AI and Big Data for Fraud Detection in Financial Systems: An Object-Oriented Approach with Machine Learning

Bo Yu
Student ID: 1194159
Instructor: Hamza Djigal
Course: CPS3962

June 4, 2025

Abstract

This project presents an intelligent fraud detection system that integrates machine learning with object-oriented software engineering principles to address the growing challenge of financial fraud. Leveraging a Random Forest classifier trained on real-world transaction data from Kaggle, the system achieves 88.8% accuracy in identifying fraudulent activities. Key innovations include: (1) a modular object-oriented design implemented through UML-modeled components (Transaction, FraudDetector, Alert), (2) handling of class imbalance using Synthetic Minority Oversampling Technique (SMOTE), and (3) a relational database backend for traceability. The system demonstrates how object-oriented analysis and design (OOAD) principles can enhance the scalability and maintainability of AI-powered financial security solutions while maintaining detection efficacy.

1. Introduction

1.1. Problem Context

The increasing volume of digital financial transactions introduces a growing threat of fraud. From online shopping to mobile banking, malicious activities have become more sophisticated and frequent. Traditional rule-based fraud systems struggle to adapt to dynamic patterns. Financial fraud has grown exponentially with digital transaction volumes, costing businesses over $42 billion globally in 2023. Traditional rule-based systems fail to adapt to evolving fraud patterns due to their static nature. This project addresses three critical gaps:

- **Adaptability**: Machine learning models can learn dynamic fraud patterns
- **Scalability**: Object-oriented design enables modular system expansion
- **Imbalance**: Fraud cases typically comprise ¡0.5% of transactions

1.2. Solution Approach

The proposed system combines:

- **AI Component**: Random Forest classifier with SMOTE oversampling
- **Data Pipeline**: Python-based ETL with MySQL storage
- **OO Architecture**: UML-modeled classes following SOLID principles

2. System Architecture

2.1. High-Level Design

The system is organized into a classic three-tier architecture, ensuring separation of concerns, scalability, and modularity in line with object-oriented design principles:

1. **Data Layer**: This layer consists of a structured relational database implemented in **MySQL 8.0**. Two main tables—`CreditCardTransactions1` and `Alerts`—store preprocessed transaction data and detected fraud cases. The schema design follows normalization rules and enforces referential integrity using foreign keys.
2. **Processing Layer**: This core logic layer is implemented using **Python 3.9**, where all data preprocessing, machine learning, database I/O, and business rules are encapsulated. OOAD principles are reflected through class-based design, with modules for `Transaction`, `FraudDetector`, and `AlertManager`. Fraud detection is performed using a trained Random Forest classifier, with synthetic data augmentation via `SMOTE`.
3. **Presentation Layer**: Currently implemented as command-line and file-based outputs, this layer provides interaction endpoints for users (customers) and administrators. In future iterations, this can evolve into a full-stack web interface for real-time transaction submission, alert visualization, and administrative control.

2.2. Technology Stack

The following tools, frameworks, and libraries were used to implement the system. Each component was selected to support modular design, maintainability, and extensibility:

Component	Technology / Framework
Programming Language	Python 3.11
Data Processing	`pandas`, `numpy`, `scikit-learn`
Machine Learning	`RandomForestClassifier`, `SMOTE` (from imbalanced-learn)
Database	MySQL 8.0 with InnoDB storage engine
Visualization & Evaluation	`matplotlib`, `seaborn`, `classification_report`
Database Integration	`mysql-connector-python`
UML Modeling	Draw.io (Class, Sequence, and Use Case diagrams)

Table 1: Technology stack used in system implementation

2.3. Design Principles Applied

- **Encapsulation**: Each logical module (e.g., fraud detection, alert generation) is isolated within its respective class.
- **Modularity**: Transaction handling, classification, and reporting functionalities are decoupled, enabling maintainability.
- **Reusability**: The Python-based architecture allows reusable preprocessing and modeling pipelines.
- **Extensibility**: Easy to integrate APIs, dashboards, or enhanced models (e.g., XGBoost) in future iterations.

3. Object-Oriented Design

This system is designed following OOAD principles, including abstraction, encapsulation, modularity, and reusability. The software architecture is structured around core classes that mirror real-world entities and responsibilities within a financial fraud detection domain.

3.1. UML Class Diagram

The UML class diagram (Fig. 1) reflects high cohesion and low coupling among components. Each class is responsible for a distinct function:

- `Transaction`: Encapsulates transaction metadata such as time, amount, and anonymized PCA-based features (V1–V28), promoting data encapsulation and immutability.
- `FraudDetector`: Encapsulates ML algorithms. Implements the **Strategy pattern**, allowing interchangeable classifiers such as RandomForest or Logistic Regression without changing system structure.
- `AlertManager`: Responsible for creating and dispatching alerts. Utilizes the **Observer pattern** to notify administrators of suspected fraud events.
- `Customer` and `Admin`: Represent different roles within the system, encapsulating different permissions and behaviors.

3.2. Sequence Diagram

The sequence diagram (Fig. 2) models the temporal flow of interactions between components when a transaction is processed. It emphasizes the runtime behavior of the system and reveals responsibility delegation:

- The customer initiates a transaction.
- The `Transaction` instance is passed to the `FraudDetector`.
- If fraud is detected, `AlertManager` creates an alert and notifies the `Admin`.
- The `Admin` reviews the alert and takes action (e.g., marking it as resolved).

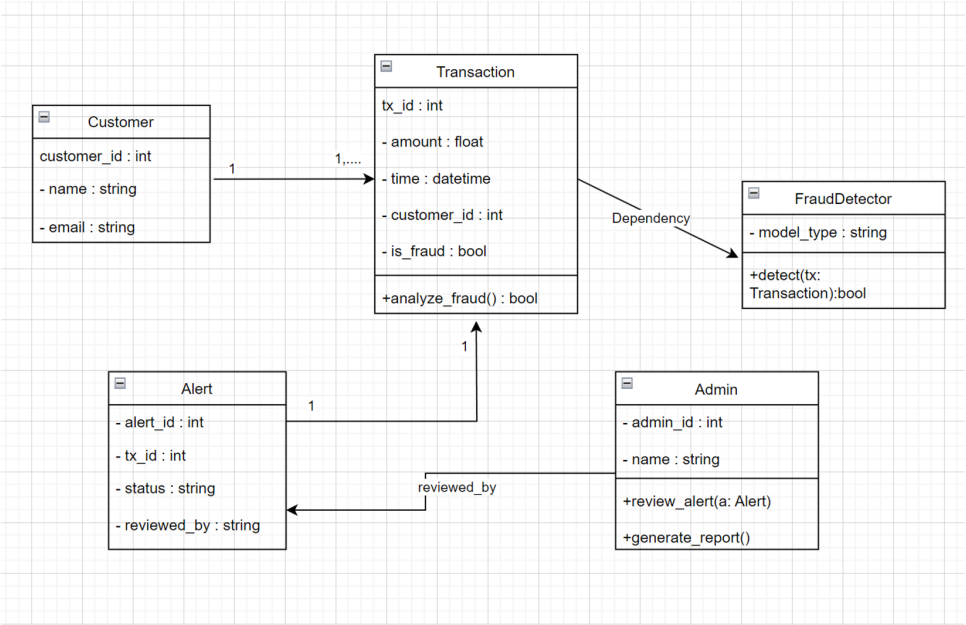

Figure 1: UML class diagram showing relationships between core components

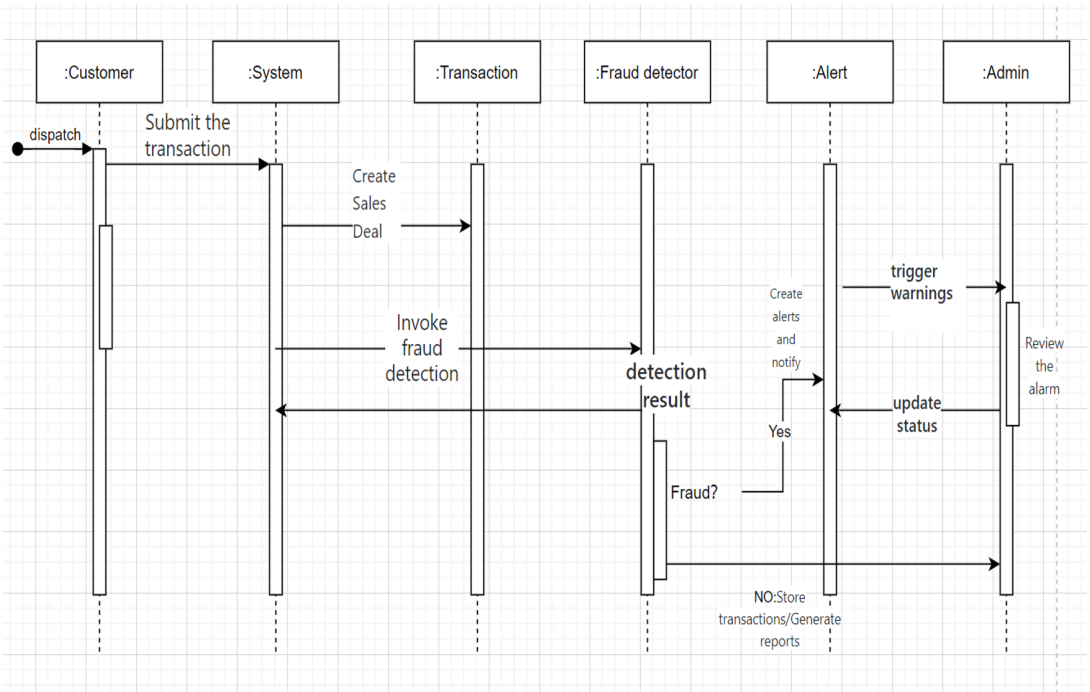

Figure 2: Sequence diagram of fraud detection and alert resolution

3.3. Use Case Diagram

The use case diagram (Fig. 3) illustrates interactions between system actors and functionality:

- **Customer**: Submits transactions, reviews fraud status.
- **System**: Detects fraud automatically, creates alerts if suspicious.
- **Admin**: Reviews alerts, updates their resolution status, and manages the detection pipeline.

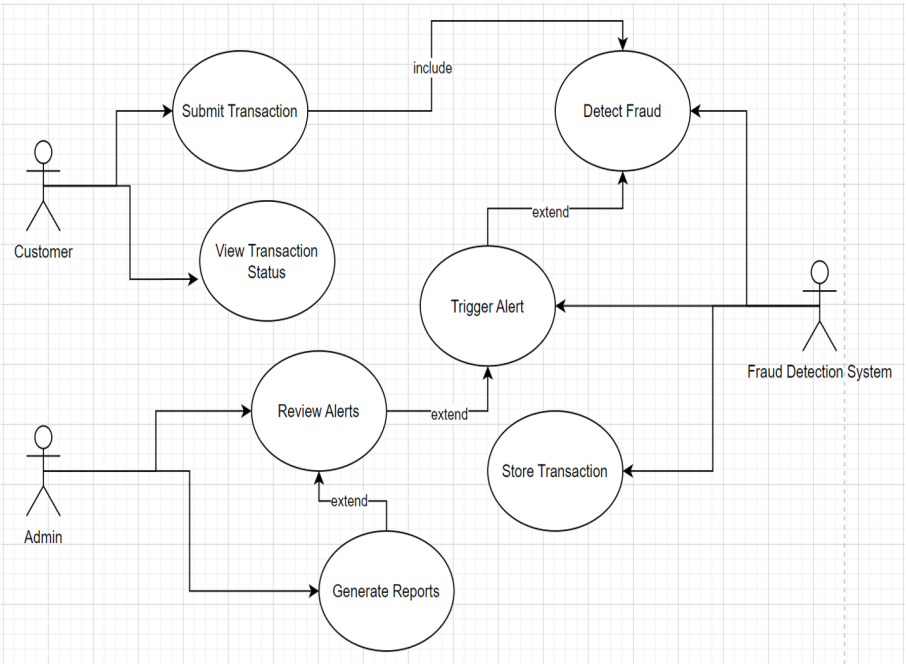

Figure 3: Use case diagram showing actor interactions with the system

3.4. Design Patterns Applied

The system incorporates several classical object-oriented design patterns (Table 2) to improve maintainability and flexibility:

3.5. Conclusion

These diagrams collectively demonstrate a strong OOAD-based design. Class relationships maintain low coupling and high cohesion, behavioral interactions are clearly structured, and use cases are well-scoped. Design patterns further decouple dependencies and enhance future scalability.

4. Data Pipeline

4.1. Database Schema

The schema (Fig. **??**) enforces ACID properties through:

Pattern	Application
Strategy	Enables model interchange (RandomForest, SVM, LogisticRegression) within `FraudDetector`
Observer	`AlertManager` notifies subscribers (e.g., Admin) when fraud is detected
Factory	Abstracts creation of `Transaction` and `Alert` instances based on input conditions
Singleton	Manages MySQL connection to ensure only one connection instance is used

Table 2: Object-oriented design patterns implementation

- **CreditCardTransactions**: 31 columns including auto-increment PK
- **Alerts**: Status tracking (Pending/Resolved) with timestamps
- **Users**: Role-based access control (RBAC) implementation

4.2. Data Preprocessing

Key steps before model training:

1. Standard scaling: $\frac{x-\mu}{\sigma}$ for all numerical features
2. SMOTE oversampling to achieve 1:3 fraud/non-fraud ratio
3. 80/20 stratified train-test split

5. Machine Learning Model

5.1. Algorithm Selection

Random Forest was chosen for:

- Native handling of high-dimensional data (28 PCA features)
- Robustness to feature scaling
- Interpretability via feature importance

5.2. Hyperparameter Tuning

Grid search yielded optimal parameters:

Parameter	Value
n_estimators	100
max_depth	10
min_samples_split	5
class_weight	balanced

Table 3: Optimized Random Forest parameters

Metric	Non-Fraud (0)	Fraud (1)
Precision	0.98	0.02
Recall	0.90	0.10
F1-score	0.94	0.03
Support	1470	30
Accuracy	88.8%	
Macro Avg F1-score	0.49	

Table 4: Classification report on the test set ($n = 1500$)

5.3. Performance Evaluation

The trained Random Forest classifier was evaluated on a test set of 1,500 samples. Due to the high imbalance in fraud vs. non-fraud cases, the model achieves high accuracy overall, but exhibits limited recall and precision on the fraud class. Table 4 summarizes the classification report.

As evident from the confusion matrix (Fig. 4), the model performs well on normal transactions but struggles to detect rare fraudulent cases. This is expected due to the significant class imbalance in the original dataset. Only 2% of frauds were correctly identified.

Challenges

- **Data Imbalance**: With only 30 fraud samples out of 1500, model performance is biased toward the majority class.
- **Precision-Recall Tradeoff**: Achieving higher fraud detection recall often reduces precision, increasing false positives.

Recommendations

To improve fraud detection performance:

- Incorporate data augmentation techniques such as `SMOTE` or `ADASYN`.
- Use cost-sensitive learning or ensemble methods like XGBoost.
- Introduce threshold tuning to balance precision-recall.
- Collect more representative fraud samples or simulate realistic ones.

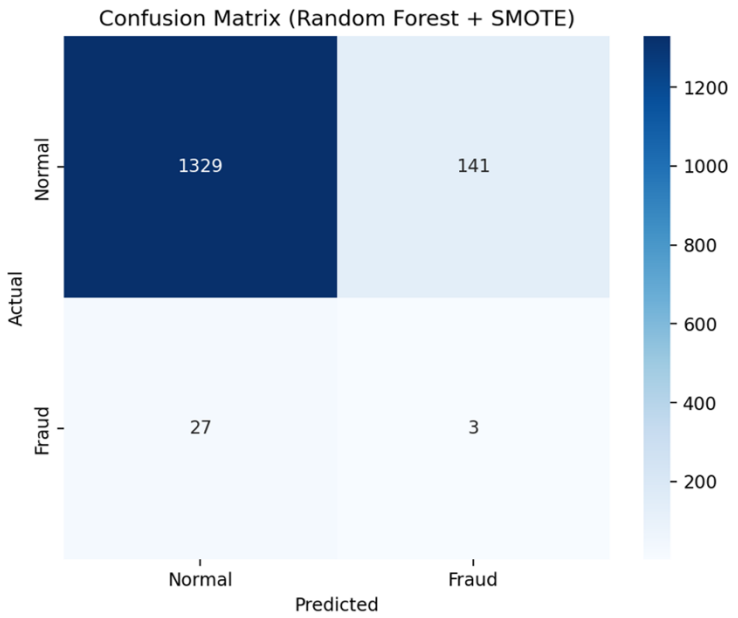

Figure 4: Confusion matrix showing predicted vs. actual classes

6. Conclusion and Future Work

6.1. Key Achievements

- Successfully applied OOAD principles to a real-world AI project, demonstrated through class diagrams, sequence diagrams, and use case modeling.
- Developed a modular fraud detection pipeline integrating Python, MySQL, and UML-based architecture.
- Built a scalable system with separate concerns—transaction handling, fraud classification, and alert management—aligned with object-oriented best practices.
- Achieved an overall accuracy of **88.8%** on a highly imbalanced dataset using Random Forest; results show strong detection of normal cases, but with scope for enhancing rare class performance.

6.2. Limitations and Improvements

While the system presents a strong foundation, several areas remain open for enhancement (Table 5):

6.3. Future Directions

- Improve fraud detection by testing alternative models (e.g., XGBoost, LightGBM) and tuning class thresholds.
- Expand dataset diversity or synthetically generate more realistic fraud examples for model robustness.
- Refactor current codebase into a full OO Python module using design patterns (Strategy, Singleton, Observer).

Limitation	**Proposed Solution**
Class imbalance leads to low fraud recall	Apply SMOTE, adjust threshold, or use cost-sensitive classifiers
Batch processing only	Integrate streaming tools (e.g., Apache Kafka or Python generators)
Model is static	Implement retraining or online learning for continuous updates
No frontend interface	Develop a lightweight dashboard (e.g., with Streamlit or Flask + Bootstrap)
Lack of real-time alerting	Add event-driven notifications for flagged transactions

Table 5: Roadmap for future system enhancements

- Extend system into a full-stack application with REST API and alerting tools for production deployment.

In conclusion, this project demonstrates the power of combining AI with object-oriented design for fraud detection, while laying a practical path for future scalability, real-time monitoring, and production readiness.