Credit Card Fraud Detection

Objective

The goal of this project is to develop a machine learning model that can accurately detect fraudulent credit card transactions using historical data. By analyzing transaction patterns, the model should be able to distinguish between normal and fraudulent activity, helping financial institutions flag suspicious behavior early and reduce potential risks.

Step 1: Importing necessary Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Step 2: Loading the Data & Cleaning

- Loading dataset into a pandas DataFrame
- About Dataset:
  - Time: This shows how many seconds have passed since the first transaction in the dataset.
  - V1-V28: These are special features created to hide sensitive information about the original data.
  - Amount: Transaction amount.
  - Class: Target variable (0 for normal transactions, 1 for fraudulent transactions).

```
data = pd.read_csv('/content/creditcard.csv')
data.head(5)
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	
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.
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.
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.
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.
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.

5 rows × 31 columns

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19898 entries, 0 to 19897
Data columns (total 31 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Time    19898 non-null  int64  
 1   V1      19898 non-null  float64
 2   V2      19898 non-null  float64
 3   V3      19898 non-null  float64
 4   V4      19898 non-null  float64
 5   V5      19898 non-null  float64
 6   V6      19898 non-null  float64
 7   V7      19898 non-null  float64
 8   V8      19898 non-null  float64
 9   V9      19898 non-null  float64
 10  V10     19898 non-null  float64
 11  V11     19897 non-null  float64
 12  V12     19897 non-null  float64
 13  V13     19897 non-null  float64
 14  V14     19897 non-null  float64
 15  V15     19897 non-null  float64
 16  V16     19897 non-null  float64
 17  V17     19897 non-null  float64
 18  V18     19897 non-null  float64
```

```
 19  V19     19897 non-null  float64
 20  V20     19897 non-null  float64
 21  V21     19897 non-null  float64
 22  V22     19897 non-null  float64
 23  V23     19897 non-null  float64
 24  V24     19897 non-null  float64
 25  V25     19897 non-null  float64
 26  V26     19897 non-null  float64
 27  V27     19897 non-null  float64
 28  V28     19897 non-null  float64
 29  Amount  19897 non-null  float64
 30  Class   19897 non-null  float64
dtypes: float64(30), int64(1)
memory usage: 4.7 MB
```

```
data.describe().T
```

	count	mean	std	min	25%	50%	75%	max
Time	19898.0	15492.416374	10512.066686	0.000000	4536.250000	14796.000000	26220.500000	30633.000000
V1	19898.0	-0.244326	1.889986	-30.552380	-0.959632	-0.302521	1.164473	1.960497
V2	19898.0	0.242420	1.527342	-40.978852	-0.329008	0.220079	0.870117	16.713389
V3	19898.0	0.745774	1.767726	-31.103685	0.309235	0.898672	1.532922	4.101716
V4	19898.0	0.277011	1.466218	-5.172595	-0.636713	0.224608	1.142143	11.927512
V5	19898.0	-0.163264	1.430821	-32.092129	-0.745156	-0.199377	0.341367	34.099309
V6	19898.0	0.092881	1.331029	-23.496714	-0.657306	-0.175434	0.486735	21.393069
V7	19898.0	-0.145279	1.338260	-26.548144	-0.599403	-0.072254	0.448572	34.303177
V8	19898.0	0.022237	1.346813	-41.484823	-0.171779	0.023822	0.279960	20.007208
V9	19898.0	0.636382	1.278839	-7.175097	-0.209565	0.620176	1.409142	10.392889
V10	19898.0	-0.220463	1.219491	-14.166795	-0.679812	-0.276776	0.228545	12.701539
V11	19897.0	0.682722	1.189811	-2.767470	-0.145385	0.656472	1.451248	12.018913
V12	19897.0	-1.088832	1.578414	-17.769143	-2.238439	-1.173521	0.198628	4.846452
V13	19897.0	0.682166	1.200808	-3.588761	-0.172367	0.682817	1.595714	4.465413
V14	19897.0	0.556135	1.340297	-19.214325	-0.052743	0.622956	1.408635	7.692209
V15	19897.0	-0.044228	0.975187	-4.152532	-0.609302	0.088551	0.627632	3.635042
V16	19897.0	-0.004940	0.965384	-12.227189	-0.487703	0.066038	0.557233	4.816252
V17	19897.0	0.289847	1.240287	-18.587366	-0.208683	0.267648	0.768742	9.253526
V18	19897.0	-0.045159	0.857511	-8.061208	-0.501786	-0.011383	0.457270	4.295648
V19	19897.0	-0.065826	0.820375	-4.932733	-0.551305	-0.070756	0.441557	4.555359
V20	19897.0	0.038985	0.630311	-13.276034	-0.158109	-0.028316	0.152608	15.815051
V21	19897.0	-0.047949	0.828385	-20.262054	-0.259497	-0.115398	0.049521	22.614889
V22	19897.0	-0.146461	0.637567	-8.593642	-0.563992	-0.118803	0.254057	5.805795
V23	19897.0	-0.038093	0.520683	-26.751119	-0.174213	-0.046994	0.073666	13.876221
V24	19897.0	0.010678	0.591180	-2.728650	-0.333314	0.061181	0.398549	3.695503
V25	19897.0	0.122924	0.437675	-7.495741	-0.138325	0.160495	0.400713	5.525093
V26	19897.0	0.033127	0.530315	-1.338556	-0.341917	-0.036546	0.332729	3.517346
V27	19897.0	0.014454	0.393002	-8.567638	-0.069146	0.003868	0.096281	8.254376
V28	19897.0	0.007312	0.244372	-3.575312	-0.010879	0.019083	0.077551	4.860769
Amount	19897.0	70.271100	205.363789	0.000000	5.750000	16.000000	59.980000	7879.420000
Class	19897.0	0.004272	0.065222	0.000000	0.000000	0.000000	0.000000	1.000000

```
null_count = data.isnull().sum()
print(null_count)
```

```
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
```

```
duplicates_record = data.duplicated().sum()
print(duplicates_record)
```

```
1081
```

```
data.shape
```

```
(284807, 31)
```

```
data = data.drop_duplicates()
data.shape
```

```
(283726, 31)
```

Step 3: Exploratory Data Analysis

```
# Calculating the ratio of fraud cases to valid cases to understand how balanced or imbalanced the dataset
fraud_transactions = data[data['Class'] == 1] # Fraudulent transactions (Class == 1)
valid_transactions = data[data['Class'] == 0] # Valid transactions (Class == 0)
outlier_fraction = len(fraud_transactions) / float(len(valid_transactions))
print(outlier_fraction)
print("fraud_transactions: {}".format(len(data[data['Class'] == 1])))
print("valid_transactions: {}".format(len(data[data['Class'] == 0])))
```

```
0.0016698852262818046
fraud_transactions: 473
valid_transactions: 283253
```

```
# Exploring Transaction Amounts:
# Help us understand if there are any significant differences in the monetary value of fraudulent transactions.
fraud_transactions.Amount.describe()
```

	Amount
count	473.000000
mean	123.871860
std	260.211041
min	0.000000
25%	1.000000
50%	9.820000
75%	105.890000
max	2125.870000

dtype: float64

```
valid_transactions.Amount.describe()
```

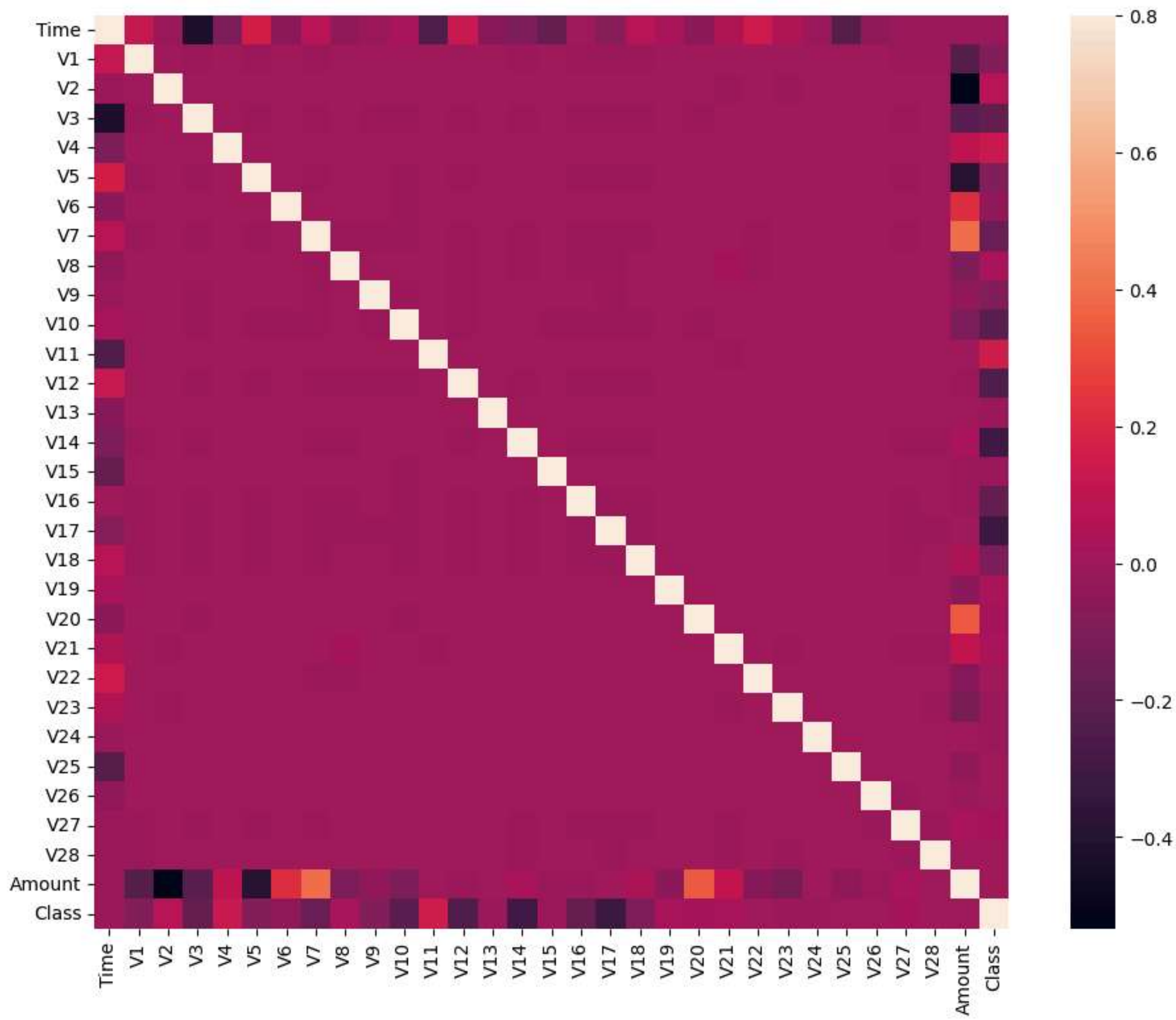
	Amount
count	283253.000000
mean	88.413575
std	250.379023
min	0.000000
25%	5.670000
50%	22.000000
75%	77.460000
max	25691.160000

dtype: float64

Correlation Matrix

The correlation between features using a heatmap using correlation matrix. Help us understanding of how the different features are correlated and which ones may be more relevant for prediction.

```
corrmat = data.corr()
fig = plt.figure(figsize = (12, 9))
sns.heatmap(corrmat, vmax = .8, square = True)
plt.show()
"""Most features do not correlate strongly with others but some features like V2 and V5 have a negative correlation with the
Amount feature. This provides valuable insights into how the features are related to the transaction amounts."""
```




```
'Most features do not correlate strongly with others but some features like V2 and V5 have a negative correlation with the \nA
mount feature. This provides valuable insights into how the features are related to the transaction amounts.'
```

Step 4: Preparing Data

Separate the input features (X) and target variable (Y) then split the data into training and testing sets

- X = data.drop(['Class'], axis = 1) removes the target column (Class) from the dataset to keep only the input features.
- Y = data["Class"] selects the Class column as the target variable (fraud or not).
- X.shape and Y.shape print the number of rows and columns in the feature set and the target set.
- xData = X.values and yData = Y.values convert the Pandas DataFrame or Series to NumPy arrays for faster processing.
- train_test_split(...) splits the data into training and testing sets into 80% for training, 20% for testing.
- random_state=42 ensures reproducibility (same split every time you run it).

```
X = data.drop(['Class'], axis = 1)
Y = data["Class"]
print(X.shape)
print(Y.shape)

xData = X.values
yData = Y.values

from sklearn.model_selection import train_test_split
xTrain, xTest, yTrain, yTest = train_test_split(
        xData, yData, test_size = 0.2, random_state = 42)
```

```
(283726, 30)
(283726,)
```

Step 5: Building and Training the Model

Train a Random Forest Classifier to predict fraudulent transactions.

- from sklearn.ensemble import RandomForestClassifier: This imports the RandomForestClassifier from sklearn.ensemble, which is used to create a random forest model for classification tasks.
- rfc = RandomForestClassifier(): Initializes a new instance of the RandomForestClassifier.
- rfc.fit(xTrain, yTrain): Trains the RandomForestClassifier model on the training data (xTrain for features and yTrain for the target labels).
- yPred = rfc.predict(xTest): Uses the trained model to predict the target labels for the test data (xTest), storing the results in yPred.

```
from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier()
rfc.fit(xTrain, yTrain)

yPred = rfc.predict(xTest)
```

Step 6: Evaluating the Model

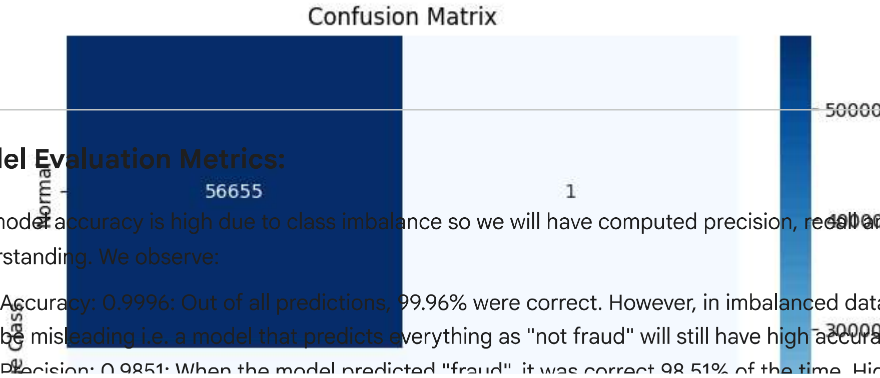
After training the model we need to evaluate its performance using various metrics such as accuracy, precision, recall, F1-score and the Matthews correlation coefficient.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, matthews_corrcoef, confusion_matrix
accuracy = accuracy_score(yTest, yPred)
precision = precision_score(yTest, yPred)
recall = recall_score(yTest, yPred)
f1 = f1_score(yTest, yPred)
mcc = matthews_corrcoef(yTest, yPred)

print("Model Evaluation Metrics:")
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")
print(f"Matthews Correlation Coefficient: {mcc:.4f}")

conf_matrix = confusion_matrix(yTest, yPred)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
            xticklabels=['Normal', 'Fraud'], yticklabels=['Normal', 'Fraud'])
plt.title("Confusion Matrix")
plt.xlabel("Predicted Class")
plt.ylabel("True Class")
plt.show()
```

```
Model Evaluation Metrics:
Accuracy: 0.9996
Precision: 0.9851
Recall: 0.7333
F1-Score: 0.8408
Matthews Correlation Coefficient: 0.8497
```



Model Evaluation Metrics:

The model accuracy is high due to class imbalance so we will have computed precision, recall and f1 score to get a more meaningful understanding. We observe:

1. Accuracy: 0.9996: Out of all predictions, 99.96% were correct. However, in imbalanced datasets (like fraud detection), accuracy can be misleading i.e. a model that predicts everything as "not fraud" will still have high accuracy.
2. Precision: 0.9851: When the model predicted "fraud", it was correct 98.51% of the time. High precision means very few false alarms