INTERNSHIP PROJECT REPORT

Submitted to

INLIGHNX GLOBAL PVT. LTD.

Bangalore, India

Date of Submission

15/01/26

Credi Card Fraud Detection

Model: LogisticRegression

Prepared by

Nadar Deepthi Chenthil

AI & ML Intern

ITID5802

Internship Duration

15/11/25 – 15/01/26

Index

Link to GitHub with the .ipynb file containing the implementation code, all dataset .csv files and documentation:

https://github.com/Deepthi-Nadar/ML-project-1/tree/main

CrediCardFraudDetection

1.Introduction

Credit card fraud detection is a paramount safety measure for financial institutions, utilizing advanced machine learning to identify and prevent unauthorized transactions. In an era where digital payments are the norm, the ability to distinguish between a legitimate purchase and a fraudulent attempt in milliseconds is critical.

- Problem Definition: The system is designed to monitor transaction patterns and flag anomalies that deviate from a user's typical spending behavior.
- Objective: To build a high-precision model that accurately identifies fraudulent "Class 1" transactions while ensuring legitimate "Class 0" transactions are not blocked.
- Input: The model processes numerical transaction features including the transaction amount, time elapsed, and 28 PCA-transformed variables (V1-V28).
- Output: A binary classification result indicating whether a transaction is "Fraud" (1) or "Legitimate" (0).
- Type of Problem: This is a Supervised Learning task addressed as a Binary Classification problem, often dealing with highly imbalanced data.
- Business Importance: It helps banks reduce massive financial losses, protects cardholders from identity theft, and automates a verification process that would be impossible to do manually.
- Pipeline Approach: The project follows a strict end-to-end Machine Learning pipeline from data cleaning and scaling to model persistence using Pickle.

2. Libraries and Dependencies

The notebook uses the following Python 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
```

The code begins by importing libraries that handle data structures, visualization, and mathematical modeling.

- **Pandas & NumPy:** Used for data frame manipulation and performing high-level mathematical functions on arrays.
- **Matplotlib & Seaborn:** Essential for generating plots like the Class distribution bar chart and Correlation Heatmaps.
- **Scikit-Learn (Preprocessing):** Includes StandardScaler to normalize the data and train_test_split to divide the data.
- **LogisticRegression:** The core algorithm chosen for its efficiency in binary probability-based classification.
- **Metrics:** Tools like accuracy_score and confusion_matrix are used to measure the success of the model.

3. Dataset Overview

The dataset contains transactions made by credit cards in September 2013 by European cardholders. This section explains the source, privacy measures, and the nature of the data being analyzed to ensure high model performance.

Key Features and Descriptions

Each feature plays a specific role in the mathematical calculation of the fraud probability score.

Feature Name	Description	Statistical Relevance
Time	Seconds elapsed from the first entry.	Used to detect "velocity" fraud (too many charges in short time).
V1 - V28	PCA-transformed numerical features.	Represents hidden patterns like location, merchant type, and device ID.
Amount	The monetary value of the transaction.	Significant for determining the "weight" of the fraud impact.
Class	The label (0 = Genuine, 1 = Fraud).	The ground truth used to calculate the Loss Function during training.

Mathematical Intuition for Data Characteristics

To understand the data's nature before training, we look at the **Imbalance Ratio**. This is the most important characteristic of a fraud dataset.

Equation to find the Fraud Percentage:

Fraud %=(Total TransactionsCount of Class 1)×100

Calculation for this dataset:

Fraud %=(284,807492)×100≈0.17%

Equation for Feature Variance (used in PCA):

σ2=N∑(xi−μ)2

The V1-V28 features are generated by maximizing this variance (σ2) to ensure that the most informative patterns of the original sensitive data are preserved while keeping the identities of the users private.

4. Data Loading

The dataset is loaded using the **pandas** library, which provides powerful data manipulation capabilities.

Steps Performed

1. Import the dataset using `pd.read_csv()`

```
# loading the dataset to a Pandas DataFrame
credit_card_data = pd.read_csv('/content/creditcard.csv')
```

Store the data in a pandas DataFrame

2. Perform initial inspection using:

- Initial Structural Check (.head()): This method displays the first 5 rows, allowing us to verify the successful loading of the V1-V28 columns and the initial "Time" and "Amount" values.

```
credit_card_data.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27
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
1	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
2	1	-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
3	1	-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
4	2	-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

5 rows × 31 columns

```
credit_card_data.head()
```

V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
72781	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	0.0
66151	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	0.0
40163	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	0.0
85226	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	0.0
77737	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	0.0

- Endpoint Inspection (.tail()): By viewing the last 5 rows, we ensure the data was not truncated during the download or import process and that the "Class" labels are present.

```
credit_card_data.tail()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26
95157	65189	1.142364	-1.090177	0.511068	-0.792356	-1.160044	0.134166	-0.997184	0.236721	-0.583865	...	0.399798	0.815983	-0.172587	-0.298962	0.310993	-0.077793
95158	65190	-2.145283	1.280406	0.014577	-2.003358	1.479294	4.673049	-2.008023	-2.933663	0.036717	...	0.276433	-0.038006	0.085076	1.049870	0.489570	1.045371
95159	65190	-3.715715	3.870511	-1.525809	0.082535	-0.244009	-0.901579	0.708830	0.070491	2.349423	...	-0.327180	0.573451	0.266379	0.040564	-0.175983	-0.494220
95160	65190	-5.164795	4.510526	-0.994499	-1.110853	-0.913228	-0.889076	0.373572	0.361552	3.841062	...	-0.908623	-1.154210	0.300341	-0.102776	0.817800	0.201861
95161	65191	-1.430966	1.192670	1.237388	1.074059	-0.997949	0.687186	-1.045570	1.012203	0.095426	...	NaN	NaN	NaN	NaN	NaN	NaN

5 rows × 31 columns

```
credit_card_data.tail()
```

V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
)0177	0.511068	-0.792356	-1.160044	0.134166	-0.997184	0.236721	-0.583865	...	0.399798	0.815983	-0.172587	-0.298962	0.310993	-0.077793	0.011223	0.019004	106.00	0.0
30406	0.014577	-2.003358	1.479294	4.673049	-2.008023	-2.933663	0.036717	...	0.276433	-0.038006	0.085076	1.049870	0.489570	1.045371	-0.363337	-0.222526	34.61	0.0
'0511	-1.525809	0.082535	-0.244009	-0.901579	0.708830	0.070491	2.349423	...	-0.327180	0.573451	0.266379	0.040564	-0.175983	-0.494220	0.257349	-0.309196	0.89	0.0
0526	-0.994499	-1.110853	-0.913228	-0.889076	0.373572	0.361552	3.841062	...	-0.908623	-1.154210	0.300341	-0.102776	0.817800	0.201861	2.384092	1.576142	7.18	0.0
)2670	1.237388	1.074059	-0.997949	0.687186	-1.045570	1.012203	0.095426	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Metadata Analysis (.info()): This step identifies the data types (mostly float64) and confirms that the memory usage is optimized for the large volume of transactions.

```
credit_card_data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 95162 entries, 0 to 95161
Data columns (total 31 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Time    95162 non-null  int64
 1   V1      95162 non-null  float64
 2   V2      95162 non-null  float64
 3   V3      95162 non-null  float64
 4   V4      95162 non-null  float64
 5   V5      95162 non-null  float64
 6   V6      95162 non-null  float64
 7   V7      95162 non-null  float64
 8   V8      95162 non-null  float64
 9   V9      95162 non-null  float64
 10  V10     95162 non-null  float64
 11  V11     95162 non-null  float64
 12  V12     95162 non-null  float64
 13  V13     95161 non-null  float64
 14  V14     95161 non-null  float64
 15  V15     95161 non-null  float64
 16  V16     95161 non-null  float64
 17  V17     95161 non-null  float64
 18  V18     95161 non-null  float64
 19  V19     95161 non-null  float64
 20  V20     95161 non-null  float64
 21  V21     95161 non-null  float64
 22  V22     95161 non-null  float64
 23  V23     95161 non-null  float64
```

```
 24  V24     95161 non-null  float64
 25  V25     95161 non-null  float64
 26  V26     95161 non-null  float64
 27  V27     95161 non-null  float64
 28  V28     95161 non-null  float64
 29  Amount  95161 non-null  float64
 30  Class   95161 non-null  float64
dtypes: float64(30), int64(1)
memory usage: 22.5 MB
```

- Integrity Verification (.isnull().sum()): This command checks for missing values across all 31 columns; for this specific dataset, the result should be 0 across all features.

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

	0
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	1
V14	1
V15	1
V16	1
V17	1
V18	1
V19	1
V20	1
V21	1
V22	1
V23	1
V24	1
V25	1
V26	1

- ○ Class Distribution Analysis (.value_counts()): This is the most critical step where we count the occurrences of 0 (Legitimate) vs 1 (Fraud) to understand the dataset's imbalance.

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

Class	count
0.0	94944
1.0	217

dtype: int64

5. Feature Scaling

Scaling is required because the "Amount" and "Time" columns have much higher ranges than the V1-V28 features.

- Standardization: Transforms data to have a mean of 0 and a standard deviation of 1.
- Equation used:

$z=\sigma x-\mu$

(Where x is the value, μ is the mean, and σ is the standard deviation).

- Consistency: Applying the scaler ensures the "Amount" of $10,000 doesn't bias the model more than a small V1 value.

This Dataset is highly imbalance

0 is Normal Transcation

1 is fraudulent Transaction

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

```
print(legit.shape)
print(fraud.shape)
```

```
(94944, 31)
(217, 31)
```

```
legit.Amount.describe()
```

	Amount
count	94944.000000
mean	98.685374
std	267.432993
min	0.000000
25%	7.600000
50%	26.720000
75%	89.752500
max	19656.530000

dtype: float64

```
fraud.Amount.describe()
```

	Amount
count	217.000000
mean	109.784286
std	243.927116
min	0.000000
25%	1.000000
50%	7.580000
75%	99.990000
max	1809.680000

dtype: float64

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

Class	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V20	V21	V22	V23	V24	V25
0.0	41232.783620	-0.25064	-0.047145	0.696491	0.15125	-0.270069	0.097914	-0.093292	0.048787	-0.032333	...	0.043401	-0.031938	-0.107340	-0.037071	0.009933	0.13190
1.0	35870.354839	-6.16618	4.217965	-8.087819	4.99603	-4.444345	-1.845983	-6.417590	2.796291	-2.953535	...	0.354728	0.731128	-0.127758	-0.247689	-0.102596	0.20696

2 rows × 30 columns

V2	V3	V4	V5	V6	V7	V8	V9	...	V20	V21	V22	V23	V24	V25	V26	V27	V28	Amount
145	0.696491	0.15125	-0.270069	0.097914	-0.093292	0.048787	-0.032333	...	0.043401	-0.031938	-0.107340	-0.037071	0.009933	0.13190	0.026607	-0.000819	0.001332	98.685374
965	-8.087819	4.99603	-4.444345	-1.845983	-6.417590	2.796291	-2.953535	...	0.354728	0.731128	-0.127758	-0.247689	-0.102596	0.20696	0.097699	0.532822	0.035472	109.784286

Under - Sampling

Build a sample dataset containing similar distribution of normal transactions and Fraudulent Transactions

Numbere of Fraudulent Transaction is 492

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

Concatenating two DataFrames

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

```
new_dataset.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26
86	55	-0.773450	0.853112	0.818254	-0.236070	0.803463	-1.438728	0.799479	-0.007989	-0.761090	...	0.035362	-0.116890	-0.178926	0.400155	-0.026231	0.165156
5745	6092	-1.048587	0.899177	2.488911	0.201191	-0.300900	0.103546	-0.096744	0.202863	1.876991	...	-0.311043	-0.335932	-0.299486	-0.157590	0.309509	0.367028
50848	44673	-0.339741	1.149495	1.302433	0.060004	0.056230	-0.985470	0.715795	-0.084107	-0.425411	...	-0.262419	-0.672120	-0.022079	0.340221	-0.154449	0.071231
94772	65023	-1.540715	1.575182	-0.224569	1.041269	0.048300	-0.830932	0.262160	0.661573	-0.295118	...	-0.017627	0.108239	0.099999	-0.067347	-0.229928	-0.340178
38847	39578	-0.752845	1.045901	0.890107	0.935372	0.140186	-0.433494	0.323012	0.393536	-0.858277	...	0.255480	0.609056	-0.063802	0.091658	-0.297356	-0.305996

5 rows × 31 columns

```
new_dataset.head()
```

V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
53112	0.818254	-0.236070	0.803463	-1.438728	0.799479	-0.007989	-0.761090	...	0.035362	-0.116890	-0.178926	0.400155	-0.026231	0.165156	0.027762	0.132980	0.76	0.0
99177	2.488911	0.201191	-0.300900	0.103546	-0.096744	0.202863	1.876991	...	-0.311043	-0.335932	-0.299486	-0.157590	0.309509	0.367028	0.385191	0.188772	7.87	0.0
49495	1.302433	0.060004	0.056230	-0.985470	0.715795	-0.084107	-0.425411	...	-0.262419	-0.672120	-0.022079	0.340221	-0.154449	0.071231	0.247692	0.098646	0.89	0.0
75182	-0.224569	1.041269	0.048300	-0.830932	0.262160	0.661573	-0.295118	...	-0.017627	0.108239	0.099999	-0.067347	-0.229928	-0.340178	0.043305	-0.085648	19.12	0.0
45901	0.890107	0.935372	0.140186	-0.433494	0.323012	0.393536	-0.858277	...	0.255480	0.609056	-0.063802	0.091658	-0.297356	-0.305996	0.066874	0.064944	18.40	0.0

```
new_dataset.tail()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26
92777	64093	-6.133987	2.941499	-5.593986	3.258845	-5.315512	-0.637328	-4.476488	1.695994	-1.606743	...	0.868340	0.793736	0.217347	-0.021985	0.145882	0.665088
93424	64412	-1.348042	2.522821	-0.782432	4.083047	-0.662280	-0.598776	-1.943552	-0.329579	-1.853274	...	1.079871	-0.352026	-0.218358	0.125866	-0.074180	0.179116
93486	64443	1.079524	0.872988	-0.303850	2.755369	0.301688	-0.350284	-0.042848	0.246625	-0.779176	...	-0.023255	-0.158601	-0.038806	-0.060327	0.358339	0.076984
93788	64585	1.080433	0.962831	-0.278065	2.743318	0.412364	-0.320778	0.041290	0.176170	-0.966952	...	-0.008996	-0.057036	-0.053692	-0.026373	0.400300	0.072828
94218	64785	-8.744415	-3.420468	-4.850575	6.606846	-2.800546	0.105512	-3.269801	0.940378	-2.558691	...	0.102913	0.311626	-4.129195	0.034639	-1.133631	0.272265

5 rows × 31 columns

```
new_dataset.tail()
```

V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
941499	-5.593986	3.258845	-5.315512	-0.637328	-4.476488	1.695994	-1.606743	...	0.868340	0.793736	0.217347	-0.021985	0.145882	0.665088	-1.684186	0.310195	294.90	1.0
522821	-0.782432	4.083047	-0.662280	-0.598776	-1.943552	-0.329579	-1.853274	...	1.079871	-0.352026	-0.218358	0.125866	-0.074180	0.179116	0.612580	0.234206	1.00	1.0
372988	-0.303850	2.755369	0.301688	-0.350284	-0.042848	0.246625	-0.779176	...	-0.023255	-0.158601	-0.038806	-0.060327	0.358339	0.076984	0.018936	0.060574	0.00	1.0
962831	-0.278065	2.743318	0.412364	-0.320778	0.041290	0.176170	-0.966952	...	-0.008996	-0.057036	-0.053692	-0.026373	0.400300	0.072828	0.027043	0.063238	0.00	1.0
420468	-4.850575	6.606846	-2.800546	0.105512	-3.269801	0.940378	-2.558691	...	0.102913	0.311626	-4.129195	0.034639	-1.133631	0.272265	1.841307	-1.796363	720.38	1.0

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

	count
Class	
0.0	492
1.0	217

dtype: int64

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

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V20	V21	V22	V23	V24	V25
Class																	
0.0	40389.132114	-0.201232	-0.070129	0.728321	0.087316	-0.291074	0.175446	-0.100706	-0.009746	-0.111128	...	0.076743	-0.041832	-0.106820	-0.033419	0.032150	0.149419
1.0	35870.354839	-6.166180	4.217965	-8.087819	4.996030	-4.444345	-1.845983	-6.417590	2.796291	-2.953535	...	0.354728	0.731128	-0.127758	-0.247689	-0.102596	0.206960

2 rows × 30 columns

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

V3	V4	V5	V6	V7	V8	V9	...	V20	V21	V22	V23	V24	V25	V26	V27	V28	Amount
0.728321	0.087316	-0.291074	0.175446	-0.100706	-0.009746	-0.111128	...	0.076743	-0.041832	-0.106820	-0.033419	0.032150	0.149419	-0.006949	-0.000812	0.009584	104.308638
-8.087819	4.996030	-4.444345	-1.845983	-6.417590	2.796291	-2.953535	...	0.354728	0.731128	-0.127758	-0.247689	-0.102596	0.206960	0.097699	0.532822	0.035472	109.784286

6. Data Pre-processing

Preparing the data for the model involves cleaning and structuring.

- Feature/Target Separation: Separating the independent variables (X) from the target class (y).
- Handling Imbalance: Since there are very few fraud cases, a "Sub-sample" or "Under-sampling" technique is often shown.
- Logic: Create a new DataFrame containing all 492 fraud cases and a random 492 non-fraud cases to create a 50/50 balanced dataset.

- Outcome: This ensures the model doesn't simply learn to predict "0" for every transaction to achieve high accuracy.

Splitting the data into Features & Targets

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

```
print(X)
```

```
        Time        V1        V2        V3        V4        V5        V6  \
86        55 -0.773450  0.853112  0.818254 -0.236070  0.803463 -1.438728
5745    6092 -1.048587  0.899177  2.488911  0.201191 -0.300900  0.103546
50848  44673 -0.339741  1.149495  1.302433  0.060004  0.056230 -0.985470
94772  65023 -1.540715  1.575182 -0.224569  1.041269  0.048300 -0.830932
38847  39578 -0.752845  1.045901  0.890107  0.935372  0.140186 -0.433494
...      ...       ...       ...       ...       ...       ...       ...
92777  64093 -6.133987  2.941499 -5.593986  3.258845 -5.315512 -0.637328
93424  64412 -1.348042  2.522821 -0.782432  4.083047 -0.662280 -0.598776
93486  64443  1.079524  0.872988 -0.303850  2.755369  0.301688 -0.350284
93788  64585  1.080433  0.962831 -0.278065  2.743318  0.412364 -0.320778
94218  64785 -8.744415 -3.420468 -4.850575  6.606846 -2.800546  0.105512

             V7        V8        V9  ...       V20       V21       V22  \
86     0.799479 -0.007989 -0.761090  ... -0.100858  0.035362 -0.116890
5745  -0.096744  0.202863  1.876991  ...  0.199320 -0.311043 -0.335932
50848  0.715795 -0.084107 -0.425411  ...  0.122899 -0.262419 -0.672120
94772  0.262160  0.661573 -0.295118  ... -0.122001 -0.017627  0.108239
38847  0.323012  0.393536 -0.858277  ... -0.046757  0.255480  0.609056
...         ...       ...       ...  ...       ...       ...       ...
92777 -4.476488  1.695994 -1.606743  ... -0.815086  0.868340  0.793736
93424 -1.943552 -0.329579 -1.853274  ...  0.348896  1.079871 -0.352026
93486 -0.042848  0.246625 -0.779176  ... -0.252115 -0.023255 -0.158601
93788  0.041290  0.176170 -0.966952  ... -0.172659 -0.008996 -0.057036
94218 -3.269801  0.940378 -2.558691  ... -1.818315  0.102913  0.311626
```

```
              V23       V24       V25       V26       V27       V28  Amount
86     -0.178926  0.400155 -0.026231  0.165156  0.027762  0.132980    0.76
5745   -0.299486 -0.157590  0.309509  0.367028  0.385191  0.188772    7.87
50848  -0.022079  0.340221 -0.154449  0.071231  0.247692  0.098646    0.89
94772   0.099999 -0.067347 -0.229928 -0.340178  0.043305 -0.085648   19.12
38847  -0.063802  0.091658 -0.297356 -0.305996  0.066874  0.064944   18.40
...          ...       ...       ...       ...       ...       ...     ...
92777   0.217347 -0.021985  0.145882  0.665088 -1.684186  0.310195  294.90
93424  -0.218358  0.125866 -0.074180  0.179116  0.612580  0.234206    1.00
93486  -0.038806 -0.060327  0.358339  0.076984  0.018936  0.060574    0.00
93788  -0.053692 -0.026373  0.400300  0.072828  0.027043  0.063238    0.00
94218  -4.129195  0.034639 -1.133631  0.272265  1.841307 -1.796363  720.38

[709 rows x 30 columns]
```

```
print(Y)
```

```
86       0.0
5745     0.0
50848    0.0
94772    0.0
38847    0.0
        ...
92777    1.0
93424    1.0
93486    1.0
93788    1.0
94218    1.0
Name: Class, Length: 709, dtype: float64
```

7. Train-Test Split

The dataset is split to evaluate how the model performs on unseen data.

- Training Set: Usually 80% of the data is used to "teach" the model the characteristics of fraud.
- Testing Set: 20% is held back to act as the "Final Exam."
- Random State: Setting random_state=2 ensures that every time you run the code, the split remains the same.

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, stratify=Y, random_state=2)
```

```
print(X.shape, X_train.shape, X_test.shape)
```

```
(709, 30) (567, 30) (142, 30)
```

8. Model Building

This is where the mathematical engine is initialized and trained.

- Selection: Logistic Regression is used because it is the "gold standard" for binary classification.
- Model Fit: The .fit(X_train, Y_train) command calculates the weights (w) and bias (b).
- The Equation:

$P(y=1)=1+e-(wX+b)1$

The model outputs a probability between 0 and 1 using this Sigmoid function.

```
model = LogisticRegression()

# training the Logistic Regression Model with Training Data
model.fit(X_train, Y_train)

/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logistic.py:465: ConvergenceWarning: lbfgs failed to converge (status=1):
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()
```

9. Model Evaluation

The output here shows how well the model learned.

- Accuracy Score: The ratio of correct predictions to total predictions.
- Equation:

$Accuracy=TP+TN+FP+FNTP+TN$

- Confusion Matrix: A 2×2 grid showing:
  - TP (True Positive): Correctly caught fraud.
  - FP (False Positive): Legitimate customer accidentally blocked.
- Precision/Recall: In fraud, Recall is most important because it measures how many actual frauds we caught.

$Recall=TP+FNTP$

```
# accuracy on training data
X_train_prediction = model.predict(X_train)
training_data_accuracy = accuracy_score(X_train_prediction, Y_train)

print('Accuracy on Training data : ', training_data_accuracy)

Accuracy on Training data :  0.9735449735449735

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

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

Accuracy score on Test Data :  0.9507042253521126
```

10. Project Workflow Summary

- Data Collection: Gathering historical CSV logs.
- Pre-processing: Balancing the classes and scaling features.
- Splitting: Dividing into Train and Test sets.
- Modeling: Training Logistic Regression.

- Evaluation: Checking Accuracy, Precision, and Recall.
- Deployment: Saving the model for real-time use.

11. Conclusion

- Summary: The project successfully built a system that can detect fraud with high accuracy even with imbalanced data.
- Finding: Logistic Regression provided a balanced approach between speed and predictive power.
- Limitation: While accurate on the sample, real-world fraud patterns evolve, requiring periodic model retraining.
- Future Work: Implementing advanced techniques like Isolation Forests or Neural Networks could further reduce False Positives.