Ashish D Fugare]170101023

Fraud Detection with Unlabeled Dataset

Report for Understanding of ML Project

Ashish D Fugare (Team Leader)
Roll: 170101023 | ashis170101023@iitg.ac.in | Group: Sinhagad

Priya Sharma
Roll: 22001081 | priya.sharma@iitg.ac.in | Group: Sinhagad

1 Introduction to Initial Problem

Understand what problem we are trying to solve?
Our problem is to perform Fraud Transaction Detection, but our dataset doesn't have labels identifying which transactions are fraudulent or normal for reference.

2 Dataset Review

Dataset Overview:

- **Total transactions:** 217,441
- **Features:** 7 (Timestamp, TransactionID, AccountID, Amount, TransactionType, Merchant, Location)
- **Missing entries:** 481 (removed from analysis)
- **Unused features:** TransactionID (dropped as it doesn't aid fraud detection)

I split the data into training (70%), test (20%), and validation (10%) division of dataset.

2.1 Feature Engineering

Considering this is a dataset of transactions, we can create additional features to better identify fraud:

Features Engineered:

Account-Based Features

- *Avg_Amount*: Average transaction amount for each account - establishes baseline spending behavior
- *Std_Amount*: Standard deviation of amounts per account - measures spending variability
- *Min_Amount/Max_Amount*: Minimum/maximum transaction amounts for each account - establishes spending boundaries
- *Tx_Count*: Number of transactions per account - measures account activity level
- *Amount_Ratio*: Current transaction amount divided by average amount for that ac-

count - detects unusually large transactions than their usual pattern

Time-Based Features

- *Hour/Day/DayOfWeek/Month*: Time components of transactions - captures temporal patterns
- *Weekend*: Binary flag (1=weekend, 0=weekday) - identifies weekend transactions
- *BusinessHours*: Binary flag (1=9am-5pm weekdays, 0=otherwise) - identifies transactions during business hours

Time-Window Features

- *Time_Since_Last_Tx*: Hours elapsed since previous transaction for the same account- measures transaction frequency , multiple transactions happening suspiciously close
- *Rapid_Succession*: Binary flag for transactions occurring less than 1 hour after previous transaction - detects unusually rapid transactions

Location-Based Features

- *Common_Location*: Most frequent transaction location for each account (intermediate feature)
- *Location_Change*: Binary flag for transactions occurring in a location different from the account's usual location - detects geographical anomalies for a specific account

3 Solving the Problem

3.1 Initial Clustering Attempt

I hypothesized that fraudulent transactions would cluster together in feature space. Testing this, I applied DBSCAN for initial fraud detection but results were disappointing - only 15 transactions flagged as fraudulent with a poor silhouette score (0.084). PCA visualization confirmed most data points were tightly packed, suggesting fraudulent transactions are scattered throughout the feature space rather than forming distinct clusters. This indicated clustering might not be the optimal approach for this problem.

Why DBSCAN didn't work?
After running the other two models (**Isolation Forest** and **LOF**), and examining how potential fraud transactions appeared, I concluded that fraudulent transactions aren't all similar across all features. They often differ in subtle but significant ways and don't cluster neatly .

Example: Transaction Anomaly

Transaction 1: $10,000, location changes (e.g., New York to Tokyo), timestamp A.
Transaction 2: $10,000, same location, timestamp B (slightly different time), all else identical.

In feature space, they might appear close since most features match. However, Transaction 1 could be fraudulent (location change signaling a stolen card) while the second is normal (same user, same place). DBSCAN would see them as part of the same dense region and group them together, assuming they are similar. This approach would miss many fraudulent transactions.

DBSCAN would only identify fraudulent transactions that consistently differ from normal transactions across all features or transactions too rare to form their own dense cluster.Also, in typical transaction datasets, fraud would be relatively small compared to normal transactions (otherwise, that institution would likely cease to operate), so these few outliers could also be labeled as noise by DBSCAN. This makes DBSCAN a poor choice for this problem.

Important Note

Fraud detection with DBSCAN was less effective due to subtle feature differences.

3.2 Next Model Selection

For the next phase, I selected **Isolation Forest** and **Local Outlier Factor (LOF)** to address fraud detection:

- **Isolation Forest** was chosen because it is specifically designed for anomaly detection, operating on the inherent assumption that anomalies such as fraud are rare.
- **Local Outlier Factor (LOF)** was selected to ensure we don't miss the local context of transactions (**e.g.**, a transaction of $500 could look normal in a set of $100-$1000 transactions but appear suspicious locally if it's in an area where $10 transactions are common).

3.3 Running the Models

We ran both Isolation Forest and LOF models. My understanding of their working principles is as follows:

How Isolation Forest works?

Isolation Forest operates by randomly partitioning the dataset and isolating individual points. The key idea is that anomalies require fewer splits to be separated because they differ significantly from the majority.Imagine a bag of 100 balls: 80 blue and 20 red. If you randomly divide the bag into two groups by picking a feature (say, color), a split might separate most red balls from the blue ones early on. For instance, if a split isolates a small group with 5 balls—4 red and 1 blue—the red balls are already closer to being singled out.

In Isolation Forest, the anomaly score reflects how quickly a transaction can be isolated within a tree, the more easier to isolate more the anomaly score

In a transaction dataset, a fraud case—like a $10,000 purchase with an unusual location—might be isolated quickly if a split on location separates it from typical transactions, marking it as an anomaly

How LOF works?

LOF works by examining density: if a point has much lower density than those around it, meaning it is in a sparser region, it is flagged as an outlier. This is great for fraud detection because it focuses on local context, not just global patterns.

Higher scores for LOF indicate transactions in unusually sparse regions. Based on these approaches, both models give an anomaly score for all transactions.

Choice of Threshold?

To identify fraud in an unlabeled dataset, I needed to define a threshold for the anomaly scores from Isolation Forest and LOF. Fruad transaction in dataset often ranging from 1% to 10% according to industry averages. I initially chose 5%—the midpoint—as a reasonable starting threshold, flagging the top 5% of scored transactions as potential fraud.

This works as in our problem the goal was to avoid flagging too many transactions as fraudulent, which could overwhelm resources or disrupt legitimate activity, while ensuring minimal missed fraud cases, which carry high costs.So this middle ground works for us.Both models consistently highlighted around 5% of transactions as anomalies across validation and test sets, which aligns with my initial estimate. While the absence of ground truth prevents definitive confirmation, this consistency, combined with fraud's expected rarity, suggests the 5% threshold choice is a practical and effective starting point.

So we got the following results

Results

```
--- Analyzing anomaly score distribution with fixed 5% contamination rate--
Using fixed 5% contamination rate:
Isolation Forest threshold: 0.000223
LOF threshold: -0.002165
Isolation Forest anomalies: 2175 (5.00%)
LOF anomalies: 2175 (5.00%)
Ensemble anomalies: 3979 (9.15%)

Validation: Isolation Forest anomalies: 1093 (5.03%)
Test: Isolation Forest anomalies: 2271 (5.22%)

Validation: LOF anomalies: 1216 (5.59%)
Test: LOF anomalies: 2207 (5.07%)
```

Model Overlap: Only 387 transactions were flagged as anomalies by both models, indicating each captures different fraud patterns.

4 Feature Importnace

Feature Importance Analysis Methodology

To understand which features drove fraud detection, I used:

1. Correlation Analysis: Measured relationship between features and anomaly scores

2. Statistical Tests:

- Kolmogorov-Smirnov tests for numerical features
- Total variation distance for categorical features

3. Standardized Differences:

$$Z = \frac{\mu_{\text{anomaly}} - \mu_{\text{normal}}}{\sigma_{\text{normal}}}$$

4. Feature Group Analysis: Evaluated logical feature groups (Amount, Time, Account Statistics, Location, Transaction Type)

5. High-Confidence Case Profiling: Examined transactions flagged by both models

5 Key Findings from Feature Importance Analysis

5.1 Most Significant Features

Our analysis revealed several features with strong fraud-detection capabilities:

- **Min_Amount:** Showed the largest separation ($Z \approx 190$ standard deviations) between normal and anomalous transactions. Normal accounts typically have miscellaneous small charges, while fraudsters often conduct exclusively large transactions.
- **Time_Since_Last_Tx:** High significance ($Z \approx 8$), indicating that extended inactivity followed by sudden transactions strongly signals potential account takeovers.
- **Location_Change:** Substantial distributional shift (0.54 total variation distance), reflecting that legitimate users rarely conduct transactions across different cities in short timeframes.
- **Tx_Count:** Negative correlation with fraud, suggesting fraudsters typically minimize transaction counts to avoid detection.
- **BusinessHours** and **Hour** showed minimal impact on fraud detection.

5.2 Feature Subset Performance

Testing with feature subsets revealed that:

- **Transaction Type** alone yielded 33% model agreement
- **Merchant** features: 10% agreement
- **Time Features**: 9.4% agreement
- All features combined: only 7.2% agreement

This suggests specific transaction types have strong fraud signals, while combining all features introduces noise that reduces detection agreement.

5.3 Complementary Model Strengths

Each model demonstrated distinct strengths:

- **Isolation Forest** excelled with:
  - Amount-based anomalies (0.39 correlation with *Max_Amount*)
  - Day-of-week patterns (0.25 with *DayOfWeek*)
  - Example: $50,000 Sunday transactions from accounts typically averaging $500 on weekdays
- **LOF** performed better on:
  - Timing anomalies (0.44 correlation with *Time_Since_Last_Tx*)
  - Sequence irregularities (e.g., 3 AM transactions from accounts normally active 9 AM-5 PM)

This complementarity explains why *Time_Since_Last_Tx*, *Location_Change*, and *Min_Amount* emerged as top fraud indicators, with each model capturing different fraud patterns.

6 What I learned:

I saw the the importance of using appropriate tools to handle datasets.As initially, I opened the dataset in Excel to inspect it, but this led to unexpected error. As the dataset's timestamp was formatted as DD/MM/YYYY HH:MM, but Excel automatically padded it with seconds (DD/MM/YYYY HH:MM:SS) while viewing, altering the data structure that I saw. This caused parsing errors when loading the dataset into Python, which were difficult to diagnose at first.

To understand feature scaling's role in fraud detection, I experimented with Isolation Forest and Local Outlier Factor (LOF) by turning off scaling and comparing it to runs with scaling. Without scaling, both models flagged 10,000 transactions (5%) in the full dataset, but the ensemble (union of anomalies) reached 19,560 (9.78%). With scaling, the ensemble dropped to 18,300 (9.15%). This showed that without scaling, we missed many unique, subtle fraud patterns and focused too much on the large-amount feature, *Amount*. For example, in the unscaled version, a $1,000 difference was seen as more anomalous than a 24-hour gap in transactions..

Feature engineering is crucial derived features like Time_Since_Last_Tx and Location_Change were more important than raw data

I also tried implementing both Isolation Forest and LOF manually in Python. Running all 200,000 dataset entries took too much waiting time and emptied my Colab compute time because we weren't doing efficient distance calculations compared to the number of entries. I saw that this implementation would be slow with Python, and for better performance close to scikit-learn, I would need to implement it in C++ for anywhere close to performance of scikit-learn. Then I subsampled to 10,000 entries and now I could atleast get results. LOF gave 100% the same output between my manual version and scikit-learn function as we are doing distance calculation and it becmoes determinstic.However, there were slight differences in anomalies between manual and scikit-learn Isolation Forest implementations. I attributed these discrepancies to the inherent randomness in the

Isolation Forest algorithm, specifically the random splitting of trees during the model construction process.

Author Contributions

- **Ashish D Fugare**: Project implementation, debugging, data analysis, model development, model interpretability,report writing
- **Priya Sharma**: