AGENTIC AI FOR FRAUD DETECTION

PS C:\Users\devpa\OneDrive\Desktop\FraudDetectionAgenticAI> python main.py
➡️ Original shape: (25, 9)
✅ Cleaned shape: (23, 9)
✅ After CleanerAgent: shape (23, 9)
✅ After FeatureEngineerAgent: columns ['transaction_id', 'customer_id', 'amount', 'timestamp', 'country', 'merchant_category', 'device_id', 'ip_address', 'is_fraud', 'txn_hour', 'day_of_week', 'night_txn', 'risky_category', 'txn_count_24h', 'geo_mismatch', 'device_usage_count']
✅ After ScalerAgent: scaled features ['amount', 'txn_hour', 'txn_count_24h', 'device_usage_count']
✅ After RuleEngineAgent: fraud flags sum = 19
✅ MemoryAgent: stored 57 flagged transactions across runs.
✅ After MLModelAgent: trained model, fraud preds sum = 10
✅ After SHAPAgent: saved SHAP summary plots
✅ RuleCheckAgent review complete.
✅ Saved final predictions to outputs/final_predictions.csv

✅ Agent suggests new rules:

Here are some improved Python fraud detection rules as code snippets, based on the provided data:

1. Rule: Transactions with unusual device usage patterns

Create a rule that flags transactions with a device usage count greater than 2 standard deviations from the mean.

```
import numpy as np

# Calculate mean and standard deviation of device usage count
mean_device_usage = np.mean([tr['device_usage_count'] for tr in transactions])
std_device_usage = np.std([tr['device_usage_count'] for tr in transactions])
```

```
# Flag transactions with unusual device usage patterns
fraudulent_transactions = [tr for tr in transactions if np.abs(tr['device_usage_count'] - mean_device_usage) > 2 * std_device_usage]
```

2. Rule: Transactions with high value and unusual time of day

Create a rule that flags transactions with a value greater than $100 and a time of day outside of the usual business hours (e.g., between 9am and 5pm).

```
# Define the business hours
business_hours = [(9, 17)]  # 9am to 5pm

# Flag transactions with high value and unusual time of day
fraudulent_transactions = [tr for tr in transactions if tr['amount'] > 100 and not any(tr['timestamp'].hour >= h[0] and tr['timestamp'].hour <= h[1] for h in business_hours)]
```

3. Rule: Transactions with unusual geographic patterns

Create a rule that flags transactions with a country that is not the usual country for the customer (e.g., if the customer is from the US and the transaction is from Russia).

```
# Define a function to calculate the usual country for a customer
def usual_country(customer_id):
    usual_countries = {1001: 'US', 1002: 'RU', 1003: 'US', 1004: 'US'}
    return usual_countries.get(customer_id)

# Flag transactions with unusual geographic patterns
fraudulent_transactions = [tr for tr in transactions if tr['country'] != usual_country(tr['customer_id'])]
```

4. Rule: Transactions with high probability of fraud

Create a rule that flags transactions with a high probability of fraud (e.g., greater than 0.5).

```
# Flag transactions with high probability of fraud
fraudulent_transactions = [tr for tr in transactions if tr['ml_prob'] > 0.5]
```

You can combine these rules to create a more robust fraud detection system. For example, you can use the first rule to flag transactions with unusual device usage patterns, and then apply the second rule to those transactions to filter out false positives.

✅ Code review by AI on the new rules:

Correctness:

The rules seem to be logically correct and aligned with the provided description. Each rule is based on a specific condition that is known to be indicative of fraudulent behavior.

Efficiency:

Here are some suggestions to improve the efficiency of the rules:

1. **Rule 1:** Instead of recalculating the mean and standard deviation for each iteration, you can calculate them once and store them in variables. This can improve performance when dealing with large datasets.
2. **Rule 2:** Instead of using a list comprehension to filter transactions, you can use a generator expression. This can be more memory-efficient when dealing with large datasets.
3. **Rule 3:** Instead of using a dictionary to store the usual country for each customer, you can use a more efficient data structure such as a pandas Series or a NumPy array. This can improve performance when dealing with large datasets.
4. **Rule 4:** Instead of using a list comprehension to filter transactions, you can use a generator expression. This can be more memory-efficient when dealing with large datasets.

Here's an updated version of the rules that incorporates these suggestions:

Rule 1:

```
import numpy as np

# Calculate mean and standard deviation of device usage count
mean_device_usage = np.mean([tr['device_usage_count'] for tr in transaction
s])
std_device_usage = np.std([tr['device_usage_count'] for tr in transactions])

# Flag transactions with unusual device usage patterns
fraudulent_transactions = (tr for tr in transactions if np.abs(tr['device_usage_c
ount'] - mean_device_usage) > 2 * std_device_usage)
```

Rule 2:

```
# Define the business hours
business_hours = [(9, 17)]  # 9am to 5pm

# Flag transactions with high value and unusual time of day
fraudulent_transactions = (tr for tr in transactions if tr['amount'] > 100 and not
any(tr['timestamp'].hour >= h[0] and tr['timestamp'].hour <= h[1] for h in busi
ness_hours))
```

Rule 3:

```
# Define a function to calculate the usual country for a customer
usual_countries = pd.Series({1001: 'US', 1002: 'RU', 1003: 'US', 1004: 'US'}, ind
ex=[1001, 1002, 1003, 1004])

# Flag transactions with unusual geographic patterns
fraudulent_transactions = (tr for tr in transactions if tr['country'] != usual_coun
tries.get(tr['customer_id']))
```

Rule 4:

```
# Flag transactions with high probability of fraud
```

```
fraudulent_transactions = (tr for tr in transactions if tr['ml_prob'] > 0.5)
```

By combining these rules, you can create a more robust fraud detection system that can effectively identify fraudulent transactions.