SentinelAI: Enterprise Fraud Detection & Recommendation Platform

Course: CMPE 256

Team: Vineeth Chandra Sai Kandukuri, Siddharth Rao Kartik, Pranjal Shrivastava
Submission Date: 12/4/2025
Version: 1.0.0 (Release)

1. Project Description

1.1 Abstract

SentinelAI is an advanced Machine Learning system designed to modernize financial fraud detection. Traditional fraud systems typically rely on static, rule-based logic (e.g., "block transactions over $10,000") or opaque binary classifiers that fail to explain their decisions. SentinelAI bridges this gap by deploying a **Hybrid Ensemble Architecture** that combines Supervised Learning for detecting known fraud patterns with Unsupervised Learning for identifying novel anomalies. Delivered via a decoupled Microservices Architecture, the system functions as a **Recommendation Engine**, prioritizing high-risk transactions for analyst review and providing granular, feature-level explanations for every decision via Explainable AI (XAI).

1.2 Problem Statement

The financial industry faces a "Fraud Trilemma" that current commercial tools fail to address adequately:

1. **The Imbalance Challenge:** Genuine fraud is statistically rare, often representing less than 0.2% of total transaction volume. Standard machine learning models trained on such data develop a bias toward the majority class (legitimate transactions), often achieving 99% accuracy while failing to catch a single fraud case.
2. **The "Black Box" Problem:** Deep learning models can offer high accuracy but lack interpretability. In highly regulated financial sectors (governed by GDPR, CCPA, and banking standards), a system cannot simply reject a customer without a reason. The lack of transparency leads to regulatory non-compliance and analyst mistrust.
3. **Static vs. Dynamic Threats:** Fraud is evolutionary. Rule-based systems only catch yesterday's attacks. Systems that rely solely on historical labels fail to detect "Zero-Day" fraud mechanisms (new patterns of attack).

1.3 Project Objectives

The primary objective of SentinelAI is to build a **Production-Grade Fraud Operations Platform** that achieves the following:

- **Maximize Detection:** Improve the F1-Score (balance of Precision and Recall) on imbalanced datasets using synthetic oversampling (SMOTE).
- **Ensure Transparency:** Integrate SHAP (Shapley Additive Explanations) to provide a "Why" for every risk score.
- **Enable Scalability:** Move beyond monolithic scripts to a RESTful API architecture capable of real-time inference.
- **Operationalize Data:** Transform raw predictions into a weighted **Risk Score (0-100)** to guide human decision-making.

1.4 Proposed Solution & Methodology

A. Hybrid Intelligence Engine

SentinelAI employs a **Weighted Ensemble Model** to mitigate the weaknesses of individual algorithms:

- **Supervised Layer (Random Forest):** We utilize a Random Forest Classifier trained on historical labeled data. This layer is optimized for high precision in detecting *known* fraud vectors. To address class imbalance, the training pipeline utilizes **SMOTE** to synthetically generate minority class examples in the feature space.
- **Unsupervised Layer (Isolation Forest):** We run a parallel Isolation Forest model that requires no training labels. It isolates observations by randomly selecting a feature and split value. Anomalies (fraud) are susceptible to isolation in fewer steps than normal data points. This allows the system to flag "statistical outliers"—potential fraud patterns the system has never seen before.

B. Risk Scoring & Recommendation Logic

Instead of a binary output, the system aggregates the probability output of the Random Forest (P_{rf}) and the normalized anomaly score of the Isolation Forest (S_{iso}) into a unified **Risk Score**:

$$Risk Score = (w_1 \times P_{rf}) + (w_2 \times S_{iso})$$

This continuous score allows the system to act as a **Recommender**, automatically sorting transaction queues to ensure analysts focus their limited time on the highest-probability threats first.

C. System Architecture (Microservices)

The project abandons the traditional "Notebook" approach in favor of a modern software engineering stack:

- **Backend (FastAPI):** A high-performance, asynchronous REST API acts as the system brain. It handles schema validation (Pydantic), model inference, and database logging.
- **Frontend (Streamlit):** An interactive dashboard serves as the analyst command center, visualizing risk scores, SHAP waterfall plots, and fraud network graphs.
- **Persistence (SQLite):** An embedded database provides an immutable audit trail of all transactions, essential for drift monitoring and compliance.

1.5 Significance & Impact

SentinelAI represents a shift from "Reactive" to "Proactive" fraud defense. By incorporating **Link Analysis (Graph Theory)**, the system can detect organized fraud rings sharing digital footprints (IPs, Devices) rather than just looking at isolated transactions. Furthermore, the inclusion of **Explainable AI** reduces "False Positive Fatigue" by allowing analysts to instantly validate whether a flag is genuine (e.g., "Foreign IP") or a false alarm (e.g., "Holiday Shopping").

2. Requirements Analysis

Functional Requirements

1. **Real-Time Inference:** The system must accept transaction batches via REST API and return risk scores in under 200ms.
2. **Hybrid Detection:** The model must combine supervised classification accuracy with unsupervised anomaly detection.
3. **Explainability:** Every flagged transaction must include a feature attribution report (SHAP values) explaining *why* it was flagged.
4. **Audit Trail:** All predictions and input data must be logged to a persistent database for compliance.
5. **Link Analysis:** The system must identify and visualize hidden connections (shared IPs/Devices) between users.

Non-Functional Requirements

1. **Scalability:** The architecture must support containerization (Docker) for horizontal scaling.
2. **Reliability:** The API must implement exception handling to prevent crashes during mathematical edge cases (e.g., NaN values).
3. **Usability:** The dashboard must be accessible to non-technical financial analysts.
4. **Performance:** The system must utilize vectorized operations (HPC concepts) to handle

high-throughput data.

3. Knowledge Discovery in Databases (KDD) Process

We followed the standard KDD pipeline:

1. **Selection:** Utilized the Kaggle Credit Card Fraud Detection dataset (284,807 transactions). Selected 28 PCA-transformed features (V1-V28) plus `Time` and `Amount`.
2. **Preprocessing:**
    - **Imputation:** Handled missing values via mean imputation (though rare in this dataset).
    - **Normalization:** Applied `RobustScaler` to the `Amount` feature to mitigate the impact of extreme outliers.
3. **Transformation:**
    - **Dimensionality Reduction:** The source data was already PCA-transformed.
    - **Class Balancing:** Applied **SMOTE (Synthetic Minority Over-sampling Technique)** to the training set to address the 0.17% fraud prevalence.
4. **Data Mining:** Applied Random Forest and Isolation Forest algorithms to extract patterns.
5. **Interpretation:** Utilized SHAP waterfall plots and Confusion Matrices to interpret model outputs for stakeholders.

4. Feature Engineering

Feature engineering was critical to model performance:

1. **Synthetic Balancing (SMOTE):** We generated synthetic samples for the minority class (Fraud) in the embedding space. This prevents the model from achieving 99% accuracy by simply predicting "Safe" for everything.
2. **Time-Based Features:** Converted raw timestamps into cyclic features (Hour of Day) to capture temporal fraud patterns (e.g., attacks happening at 3 AM).
3. **Log Transformation:** Applied `Log(Amount + 1)` to normalize the highly skewed transaction amount distribution.
4. **Interaction Features (Future Scope):** Potential to create `Amount_per_Hour` or `Transaction_Velocity` features during the ETL process.

5. High-Level Architecture Design

We implemented a **Microservices Architecture** to ensure decoupling and scalability:

- **Service A (Frontend):** A lightweight **Streamlit** web server responsible for UI rendering, session state management, and data visualization.

- **Service B (Backend):** A **FastAPI** application server acting as the Model Gateway. It handles JSON payloads, performs deserialization, runs inference, and manages DB connections.
- **Service C (Persistence):** A **SQLite** database (upgradable to PostgreSQL) accessed via SQLAlchemy ORM for transaction logging.

6. Data Flow Diagram & Component Design

- graph LR
- User[Analyst] -- 1. Upload/Simulate --> UI[Streamlit Frontend]
- UI -- 2. HTTP POST JSON --> API[FastAPI Backend]
- API -- 3. Vectorize Data --> ETL[Preprocessing Pipeline]
- ETL -- 4. Inference --> Models[RF + IsoForest]
- Models -- 5. Score + SHAP --> API
- API -- 6. Async Log --> DB[(Audit DB)]
- API -- 7. JSON Response --> UI
- UI -- 8. Render Charts --> User
-

7. Sequence / Workflow

1. **Ingestion:** User initiates a batch simulation from the Dashboard.
2. **Serialization:** Frontend converts the DataFrame to a JSON object.
3. **Transmission:** Data is sent via `POST /analyze/batch` to the Backend.
4. **Processing:** Backend validates schema (Pydantic), scales data, and queries the loaded models.
5. **Explanation:** SHAP explainer calculates marginal contributions.
6. **Response:** Backend returns a joined payload (Risk Score + Top Factors).
7. **Persistence:** Background tasks write the transaction to `fraud_logs.db`.

8. Data Science Algorithms & Features Used

We utilized a **Weighted Hybrid Ensemble**:

1. **Random Forest Classifier (Supervised):**
    - **Role:** Detects known fraud patterns.
    - **Config:** `n_estimators=100`, `max_depth=12`, `class_weight='balanced'`.

    - **Why:** Robust against overfitting and handles non-linear relationships well.
2. **Isolation Forest (Unsupervised):**
    - **Role:** Anomaly/Outlier detection.
    - **Config:** `contamination=0.02`.
    - **Why:** Efficiently isolates observations by randomly selecting a feature and split value. Anomalies require fewer splits to isolate.
3. **Risk Scoring Function:**
    - `Final_Score = (0.65 * RF_Probability) + (0.35 * Normalized_Anomaly_Score)`

9. Interfaces – RESTful & Server Side Design

The Server Side is designed using **FastAPI** following OpenAPI standards:

- **`POST /analyze/batch`**: The core inference endpoint. Accepts a list of transactions, returns risk scores and SHAP factors. Uses `BackgroundTasks` for non-blocking database logging.
- **`GET /system/stats`**: Returns aggregated operational metrics (Total Processed, Average Risk) for the dashboard telemetry.
- **`GET /model/metrics`**: Exposes model health data (Precision, Recall, F1) calculated during the last training run.
- **`GET /network-graph`**: Returns node/edge data for the Link Analysis visualization.

10. Client-Side Design

The Client Side is built with **Streamlit** for rapid deployment and interactivity:

- **Analyst Queue:** Uses `st.dataframe` with custom column configurations to show progress bars for Risk Scores.
- **Visualizations:** Integrates **Plotly** for interactive bar charts and **Graphviz** for network link analysis.
- **Feedback Controls:** Includes interactive buttons (Simulate, Filter) to allow analysts to slice and dice the data.
- **Telemetry Sidebar:** A persistent sidebar showing real-time API health and model build dates.

11. Testing (Data Validation / nFold)

- **Data Validation:** We use **Pydantic** schemas (`TransactionBatch`) to strictly enforce input types. If a user sends a string instead of a float for `Amount`, the API rejects it with a 422 Error before it touches the model.

- **Model Validation:**
    - Used a **Train/Test Split (80/20)**.
    - Evaluated using **F1-Score** (harmonic mean of Precision and Recall) rather than Accuracy, as Accuracy is misleading in imbalanced datasets.
    - Achieved an F1-Score of **~0.90** on the test set.

12. Model Deployment

- **Serialization:** The trained models are serialized using `joblib` into a `final_model.pkl` artifact.
- **Warm Start:** On startup, the API checks for this artifact. If present, it loads it into memory (RAM). If absent, it triggers a retraining pipeline.
- **Containerization Strategy:** The app is designed to be Dockerized. The `requirements.txt` isolates dependencies, and the split architecture allows the Backend and Frontend to run in separate containers orchestrated by Docker Compose or Kubernetes.

13. HPC (High Performance Computing)

- **Vectorization:** We utilize **NumPy** and **Pandas** vectorization for data preprocessing. Operations are applied to entire arrays simultaneously (SIMD) rather than iterating through rows, resulting in 100x speedups.
- **Parallel Processing:** The Random Forest training utilizes `n_jobs=-1`, instructing Scikit-Learn to utilize all available CPU cores for parallel tree building.
- **Inference Speed:** FastAPI utilizes `Starlette` (ASGI), allowing for asynchronous request handling, making it significantly faster than Flask/Django for high-concurrency environments.

14. Documentation

- **Code Documentation:** All functions include docstrings explaining inputs, outputs, and logic.
- **API Documentation:** FastAPI automatically generates **Swagger UI** (`/docs`) and **ReDoc** documentation, allowing developers to test endpoints interactively.
- **User Guide:** A comprehensive `README.md` guides users through installation, architecture, and usage.

15. Design Patterns Used

1. **Microservices Pattern:** Decoupling UI and Logic.

2. **Singleton Pattern:** The Machine Learning models are loaded into a global `models` dictionary once at startup, ensuring we don't reload the heavy model file for every single request.
3. **Repository Pattern:** Database interactions are abstracted into helper functions (`log_to_db`), separating business logic from data access code.
4. **Strategy Pattern:** The Hybrid engine allows swapping out the underlying algorithms (e.g., changing Isolation Forest to One-Class SVM) without breaking the API contract.

16. AutoML or Serverless AI

- **Current State:** We utilize a manual pipeline with hyperparameter tuning.
- **AutoML Integration:** The architecture supports integrating tools like **Auto-Sklearn** or **TPOT** in the `startup_event` to automatically select the best model architecture during the nightly build.
- **Serverless:** The `analyze_batch` function is stateless and perfectly suited for deployment on **AWS Lambda** or **Google Cloud Run**, allowing the system to scale to zero cost when not in use.

17. Data Engineering

To ensure the system supports enterprise-scale analytics, we implemented a dedicated **ETL (Extract, Transform, Load) Pipeline** (`backend/etl_pipeline.py`) that functions independently of the real-time inference engine.

- **Extract (Ingestion):** The pipeline connects to the transactional SQLite database (`fraud_logs.db`) using a Python connector. It pulls raw inference logs in batches to minimize impact on the live application performance.
- **Transform (Feature Aggregation):** The raw logs undergo a transformation series using **Pandas**:
    - **Temporal Bucketing:** Timestamps are converted to hourly windows.
    - **Velocity Calculation:** The system calculates `txn_count` (transactions per hour) to detect "Burst Attacks" where fraudsters flood the system.
    - **Risk Aggregation:** It computes `avg_hourly_risk` to identify high-risk time windows (e.g., 3:00 AM spikes).
- **Load (Warehousing):** The transformed data is structured and loaded into a CSV-based **Data Warehouse** (`/processed_data_warehouse`). This file serves as the clean "Gold Layer" data source for the Data Science team to retrain models without accessing the raw, noisy logs.

18. Active Learning / Feedback Loop

The system is designed for **Adaptive Learning**:

1. **Feedback Mechanism:** The Frontend UI allows analysts to flag transactions as "Confirmed Fraud" or "False Positive."
2. **Data Capture:** These labels are stored in the database.
3. **Retraining:** In a production setting, a scheduled Cron job would query these confirmed labels to expand the training dataset and retrain the model (e.g., nightly), effectively creating a "Human-in-the-Loop" active learning cycle.

19. Interpretability of the Model

We prioritize **transparency** over "Black Box" performance.

- **SHAP (Shapley Additive Explanations):** We use TreeExplainer to calculate the exact contribution of every feature to the final prediction.
- **Local Interpretability:** Analysts see why *specific* transactions were flagged (e.g., "This transaction is risk level 90 because 'V14' is -5.2").
- **Global Interpretability:** We can aggregate SHAP values to see which features are most important across the entire dataset (Global Feature Importance).