MODEL COMPARISONS

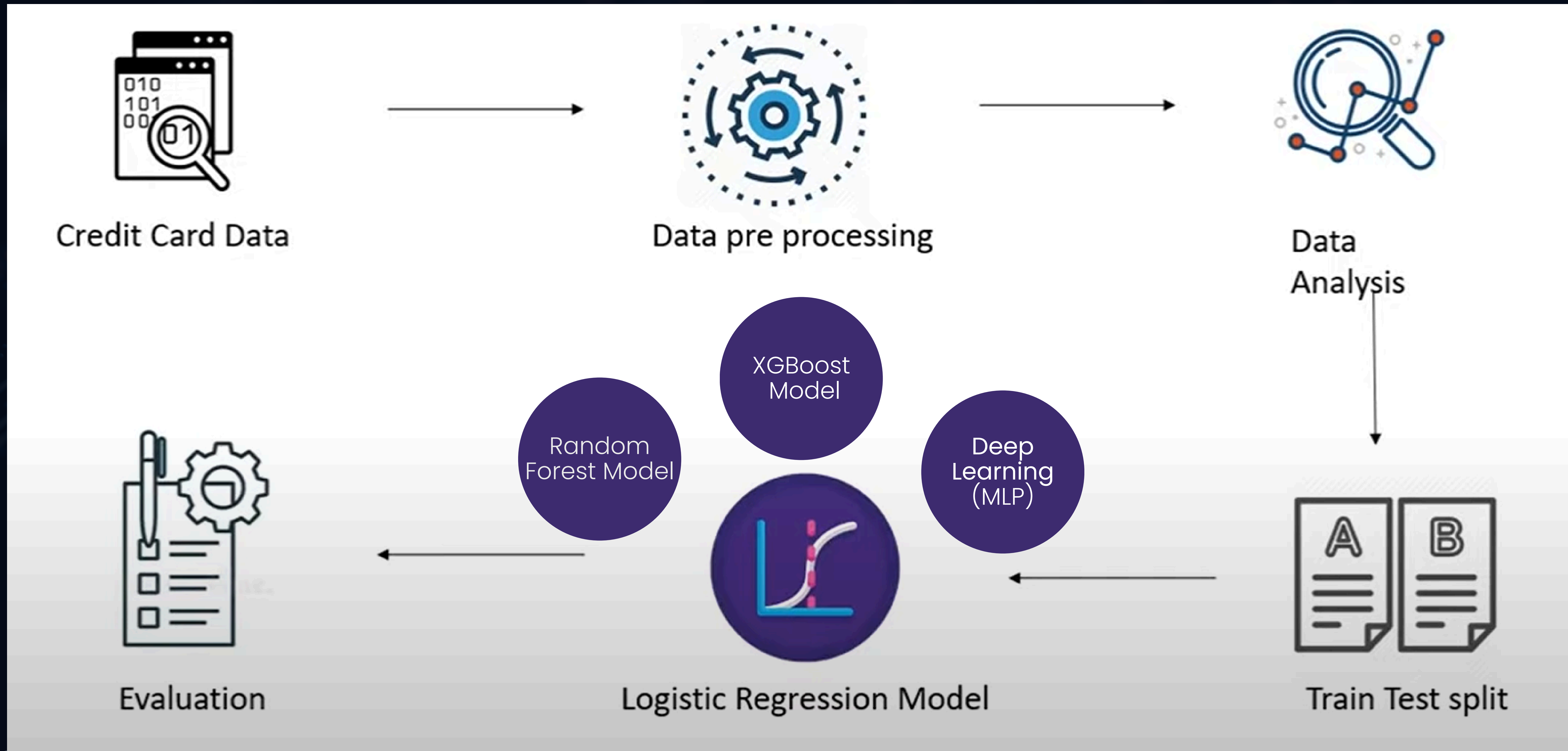
Model	Accuracy	Precision	Recall	F1 Score	AUPRC (Positive Fraud)	ROC AUC
Logistic Regression	**97.9%**	98.8%	**96.6%**	**97.7%**	0.992	0.989
XGBoost	97.4%	98.8%	95.5%	97.1%	**0.998**	**0.998**
Random Forest	97.4%	**100.0%**	94.3%	97.1%	0.969	0.997
Deep Learning	97.4%	98.8%	95.5%	97.1%	0.994	0.994

All of the models have high accuracy.
XG Boost has the best AUPRC & ROCAUC
Random Forest has 0 False Positives

CREDIT CARD FRAUD DETECTION

MACHINE LEARNING CLASSIFICATION PROBLEM

PROJECT OVERVIEW



LIBRARY AND MODULES

- pandas & numpy for data handling and calculations
- matplotlib & seaborn for visualizations
- StandardScaler to normalize Amount and Time
- train_test_split to divide data for training/testing
- RandomForestClassifier for model training
- sklearn.metrics for model evaluation (accuracy, precision, ROC, etc.)
- warnings to hide warnings in output

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    classification_report, confusion_matrix,
    precision_recall_curve, average_precision_score, roc_curve, auc
)
import warnings
warnings.filterwarnings("ignore")
```

3.0s

Loading The Dataset

```
df = pd.read_csv("creditcard.csv")
```

[59]

```
df.head()
```

[60]

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	149.62	0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	123.50	0
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	0

5 rows × 31 columns

Data Pre-processing

Checking for Null Values

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

[61] Python

```
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
...
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Data Pre-processing

Checking for Duplicated Values

```
[62]
df.duplicated().sum()
```

```
1081
```

```
[63]
df.shape
```

```
(284807, 31)
```

```
[64]
df = df.drop_duplicates()

df.shape
```

```
(283726, 31)
```

Data Analysis

Info

```
#Info & Statistics
df.info()
```

[65]

```
<class 'pandas.core.frame.DataFrame'>
Index: 283726 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   Time    283726 non-null  float64
 1   V1      283726 non-null  float64
 2   V2      283726 non-null  float64
 3   V3      283726 non-null  float64
 4   V4      283726 non-null  float64
 5   V5      283726 non-null  float64
 6   V6      283726 non-null  float64
 7   V7      283726 non-null  float64
 8   V8      283726 non-null  float64
 9   V9      283726 non-null  float64
 10  V10     283726 non-null  float64
 11  V11     283726 non-null  float64
 12  V12     283726 non-null  float64
 13  V13     283726 non-null  float64
 14  V14     283726 non-null  float64
 15  V15     283726 non-null  float64
 16  V16     283726 non-null  float64
 17  V17     283726 non-null  float64
 18  V18     283726 non-null  float64
 19  V19     283726 non-null  float64
...
 29  Amount  283726 non-null  float64
 30  Class   283726 non-null  int64  
dtypes: float64(30), int64(1)
memory usage: 69.3 MB
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Data Analysis

Statistics

```
df.describe()
```

[10] ✓ 0.6s

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
count	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	...	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000	283726.000000
mean	94811.077600	0.005917	-0.004135	0.001613	-0.002966	0.001828	-0.001139	0.001801	-0.000854	-0.001596	...	-0.000371	-0.000015	0.000198	0.000214	-0.000232	0.000149	0.001763	0.000547	88.472687	0.001667
std	47481.047891	1.948026	1.646703	1.508682	1.414184	1.377008	1.331931	1.227664	1.179054	1.095492	...	0.723909	0.724550	0.623702	0.605627	0.521220	0.482053	0.395744	0.328027	250.399437	0.040796
min	0.000000	-56.407510	-72.715728	-48.325589	-5.683171	-113.743307	-26.160506	-43.557242	-73.216718	-13.434066	...	-34.830382	-10.933144	-44.807735	-2.836627	-10.295397	-2.604551	-22.565679	-15.430084	0.000000	0.000000
25%	54204.750000	-0.915951	-0.600321	-0.889682	-0.850134	-0.689830	-0.769031	-0.552509	-0.208828	-0.644221	...	-0.228305	-0.542700	-0.161703	-0.354453	-0.317485	-0.326763	-0.070641	-0.052818	5.600000	0.000000
50%	84692.500000	0.020384	0.063949	0.179963	-0.022248	-0.053468	-0.275168	0.040859	0.021898	-0.052596	...	-0.029441	0.006675	-0.011159	0.041016	0.016278	-0.052172	0.001479	0.011288	22.000000	0.000000
75%	139298.000000	1.316068	0.800283	1.026960	0.739647	0.612218	0.396792	0.570474	0.325704	0.595977	...	0.186194	0.528245	0.147748	0.439738	0.350667	0.240261	0.091208	0.078276	77.510000	0.000000
max	172792.000000	2.454930	22.057729	9.382558	16.875344	34.801666	73.301626	120.589494	20.007208	15.594995	...	27.202839	10.503090	22.528412	4.584549	7.519589	3.517346	31.612198	33.847808	25691.160000	1.000000

8 rows × 31 columns

- Time ranges from 0 to ~172,000 seconds (~48 hours).
- Mean is around 94,000 — the dataset has more higher values
- Amount has a wide range, from 0 to over 25,000, with a skewed mean of about 88.
- This suggests that just a few high-value transactions pull the mean right.
- Class is binary, with a mean close to 0, confirming imbalance.”

Data Analysis

Class Value Counts

```
[67]
df['Class'].value_counts()
```

```
Class
0    283253
1       473
Name: count, dtype: int64
```

Data Analysis

Visualisation

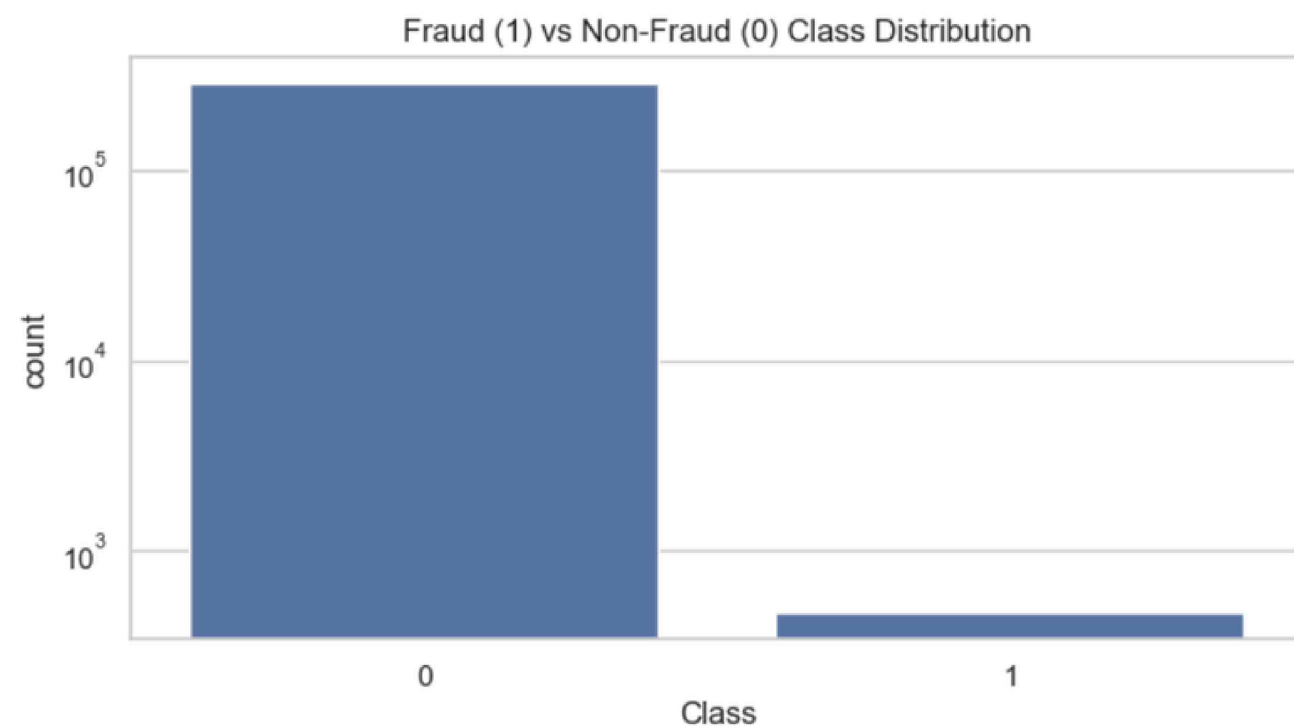
Visualization

```
sns.set(style="whitegrid")
```

[68]

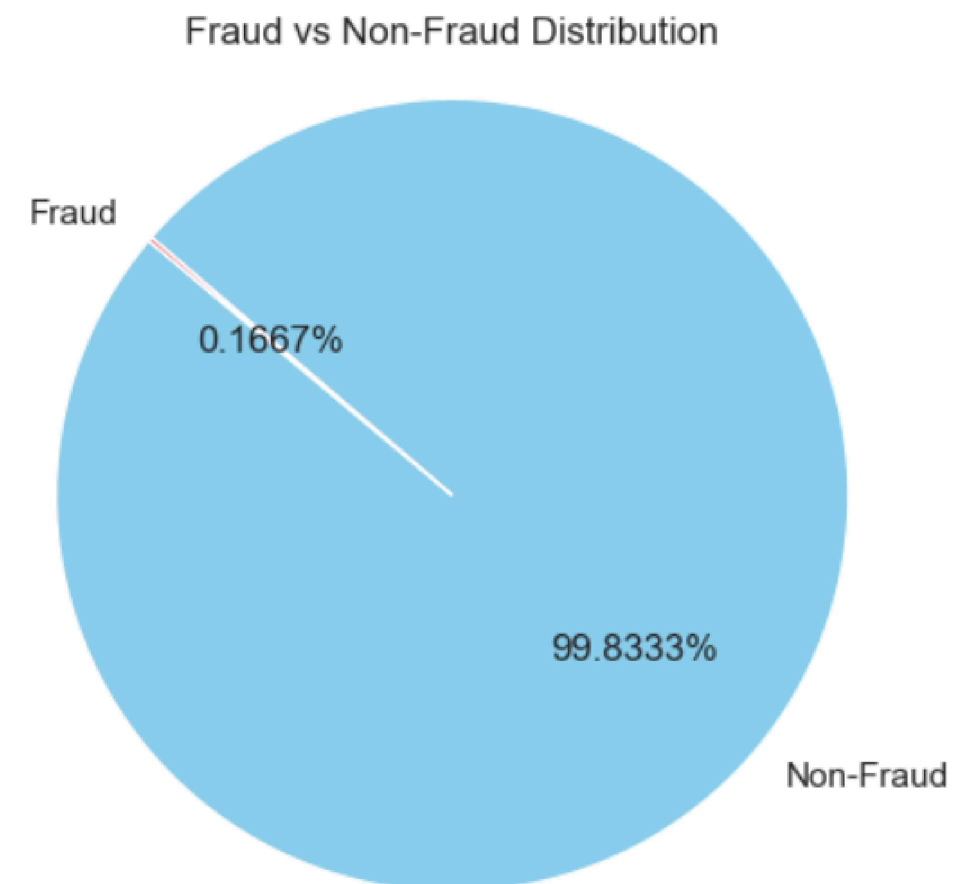
```
#Class distribution countplot
plt.figure(figsize=(8, 4))
sns.countplot(x='Class', data=df)
plt.title("Fraud (1) vs Non-Fraud (0) Class Distribution")
plt.yscale('log')
plt.show()
```

[69]



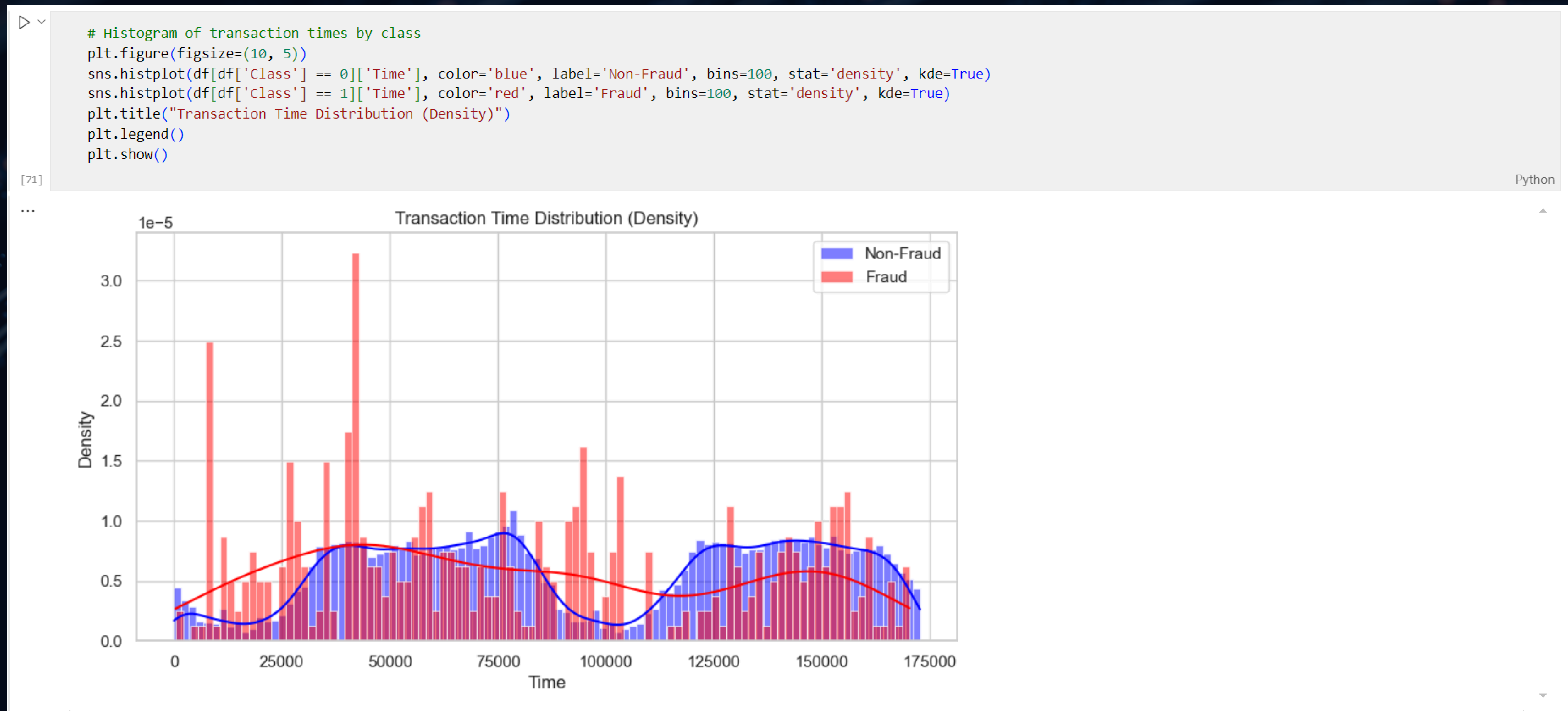

```
# Pie chart of class
plt.figure(figsize=(5, 5))
plt.pie(df['Class'].value_counts(), labels=['Non-Fraud', 'Fraud'],
        autopct='%1.4f%%', colors=['skyblue', 'red'], startangle=140)
plt.title("Fraud vs Non-Fraud Distribution")
plt.axis('equal')
plt.show()
```

[70]



Data Analysis

Transaction Time Distribution



Around 0–50,000 seconds, fraud activity is relatively high Fraud density drops and picks up again in certain mid-ranges.

Data Analysis

Histplot Fraud Transactions By Hour

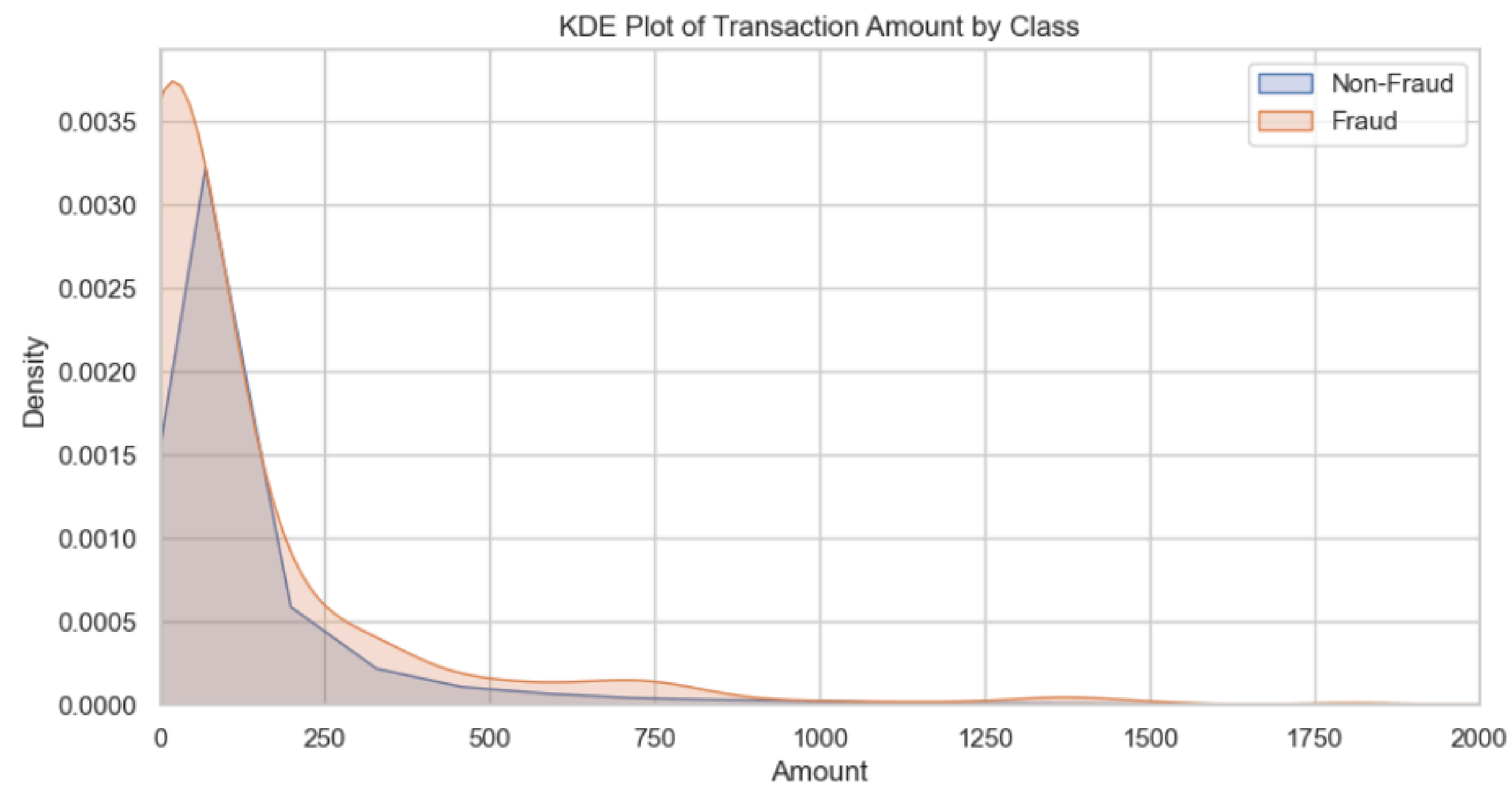
During the day, there are fraud peaks around hour 10–12

Data Analysis

KDE Plot of Transaction Amount by Class

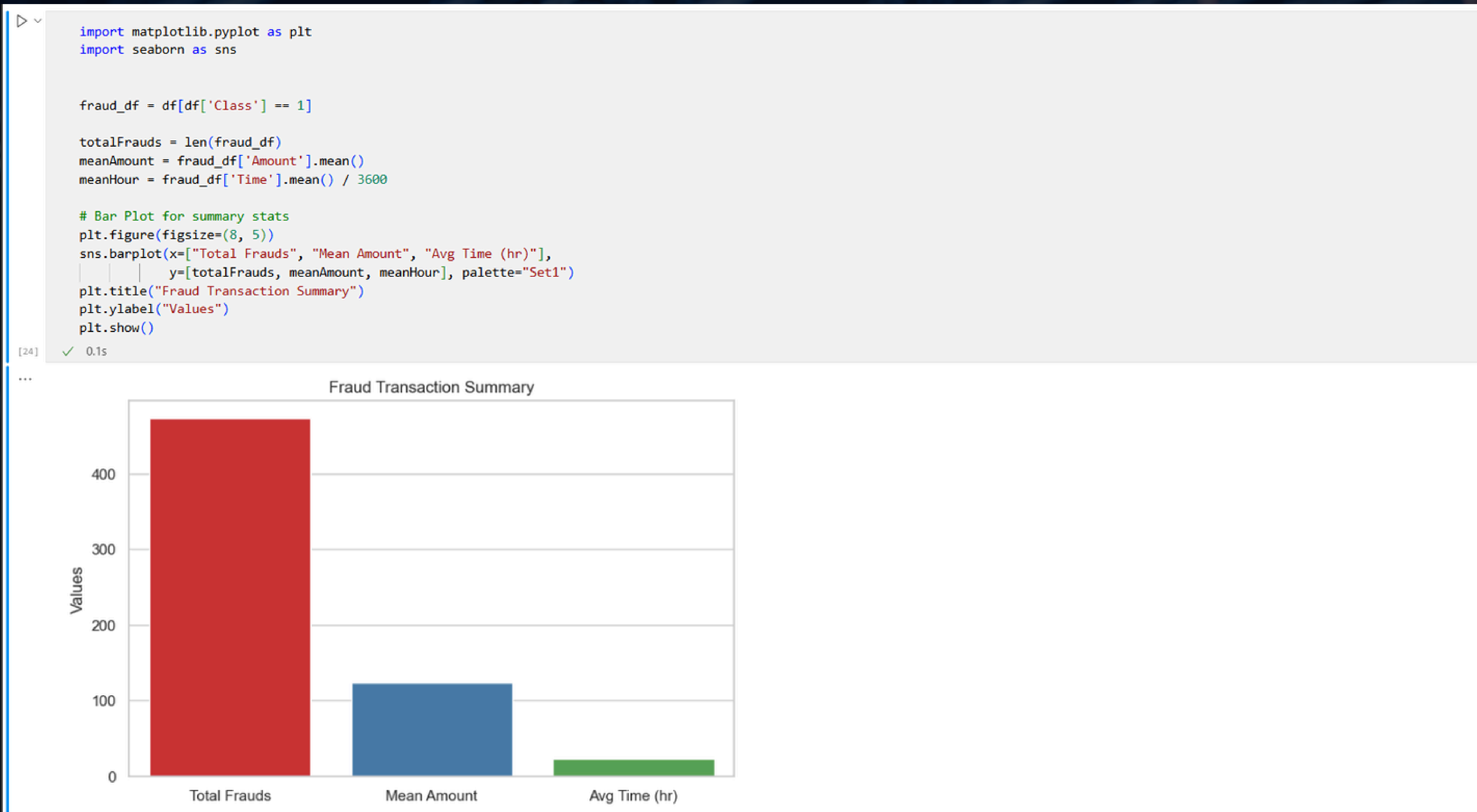
```
plt.figure(figsize=(10, 5))
sns.kdeplot(df[df['Class'] == 0]['Amount'], label='Non-Fraud', fill=True)
sns.kdeplot(df[df['Class'] == 1]['Amount'], label='Fraud', fill=True)
plt.xlim(0, 2000)  # focus on most common amounts
plt.title('KDE Plot of Transaction Amount by Class')
plt.xlabel('Amount')
plt.ylabel('Density')
plt.legend()
plt.show()
```

[20] ✓ 0.9s



Data Analysis

Bar Plot For Summary



Data Analysis

Correlation

CORRELATION

```python
corr_matrix = df.corr()
```

[75] Python

```python
# Correlation with Class
corr_with_class = corr_matrix['Class'].drop('Class').sort_values(ascending=False)
print("\nTop Positively Correlated Features with Class:")
print(corr_with_class.head(10))
print("\nTop Negatively Correlated Features with Class:")
print(corr_with_class.tail(10))
```

[76] Python

```
Top Positively Correlated Features with Class:
V11       0.149067
V4        0.129326
V2        0.084624
V19       0.033631
V8        0.033068
V21       0.026357
V27       0.021892
V20       0.021486
V28       0.009682
Amount    0.005777
Name: Class, dtype: float64

Top Negatively Correlated Features with Class:
V9    -0.094021
V1    -0.094486
V18   -0.105340
V7    -0.172347
V3    -0.182322
V16   -0.187186
V10   -0.206971
V12   -0.250711
V14   -0.293375
V17   -0.313498
Name: Class, dtype: float64
```

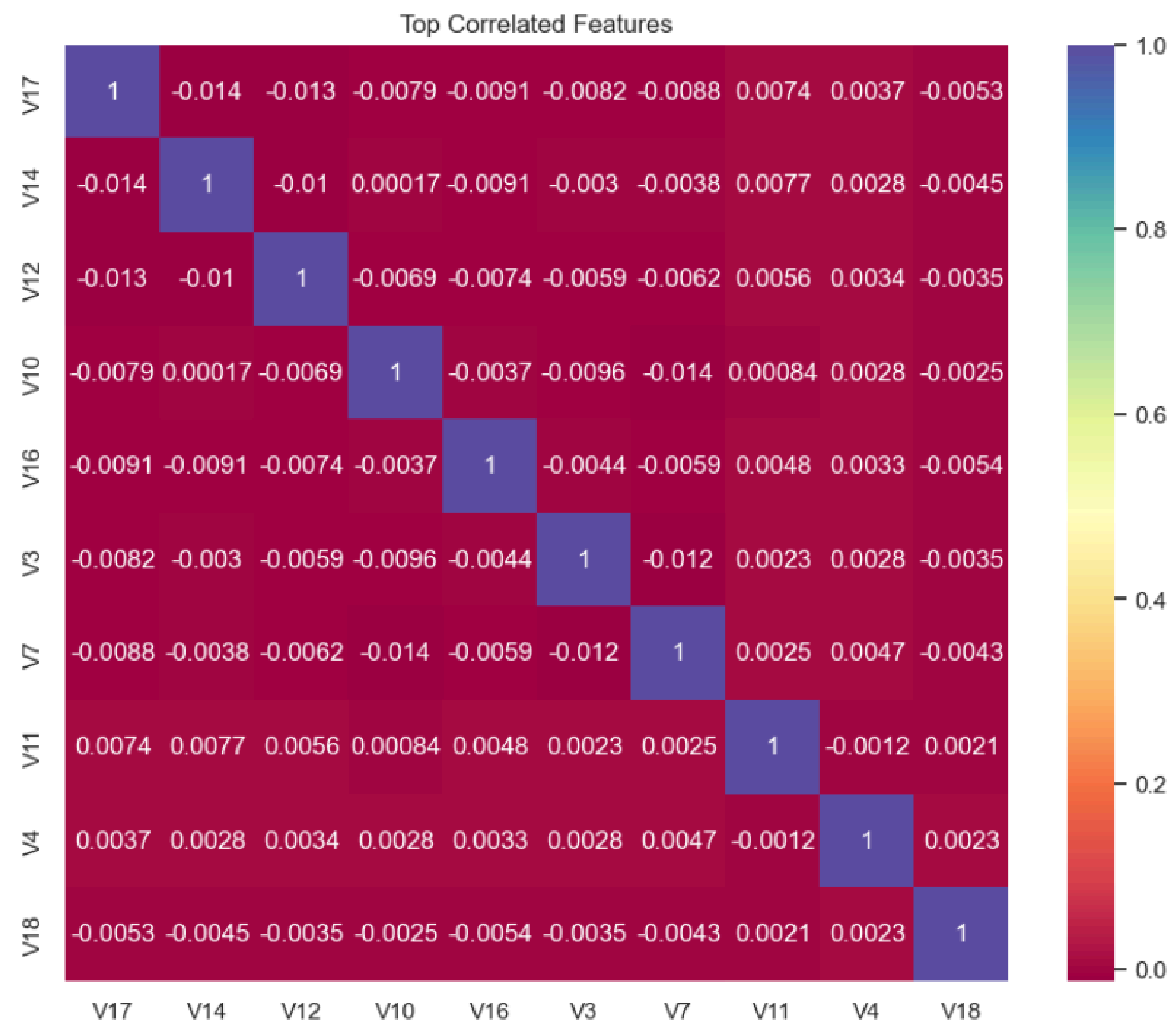
Top Positively Correlated: V11, V4, V2, V19, V8, V21, V27, V20, V28
Top Negatively Correlated: V17, V14, V12, V10, V16, V7, V3, V18, V1, V9

Data Analysis

HeatMap

```python
# Heatmap of top 10 correlated features
plt.figure(figsize=(10, 8))
top_features = corr_with_class.abs().sort_values(ascending=False).head(10).index
sns.heatmap(df[top_features].corr(), annot=True, cmap='Spectral')
plt.title("Top Correlated Features")
plt.show()
```

[77]



Data Analysis

Feature Scaling

Feature Scaling

```
scaler = StandardScaler()
df['scaled_amount'] = scaler.fit_transform(df[['Amount']])
df['scaled_time'] = scaler.fit_transform(df[['Time']])
df.drop(['Amount', 'Time'], axis=1, inplace=True)
```

[78]

Data Analysis

UnderSampling

Undersampling

```
fraud = df[df['Class'] == 1]
non_fraud = df[df['Class'] == 0]

print("\nBefore resampling:")
print(f"Fraud cases: {len(fraud)} | Non-fraud cases: {len(non_fraud)}")

# Undersample majority class to match minority
non_fraud_sampled = non_fraud.sample(n=len(fraud), random_state=42)
new_df = pd.concat([fraud, non_fraud_sampled], ignore_index=True)

print("\nAfter undersampling:")
print(new_df['Class'].value_counts())

X = new_df.drop('Class', axis=1)
y = new_df['Class']
```

[80]

```
Before resampling:
Fraud cases: 473 | Non-fraud cases: 283253

After undersampling:
Class
1    473
0    473
Name: count, dtype: int64
```

LOGISTIC REGRESSION

Logistic Regression is a interpretable model used for binary classification. Here, it learns to separate fraudulent from legitimate transactions by finding the best decision boundary based on the training data.

```
from sklearn.linear_model import LogisticRegression
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Model training
lr = LogisticRegression(random_state=42, max_iter=1000)
lr.fit(x_train, y_train)
```

This code shows how well the Logistic Regression model performs using accuracy, precision, recall, F1 score, and AUPRC. These values show high accuracy and excellent balance between detecting fraud and minimizing false positives."

```
# Evaluation
print("\nLogistic Regression Metrics:")
print("Accuracy:", accuracy_score(y_test, y_pred_lr))
print("Precision:", precision_score(y_test, y_pred_lr))
print("Recall:", recall_score(y_test, y_pred_lr))
print("F1 Score:", f1_score(y_test, y_pred_lr))
print("AUPRC:", average_precision_score(y_test, y_scores_lr))

print("\nClassification Report:\n", classification_report(y_test, y_pred_lr))
```

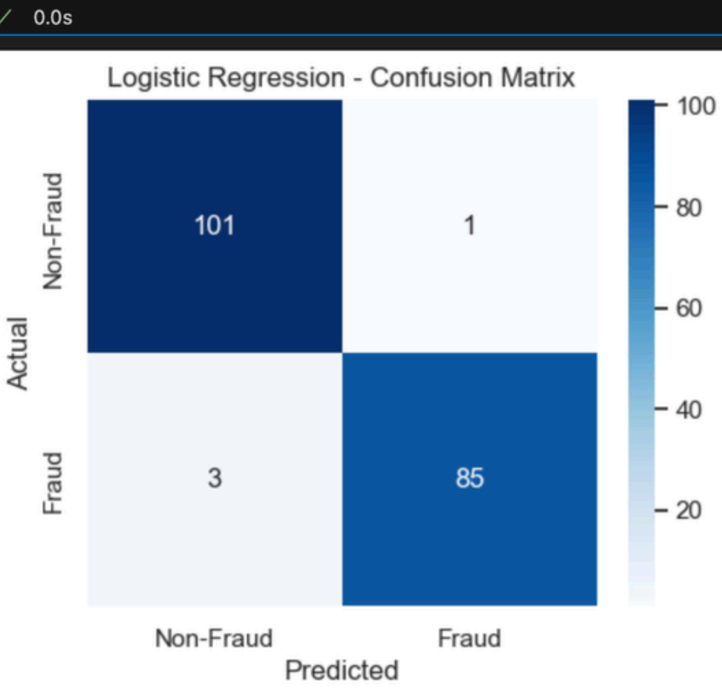
✓ 0.0s

```
Logistic Regression Metrics:
Accuracy: 0.9789473684210527
Precision: 0.9883720930232558
Recall: 0.9659090909090909
F1 Score: 0.9770114942528736
AUPRC: 0.992045528623561

Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       102
           1       0.99      0.97      0.98        88

    accuracy                           0.98       190
   macro avg       0.98      0.98      0.98       190
weighted avg       0.98      0.98      0.98       190
```

LOGISTIC REGRESSION

```
# Confusion Matrix plot
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_lr), annot=True, fmt='d', cmap='Blues',
            xticklabels=['Non-Fraud', 'Fraud'], yticklabels=['Non-Fraud', 'Fraud'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Logistic Regression - Confusion Matrix")
plt.show()
```

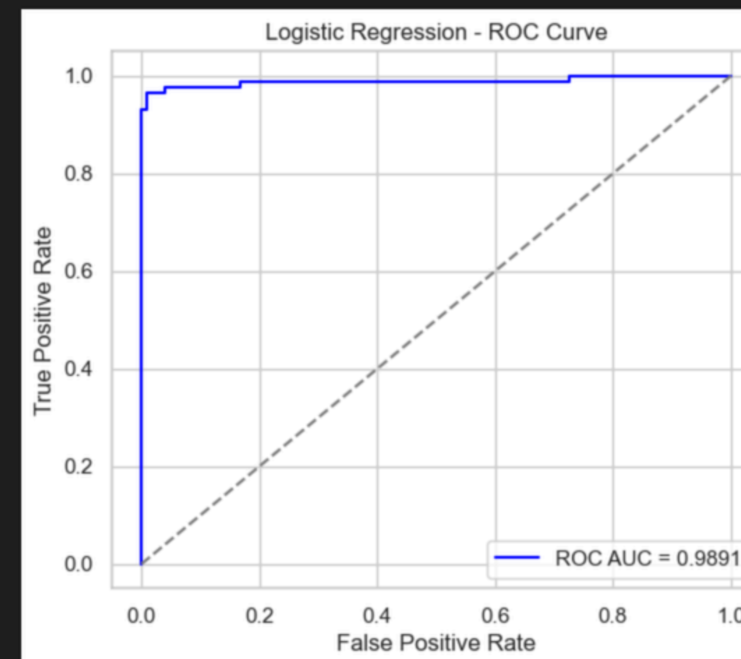
✓ 0.0s



This plot visualizes the model's predictions vs actual labels, showing how many fraud and non-fraud cases were correctly or incorrectly classified.

```
plt.figure(figsize=(6, 5))
plt.plot(fpr_lr, tpr_lr, label=f"ROC AUC = {roc_auc_lr:.4f}", color='blue')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Logistic Regression - ROC Curve')
plt.legend()
plt.grid(True)
plt.show()
```

✓ 0.1s



This curve shows the trade-off between true positive rate and false positive rate. The ROC AUC score indicates overall model performance — higher is better.

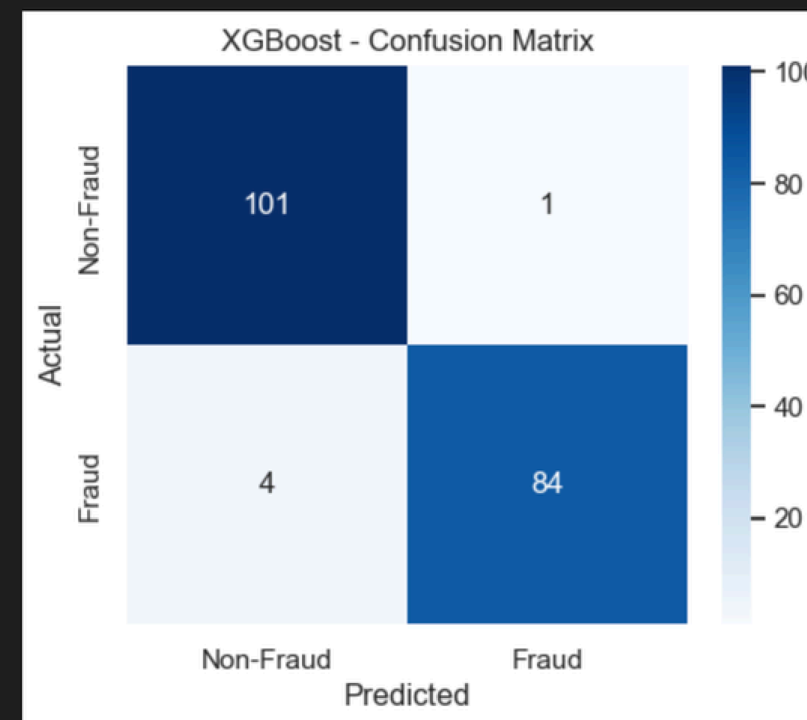
XGBOOST MODEL

```python
import xgboost as xgb

# Model training
xgb_clf = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42)
xgb_clf.fit(x_train, y_train)
```

XGBoost is an advanced gradient boosting method that builds multiple decision trees sequentially. It focuses on correcting errors from previous trees to improve prediction accuracy, making it very effective for complex datasets.

```python
# Confusion Matrix plot
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_xgb), annot=True, fmt='d', cmap='Blues',
            xticklabels=['Non-Fraud', 'Fraud'], yticklabels=['Non-Fraud', 'Fraud'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("XGBoost - Confusion Matrix")
plt.show()
```

✓ 0.0s

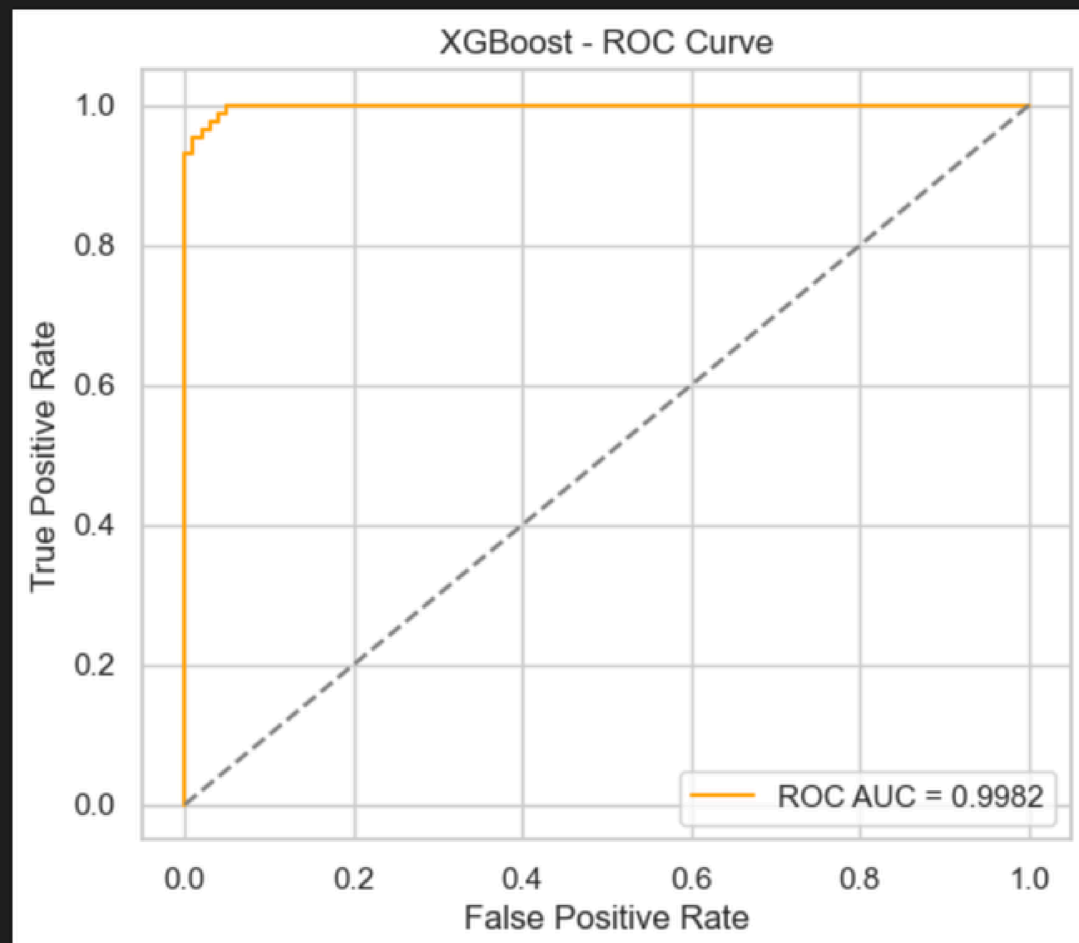


This heatmap shows correct and incorrect predictions by the XGBoost model. It helps understand model accuracy on each class.

XGBOOST MODEL

```
# ROC Curve
fpr_xgb, tpr_xgb, _ = roc_curve(y_test, y_scores_xgb)
roc_auc_xgb = auc(fpr_xgb, tpr_xgb)

plt.figure(figsize=(6, 5))
plt.plot(fpr_xgb, tpr_xgb, label=f"ROC AUC = {roc_auc_xgb:.4f}", color='orange')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('XGBoost - ROC Curve')
plt.legend()
plt.grid(True)
plt.show()
```

✓ 0.0s



- This ROC curve shows the trade-off between true positive rate and false positive rate.
- The closer the curve is to the top-left, the better the model.
- The AUC score tells overall performance (higher is better).

RANDOM FOREST MODEL

```
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(x_train, y_train)
y_pred = rf.predict(x_test)
```

✓ 0.2s

Train a random forest with 100 decision trees to learn fraud patterns.

```
# Classification Report
print("\nClassification Report:\n")
print(classification_report(y_test, y_pred))
```

✓ 0.0s

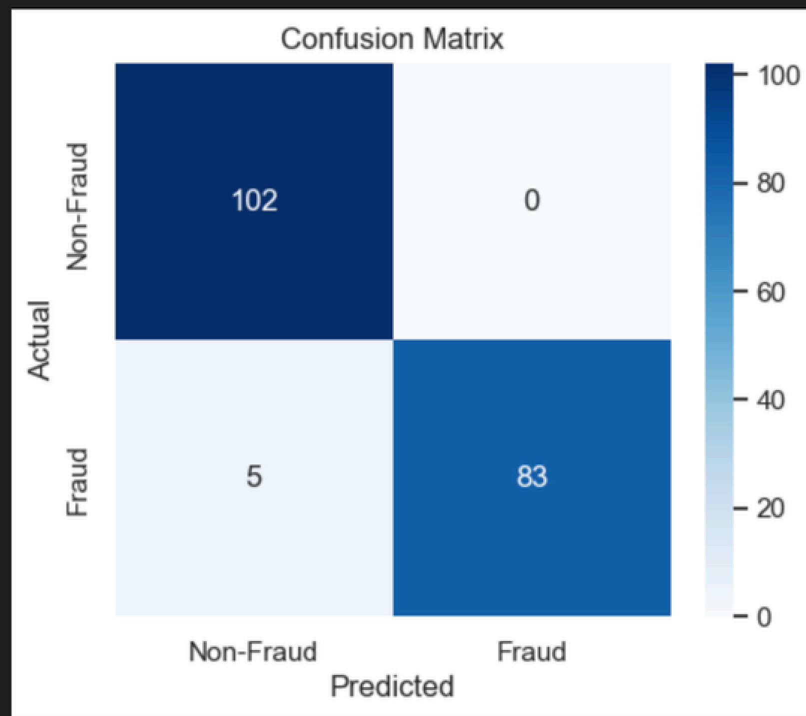
```
Classification Report:

              precision    recall  f1-score   support

           0       0.95      1.00      0.98       102
           1       1.00      0.94      0.97        88

    accuracy                           0.97       190
   macro avg       0.98      0.97      0.97       190
weighted avg       0.97      0.97      0.97       190
```

This report shows precision, recall, F1-score, and support for both classes (fraud and non-fraud).
It's a quick summary of how well the model is doing on each class.

RANDOM FOREST MODEL

```python
# Confusion Matrix
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt='d', cmap='Blues',
            xticklabels=['Non-Fraud', 'Fraud'], yticklabels=['Non-Fraud', 'Fraud'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Confusion Matrix")
plt.show()
```

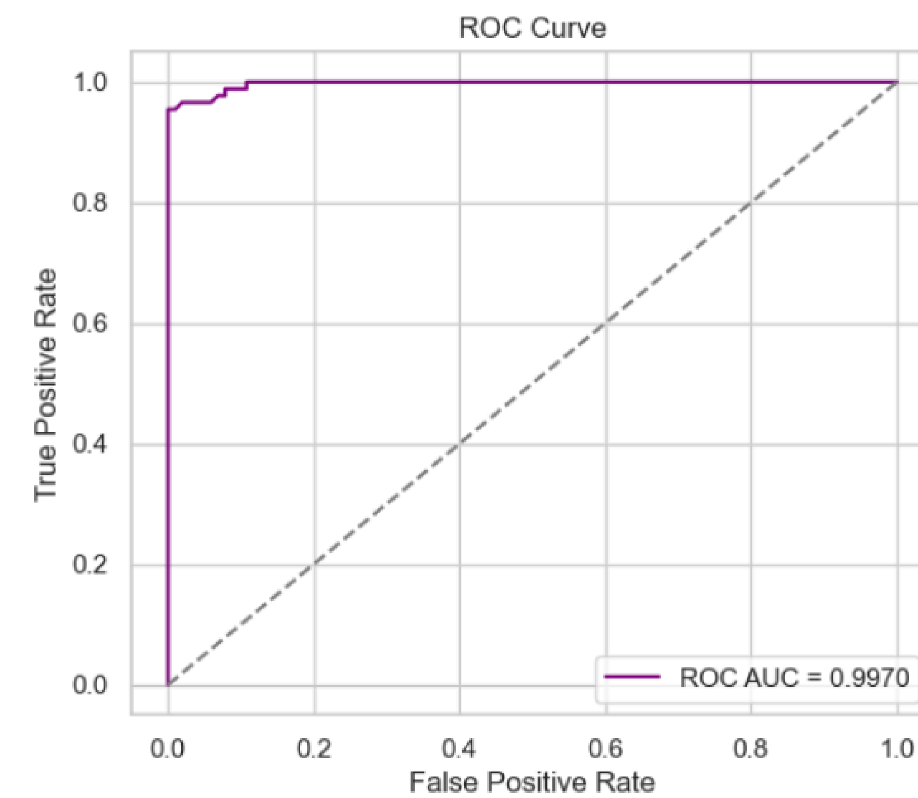
✓ 0.0s



This heatmap shows the number of fraud and non fraud predictions.

```python
# ROC Curve
fpr, tpr, _ = roc_curve(y_test, y_scores)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, label=f"ROC AUC = {roc_auc:.4f}", color='purple')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.grid(True)
plt.show()
```

[97]



DEEP LEARNING MODEL (KERAS)

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
```
[39] ✓ 4.0s Python

```
#Model architecture
model = Sequential([
    Dense(32, input_dim=x_train.shape[1], activation='relu'),
    Dropout(0.3),
    Dense(16, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')  # Binary classification output
])
```
[40] ✓ 0.2s Python

```
2025-06-16 10:13:26.066073: I metal_plugin/src/device/metal_device.cc:1154] Metal device set to: Apple M1
2025-06-16 10:13:26.066260: I metal_plugin/src/device/metal_device.cc:296] systemMemory: 8.00 GB
2025-06-16 10:13:26.066266: I metal_plugin/src/device/metal_device.cc:313] maxCacheSize: 2.67 GB
2025-06-16 10:13:26.066597: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:306] Could not identify NUMA node of
2025-06-16 10:13:26.066917: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:272] Created TensorFlow device (/job
```

```
# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy', tf.keras.metrics.Precision(), tf.keras.metrics.Recall()])

# Early stopping to avoid overfitting
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
```
[41] ✓ 0.0s Python

This neural network has layers that learn complex data patterns by transforming inputs through nonlinear functions (ReLU). Dropout layers randomly deactivate neurons during training to prevent overfitting. Early stopping halts training when performance on validation data stops improving.

DEEP LEARNING MODEL (KERAS)

```
# Train the model
history = model.fit(
    x_train, y_train,
    epochs=50,
    batch_size=64,
    validation_split=0.2,
    callbacks=[early_stop],
    verbose=2
)
```

[101]

```
Epoch 1/50
WARNING:tensorflow:From c:\Users\cholt\anaconda3\envs\env\Lib\site-packages\keras\src\utils\tf_utils.py:492: The name tf.ragged.RaggedTensorValue is deprecated. Please use tf.compat.v1.ragged.RaggedTensorValue instead.

WARNING:tensorflow:From c:\Users\cholt\anaconda3\envs\env\Lib\site-packages\keras\src\engine\base_layer_utils.py:384: The name tf.executing_eagerly_outside_functions is deprecated. Please use tf.compat.v1.executing_eagerly_

10/10 - 2s - loss: 1.1789 - accuracy: 0.5629 - precision: 0.5467 - recall: 0.7869 - val_loss: 0.5150 - val_accuracy: 0.7171 - val_precision: 0.6637 - val_recall: 0.9375 - 2s/epoch - 198ms/step
Epoch 2/50
10/10 - 0s - loss: 0.8849 - accuracy: 0.6159 - precision: 0.5910 - recall: 0.7770 - val_loss: 0.3761 - val_accuracy: 0.8355 - val_precision: 0.8395 - val_recall: 0.8500 - 72ms/epoch - 7ms/step
Epoch 3/50
10/10 - 0s - loss: 0.6167 - accuracy: 0.7086 - precision: 0.6817 - recall: 0.7934 - val_loss: 0.3358 - val_accuracy: 0.8421 - val_precision: 0.8784 - val_recall: 0.8125 - 79ms/epoch - 8ms/step
Epoch 4/50
10/10 - 0s - loss: 0.5571 - accuracy: 0.7318 - precision: 0.7147 - recall: 0.7803 - val_loss: 0.3144 - val_accuracy: 0.8487 - val_precision: 0.8904 - val_recall: 0.8125 - 75ms/epoch - 8ms/step
Epoch 5/50
10/10 - 0s - loss: 0.5337 - accuracy: 0.7798 - precision: 0.7575 - recall: 0.8295 - val_loss: 0.2995 - val_accuracy: 0.8487 - val_precision: 0.8904 - val_recall: 0.8125 - 76ms/epoch - 8ms/step
Epoch 6/50
10/10 - 0s - loss: 0.4685 - accuracy: 0.8046 - precision: 0.8026 - recall: 0.8131 - val_loss: 0.2902 - val_accuracy: 0.8487 - val_precision: 0.8904 - val_recall: 0.8125 - 74ms/epoch - 7ms/step
Epoch 7/50
10/10 - 0s - loss: 0.4861 - accuracy: 0.7831 - precision: 0.7736 - recall: 0.8066 - val_loss: 0.2840 - val_accuracy: 0.8487 - val_precision: 0.8904 - val_recall: 0.8125 - 71ms/epoch - 7ms/step
Epoch 8/50
10/10 - 0s - loss: 0.4601 - accuracy: 0.7964 - precision: 0.7935 - recall: 0.8066 - val_loss: 0.2769 - val_accuracy: 0.8487 - val_precision: 0.8904 - val_recall: 0.8125 - 71ms/epoch - 7ms/step
Epoch 9/50
10/10 - 0s - loss: 0.4680 - accuracy: 0.8162 - precision: 0.8170 - recall: 0.8197 - val_loss: 0.2698 - val_accuracy: 0.8487 - val_precision: 0.8904 - val_recall: 0.8125 - 71ms/epoch - 7ms/step
```

```
# Predictions and evaluation
y_pred_dl_prob = model.predict(x_test).flatten()
y_pred_dl = (y_pred_dl_prob > 0.5).astype(int)
```

```
6/6 [==============================] - 0s 2ms/step
```

Early stopping to avoid overfitting and train over 50 epochs.

DEEP LEARNING MODEL (KERAS)

```
print("\nDeep Learning Model Metrics:")
print("Accuracy:", accuracy_score(y_test, y_pred_dl))
print("Precision:", precision_score(y_test, y_pred_dl))
print("Recall:", recall_score(y_test, y_pred_dl))
print("F1 Score:", f1_score(y_test, y_pred_dl))
print("AUPRC:", average_precision_score(y_test, y_pred_dl_prob))

print("\nClassification Report:\n", classification_report(y_test, y_pred_dl))
```

```
[45]  ✓ 0.0s
```

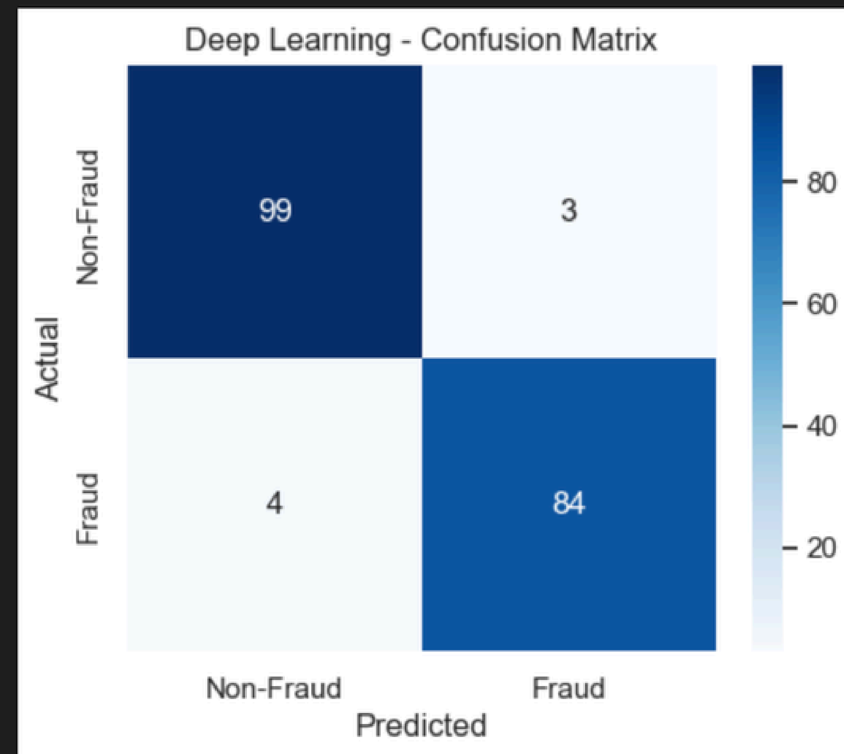
```
Deep Learning Model Metrics:
Accuracy: 0.9631578947368421
Precision: 0.9655172413793104
Recall: 0.9545454545454546
F1 Score: 0.96
AUPRC: 0.9937471319415195

Classification Report:
               precision    recall  f1-score   support

           0       0.96      0.97      0.97       102
           1       0.97      0.95      0.96        88

    accuracy                           0.96       190
   macro avg       0.96      0.96      0.96       190
weighted avg       0.96      0.96      0.96       190
```

- Shows how well it detects fraud (Precision, Recall, F1).
- AUPRC tells how well the model ranks fraud cases.
- classification_report gives detailed per-class metrics.

DEEP LEARNING MODEL (KERAS)

```python
# Confusion Matrix plot
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_dl), annot=True, fmt='d', cmap='Blues',
            xticklabels=['Non-Fraud', 'Fraud'], yticklabels=['Non-Fraud', 'Fraud'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title("Deep Learning - Confusion Matrix")
plt.show()
```

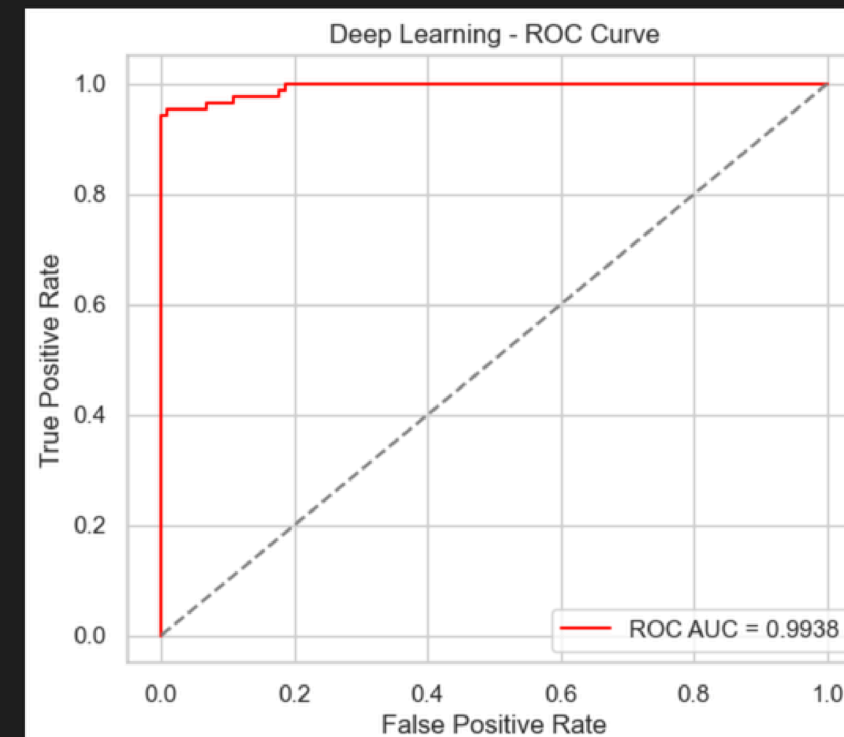
✓ 0.0s



This plot shows how many fraud and non-fraud cases the deep learning model predicted correctly vs incorrectly

```python
# ROC Curve
fpr_dl, tpr_dl, _ = roc_curve(y_test, y_pred_dl_prob)
roc_auc_dl = auc(fpr_dl, tpr_dl)

plt.figure(figsize=(6, 5))
plt.plot(fpr_dl, tpr_dl, label=f"ROC AUC = {roc_auc_dl:.4f}", color='red')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Deep Learning - ROC Curve')
plt.legend()
plt.grid(True)
plt.show()
```

✓ 0.0s



This ROC curve shows how well the deep learning model distinguishes fraud from non-fraud at different thresholds.
The closer the curve follows the top-left corner, the better the model.
The AUC score summarizes the performance.

MODEL COMPARISONS

Model	Accuracy	Precision	Recall	F1 Score	AUPRC (Positive Fraud)	ROC AUC
Logistic Regression	**97.9%**	98.8%	**96.6%**	**97.7%**	0.992	0.989
XGBoost	97.4%	98.8%	95.5%	97.1%	**0.998**	**0.998**
Random Forest	97.4%	**100.0%**	94.3%	97.1%	0.969	0.997
Deep Learning	97.4%	98.8%	95.5%	97.1%	0.994	0.994

All of the models have high accuracy.
XG Boost has the best AUPRC & ROCAUC
Random Forest has 0 False Positives

THANK YOU