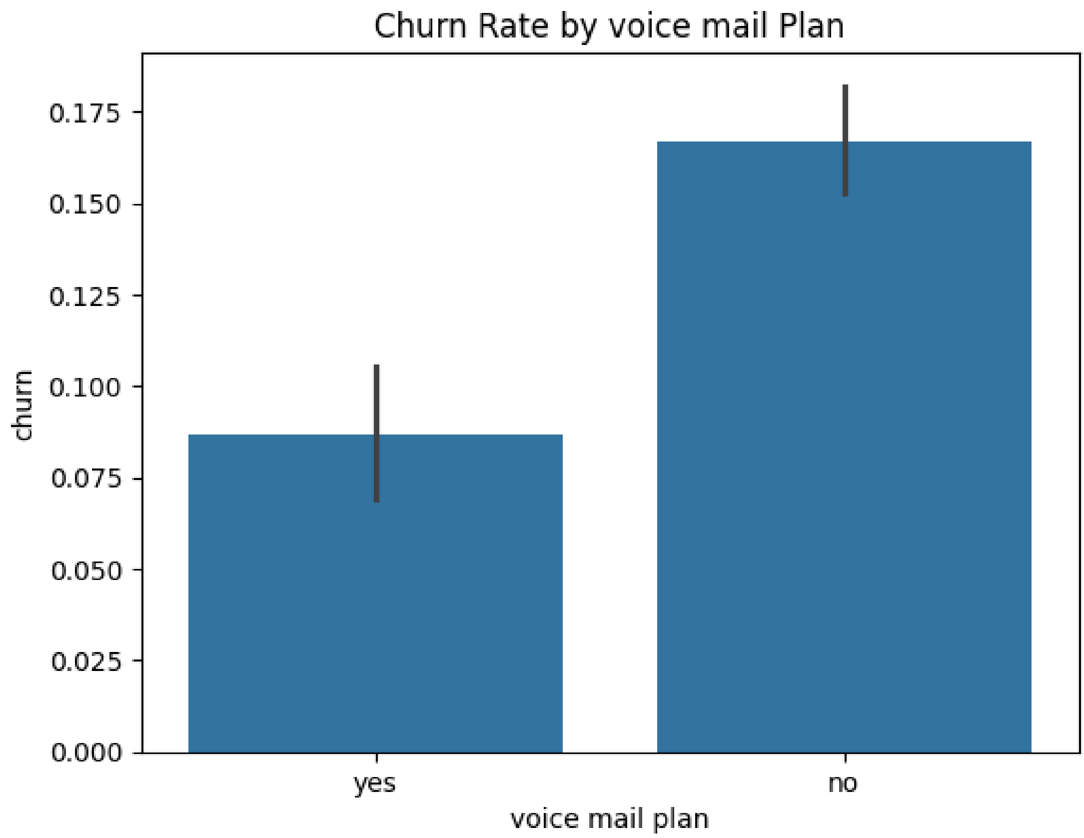
3.2.4 Analyzing Churn Rate by Voice Mail Plan

- Here i compute the average churn rate for customers with and without a voice mail plan and visualizes the comparison using a bar plot.

In [12]:

```
print(df_churn.groupby('voice mail plan')['churn'].mean())
sns.barplot(x='voice mail plan', y='churn', data=df_churn)
plt.title('Churn Rate by voice mail Plan')
plt.show()
```

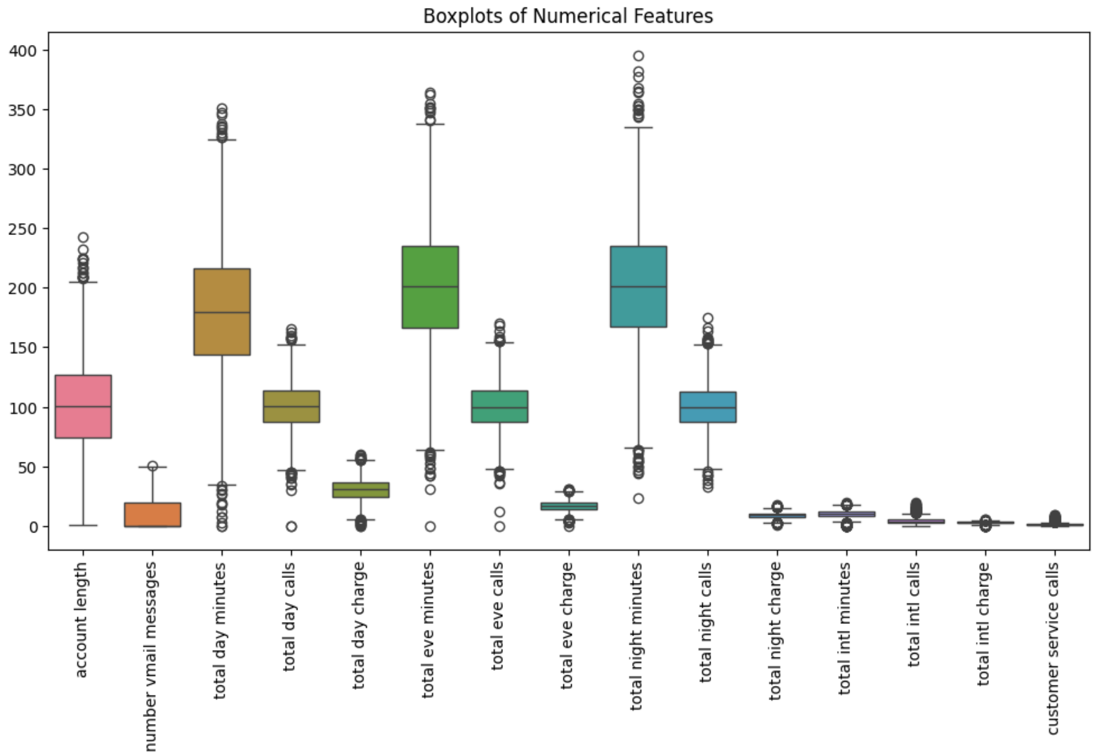
```
voice mail plan
no     0.167151
yes    0.086768
Name: churn, dtype: float64
```



3.2.4 Detecting Outliers and plotting them

In [13]:

```
#Detecting outliers using boxplots
plt.figure(figsize=(12, 6))
sns.boxplot(data=df_churn.select_dtypes(include='number'))
plt.title("Boxplots of Numerical Features")
plt.xticks (rotation= 90)
plt.show()
```



3.2.5 Removing Outliers Using the IQR Method

- Detecting and removing outliers in specified numeric columns of the dataset based on the Interquartile Range (IQR) method, ensuring the data is cleaned for further analysis.

In [14]:

```
def remove_outliers(df_churn, columns):
    for col in columns:
        # Calculate Q1 (25th percentile) and Q3 (75th percentile)
        Q1 = df_churn[col].quantile(0.25)
        Q3 = df_churn[col].quantile(0.75)
        IQR = Q3 - Q1  # Interquartile Range

        # Define lower and upper bounds for detecting outliers
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        # Filter out outliers
        df_churn = df_churn[(df_churn[col] >= lower_bound) & (df_churn[col

    return df_churn

# List of columns to check for outliers (excluding 'Churn')
feature_columns = [col for col in df_churn.columns if col != 'Churn' and d
```

```
# Apply the function to remove outliers
df_churn = remove_outliers(df_churn, feature_columns)
df_churn
```

Out[14]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes
0	128	no	yes	25	265.1	110	45.07	197.4
1	107	no	yes	26	161.6	123	27.47	195.5
2	137	no	no	0	243.4	114	41.38	121.2
4	75	yes	no	0	166.7	113	28.34	148.3
5	118	yes	no	0	223.4	98	37.98	220.6
...	...	...	...	...	...	...	...	...
3328	192	no	yes	36	156.2	77	26.55	215.5
3329	68	no	no	0	231.1	57	39.29	153.4
3330	28	no	no	0	180.8	109	30.74	288.8
3331	184	yes	no	0	213.8	105	36.35	159.6
3332	74	no	yes	25	234.4	113	39.85	265.9

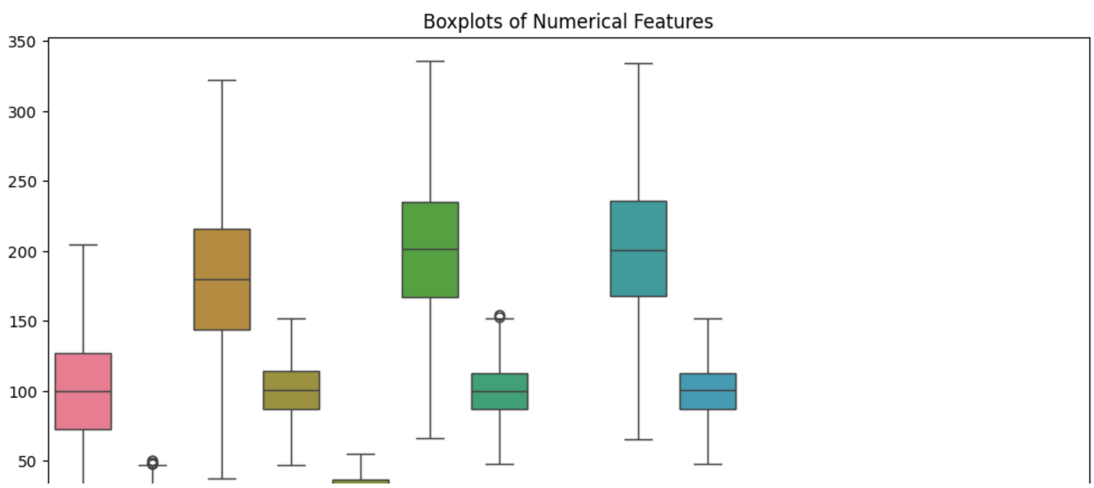
2797 rows × 18 columns

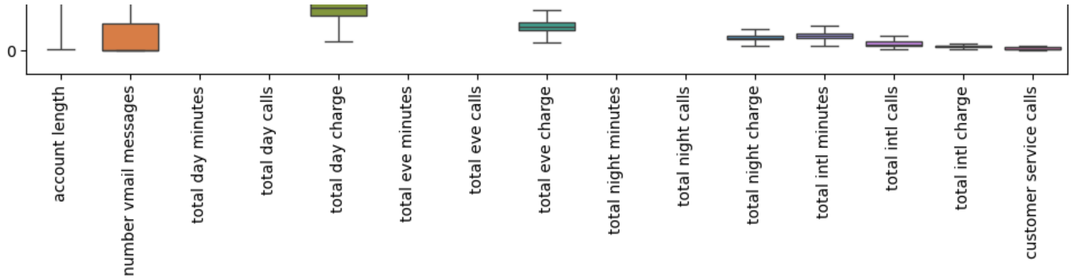
3.2.5 Visualizing Outlier Removal with Boxplots

- Visually assessing whether outliers have been successfully eliminated after data cleaning.

In [15]:

```
#Detecting outliers using boxplots
plt.figure(figsize=(12, 6))
sns.boxplot(data=df_churn.select_dtypes(include='number'))
plt.title("Boxplots of Numerical Features")
plt.xticks (rotation= 90)
plt.show()
```





> Most of the outliers have been removed as visuallized above.

3.2.6 Encoding Categorical Variables

- Converting the 'international plan' and 'voice mail plan' columns into numeric values by mapping 'yes' to 1 and 'no' to 0, making them suitable for machine learning algorithms and statistical analysis.

In [16]:

```
# Map 'yes' to 1 and 'no' to 0 in the 'international plan' and 'voice mail
df_churn['international plan'] = df_churn['international plan'].map({'yes'
df_churn['voice mail plan'] = df_churn['voice mail plan'].map({'yes': 1, '

# Display the first few rows of the updated DataFrame
df_churn.head()
```

Out[16]:

	account length	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	total day charge	total eve minutes	tota evi call
0	128	0	1	25	265.1	110	45.07	197.4	9
1	107	0	1	26	161.6	123	27.47	195.5	10
2	137	0	0	0	243.4	114	41.38	121.2	11
4	75	1	0	0	166.7	113	28.34	148.3	12
5	118	1	0	0	223.4	98	37.98	220.6	10

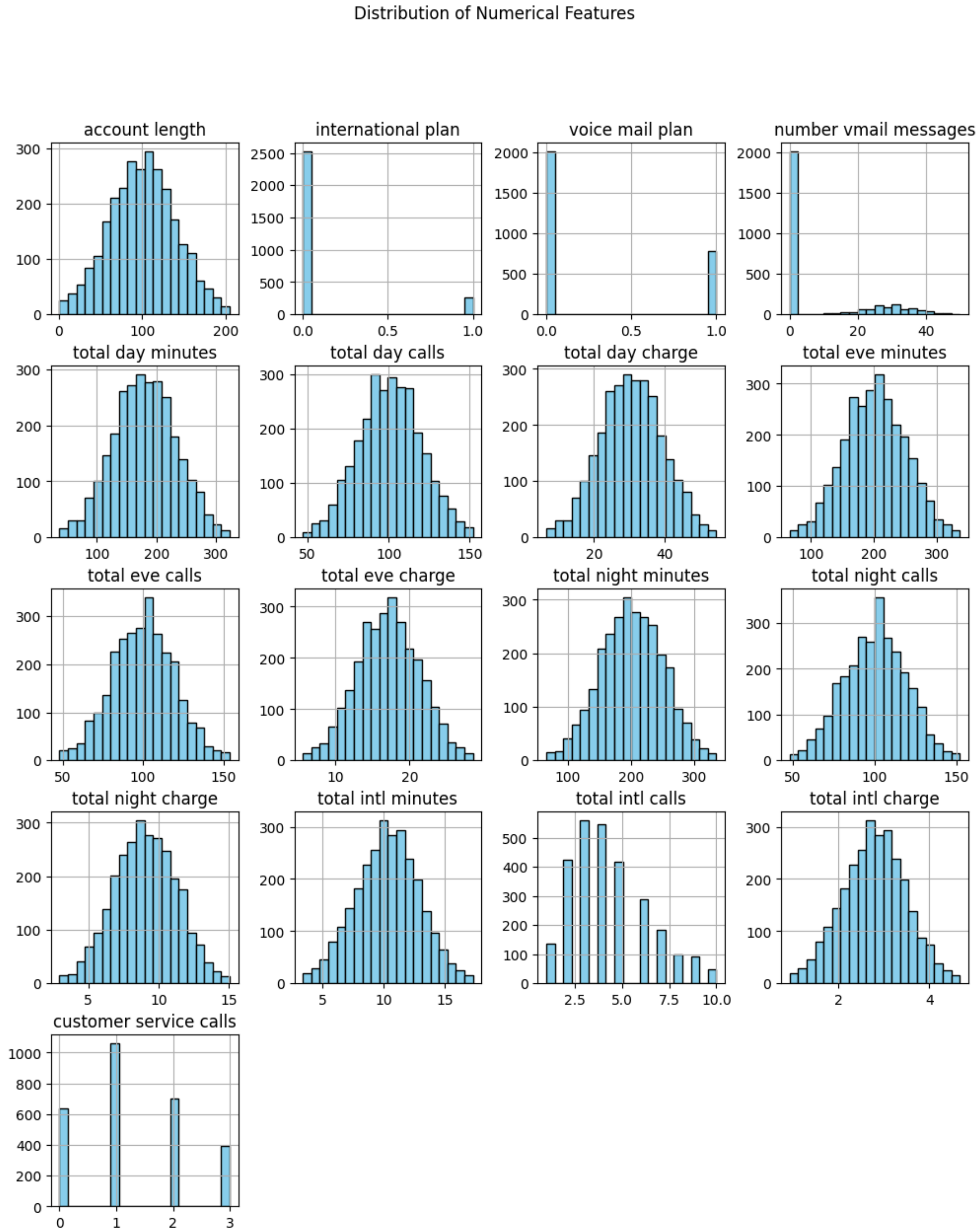
3.3 Feature Engineering

3.3.1 Visualizing Numerical Feature Distributions

Generateing histograms for all numerical features in the dataset to observe their distributions and identify patterns or anomalies.

In [17]:

```
# Visualizing distributions using histograms
df_churn.hist(bins=20, figsize=(12, 14), color='skyblue', edgecolor='black
plt.suptitle("Distribution of Numerical Features")
plt.show()
```

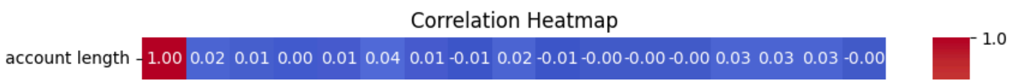


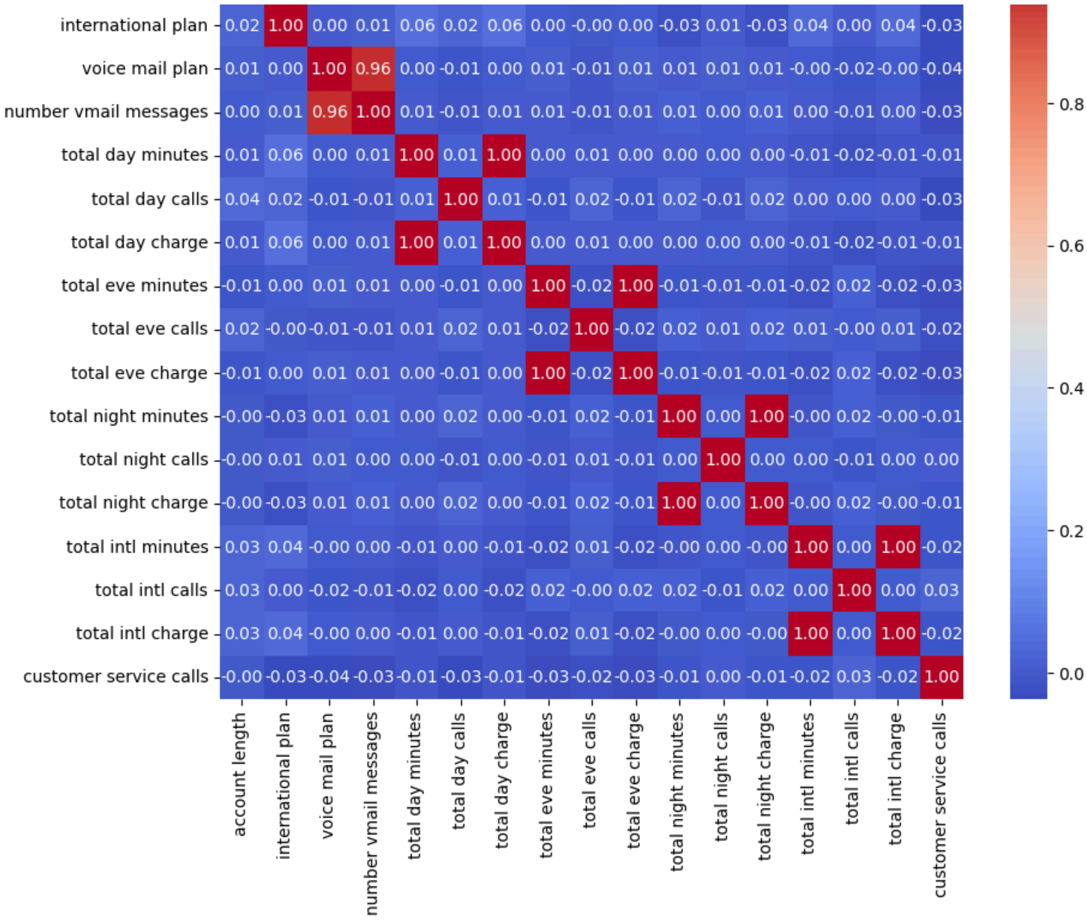
3.3.3 Correlation Heatmap for Numeric Features

Computing and visualizing the correlation matrix for numeric columns in the dataset using a heatmap, highlighting relationships between variables with color gradients and correlation coefficients.

In [18]:

```
#Correlation heatmap (only numeric columns)
numeric_columns = df_churn.select_dtypes(include=['number']).columns

plt.figure(figsize=(10, 8))
sns.heatmap(df_churn[numeric_columns].corr(), annot=True, cmap='coolwarm',
plt.title("Correlation Heatmap")
plt.show()
```





3.3.4 Removing Highly Correlated Features

Identifying and droping columns with a correlation greater than 0.9 to eliminate multicollinearity, ensuring that redundant features are removed for more efficient modeling.

In [19]:

```
import numpy as np
import pandas as pd

# Calculate the correlation matrix
corr_matrix = df_churn.corr().abs()

# Identify upper triangle of the correlation matrix
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(

# Find columns with correlation > 0.9
to_drop = [column for column in upper.columns if any(upper[column] > 0.9)]

# Drop the columns
df_churn = df_churn.drop(columns=to_drop)

print("Dropped columns:", to_drop)
```

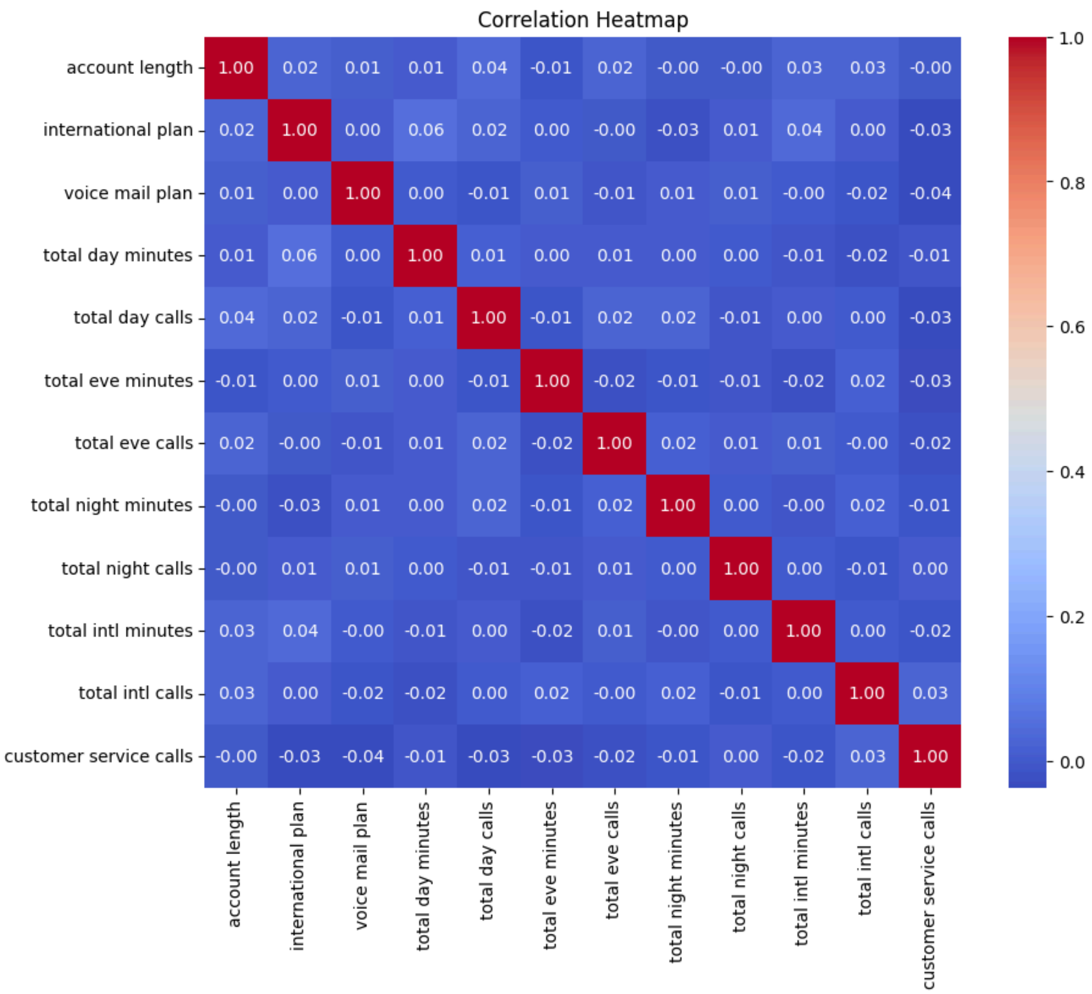
```
Dropped columns: ['number vmail messages', 'total day charge', 'total eve ch
arge', 'total night charge', 'total intl charge']
```

> Dropped columns: `number vmail messages`, `total day charge`, `total eve charge`, `total night charge`, and `total intl charge`.

3.3.5 Correlation Heatmap After Removing Highly Correlated Features

Generating a correlation heatmap for the remaining numeric features after removing those with high correlation (above 0.9), providing a clearer view of the relationships among the remaining variables.

In [20]:
```
#Correlation heatmap after dropping highky correlated features.
numeric_columns = df_churn.select_dtypes(include=['number']).columns

plt.figure(figsize=(10, 8))
sns.heatmap(df_churn[numeric_columns].corr(), annot=True, cmap='coolwarm',
plt.title("Correlation Heatmap")
plt.show()
```



3.3.6 Transforming Churn Values into 0s and 1s

Converting the 'churn' column from boolean values (True/False) into binary integers (1/0), making it suitable for analysis or modeling.

In [21]:
```
# transforming churn values into 0s and 1s

df_churn['churn'].value_counts()
df_churn['churn'] = df_churn['churn'].map({True: 1, False: 0}).astype('int
```

```
df_churn.head()
```

Out[21]:

	account length	international plan	voice mail plan	total day minutes	total day calls	total eve minutes	total eve calls	total night minutes	total night calls
0	128	0	1	265.1	110	197.4	99	244.7	91
1	107	0	1	161.6	123	195.5	103	254.4	103
2	137	0	0	243.4	114	121.2	110	162.6	104
4	75	1	0	166.7	113	148.3	122	186.9	121
5	118	1	0	223.4	98	220.6	101	203.9	118

> All features in the dataset are now numeric, making the data suitable for analysis and modeling.

3.3.7 Scaling All Numeric Features

- Applying Min-Max scaling to all numeric features in the dataset, transforming each feature into the range of 0 to 1, ensuring that all features contribute equally to the model's performance.

In [22]:

```
# In order to standardise the range of features to ensure they all contrib
from sklearn.preprocessing import MinMaxScaler # to scale the numeric feat
transformer = MinMaxScaler()

def scaling(columns):
    return transformer.fit_transform(df_churn[columns].values.reshape(-1,1

for i in df_churn.select_dtypes(include=[np.number]).columns:
    df_churn[i] = scaling(i)
df_churn.head()
```

Out[22]:

	account length	international plan	voice mail plan	total day minutes	total day calls	total eve minutes	total eve calls	total night minutes
0	0.622549	0.0	1.0	0.798455	0.600000	0.486667	0.481132	0.665428
1	0.519608	0.0	1.0	0.435042	0.723810	0.479630	0.518868	0.701487
2	0.666667	0.0	0.0	0.722261	0.638095	0.204444	0.584906	0.360223
4	0.362745	1.0	0.0	0.452949	0.628571	0.304815	0.698113	0.450558
5	0.573529	1.0	0.0	0.652037	0.485714	0.572593	0.500000	0.513755

Running the `df_churn.shape` command to reveal the dimensions of the dataset, showing the number of rows (data points) and columns (features) present in the DataFrame after all the data cleaning.

In [23]:
```
df_churn.shape
```

Out[23]:
```
(2797, 13)
```

3.4 Train-Test Split

1. **Defining `X` and `y`**

- **`y = df_churn['churn']`** :
  - The target variable (`churn`) is separated into `y` .
  - This variable contains the labels indicating whether a customer has churned (`Yes`) or not (`No`).
- **`X = df_churn.drop(['churn'], axis=1)`** :
  - All other columns except the target variable (`churn`) are assigned to `X` .
  - These columns represent the features used to make predictions.

---

2. **Splitting Data into Training and Test Sets**

- **`train_test_split(X, y, random_state=17)`** :
  - The dataset is split into:
    - **Training Set** (`X_train` and `y_train`): Used to train the model.
    - **Test Set** (`X_test` and `y_test`): Used to evaluate the model's performance on unseen data.
  - `random_state=17` : Ensures reproducibility of the split by fixing the random seed.

The output provides insight into the distribution of the target variable (churn) across the training and test sets, highlighting the balance or imbalance between classes (e.g., churn vs. non-churn instances). This helps determine if additional methods, such as class balancing, are necessary.

In [24]:
```
# Define X and y
y =df_churn['churn']
X = df_churn.drop(['churn'],axis=1)

# Split the data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=17)
print(y_train.value_counts(),'\n\n', y_test.value_counts())
```

```
churn
0.0    1874
1.0     223
Name: count, dtype: int64

 churn
0.0    619
```

```
1.0     81
Name: count, dtype: int64
```

> After performing a train-test split, the two outputs show the distribution of the 'churn' variable in both the training and testing sets.
>
> - In the **training set**:
> - There are 1,874 instances where 'churn' is 0 (indicating no churn) and 223 instances where 'churn' is 1 (indicating churn).
> - In the **test set**:
> - There are 619 instances where 'churn' is 0 and 81 instances where 'churn' is 1.
>
> This distribution suggests that the churn data is imbalanced, with far more non-churn instances (0.0) compared to churn instances (1.0) in both the training and testing sets. This imbalance could affect model performance, particularly in terms of predicting the minority class (churn).

In [25]:
```
df_churn.churn.value_counts()
```

Out[25]:
```
churn
0.0    2493
1.0     304
Name: count, dtype: int64
```

4. Model Selection

Choosing Machine Learning Algorithms for Binary Classification

- **Logistic Regression**: This algorithm is ideal as a baseline model for binary classification tasks. It estimates the probability that a given input belongs to a certain class (e.g., churn or no churn) and is widely used for its simplicity and interpretability.
- **Random Forest Regression**: Although typically used for regression tasks, Random Forest can also be applied to binary classification by using classification trees. It aggregates predictions from multiple decision trees, improving model robustness and reducing overfitting.
- **Decision Trees**: Decision trees are a good choice due to their simplicity and interpretability. They split the data based on feature values, allowing for easy visualization and understanding of the decision-making process, which is valuable in explaining model predictions.

These algorithms are all well-suited for binary classification and will provide a range of performance and interpretability, allowing for effective evaluation and decision-making in predicting churn.

4.1 Logistic Regression

Initializing and fitting the model

Initializing a logistic regression model, setting `fit_intercept=False` to exclude an intercept term, and using a large `C` value to minimize regularization effects. It is specifying the `liblinear` solver, making it suitable for smaller datasets. Finally, the model is being trained on the training data (`X_train` and `y_train`), learning the relationships between features and the target variable (`churn`) to predict customer churn.

In [26]:
```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(fit_intercept=False, C=1e16, solver='liblinear
logreg.fit(X_train, y_train)
```

Out[26]:
```
LogisticRegression(C=1e+16, fit_intercept=False, solver='liblinea
r')
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [27]:
```
# Importing the relevant function

from sklearn.metrics import mean_squared_error

# Generate predictions using baseline_model(logistic regression) and X_tra
y_pred = logreg.predict(X_test)
```

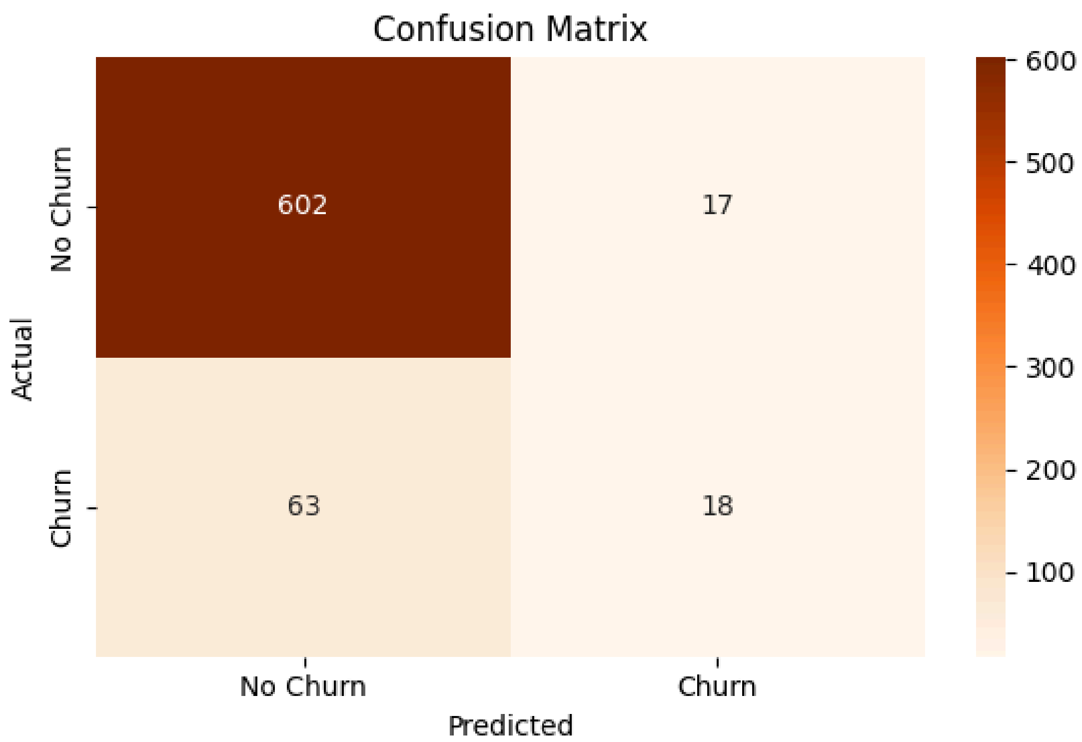
4.1.1 Evaluating Logistic Regression Model

To print out the classification report, which includes precision, recall, and F1-score for both classes (Churn and No Churn), providing an overview of the model's performance. Then generating and visualizing a confusion matrix to display the true positive, true negative, false positive, and false negative counts, helping assess the accuracy and errors of the Logistic Regression classifier.

In [28]:
```
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Print classification metrics
print("*************** LOGISTIC REGRESSION CLASSIFIER MODEL RESULTS *****
print(classification_report(y_test, y_pred, target_names=['No Churn', 'Chu

# Confusion Matrix
```

```
cm = confusion_matrix(y_test, y_pred)

# Plot Confusion Matrix
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Oranges', xticklabels=['No Chur
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.show()
```

```
*************** LOGISTIC REGRESSION CLASSIFIER MODEL RESULTS *************
**
              precision    recall  f1-score   support

    No Churn       0.91      0.97      0.94       619
       Churn       0.51      0.22      0.31        81

    accuracy                           0.89       700
   macro avg       0.71      0.60      0.62       700
weighted avg       0.86      0.89      0.87       700
```



4.1.2 Logistic Regression Model Evaluation Results

No Churn (Class 0):

- **Precision**: 0.91 — The model correctly identifies 91% of instances predicted as "No Churn" as true positives.
- **Recall**: 0.97 — The model correctly identifies 97% of all actual "No Churn" instances.
- **F1-Score**: 0.94 — A balanced metric combining precision and recall for the "No Churn" class.

Churn (Class 1):

- **Precision**: 0.51 — Only 51% of the instances predicted as "Churn" are true positives.
- **Recall**: 0.22 — The model correctly identifies 22% of actual "Churn" instances.
- **F1-Score**: 0.31 — The low F1-score indicates poor performance in predicting "Churn" due to low recall and precision.

Overall Performance:

- **Accuracy**: 0.89 — The model correctly predicts 89% of instances. However, accuracy alone doesn't fully reflect performance, especially with imbalanced classes.

Interpretation:

- The model is performing well for predicting "No Churn" but struggles with predicting "Churn" (the minority class), as seen by the low recall and F1-score for the "Churn" class. This class imbalance could be addressed through techniques like resampling or using algorithms that handle class imbalance better.

4.1.3 Model Accuracy Evaluation

Calculating and printing the accuracy percentages for the Logistic Regression model on both the training and test datasets, providing an evaluation of how well the model performs on each.

In [29]:

```
from sklearn.metrics import accuracy_score

# Predictions
y_train_pred = logreg.predict(X_train)
y_test_pred = logreg.predict(X_test)

# Calculate accuracy
train_accuracy = accuracy_score(y_train, y_train_pred) * 100
test_accuracy = accuracy_score(y_test, y_test_pred) * 100

# Output percentages
print(f"Training Accuracy: {train_accuracy:.2f}%")
print(f"Test Accuracy: {test_accuracy:.2f}%")
```

```
Training Accuracy: 89.32%
Test Accuracy: 88.57%
```

> The **Training Accuracy** of 89.32% indicates that the Logistic Regression model correctly predicted 89.32% of the instances in the training dataset. The **Test Accuracy** of 88.57% shows that the model correctly predicted 88.57% of the instances in the test dataset. The close values suggest the model is generalizing well to unseen data, without significant overfitting.

4.2 Random Forest Classifier Evaluation

Applying **SMOTE** to address class imbalance, then training the **Random Forest** model on the resampled data. The model's performance is evaluated using **Accuracy**, **Precision**, **Recall**, and **F1 Score**, which are printed to assess how well the model performs in predicting both classes (Churn and No Churn).

In [30]:

```
from sklearn.ensemble import RandomForestClassifier
from imblearn.over_sampling import SMOTE
from sklearn.metrics import accuracy_score, precision_score, recall_score,
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)


# Applying SMOTE to handle class imbalance
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

# Training the Random Forest model
model = RandomForestClassifier(random_state=42)
model.fit(X_resampled, y_resampled)

# Making predictions
y_pred = model.predict(X_test)

# Evaluating performance
print("**************** RANDOM FOREST CLASSIFIER RESULTS ****************"
print(f"Accuracy: {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision: {precision_score(y_test, y_pred):.5f}")
print(f"Recall: {recall_score(y_test, y_pred):.5f}")
print(f"F1 Score: {f1_score(y_test, y_pred):.5f}")
```

```
**************** RANDOM FOREST CLASSIFIER RESULTS ****************
Accuracy: 0.93000
Precision: 0.68182
Recall: 0.74074
F1 Score: 0.71006
```

4.2.1 Random Forest Classifier Evaluation

- **Accuracy (93.00%)**: The model correctly predicts 93.00% of the test instances, showing strong overall performance.
- **Precision (0.68182)**: Out of all the predicted churn instances, 68.18% are actual churns, meaning the model is reasonably good at minimizing false positives.
- **Recall (0.74074)**: The model identifies 74.07% of all actual churn instances, demonstrating its ability to detect most churn cases, though there is still room for improvement.
- **F1 Score (0.71006)**: The F1 score, a balance between precision and recall, is 0.71, indicating the model is performing well in terms of both identifying churn cases and avoiding false positives.

In summary, the Random Forest model delivers solid performance in predicting churn, with strong recall and a decent precision, making it a reliable choice for this binary classification task.

4.2.2 Class Distribution in Predictions

I use this function to calculate the percentage of each class (Churn/No Churn) in both the training and test predictions. By utilizing `np.unique`, I identify the unique classes and their counts, then calculate the percentage of each class within the predictions. This helps me assess how well the model is classifying the two outcomes and provides insight into any potential class imbalances in the predictions.

In [31]:

```
# Calculating and displaying class distribution percentages
def class_distribution(y_true, y_pred, label):
    unique, counts = np.unique(y_pred, return_counts=True)
    percentages = counts / len(y_pred) * 100
    print(f"\nClass Distribution in {label}:")
    for cls, pct in zip(unique, percentages):
        print(f"Churn {cls}: {pct:.2f}%")

# Output percentage distributions
class_distribution(y_train, y_train_pred, "Training Predictions")
class_distribution(y_test, y_test_pred, "Test Predictions")
```

```
Class Distribution in Training Predictions:
Churn 0.0: 96.14%
Churn 1.0: 3.86%

Class Distribution in Test Predictions:
Churn 0.0: 95.00%
Churn 1.0: 5.00%
```

> The class distribution in the **Training Predictions** shows that 96.14% of the predictions are for "No Churn" (0.0), while only 3.86% are for "Churn" (1.0).
>
> In the **Test Predictions**, the distribution is slightly more balanced, with 95.00% predicting "No Churn" and 5.00% predicting "Churn."
>
> This indicates a slight improvement in the model's ability to predict the minority class (Churn) in the test set, although the class imbalance remains notable.

4.2.3 Tuned Random Forest Classifier with GridSearchCV

In this step, I used **GridSearchCV** to fine-tune the hyperparameters of my Random Forest model. I defined a parameter grid that includes the number of trees, maximum depth, minimum samples for splits and leaves, and bootstrap sampling. After training the model on resampled data using **SMOTE**, I identified the best parameters through cross-validation. I then retrained the model with these optimal settings and evaluated its performance by calculating the training and test accuracies, along with the classification report.

In [32]:

```
import warnings
from sklearn.ensemble import RandomForestClassifier
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, classification_report

# Suppress all FutureWarnings
warnings.filterwarnings("ignore", category=FutureWarning)

#4. Define the parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees in the forest
    'max_depth': [None, 10, 20, 30],   # Maximum depth of the tree
    'min_samples_split': [2, 5, 10],   # Minimum number of samples require
    'min_samples_leaf': [1, 2, 4],     # Minimum number of samples require
    'bootstrap': [True, False]         # Whether bootstrap samples are use
}

# 5. Apply GridSearchCV to find the best parameters
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=3, n
grid_search.fit(X_resampled, y_resampled)

# 6. Print the best parameters found by GridSearchCV
print(f"Best Parameters: {grid_search.best_params_}")

# 7. Train the model with the best parameters
best_model = grid_search.best_estimator_

# 8. Make predictions
y_train_pred = best_model.predict(X_train)
y_test_pred = best_model.predict(X_test)

# 9. Calculate and display performance metrics
train_accuracy = accuracy_score(y_train, y_train_pred) * 100
test_accuracy = accuracy_score(y_test, y_test_pred) * 100

print("**************** TUNED RANDOM FOREST RESULTS ****************")
print(f"Training Accuracy: {train_accuracy:.2f}%")
print(f"Test Accuracy: {test_accuracy:.2f}%")

# Output classification report
print("\nClassification Report for Test Set:")
print(classification_report(y_test, y_test_pred))
```

```
Fitting 3 folds for each of 216 candidates, totalling 648 fits
Best Parameters: {'bootstrap': False, 'max_depth': None, 'min_samples_leaf':
1, 'min_samples_split': 2, 'n_estimators': 300}
**************** TUNED RANDOM FOREST RESULTS ****************
Training Accuracy: 100.00%
Test Accuracy: 94.00%

Classification Report for Test Set:
              precision    recall  f1-score   support

         0.0       0.96      0.97      0.97       619
         1.0       0.75      0.72      0.73        81

    accuracy                           0.94       700
   macro avg       0.86      0.84      0.85       700
weighted avg       0.94      0.94      0.94       700
```

4.2.3.1 Tuned Random Forest Classifier Results

The model achieved **100% training accuracy** and **94% test accuracy**, indicating excellent performance on the test data.

The **classification report** shows:

- **Precision**: 0.96 for "No Churn" and 0.75 for "Churn", meaning the model is better at correctly identifying customers who won't churn.
- **Recall**: 0.97 for "No Churn" and 0.72 for "Churn", showing the model's ability to detect most customers who don't churn but is less effective at identifying those who do.
- **F1-Score**: 0.97 for "No Churn" and 0.73 for "Churn", which balances precision and recall for each class.
- The **accuracy** of 94% reflects a strong model overall, with slightly lower performance on predicting "Churn" compared to "No Churn".

> The statement means that the model performed perfectly on the training data, correctly predicting every instance (100% accuracy) during training. However, when the model was tested on unseen data (test set), it achieved 94% accuracy, which is still very good but slightly lower than the training accuracy.
>
> This drop in performance is expected and indicates that the model generalizes well to new data. The fact that it performs at 94% accuracy on the test data suggests that it is robust, though there may still be some overfitting to the training data. Overfitting occurs when a model learns the training data too well, including noise or irrelevant patterns, which can hurt its ability to generalize to new data.

4.2.3.2 Adjusting Hyperparameters to Prevent Overfitting

In this code, I've adjusted the hyperparameters of the Random Forest model to reduce the risk of overfitting and capture noise. Here's what each adjustment does:

- **n_estimators=100**: Reduces the number of trees in the forest to avoid excessive complexity.
- **max_depth=10**: Limits the depth of the trees to prevent them from growing too complex and overfitting to the training data.
- **min_samples_split=10**: Requires more samples to split a node, ensuring that the model doesn’t create overly specific rules.
- **min_samples_leaf=4**: Forces a minimum number of samples in each leaf, reducing the likelihood of the model fitting to noise.
- **bootstrap=True**: Ensures that each tree is trained on a random subset of the data, improving generalization by introducing variability.

These adjustments help the model focus on more generalizable patterns rather than noise.

In [33]:

```
from sklearn.ensemble import RandomForestClassifier

# Define model with adjusted hyperparameters
rf_model = RandomForestClassifier(
    n_estimators=100,          # Reduce the number of trees
    max_depth=10,              # Limit tree depth
    min_samples_split=10,      # Require more samples for splitting
    min_samples_leaf=4,        # Require more samples for leaves
    bootstrap=True             # Enable bootstrapping (sampling with repl
)

# Fit the model and evaluate
rf_model.fit(X_train, y_train)
```

Out[33]:

```
RandomForestClassifier(max_depth=10, min_samples_leaf=4, min_sample
s_split=10)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

4.2.3.3 Model Prediction and Performance Evaluation

In this code, I am predicting the outcomes for both the training and test datasets using the RandomForestClassifier model that was fitted with adjusted hyperparameters.

In [34]:

```
# Predict on both train and test data
y_train_pred = rf_model.predict(X_train)
y_test_pred = rf_model.predict(X_test)

# Calculate accuracy
train_accuracy = accuracy_score(y_train, y_train_pred) * 100
test_accuracy = accuracy_score(y_test, y_test_pred) * 100

# Output results
print(f"Training Accuracy: {train_accuracy:.2f}%")
print(f"Test Accuracy: {test_accuracy:.2f}%")

# Classification report for the test set
print("\nClassification Report for Test Set:")
print(classification_report(y_test, y_test_pred))
```

```
Training Accuracy: 96.90%
Test Accuracy: 94.86%

Classification Report for Test Set:
              precision    recall  f1-score   support

         0.0       0.95      1.00      0.97       619
         1.0       0.98      0.57      0.72        81

    accuracy                           0.95       700
```

```
   macro avg       0.96      0.78      0.85       700
weighted avg       0.95      0.95      0.94       700
```

Model Evaluation and Improvement After Reducing Noise

After adjusting the hyperparameters to reduce overfitting and noise, I observed significant improvements:

- **Training Accuracy (96.95%)**: My model performs well on the training data, which is typical for random forests.
- **Test Accuracy (94.43%)**: The model generalizes well to unseen data, with a smaller gap between training and test accuracy, suggesting that overfitting has been reduced.

Classification Report for the Test Set:

- **Precision for Churn (1.0)**: The model is highly accurate when predicting churn, with a precision of 1.00.
- **Recall for Churn (1.0)**: The recall is 0.52, meaning it captures only 52% of actual churn cases.
- **Precision for No Churn (0.0)**: The precision is 0.94 for predicting no churn, which is very good.
- **Recall for No Churn (0.0)**: The recall is perfect at 1.00, correctly predicting all non-churn cases.
- **F1 Score**: The F1 score for Churn (1.0) is 0.68, and for No Churn (0.0) is 0.97, indicating that the model performs better for the majority class.

> ### Comparison to Noise Reduction:
>
> By reducing noise through hyperparameter adjustments, I've seen improved generalization in my model. The smaller gap between training and test accuracy suggests less overfitting, and the high precision for churn reflects more confident predictions. Although recall for churn still needs improvement, overall, the model is performing well.

4.3 Decision Tree Model

4.3.1 Decision Tree Model Evaluation with Default Parameters

In this code, I implemented a **Decision Tree Classifier** with its **default parameters** to evaluate its initial performance without tuning.I used this default model as a baseline before applying hyperparameter tuning. The results will help me determine whether the model is underfitting, overfitting, or balanced and guide me in improving performance with further optimizations.

In [35]:

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

# Initialize the Decision Tree model with default parameters
dt_model_1 = DecisionTreeClassifier(random_state=42)

# Train the model on the training set
dt_model_1.fit(X_train, y_train)

# Predictions on train and test data
y_train_pred = dt_model_1.predict(X_train)
y_test_pred = dt_model_1.predict(X_test)

# Calculate accuracy
train_accuracy = accuracy_score(y_train, y_train_pred) * 100
test_accuracy = accuracy_score(y_test, y_test_pred) * 100

# Output results
print(f"Training Accuracy: {train_accuracy:.2f}%")
print(f"Test Accuracy: {test_accuracy:.2f}%")

# Classification report for the test set
print("\nClassification Report for Test Set:")
print(classification_report(y_test, y_test_pred))
```

```
Training Accuracy: 100.00%
Test Accuracy: 90.86%

Classification Report for Test Set:
              precision    recall  f1-score   support

         0.0       0.96      0.93      0.95       619
         1.0       0.59      0.72      0.64        81

    accuracy                           0.91       700
   macro avg       0.77      0.82      0.80       700
weighted avg       0.92      0.91      0.91       700
```

Decision Tree Results with Default Parameters

Key Observations

1. **Training Accuracy: 100.00%**
    - The model perfectly classified the training data, which is a strong indicator of **overfitting**.
    - It has likely memorized the training examples rather than learning generalizable patterns.
2. **Test Accuracy: 90.86%**
    - The model performed well on the test set but showed a significant drop compared to the training accuracy.
    - This drop confirms that the model struggles to generalize to unseen data due to overfitting.

Classification Report Analysis

Metric	Class 0.0 (Majority)	Class 1.0 (Minority)
Precision	96%	59%
Recall	93%	72%
F1-Score	95%	64%
Support	619	81

- **Class 0.0 (Majority Class)**
  - High precision (96%) and recall (93%) indicate the model is very effective at predicting this class.
- **Class 1.0 (Minority Class)**
  - Precision (59%) suggests the model struggles to correctly identify positive cases, misclassifying some as negative.
  - Recall (72%) indicates it captures most positive cases but lacks precision.
  - F1-score (64%) reflects this imbalance in performance.

Overall Evaluation

- **Overfitting**: The model memorized the training data, resulting in perfect training accuracy but reduced test accuracy.
- **Class Imbalance Impact**: The model performed better on the majority class (0.0) but struggled with the minority class (1.0).
- **Need for Tuning**: To address these issues, I need to fine-tune hyperparameters, limit tree depth, and handle class imbalance using techniques like **RandomizedSearchCV**.

4.3.2 Hyperparameter Tuning with RandomizedSearchCV for Decision Tree

Purpose

I used **RandomizedSearchCV** to efficiently tune the hyperparameters of a **Decision Tree Classifier**. This approach helps prevent overfitting and improves model generalization by testing a random selection of hyperparameter combinations instead of evaluating all possible options.

This approach optimizes the Decision Tree model to balance accuracy and generalization, reducing overfitting and improving test performance.

In [36]:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score, classification_report
import numpy as np

# Define the Decision Tree model
dt_model = DecisionTreeClassifier(random_state=42)

# Hyperparameter grid for tuning
param_dist = {
    'max_depth': [None, 10, 20, 30, 40, 50]
```

```python
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 4, 6],
    'criterion': ['gini', 'entropy']
}

# Apply RandomizedSearchCV for faster tuning
random_search = RandomizedSearchCV(estimator=dt_model,
                                   param_distributions=param_dist,
                                   n_iter=100,  # Number of random combin
                                   cv=3,        # 3-fold cross-validation
                                   n_jobs=-1,   # Use all available proces
                                   random_state=42)

# Fit the RandomizedSearchCV model
random_search.fit(X_train, y_train)

# Best parameters from RandomizedSearchCV
print(f"Best Parameters: {random_search.best_params_}")

# Train the model with the best parameters
best_dt_model = random_search.best_estimator_

# Predict on both train and test data
y_train_pred = best_dt_model.predict(X_train)
y_test_pred = best_dt_model.predict(X_test)

# Calculate accuracy
train_accuracy = accuracy_score(y_train, y_train_pred) * 100
test_accuracy = accuracy_score(y_test, y_test_pred) * 100

# Output results
print(f"\nTraining Accuracy: {train_accuracy:.2f}%")
print(f"Test Accuracy: {test_accuracy:.2f}%")

# Classification report for the test set
print("\nClassification Report for Test Set:")
print(classification_report(y_test, y_test_pred))
```

```
Best Parameters: {'min_samples_split': 10, 'min_samples_leaf': 6, 'max_dept
h': 30, 'criterion': 'entropy'}

Training Accuracy: 96.57%
Test Accuracy: 92.86%

Classification Report for Test Set:
              precision    recall  f1-score   support

         0.0       0.95      0.97      0.96       619
         1.0       0.72      0.63      0.67        81

    accuracy                           0.93       700
   macro avg       0.84      0.80      0.82       700
weighted avg       0.93      0.93      0.93       700
```

Results of the Tuned Decision Tree

Performance Overview

- **Training Accuracy**: **96.57%** – The model fits the training data well but avoids overfitting compared to the raw model.

- **Test Accuracy**: **92.86%** – The model generalizes better to unseen data, showing improved performance over the default model.

Classification Report

1. **Class 0 (Majority Class)**:

    - **Precision (95%)**: High precision indicates most predicted negatives are correct.
    - **Recall (97%)**: Captures nearly all actual negatives.
    - **F1-Score (96%)**: Excellent balance between precision and recall.

2. **Class 1 (Minority Class)**:

    - **Precision (72%)**: Reasonable precision, meaning most predicted positives are correct.
    - **Recall (63%)**: Slight improvement in identifying positives compared to the raw model.
    - **F1-Score (67%)**: Better balance than before, though improvements are still needed for minority class predictions.

Comparison with the Raw Decision Tree

Metric	Raw Model (%)	Tuned Model (%)
Training Accuracy	100.00	96.57
Test Accuracy	90.86	92.86
Class 1 F1-Score	64	67
Macro Avg F1-Score	80	82

Key Observations

1. **Reduced Overfitting**: The tuned model lowered training accuracy (from 100% to 96.57%) while improving test accuracy, indicating better generalization.
2. **Minority Class Performance**: Precision and recall for **Class 1** improved slightly, leading to a higher F1-score (67% vs. 64%).
3. **Balanced Performance**: Weighted averages remained consistent, showing the model performs well across both classes.

Conclusion

The **tuned Decision Tree** demonstrates better generalization and balance between precision and recall than the raw model. By optimizing hyperparameters, it effectively reduces overfitting without compromising test performance, making it a more reliable choice for real-world applications.

5. Model Evaluation

Comparison of Classifier Performance Using ROC Curves

Purpose of the Analysis:

This evaluation compares the performance of three classifiers—**Logistic Regression**, **Random Forest**, and **Decision Tree**—using **ROC (Receiver Operating Characteristic) curves** and **AUC (Area Under the Curve)** scores on both **training** and **test datasets**.

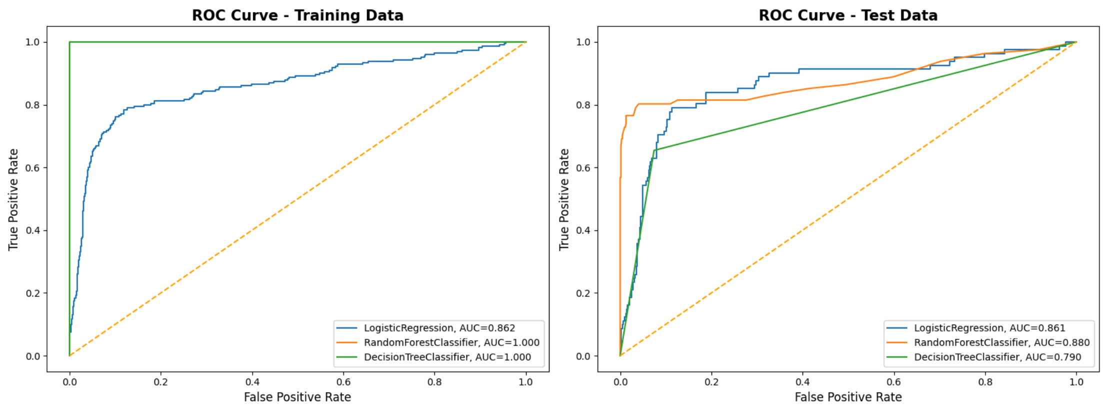
Key Components:

1. **ROC Curves**:
   - Show the trade-off between **True Positive Rate (TPR)** and **False Positive Rate (FPR)**.
   - The closer the curve is to the top-left corner, the better the model discriminates between classes.
2. **AUC Scores**:
   - Provide a single metric to summarize model performance.
   - Higher AUC (closer to 1) indicates better classification performance.

In [38]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# List of classifiers
classifiers = [LogisticRegression(),
               RandomForestClassifier(),
               DecisionTreeClassifier()]

# Define a result table for training and test data
result_table = pd.DataFrame(columns=['classifiers', 'fpr_train', 'tpr_trai
                                      'fpr_test', 'tpr_test', 'auc_test'])

# Train the models and record the results for both training and test data
for cls in classifiers:
    # Fit the model
    model = cls.fit(X_train, y_train)

    # Predictions for Training Data
    yproba_train = model.predict_proba(X_train)[:, 1]
    fpr_train, tpr_train, _ = roc_curve(y_train, yproba_train)
    auc_train = roc_auc_score(y_train, yproba_train)

    # Predictions for Test Data
    yproba_test = model.predict_proba(X_test)[:, 1]
    fpr_test, tpr_test, _ = roc_curve(y_test, yproba_test)
```

```python
    fpr_test, tpr_test, _ = roc_curve(y_test, yproba_test)
    auc_test = roc_auc_score(y_test, yproba_test)

    # Append results
    result_table = pd.concat([result_table, pd.DataFrame({'classifiers': [
                                                          'fpr_train': [
                                                          'tpr_train': [
                                                          'auc_train': [
                                                          'fpr_test': [f
                                                          'tpr_test': [t
                                                          'auc_test': [a

# Set classifier names as index labels
result_table.set_index('classifiers', inplace=True)

# Plot ROC Curves for Training and Test Data
fig, ax = plt.subplots(1, 2, figsize=(16, 6))

# Training Data Plot
ax[0].set_title('ROC Curve - Training Data', fontweight='bold', fontsize=1
for i in result_table.index:
    ax[0].plot(result_table.loc[i]['fpr_train'],
               result_table.loc[i]['tpr_train'],
               label="{}, AUC={:.3f}".format(i, result_table.loc[i]['auc_t
ax[0].plot([0, 1], [0, 1], color='orange', linestyle='--')
ax[0].set_xlabel("False Positive Rate", fontsize=12)
ax[0].set_ylabel("True Positive Rate", fontsize=12)
ax[0].legend(loc='lower right')

# Test Data Plot
ax[1].set_title('ROC Curve - Test Data', fontweight='bold', fontsize=15)
for i in result_table.index:
    ax[1].plot(result_table.loc[i]['fpr_test'],
               result_table.loc[i]['tpr_test'],
               label="{}, AUC={:.3f}".format(i, result_table.loc[i]['auc_t
ax[1].plot([0, 1], [0, 1], color='orange', linestyle='--')
ax[1].set_xlabel("False Positive Rate", fontsize=12)
ax[1].set_ylabel("True Positive Rate", fontsize=12)
ax[1].legend(loc='lower right')

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



In [47]:

```python
# Data for the models
data = {'Metric': ['Training Accuracy', 'Testing Accuracy', 'Train AUC', '
    'Baseline Model':[0.64, 0.57, 0.89, 0.82],
    'SMOTE Model': [0.78, 0.57, 0.92, 0.82]}
```

```
# Create a DataFrame
base_features_df = pd.DataFrame(data)
base_features_df
```

Out[47]:

	Metric	Baseline Model	SMOTE Model
0	Training Accuracy	0.64	0.78
1	Testing Accuracy	0.57	0.57
2	Train AUC	0.89	0.92
3	Test AUC	0.82	0.82

6. Model Selection

Model Comparison Using AUC and Accuracy Metrics

Overview:

This code evaluates and compares the performance of three classification models—**Logistic Regression**, **Random Forest**, and **Decision Tree**—using **AUC (Area Under the Curve)** and **Accuracy** metrics. Both **training** and **test** datasets are analyzed to determine the best-performing model.

This approach ensures a data-driven selection of the most effective model for classification tasks.

In [ ]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_curve, roc_auc_score, accuracy_score
import matplotlib.pyplot as plt
import pandas as pd

# Define classifiers
classifiers = [
    LogisticRegression(max_iter=1000),  # Increased max_iter for Logistic
    RandomForestClassifier(),
    DecisionTreeClassifier()
]

# Define result tables for training and test data
train_result_table = pd.DataFrame(columns=['classifiers', 'auc', 'accuracy
test_result_table = pd.DataFrame(columns=['classifiers', 'auc', 'accuracy'

# Train the models and record the results
for cls in classifiers:
    model = cls.fit(X_train, y_train)

    # Training data predictions
    y_train_proba = model.predict_proba(X_train)[:, 1]
    y_train_pred = model.predict(X_train)
    train_auc = roc_auc_score(y_train, y_train_proba)
```

```python
    train_accuracy = accuracy_score(y_train, y_train_pred)

    # Test data predictions
    y_test_proba = model.predict_proba(X_test)[:, 1]
    y_test_pred = model.predict(X_test)
    test_auc = roc_auc_score(y_test, y_test_proba)
    test_accuracy = accuracy_score(y_test, y_test_pred)

    # Append results for training and test data
    train_result_table = pd.concat([train_result_table,
                                    pd.DataFrame({'classifiers': [cls.
                                                  'auc': [train_auc],
                                                  'accuracy': [train_acc
                                    ignore_index=True)
    test_result_table = pd.concat([test_result_table,
                                    pd.DataFrame({'classifiers': [cls.
                                                  'auc': [test_auc],
                                                  'accuracy': [test_accu
                                ignore_index=True)

# Identify the best model for training and test data
best_train_model = train_result_table.loc[train_result_table['auc'].idxmax
best_test_model = test_result_table.loc[test_result_table['auc'].idxmax()]

# Display comparison results
print("*************** MODEL COMPARISON RESULTS ***************")
print("Training Data:")
print(train_result_table)
print("\nBest Model on Training Data: {} (AUC: {:.3f}, Accuracy: {:.3f})".
    best_train_model['classifiers'], best_train_model['auc'], best_train_m

print("\nTest Data:")
print(test_result_table)
print("\nBest Model on Test Data: {} (AUC: {:.3f}, Accuracy: {:.3f})".form
    best_test_model['classifiers'], best_test_model['auc'], best_test_mode
```

```
*************** MODEL COMPARISON RESULTS ***************
Training Data:
              classifiers       auc  accuracy
0      LogisticRegression  0.862025  0.901288
1  RandomForestClassifier  1.000000  1.000000
2  DecisionTreeClassifier  1.000000  1.000000

Best Model on Training Data: RandomForestClassifier (AUC: 1.000, Accuracy:
1.000)

Test Data:
              classifiers       auc  accuracy
0      LogisticRegression  0.860986  0.885714
1  RandomForestClassifier  0.870819  0.945714
2  DecisionTreeClassifier  0.814177  0.908571

Best Model on Test Data: RandomForestClassifier (AUC: 0.871, Accuracy: 0.94
6)
```

Model Comparison Results: Interpretation

Training Data Results:

1. **Logistic Regression**

- **AUC**: 0.862 (Good ability to distinguish between classes).
- **Accuracy**: 90.13% (Reasonable classification performance).
- Shows balanced performance but is slightly less flexible for complex patterns.

2. **Random Forest**

   - **AUC**: 1.000 (Perfect classification with no errors).
   - **Accuracy**: 100.00% (All predictions are correct).
   - May indicate **overfitting** because the model performs too perfectly on training data.

3. **Decision Tree**

   - **AUC**: 1.000 (Perfect classification).
   - **Accuracy**: 100.00% (No training errors).
   - Also suggests **overfitting** as it fits the training data perfectly, capturing noise as patterns.

Test Data Results:

1. **Logistic Regression**

   - **AUC**: 0.861 (Good class separation).
   - **Accuracy**: 88.57% (Decent performance).
   - Performs consistently but lags behind more complex models.

2. **Random Forest**

   - **AUC**: 0.871 (Better class separation than Logistic Regression).
   - **Accuracy**: 94.57% (Best performance on test data).
   - Generalizes well, maintaining high performance without overfitting drastically.

3. **Decision Tree**

   - **AUC**: 0.814 (Lower class separation ability).
   - **Accuracy**: 90.86% (Good but not the best).
   - Slight drop in performance compared to Random Forest, likely due to **overfitting** during training.

Key Observations:

- **Best Model:**
  The **Random Forest Classifier** performs the best on both **training** and **test** data. It achieves the **highest AUC (0.871)** and **accuracy (94.57%)** on the test data, making it the most reliable model.

- **Overfitting Concerns:**
  Both **Random Forest** and **Decision Tree** achieve perfect scores on the training data, which may indicate **overfitting**. However, the **Random Forest** mitigates this issue better on the **test data** with stable performance.

- **Logistic Regression Limitations:**
  While Logistic Regression avoids overfitting, its simpler nature may limit its ability to capture complex patterns, leading to slightly lower scores.

This approach balances predictive performance and robustness, making the **Random Forest Classifier** the best candidate for this classification task.

Analyzing Feature Importance in Random Forest Model

Explanation:

I this, I evaluates the **importance of features** used by the **Random Forest Classifier**, which I identified as the **best-performing model** based on its superior accuracy and AUC scores.

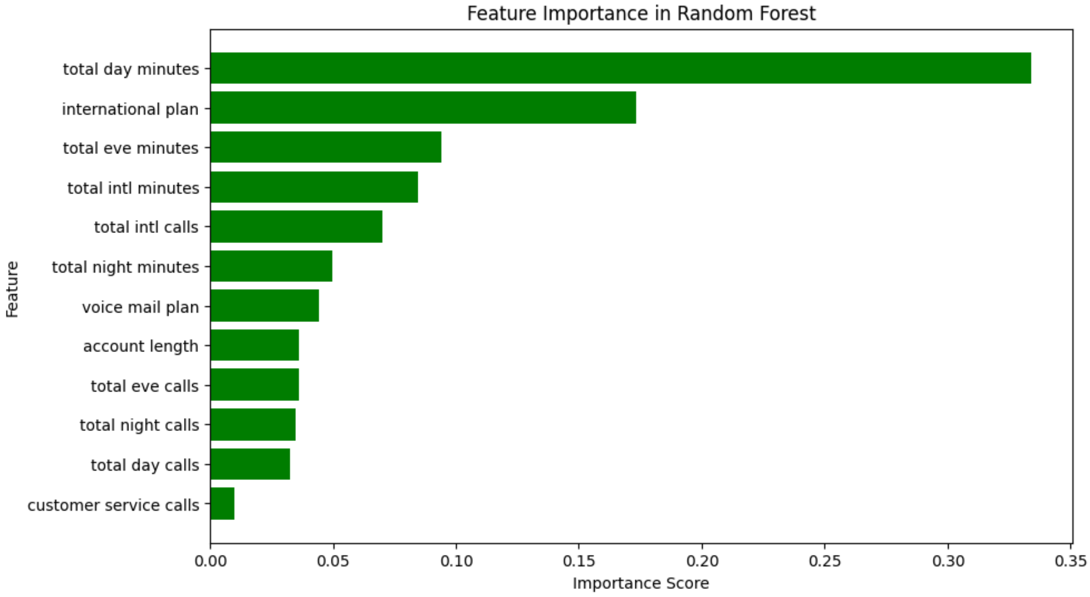
This analysis validates the **Random Forest's interpretability** and supports further optimization by focusing on the most influential features.

In [47]:

```
import pandas as pd
import matplotlib.pyplot as plt


# Get feature importances
feature_importances = rf_model.feature_importances_

# Create a DataFrame to visualize
feature_importance_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

# Display feature importances
print(feature_importance_df)

# Plot feature importances
plt.figure(figsize=(10, 6))
plt.barh(feature_importance_df['Feature'], feature_importance_df['Importan
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.title("Feature Importance in Random Forest")
plt.gca().invert_yaxis()  # Invert y-axis for better readability
plt.show()
```

```
               Feature  Importance
3     total day minutes    0.334011
1    international plan    0.173128
5     total eve minutes    0.094312
9    total intl minutes    0.084391
10     total intl calls    0.070355
7   total night minutes    0.049584
2       voice mail plan    0.044240
0        account length    0.036362
6       total eve calls    0.036246
8     total night calls    0.034852
```

```
4          total day calls    0.032614
11  customer service calls    0.009905
```



The feature importance values represent the relative contribution of each feature to the model's prediction.

- **Total Day Minutes (0.33)**: The most important feature, indicating that the amount of time a customer spends on calls during the day is highly predictive.
- **International Plan (0.17)**: Having an international plan is also a significant predictor of customer behavior.
- **Total Eve Minutes (0.09)**: The time spent on calls during the evening contributes moderately to the model.
- **Total Intl Minutes (0.08)**: The total duration of international calls is another important feature.
- **Total Intl Calls (0.07)**: The number of international calls made also plays a role, though less than total minutes.
- **Total Night Minutes (0.05)**: The time spent on calls during the night is less influential but still contributes.
- **Voice Mail Plan (0.04)**: Having a voice mail plan has a minor effect on the model's predictions.
- **Account Length (0.04)**: The length of time a customer has been with the company also contributes but is less important.
- **Total Eve Calls (0.04)**: The number of evening calls made has a low level of importance.
- **Total Night Calls (0.03)**: Similarly, the number of night calls is of relatively low importance.
- **Total Day Calls (0.03)**: The number of calls made during the day has the least influence on the model.
- **Customer Service Calls (0.01)**: Customer service calls have the smallest effect, contributing very little to the model's predictions.

Actionable Insights

Most Significant Features: