Credit Card Fraud Detection

By: Abhishek Sinha **Intern ID : ITID1965**

Submitted To: Inlign Tech as my internship project for the month 1/7/25 to 1/8/25.

Abstract

Through this project we are aiming to detect fraudulent credit card transactions using machine learning techniques. The dataset for this project is available here on Kaggle.

Due to the presence of significant class imbalance in the dataset — where fraudulent transactions form a tiny fraction of the data (< 0.1%), I have built the model using robust preprocessing steps, oversampling techniques, and XGBoost as the final classifier.

Performance is evaluated with appropriate metrics for imbalanced datasets.

Problem Statement

Develop a system to detect fraudulent credit card transactions in real-time using transaction data.

Dataset Overview

- **Source**: Kaggle Credit Card Fraud Dataset
- **Size**: 284,807 transactions
- **Features**:
  - 28 PCA-anonymized features (V1 to V28)
  - Amount – transaction amount
  - Time – seconds since the first transaction
  - Class – target (1 = Fraud, 0 = Legit)

```
print(data.info())
data.head()
```

```
<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
```

Exploratory Data Analysis (EDA)

- No missing values found.

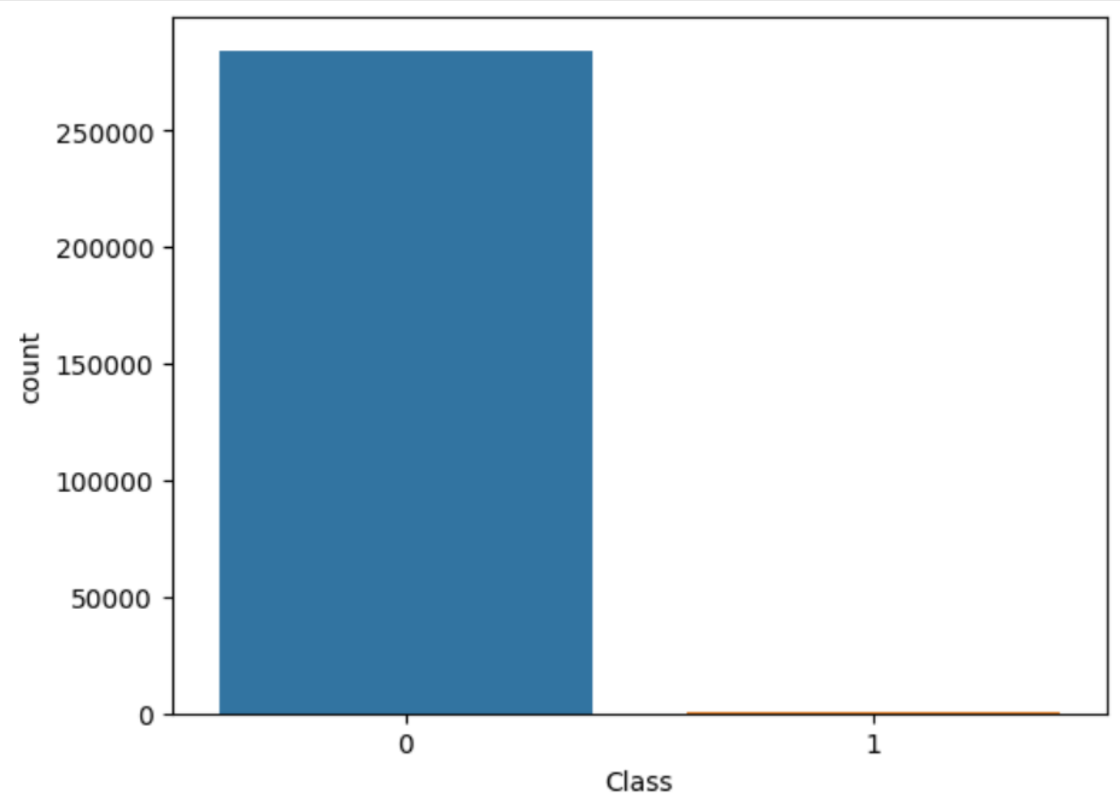
```python
data.isnull().sum()
```

```
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
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

- Fraudulent transactions are significantly fewer than legitimate ones.

```python
# Class distribution check
sns.countplot(x = 'Class', data = data)
```

```
<Axes: xlabel='Class', ylabel='count'>
```



Above countplot shows that about only 0.17% of the data is fraudulent and the rest are all legitimate.

- Fraudulent transactions tend to have higher mean transaction amounts.

```
#Lets seperate the data on the basis of fraud or legit
dataFraud = data[data['Class'] == 1]
dataLegit = data[data['Class'] == 0]

#We can start by checking the amount in both cases
print("Details about Fraud amount:\n",dataFraud['Amount'].describe())
print("\nDetails about Legit amount:\n",dataLegit['Amount'].describe())
```

```
Details about Fraud amount:
 count     492.000000
mean      122.211321
std       256.683288
min         0.000000
25%         1.000000
50%         9.250000
75%       105.890000
max      2125.870000
Name: Amount, dtype: float64

Details about Legit amount:
 count    284315.000000
mean         88.291022
std         250.105092
min           0.000000
25%           5.650000
50%          22.000000
75%          77.050000
max       25691.160000
```

From above details, we can see that the mean amount for frauds are much higher than the amount for the legitimate transactions.

- Some features showed mild correlation with fraud class (both positive and negative).

Lets take the min correlation required for a feature to be

1. Negative Correlation: -0.1
2. Positive Correlation: +0.1

```
#We take out the
correlated_features = data.corr()[['Class']][(data.corr()['Class'] <= -0.1) | (data.corr()['Class'] >= 0.1)]
columns = correlated_features.index
print(columns)
```

```
Index(['V1', 'V3', 'V4', 'V7', 'V10', 'V11', 'V12', 'V14', 'V16', 'V17', 'V18',
       'Class'],
      dtype='object')
```

- Boxplots revealed the presence of outliers in multiple columns.

Preprocessing & Feature Engineering

- I took two feature sets here:
  - All features
  - Correlated features

    I've done this so as to later evaluate whether correlated features perform better or all features perform better.
- Standardization and dimensionality reduction were not required due to PCA-processed features.

Handling Imbalanced Data

- Applied **SMOTE** from smote_variants library to oversample the minority class.

  I oversampled the fraudulent transaction data, but only after splitting it, since it can lead to data leakage to the test set. This might give rise to overfitting, so we need to take care of it here.

```
# split into trainset and testset
X_train, X_test, Y_train, Y_test = train_test_split(X,
                                                    Y,
                                                    test_size = 0.2,
                                                    random_state = 42)
```

```
# begin oversample for trainSet
smote = sv.SMOTE()
X_train_resampled, Y_train_resampled = smote.sample(X_train.values, Y_train.values)
```

```
2025-07-31 05:54:37,343:INFO:SMOTE: Running sampling via ('SMOTE', "{'proportion': 1.0, 'n_neighbors': 5, 'nn_params': {}, 'n_job
s': 1, 'ss_params': {'n_dim': 2, 'simplex_sampling': 'random', 'within_simplex_sampling': 'random', 'gaussian_component': {}}, 'r
andom_state': None, 'class_name': 'SMOTE'}")
2025-07-31 05:54:37,348:INFO:NearestNeighborsWithMetricTensor: NN fitting with metric minkowski
2025-07-31 05:54:37,350:INFO:NearestNeighborsWithMetricTensor: kneighbors query minkowski
2025-07-31 05:54:37,407:INFO:SMOTE: simplex sampling with n_dim 2
```

- This balanced the class distribution without affecting test data (to prevent leakage).
- The data was returned in numpy array so I later also converted in back to DataFrame.

```
# Convert back to dataframe
X_train_resampled = pd.DataFrame(X_train_resampled, columns = X.columns)
Y_train_resampled = pd.Series(Y_train_resampled)
```

Modeling with XGBoost

- I've used an XGBoost classifier on both
  - the oversampled training data.
  - the original unsampled data.

    This was done to see whether the oversampling had a positive effect on the model's ability to detect fraudulent transactions.

- Key model parameters: learning rate, number of estimators, max depth, etc. (tuned through optuna).
- The model was selected for its performance with tabular data and imbalanced learning.

<u>Oversampled XGBoost optuna objective function</u>

XGBoost

```python
# first we define our optuna objective function

#with oversampled data
def objective(trial):
    param = {
    'objective':'binary:logistic',
    'eval_metric':'auc',
    'tree_method':'hist',
    'scale_pos_weight':trial.suggest_float('scale_pos_weight',1,100),
    'max_depth':trial.suggest_int('max_depth',3,12),
    'learning_rate':trial.suggest_float('learning_rate',0.01, 0.3),
    'subsample':trial.suggest_float('subsample',0.5,1.0),
    'colsample_bytree':trial.suggest_float('colsample_bytree', 0.5, 1.0),
    'lambda':trial.suggest_float('lambda',1e-3, 10.0),
    'alpha':trial.suggest_float('alpha',1e-3,10.0)
    }

    model = XGBClassifier(**param, use_label_encoder = False)
    model.fit(X_train_resampled, Y_train_resampled)
    preds = model.predict(X_test)
    return precision_score(Y_test, preds)
```

Unsampled XGBoost optuna objective function

```python
#without oversampled data
def objective_no_osmpl(trial):
    param = {
    'objective':'binary:logistic',
    'eval_metric':'auc',
    'tree_method':'hist',
    'scale_pos_weight':trial.suggest_float('scale_pos_weight',1,100),
    'max_depth':trial.suggest_int('max_depth',3,12),
    'learning_rate':trial.suggest_float('learning_rate',0.01, 0.3),
    'subsample':trial.suggest_float('subsample',0.5,1.0),
    'colsample_bytree':trial.suggest_float('colsample_bytree', 0.5, 1.0),
    'lambda':trial.suggest_float('lambda',1e-3, 10.0),
    'alpha':trial.suggest_float('alpha',1e-3,10.0)
    }

    model = XGBClassifier(**param, use_label_encoder = False)
    model.fit(X_train, Y_train)
    preds = model.predict(X_test)
    return precision_score(Y_test, preds)
```

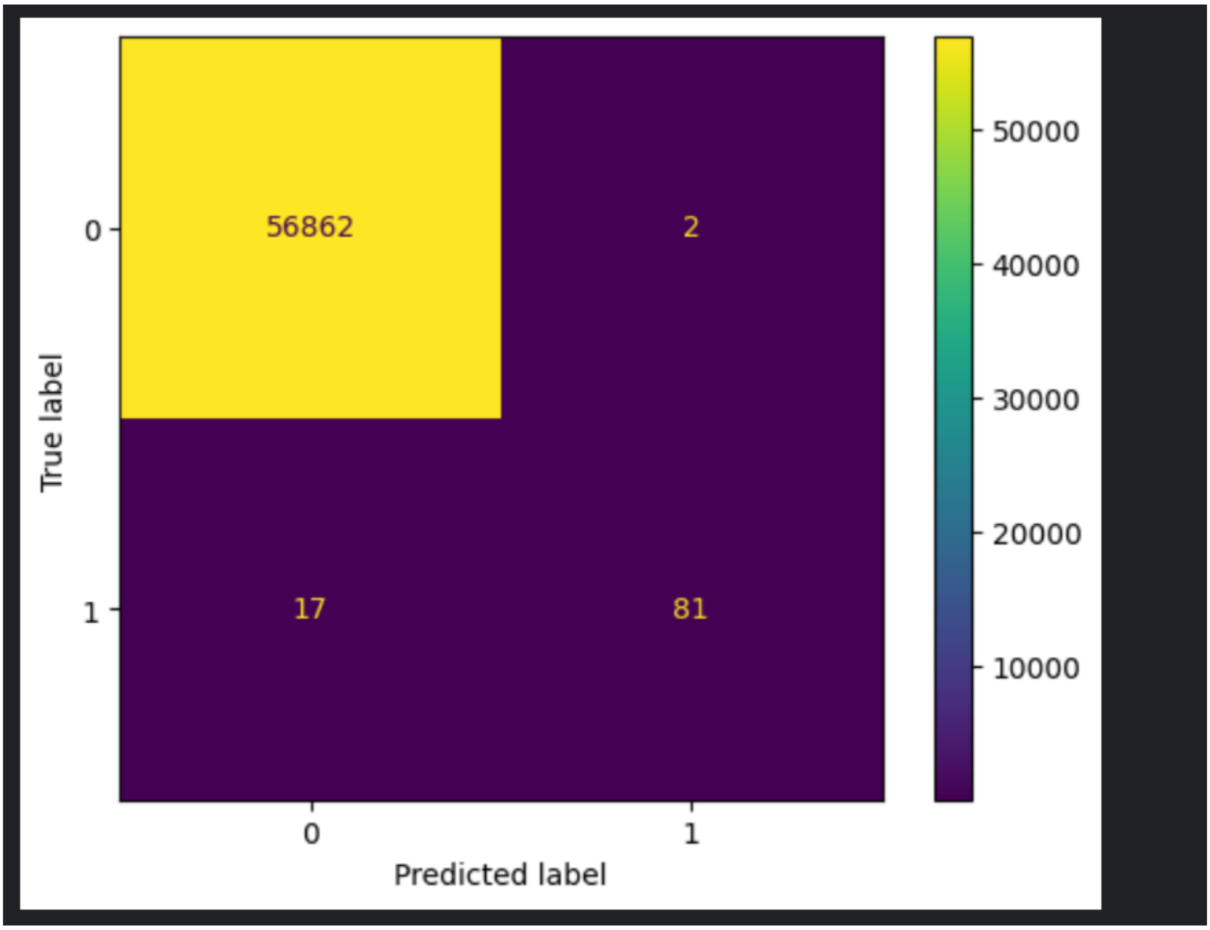
- The best classifier model obtained was one without any oversampling, meaning that there was no need of oversampling the data here.

```
print(f"Model: XGBClassifier with hyperparams tuned with Optuna(no oversampling)")
print(f"Acuracy Score : {accuracy_score(Y_test, final_preds_no_osmpl)}")
print(f"ROC_AUC Score : {roc_auc_score(Y_test, final_preds_proba_no_osmpl)}")
print(f"Precision Score : {precision_score(Y_test, final_preds_no_osmpl)}")
print(f"Recall Score : {recall_score(Y_test, final_preds_no_osmpl)}")
print(f"Classification Report : \n{classification_report(Y_test, final_preds_no_osmpl)}")
cm = confusion_matrix(Y_test, final_preds_no_osmpl)
cm_disp = ConfusionMatrixDisplay(confusion_matrix = cm)
cm_disp.plot()
plt.show()
```

```
Model: XGBClassifier with hyperparams tuned with Optuna(no oversampling)
Acuracy Score : 0.9996664442961974
ROC_AUC Score : 0.9850046799811651
Precision Score : 0.9759036144578314
Recall Score : 0.826530612244898
Classification Report :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.98      0.83      0.90        98

    accuracy                           1.00     56962
   macro avg       0.99      0.91      0.95     56962
weighted avg       1.00      1.00      1.00     56962
```



Evaluation Metrics

1. Accuracy Score : 0.99
   While a high accuracy score seems to be very nice, in this case it can be easily misinterpreted. This is since +99% of the data is for legitimate transactions, if the model classifies all of them as legitimate, it can reach ~99% accuracy.

2. ROC_AUC Score : 0.98
   roc_auc score being high (98%) tells us that our model is very much proficient in detecting both the classes and differeing betwen them.

3. Precision Score : 0.98
   We have a high precision score of 91%. This means that of all the transactions that our model classified as frauds, 91% were truly fraudulent transactions. We have drastically reduced false positives through this.

4. Recall Score : 0.83
   This tells us that out of all the present fraudulent transactions in the dataset, we managed to classify ~83% of them.

5. F1 -Score(fraud class) : 0.90
   This tells us that our model is very much well balanced for deployement.

This model gives an optimal trade-off between false positives and missed frauds, making it a strong model for real-world deployment.
Its performance suggests robust generalization even without oversampling, indicating the effectiveness of hyperparameter tuning and the inherent power of XGBoost.

Real-Time Simulation

A real-time fraud detection simulation is proposed, where:

- Transactions are streamed one by one (simulated via time.sleep()).
- Each transaction is passed to the trained model.
- Predictions and probabilities are printed or visualized in real time.
- Integrated into a Streamlit dashboard.

RealTime Simulation of Classification model

> Now we can conduct the Realtime simulation of the model when it is classifying.

```
# We need to conduct realtime simulation of the data
# So i will split part of it since dataset is very large
realtime_split = X_test

# also taking best model
best_model = final_model_no_osmpl
```

```python
# define function for realtime detection
def realtime_analysis():
    legit = 0
    fraud = 0
    for i in range(len(realtime_split)):
        current = realtime_split.iloc[i:i+1]

        pred = best_model.predict(current)[0]
        proba = best_model.predict_proba(current)[0][1]

        if(pred == 1):
            fraud += 1
            print(f"Transaction #{i+1}: {'FRAUDULENT' if pred else 'LEGITIMATE'} | Confidence: {proba:.2f}")
        else:
            legit += 1

    print(f"==================Final Results================\nLegitimate Transactions : {legit}\nFraudulent Transactions : {fraud}")
```

```python
# Commence Realtime Analysis
realtime_analysis()
```

```
Transaction #48679: FRAUDULENT | Confidence: 0.98
Transaction #50501: FRAUDULENT | Confidence: 1.00
Transaction #51017: FRAUDULENT | Confidence: 0.98
Transaction #51623: FRAUDULENT | Confidence: 0.99
Transaction #53466: FRAUDULENT | Confidence: 1.00
Transaction #53921: FRAUDULENT | Confidence: 1.00
Transaction #54606: FRAUDULENT | Confidence: 0.98
Transaction #55410: FRAUDULENT | Confidence: 1.00
Transaction #55460: FRAUDULENT | Confidence: 1.00
Transaction #55765: FRAUDULENT | Confidence: 1.00
Transaction #55800: FRAUDULENT | Confidence: 1.00
Transaction #56197: FRAUDULENT | Confidence: 1.00
Transaction #56283: FRAUDULENT | Confidence: 1.00
Transaction #56289: FRAUDULENT | Confidence: 0.97
Transaction #56468: FRAUDULENT | Confidence: 0.60
===================Final Results================
Legitimate Transactions : 56878
Fraudulent Transactions : 84
```

Conclusion

- The model shows strong potential in detecting credit card fraud.
- SMOTE-based oversampling helped address class imbalance effectively.
- XGBoost proved to be a suitable model for this tabular, imbalanced data scenario.

Future Work

- Deploy real-time fraud prediction via a REST API or dashboard.
- Use time-series modeling or transaction sequence analysis.
- Apply anomaly detection techniques for unsupervised fraud spotting.