Credit Card Fraud Detection

In [ ]:
```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import RobustScaler
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.metrics import classification_report, roc_auc_score, precision_recall_cur
import matplotlib.pyplot as plt
import seaborn as sns
import time
```

Introduction

In this project, we aim to identify fraudulent transactions within a highly imbalanced dataset. This notebook details the step-by-step process of building a fraud detection model, starting from data preprocessing and exploratory data analysis (EDA) to model selection, hyperparameter tuning, and performance evaluation. By applying techniques such as log transformation, SMOTE for handling class imbalance, and evaluating multiple machine learning models, we aim to develop a robust model capable of distinguishing fraudulent transactions from legitimate ones.

Data Preprocessing

1. Loading Data

This data originates from a study done in collaboration by Worldline and a research group from the Université Libre de Bruxelles, and can be found here. This dataset contains real-world transaction data from European card holders over the span of two days. To protect sensitive consumer data, all features except for Time, Amount, and Class, have been transformed using PCA.

In [ ]:
```python
# Load data
data = pd.read_csv('../Data/creditcard.csv')
data.head()
```

Out[ ]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.3637
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.2554
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.5146
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.3870
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.8177

5 rows × 31 columns

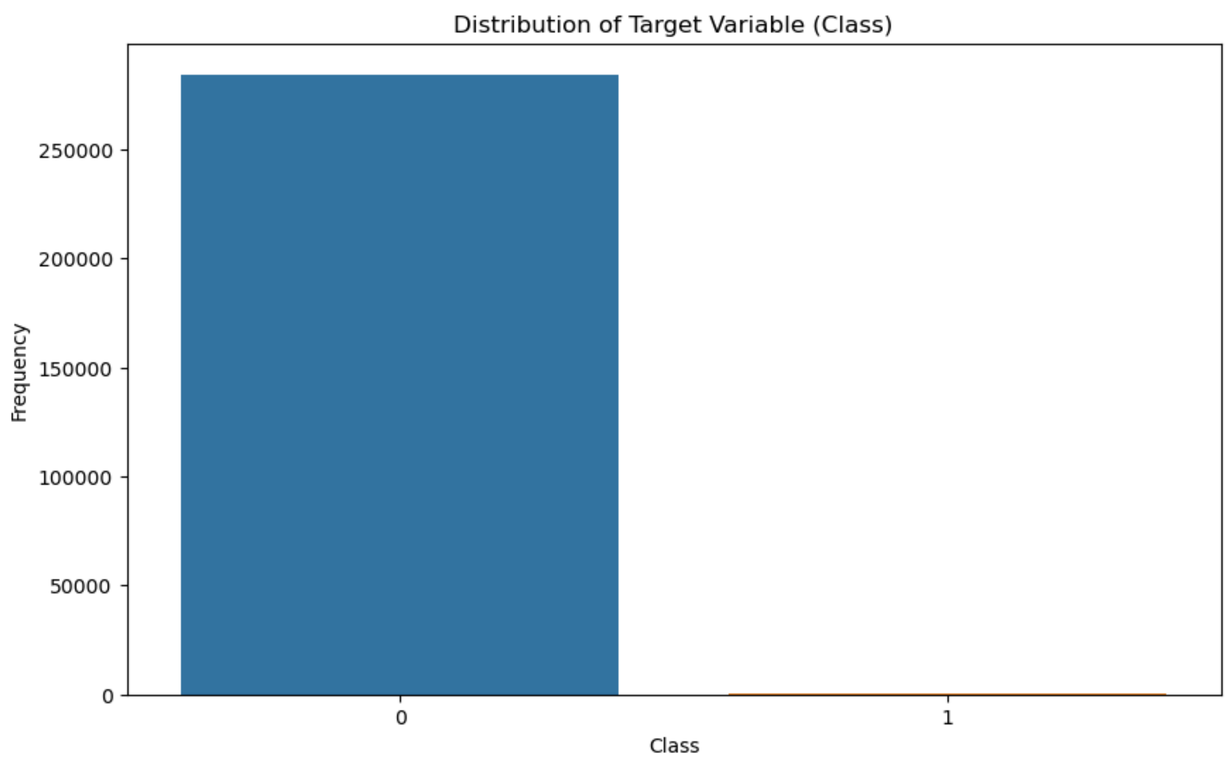
In [ ]:
```python
# check for null values
data.isnull().sum().max()
```

Out[ ]:
```
0
```

Exploratory Data Analysis

Class Imbalance

In [ ]:
```python
# Distribution of the target variable 'Class'
plt.figure(figsize=(10, 6))
sns.countplot(x='Class', data=data)
plt.title('Distribution of Target Variable (Class)')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.show()

data['Class'].value_counts(normalize=True)
```

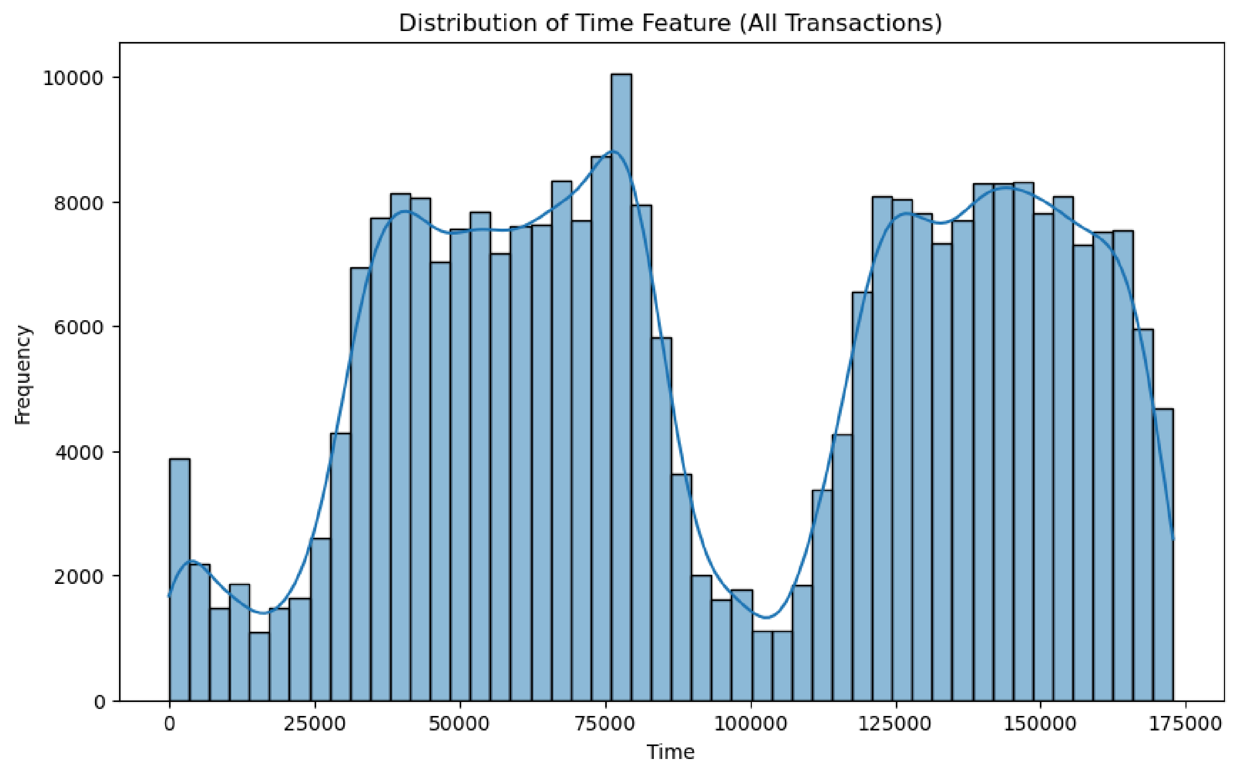


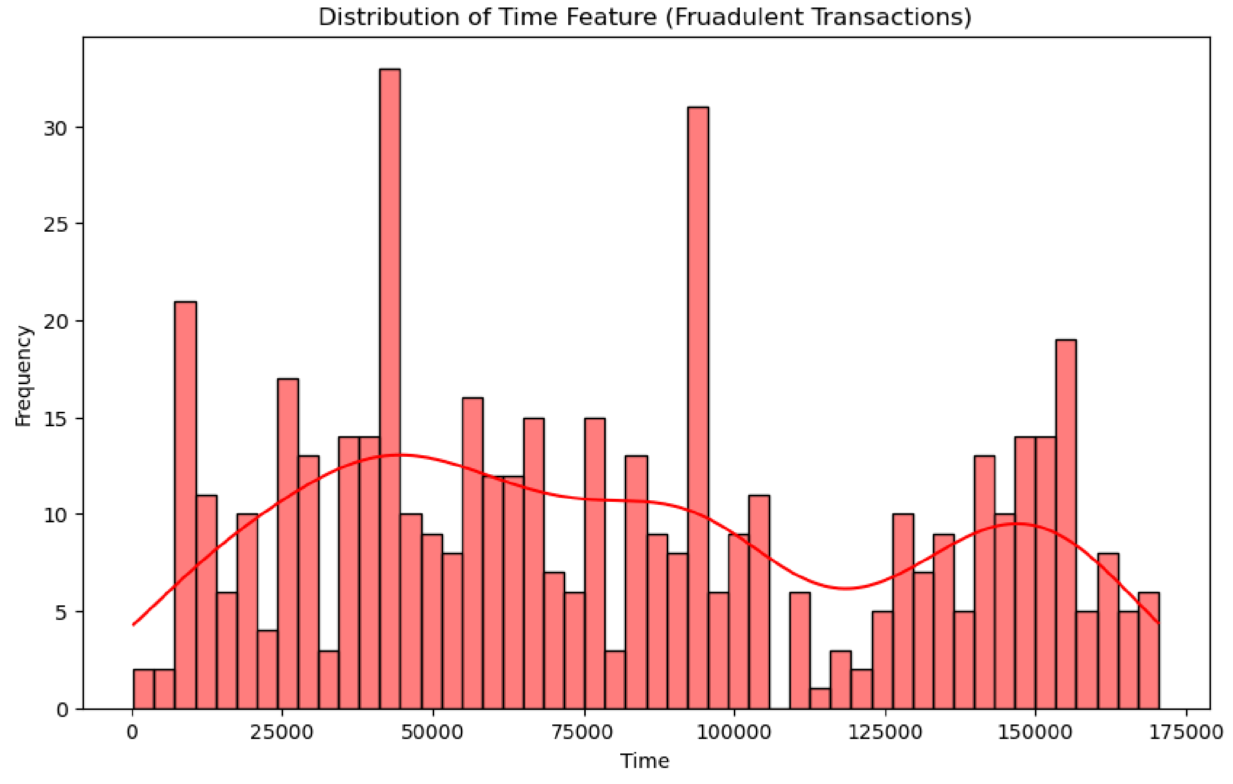
Out[ ]:
```
0    0.998273
1    0.001727
Name: Class, dtype: float64
```

The graphic above illustrates the severe class imbalance present in the dataset. Non-fraudulent transactions (Class 0) comprise the overwhelming majority or the dataset at over 99.8% of transactions present, fraudulent transactions (Class 1) make up less than 0.2%. This severe imbalance poses a risk of the model becoming biased towards predicting the majority class, resulting in poor detection of fraud. This must be addressed using techniques like SMOTE to rebalance the dataset and evaluating models with the appropriate metrics.

Distributions

In [ ]:
```python
# Distribution of the 'Time' feature
plt.figure(figsize=(10, 6))
sns.histplot(data['Time'], bins=50, kde=True)
plt.title('Distribution of Time Feature (All Transactions)')
plt.xlabel('Time')
plt.ylabel('Frequency')
plt.show()

# Create datafame only containing fraud cases
fraud_data = data[data['Class'] == 1]

# Distribution of the 'Time' feature for fraudulent transactions
plt.figure(figsize=(10, 6))
sns.histplot(fraud_data['Time'], bins=50, kde=True, color='red')
plt.title('Distribution of Time Feature (Fruadulent Transactions)')
plt.xlabel('Time')
plt.ylabel('Frequency')
plt.show()
```



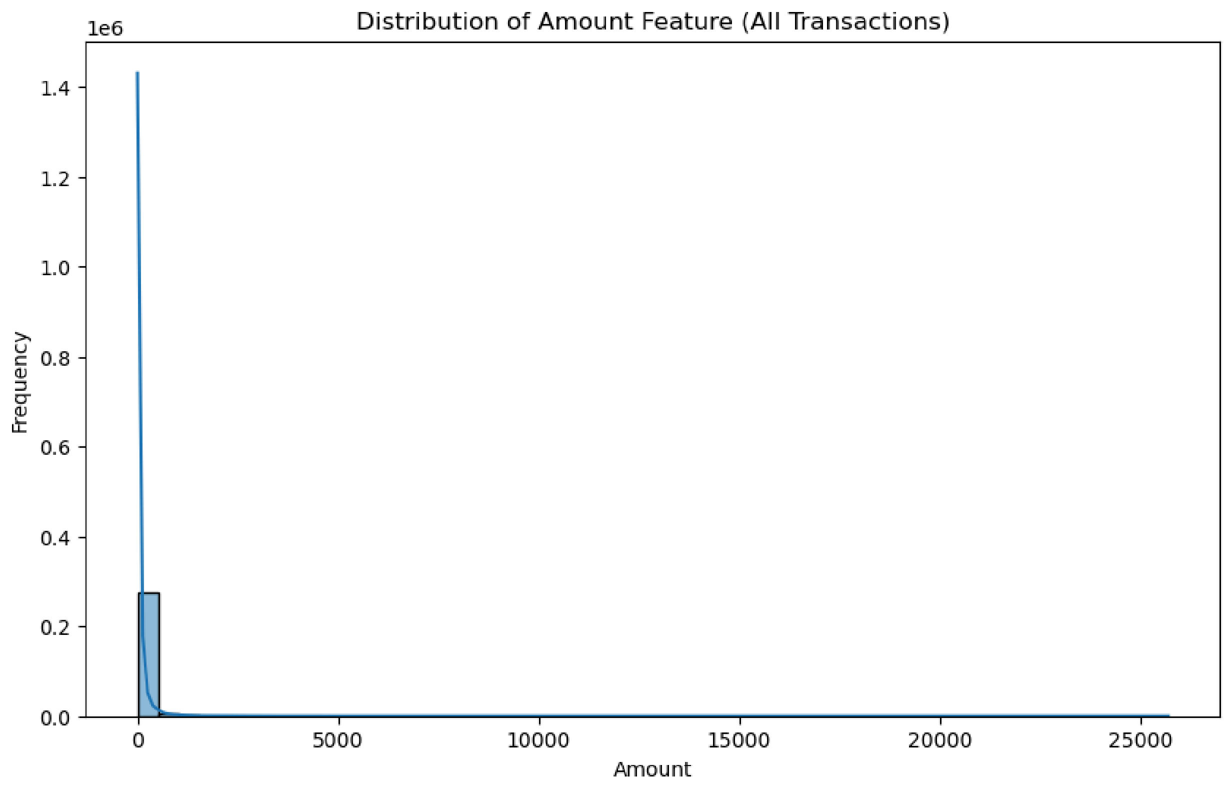


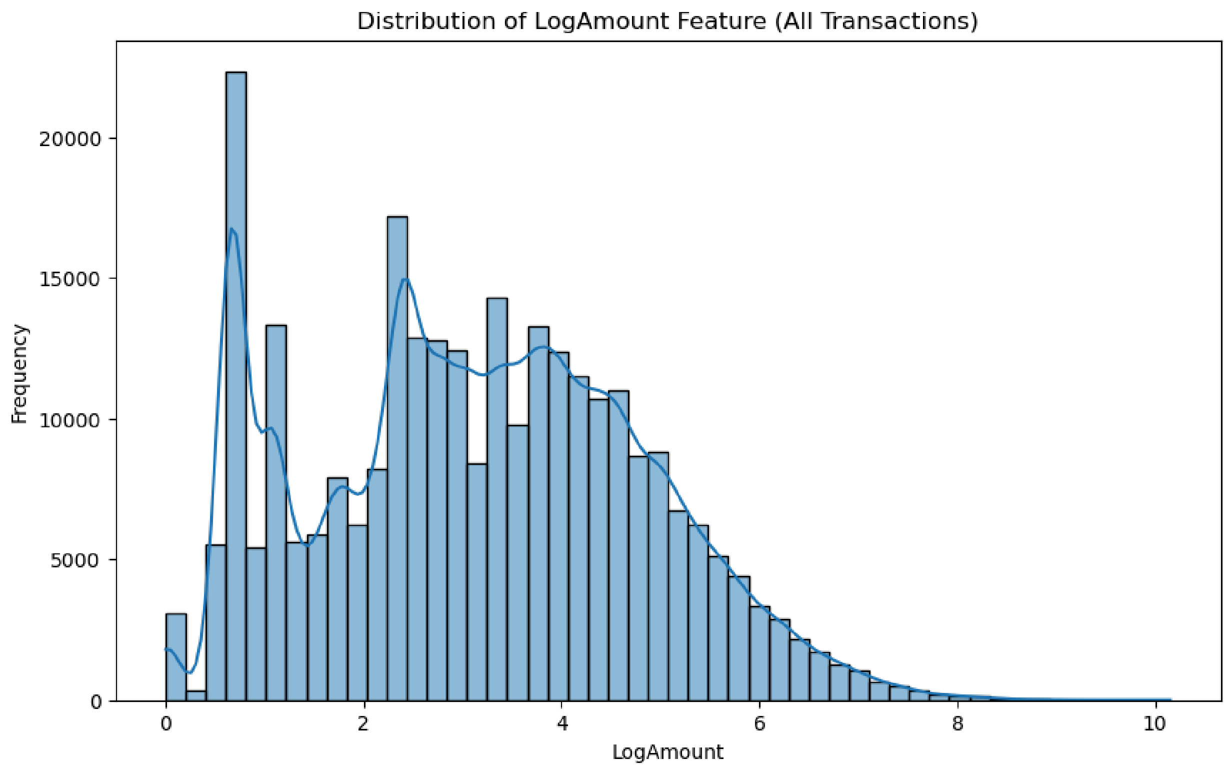
The histograms above show the distribution of the Time feature for all transactions and fraudulent transactions. The Time feature represents the time in seconds elapsed from a specific starting point. For all transactions, the distribution exhibits a bimodal pattern with two prominent peaks, indicating periods of high transaction activity. This suggests that transaction volume varies significantly based on the time of day.

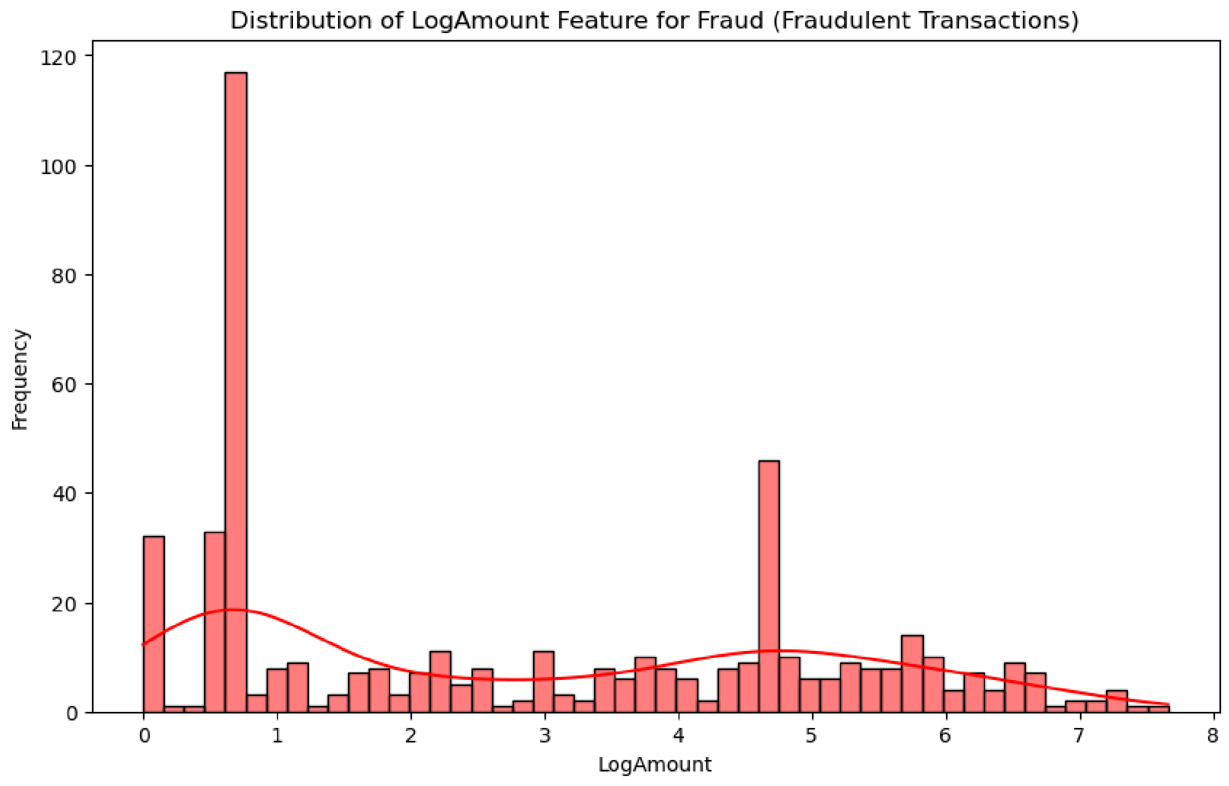
For fraudulent transactions, the distribution is more uniform and less pronounced, lacking the clear peaks observed in the overall transaction distribution. This indicates that fraud can occur at any time, without a strong temporal pattern.

```
In [ ]: # Distribution of the 'Amount' feature
plt.figure(figsize=(10, 6))
sns.histplot(data['Amount'], bins=50, kde=True)
plt.title('Distribution of Amount Feature (All Transactions)')
plt.xlabel('Amount')
plt.ylabel('Frequency')
plt.show()

# Apply log transformation to the Amount feature
data['LogAmount'] = np.log1p(data['Amount'])

# Distribution of the 'LogAmount' feature
plt.figure(figsize=(10, 6))
sns.histplot(data['LogAmount'], bins=50, kde=True)
plt.title('Distribution of LogAmount Feature (All Transactions)')
```

```python
plt.xlabel('LogAmount')
plt.ylabel('Frequency')
plt.show()

# Recopy fraudulent data table so it includes LogAmount
fraud_data = data[data['Class'] == 1]

# Distribution of the 'LogAmount' feature
plt.figure(figsize=(10, 6))
sns.histplot(fraud_data['LogAmount'], bins=50, kde=True, color='red')
plt.title('Distribution of LogAmount Feature for Fraud (Fraudulent Transactions)')
plt.xlabel('LogAmount')
plt.ylabel('Frequency')
plt.show()
```





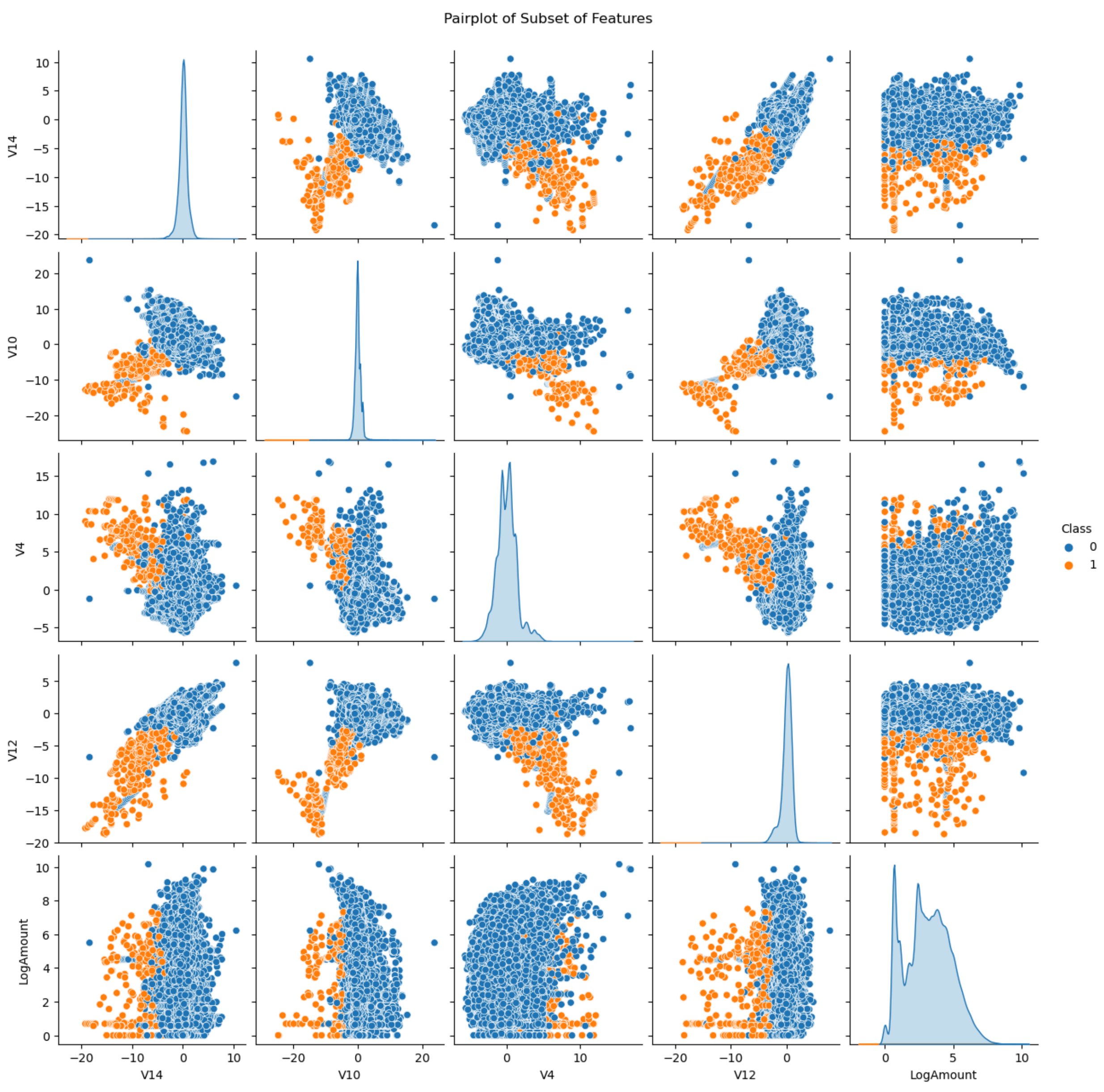


The first graphic illustrates the distribution of the Amount feature for all transactions, which is highly right-skewed with most transactions having lower amounts. This skewness can affect the performance of machine learning models by emphasizing extreme values. To address this issue, a log transformation was applied to the Amount feature, resulting in a more normalized distribution as shown in the second graphic. The normalized LogAmount distribution reduces the impact of extreme values and enhances model performance.

The third graphic presents the distribution of the LogAmount feature specifically for fraudulent transactions. It shows that fraudulent transactions are more frequent at lower transaction amounts, with a significant peak around the log-transformed value of 0. Additionally, there are other smaller peaks across the range, indicating that fraud can occur at various transaction amounts, though it is more common at smaller values. These visualizations highlight the necessity of the log transformation and provide valuable insights into the patterns of fraudulent transactions, which are critical for developing an effective fraud detection model.

Pairplot

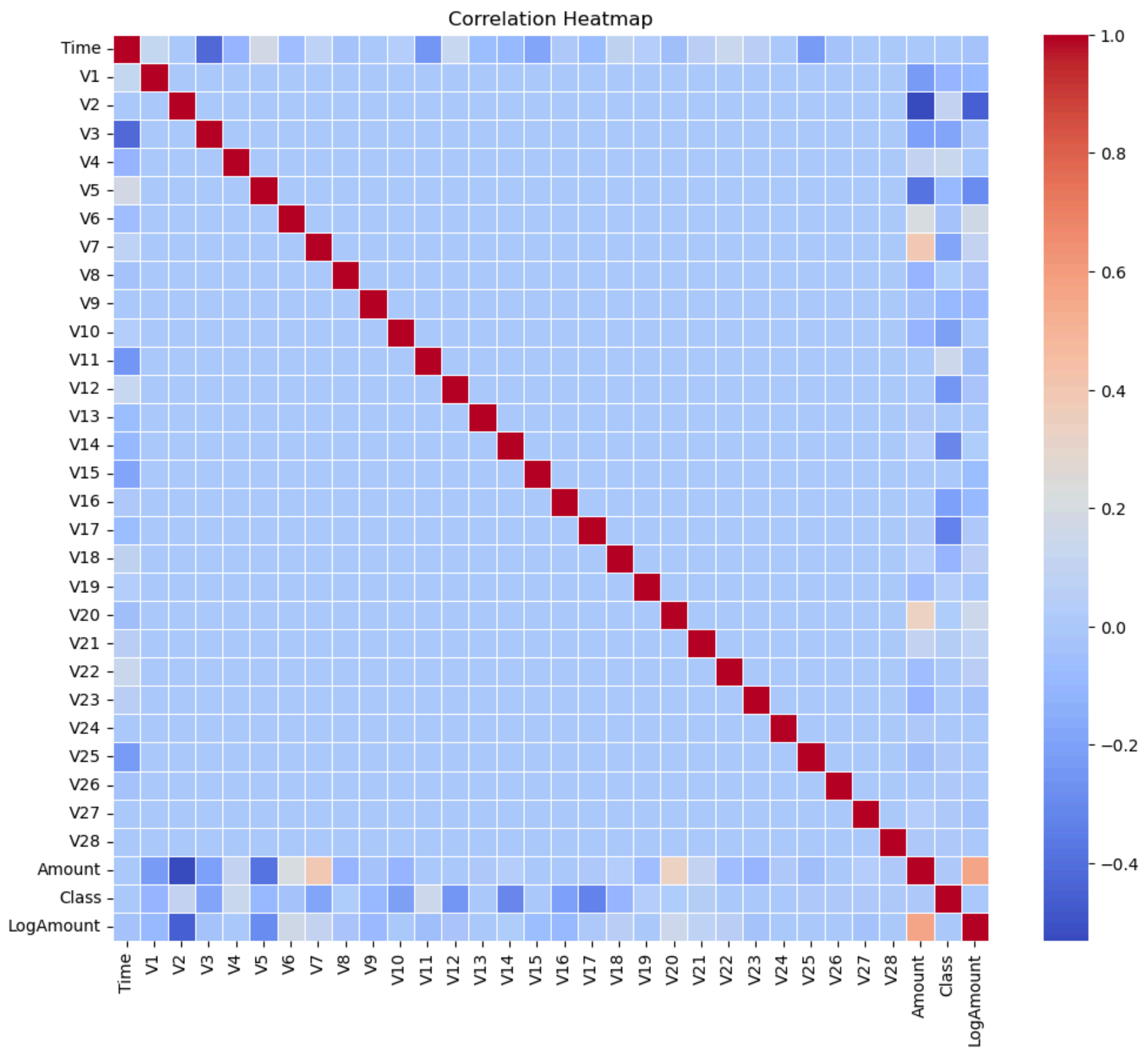
In [ ]:
```
# Pairplot of a subset of features
subset_features = ['V14', 'V10', 'V4', 'V12', 'LogAmount']
sns.pairplot(data[subset_features + ['Class']], hue='Class', diag_kind='kde')
plt.suptitle('Pairplot of Subset of Features', y=1.02)
plt.show()
```



This pairplot visualizes the relationships between a subset of features (V14, V10, V4, V12, and LogAmount) and distinguishes between non-fraudulent (blue) and fraudulent (orange) transactions. The scatter plots reveal distinct clusters of fraudulent transactions, suggesting that specific combinations of these features are indicative of fraud. For example, in the V14 vs. V10 and V4 vs. V10 plots, fraudulent transactions cluster separately from non-fraudulent ones, highlighting potential patterns that can be leveraged for fraud detection.

Correlations

In [ ]:
```
# Correlation heatmap
plt.figure(figsize=(12, 10))
correlation_matrix = data.corr()
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()
```



The correlation heatmap above provides a visual representation of the relationships between different features in the dataset. The color intensity indicates the strength of the correlation, with red representing strong positive correlations and blue representing strong negative correlations. Most features show little to no correlation with each other, as indicated by the light blue color across most of the heatmap. This lack of correlation is expected because many features have been transformed using Principal Component Analysis (PCA), which aims to create uncorrelated features to capture the underlying variance in the data. Notably, the Amount feature shows a stronger correlation with LogAmount, confirming that the log transformation effectively normalizes the data while preserving the relationship with the original Amount values.

The Class feature, representing whether a transaction is fraudulent, does not exhibit strong correlations with most features, underscoring the challenge of distinguishing fraudulent transactions based solely on individual features. This emphasizes the importance of using a combination of features and advanced techniques like machine learning models to accurately identify fraudulent activities.

Preprocessing

Training and Testing Split

In [ ]:
```
# Define features and target
X = data.drop(columns=['Class', 'Amount'])
y = data['Class']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

Rescaling

In [ ]:
```
# Define the scaler
scaler = RobustScaler()
```

```
# Fit the scaler on the training data and transform the training data
X_train_scaled = scaler.fit_transform(X_train)

# Transform the testing data using the scaler fitted on the training data
X_test_scaled = scaler.transform(X_test)
```

SMOTE

In [ ]:
```
# Apply SMOTE to the training data
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train_scaled, y_train)
```

Severe class imbalance can negatively affect the training and evaluation of our models in several ways. The model may become biased towards the majority class and perform poorly in predicting the minority class. Accuracy metrics can be misleading; for instance, a model that simply predicts the majority class 100% of the time would still achieve over 99.8% accuracy due to the imbalance. Additionally, the model may fail to properly learn the characteristics that define the minority class, resulting in poor detection of fraudulent transactions.

These issues can be addressed by using SMOTE (Synthetic Minority Over-sampling Technique). SMOTE balances the dataset by generating synthetic samples for the minority class by interpolating between existing minority class samples. This process enhances the model's ability to learn the characteristics of both classes, leading to improved detection performance and overall model robustness.

Baseline Models

Logistic Regression, Random Forest, Gradient Boosting, and XGBoost were chosen as baseline models for their proven effectiveness in classification tasks. Logistic Regression is a straightforward linear model that provides a strong baseline. Random Forest, an ensemble method, enhances accuracy and robustness by combining multiple decision trees. Gradient Boosting builds trees sequentially to correct errors from previous trees, improving performance. Lastly, XGBoost is a highly efficient implementation of gradient boosting renowned for its speed and accuracy.

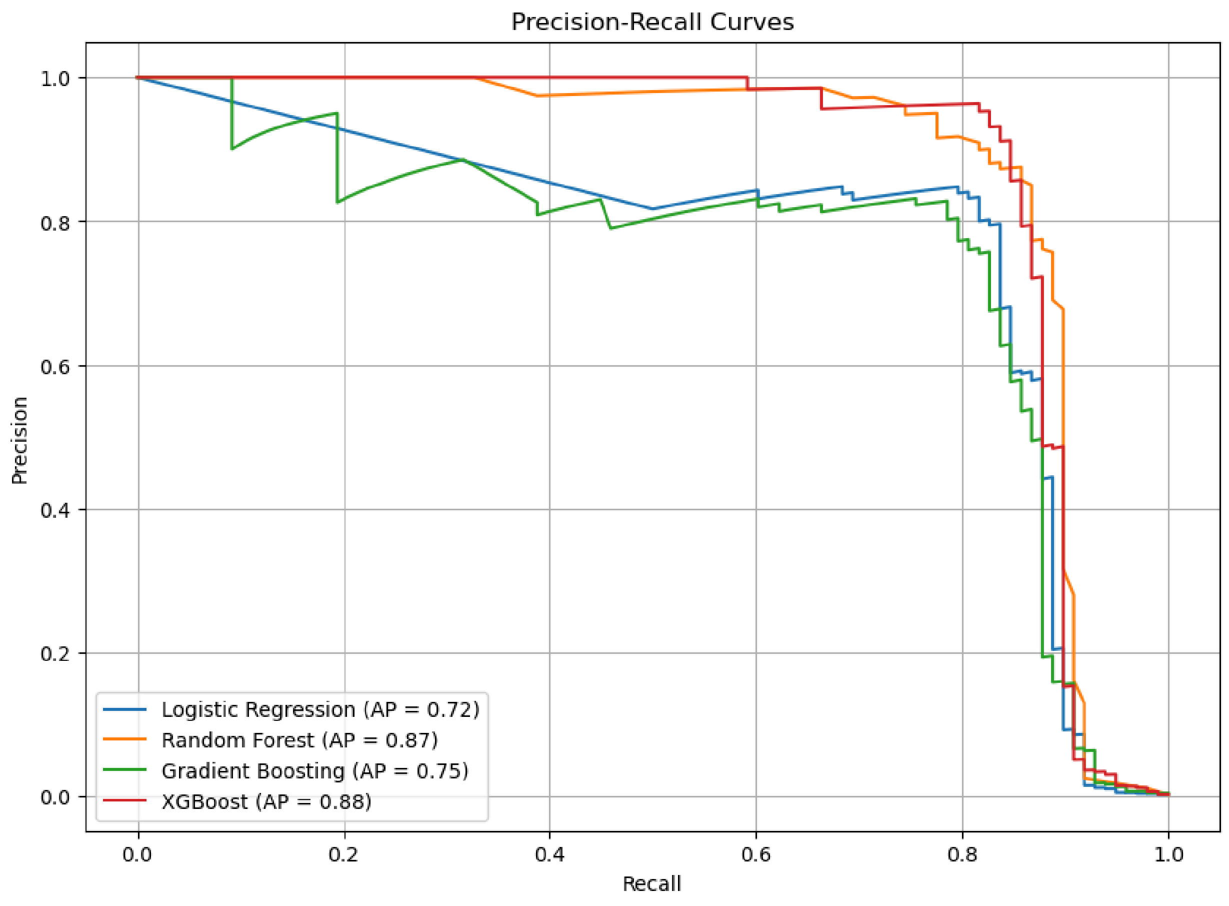
In [ ]:
```
# Function to compute metrics and return results
def compute_metrics(model, X_test, y_test):
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)[:, 1]  # Probabilities for the positive class
    precision, recall, _ = precision_recall_curve(y_test, y_prob)
    fpr, tpr, _ = roc_curve(y_test, y_prob)
    avg_precision = average_precision_score(y_test, y_prob)
    roc_auc = roc_auc_score(y_test, y_prob)
    report = classification_report(y_test, y_pred, output_dict=True)
    return {
        'precision': precision,
        'recall': recall,
        'fpr': fpr,
        'tpr': tpr,
        'avg_precision': avg_precision,
        'roc_auc': roc_auc,
        'classification_report': report
    }
```

In [ ]:
```
# Initialize models
log_reg = LogisticRegression(random_state=42, solver='liblinear')
random_forest = RandomForestClassifier(random_state=42)
gradient_boosting = GradientBoostingClassifier(random_state=42)
xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42)

# List of models
models = [
    ("Logistic Regression", log_reg),
    ("Random Forest", random_forest),
    ("Gradient Boosting", gradient_boosting),
    ("XGBoost", xgb)
]
```

In [ ]:
```
# Dictionary to store model performance metrics
baseline_model_metrics = {}

# Train and evaluate models with best hyperparameters
for name, model in models:
    start_time = time.time()
    model.fit(X_train_smote, y_train_smote)
    training_time = time.time() - start_time
    metrics = compute_metrics(model, X_test_scaled, y_test)

    baseline_model_metrics[name] = {
        'training_time': training_time,
        'roc_auc': metrics['roc_auc'],
        'average_precision': metrics['avg_precision'],
        'classification_report': metrics['classification_report'],
        'precision_curve': metrics['precision'],
        'recall_curve': metrics['recall'],
        'fpr_curve': metrics['fpr'],
        'tpr_curve': metrics['tpr']
    }
```

Precision-Recall Curves

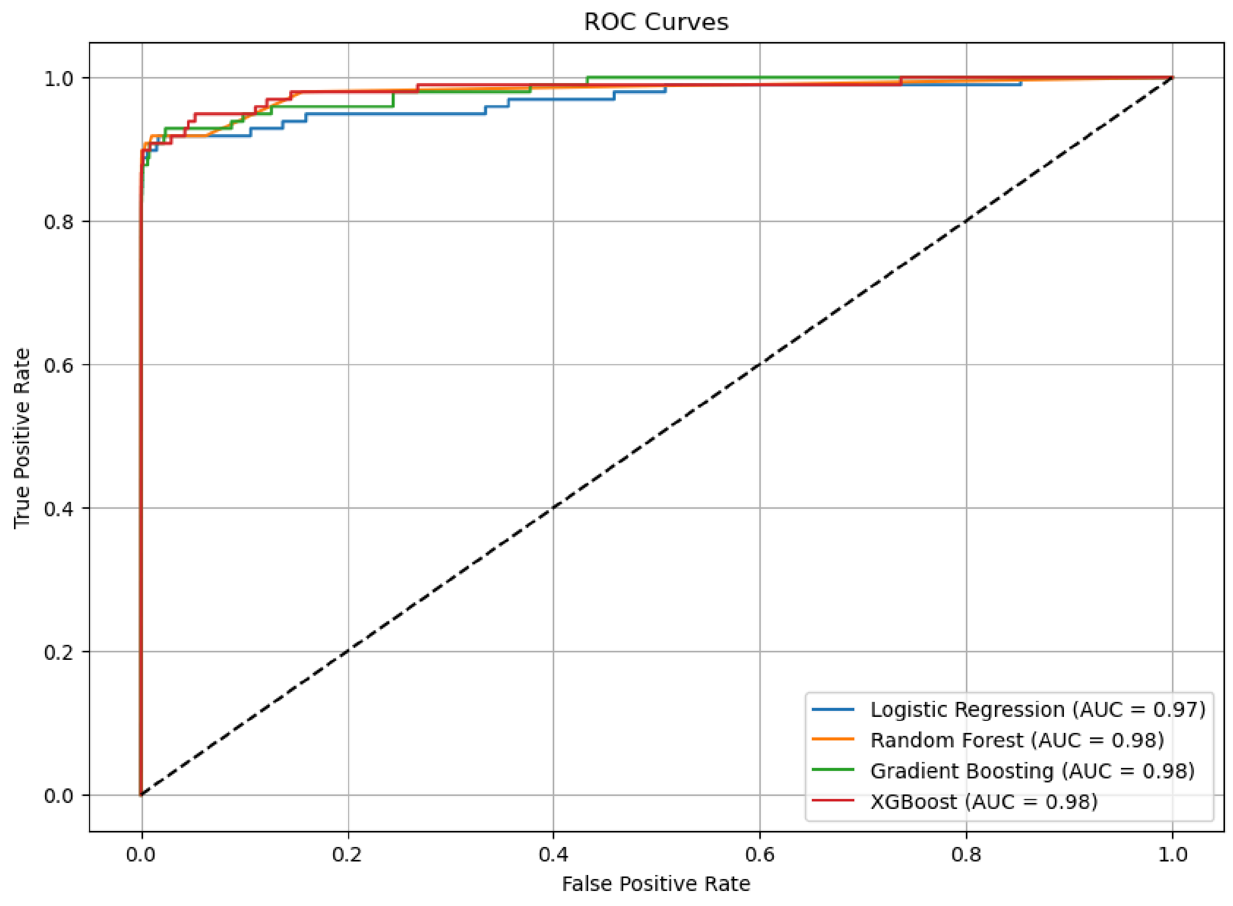
In [ ]:
```
# Plot Precision-Recall Curves
plt.figure(figsize=(10, 7))
for name, metrics in baseline_model_metrics.items():
    plt.plot(metrics['recall_curve'], metrics['precision_curve'], label=f'{name} (AP =
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend()
plt.grid(True)
plt.show()
```



The Precision-Recall curves show that XGBoost and Random Forest significantly outperform the other models, with XGBoost achieving the highest average precision (AP = 0.88) and Random Forest following closely (AP = 0.87). These models demonstrate the best balance between precision and recall, making them good contenders for the final model.

ROC Curves

```
In [ ]: # Plot ROC Curves
        plt.figure(figsize=(10, 7))
        for name, metrics in baseline_model_metrics.items():
            plt.plot(metrics['fpr_curve'], metrics['tpr_curve'], label=f'{name} (AUC = {metric
        plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line for random classifier
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curves')
        plt.legend()
        plt.grid(True)
        plt.show()
```



The ROC curves show that XGBoost, Random Forest, and Gradient Boosting all achieve high performance with an AUC of 0.98, indicating excellent ability to distinguish between fraudulent and non-fraudulent transactions. Logistic Regression also performs well with an AUC of 0.97.

Overall Metrics

```
In [ ]: # Display the performance metrics in a DataFrame
        metrics_df = pd.DataFrame.from_dict(baseline_model_metrics, orient='index')
        metrics_df['precision'] = metrics_df['classification_report'].apply(lambda x: x['1']['
        metrics_df['recall'] = metrics_df['classification_report'].apply(lambda x: x['1']['rec
        metrics_df['f1-score'] = metrics_df['classification_report'].apply(lambda x: x['1']['f
        metrics_df = metrics_df.drop(columns=['classification_report', 'precision_curve', 'rec
        metrics_df
```

Out[ ]:

	training_time	roc_auc	average_precision	precision	recall	f1-score
Logistic Regression	3.492330	0.969693	0.723478	0.058670	0.918367	0.110294
Random Forest	225.781602	0.981322	0.874102	0.872340	0.836735	0.854167
Gradient Boosting	414.145973	0.982658	0.746354	0.102299	0.908163	0.183884
XGBoost	1.994773	0.983994	0.875445	0.688000	0.877551	0.771300

Given these pre-tuning models, Random Forest and XGBoost appear to be the best options. These models have high ROC AUC scores and a good balance between precision and recall as illustrated by their f1-scores. The Logistic Regression and Gradient Boosting models appear to struggle with precision, meaning that these models will produce many false positives compared to Random Forest and XGBoost making them inferior. Training time will also be key consideration moving forward. Models with relatively longer training times may be less practical to tune and adapt for practical use cases.

Hyperparameter Tuning

In [ ]:
```
# Hyperparameter grids have been limited to avoid overly long processing times
# Define hyperparameter grid for Logistic Regression
log_reg_params = {
    'C': [0.1, 1, 10],
    'solver': ['liblinear']
}

# Define hyperparameter grid for Random Forest
random_forest_params = {
    'n_estimators': [100, 200],
    'max_depth': [10, 20],
    'min_samples_split': [2, 5]
}

# Define hyperparameter grid for Gradient Boosting
gradient_boosting_params = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1],
    'max_depth': [3, 4]
}

# Define hyperparameter grid for XGBoost
xgb_params = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1],
    'max_depth': [3, 4]
}
```

In [ ]:
```
# Logistic Regression Grid Search
log_reg_grid = GridSearchCV(LogisticRegression(random_state=42), log_reg_params, cv=3,
log_reg_grid.fit(X_train_smote, y_train_smote)

# Random Forest Grid Search
random_forest_grid = GridSearchCV(RandomForestClassifier(random_state=42), random_fore
random_forest_grid.fit(X_train_smote, y_train_smote)

# Gradient Boosting Grid Search
gradient_boosting_grid = GridSearchCV(GradientBoostingClassifier(random_state=42), gra
gradient_boosting_grid.fit(X_train_smote, y_train_smote)

# XGBoost Grid Search
xgb_grid = GridSearchCV(XGBClassifier(random_state=42, use_label_encoder=False, eval_m
xgb_grid.fit(X_train_smote, y_train_smote)

# Best parameters
best_params_log_reg = log_reg_grid.best_params_
best_params_random_forest = random_forest_grid.best_params_
best_params_gradient_boosting = gradient_boosting_grid.best_params_
best_params_xgb = xgb_grid.best_params_

best_params_log_reg, best_params_random_forest, best_params_gradient_boosting, best_pa
```

Out[ ]:
```
({'C': 10, 'solver': 'liblinear'},
 {'max_depth': 20, 'min_samples_split': 5, 'n_estimators': 200},
 {'learning_rate': 0.1, 'max_depth': 4, 'n_estimators': 200},
 {'learning_rate': 0.1, 'max_depth': 4, 'n_estimators': 200})
```

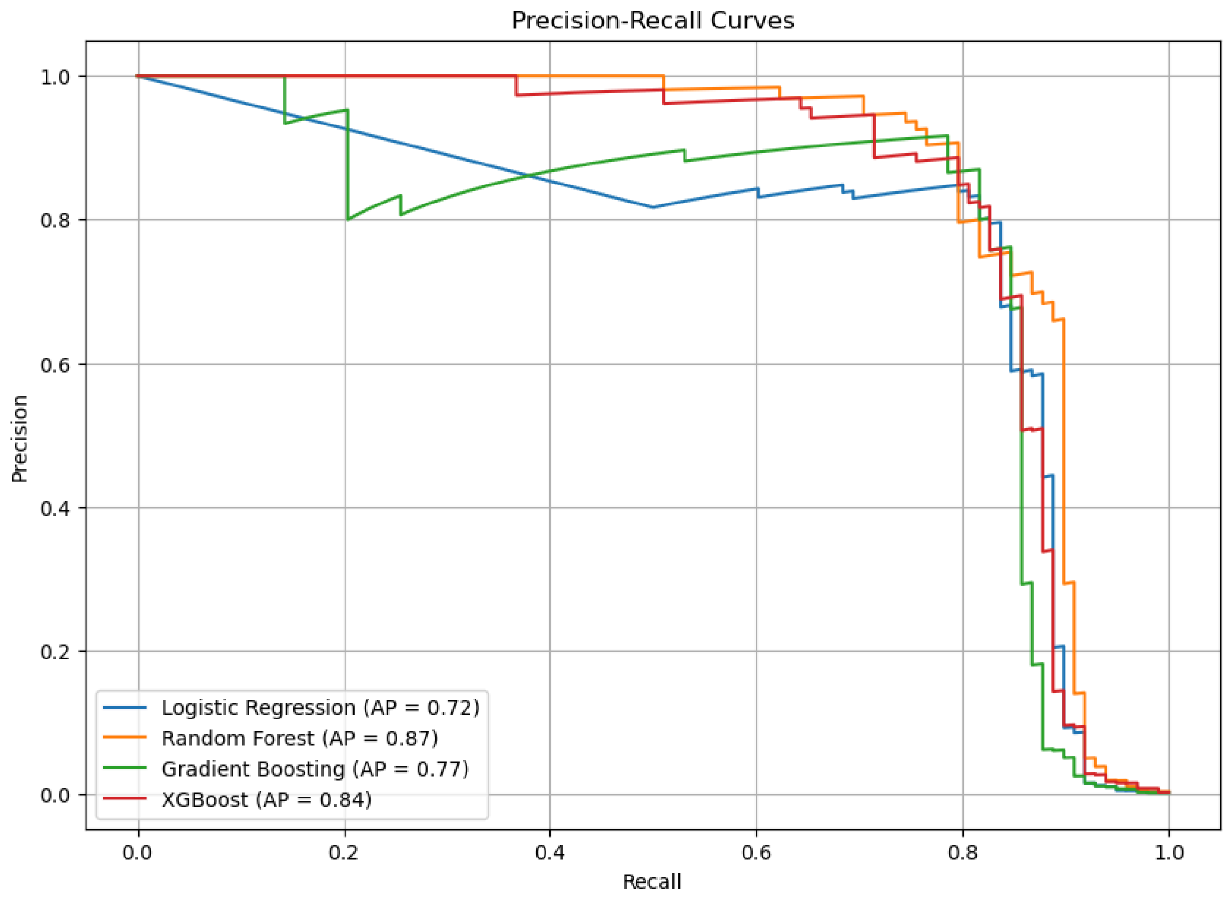
In [ ]:
```
# initiate models with best hyperparameters
log_reg_best = LogisticRegression(random_state=42, **best_params_log_reg)
random_forest_best = RandomForestClassifier(random_state=42, **best_params_random_fore
gradient_boosting_best = GradientBoostingClassifier(random_state=41, **best_params_gra
xgb_best = XGBClassifier(random_state=41, use_label_encoder=False, eval_metric='loglos

# List of models with best hyperparameters
models_best = [
    ("Logistic Regression", log_reg_best),
    ("Random Forest", random_forest_best),
    ("Gradient Boosting", gradient_boosting_best),
    ("XGBoost", xgb_best)
]
```

In [ ]:
```
# Dictionary to store model performance metrics
model_metrics = {}
```

```
# Train and evaluate models with best hyperparameters
for name, model in models_best:
    start_time = time.time()
    model.fit(X_train_smote, y_train_smote)
    training_time = time.time() - start_time
    metrics = compute_metrics(model, X_test_scaled, y_test)

    model_metrics[name] = {
        'training_time': training_time,
        'roc_auc': metrics['roc_auc'],
        'average_precision': metrics['avg_precision'],
        'classification_report': metrics['classification_report'],
        'precision_curve': metrics['precision'],
        'recall_curve': metrics['recall'],
        'fpr_curve': metrics['fpr'],
        'tpr_curve': metrics['tpr']
    }
```

Precision Recall Curves

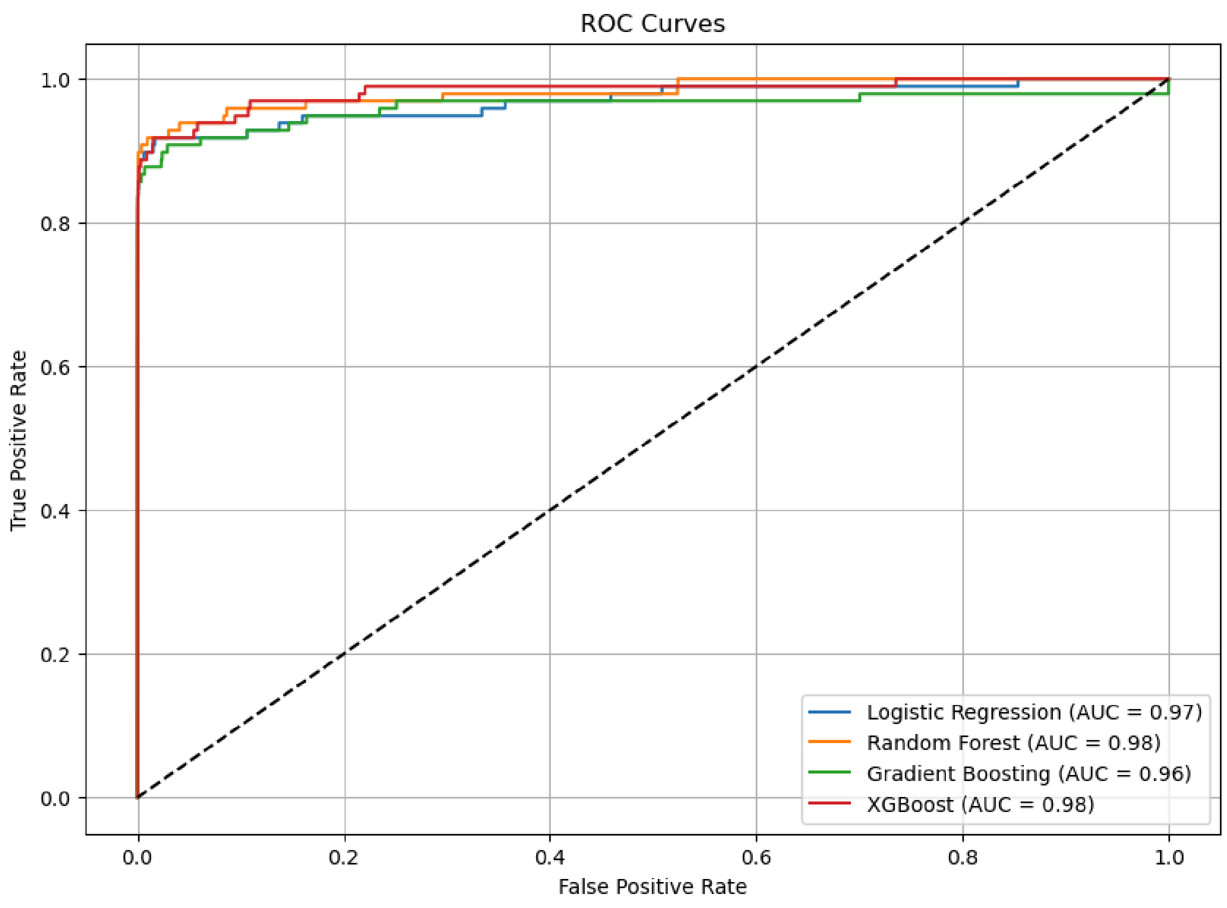
In [ ]:
```
# Plot Precision-Recall Curves
plt.figure(figsize=(10, 7))
for name, metrics in model_metrics.items():
    plt.plot(metrics['recall_curve'], metrics['precision_curve'], label=f'{name} (AP =
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend()
plt.grid(True)
plt.show()
```



The Precision-Recall curves for the tuned models show that Random Forest achieves the highest average precision (AP = 0.87), followed by XGBoost (AP = 0.84). These results indicate that Random Forest and XGBoost provide the best balance between precision and recall after hyperparameter tuning.

ROC Curves

In [ ]:
```
# Plot ROC Curves
plt.figure(figsize=(10, 7))
for name, metrics in model_metrics.items():
    plt.plot(metrics['fpr_curve'], metrics['tpr_curve'], label=f'{name} (AUC = {metric
plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line for random classifier
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves')
plt.legend()
plt.grid(True)
plt.show()
```



The ROC curves for the tuned models indicate that Random Forest and XGBoost achieve the highest performance with an AUC of 0.98.

Overall Metrics

In [ ]:
```
# Display the performance metrics in a DataFrame
metrics_df = pd.DataFrame.from_dict(model_metrics, orient='index')
metrics_df['precision'] = metrics_df['classification_report'].apply(lambda x: x['1']['
metrics_df['recall'] = metrics_df['classification_report'].apply(lambda x: x['1']['rec
metrics_df['f1-score'] = metrics_df['classification_report'].apply(lambda x: x['1']['f
metrics_df = metrics_df.drop(columns=['classification_report', 'precision_curve', 'rec
metrics_df
```

Out[ ]:

	training_time	roc_auc	average_precision	precision	recall	f1-score
Logistic Regression	3.497718	0.969683	0.723520	0.058632	0.918367	0.110227
Random Forest	442.115498	0.981962	0.866052	0.741071	0.846939	0.790476
Gradient Boosting	1108.923300	0.961636	0.774926	0.324324	0.857143	0.470588
XGBoost	2.980357	0.983209	0.840912	0.277955	0.887755	0.423358

The performance of each model may be summarized as follows:

1. **Logistic Regression:** Good recall, but low precision and F1-score. This indicates that this model will be able to detect many fraudulent transactions, but at the cost of having a very high false positive rate relative to our other models.
2. **Random Forest:** Performs well across all metrics, and has a good balance between precision and recall. Has a relatively long training time when compared to faster models like XGBoost.
3. **Gradient Boosting:** Good recall, but relatively low precision and f1-score. This is indicative of similar issues to Logistic Regression, but with better overall performance.
4. **XGBoost:** Performs well, but has lower precision and f1-score relative to Random Forest. XGBoost, however, has the shortest training time of all the models potentially making it more useful for practical applications.

Final Model

Random Forest was chosen as the final model due to its strong performance in identifying fraud. It achieved an excellent AUC of 0.98, showing it can effectively distinguish between fraudulent and legitimate transactions. Additionally, it had the best F1 score, balancing precision and recall, which means it not only catches most fraud cases but also avoids mislabeling too many legitimate transactions as fraud.

In [ ]:

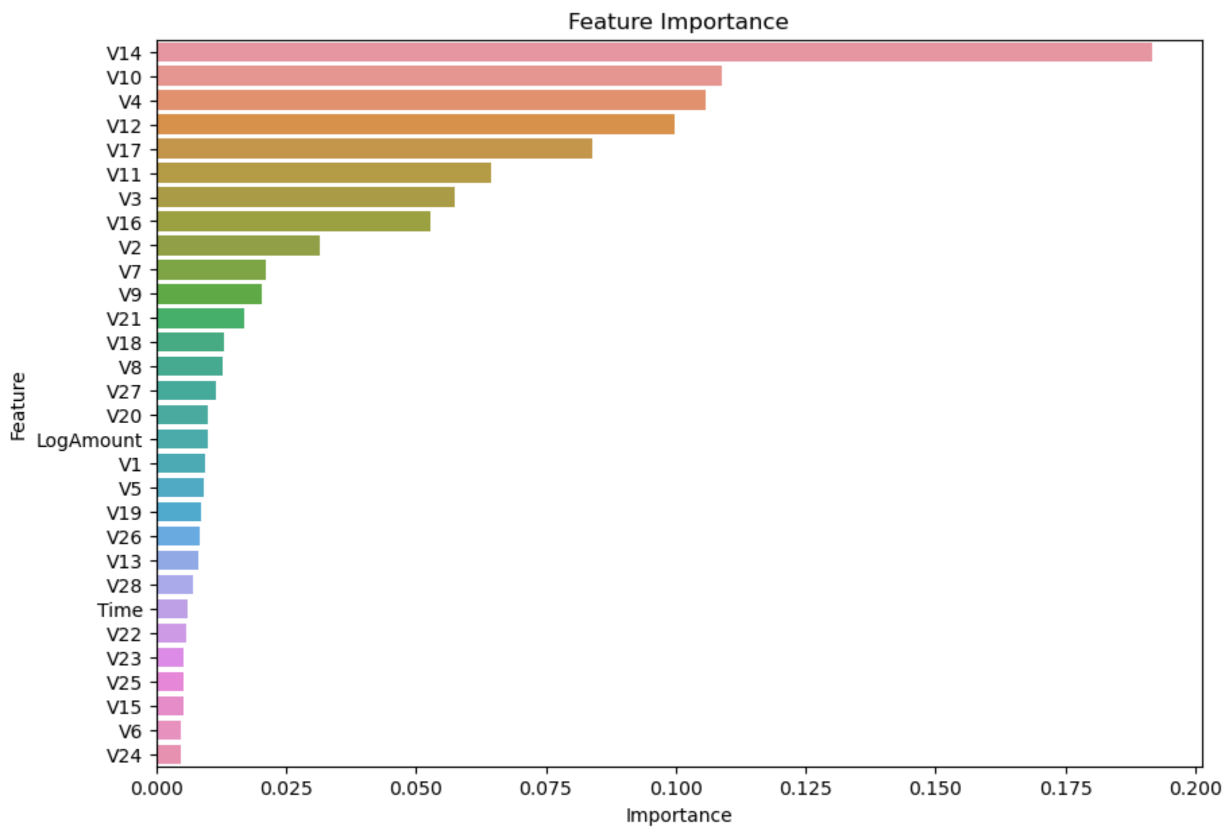
```
# Train Random Forest model with best hyperparameters
random_forest_best = RandomForestClassifier(random_state=42, **best_params_random_fore
start_time = time.time()
random_forest_best.fit(X_train_smote, y_train_smote)
training_time = time.time() - start_time

# Evaluate the model
y_pred = random_forest_best.predict(X_test_scaled)
y_prob = random_forest_best.predict_proba(X_test_scaled)[:, 1]

# Compute metrics
roc_auc = roc_auc_score(y_test, y_prob)
avg_precision = average_precision_score(y_test, y_prob)
classification_report_dict = classification_report(y_test, y_pred, output_dict=True)

# Print summary statistics
print(f"Random Forest Training Time: {training_time:.2f} seconds")
print(f"ROC AUC Score: {roc_auc:.4f}")
print(f"Average Precision Score: {avg_precision:.4f}")
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

```
Random Forest Training Time: 400.60 seconds
ROC AUC Score: 0.9820
Average Precision Score: 0.8661
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.74      0.85      0.79        98

    accuracy                           1.00     56962
   macro avg       0.87      0.92      0.90     56962
weighted avg       1.00      1.00      1.00     56962
```

Feature Importance

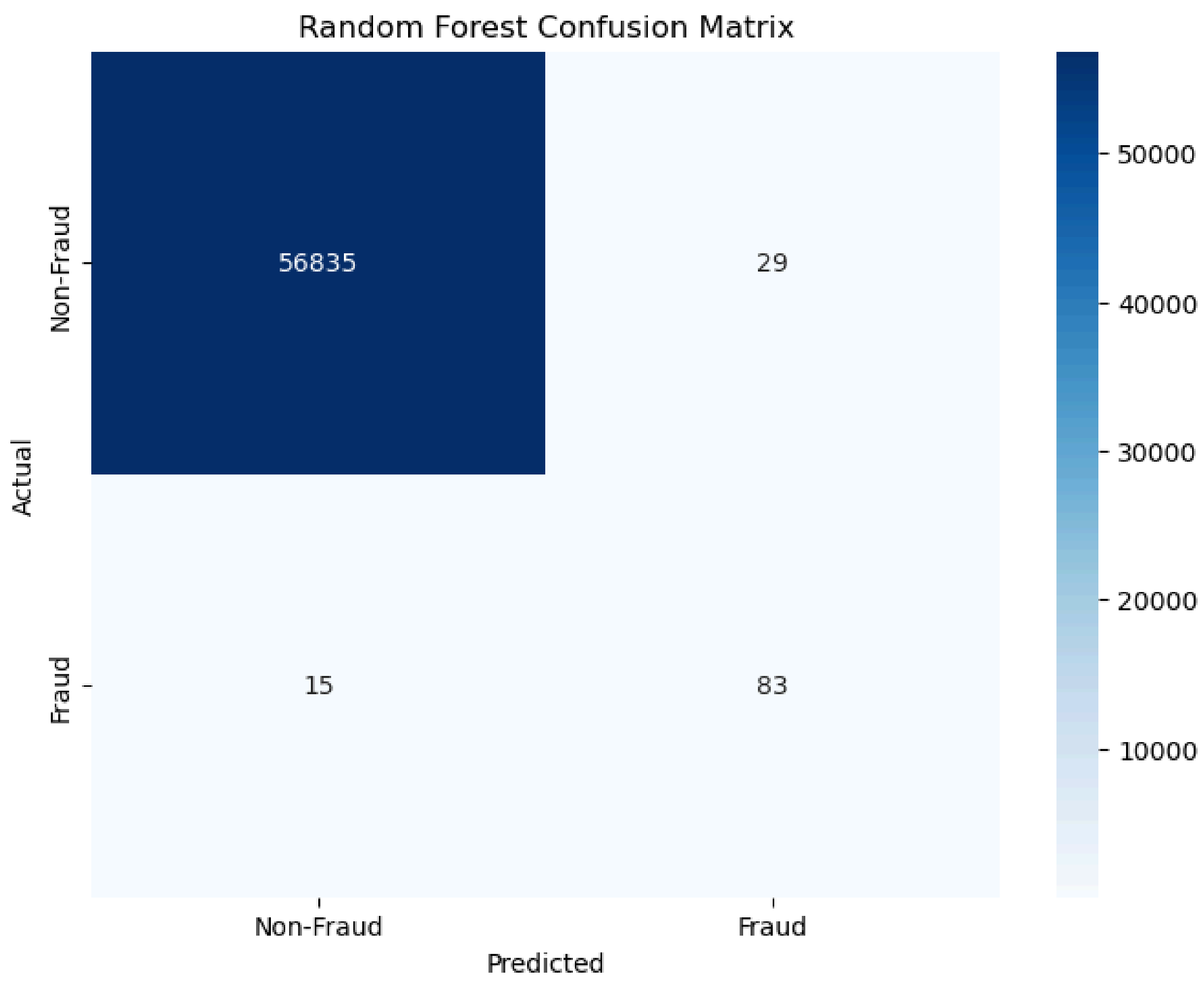
In [ ]:

```
# Feature Importance
feature_importances = random_forest_best.feature_importances_
feature_names = X.columns
feature_importance_df = pd.DataFrame({'Feature': feature_names, 'Importance': feature_
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=F

# Plot Feature Importance
plt.figure(figsize=(10, 7))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df)
plt.title('Feature Importance')
plt.show()
```



Confusion Matrix

```
In [ ]: # Confusion Matrix
        conf_matrix = confusion_matrix(y_test, y_pred)

        # Plot Confusion Matrix
        plt.figure(figsize=(8, 6))
        sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Non-Fraud',
        plt.xlabel('Predicted')
        plt.ylabel('Actual')
        plt.title('Random Forest Confusion Matrix')
        plt.show()
```



This confusion matrix helps us understand how well our fraud detection model is performing by breaking down how many transactions it correctly and incorrectly identifies as fraudulent or non-fraudulent. With over 56,800 transactions correctly classified as non-fraudulent and only 29 false positives, our model is quite good at identifying valid transactions. This is imporantant to ensure that customers are not troubled with declined transactions or false fraud alerts. Perhaps more importantly, the model was able to corretly identify 83 fraudulent transactions while only missing 15. This suggests that we can reliably detect fraudulent transactions to enhance user security and experience.

Conclusion

The Random Forest model is highly accurate and reliable for detecting fraudulent transactions. This ensures that genuine transactions are processed smoothly while effetively catching most fraud cases. Customers can trust that their legitimate transactions will go through without unnecessary interruptions, while the system actively works to prevent fraud. While the model performs well overall, continuous monitoring and improvement are essential to adapt to new fraud patterns and further enhance security and user experience.

Further tuning and the encorporation of additional features could improve model performance even more. Exploring ensemble methods that combine multiple models could yield even better results. Implementing real-time fraud detection systems will help in proactively stopping fraudulent transactions as they occur.

This project demonstrates a robust approach to building and evaluating a fraud detection mode, providing a solid foundation for deploying an effective fraud detection system in a real-world financial environment.