Credit Card Fraud Detection – End-to-End ML Pipeline (Production-Oriented)

Project Overview

This notebook demonstrates a production-oriented machine learning pipeline for credit card fraud detection using an imbalanced dataset.

Key focus:

- ETL and data preprocessing
- Handling class imbalance (SMOTE)
- Model training and hyperparameter tuning
- Evaluation metrics suitable for fraud detection
- Deployment and monitoring considerations

Note: For demonstration purposes, we generate a small sample dataset. The full Kaggle dataset workflow is included as commented code.

```
# Step 1: Import Libraries
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
from imblearn.over_sampling import SMOTE

import warnings
warnings.filterwarnings("ignore")
```

```
# Step 2: Dataset Acquisition (Kaggle API – Optional / Production Demo)
"""
This section demonstrates how to download the full Credit Card Fraud dataset from Kaggle.
It is commented out to keep the notebook lightweight and reproducible.

Skills demonstrated:
- Installing Kaggle package
- Uploading API credentials (kaggle.json)
- Setting up local environment for Kaggle API
- Downloading and unzipping datasets
- Loading the CSV into a DataFrame
"""

# 1 Install Kaggle package (only once)
# !pip install kaggle --quiet

# 2 Upload Kaggle API credentials (kaggle.json)
# from google.colab import files
# files.upload()  # Select your kaggle.json

# 3 Set up API credentials
# import os
# os.makedirs(os.path.expanduser("~/.kaggle"), exist_ok=True)
# !cp kaggle.json ~/.kaggle/
# !chmod 600 ~/.kaggle/kaggle.json

# 4 Download and unzip dataset from Kaggle
# !kaggle datasets download -d mlg-ulb/creditcardfraud -p ./data --unzip

# 5 Load full dataset into a DataFrame
# import pandas as pd
# df_full = pd.read_csv("./data/creditcard.csv")
# print("Full dataset shape:", df_full.shape)

"""
Notes:
- All commands are commented to avoid execution issues in environments without Kaggle access.
- In a real production workflow, this allows you to automatically fetch the latest dataset.
- A smaller sample CSV is generated from df_full for HR-friendly demos in Notebook 1.
```

```
"""
```

```
'\nNotes:\n- All commands are commented to avoid execution issues in environments without Kaggle access.\n- In a real productio
n workflow, this allows you to automatically fetch the latest dataset.\n- A smaller sample CSV is generated from df_full for HR
-friendly demos in Notebook 1.\n'
```

```
# Step 2.1: Sample Data for Demonstration
# To keep this notebook lightweight and reproducible, we generate a small sample dataset.

np.random.seed(42)

X_sample = np.random.randn(1000, 30)
y_sample = np.random.choice([0, 1], size=1000, p=[0.98, 0.02])

df_sample = pd.DataFrame(X_sample, columns=[f"V{i}" for i in range(1, 31)])
df_sample["Class"] = y_sample

df_sample.head()
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V22	V23
0	0.496714	-0.138264	0.647689	1.523030	-0.234153	-0.234137	1.579213	0.767435	-0.469474	0.542560	...	-0.225776	0.067528
1	-0.601707	1.852278	-0.013497	-1.057711	0.822545	-1.220844	0.208864	-1.959670	-1.328186	0.196861	...	-0.385082	-0.676922
2	-0.479174	-0.185659	-1.106335	-1.196207	0.812526	1.356240	-0.072010	1.003533	0.361636	-0.645120	...	0.357113	1.477894
3	0.097078	0.968645	-0.702053	-0.327662	-0.392108	-1.463515	0.296120	0.261055	0.005113	-0.234587	...	-0.026514	0.060230
4	0.791032	-0.909387	1.402794	-1.401851	0.586857	2.190456	-0.990536	-0.566298	0.099651	-0.503476	...	1.307143	-1.607483

5 rows × 31 columns

```
# Step 3: Data Cleaning & ETL Preparation
# Basic data quality checks:

df_sample.info()
df_sample.isnull().sum().head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 31 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   V1      1000 non-null   float64
 1   V2      1000 non-null   float64
 2   V3      1000 non-null   float64
 3   V4      1000 non-null   float64
 4   V5      1000 non-null   float64
 5   V6      1000 non-null   float64
 6   V7      1000 non-null   float64
 7   V8      1000 non-null   float64
 8   V9      1000 non-null   float64
 9   V10     1000 non-null   float64
 10  V11     1000 non-null   float64
 11  V12     1000 non-null   float64
 12  V13     1000 non-null   float64
 13  V14     1000 non-null   float64
 14  V15     1000 non-null   float64
 15  V16     1000 non-null   float64
 16  V17     1000 non-null   float64
 17  V18     1000 non-null   float64
 18  V19     1000 non-null   float64
 19  V20     1000 non-null   float64
 20  V21     1000 non-null   float64
 21  V22     1000 non-null   float64
 22  V23     1000 non-null   float64
 23  V24     1000 non-null   float64
 24  V25     1000 non-null   float64
 25  V26     1000 non-null   float64
 26  V27     1000 non-null   float64
 27  V28     1000 non-null   float64
 28  V29     1000 non-null   float64
 29  V30     1000 non-null   float64
 30  Class   1000 non-null   int64  
dtypes: float64(30), int64(1)
memory usage: 242.3 KB
```

	0
V1	0
V2	0
V3	0
V4	0
V5	0

dtype: int64

Observations:

No missing values detected.

Features are numerical and anonymized (PCA-like structure).

Dataset structure is suitable for direct ML modeling after scaling.

```
# Step 3.1: ETL Pipeline Function
def etl_pipeline(df, target_col="Class"):
    """
    Production-style ETL pipeline:
    - Separate features and target
    - Scale numerical features
    """
    df = df.copy()
    X = df.drop(target_col, axis=1)
    y = df[target_col]

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

    return X_scaled, y

# Apply ETL pipeline
X_processed, y_processed = etl_pipeline(df_sample)
```

```
# Step 4: Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X_processed,
    y_processed,
    test_size=0.2,
    stratify=y_processed,
    random_state=42)
```

```
# Step 5: Handle Class Imbalance with SMOTE
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

pd.Series(y_train_resampled).value_counts()
```

Class	count
0	786
1	786

dtype: int64

Notes:

Fraud detection datasets are highly imbalanced.

SMOTE is applied only to training data to avoid data leakage.

```
# Step 6: Model Training & Hyperparameter Tuning
log_reg = LogisticRegression(max_iter=1000)

param_grid = {
    "C": [0.01, 0.1, 1, 10],    "penalty": ["l2"]}

grid_search = GridSearchCV(
    log_reg,
    param_grid,
    scoring="roc_auc",
    cv=5,
    n_jobs=-1)

grid_search.fit(X_train_resampled, y_train_resampled)

best_model = grid_search.best_estimator_
grid_search.best_params_
```

```
{'C': 0.1, 'penalty': 'l2'}
```

```
# Step 7: Model Evaluation
y_pred = best_model.predict(X_test)
y_prob = best_model.predict_proba(X_test)[:, 1]

print("Classification Report:")
print(classification_report(y_test, y_pred))

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

roc_auc = roc_auc_score(y_test, y_prob)
print("ROC-AUC Score:", roc_auc)
```

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

           0       0.98      0.76      0.85       197
           1       0.00      0.00      0.00         3

    accuracy                           0.74       200
   macro avg       0.49      0.38      0.43       200
weighted avg       0.97      0.74      0.84       200
```

```
Confusion Matrix:
[[149  48]
 [  3   0]]
ROC-AUC Score: 0.626057529610829
```

> ### Evaluation Focus
>
> - **ROC-AUC** is prioritized over accuracy due to class imbalance.
> - **Confusion matrix** helps assess false negatives, which are costly in fraud detection.

> **Note:** The output shows that the sample data is highly imbalanced, resulting in a recall of 0.
> In practice, when using the full dataset, techniques such as **SMOTE** or **class weighting** are needed to improve recall.