Anomaly Detection in Credit Card Transactions

In today's digital age, credit card fraud has become an increasingly prevalent and sophisticated threat. As millions of transactions occur daily, detecting fraudulent activities among them presents a significant challenge for financial institutions and consumers alike. The sheer volume of data makes it virtually impossible for human analysts to identify meaningful patterns or anomalies in transaction data manually. This is where the power of machine learning comes into play, offering a robust solution to this complex problem.

The Challenge of Credit Card Fraud

Credit card fraud is not just a financial inconvenience; it's a serious crime that can have far-reaching consequences for individuals, businesses, and the economy as a whole. Fraudsters are constantly evolving their techniques, making it crucial for detection methods to be equally dynamic and advanced. Traditional rule-based systems, while still useful, are often not sufficient to catch more subtle or novel forms of fraud.

Leveraging Machine Learning for Fraud Detection

Machine learning models have emerged as a powerful tool against credit card fraud. These models can analyze vast amounts of transaction data, learning to recognize patterns and anomalies that might be invisible to the human eye. By employing sophisticated algorithms, machine learning can adapt to new fraud patterns over time, providing a more flexible and effective approach to fraud detection.

Project Goal: Anomaly Detection in Financial Transactions

The primary objective of this project is to develop a machine learning model capable of detecting anomalous transactions within a large dataset of financial activities. An anomalous transaction is defined as one that significantly deviates from the typical behavior observed in the majority of transactions. By accurately identifying these outliers, we aim to flag potential fraudulent activities for further investigation.

Dataset: A Real-World Challenge

For this project, we're utilizing the Credit Card Fraud Detection dataset from Kaggle. This dataset provides a real-world scenario, containing transactions made by European credit card holders in September 2013. It comprises a total of 284,807 transactions, of which 492 are fraudulent.

One of the key challenges presented by this dataset is its highly imbalanced nature. Fraudulent transactions account for only 0.17% of the total transactions, reflecting the reality of fraud detection where legitimate transactions far outnumber fraudulent ones. This imbalance adds an extra layer of complexity to our task, requiring careful consideration in our modelling approach to ensure we can effectively identify the rare fraudulent cases without being overwhelmed by the volume of legitimate transactions.

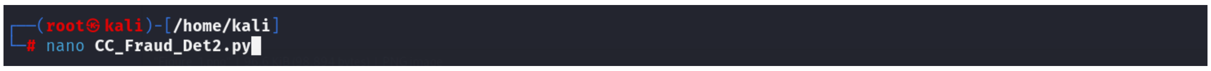
In the following sections, we will delve into the details of our approach, exploring the data, implementing various machine learning techniques, and evaluating their effectiveness in detecting credit card fraud. Through this process, we aim to contribute to the ongoing efforts to make financial transactions safer and more secure in our increasingly digital world.

Steps to Visualize the data and develop the Machine Learning Model: -

Begin by downloading the dataset from Kaggle https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud?resource=download

Open the nano editor with the desired file name:

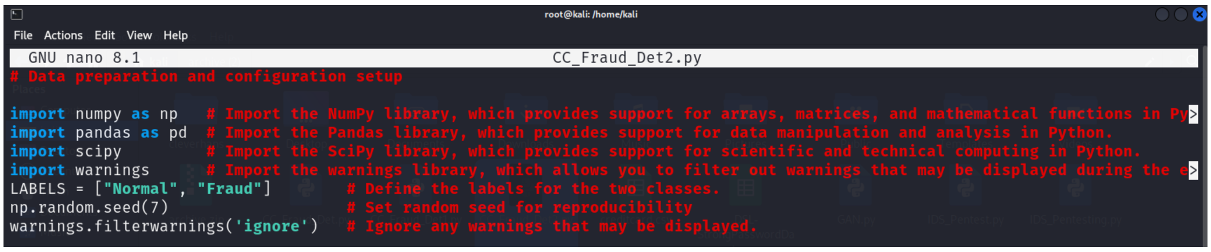
$ nano CC_Fraud_Det2.py



```
┌──(root㉿kali)-[/home/kali]
└─# nano CC_Fraud_Det2.py
```

Begin writing the code. It can be obtained from https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/Capstone%20Projects/CC_Fraud_Det2.py

Step 1 - Importing Libraries and Setting Up



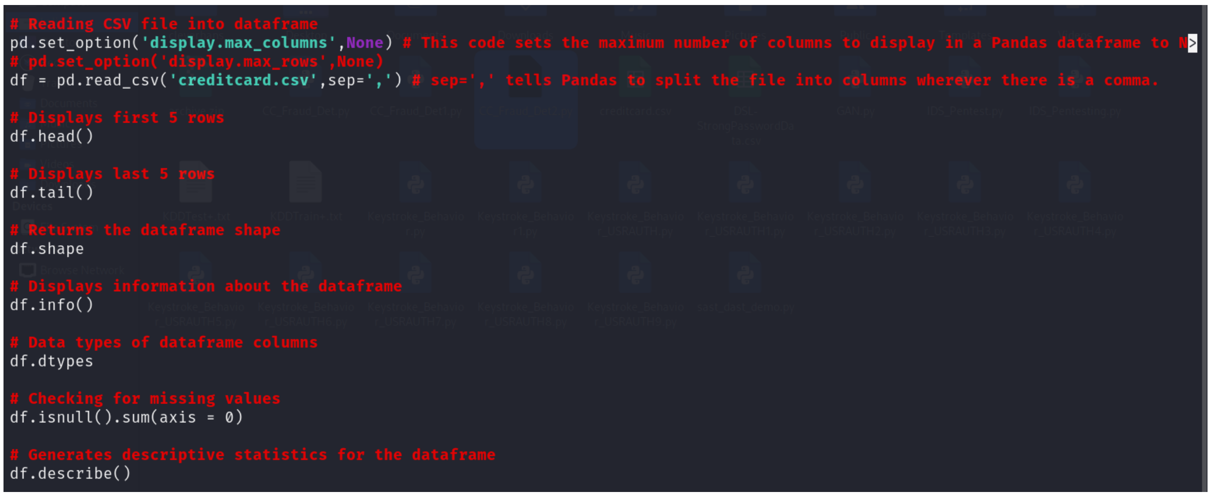
```
  GNU nano 8.1                                CC_Fraud_Det2.py
# Data preparation and configuration setup

import numpy as np   # Import the NumPy library, which provides support for arrays, matrices, and mathematical functions in Py>
import pandas as pd  # Import the Pandas library, which provides support for data manipulation and analysis in Python.
import scipy         # Import the SciPy library, which provides support for scientific and technical computing in Python.
import warnings      # Import the warnings library, which allows you to filter out warnings that may be displayed during the e>
LABELS = ["Normal", "Fraud"]           # Define the labels for the two classes.
np.random.seed(7)                      # Set random seed for reproducibility
warnings.filterwarnings('ignore')      # Ignore any warnings that may be displayed.
```

This step imports necessary Python libraries:

- numpy (imported as np): For numerical operations and array handling.
- pandas (imported as pd): For data manipulation and analysis.
- scipy: For scientific computing (though it's not used explicitly in the rest of the code).
- warnings: To suppress warning messages.
- It also sets up some initial configurations:

- LABELS: A list defining the two classes we're dealing with: "Normal" and "Fraud".
- np.random.seed(7): Sets a fixed random seed for reproducibility.
- warnings.filterwarnings('ignore'): Suppresses warning messages.

Step 2 - Loading and Exploring the Dataset

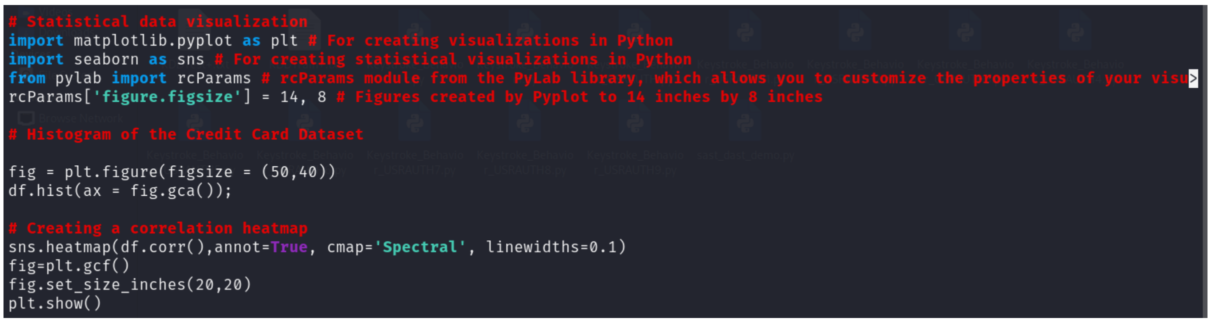


```
# Reading CSV file into dataframe
pd.set_option('display.max_columns',None) # This code sets the maximum number of columns to display in a Pandas dataframe to N>
# pd.set_option('display.max_rows',None)
df = pd.read_csv('creditcard.csv',sep=',') # sep=',' tells Pandas to split the file into columns wherever there is a comma.

# Displays first 5 rows
df.head()

# Displays last 5 rows
df.tail()

# Returns the dataframe shape
df.shape

# Displays information about the dataframe
df.info()

# Data types of dataframe columns
df.dtypes

# Checking for missing values
df.isnull().sum(axis = 0)

# Generates descriptive statistics for the dataframe
df.describe()
```

This step loads the dataset and performs initial exploratory data analysis:

- pd.set_option('display.max_columns', None): Ensures all columns are displayed when printing DataFrames.
- df = pd.read_csv('creditcard.csv', sep=','): Loads the credit card dataset from a CSV file into a pandas DataFrame.
- df.head() and df.tail(): Display the first and last 5 rows of the DataFrame.
- df.shape: Shows the dimensions of the DataFrame (number of rows and columns).
- df.info(): Provides a concise summary of the DataFrame, including column names and data types.
- df.dtypes: Shows the data type of each column.
- df.isnull().sum(axis = 0): Checks for missing values in each column.
- df.describe(): Generates descriptive statistics of the DataFrame.

Step 3 - Data Visualization

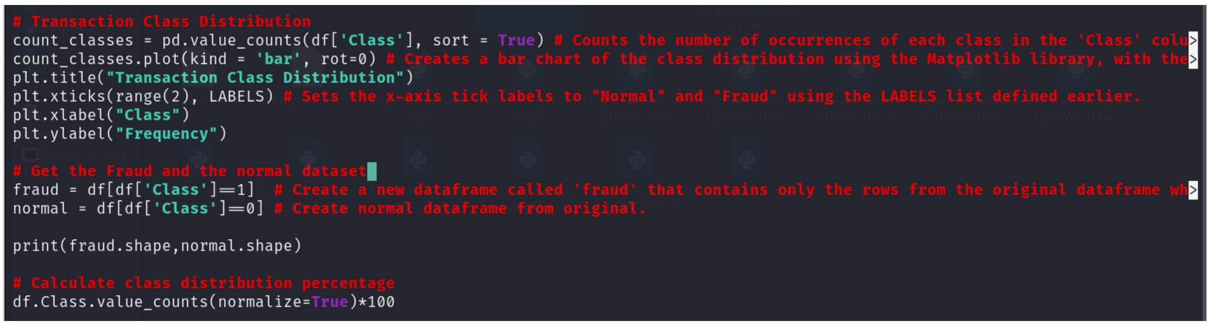


```
# Statistical data visualization
import matplotlib.pyplot as plt # For creating visualizations in Python
import seaborn as sns # For creating statistical visualizations in Python
from pylab import rcParams # rcParams module from the PyLab library, which allows you to customize the properties of your visu>
rcParams['figure.figsize'] = 14, 8 # Figures created by Pyplot to 14 inches by 8 inches

# Histogram of the Credit Card Dataset

fig = plt.figure(figsize = (50,40))
df.hist(ax = fig.gca());

# Creating a correlation heatmap
sns.heatmap(df.corr(),annot=True, cmap='Spectral', linewidths=0.1)
fig=plt.gcf()
fig.set_size_inches(20,20)
plt.show()
```

This section creates visualizations to better understand the data:

- It imports matplotlib.pyplot and seaborn for plotting.
- rcParams['figure.figsize'] = 14, 8: Sets the default figure size.
- df.hist(ax = fig.gca()): Creates histograms for all numerical columns in the DataFrame.
- sns.heatmap(df.corr(), ...): Generates a correlation heatmap to visualize relationships between features.

Step 4 - Analyzing Class Distribution

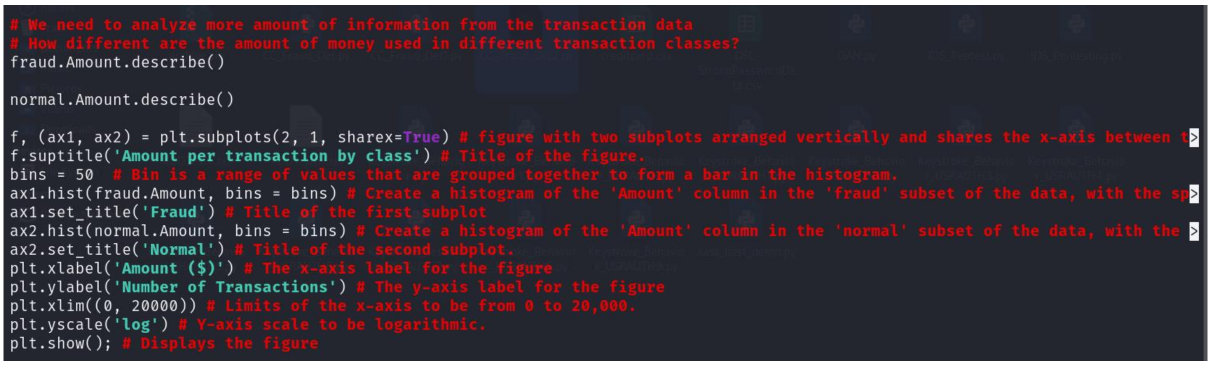


```
# Transaction Class Distribution
count_classes = pd.value_counts(df['Class'], sort = True) # Counts the number of occurrences of each class in the 'Class' colu>
count_classes.plot(kind = 'bar', rot=0) # Creates a bar chart of the class distribution using the Matplotlib library, with the>
plt.title("Transaction Class Distribution")
plt.xticks(range(2), LABELS) # Sets the x-axis tick labels to "Normal" and "Fraud" using the LABELS list defined earlier.
plt.xlabel("Class")
plt.ylabel("Frequency")

# Get the Fraud and the normal dataset
fraud = df[df['Class']==1]  # Create a new dataframe called 'fraud' that contains only the rows from the original dataframe wh>
normal = df[df['Class']==0] # Create normal dataframe from original.

print(fraud.shape,normal.shape)

# Calculate class distribution percentage
df.Class.value_counts(normalize=True)*100
```

This part analyzes the distribution of fraudulent and normal transactions:

- It creates a bar plot showing the frequency of each class.
- It separates the DataFrame into two: fraud (Class = 1) and normal (Class = 0).
- It prints the shapes of these new DataFrames and calculates the percentage of each class.

Step 5 - Analyzing Transaction Amounts

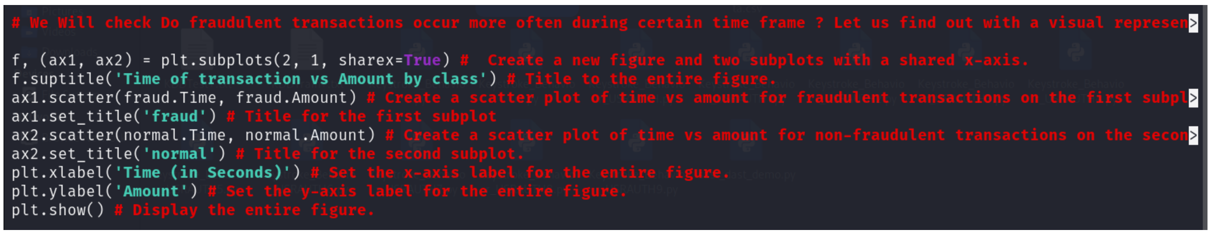


```
# We need to analyze more amount of information from the transaction data
# How different are the amount of money used in different transaction classes?
fraud.Amount.describe()

normal.Amount.describe()

f, (ax1, ax2) = plt.subplots(2, 1, sharex=True) # figure with two subplots arranged vertically and shares the x-axis between t>
f.suptitle('Amount per transaction by class') # Title of the figure.
bins = 50  # Bin is a range of values that are grouped together to form a bar in the histogram.
ax1.hist(fraud.Amount, bins = bins) # Create a histogram of the 'Amount' column in the 'fraud' subset of the data, with the sp>
ax1.set_title('Fraud') # Title of the first subplot
ax2.hist(normal.Amount, bins = bins) # Create a histogram of the 'Amount' column in the 'normal' subset of the data, with the >
ax2.set_title('Normal') # Title of the second subplot.
plt.xlabel('Amount ($)') # The x-axis label for the figure
plt.ylabel('Number of Transactions') # The y-axis label for the figure
plt.xlim((0, 20000)) # Limits of the x-axis to be from 0 to 20,000.
plt.yscale('log') # Y-axis scale to be logarithmic.
plt.show(); # Displays the figure
```

This section analyzes the transaction amounts:

- It provides descriptive statistics for amounts in both fraudulent and normal transactions.
- It creates histograms to visualize the distribution of transaction amounts for both classes.

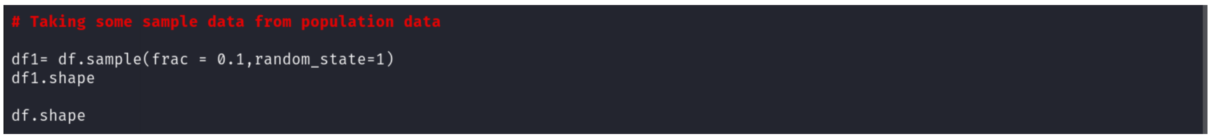
Step 6 - Analyzing Transaction Time



```
# We Will check Do fraudulent transactions occur more often during certain time frame ? Let us find out with a visual represen>

f, (ax1, ax2) = plt.subplots(2, 1, sharex=True) #  Create a new figure and two subplots with a shared x-axis.
f.suptitle('Time of transaction vs Amount by class') # Title to the entire figure.
ax1.scatter(fraud.Time, fraud.Amount) # Create a scatter plot of time vs amount for fraudulent transactions on the first subpl>
ax1.set_title('fraud') # Title for the first subplot
ax2.scatter(normal.Time, normal.Amount) # Create a scatter plot of time vs amount for non-fraudulent transactions on the secon>
ax2.set_title('normal') # Title for the second subplot.
plt.xlabel('Time (in Seconds)') # Set the x-axis label for the entire figure.
plt.ylabel('Amount') # Set the y-axis label for the entire figure.
plt.show() # Display the entire figure.
```

This part creates scatter plots to visualize the relationship between transaction time and amount for both classes.

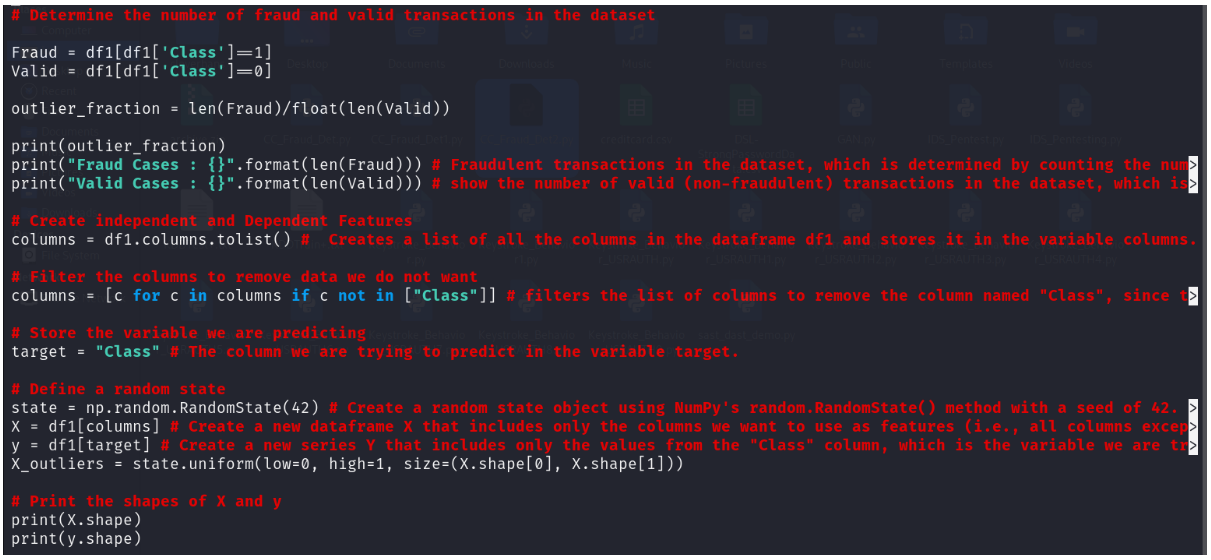
Step 7 - Sampling the Dataset



```
# Taking some sample data from population data

df1= df.sample(frac = 0.1,random_state=1)
df1.shape

df.shape
```

This step creates a smaller sample (10%) of the original dataset for faster processing.

Step 8 - Preparing Features and Target Variable



```
# Determine the number of fraud and valid transactions in the dataset

Fraud = df1[df1['Class']==1]
Valid = df1[df1['Class']==0]

outlier_fraction = len(Fraud)/float(len(Valid))

print(outlier_fraction)
print("Fraud Cases : {}".format(len(Fraud))) # Fraudulent transactions in the dataset, which is determined by counting the num>
print("Valid Cases : {}".format(len(Valid))) # show the number of valid (non-fraudulent) transactions in the dataset, which is>

# Create independent and Dependent Features
columns = df1.columns.tolist() #  Creates a list of all the columns in the dataframe df1 and stores it in the variable columns.

# Filter the columns to remove data we do not want
columns = [c for c in columns if c not in ["Class"]] # filters the list of columns to remove the column named "Class", since t>

# Store the variable we are predicting
target = "Class" # The column we are trying to predict in the variable target.

# Define a random state
state = np.random.RandomState(42) # Create a random state object using NumPy's random.RandomState() method with a seed of 42. >
X = df1[columns] # Create a new dataframe X that includes only the columns we want to use as features (i.e., all columns excep>
y = df1[target] # Create a new series Y that includes only the values from the "Class" column, which is the variable we are tr>
X_outliers = state.uniform(low=0, high=1, size=(X.shape[0], X.shape[1]))

# Print the shapes of X and y
print(X.shape)
print(y.shape)
```

This step is crucial for preparing the data for machine learning. The following takes place in the code snippet above:

Separating Fraud and Valid Transactions:

- Fraud = df1[df1['Class']==1]: This creates a DataFrame containing only the fraudulent transactions (where 'Class' is 1).
- Valid = df1[df1['Class']==0]: This creates a DataFrame containing only the valid transactions (where 'Class' is 0).

Calculating the Outlier Fraction:

outlier_fraction = len(Fraud)/float(len(Valid)): This calculates the ratio of fraudulent transactions to valid transactions. This fraction is important for some anomaly detection algorithms to understand the expected proportion of anomalies.

Printing Statistics: The code prints the outlier fraction and the number of fraud and valid cases, which gives us an idea of the class imbalance in the dataset.

Preparing Features and Target:

- columns = df1.columns.tolist(): This creates a list of all column names in the DataFrame.

- columns = [c for c in columns if c not in ["Class"]]: This removes the 'Class' column from the list of features, as it's our target variable.
- target = "Class": This specifies 'Class' as our target variable.

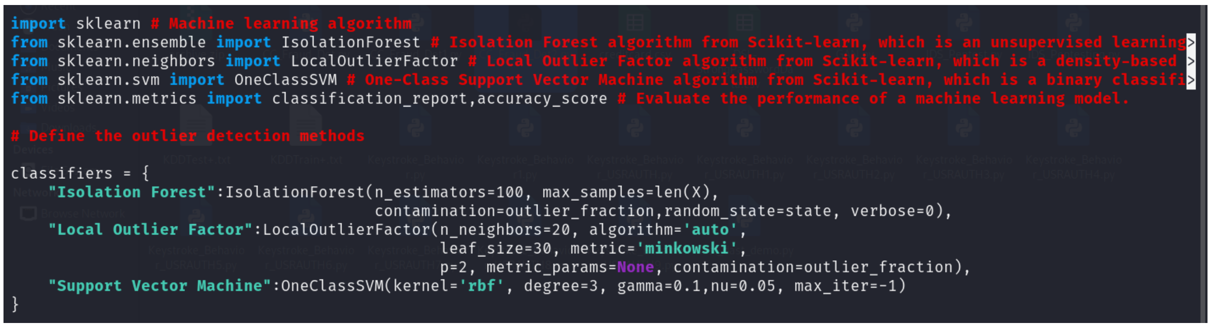
Creating Feature Matrix and Target Vector:

- X = df1[columns]: This creates our feature matrix X, containing all columns except 'Class'.
- y = df1[target]: This creates our target vector y, containing only the 'Class' column.

Printing Shapes:

The code prints the shapes of X and y to confirm they have been split correctly.

Step 9 - Implementing Anomaly Detection Algorithms



```
import sklearn # Machine learning algorithm
from sklearn.ensemble import IsolationForest # Isolation Forest algorithm from Scikit-learn, which is an unsupervised learning>
from sklearn.neighbors import LocalOutlierFactor # Local Outlier Factor algorithm from Scikit-learn, which is a density-based >
from sklearn.svm import OneClassSVM # One-Class Support Vector Machine algorithm from Scikit-learn, which is a binary classifi>
from sklearn.metrics import classification_report,accuracy_score # Evaluate the performance of a machine learning model.

# Define the outlier detection methods

classifiers = {
    "Isolation Forest":IsolationForest(n_estimators=100, max_samples=len(X),
                                       contamination=outlier_fraction,random_state=state, verbose=0),
    "Local Outlier Factor":LocalOutlierFactor(n_neighbors=20, algorithm='auto',
                                              leaf_size=30, metric='minkowski',
                                              p=2, metric_params=None, contamination=outlier_fraction),
    "Support Vector Machine":OneClassSVM(kernel='rbf', degree=3, gamma=0.1,nu=0.05, max_iter=-1)
}
```

This step sets up three different anomaly detection algorithms:

Isolation Forest:

This algorithm isolates anomalies by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

Parameters:

- n_estimators=100: The number of base estimators in the ensemble.

- max_samples=len(X): The number of samples to draw from X to train each base estimator.
- contamination=outlier_fraction: The proportion of outliers in the data set.

Local Outlier Factor (LOF):

This algorithm computes a score reflecting the degree of abnormality of the observations. It measures the local deviation of density of a given sample with respect to its neighbors.

Parameters:

- n_neighbors=20: Number of neighbors to use by default for k-neighbors queries.
- contamination=outlier_fraction: The proportion of outliers in the data set.

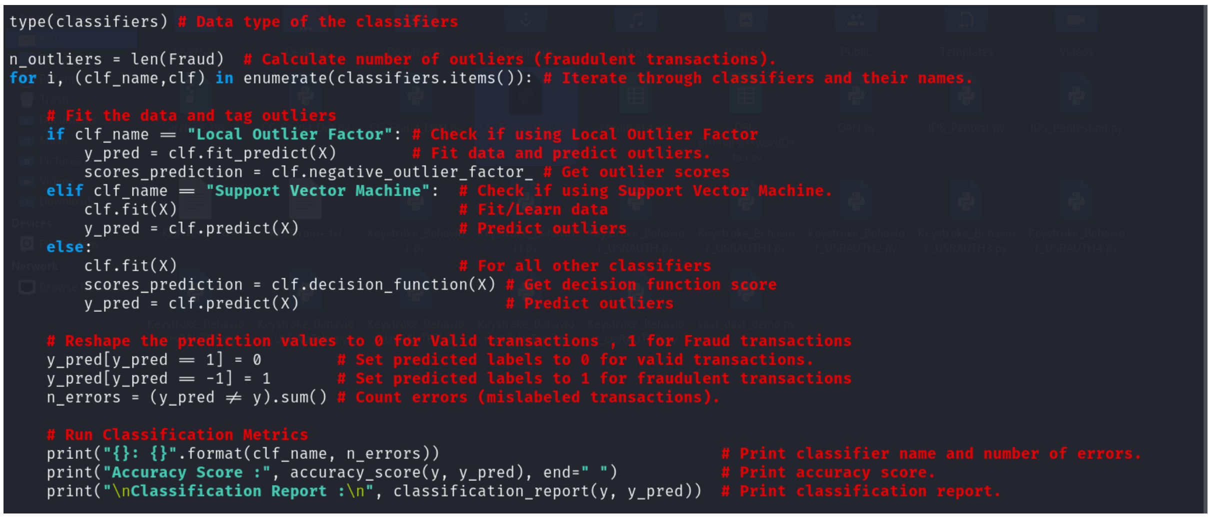
One-Class SVM:

This algorithm learns a boundary that encloses the majority of the data points. Points that fall outside this boundary are classified as anomalies.

Parameters:

- kernel='rbf': Specifies the kernel type to be used in the algorithm. RBF stands for Radial Basis Function.
- nu=0.05: An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors.

Step 10 - Evaluating the Algorithms



```python
type(classifiers) # Data type of the classifiers

n_outliers = len(Fraud)  # Calculate number of outliers (fraudulent transactions).
for i, (clf_name,clf) in enumerate(classifiers.items()): # Iterate through classifiers and their names.

    # Fit the data and tag outliers
    if clf_name == "Local Outlier Factor": # Check if using Local Outlier Factor
        y_pred = clf.fit_predict(X)        # Fit data and predict outliers.
        scores_prediction = clf.negative_outlier_factor_ # Get outlier scores
    elif clf_name == "Support Vector Machine":  # Check if using Support Vector Machine.
        clf.fit(X)                             # Fit/Learn data
        y_pred = clf.predict(X)                # Predict outliers
    else:
        clf.fit(X)                             # For all other classifiers
        scores_prediction = clf.decision_function(X) # Get decision function score
        y_pred = clf.predict(X)                      # Predict outliers

    # Reshape the prediction values to 0 for Valid transactions , 1 for Fraud transactions
    y_pred[y_pred == 1] = 0        # Set predicted labels to 0 for valid transactions.
    y_pred[y_pred == -1] = 1       # Set predicted labels to 1 for fraudulent transactions
    n_errors = (y_pred != y).sum() # Count errors (mislabeled transactions).

    # Run Classification Metrics
    print("{}: {}".format(clf_name, n_errors))                       # Print classifier name and number of errors.
    print("Accuracy Score :", accuracy_score(y, y_pred), end=" ")     # Print accuracy score.
    print("\nClassification Report :\n", classification_report(y, y_pred))  # Print classification report.
```

This step evaluates each of the anomaly detection algorithms:

Iterating Through Classifiers: The code loops through each classifier in the classifiers dictionary.

Fitting and Predicting: For each classifier, it fits the model to the data (X) and generates predictions.

The process is slightly different for each algorithm:

- For LOF, it uses fit_predict() which does both in one step.
- For One-Class SVM and Isolation Forest, it first fits the model, then predicts.

Adjusting Predictions:

y_pred[y_pred == 1] = 0 and y_pred[y_pred == -1] = 1: This adjusts the predictions to match our labeling scheme (0 for normal, 1 for fraud).

Calculating Errors:

n_errors = (y_pred != y).sum(): This calculates the number of misclassifications.

Printing Results:

For each classifier, it prints:

- The number of errors
- The accuracy score
- A detailed classification report, which includes precision, recall, and F1-score for each class

This evaluation step is crucial as it allows us to compare the performance of different anomaly detection algorithms on our credit card fraud detection task. It helps us understand which algorithm is most effective at identifying fraudulent transactions while minimizing false positives.

Balancing the Dataset: In this code, the dataset isn't explicitly balanced. Instead, the code uses techniques that are suitable for imbalanced datasets:

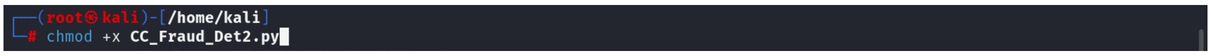
1. It uses the entire dataset without undersampling or oversampling.
2. It sets the 'contamination' parameter in Isolation Forest and Local Outlier Factor to the actual fraction of outliers in the dataset:

```
outlier_fraction = len(Fraud)/float(len(Valid))
```

After entering the code save and exit the file by pressing “CTRL+x” followed by “y” and “Enter”.

convert the file to an executable one using the command:

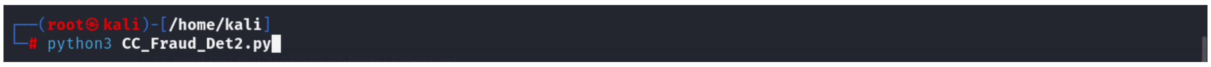
```
$ chmod +x CC_Fraud_Det2.py
```



```
┌──(root㉿kali)-[/home/kali]
└─# chmod +x CC_Fraud_Det2.py
```

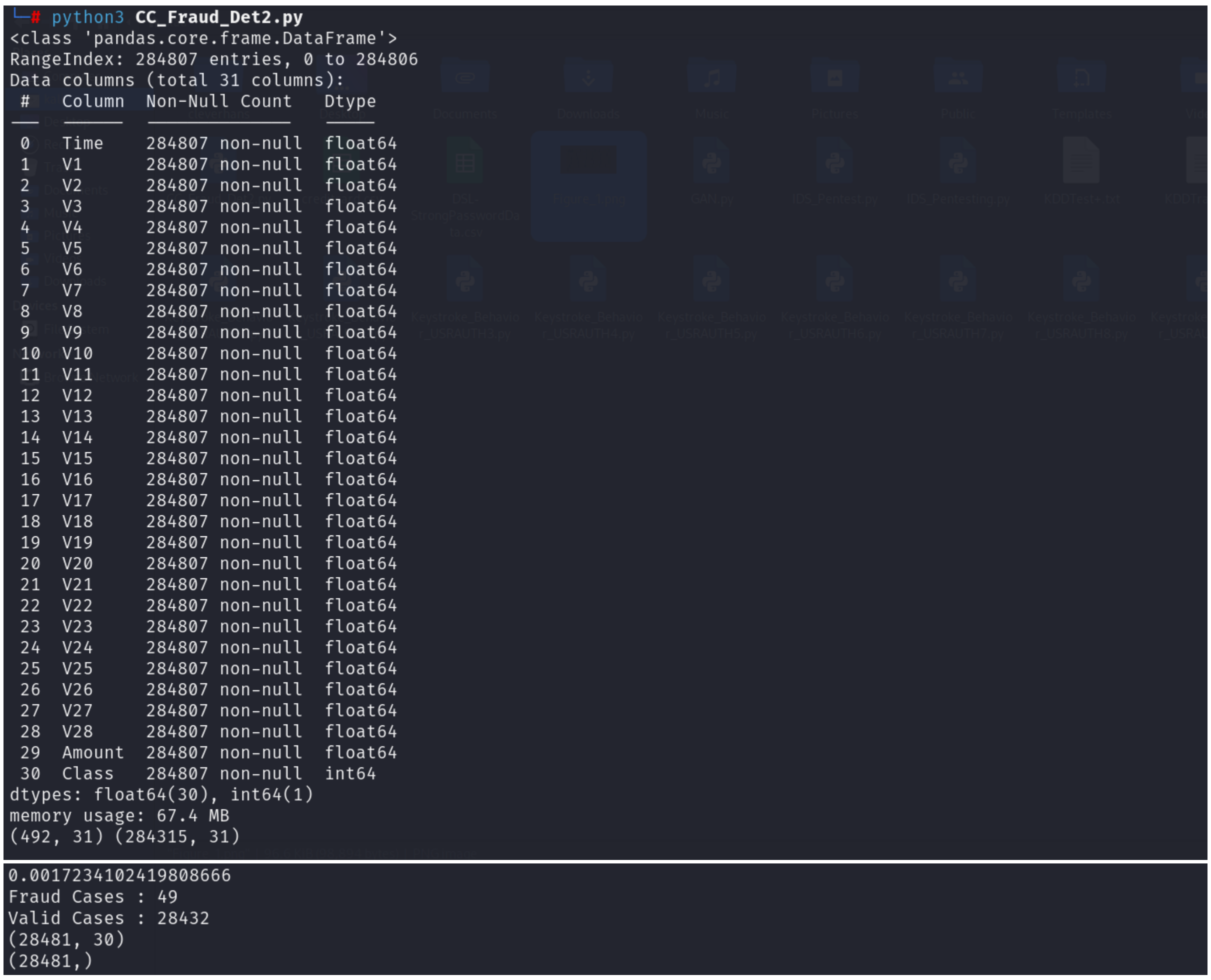
Finally run the program using the command:

```
$ python3 CC_Fraud_Det2.py
```



```
┌──(root㉿kali)-[/home/kali]
└─# python3 CC_Fraud_Det2.py
```

Output



```
└─# python3 CC_Fraud_Det2.py
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   Time     284807 non-null  float64
 1   V1       284807 non-null  float64
 2   V2       284807 non-null  float64
 3   V3       284807 non-null  float64
 4   V4       284807 non-null  float64
 5   V5       284807 non-null  float64
 6   V6       284807 non-null  float64
 7   V7       284807 non-null  float64
 8   V8       284807 non-null  float64
 9   V9       284807 non-null  float64
 10  V10      284807 non-null  float64
 11  V11      284807 non-null  float64
 12  V12      284807 non-null  float64
 13  V13      284807 non-null  float64
 14  V14      284807 non-null  float64
 15  V15      284807 non-null  float64
 16  V16      284807 non-null  float64
 17  V17      284807 non-null  float64
 18  V18      284807 non-null  float64
 19  V19      284807 non-null  float64
 20  V20      284807 non-null  float64
 21  V21      284807 non-null  float64
 22  V22      284807 non-null  float64
 23  V23      284807 non-null  float64
 24  V24      284807 non-null  float64
 25  V25      284807 non-null  float64
 26  V26      284807 non-null  float64
 27  V27      284807 non-null  float64
 28  V28      284807 non-null  float64
 29  Amount   284807 non-null  float64
 30  Class    284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
(492, 31) (284315, 31)
```

```
0.0017234102419808666
Fraud Cases : 49
Valid Cases : 28432
(28481, 30)
(28481,)
```

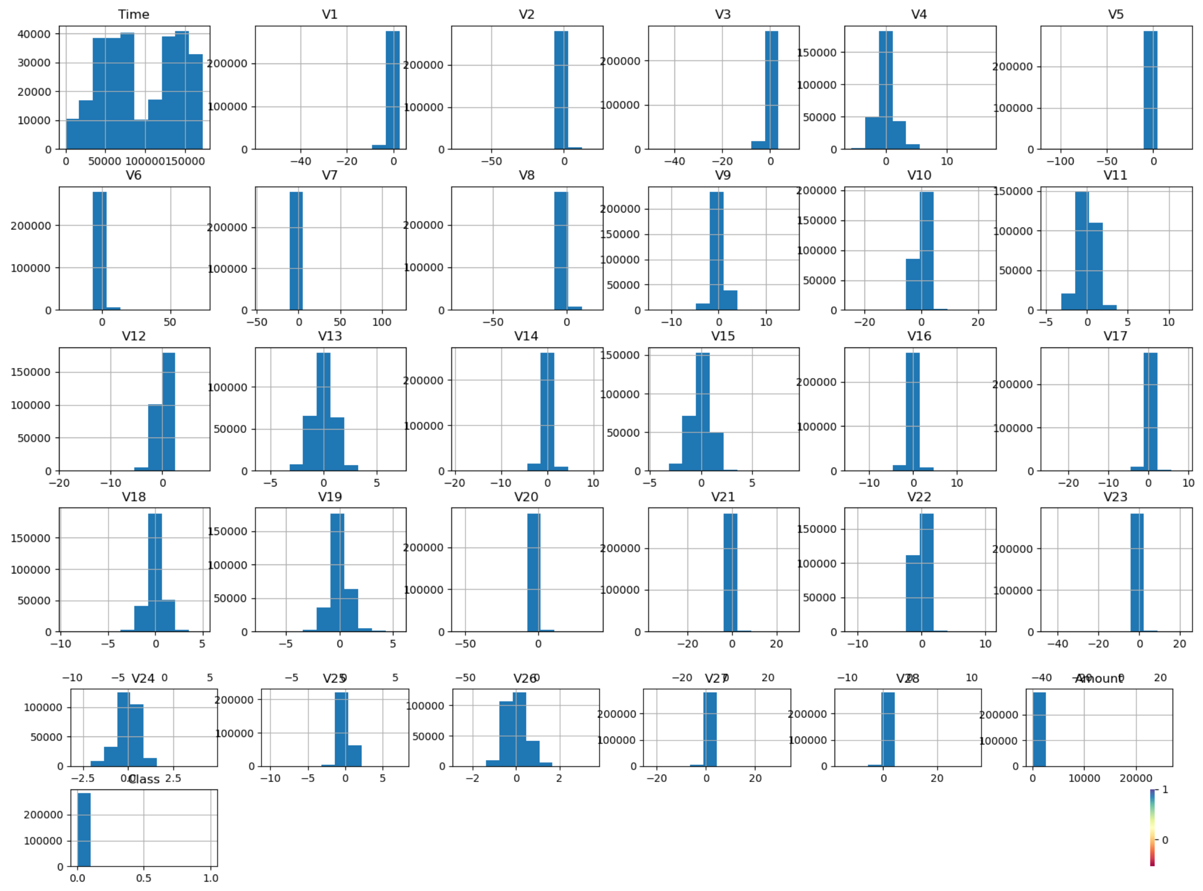
This output image shows the structure of our dataset, which is stored in a pandas DataFrame. Let's break it down:

1. We have 284,807 rows (or entries) in our dataset. Each row represents a single credit card transaction.
2. There are 31 columns in total. Each column represents a different piece of information about the transaction:
   - The first column is "Time", which likely represents when the transaction occurred.

- Columns V1 through V28 are unnamed features. These are due to the result of a data transformation to protect the privacy of the cardholders.
- The "Amount" column likely represents the transaction amount.
- The "Class" column is our target variable - it tells us whether a transaction is fraudulent (1) or not (0).

3. All columns have 284,807 non-null values, which means there's no missing data.
4. Most columns are of type "float64", which means they contain decimal numbers. The "Class" column is "int64", which means it contains whole numbers (in this case, just 0 and 1).
5. The dataset uses about 67.4 MB of memory.
6. 0.0017234102419808666: This is the fraction of fraudulent transactions in our dataset. It's a very small number, which means fraudulent transactions are rare.
7. Fraud Cases: 49: This tells us there are 49 fraudulent transactions in our dataset.
8. Valid Cases: 28432: This shows there are 28,432 valid (non-fraudulent) transactions.
9. (28481, 30): This indicates that we're working with a subset of the original data, containing 28,481 total transactions and 30 features (probably excluding the "Class" column).

These statistics highlight a crucial aspect of our problem: class imbalance. We have far fewer fraudulent transactions than valid ones, which can make it challenging for machine learning models to accurately detect fraud. This imbalance is why we need to use special techniques like anomaly detection algorithms to effectively identify fraudulent transactions.

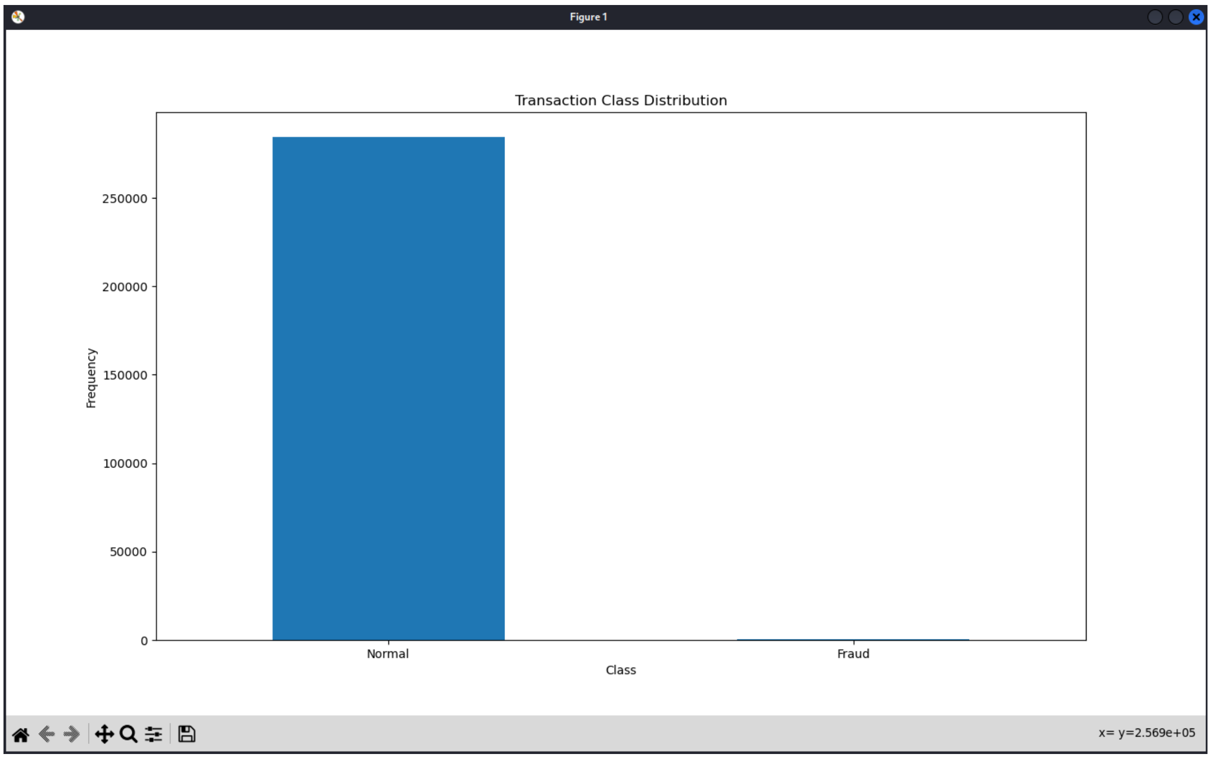


This output image shows histograms for each feature (V1 to V28, Time, and Amount) in our dataset. A histogram is like a bar chart that shows how often different values occur in our data.

Key points:

1. Most features (V1 to V28) are centered around 0, with some having more spread than others.
2. The Time feature shows multiple peaks, suggesting transactions happen more frequently at certain times.
3. The Amount feature shows most transactions are for smaller amounts, with fewer large transactions.

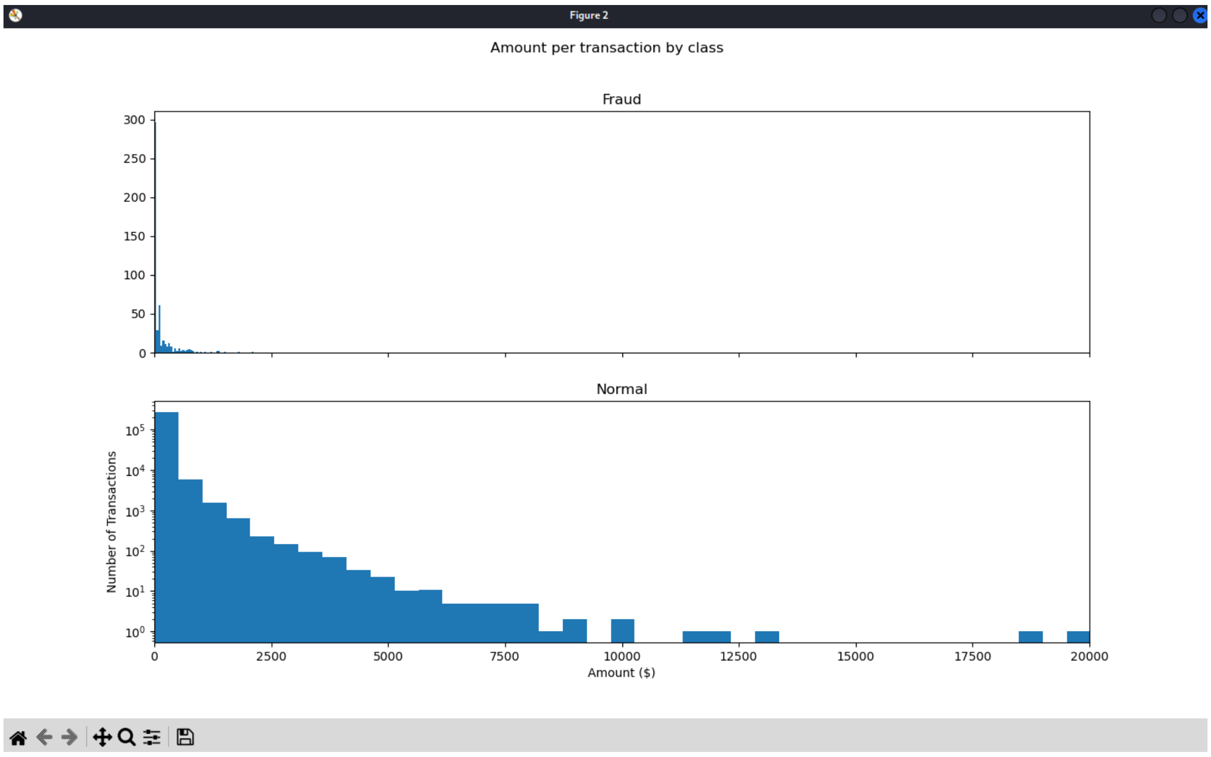
4. The Class feature (last histogram) shows a huge imbalance - there's a very tall bar at 0 (normal transactions) and a tiny bar at 1 (fraudulent transactions).



This bar chart shows the Transaction Class Distribution. It's a visual representation of how imbalanced our dataset is.

Key points:

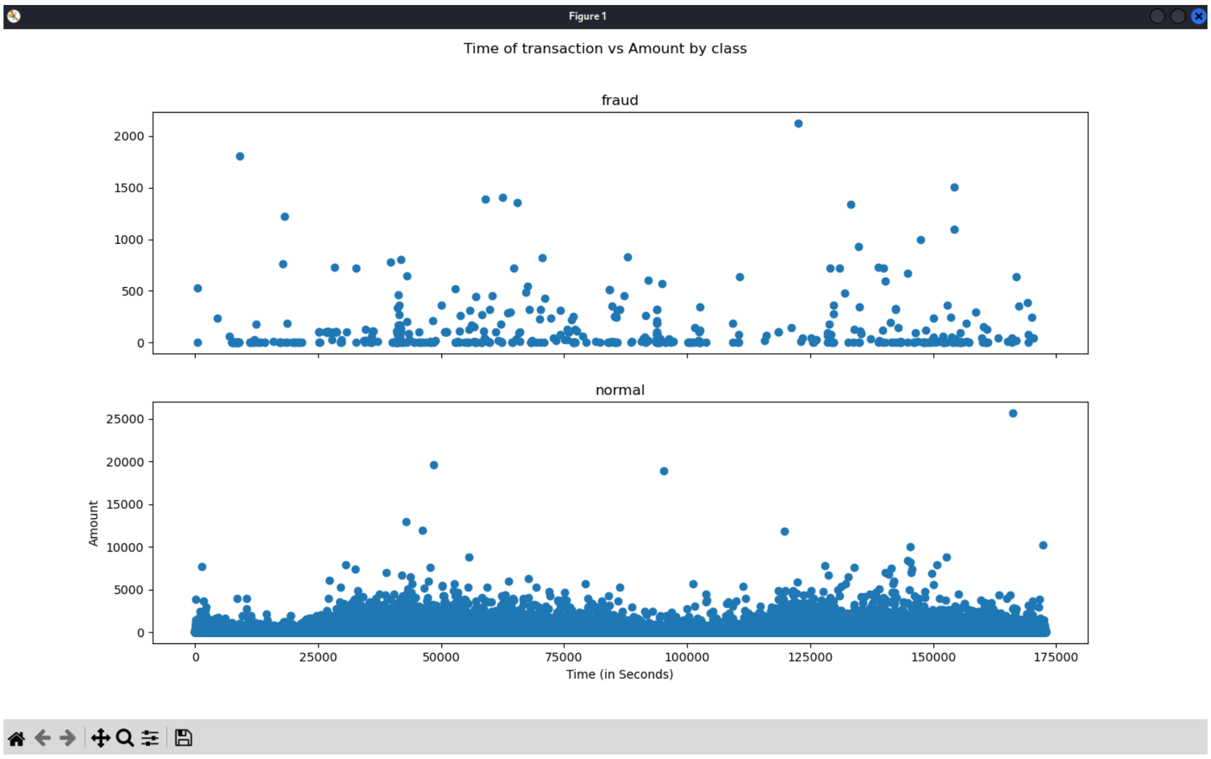
1. The blue bar for "Normal" transactions is extremely tall, representing over 250,000 transactions.
2. The bar for "Fraud" transactions is so small it's barely visible, representing just a few hundred transactions.
3. This imbalance makes it challenging to detect fraud, as there are so few examples to learn from.



This image shows two histograms comparing the Amount per transaction for Fraud and Normal classes.

Key points:

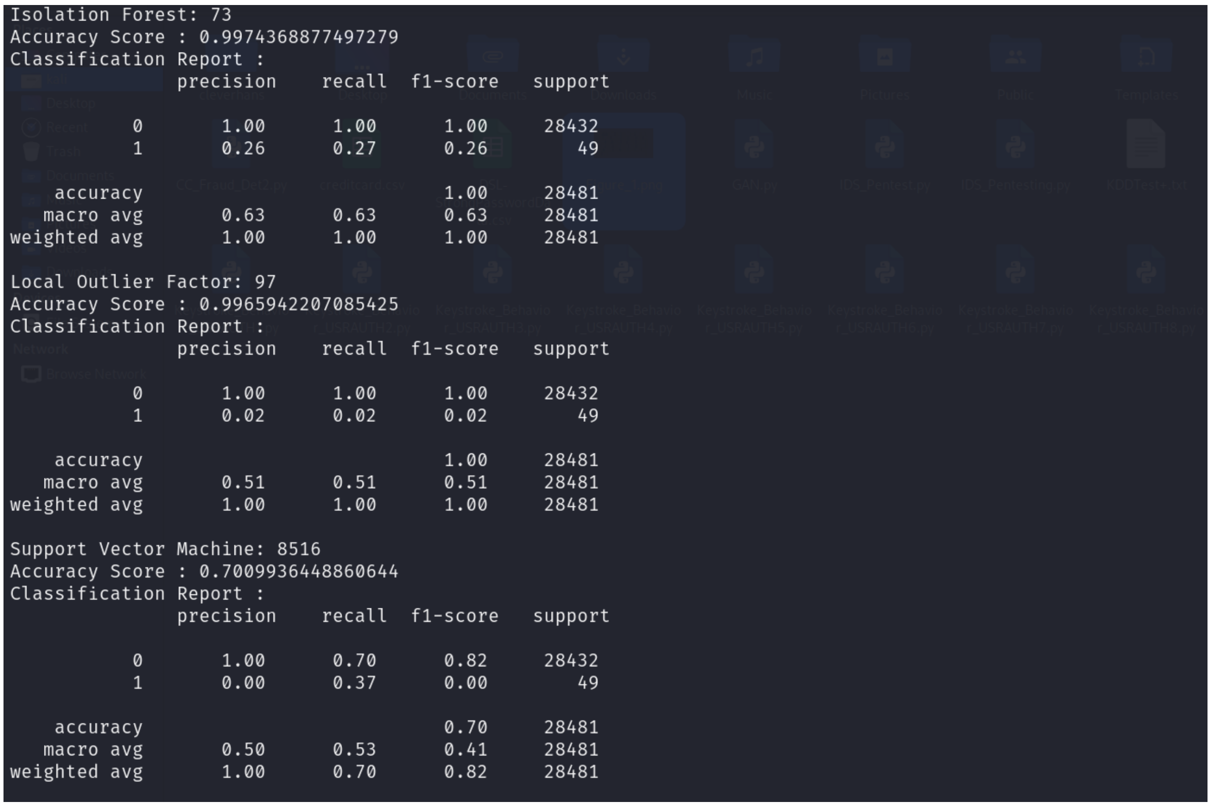
1. For Fraud transactions (top graph):
   - Most fraudulent transactions are for smaller amounts.
   - There are a few larger fraudulent transactions, but they're rare.
2. For Normal transactions (bottom graph):
   - The y-axis is logarithmic, meaning each tick represents a 10x increase.
   - Most normal transactions are also for smaller amounts.
   - There's a long "tail" of larger transactions, but they become less frequent as the amount increases.



This scatter plot shows the Time of transaction vs Amount, separated for fraud and normal transactions.

Key points:

1. For Fraud transactions (top graph):
    - Fraudulent transactions occur throughout the time period.
    - Most fraudulent transactions are for smaller amounts, but there are some larger ones.
    - There's no clear pattern in when fraudulent transactions occur.
2. For Normal transactions (bottom graph):
    - Normal transactions also occur throughout the time period.
    - There's a dense cluster of transactions for smaller amounts.
    - There are some very large transactions, but they're less common.
    - The pattern looks similar to fraudulent transactions, which is why detecting fraud is challenging.



```
Isolation Forest: 73
Accuracy Score : 0.9974368877497279
Classification Report :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28432
           1       0.26      0.27      0.26        49

    accuracy                           1.00     28481
   macro avg       0.63      0.63      0.63     28481
weighted avg       1.00      1.00      1.00     28481

Local Outlier Factor: 97
Accuracy Score : 0.9965942207085425
Classification Report :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28432
           1       0.02      0.02      0.02        49

    accuracy                           1.00     28481
   macro avg       0.51      0.51      0.51     28481
weighted avg       1.00      1.00      1.00     28481

Support Vector Machine: 8516
Accuracy Score : 0.7009936448860644
Classification Report :
              precision    recall  f1-score   support

           0       1.00      0.70      0.82     28432
           1       0.00      0.37      0.00        49

    accuracy                           0.70     28481
   macro avg       0.50      0.53      0.41     28481
weighted avg       1.00      0.70      0.82     28481
```

This output image shows the results of three different machine learning models used for detecting credit card fraud:

1. Isolation Forest
2. Local Outlier Factor
3. Support Vector Machine

For each model, we see an "Accuracy Score" and a "Classification Report." Let's go through what these mean:

Accuracy Score: This is a percentage that tells us how often the model is correct overall. For example, the Isolation Forest has an accuracy of about 99.74%, which means it's right about 99.74% of the time.

Classification Report: This gives us more detailed information about how well the model performs. It has four columns:

1. **Precision:** How often the model is correct when it predicts fraud.

2. **Recall:** How often the model correctly identifies actual fraud cases.
3. **F1-score:** A balance between precision and recall.
4. **Support:** The number of transactions in each category.

And it has rows for:

- 0: Normal transactions
- 1: Fraudulent transactions
- accuracy: Overall accuracy
- macro avg: Average of all classes, treating them equally
- weighted avg: Average weighted by the number of samples in each class

Let's look at the Isolation Forest model as an example:

For normal transactions (0):

- Precision, recall, and F1-score are all 1.00, which is perfect.
- There were 28,432 normal transactions.

For fraudulent transactions (1):

- Precision is 0.26, recall is 0.27, and F1-score is 0.26.
- There were 49 fraudulent transactions.