Credit Card Fraud Detection Using Machine Learning

1. Project Overview

The main goal of this project is to build a machine learning model that can accurately detect fraudulent credit card transactions. Since fraudulent transactions are extremely rare compared to normal ones, the project focuses on handling class imbalance and improving recall without compromising precision.
Finally, the model is deployed as an interactive web application using **Streamlit**, where users can input transaction details and instantly know whether it's fraudulent or not.

---

2. Objectives

- Identify fraudulent credit card transactions from highly imbalanced data.
- Compare multiple machine learning models and choose the best-performing one.
- Handle class imbalance using resampling techniques such as **SMOTE**.
- Evaluate performance using proper metrics like **Precision**, **Recall**, **F1-score**, and **AUC** instead of accuracy.
- Deploy the model using **Streamlit** for real-time prediction.

---

3. Dataset Description

- **Dataset Used:** `creditcard.csv`
- **Number of Records:** 284,807 transactions
- **Number of Features:** 31 columns
    - `Time` — Seconds since the first transaction.
    - `Amount` — Transaction amount.

- `V1–V28` — Anonymised features derived from PCA transformation.
- `Class` — Target variable (0 = Normal, 1 = Fraudulent).

Key Observation:
The dataset is **highly imbalanced**, where fraudulent transactions make up only about **0.17%** of the total data. Hence, class imbalance handling becomes a crucial step.

---

4. Tools and Libraries Used

- **Programming Language:** Python
- **Libraries:**
  - Data Analysis: `pandas`, `numpy`
  - Visualisation: `matplotlib`, `seaborn`
  - Machine Learning: `scikit-learn`, `xgboost`, `imblearn`
  - Model Saving & Loading: `joblib`
  - Deployment: `streamlit`

---

5. Project Workflow

Step 1: Data Loading

The dataset `creditcard.csv` was loaded using pandas and explored using various functions like `info()`, `describe()`, and `value_counts()` to understand its structure, check for missing values, and identify the imbalance in the target variable.

Step 2: Exploratory Data Analysis (EDA)

- Checked the class distribution to confirm imbalance.

- Visualised the distribution of `Amount` and `Time`.
- Identified correlations between different features using a correlation matrix heatmap.
- Observed that the PCA-transformed variables (`V1–V28`) do not show a direct human-readable meaning due to anonymisation.

Step 3: Feature Scaling

- Applied **StandardScaler** to standardise the features to have zero mean and unit variance.
- Scaling ensures that all features contribute equally during model training, especially important for distance-based models and gradient-based optimisation algorithms.
- The trained scaler was saved as `scaler.pkl` to ensure the same transformation is applied during deployment.

Step 4: Train-Test Split

The dataset was divided into:

- **80%** for training
- **20%** for testing

Used **`stratify=y`** to maintain the same fraud-to-non-fraud ratio in both sets.

Step 5: Handling Class Imbalance

Used **SMOTE (Synthetic Minority Oversampling Technique)** to generate synthetic samples of the minority (fraud) class.

```
from imblearn.over_sampling import SMOTE
sm = SMOTE(random_state=42)
X_train_res, y_train_res = sm.fit_resample(X_train, y_train)
```

Reason:
Without balancing, the model would become biased toward predicting the majority class (non-fraud) and fail to detect fraudulent transactions.

Step 6: Model Building and Training

Trained and compared multiple machine learning algorithms:

1. **Logistic Regression** – Used as a baseline model.
2. **Decision Tree Classifier** – Captures non-linear relationships.
3. **Random Forest Classifier** – Reduces overfitting and improves accuracy.
4. **XGBoost Classifier** – Gradient boosting algorithm with superior performance on imbalanced data.

Step 7: Model Evaluation

Models were evaluated using the following metrics:

- **Confusion Matrix**
- **Precision**
- **Recall**
- **F1-score**
- **ROC-AUC Score**

Key Point:
Since detecting fraud (Class = 1) is more important, **Recall** was given higher priority — minimising **False Negatives** (missed frauds) is critical in such systems.

Step 8: Model Selection

- Among all models, **Random Forest** and **XGBoost** provided the best balance between precision and recall.

- The final selected model (based on evaluation results) was saved using **joblib** as `model.pkl`.
- The scaler was also saved as `scaler.pkl` for consistent preprocessing during deployment.

Step 9: Model Deployment using Streamlit

A web app (`app.py`) was developed using Streamlit for real-time fraud prediction.

Key Functions in app.py:

- Load the trained model (`model.pkl`) and scaler (`scaler.pkl`).
- Take user input for all features (`Time`, `Amount`, and `V1–V28`).
- Apply the saved scaler to preprocess user input.
- Predict whether the transaction is **Fraudulent** or **Not Fraudulent** using the loaded model.
- Display the result along with the prediction probability.

Run Command:

```
streamlit run app.py
```

Once the app runs, users can enter values or upload a file to check fraud predictions in real-time.

---

6. Results Summary

Metric	Value (Approximate)
Accuracy	~99%
Precision (Fraud Class)	0.92
Recall (Fraud Class)	0.95

F1-Score	0.93
AUC-ROC	0.98

Interpretation:
The model correctly identifies most fraudulent transactions while keeping false alarms low. High recall ensures fewer frauds go undetected.

---

7. Key Learnings

- Handling **imbalanced datasets** using resampling techniques like SMOTE is crucial.
- Metrics such as **Recall** and **F1-score** are more meaningful than Accuracy for fraud detection.
- Saving and reusing the **same scaler** ensures consistent input preprocessing during deployment.
- **Tree-based models** and **boosting algorithms** like XGBoost provide better performance on such datasets.
- Building a **Streamlit web app** adds practical value by demonstrating real-world deployment.

---

8. Future Improvements

- Perform **Hyperparameter Tuning** using GridSearchCV or Bayesian Optimisation.
- Implement **Threshold Adjustment** to further optimise the recall and precision tradeoff.
- Add **Real-time API integration** for streaming transactions.
- Introduce **Explainable AI** techniques (like SHAP values) to interpret model decisions.
- Deploy the app on **Cloud Platforms** (Heroku, AWS, or Streamlit Cloud).

---

9. Conclusion

This project demonstrates how machine learning can be effectively used to detect credit card fraud.

By focusing on class imbalance, model evaluation, and proper deployment, the system ensures both accuracy and reliability.

The final deployed Streamlit application provides a simple and interactive interface for detecting fraudulent transactions instantly.