Identify Fraudulent Transactions in Credit Card Transactions Dataset

Names:

Mohammed AL-Weatishi 201967350

Ahmed AL-Johani 202065800

Ghaleb AL-Hashim 201925590

Gi Repository: Identify-Fraudulent-Transactions-in-Credit-Card-Transactions-Dataset-ICS474

Dataset Overview

The Credit Card Transactions Dataset is a comprehensive collection of credit card transaction records designed for fraud detection analysis. While it's a simulated dataset, it's structured to mirror real-world credit card transaction patterns and fraud scenarios.

- The dataset simulates real credit card transactions with both legitimate and fraudulent cases
- The dataset contains detailed transaction information including temporal, geographical, and demographic data
- The datasets specifically designed for developing fraud detection models

Overall Statistics

- Total Records: 616,615
- Memory Usage: 112.9+ MB
- Missing Values: Only in merch_zipcode (93,225 missing)
- Data Types Distribution:
  - float64: 7 columns
  - int64: 5 columns
  - object: 12 columns

```
import pandas as pd

df = pd.read_csv('credit_card_transactions02.csv',delimiter = ',')
display(df.head())
```

Merchant Information

7. **merchant**
   - Definition: Name of the merchant/business
   - Type: object (string)
8. **category**
   - Definition: Type of purchase/merchant category
   - Values: misc_net, grocery_pos, entertainment, gas_transport, etc.
   - Type: object (string)
9. **merch_lat**
   - Definition: Merchant's latitude location
   - Range: Typically within US boundaries
   - Type: float64
10. **merch_long**
    - Definition: Merchant's longitude location
    - Range: Typically within US boundaries
    - Type: float64
11. **merch_zipcode**
    - Definition: Merchant's ZIP code
    - Note: Has some null values (8461 non-null out of 9999)
    - Type: float64

Cardholder Information

12. **first**
    - Definition: Cardholder's first name
    - Type: object (string)
13. **last**
    - Definition: Cardholder's last name
    - Type: object (string)
14. **gender**
    - Definition: Cardholder's gender
    - Values: 'F' or 'M'
    - Type: object (string)

15. **dob**
    - Definition: Cardholder's date of birth
    - Type: object (string)
16. **job**
    - Definition: Cardholder's occupation
    - Type: object (string)

Transaction Information

1. **trans_date_trans_time**
    - Definition: Timestamp of the transaction
    - Type: object (datetime string)
2. **trans_num**
    - Definition: Unique transaction identifier
    - Type: object (string)
3. **unix_time**
    - Definition: Transaction timestamp in Unix format
    - Type: int64
4. **amt**
    - Definition: Transaction amount in dollars
    - Type: float64
5. **is_fraud**
    - Definition: Binary indicator of fraudulent transaction
    - Values: 0 (legitimate) or 1 (fraudulent)
    - Type: int64
    - Target variable for fraud detection

Card Information

6. **cc_num**
    - Definition: Credit card number
    - Type: float64

Location Information

17. **street**
    - Definition: Cardholder's street address
    - Type: object (string)
18. **city**
    - Definition: Cardholder's city
    - Type: object (string)
19. **state**
    - Definition: Cardholder's state
    - Type: object (string)
20. **zip**
    - Definition: Cardholder's ZIP code
    - Format: 5-digit US ZIP code
    - Type: int64
21. **lat**
    - Definition: Cardholder's latitude location
    - Range: Typically within US boundaries
    - Type: float64
22. **long**
    - Definition: Cardholder's longitude location
    - Range: Typically within US boundaries
    - Type: float64
23. **city_pop**
    - Definition: Population of the cardholder's city
    - Type: int64

Other

24. **Unnamed: 0**
    - Definition: Index column
    - Type: int64

Feature Selection

Columns to REMOVE

 # The first thing we do is to remove columns that doesn't affect our model in fraud detection

1) **Unnamed: 0** - This is likely just an index column with no predictive value
2) **first** - Personal names shouldn't influence fraud detection
3) **last** - Personal names shouldn't influence fraud detection
4) **street** - Specific street addresses are too granular and could lead to overfitting
5) **city** - The city_pop and lat/long provide better geographical indicators
6) **state** - Already represented by geographical coordinates
7) **zip** - Merchant zipcode provides sufficient location information
8) **dob** - Age might be relevant, but you can calculate it from dob if needed
9) **trans_num** - Transaction ID has no predictive value
10) **unix_time** - Already have trans_date_trans_time in a more usable format

 # Implementation

```
# List of columns to remove
columns_to_drop = ['Unnamed: 0', 'first', 'last', 'street', 'city', 'state',
                   'zip', 'dob', 'trans_num', 'unix_time']

# Drop the columns and create new dataframe
df_cleaned = df.drop(columns=columns_to_drop)

df_cleaned.head()
```

Checking & Handeling for Missing Values

Check for missing values

```python
# Check for missing values
print("Missing Values in Each Column:")
print(df_cleaned.isnull().sum())

# Calculate percentage of missing values
print("\nPercentage of Missing Values:")
print((df_cleaned.isnull().sum() / len(df_cleaned)) * 100)
```

```
Missing Values in Each Column:
trans_date_trans_time       0
cc_num                      0
merchant                    0
category                    0
amt                         0
gender                      0
lat                         0
long                        0
city_pop                    0
job                         0
merch_lat                   0
merch_long                  0
is_fraud                    0
merch_zipcode            1538
dtype: int64
```

 # seems that only merch_zipcode column contain missing values

Remove unknown values

```python
# Check how many rows have Unknown zipcode
print("Number of rows with Unknown zipcode:", (df_cleaned['merch_zipcode'] == 'Unknown').sum())
print("Total rows before removal:", len(df_cleaned))

# Remove rows where merch_zipcode is Unknown
df_cleaned = df_cleaned[df_cleaned['merch_zipcode'] != 'Unknown']

# Convert merch_zipcode to numeric type since all values are now numbers
df_cleaned['merch_zipcode'] = pd.to_numeric(df_cleaned['merch_zipcode'])

print("\nTotal rows after removal:", len(df_cleaned))

# Verify no more Unknown values
print("\nUnique values in merch_zipcode (first 5):", df_cleaned['merch_zipcode'].unique()[:5])

# Display first few rows to verify changes
print("\nFirst few rows after cleaning:")
print(df_cleaned.head())
```

Checking & Handeling for duplicate Values

 # Note: two records are considered duplicate if they share the same card, time and amount.

```python
# now we need to check for duplicate: (same card, time and amount)

# 1. Check for completely duplicate rows (all columns identical)
complete_duplicates = df_cleaned.duplicated().sum()
print("Complete duplicate rows:", complete_duplicates)

# 2. Check for suspicious transaction duplicates
# Same card, same amount, same timestamp (potential fraud or error)
suspicious_duplicates = df_cleaned.duplicated(
    subset=['cc_num', 'amt', 'trans_date_trans_time'],
    keep='first'
).sum()
print("\nSuspicious duplicates (same card, amount, and timestamp):", suspicious_duplicates)

# 3. Check for transactions per credit card
transactions_per_card = df_cleaned['cc_num'].value_counts()
print("\nTransactions per credit card:")
print("Min transactions:", transactions_per_card.min())
print("Max transactions:", transactions_per_card.max())
print("Mean transactions:", transactions_per_card.mean())

# 4. Check cards with unusually high number of transactions
print("\nCards with highest number of transactions:")
print(transactions_per_card.head())
```

```
Complete duplicate rows: 0

Suspicious duplicates (same card, amount, and timestamp): 0

Transactions per credit card:
Min transactions: 1
Max transactions: 459
Mean transactions: 31.945686900958467

Cards with highest number of transactions:
6010000000000000    459
213000000000000     389
180000000000000     288
3520000000000000    201
676000000000        197
Name: cc_num, dtype: int64
```

 # seems there is no duplicate records

comprehensive statistical analysis

Numerical Features Statistics

```
=== Numerical Features Statistics ===
               amt          lat          long      city_pop    merch_lat  \
count  9999.000000  9999.000000  9999.000000  9.999000e+03  9999.000000
mean     68.415379    38.594648   -90.617096  8.985949e+04    38.594499
std     111.533341     5.180174    14.459441  3.001342e+05     5.208998
min       1.010000    20.027100  -165.672300  2.300000e+01    19.165823
25%       9.690000    34.778900   -97.171400  7.410000e+02    34.848237
50%      48.670000    39.390000   -87.724600  2.395000e+03    39.373000
75%      83.095000    41.846700   -80.128400  1.909000e+04    41.899503
max    3178.510000    65.689900   -67.950300  2.906700e+06    66.645176
```

Transaction Amount Analysis

```
=== Transaction Amount Analysis ===
Total Transaction Volume: 684,085.37
Average Transaction Amount: 68.42
Median Transaction Amount: 48.67
Transaction Amount Std Dev: 111.53
Max Transaction Amount: 3,178.51
Min Transaction Amount: 1.01
```

Categorical Features Analysis

```
Transaction Categories Distribution:
gas_transport     1071
grocery_pos       1015
home               959
shopping_pos       877
kids_pets          835
shopping_net       739
personal_care      715
food_dining        703
entertainment      688
health_fitness     634
misc_pos           572
misc_net           546
grocery_net        350
travel             295
Name: category, dtype: int64
```

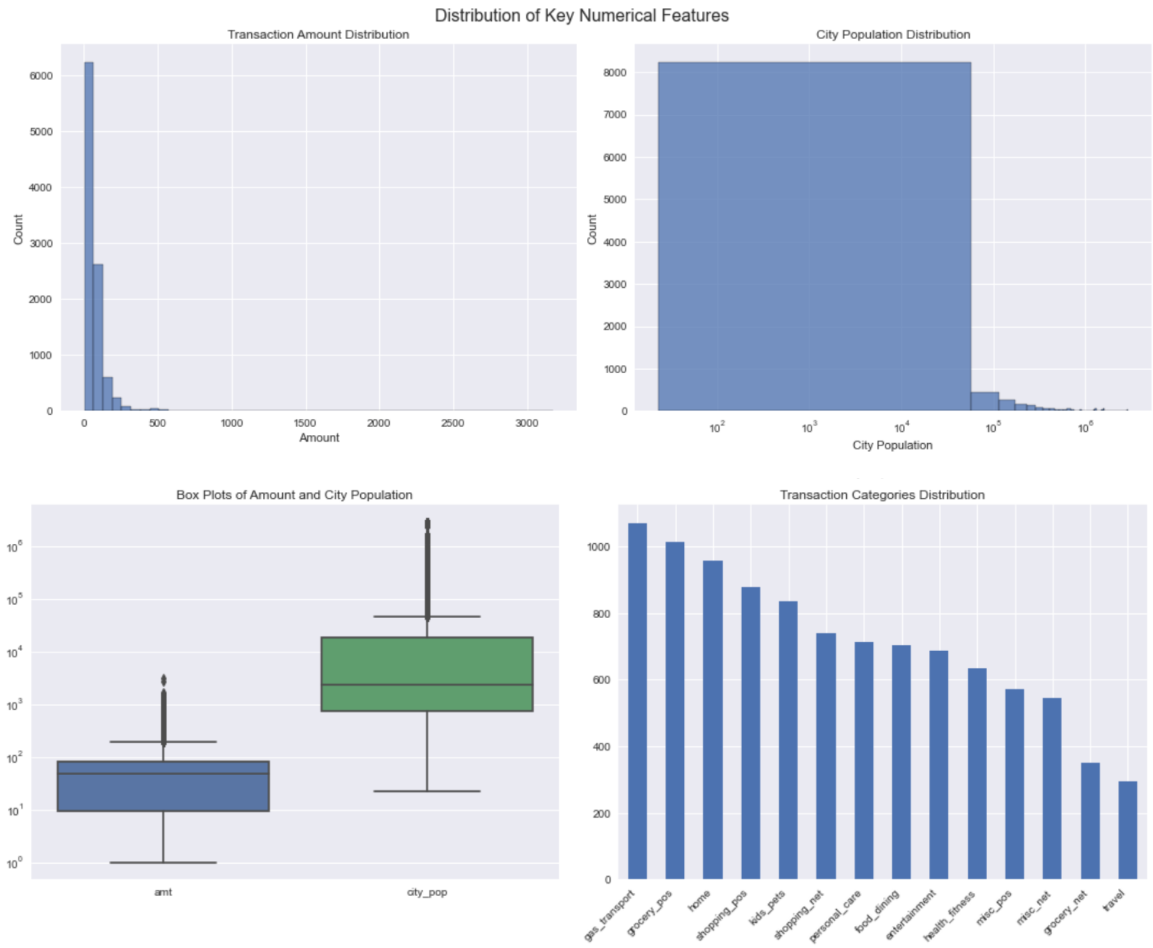
```
Top 10 Jobs:
Designer, ceramics/pottery     78
Exhibition designer            73
Film/video editor              69
Systems developer              69
Scientist, research (maths)    60
Copywriter, advertising        59
IT trainer                     59
Financial adviser              58
Barrister                      58
Comptroller                    58
Name: job, dtype: int64
```

Target Feature

```
Fraud Distribution (%):
0    99.529953
1     0.470047
Name: is_fraud, dtype: float64
```

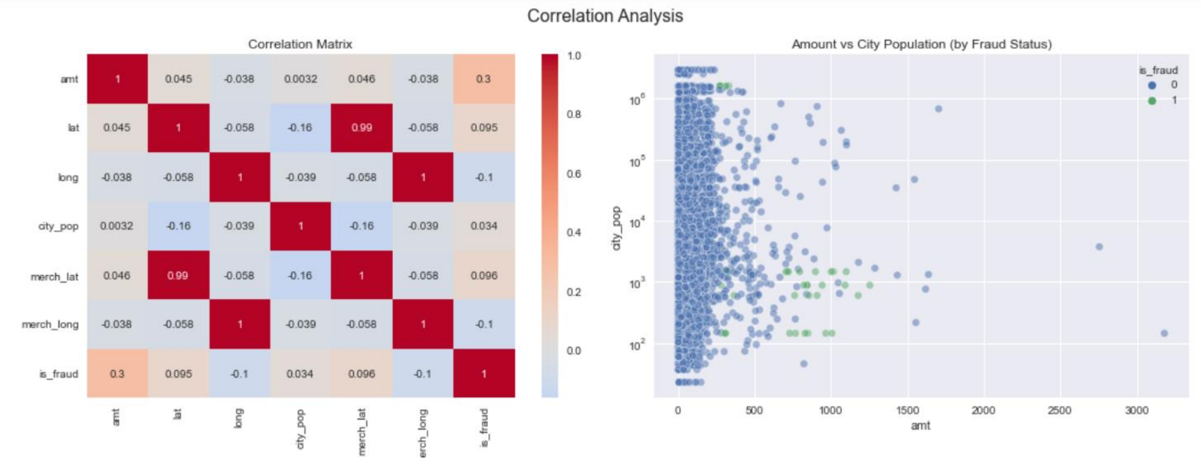
Comprehensive analysis of distributions, correlations, and outliers

Distribution of Key Numerical Features

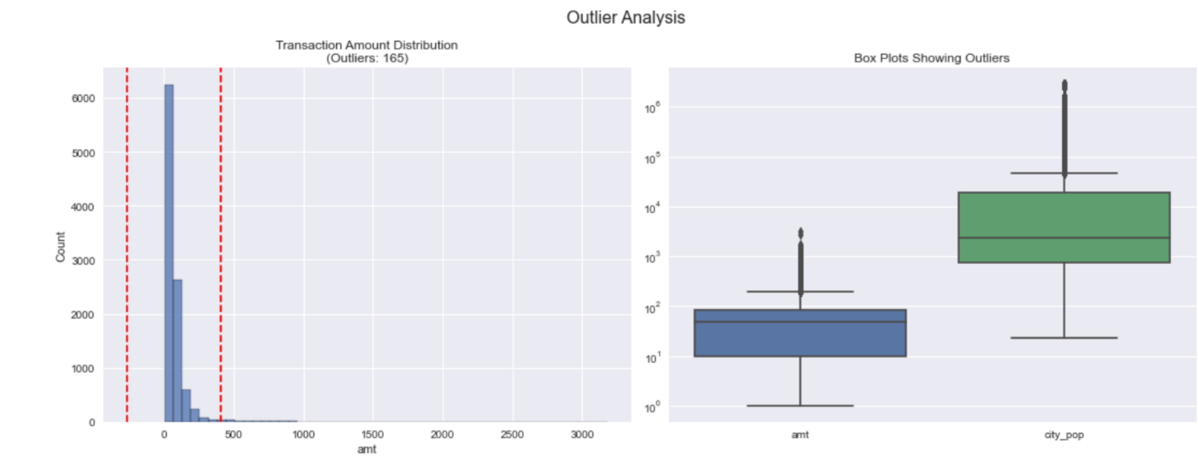


- Transaction amounts show a right-skewed distribution, with most transactions being of lower value
- City population has a highly skewed distribution with many small cities and few large ones
- Transaction categories are relatively well-balanced, with gas_transport and grocery_pos being the most common

Correlation Analysis



Outlier Analysis



Correlation Analysis

- Most features show weak to moderate correlations with each other
- Geographic features (lat/long) show expected correlations with their merchant counterparts
- Transaction amount has a weak correlation with fraud status
- City population shows minimal correlation with other features

Outlier Detection

- Transaction amounts have several outliers, particularly on the high end
- City population data contains extreme outliers, representing major metropolitan areas
- The outliers appear to be legitimate data points rather than errors, as they follow expected patterns

Data Preprocessing

Encoding Categorical Variables

 # there are four categorical features in our data need to be encoded: merchant, category, gender, and job.

Encoding Gender and Category features

```
# 1. Gender - Using Label Encoder since it's binary
from sklearn.preprocessing import LabelEncoder
le_gender = LabelEncoder()
df_cleaned['gender_encoded'] = le_gender.fit_transform(df_cleaned['gender'])

# 2. Category - Using One-Hot Encoding
category_dummies = pd.get_dummies(df_cleaned['category'], prefix='category')

# 3. Remove original columns and add encoded ones
df_cleaned = df_cleaned.drop(['gender', 'category'], axis=1)

# 4. Add category dummy columns to the main dataframe
df_cleaned = pd.concat([df_cleaned, category_dummies], axis=1)

# Display first few rows to verify changes
print("\nFirst few rows of transformed dataframe:")
print(df_cleaned.head())

# Verify the shape and new columns
print("\nDataframe shape:", df_cleaned.shape)
print("\nCategory columns added:", list(category_dummies.columns))
```

Gender

- Used Label Encoder (0 and 1)
- Original 'gender' column is replaced with 'gender_encoded'
- Typically encodes F as 0 and M as 1

Category

- Used One-Hot Encoding
- Created separate binary columns for each category
- Each column starts with 'category_'
- Each row has a 1 in exactly one category column

Removing Job Column

```
# 1. Remove job column
df_cleaned = df_cleaned.drop('job', axis=1)
```

Hot Encoding the Merchant Feature

```
# 2. Get top 50 most frequent merchants
top_50_merchants = df_cleaned['merchant'].value_counts().nlargest(50).index

# 3. Replace less frequent merchants with 'OTHER'
df_cleaned['merchant'] = df_cleaned['merchant'].apply(lambda x: x if x in top_50_merchants else 'OTHER')

# 4. Create dummy variables
merchant_dummies = pd.get_dummies(df_cleaned['merchant'], prefix='merchant')

# 5. Remove original merchant column and add encoded columns
df_cleaned = df_cleaned.drop('merchant', axis=1)
df_cleaned = pd.concat([df_cleaned, merchant_dummies], axis=1)
```

Given this is a fraud detection model:

- merchant might be important as fraud patterns could be associated with specific merchants
- job is likely less relevant for predicting credit card fraud

cc_num can’t be encoded as it is an identifier, also has two many digits so can’t be converted into integers. So we decided to remove it.

```
# Remove cc_num column
df_cleaned = df_cleaned.drop('cc_num', axis=1)
```

Process data with Date-Time type

 # We have only one such feature: trans_date_trans_time. we need to convert that type so that this feature can be processed in our model.

Extract 5 features from trans_date_trans_time.

Hour of day (0-23)

- Crucial for fraud detection
- Fraudulent transactions often occur at unusual hours
- Captures daily transaction patterns

Day of week (0-6)

- Different patterns between weekdays and weekends
- Some fraudsters might target specific days

Day of month (1-31)

- Can capture patterns related to salary payments
- Monthly billing cycles
- End-of-month activities

Month (1-12)

- Seasonal patterns
- Holiday-related fraud patterns

Minutes: just to be more precise

```
# First create all new temporal columns before dropping the original
df_cleaned['hour'] = df_cleaned['trans_date_trans_time'].dt.hour
df_cleaned['minute'] = df_cleaned['trans_date_trans_time'].dt.minute
df_cleaned['day_of_week'] = df_cleaned['trans_date_trans_time'].dt.dayofweek
df_cleaned['day_of_month'] = df_cleaned['trans_date_trans_time'].dt.day
df_cleaned['month'] = df_cleaned['trans_date_trans_time'].dt.month

# Drop the original trans_date_trans_time column
df_cleaned = df_cleaned.drop('trans_date_trans_time', axis=1)

# Let's look at the first few rows to verify the changes
print("First few rows of the transformed dataset:")
print(df_cleaned.head())

# Verify the ranges of all new temporal columns
print("\nValue ranges for new temporal features:")
for col in ['hour', 'minute', 'day_of_week', 'day_of_month', 'month']:
    print(f"\n{col}:")
    print(f"Range: {df_cleaned[col].min()} to {df_cleaned[col].max()}")
    print(f"Unique values: {sorted(df_cleaned[col].unique())}")

# Display the new column data types
print("\nData types of new temporal columns:")
print(df_cleaned[['hour', 'minute', 'day_of_week', 'day_of_month', 'month']].dtypes)
```

Tree-Based Classification Models for Predecting Fraud Accounts

Problem Type Match

- we have a binary classification problem (fraud vs. non-fraud)
- Tree-based models like Decision Trees, Random Forests, or XGBoost are well-suited for classification tasks
- The target variable is_fraud is clearly defined as a binary outcome

Advantages

Handle Mixed Data Types: our dataset contains:

- Numerical features (amt, lat, long, city_pop)
- Categorical features (merchant, category, job, gender)
- Tree models can naturally handle both without extensive preprocessing

Random Forest Classifier

Why

- Robust against overfitting through ensemble learning
- Handles high-dimensional data well (we have 80 features)
- Provides feature importance rankings
- Good with imbalanced datasets when properly configured

Cross Validation-Training/Testing

 # The method of splitting we are going to choose is cross validation

Method Choice: K-Fold Cross-Validation with Stratification

Using 5-fold stratified cross-validation.This means our data will be split into 5 equal parts, maintaining fraud/non-fraud proportions in each fold

Rationale for Choosing Cross-Validation:

- More Robust Performance Estimation
- Each data point will be used for both training and testing
- Gets performance metrics from 5 different train-test combinations
- Provides a more reliable estimate of model performance than a single train-test split

Better for Imbalanced Data

- Our fraud detection dataset is imbalanced (few fraud cases)
- Stratification ensures each fold maintains the same ratio of fraud/non-fraud cases
- Reduces the risk of having folds with too few fraud cases

```
from sklearn.model_selection import StratifiedKFold

# Initialize StratifiedKFold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Prepare data
X = df_cleaned.drop('is_fraud', axis=1)
y = df_cleaned['is_fraud']
```

Training Details

Using Random Forest with parameters optimized for imbalanced fraud detection

- class_weight='balanced' is crucial given your 0.37% fraud rate
- Higher number of trees (200) to better capture rare fraud patterns
- No max_depth restriction to allow model to learn complex patterns

```
from sklearn.ensemble import RandomForestClassifier

# Initialize model with parameters for imbalanced data
rf_model = RandomForestClassifier(
    n_estimators=200,            # More trees for better performance
    max_depth=None,              # Allow full depth for complex fraud patterns
    min_samples_split=2,         # Default value
    min_samples_leaf=1,          # Default value
    class_weight='balanced',     # Critical for handling 0.37% fraud ratio
    random_state=42,
    n_jobs=-1                    # Use all CPU cores
)

# Train and evaluate
for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
    X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
    y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

    rf_model.fit(X_train, y_train)
```

Model Evaluation Metrics Choice

Precision: Measures false positives (important for reducing false fraud alerts)

Recall: Measures missed frauds (crucial - we want to catch most frauds)

F1-Score: Balances precision and recall

PR-AUC: Better than ROC-AUC for imbalanced data

```
# Print average metrics
print("\nAverage Metrics across folds:")
print(f"Precision: {np.mean(precision_scores):.3f} (+/- {np.std(precision_scores):.3f})")
print(f"Recall: {np.mean(recall_scores):.3f} (+/- {np.std(recall_scores):.3f})")
print(f"F1-score: {np.mean(f1_scores):.3f} (+/- {np.std(f1_scores):.3f})")
print(f"PR-AUC: {np.mean(pr_aucs):.3f} (+/- {np.std(pr_aucs):.3f})")
```

```
Average Metrics across folds:
Precision: 1.000 (+/- 0.000)
Recall: 0.476 (+/- 0.244)
F1-score: 0.608 (+/- 0.227)
PR-AUC: 0.965 (+/- 0.043)
```

Performance Analysis of Your Model:

Precision (1.000 ± 0.000)

Perfect precision (1.0) means no false positives

When model predicts fraud, it's always correct

Very good for minimizing false fraud alerts

Recall (0.476 ± 0.244)

Model catches about 48% of actual fraud cases

High variation (±0.244) across folds

Looking at individual folds:

Key Observations:

Model is very conservative (high precision, lower recall)

Performance varies significantly between folds

Overall accuracy looks inflated due to imbalanced data

Suggestions for Improvement:

Could adjust class weights to improve recall

Consider using SMOTE or other sampling techniques

Might try different threshold for classification

Model Improvement

Problem Identification

Initial model had good precision (100%) but poor recall (47.6%)

High class imbalance (only 0.37% fraud cases)

Performance varied significantly between folds

Need to catch more fraud cases while maintaining precision

Solution Approach

Used SMOTE (Synthetic Minority Over-sampling Technique) to balance training data

Modified Random Forest parameters for better performance

Combined multiple techniques to handle imbalanced data

Kept validation data in original distribution for realistic evaluation

Key Improvements Made

Data Level (SMOTE):

Created synthetic fraud cases in training data only

Balanced the class distribution for better learning

Maintained original validation data to test real-world performance

Model Level (Random Forest):

Increased number of trees to 300 for better learning

Removed depth restrictions to capture complex fraud patterns

Added class weights to further handle imbalance

Utilized all CPU cores for efficient training

Results

```
Improved Model - Average Metrics:
Precision: 0.967 (+/- 0.067)
Recall: 0.819 (+/- 0.139)
F1-score: 0.879 (+/- 0.092)

Top 10 Most Important Features:
           feature  importance
0              amt    0.265944
3         city_pop    0.110792
1              lat    0.089407
2             long    0.074593
4        merch_lat    0.072970
10     day_of_week    0.061568
5       merch_long    0.058379
6    merch_zipcode    0.050963
11    day_of_month    0.050243
28  merchant_OTHER    0.042212
```

Improved Metrics:

Recall increased from 47.6% to 81.9% (catching more fraud)

Precision remained high at 96.7% (few false alarms)

F1-score improved from 0.608 to 0.879 (better overall)

More consistent performance across folds

Feature Insights:

Transaction amount most important (26.6%)

City population second most important (11.1%)

Geographic features (lat/long) also significant

Why It Worked

SMOTE provided better examples of fraud patterns

More trees captured complex relationships

No depth restriction allowed detailed pattern learning

Combined approaches (SMOTE + class weights) handled imbalance effectively

Original validation data ensured realistic performance measurement