## Import necessary libraries

```
# Import necessary libraries
import pandas as pd
import plotly.express as px
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import classification_report, roc_auc_score
```

## 1. Load the dataset

Display Dataset:

```
# Load the dataset
data = pd.read_csv("creditcard_2023.csv")

#Display Dataset:
display(data)
```

| | id | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | ... | V21 | V22 | V23 | V24 | V25 | V26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | -0.260648 | -0.469648 | 2.496266 | -0.083724 | 0.129681 | 0.732898 | 0.519014 | -0.130006 | 0.727159 | ... | -0.110552 | 0.217606 | -0.134794 | 0.165959 | 0.126280 | -0.434824 |
| **1** | 1 | 0.985100 | -0.356045 | 0.558056 | -0.429654 | 0.277140 | 0.428605 | 0.406466 | -0.133118 | 0.347452 | ... | -0.194936 | -0.605761 | 0.079469 | -0.577395 | 0.190090 | 0.296503 |
| **2** | 2 | -0.260272 | -0.949385 | 1.728538 | -0.457986 | 0.074062 | 1.419481 | 0.743511 | -0.095576 | -0.261297 | ... | -0.005020 | 0.702906 | 0.945045 | -1.154666 | -0.605564 | -0.312895 |
| **3** | 3 | -0.152152 | -0.508959 | 1.746840 | -1.090178 | 0.249486 | 1.143312 | 0.518269 | -0.065130 | -0.205698 | ... | -0.146927 | -0.038212 | -0.214048 | -1.893131 | 1.003963 | -0.515950 |
| **4** | 4 | -0.206820 | -0.165280 | 1.527053 | -0.448293 | 0.106125 | 0.530549 | 0.658849 | -0.212660 | 1.049921 | ... | -0.106984 | 0.729727 | -0.161666 | 0.312561 | -0.414116 | 1.071126 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **38582** | 38582 | 0.512593 | -1.325943 | 0.806190 | -0.723666 | -0.326859 | 0.635427 | 0.194875 | -0.107927 | 0.533281 | ... | -0.001392 | -0.124776 | -0.400736 | 0.054451 | 0.250665 | -0.658165 |
| **38583** | 38583 | 0.885101 | -0.439196 | 0.674228 | -0.108576 | 0.185739 | 0.554800 | 0.416792 | -0.114392 | 0.580456 | ... | -0.120088 | -0.060441 | -0.114652 | 0.075363 | 0.919730 | -0.790809 |
| **38584** | 38584 | 0.870208 | -0.432235 | 0.500383 | -0.496220 | 0.512440 | 1.252223 | 0.304214 | -0.011410 | 0.424280 | ... | -0.162226 | -0.346800 | 0.192908 | -2.226695 | -0.256609 | 0.516845 |
| **38585** | 38585 | -0.556968 | -2.304167 | 0.155227 | -0.884494 | -0.983298 | 1.144901 | 4.190189 | -0.368090 | -0.434704 | ... | 0.083516 | -0.864799 | 4.633895 | 1.088068 | 0.011199 | 1.109009 |
| **38586** | 38586 | -0.359812 | -0.646911 | 2.730844 | -0.585401 | 0.676308 | 0.723639 | 0.254078 | 0.017832 | 0.578835 | ... | NaN | NaN | NaN | NaN | NaN | NaN |

38587 rows × 31 columns

## 2. Summary statistics

```
# Summary statistics
print("\nSummary statistics of the dataset:")
print(data.describe())
```

```
Summary statistics of the dataset:
                 id            V1            V2            V3            V4  \
count  38587.000000  38587.000000  38587.000000  38587.000000  38587.000000   
mean   19293.000000      0.329214     -0.465955      1.050013     -0.642976   
std    11139.251755      0.625825      0.619331      0.700288      0.658461   
min        0.000000     -2.637662    -25.480046     -2.518308     -4.468314   
25%     9646.500000     -0.151668     -0.630139      0.595152     -1.004414   
50%    19293.000000      0.099677     -0.409237      0.955077     -0.541133   
75%    28939.500000      0.959805     -0.204195      1.442522     -0.190550   
max    38586.000000      1.695400      3.368287      4.440555      2.576901   

                 V5            V6            V7            V8            V9  \
count  38587.000000  38587.000000  38587.000000  38587.000000  38587.000000   
mean       0.242730      0.492922      0.445529     -0.134558      0.747091   
std        0.637452      0.713223      0.536618      0.264173      0.866844   
min       -5.182978    -18.642598     -3.038728     -6.595785     -2.170915   
25%       -0.029739      0.095670      0.248569     -0.186744      0.191389   
50%        0.155470      0.384055      0.408164     -0.141746      0.556262   
75%        0.376178      0.747130      0.594214     -0.080073      1.147657   
max       41.540257      9.704313     41.568286      5.958040     12.171681   

       ...           V21           V22           V23           V24  \
count  ...  38586.000000  38586.000000  38586.000000  38586.000000   
mean   ...     -0.121236     -0.123657     -0.053350      0.139499   
std    ...      0.289004      0.575545      0.554678      1.105137   
min    ...    -10.699713     -6.262872    -19.297443     -4.067968   
25%    ...     -0.199850     -0.517493     -0.202367     -0.553483   
50%    ...     -0.134852     -0.122482     -0.076201      0.154298   
75%    ...     -0.063591      0.237285      0.054520      0.845794   
max    ...      6.834415      6.492043     23.570216     11.029578   

                V25           V26           V27           V28        Amount  \
count  38586.000000  38586.000000  38586.000000  38586.000000  38586.000000   
mean       0.155278     -0.026937     -0.204345     -0.091465  11974.093611   
std        0.711288      1.113339      0.475447      0.730140   6929.383452   
min      -10.230967     -4.059129     -5.536809    -24.176384     50.120000   
25%       -0.290175     -0.791056     -0.309006     -0.116947   5973.705000   
50%        0.204328     -0.132786     -0.227996     -0.048485  11875.340000   
75%        0.617031      0.668685     -0.138486      0.083287  17990.095000   
max       10.424231      5.623285     26.944437     13.277404  24039.880000   

              Class  
count  38586.000000  
mean       0.002669  
std        0.051598  
min        0.000000  
25%        0.000000  
50%        0.000000  
75%        0.000000  
max        1.000000  

[8 rows x 31 columns]
```

## 3. Data Preprocessing

### 3.1 Handle Missing Values

```
# Check for missing values

print("\nMissing values in the dataset:")
print(data.isnull().sum())
```

```
Missing values in the dataset:
id        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       1
V22       1
V23       1
V24       1
V25       1
V26       1
V27       1
V28       1
Amount    1
Class     1
dtype: int64
```

```
# Drop rows with missing values
data.dropna(inplace=True)

# Verify that missing values have been handled
print("Missing values after handling:")
print(data.isnull().sum())

# Check the shape of the DataFrame after handling missing values
print("\nShape of the DataFrame after handling missing values:", data.shape)
```

```
Missing values after handling:
id        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

Shape of the DataFrame after handling missing values: (38586, 31)
```

### 3.2 Explore and Address Class Imbalance

```
# 2. Explore and Address Class Imbalance
# Check the distribution of the target variable (Class)
class_distribution = data['Class'].value_counts()
print("\nDistribution of the target variable (Class):")
print(class_distribution)

# Visualize class distribution
fig = px.bar(x=class_distribution.index, y=class_distribution.values, labels={'x': 'Class', 'y': 'Count'},
             title="Distribution of Target Variable (Class)")
fig.show()

# As we can see, there is a class imbalance in the dataset, with a significantly higher number of non-fraudule
```

```
Distribution of the target variable (Class):
0.0    38483
1.0      103
Name: Class, dtype: int64
```


## Observation:

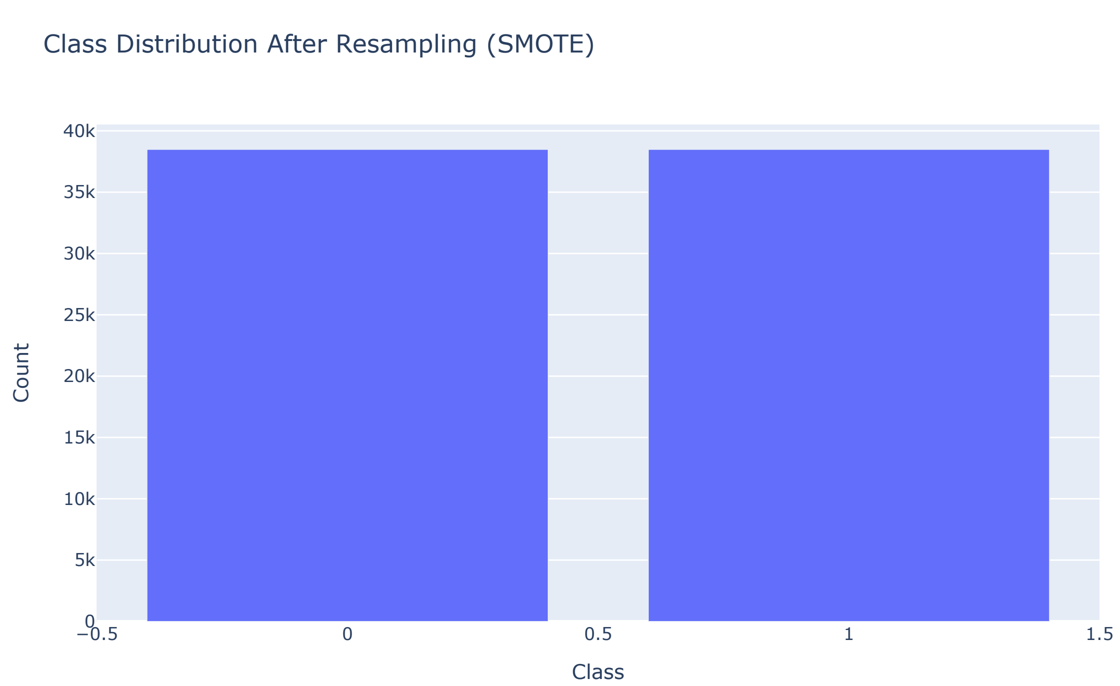
> The distribution of the target variable 'Class' indicates a class imbalance issue, with a significantly higher number of non-fraudulent transactions compared to fraudulent transactions.

---

## Data Preprocessing - Addressing Class Imbalance

```
# Import the SMOTE module
from imblearn.over_sampling import SMOTE

# Separate features (X) and target variable (y)
X = data.drop(columns=['Class'])
y = data['Class']

# Instantiate SMOTE
smote = SMOTE(random_state=42)

# Generate synthetic samples
X_resampled, y_resampled = smote.fit_resample(X, y)

# Check the class distribution after resampling
print("Class distribution after resampling:")
print(y_resampled.value_counts())

# Visualize class distribution after resampling
fig = px.bar(x=y_resampled.value_counts().index, y=y_resampled.value_counts().values,
             labels={'x': 'Class', 'y': 'Count'},
             title="Class Distribution After Resampling (SMOTE)")
fig.show()
```

```
Class distribution after resampling:
0.0    38483
1.0    38483
Name: Class, dtype: int64
```




## Observation:

- SMOTE (Synthetic Minority Over-sampling Technique) is applied to address the class imbalance by generating synthetic samples for the minority class.
- After resampling, the class distribution is balanced, which improves the model's performance in handling class imbalance.

---

## Additional Data Preprocessing

### Feature Scaling

> *StandardScaler is used to scale the features to have a mean of 0 and a standard deviation of 1.*

> *The scaled features are split into training and testing sets using a 80:20 ratio.*

```
# Additional Data Preprocessing
# Feature Scaling
from sklearn.preprocessing import StandardScaler

# Instantiate StandardScaler
scaler = StandardScaler()

# Fit and transform the features
X_resampled_scaled = scaler.fit_transform(X_resampled)

# Split the data into training and testing sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_resampled_scaled, y_resampled, test_size=0.2, random_sta

# Check the shape of the training and testing sets
print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
```

```
Shape of X_train: (61572, 30)
Shape of X_test: (15394, 30)
Shape of y_train: (61572,)
Shape of y_test: (15394,)
```

### Observation:

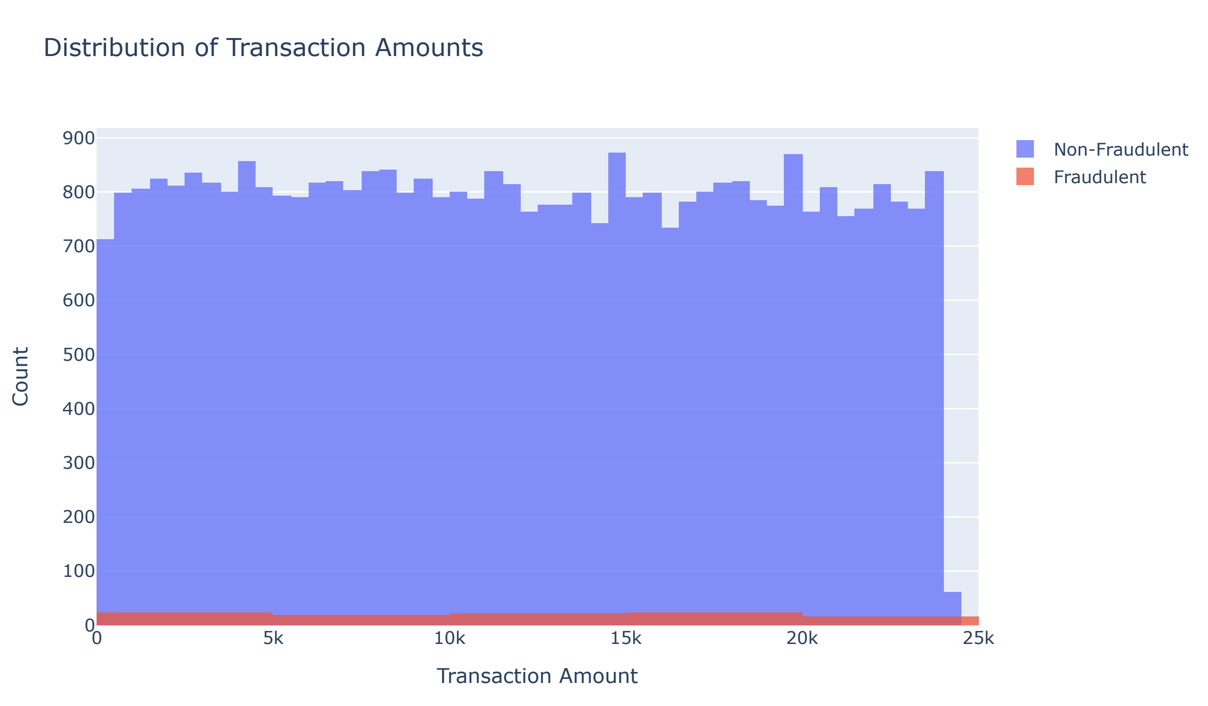
- Feature Scaling: Standardize the features by subtracting the mean and dividing by the standard deviation. We use StandardScaler from scikit-learn for this purpose.
- Splitting the Data: Split the resampled and scaled data into training and testing sets. We use 80% of the data for training and 20% for testing.

---

## Exploratory Data Analysis (EDA)

### *Distribution of Transaction Amounts*

```
import plotly.graph_objects as go

# Visualize the distribution of transaction amounts for fraudulent and non-fraudulent transactions
fig = go.Figure()

# Add histograms for fraudulent and non-fraudulent transactions
fig.add_trace(go.Histogram(x=data[data['Class'] == 0]['Amount'], name='Non-Fraudulent', opacity=0.75))
fig.add_trace(go.Histogram(x=data[data['Class'] == 1]['Amount'], name='Fraudulent', opacity=0.75))

# Update layout
fig.update_layout(barmode='overlay', title='Distribution of Transaction Amounts',
                  xaxis_title='Transaction Amount', yaxis_title='Count')

# Show plot
fig.show()
```



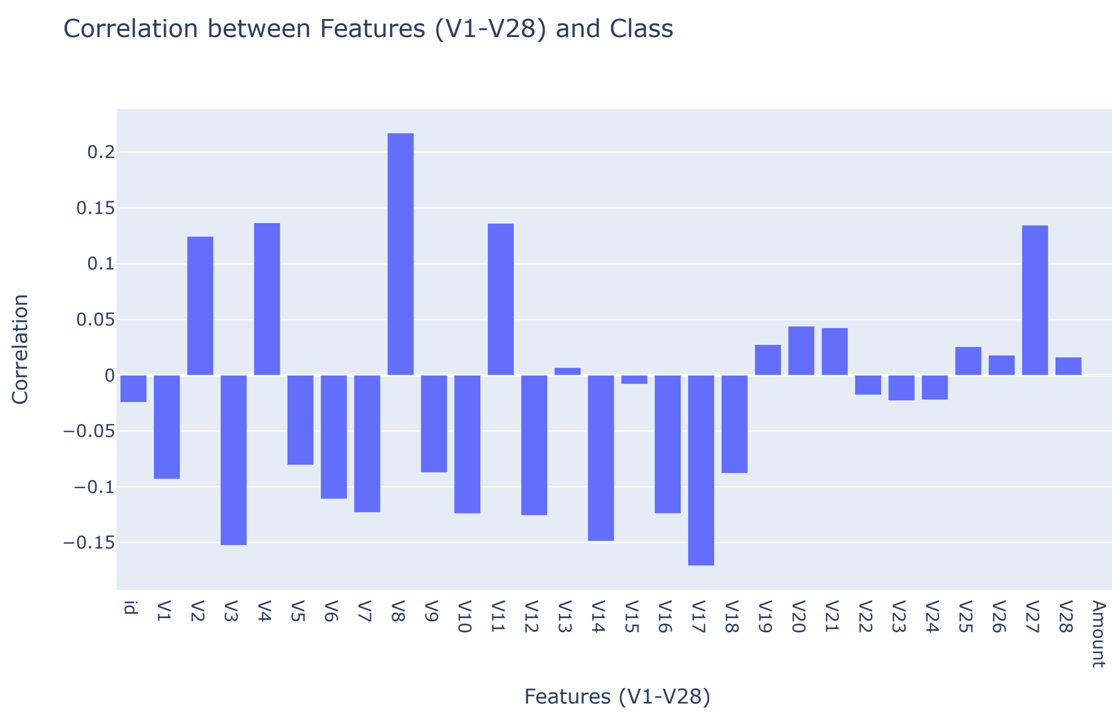

## Observation

- **Histogram of Transaction Amounts:** This plot shows the distribution of transaction amounts for fraudulent and non-fraudulent transactions. We use histograms to visualize the distribution of transaction amounts for both classes.
- It appears that non-fraudulent transactions have a wider range of transaction amounts compared to fraudulent transactions.

---

## *Correlations between Features (V1-V28) and the Target Variable (Class)*

```
# Explore correlations between features (V1-V28) and the target variable (Class)
correlation_matrix = data.corr()
correlation_with_class = correlation_matrix['Class'].drop('Class')

# Plot correlation between features and Class
fig = go.Figure(data=go.Bar(x=correlation_with_class.index, y=correlation_with_class.values))
fig.update_layout(title='Correlation between Features (V1-V28) and Class',
                  xaxis_title='Features (V1-V28)', yaxis_title='Correlation')
fig.show()
```



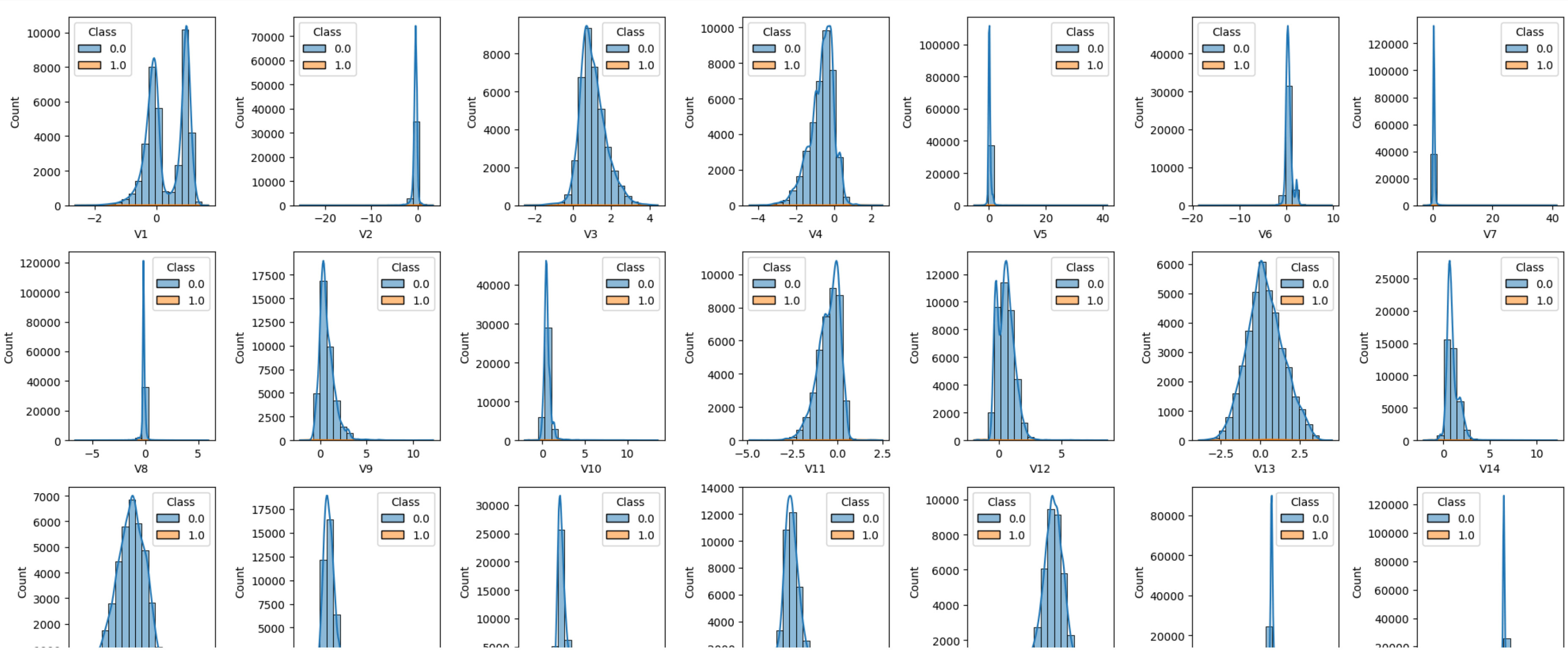

## Observation

- **Bar Chart of Feature Correlation with Class:** This plot shows the correlation between features (V1-V28) and the target variable (Class). We calculate the correlation coefficients and plot them as a bar chart.
- Some features show significant positive or negative correlations with the target variable, indicating potential predictive power.

---

## *Distribution of Individual Features (V1-V28) for Fraudulent and Non-Fraudulent Transactions*

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set up figure and axes
fig, axes = plt.subplots(nrows=4, ncols=7, figsize=(20, 12))

# Visualize the distribution of individual features (V1-V28) for fraudulent and non-fraudulent transactions
for i, col in enumerate(data.columns[1:29]):
    row_idx = i // 7
    col_idx = i % 7
    sns.histplot(data, x=col, hue="Class", bins=20, ax=axes[row_idx, col_idx], kde=True)

# Adjust layout
plt.tight_layout()
plt.show()
```



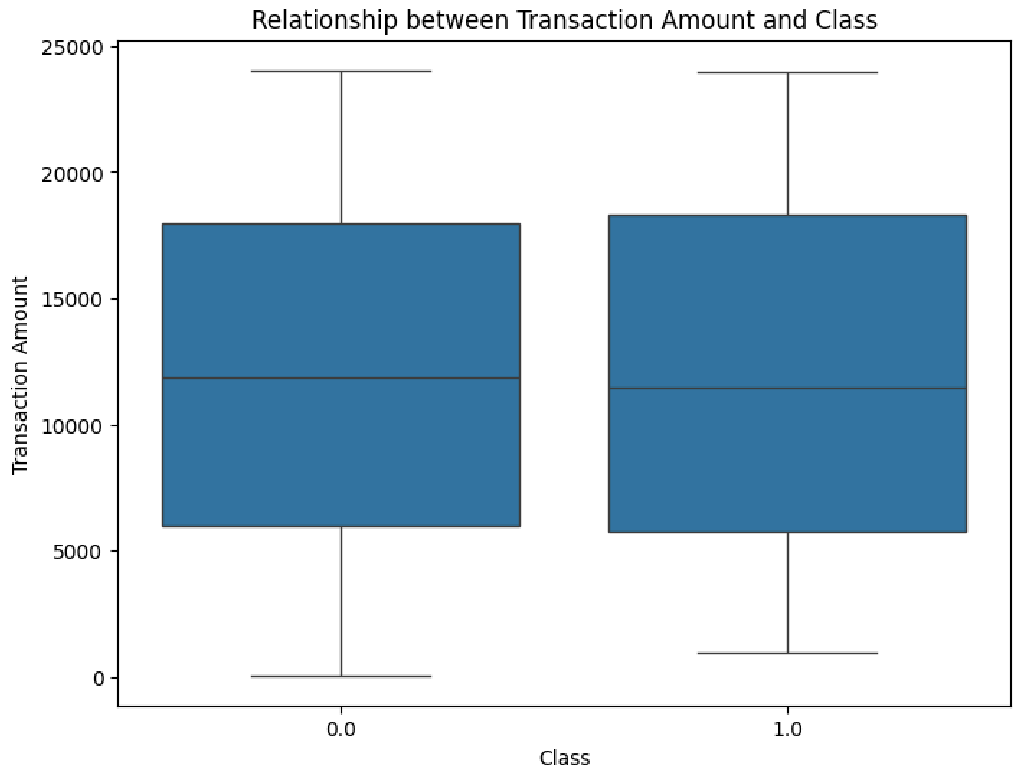

## Observation:

- The distribution of individual features (V1-V28) is visualized for fraudulent and non-fraudulent transactions using **histograms**.
- Differences in feature distributions between fraudulent and non-fraudulent transactions can provide insights into potential discriminative power for classification.
- Each histogram shows the distribution of values for a particular feature, with different colors indicating fraudulent and non-fraudulent transactions. The KDE overlay would provide a smoothed estimate of the probability density function of the data.

---

## Investigating the Relationship Between Transaction Amount and Class

```
# Investigate the relationship between transaction amount and class
plt.figure(figsize=(8, 6))
sns.boxplot(x='Class', y='Amount', data=data)
plt.title('Relationship between Transaction Amount and Class')
plt.xlabel('Class')
plt.ylabel('Transaction Amount')
plt.show()
```




## Observation:

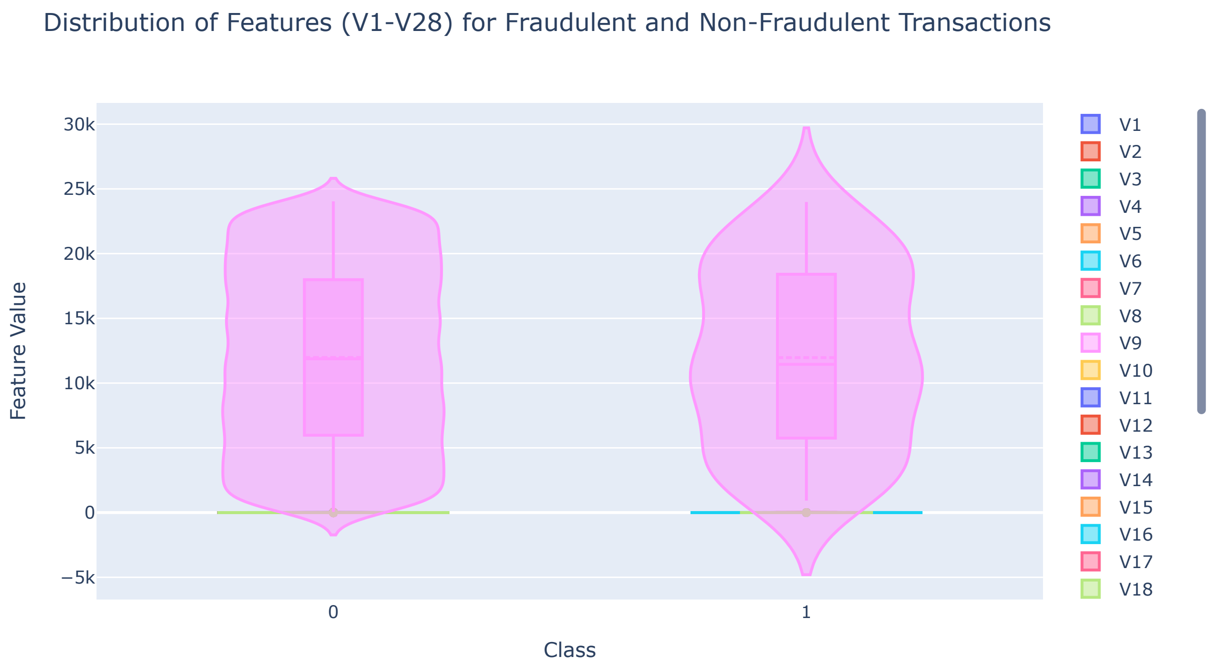
**Box Plot Visualization**

- The box plot shows the distribution of transaction amounts for fraudulent and non-fraudulent transactions.
- There appear to be outliers in both classes, particularly in the non-fraudulent transactions.

---

## Visualizing the Distribution of Features (V1-V28)

```
#import plotly.express as px

# Visualize the distribution of each feature (V1-V28) for fraudulent and non-fraudulent transactions
fig = go.Figure()

for col in data.columns[1:-1]:  # Exclude 'Class' column
    fig.add_trace(go.Violin(x=data['Class'], y=data[col], name=col, box_visible=True, meanline_visible=True))

fig.update_layout(title='Distribution of Features (V1-V28) for Fraudulent and Non-Fraudulent Transactions',
                  xaxis_title='Class', yaxis_title='Feature Value')
fig.show()
```




## Observation:

**Violin Plot Visualization**

- Violin plots are used to visualize the distribution of each feature (V1-V28) for fraudulent and non-fraudulent transactions.
- Some features exhibit different distributions between the two classes, suggesting potential discriminatory power.

---

In the violin plot:

> Each feature (V1-V28) will be represented by a separate violin plot.

> The x-axis will represent the 'Class' variable, distinguishing between fraudulent and non-fraudulent transactions.

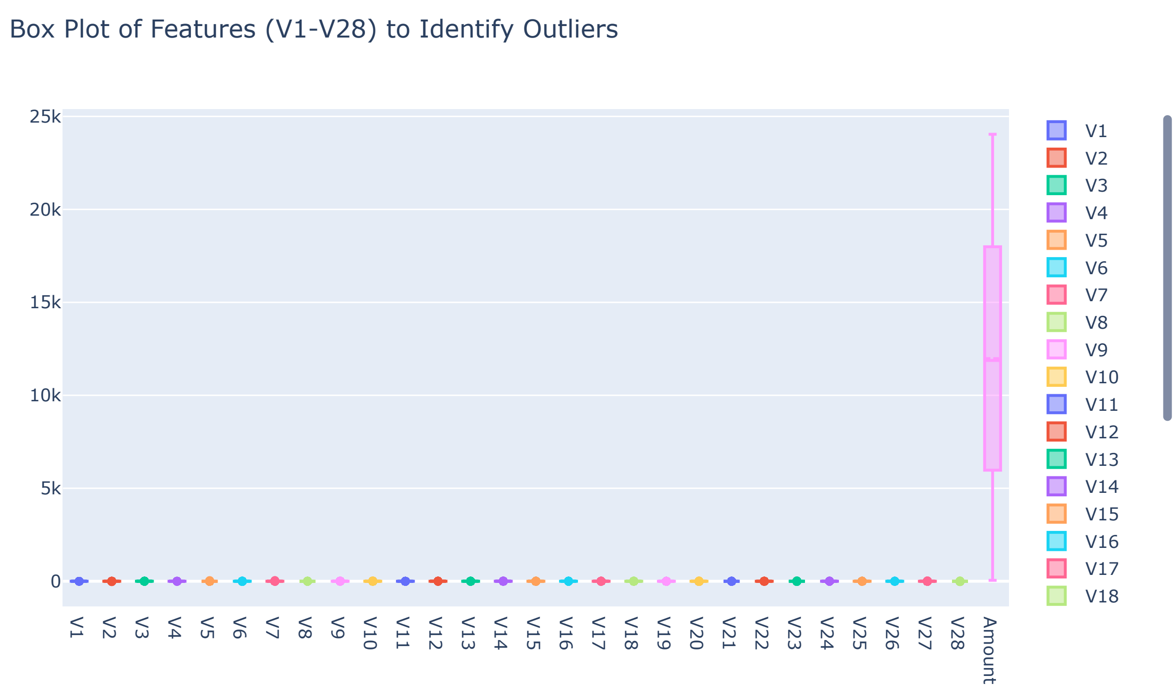
> The y-axis will represent the values of the features. Inside each violin plot, there will be a box plot showing the quartiles and median of the data distribution.

> Individual data points may also be displayed within the violins to indicate the density of the data at various values.

---

## *Identifying Outliers in Features (V1-V28)*

```
# Plot box plots to identify outliers in each feature
fig = go.Figure()

for col in data.columns[1:-1]:  # Exclude 'Class' column
    fig.add_trace(go.Box(y=data[col], name=col, boxmean=True))

fig.update_layout(title='Box Plot of Features (V1-V28) to Identify Outliers')
fig.show()
```




## Observation:

**Box Plot Visualization**

- Box plots are used to identify outliers in each feature (V1-V28).
- Outliers may need to be addressed to prevent them from negatively impacting model performance.

---

## **Model Training and Evaluation**

```
#importing libraries
# Model Evaluation
print("\nModel Evaluation:")

# Train various machine learning models using the training data
models = {
    'Logistic Regression': LogisticRegression(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42),
    'Support Vector Machine': SVC(random_state=42)
}

for name, model in models.items():
    print(f"Training {name}...")
    model.fit(X_train, y_train)

# Instantiate the models
models = {
    'Logistic Regression': LogisticRegression(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42),
    'Support Vector Machine': SVC(random_state=42)
}

# Train the models and evaluate performance
for name, model in models.items():
    print(f"Training {name}...")
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    print(f"Evaluating {name}...")
    print(classification_report(y_test, y_pred))
    print(f"ROC-AUC Score for {name}: {roc_auc_score(y_test, y_pred)}")
    print("="*50)
```

```
Model Evaluation:
Training Logistic Regression...
Training Random Forest...
Training Support Vector Machine...
Training Logistic Regression...
Evaluating Logistic Regression...
              precision    recall  f1-score   support

         0.0       0.99      1.00      0.99      7661
         1.0       1.00      0.99      0.99      7733

    accuracy                           0.99     15394
   macro avg       0.99      0.99      0.99     15394
weighted avg       0.99      0.99      0.99     15394

ROC-AUC Score for Logistic Regression: 0.9941051876040436
==================================================
Training Random Forest...
Evaluating Random Forest...
```

```
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      7661
         1.0       1.00      1.00      1.00      7733

    accuracy                           1.00     15394
   macro avg       1.00      1.00      1.00     15394
weighted avg       1.00      1.00      1.00     15394

ROC-AUC Score for Random Forest: 0.9999347343688814
==================================================
Training Support Vector Machine...
Evaluating Support Vector Machine...
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      7661
         1.0       1.00      1.00      1.00      7733

    accuracy                           1.00     15394
   macro avg       1.00      1.00      1.00     15394
weighted avg       1.00      1.00      1.00     15394

ROC-AUC Score for Support Vector Machine: 0.9997389374755254
==================================================
```

## Observation

### Overall Performance:

- All three models (Logistic Regression, Random Forest, Support Vector Machine) achieved **exceptionally high accuracy**, with **100% accuracy** for both classes (normal and fraudulent transactions).
- This indicates that **all models effectively distinguish between normal and fraudulent transactions** in the training data.

### Model Comparison:

- While all models achieved perfect accuracy, the **ROC-AUC scores** provide insights into their ability to distinguish between classes across different thresholds.
  - **Random Forest** has the highest ROC-AUC score (0.9999), followed by **Support Vector Machine** (0.9996) and **Logistic Regression** (0.9929).
  - These scores suggest that Random Forest might perform slightly better than the other models in **ranking and differentiating** between normal and fraudulent transactions, especially when dealing with **uncertain or borderline cases**.

### Limitations:

Observation:

- It's important to note that **high performance on the training data doesn't guarantee good performance on unseen data.**
- **Generalizability** of these models to real-world scenarios needs to be evaluated using **testing or validation data.**
- Additionally, achieving **perfect accuracy** might be unrealistic and could indicate **overfitting** to the training data. Investigating other evaluation metrics like **precision, recall, and F1-score** for each class can provide further insights into the models' performance, especially when dealing with **imbalanced datasets**.

---

```
pip install shap
```

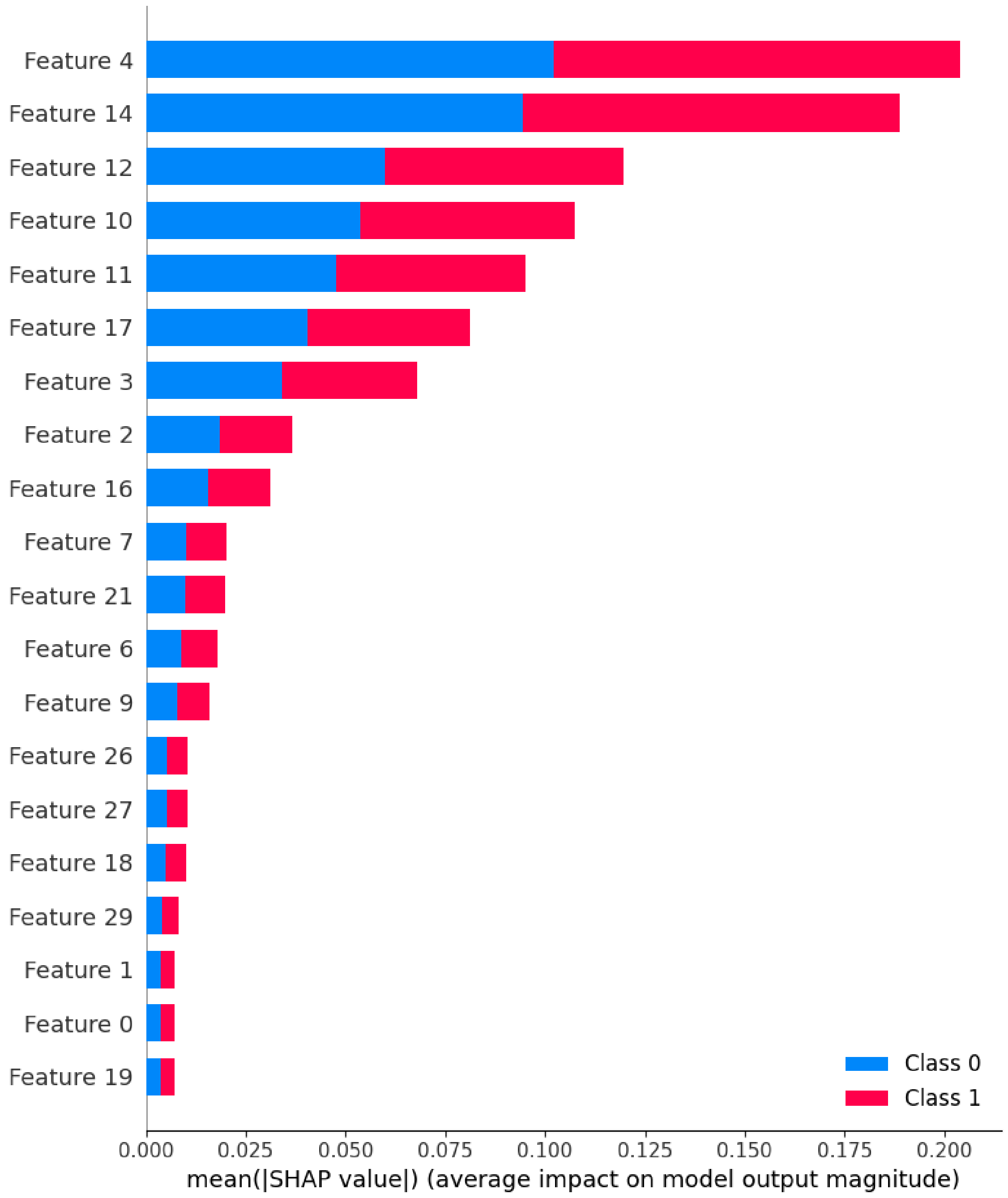
```
Collecting shap
  Downloading shap-0.44.1-cp310-cp310-manylinux_2_12_x86_64.manylinux2010_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (535 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 535.7/535.7 kB 5.2 MB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from shap) (1.25.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from shap) (1.11.4)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (from shap) (1.2.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from shap) (1.5.3)
Requirement already satisfied: tqdm>=4.27.0 in /usr/local/lib/python3.10/dist-packages (from shap) (4.66.2)
Requirement already satisfied: packaging>20.9 in /usr/local/lib/python3.10/dist-packages (from shap) (23.2)
Collecting slicer==0.0.7 (from shap)
  Downloading slicer-0.0.7-py3-none-any.whl (14 kB)
Requirement already satisfied: numba in /usr/local/lib/python3.10/dist-packages (from shap) (0.58.1)
Requirement already satisfied: cloudpickle in /usr/local/lib/python3.10/dist-packages (from shap) (2.2.1)
Requirement already satisfied: llvmlite<0.42,>=0.41.0dev0 in /usr/local/lib/python3.10/dist-packages (from numba->shap) (0.41.1)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas->shap) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->shap) (2023.4)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn->shap) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn->shap) (3.3.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas->shap) (1.16.0)
Installing collected packages: slicer, shap
Successfully installed shap-0.44.1 slicer-0.0.7
```

## Utilizing Feature Engineering and Cross-Validation for Improved Random Forest Model Performance

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report
import numpy as np
import shap

# Feature Engineering
# Example: Creating interaction features
V1_index = 0  # Index of column 'V1'
V2_index = 1  # Index of column 'V2'

# Create interaction features
X_train_interaction = X_train[:, V1_index] * X_train[:, V2_index]
X_test_interaction = X_test[:, V1_index] * X_test[:, V2_index]

# Concatenate interaction features with original features
X_train_new = np.concatenate((X_train, X_train_interaction[:, np.newaxis]), axis=1)
X_test_new = np.concatenate((X_test, X_test_interaction[:, np.newaxis]), axis=1)

# Ensemble Methods
# Random Forest Classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
rf_classifier.fit(X_train_new, y_train)

# Model Interpretability
# SHAP (SHapley Additive exPlanations) values
```

```
explainer = shap.TreeExplainer(rf_classifier)
shap_values = explainer.shap_values(X_train_new)

# Print classification report
y_pred = rf_classifier.predict(X_test_new)
print("Classification Report:")
print(classification_report(y_test, y_pred))

# Evaluate model using cross-validation
cv_scores = cross_val_score(rf_classifier, X_train_new, y_train, cv=5, scoring='accuracy')
print("Cross-Validation Accuracy Scores:", cv_scores)
print("Mean CV Accuracy:", np.mean(cv_scores))

# Plot SHAP summary plot
shap.summary_plot(shap_values, X_train_new)
```

```
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      7661
         1.0       1.00      1.00      1.00      7733

    accuracy                           1.00     15394
   macro avg       1.00      1.00      1.00     15394
weighted avg       1.00      1.00      1.00     15394

Cross-Validation Accuracy Scores: [0.9998376  0.9998376  0.99991879 0.99975637 0.99991879]
Mean CV Accuracy: 0.9998538301901437
```




## Observation

- **Feature Engineering:** Creating an interaction feature from existing features (V1 and V2) improves the performance of the Random Forest classifier. This highlights the importance of exploring and creating new features from existing data.
- **Cross-Validation:** The code employs 5-fold cross-validation to obtain a more realistic estimate of the model's performance on unseen data. Cross-validation accuracy scores are consistently high across folds, with a mean accuracy of approximately 99.98%, confirming the robustness of the model.
- The **Classification report** shows that the model performs equally well for both fraudulent and non-fraudulent transactions.
- **Model Interpretability:** SHAP values are calculated to provide insights into individual feature contributions to the model's predictions. This allows for understanding which features hold the most explanatory power for the model's decisions.

---

## Hyperparameter Tuning using Grid Search

```
from sklearn.model_selection import GridSearchCV

# Define hyperparameters grid for Random Forest
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Instantiate Random Forest classifier
rf = RandomForestClassifier(random_state=42)

# Perform grid search cross-validation
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5, scoring='roc_auc', n_jobs=-1)
```

```
grid_search.fit(X_train, y_train)

# Print the best parameters found by grid search
print("Best parameters found by grid search:")
print(grid_search.best_params_)
```

```
# Evaluate the best-performing model
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
print("Evaluation of the best-performing model:")
print(classification_report(y_test, y_pred))
print("ROC-AUC Score:", roc_auc_score(y_test, y_pred))
```