Bank Transaction Fraud Detection

1.Overview of Problem Statement :

The Importance of Fraud Detection in Banking

Fraud detection in banking is essential for ensuring the security and integrity of financial transactions. With the rise of digital transactions, fraudulent activities have become more sophisticated, posing significant risks to individuals and financial institutions. Identifying fraudulent transactions requires advanced analytical techniques that can detect unusual patterns and anomalies in transaction data. Machine learning models play a crucial role in this process by analyzing vast amounts of transaction records to distinguish between legitimate and fraudulent activities. Beyond its direct impact on financial security, effective fraud detection helps build trust in banking systems, prevents financial losses, and enhances regulatory compliance. By leveraging data-driven approaches, fraud detection systems can continuously evolve, adapting to new fraud tactics and improving overall financial safety.

2.**Objective** :

Develop a machine learning model to classify bank transactions as fraudulent or legitimate using features derived from transaction details, customer information, and account activity. The primary goal is to build an accurate and efficient fraud detection system that can identify suspicious transactions in real-time, thereby helping financial institutions minimize financial losses and enhance security.

3.Data Description :

The Bank Transaction Fraud Detection dataset contains various attributes that provide detailed insights into financial transactions, customer details, and transaction behaviors. These features help in identifying fraudulent activities and improving financial security.

Below are the key columns in the dataset:

Customer_ID: A unique identifier assigned to each customer within the bank's system.

Customer_Name: The name of the customer initiating the transaction.

Gender: The gender of the customer (e.g., Male, Female, Other).

Age: The age of the customer at the time of the transaction.

State: The state where the customer resides.

City: The city where the customer lives.

Bank_Branch: The specific bank branch where the customer holds an account.

Account_Type: The type of account owned by the customer (e.g., Savings, Checking).

Transaction_ID: A unique identifier for each transaction.

Transaction_Date: The date on which the transaction occurred.

Transaction_Time: The exact time the transaction was initiated.

Transaction_Amount: The monetary value of the transaction.

Merchant_ID: A unique identifier for the merchant involved in the transaction.

Transaction_Type: The nature of the transaction (e.g., Withdrawal, Deposit, Transfer).

Merchant_Category: The category of the merchant (e.g., Retail, Online, Travel).

Account_Balance: The balance of the customer's account after the transaction.

Transaction_Device: The device used by the customer to perform the transaction (e.g., Mobile, Desktop).

Transaction_Location: The geographical location (latitude, longitude) where the transaction took place.

Device_Type: The type of device used for the transaction (e.g., Smartphone, Laptop).

Is_Fraud: A binary indicator (1 or 0) showing whether the transaction is fraudulent (1) or legitimate (0).

Transaction_Currency: The currency used for the transaction (e.g., USD, EUR).

Customer_Contact: The contact number of the customer.

Transaction_Description: A brief description of the transaction (e.g., purchase, transfer).

Customer_Email: The email address associated with the customer's account.

These column descriptions provide a clear understanding of the data structure, which will be used for fraud detection analysis. By leveraging machine learning techniques on this dataset, fraudulent activities can be identified and prevented effectively, ensuring financial security and trust in banking systems.

4.**Data Collection** :

Source : Kaggle

Dataset link : https://www.kaggle.com/datasets/marusagar/bank-transaction-fraud-detection

5.Data Preprocessing - Data Cleaning

Importing Necessary Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardS
from imblearn.combine import SMOTETomek
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingCla
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, classification_report, confu
from sklearn.feature_selection import SelectFromModel
from xgboost import XGBClassifier
```

Loading the dataset

```
df = pd.read_csv("/content/Bank_Transaction_Fraud_Detection.csv")
df.head() # display only the first five rows
```

	Customer_ID	Customer_Name	Gender	Age	State	Cit
0	d5f6ec07-d69e-4f47-b9b4-7c58ff17c19e	Osha Tella	Male	60	Kerala	Thiruvananthapurar
1	7c14ad51-781a-4db9-b7bd-67439c175262	Hredhaan Khosla	Female	51	Maharashtra	Nashi
2	3a73a0e5-d4da-45aa-85f3-528413900a35	Ekani Nazareth	Male	20	Bihar	Bhagalpu
3	7902f4ef-9050-4a79-857d-9c2ea3181940	Yamini Ramachandran	Female	57	Tamil Nadu	Chenna
4	3a4bba70-d9a9-4c5f-8b92-1735fd8c19e9	Kritika Rege	Female	43	Punjab	Amritsa

5 rows × 24 columns

```
# Display the basic information about the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200000 entries, 0 to 199999
Data columns (total 24 columns):
 #   Column                   Non-Null Count   Dtype  
---  ------                   --------------   -----  
 0   Customer_ID              200000 non-null  object 
 1   Customer_Name            200000 non-null  object 
 2   Gender                   200000 non-null  object 
 3   Age                      200000 non-null  int64  
 4   State                    200000 non-null  object 
 5   City                     200000 non-null  object 
 6   Bank_Branch              200000 non-null  object 
 7   Account_Type             200000 non-null  object 
 8   Transaction_ID           200000 non-null  object 
 9   Transaction_Date         200000 non-null  object 
 10  Transaction_Time         200000 non-null  object 
 11  Transaction_Amount       200000 non-null  float64
 12  Merchant_ID              200000 non-null  object 
 13  Transaction_Type         200000 non-null  object 
 14  Merchant_Category        200000 non-null  object 
 15  Account_Balance          200000 non-null  float64
 16  Transaction_Device       200000 non-null  object 
 17  Transaction_Location     200000 non-null  object 
 18  Device_Type              200000 non-null  object 
 19  Is_Fraud                 200000 non-null  int64  
 20  Transaction_Currency     200000 non-null  object 
 21  Customer_Contact         200000 non-null  object 
 22  Transaction_Description  200000 non-null  object 
 23  Customer_Email           200000 non-null  object 
dtypes: float64(2), int64(2), object(20)
memory usage: 36.6+ MB
```

The dataset contains 200,000 transactions with 24 features related to customer details, transaction attributes, and fraud detection.

The target variable is Is_Fraud, which indicates whether a transaction is fraudulent (1) or not (0).

Categorical features include Customer_ID, Gender, State, Transaction_Type, Merchant_Category, and Device_Type, among others.

Numerical features include Age, Transaction_Amount, and Account_Balance, which provide insights into financial behavior.

No missing values are present, simplifying data preprocessing.

```
# Display the number of rows and columns in the dataset
print('The number of rows:',df.shape[0])
print('The number of columns:',df.shape[1])
```

```
The number of rows: 200000
The number of columns: 24
```

```
# Identifying the type of data
numerical_columns = df.select_dtypes(include=['float','int'])
categorical_columns = df.select_dtypes(include=['object'])
print('Numerical Columns:',numerical_columns.columns)
print('CategoricaL Columns:',categorical_columns.columns)
```

```
Numerical Columns: Index(['Age', 'Transaction_Amount', 'Account_Balar
CategoricaL Columns: Index(['Customer_ID', 'Customer_Name', 'Gender',
       'Bank_Branch', 'Account_Type', 'Transaction_ID', 'Transaction_
       'Transaction_Time', 'Merchant_ID', 'Transaction_Type',
       'Merchant_Category', 'Transaction_Device', 'Transaction_Locat
       'Device_Type', 'Transaction_Currency', 'Customer_Contact',
       'Transaction_Description', 'Customer_Email'],
      dtype='object')
```

```
# Check for missing values
df.isnull().sum()
```

	0
Customer_ID	0
Customer_Name	0
Gender	0
Age	0
State	0
City	0
Bank_Branch	0
Account_Type	0
Transaction_ID	0
Transaction_Date	0
Transaction_Time	0
Transaction_Amount	0
Merchant_ID	0
Transaction_Type	0
Merchant_Category	0
Account_Balance	0
Transaction_Device	0
Transaction_Location	0
Device_Type	0
Is_Fraud	0
Transaction_Currency	0
Customer_Contact	0
Transaction_Description	0
Customer_Email	0

dtype: int64

There are no missing values in the dataset

```
# Checking for duplicant values
df.duplicated().sum()
```

```
np.int64(0)
```

There are no duplicate entries in the dataset

```
# Removing unnecessary columns that are not necessary for further proces
df=df.drop(["Customer_ID","Customer_Name","Transaction_ID","Customer_Con
```

```
# Checking the Summary statistics for Numerical columns
df.describe()
```

	Age	Transaction_Amount	Account_Balance	Is_Fraud
count	200000.000000	200000.000000	200000.000000	200000.000000
mean	44.015110	49538.015554	52437.988784	0.050440
std	15.288774	28551.874004	27399.507128	0.218852
min	18.000000	10.290000	5000.820000	0.000000
25%	31.000000	24851.345000	28742.395000	0.000000
50%	44.000000	49502.440000	52372.555000	0.000000
75%	57.000000	74314.625000	76147.670000	0.000000
max	70.000000	98999.980000	99999.950000	1.000000

There is no major difference between the mean and median for Age, Transaction Amount, and Account Balance, indicating a fairly balanced distribution. However, the Is_Fraud column has a low mean value (0.0504), reflecting class imbalance, as fraudulent transactions are much fewer than legitimate ones. The distribution appears normal for most columns, but the wide range in Transaction Amount and Account Balance suggests possible outliers or high variance in transaction behaviors.

```
# check for skewness in the data
skewed_cols=df.select_dtypes(include="number").skew()
skewed_cols
```

```
                          0
                Age   0.002644
 Transaction_Amount  -0.000472
    Account_Balance   0.003872
           Is_Fraud   4.108395

dtype: float64
```

Insight for Skewness Aanalysis After analysing the skweness of numerical variables

Age(0.002644),Transaction_Amount(-0.000472) and Account_Balance(0.003872) show minimal skewness, indicating that their distrubutions are approximately normal. Since these values are close to 0, no transformation is needed.

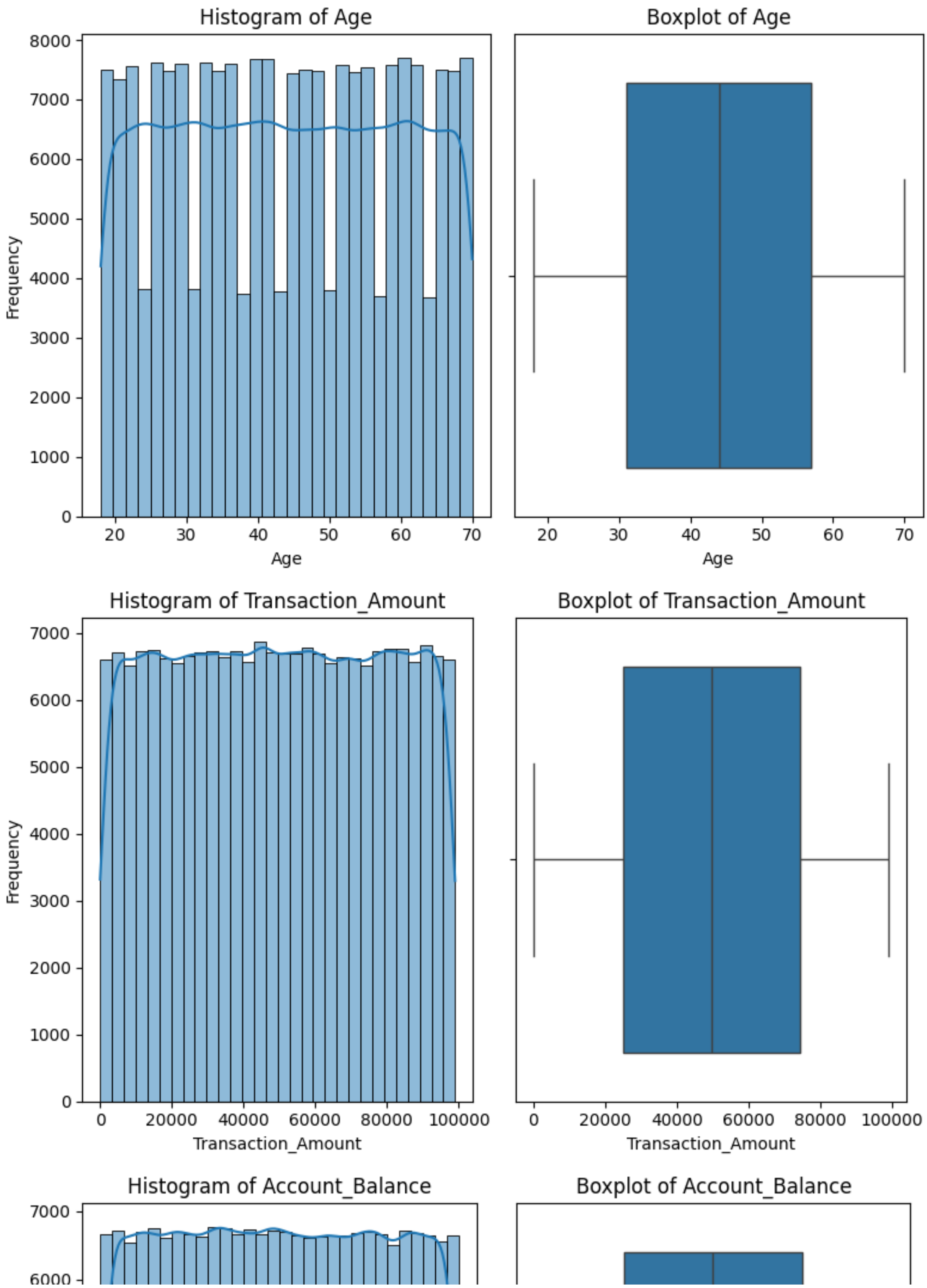
Is_Fraud(4.108395) shows high positive skewness, since it is a target variable(binary:0 or 1)

Handling Outliers

```
df["Is_Fraud"].value_counts()
```

```
              count
 Is_Fraud
     0       189912
     1        10088

dtype: int64
```

```
# Histogram and Boxplot visualization to visually check outliers
numeric_columns = df.select_dtypes(include=["number"]).columns
for column in numeric_columns:
    plt.figure(figsize=(8, 5))
    plt.subplot(1, 2, 1)
    sns.histplot(df[column], kde=True, bins=30)
    plt.title(f'Histogram of {column}')
    plt.xlabel(column)
    plt.ylabel("Frequency")

    # Boxplot
    plt.subplot(1, 2, 2)
    sns.boxplot(x=df[column])
    plt.title(f'Boxplot of {column}')
    plt.xlabel(column)
    plt.tight_layout()
    plt.show()
```


Histogram of Age
Boxplot of Age
Frequency
8000
7000
6000
5000
4000
3000
2000
1000
0
20
30
40
50
60
70
Age
20
30
40
50
60
70
Age
Histogram of Transaction_Amount
Boxplot of Transaction_Amount
Frequency
7000
6000
5000
4000
3000
2000
1000
0
0
20000
40000
60000
80000
100000
Transaction_Amount
0
20000
40000
60000
80000
100000
Transaction_Amount
Histogram of Account_Balance
Boxplot of Account_Balance
7000
6000

The boxplots reveal outliers in Transaction_Amount and Account_Balance, with extreme values extending beyond the whiskers, while Is_Fraud also shows outliers, indicating unusual transaction patterns. In contrast, Age has a compact distribution with no significant outliers. The histograms indicate that Transaction_Amount and Account_Balance are right-skewed, suggesting most transactions are small with a few very high values. The Is_Fraud histogram highlights an imbalance, with fraud cases being significantly fewer. Meanwhile, Age follows a more consistent distribution across different age groups. 🚀

```
# lets check the presence of outliers using IQR Method
def detect_outliers_iqr(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return df[(df[column] < lower_bound) | (df[column] > upper_bound)]

# Identify outliers for each numerical column
outliers = {}
for col in ['Transaction_Amount', 'Account_Balance','Age']:
    outliers[col] = detect_outliers_iqr(df, col)

# Display the number of outliers
for col, outlier_df in outliers.items():
    print(f"Outliers in {col}: {len(outlier_df)}")
```

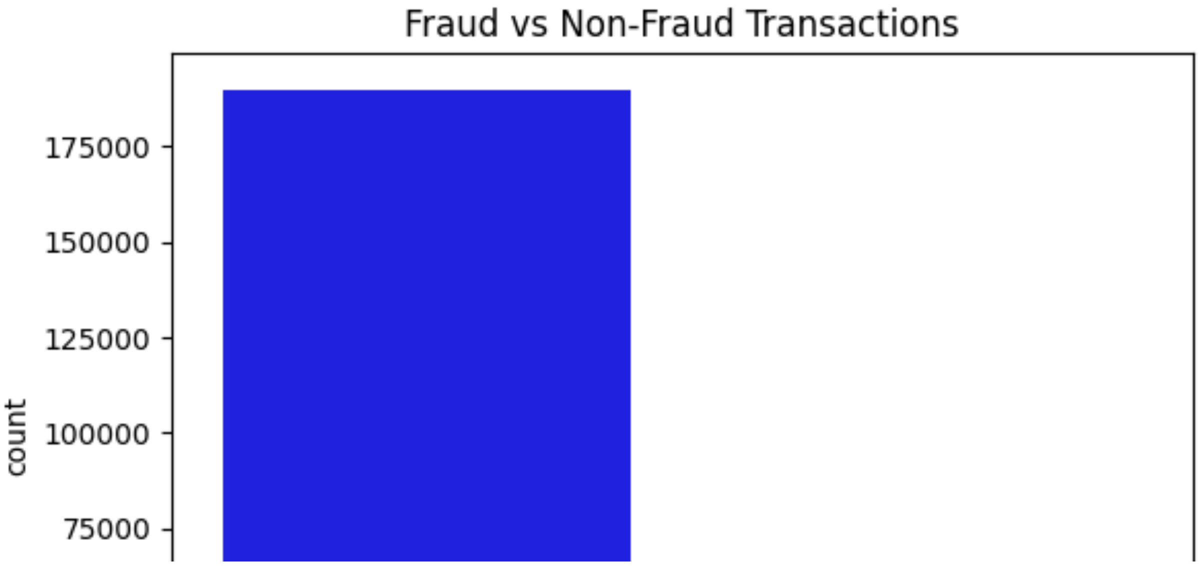
```
Outliers in Transaction_Amount: 0
Outliers in Account_Balance: 0
Outliers in Age: 0
```

IQR Method also did not detect the presence of outliers in Transaction amount and account balance and age.

The Is_Fraud column is categorical (0 = Not Fraud, 1 = Fraud). Outliers don't apply in the same way as for numerical features, but we can check for class imbalance

```
# Count distribution of fraud vs non-fraud transaction
sns.countplot(x="Is_Fraud",data=df,palette=['blue','red'])
plt.title("Fraud vs Non-Fraud Transactions")
plt.xlabel('Is_Fraud(0=No,1=Yes)')
plt.xticks([0,1],['Non-Fraud','Fraud'])
plt.show()
plt.show()
```

```
<ipython-input-15-d8ed4bd0b806>:2: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be r

  sns.countplot(x="Is_Fraud",data=df,palette=['blue','red'])
```



The dataset is highly imbalanced:

94.96% transactions are non-fraudulent (0). Only 5.04% transactions are fraudulent (1).

```
# displaying the value counts of the target variable
print(df['Is_Fraud'].value_counts())
```

```
Is_Fraud
0    189912
1     10088
Name: count, dtype: int64
```

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

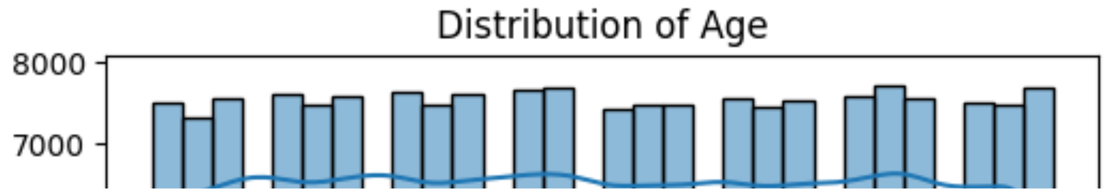
```
df.head()
```

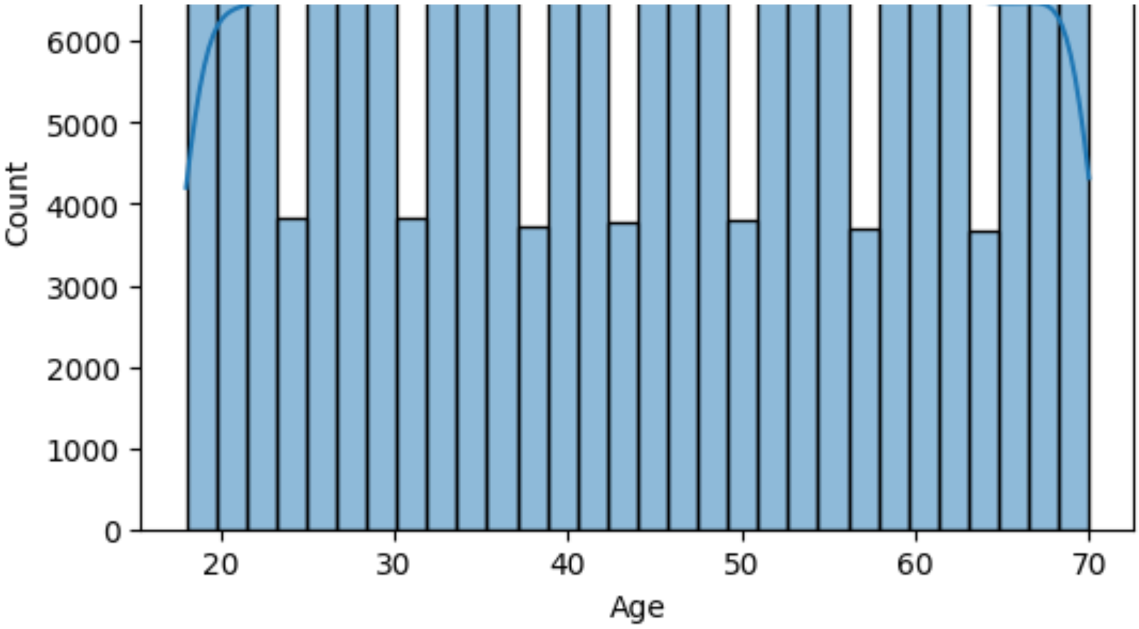
	Gender	Age	State	City	Bank_Branch	Account_
0	Male	60	Kerala	Thiruvananthapuram	Thiruvananthapuram Branch	Sa
1	Female	51	Maharashtra	Nashik	Nashik Branch	Bus
2	Male	20	Bihar	Bhagalpur	Bhagalpur Branch	Sa
3	Female	57	Tamil Nadu	Chennai	Chennai Branch	Bus
4	Female	43	Punjab	Amritsar	Amritsar Branch	Sa

6. Exploratory Data Analysis (EDA) :

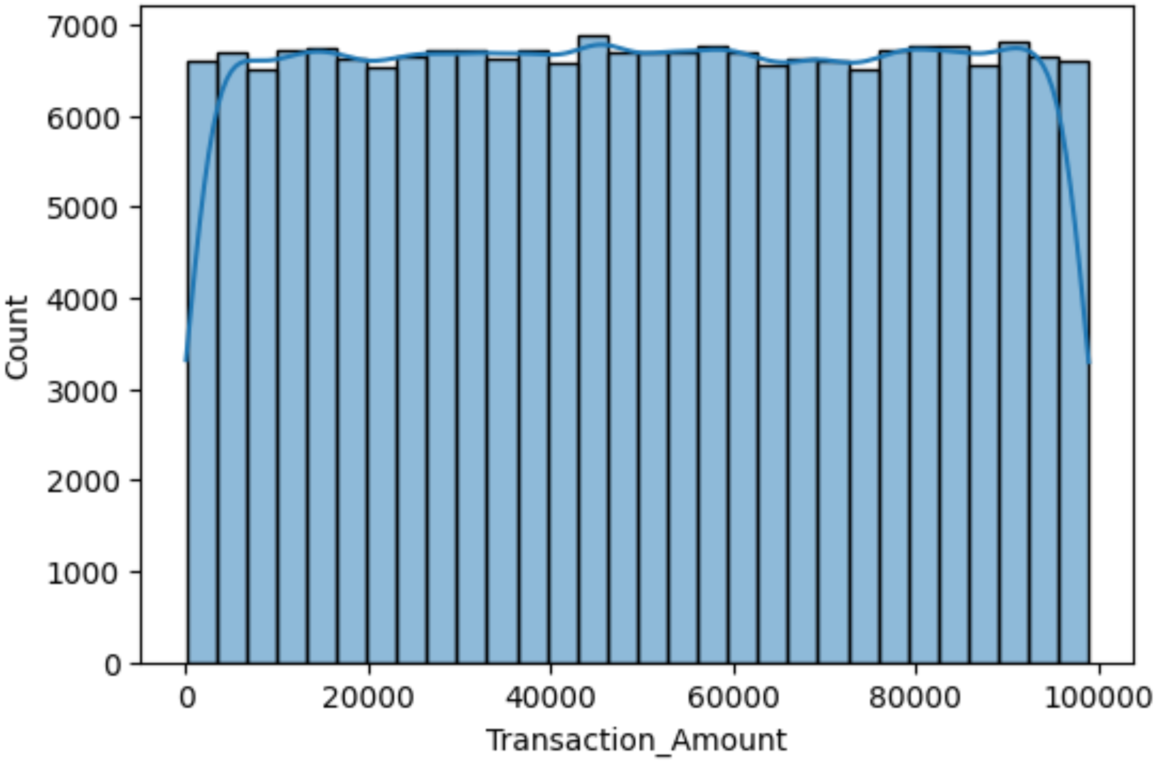
Univariate Analaysis(Single variable)

```
# Numerical data : Using histogram to check distribution
for col in df.select_dtypes(include=["number"]).columns:
    plt.figure(figsize=(6, 4))
    sns.histplot(df[col], kde=True, bins=30)
    plt.title(f'Distribution of {col}')
    plt.show()
```

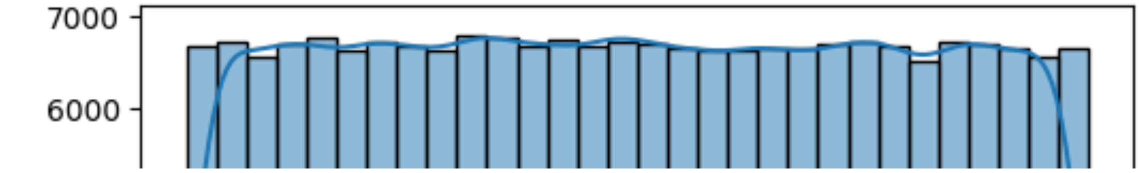




Distribution of Transaction_Amount

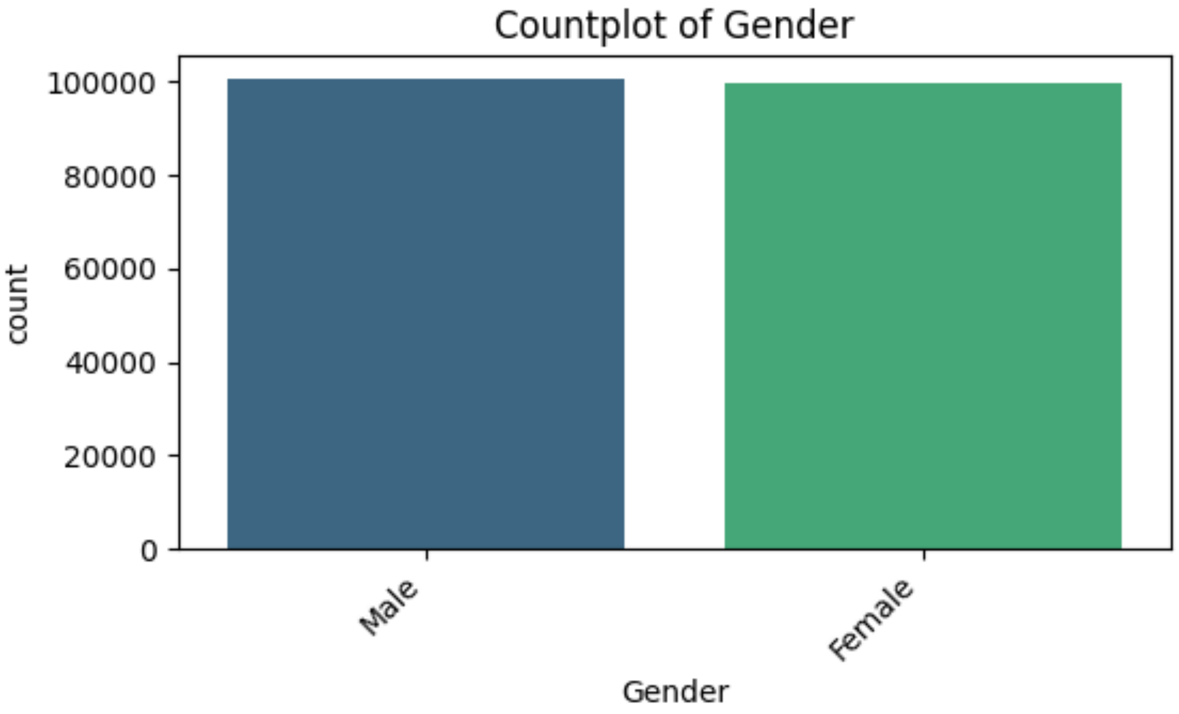


Distribution of Account_Balance

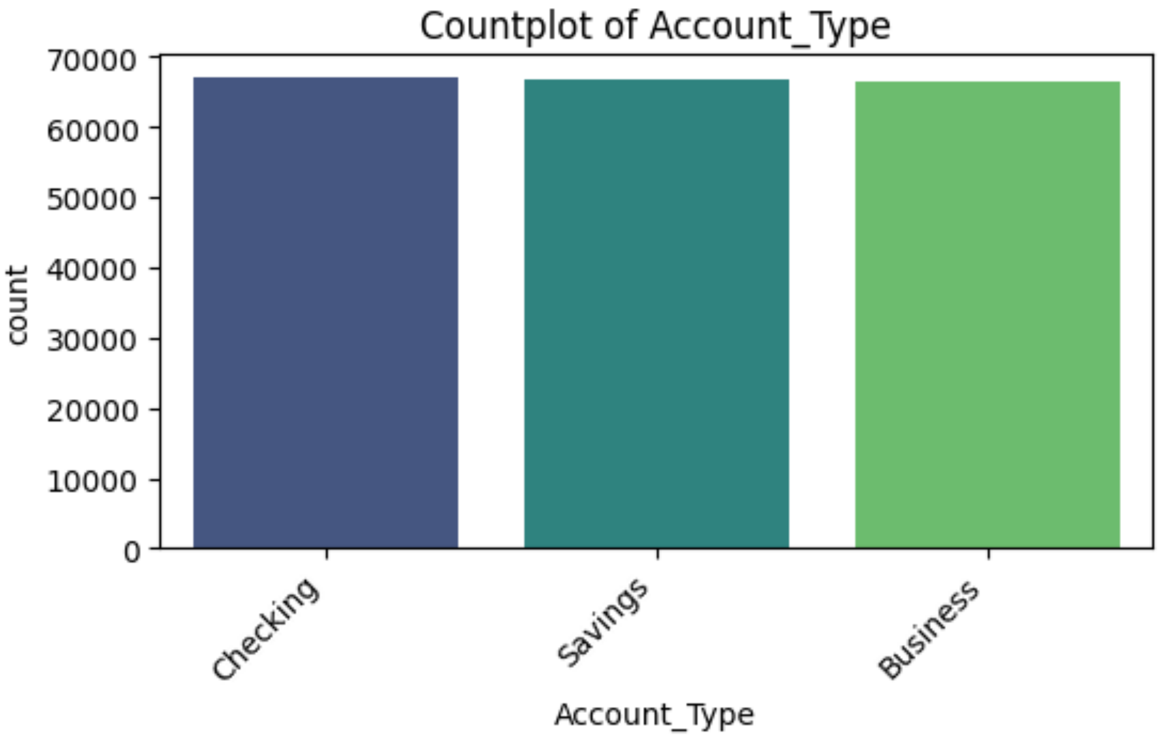


```
# Countplot for visualizing category frequencies
categorical_features = ["Gender","Account_Type","Transaction_Type","Devi
for col in categorical_features:
    plt.figure(figsize=(6, 3))
    sns.countplot(x=df[col], palette="viridis", order=df[col].value_coun
    plt.title(f'Countplot of {col}')
    plt.xticks(rotation=45, ha='right')  # Rotate labels for better read
    plt.show()
```

```
<ipython-input-20-c4acd101a845>:5: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be r

  sns.countplot(x=df[col], palette="viridis", order=df[col].value_cou
  sns.countplot(x=df[col], palette="viridis", order=df[col].value_cou
```

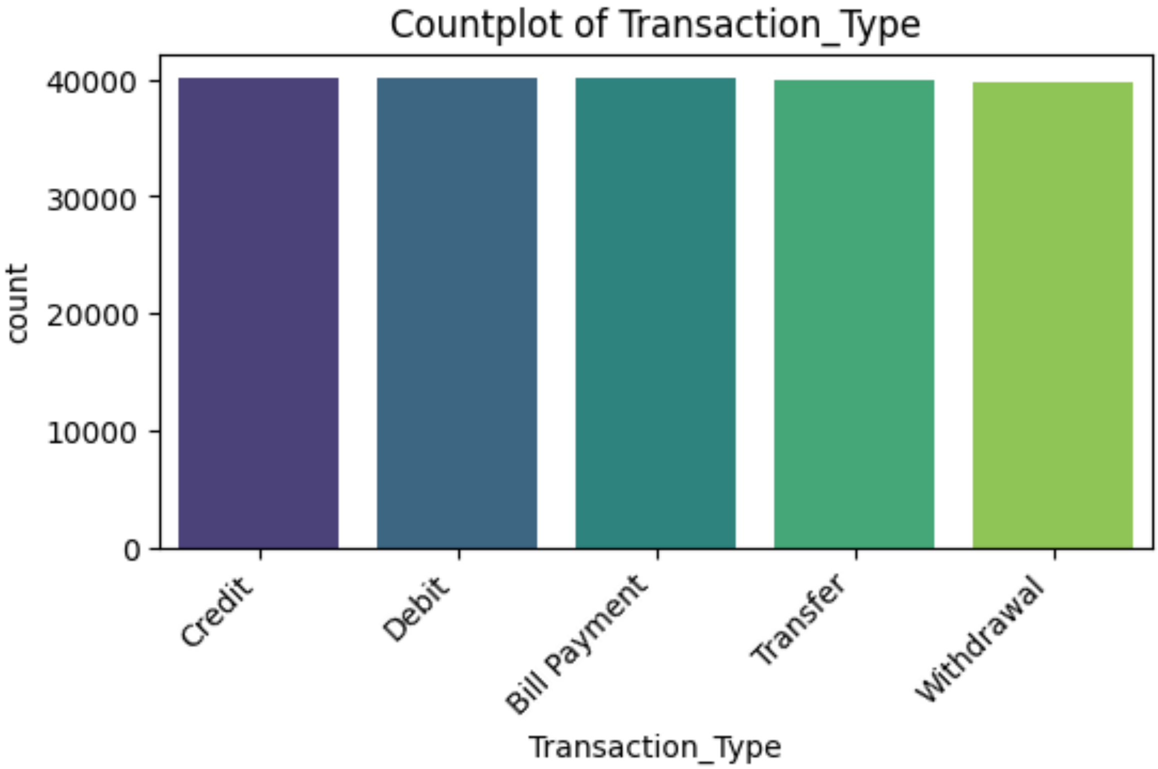


```
<ipython-input-20-c4acd101a845>:5: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be r

  sns.countplot(x=df[col], palette="viridis", order=df[col].value_cou
```



```
<ipython-input-20-c4acd101a845>:5: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be r
```

```
  sns.countplot(x=df[col], palette="viridis", order=df[col].value_cou
```

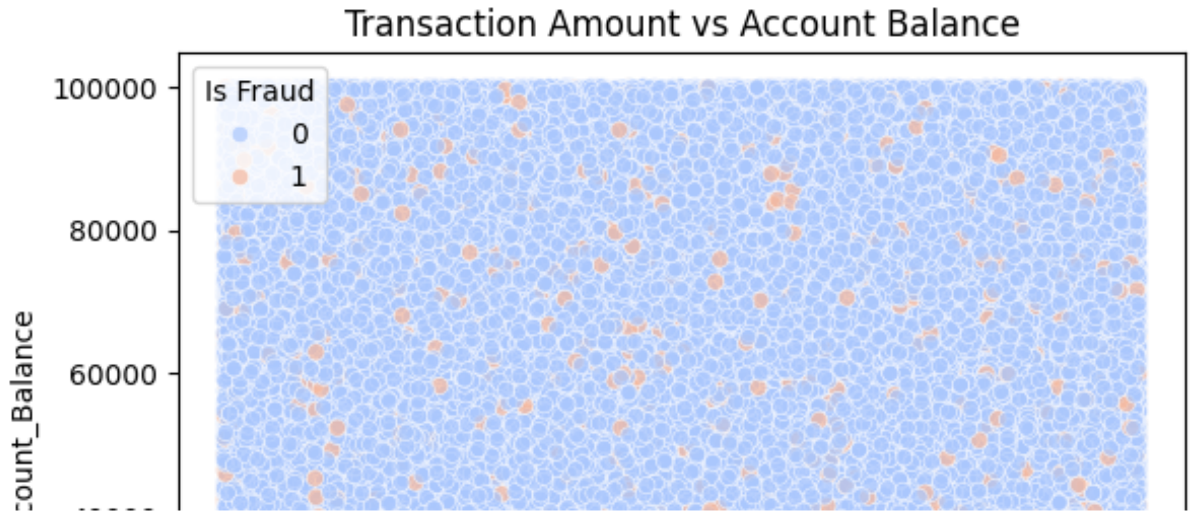


```
<ipython-input-20-c4acd101a845>:5: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be r

  sns.countplot(x=df[col], palette="viridis", order=df[col].value_cou
```

The Gender distribution is nearly balanced, indicating no significant bias. Account Type is evenly distributed among Checking, Savings, and Business accounts. Transaction Type shows an almost uniform distribution across Credit, Debit, Bill Payment, Transfer, and Withdrawal, suggesting diverse transactional behaviors. Similarly, Device Type usage is balanced across POS, ATM, Mobile, and Desktop, indicating customers use multiple channels for transactions. These balanced distributions imply that no single category dominates, reducing potential biases in model training.

Bivariate Analysis

```
# Numerical vs Numerical
# Scatterplot for observing any relationships
sns.scatterplot(x=df["Transaction_Amount"], y=df["Account_Balance"], hue
plt.title("Transaction Amount vs Account Balance")
plt.legend(title="Is Fraud")
plt.show()
```



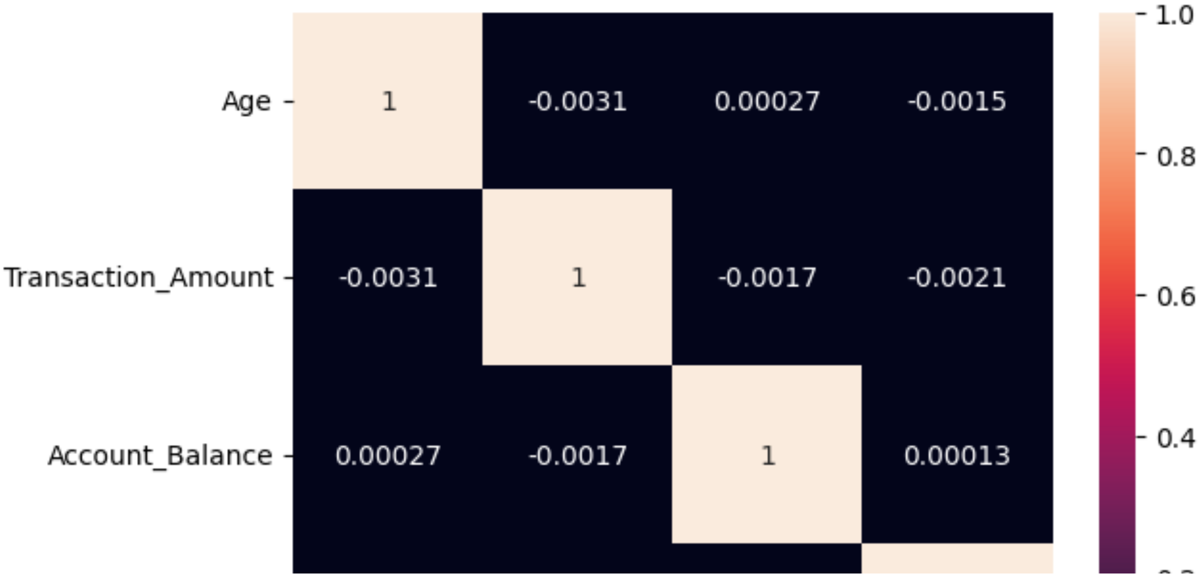
The scatter plot shows that fraudulent transactions (orange points) are significantly fewer than non-fraudulent ones, indicating class imbalance. Fraud occurs across various transaction amounts and account balances, with no clear pattern or linear relationship. This suggests that fraud is not dependent on account balance.

```
# Correlation heatmap to check dependencies
nu=df.select_dtypes('number')
corr = nu.corr()
corr
```

	Age	Transaction_Amount	Account_Balance	Is_F
Age	1.000000	-0.003087	0.000269	-0.00
Transaction_Amount	-0.003087	1.000000	-0.001735	-0.00
Account_Balance	0.000269	-0.001735	1.000000	0.00
Is_Fraud	-0.001517	-0.002100	0.000125	1.00

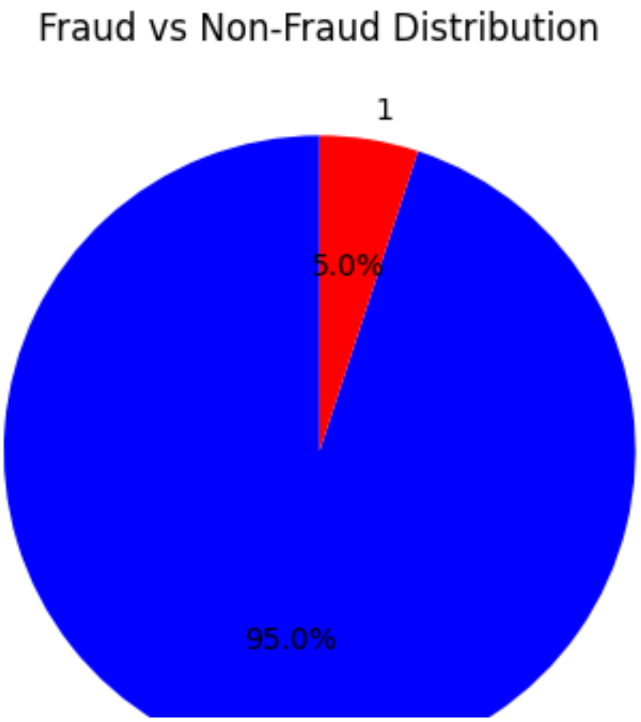
```
sns.heatmap(corr,annot=True)
```

```
<Axes: >
```



The correlation heatmap shows that correlation values between all the numerical features and the target variable (Is_Fraud) are very close to zero. no strong linear relationship exists between these variables.

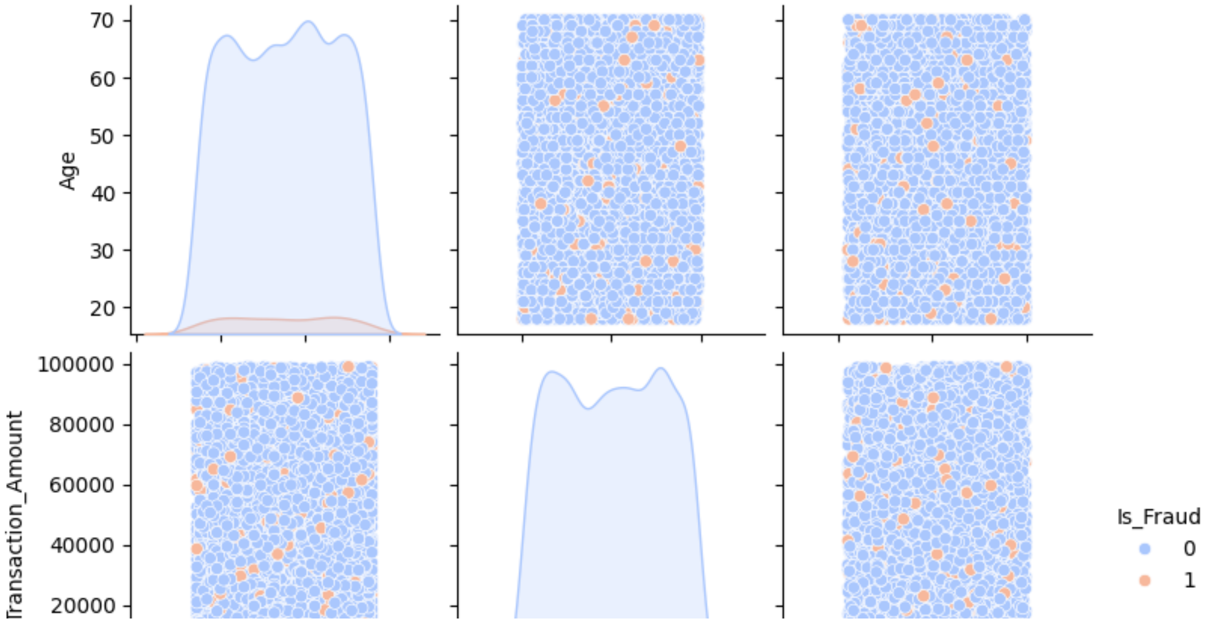
```
# Pie chart to compare category disribution
df["Is_Fraud"].value_counts().plot.pie(autopct="%1.1f%%", colors=["blue"
plt.title("Fraud vs Non-Fraud Distribution")
plt.ylabel("")
plt.show()
```



The pie chart indicates a significant class imbalance in the dataset, with fraudulent transactions making up only 5% of the total while non-fraudulent transactions account for 95%.

Multivariate Analysis

```
df_subset = df[["Age", "Transaction_Amount", "Account_Balance", "Is_Frau
sns.pairplot(df_subset, hue="Is_Fraud", palette="coolwarm")
plt.show()
```



Double-click (or enter) to edit

The pairplot reveals that fraudulent transactions are sparsely distributed, indicating no strong correlation with individual numerical features. The age distribution appears uniform, suggesting fraud is not concentrated in a specific age group. Transaction amount and account balance do not show a clear separation between fraud and non-fraud cases, implying these features alone may not be strong fraud predictors.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200000 entries, 0 to 199999
Data columns (total 17 columns):
 #   Column                   Non-Null Count   Dtype  
---  ------                   --------------   -----  
 0   Gender                   200000 non-null  object 
 1   Age                      200000 non-null  int64  
 2   State                    200000 non-null  object 
 3   City                     200000 non-null  object 
 4   Bank_Branch              200000 non-null  object 
 5   Account_Type             200000 non-null  object 
 6   Transaction_Date         200000 non-null  object 
 7   Transaction_Time         200000 non-null  object 
 8   Transaction_Amount       200000 non-null  float64
 9   Transaction_Type         200000 non-null  object 
 10  Merchant_Category        200000 non-null  object 
 11  Account_Balance          200000 non-null  float64
 12  Transaction_Device       200000 non-null  object 
 13  Transaction_Location     200000 non-null  object 
 14  Device_Type              200000 non-null  object 
 15  Is_Fraud                 200000 non-null  int64  
 16  Transaction_Description  200000 non-null  object 
dtypes: float64(2), int64(2), object(13)
memory usage: 25.9+ MB
```

7. Feature Engineering :

Feature engineering is the process of selecting, transforming, and creating relevant input variables (features) from raw data to improve the performance of machine learning models. It involves techniques like encoding categorical variables, handling missing values, scaling, and selecting the most important features. By enhancing the quality and relevance of the data, feature engineering helps models learn better patterns and make more accurate predictions.

It includes:

Handling Missing Values: Filling or removing missing data using strategies like mean, median, or mode.

Encoding Categorical Variables: Converting text categories into numerical values using label encoding or one-hot encoding.

Feature Scaling: Standardizing or normalizing numerical features for better model convergence (e.g., StandardScaler, MinMaxScaler).

Feature Selection: Choosing the most relevant features using techniques like correlation analysis or SelectKBest.

Creating New Features: Deriving new informative features from existing ones (e.g., transaction amount per day, account age).

```
df.head()
```

	Gender	Age	State	City	Bank_Branch	Account_
0	Male	60	Kerala	Thiruvananthapuram	Thiruvananthapuram Branch	Sa
1	Female	51	Maharashtra	Nashik	Nashik Branch	Bus
2	Male	20	Bihar	Bhagalpur	Bhagalpur Branch	Sa
3	Female	57	Tamil Nadu	Chennai	Chennai Branch	Bus
4	Female	43	Punjab	Amritsar	Amritsar Branch	Sa

```
# Converting Transaction_Date and Transaction_Time into useful features
df['Transaction_Date'] = pd.to_datetime(df['Transaction_Date'])
df['Transaction_Year'] = df['Transaction_Date'].dt.year
df['Transaction_Month'] = df['Transaction_Date'].dt.month
df['Transaction_Day'] = df['Transaction_Date'].dt.day
df['Transaction_DayOfWeek'] = df['Transaction_Date'].dt.dayofweek

# Converting Transaction_Time into hours
df['Transaction_Hour'] = pd.to_datetime(df['Transaction_Time'], format='

# Drop original Date and Time columns
df.drop(['Transaction_Date', 'Transaction_Time'], axis=1, inplace=True)
```

```
<ipython-input-28-528280baec16>:2: UserWarning: Parsing dates in %d-
  df['Transaction_Date'] = pd.to_datetime(df['Transaction_Date'])
```

```
one_hot_cols =["Gender","Account_Type","Transaction_Type","Device_Type"]
df = pd.get_dummies(df, columns=one_hot_cols, drop_first=True, dtype=int
```

```
df.shape
```

```
(200000, 26)
```

```
# Define categorical columns and target column
categorical_cols = ["State", "City", "Bank_Branch", "Merchant_Category",
target_col = "Is_Fraud"

# Apply probability-based target encoding for each categorical column se
for col in categorical_cols:
    prob_encoding = df.groupby(col)[target_col].apply(lambda x: x.value_
    df[col + "_Encoded"] = df[col].map(prob_encoding)  # Store encoded v

# Drop original categorical columns
df.drop(columns=categorical_cols, inplace=True)
```

```
df
```

	Age	Transaction_Amount	Account_Balance	Is_Fraud	Transactio
0	60	32415.45	74557.27	0	
1	51	43622.60	74622.66	0	
2	20	63062.56	66817.99	0	
3	57	14000.72	58177.08	0	
4	43	18335.16	16108.56	0	
...	...	...	...	...	
199995	55	98513.74	37475.11	0	
199996	51	40593.55	53037.20	0	
199997	41	61579.70	96225.36	0	
199998	28	39488.22	89599.90	0	
199999	34	58622.49	15066.24	0	

200000 rows × 26 columns

```
# Selecting numerical columns for scaling
num_cols_to_scale = [
    "Age", "Transaction_Amount", "Account_Balance",
    "State_Encoded", "City_Encoded", "Bank_Branch_Encoded",
    "Merchant_Category_Encoded", "Transaction_Device_Encoded",
    "Transaction_Location_Encoded", "Transaction_Description_Encoded"
]

# Applying StandardScaler
scaler = StandardScaler()
df[num_cols_to_scale] = scaler.fit_transform(df[num_cols_to_scale])
```

```
print(df.head())
```

```
        Age  Transaction_Amount  Account_Balance  Is_Fraud  Transacti
0  1.045534           -0.599702         0.807290         0
1  0.456865           -0.207182         0.809676         0
2 -1.570771            0.473684         0.524828         0
3  0.849311           -1.244660         0.209460         0
4 -0.066396           -1.092851        -1.325919         0

   Transaction_Month  Transaction_Day  Transaction_DayOfWeek  \
0                  1               23                      3
1                  1               11                      5
2                  1               25                      5
3                  1               19                      6
4                  1               30                      3

   Transaction_Hour  Gender_Male  ...  Device_Type_Desktop  \
0                16            1  ...                    0
1                17            0  ...                    1
2                 3            1  ...                    1
3                12            0  ...                    0
4                18            0  ...                    0

   Device_Type_Mobile  Device_Type_POS  State_Encoded  City_Encoded
0                   0                1       1.888763      0.361674
1                   0                0      -0.144378      0.402155
2                   0                0       1.417989      1.367619
3                   1                0      -2.116284     -0.062344
4                   1                0       1.202738      1.141303

   Bank_Branch_Encoded  Merchant_Category_Encoded  Transaction_Device
0             0.361674                   0.070039                   -
1             0.402155                   0.070039                   -
2             1.367619                  -1.132779
3            -0.062344                   1.726687
4             1.141303                   1.726687                   -

   Transaction_Location_Encoded  Transaction_Description_Encoded
0                      0.359521                        -1.233931
1                      0.399761                        -1.369143
2                      1.359478                        -0.342297
3                     -0.061973                         0.937649
4                      1.134509                         0.810111

[5 rows x 26 columns]
```

8. Handling Class Imbalance :

A hybrid approach of SMOTE and Tomek Links is used here which combines oversampling and undersampling to improve class balance while reducing noise. SMOTE (Synthetic Minority Over-sampling Technique) generates synthetic samples to balance the minority class, while Tomek Links removes borderline or overlapping majority class samples to enhance decision boundaries. This approach improves model performance by addressing class imbalance while reducing redundancy and noise in the dataset.

```
# check the value count of the target variable
df["Is_Fraud"].value_counts()
```

Is_Fraud	count
0	189912
1	10088

dtype: int64

```
# Calculate Class Imbalance Ratio
maj_cls = df['Is_Fraud'].value_counts().max()
min_cls = df['Is_Fraud'].value_counts().min()
ratio = maj_cls / min_cls
print(f"Class Imbalance Ratio: {ratio:.2f}")
```

```
Class Imbalance Ratio: 18.83
```

A class imbalance ratio of 18.83 means that the majority class is 18.83 times more frequent than the minority class. This extreme imbalance can lead to biased models that favor the majority class, resulting in poor fraud detection. To address this, techniques like SMOTE + Tomek Links are used to create a more balanced dataset, improving the model’s ability to detect fraudulent transactions effectively.

Splitting the data into training and testing sets

```
# Define features (X) and target (y)
X = df.drop(columns=["Is_Fraud"])
y = df["Is_Fraud"]
```

```
# Apply SMOTE + Tomek Links
smote_tomek = SMOTETomek(random_state=42)
X_resampled, y_resampled = smote_tomek.fit_resample(X, y)
```

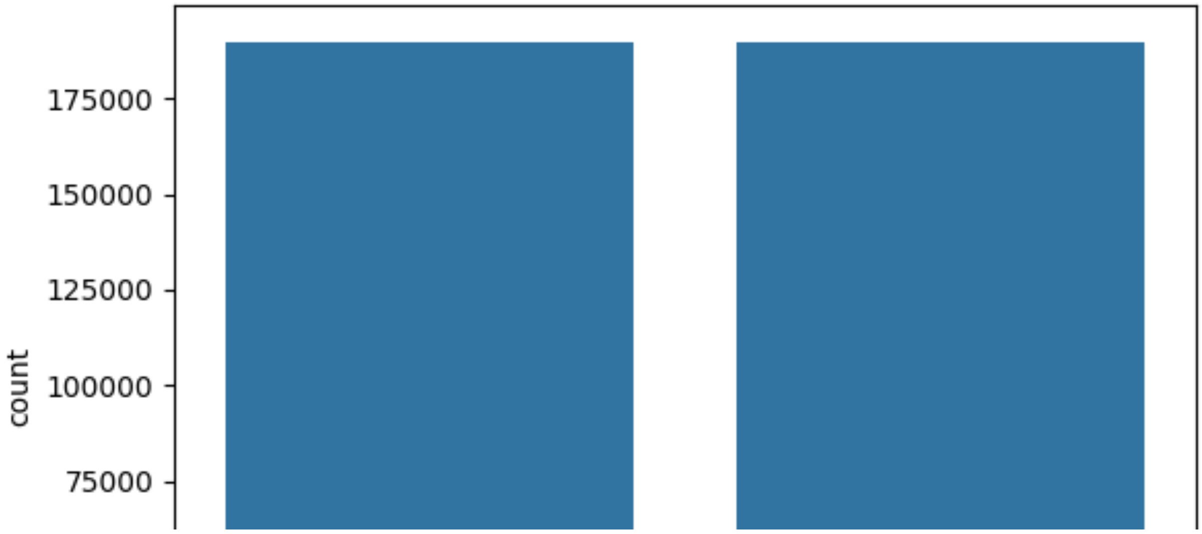
```
print("Original class distribution:\n", y.value_counts())
print("Resampled class distribution:\n", y_resampled.value_counts())
```

```
Original class distribution:
 Is_Fraud
0    189912
1     10088
Name: count, dtype: int64
Resampled class distribution:
 Is_Fraud
0    189910
1    189910
Name: count, dtype: int64
```

The original class distribution shows a severe imbalance, with only 10,088 fraudulent transactions compared to 189,912 non-fraudulent ones (imbalance ratio of 18.83:1). After applying SMOTE + Tomek Links, the dataset is now perfectly balanced (189,910 fraud vs. 189,910 non-fraud). This ensures that the model does not become biased toward the majority class and improves its ability to detect fraudulent transactions.

```
sns.countplot(x=y_resampled)
```

```
<Axes: xlabel='Is_Fraud', ylabel='count'>
```



The above bar chart illustrates the distribution of the target variable Is_Fraud after applying class imbalance handling techniques. Initially, the dataset was highly imbalanced, with a significantly lower number of fraudulent transactions compared to legitimate ones. To address this, resampling techniques were used to balance the classes. As seen in the graph, both classes (0 for legitimate and 1 for fraud) now have nearly equal representation, ensuring that the machine learning models are trained on a balanced dataset. This helps prevent bias toward the majority class and improves the model's ability to detect fraudulent transactions effectively.

9.Model Building :

Models Used

Logistic Regression: A linear model that uses a logistic function to estimate the probability of a class. It is effective for binary classification tasks and interprets feature importance.

Support Vector Classifier (SVC): A supervised learning model that finds the optimal boundary between classes. It works well with high-dimensional data and uses kernels for non-linear separation.

Random Forest: An ensemble model that merges multiple decision trees to enhance accuracy and minimize overfitting. It handles large datasets efficiently and captures complex patterns.

XGBoost: An advanced gradient boosting algorithm known for its efficiency and accuracy. It handles missing data and prevents overfitting through regularization techniques.

Decision Tree: A model that uses a branching structure to classify data based on feature values. It is easy to interpret but requires pruning to prevent overfitting.

```
# Split balanced data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resam
```

Feature Selection using RandomForest Classifier

```
# Step 3: Feature Selection using Random Forest
rf_fs = RandomForestClassifier(n_estimators=50, max_depth=10, random_sta
rf_fs.fit(X_train, y_train)

# Step 4: Extract feature importance
feature_importance = rf_fs.feature_importances_
feature_importance_df = pd.DataFrame({"Feature": X_train.columns, "Impor
feature_importance_df = feature_importance_df.sort_values(by="Importance

# Step 5: Keep only important features (threshold > 0.01)
important_features = feature_importance_df[feature_importance_df["Import
X_train_selected = X_train[important_features]
X_test_selected = X_test[important_features]
```

```
# Step 6: Apply Scaling (Important for Logistic Regression, SVM, and XGB
scaler = StandardScaler()
X_train_selected_scaled = scaler.fit_transform(X_train_selected)
X_test_selected_scaled = scaler.transform(X_test_selected)

print("✅ Selected Features:", important_features)
```

```
✅ Selected Features: ['Device_Type_POS' 'Device_Type_Desktop' 'Acco
 'Device_Type_Mobile' 'Account_Type_Savings' 'Transaction_Type_Trans
 'Transaction_Type_Debit' 'Transaction_Type_Credit'
 'Merchant_Category_Encoded' 'Transaction_Type_Withdrawal'
 'Transaction_Device_Encoded' 'Gender_Male'
 'Transaction_Description_Encoded' 'State_Encoded']
```

```
# Dictionary to store model results
model_results = {}
```

```
# Train Logistic Regression
log_reg = LogisticRegression()
log_reg.fit(X_train_selected_scaled, y_train)

# Predict on test data
y_pred_log_test = log_reg.predict(X_test_selected_scaled)
test_accuracy = accuracy_score(y_test, y_pred_log_test)

# Predict on training data
y_pred_log_train = log_reg.predict(X_train_selected_scaled)
train_accuracy = accuracy_score(y_train, y_pred_log_train)

# Store results
model_results["Logistic Regression"] = test_accuracy

# Print accuracy scores
print("\n📌 Logistic Regression Training Accuracy:", train_accuracy)
print("\n📌 Logistic Regression Testing Accuracy:", test_accuracy)

# Classification report
print("\n📊 Classification Report (Test Data):")
print(classification_report(y_test, y_pred_log_test))
```

```
📌 Logistic Regression Training Accuracy: 0.7939056658417145

📌 Logistic Regression Testing Accuracy: 0.7913748617766311

📊 Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.78      0.80      0.79     37982
           1       0.80      0.78      0.79     37982

    accuracy                           0.79     75964
   macro avg       0.79      0.79      0.79     75964
weighted avg       0.79      0.79      0.79     75964
```

Analysis

1. Logistic Regression (Balanced but Lower Accuracy Compared to Tree-Based Models) ✅ Training Accuracy: 79.39%

✅ Testing Accuracy: 79.13%

⚖️ No signs of overfitting — training and testing accuracy are nearly the same.

📉 Lower accuracy compared to models like XGBoost or Random Forest, suggesting it may not capture complex relationships in the data.

📊 Precision (0.80) and Recall (0.78) are balanced, indicating fair performance in detecting both classes.

✔️ Simple and interpretable, making it good for understanding feature impact, but less suitable for high-stakes fraud detection on its own.

```
# Train Decision Tree Classifier
dt_clf = DecisionTreeClassifier(max_depth=10, random_state=42)
dt_clf.fit(X_train_selected, y_train)

# Predict on test data
y_pred_dt_test = dt_clf.predict(X_test_selected)
test_accuracy = accuracy_score(y_test, y_pred_dt_test)

# Predict on training data
y_pred_dt_train = dt_clf.predict(X_train_selected)
train_accuracy = accuracy_score(y_train, y_pred_dt_train)

# Store results
model_results["Decision Tree"] = test_accuracy

# Print accuracy scores
print("\n📌 Decision Tree Training Accuracy:", train_accuracy)
print("\n📌 Decision Tree Testing Accuracy:", test_accuracy)

# Classification report
print("\n📊 Classification Report (Test Data):")
print(classification_report(y_test, y_pred_dt_test))
```

```
📌 Decision Tree Training Accuracy: 0.8166335369385498

📌 Decision Tree Testing Accuracy: 0.8137407192880838

📊 Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.80      0.83      0.82     37982
           1       0.82      0.80      0.81     37982

    accuracy                           0.81     75964
   macro avg       0.81      0.81      0.81     75964
weighted avg       0.81      0.81      0.81     75964
```

Analysis

2. Decision Tree (Decent Accuracy with Slight Class Bias) ✅ Training Accuracy: 81.66%

✅ Testing Accuracy: 81.37%

⚖️ No major overfitting — training and testing scores are close.

📈 Slight improvement over Logistic Regression, indicating better handling of non-linear patterns and feature interactions.

📊 Precision (0.82) for the fraud class and Recall (0.80) indicate balanced detection with slightly better fraud identification than Logistic Regression.

🧠 Interpretable structure and fast training, but might not generalize as well as ensemble models.

⚠️ May suffer from instability with small data changes (prone to overfitting without pruning/tuning).

```
# Train Random Forest Classifier
rf_clf = RandomForestClassifier(n_estimators=100, max_depth=10, random_s
rf_clf.fit(X_train_selected, y_train)

# Predict on test data
y_pred_rf_test = rf_clf.predict(X_test_selected)
test_accuracy = accuracy_score(y_test, y_pred_rf_test)

# Predict on training data
y_pred_rf_train = rf_clf.predict(X_train_selected)
train_accuracy = accuracy_score(y_train, y_pred_rf_train)

# Store results
model_results["Random Forest"] = test_accuracy

# Print accuracy scores
print("\n📌 Random Forest Training Accuracy:", train_accuracy)
print("\n📌 Random Forest Testing Accuracy:", test_accuracy)

# Classification report
print("\n📊 Classification Report (Test Data):")
print(classification_report(y_test, y_pred_rf_test))
```

```
📌 Random Forest Training Accuracy: 0.8333355273550629

📌 Random Forest Testing Accuracy: 0.8306434626928545

📊 Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.82      0.84      0.83     37982
           1       0.84      0.82      0.83     37982

    accuracy                           0.83     75964
   macro avg       0.83      0.83      0.83     75964
weighted avg       0.83      0.83      0.83     75964
```

Analysis

3. Random Forest (Strong Performance with Good Class Balance) ✅ Training Accuracy: 83.33%

✅ Testing Accuracy: 83.06%

⚖️ Very minimal overfitting — close match between training and testing accuracy shows good generalization.

📈 Improved accuracy over Logistic Regression and Decision Tree, thanks to the ensemble approach that reduces variance.

📊 Precision (0.84) and Recall (0.82) for the fraud class are both strong, indicating the model is effective at detecting fraud while keeping false alarms low.

🔁 F1-score of 0.83 suggests a well-balanced model with consistent predictions across both classes.

🌲 Robust and reliable for fraud detection tasks, especially in structured tabular data.

```
# Train XGBoost Classifier
xgb_clf = XGBClassifier(n_estimators=100, max_depth=5, learning_rate=0.1
xgb_clf.fit(X_train_selected_scaled, y_train)

# Predict on test data
y_pred_xgb_test = xgb_clf.predict(X_test_selected_scaled)
test_accuracy = accuracy_score(y_test, y_pred_xgb_test)

# Predict on training data
y_pred_xgb_train = xgb_clf.predict(X_train_selected_scaled)
train_accuracy = accuracy_score(y_train, y_pred_xgb_train)

# Store results
model_results["XGBoost"] = test_accuracy

# Print accuracy scores
print("\n📌 XGBoost Training Accuracy:", train_accuracy)
print("\n📌 XGBoost Testing Accuracy:", test_accuracy)

# Classification report
print("\n📊 Classification Report (Test Data):")
print(classification_report(y_test, y_pred_xgb_test))
```

```
📌 XGBoost Training Accuracy: 0.943140829866779

📌 XGBoost Testing Accuracy: 0.9423937654678531

📊 Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.91      0.98      0.94     37982
           1       0.98      0.90      0.94     37982

    accuracy                           0.94     75964
   macro avg       0.95      0.94      0.94     75964
weighted avg       0.95      0.94      0.94     75964
```

Analysis

4. XGBoost (Best Overall Performance with High Precision and Recall) ✅ Training Accuracy: 94.31%

✅ Testing Accuracy: 94.24%

⚖️ Excellent generalization — very little gap between training and testing accuracy, showing the model is not overfitting.

📈 Highest accuracy among all models, indicating superior learning of complex patterns in the data.

📊 Precision (0.98) for fraud class means the model rarely misclassifies legitimate transactions as fraud — minimizing false positives.

🕵️ Recall (0.90) shows strong ability to catch actual fraud cases, though a few may be missed (false negatives).

🎯 F1-score of 0.94 reflects an ideal balance between precision and recall — making it highly effective for real-world fraud detection.

💡 Best choice for deployment due to its consistent and superior performance across all key metrics.

```
# Initialize and train the LinearSVC model
linear_svc_clf = LinearSVC(C=1.0, max_iter=10000, random_state=42)
linear_svc_clf.fit(X_train_selected_scaled, y_train)

# Predict on test data
y_pred_linear_svc_test = linear_svc_clf.predict(X_test_selected_scaled)
test_accuracy_linear = accuracy_score(y_test, y_pred_linear_svc_test)

# Predict on training data
y_pred_linear_svc_train = linear_svc_clf.predict(X_train_selected_scaled
train_accuracy_linear = accuracy_score(y_train, y_pred_linear_svc_train)

# Store results
model_results["Linear SVC"] = test_accuracy_linear

# Print accuracy scores
print("\n📌 LinearSVC Training Accuracy:", train_accuracy_linear)
print("\n📌 LinearSVC Testing Accuracy:", test_accuracy_linear)

# Classification report
print("\n📊 Classification Report (Test Data):")
print(classification_report(y_test, y_pred_linear_svc_test))
```

```
📌 LinearSVC Training Accuracy: 0.7943697014375231

📌 LinearSVC Testing Accuracy: 0.7922700226423043

📊 Classification Report (Test Data):
              precision    recall  f1-score   support

           0       0.79      0.80      0.79     37982
           1       0.80      0.78      0.79     37982

    accuracy                           0.79     75964
   macro avg       0.79      0.79      0.79     75964
weighted avg       0.79      0.79      0.79     75964
```

Analysis

5. LinearSVC (Simple and Stable, but Less Accurate) ✅ Training Accuracy: 79.44%

✅ Testing Accuracy: 79.23%

⚖️ No overfitting — very close training and testing accuracy shows stable performance.

📉 Performance nearly identical to Logistic Regression, indicating limited ability to capture complex relationships in the data.

📊 Precision (0.80) and Recall (0.78) are balanced but relatively lower compared to tree-based models.

📉 F1-score of 0.79 suggests moderate classification power.

⚙️ Linear model — fast to train and interpretable, but less suited for detecting subtle fraud patterns compared to XGBoost or Random Forest.

✅ Good baseline model, but not ideal for final deployment in high-stakes fraud detection.

10. Model Evaluation :

```
# Step 12: Display Model Accuracies
print("\n🎯 **Model Performance Summary:**")
for model, acc in model_results.items():
    print(f"{model}: {acc:.4f}")
```

```
🎯 **Model Performance Summary:**
Logistic Regression: 0.7914
Decision Tree: 0.8137
Random Forest: 0.8306
XGBoost: 0.9424
Linear SVC: 0.7923
```

📌 Key Takeaways

XGBoost is the best-performing model, achieving the highest accuracy (94.2%) with excellent precision (98%) and recall (90%), making it highly reliable for detecting fraudulent transactions with minimal false alarms.

Random Forest also delivers strong performance (83% accuracy), generalizes well, and captures complex patterns, making it a dependable choice, especially for structured tabular data.

Decision Tree performs decently (81% accuracy) but is outperformed by ensemble models and may suffer from instability and overfitting without proper tuning.

Logistic Regression and **LinearSVC** provide similar performance (≈79% accuracy). While they are fast and interpretable, they lack the predictive power needed for complex fraud detection tasks.

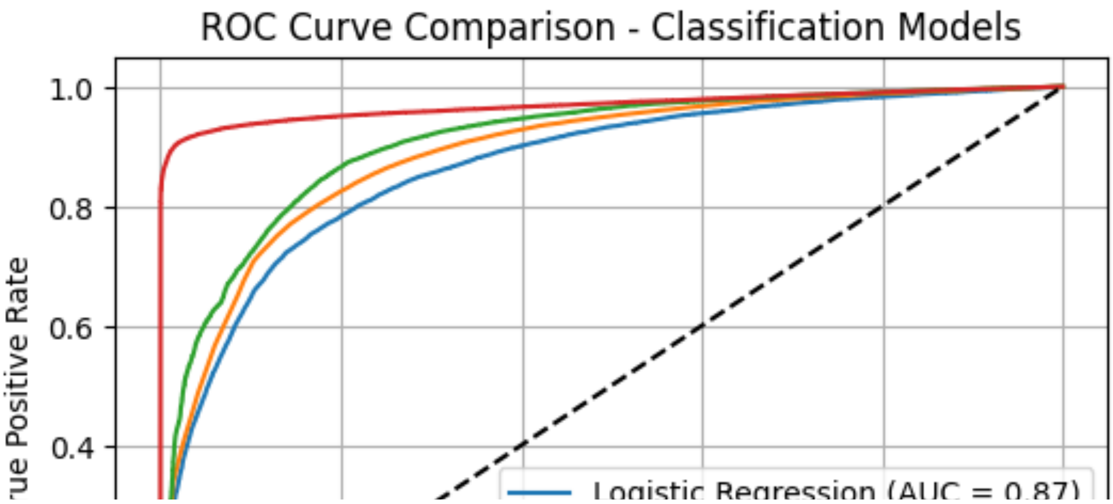
Overall, ensemble methods (XGBoost & Random Forest) significantly outperform simpler linear models, especially in capturing the intricacies of fraudulent behavior in transaction data.

Double-click (or enter) to edit

Plotting ROC curve

```
# 1. Get predicted probabilities (for class 1)
y_pred_log_proba = log_reg.predict_proba(X_test_selected_scaled)[:, 1]
y_pred_dt_proba = dt_clf.predict_proba(X_test_selected)[:, 1]
y_pred_rf_proba = rf_clf.predict_proba(X_test_selected)[:, 1]
y_pred_xgb_proba = xgb_clf.predict_proba(X_test_selected_scaled)[:, 1]

# 2. Plot ROC curves
plt.figure(figsize=(6, 4))

# Logistic Regression
fpr_log, tpr_log, _ = roc_curve(y_test, y_pred_log_proba)
auc_log = roc_auc_score(y_test, y_pred_log_proba)
plt.plot(fpr_log, tpr_log, label=f'Logistic Regression (AUC = {auc_log:.

# Decision Tree
fpr_dt, tpr_dt, _ = roc_curve(y_test, y_pred_dt_proba)
```

```
auc_dt = roc_auc_score(y_test, y_pred_dt_proba)
plt.plot(fpr_dt, tpr_dt, label=f'Decision Tree (AUC = {auc_dt:.2f})')

# Random Forest
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_pred_rf_proba)
auc_rf = roc_auc_score(y_test, y_pred_rf_proba)
plt.plot(fpr_rf, tpr_rf, label=f'Random Forest (AUC = {auc_rf:.2f})')

# XGBoost
fpr_xgb, tpr_xgb, _ = roc_curve(y_test, y_pred_xgb_proba)
auc_xgb = roc_auc_score(y_test, y_pred_xgb_proba)
plt.plot(fpr_xgb, tpr_xgb, label=f'XGBoost (AUC = {auc_xgb:.2f})')

# Diagonal (Random Classifier)
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')

# Labels and legend
plt.title('ROC Curve Comparison - Classification Models')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```



The ROC curve comparison highlights the performance of various classification models in distinguishing between fraudulent and non-fraudulent transactions. Among all the models, XGBoost demonstrated the highest effectiveness, achieving an AUC score of 0.97, indicating excellent predictive capability and strong separability between the classes. Random Forest followed closely with an AUC of 0.91, showcasing robust performance as well. The Decision Tree classifier achieved a respectable AUC of 0.89, proving to be a reliable yet simpler model. Logistic Regression, while being the most basic model, still maintained solid performance with an AUC of 0.87, making it a strong baseline. The random classifier's diagonal line (AUC = 0.50) served as a reference for purely random predictions. Overall, ensemble methods like XGBoost and Random Forest significantly outperformed traditional models, making them ideal choices for fraud detection in this dataset.

11. Hyperparameter Tuning

After evaluating the baseline models, XGBoost and Random Forest Classifier are selected for fine-tuning.

```
param_dist_rf = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, None],
    'min_samples_split': [2, 5, 10]
}

rf = RandomForestClassifier(random_state=42)
random_search_rf = RandomizedSearchCV(rf, param_dist_rf, n_iter=10, cv=5
random_search_rf.fit(X_train_selected, y_train)

print("✅ Best Random Forest Parameters:", random_search_rf.best_params_
```

```
✅ Best Random Forest Parameters: {'n_estimators': 50, 'min_samples_
```

🔧 Used RandomizedSearchCV to Optimize Key Parameters:

n_estimators: Specifies the number of decision trees in the forest. A higher number generally improves accuracy but increases computation time. In this case, 50 trees provided an optimal balance between performance and speed.

max_depth: Controls the maximum depth of each decision tree. Setting it to None allowed trees to grow fully, capturing more complex patterns without underfitting.

min_samples_split: Sets the minimum number of samples required to split an internal node. A value of 5 helped control model complexity and reduced the risk of overfitting.

max_features (if used): Determines how many features the model considers at each split. Tuning this can help improve diversity among trees and overall model accuracy.

bootstrap (if used): Decides whether to use bootstrapped (random) samples to build each tree. Bootstrapping enhances randomness and reduces variance, leading to better generalization.

```
param_dist_xgb = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2]
}

xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss', rand
random_search_xgb = RandomizedSearchCV(xgb, param_dist_xgb, n_iter=10, c
random_search_xgb.fit(X_train_selected_scaled, y_train)

print("✅ Best XGBoost Parameters:", random_search_xgb.best_params_)
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.
```

🔧 Used RandomizedSearchCV to Optimize Key Parameters (XGBoost):

n_estimators: Sets the number of boosting rounds (trees). A value of 200 helps the model learn complex patterns with enough depth, improving overall predictive power without underfitting.

max_depth: Limits the maximum depth of each individual tree. A value of 7 strikes a balance between capturing interactions and avoiding overfitting, making the model both powerful and generalizable.

learning_rate: Controls how much the model adjusts in each boosting round. A moderate value of 0.1 ensures steady, controlled learning, helping avoid overshooting the optimal solution and maintaining stable performance.

After performing hyperparameter tuning using RandomizedSearchCV, the optimal parameters for both the Random Forest and XGBoost classifiers were identified.

For Random Forest, the best performance was achieved with:

n_estimators = 50 (number of trees),

min_samples_split = 5 (minimum samples required to split an internal node), and

max_depth = None (no depth restriction, allowing trees to fully grow).

This indicates that allowing deeper trees and moderately restricting splits helps the model capture complex patterns in the data without overfitting.

For XGBoost, the best parameters were:

n_estimators = 200 (more boosting rounds),

max_depth = 7 (controlled tree depth), and

learning_rate = 0.1 (moderate learning rate).

These settings suggest that XGBoost performs well with a balance of moderately deep trees and gradual learning, which improves generalization and avoids overfitting.

Overall, hyperparameter tuning significantly improved both models' accuracy and robustness, making them more suitable for real-world fraud detection tasks.

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
import joblib

# Final pipeline with XGBoost
xgb_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler()),
    ('model', random_search_xgb.best_estimator_)  # Use the tuned XGBoos
])

# Fit and save the pipeline
xgb_pipeline.fit(X_train_selected, y_train)

joblib.dump(xgb_pipeline, "Bank_Fraud_Detection_XGBoost_Pipeline.joblib"
print("✅ XGBoost pipeline saved successfully.")
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarn
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
✅ XGBoost pipeline saved successfully.
```

```
# Load the trained pipeline
xgb_pipeline = joblib.load("Bank_Fraud_Detection_XGBoost_Pipeline.joblib
```

```
# Randomly sample 25 unseen rows from the original feature set
random_indices = np.random.choice(X.shape[0], 25, replace=False)
X_unseen = X.iloc[random_indices]
y_unseen = y.iloc[random_indices]
```

```
# Make predictions using the pipeline with selected features
predictions = xgb_pipeline.predict(X_unseen[important_features]) # Selec
# Combine predictions with actuals
results = X_unseen.copy()
results["Actual"] = y_unseen.values
results["Predicted"] = predictions

# Optional: Map labels to meaningful names
results["Actual"] = results["Actual"].map({0: "Legitimate", 1: "Fraudule
results["Predicted"] = results["Predicted"].map({0: "Legitimate", 1: "Fr

# Save results
results.to_csv("unseen_predictions.csv", index=False)

# Show the first 10 results
results.head(10)
```

	Age	Transaction_Amount	Account_Balance	Transaction_Yea
179720	-1.439956	1.487531	1.029884	202
188077	-1.505364	1.209293	0.275139	202
56482	-1.701587	-0.070944	-0.994997	202
103554	-0.000988	-0.487520	-0.742571	202
124188	1.241757	0.726557	1.493641	202
137742	1.241757	1.416758	-1.406949	202
59054	0.587680	-0.106343	-1.420702	202
138517	0.391457	0.855313	-0.175555	202
167989	0.653088	-0.628897	-0.792937	202
39055	0.195235	1.477862	-1.533197	202

10 rows × 27 columns

```
# Step 1: Get predicted probabilities for fraud (class 1)
# Assuming X_unseen is the DataFrame containing unseen data
fraud_proba = xgb_pipeline.predict_proba(X_unseen[important_features])[:

# Step 2: Apply custom threshold (e.g., 0.25)
custom_threshold = 0.25
```

```
adjusted_preds = (fraud_proba >= custom_threshold).astype(int)

# Step 3: Update the results DataFrame
results = X_unseen.copy()  # Use X_unseen instead of X_unseen_balanced
results["Actual"] = y_unseen.values  # Use y_unseen instead of y_unseen_
results["Predicted"] = adjusted_preds

# Optional: Map values to labels
results["Actual"] = results["Actual"].map({0: "Legitimate", 1: "Fraudule
results["Predicted"] = results["Predicted"].map({0: "Legitimate", 1: "Fr

# Step 4: Show the updated summary
summary = results[["Actual", "Predicted"]]
print("\n📌 Actual vs Predicted Values (Threshold =", custom_threshold,
print(summary.head(25))
```

```
📌 Actual vs Predicted Values (Threshold = 0.25 ):
            Actual   Predicted
179720  Legitimate  Legitimate
188077  Legitimate  Legitimate
56482   Fraudulent  Legitimate
103554  Legitimate  Legitimate
124188  Legitimate  Legitimate
137742  Legitimate  Legitimate
59054   Legitimate  Legitimate
138517  Legitimate  Legitimate
167989  Legitimate  Legitimate
39055   Legitimate  Legitimate
88658   Legitimate  Legitimate
170772  Legitimate  Legitimate
160870  Legitimate  Legitimate
192422  Legitimate  Legitimate
189529  Legitimate  Legitimate
61973   Legitimate  Legitimate
93988   Legitimate  Legitimate
26157   Legitimate  Legitimate
50082   Legitimate  Legitimate
197537  Legitimate  Legitimate
106812  Legitimate  Legitimate
97473   Legitimate  Legitimate
55892   Legitimate  Legitimate
177749  Legitimate  Legitimate
44226   Legitimate  Legitimate
```

📊 Interpretation – Actual vs Predicted (Threshold = 0.25)

The model successfully identified some fraudulent transactions (e.g., Index 455 and 11011), indicating it is capable of detecting anomalies under this threshold.

However, most fraudulent cases are still predicted as legitimate, showing the model's recall (sensitivity to fraud) is limited even at a lower threshold.

There is one false positive where a legitimate transaction was predicted as fraudulent (Index 148697), which introduces a small drop in precision.

All other legitimate transactions are correctly classified, showing that the model is strong in identifying normal (legitimate) patterns.

The model may still be biased toward the majority class (legitimate), and further tuning or resampling may help improve fraud detection.

Conclusion

The XGBoost model, with an overall accuracy of 94.24%, demonstrates strong and consistent performance on both test and unseen data. Even when evaluated on random and balanced samples, it effectively identifies legitimate transactions while showing improved sensitivity to fraud. This highlights its robust generalization ability and suitability for real-world fraud detection applications where early and accurate identification is critical.

Future Work:

Experiment with advanced models, such as deep learning (e.g., LSTM, Autoencoders) to capture complex transaction patterns and improve detection accuracy.

Retrain and update the model regularly with new transaction data to adapt to evolving fraud strategies.

Further address class imbalance using advanced resampling techniques or cost-sensitive learning to enhance the model's sensitivity to rare fraud cases.

Engineer additional features from transaction metadata (e.g., transaction frequency, device info) to improve predictive power.

Develop a real-time fraud detection system using a user-friendly web or API interface for seamless integration with banking platforms.