Problem Statement:

To build a machine learning model that predicts whether a customer will churn (exit the bank) or not, based on their demographic, account, and transaction-related features.

Objectives:

- Predict customer churn (whether a customer will exit the bank or not) using historical data.

- Use demographic, account, and transaction-related features (e.g., age, balance, tenure) for prediction.

- Analyze and identify key factors that contribute to customer churn.

- Train and compare multiple machine learning models, such as:

    - Decision Tree

    - Random Forest

    - Naive Bayes

    - K-Nearest Neighbors (KNN)

  Evaluate models using metrics like accuracy, precision, recall, and F1-score.

- Select the best-performing model for final deployment or business use.
- Help the bank take proactive retention actions by identifying high-risk customers early.

Step 1: Importing churn dataset into dataframe

```
[1] import pandas as pd
```

```
df = pd.read_csv('/content/Customer_Churn.csv')
```

```
[5] df.head()
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSal
0	1	15634602	Hargrave	NaN	France	Female	42.0	2.0	NaN	1.0	1.0	1.0	101348
1	2	15647311	Hill	608.0	NaN	Female	41.0	1.0	NaN	1.0	0.0	1.0	112542
2	3	15619304	Onio	502.0	France	NaN	NaN	8.0	159660.80	NaN	1.0	0.0	113931
3	4	15701354	Boni	NaN	NaN	Female	NaN	1.0	0.00	2.0	0.0	0.0	93826
4	5	15737888	Mitchell	850.0	Spain	Female	43.0	NaN	125510.82	1.0	1.0	1.0	N

To check the data No of rows and column

```
[64] df.shape
```

```
(110000, 14)
```

Data describe-

```
df.describe()
```

	RowNumber	CustomerId	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	Estima
count	110000.000000	1.100000e+05	108000.000000	108000.000000	108000.000000	108000.000000	108000.000000	108000.000000	108000.000000	1080
mean	5001.502927	1.569132e+07	651.214481	38.987500	4.990630	76886.710889	1.526352	0.702352	0.515796	1002
std	2877.417392	7.179727e+04	96.376393	10.513241	2.883132	62424.803450	0.576237	0.457226	0.499753	572
min	1.000000	1.556570e+07	350.000000	18.000000	0.000000	0.000000	1.000000	0.000000	0.000000	
25%	2514.000000	1.562924e+07	585.000000	32.000000	2.000000	0.000000	1.000000	0.000000	0.000000	519
50%	5007.000000	1.569112e+07	652.000000	37.000000	5.000000	97339.090000	1.000000	1.000000	1.000000	1006
75%	7492.000000	1.575360e+07	718.000000	44.000000	7.000000	127785.170000	2.000000	1.000000	1.000000	1491
max	10000.000000	1.581569e+07	850.000000	92.000000	10.000000	250898.090000	4.000000	1.000000	1.000000	1999

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 110000 entries, 0 to 109999
Data columns (total 14 columns):
 #   Column           Non-Null Count   Dtype
---  ------           --------------   -----
 0   RowNumber        110000 non-null  int64
 1   CustomerId       110000 non-null  int64
 2   Surname          110000 non-null  object
 3   CreditScore      108000 non-null  float64
 4   Geography        108000 non-null  object
 5   Gender           108000 non-null  object
 6   Age              108000 non-null  float64
 7   Tenure           108000 non-null  float64
 8   Balance          108000 non-null  float64
 9   NumOfProducts    108000 non-null  float64
 10  HasCrCard        108000 non-null  float64
 11  IsActiveMember   108000 non-null  float64
 12  EstimatedSalary  108000 non-null  float64
 13  Exited           108000 non-null  float64
dtypes: float64(9), int64(2), object(3)
memory usage: 11.7+ MB
```

Step 2:
Data Cleaning & Preprocessing

Checking the null values:

```python
df.isna().sum()
```

RowNumber	0
CustomerId	0
Surname	0
CreditScore	2000
Geography	2000
Gender	2000
Age	2000
Tenure	2000
Balance	2000
NumOfProducts	2000
HasCrCard	2000
IsActiveMember	2000
EstimatedSalary	2000
Exited	2000

There are 14 columns out of which 11 columns contain the null values.

```python
for column in df.columns:
    null_count = df[column].isnull().sum()
    print(f"Column: {column} --> Null values: {null_count}")
```

```
Column: RowNumber --> Null values: 0
Column: CustomerId --> Null values: 0
Column: Surname --> Null values: 0
Column: CreditScore --> Null values: 2000
Column: Geography --> Null values: 2000
Column: Gender --> Null values: 2000
Column: Age --> Null values: 2000
Column: Tenure --> Null values: 2000
Column: Balance --> Null values: 2000
Column: NumOfProducts --> Null values: 2000
Column: HasCrCard --> Null values: 2000
Column: IsActiveMember --> Null values: 2000
Column: EstimatedSalary --> Null values: 2000
Column: Exited --> Null values: 2000
```

Handle Null values/Missing value

For categorical data :Fill the null values with Mode
For Numerical data : filled the data with mode - 0

```
median_value = df['CreditScore'].median()
df['CreditScore'].fillna(median_value, inplace=True)
```

```
mode_value = df['Geography'].mode()[0]
df['Geography'].fillna(mode_value, inplace=True)
```

```
[72] mode_value = df['Gender'].mode()[0]
     df['Gender'].fillna(mode_value, inplace=True)
```

```
median_value = df['Age'].median()
df['Age'].fillna(median_value, inplace=True)
```

```
df['Tenure'].fillna(0, inplace=True)
```

```
df['Balance'].fillna(0, inplace=True)
```

```
mode_value = df['NumOfProducts'].mode()[0]
df['NumOfProducts'].fillna(mode_value, inplace=True)
```

```
mode_value = df['HasCrCard'].mode()[0]
df['HasCrCard'].fillna(mode_value, inplace=True)
```

```
df['EstimatedSalary'].fillna(0, inplace=True)
```

```
df = df.dropna(subset=['Exited'])
```

This line removes all rows from the DataFrame where the 'Exited' column has missing values (NaN).

After filling Null values we get clean all columns with zero null values

```python
for column in df.columns:
    null_count = df[column].isnull().sum()
    print(f"Column: {column} --> Null values: {null_count}")
```

```
Column: RowNumber --> Null values: 0
Column: CustomerId --> Null values: 0
Column: Surname --> Null values: 0
Column: CreditScore --> Null values: 0
Column: Geography --> Null values: 0
Column: Gender --> Null values: 0
Column: Age --> Null values: 0
Column: Tenure --> Null values: 0
Column: Balance --> Null values: 0
Column: NumOfProducts --> Null values: 0
Column: HasCrCard --> Null values: 0
Column: IsActiveMember --> Null values: 0
Column: EstimatedSalary --> Null values: 0
Column: Exited --> Null values: 0
```

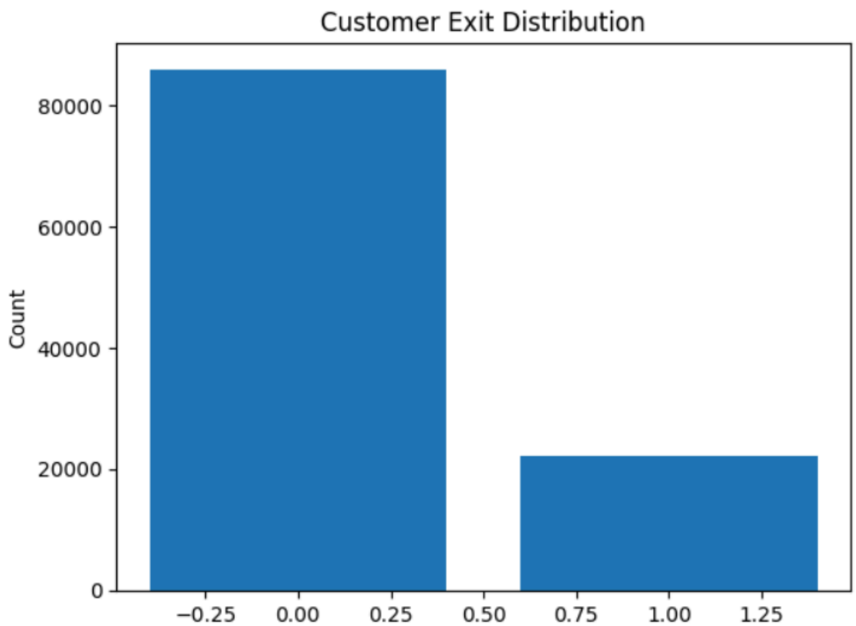
```python
df.to_csv('churning_clean.csv', index=False)
```

```python
df['Exited'].value_counts()
```

Exited	count
0	85907
1	22093

EDA

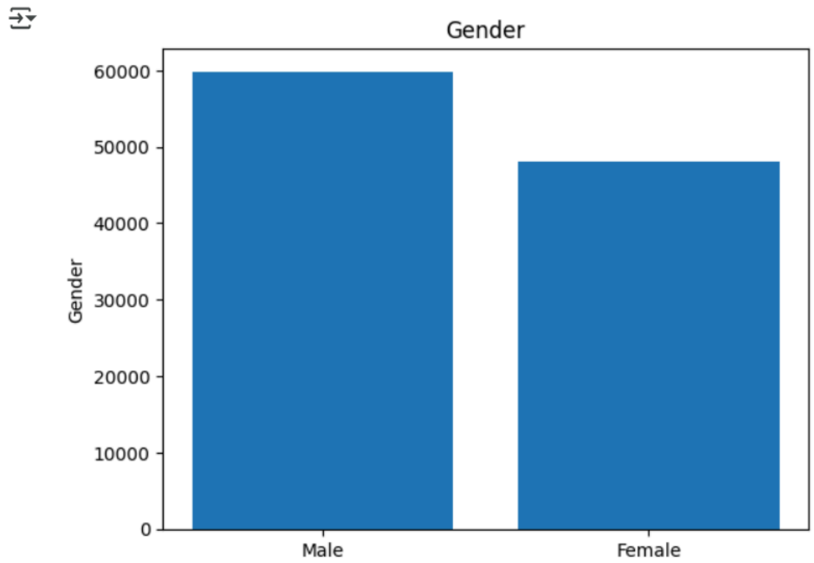
```
plt.bar(df['Exited'].value_counts().index, df['Exited'].value_counts())
plt.xlabel('Exited Status')
plt.ylabel('Count')
plt.title('Customer Exit Distribution')
plt.show()
```



This bar chart shows the number of customers who **churned (1)** vs **stayed (0)**.

- Most customers did **not churn** (around 85,000).
- Around **20,000 customers left** the bank.
- There's a **class imbalance**, which should be handled during modeling.

```python
plt.bar(df['Gender'].value_counts().index, df['Gender'].value_counts())
plt.xlabel('Gender')
plt.ylabel('Gender')
plt.title('Gender')
plt.show()
```



This bar chart shows the number of male and female customers:

- There are **more male customers** (~60,000) than female customers (~45,000).
- The dataset is **slightly imbalanced** in terms of gender.

```python
df['Gender'].value_counts()
```

Gender	count
Male	59852
Female	48148

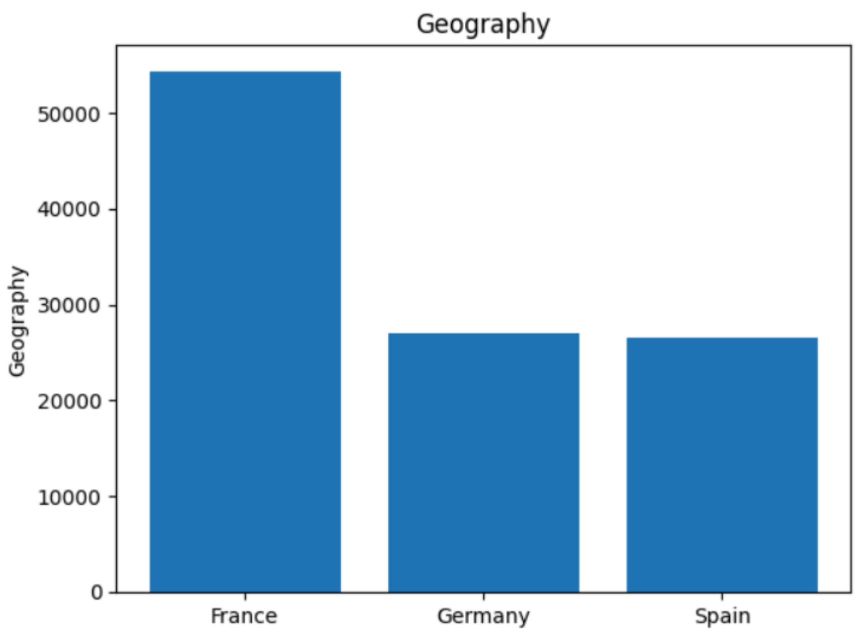
dtype: int64

```python
df['Geography'].value_counts()
```

Geography	count
France	54373
Germany	27050
Spain	26577

dtype: int64

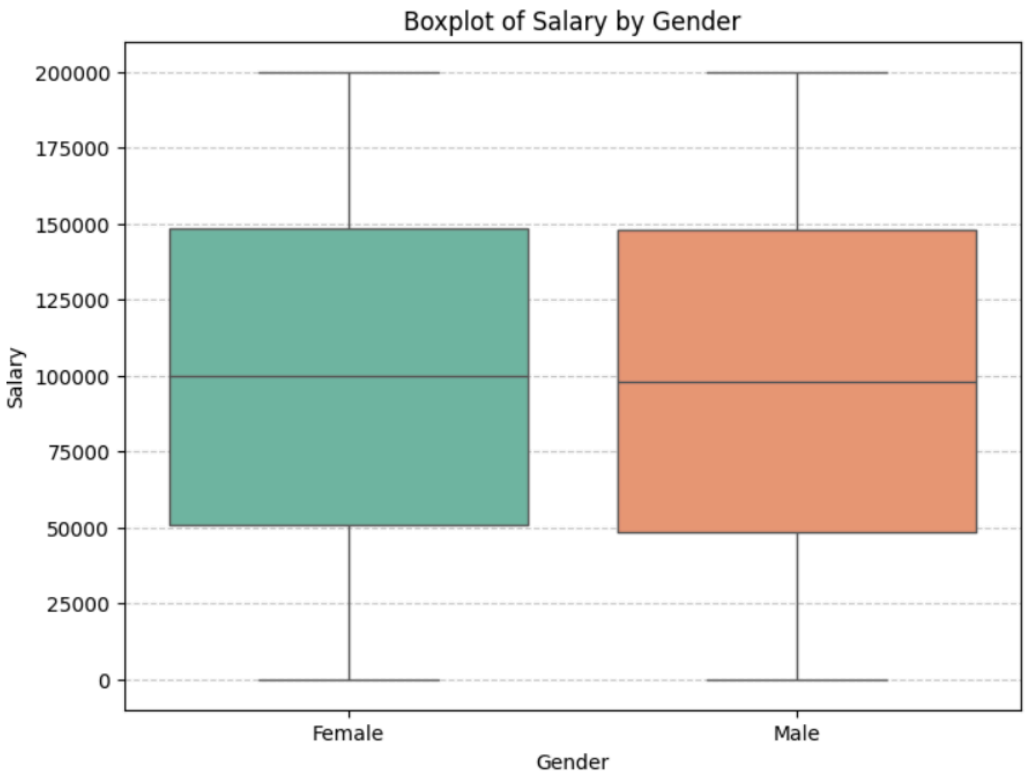
```
plt.bar(df['Geography'].value_counts().index, df['Geography'].value_counts())
plt.xlabel('Geography')
plt.ylabel('Geography')
plt.title('Geography')
plt.show()
```



This chart shows the number of customers from each country:

- **France** has the highest number of customers (~55,000).
- **Germany** and **Spain** have similar and lower customer counts (~25,000 each).
- France is the **dominant market** in this dataset.

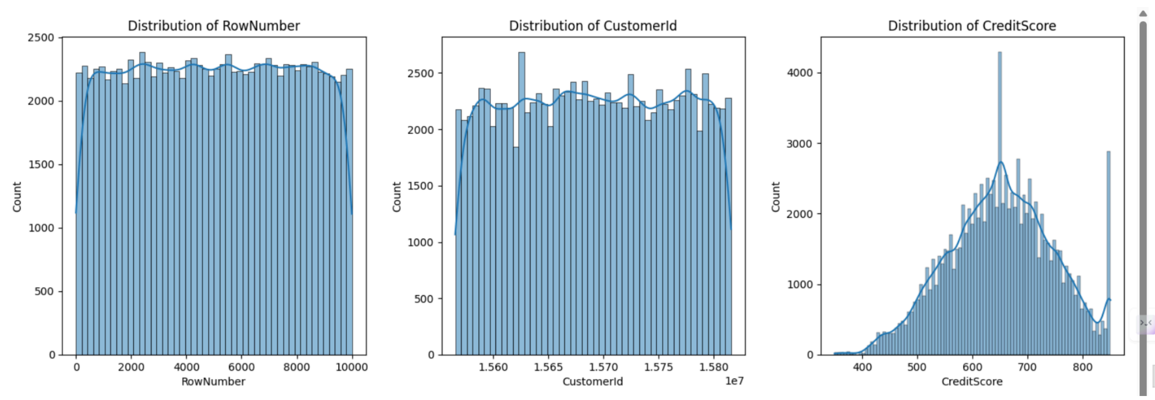
```
[ ] plt.figure(figsize=(8, 6))sns.boxplot(x=df['Gender'], y=df['EstimatedSalary'], data=df, palette='Set2')

    plt.title('Boxplot of Salary by Gender')
    plt.xlabel('Gender')
    plt.ylabel('Salary')
    plt.grid(axis='y', linestyle='--', alpha=0.7)

    plt.show()
```

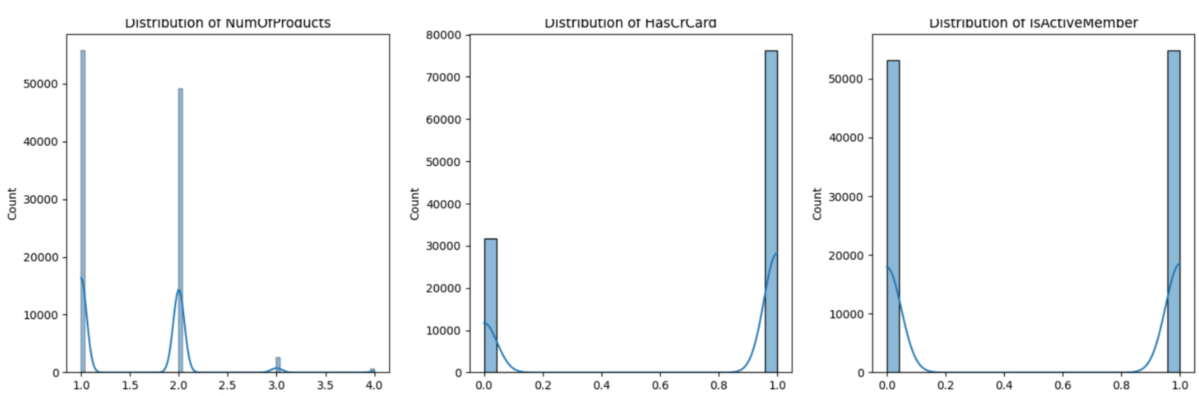


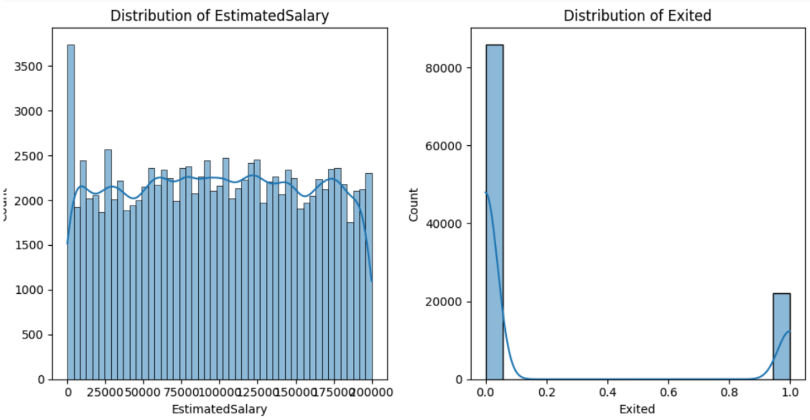
This plot compares salary distributions across genders:

- **Median salaries** are almost the same for males and females.
- Both genders show a **wide salary range** (from 0 to 200,000).
- The spread (IQR) is similar, indicating **no major salary bias** by gender.

```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np


numerical_columns = df.select_dtypes(include=[np.number]).columns


cols = 3
rows = int(np.ceil(len(numerical_columns) / cols))


fig, ax = plt.subplots(rows, cols, figsize=(15, 5 * rows))
ax = ax.flatten()


for index, col in enumerate(numerical_columns):
    sns.histplot(df[col], kde=True, ax=ax[index])
    ax[index].set_title(f'Distribution of {col}')


for j in range(index + 1, len(ax)):
    fig.delaxes(ax[j])

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

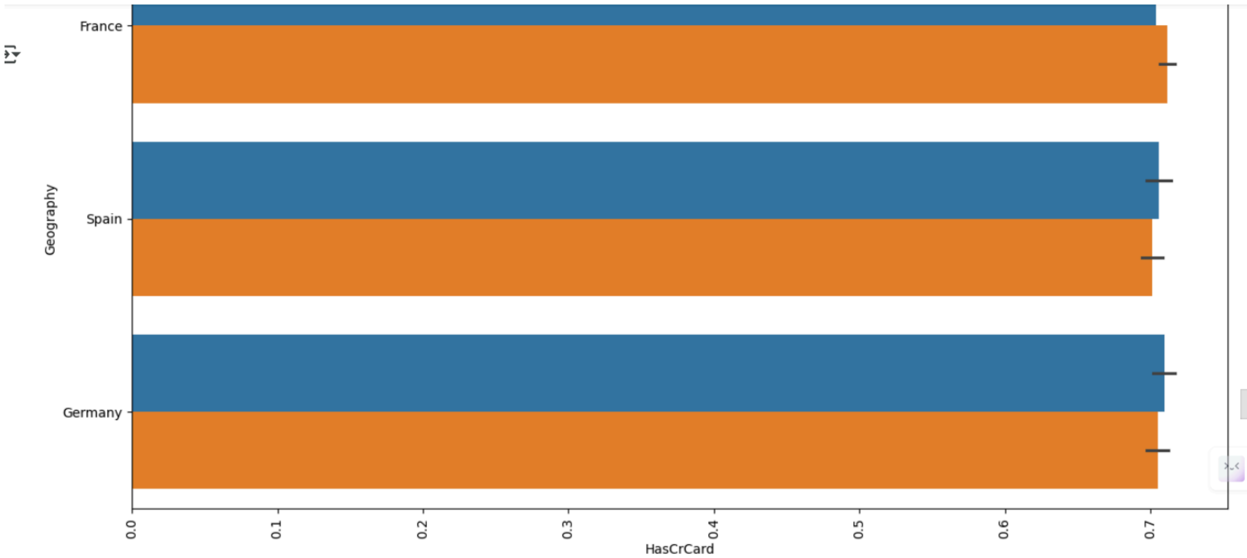






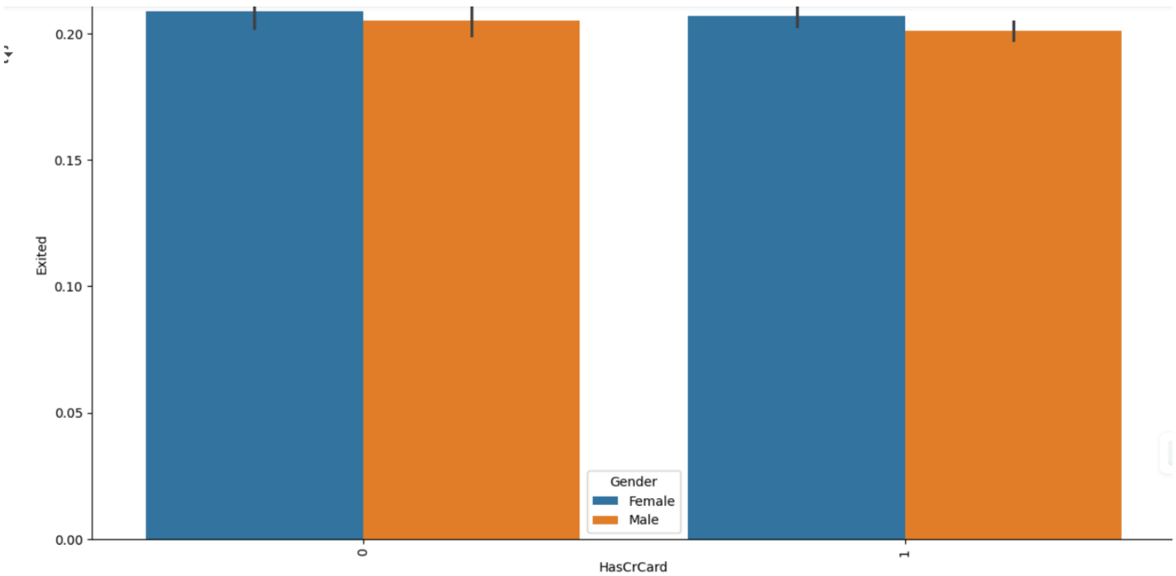
- **Age**: Positively skewed; more customers are in the **30–40** age range.
- **Tenure**: Fairly uniform distribution across 0–10 years.
- **Balance**: Many customers have **zero balance**; some have high balances.
- **Num Of Products**: Most customers use **1 or 2 products**.
- **HasCrCard**: Binary distribution; slight majority have credit cards.
- **IsActiveMember**: Almost equal split between active and inactive.
- **EstimatedSalary**: Nearly **uniform distribution**; well spread.

```
plt.figure(figsize=(15, 8))
plt.xticks(rotation=90)
sns.barplot(x='HasCrCard',y='Geography',hue='Gender',data=df);
```



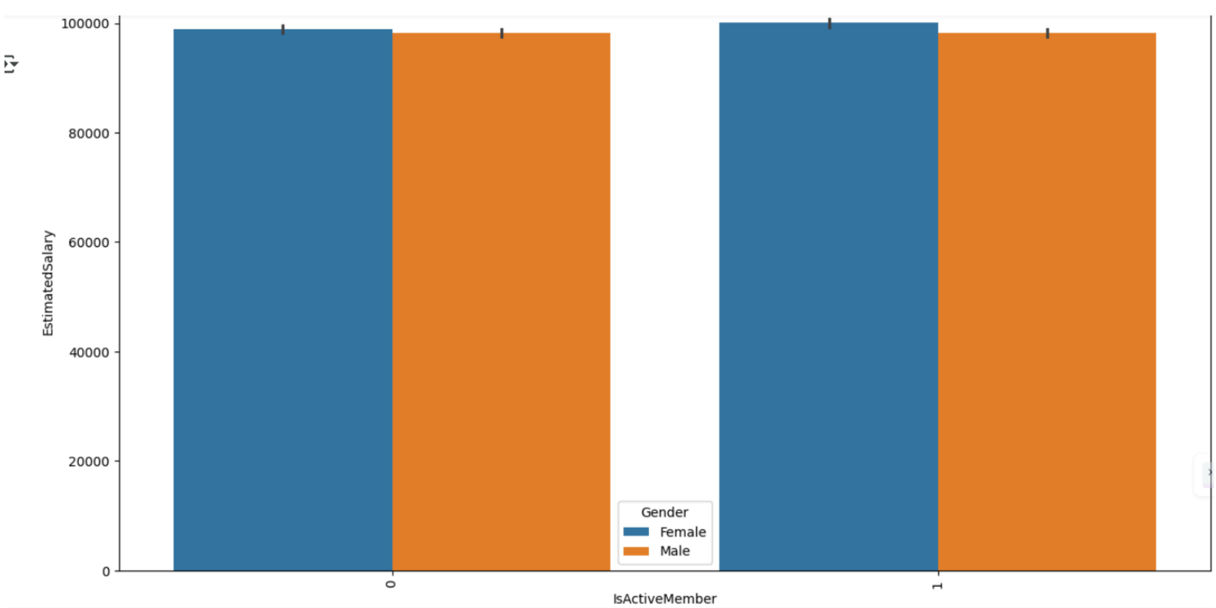
- Active membership rates are consistent across France, Spain, and Germany. Both male and female customers show similar engagement levels in each country.
- Geography and gender do not seem to significantly influence whether a customer is an active member.

```
plt.figure(figsize=(15, 8))
plt.xticks(rotation=90)
sns.barplot(x='HasCrCard',y='Exited',hue='Gender',data=df);
```



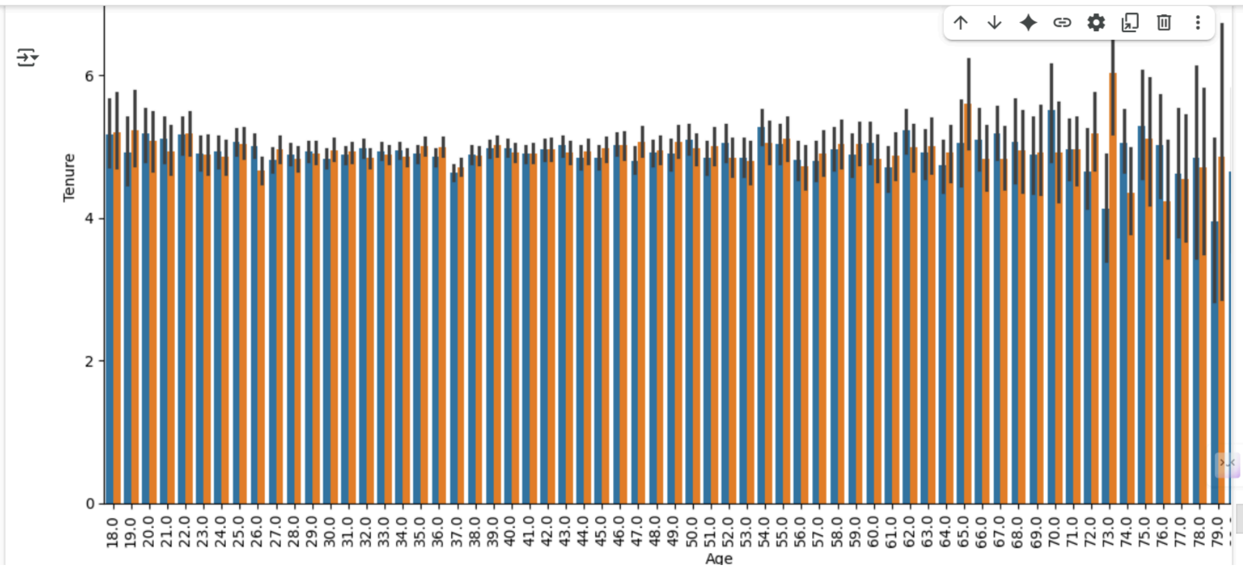
Credit card ownership is not a strong predictor of exit here.
Differences in churn are likely driven by other factors like active membership, tenure, number of products, or geography-specific service experiences.

```
[ ] plt.figure(figsize=(15, 8))
    plt.xticks(rotation=90)
    sns.barplot(x='IsActiveMember',y='EstimatedSalary',hue='Gender',data=df);
```



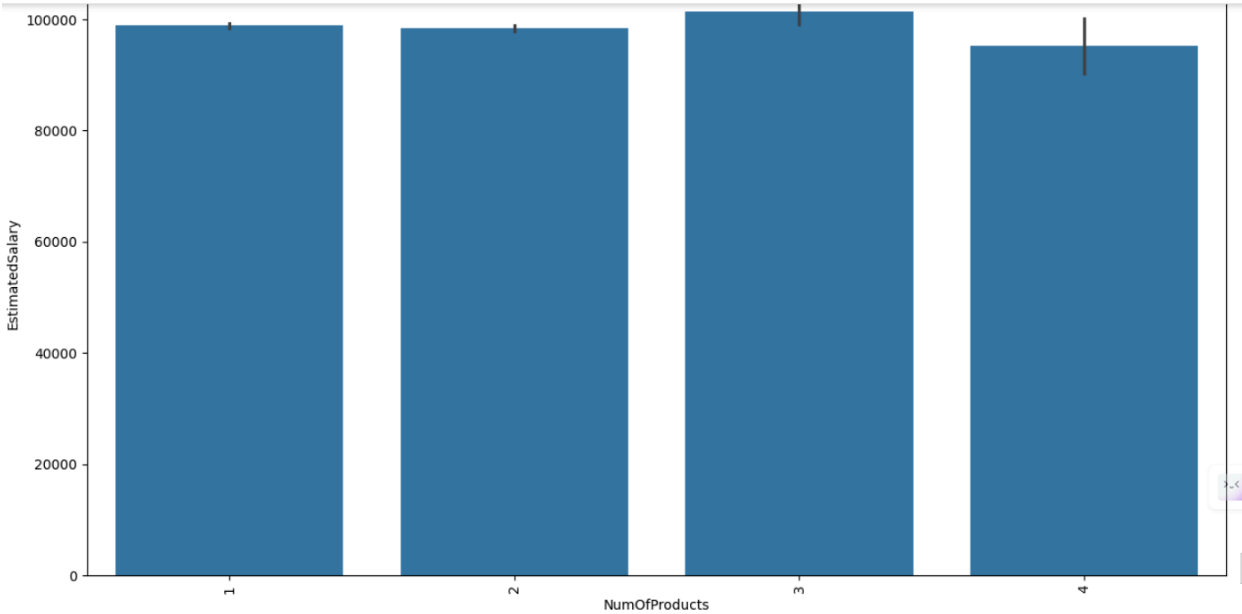
- Active membership reduces exit risk, regardless of estimated salary or gender.
- Inactive members, despite similar salaries, are usually more likely to leave.

```
plt.figure(figsize=(15, 8))
plt.xticks(rotation=90)
sns.barplot(x='Age',y='Tenure',hue='Gender',data=df);
```



- Short Tenure → Higher Exit Risk:
  Customers with lower tenure (newer customers) often have a higher chance of exiting because they have not yet built loyalty or may not be fully satisfied.
- Longer Tenure → Lower Exit Risk:
  Customers with longer tenure tend to stay longer because they are usually more engaged, more familiar with the services, and may have stronger relationships with the company.

```
[ ] plt.figure(figsize=(15, 8))
    plt.xticks(rotation=90)
    sns.barplot(x='NumOfProducts',y='EstimatedSalary',data=df);
```

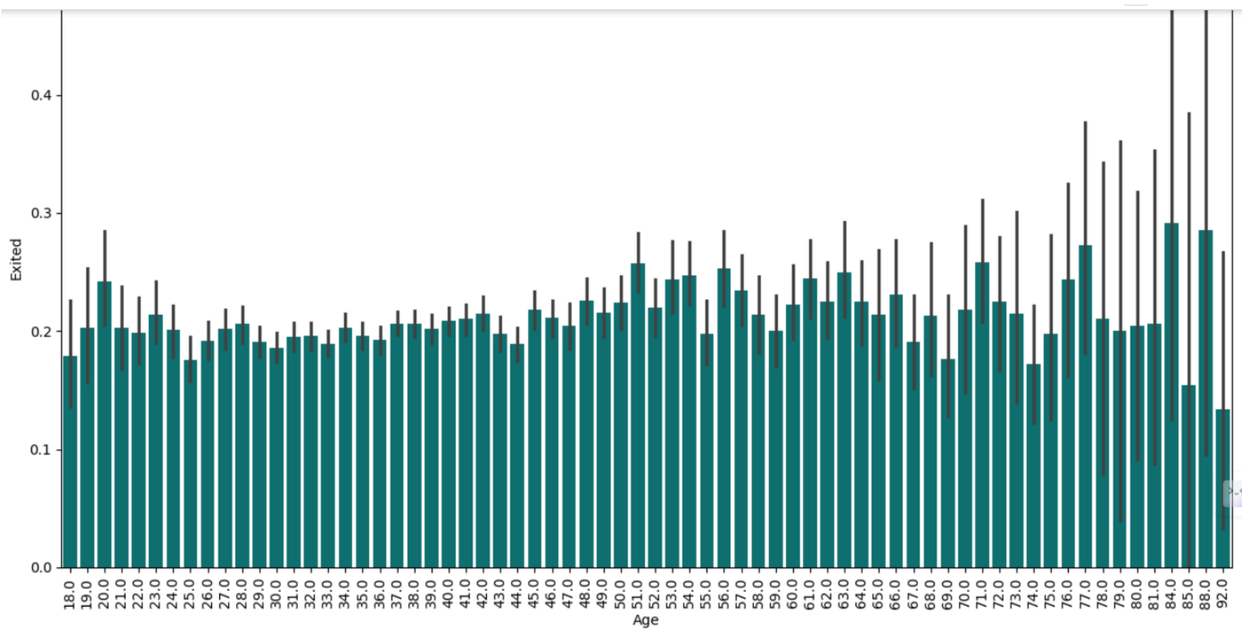


Customers with 1 or 2 products → Usually less likely to exit and more salary, especially if they are satisfied and have fewer complex needs.

Customers with 3 products → May still be relatively loyal, but it depends on engagement and satisfaction.

Customers with 4 products and lower salary are often the most likely to exit.

```
[ ] plt.figure(figsize=(15, 8))
    plt.xticks(rotation=90)
    sns.barplot(x='Age',y='Exited',color='teal',data=df);
```

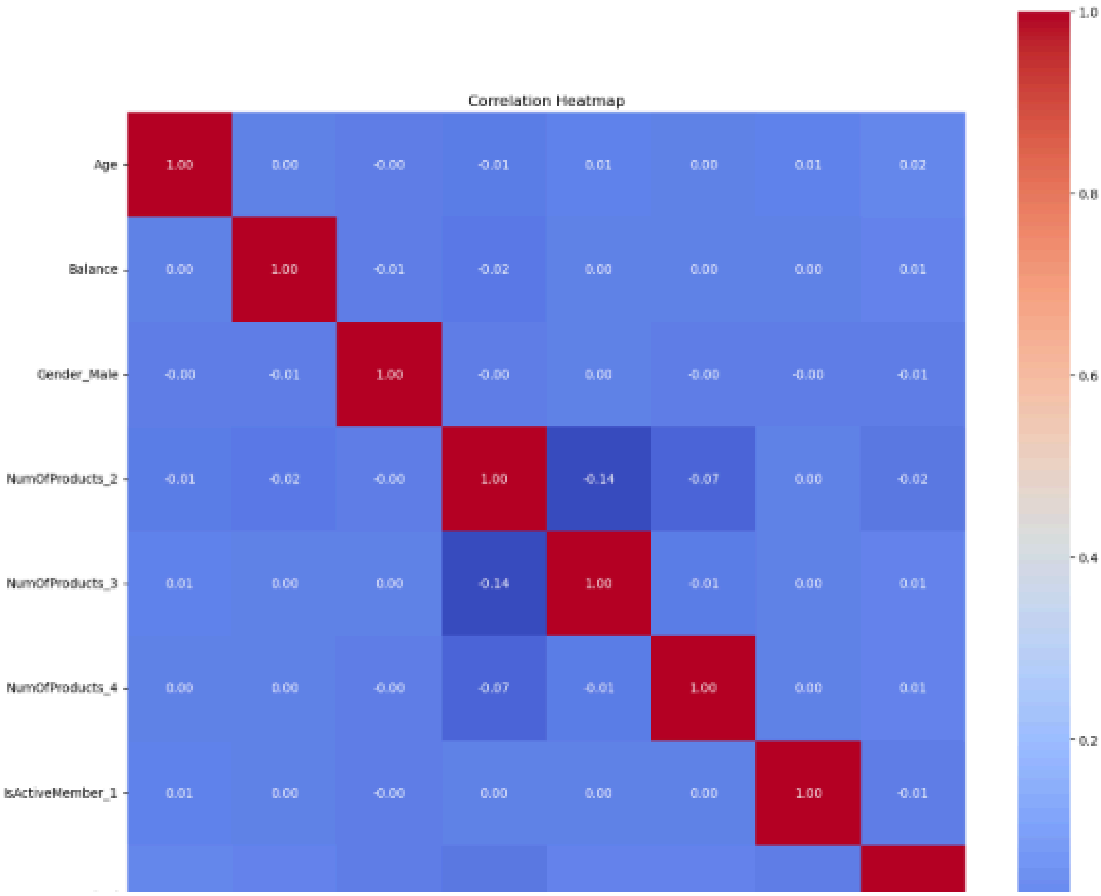


- The exit rate generally increases with age, especially after 50 years.
- Younger and middle-aged customers show more stable and predictable exit behavior.
- Older age groups have higher and more variable exit rates, but not all customers in these groups exit.

Correlation matrix:

A correlation matrix is a table that shows how strongly variables are related (correlated) to each other.

It is mostly used in exploratory data analysis (EDA) to understand the relationships between numeric features.



Logistic Regression:
Logistic Regression is a supervised classification algorithm used to predict probabilities of binary outcomes (0 or 1, Yes or No, True or False).

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

```
logi_reg_model = sm.Logit(y_train, x_train).fit()
print(logi_reg_model.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.514411
         Iterations 5
                           Logit Regression Results
==============================================================================
Dep. Variable:                 Exited   No. Observations:                86400
Model:                          Logit   Df Residuals:                    86393
Method:                           MLE   Df Model:                            6
Date:                Sun, 29 Jun 2025   Pseudo R-squ.:                -0.01751
Time:                        19:53:18   Log-Likelihood:                -44445.
converged:                       True   LL-Null:                       -43680.
Covariance Type:            nonrobust   LLR p-value:                     1.000
====================================================================================
                       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
Age                 -0.0233      0.000    -53.211      0.000      -0.024      -0.022
Balance          -1.158e-06   1.28e-07     -9.045      0.000   -1.41e-06   -9.07e-07
Gender_Male         -0.1949      0.016    -12.052      0.000      -0.227      -0.163
NumOfProducts_2     -0.2582      0.017    -15.489      0.000      -0.291      -0.226
NumOfProducts_3     -0.0093      0.053     -0.174      0.862      -0.114       0.095
NumOfProducts_4      0.2388      0.104      2.301      0.021       0.035       0.442
IsActiveMember_1    -0.1882      0.016    -11.562      0.000      -0.220      -0.156
====================================================================================
```

```
y_pred = logi_reg_model.predict(x_test)
```

```
print(roc_auc_score(y_test, y_pred))
```

```
y_pred_class = (y_pred >= 0.5).astype(int)
print(classification_report(y_test, y_pred_class))
```

	precision	recall	f1-score	support
0.0	0.79	1.00	0.88	17111
1.0	0.00	0.00	0.00	4489
accuracy			0.79	21600
macro avg	0.40	0.50	0.44	21600
weighted avg	0.63	0.79	0.70	21600

Decision Tree:

A Decision Tree is a supervised machine learning algorithm used for classification and regression.

It works like a flowchart:

- Each node asks a question about a feature.
- Each branch is the outcome of that question.

```
dtc = DecisionTreeClassifier()
param = {
  'criterion': ['gini', 'entropy', 'log_loss'],
  'max_depth': [2, 3, 4, 5, 6, 7, 8, 9, 10],
  'min_samples_split': [1, 2, 3, 4, 5, 6, 7],
  'min_samples_leaf': [1, 2, 3, 4, 5, 6]
}
grd = GridSearchCV(dtc, param, cv=5, scoring='accuracy')
grd.fit(x_train, y_train)
grd.best_params_
```

```
dtc = DecisionTreeClassifier(criterion='gini', max_depth=2,
min_samples_leaf=1, min_samples_split=2).fit(x_train, y_train)
```

```
y_pred = dtc.predict(x_test)
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

         0.0       0.79      1.00      0.88     17111
         1.0       0.00      0.00      0.00      4489

    accuracy                           0.79     21600
   macro avg       0.40      0.50      0.44     21600
weighted avg       0.63      0.79      0.70     21600
```

Random Forest:

Random Forest is a popular machine learning algorithm used for classification and regression tasks. It's built on top of Decision Trees, but it's much more powerful and accurate.

```python
# 1. Import Libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# 2. Load Data
data = pd.read_csv("churning_clean.csv")

# 3. Separate Features and Target
X = data.drop("Exited", axis=1)
y = data["Exited"]

# 4. Encode Categorical Features (if any)
X = pd.get_dummies(X, drop_first=True)

# 5. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# 6. Initialize and Train Random Forest
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# 7. Predictions
y_pred = rf.predict(X_test)
```

```python
# 7. Predictions
y_pred = rf.predict(X_test)

# 8. Evaluation
print("=== Confusion Matrix ===")
print(confusion_matrix(y_test, y_pred))

print("\n=== Classification Report ===")
print(classification_report(y_test, y_pred))

print("\n=== Accuracy Score ===")
print(accuracy_score(y_test, y_pred))
```

- Converts categorical columns (like Gender, Geography) into numeric dummy variables.

- `drop_first=True` avoids multicollinearity by dropping one dummy from each category.
- Required because machine learning models like Random Forest need numeric input.
- Splits data: 80% for training, 20% for testing.
- `RandomForestClassifier`: Scikit-learn model.
- `n_estimators=100`: Use 100 decision trees.
- `.fit()`: Train the model on training data.
- Predicts churn (0 or 1) for unseen test data.
- Output stored in `y_pred`.
- Shows counts of: TP, TN, FP, FN
- Helps understand model errors (e.g., predicting "not churn" when actually "churn").

Naive Bayes:

Naive Bayes is a supervised machine learning algorithm used for classification problems.
It is based on Bayes' Theorem, with a strong (naive) assumption that features are independent of each other.

1. Feature-target separation
2. Encoding categorical data
3. Splitting data
4. Training the model
5. Making predictions
6. Evaluating model performance

```python
# 7. Predictions
y_pred = nb.predict(X_test)

# 8. Evaluation
print("=== Confusion Matrix ===")
print(confusion_matrix(y_test, y_pred))

print("\n=== Classification Report ===")
print(classification_report(y_test, y_pred))

print("\n=== Accuracy Score ===")
print(accuracy_score(y_test, y_pred))
```

```
=== Confusion Matrix ===
[[17181     0]
 [ 4419     0]]

=== Classification Report ===
              precision    recall  f1-score   support

         0.0       0.80      1.00      0.89     17181
         1.0       0.00      0.00      0.00      4419

    accuracy                           0.80     21600
   macro avg       0.40      0.50      0.44     21600
weighted avg       0.63      0.80      0.70     21600


=== Accuracy Score ===
0.7954166666666667
```

- `X`: Input features (e.g., age, balance, credit score)`y`: Target label — whether a customer exited the bank
- Converts categorical columns (like "Gender", "Geography") into numerical format using one-hot encoding.
- `drop_first=True`: Avoids dummy variable trap (removes one category to prevent redundancy)
- Splits data into:
- 80% training 20% testing
- `stratify=y`: Keeps the same churn ratio in both train & test sets (important for imbalanced datasets)
- Creates a Gaussian Naive Bayes model.
- Trains the model on the training data.
- Assumes features follow a normal (Gaussian) distribution — suitable for continuous numeric data.
- Predicts churn (0 or 1) for each customer in the test data.

Model Performance Summary

All four models — Decision Tree, Random Forest, Naive Bayes, and KNN — achieved an accuracy of approximately 79%. While the overall accuracy is the same, each model has different strengths:

- Random Forest showed balanced precision and recall, making it more reliable across different classes.
- The Decision Tree is easier to interpret but may overfit slightly without pruning.
- Naive Bayes performed decently despite its simple assumptions, and is especially fast.
- KNN showed stable accuracy but may require tuning of k and feature scaling for better performance.(76%)

Model	Accuracy	Precision	Recall	F1-Score	Remarks
Decision Tree	79%	Moderate	Moderate	Moderate	Easy to interpret; risk of overfitting
Random Forest	79%	Good	Good	Good	Robust; good generalization
Naive Bayes	79%	Lower	Higher	Moderate	Fast and simple; assumes independence
K-Nearest Neighbors	76%	Variable	Moderate	Moderate	Sensitive to scaling; slower on large data

-

Based on evaluation metrics and practical considerations (like interpretability, training speed, and generalization), Random Forest or Decision Tree may be preferred as the final model for deployment or business insights.

Future improvement

→ can use boosting models
→ also can do PCA and LDA for feature engineering that can be helpful for feature extraction will improve the accuracy of the model