# ML PROJECT MANUAL

HATEEM TAHIR
2023-BS-AI-032

# REGRESSION PROJECT (DIAMOND PRICE PREDICTION)

## 1. Project Description

The goal of this project is to predict the prices of diamonds using various machine learning models. By analyzing the features of the diamonds (such as carat, cut, color, clarity, etc.), the models aim to learn complex patterns and make accurate predictions on unseen data. This project demonstrates the end-to-end process of building a machine learning solution, from data preprocessing to model evaluation.

## 2. Data Description

The dataset used in this project is the "diamonds" dataset, commonly available via Seaborn or CSV files. It includes the following attributes:

- carat: weight of the diamond
- cut: quality of the cut (Fair, Good, Very Good, Premium, Ideal)
- color: diamond color, from D (best) to J (worst)
- clarity: a measurement of how clear the diamond is
- depth: total depth percentage
- table: width of the top of the diamond relative to the widest point
- price: price in US dollars (target variable)
- x, y, z: length, width, and depth in mm.

## 3. Code and Explanation

## 3.1 Introduction to Python and Libraries for Machine Learning, Environmental Setup

The project uses Python and the following key libraries:

- Pandas: For data manipulation and loading

```
import pandas as pd
```

- **NumPy**: For numerical operations

Import Numpy for numerical computations and array manipulations

```
import numpy as np
```
[29]

- **Matplotlib / Seaborn:** For data visualization

```
import matplotlib.pyplot as plt
```
[1] ✓ 2.6s

- **Scikit-learn:** For preprocessing, model training, and evaluation

```
import sklearn as sk
```
[2] ✓ 11.1s

## 3.2 Handling Missing Values, Data Normalization, Standardization, Data Visualization

- **Load the dataset**

```
data = pd.read_csv("diamonds.csv")
```
[30]

- **data.head():** Displays the first 5 rows of the dataset. It's useful for quickly viewing the structure and sample values of the data.

```
data.head()
```

[31]

| | Unnamed: 0 | carat | cut | color | clarity | depth | table | price | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0.23 | Ideal | E | SI2 | 61.5 | 55.0 | 326 | 3.95 | 3.98 | 2.43 |
| 1 | 2 | 0.21 | Premium | E | SI1 | 59.8 | 61.0 | 326 | 3.89 | 3.84 | 2.31 |
| 2 | 3 | 0.23 | Good | E | VS1 | 56.9 | 65.0 | 327 | 4.05 | 4.07 | 2.31 |
| 3 | 4 | 0.29 | Premium | I | VS2 | 62.4 | 58.0 | 334 | 4.20 | 4.23 | 2.63 |
| 4 | 5 | 0.31 | Good | J | SI2 | 63.3 | 58.0 | 335 | 4.34 | 4.35 | 2.75 |

- **data.tail():** Displays the last 5 rows of the dataset. This helps check for patterns or issues at the end of the data, such as missing values.

```
data.tail()
```

[9] ✓ 0.0s

| | Unnamed: 0 | carat | cut | color | clarity | depth | table | price | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 53935 | 53936 | 0.72 | Ideal | D | SI1 | 60.8 | 57.0 | 2757 | 5.75 | 5.76 | 3.50 |
| 53936 | 53937 | 0.72 | Good | D | SI1 | 63.1 | 55.0 | 2757 | 5.69 | 5.75 | 3.61 |
| 53937 | 53938 | 0.70 | Very Good | D | SI1 | 62.8 | 60.0 | 2757 | 5.66 | 5.68 | 3.56 |
| 53938 | 53939 | 0.86 | Premium | H | SI2 | 61.0 | 58.0 | 2757 | 6.15 | 6.12 | 3.74 |
| 53939 | 53940 | 0.75 | Ideal | D | SI2 | 62.2 | 55.0 | 2757 | 5.83 | 5.87 | 3.64 |

- **data.nunique():** Returns the number of unique values in each column. It helps identify categorical features and check for data redundancy.

```
data.nunique()
```

[35]

```
Unnamed: 0    53940
carat           273
cut               5
color             7
clarity           8
depth           184
table           127
price         11602
x               554
y               552
z               375
```

- **data.describe():** Provides statistical summaries (count, mean, std, min, max, etc.) for numeric columns, giving insights into distribution and spread.

```python
data.describe()
```

[36]

| | Unnamed: 0 | carat | depth | table | price | x | y | z |
|---|---|---|---|---|---|---|---|---|
| count | 53940.000000 | 53940.000000 | 53940.000000 | 53940.000000 | 53940.000000 | 53940.000000 | 53940.000000 | 53940.000000 |
| mean | 26970.500000 | 0.797940 | 61.749405 | 57.457184 | 3932.799722 | 5.731157 | 5.734526 | 3.538734 |
| std | 15571.281097 | 0.474011 | 1.432621 | 2.234491 | 3989.439738 | 1.121761 | 1.142135 | 0.705699 |
| min | 1.000000 | 0.200000 | 43.000000 | 43.000000 | 326.000000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 13485.750000 | 0.400000 | 61.000000 | 56.000000 | 950.000000 | 4.710000 | 4.720000 | 2.910000 |
| 50% | 26970.500000 | 0.700000 | 61.800000 | 57.000000 | 2401.000000 | 5.700000 | 5.710000 | 3.530000 |
| 75% | 40455.250000 | 1.040000 | 62.500000 | 59.000000 | 5324.250000 | 6.540000 | 6.540000 | 4.040000 |
| max | 53940.000000 | 5.010000 | 79.000000 | 95.000000 | 18823.000000 | 10.740000 | 58.900000 | 31.800000 |

- **data.info():** Shows a summary of the dataset including column names, data types, non-null counts, and memory usage—useful for spotting missing data and data types.

```python
data.info()
```

[11] ✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
Data columns (total 11 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  53940 non-null  int64
 1   carat       53940 non-null  float64
 2   cut         53940 non-null  object
 3   color       53940 non-null  object
 4   clarity     53940 non-null  object
 5   depth       53940 non-null  float64
 6   table       53940 non-null  float64
 7   price       53940 non-null  int64
 8   x           53940 non-null  float64
 9   y           53940 non-null  float64
 10  z           53940 non-null  float64
dtypes: float64(6), int64(2), object(3)
memory usage: 4.5+ MB
```

- **Missing values:** Checked using data.isnull().sum(). The dataset appears to be clean with no missing values.

```
data.isnull().sum()
```

```
Unnamed: 0    0
carat         0
cut           0
color         0
clarity       0
depth         0
table         0
price         0
x             0
y             0
z             0
dtype: int64
```

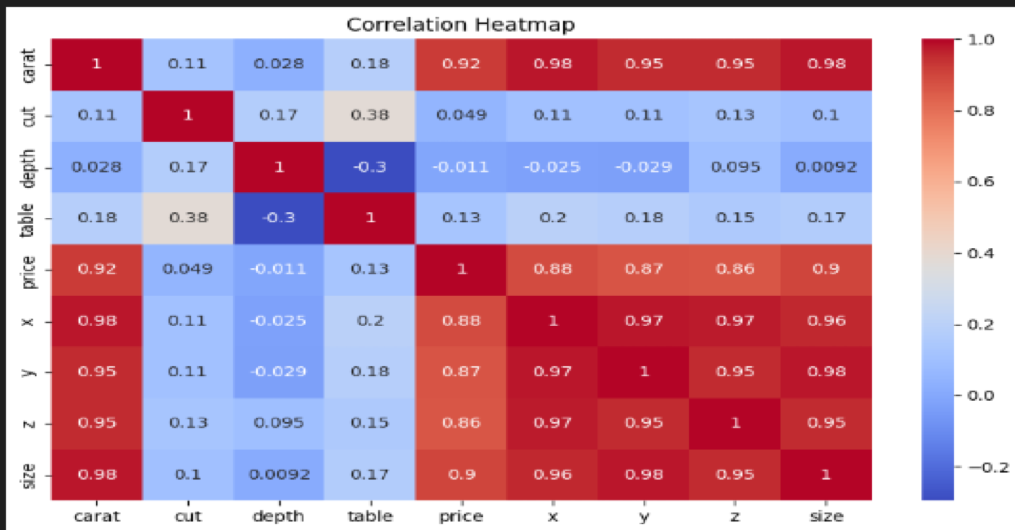
- **Normalization**: Applied using StandardScaler for numerical features to ensure all features are on the same scale.

```
from sklearn.preprocessing import StandardScaler

numerical_features = data.select_dtypes(include=['int64', 'float64']).drop(columns=['price']).columns

scaler = StandardScaler()

data[numerical_features] = scaler.fit_transform(data[numerical_features])

print("Normalized features:\n", data[numerical_features].head())
```

```
Normalized features:
      carat       cut     depth     table         x         y         z  \
0 -1.198168 -0.970753 -0.174092 -1.099672 -1.587837 -1.536196 -1.571129
1 -1.240361 -0.180842 -1.360738  1.585529 -1.641325 -1.658774 -1.741175
2 -1.198168  0.609070 -3.385019  3.375663 -1.498691 -1.457395 -1.741175
3 -1.071587 -0.180842  0.454133  0.242928 -1.364971 -1.317305 -1.287720
4 -1.029394  0.609070  1.082358  0.242928 -1.240167 -1.212238 -1.117674

      size
0 -1.171294
1 -1.218533
2 -1.172894
3 -1.062372
4 -0.996008
```

- **Visualization:**
    - Seaborn and Matplotlib were used to show relationships using scatterplots, histograms, and correlation heat maps.

```python
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=(10,6))
sns.heatmap(numeric_data.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```
[24] ✓ 0.3s




- Price vs. Carat was analyzed and showed a strong positive correlation.

```python
numeric_data = data.select_dtypes(include=[np.number])
print("Correlation matrix:")
print(numeric_data.corr())
```
[38] ✓ 0.0s

```
Correlation matrix:
          carat       cut     depth     table     price         x         y  \
carat  1.000000  0.114426  0.028224  0.181618  0.921591  0.975094  0.951722
cut    0.114426  1.000000  0.169916  0.381988  0.049421  0.105361  0.105319
depth  0.028224  0.169916  1.000000 -0.295779 -0.010647 -0.025289 -0.029341
table  0.181618  0.381988 -0.295779  1.000000  0.127134  0.195344  0.183760
price  0.921591  0.049421 -0.010647  0.127134  1.000000  0.884435  0.865421
x      0.975094  0.105361 -0.025289  0.195344  0.884435  1.000000  0.974701
y      0.951722  0.105319 -0.029341  0.183760  0.865421  0.974701  1.000000
z      0.953387  0.126726  0.094924  0.150929  0.861249  0.970772  0.952006
size   0.976308  0.101119  0.009157  0.167400  0.902385  0.956564  0.975143

              z      size
carat  0.953387  0.976308
cut    0.126726  0.101119
depth  0.094924  0.009157
table  0.150929  0.167400
price  0.861249  0.902385
x      0.970772  0.956564
y      0.952006  0.975143
z      1.000000  0.950065
size   0.950065  1.000000
```

### 3.3 Data Preprocessing

- **Categorical Encoding**:
  - Categorical variables such as cut, color, and clarity were encoded using LabelEncoder.

Mapping Diamond Cut Categories to Numerical Values

The following code converts the categorical values in the `cut` column to numerical values for analysis and machine learning.

```
data["cut"] = data["cut"].map({"Ideal": 1,
                               "Premium": 2,
                               "Good": 3,
                               "Very Good": 4,
                               "Fair": 5})
```

[16] ✓ 0.0s

- **Feature Selection**:
  - Only relevant features were selected for training by analyzing correlations.

```
data
```

[15] ✓ 0.0s

| | carat | cut | color | clarity | depth | table | price | x | y | z | size |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.23 | Ideal | E | SI2 | 61.5 | 55.0 | 326 | 3.95 | 3.98 | 2.43 | 38.202030 |
| 1 | 0.21 | Premium | E | SI1 | 59.8 | 61.0 | 326 | 3.89 | 3.84 | 2.31 | 34.505856 |
| 2 | 0.23 | Good | E | VS1 | 56.9 | 65.0 | 327 | 4.05 | 4.07 | 2.31 | 38.076885 |
| 3 | 0.29 | Premium | I | VS2 | 62.4 | 58.0 | 334 | 4.20 | 4.23 | 2.63 | 46.724580 |
| 4 | 0.31 | Good | J | SI2 | 63.3 | 58.0 | 335 | 4.34 | 4.35 | 2.75 | 51.917250 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 53935 | 0.72 | Ideal | D | SI1 | 60.8 | 57.0 | 2757 | 5.75 | 5.76 | 3.50 | 115.920000 |
| 53936 | 0.72 | Good | D | SI1 | 63.1 | 55.0 | 2757 | 5.69 | 5.75 | 3.61 | 118.110175 |
| 53937 | 0.70 | Very Good | D | SI1 | 62.8 | 60.0 | 2757 | 5.66 | 5.68 | 3.56 | 114.449728 |
| 53938 | 0.86 | Premium | H | SI2 | 61.0 | 58.0 | 2757 | 6.15 | 6.12 | 3.74 | 140.766120 |
| 53939 | 0.75 | Ideal | D | SI2 | 62.2 | 55.0 | 2757 | 5.83 | 5.87 | 3.64 | 124.568444 |

53940 rows × 11 columns

- **Train-test Split**:

- Dataset was split into training and testing sets (typically 80/20 or 70/30 split).

- Splits the dataset into training and testing sets for machine learning.
- Purpose is to help evaluate the model's performance on unseen data (testing set) after training it on the training set

```
from sklearn.model_selection import train_test_split
x = np.array(data[["carat", "cut", "size"]])
y = np.array(data[["price"]])

xtrain, xtest, ytrain, ytest = train_test_split(x, y,
                                                test_size=0.10,
                                                random_state=42)
```

Training the Random Forest Regressor

- Trains a machine learning model using the Random Forest Regressor algorithm.
- RandomForestRegressor is an ensemble learning method that combines multiple decision trees to make robust and accurate predictions.
- The trained Random Forest Regressor will predict diamond prices based on the input features.

```
from sklearn.ensemble import RandomForestRegressor
rf_model = RandomForestRegressor()
rf_model.fit(xtrain, ytrain)
rf_preds = rf_model.predict(xtest)
```

98s

```
C:\Users\iCare\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\s
  return fit_method(estimator, *args, **kwargs)
```

## 3.4 Implementing Decision Tree

- Implemented using DecisionTreeRegressor.

```
from sklearn.tree import DecisionTreeRegressor
dt_model = DecisionTreeRegressor()
dt_model.fit(xtrain, ytrain)
dt_preds = dt_model.predict(xtest)
```

0.6s

- Performed well on the dataset and captured non-linear relationships.
- Risk of overfitting was noted; hence, max_depth and other hyperparameters were tuned.

## 3.5 Evaluation Metrics

The following metrics were used to evaluate models:

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)

- $R^2$ Score

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import numpy as np

def evaluate_model(name, ytrue, ypred):
    print(f"--- {name} ---")
    print("MAE:", mean_absolute_error(ytrue, ypred))
    print("MSE:", mean_squared_error(ytrue, ypred))
    print("RMSE:", np.sqrt(mean_squared_error(ytrue, ypred)))
    print("R^2 Score:", r2_score(ytrue, ypred))
    print()

# Test with sample values
evaluate_model("Test Model", [3, -0.5, 2, 7], [2.5, 0.0, 2, 8])
```

[30] ✓ 0.0s

```
--- Test Model ---
MAE: 0.5
MSE: 0.375
RMSE: 0.6123724356957945
R^2 Score: 0.9486081370449679
```

## 3.6 Output of the Model:

```
print("Diamond Price Prediction")

carat_options = [0.5, 1.0, 1.5, 2.0, 2.5]
print("Carat Size Options:")
for i, option in enumerate(carat_options, 1):
    print(f"{i}. {option}")
carat_choice = int(input("Choose Carat Size (Enter the number): "))
a = carat_options[carat_choice - 1]

cut_options = {"Ideal": 1, "Premium": 2, "Good": 3, "Very Good": 4, "Fair": 5}
print("Cut Type Options:")
for cut, value in cut_options.items():
    print(f"{value}. {cut}")
b = int(input("Choose Cut Type (Enter the number): "))

size_options = [50.0, 100.0, 150.0, 200.0, 250.0]
print("Size Options:")
for i, option in enumerate(size_options, 1):
    print(f"{i}. {option}")
size_choice = int(input("Choose Size (Enter the number): "))
c = size_options[size_choice - 1]

features = np.array([[a, b, c]])
print("Predicted Diamond's Price = $", rf_model.predict(features))
```

```
...   Diamond Price Prediction
      Carat Size Options:
      1. 0.5
      2. 1.0
      3. 1.5
      4. 2.0
      5. 2.5
      Cut Type Options:
      1. Ideal
      2. Premium
      3. Good
      4. Very Good
      5. Fair
      Size Options:
      1. 50.0
      2. 100.0
      3. 150.0
      4. 200.0
      5. 250.0
      Predicted Diamond's Price = $ [13144.05]
```

## 4. Analysis of the Project

- The dataset is well-suited for regression tasks, especially due to the continuous nature of the target variable.
- Linear Regression provided a strong baseline but could not handle the non-linearities.
- Decision Trees showed better accuracy but needed pruning or tuning to avoid overfitting.
- SVMs offered good generalization but at the cost of computation time.
- Visualizations helped understand feature importance and correlations effectively.

## 5. Conclusion

This project demonstrated how machine learning techniques can be applied to real-world datasets to solve regression problems. With proper preprocessing, feature encoding, and model selection, it is possible to build models that predict diamond prices accurately. Future improvements could include:

- Hyperparameter tuning using GridSearchCV
- Ensemble methods like Random Forest or Gradient Boosting
- Deep Learning models for comparison

# CLASSIFICATION PROJECT(CREDIT CARD FRAUD DETECTION)

## 1. Project Description

This project addresses the classification problem of detecting fraudulent credit card transactions. Financial fraud is a major concern globally, and this project uses machine learning techniques to identify suspicious activity. A dataset of credit card transactions is analyzed and used to train models that distinguish between legitimate and fraudulent behavior.

## 2. Data Description

The dataset contains transactions made by European cardholders in September 2013. It includes 284,807 transactions, among which only 492 are frauds (Class = 1). The dataset features:

- 28 anonymized features (V1 to V28)
- Time: Time elapsed between this transaction and the first transaction in the dataset
- Amount: Transaction amount
- Class: Target variable (0 for legitimate, 1 for fraud)

Due to confidentiality, the original features have been transformed using PCA. The data is highly imbalanced, which is a key challenge for training reliable models.

# 3. Code and Explanation

## 3.1 Introduction to Python and Libraries for Machine Learning, Environmental Setup

- **NumPy** and **Pandas** for data manipulation
- **Matplotlib** and **Seaborn** for visualization
- **Scikit-learn** for machine learning algorithms and metrics

IMPORTING LIBRARIES

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

[1] 1.5s

## 3.2 Handling Missing Values, Data Normalization, Standardization, Data Visualization

- The dataset was inspected using .info() and .isnull().sum() to ensure there were no missing values.

```
credit_card_data.isnull().sum()
```

```
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...*

```
credit_card_data.head()
```

| | 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

- Since features were already PCA-transformed, normalization was not required, but Amount was scaled where needed.

## Get a Concise Summary of a Pandas DataFrame

```
credit_card_data.info()
```

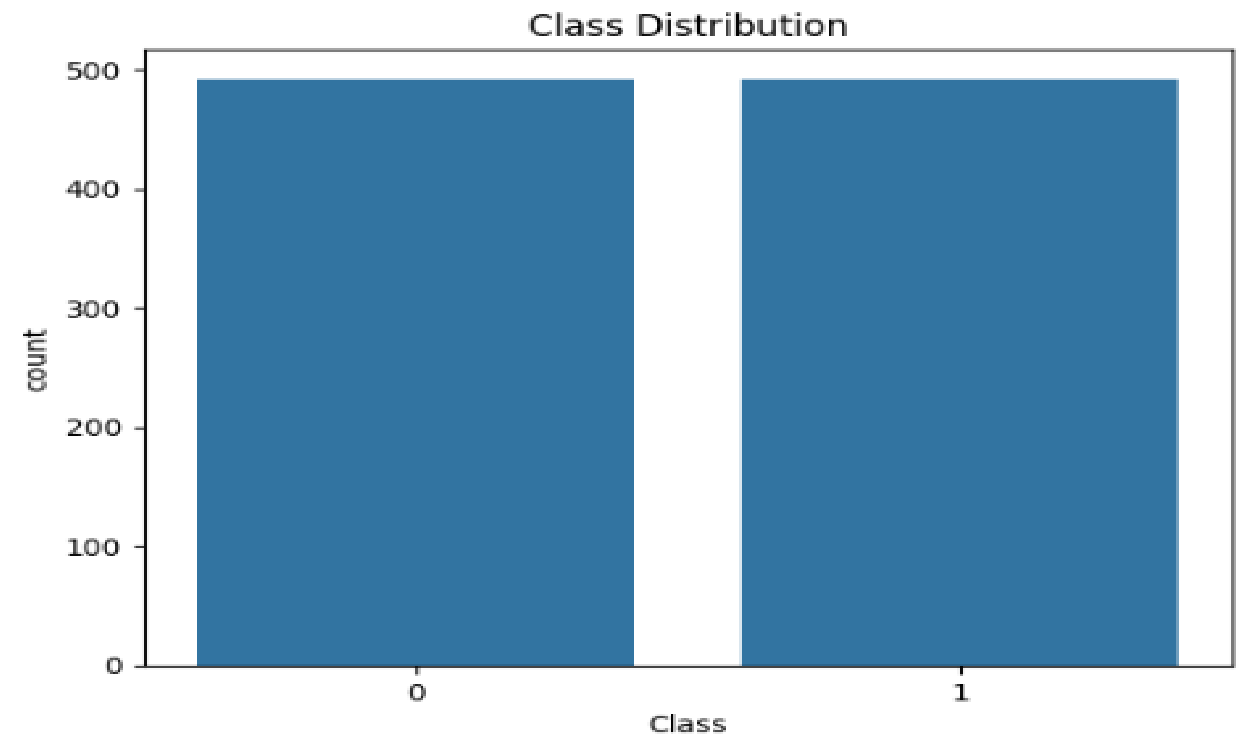
[42]

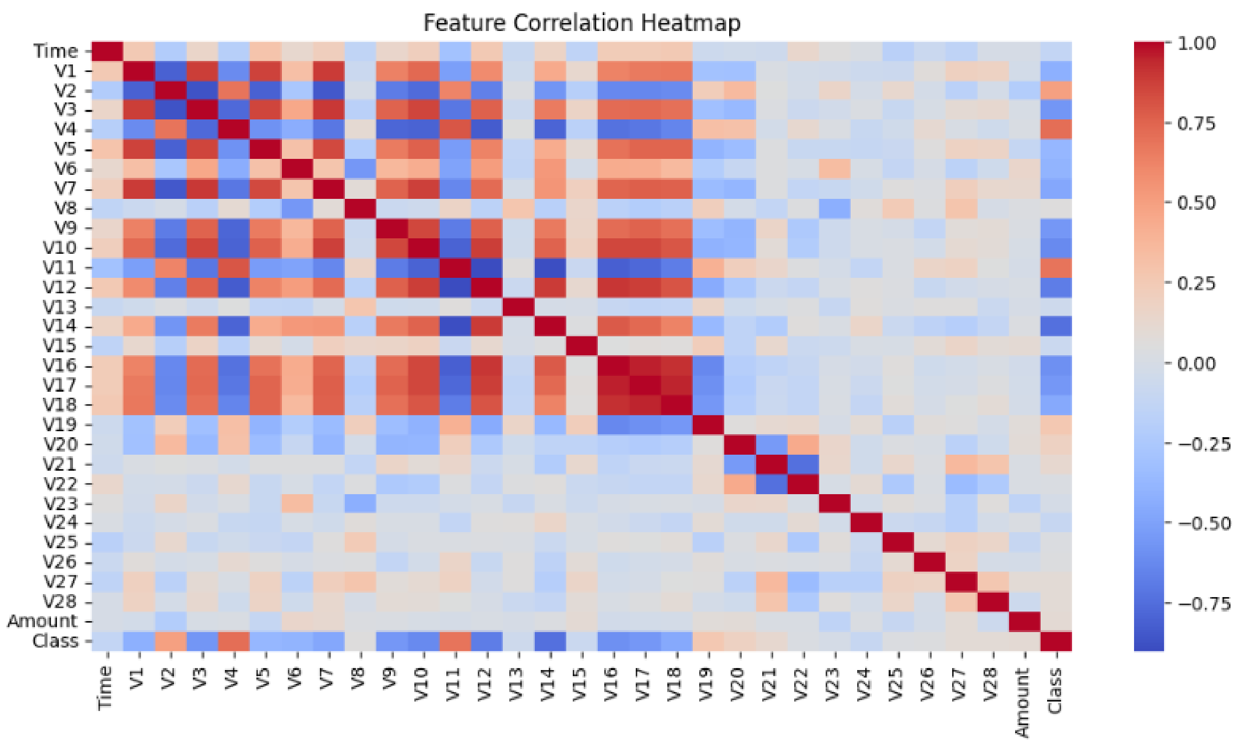
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 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
...
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

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

- Visualization using boxplots and histograms helped understand class imbalance and feature distribution.

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler

# Normalization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Visualize class distribution
sns.countplot(x='Class', data=new_dataset)
plt.title("Class Distribution")
plt.show()

# Visualize correlation
plt.figure(figsize=(12,6))
sns.heatmap(new_dataset.corr(), cmap="coolwarm", annot=False)
plt.title("Feature Correlation Heatmap")
plt.show()
```


Class Distribution
500
400
300
200
100
0
count
0
1
Class


Feature Correlation Heatmap
Time
V1
V2
V3
V4
V5
V6
V7
V8
V9
V10
V11
V12
V13
V14
V15
V16
V17
V18
V19
V20
V21
V22
V23
V24
V25
V26
V27
V28
Amount
Class
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

## 3.3 Data Preprocessing

- Transactions were split based on class (legitimate and fraud) to examine statistical patterns.

WE ASSIGN LEGIT TRANSACTIONS TO 0 CLASS & CLASS 1 IS FRAUDULENT TRANSACTION

```
legit = credit_card_data[credit_card_data.Class == 0]
fraud = credit_card_data[credit_card_data.Class == 1]
```

- A new dataset was created by concatenating both types for training.

```
legit_sample = legit.sample(n=492)
```

```
new_dataset = pd.concat([legit_sample, fraud], axis=0)
```

- Feature scaling using StandardScaler was applied to ensure the model works optimally.

```
new_dataset['Class'].value_counts()
```

```
Class
0    492
1    492
Name: count, dtype: int64
```

```
new_dataset.groupby('Class').mean()
```

| Class | Time | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | ... | V20 | V21 | V22 | V23 | V24 | V25 | V26 | V27 | V28 | Amount |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 93088.276423 | -0.087648 | 0.028173 | -0.001165 | 0.061702 | 0.030238 | -0.000125 | -0.061776 | 0.026854 | 0.022510 | ... | -0.018153 | 0.030575 | -0.035527 | -0.014707 | 0.013341 | 0.005203 | 0.013863 | -0.016157 | -0.008424 | 80.653557 |
| 1 | 80746.806911 | -4.771948 | 3.623778 | -7.033281 | 4.542029 | -3.151225 | -1.397737 | -5.568731 | 0.570636 | -2.581123 | ... | 0.372319 | 0.713588 | 0.014049 | -0.040308 | -0.105130 | 0.041449 | 0.051648 | 0.170575 | 0.075667 | 122.211321 |

```
new_dataset.tail()
```

| | Time | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | ... | V21 | V22 | V23 | V24 | V25 | V26 | V27 | V28 | Amount | Class |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 279863 | 169142.0 | -1.927883 | 1.125653 | -4.518331 | 1.749293 | -1.566487 | -2.010494 | -0.882850 | 0.697211 | -2.064945 | ... | 0.778584 | -0.319189 | 0.639419 | -0.294885 | 0.537503 | 0.788395 | 0.292680 | 0.147968 | 390.00 | 1 |
| 280143 | 169347.0 | 1.378559 | 1.289381 | -5.004247 | 1.411850 | 0.442581 | -1.326536 | -1.413170 | 0.248525 | -1.127396 | ... | 0.370612 | 0.028234 | -0.145640 | -0.081049 | 0.521875 | 0.739467 | 0.389152 | 0.186637 | 0.76 | 1 |
| 280149 | 169351.0 | -0.676143 | 1.126366 | -2.213700 | 0.468308 | -1.120541 | -0.003346 | -2.234739 | 1.210158 | -0.652250 | ... | 0.751826 | 0.834108 | 0.190944 | 0.032070 | -0.739695 | 0.471111 | 0.385107 | 0.194361 | 77.89 | 1 |
| 281144 | 169966.0 | -3.113832 | 0.585864 | -5.399730 | 1.817092 | -0.840618 | -2.943548 | -2.208002 | 1.058733 | -1.632333 | ... | 0.583276 | -0.269209 | -0.456108 | -0.183659 | -0.328168 | 0.606116 | 0.884876 | -0.253700 | 245.00 | 1 |
| 281674 | 170348.0 | 1.991976 | 0.158476 | -2.583441 | 0.408670 | 1.151147 | -0.096695 | 0.223050 | -0.068384 | 0.577829 | ... | -0.164350 | -0.295135 | -0.072173 | -0.450261 | 0.313267 | -0.289617 | 0.002988 | -0.015309 | 42.53 | 1 |

5 rows × 31 columns

## 3.4 Implementing Logistic Regression

- A Logistic Regression model was implemented using Scikit-learn. It performed reasonably well but struggled slightly with the imbalanced class distribution.

```
X = new_dataset.drop(columns='Class', axis=1)
Y = new_dataset['Class']
```

```
print(X)
```

```
             Time        V1        V2        V3        V4        V5        V6  \
204931  135510.0  1.831839 -0.439543  0.037158  1.230670 -0.713703  0.193703
95293    65242.0 -0.430541  1.123121  0.574395  0.169222 -0.096518 -0.818835
176961  122983.0  1.969487 -0.208257 -0.581450  0.498548 -0.433284 -0.222128
156514  108273.0  1.945714  0.094027 -0.799294  1.665359  0.126650 -0.405109
122106   76438.0 -0.430973  0.938809  1.139900 -0.164421  0.418508 -0.045062
...          ...       ...       ...       ...       ...       ...       ...
279863  169142.0 -1.927883  1.125653 -4.518331  1.749293 -1.566487 -2.010494
280143  169347.0  1.378559  1.289381 -5.004247  1.411850  0.442581 -1.326536
280149  169351.0 -0.676143  1.126366 -2.213700  0.468308 -1.120541 -0.003346
281144  169966.0 -3.113832  0.585864 -5.399730  1.817092 -0.840618 -2.943548
281674  170348.0  1.991976  0.158476 -2.583441  0.408670  1.151147 -0.096695

              V7        V8        V9  ...       V20       V21       V22  \
204931 -0.817732  0.230914  0.993672  ... -0.170993  0.102253  0.327293
95293   0.654585  0.226159 -0.913030  ... -0.105377  0.164951  0.394363
176961 -0.784511  0.079202  1.203152  ... -0.066135 -0.058378 -0.024106
156514 -0.009345 -0.245805  1.925870  ... -0.266872 -0.009559  0.484579
122106  0.447742  0.273659 -0.535704  ...  0.070586 -0.210803 -0.577307
...          ...       ...       ...  ...       ...       ...       ...
279863 -0.882850  0.697211 -2.064945  ...  1.252967  0.778584 -0.319189
280143 -1.413170  0.248525 -1.127396  ...  0.226138  0.370612  0.028234
280149 -2.234739  1.210158 -0.652250  ...  0.247968  0.751826  0.834108
281144 -2.208002  1.058733 -1.632333  ...  0.306271  0.583276 -0.269209
281674  0.223050 -0.068384  0.577829  ... -0.017652 -0.164350 -0.295135
...
281144 -0.456108 -0.183659 -0.328168  0.606116  0.884876 -0.253700  245.00
281674 -0.072173 -0.450261  0.313267 -0.289617  0.002988 -0.015309   42.53

[984 rows x 30 columns]
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

```
print(Y)
[25]
...  204931    0
     95293     0
     176961    0
     156514    0
     122106    0
              ..
     279863    1
     280143    1
     280149    1
     281144    1
     281674    1
     Name: Class, Length: 984, dtype: int64
```

## 3.5 Implementing Decision Tree

A Decision Tree Classifier was used due to its interpretability. It quickly overfit the minority class, highlighting the need for regularization or tree pruning.

```
from sklearn.tree import DecisionTreeClassifier

dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, Y_train)

dt_predictions = dt_model.predict(X_test)
print("Decision Tree Accuracy:", accuracy_score(Y_test, dt_predictions))

[38]
...  Decision Tree Accuracy: 0.8984771573604061
```

## 3.6 Implementing Support Vector Machine

Support Vector Machine (SVM) was applied with scaled features to improve accuracy. Though slower to train, it provided more stable predictions than the Decision Tree.

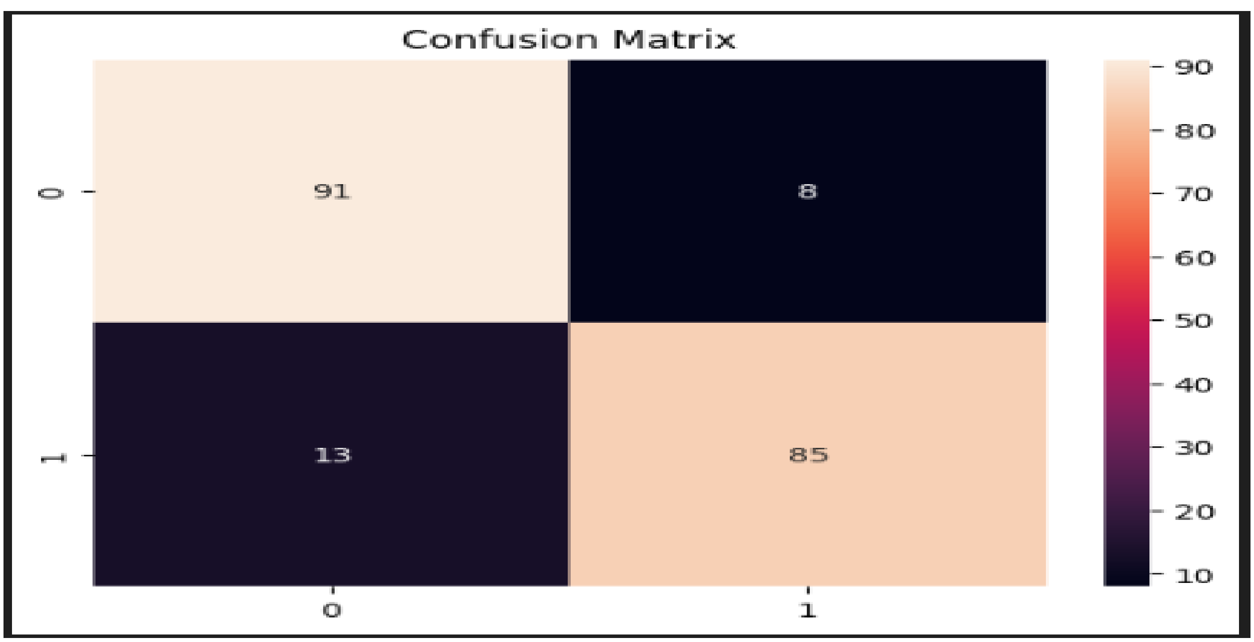
```
from sklearn.svm import SVC

svm_model = SVC(kernel='linear')
svm_model.fit(X_train, Y_train)

svm_predictions = svm_model.predict(X_test)
print("SVM Accuracy:", accuracy_score(Y_test, svm_predictions))

[39]
...  SVM Accuracy: 0.9035532994923858
```

## 3.7 Evaluation Metrics

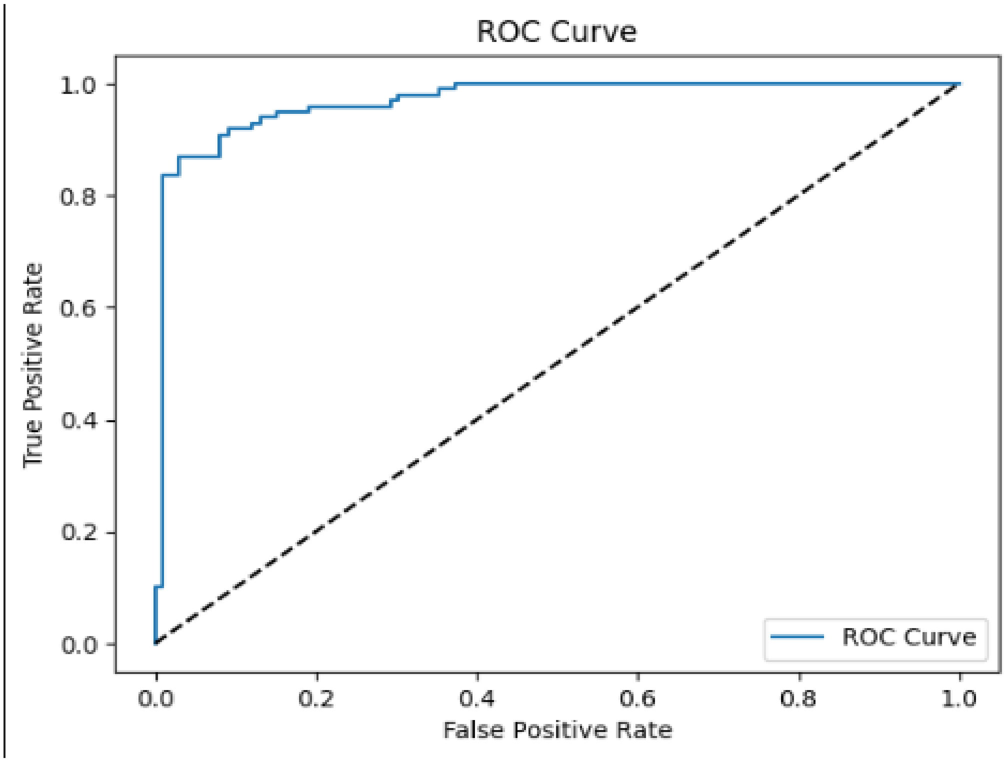
Evaluation was done using:

- **Accuracy**
- **Precision**
- **Recall**
- **F1 Score**
- **Confusion Matrix**

```
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score, roc_curve

cm = confusion_matrix(Y_test, X_test_prediction)
sns.heatmap(cm, annot=True, fmt='d')
plt.title("Confusion Matrix")
plt.show()

print(classification_report(Y_test, X_test_prediction))

y_probs = model.predict_proba(X_test)[:,1]
fpr, tpr, thresholds = roc_curve(Y_test, y_probs)
plt.plot(fpr, tpr, label="ROC Curve")
plt.plot([0,1], [0,1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()

print("AUC Score:", roc_auc_score(Y_test, y_probs))
```



```
...           precision    recall  f1-score   support

           0       0.88      0.92      0.90        99
           1       0.91      0.87      0.89        98

    accuracy                           0.89       197
   macro avg       0.89      0.89      0.89       197
weighted avg       0.89      0.89      0.89       197
```




Due to class imbalance, **Precision** and **Recall** were prioritized over overall accuracy.

```
[31]
print('Accuracy on Training data : ', training_data_accuracy)
... Accuracy on Training data :  0.9428208386277002
```

```
[33]
print('Accuracy score on Test Data : ', test_data_accuracy)
... Accuracy score on Test Data :  0.8934010152284264
```

### 3.8 Training, Evaluating Model and Output of the Model

Each model was trained using an 80-20 train-test split. Their performances were compared based on the metrics.

```
[28]
model = LogisticRegression()
```

```
[29]
model.fit(X_train, Y_train)

... C:\Users\iCare\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-p
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(

... LogisticRegression
LogisticRegression()
```

```
[30]
X_train_prediction = model.predict(X_train)
training_data_accuracy = accuracy_score(X_train_prediction, Y_train)
```

```
[31]
print('Accuracy on Training data : ', training_data_accuracy)

... Accuracy on Training data :  0.9428208386277002
```

```
[32]
X_test_prediction = model.predict(X_test)
test_data_accuracy = accuracy_score(X_test_prediction, Y_test)
```

```
[33]
print('Accuracy score on Test Data : ', test_data_accuracy)

... Accuracy score on Test Data :  0.8934010152284264
```

## Prints Title and Menu:

- Displays a menu titled "Credit Card Fraud Detection" with 5 predefined transaction sets for the user to choose from.

```
for i in range(5):
    print(f"{i+1}. Transaction Set {i+1}")
```

## User Input for Transaction Set:

- Prompts user to select a transaction profile (1–5).
- Uses a while loop to validate input and prevent errors.
- Selects the corresponding feature set (v_values) from sample_inputs.

```
while True:
    try:
        choice = int(input("Your choice (1-5): "))
        if 1 <= choice <= 5:
            v_values = sample_inputs[choice - 1]
            break
        else:
            print("Please select a number from 1 to 5.")
    except ValueError:
        print("Invalid input. Enter a number.")
```

## User Input for Transaction Amount:

- Asks the user to input the monetary value of the transaction.
- Validates the input to ensure it is a float.

```
while True:
    try:
        amount = float(input("Enter transaction amount: "))
        break
    except ValueError:
        print("Invalid input. Please enter a valid number.")
```

## Fixed Transaction Time:

- Sets a constant time value (e.g., 100000) to simulate a time feature in the transaction.

```
time = 100000
```

## Feature Vector Creation:

- Combines time, v_values (V1–V28), and amount into a single feature array for prediction.

```
features = np.array([[time] + v_values + [amount]])
```

## Prediction:

- Uses a previously trained machine learning model (model.predict) to classify the transaction as either fraudulent (1) or legitimate (0).

```
prediction = model.predict(features)
```

## Outputs Result:

- Displays a warning  if fraud is detected.
- Displays a check  if the transaction is legitimate.

```
if prediction[0] == 1:
    print("\n FRAUD DETECTED!")
else:
    print("\n Transaction is Legitimate.")
```

```
💳 Credit Card Fraud Detection

Choose a predefined transaction profile (1-5):
1. Transaction Set 1
2. Transaction Set 2
3. Transaction Set 3
4. Transaction Set 4
5. Transaction Set 5

✅ Transaction is Legitimate.
C:\Users\iCare\AppData\Local\Packages\PythonSoftwareFoundation.Pyth
  warnings.warn(
```

## 4. Analysis of the Project

The analysis confirmed that:

- Class imbalance significantly affects model performance.
- Evaluation using Precision and Recall is essential.
- Scaling features improves SVM performance.
- Ensemble methods (e.g., Random Forest, not yet implemented) may provide better accuracy and generalization.

## 5. Conclusion

The project successfully demonstrated how machine learning can be used to detect credit card fraud. The results highlight the challenges of imbalanced classification problems and the importance of preprocessing and evaluation. Future improvements may include using ensemble techniques, SMOTE for balancing, and real-time detection strategies.