**fraud\_detection.py**

"""

fraud\_detection.py

Executable end-to-end credit card fraud detection:

- loads 'creditcard.csv' if present, otherwise synthesize data

- preprocesses, trains (XGBoost) with SMOTE pipeline

- evaluates using precision/recall/F1/ROC-AUC and precision-recall curve

- saves trained model and scaler

- starts a small Flask app for inference

Usage:

python fraud\_detection.py # trains and runs evaluation, then starts Flask API

"""

import os

import numpy as np

import pandas as pd

import joblib

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.metrics import (confusion\_matrix, classification\_report,

roc\_auc\_score, precision\_recall\_curve, auc)

from imblearn.over\_sampling import SMOTE

from imblearn.pipeline import Pipeline as ImbPipeline

from xgboost import XGBClassifier

# Flask for prediction endpoint

from flask import Flask, request, jsonify

# Data loading / generation

DATAFILE = "creditcard.csv" # if available, must have 'Class' column where 1=fraud, 0=normal

def load\_or\_synthesize():

if os.path.exists(DATAFILE):

print(f"Loading dataset from {DATAFILE} ...")

df = pd.read\_csv(DATAFILE)

if 'Class' not in df.columns:

raise ValueError("Expected 'Class' column in CSV with 1=fraud, 0=normal")

return df

else:

print(f"{DATAFILE} not found — synthesizing imbalanced dataset for demo...")

# synthesize data: 30 features, heavy class imbalance

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=20000, n\_features=30, n\_informative=10,

n\_redundant=10, n\_repeated=0, n\_classes=2,

weights=[0.995, 0.005], flip\_y=0.01, class\_sep=1.0,

random\_state=42)

cols = [f"V{i}" for i in range(1, 31)]

df = pd.DataFrame(X, columns=cols)

df['Amount'] = np.abs(np.random.randn(len(df)) \* 100) # pseudo amounts

df['Class'] = y

return df

# Preprocessing and split

def prepare\_data(df, test\_size=0.2, random\_state=42):

X = df.drop(columns=['Class'])

y = df['Class'].astype(int)

# Simple features: scale numerical features

numeric\_cols = X.select\_dtypes(include=[np.number]).columns.tolist()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=test\_size,

stratify=y, random\_state=random\_state)

print("Train class distribution:\n", y\_train.value\_counts(normalize=True))

print("Test class distribution:\n", y\_test.value\_counts(normalize=True))

return X\_train, X\_test, y\_train, y\_test, numeric\_cols

# Train model

def train\_model(X\_train, y\_train, numeric\_cols):

# Pipeline: scaler -> SMOTE -> XGBoost

scaler = StandardScaler()

smote = SMOTE(random\_state=42, sampling\_strategy='auto', n\_jobs=-1)

xgb = XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss', tree\_method='hist', random\_state=42)

pipeline = ImbPipeline(steps=[

('scaler', scaler),

('smote', smote),

('clf', xgb)

])

# We do a small grid search for n\_estimators and max\_depth; keep it small for speed

param\_grid = {

'clf\_\_n\_estimators': [50, 150],

'clf\_\_max\_depth': [3, 6],

'clf\_\_learning\_rate': [0.1]

}

grid = GridSearchCV(pipeline, param\_grid, scoring='f1', cv=3, n\_jobs=-1, verbose=1)

grid.fit(X\_train, y\_train)

print("Best params:", grid.best\_params\_)

print("Best CV f1:", grid.best\_score\_)

return grid.best\_estimator\_

# Evaluate

def evaluate\_model(model, X\_test, y\_test):

y\_pred = model.predict(X\_test)

y\_proba = model.predict\_proba(X\_test)[:,1]

print("\nClassification report:")

print(classification\_report(y\_test, y\_pred, digits=4))

roc\_auc = roc\_auc\_score(y\_test, y\_proba)

print(f"ROC AUC: {roc\_auc:.4f}")

# Precision-Recall curve and area

precision, recall, thresholds = precision\_recall\_curve(y\_test, y\_proba)

pr\_auc = auc(recall, precision)

print(f"PR AUC: {pr\_auc:.4f}")

# Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

print("Confusion matrix:")

print(cm)

# Plot PR curve

plt.figure(figsize=(6,5))

plt.plot(recall, precision, label=f'PR AUC = {pr\_auc:.4f}')

plt.xlabel('Recall')

plt.ylabel('Precision')

plt.title('Precision-Recall curve')

plt.legend()

plt.tight\_layout()

plt.savefig("pr\_curve.png")

print("Saved precision-recall curve to pr\_curve.png")

# Save artifacts

def save\_artifacts(model):

joblib.dump(model, "fraud\_xgb\_pipeline.joblib")

print("Saved model pipeline to fraud\_xgb\_pipeline.joblib")

# Simple Flask API

def create\_app(model):

app = Flask(\_\_name\_\_)

@app.route("/predict", methods=["POST"])

def predict():

"""

Expects JSON: {"data": [[feat1, feat2, ...], [...]]}

or a single record: {"data": [feat1, feat2, ...]}

Returns predicted class and probability for fraud (1).

"""

payload = request.get\_json(force=True)

if payload is None or "data" not in payload:

return jsonify({"error": "send JSON with 'data' field"}), 400

data = payload["data"]

arr = np.array(data)

# If single sample provided as 1D, reshape

if arr.ndim == 1:

arr = arr.reshape(1, -1)

try:

proba = model.predict\_proba(arr)[:,1]

pred = model.predict(arr)

except Exception as e:

return jsonify({"error": f"model prediction failed: {str(e)}"}), 500

results = []

for p, pr in zip(pred, proba):

results.append({"class": int(p), "fraud\_probability": float(pr)})

return jsonify({"results": results})

return app

# Main

def main():

df = load\_or\_synthesize()

X\_train, X\_test, y\_train, y\_test, numeric\_cols = prepare\_data(df)

model = train\_model(X\_train, y\_train, numeric\_cols)

evaluate\_model(model, X\_test, y\_test)

save\_artifacts(model)

# Start Flask app for demo (on localhost:5000)

app = create\_app(model)

print("\nStarting Flask API at http://127.0.0.1:5000")

print("POST JSON to /predict with key 'data'. Example single record:")

example = X\_test.iloc[0].tolist()

print(f"Example payload: {{'data': {example}}}")

# Run app

app.run(host="127.0.0.1", port=5000, debug=False)

if \_\_name\_\_ == "\_\_main\_\_":

main()