Project Title: Credit Card Fraud Detection using Machine Learning (Random Forest)

Objective: To build a robust model capable of detecting highly rare fraudulent credit card transactions, prioritizing the identification of genuine fraud cases.

This code sets up the project by importing all necessary Python tools (pandas, sklearn, SMOTE). It then safely loads the large credit card transaction dataset from Kaggle.

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
from imblearn.over_sampling import SMOTE
import kagglehub
```

Data Collection and Loading

Source: The project uses a dataset from Kaggle titled "Credit Card Transactions Fraud Detection Dataset"

Datasets: fraudTest.csv and fraudTrain.csv

```
print("Loading dataset...")


# --- Load the dataset using kagglehub ---
try:
    from kagglehub import KaggleDatasetAdapter
    file_path = "creditcard.csv"
    # Using a different publicly available Kaggle dataset for the credit card fraud task.
    df = kagglehub.load_dataset(
        KaggleDatasetAdapter.PANDAS,
        "mlg-ulb/creditcardfraud",
        file_path,
    )
    print("Dataset loaded successfully from KaggleHub.")
except Exception as e:
    print(f"Error loading from KaggleHub: {e}. Attempting to load from a local file.")
    try:
        df = pd.read_csv('creditcard.csv')
        print("Dataset loaded successfully from local file.")
    except FileNotFoundError:
        print("Error: 'creditcard.csv' not found. Please ensure the file is in the correct directory.")
        exit()
```

```
Loading dataset...
/tmp/ipython-input-261276384.py:9: DeprecationWarning: Use dataset_load() instead of load_dataset(). load_dataset() will b
  df = kagglehub.load_dataset(
Using Colab cache for faster access to the 'creditcardfraud' dataset.
Dataset loaded successfully from KaggleHub.
```

Exploratory Data Analysis (EDA)

The primary task here is to inspect the class distribution of the target variable (Class). The output confirms the extreme problem we must solve: only 0.17% of transactions are fraud. This imbalance necessitates the use of advanced techniques like SMOTE to prevent the model from becoming biased toward the majority class.

```
# --- Exploratory Data Analysis (EDA) ---
print("\n--- Class Distribution ---")
print(df['Class'].value_counts())
print(f"\nFraudulent transactions make up {round(df['Class'].value_counts()[1]/len(df) * 100, 2)}% of the dataset.")
```

```
--- Class Distribution ---
Class
0    284315
1       492
Name: count, dtype: int64

Fraudulent transactions make up 0.17% of the dataset.
```

Data Preprocessing (Feature Scaling)

Before modeling, the data must be standardized. We use StandardScaler to perform feature scaling on the 'Amount' and 'Time' columns. This normalization process ensures these two features, which have not been transformed by PCA, do not disproportionately influence the model simply because of their larger numerical ranges.

```
# --- Data Preprocessing ---
print("\n--- Preprocessing Data ---")

# Separate features (X) and target (y)
X = df.drop('Class', axis=1)
y = df['Class']

# Standardize the 'Amount' and 'Time' features
scaler = StandardScaler()
X['Amount'] = scaler.fit_transform(X[['Amount']])
X['Time'] = scaler.fit_transform(X[['Time']])
```

```
--- Preprocessing Data ---
```

Data Splitting

This step partitions the data into training (80%) and testing (20%) subsets. The critical component here is the use of stratify=y. This ensures that the extremely rare fraudulent transactions are equally distributed between the two sets, guaranteeing that the model evaluation on the test set is fair and reflective of real-world class proportions.

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
print(f"Original training set size: {X_train.shape[0]} samples")
print(f"Original testing set size: {X_test.shape[0]} samples")
```

```
Original training set size: 227845 samples
Original testing set size: 56962 samples
```

Handling Imbalance with SMOTE

This block implements the most vital step: solving the data imbalance. We apply SMOTE (Synthetic Minority Over-sampling Technique) to the training data only. SMOTE mathematically generates synthetic samples for the minority class (fraud) until the dataset is perfectly balanced, providing the Random Forest with sufficient positive examples to accurately learn fraud patterns.

```
# --- Handling Imbalanced Data with SMOTE ---
print("\n--- Applying SMOTE to Balance the Training Data ---")
sm = SMOTE(random_state=42)
X_train_res, y_train_res = sm.fit_resample(X_train, y_train)

print(f"Balanced training set size: {X_train_res.shape[0]} samples")
print(f"New class distribution: \n{y_train_res.value_counts()}")
```

```
--- Applying SMOTE to Balance the Training Data ---
Balanced training set size: 454902 samples
New class distribution:
Class
0    227451
1    227451
Name: count, dtype: int64
```

Random Forest Model Training and Prediction

We train the highly effective Random Forest Classifier on our newly balanced training data. Random Forest, an ensemble learning method, is inherently robust against overfitting. We set the parameter n_estimators=50 to optimize the balance between high performance and fast training time. Finally, we generate predictions (y_pred) on the original, untouched test set.

```
# --- Random Forest Model Training and Evaluation ---
print("\n--- Training Random Forest Classifier ---")
# Use a Random Forest Classifier, which is highly effective for this problem
rf_classifier = RandomForestClassifier(n_estimators=50, random_state=42)
rf_classifier.fit(X_train_res, y_train_res)
print("\n--- Evaluating Random Forest on the Test Set ---")
y_pred = rf_classifier.predict(X_test)
```

```
--- Training Random Forest Classifier ---

--- Evaluating Random Forest on the Test Set ---
```

Final Evaluation and Results

The final block evaluates the model using the Confusion Matrix and Classification Report. While overall Accuracy (0.9994) is high, the model's true success is measured by its Recall (0.84 or 84%) for the fraud class, which indicates that the model successfully caught the vast majority of actual fraud cases. This high Recall rate achieves the project's objective of minimizing missed financial threats.

```
# Evaluate the model using a confusion matrix and classification report
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, zero_division=0)
cm = confusion_matrix(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}")
print("Classification Report:")
print(report)
print("Confusion Matrix:")
print(cm)
```

```
Accuracy: 0.9994
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.84      0.84      0.84        98

    accuracy                           1.00     56962
   macro avg       0.92      0.92      0.92     56962
weighted avg       1.00      1.00      1.00     56962

Confusion Matrix:
[[56848    16]
 [   16    82]]
```