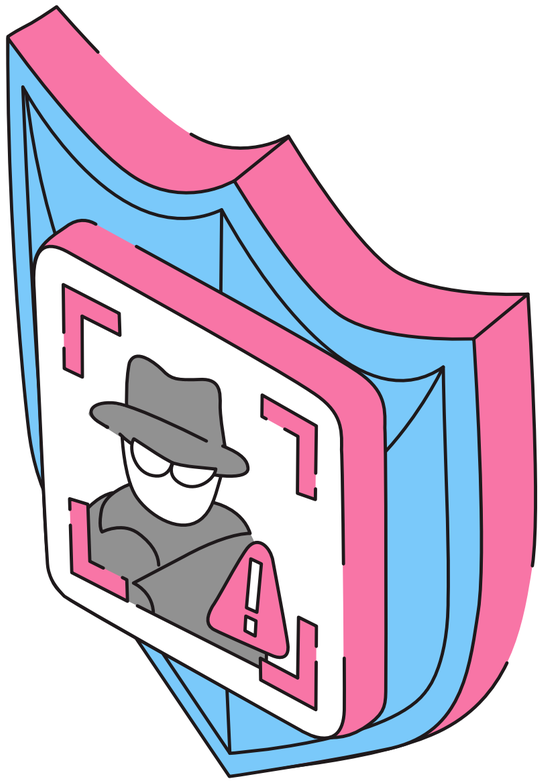
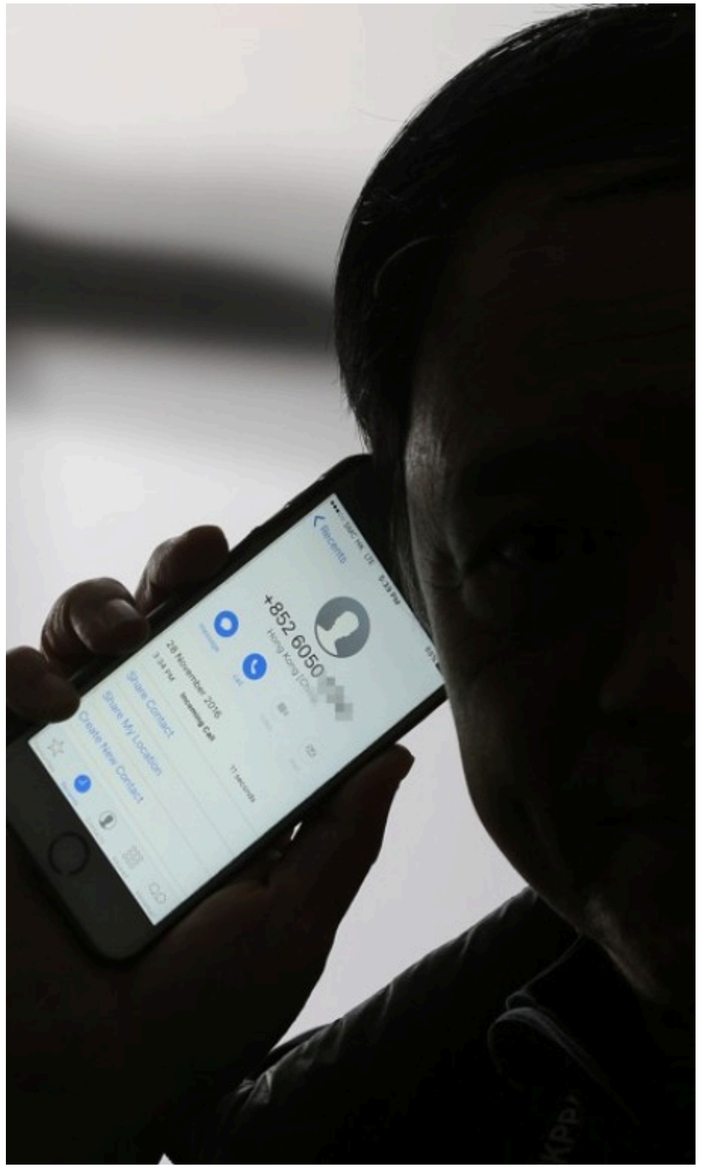


Fraud Detection

Using Machine Learning

Presenters:
CHAN Ho Lam Myles
CHONG Tin Tak
LAI Chi Shing
YANG Huiyan



Escalating Digital Fraud Risks

40,000 fraud cases[1]

HK$9.18 billion Losses[1]

2024
HK$343.6 billion[2]

2029
HK$776 billion[2]

Machine Learning

- Detect anomalies
- New fraud patterns

Objective: Comparative Analysis of the **Effectiveness of Fraud Detection** using Different Machine Learning Algorithms

[1] Cybersource. (2023). Global Fraud and Payments Report 2023. Retrieved from https://www.cybersource.com/en-ap/solutions/fraud-and-risk-management/fraud-report.html
[2] Statista. (2024, June 16). Global e-commerce payment fraud losses 2029. Retrieved from https://www.statista.com/statistics/1273177/ecommerce-payment-fraud-losses-globally/

Data Overview

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	\
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	

	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	M1979787155	0.0	0.0	0.0	0.0
1	M2044282225	0.0	0.0	0.0	0.0
2	C553264065	0.0	0.0	1.0	0.0
3	C38997010	21182.0	0.0	1.0	0.0
4	M1230701703	0.0	0.0	0.0	0.0

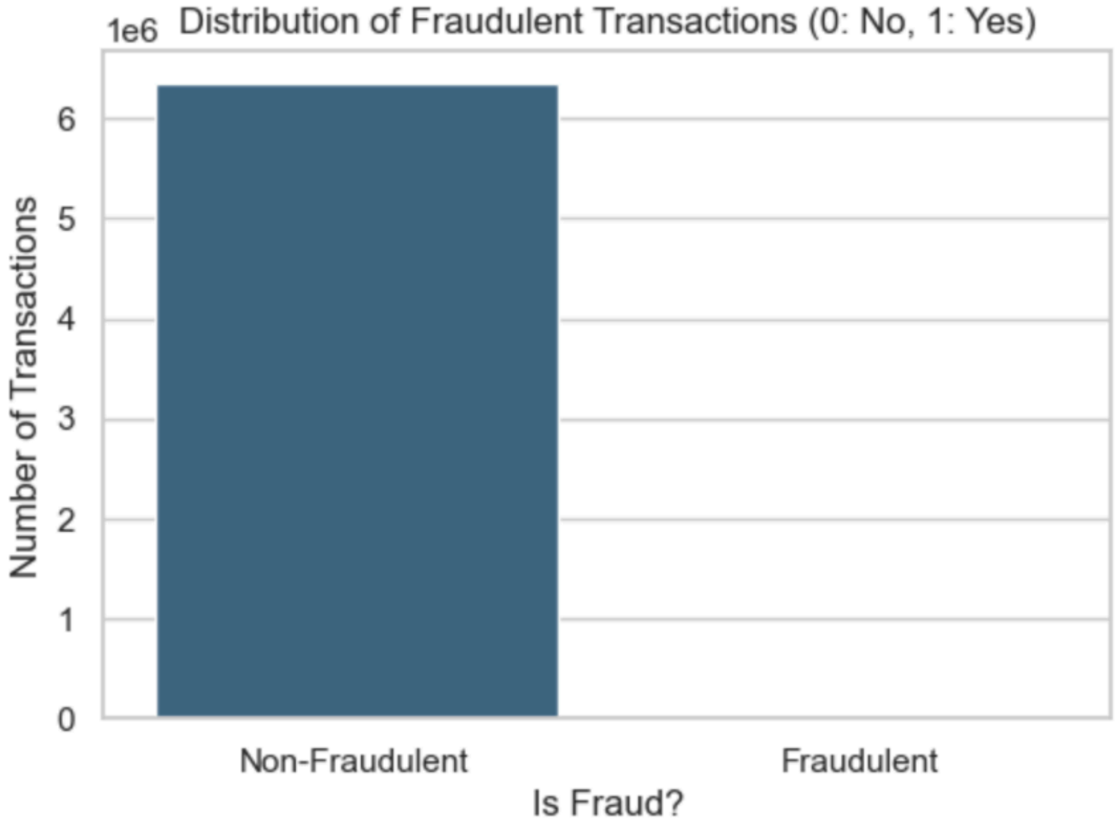
Features	Feature Name	Description
Numerical	Step	Unit of time (1 step = 1 hour)
	Amount	Amount of the transaction
	oldbalanceOrgsort	Balance before the transaction
	newbalanceOrigsort	Balance after the transaction
	oldbalanceDestsort	Initial balance of recipient before the transaction
	newbalanceDestsort	New balance of recipient after the transaction
Categorical	Type	Type of online transaction
Target	isFraud	Sort fraud transaction

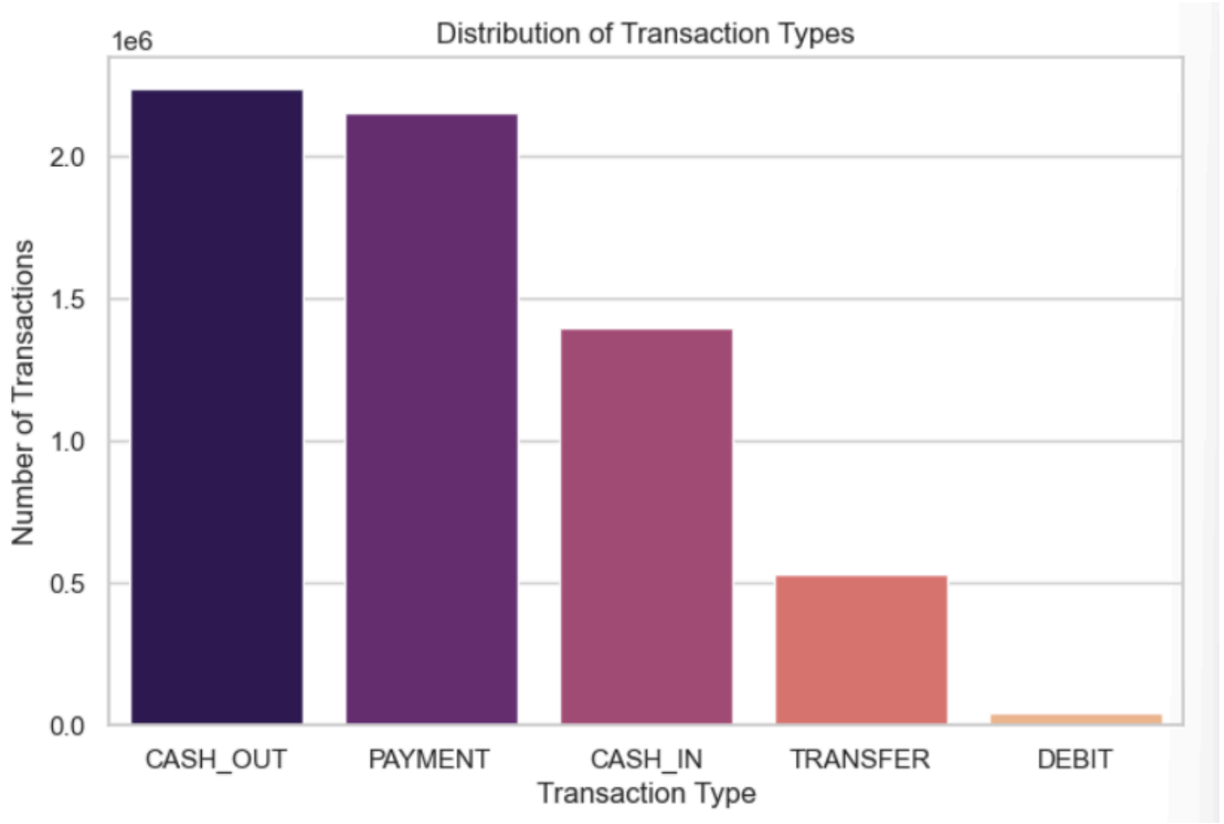
Data Overview

```
isFraud
0.0    2896409
1.0       2567
```

0.0885%

Significant class imbalance!





Data Processing

01

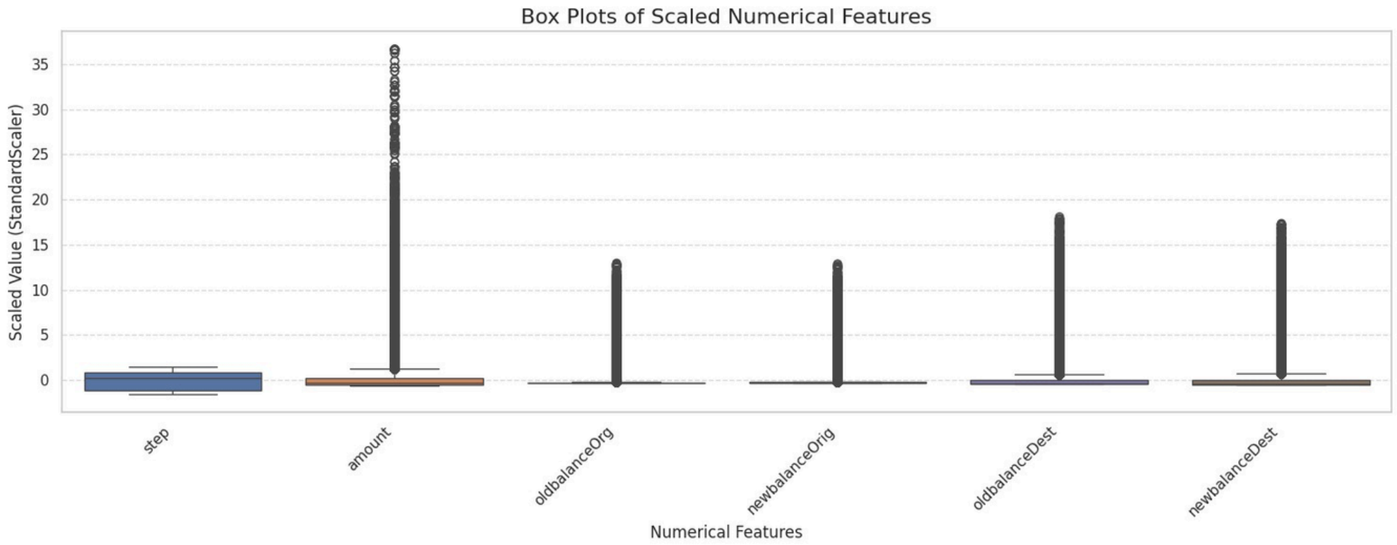
Data Preprocessing

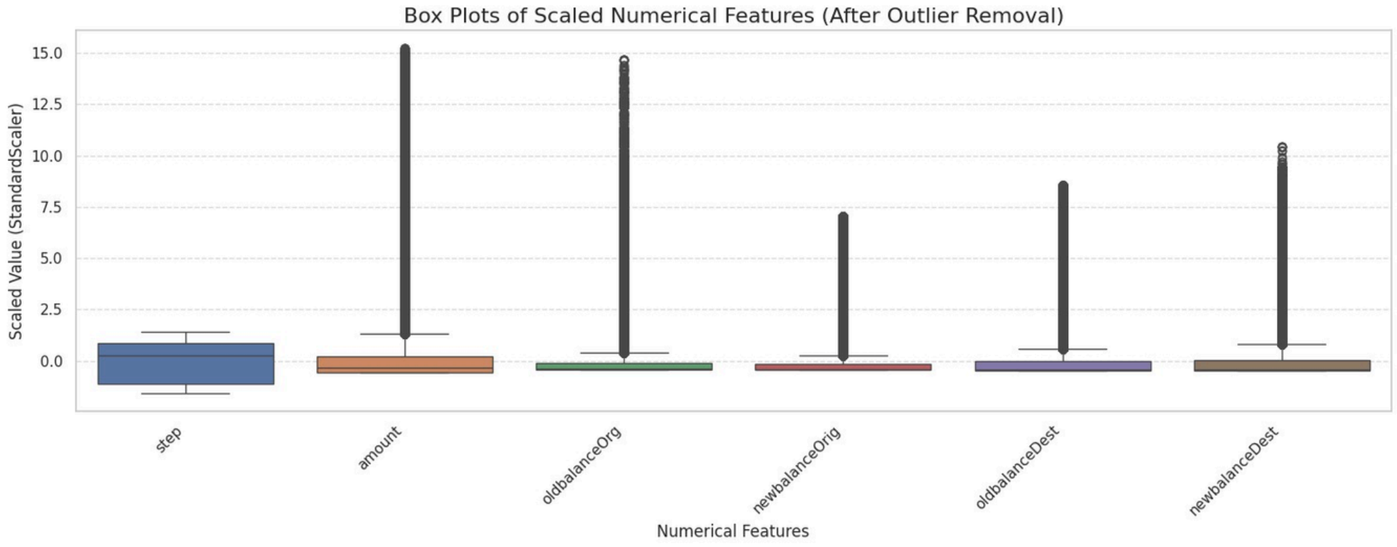
- Data Cleaning
- Remove Outliers: IQR

02

Feature Engineering

- Numerical: StandardScaler
- Categorical: OneHotEncoder
- Dimension Reduction: PCA





Data Processing

01

Data Preprocessing

- Data Cleaning
- Remove Outliers: IQR

02

Feature Engineering

- Numerical: StandardScaler
- Categorical: OneHotEncoder
- Dimension Reduction: PCA

```
   num__step  num__amount  num__oldbalanceOrg  num__newbalanceOrig  \
0  -1.598696    -0.549787            0.636976             0.491401
1  -1.598696    -0.579917           -0.302535            -0.309989
2  -1.598696    -0.586276           -0.435478            -0.420233
3  -1.598696    -0.586276           -0.435478            -0.420233
4  -1.598696    -0.542879           -0.174405            -0.250267

   num__oldbalanceDest  num__newbalanceDest  cat__type_CASH_OUT  \
0            -0.453255            -0.497089                 0.0
1            -0.453255            -0.497089                 0.0
2            -0.453255            -0.497089                 0.0
3            -0.441626            -0.497089                 1.0
4            -0.453255            -0.497089                 0.0

   cat__type_DEBIT  cat__type_PAYMENT  cat__type_TRANSFER
0              0.0                1.0                 0.0
1              0.0                1.0                 0.0
2              0.0                0.0                 1.0
3              0.0                0.0                 0.0
4              0.0                1.0                 0.0

Shape of X after scaling/encoding: (2565117, 10)
```

Feature Engineering

	PC1	PC2	PC3	PC4	PC5
0	-0.738856	0.040836	1.677014	-0.054083	0.686347
1	-0.769263	0.103693	1.678303	-0.068545	0.687060
2	-0.631551	0.123705	1.689476	0.060162	0.049350
3	-0.659362	0.172005	1.704730	-0.090991	-0.718942
4	-0.760610	0.099718	1.678123	-0.053749	0.687259
...	...	...	...	...	...
6362615	-0.228525	0.373018	-3.503145	0.297089	-0.782377
6362616	3.887739	0.763864	-3.574893	9.359431	0.143809
6362617	4.929845	1.231314	-3.508941	8.867335	-0.520536
6362618	0.100335	0.343157	-3.525524	1.223894	-0.005266
6362619	2.423553	1.267017	-3.400889	-0.052649	-0.538406

6362620 rows × 5 columns

PCA

- Dimension reduction
- Creates new features to represent data more efficiently

Hyperparameter tuning

GridSearchCV with 5-fold cross-validation

- Prevent overfitting
- Average performance across subsets
- Maximize performance

```
Shape of X after PCA: (2565117, 5)
```

Data Splitting

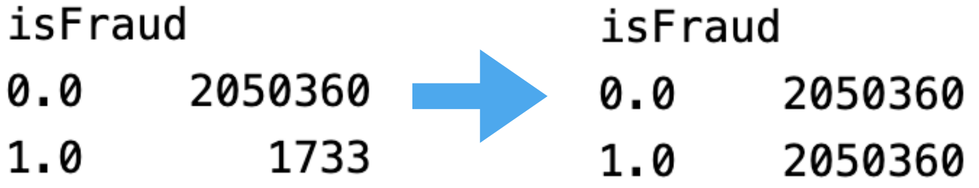
- Training data = 80%
- Test data = 20%

Dataset	Shape	Non-Fraud (%)	Fraud (%)
X_train	(5090096, 5)	-	-
X_test	(1272524, 5)	-	-
y_train	(5090096,)	0.9987092581357994	0.0012907418642005967
y_test	(1272524,)	0.9987088652159016	0.0012911347840983745

Resampling

SMOTE
(Synthetic Minority Over-sampling Technique)

- Address class imbalance
- Steps:
  - Selecting a ‘fraud’ sample at random
  - Finding its **k nearest neighbors** among other ‘fraud’ samples
  - Randomly choosing one of these neighbors and creating a new synthetic sample

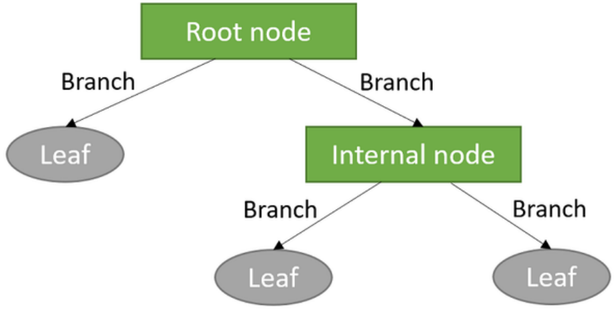


Model Selection

01

Decision Tree

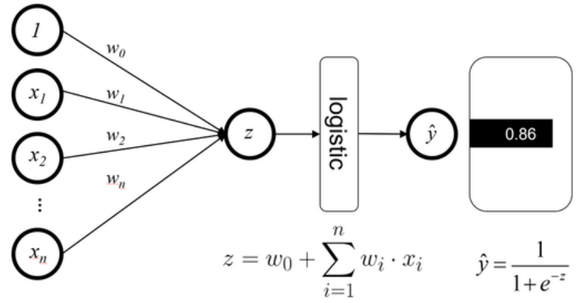
- Interpretability and Transparency
- Speed and Efficiency



02

Logistic Regression

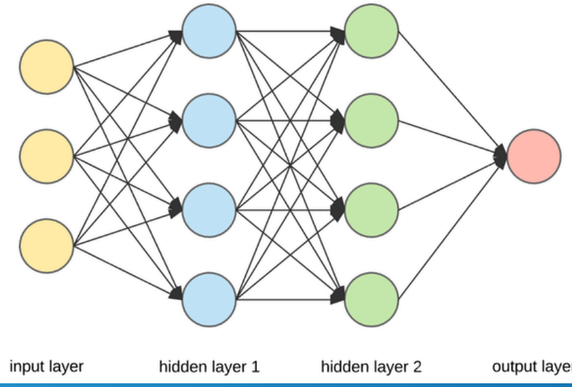
- Simplicity and Robustness
- Probability Outputs for Risk Scoring



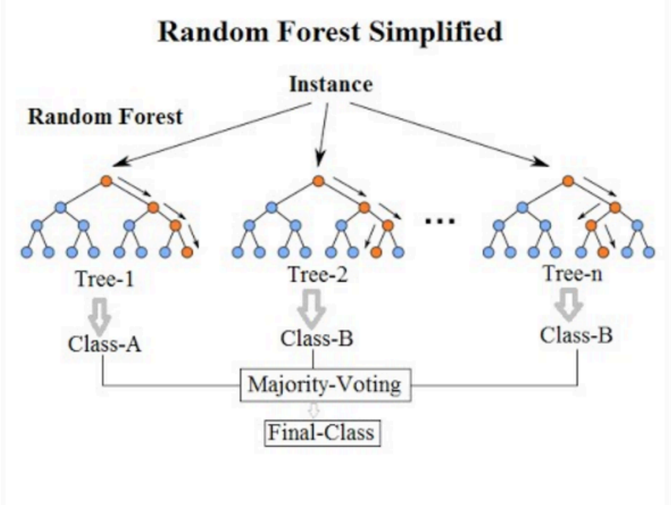
03

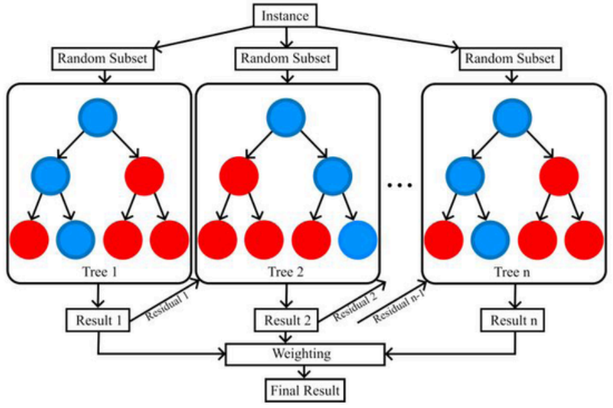
Neural Network

- Model Complex Patterns
- Adaptability and Continuous Learning



Model Selection





04

Random Forest

- High Accuracy and Robustness
- Resistance to Overfitting

05

XGBoost

- Effectiveness with Imbalanced Data
- Customizable and Tunable
- Scalability and Speed

Why SVM Was Excluded...

Impractical for hyperparameter tuning

- SVM's **Non-Linear Power**:
  - Abstract kernel transformation
  - Achieves high performance using kernel functions (like RBF) to handle complex, non-linear patterns

- **Computational Complexity**:
  - Training time complexity scales poorly with the number of samples (N)
  - Often $O(N^2)$ to $O(N^3)$ for standard implementations.

~2.5 million rows of data → Hyperparameter tuning: **Time prohibitive!**

Due to the $O(N^2)$ complexity and large dataset size, finding optimal SVM hyperparameters via 5-Fold CV was **computationally infeasible**.

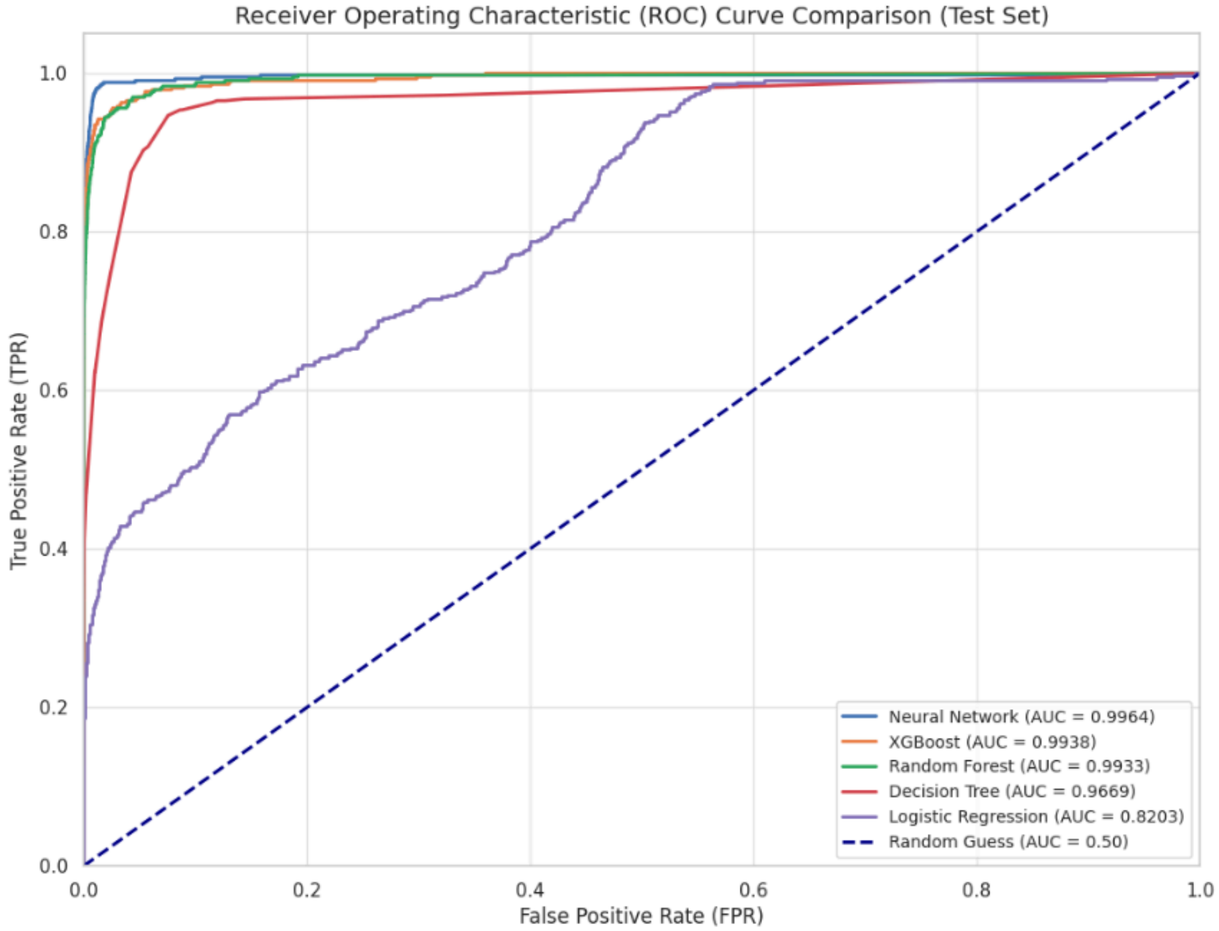
Optimizing Model Performance

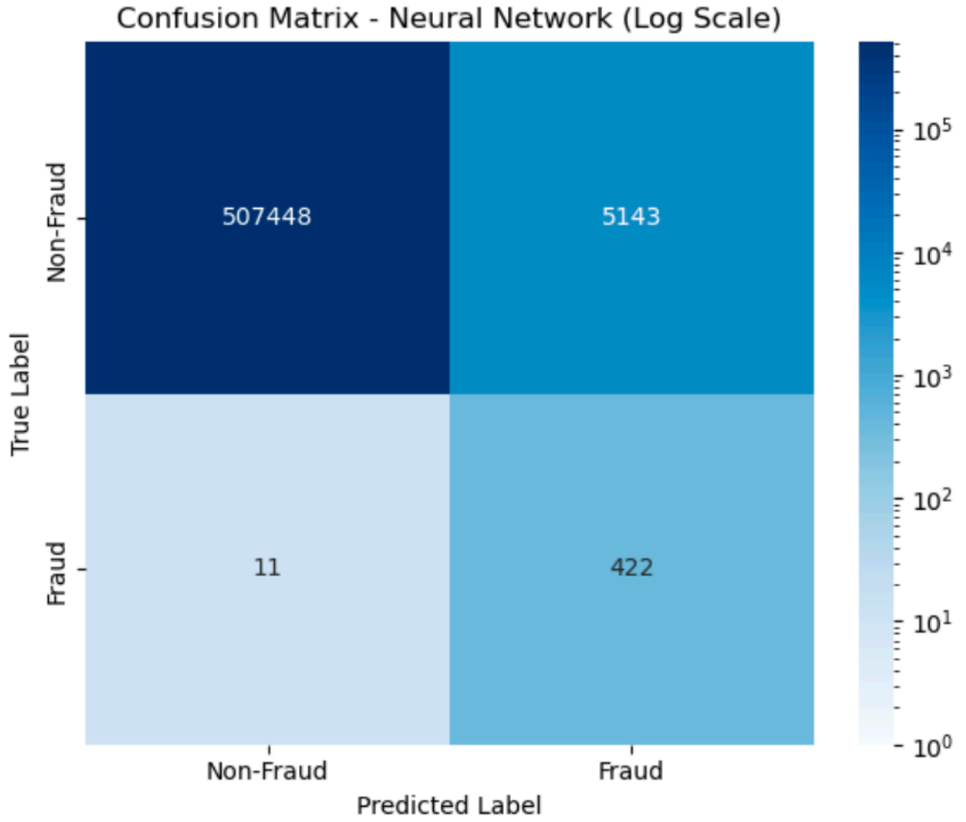
	Randomized Search (Method)	+ **5-fold Stratified CV** (Validation)	→ **ROC AUC** (Objective)
Feature	**Efficiently samples** hyperparameter combinations from defined ranges	**Robustly** evaluates performance and **preserves class ratios**	Measures model's ability to **distinguish between classes** across all thresholds.
Benefit	Faster than Grid Search for exploring large parameter spaces	Reliable performance estimate; **Stratification is crucial** for our imbalanced dataset	Excellent metric for imbalanced classification; **threshold-independent**
Goal	Identify high-performing parameter sets without exhaustive testing	Minimize bias in performance estimation during tuning	Select models/parameters best at distinguishing fraudulent transactions

Optimizing Model Performance

Model	Best Parameters / Configuration
Decision Tree	{'criterion': 'entropy', 'max_depth': 8, 'min_samples_leaf': 22, 'min_samples_split': 43}
Logistic Regression	{'C': 9.83755518841459, 'penalty': 'l1', 'solver': 'liblinear'}
Random Forest	{'max_depth': 15, 'max_features': 'log2', 'min_samples_leaf': 6, 'min_samples_split': 3, 'n_estimators': 241}
XGBoost	{'colsample_bytree': 0.8391599915244341, 'gamma': 0.4609371175115584, 'learning_rate': 0.03654775061557585, 'max_depth': 9, 'n_estimators': 301, 'subsample': 0.6180909155642152}
Neural Network	{'Architecture': 'MLP: Input(5) -> Dense(64,ReLU) -> Dropout(0.3) -> Dense(32,ReLU) -> Dropout(0.3) -> Dense(1,Sigmoid)', 'Optimizer': 'Adam (lr=0.001)', 'Loss': 'binary_crossentropy', 'Epochs': 15, 'Batch Size': 2048, 'Class Weights Applied': True}

Performance Evaluation





Performance Evaluation

Model	ROC AUC	F1 Score	Recall	Precision	Accuracy
Neural Network	**0.9964**	0.1407	**0.9746**	0.0758	0.9900
XGBoost	0.9938	**0.2171**	0.8961	**0.1235**	0.9945
Random Forest	0.9933	0.2152	0.8591	0.1230	**0.9947**
Decision Tree	0.9669	0.0182	0.9538	0.0092	0.9132
Logistic Regression	0.8203	0.0040	0.7021	0.0020	0.7058

- **Linear Nature**: Inability to capture non-linear decision boundaries essential for detecting fraud.
- **PCA Limitations**: PCA does not guarantee class separability for linear classifiers.
- **SMOTE Overfitting**: While recall improved, precision dropped drastically due to overfitting on synthetic minority class samples.
- **No Interaction Modeling**: Lacks native ability to model feature interactions, unlike tree-basedmethods.

Conclusion

Neural Network > XGBoost > Random Forest > Decision Tree > Logistic Regression

Future Enhancement Methods

- **Feature Engineering**:
  - Supervised Dimensionality Reduction using Linear Discriminant Analysis
  - Kernel PCA
  - Domain-specific feature engineering

- **Adopt Ensemble/Hybrid methods**:
  - Stacking Classifier and Voting Classifier
  - Deep Learning algorithms

Thank you!

Q&A

Appendix

Cleaning

```
# Import Libraries
import warnings; warnings.filterwarnings("ignore")
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
import kagglehub
```

```
path = kagglehub.dataset_download("jainilcoder/online-payment-fraud-detection", path=os.getcwd())
print(f"Dataset downloaded to: {path}")
```

Appendix

Understand the data

Print the details

[94]:
```
print("\n1. First 5 Rows of the Dataset:")
print(df.head())
```

```
1. First 5 Rows of the Dataset:
   step      type    amount     nameOrig  oldbalanceOrg  newbalanceOrig  \
0     1   PAYMENT   9839.64  C1231006815       170136.0       160296.36
1     1   PAYMENT   1864.28  C1666544295        21249.0        19384.72
2     1  TRANSFER    181.00  C1305486145          181.0            0.00
3     1  CASH_OUT    181.00   C840083671          181.0            0.00
4     1   PAYMENT  11668.14  C2048537720        41554.0        29885.86

      nameDest  oldbalanceDest  newbalanceDest  isFraud  isFlaggedFraud
0  M1979787155             0.0             0.0      0.0             0.0
1  M2044282225             0.0             0.0      0.0             0.0
2   C553264065             0.0             0.0      1.0             0.0
3    C38997010         21182.0             0.0      1.0             0.0
4  M1230701703             0.0             0.0      0.0             0.0
```

[95]:
```
print("\n2. Dataset Dimensions (Rows, Columns):")
print(df.shape)
```

```
2. Dataset Dimensions (Rows, Columns):
(2898977, 11)
```

[96]:
```
print("\n3. Column Information (Data Types, Non-Null Counts):")
df.info()
```

```
3. Column Information (Data Types, Non-Null Counts):
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2898977 entries, 0 to 2898976
Data columns (total 11 columns):
 #   Column          Dtype
---  ------          -----
 0   step            int64
 1   type            object
 2   amount          float64
 3   nameOrig        object
 4   oldbalanceOrg   float64
 5   newbalanceOrig  float64
 6   nameDest        object
 7   oldbalanceDest  float64
 8   newbalanceDest  float64
 9   isFraud         float64
 10  isFlaggedFraud  float64
dtypes: float64(7), int64(1), object(3)
memory usage: 243.3+ MB
```

Appendix

```
print("\n4. Missing Values Count per Column:")
print(df.isnull().sum())
```

```
4. Missing Values Count per Column:
step              0
type              0
amount            1
nameOrig          1
oldbalanceOrg     1
newbalanceOrig    1
nameDest          1
oldbalanceDest    1
newbalanceDest    1
isFraud           1
isFlaggedFraud    1
dtype: int64
```

```
df = df.dropna()
```

```
print("\n5. Descriptive Statistics for Numerical Columns:")
print(df.describe().T)
```

```
5. Descriptive Statistics for Numerical Columns:
                   count          mean           std  min        25%  \
step            2898976.0  1.205408e+02  7.495810e+01  1.0     37.000
amount          2898976.0  1.578971e+05  2.681763e+05  0.0  12550.235
oldbalanceOrg   2898976.0  8.489864e+05  2.926095e+06  0.0      0.000
newbalanceOrig  2898976.0  8.703853e+05  2.962512e+06  0.0      0.000
oldbalanceDest  2898976.0  9.943366e+05  2.300190e+06  0.0      0.000
newbalanceDest  2898976.0  1.102537e+06  2.384246e+06  0.0      0.000
isFraud         2898976.0  8.854851e-04  2.974393e-02  0.0      0.000
isFlaggedFraud  2898976.0  3.449494e-07  5.873239e-04  0.0      0.000

                       50%           75%          max
step                140.000  1.850000e+02       228.00
amount            76240.610  2.115956e+05  10000000.00
oldbalanceOrg     14710.000  1.143100e+05  38939424.03
newbalanceOrig        0.000  1.536623e+05  38946233.02
oldbalanceDest   137061.875  9.440713e+05  42655769.20
newbalanceDest   223511.895  1.130109e+06  42655769.20
isFraud               0.000  0.000000e+00         1.00
isFlaggedFraud        0.000  0.000000e+00         1.00
```

Appendix

```
print("\n6. Descriptive Statistics for Categorical Columns:")
print(df.describe(include=['object']))
```

```
6. Descriptive Statistics for Categorical Columns:
            type     nameOrig    nameDest
count    2898976      2898976     2898976
unique         5      2896983     1235172
top     CASH_OUT  C1999539787  C985934102
freq     1030687            3         103
```

```
print("\n7. Distribution of the Target Variable ('isFraud'):")
fraud_counts = df['isFraud'].value_counts()
print(fraud_counts)
fraud_percentage = (fraud_counts.get(1, 0) / df.shape[0]) * 100
print(f"\nPercentage of Fraudulent Transactions: {fraud_percentage:.4f}%")
```

```
7. Distribution of the Target Variable ('isFraud'):
isFraud
0.0    2896409
1.0       2567
Name: count, dtype: int64

Percentage of Fraudulent Transactions: 0.0885%
```

```
print("\n8. Distribution of Transaction Types ('type'):")
print(df['type'].value_counts())
```

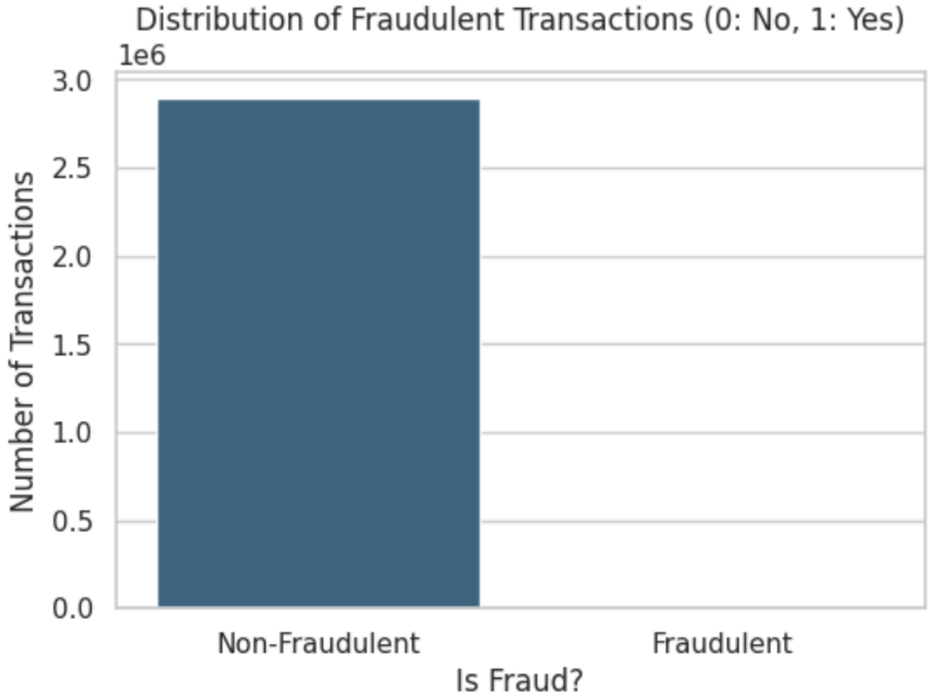
```
8. Distribution of Transaction Types ('type'):
type
CASH_OUT    1030687
PAYMENT      974138
CASH_IN      636005
TRANSFER     239912
DEBIT         18234
Name: count, dtype: int64
```

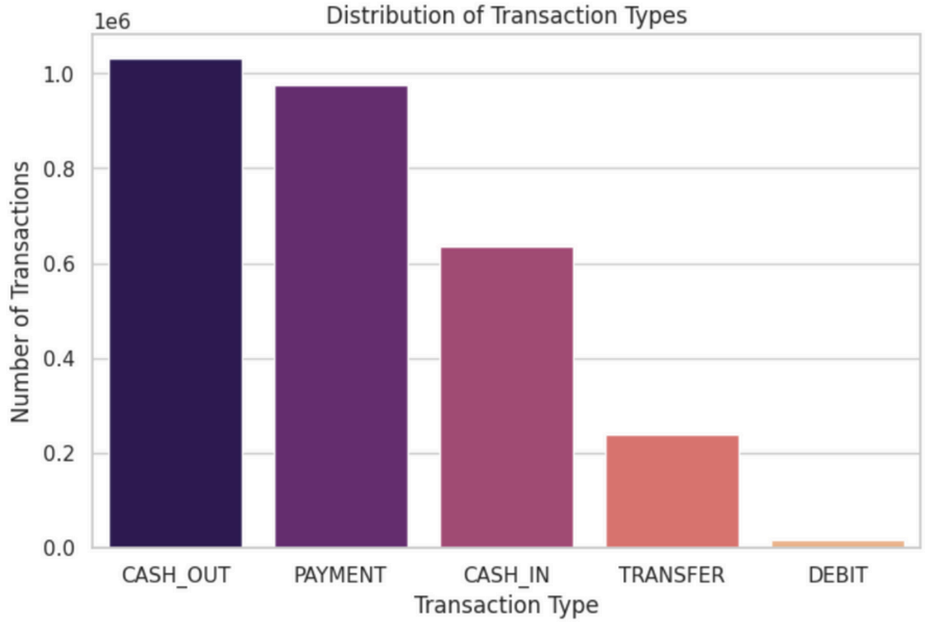
Visualizations

Appendix

Visualizations

```
# Set plot style
sns.set(style="whitegrid")

# Plot Fraud Distribution
plt.figure(figsize=(6, 4))
sns.countplot(x='isFraud', data=df, palette='viridis')
plt.title('Distribution of Fraudulent Transactions (0: No, 1: Yes)')
plt.xlabel("Is Fraud?")
plt.ylabel("Number of Transactions")
plt.xticks([0, 1], ['Non-Fraudulent', 'Fraudulent']) # More descriptive labels
plt.show()

# Plot Transaction Type Distribution
plt.figure(figsize=(8, 5))
sns.countplot(x='type', data=df, order=df['type'].value_counts().index, palette='magma') # Order bars by fre
plt.title('Distribution of Transaction Types')
plt.xlabel("Transaction Type")
plt.ylabel("Number of Transactions")
# plt.xticks(rotation=45) # Uncomment if labels overlap
plt.show()
```





Appendix

Data preprocessing

[105]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.decomposition import PCA
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
```

[109]:
```
columns_to_drop = ['nameOrig', 'nameDest', 'isFlaggedFraud']
columns_present = [col for col in columns_to_drop if col in df.columns]
df_processed = df.drop(columns=columns_present)
print(f"Dropped columns: {columns_present}")
print("Remaining columns:", df_processed.columns.tolist())
```

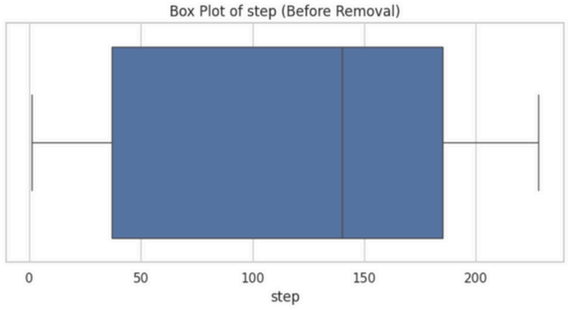
```
Dropped columns: ['nameOrig', 'nameDest', 'isFlaggedFraud']
Remaining columns: ['step', 'type', 'amount', 'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest', 'newbalanceDest', 'isFraud']
```

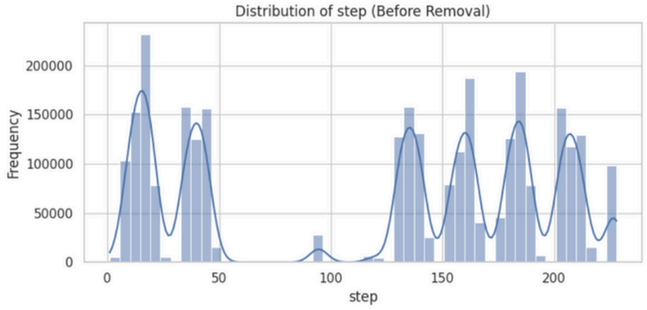
[110]:
```
numerical_features = df_processed.select_dtypes(include=np.number).columns.tolist()
if 'isFraud' in numerical_features:
    numerical_features.remove('isFraud') # Exclude target
categorical_features = df_processed.select_dtypes(include='object').columns.tolist()
print(f"Numerical features identified: {numerical_features}")
print(f"Categorical features identified: {categorical_features}")
```

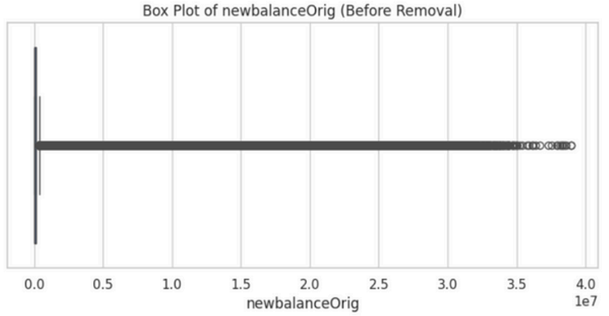
```
Numerical features identified: ['step', 'amount', 'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest', 'newbalanceDest']
Categorical features identified: ['type']
```

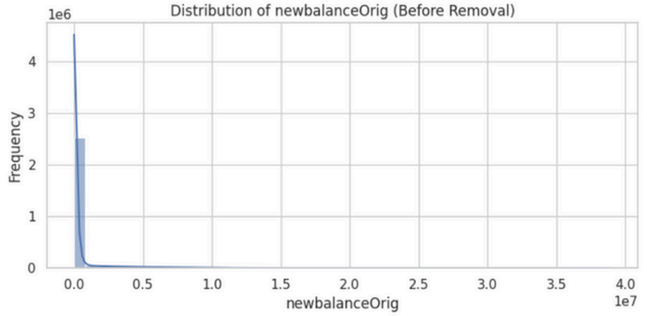
[112]:
```
for col in numerical_features:
    plt.figure(figsize=(15, 4))

    plt.subplot(1, 2, 1)
    sns.boxplot(x=df_processed[col])
    plt.title(f'Box Plot of {col} (Before Removal)')
    plt.xlabel(col)

    plt.subplot(1, 2, 2)
    sns.histplot(df_processed[col], kde=True, bins=50)
    plt.title(f'Distribution of {col} (Before Removal)')
    plt.xlabel(col)
    plt.ylabel('Frequency')

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

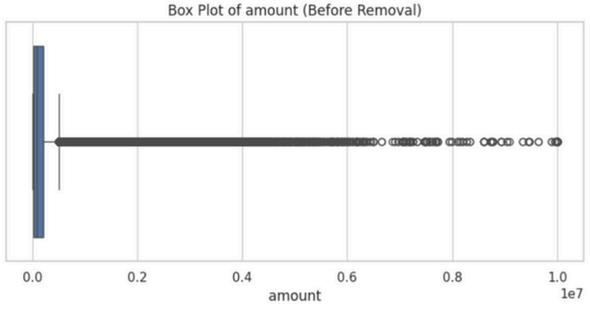
Appendix

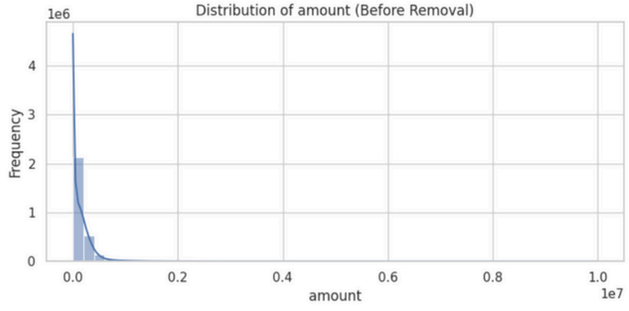

Box Plot of step (Before Removal)
step

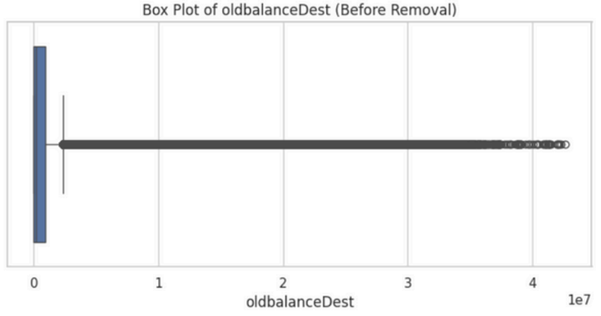

Distribution of step (Before Removal)
Frequency
step

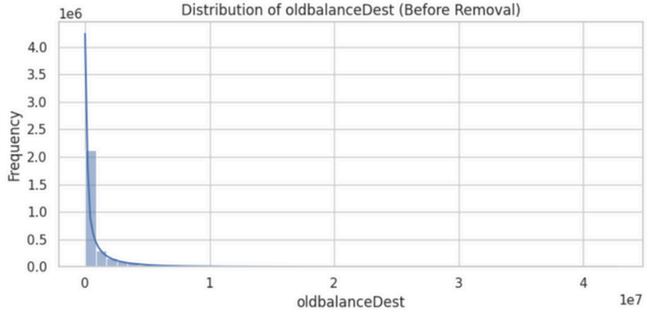

Box Plot of newbalanceOrig (Before Removal)
newbalanceOrig
1e7

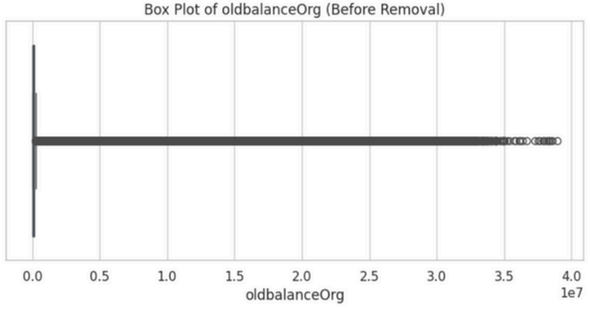

Distribution of newbalanceOrig (Before Removal)
1e6
Frequency
newbalanceOrig
1e7

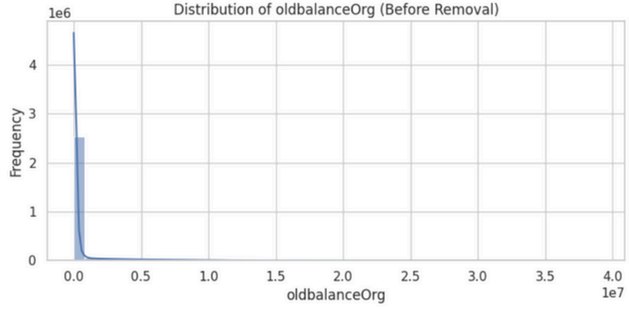

Box Plot of amount (Before Removal)
amount
1e7

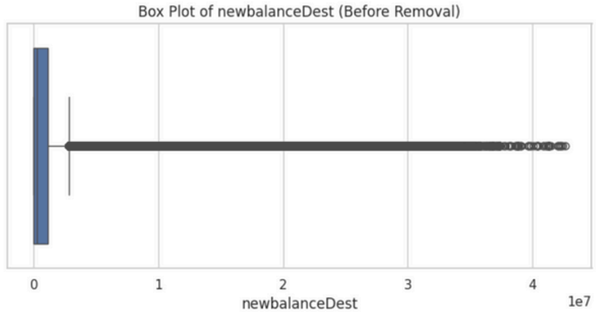

Distribution of amount (Before Removal)
1e6
Frequency
amount
1e7

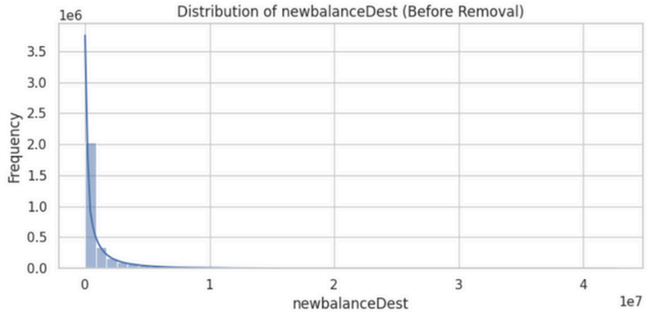

Box Plot of oldbalanceDest (Before Removal)
oldbalanceDest
1e7


Distribution of oldbalanceDest (Before Removal)
1e6
Frequency
oldbalanceDest
1e7


Box Plot of oldbalanceOrg (Before Removal)
oldbalanceOrg
1e7


Distribution of oldbalanceOrg (Before Removal)
1e6
Frequency
oldbalanceOrg
1e7


Box Plot of newbalanceDest (Before Removal)
newbalanceDest
1e7


Distribution of newbalanceDest (Before Removal)
1e6
Frequency
newbalanceDest
1e7

Appendix

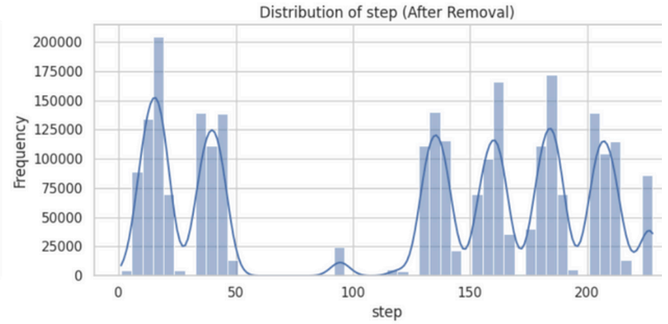
```python
df_no_outliers = df_processed.copy()
initial_rows = df_no_outliers.shape[0]
print(f"Shape before outlier removal: {df_no_outliers.shape}")

# Apply IQR filtering cumulatively
outliers_removed_details = {} # Dictionary to store count per column

for col in numerical_features:
    rows_before_col = df_no_outliers.shape[0]
    print(f"\nProcessing column: {col}")
    Q1 = df_no_outliers[col].quantile(0.25)
    Q3 = df_no_outliers[col].quantile(0.75)
    IQR_val = Q3 - Q1

    lower_bound = Q1 - 20 * IQR_val
    upper_bound = Q3 + 20 * IQR_val

    print(f"  Q1: {Q1:.2f}, Q3: {Q3:.2f}, IQR: {IQR_val:.2f}")
    print(f"  Lower Bound: {lower_bound:.2f}, Upper Bound: {upper_bound:.2f}")

    # Keep rows within the bounds for the current column
    df_no_outliers = df_no_outliers[
        (df_no_outliers[col] >= lower_bound) & (df_no_outliers[col] <= upper_bound)
    ]
    rows_after_col = df_no_outliers.shape[0]
    removed_count = rows_before_col - rows_after_col
    outliers_removed_details[col] = removed_count
    print(f"  Removed {removed_count} outliers based on {col}. Current shape: {df_no_outliers.shape}")

print("\n--- Outlier Removal Summary ---")
total_rows_removed = initial_rows - df_no_outliers.shape[0]
percentage_removed = (total_rows_removed / initial_rows) * 100 if initial_rows > 0 else 0
print(f"Final shape after outlier removal: {df_no_outliers.shape}")
print(f"Total number of rows removed: {total_rows_removed} ({percentage_removed:.2f}%)")
print("Outliers removed per feature (cumulative effect):")
for col, count in outliers_removed_details.items():
    print(f"  - {col}: {count}")

# Check if data remains
if df_no_outliers.empty:
    print("\nWarning: All data removed after outlier detection. IQR might be too strict for this dataset.")
    # Consider adjusting the 1.5 multiplier or using a different outlier method if this happens.
    # exit() # Optional: stop script if no data left
```

```
Shape before outlier removal: (2898976, 8)

Processing column: step
  Q1: 37.00, Q3: 185.00, IQR: 148.00
  Lower Bound: -2923.00, Upper Bound: 3145.00
  Removed 0 outliers based on step. Current shape: (2898976, 8)

Processing column: amount
  Q1: 12550.24, Q3: 211595.61, IQR: 199045.38
  Lower Bound: -3968357.31, Upper Bound: 4192503.16
  Removed 539 outliers based on amount. Current shape: (2898437, 8)

Processing column: oldbalanceOrg
  Q1: 0.00, Q3: 114253.04, IQR: 114253.04
  Lower Bound: -2285060.80, Upper Bound: 2399313.84
  Removed 263959 outliers based on oldbalanceOrg. Current shape: (2634478, 8)

Processing column: newbalanceOrig
  Q1: 0.00, Q3: 62472.80, IQR: 62472.80
  Lower Bound: -1249456.05, Upper Bound: 1311928.85
  Removed 61707 outliers based on newbalanceOrig. Current shape: (2572771, 8)

Processing column: oldbalanceDest
  Q1: 0.00, Q3: 782507.89, IQR: 782507.89
  Lower Bound: -15650157.80, Upper Bound: 16432665.69
  Removed 7599 outliers based on oldbalanceDest. Current shape: (2565172, 8)

Processing column: newbalanceDest
  Q1: 0.00, Q3: 1014203.67, IQR: 1014203.67
  Lower Bound: -20284073.30, Upper Bound: 21298276.96
  Removed 55 outliers based on newbalanceDest. Current shape: (2565117, 8)

--- Outlier Removal Summary ---
Final shape after outlier removal: (2565117, 8)
Total number of rows removed: 333859 (11.52%)
Outliers removed per feature (cumulative effect):
  - step: 0
  - amount: 539
  - oldbalanceOrg: 263959
  - newbalanceOrig: 61707
  - oldbalanceDest: 7599
  - newbalanceDest: 55
```

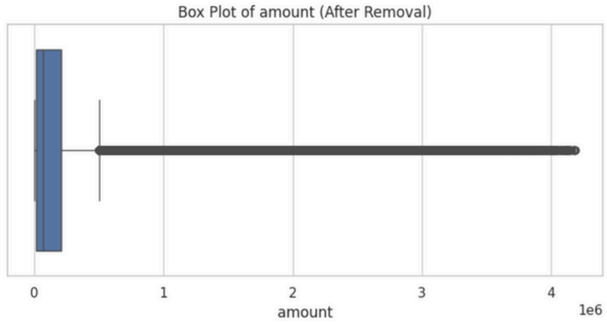
Appendix

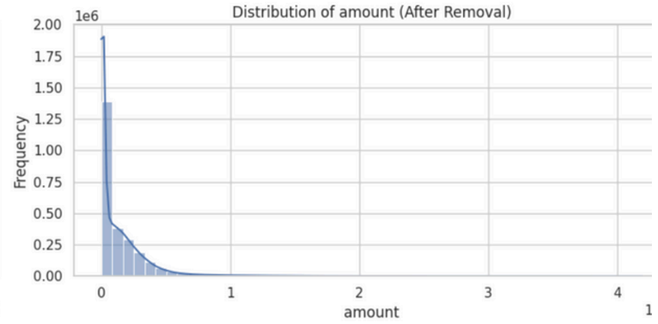
```
print("\n--- Visualizing Data After Outlier Removal ---")

# Check if the DataFrame is empty after removal
if df_no_outliers.empty:
    print("DataFrame 'df_no_outliers' is empty. Cannot visualize.")
else:
    print(f"Visualizing distributions for {len(numerical_features)} numerical features in df_no_outliers (shape: {df_no_outliers.shape})...")
    for col in numerical_features:
        # Ensure the column still exists in the dataframe
        if col in df_no_outliers.columns:
            print(f"Plotting for column: {col}")
            plt.figure(figsize=(15, 4)) # Create a new figure for each column

            # 1. Box Plot (After Removal)
            plt.subplot(1, 2, 1) # 1 row, 2 columns, 1st subplot
            try:
                sns.boxplot(x=df_no_outliers[col])
                plt.title(f'Box Plot of {col} (After Removal)')
                plt.xlabel(col)
            except Exception as e:
                print(f"  Could not generate box plot for {col}: {e}")


            # 2. Histogram / KDE Plot (After Removal)
            plt.subplot(1, 2, 2) # 1 row, 2 columns, 2nd subplot
            try:
                # Check if there's enough variance to plot KDE
                if df_no_outliers[col].nunique() > 1:
                    sns.histplot(df_no_outliers[col], kde=True, bins=50)
                else:
                    # If only one unique value, just plot histogram
                    sns.histplot(df_no_outliers[col], kde=False, bins=1)
                plt.title(f'Distribution of {col} (After Removal)')
                plt.xlabel(col)
                plt.ylabel('Frequency')
            except Exception as e:
                print(f"  Could not generate histogram/KDE for {col}: {e}")


            plt.tight_layout() # Adjust spacing
            plt.show() # Display the plots for the current column
        else:
            print(f"  Column '{col}' not found in df_no_outliers. Skipping visualization.")

print("\n--- Visualization after outlier removal complete ---")
```

```
--- Visualizing Data After Outlier Removal ---
Visualizing distributions for 6 numerical features in df_no_outliers (shape: (2565117, 8))...
Plotting for column: step
```

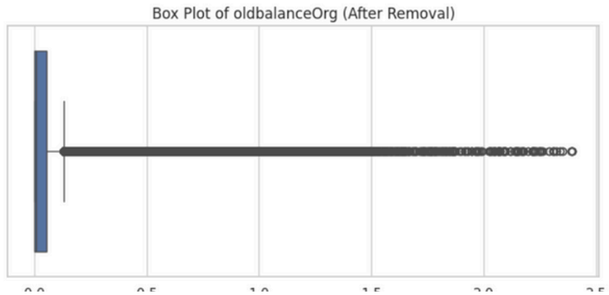


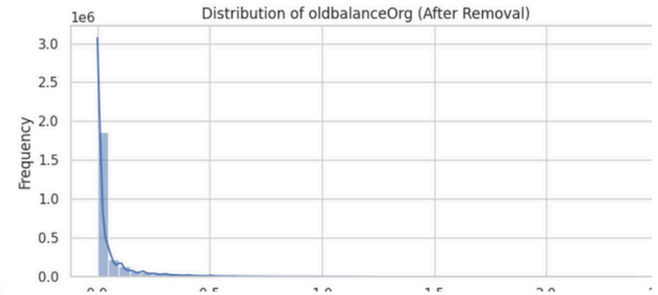

```
Plotting for column: amount
```





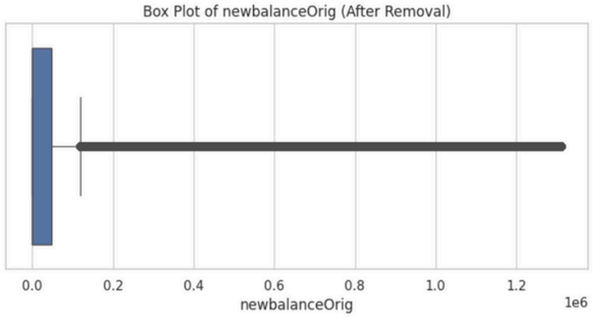

```
Plotting for column: oldbalanceOrg
```

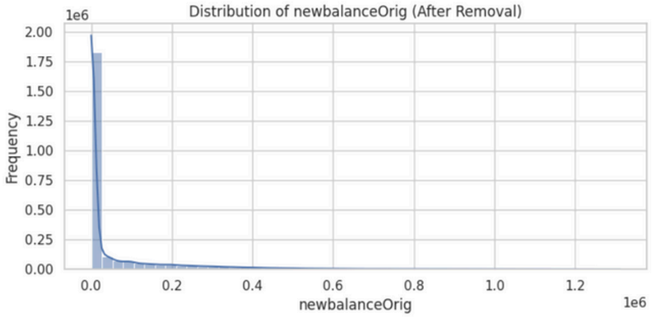




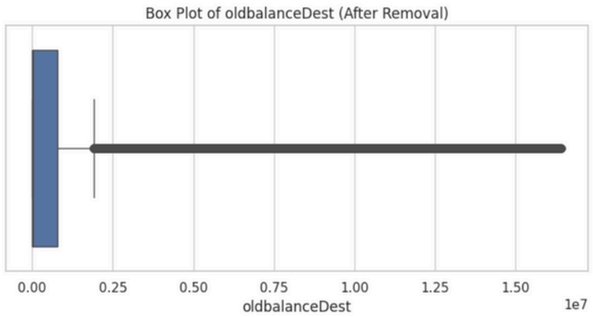
Appendix

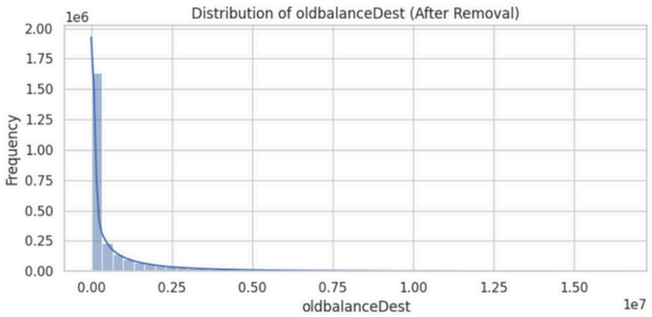
```
Plotting for column: newbalanceOrig
```



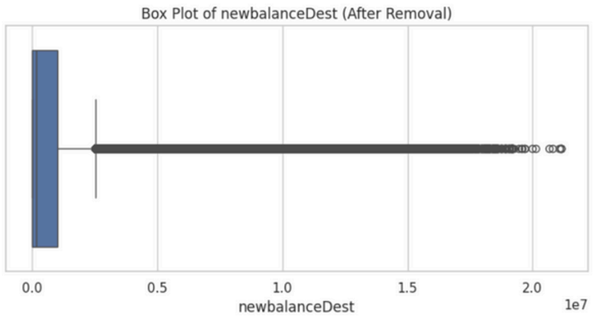


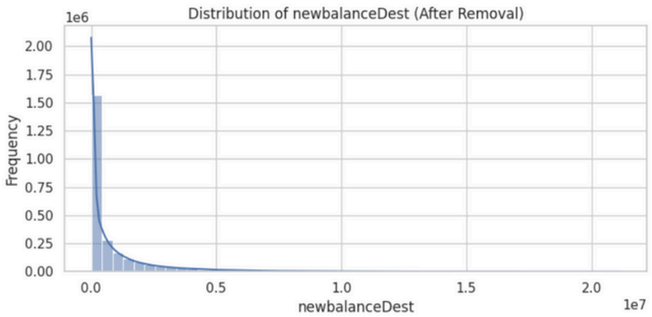

```
Plotting for column: oldbalanceDest
```






```
Plotting for column: newbalanceDest
```





Appendix

```
# define feature and target
X = df_no_outliers.drop('isFraud', axis=1)
y = df_no_outliers['isFraud']

print(f"\nShape of features (X) before preprocessing: {X.shape}")
print(f"Shape of target (y): {y.shape}")
```

```
Shape of features (X) before preprocessing: (2565117, 7)
Shape of target (y): (2565117,)
```

handle numerical and categorical features

```
# Identify numerical and categorical features
numerical_features = X.select_dtypes(include=np.number).columns.tolist()
categorical_features = X.select_dtypes(include='object').columns.tolist()
print(f"\nNumerical features identified: {numerical_features}")
print(f"Categorical features identified: {categorical_features}")
```

```
Numerical features identified: ['step', 'amount', 'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest', 'newbalanceDest']
Categorical features identified: ['type']
```

```
# Create preprocessing pipelines
numerical_transformer = StandardScaler()
categorical_transformer = OneHotEncoder(handle_unknown='ignore', drop='first', sparse_output=False)
```

```
# Create ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)
    ],
    remainder='passthrough'
)
```

```
# Fit and transform the *entire* feature set X
X_processed = preprocessor.fit_transform(X)
```

```
feature_names_out = preprocessor.get_feature_names_out()
feature_names_out
```

```
array(['num__step', 'num__amount', 'num__oldbalanceOrg',
       'num__newbalanceOrig', 'num__oldbalanceDest',
       'num__newbalanceDest', 'cat__type_CASH_OUT', 'cat__type_DEBIT',
       'cat__type_PAYMENT', 'cat__type_TRANSFER'], dtype=object)
```

Appendix

handle numerical and categorical features

```
# Identify numerical and categorical features
numerical_features = X.select_dtypes(include=np.number).columns.tolist()
categorical_features = X.select_dtypes(include='object').columns.tolist()
print(f"\nNumerical features identified: {numerical_features}")
print(f"Categorical features identified: {categorical_features}")
```

```
Numerical features identified: ['step', 'amount', 'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest', 'newbalanceDest']
Categorical features identified: ['type']
```

```
# Create preprocessing pipelines
numerical_transformer = StandardScaler()
categorical_transformer = OneHotEncoder(handle_unknown='ignore', drop='first', sparse_output=False)
```

```
# Create ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)
    ],
    remainder='passthrough'
)
```

```
# Fit and transform the *entire* feature set X
X_processed = preprocessor.fit_transform(X)
```

```
feature_names_out = preprocessor.get_feature_names_out()
feature_names_out
```

```
array(['num__step', 'num__amount', 'num__oldbalanceOrg',
       'num__newbalanceOrig', 'num__oldbalanceDest',
       'num__newbalanceDest', 'cat__type_CASH_OUT', 'cat__type_DEBIT',
       'cat__type_PAYMENT', 'cat__type_TRANSFER'], dtype=object)
```

Appendix

PCA

```
pca_analyzer = PCA(random_state=42)
pca_analyzer.fit(X_processed)
cumulative_explained_variance = np.cumsum(pca_analyzer.explained_variance_ratio_)
n_components_95 = np.argmax(cumulative_explained_variance >= 0.95) + 1
print(f"Number of components needed for >= 95% variance: {n_components_95}")
```

```
Number of components needed for >= 95% variance: 5
```

```
pca = PCA(n_components=n_components_95, random_state=42)
X_pca = pca.fit_transform(X_processed)
```

```
print(f"\nShape of X after PCA: {X_pca.shape}")
pca_columns = [f'PC{i+1}' for i in range(n_components_95)]
X_pca_df = pd.DataFrame(X_pca, columns=pca_columns, index=X.index)
print("First 5 rows of PCA-transformed data:\n", X_pca_df.head())
```

```
Shape of X after PCA: (2565117, 5)
First 5 rows of PCA-transformed data:
          PC1       PC2       PC3       PC4       PC5
0 -1.086998  0.742401 -1.522433 -0.420936 -0.475533
1 -0.890424 -0.465860 -1.590329 -0.473069 -0.551599
2 -0.637245 -0.604853 -1.619794 -0.199640 -0.113378
3 -0.602021 -0.661038 -1.617156 -0.382883  0.793343
4 -0.898815 -0.333727 -1.585785 -0.437364 -0.542048
```

Appendix

Data split

```
X_train, X_test, y_train, y_test = train_test_split(
    X_pca,
    y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
print(f"\nShape of X_train: {X_train.shape}")
print(f"Shape of X_test: {X_test.shape}")
print(f"Shape of y_train: {y_train.shape}")
print(f"Shape of y_test: {y_test.shape}")
print(f"Fraud proportion in y_train:\n{y_train.value_counts(normalize=True)}")
print(f"Fraud proportion in y_test:\n{y_test.value_counts(normalize=True)}")
```

```
Shape of X_train: (2052093, 5)
Shape of X_test: (513024, 5)
Shape of y_train: (2052093,)
Shape of y_test: (513024,)
Fraud proportion in y_train:
isFraud
0.0    0.999155
1.0    0.000845
Name: proportion, dtype: float64
Fraud proportion in y_test:
isFraud
0.0    0.999156
1.0    0.000844
Name: proportion, dtype: float64
```

Appendix

resampling

```
from imblearn.over_sampling import SMOTE
print("\n--- 8. Resampling Training Data with SMOTE ---")
print(f"Class distribution in y_train before SMOTE:\n{y_train.value_counts()}")

smote = SMOTE(random_state=42) # Use n_jobs for potential speedup if available
start_smote_time = time.time()
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
end_smote_time = time.time()

print(f"SMOTE completed in {end_smote_time - start_smote_time:.2f} seconds.")
print(f"Shape of X_train_resampled: {X_train_resampled.shape}")
print(f"Shape of y_train_resampled: {y_train_resampled.shape}")
print(f"Class distribution in y_train_resampled after SMOTE:\n{pd.Series(y_train_resampled).value_counts()}"
```

```
--- 8. Resampling Training Data with SMOTE ---
Class distribution in y_train before SMOTE:
isFraud
0.0    2050360
1.0       1733
Name: count, dtype: int64
SMOTE completed in 0.26 seconds.
Shape of X_train_resampled: (4100720, 5)
Shape of y_train_resampled: (4100720,)
Class distribution in y_train_resampled after SMOTE:
isFraud
0.0    2050360
1.0    2050360
Name: count, dtype: int64
```

Appendix

Model training

```
import time
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    roc_auc_score,
    classification_report,
    confusion_matrix,
    make_scorer
)
import warnings; warnings.filterwarnings("ignore")
# Models
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb

# Tuning and CV
from sklearn.model_selection import RandomizedSearchCV, StratifiedKFold

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Helper for parameter distributions
from scipy.stats import uniform, randint
```

Appendix

Define Parameter Distributions for RandomizedSearch

```
### setup
from sklearn import set_config
set_config(display='text') # Use text display for estimators to avoid fitting errors on display
```

```
param_dist = {
    "Decision Tree": {
        'criterion': ['gini', 'entropy'],
        'max_depth': randint(3, 20),
        'min_samples_split': randint(2, 50),
        'min_samples_leaf': randint(1, 30)
    },
    "Logistic Regression": {
        'C': uniform(0.1, 10),
        'penalty': ['l1', 'l2'],
        'solver': ['liblinear']
    },
    "Random Forest": {
        'n_estimators': randint(50, 300),
        'max_depth': [None] + list(randint(5, 30).rvs(5)),
        'min_samples_split': randint(2, 50),
        'min_samples_leaf': randint(1, 30),
        'max_features': ['sqrt', 'log2', None]
    },
    "XGBoost": {
        'n_estimators': randint(50, 400),
        'learning_rate': uniform(0.01, 0.3),
        'max_depth': randint(3, 15),
        'subsample': uniform(0.6, 0.4),
        'colsample_bytree': uniform(0.6, 0.4),
        'gamma': uniform(0, 0.5)
    }
}
base_models = {
    "Decision Tree": DecisionTreeClassifier(random_state=42, class_weight='balanced'),
    "Logistic Regression": LogisticRegression(random_state=42, class_weight='balanced', max_iter=1000),
    #"SVM": SVC(random_state=42, class_weight='balanced', probability=True),
    "Random Forest": RandomForestClassifier(random_state=42, class_weight='balanced', n_jobs=-1),
    "XGBoost": xgb.XGBClassifier(random_state=42, eval_metric='logloss')
}
# Calculate scale_pos_weight for XGBoost based on the training data
fraud_count_train = np.sum(y_train == 1)
non_fraud_count_train = np.sum(y_train == 0)
scale_pos_weight = non_fraud_count_train / fraud_count_train
print(f"Calculated scale_pos_weight for XGBoost (using training data): {scale_pos_weight:.2f}")

# Add a semicolon at the end to suppress output in interactive environments
base_models["XGBoost"].set_params(scale_pos_weight=scale_pos_weight);
```

```
Calculated scale_pos_weight for XGBoost (using training data): 1183.13
```

Appendix

5 fold CV

```
import warnings; warnings.filterwarnings("ignore")
print("\n--- Starting Hyperparameter Tuning with 5-Fold Cross-Validation ---")

# --- Setup Cross-Validation ---
cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
print(f"Using Stratified K-Fold Cross-Validation with {cv_strategy.get_n_splits()} splits.")

# --- Store Best Estimators and Tuning Results ---
best_estimators = {}
tuning_results_list = []

# --- Run Randomized Search for each Scikit-learn/XGBoost model ---
n_iter_search = 20 # Number of parameter settings sampled per model. Adjust as needed.
print(f"Running RandomizedSearchCV with n_iter={n_iter_search} for each model.")

# Define the primary scoring metric for RandomizedSearchCV
# roc_auc is generally good for imbalanced classification.
# Alternatives: 'f1', 'recall', 'precision'
tuning_scoring_metric = 'roc_auc'
print(f"Tuning scoring metric: {tuning_scoring_metric}")


for name in param_dist.keys(): # Iterate through models defined in param_dist
    print(f"\n--- Tuning {name} ---")
    start_time = time.time()

    # Ensure solver compatibility for Logistic Regression if penalty='l1' is possible
    # 'liblinear' supports both L1 and L2, which is specified in param_dist
    if name == "Logistic Regression":
        # Check if the base model needs the solver explicitly set if not already 'liblinear'
        if base_models[name].solver != 'liblinear':
             print(f"Updating Logistic Regression solver to 'liblinear' for L1/L2 compatibility.")
             base_models[name].set_params(solver='liblinear')

    # Setup RandomizedSearchCV
    random_search = RandomizedSearchCV(
        estimator=base_models[name],         # Base model instance
        param_distributions=param_dist[name],# Parameter space to search
        n_iter=n_iter_search,                # Number of iterations (samples)
        scoring=tuning_scoring_metric,       # Metric to optimize
        cv=cv_strategy,                      # Cross-validation strategy
        n_jobs=-1,                           # Use all available CPU cores
        random_state=42,                     # Reproducibility for sampling parameters
        verbose=1                            # Show progress (0=silent, 1=some, 2=more)
        # refit=True is the default, meaning the best estimator found is refit on the whole training set
    )
```

Appendix

```
    # Fit RandomizedSearchCV on the training data
    # This performs the search and finds the best parameters via cross-validation
    random_search.fit(X_train, y_train)

    tuning_time = time.time() - start_time
    print(f"Tuning completed in {tuning_time:.2f} seconds.")

    # Store the best estimator found (already refit on the full training data)
    best_estimators[name] = random_search.best_estimator_
    best_params = random_search.best_params_
    best_score = random_search.best_score_ # Mean cross-validated score of the best_estimator

    print(f"Best Parameters for {name}: {best_params}")
    print(f"Best Cross-Validation {tuning_scoring_metric.upper()} for {name}: {best_score:.4f}")

    # Store tuning summary
    tuning_results_list.append({
        "Model": name,
        f"Best CV {tuning_scoring_metric.upper()}": best_score,
        "Best Params": best_params,
        "Tuning Time (s)": tuning_time
    })

# Display Tuning Summary for the searched models
print("\n--- Hyperparameter Tuning Summary (Scikit-learn/XGBoost) ---")
if tuning_results_list: # Check if list is not empty
    tuning_summary_df = pd.DataFrame(tuning_results_list)
    tuning_summary_df.set_index("Model", inplace=True)
    print(tuning_summary_df[[f"Best CV {tuning_scoring_metric.upper()}", 'Tuning Time (s)']])
else:
    print("No tuning results available.")
    tuning_summary_df = pd.DataFrame() # Create empty df to avoid errors later


# --- Train and Evaluate Neural Network (Separately) ---
# Neural network tuning often uses different libraries (like KerasTuner) or manual experimentation.
# Here, we train a single architecture defined below.
print("\n--- Training and Evaluating Neural Network (Fixed Architecture) ---")

# Define NN parameters
input_shape = (X_train.shape[1],) # Number of features after PCA
epochs = 15 # Adjust epochs based on observing training/validation loss curves if needed
batch_size = 2048 # Larger batch size can speed up training on large datasets
print(f"NN Parameters: epochs={epochs}, batch_size={batch_size}, input_shape={input_shape}")

# Define class weights for Keras to handle imbalance (using counts from y_train)
# Ensure fraud_count_train and non_fraud_count_train were calculated correctly earlier
if fraud_count_train > 0: # Avoid division by zero
    total_samples = len(y_train)
    weight_for_0 = (1 / non_fraud_count_train) * (total_samples / 2.0)
```

Appendix

```
    # Fit RandomizedSearchCV on the training data
    # This performs the search and finds the best parameters via cross-validation
    random_search.fit(X_train, y_train)

    tuning_time = time.time() - start_time
    print(f"Tuning completed in {tuning_time:.2f} seconds.")

    # Store the best estimator found (already refit on the full training data)
    best_estimators[name] = random_search.best_estimator_
    best_params = random_search.best_params_
    best_score = random_search.best_score_ # Mean cross-validated score of the best_estimator

    print(f"Best Parameters for {name}: {best_params}")
    print(f"Best Cross-Validation {tuning_scoring_metric.upper()} for {name}: {best_score:.4f}")

    # Store tuning summary
    tuning_results_list.append({
        "Model": name,
        f"Best CV {tuning_scoring_metric.upper()}": best_score,
        "Best Params": best_params,
        "Tuning Time (s)": tuning_time
    })

# Display Tuning Summary for the searched models
print("\n--- Hyperparameter Tuning Summary (Scikit-learn/XGBoost) ---")
if tuning_results_list: # Check if list is not empty
    tuning_summary_df = pd.DataFrame(tuning_results_list)
    tuning_summary_df.set_index("Model", inplace=True)
    print(tuning_summary_df[[f"Best CV {tuning_scoring_metric.upper()}", 'Tuning Time (s)']])
else:
    print("No tuning results available.")
    tuning_summary_df = pd.DataFrame() # Create empty df to avoid errors later


# --- Train and Evaluate Neural Network (Separately) ---
# Neural network tuning often uses different libraries (like KerasTuner) or manual experimentation.
# Here, we train a single architecture defined below.
print("\n--- Training and Evaluating Neural Network (Fixed Architecture) ---")

# Define NN parameters
input_shape = (X_train.shape[1],) # Number of features after PCA
epochs = 15 # Adjust epochs based on observing training/validation loss curves if needed
batch_size = 2048 # Larger batch size can speed up training on large datasets
print(f"NN Parameters: epochs={epochs}, batch_size={batch_size}, input_shape={input_shape}")

# Define class weights for Keras to handle imbalance (using counts from y_train)
# Ensure fraud_count_train and non_fraud_count_train were calculated correctly earlier
if fraud_count_train > 0: # Avoid division by zero
    total_samples = len(y_train)
    weight_for_0 = (1 / non_fraud_count_train) * (total_samples / 2.0)
```

Appendix

```
# Train the model
print("\nTraining Neural Network...")
start_time = time.time()
history = nn_model.fit(
    X_train,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.1, # Use 10% of training data for validation during training
    class_weight=nn_class_weight, # Apply class weights if calculated
    verbose=1 # Show training progress per epoch
)
nn_training_time = time.time() - start_time
print(f"NN Training completed in {nn_training_time:.2f} seconds.")

# Store the trained NN model in the best_estimators dictionary for consistency in the next step
# Although not found via RandomizedSearch, we treat it as the "best" NN we trained.
best_estimators["Neural Network"] = nn_model

print("\n--- Model Tuning and NN Training Complete ---")
```

Appendix

```
--- Starting Hyperparameter Tuning with 5-Fold Cross-Validation ---
Using Stratified K-Fold Cross-Validation with 5 splits.
Running RandomizedSearchCV with n_iter=20 for each model.
Tuning scoring metric: roc_auc

--- Tuning Decision Tree ---
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Tuning completed in 93.04 seconds.
Best Parameters for Decision Tree: {'criterion': 'entropy', 'max_depth': 8, 'min_samples_leaf': 22, 'min_sa
mples_split': 43}
Best Cross-Validation ROC_AUC for Decision Tree: 0.9619

--- Tuning Logistic Regression ---
Updating Logistic Regression solver to 'liblinear' for L1/L2 compatibility.
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Tuning completed in 17.35 seconds.
Best Parameters for Logistic Regression: {'C': np.float64(9.83755518841459), 'penalty': 'l1', 'solver': 'li
blinear'}
Best Cross-Validation ROC_AUC for Logistic Regression: 0.8193

--- Tuning Random Forest ---
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Tuning completed in 746.94 seconds.
Best Parameters for Random Forest: {'max_depth': np.int64(15), 'max_features': 'log2', 'min_samples_leaf':
6, 'min_samples_split': 3, 'n_estimators': 241}
Best Cross-Validation ROC_AUC for Random Forest: 0.9871

--- Tuning XGBoost ---
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Tuning completed in 145.28 seconds.
Best Parameters for XGBoost: {'colsample_bytree': np.float64(0.8391599915244341), 'gamma': np.float64(0.460
9371175115584), 'learning_rate': np.float64(0.03654775061557585), 'max_depth': 9, 'n_estimators': 301, 'sub
sample': np.float64(0.6180909155642152)}
Best Cross-Validation ROC_AUC for XGBoost: 0.9899

--- Hyperparameter Tuning Summary (Scikit-learn/XGBoost) ---
                     Best CV ROC_AUC  Tuning Time (s)
Model
Decision Tree               0.961877        93.035247
Logistic Regression         0.819328        17.347903
Random Forest               0.987138       746.944452
XGBoost                     0.989931       145.283262

--- Training and Evaluating Neural Network (Fixed Architecture) ---
NN Parameters: epochs=15, batch_size=2048, input_shape=(5,)
Calculated class weights for Keras: {0: np.float64(0.5004226087126163), 1: np.float64(592.0637622619735)}

Neural Network Model Summary:
Model: "Fraud_Detection_NN"
```

Appendix

```
Best Cross-Validation ROC_AUC for Random Forest: 0.9871

--- Tuning XGBoost ---
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Tuning completed in 145.28 seconds.
Best Parameters for XGBoost: {'colsample_bytree': np.float64(0.8391599915244341), 'gamma': np.float64(0.460
9371175115584), 'learning_rate': np.float64(0.03654775061557585), 'max_depth': 9, 'n_estimators': 301, 'sub
sample': np.float64(0.6180909155642152)}
Best Cross-Validation ROC_AUC for XGBoost: 0.9899

--- Hyperparameter Tuning Summary (Scikit-learn/XGBoost) ---
                    Best CV ROC_AUC  Tuning Time (s)
Model
Decision Tree              0.961877        93.035247
Logistic Regression        0.819328        17.347903
Random Forest              0.987138       746.944452
XGBoost                    0.989931       145.283262

--- Training and Evaluating Neural Network (Fixed Architecture) ---
NN Parameters: epochs=15, batch_size=2048, input_shape=(5,)
Calculated class weights for Keras: {0: np.float64(0.5004226087126163), 1: np.float64(592.0637622619735)}

Neural Network Model Summary:
```

Model: "Fraud_Detection_NN"

Layer (type)	Output Shape	Param #
Dense_1 (Dense)	(None, 64)	384
Dropout_1 (Dropout)	(None, 64)	0
Dense_2 (Dense)	(None, 32)	2,080
Dropout_2 (Dropout)	(None, 32)	0
Output_Sigmoid (Dense)	(None, 1)	33

Total params: 2,497 (9.75 KB)

Trainable params: 2,497 (9.75 KB)

Non-trainable params: 0 (0.00 B)

Appendix

Evaluation using test set and comparison (models stored in the previous step)

```
from sklearn.metrics import roc_curve
print("\n--- Evaluating Tuned Models and Trained NN on the Hold-Out Test Set ---")

final_results_list = []
model_probabilities = {} # To store probabilities for ROC curve

# --- Evaluate Scikit-Learn / XGBoost Models ---
for name, model in best_estimators.items():
    # Skip the Neural Network for now, handle it separately as it's a Keras model
    if name == "Neural Network":
        continue

    print(f"\nEvaluating {name}...")
    start_time = time.time()

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

    # Get probabilities for ROC AUC calculation
    try:
        y_pred_proba = model.predict_proba(X_test)[:, 1] # Probability of positive class (1)
        model_probabilities[name] = y_pred_proba
    except AttributeError:
        print(f"Warning: {name} does not have predict_proba method. ROC AUC cannot be calculated accurately.")
        # Assign 0/1 predictions if predict_proba is unavailable; ROC AUC will be less meaningful
        y_pred_proba = y_pred
        model_probabilities[name] = None # Mark as unavailable

    eval_time = time.time() - start_time

    # Calculate Metrics
    accuracy = accuracy_score(y_test, y_pred)
    # Use zero_division=0 for precision/f1 to avoid errors if no positive predictions are made
    precision = precision_score(y_test, y_pred, pos_label=1, zero_division=0)
    recall = recall_score(y_test, y_pred, pos_label=1)
    f1 = f1_score(y_test, y_pred, pos_label=1)
    roc_auc = np.nan # Initialize as NaN

    if model_probabilities[name] is not None:
        try:
            roc_auc = roc_auc_score(y_test, y_pred_proba)
        except ValueError as e:
             print(f"Could not calculate ROC AUC for {name}: {e}")
```

Appendix

```
    if model_probabilities[name] is not None:
        try:
            roc_auc = roc_auc_score(y_test, y_pred_proba)
        except ValueError as e:
             print(f"Could not calculate ROC AUC for {name}: {e}")
             # roc_auc remains NaN

    # Retrieve tuning time (if available in tuning_summary_df)
    try:
        # Use .get() with a default value in case the model name isn't in the index
        training_time = tuning_summary_df.loc[name].get('Tuning Time (s)', 0)
    except KeyError:
        training_time = 0 # Assign 0 if model not found in summary (shouldn't happen ideally)
        print(f"Warning: Tuning time not found for {name} in tuning_summary_df.")


    # Store results
    final_results_list.append({
        "Model": name,
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1 Score": f1,
        "ROC AUC": roc_auc,
        "Training/Tuning Time (s)": training_time,
        "Prediction Time (s)": eval_time
    })
    print(f"Evaluation completed for {name} in {eval_time:.2f} seconds.")
    # Optional: print intermediate results
    # print(f"  Accuracy: {accuracy:.4f}, Precision: {precision:.4f}, Recall: {recall:.4f}, F1: {f1:.4f}, ROC AUC: {roc_auc:.4f}")


# --- Evaluate Neural Network ---
if "Neural Network" in best_estimators:
    print("\nEvaluating Neural Network...")
    start_time = time.time()
    nn_model_eval = best_estimators["Neural Network"] # Retrieve the trained Keras model

    # Keras evaluate method provides loss and compiled metrics
    nn_loss, nn_accuracy, nn_precision, nn_recall, nn_roc_auc = nn_model_eval.evaluate(
        X_test, y_test, verbose=0 # Evaluate silently
    )

    # Get probabilities and thresholded predictions for sklearn metrics (like F1, confusion matrix)
    y_pred_proba_nn = nn_model_eval.predict(X_test).flatten() # Get probabilities
    y_pred_nn = (y_pred_proba_nn > 0.5).astype(int) # Apply 0.5 threshold

    eval_time = time.time() - start_time

    # Calculate F1 score using sklearn for consistency
    nn_f1 = f1_score(y_test, y_pred_nn, pos_label=1, zero_division=0)
```

Appendix

```
    # Store NN probabilities for ROC curve
    model_probabilities["Neural Network"] = y_pred_proba_nn

    # Store NN results (use metrics from Keras evaluate where available, F1 from sklearn)
    final_results_list.append({
        "Model": "Neural Network",
        "Accuracy": nn_accuracy,
        "Precision": nn_precision, # From Keras evaluate
        "Recall": nn_recall,       # From Keras evaluate
        "F1 Score": nn_f1,         # From sklearn calculation
        "ROC AUC": nn_roc_auc,     # From Keras evaluate
        "Training/Tuning Time (s)": nn_training_time, # From NN training step
        "Prediction Time (s)": eval_time
    })
    print(f"Evaluation completed for Neural Network in {eval_time:.2f} seconds.")
    # print(f"  Accuracy: {nn_accuracy:.4f}, Precision: {nn_precision:.4f}, Recall: {nn_recall:.4f}, F1: {nn_f1:.4f}, ROC AUC: {nn_roc_auc:.4f}")


# --- Compare Final Results ---
print("\n--- Comparison of Final Model Results (After Tuning) ---")
if final_results_list: # Check if list is not empty
    final_results_df = pd.DataFrame(final_results_list)
    final_results_df.set_index("Model", inplace=True)

    # Sort by a chosen metric, e.g., ROC AUC or F1 Score. Handle NaNs.
    sort_metric = 'ROC AUC'
    final_results_df_sorted = final_results_df.sort_values(
        by=sort_metric, ascending=False, na_position='last'
    )
    print(f"Results sorted by {sort_metric} (descending):")
    # Display relevant columns, formatted
    print(final_results_df_sorted[[ 'ROC AUC', 'F1 Score', 'Recall', 'Precision', 'Accuracy', 'Prediction Time (s)', 'Training/Tuning Time (s)']].to_stri

    # Optionally save results
    # final_results_df_sorted.to_csv("final_model_comparison_results.csv")
    # print("\nFinal results saved to final_model_comparison_results.csv")

else:
    print("No final results to display.")
    final_results_df_sorted = pd.DataFrame() # Ensure it exists as an empty df


# --- Plot ROC Curves ---
print("\n--- Generating ROC Curve Comparison Plot ---")

plt.figure(figsize=(12, 9)) # Make figure slightly larger

# Iterate through the results DataFrame, sorted by AUC, to plot curves
# This ensures the legend order matches performance (best first)
for model_name in final_results_df_sorted.index:
    if model_name in model_probabilities and model_probabilities[model_name] is not None:
```

Appendix

```
# This ensures the legend order matches performance (best first)
for model_name in final_results_df_sorted.index:
    if model_name in model_probabilities and model_probabilities[model_name] is not None:
        probabilities = model_probabilities[model_name]
        fpr, tpr, thresholds = roc_curve(y_test, probabilities)
        auc_score = final_results_df_sorted.loc[model_name, 'ROC AUC'] # Get AUC from DataFrame

        if pd.notna(auc_score):
            plt.plot(fpr, tpr, lw=2, label=f'{model_name} (AUC = {auc_score:.4f})') # Thicker lines
        else:
             print(f"Skipping ROC plot for {model_name} due to invalid AUC score.")

    elif model_name not in model_probabilities:
         print(f"Skipping ROC plot for {model_name} - probabilities not found.")
    else: # Probabilities were None (e.g., predict_proba failed)
         print(f"Skipping ROC plot for {model_name} - probabilities unavailable.")


# Plot the baseline (random guess)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Guess (AUC = 0.50)')

# Customize the plot
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05]) # Slightly larger y-limit
plt.xlabel('False Positive Rate (FPR)', fontsize=12)
plt.ylabel('True Positive Rate (TPR)', fontsize=12)
plt.title('Receiver Operating Characteristic (ROC) Curve Comparison (Test Set)', fontsize=14)
plt.legend(loc="lower right", fontsize=10)
plt.grid(alpha=0.5) # Add a subtle grid
plt.show()

print("\n--- ROC Curve Plot Displayed ---")


# --- Detailed Report for Best Tuned Model ---
if not final_results_df_sorted.empty:
    # Get the name of the best model based on the sorting metric used above
    best_model_name = final_results_df_sorted.index[0]
    print(f"\n--- Detailed Report for Best Model ({best_model_name}) based on {sort_metric} ---")

    if best_model_name == "Neural Network":
         # Use the predictions calculated during NN evaluation
         if 'y_pred_nn' in locals():
              print(f"\nConfusion Matrix (Neural Network):\n{confusion_matrix(y_test, y_pred_nn)}")
              print(f"\nClassification Report (Neural Network):\n{classification_report(y_test, y_pred_nn, digits=4)}")
         else:
              print("Neural Network predictions (y_pred_nn) not available for detailed report.")

    elif best_model_name in best_estimators:
        # Retrieve the best estimator object
```

Appendix

```
# This ensures the legend order matches performance (best first)
for model_name in final_results_df_sorted.index:
    if model_name in model_probabilities and model_probabilities[model_name] is not None:
        probabilities = model_probabilities[model_name]
        fpr, tpr, thresholds = roc_curve(y_test, probabilities)
        auc_score = final_results_df_sorted.loc[model_name, 'ROC AUC'] # Get AUC from DataFrame

        if pd.notna(auc_score):
            plt.plot(fpr, tpr, lw=2, label=f'{model_name} (AUC = {auc_score:.4f})') # Thicker lines
        else:
             print(f"Skipping ROC plot for {model_name} due to invalid AUC score.")

    elif model_name not in model_probabilities:
         print(f"Skipping ROC plot for {model_name} - probabilities not found.")
    else: # Probabilities were None (e.g., predict_proba failed)
         print(f"Skipping ROC plot for {model_name} - probabilities unavailable.")


# Plot the baseline (random guess)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Guess (AUC = 0.50)')

# Customize the plot
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05]) # Slightly larger y-limit
plt.xlabel('False Positive Rate (FPR)', fontsize=12)
plt.ylabel('True Positive Rate (TPR)', fontsize=12)
plt.title('Receiver Operating Characteristic (ROC) Curve Comparison (Test Set)', fontsize=14)
plt.legend(loc="lower right", fontsize=10)
plt.grid(alpha=0.5) # Add a subtle grid
plt.show()

print("\n--- ROC Curve Plot Displayed ---")


# --- Detailed Report for Best Tuned Model ---
if not final_results_df_sorted.empty:
    # Get the name of the best model based on the sorting metric used above
    best_model_name = final_results_df_sorted.index[0]
    print(f"\n--- Detailed Report for Best Model ({best_model_name}) based on {sort_metric} ---")

    if best_model_name == "Neural Network":
         # Use the predictions calculated during NN evaluation
         if 'y_pred_nn' in locals():
              print(f"\nConfusion Matrix (Neural Network):\n{confusion_matrix(y_test, y_pred_nn)}")
              print(f"\nClassification Report (Neural Network):\n{classification_report(y_test, y_pred_nn, digits=4)}")
         else:
              print("Neural Network predictions (y_pred_nn) not available for detailed report.")

    elif best_model_name in best_estimators:
        # Retrieve the best estimator object
```

Appendix

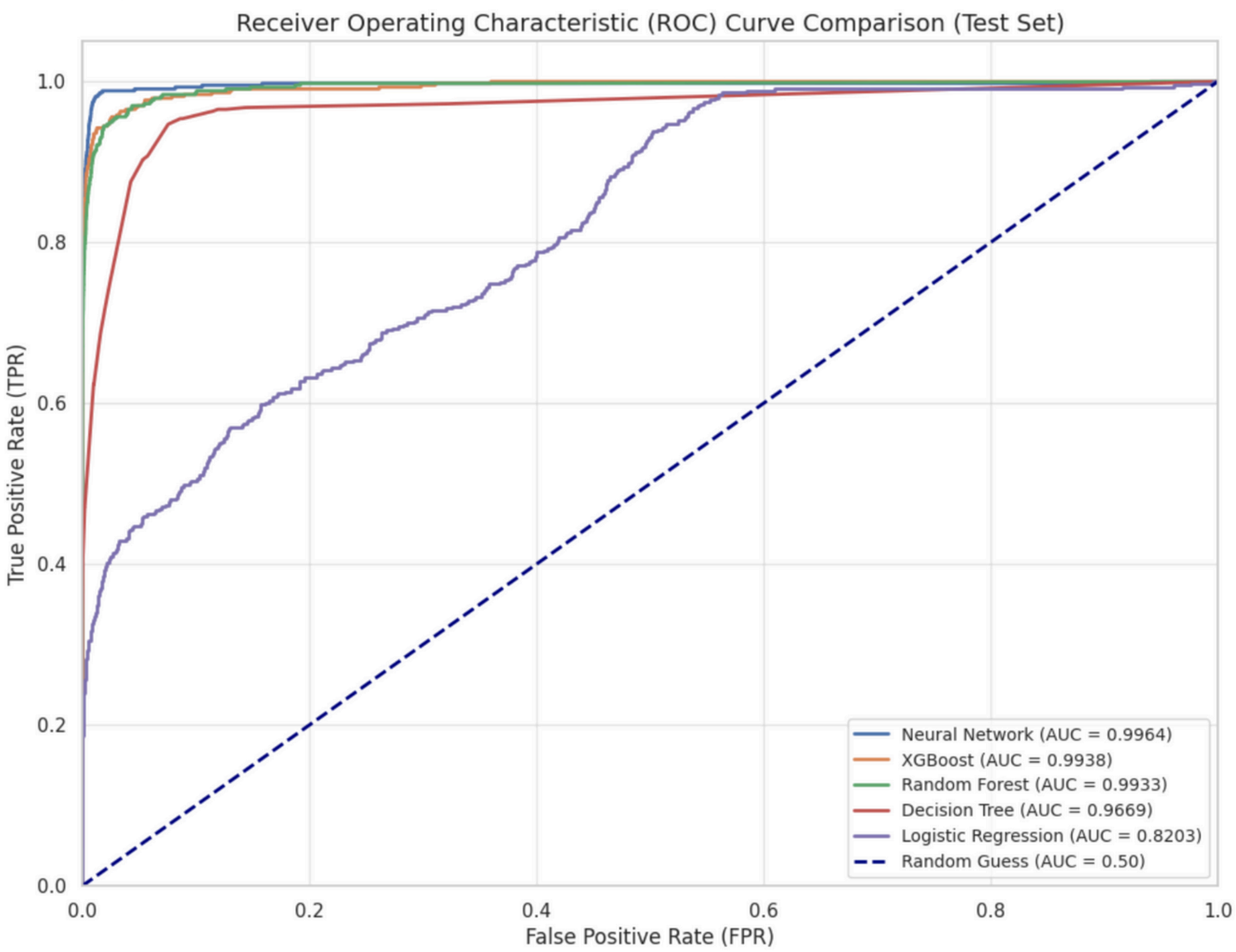
```
    elif best_model_name in best_estimators:
        # Retrieve the best estimator object
        best_model_obj = best_estimators[best_model_name]
        # Generate predictions using the best model
        y_pred_best = best_model_obj.predict(X_test)

        # Display best parameters found during tuning (if available)
        try:
            best_params_str = tuning_summary_df.loc[best_model_name].get('Best Params', 'N/A')
            print(f"Best Parameters Found: {best_params_str}")
        except KeyError:
            print("Best parameters not found in tuning summary.")

        # Display Confusion Matrix and Classification Report
        print(f"\nConfusion Matrix ({best_model_name}):\n{confusion_matrix(y_test, y_pred_best)}")
        print(f"\nClassification Report ({best_model_name}):\n{classification_report(y_test, y_pred_best, digits=4)}") # Use 4 digits for precision

    else:
         print(f"Could not retrieve best model object or predictions for {best_model_name}.")

else:
    print("\nCannot generate detailed report as no final results are available.")

print("\n--- End of Evaluation and Comparison ---")
```

Appendix

```
--- Evaluating Tuned Models and Trained NN on the Hold-Out Test Set ---

Evaluating Decision Tree...
Evaluation completed for Decision Tree in 0.04 seconds.

Evaluating Logistic Regression...
Evaluation completed for Logistic Regression in 0.01 seconds.

Evaluating Random Forest...
Evaluation completed for Random Forest in 0.87 seconds.

Evaluating XGBoost...
Evaluation completed for XGBoost in 0.13 seconds.

Evaluating Neural Network...
16032/16032 ━━━━━━━━━━━━━━━━━━━━ 7s 451us/step
Evaluation completed for Neural Network in 24.64 seconds.

--- Comparison of Final Model Results (After Tuning) ---
Results sorted by ROC AUC (descending):
                    ROC AUC  F1 Score  Recall  Precision  Accuracy  Prediction Time (s)  Training/Tuning Time (s)
Model
Neural Network       0.9964    0.1407  0.9746     0.0758    0.9900              24.6414                   97.5048
XGBoost              0.9938    0.2171  0.8961     0.1235    0.9945               0.1349                  145.2833
Random Forest        0.9933    0.2152  0.8591     0.1230    0.9947               0.8696                  746.9445
Decision Tree        0.9669    0.0182  0.9538     0.0092    0.9132               0.0420                   93.0352
Logistic Regression  0.8203    0.0040  0.7021     0.0020    0.7058               0.0139                   17.3479

--- Generating ROC Curve Comparison Plot ---
```

Appendix

```
--- Generating ROC Curve Comparison Plot ---
```



Appendix

```
--- ROC Curve Plot Displayed ---

--- Detailed Report for Best Model (Neural Network) based on ROC AUC ---

Confusion Matrix (Neural Network):
[[507448   5143]
 [    11    422]]

Classification Report (Neural Network):
              precision    recall  f1-score   support

         0.0     1.0000    0.9900    0.9949    512591
         1.0     0.0758    0.9746    0.1407       433

    accuracy                         0.9900    513024
   macro avg     0.5379    0.9823    0.5678    513024
weighted avg     0.9992    0.9900    0.9942    513024


--- End of Evaluation and Comparison ---
```

References

[1] Cybersource, "Global Fraud and Payments Report 2023," [Online]. Available: https://www.cybersource.com/en/solutions/fraud-and-risk-management/fraud-report.html. [Accessed: 5-May-2025].

[2] J. Shah, “Online Payment Fraud Detection,” Kaggle, Oct. 26, 2022. [Online]. Available: https://www.kaggle.com/datasets/jainilcoder/online-payment-fraud-detection. [Accessed: Apr 20, 2025].

[3] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: synthetic minority over-sampling technique,” Journal of Artificial Intelligence Research, vol.16, pp. 321–357, 2002.

[4] Statista, "Value of e-commerce losses to online payment fraud worldwide from 2020 to 2023 (in billion U.S. dollars)," [Online]. Available: https://www.statista.com/statistics/1273177/ecommerce-payment-fraud-losses-globally/. [Accessed: 5-May-2025].