AI Driven Fraud Detection System for Financial Transactions

This project focuses on building an AI-driven fraud detection system for financial transactions using machine learning. The main goal is to accurately identify fraudulent transactions within a highly imbalanced, anonymised dataset.

The key steps followed in this notebook are:

- Exploratory Data Analysis (EDA) to understand class imbalance
- Preprocessing of important features ('Amount' and 'Time') to improve model learning
- Handling severe class imbalance using oversampling techniques (SMOTE and ADASYN)
- Training and evaluating multiple machine learning models (Random Forest, Logistic Regression, SVM, KNN)
- Comparing models based on Recall, F1 Score, and Precision-Recall AUC to select the best-performing pipeline

Throughout the notebook, a strong focus is placed on:

- Handling imbalanced data effectively
- Maintaining reproducibility and fairness
- Using clear metrics that reflect real-world fraud detection needs

The results from this project provide insights into how different models and resampling techniques affect fraud detection performance in a practical setting and which is best.

```python
# Import necessary libraries for data analysis and visualization
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Machine Learning libraries
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score, confusion_matrix, roc_auc_score, roc_curve
from sklearn.metrics import classification_report
from sklearn.metrics import roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import auc

# Resampling techniques for imbalanced dataset
from imblearn.over_sampling import SMOTE
from imblearn.over_sampling import ADASYN
from collections import Counter

# For visualizations
sns.set(style="whitegrid")

import json # For Saving and loading metrics
import joblib  # For saving and loading models
import pickle  # For saving and loading metrics
import os  # For checking file existences
from scipy.stats import boxcox # For box cox transformations

# Mount Google Drive to access the file
from google.colab import drive
drive.mount('/content/drive')

# Define the file path to your dataset in Google Drive
file_path = '/content/drive/My Drive/Colab Notebooks/creditcard.csv'

# Load the dataset
df = pd.read_csv(file_path)

# Display the first few rows of the dataset
df.head()
```

```
Mounted at /content/drive
```

{"type":"dataframe","variable_name":"df"}

```
df.describe()
```

{"type":"dataframe"}

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
```

```
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

```python
# Check for null values
print("Missing values per column:\n", df.isnull().sum())
```

```
Missing values per column:
 Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
```

```
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

```
#Finding all rows where fraud (Class = 1)
frauds = df.loc[df['Class'] == 1]
#Finding all rows that are not fraud (Class = 0)
non_frauds = df.loc[df['Class'] == 0]

#Calculating the counts and percentages
fraud_count = len(frauds)
non_fraud_count = len(non_frauds)

#Calculating percentages
fraud_percentage = round(fraud_count / len(df) * 100, 2)
non_fraud_percentage = round(non_fraud_count / len(df) * 100, 2)

#Printing the results
print('Fraud - ', fraud_count, 'transactions or ', fraud_percentage,
'% of the dataset')
print('Legitimate -', non_fraud_count, 'transactions or ',
non_fraud_percentage, '% of the dataset')
```

```
Fraud -  492 transactions or  0.17 % of the dataset
Legitimate - 284315 transactions or  99.83 % of the dataset
```

#**Train/Test Split and Baseline**

```
from sklearn.model_selection import train_test_split

# Define features (X) and target (y)
X = df.drop(columns=['Class'])  # drop target column
y = df['Class']                 # define target
```

```
# Train-test split with stratification for class balance
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# Display split details
print(f"Training set shape: {X_train.shape}, Target: {y_train.shape}")
print(f"Testing set shape: {X_test.shape}, Target: {y_test.shape}")
print(f"\nFraud cases in training set: {y_train.sum()}")
print(f"Fraud cases in testing set: {y_test.sum()}")
```

```
Training set shape: (227845, 30), Target: (227845,)
Testing set shape: (56962, 30), Target: (56962,)

Fraud cases in training set: 394
Fraud cases in testing set: 98
```

We split the original dataset into a training set (80%) and a testing set (20%) using stratified sampling to maintain the class distribution. This ensures that the minority fraud cases are proportionally represented in both sets, avoiding skewed evaluation results later.

Random Forest

```
# File path for Random Forest metrics
rf_path = "/content/drive/My Drive/Colab
Notebooks/rf_metrics_RAW.json"

# Load or train Random Forest
if os.path.exists(rf_path):
    print("Loaded saved Random Forest classification report:")
    with open(rf_path, "r") as f:
        saved_metrics = json.load(f)
    print(pd.DataFrame(saved_metrics).transpose())
else:
    print("No saved classification report found. Training Random
Forest model on RAW data...")
    rf_model = RandomForestClassifier(random_state=42)
    rf_model.fit(X_train, y_train)
    y_pred_rf = rf_model.predict(X_test)
    report = classification_report(y_test, y_pred_rf,
output_dict=True)
    with open(rf_path, "w") as f:
        json.dump(report, f, indent=4)
    print(pd.DataFrame(report).transpose())
```

```
Loaded saved Random Forest classification report:
              precision    recall  f1-score       support
0              0.999684  0.999912  0.999798  56864.000000
1              0.941176  0.816327  0.874317     98.000000
```

```
accuracy       0.999596  0.999596  0.999596      0.999596
macro avg      0.970430  0.908119  0.937057  56962.000000
weighted avg   0.999583  0.999596  0.999582  56962.000000
```

Logistic Regression

```
# File path for Logistic Regression metrics
lr_path = "/content/drive/My Drive/Colab
Notebooks/lr_metrics_RAW.json"

# Load or train Logistic Regression
if os.path.exists(lr_path):
    print("Loaded saved Logistic Regression classification report:")
    with open(lr_path, "r") as f:
        saved_metrics = json.load(f)
    print(pd.DataFrame(saved_metrics).transpose())
else:
    print("No saved classification report found. Training Logistic
Regression model on RAW data...")
    lr_model = LogisticRegression(max_iter=1000, random_state=42)
    lr_model.fit(X_train, y_train)
    y_pred_lr = lr_model.predict(X_test)
    report = classification_report(y_test, y_pred_lr,
output_dict=True)
    with open(lr_path, "w") as f:
        json.dump(report, f, indent=4)
    print(pd.DataFrame(report).transpose())
```

```
Loaded saved Logistic Regression classification report:
              precision    recall  f1-score      support
0              0.999437  0.999771  0.999604  56864.00000
1              0.835443  0.673469  0.745763     98.00000
accuracy       0.999210  0.999210  0.999210      0.99921
macro avg      0.917440  0.836620  0.872684  56962.00000
weighted avg   0.999155  0.999210  0.999168  56962.00000
```

KNN

```
# File path for KNN metrics
knn_path = "/content/drive/My Drive/Colab
Notebooks/knn_metrics_RAW.json"

# Load or train KNN
if os.path.exists(knn_path):
    print("Loaded saved KNN classification report:")
    with open(knn_path, "r") as f:
        saved_metrics = json.load(f)
    print(pd.DataFrame(saved_metrics).transpose())
else:
```

```
    print("No saved classification report found. Training KNN model on
RAW data...")
    knn_model = KNeighborsClassifier()
    knn_model.fit(X_train, y_train)
    y_pred_knn = knn_model.predict(X_test)
    report = classification_report(y_test, y_pred_knn,
output_dict=True)
    with open(knn_path, "w") as f:
        json.dump(report, f, indent=4)
    print(pd.DataFrame(report).transpose())
```

```
Loaded saved KNN classification report:
              precision    recall  f1-score       support
0              0.998332  1.000000  0.999165  56864.000000
1              1.000000  0.030612  0.059406     98.000000
accuracy       0.998332  0.998332  0.998332      0.998332
macro avg      0.999166  0.515306  0.529286  56962.000000
weighted avg   0.998335  0.998332  0.997549  56962.000000
```

SVM

```
# File path for SVM metrics
svm_path = "/content/drive/My Drive/Colab
Notebooks/svm_metrics_RAW.json"

# Load or train SVM
if os.path.exists(svm_path):
    print("Loaded saved SVM classification report:")
    with open(svm_path, "r") as f:
        saved_metrics = json.load(f)
    print(pd.DataFrame(saved_metrics).transpose())
else:
    print("No saved classification report found. Training SVM model on
RAW data...")
    svm_model = SVC(kernel='rbf', random_state=42)
    svm_model.fit(X_train, y_train)
    y_pred_svm = svm_model.predict(X_test)
    report = classification_report(y_test, y_pred_svm,
output_dict=True)
    with open(svm_path, "w") as f:
        json.dump(report, f, indent=4)
    print(pd.DataFrame(report).transpose())
```

```
Loaded saved SVM classification report:
              precision   recall  f1-score      support
0              0.998280  1.00000  0.999139  56864.00000
1              0.000000  0.00000  0.000000     98.00000
accuracy       0.998280  0.99828  0.998280      0.99828
```

```
macro avg      0.499140  0.50000  0.499570  56962.00000
weighted avg   0.996562  0.99828  0.997420  56962.00000
```

#**Exploratory Data Analysis**

Lets have a look at how many of the transactions are fraudlent and how many are legitimate.

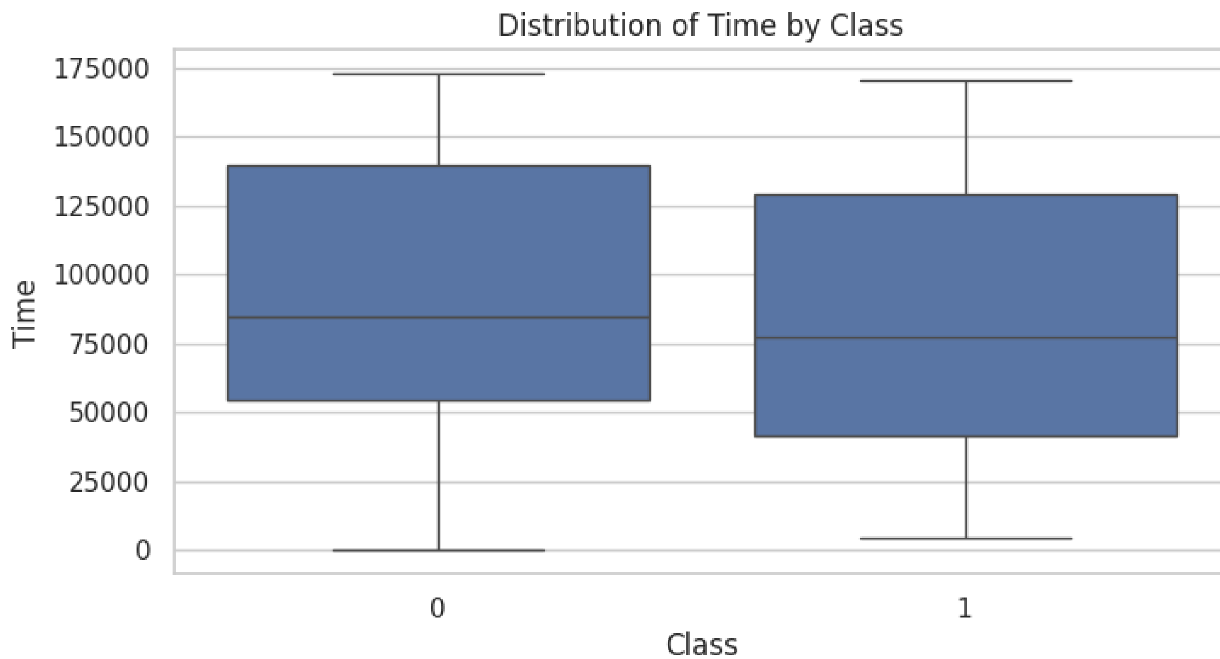
```
# 1. Check class distribution in the training set (clear format)
print("Class Distribution in Training Set:\n")
class_counts = y_train.value_counts()
class_percentages = y_train.value_counts(normalize=True) * 100
distribution_df = pd.DataFrame({
    "Count": class_counts,
    "Percentage": class_percentages.round(4)
})
distribution_df.index = distribution_df.index.map({0: 'Legitimate
(0)', 1: 'Fraudulent (1)'})
display(distribution_df)


# Bar plot of class distribution
plt.figure(figsize=(6,4))
sns.countplot(x=y_train)
plt.title('Class Distribution (Training Set)')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```

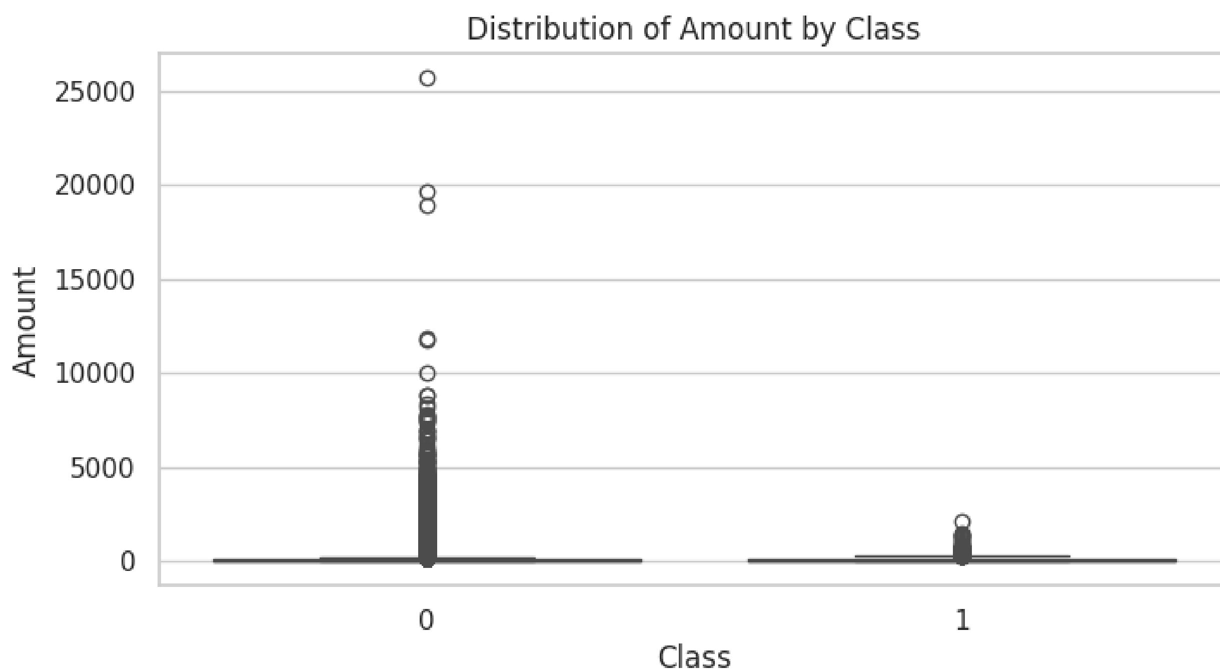
```
Class Distribution in Training Set:
```


{"summary":"{\n \"name\": \"distribution_df\",\n \"rows\": 2,\n \"fields\": [\n   {\n   \"column\": \"Class\",\n   \"properties\": {\n   \"dtype\": \"string\",\n   \"num_unique_values\": 2,\n   \"samples\": [\n   \"Fraudulent (1)\",\n   \"Legitimate (0)\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Count\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 160553,\n   \"min\": 394,\n   \"max\": 227451,\n   \"num_unique_values\": 2,\n   \"samples\": [\n   394,\n   227451\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Percentage\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 70.46616059372045,\n   \"min\": 0.1729,\n   \"max\": 99.8271,\n   \"num_unique_values\": 2,\n   \"samples\": [\n   0.1729,\n   99.8271\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe","variable_name":"distribution_df"}

We checked the class distribution in the training set. As expected, fraud cases are extremely rare compared to non-fraud cases, highlighting the severe imbalance problem that needs to be addressed before model training.

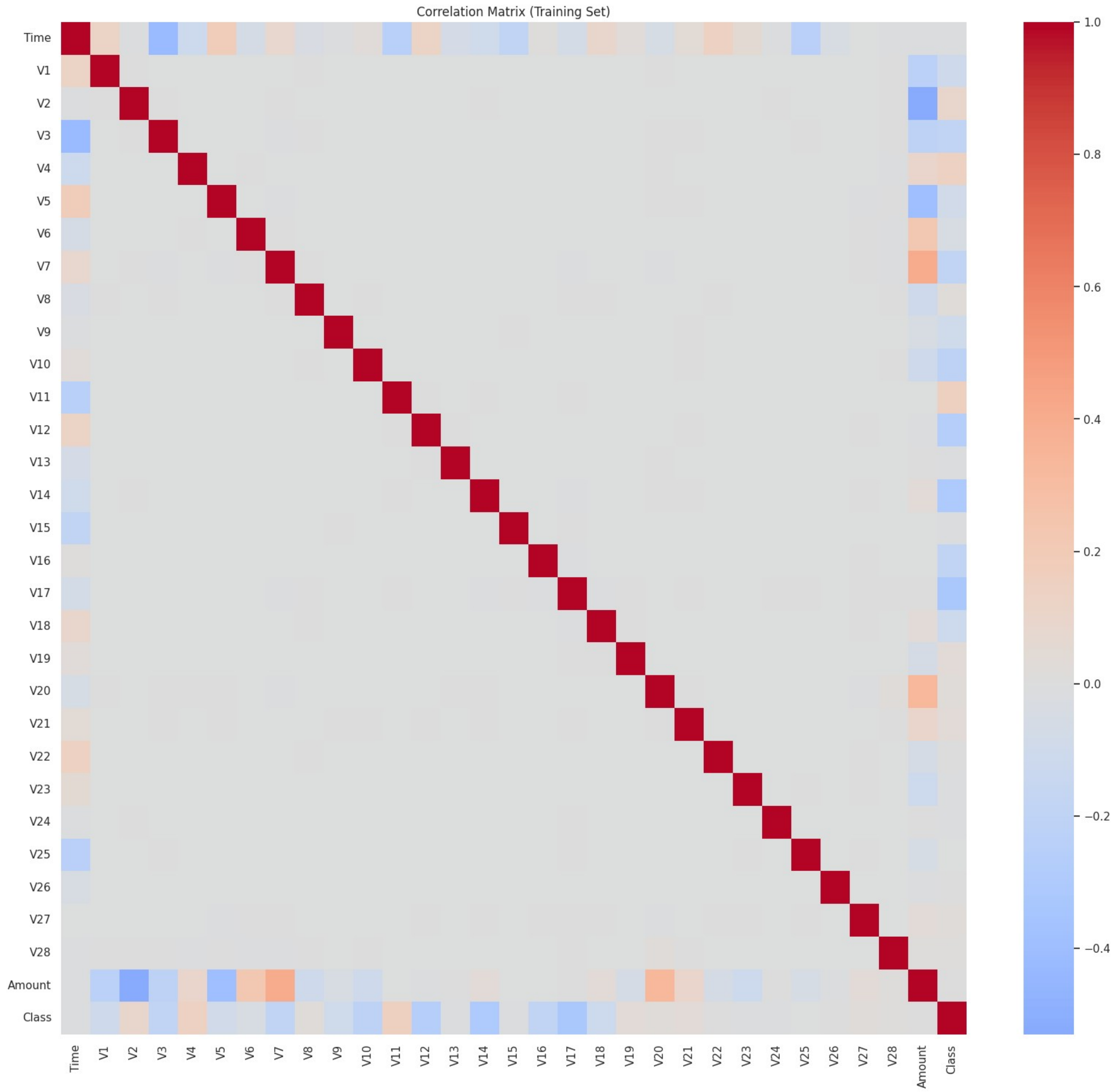
```
for col in ['Time', 'Amount']:
    plt.figure(figsize=(8, 4))
    sns.boxplot(x=y_train, y=X_train[col])
    plt.title(f'Distribution of {col} by Class')
    plt.show()
```






```
# Correlation matrix on training set (for all features + class)
df_train_full = X_train.copy()
df_train_full['Class'] = y_train
corr = df_train_full.corr()

# Plot correlation heatmap
plt.figure(figsize=(20, 18))
sns.heatmap(corr, annot=False, cmap='coolwarm', center=0)
```

```
plt.title('Correlation Matrix (Training Set)')
plt.show()
```



We plotted the correlation matrix to explore relationships between the available features. Although the dataset is PCA-transformed (making feature meanings unknown), the correlation matrix still helps identify any slight patterns or dependencies. Understanding feature correlations can guide model expectations — for example, highly correlated features could lead to redundant information, while uncorrelated features suggest independent contributions.

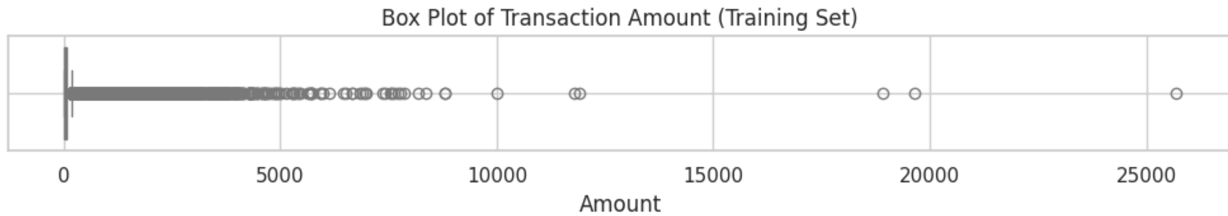
```
#Top 10 positively/negatively correlated features with Class
class_corr = corr['Class'].drop('Class')
top_positive = class_corr.sort_values(ascending=False).head(10)
```

```
top_negative = class_corr.sort_values().head(10)

print("\nTop 10 Positively Correlated Features with 'Class':")
print(top_positive)
print("\nTop 10 Negatively Correlated Features with 'Class':")
print(top_negative)
```

```
Top 10 Positively Correlated Features with 'Class':
V11       0.153709
V4        0.135014
V2        0.090586
V21       0.035588
V19       0.032380
V8        0.020552
V20       0.019385
V27       0.016034
V28       0.009810
Amount    0.006211
Name: Class, dtype: float64

Top 10 Negatively Correlated Features with 'Class':
V17   -0.321937
V14   -0.301054
V12   -0.259989
V10   -0.217894
V3    -0.194135
V16   -0.193826
V7    -0.186184
V18   -0.108732
V1    -0.100041
V9    -0.098247
Name: Class, dtype: float64
```

Data Preprocessing

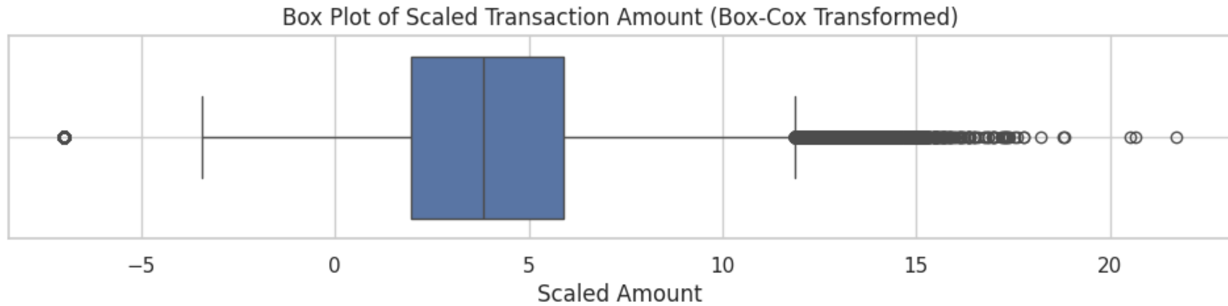
```
plt.figure(figsize=(10, 2))
sns.boxplot(x=X_train['Amount'], orient='h', color='lightblue')
plt.title("Box Plot of Transaction Amount (Training Set)")
plt.xlabel("Amount")
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
#Shift amount to avoid zero (Box-Cox needs strictly positive values)
X_train['Amount_shifted'] = X_train['Amount'] + 1e-9
X_test['Amount_shifted'] = X_test['Amount'] + 1e-9

#Fit Box-Cox on training set only
X_train['scaled_amount'], fitted_lambda =
boxcox(X_train['Amount_shifted'])

# Save the fitted lambda for Box-Cox transformation
joblib.dump(fitted_lambda, 'boxcox_lambda_amount.pkl')

# Load existing saved Box-Cox lambda (.pkl file)
boxcox_lambda = joblib.load('boxcox_lambda_amount.pkl')

# Save the lambda into a JSON file
with open('boxcox_lambda_amount.json', 'w') as f:
    json.dump({'lambda': float(boxcox_lambda)}, f)

print("Saved Box-Cox lambda as boxcox_lambda_amount.json")


# Apply the same transformation using the learned lambda to the test
set
X_test['scaled_amount'] = boxcox(X_test['Amount_shifted'],
lmbda=fitted_lambda)

# Check skewness before transformation
original_skew = X_train['Amount'].skew()
print(f"Skewness of original 'Amount': {original_skew:.4f}")

# Check skewness after Box-Cox transformation
transformed_skew = pd.Series(X_train['scaled_amount']).skew()
print(f"Skewness of Box-Cox transformed 'Amount':
{transformed_skew:.4f}")

# Drop the original and intermediate columns
X_train.drop(['Amount', 'Amount_shifted'], axis=1, inplace=True)
X_test.drop(['Amount', 'Amount_shifted'], axis=1, inplace=True)

plt.figure(figsize=(12, 2))
sns.boxplot(x=X_train['scaled_amount'], orient='h')
plt.title("Box Plot of Scaled Transaction Amount (Box-Cox
Transformed)")
```

```
plt.xlabel("Scaled Amount")
plt.grid(True)
plt.show()
```

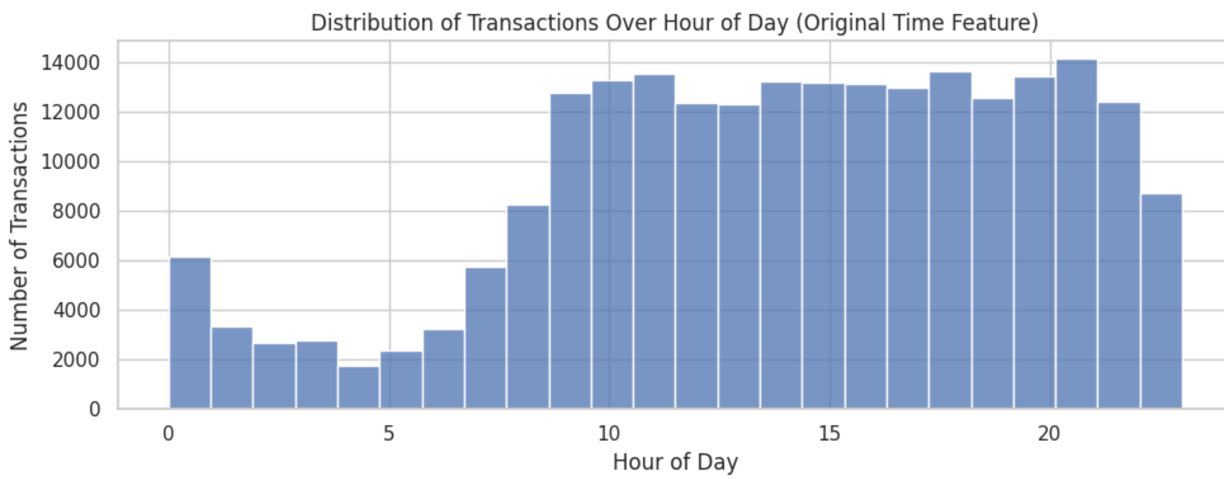
```
Saved Box-Cox lambda as boxcox_lambda_amount.json
Skewness of original 'Amount': 18.1939
Skewness of Box-Cox transformed 'Amount': 0.1146
```



We applied Box-Cox transformation to the 'Amount' feature to reduce its skewness. Normalizing the feature helps machine learning models perform better because highly skewed data can bias the model towards certain feature values.

Now lets move onto time

```
# Convert time to hours for better readability
X_train_time_dist = X_train.copy()
X_train_time_dist['Hour'] = (X_train_time_dist['Time'] // 3600) % 24

plt.figure(figsize=(10, 4))
sns.histplot(X_train_time_dist['Hour'], bins=24, kde=False)
plt.title("Distribution of Transactions Over Hour of Day (Original
Time Feature)")
plt.xlabel("Hour of Day")
plt.ylabel("Number of Transactions")
plt.grid(True)
plt.tight_layout()
plt.show()
```

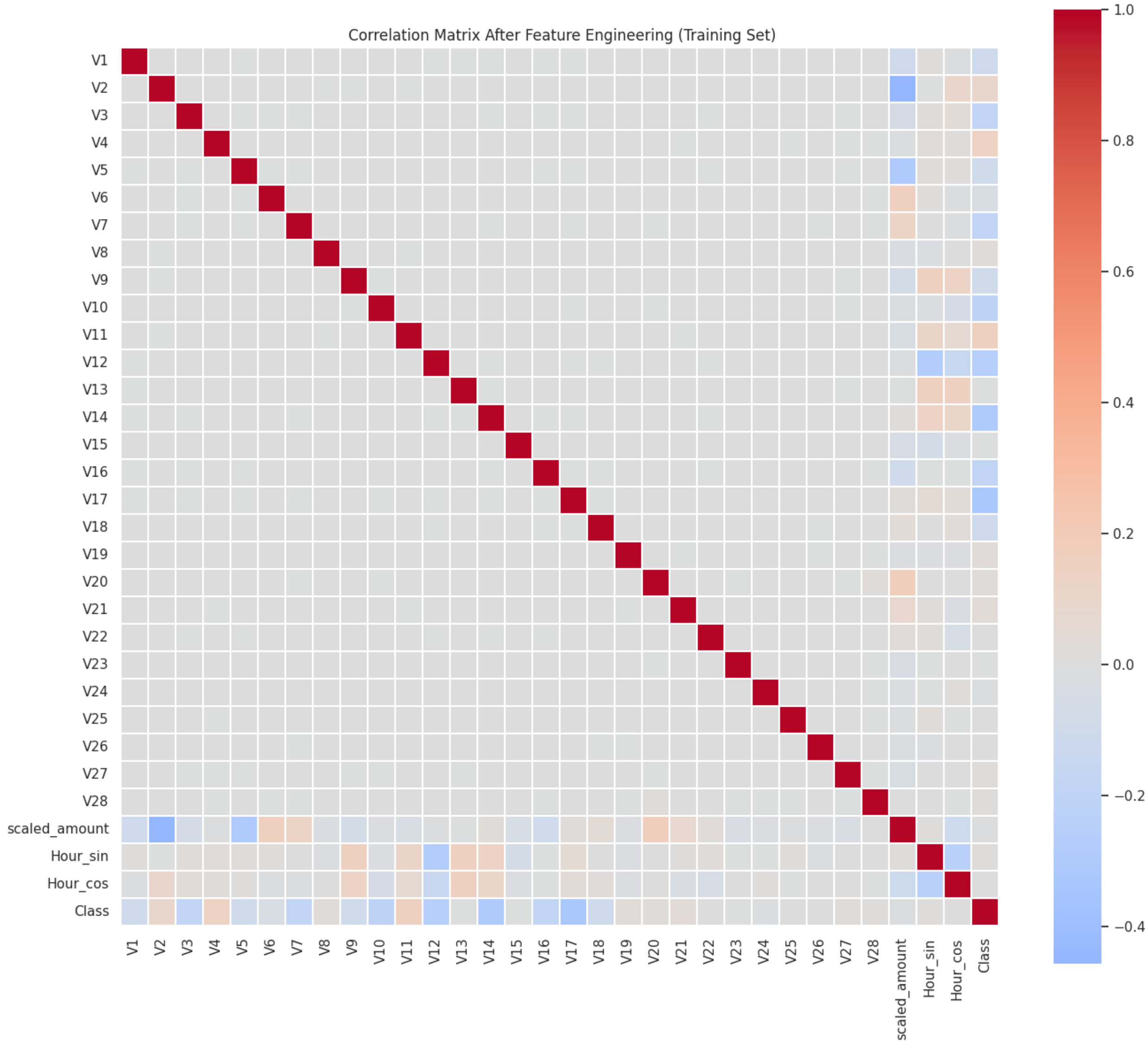



```
# Copy original datasets
X_train_time = X_train.copy()
X_test_time = X_test.copy()

# Extract Hour from 'Time' column (seconds since first transaction)
X_train_time['Hour'] = (X_train_time['Time'] // 3600) % 24
X_test_time['Hour'] = (X_test_time['Time'] // 3600) % 24

# Apply sin/cos transformation to encode cyclical nature
X_train_time['Hour_sin'] = np.sin(2 * np.pi * X_train_time['Hour'] /
24)
X_train_time['Hour_cos'] = np.cos(2 * np.pi * X_train_time['Hour'] /
24)

X_test_time['Hour_sin'] = np.sin(2 * np.pi * X_test_time['Hour'] / 24)
X_test_time['Hour_cos'] = np.cos(2 * np.pi * X_test_time['Hour'] / 24)

# Drop raw 'Time' column and 'Hour'
X_train_time.drop(columns=['Time', 'Hour'], inplace=True)
X_test_time.drop(columns=['Time', 'Hour'], inplace=True)
```

We engineered new 'Hour_sin' and 'Hour_cos' features from the 'Time' column using sine and cosine transformations. This approach captures the cyclic nature of time (e.g., transactions occurring at similar times of day) which could be useful for detecting fraud patterns.

```
# Merge the updated X_train_time with y_train for correlation analysis
train_df = X_train_time.copy()
train_df['Class'] = y_train.values

# Compute correlation matrix
corr_matrix = train_df.corr()

# Plot correlation heatmap
plt.figure(figsize=(16, 14))
```

```
sns.heatmap(corr_matrix, cmap='coolwarm', annot=False, center=0,
square=True, linewidths=0.3)
plt.title(" Correlation Matrix After Feature Engineering (Training
Set)")
plt.show()

# Show top correlated features with 'Class'
corr_with_target = corr_matrix['Class'].drop('Class')

top_positive_corr =
corr_with_target.sort_values(ascending=False).head(10)
top_negative_corr = corr_with_target.sort_values().head(10)

print("Top 10 Positively Correlated Features with 'Class':")
print(top_positive_corr)
print("\nTop 10 Negatively Correlated Features with 'Class':")
print(top_negative_corr)
```




```
Top 10 Positively Correlated Features with 'Class':
V11         0.153709
V4          0.135014
V2          0.090586
V21         0.035588
V19         0.032380
V8          0.020552
V20         0.019385
V27         0.016034
Hour_sin    0.011816
V28         0.009810
Name: Class, dtype: float64

Top 10 Negatively Correlated Features with 'Class':
V17   -0.321937
```

```
V14   -0.301054
V12   -0.259989
V10   -0.217894
V3    -0.194135
V16   -0.193826
V7    -0.186184
V18   -0.108732
V1    -0.100041
V9    -0.098247
Name: Class, dtype: float64
```

Pre Processed Baseline

```python
# Check structure and basic stats of preprocessed training set
print("\n Structure of Preprocessed Training Set:")
X_train_time.info()

print("\n Descriptive Statistics of Preprocessed Features:")
display(X_train_time.describe())

print("\n Sample of Preprocessed Training Data:")
display(X_train_time.head())
```

```
 Structure of Preprocessed Training Set:
<class 'pandas.core.frame.DataFrame'>
Index: 227845 entries, 265518 to 17677
Data columns (total 31 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   V1              227845 non-null  float64
 1   V2              227845 non-null  float64
 2   V3              227845 non-null  float64
 3   V4              227845 non-null  float64
 4   V5              227845 non-null  float64
 5   V6              227845 non-null  float64
 6   V7              227845 non-null  float64
 7   V8              227845 non-null  float64
 8   V9              227845 non-null  float64
 9   V10             227845 non-null  float64
 10  V11             227845 non-null  float64
 11  V12             227845 non-null  float64
 12  V13             227845 non-null  float64
 13  V14             227845 non-null  float64
 14  V15             227845 non-null  float64
 15  V16             227845 non-null  float64
 16  V17             227845 non-null  float64
 17  V18             227845 non-null  float64
```

```
 18  V19             227845 non-null  float64
 19  V20             227845 non-null  float64
 20  V21             227845 non-null  float64
 21  V22             227845 non-null  float64
 22  V23             227845 non-null  float64
 23  V24             227845 non-null  float64
 24  V25             227845 non-null  float64
 25  V26             227845 non-null  float64
 26  V27             227845 non-null  float64
 27  V28             227845 non-null  float64
 28  scaled_amount   227845 non-null  float64
 29  Hour_sin        227845 non-null  float64
 30  Hour_cos        227845 non-null  float64
dtypes: float64(31)
memory usage: 55.6 MB

 Descriptive Statistics of Preprocessed Features:

{"type":"dataframe"}


 Sample of Preprocessed Training Data:

{"type":"dataframe"}
```

Before we Preprocess the data and create the splits, we will run it on the raw dataset to set the baseline

Random Forest Model

```
import os, json
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# File path for Random Forest metrics
rf_path = "/content/drive/My Drive/Colab Notebooks/rf_metrics.json"

# Function to save classification report
def save_classification_report(model_name, y_true, y_pred, path):
    report = classification_report(y_true, y_pred, output_dict=True)
    with open(path, "w") as f:
        json.dump(report, f, indent=4)
    print(f"{model_name} classification report saved to {path}")

# Load or train Random Forest
if os.path.exists(rf_path):
    print("Loaded saved Random Forest classification report:")
    with open(rf_path, "r") as f:
        saved_metrics = json.load(f)
    df_metrics_rf = pd.DataFrame(saved_metrics).transpose()
```

```
    print(df_metrics_rf)
else:
    print("No saved classification report found. Training Random
Forest model...")
    rf_model = RandomForestClassifier(random_state=42)
    rf_model.fit(X_train_time, y_train)
    y_pred_rf = rf_model.predict(X_test_time)
    save_classification_report("Random Forest", y_test, y_pred_rf,
rf_path)
```

```
Loaded saved Random Forest classification report:
              precision    recall  f1-score       support
0              0.999701  0.999930  0.999815  56864.000000
1              0.952941  0.826531  0.885246     98.000000
accuracy       0.999631  0.999631  0.999631      0.999631
macro avg      0.976321  0.913230  0.942531  56962.000000
weighted avg   0.999621  0.999631  0.999618  56962.000000
```

Logisitic Regression

```
from sklearn.linear_model import LogisticRegression

# File path for Logistic Regression metrics
lr_path = "/content/drive/My Drive/Colab Notebooks/lr_metrics.json"

# Load or train Logistic Regression
if os.path.exists(lr_path):
    print("Loaded saved Logistic Regression classification report:")
    with open(lr_path, "r") as f:
        saved_metrics = json.load(f)
    df_metrics_lr = pd.DataFrame(saved_metrics).transpose()
    print(df_metrics_lr)
else:
    print("No saved classification report found. Training Logistic
Regression model...")
    lr_model = LogisticRegression(max_iter=1000, random_state=42)
    lr_model.fit(X_train_time, y_train)
    y_pred_lr = lr_model.predict(X_test_time)
    save_classification_report("Logistic Regression", y_test,
y_pred_lr, lr_path)
```

```
Loaded saved Logistic Regression classification report:
              precision    recall  f1-score       support
0              0.999402  0.999754  0.999578  56864.000000
1              0.820513  0.653061  0.727273     98.000000
accuracy       0.999157  0.999157  0.999157      0.999157
macro avg      0.909958  0.826408  0.863425  56962.000000
weighted avg   0.999095  0.999157  0.999110  56962.000000
```

KNN Model

```python
from sklearn.neighbors import KNeighborsClassifier

# File path for KNN metrics
knn_path = "/content/drive/My Drive/Colab Notebooks/knn_metrics.json"

# Load or train KNN
if os.path.exists(knn_path):
    print("Loaded saved KNN classification report:")
    with open(knn_path, "r") as f:
        saved_metrics = json.load(f)
    df_metrics_knn = pd.DataFrame(saved_metrics).transpose()
    print(df_metrics_knn)
else:
    print("No saved classification report found. Training KNN
model...")
    knn_model = KNeighborsClassifier()
    knn_model.fit(X_train_time, y_train)
    y_pred_knn = knn_model.predict(X_test_time)
    save_classification_report("KNN", y_test, y_pred_knn, knn_path)
```

```
Loaded saved KNN classification report:
              precision    recall  f1-score       support
0              0.999631  0.999842  0.999736  56864.000000
1              0.895349  0.785714  0.836957     98.000000
accuracy       0.999473  0.999473  0.999473      0.999473
macro avg      0.947490  0.892778  0.918346  56962.000000
weighted avg   0.999451  0.999473  0.999456  56962.000000
```

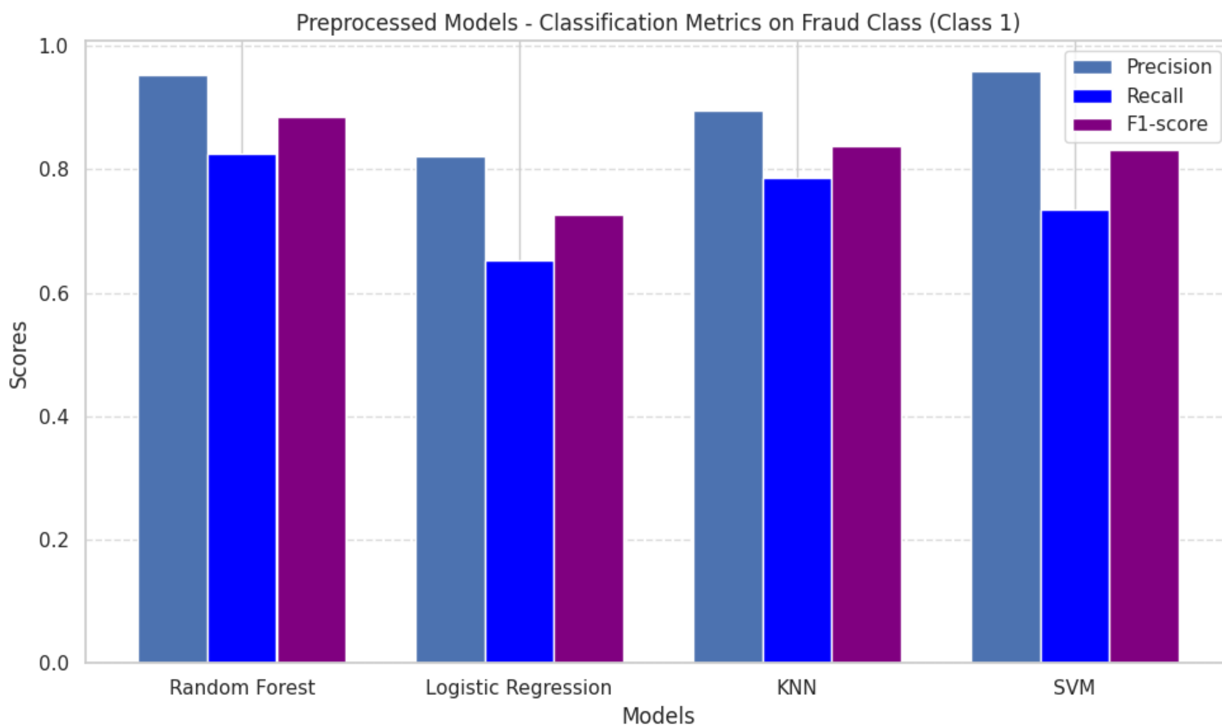
SVM

```python
from sklearn.svm import SVC

# File path for SVM metrics
svm_path = "/content/drive/My Drive/Colab Notebooks/svm_metrics.json"

# Load or train SVM
if os.path.exists(svm_path):
    print("Loaded saved SVM classification report:")
    with open(svm_path, "r") as f:
        saved_metrics = json.load(f)
    df_metrics_svm = pd.DataFrame(saved_metrics).transpose()
    print(df_metrics_svm)
else:
    print("No saved classification report found. Training SVM
model...")
    svm_model = SVC(random_state=42)
    svm_model.fit(X_train_time, y_train)
    y_pred_svm = svm_model.predict(X_test_time)
    save_classification_report("SVM", y_test, y_pred_svm, svm_path)
```

```
Loaded saved SVM classification report:
              precision    recall  f1-score       support
0              0.999543  0.999947  0.999745  56864.000000
1              0.960000  0.734694  0.832370     98.000000
accuracy       0.999491  0.999491  0.999491      0.999491
macro avg      0.979771  0.867321  0.916057  56962.000000
weighted avg   0.999475  0.999491  0.999457  56962.000000
```

Comparing the baseline results

```
import matplotlib.pyplot as plt
import numpy as np

# Preprocessed metrics for Class 1 (fraud)
metrics = {
    "Random Forest": {"precision": 0.952941, "recall": 0.826531, "f1-
score": 0.885246},
    "Logistic Regression": {"precision": 0.820513, "recall": 0.653061,
"f1-score": 0.727273},
    "KNN": {"precision": 0.895349, "recall": 0.785714, "f1-score":
0.836957},
    "SVM": {"precision": 0.960000, "recall": 0.734694, "f1-score":
0.832370},
}

models = list(metrics.keys())
precision = [metrics[m]["precision"] for m in models]
recall = [metrics[m]["recall"] for m in models]
f1_score = [metrics[m]["f1-score"] for m in models]

x = np.arange(len(models))
width = 0.25

fig, ax = plt.subplots(figsize=(10, 6))
bars1 = ax.bar(x - width, precision, width, label="Precision")
bars2 = ax.bar(x, recall, width, label="Recall", color="blue")
bars3 = ax.bar(x + width, f1_score, width, label="F1-score",
color="purple")

ax.set_xlabel("Models")
ax.set_ylabel("Scores")
ax.set_title("Preprocessed Models - Classification Metrics on Fraud
Class (Class 1)")
ax.set_xticks(x)
ax.set_xticklabels(models)
ax.legend()
ax.grid(axis="y", linestyle="--", alpha=0.7)

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



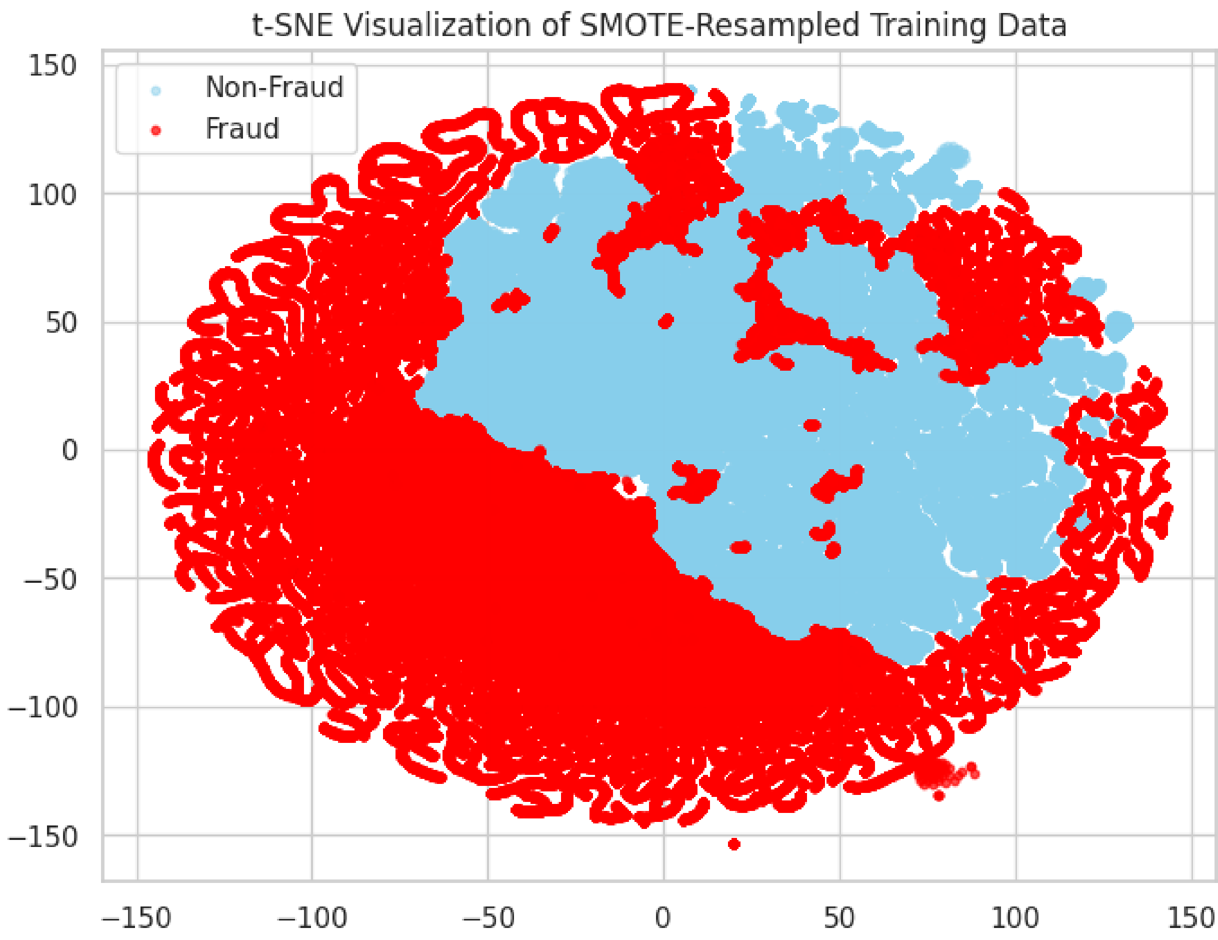
Data Preprocessing (SMOTE/ADASYN)

```
# Before SMOTE
print("Before SMOTE:", Counter(y_train))

# Initialize SMOTE
smote = SMOTE(random_state=42)

# Apply SMOTE to preprocessed training data
X_train_smote, y_train_smote = smote.fit_resample(X_train_time,
y_train)

# After SMOTE
print("After SMOTE:", Counter(y_train_smote))

Before SMOTE: Counter({0: 227451, 1: 394})
After SMOTE: Counter({0: 227451, 1: 227451})
```

We used SMOTE (Synthetic Minority Over-sampling Technique) to balance the training data by generating synthetic fraud examples. This reduces the bias toward the majority class and improves the model's ability to correctly identify fraudulent transactions.

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# Run t-SNE with fixed random state
```

```
tsne_smote = TSNE(n_components=2, random_state=42, perplexity=30)
X_smote_tsne = tsne_smote.fit_transform(X_train_smote)

# Plot
plt.figure(figsize=(8, 6))
plt.scatter(X_smote_tsne[y_train_smote == 0, 0],
X_smote_tsne[y_train_smote == 0, 1],
            label='Non-Fraud', alpha=0.5, s=10, c='skyblue')
plt.scatter(X_smote_tsne[y_train_smote == 1, 0],
X_smote_tsne[y_train_smote == 1, 1],
            label='Fraud', alpha=0.7, s=10, c='red')
plt.title("t-SNE Visualization of SMOTE-Resampled Training Data")
plt.legend()
plt.grid(True)
plt.show()
```



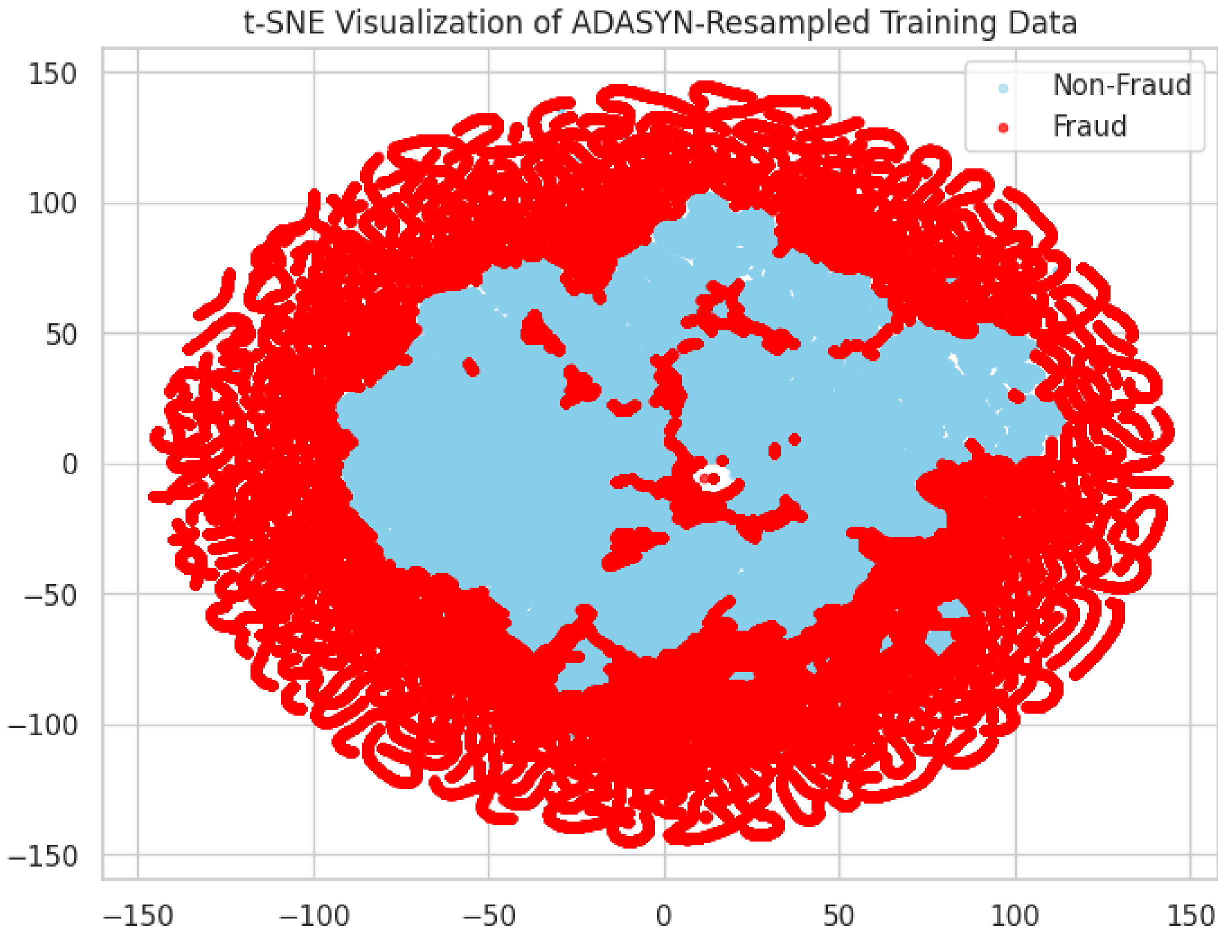

```
# Before ADASYN
print("Before ADASYN:", Counter(y_train))

# Initialize ADASYN
```

```
adasyn = ADASYN(random_state=42)

# Apply ADASYN to preprocessed training data
X_train_adasyn, y_train_adasyn = adasyn.fit_resample(X_train_time,
y_train)

# After ADASYN
print("After ADASYN:", Counter(y_train_adasyn))
```

```
Before ADASYN: Counter({0: 227451, 1: 394})
After ADASYN: Counter({1: 227460, 0: 227451})
```

We also applied ADASYN (Adaptive Synthetic Sampling) to generate more synthetic minority examples, focusing more on harder-to-classify cases. This provides another approach to balance the dataset and allows us to compare results between SMOTE and ADASYN later.

```
# Run t-SNE with fixed random state
tsne_adasyn = TSNE(n_components=2, random_state=42, perplexity=30)
X_adasyn_tsne = tsne_adasyn.fit_transform(X_train_adasyn)

# Plot
plt.figure(figsize=(8, 6))
plt.scatter(X_adasyn_tsne[y_train_adasyn == 0, 0],
X_adasyn_tsne[y_train_adasyn == 0, 1],
            label='Non-Fraud', alpha=0.5, s=10, c='skyblue')
plt.scatter(X_adasyn_tsne[y_train_adasyn == 1, 0],
X_adasyn_tsne[y_train_adasyn == 1, 1],
            label='Fraud', alpha=0.7, s=10, c='red')
plt.title("t-SNE Visualization of ADASYN-Resampled Training Data")
plt.legend()
plt.grid(True)
plt.show()
```



Model Training

##SMOTE

```
import json
import os
import pandas as pd
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    roc_auc_score,
    precision_recall_curve,
    auc
)
import matplotlib.pyplot as plt
import seaborn as sns

# Dictionary to store AUC-ROC and PR AUC scores
internal_auc_scores = {}
```

```python
# Helper function to save and display results
def save_metrics_and_outputs(model_name, y_true, y_pred, y_proba,
report_path, internal_auc_dict):
    # === Save and display classification report ===
    report = classification_report(y_true, y_pred, output_dict=True)
    with open(report_path, 'w') as f:
        json.dump(report, f, indent=4)
    print(f"\n{model_name} Classification Report:")
    display(pd.DataFrame(report).transpose())

    # === Display confusion matrix ===
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=[0,
1], yticklabels=[0, 1])
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(f"{model_name} Confusion Matrix")
    plt.show()

    # === Calculate and store AUC-ROC and PR AUC ===
    auc_roc = roc_auc_score(y_true, y_proba[:, 1])
    precision, recall, _ = precision_recall_curve(y_true, y_proba[:,
1])
    pr_auc = auc(recall, precision)
    internal_auc_dict[model_name] = {
        "AUC_ROC": auc_roc,
        "PR_AUC": pr_auc
    }
    print(f"{model_name} AUC-ROC: {auc_roc:.4f}, PR AUC: {pr_auc:.4f}
(Saved internally)")
```

Random Forest

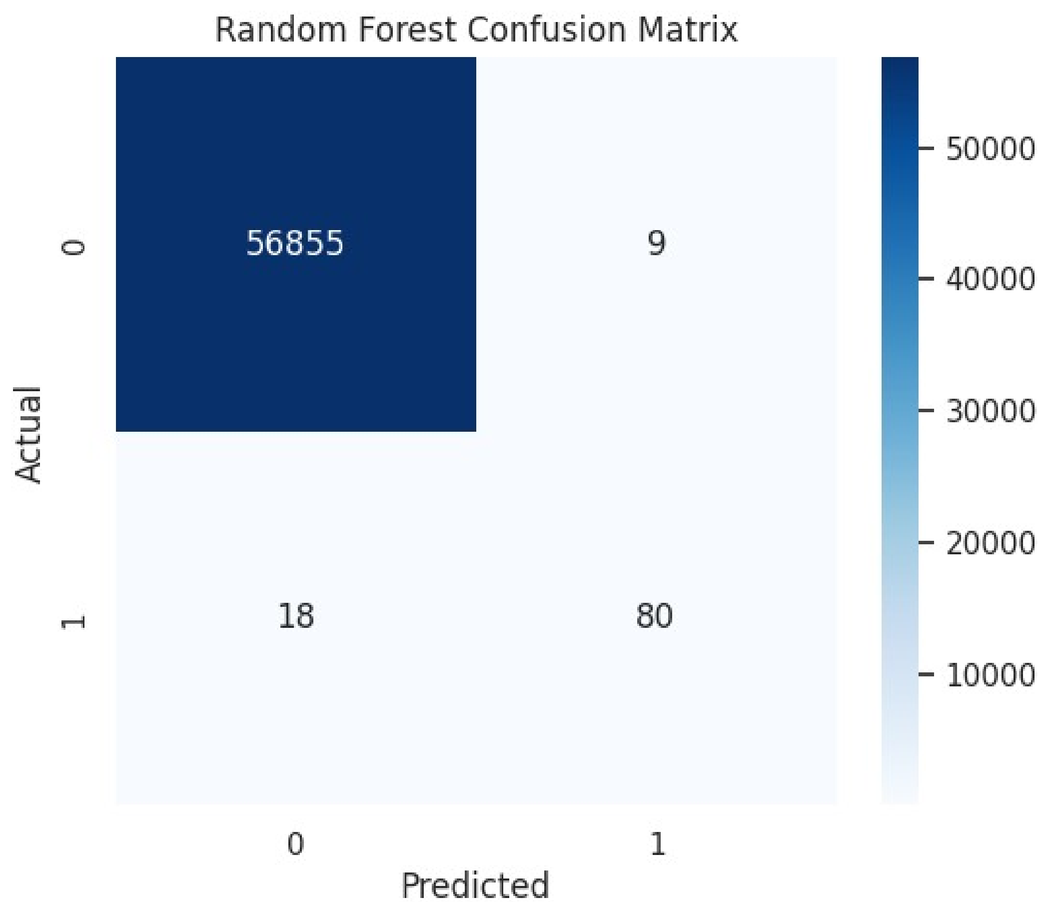
```python
rf_smote_path = "/content/drive/My Drive/Colab
Notebooks/rf_SMOTE_metrics.json"

if os.path.exists(rf_smote_path):
    with open(rf_smote_path, 'r') as f:
        saved_report = json.load(f)
    print("Random Forest Classification Report:")
    display(pd.DataFrame(saved_report).transpose())

    rf_model = RandomForestClassifier(random_state=42)
    rf_model.fit(X_train_smote, y_train_smote)
    y_pred_rf = rf_model.predict(X_test_time)
    y_proba_rf = rf_model.predict_proba(X_test_time)
    save_metrics_and_outputs("Random Forest", y_test, y_pred_rf,
```

```
y_proba_rf, rf_smote_path, internal_auc_scores)

else:
    print("No saved report found. Training Random Forest on SMOTE data...")
    rf_model = RandomForestClassifier(random_state=42)
    rf_model.fit(X_train_smote, y_train_smote)
    y_pred_rf = rf_model.predict(X_test_time)
    y_proba_rf = rf_model.predict_proba(X_test_time)
    save_metrics_and_outputs("Random Forest", y_test, y_pred_rf, y_proba_rf, rf_smote_path, internal_auc_scores)
```

```
No saved report found. Training Random Forest on SMOTE data...
```

```
Random Forest Classification Report:
```

{"summary":"{\n \"name\": \"   save_metrics_and_outputs(\\\"Random Forest\\\", y_test, y_pred_rf, y_proba_rf, rf_smote_path, internal_auc_scores)\",\n \"rows\": 5,\n \"fields\": [\n   {\n   \"column\": \"precision\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.04502686328713626,\n   \"min\": 0.898876404494382,\n   \"max\": 0.9996835053540344,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.898876404494382,\n   0.9995100722603184,\n   0.9995259997893332\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.08196470676738973,\n   \"min\": 0.8163265306122449,\n   \"max\": 0.9998417276308385,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   0.8163265306122449,\n   0.9080841291215417,\n   0.9998417276308385\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"f1-score\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.06438359730356975,\n   \"min\": 0.8556149732620321,\n   \"max\": 0.9997626102323782,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.8556149732620321,\n   0.9995146121209514,\n   0.9995259997893332\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"support\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 31154.412478809652,\n   \"min\": 0.9995259997893332,\n   \"max\": 56962.0,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   98.0,\n   56962.0,\n   56864.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}




```
Random Forest AUC-ROC: 0.9683, PR AUC: 0.8684 (Saved internally)
```

Logistic regression

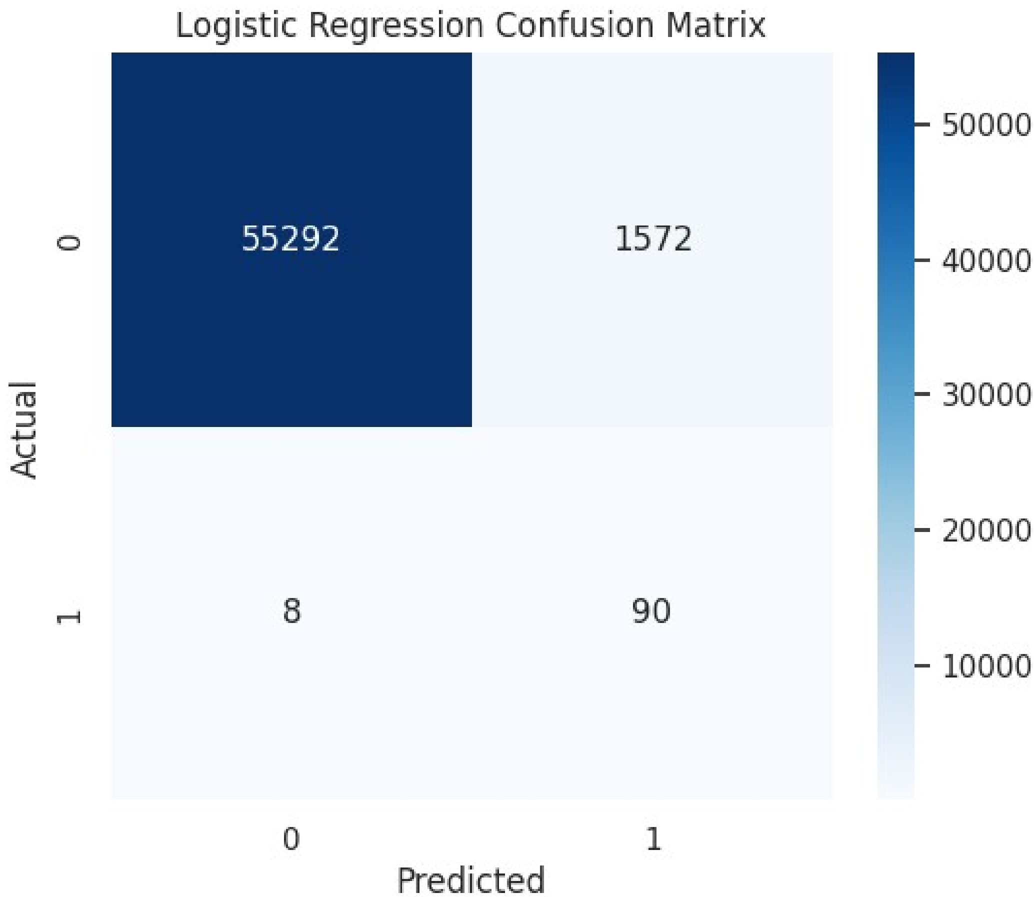
```
lr_smote_path = "/content/drive/My Drive/Colab
Notebooks/lr_SMOTE_metrics.json"

if os.path.exists(lr_smote_path):
    with open(lr_smote_path, 'r') as f:
        saved_report = json.load(f)
    print("Logistic Regression Classification Report:")
    display(pd.DataFrame(saved_report).transpose())

    lr_model = LogisticRegression(max_iter=1000, random_state=42)
    lr_model.fit(X_train_smote, y_train_smote)
    y_pred_lr = lr_model.predict(X_test_time)
    y_proba_lr = lr_model.predict_proba(X_test_time)
    save_metrics_and_outputs("Logistic Regression", y_test, y_pred_lr,
y_proba_lr, lr_smote_path, internal_auc_scores)

else:
```

```
    print("No saved report found. Training Logistic Regression on 
SMOTE data...")
    lr_model = LogisticRegression(max_iter=1000, random_state=42)
    lr_model.fit(X_train_smote, y_train_smote)
    y_pred_lr = lr_model.predict(X_test_time)
    y_proba_lr = lr_model.predict_proba(X_test_time)
    save_metrics_and_outputs("Logistic Regression", y_test, y_pred_lr, 
y_proba_lr, lr_smote_path, internal_auc_scores)
```

```
No saved report found. Training Logistic Regression on SMOTE data...
```

```
Logistic Regression Classification Report:
```

{"summary":"{\n \"name\": \"   save_metrics_and_outputs(\\\"Logistic Regression\\\", y_test, y_pred_lr, y_proba_lr, lr_smote_path, internal_auc_scores)\",\n \"rows\": 5,\n \"fields\": [\n   {\n   \"column\": \"precision\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.41817983689130067,\n   \"min\": 0.05415162454873646,\n   \"max\": 0.9998553345388789,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.05415162454873646,\n   0.9982283031218108,\n   0.9722622098943156\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.02411293361417645,\n   \"min\": 0.9183673469387755,\n   \"max\": 0.9723550928531233,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   0.9183673469387755,\n   0.9453612198959493,\n   0.9723550928531233\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"f1-score\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.3926689807140676,\n   \"min\": 0.10227272727272728,\n   \"max\": 0.985913483827253,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.10227272727272728,\n   0.9843932283210849,\n   0.9722622098943156\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"support\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 31154.419955907684,\n   \"min\": 0.9722622098943156,\n   \"max\": 56962.0,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   98.0,\n   56962.0,\n   56864.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}




```
Logistic Regression AUC-ROC: 0.9720, PR AUC: 0.7622 (Saved internally)
```

KNN

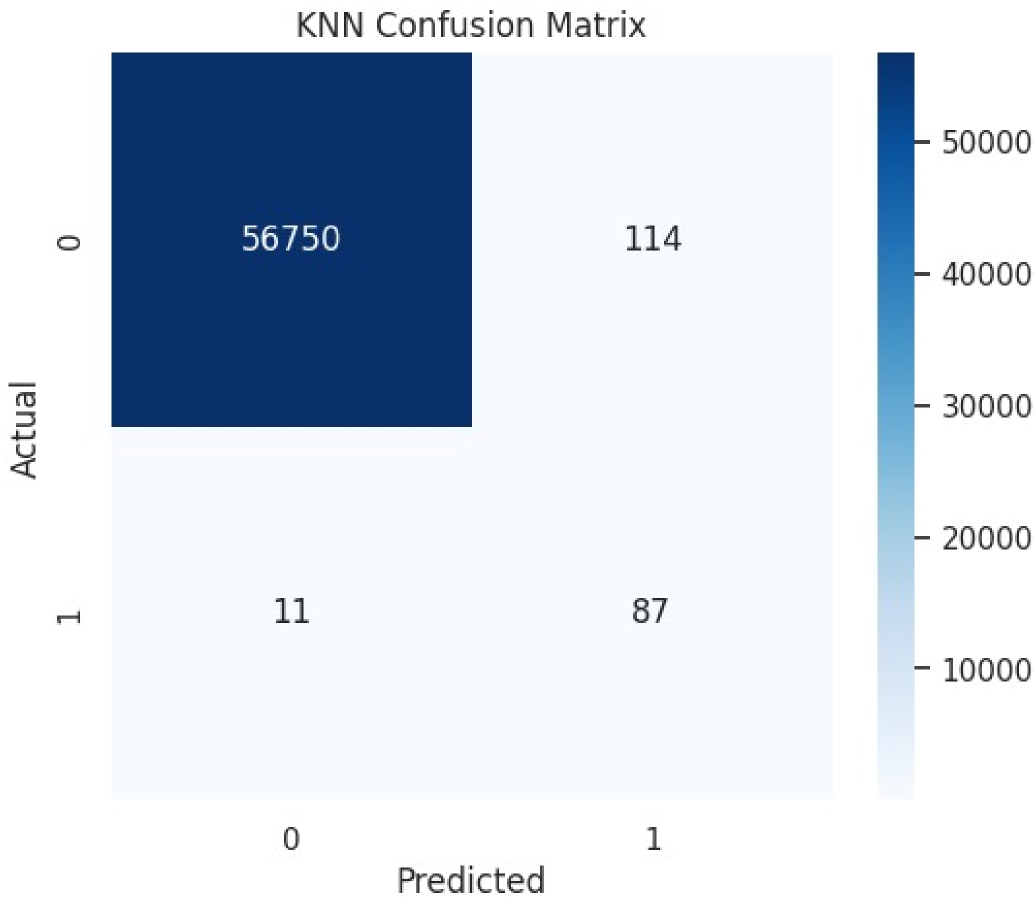
```
knn_smote_path = "/content/drive/My Drive/Colab
Notebooks/knn_SMOTE_metrics.json"

if os.path.exists(knn_smote_path):
    with open(knn_smote_path, 'r') as f:
        saved_report = json.load(f)
    print("KNN Classification Report:")
    display(pd.DataFrame(saved_report).transpose())

    knn_model = KNeighborsClassifier()
    knn_model.fit(X_train_smote, y_train_smote)
    y_pred_knn = knn_model.predict(X_test_time)
    y_proba_knn = knn_model.predict_proba(X_test_time)
    save_metrics_and_outputs("KNN", y_test, y_pred_knn, y_proba_knn,
knn_smote_path, internal_auc_scores)

else:
```

```python
    print("No saved report found. Training KNN on SMOTE data...")
    knn_model = KNeighborsClassifier()
    knn_model.fit(X_train_smote, y_train_smote)
    y_pred_knn = knn_model.predict(X_test_time)
    y_proba_knn = knn_model.predict_proba(X_test_time)
    save_metrics_and_outputs("KNN", y_test, y_pred_knn, y_proba_knn,
knn_smote_path, internal_auc_scores)
```

```
No saved report found. Training KNN on SMOTE data...
```

```
KNN Classification Report:
```

{"summary":"{\n \"name\": \"   save_metrics_and_outputs(\\\"KNN\\\", y_test, y_pred_knn, y_proba_knn, knn_smote_path, internal_auc_scores)\",\n \"rows\": 5,\n \"fields\": [\n   {\n   \"column\": \"precision\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.25305883813054986,\n   \"min\": 0.43283582089552236,\n   \"max\": 0.9998062049646764,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.43283582089552236,\n   0.9988307634837106,\n   0.9978055545802464\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.0492373319231268,\n   \"min\": 0.8877551020408163,\n   \"max\": 0.9979952166572875,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   0.8877551020408163,\n   0.9428751593490519,\n   0.9979952166572875\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"f1-score\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.18616685219859616,\n   \"min\": 0.5819397993311036,\n   \"max\": 0.9988998899889989,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.5819397993311036,\n   0.9981825329986461,\n   0.9978055545802464\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"support\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 31154.412950641894,\n   \"min\": 0.9978055545802464,\n   \"max\": 56962.0,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   98.0,\n   56962.0,\n   56864.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}




```
KNN AUC-ROC: 0.9484, PR AUC: 0.7570 (Saved internally)
```

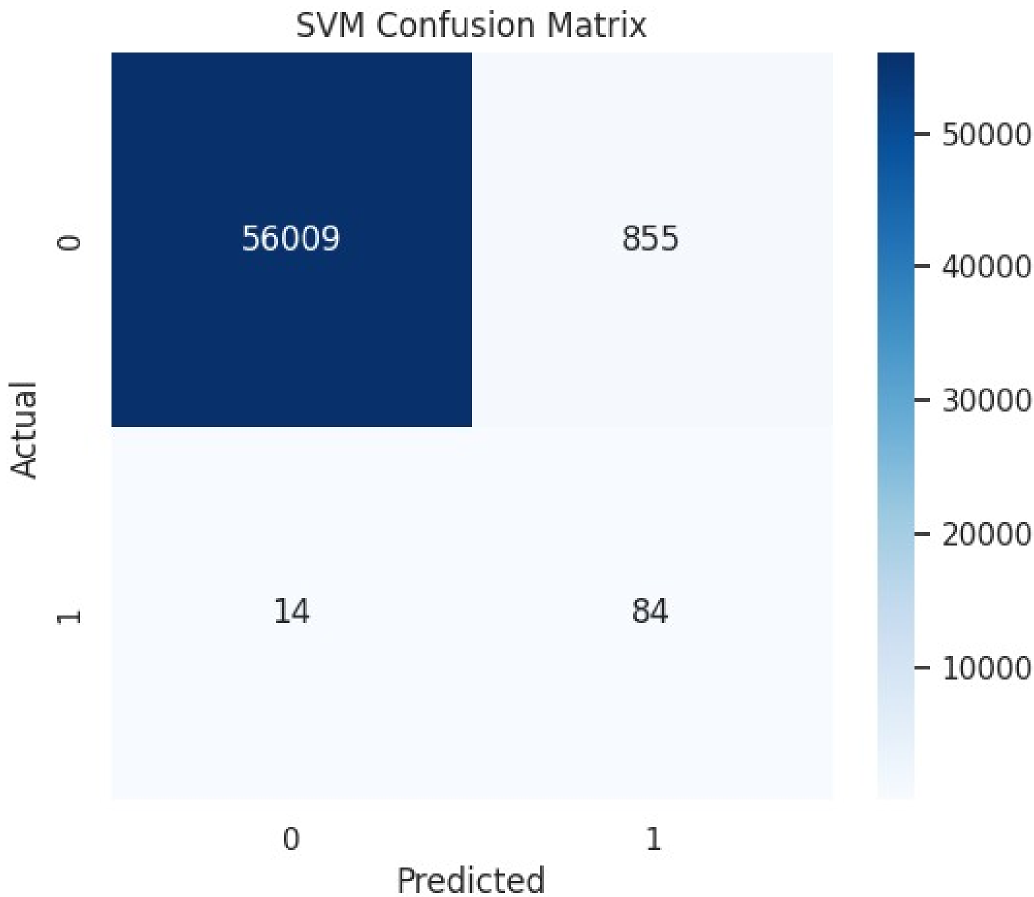
SVM

```
svm_smote_path = "/content/drive/My Drive/Colab
Notebooks/svm_SMOTE_metrics.json"

if os.path.exists(svm_smote_path):
    with open(svm_smote_path, 'r') as f:
        saved_report = json.load(f)
    print("SVM Classification Report:")
    display(pd.DataFrame(saved_report).transpose())

    svm_model = SVC(probability=True, random_state=42)
    svm_model.fit(X_train_smote, y_train_smote)
    y_pred_svm = svm_model.predict(X_test_time)
    y_proba_svm = svm_model.predict_proba(X_test_time)
    save_metrics_and_outputs("SVM", y_test, y_pred_svm, y_proba_svm,
svm_smote_path, internal_auc_scores)

else:
```

```
    print("No saved report found. Training SVM on SMOTE data...")
    svm_model = SVC(probability=True, random_state=42)
    svm_model.fit(X_train_smote, y_train_smote)
    y_pred_svm = svm_model.predict(X_test_time)
    y_proba_svm = svm_model.predict_proba(X_test_time)
    save_metrics_and_outputs("SVM", y_test, y_pred_svm, y_proba_svm,
svm_smote_path, internal_auc_scores)
```

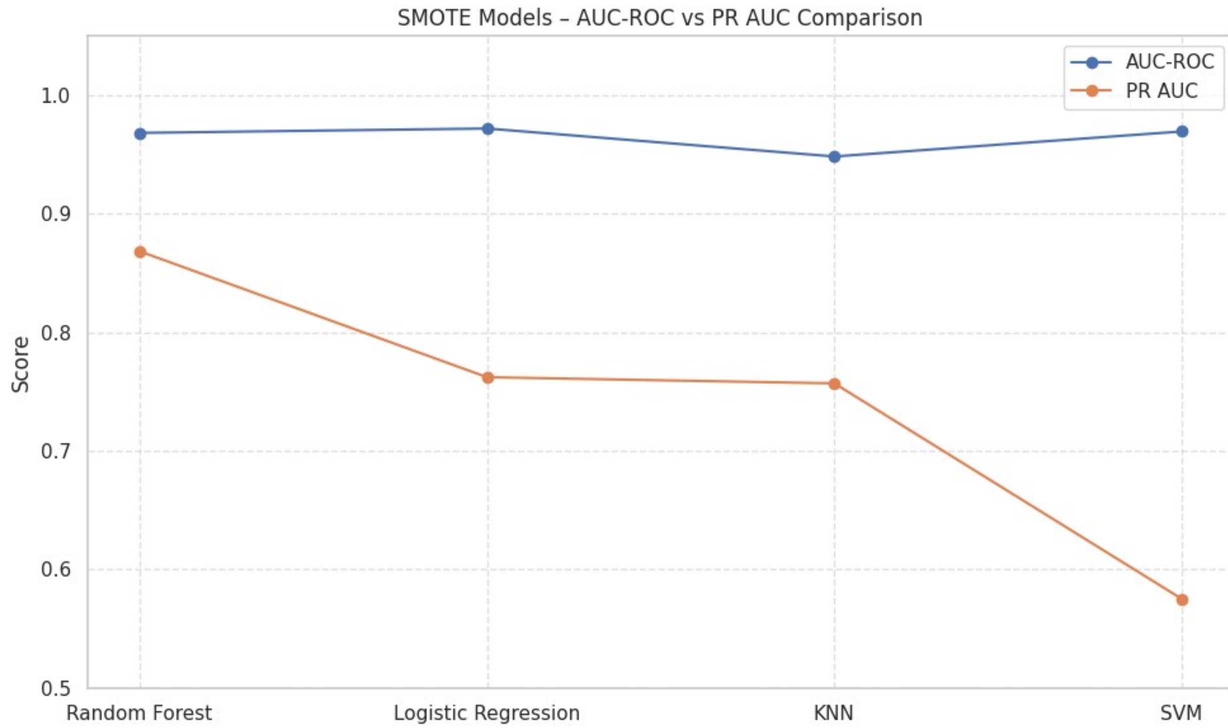
```
No saved report found. Training SVM on SMOTE data...

SVM Classification Report:
```

{"summary":"{\n \"name\": \"   save_metrics_and_outputs(\\\"SVM\\\", y_test, y_pred_svm, y_proba_svm, svm_smote_path, internal_auc_scores)\",\n \"rows\": 5,\n \"fields\": [\n   {\n   \"column\": \"precision\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.40436014088181743,\n   \"min\": 0.08945686900958466,\n   \"max\": 0.9997501026364172,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.08945686900958466,\n   0.9981839930037598,\n   0.9847442154418735\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.057089728278576994,\n   \"min\": 0.8571428571428571,\n   \"max\": 0.9849641249296567,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   0.8571428571428571,\n   0.9210534910362569,\n   0.9849641249296567\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"f1-score\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.3698241902127021,\n   \"min\": 0.16200578592092574,\n   \"max\": 0.9923020365498242,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.16200578592092574,\n   0.990873557343307,\n   0.9847442154418735\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"support\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 31154.41653271562,\n   \"min\": 0.9847442154418735,\n   \"max\": 56962.0,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   98.0,\n   56962.0,\n   56864.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}




```
SVM AUC-ROC: 0.9695, PR AUC: 0.5752 (Saved internally)
```

```python
import matplotlib.pyplot as plt

# Model names and manually provided scores
models = ['Random Forest', 'Logistic Regression', 'KNN', 'SVM']
auc_roc_scores = [0.9683, 0.9720, 0.9484, 0.9695]
pr_auc_scores = [0.8684, 0.7622, 0.7570, 0.5752]

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(models, auc_roc_scores, marker='o', label='AUC-ROC')
plt.plot(models, pr_auc_scores, marker='o', label='PR AUC')

# Labels and title
plt.title('SMOTE Models – AUC-ROC vs PR AUC Comparison')
plt.ylabel('Score')
plt.ylim(0.5, 1.05)
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.show()
```



ADASYN

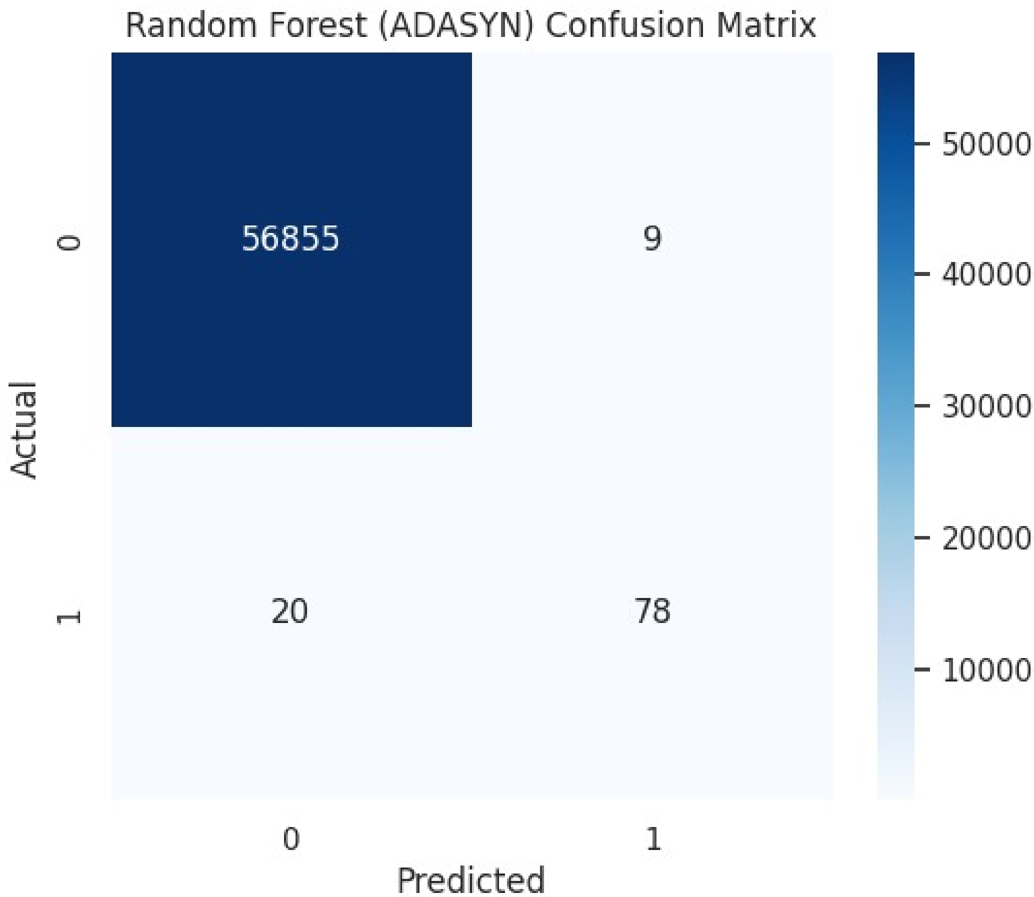
Random Forest

```
model_name = "Random Forest (ADASYN)"
rf_adasyn_path = "/content/drive/My Drive/Colab
Notebooks/rf_ADASYN_metrics.json"

if os.path.exists(rf_adasyn_path):
    with open(rf_adasyn_path, 'r') as f:
        saved_report = json.load(f)
    print(f"{model_name} Classification Report:")
    display(pd.DataFrame(saved_report).transpose())

    rf_model = RandomForestClassifier(random_state=42)
    rf_model.fit(X_train_adasyn, y_train_adasyn)
    y_pred_rf = rf_model.predict(X_test_time)
    y_proba_rf = rf_model.predict_proba(X_test_time)

    save_metrics_and_outputs(model_name, y_test, y_pred_rf,
y_proba_rf, rf_adasyn_path, internal_auc_scores)

else:
    print(f"No saved report found. Training {model_name} on ADASYN
data...")
    rf_model = RandomForestClassifier(random_state=42)
    rf_model.fit(X_train_adasyn, y_train_adasyn)
```

```
    y_pred_rf = rf_model.predict(X_test_time)
    y_proba_rf = rf_model.predict_proba(X_test_time)

    save_metrics_and_outputs(model_name, y_test, y_pred_rf,
y_proba_rf, rf_adasyn_path, internal_auc_scores)
```

```
No saved report found. Training Random Forest (ADASYN) on ADASYN
data...

Random Forest (ADASYN) Classification Report:
```


{"summary":"{\n \"name\": \"   save_metrics_and_outputs(model_name, y_test, y_pred_rf, y_proba_rf, rf_adasyn_path, internal_auc_scores)\",\n \"rows\": 5,\n \"fields\": [\n   {\n   \"column\": \"precision\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.04605011752147693,\n   \"min\": 0.896551724137931,\n   \"max\": 0.9996483516483516,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.896551724137931,\n   0.9994709795494783,\n   0.9994908886626171\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.09107975088840796,\n   \"min\": 0.7959183673469388,\n   \"max\": 0.9998417276308385,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   0.7959183673469388,\n   0.8978800474888886,\n   0.9998417276308385\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"f1-score\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.06990204303580122,\n   \"min\": 0.8432432432432433,\n   \"max\": 0.9997450302886433,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.8432432432432433,\n   0.9994757775389074,\n   0.9994908886626171\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"support\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 31154.412488438877,\n   \"min\": 0.9994908886626171,\n   \"max\": 56962.0,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   98.0,\n   56962.0,\n   56864.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}




```
Random Forest (ADASYN) AUC-ROC: 0.9746, PR AUC: 0.8606 (Saved
internally)
```

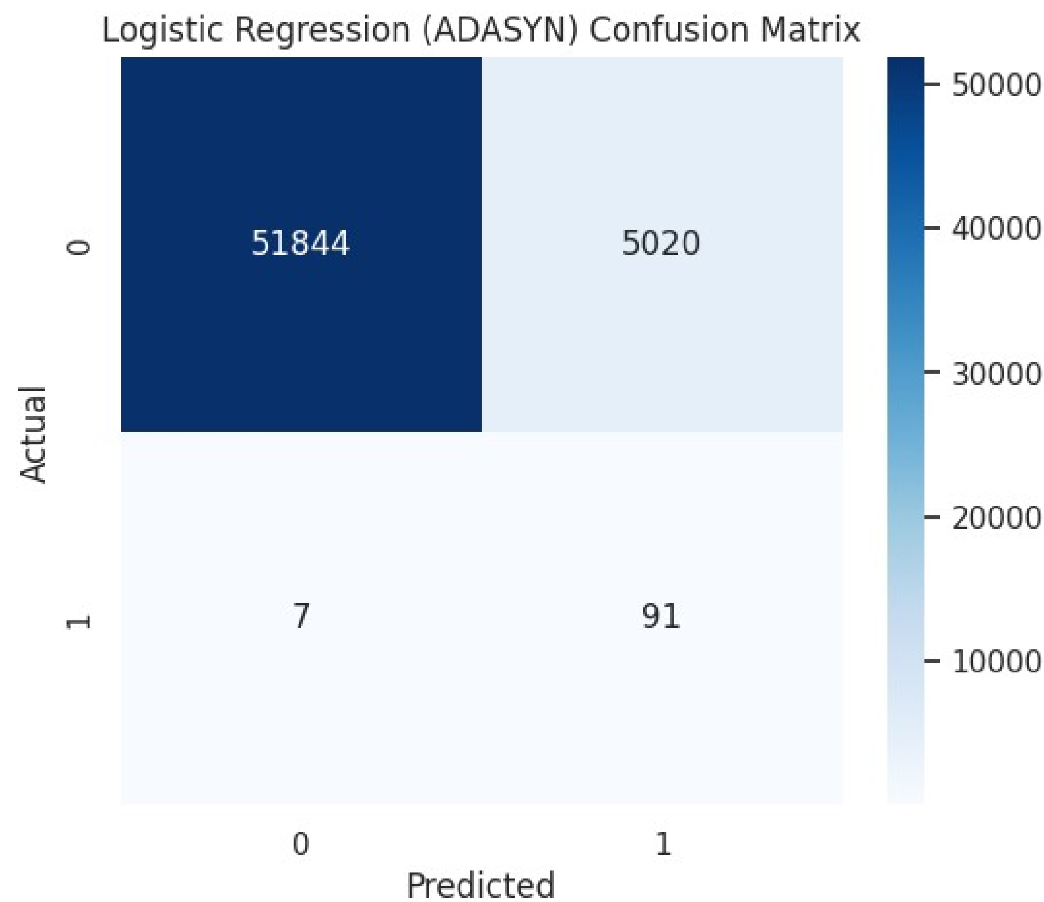
Logistic regression

```
model_name = "Logistic Regression (ADASYN)"
lr_adasyn_path = "/content/drive/My Drive/Colab
Notebooks/lr_ADASYN_metrics.json"

if os.path.exists(lr_adasyn_path):
    with open(lr_adasyn_path, 'r') as f:
        saved_report = json.load(f)
    print(f"{model_name} Classification Report:")
    display(pd.DataFrame(saved_report).transpose())

    lr_model = LogisticRegression(max_iter=1000)
    lr_model.fit(X_train_adasyn, y_train_adasyn)
    y_pred_lr = lr_model.predict(X_test_time)
    y_proba_lr = lr_model.predict_proba(X_test_time)

    save_metrics_and_outputs(model_name, y_test, y_pred_lr,
```

```
y_proba_lr, lr_adasyn_path, internal_auc_scores)

else:
    print(f"No saved report found. Training {model_name} on ADASYN
data...")
    lr_model = LogisticRegression(max_iter=1000)
    lr_model.fit(X_train_adasyn, y_train_adasyn)
    y_pred_lr = lr_model.predict(X_test_time)
    y_proba_lr = lr_model.predict_proba(X_test_time)

    save_metrics_and_outputs(model_name, y_test, y_pred_lr,
y_proba_lr, lr_adasyn_path, internal_auc_scores)
```

```
No saved report found. Training Logistic Regression (ADASYN) on ADASYN
data...

Logistic Regression (ADASYN) Classification Report:
```


```
{"summary":"{\n  \"name\": \"    save_metrics_and_outputs(model_name,
y_test, y_pred_lr, y_proba_lr, lr_adasyn_path,
internal_auc_scores)\",\n  \"rows\": 5,\n  \"fields\": [\n    {\n
\"column\": \"precision\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0.42567332029439975,\n
\"min\": 0.01780473488554099,\n        \"max\": 0.9998649977821065,\n
\"num_unique_values\": 5,\n        \"samples\": [\n
0.01780473488554099,\n          0.9981754169077716,\n
0.9117481829991925\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"recall\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 0.007526836166054648,\n        \"min\": 0.9117191896454699,\n
\"max\": 0.9285714285714286,\n        \"num_unique_values\": 4,\n
\"samples\": [\n          0.9285714285714286,\n
0.9201453091084493,\n          0.9117191896454699\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"f1-score\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\":
0.40396158611943783,\n        \"min\": 0.03493952774044922,\n
\"max\": 0.9537598307501265,\n        \"num_unique_values\": 5,\n
\"samples\": [\n          0.03493952774044922,\n
0.9521790507618019,\n          0.9117481829991925\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"support\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\":
31154.436551898303,\n        \"min\": 0.9117481829991925,\n
\"max\": 56962.0,\n        \"num_unique_values\": 4,\n
\"samples\": [\n          98.0,\n          56962.0,\n
56864.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\n}","type":"dataframe"}
```




```
Logistic Regression (ADASYN) AUC-ROC: 0.9739, PR AUC: 0.7670 (Saved
internally)
```

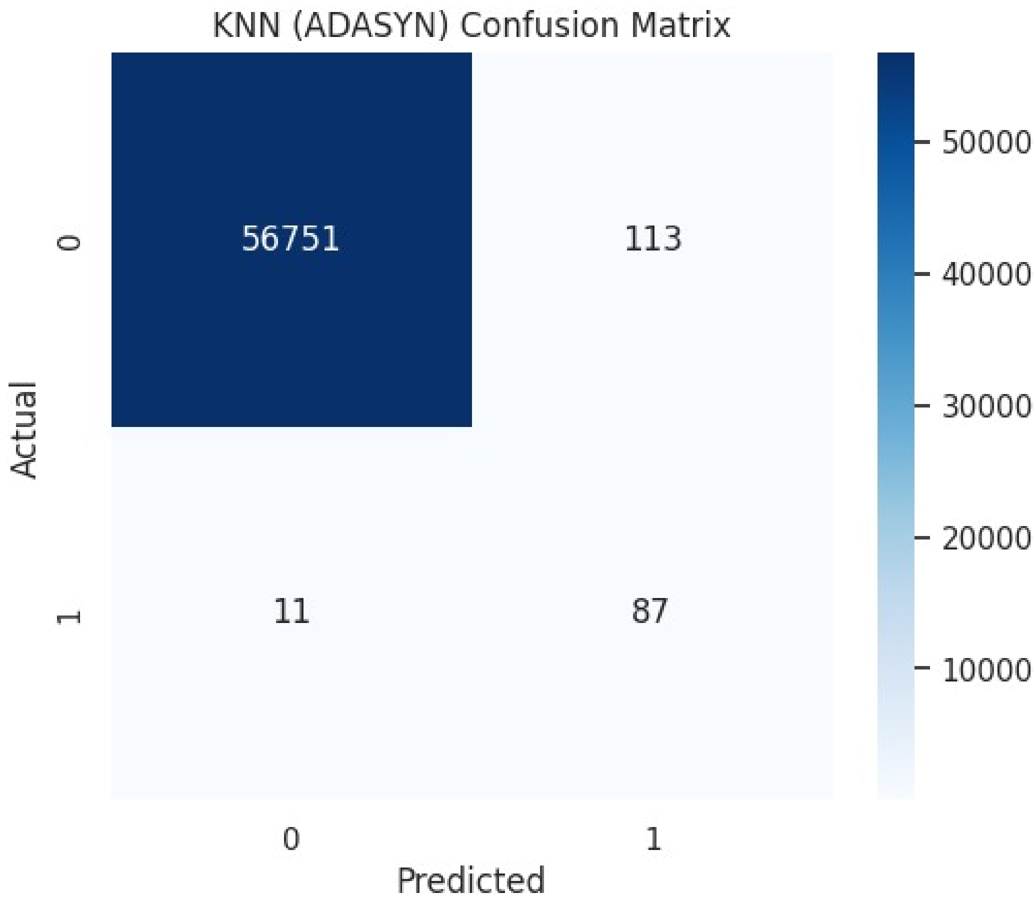
KNN

```
model_name = "KNN (ADASYN)"
knn_adasyn_path = "/content/drive/My Drive/Colab
Notebooks/knn_ADASYN_metrics.json"

if os.path.exists(knn_adasyn_path):
    with open(knn_adasyn_path, 'r') as f:
        saved_report = json.load(f)
    print(f"{model_name} Classification Report:")
    display(pd.DataFrame(saved_report).transpose())

    knn_model = KNeighborsClassifier()
    knn_model.fit(X_train_adasyn, y_train_adasyn)
    y_pred_knn = knn_model.predict(X_test_time)
    y_proba_knn = knn_model.predict_proba(X_test_time)

    save_metrics_and_outputs(model_name, y_test, y_pred_knn,
```

```
y_proba_knn, knn_adasyn_path, internal_auc_scores)

else:
    print(f"No saved report found. Training {model_name} on ADASYN data...")
    knn_model = KNeighborsClassifier()
    knn_model.fit(X_train_adasyn, y_train_adasyn)
    y_pred_knn = knn_model.predict(X_test_time)
    y_proba_knn = knn_model.predict_proba(X_test_time)

    save_metrics_and_outputs(model_name, y_test, y_pred_knn,
y_proba_knn, knn_adasyn_path, internal_auc_scores)
```

```
No saved report found. Training KNN (ADASYN) on ADASYN data...

KNN (ADASYN) Classification Report:
```

{"summary":"{\n \"name\": \"   save_metrics_and_outputs(model_name, y_test, y_pred_knn, y_proba_knn, knn_adasyn_path, internal_auc_scores)\",\n \"rows\": 5,\n \"fields\": [\n   {\n   \"column\": \"precision\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.2520945427647179,\n   \"min\": 0.435,\n   \"max\": 0.999806208378845,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.435,\n   0.9988344902435772,\n   0.9978231101436045\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.0492451864033239 26,\n   \"min\": 0.8877551020408163,\n   \"max\": 0.9980128024760833,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   0.8877551020408163,\n   0.9428839522584498,\n   0.9980128024760833\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"f1-score\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.18529948832050916,\n   \"min\": 0.5838926174496645,\n   \"max\": 0.9989087004734832,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.5838926174496645,\n   0.9981946880417509,\n   0.9978231101436045\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"support\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 31154.412945827276,\n   \"min\": 0.9978231101436045,\n   \"max\": 56962.0,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   98.0,\n   56962.0,\n   56864.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}




```
KNN (ADASYN) AUC-ROC: 0.9485, PR AUC: 0.7618 (Saved internally)
```

SVM

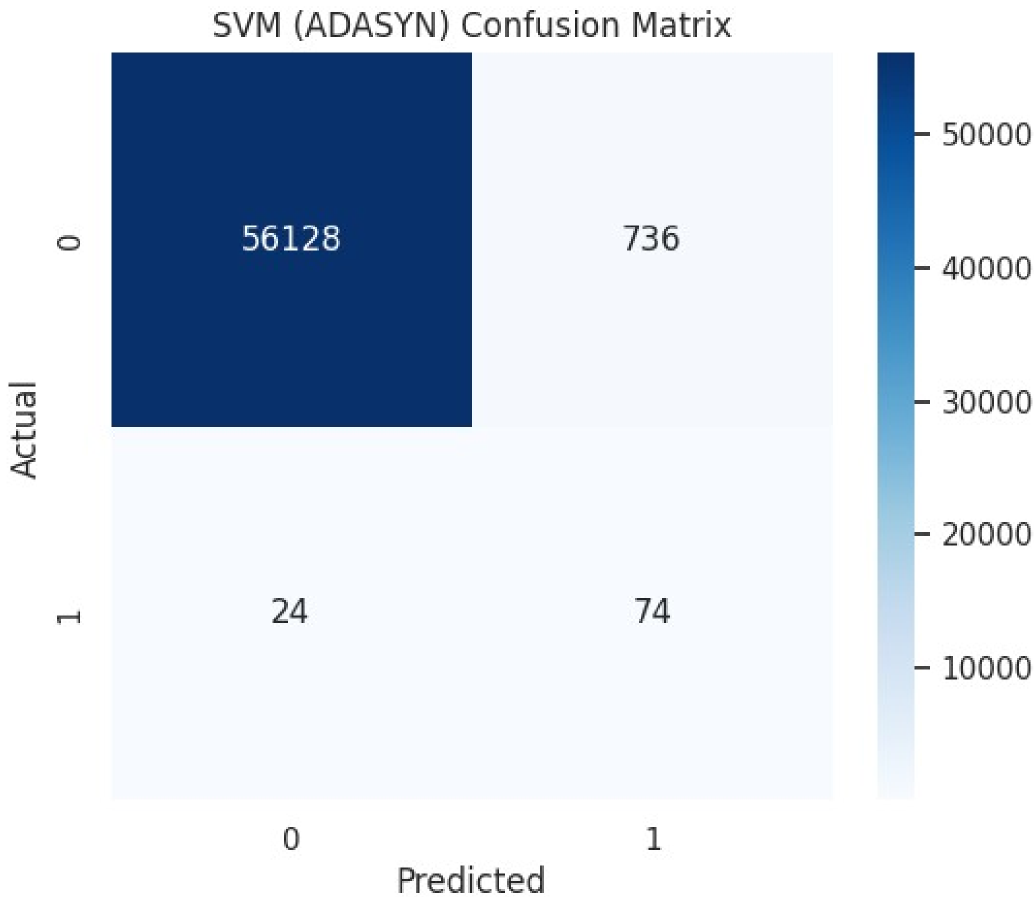
```
model_name = "SVM (ADASYN)"
svm_adasyn_path = "/content/drive/My Drive/Colab
Notebooks/svm_ADASYN_metrics.json"

if os.path.exists(svm_adasyn_path):
    with open(svm_adasyn_path, 'r') as f:
        saved_report = json.load(f)
    print(f"{model_name} Classification Report:")
    display(pd.DataFrame(saved_report).transpose())

    svm_model = SVC(probability=True)
    svm_model.fit(X_train_adasyn, y_train_adasyn)
    y_pred_svm = svm_model.predict(X_test_time)
    y_proba_svm = svm_model.predict_proba(X_test_time)

    save_metrics_and_outputs(model_name, y_test, y_pred_svm,
y_proba_svm, svm_adasyn_path, internal_auc_scores)
```

```python
else:
    print(f"No saved report found. Training {model_name} on ADASYN data...")
    svm_model = SVC(probability=True)
    svm_model.fit(X_train_adasyn, y_train_adasyn)
    y_pred_svm = svm_model.predict(X_test_time)
    y_proba_svm = svm_model.predict_proba(X_test_time)

    save_metrics_and_outputs(model_name, y_test, y_pred_svm, y_proba_svm, svm_adasyn_path, internal_auc_scores)
```

```
No saved report found. Training SVM (ADASYN) on ADASYN data...
```

```
SVM (ADASYN) Classification Report:
```

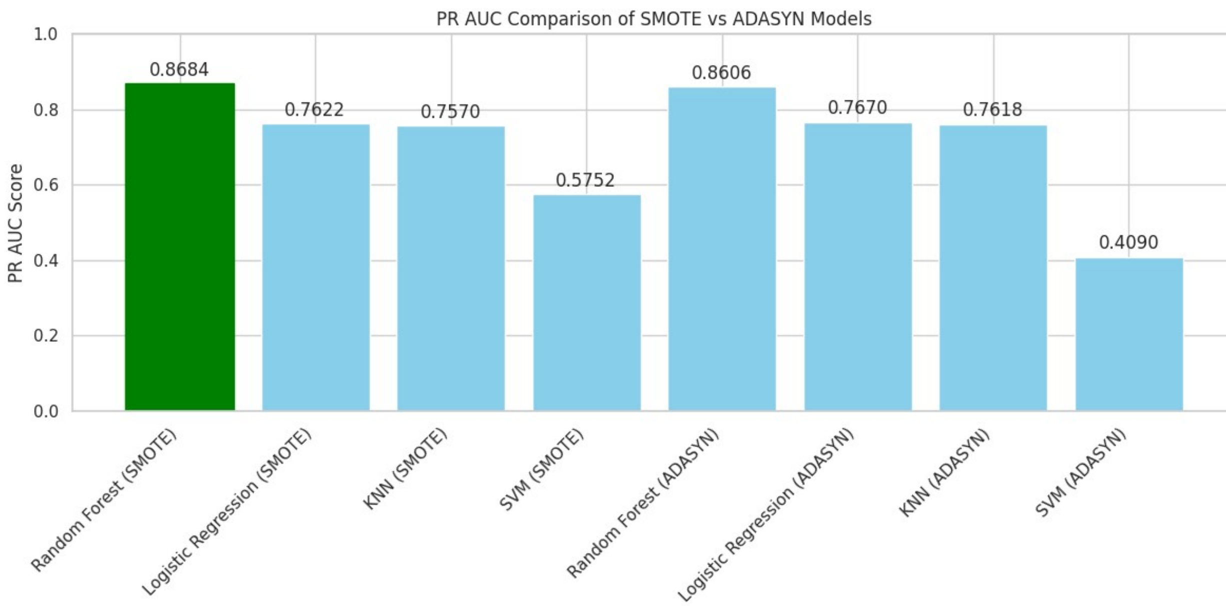
{"summary":"{\n \"name\": \"   save_metrics_and_outputs(model_name, y_test, y_pred_svm, y_proba_svm, svm_adasyn_path, internal_auc_scores)\",\n \"rows\": 5,\n \"fields\": [\n   {\n   \"column\": \"precision\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.40377009128828484,\n   \"min\": 0.09135802469135802,\n   \"max\": 0.9995725886878473,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.09135802469135802,\n   0.9980100552923967,\n   0.9866577718478986\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"recall\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.10359963202712717,\n   \"min\": 0.7551020408163265,\n   \"max\": 0.9870568373663478,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   0.7551020408163265,\n   0.8710794390913371,\n   0.9870568373663478\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"f1-score\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.3699716672858355,\n   \"min\": 0.16299559471365638,\n   \"max\": 0.9932752884547321,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.16299559471365638,\n   0.9918468377334332,\n   0.9866577718478986\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"support\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 31154.416007922515,\n   \"min\": 0.9866577718478986,\n   \"max\": 56962.0,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   98.0,\n   56962.0,\n   56864.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}




```
SVM (ADASYN) AUC-ROC: 0.9254, PR AUC: 0.4090 (Saved internally)
```

```
import matplotlib.pyplot as plt

# PR AUC scores
models = [
    'Random Forest (SMOTE)', 'Logistic Regression (SMOTE)',
    'KNN (SMOTE)', 'SVM (SMOTE)',
    'Random Forest (ADASYN)', 'Logistic Regression (ADASYN)',
    'KNN (ADASYN)', 'SVM (ADASYN)'
]
pr_auc_scores = [0.8684, 0.7622, 0.7570, 0.5752, 0.8606, 0.7670,
0.7618, 0.4090]

# Plotting
plt.figure(figsize=(12, 6))
bars = plt.bar(models, pr_auc_scores, color='skyblue')
bars[0].set_color('green')  # Highlight best performing (Random Forest
SMOTE)

# Annotate scores
for bar in bars:
```

```python
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2.0, yval + 0.01,
f'{yval:.4f}', ha='center', va='bottom')

plt.title('PR AUC Comparison of SMOTE vs ADASYN Models')
plt.ylabel('PR AUC Score')
plt.ylim(0, 1)
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```




```python
import joblib

# === Finalize and Save the Model ===
final_rf_model = RandomForestClassifier(random_state=42)
final_rf_model.fit(X_train_smote, y_train_smote)

# Save the trained model
joblib.dump(final_rf_model, "final_rf_smote_thresh050_model.pkl")
print(" Final Random Forest model (SMOTE, threshold=0.50) saved
successfully.")

# Optional: Save your threshold value separately (for consistent
inference)
with open("rf_threshold.txt", "w") as f:
    f.write("0.50")
```

```
 Final Random Forest model (SMOTE, threshold=0.50) saved successfully.
```