Fraud Detection System - High-Level Overview and Requirements

Industry: Payment processing
Objective: To combat online payment fraud and protect both merchants and users by identifying and blocking suspicious activity.
System Overview:

- Data Sources: Transaction data (time, card details, IP address, amount, etc.), user profile information, device fingerprinting.
- Algorithms: Rule-based checks, statistical analysis, machine learning models for anomaly detection.

For this task, the algorithms can be separated into two.

- None Machine learning : Using rule base to check – if, elif and else
- Machine learning : Classifications model

The criteria for whether or not to use machine learning is the complexity of data and its condition. This should be considered after modeling exploration.

Nonetheless, the fraud engine component must consist of two algorithms and use them dependently on circumstance.

High-Level Requirements:

Functional:

- Real-time Transaction Monitoring: Analyze every transaction for suspicious patterns based on defined rules and algorithms.
- Risk Scoring: Assign a risk score to each transaction based on various factors like time, location, user behavior, and card information.
- Alert Generation: Trigger alerts for high-risk transactions for manual review and potential blocking.
- Case Management: Facilitate investigation and resolution of suspected fraudulent cases.
- Reporting and Analytics: Provide insights into fraud trends and system performance for continuous improvement.

Non-Functional:

- Performance: The system must be able to handle high transaction volumes with minimal latency.
- Security: All data must be stored and transmitted securely, adhering to industry standards like PCI DSS.
- Usability: The system interface should be easy to use for both merchants and fraud analysts.

Development Steps:

- Develop detailed functional and non-functional requirements for each component of the system.
- Design and implement the fraud detection system based on the chosen architecture and algorithms.
- Conduct thorough testing and validation to ensure system accuracy and performance.
- Continuously monitor and update the system to adapt to evolving fraud tactics.

By implementing a robust fraud detection system based on these high-level requirements, you can significantly reduce fraudulent activity and protect your business and users from financial losses.

Remember, this is a high-level overview and should be further customized based on your specific needs and technical considerations.

Please let me know if you have any questions or need further assistance in refining these requirements.

None Machine learning Engine

High-Level Requirements

This engine is much more affordable and business understands. By adding the suspect cases to the class, the fraud transaction would explode.
Additionally, each detected case has its weight and reliability.

The result must return the holder's name, matched cases, and fraud score. This format allowed the management team to investigate and take some serious action.

For example,

- Audrey uses the card between Saturday 22:00 - 02:00.
- Binti have the same case, but he frequently uses different card but same IP for a short time.

Transaction Date	Card/Account	Holder Name	Payer IP
1/8/23 22:10	527517******6525	Alexis Hernandez	213.255.247.56
1/7/23 22:30	543215******6187	Audrey Warner	103.144.245.176
1/9/23 23:01	473310******1168	Ben Hampson	111.1.213.122
1/8/23 23:39	437551******2016	Binti Daud	58.136.190.229
1/8/23 23:39	485498******2922	Binti Daud	58.136.190.229
1/8/23 23:40	521729******8378	Binti Daud	49.228.232.120

Figure 1.1: The example raw data.

Holder Name	Fraud_score	Cases
Alexis Hernandez	1	| use_between_time
Audrey Warner	1	| use_between_time
Ben Hampson	0	
Binti Daud	4	| use_between_time | use_between_time | use_between_time | same_ip_different_card_transactions
Cecilia Park	0	
Chittrawan Rangsimahariwong	0	
Cynthia Cusumano	0	
Danielle Ash	0	
Dany j Castaneda	0	
Franklin Salasky	0	
Jimmie Hayes	0	
Kate Dennis	1	| same_ip_different_card_transactions

Figure 1.2: The example result.

Example of cases:

1. Time-Based Detection: Flag transactions occurring between 10 PM and 4 AM or on weekends as potentially high-risk.
2. Rapid Card Use: Identify suspicious scenarios where multiple cards are used from the same IP address within a short timeframe.
3. Excessive IP Switching: Alert on transactions where IPs are changed frequently (3-6 times) in a day, especially with different cards.
4. Escalating Transaction Amounts: Monitor for users gradually increasing spending amounts, especially after small initial transactions.
5. Lost Card Misuse: Detects attempts to use known lost credit cards after initial trials with valid cards.
6. … Could be adding if new cases found

Low level requirements

1. Python language would be selected, compatible with machine learning algorithms.
2. Clear DocString and standard name convention.
3. Containerize package: compatible with various environment
4. Robust class: The new case could be easily adding
5. OOP practise: To utilize encapsulation and polymorphism concepts.
6. Capture all transactions that meet the cases.
7. All constant parameters must be the attribute of the class, can be obtained and set.
8. Low resource consumption.
9. All parameters must be set and get, to prevent the constant setting at code.

Coding example

Project structure

Follow the best, I hope, practice in python package development.

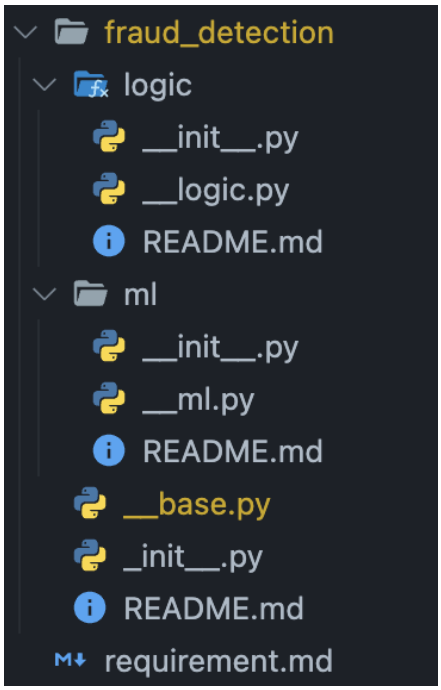


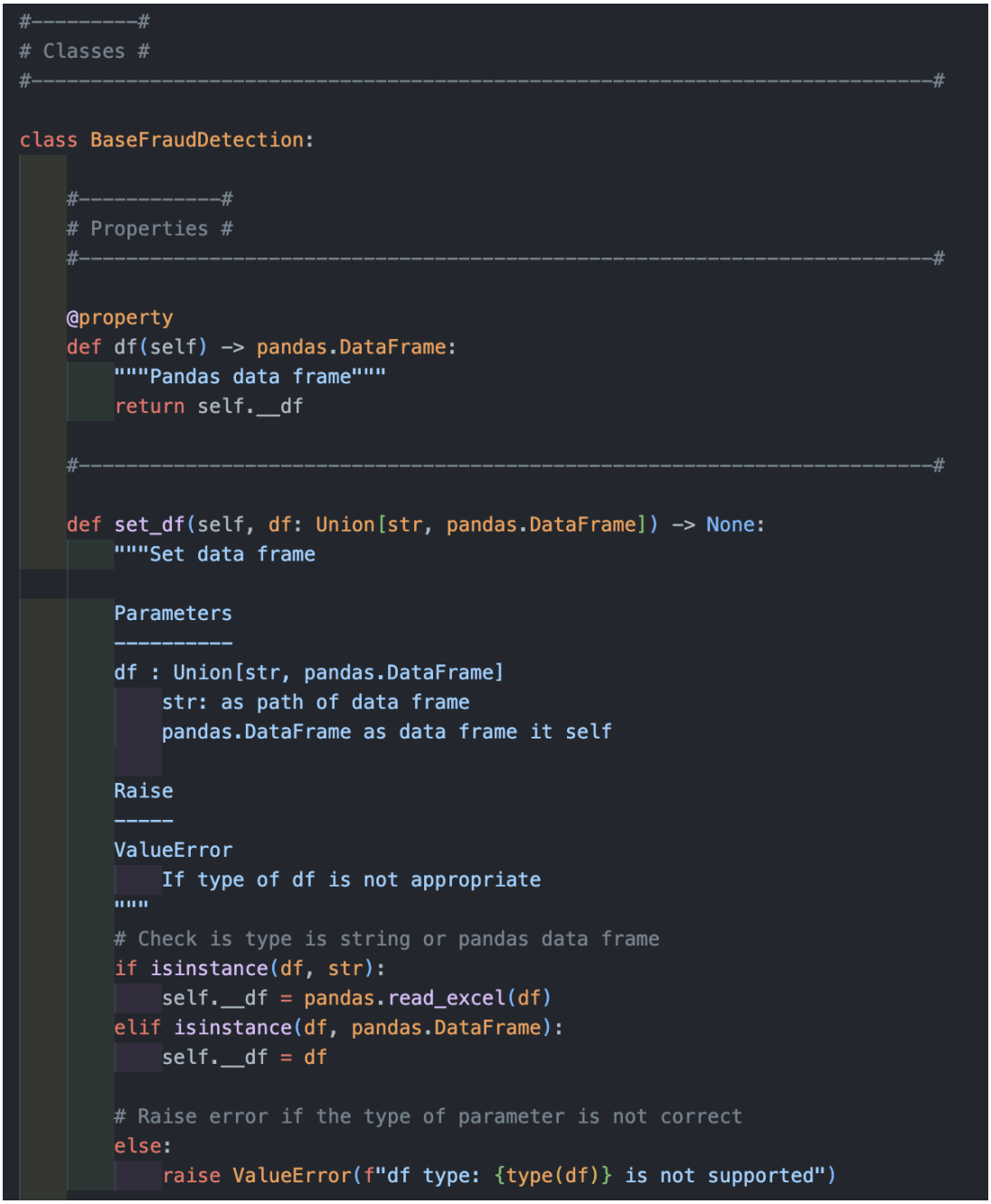
Figure 2: The project structure

The additional file can be added – eg. utilities, assets and ect.

Base class

The base class for inherited guidelines, this class consolidated with the properties and methods shared with the other class.

1. Source of data must be **pandas.DataFrame.** This data structure is tabular and easy to manipulate.



```python
#---------#
# Classes #
#-----------------------------------------------------------------------------#

class BaseFraudDetection:

    #------------#
    # Properties #
    #-----------------------------------------------------------------------------#

    @property
    def df(self) -> pandas.DataFrame:
        """Pandas data frame"""
        return self.__df

    #-----------------------------------------------------------------------------#

    def set_df(self, df: Union[str, pandas.DataFrame]) -> None:
        """Set data frame

        Parameters
        ----------
        df : Union[str, pandas.DataFrame]
            str: as path of data frame
            pandas.DataFrame as data frame it self

        Raise
        -----
        ValueError
            If type of df is not appropriate
        """
        # Check is type is string or pandas data frame
        if isinstance(df, str):
            self.__df = pandas.read_excel(df)
        elif isinstance(df, pandas.DataFrame):
            self.__df = df

        # Raise error if the type of parameter is not correct
        else:
            raise ValueError(f"df type: {type(df)} is not supported")
```

Figure 3: The example of set and get df, this attribute would be utilized by others.

2. Abstract method as a guildling of the other class.

```
#--------#
# Method #
#-----------------------------------------------------------------------------#

@abstractmethod
def get_fraud_transaction(self, df: pandas.DataFrame) -> pandas.DataFrame:
    """Would be overrides at child class. The High level method
    to consume data and returned the result.

    Parameters
    ----------
    df : pandas.DataFrame
        Target dat frame

    Returns
    -------
    pandas.DataFrame
        Fraud transaction
    """

#-----------------------------------------------------------------------------#
```

Figure 4: abstract method for all classes.

3. Utilities function at BaseClass

```
#-----------------------------------------------------------------------------#
# Utilities #
#-----------#

@staticmethod
def extract_date(
        df: Union[str, pd.DataFrame],
        date_column: str
) -> pd.DataFrame:
    """Extract date column to useful information

    Parameters
    ----------
    df : Union[str, pd.DataFrame]
        Data frame
    date_column : str
        target date column

    Returns
    -------
    pd.DataFrame
        data frame with column `day_of_week` and `time`
    """
    df["day_of_week"] = df[date_column].dt.day_name()
    df["time"] = df[date_column].dt.time

    return df


#-----------------------------------------------------------------------------#
```

Figure 5: Utility function.

LogicFraudDetection class

This is the main engine for the fraud detection package. The logic can be assigned to the main call. This practice confirms the robust detection pipeline.

```
fraud = LogicFraudDetection(
    df = "fraud-detection-data (1).xlsx",
    date_column = "Transaction Date",
    card_holder_column = "Holder Name",
    ip_column_name = "Payer IP",
    card_column = "Card/Account"
)
```

Figure 6.0: Example of class initiation.

```
df = fraud.get_fraud_transaction(
    check_name_list=[
        "use_between_time",
        "same_ip_different_card_transactions",
    ]
)
```

Figure 6.1: Calling method, the logics could be assigned.

```
df = fraud.get_fraud_transaction(
    check_name_list=[
        "same_ip_different_card_transactions",
    ]
)
```

Figure 6.2: Changing logic.

The flow can be ended by design a robust calling method.

```python
def get_fraud_transaction(self, check_name_list: list[str]) -> DataFrame:
    """Check all applied logic and return the suspects

    Parameters
    ----------
    df : DataFrame
        Target data frame

    Returns
    -------
    DataFrame
        Result
    """
    # Set parameter
    df = self.df
    user_df = pd.DataFrame(
        self.df[self.card_holder].unique(),
        columns=[self.card_holder]
    )
    user_df["fraud_score"] = 0
    user_df["cases"] = ""

    # Iterate over function name to activate and return df
    for check_name in check_name_list:

        # Get function for each plot name, using name
        check_function = getattr(self, check_name)

        # Check if the plot function is found
        if check_function is None or not callable(check_function):
            print(f"Warning: FE function '{check_name}' not found.")
            continue

        # Execute plot function
        user_df = check_function(df, user_df)

    return user_df
```

Figure 7.0: Main calling function.

```
def same_ip_different_card_transactions(
        self,
        df: DataFrame,
        user_df: DataFrame,
) -> DataFrame:
    # Set a time window for considering transactions
    # as part of the same session
    time_window = timedelta(minutes=self.use_count_threshold_minute)

    df = df.sort_values(by=[self.card_holder, self.date_column])

    df['short_time_diff'] = df.groupby(self.card_holder)[self.date_column].diff

    suspected_fraud_cases = df[(df['short_time_diff'] <= time_window) & (df[sel

    suspected_fraud_user_count = suspected_fraud_cases.groupby(self.card_holder

    for idx, row in suspected_fraud_user_count.iterrows():
        number_uses_condition = row["count"] > 0

        user_df = self.add_fraud_score(
            user_df,
            row[self.card_holder],
            number_uses_condition,
            "same_ip_different_card_transactions",
            1,
            self.use_count_weight
        )
    return user_df

#----------#
```

Figure 7.1: Example of detection function.

Noted, you can go to attratched code to investigate the docstring and more logic.