In [1]:
```
!nvidia-smi  # this should display information about available GPUs
```

```
Sun Jan 14 18:32:44 2024
+-------------------------------------------------------------------------------------
---+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2
|
|-----------------------------------------+----------------------+--------------------
---+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. E
CC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute
M. |
|                                         |                      |               MIG
M. |
|=========================================+======================+====================
===|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
0 |
| N/A   47C    P8              10W /  70W |      0MiB / 15360MiB |      0%      Defau
lt |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+--------------------
---+

+-------------------------------------------------------------------------------------
---+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                            GPU Memo
ry |
|        ID   ID                                                             Usage
|
|=====================================================================================
===|
|  No running processes found
|
+-------------------------------------------------------------------------------------
---+
```

In [ ]:
```
!pip install cudf-cu12 --extra-index-url=https://pypi.nvidia.com
```

In [3]:
```
import cudf  # this should work without any errors
```

In [ ]:
```
!pip install plotly-express
```

Importing Libraries

In [5]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, c
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler
```

In [7]:
```
# Load your dataset
df = pd.read_csv("/content/Churn_Modelling.csv")
```

Data Preprocessing

Data Statistics

In [9]:
```
data_statistics = df.describe()
print(data_statistics)
```

```
         RowNumber    CustomerId   CreditScore           Age        Tenure  \
count  10000.00000  1.000000e+04  10000.000000  10000.000000  10000.000000   
mean    5000.50000  1.569094e+07    650.528800     38.921800      5.012800   
std     2886.89568  7.193619e+04     96.653299     10.487806      2.892174   
min        1.00000  1.556570e+07    350.000000     18.000000      0.000000   
25%     2500.75000  1.562853e+07    584.000000     32.000000      3.000000   
50%     5000.50000  1.569074e+07    652.000000     37.000000      5.000000   
75%     7500.25000  1.575323e+07    718.000000     44.000000      7.000000   
max    10000.00000  1.581569e+07    850.000000     92.000000     10.000000   

             Balance  NumOfProducts    HasCrCard  IsActiveMember  \
count   10000.000000   10000.000000  10000.00000    10000.000000   
mean    76485.889288       1.530200      0.70550        0.515100   
std     62397.405202       0.581654      0.45584        0.499797   
min         0.000000       1.000000      0.00000        0.000000   
25%         0.000000       1.000000      0.00000        0.000000   
50%     97198.540000       1.000000      1.00000        1.000000   
75%    127644.240000       2.000000      1.00000        1.000000   
max    250898.090000       4.000000      1.00000        1.000000   

       EstimatedSalary        Exited  
count     10000.000000  10000.000000  
mean     100090.239881      0.203700  
std       57510.492818      0.402769  
min          11.580000      0.000000  
25%       51002.110000      0.000000  
50%      100193.915000      0.000000  
75%      149388.247500      0.000000  
max      199992.480000      1.000000  
```

In [10]:
```
missing_values = df.isnull().sum()
print(missing_values)
```

```
RowNumber          0
CustomerId         0
Surname            0
CreditScore        0
Geography          0
Gender             0
Age                0
Tenure             0
Balance            0
NumOfProducts      0
HasCrCard          0
IsActiveMember     0
EstimatedSalary    0
Exited             0
dtype: int64
```

In [42]:
```
# dropping unnecessary columns
drop_cols = ['CustomerId','Surname','RowNumber']
df.drop(drop_cols, axis=1, inplace=True)
df.head()
```

Out[42]:

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMe
0	619	France	Female	42	2	0.00	1	1	
1	608	Spain	Female	41	1	83807.86	1	0	
2	502	France	Female	42	8	159660.80	3	1	
3	699	France	Female	39	1	0.00	2	0	
4	850	Spain	Female	43	2	125510.82	1	1	

In [43]:
```
print(df['Geography'].unique())
df['Geography'].value_counts()
```

```
['France' 'Spain' 'Germany']
```

Out[43]:
```
France     5014
Germany    2509
Spain      2477
Name: Geography, dtype: int64
```

In [14]:
```
# Identify outliers using Interquartile Range (IQR)
def detect_outliers_iqr(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    outliers = data[(data[column] < lower_bound) | (data[column] > upper_bound)]
    return outliers

# Assume 'Balance' is the column you want to check for outliers
numeric_column = 'Balance'

outliers = detect_outliers_iqr(df, numeric_column)
```

```
# Display the outliers
print("\nOutliers:")
print(outliers)
```

```
Outliers:
Empty DataFrame
Columns: [RowNumber, CustomerId, Surname, CreditScore, Geography, Gender, Age, Tenur
e, Balance, NumOfProducts, HasCrCard, IsActiveMember, EstimatedSalary, Exited]
Index: []
```

In [20]:

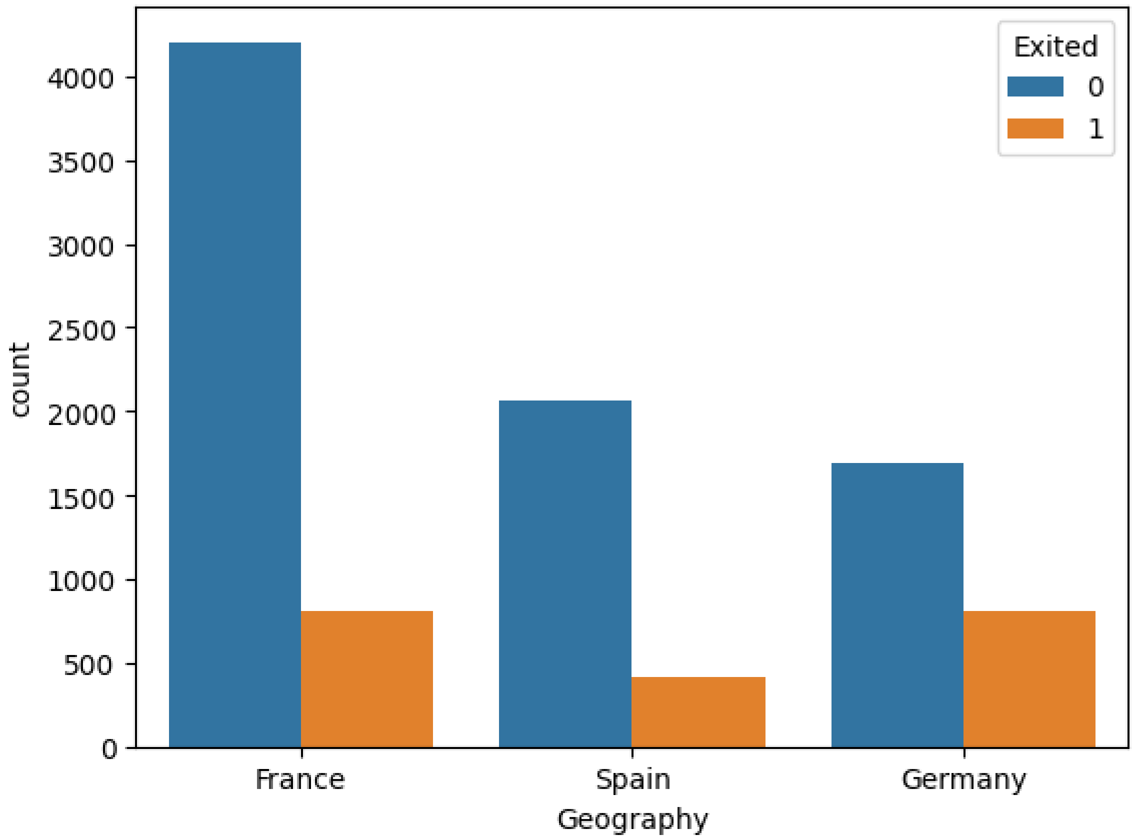
```
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

# Assuming 'df' is your DataFrame, and you want to use certain columns for clustering
selected_columns = ['CreditScore', 'Age', 'Balance', 'NumOfProducts', 'EstimatedSalary

# Extract the features (X) from the DataFrame
X = df[selected_columns]

# Standardize the features (important for DBSCAN)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Now, you can use X_scaled with DBSCAN
outlier_detector = DBSCAN(eps=3, min_samples=2)
outliers = outlier_detector.fit_predict(X_scaled)
print(outliers)

num_outliers = len(outliers[outliers == -1])
print(f"Number of outliers: {num_outliers}")
```

```
[0 0 0 ... 0 0 0]
Number of outliers: 0
```

Exploratory Data Analysis EDA

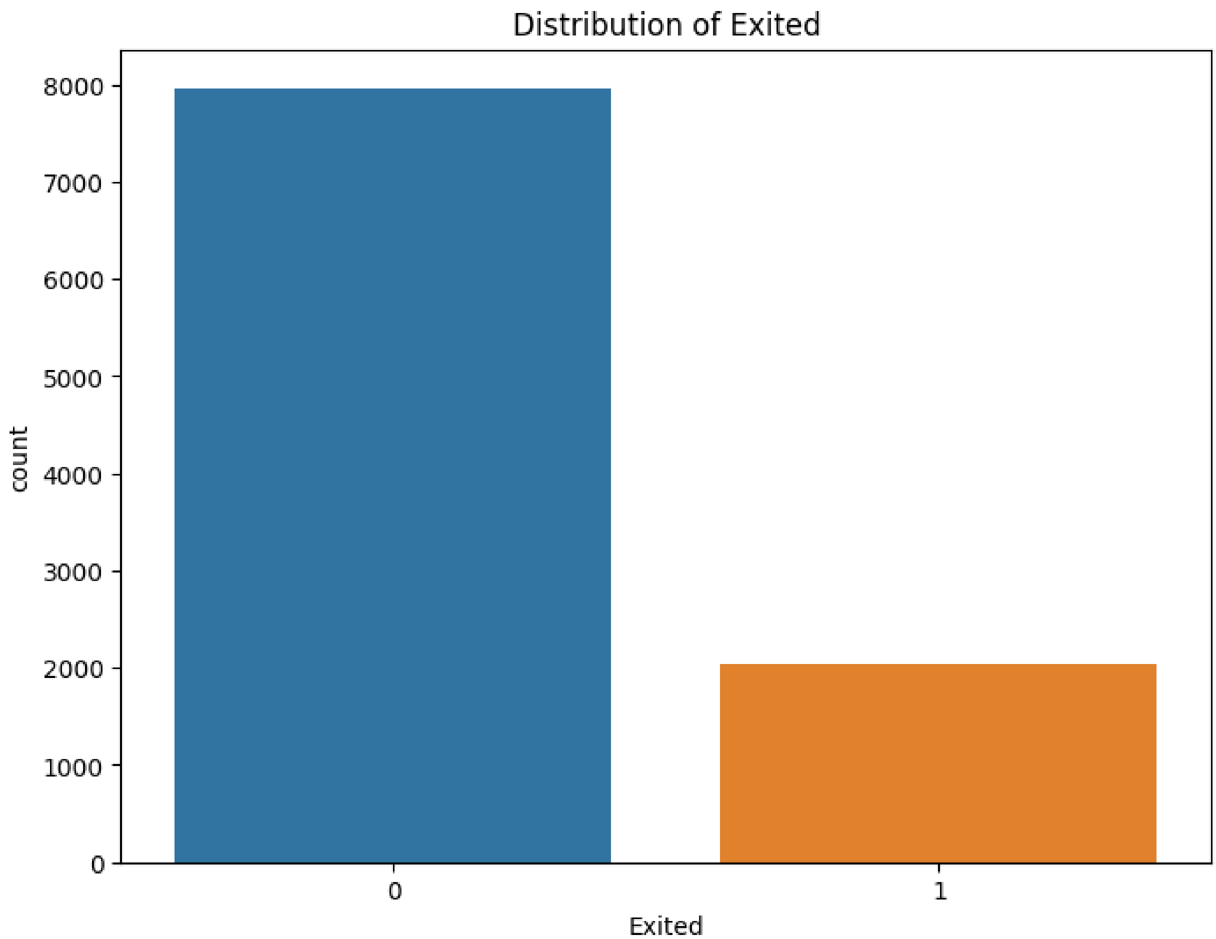
In [21]:

```
sns.countplot(x='Geography', hue='Exited', data=df)
plt.show()
```

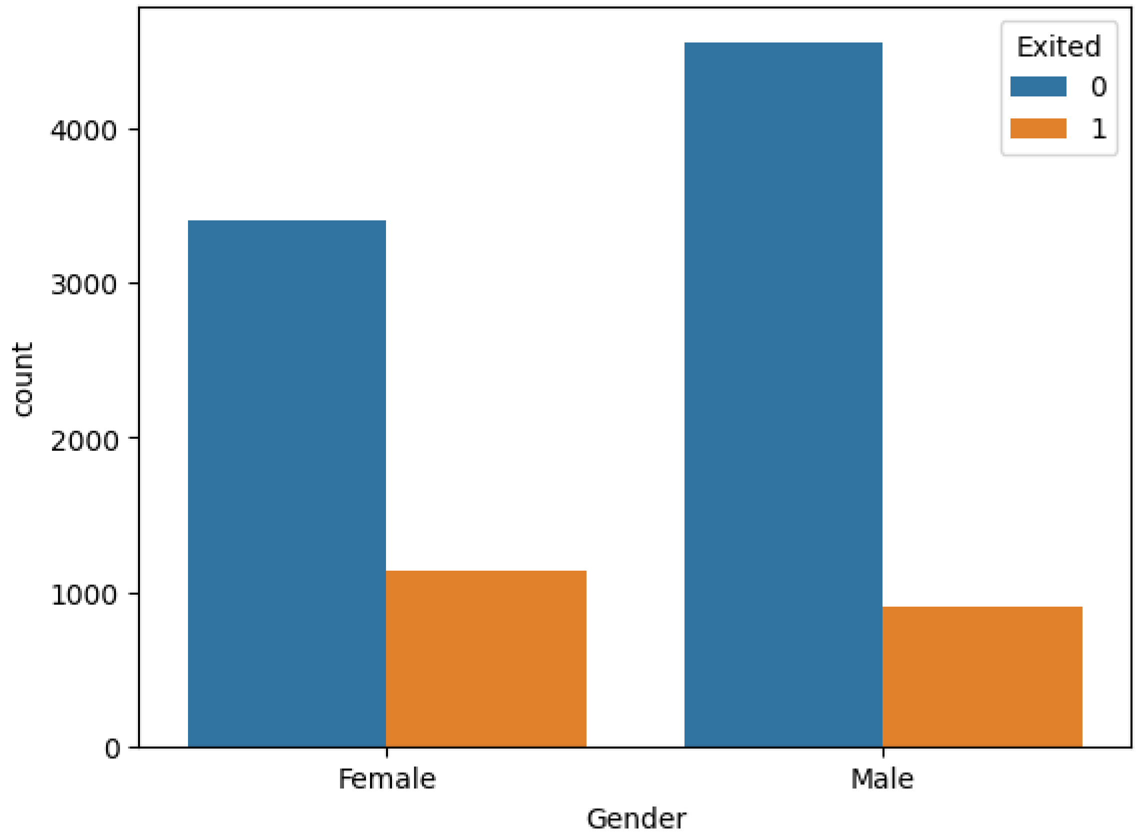


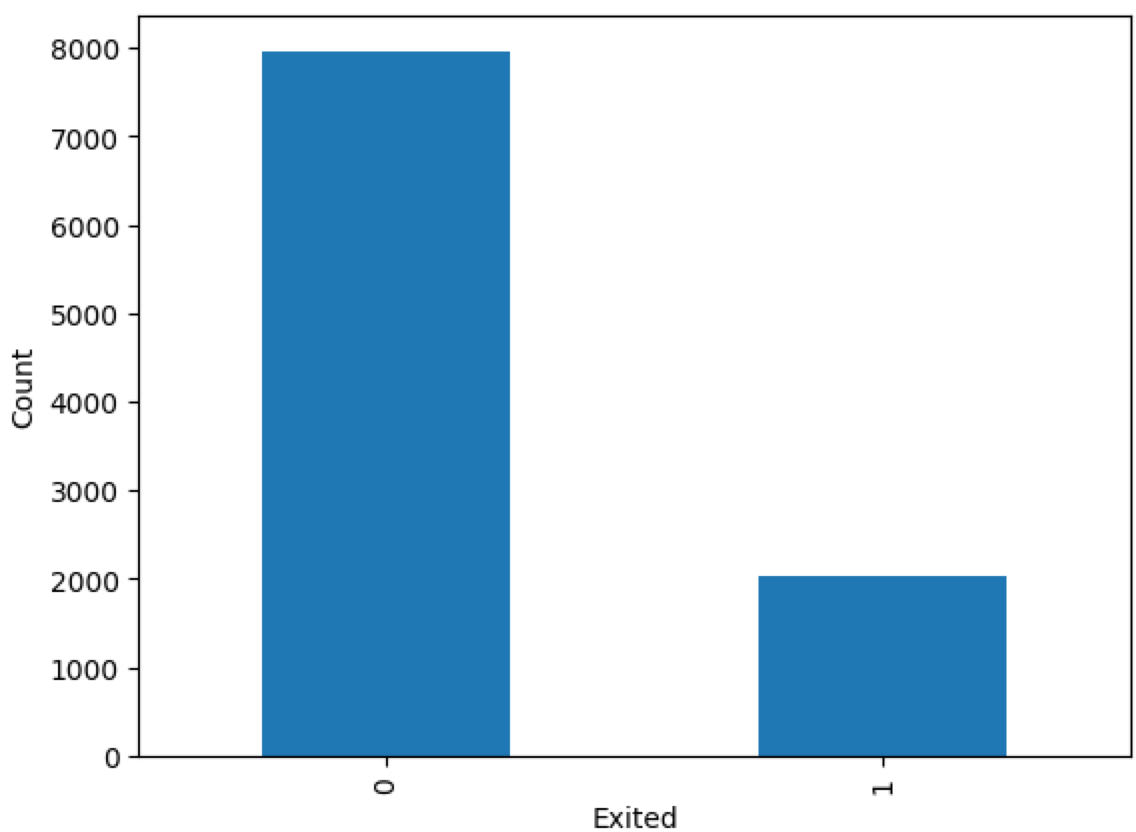
In [22]:

```
# Distribution of the target variable 'Exited'
plt.figure(figsize=(8, 6))
sns.countplot(x='Exited', data=df)
plt.title('Distribution of Exited')
plt.show()
```



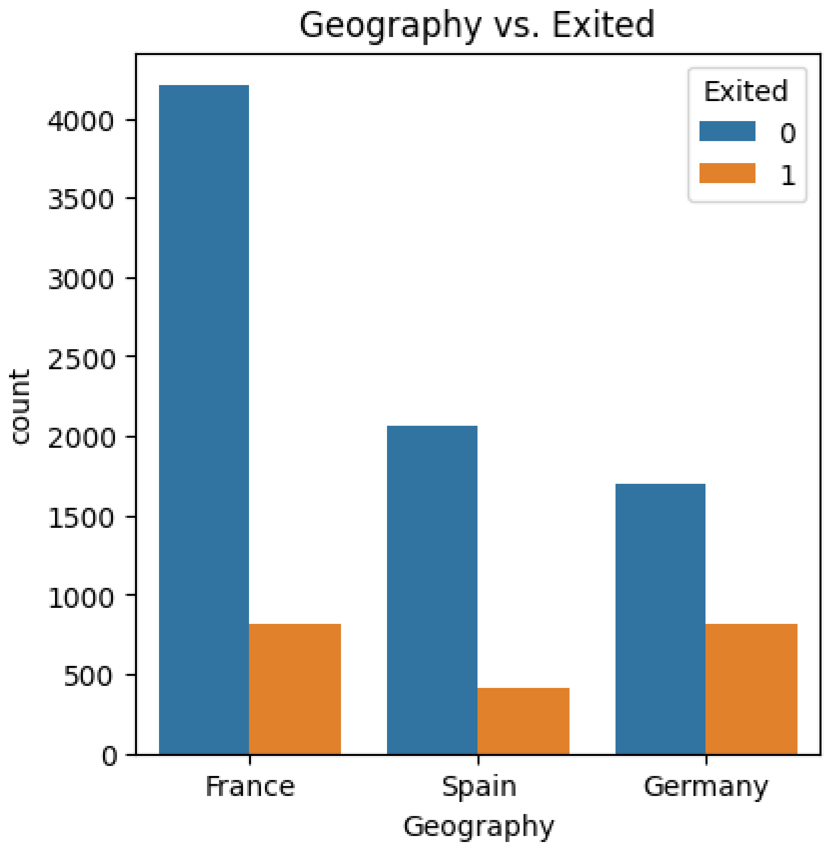
In [36]:
```
sns.countplot(x='Gender', hue='Exited', data=df)
plt.show()
df['Exited'].value_counts().plot(kind='bar')
plt.xlabel('Exited')
plt.ylabel('Count')
plt.show()
```





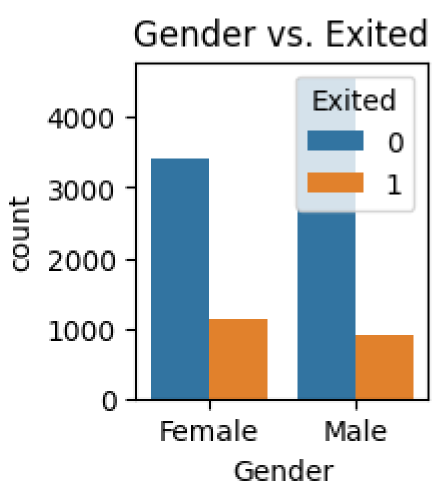
In [23]:
```
# Distribution of categorical features
plt.figure(figsize=(15, 10))
plt.subplot(2, 3, 1)
sns.countplot(x='Geography', hue='Exited', data=df)
plt.title('Geography vs. Exited')
```

Out[23]: Text(0.5, 1.0, 'Geography vs. Exited')



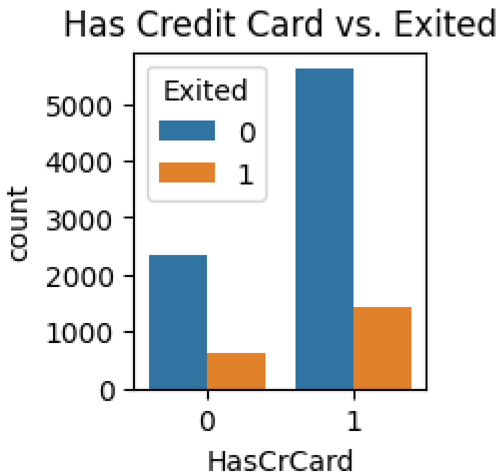
In [24]:
```
plt.subplot(2, 3, 2)
sns.countplot(x='Gender', hue='Exited', data=df)
plt.title('Gender vs. Exited')
```

Out[24]: Text(0.5, 1.0, 'Gender vs. Exited')



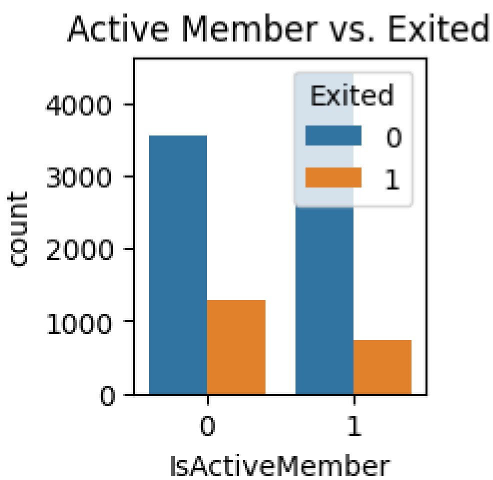
In [25]:
```
plt.subplot(2, 3, 3)
sns.countplot(x='HasCrCard', hue='Exited', data=df)
plt.title('Has Credit Card vs. Exited')
```

Out[25]: Text(0.5, 1.0, 'Has Credit Card vs. Exited')



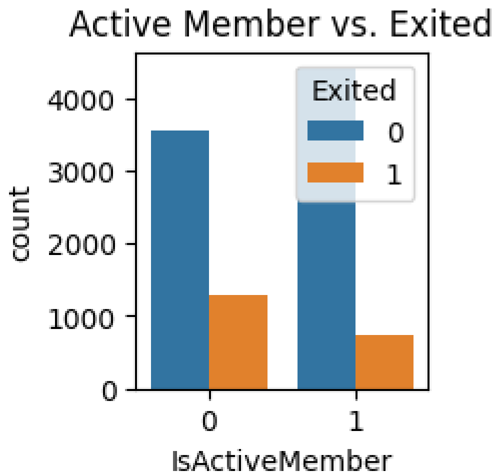
In [26]:
```
plt.subplot(2, 3, 4)
sns.countplot(x='IsActiveMember', hue='Exited', data=df)
plt.title('Active Member vs. Exited')
```

Out[26]:
```
Text(0.5, 1.0, 'Active Member vs. Exited')
```

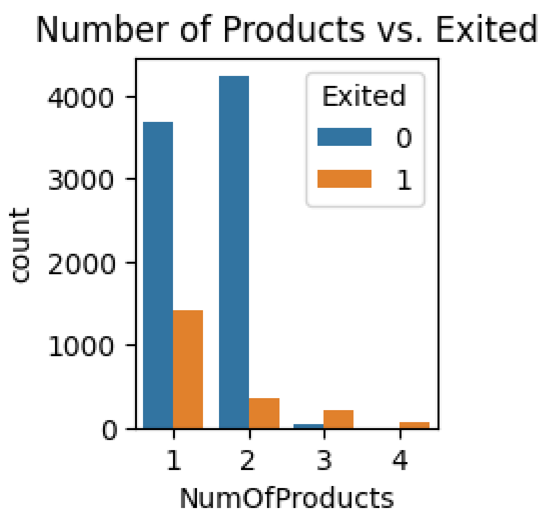


In [27]:
```
plt.subplot(2, 3, 4)
sns.countplot(x='IsActiveMember', hue='Exited', data=df)
plt.title('Active Member vs. Exited')
```

Out[27]:
```
Text(0.5, 1.0, 'Active Member vs. Exited')
```

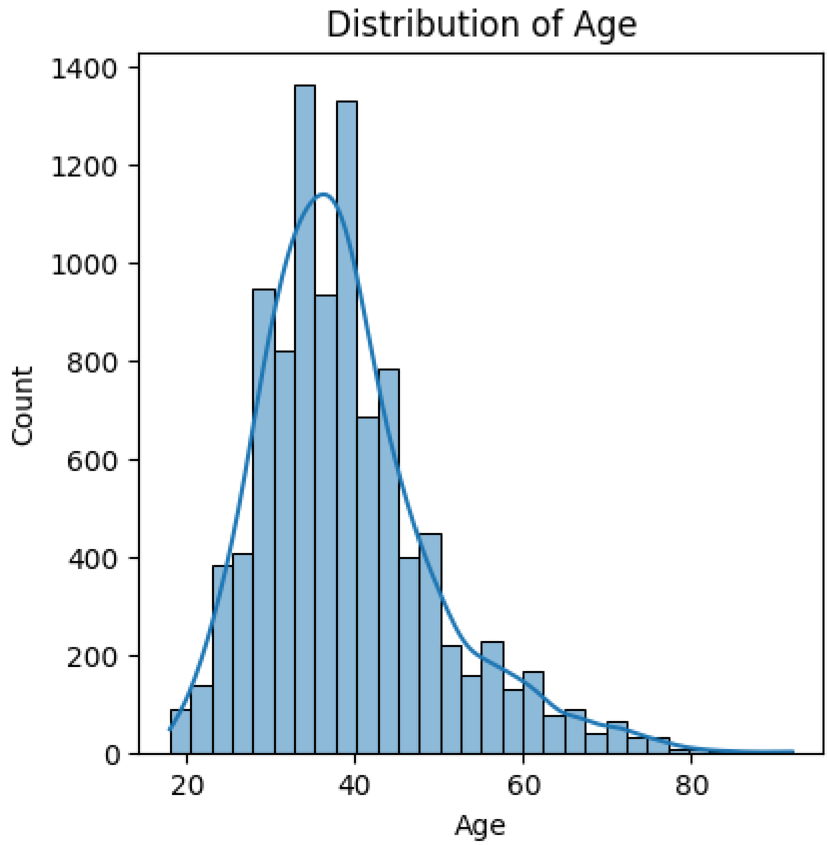


In [29]:
```
plt.subplot(2, 3, 5)
sns.countplot(x='NumOfProducts', hue='Exited', data=df)
plt.title('Number of Products vs. Exited')
plt.tight_layout()
plt.show()
```



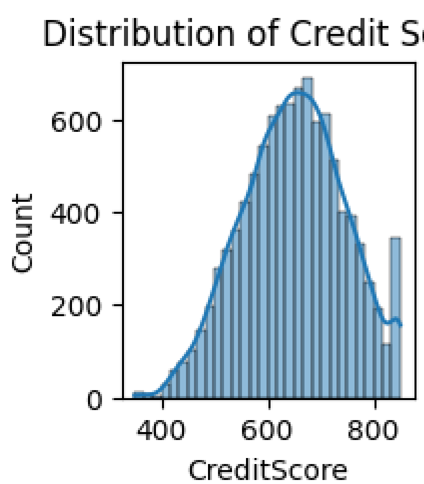
In [30]:
```
# Distribution of numerical features
plt.figure(figsize=(15, 10))
plt.subplot(2, 3, 1)
sns.histplot(df['Age'], bins=30, kde=True)
plt.title('Distribution of Age')
```

Out[30]:
```
Text(0.5, 1.0, 'Distribution of Age')
```



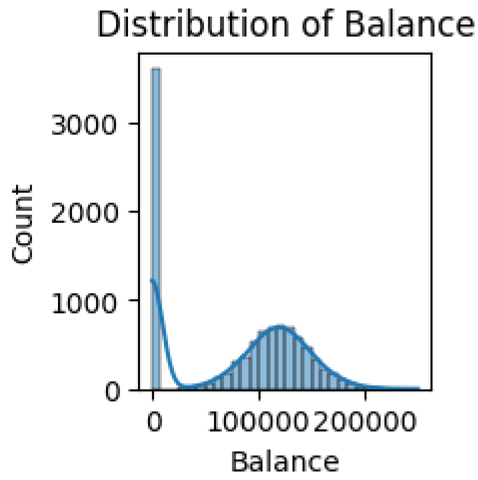
In [31]:
```
plt.subplot(2, 3, 2)
sns.histplot(df['CreditScore'], bins=30, kde=True)
plt.title('Distribution of Credit Score')
```

Out[31]: Text(0.5, 1.0, 'Distribution of Credit Score')



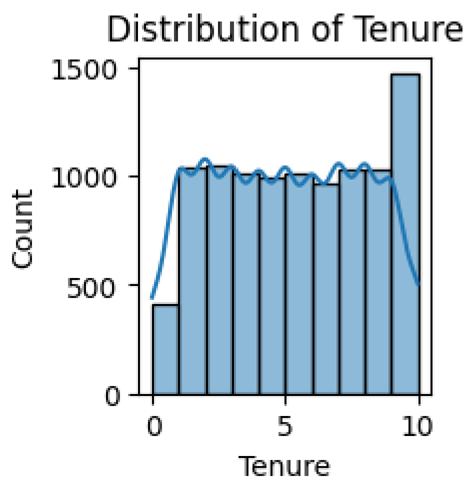
In [32]:
```
plt.subplot(2, 3, 3)
sns.histplot(df['Balance'], bins=30, kde=True)
plt.title('Distribution of Balance')
```

Out[32]: Text(0.5, 1.0, 'Distribution of Balance')

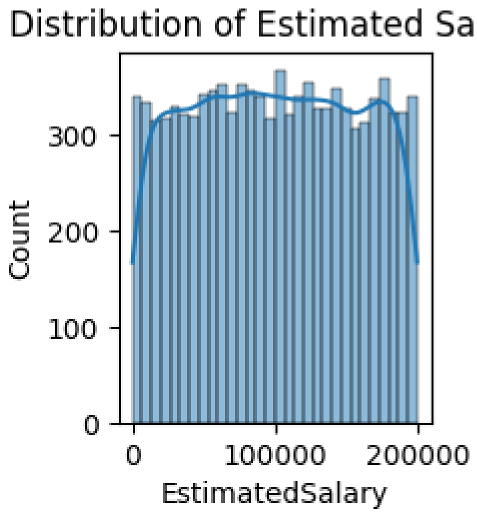


In [33]:
```
plt.subplot(2, 3, 4)
sns.histplot(df['Tenure'], bins=10, kde=True)
plt.title('Distribution of Tenure')
```

Out[33]:
```
Text(0.5, 1.0, 'Distribution of Tenure')
```



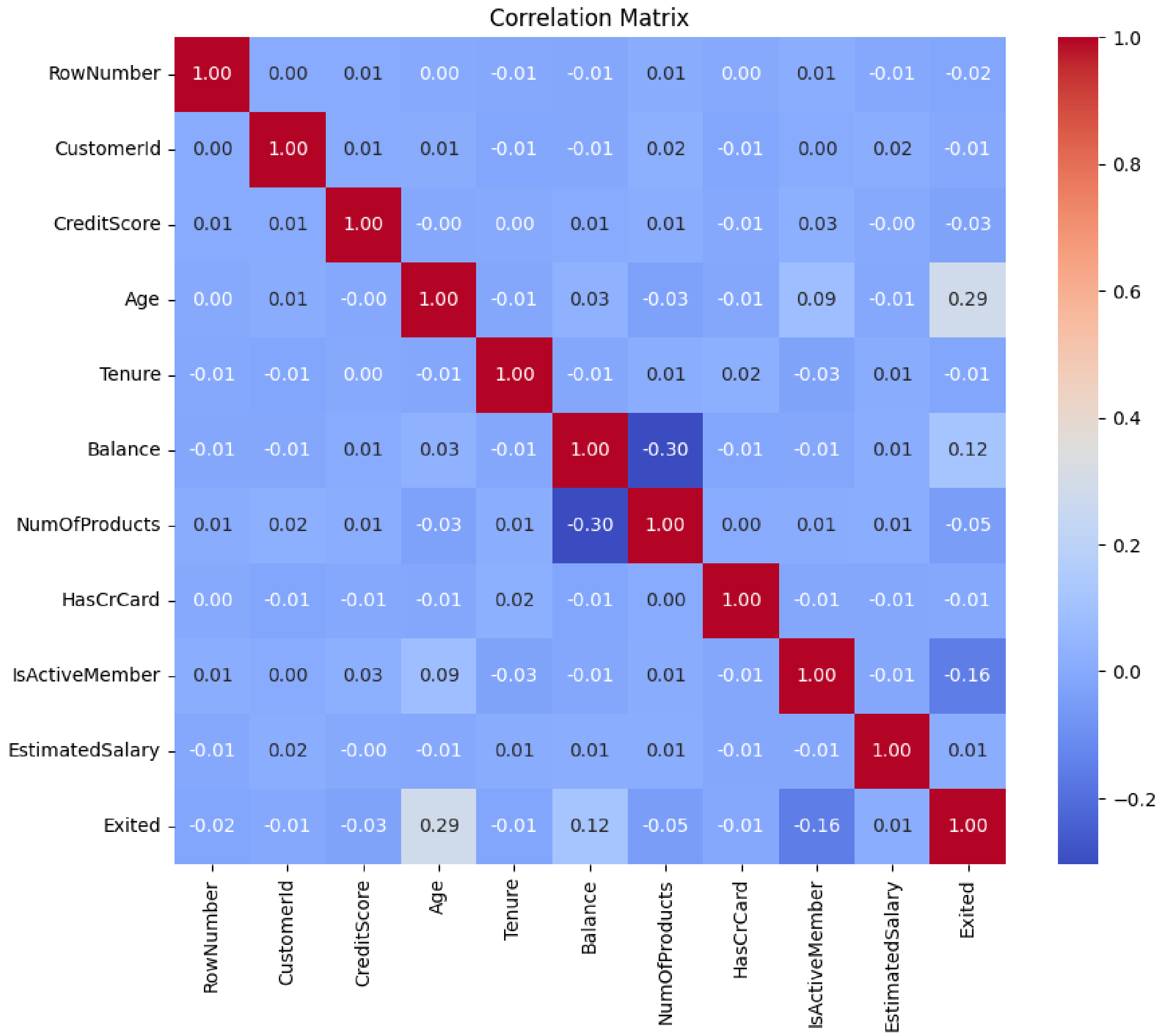
In [34]:
```
plt.subplot(2, 3, 5)
sns.histplot(df['EstimatedSalary'], bins=30, kde=True)
plt.title('Distribution of Estimated Salary')
plt.tight_layout()
plt.show()
```



In [35]:

```
# Correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix')
plt.show()
```

```
<ipython-input-35-01f47f28811d>:3: FutureWarning: The default value of numeric_only i
n DataFrame.corr is deprecated. In a future version, it will default to False. Select
only valid columns or specify the value of numeric_only to silence this warning.
  sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt='.2f')
```



Data Transformation Techniques

Imagine you have a dataset with information about bank customers, including their gender and the country they are from (categorical features). Machines often find it easier to work with numbers than words, so this code helps convert these categorical features into a numerical format.

The **pd.get_dummies() function** is used to create new columns for each category in the original categorical columns. For instance, it creates columns like 'Gender_Female', 'Gender_Male', 'Geography_France', 'Geography_Germany', 'Geography_Spain', assigning 1 or 0 based on the presence of that category.

Finally, the code reorganizes the columns to have a specific order, and the resulting DataFrame becomes a mix of original and new columns. This transformed dataset is now more suitable for machine learning algorithms that require numerical input. The .head() function is just used to show the first few rows of the transformed dataset for a quick look.

Encoding Categorical Features

In [44]:
```
df = pd.get_dummies(df, columns=['Gender','Geography'])
order = ['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts',
       'HasCrCard', 'IsActiveMember', 'EstimatedSalary', 'Gender_Female',
       'Gender_Male', 'Geography_France', 'Geography_Germany', 'Geography_Spain', 'Exi
df = df[order]
df.head()
```

Out[44]:

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
0	619	42	2	0.00	1	1	1	101348.88
1	608	41	1	83807.86	1	0	1	112542.58
2	502	42	8	159660.80	3	1	0	113931.57
3	699	39	1	0.00	2	0	0	93826.63
4	850	43	2	125510.82	1	1	1	79084.10

In [45]:
```
X = df.drop(columns=['Exited'])
y = df['Exited']
```

In [46]:
```
print(X.shape)
y.shape
```

```
(10000, 13)
```

Out[46]:
```
(10000,)
```

Generalization

Imagine you have a big list of customers with their locations and bank balances. This code helps organize this information by regions, like grouping people from the same city or country together.

So, it calculates the average bank balance for each group, creating a summary that's easier to understand. It's like saying, "On average, people in City A have this much money in their bank accounts, people in City B have this much, and so on."

Finally, the code displays this summarized information, making it easier to compare the average bank balances in different regions.

In [37]:
```
generalized_data = df.groupby('Geography')['Balance'].mean().reset_index()

# Display the generalized data
print("\nGeneralized Data:")
print(generalized_data)
```

```
Generalized Data:
  Geography        Balance
0    France   62092.636516
1   Germany  119730.116134
2     Spain   61818.147763
```

Normalization

Imagine you have a big list of information about customers, like their credit scores, ages, and more. This code is like organizing and adjusting these details to make them more comparable and easier to understand.

There are two methods used here:

Min-Max Normalization: It's like making sure all the numbers fit into a similar scale, so they are easy to compare. Think of it like converting scores from 1 to 100 into a scale from 0 to 1, where 0 is the lowest and 1 is the highest.

Z-Score Normalization (Standardization): This is another way of making things comparable. It's like adjusting the numbers to show how far each value is from the average. Imagine everyone's scores being adjusted so that they're like how many 'average' units away from the typical score.

So, the code takes these customer details and transforms them in these two ways, making the information more straightforward and comparable, just like making sure everyone speaks the same language when discussing their details.

In [38]:
```
# Assuming 'df' is your DataFrame and you want to normalize certain columns
columns_to_normalize = ['CreditScore', 'Age', 'Balance', 'NumOfProducts', 'EstimatedSa
```

In [39]:
```
# Min-Max Normalization
min_max_scaler = MinMaxScaler()
df_min_max_normalized = df.copy()
df_min_max_normalized[columns_to_normalize] = min_max_scaler.fit_transform(df[columns_
```

In [40]:
```
# Z-Score Normalization (Standardization)
z_score_scaler = StandardScaler()
df_z_score_normalized = df.copy()
df_z_score_normalized[columns_to_normalize] = z_score_scaler.fit_transform(df[columns_
```

In [41]:
```
# Display the original and normalized DataFrames
print("Original Data:")
print(df[columns_to_normalize].head())

print("\nMin-Max Normalized Data:")
print(df_min_max_normalized[columns_to_normalize].head())

print("\nZ-Score Normalized Data:")
print(df_z_score_normalized[columns_to_normalize].head())
```

```
Original Data:
   CreditScore  Age    Balance  NumOfProducts  EstimatedSalary
0          619   42       0.00              1        101348.88
1          608   41   83807.86              1        112542.58
2          502   42  159660.80              3        113931.57
3          699   39       0.00              2         93826.63
4          850   43  125510.82              1         79084.10

Min-Max Normalized Data:
   CreditScore       Age   Balance  NumOfProducts  EstimatedSalary
0        0.538  0.324324  0.000000       0.000000         0.506735
1        0.516  0.310811  0.334031       0.000000         0.562709
2        0.304  0.324324  0.636357       0.666667         0.569654
3        0.698  0.283784  0.000000       0.333333         0.469120
4        1.000  0.337838  0.500246       0.000000         0.395400

Z-Score Normalized Data:
   CreditScore       Age   Balance  NumOfProducts  EstimatedSalary
0    -0.326221  0.293517 -1.225848      -0.911583         0.021886
1    -0.440036  0.198164  0.117350      -0.911583         0.216534
2    -1.536794  0.293517  1.333053       2.527057         0.240687
3     0.501521  0.007457 -1.225848       0.807737        -0.108918
4     2.063884  0.388871  0.785728      -0.911583        -0.365276
```

In [ ]:

```
print(X.shape)
y.shape
```

Dimensionality Reduction

Principal Component Analysis

In [47]:

```
# Assuming 'X' is your feature matrix with the listed columns
selected_columns = ['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCr

# Extract the features (X) from the DataFrame
X = df[selected_columns]

# Standardize the features before applying PCA
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)

# Apply PCA with 2 components
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X_standardized)

# Create a DataFrame with the reduced features
df_reduced = pd.DataFrame(data=X_reduced, columns=['Principal Component 1', 'Principal

# Display the reduced features DataFrame
print("Reduced Features:")
print(df_reduced.head())
```

```
Reduced Features:
   Principal Component 1  Principal Component 2
0               0.137838               0.977294
1              -0.918809               1.307642
2               0.873885              -1.150707
3               1.256769               0.258700
4              -1.246896               1.325233
```

Imagine you have a lot of information about customers, like their credit score, age, and more. This code is like using magic glasses to simplify and highlight the most important information. It takes all the details about customers and transforms them into just two key things, like the main ingredients in a recipe.

These two things, called 'Principal Components,' capture the essence of the customer information. They are like a summary that keeps the crucial parts but makes everything much simpler. The code prints out these two key components, making it easier to see and understand the main patterns in the customer data. It's like turning a complex story into a short and sweet summary!

Feature Selection

In [52]:

```
import pandas as pd
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Assuming 'df' is your DataFrame and 'Exited' is the target variable
X = df.drop(columns=['Exited'])  # Features
y = df['Exited']  # Target

# Attribute Subset Selection using SelectKBest
selector = SelectKBest(f_classif, k=5)
X_selected = selector.fit_transform(X, y)

# Get the selected feature names
selected_feature_names = X.columns[selector.get_support()]

print(selected_feature_names)
```

```
Index(['Age', 'Balance', 'IsActiveMember', 'Gender_Female',
       'Geography_Germany'],
      dtype='object')
```

This code is all about finding the most important information from a dataset to help predict whether customers will leave a service or not. Imagine you have a lot of information about customers, like their age, how much money they have, and so on. We want to **figure out which pieces of information (features) are the most helpful in making predictions.**

The code uses a **technique called 'SelectKBest' to choose the top 5 most important features from all the available information**. It's like **selecting the most important ingredients to make a delicious dish**. In this case, the dish is a model (like a smart system) that can predict if a customer will leave the service. The **selected features are the most influential factors that contribute to making accurate predictions**. The code prints out the names of these selected features, so it's easier to understand what information is most important for predicting customer behavior.

Discretization

In [54]:

```
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
# Assuming 'df' is your DataFrame and 'Exited' is the target variable
X = df.drop(columns=['Exited'])  # Features
y = df['Exited']  # Target

# Discretization using KBinsDiscretizer
discretizer = KBinsDiscretizer(n_bins=5, encode='ordinal', strategy='uniform')
X_discretized = discretizer.fit_transform(X)

# Convert the discretized features back to a DataFrame
X_discretized_df = pd.DataFrame(X_discretized, columns=X.columns)

# Split the data into train and test sets for model evaluation
X_train, X_test, y_train, y_test = train_test_split(X_discretized_df, y, test_size=0.2

# Train a model using the discretized features
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Model Accuracy with Discretized Features:", accuracy)
```

```
Model Accuracy with Discretized Features: 0.841
```

This code is about training a machine learning model to predict whether customers will exit a service or not. The features used for prediction are first divided into five different intervals using a technique called discretization. Imagine sorting these features into five bins like sorting items into different shelves based on their size. Then, a model, specifically a RandomForest, is trained using these discretized features. This trained model is like a smart assistant that learns patterns from the past behavior of customers.

After the model is trained, it is tested on a set of data it has never seen before (like a final exam). The accuracy of the model, which indicates how well it predicts whether customers will exit or not, is then printed out. In simpler terms, this code is a part of building a smart system to predict if customers are likely to leave a service based on their behavior. The better the accuracy, the smarter and more reliable the system is at making predictions.

Model Training

Data Validation

In [55]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

In [56]:
```
# Model Training
lg = LogisticRegression()
rf = RandomForestClassifier(n_estimators=50, random_state=2)
gb = GradientBoostingClassifier(n_estimators=50, random_state=2)
```

In [57]:
```
clfs = {
    'lg': lg,
    'rf': rf,
    'gb': gb
}

def train_clfs_and_predict(clfs, X_train, X_test, y_train, y_test):
    acc = []
    prec = []
    recall = []
    f1 = []
    conf_mat = []

    for clf in clfs:
        model = clfs[clf]
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        acc.append(accuracy_score(y_test, y_pred))
        prec.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        f1.append(f1_score(y_test, y_pred))
        conf_mat.append(confusion_matrix(y_test, y_pred))

    return acc, prec, recall, f1, conf_mat
```

This code defines a few machine learning models (**logistic regression, random forest, and gradient boost**) and puts them into a **dictionary called clfs**. Then, there's a **function called train_clfs_and_predict that trains these models on a training dataset (X_train, y_train)** and **evaluates their performance on a testing dataset (X_test, y_test)**. It **computes and collects various metrics such as accuracy, precision, recall, F1 score, and confusion matrix for each model**. These **metrics help understand how well the models are making predictions and are crucial for comparing and selecting the best-performing model**. Overall, this code is part of the process to assess and choose the most effective machine learning model for a specific task.

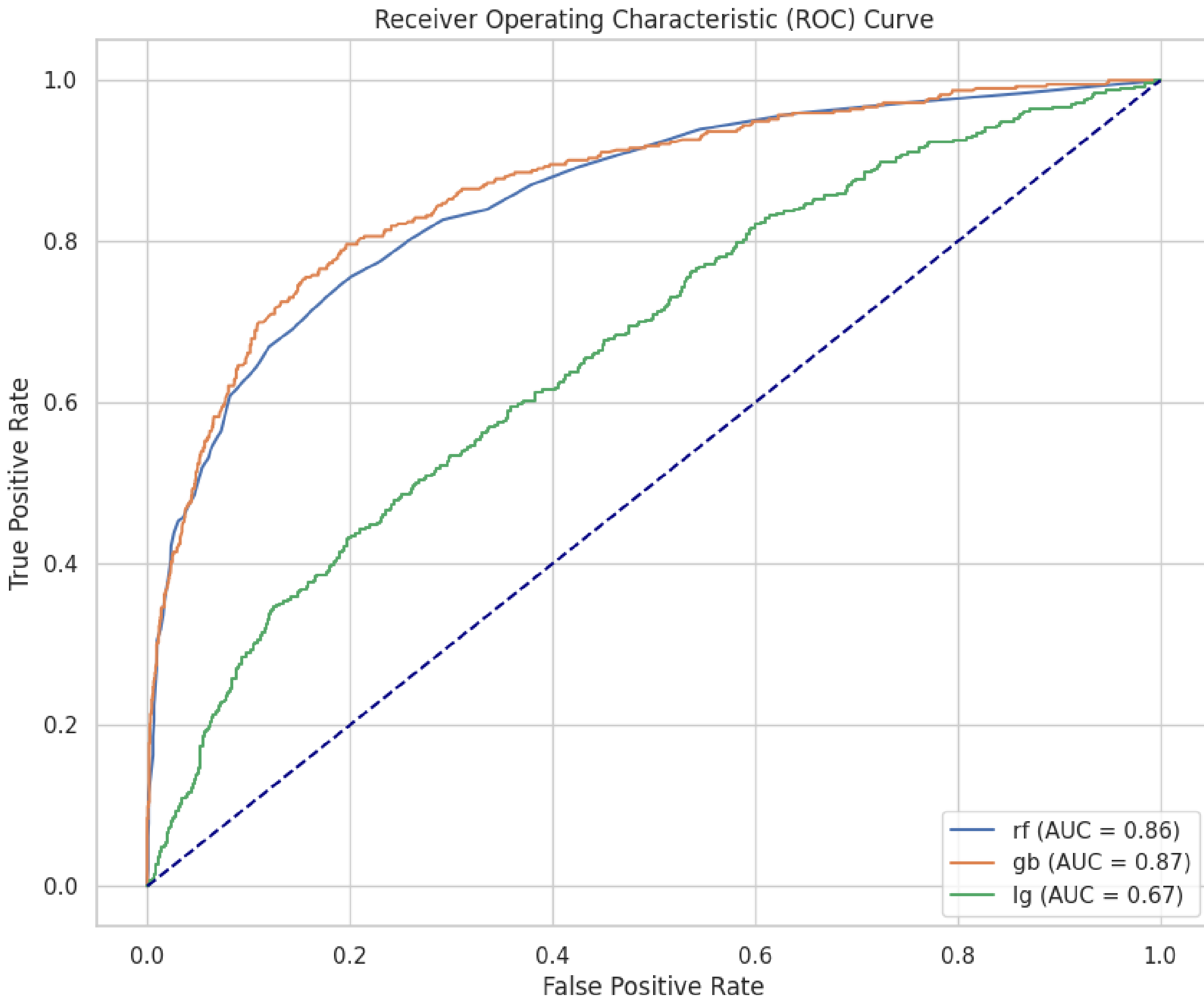
In [58]:
```
accuracy, precision, recall, f1, conf_mat = train_clfs_and_predict(clfs, X_train, X_te

performance = {
    'classifiers': list(clfs.keys()),
    'accuracy': accuracy,
    'precision': precision,
```

```python
    'recall': recall,
    'f1_score': f1,
    'confusion_matrix': conf_mat,
}

perf_df = pd.DataFrame(performance).sort_values(by='accuracy', ascending=False)
```

ROC-AUC Curve

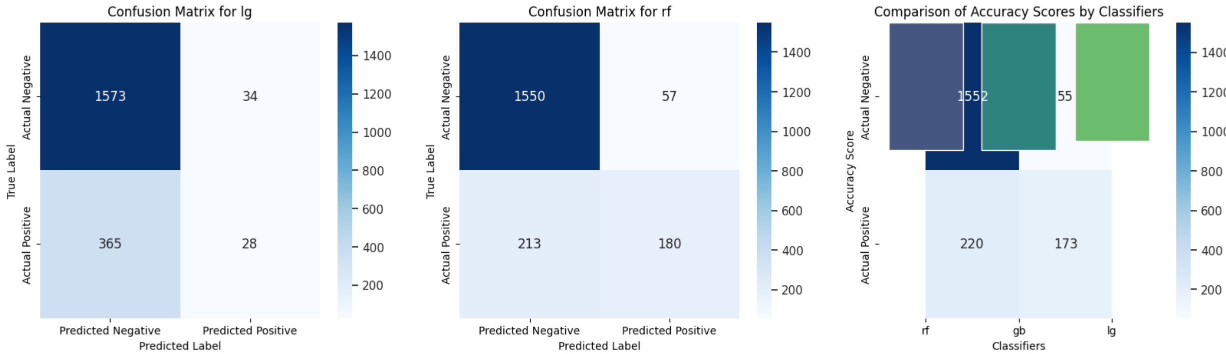
In [71]:

```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier

# Binarize the labels if you have a multi-class classification problem
y_bin = label_binarize(y_test, classes=[0, 1, 2])  # Adjust the classes based on your

# Initialize a figure for the ROC-AUC curves
plt.figure(figsize=(10, 8))

# For each classifier, calculate ROC curve and AUC
for i, classifier in enumerate(perf_df['classifiers']):
    model = clfs[classifier]
    y_score = model.predict_proba(X_test)[:, 1]

    # Compute ROC curve and ROC area
    fpr, tpr, _ = roc_curve(y_bin[:, 1], y_score)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.plot(fpr, tpr, label=f'{classifier} (AUC = {roc_auc:.2f})')

# Plot the random guessing line
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')

# Set labels and title
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```



In [59]:

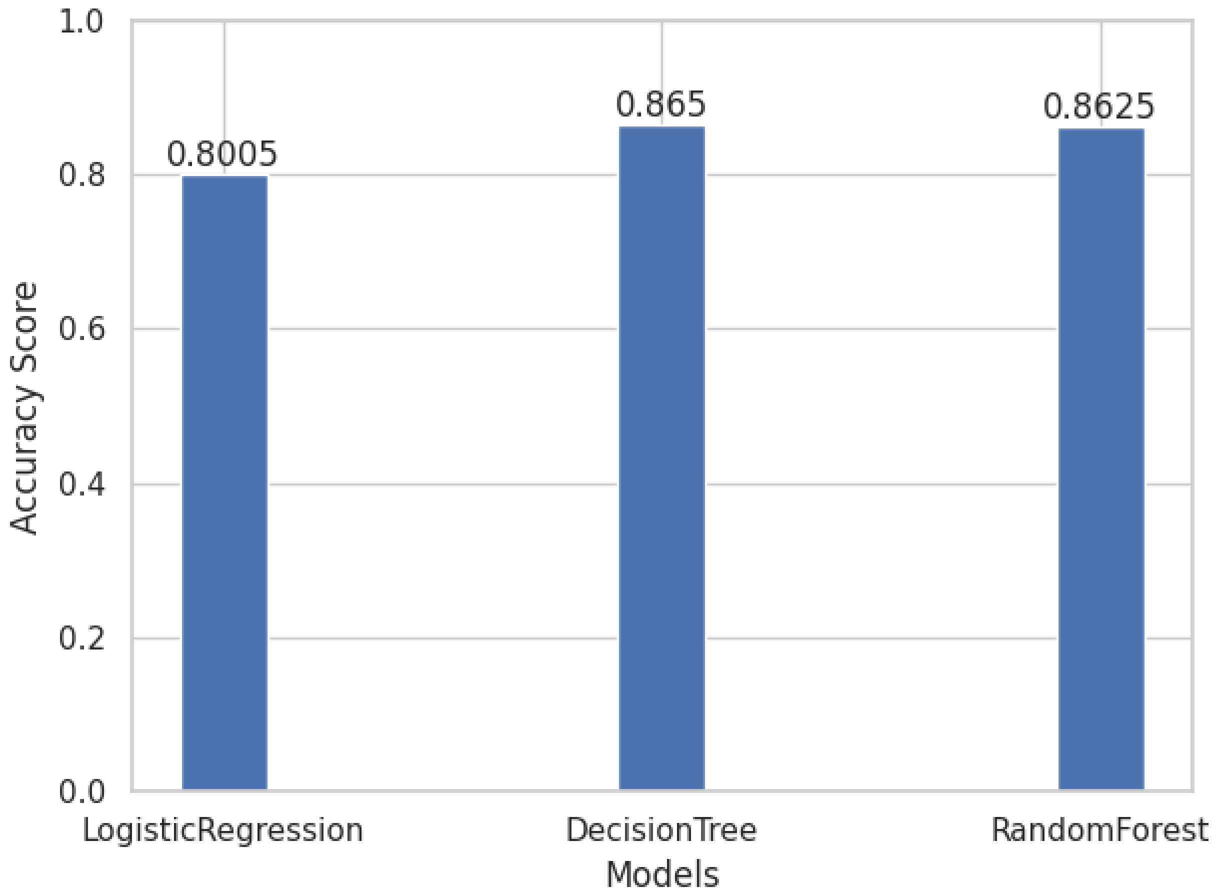
```
# Plotting confusion matrices of classifiers
num_classifiers = len(conf_mat)

fig, axes = plt.subplots(1, num_classifiers, figsize=(20, 5))

for i, (matrix, classifier) in enumerate(zip(conf_mat, list(clfs.keys()))):
    sns.set(font_scale=1)
    sns.heatmap(matrix, annot=True, fmt="d", cmap="Blues",
                xticklabels=["Predicted Negative", "Predicted Positive"],
                yticklabels=["Actual Negative", "Actual Positive"],
                ax=axes[i])
    axes[i].set_title(f"Confusion Matrix for {classifier}")
    axes[i].set_xlabel("Predicted Label")
    axes[i].set_ylabel("True Label")

# Comparison graph of accuracy scores
sns.set(style="whitegrid")
sns.barplot(x=perf_df.classifiers, y=perf_df.accuracy, palette="viridis")
plt.title("Comparison of Accuracy Scores by Classifiers")
plt.xlabel("Classifiers")
plt.ylabel("Accuracy Score")
plt.show()
```



This code creates two visualizations to assess the performance of different machine learning models. The first set of visualizations consists of confusion matrices for each model. A confusion matrix helps to understand how well a model is at predicting positive and negative outcomes. The second visualization is a bar plot comparing the accuracy scores of various models. Each bar represents the accuracy of a specific model, indicating how often the model's predictions are correct. These visualizations offer a clear and concise way to evaluate and compare the effectiveness of different machine learning classifiers.

In [70]:

```
Models = ['LogisticRegression', 'DecisionTree', 'RandomForest', 'KNeighbors', 'Gaussia
scores = accuracy  # Replace with your actual accuracy scores

fig, ax = plt.subplots()

x = np.arange(len(Models))

# Use the minimum length between Models and scores
min_length = min(len(Models), len(scores))

ax.bar(x[:min_length], scores[:min_length], width=0.2)
ax.set_xticks(x[:min_length])
ax.set_xticklabels(Models[:min_length])
ax.set_xlabel('Models')
ax.set_ylabel('Accuracy Score')
ax.set_ylim(0, 1.0)  # Adjust the y-axis limit based on your data range

for index, value in enumerate(scores[:min_length]):
    plt.text(x=index, y=value + 0.01, s=str(round(value, 5)), ha='center')

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



The code creates a visual representation of the performance of different machine learning models. The models include Logistic Regression, Decision Tree, Random Forest, K-Nearest Neighbors, and Gaussian Naive Bayes. Each bar in the plot represents the accuracy score of a specific model, showing how well each model performed in terms of making correct predictions. The higher the bar, the better the model accuracy. This type of visualization helps in easily comparing and understanding which machine learning model is more effective for a given task.

In [ ]:

In [ ]: